Entwicklung eines Oberon-2-Compilers mit Hilfe einer Compiler-Compiler-Toolbox

– Synthesephase des Compilers

Diplomarbeit
Dezember 1995

Ralf Bauer Maximilian Spring

Technische Universität Berlin
Fachbereich Informatik
Institut für Angewandte Informatik
FG Programmiersprachen und Compiler

Ralf Bauer, Matr.-Nr. 100 377
Maximilian Spring, Matr.-Nr. 100 386

Entwicklung eines Oberon-2-Compilers mit Hilfe einer Compiler-Compiler-Toolbox – Synthesephase des Compilers

Diplomarbeit an der
Technischen Universität Berlin
Fachbereich Informatik
Institut für Angewandte Informatik
FG Programmiersprachen und Compiler
im Dezember 1995

Die Aufgabe wurde von Prof. Dr. Bleicke Eggers gestellt.

Vorwort

Die Aufteilung der Diplomarbeit im Rahmen der Gruppenarbeit:

Ralf Bauer befaßte sich mit der Portierung des Frontends nach Linux, der Spezifikation des Codeerzeugers sowie der Erstellung der notwendigen Oberon-2-Bibliotheksmodule. Er schrieb Kapitel 2 (BEG – Backend-Generator) sowie die Abschnitte 3.1 (Überblick) und 3.2 (Frontend) sowie die Abschnitte 3.3.2.1 (Kommentierungen) bis 3.3.2.3 (Codegeneratorbeschreibung).

Maximilian Spring befaßte sich mit dem Entwurf der Laufzeitdatenstrukturen, der Einbettung des spezifizierten Codeerzeugers sowie der Entwicklung des Laufzeitsystems einschließlich der Freispeicherverwaltung. Er schrieb Kapitel 1 (Zielsprache) sowie die Abschnitte 3.3.1 (Assemblersubsystem) und 3.3.2.4 (Baumtraversierung) sowie die Abschnitte 3.4 (Umgebende Zusatzmodule) bis 3.6 (Spezielle Problembereiche).

Die erstellten Spezifikations- und Programmtexte sind in einem eigenen Band „Anhang zur Diplomarbeit“ enthalten.

Die Arbeit wurde von Mario Kröplin betreut.

Inhaltsverzeichnis

Überblick

Dies ist die Dokumentation der Spezifikation eines Compilers, der Programme in der Sprache Oberon-2 akzeptiert und Intel 80386 Assemblercode unter dem Betriebssystem Linux erzeugt. Die Spezifikation stützt sich auf die Studienarbeit von BAUER und SPRING [1994] und erfolgte mit der Compiler-Compiler-Toolbox von Dr. Josef Grosch (GMD Karlsruhe), einer Sammlung von Einzelwerkzeugen für die Generierung von Programmfragmenten für verschiedene Compiler-Phasen. Für die Synthesephase des Compilers wurde der Backend-Generator BEG von Helmut Emmelmann (ebenfalls GMD Karlsruhe) eingesetzt.

Die Motivation zu dieser Diplomarbeit war das Interesse an der Untersuchung der praktischen Benutzbarkeit dieser Werkzeuge anhand ihres Einsatzes am Beispiel eines *echten* Compilers.

Die Wahl der Quellsprache fiel auf Oberon-2, da diese eine vom Sprachumfang kleine, aber durch ihren Typerweiterungsmechanismus mächtige Programmiersprache ist, die objektorientiertes Programmieren unterstützt. Wegen der großen Verbreitung von Rechnern mit Intel Prozessor ab 80386 (als PCs), erzeugt der Compiler Code für diesen Prozessor. Das Betriebssystem Linux wurde als Zielsystem gewählt, da vor allem für das Laufzeitsystem des Compilers die Kenntnis der internen Strukturen notwendig ist und diese für Linux gut dokumentiert sind. Zudem existieren unter Linux für alle eingesetzten Werkzeuge frei verfügbare Portierungen, so daß es auch als Entwicklungssystem eingesetzt werden konnte. Nicht zuletzt nutzt Linux weitestgehend alle Möglichkeiten dieser Intel Prozessoren.

Die Dokumentation gliedert sich in vier Kapitel. Die Zielspache wird im ersten Kapitel vorgestellt. Hier werden die Strukturen eines ausführbaren Oberon-2-Programms und wird der eingesetzte Assembler näher erläutert. Das zweite Kapitel beschreibt das für die Codegenerierung eingesetzte Werkzeug. Im dritten Kapitel wird auf die erstellte Implementierung eingegangen. Es besteht aus einer Beschreibung der einzelnen Spezifikationsdokumente und der Darstellung der gewählten Lösungen zu speziellen Problembereichen. Das vierte und letzte Kapitel schließt die Arbeit mit einer Bewertung des Werkzeugs und einem Vergleich der erstellten Implementierung mit anderen Compilern ab.

Obwohl die Dokumentation auf deutsch vorliegt, wurden die im Compilerbau gängigen englischen Fachbegriffe dort beibehalten, wo die Verwendung von entsprechenden deutschen Begriffen nicht ratsam erschien. In Wortzusammensetzungen aus verschiedensprachigen Gliedern wird der Bindestrich zur Trennung eingesetzt (z.B. Record-Typ).

Innerhalb der Dokumenation werden verschiedene Schriftarten benutzt, um dem Leser das Erkennen von Textpassagen mit bestimmter Bedeutung zu erleichtern. Spezifikations- und Programmcode ist in der Schriftart `Courier` gesetzt. Für Grammatikregeln, die in der Dokumentation der Beschreibung einer Sprache dienen, wird die Schriftart Sans-Serif verwendet.

1 Zielsprache

Für das Verständnis der Codeerzeugung ist die Kenntnis der Zielsprache unabdingbar. Aus diesem Grund gibt das vorliegende Kapitel einen Überblick über die Zielmaschine und die Strukturen eines übersetzten und ausführbaren Oberon-2-Programms.

Der Codeerzeuger produziert Assemblercode, der mit dem GNU Assembler in binären Maschinencode umgewandelt wird. Die Verwendung des GNU Assemblers und des Linkers `ld` ermöglicht es, die Erzeugung der binären Darstellung von übersetzten Modulen und Programmen diesen Werkzeugen zu überlassen. Der GNU Assembler wird in einem eigenen Abschnitt beschrieben.

1.1 Prozessor

Der Intel 80386 ist der erste Prozessor der Intel 80x86-Familie, der im sogenannten *Protected Mode* entsprechend der üblichen 32-Bit-Architektur betrieben werden kann.

Zusammen mit dem 80386 kann der Koprozessor 80387 zur Beschleunigung von Fließkommaarithmetik eingesetzt werden.

Unter dem Betriebssystem Linux übernimmt bei fehlendem Koprozessor eine Software-Emulation dessen Funktionalität. Die Weiterentwicklungen des Prozessors in Gestalt des 80486 und Pentium entsprechen einer Kombination von 80386 und 80387. Die neu hinzugekommenen Eigenschaften dieser Prozessoren sind für die hier angestellten Betrachtungen ohne Bedeutung, so daß alle Aussagen über den 80386/387 auch für die beiden Nachfolgeprozessoren Gültigkeit besitzen.

1.1.1 Registersatz

Der Registersatz des 80386 besteht aus

- acht sogenannten allgemeinen Registern `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp` und `esp` mit einer Breite von 32 Bit,
- zwei Statusregistern `eflags` und `eip` (Programmzähler) mit einer Breite von 32 Bit,
- sechs Segmentregistern `cs`, `ss`, `ds`, `es`, `fs` und `gs` mit einer Breite von 16 Bit,
- diversen Schutz-, Steuer-, Debug- und Testregistern, welche aber für die Anwendungsprogrammierung hier ohne Bedeutung sind.

Die niederwertigen 16 Bits der allgemeinen Register und der Statusregister besitzen eigene Namen, aus denen sich die entsprechenden Registernamen durch den führenden Buchstaben `E` (Extended) ergeben haben. Zusätzlich können die niederwertigen bzw. höherwertigen 8 Bits der Register `ax`, `bx`, `cx` und `dx` als `al`, `bl`, `cl` und `dl` bzw. `ah`, `bh`, `ch` und `dh` angesprochen werden. Abbildung 1 gibt einen Überblick über die konzeptionellen Register und deren Teilregister.

Der Koprozessor 80387 besitzt 8 Fließkommaregister mit je 80 Bits und einige Status- und Steuerregister. Die Fließkommaregister werden in einem Ringspeicher als Keller verwaltet.

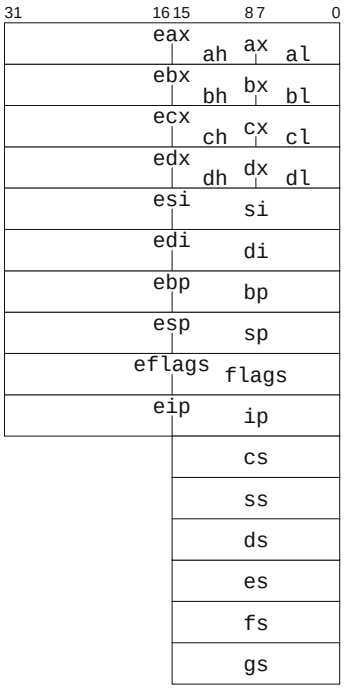

Abbildung 1: Registersatz des 80386

Im 16 Bit breiten Statuswort `SW` existieren drei Bits, genannt `TOP`, welche die zuletzt gekellerte Fließkommazahl adressieren. Diese Fließkommazahl wird mit `ST(0)` oder kurz `ST` angesprochen, die davor gekellerte mit `ST(1)`, die davor mit `ST(2)` usw. Die Belegung der Fließkommaregister nach dem Laden der drei Fließkommawerte `1.0`, `2.0` und `3.0` wird in Abbildung 2 dargestellt.

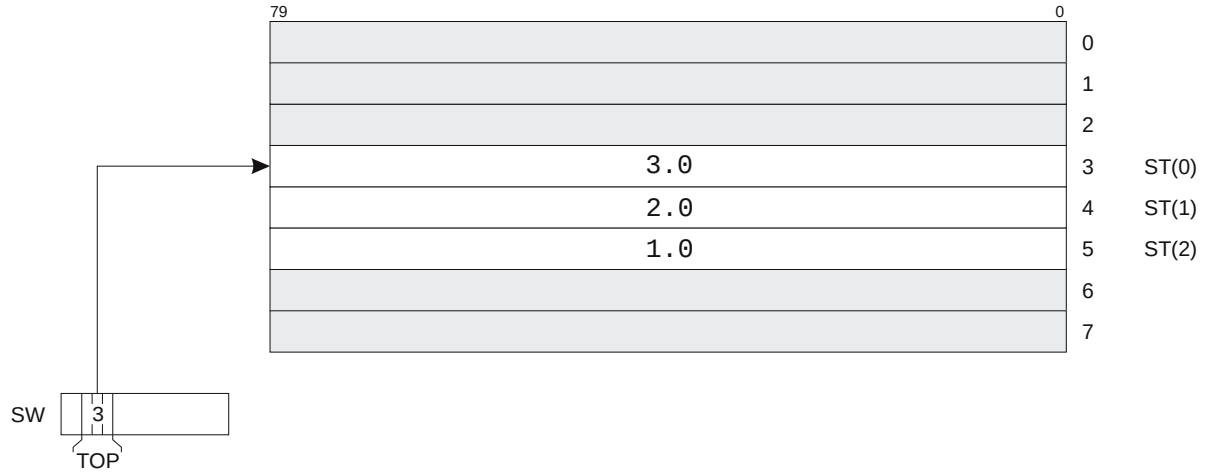

Abbildung 2: Beispielhafte Belegung der 80387 Register

1.1.2 Adressierungsarten

Die Ausführung von Instruktionen durch den Prozessor manipuliert Operanden. Es existieren unterschiedliche Arten von Operanden.

- Ein *impliziter Operand* ist bereits durch die Instruktion selbst definiert.
- Ein *Registeroperand* wird durch eine explizite Registerangabe definiert.
- Ein *unmittelbarer Operand* ist direkt hinter dem Instruktionscode als Wert angegeben und damit Teil der Instruktion.
- Ein *Speicheroperand* befindet sich im Hauptspeicher an einer bestimmten Adresse. Es existieren hierbei unterschiedliche Möglichkeiten der Adreßbildung, welche als Adressierungsarten bezeichnet werden.
  - Bei der *direkten Adressierung* steht die Adresse als Wert unmittelbar in der Instruktion.
  - Bei der *Basisadressierung* befindet sich die Adresse in einem explizit angegebenen Register.
  - Bei der *Indexadressierung* kann zusätzlich ein sogenannter *Skalierungsfaktor* mit dem Wert 1, 2, 4 oder 8 angegeben werden, mit dem der Inhalt des Indexregisters bei der Adreßbildung multipliziert wird.
  - Es existiert weiter eine Kombination aus Basis- und Indexadressierung, bei der die Adresse als Summe gebildet wird.
  - Bei der Basis- und Indexadressierung sowie ihrer Kombination besteht jeweils die Möglichkeit, zusätzlich einen Wert anzugeben, der zu der Adresse addiert wird. Die erweiterte Adressierungsart wird dann als „*mit Displacement*“ bezeichnet.
- Ein *E/A-Operand* schließlich ist die Bezugnahme auf einen E/A-Port, dessen Adresse entweder unmittelbar oder über ein Register angegeben wird.

1.2 Betriebssystem

Linux ist ein frei verfügbares, UNIX-ähnliches Betriebssystem für Personal Computer. Seine Hauptverbreitung hat es auf Rechnern mit *IBM PC*-kompatibler Architektur und Intel Prozessor ab 80386. Es umfaßt Teile der Funktionalität von UNIX System V, der POSIX-Spezifikation und BSD. Der finnische Informatikstudent Linus Torvalds entwickelte wesentliche Teile des Linux-Kerns, dessen erste Version Ende 1991 im Internet verfügbar gemacht wurde.

Nach Hetze et. al. [1994] läuft praktisch das gesamte Angebot an freier Software auch unter Linux, und die Linux-Distributionen haben heute einen Umfang erreicht, der die Angebote kommerzieller UNIX-Systeme leicht in den Schatten stellt (S. 19f.). Nicht zuletzt begründet die hohe Stabilität der vorhandenen GNU Software nach Miller et. al. [1995] die große Attraktivität von Linux. Unter anderem wurde diese Arbeit mit LaTeX unter Linux erstellt.

Linux läuft im *Protected Mode* des 80386 und kann damit die Möglichkeiten dieser Prozessorarchitektur, wie u.a. die virtuelle Speicherverwaltung, den 4 Gigabyte großen Adreßraum und die Speicherschutzmechanismen nutzen.

1.3 Assembler

Der zur Erzeugung des Objektcodes eingesetzte GNU Assembler `as` ist der einzige unter Linux verbreitete Assembler. Er wird in diesem Abschnitt soweit beschrieben, wie es zum Verständnis der nachfolgenden Kapitel notwendig ist. Eigenschaften von `as`, welche in dieser Arbeit nicht verwendet wurden, bleiben unerwähnt. Die Beschreibung stützt sich auf die Assemblerdokumentation von ELSNER und FENLASON [1993]. Die Kenntnis der Maschineninstruktionen des Intel 80386/387 wird vorausgesetzt. Eine umfassende Dokumentation hierzu ist u.a. bei NELSON [1991] zu finden.

1.3.1 Vokabular und Repräsentation

Leerzeichen, Tabulatoren und Zeilenumbrüche dienen der Trennung von Symbolen. Symbole sind Namen, Zahlen, Zeichenketten, Operatoren und Begrenzer sowie Register. Groß- und Kleinbuchstaben werden als unterschiedlich angesehen.

1. *Namen* sind Folgen von Buchstaben, Ziffern und den Zeichen '_', '.' und '$'. Das erste Zeichen darf keine Ziffer sein.

```
ident          =  extendedLetter { extendedLetter | digit }.
extendedLetter =  letter | '_' | '.' | '$'.
letter         =  'A' | ... | 'Z' | 'a' | ... | 'z'.
```

2. *Zahlen* sind Folgen von Ziffern und den Buchstaben `A` bis `F` sowie `a` bis `f`. Besitzt eine Zahl ein Präfix '`0b`', '`0`' bzw. '`0x`', so wird sie als zur Basis 2, 8 bzw. 16 angesehen.

```
number     =  '0b' binInteger | '0' octInteger | decInteger | '0x' hexInteger.
binInteger =  binDigit { binDigit }.
octInteger =  { octDigit }.
decInteger =  ('1' | ... | '9') { digit }.
hexInteger =  hexDigit { hexDigit }.
binDigit   =  '0' | '1'.
octDigit   =  binDigit | '2' | ... | '7'.
digit      =  octDigit | '8' | '9'.
hexDigit   =  digit | 'A' | ... | 'F' | 'a' | ... | 'f'.
```

3. *Zeichenketten* sind Folgen von Zeichen zwischen Anführungszeichen ("). Innerhalb einer Zeichenkette können die folgenden Kodierungen für spezielle Zeichen verwendet werden:

Kodierung	*Zeichen*
\b	`BS` (ASCII 8, Backspace)
\t	`HT` (ASCII 9, Tabulator)
\n	`LF` (ASCII 10, Newline)
\f	`FF` (ASCII 12, Formfeed)
\r	`CR` (ASCII 13, Return)
ddd	Zeichen gemäß oktalem ASCII-Code

```
string  =  '"' { char } '"'.
char    =  ' ' | '!' | '#' | ... | '[' | ']' | ... | '~' | special.
special =  '\b' | '\t' | '\n' | '\f' | '\r' | '\' octDigit octDigit octDigit.
```

4. *Operatoren* und *Begrenzer* sind die Zeichen '+', '-' und ',.

5. *Register* sind die nachfolgend aufgeführten Folgen von Zeichen, die mit dem Zeichen '%'beginnen, gefolgt von der Registerbezeichnung in Kleinbuchstaben.

```
%eax    %ax     %ah     %al     %cs     %st(0)    %st
%ebx    %bx     %bh     %bl     %ds     %st(1)
%ecx    %cx     %ch     %cl     %ss     %st(2)
%edx    %dx     %dh     %dl     %es     %st(3)
%esi    %si                     %fs     %st(4)
%edi    %di                     %gs     %st(5)
%ebp    %bp                             %st(6)
%esp    %sp                             %st(7)
```

6. *Kommentare* sind Folgen von Zeichen, die vom Assembler ignoriert werden. Sie werden durch das Zeichen '#' eingeleitet und erstrecken sich bis zum Ende der jeweiligen Zeile.

1.3.2 Ausdrücke

Ausdrücke beschreiben Regeln zur Berechnung von Werten. In ihnen werden Zahlen und Namen kombiniert, um andere Werte durch die Anwendung von Operatoren zu berechnen. Als Operatoren stehen + und - zur Verfügung. Als monadischer Operator bedeutet - Vorzeichenumkehrung (Zweierkomplement).

```
Expression       =  [ '-' ] SimpleExpression.
SimpleExpression =  Term { Operator Term }.
Term             =  number | ident | '(' Expression ')'.
Operator         =  '+' | '-'.
```

1.3.3 Anweisungen

Anweisungen sind die Bestandteile eines Assemblerprogramms. Eine Anweisung kann eine Deklaration, eine Direktive oder eine (Maschinen-)Instruktion sein. Sie wird durch das Zeilenende abgeschlossen.

```
Statement  =  Declaration | Directive | Instruction.
```

1.3.4 Deklarationen

Eine Deklaration verknüpft einen Namen mit einem Wert. Ein Name darf im gesamten Assemblerdokument höchstens einmal deklariert sein. Die Applikation eines Namens kann vor seiner Deklaration erfolgen. Ein nicht deklarierter Name muß in einem anderen Assemblerdokument deklariert sein, so daß die Verknüpfung des Namens mit einem Wert durch den Linker durchgeführt werden kann.

Es stehen zwei Arten von Deklarationen zur Verfügung. Ein Name, dem unmittelbar (ohne Zwischenraum) ein Doppelpunkt folgt, wird Label genannt und repräsentiert den aktuellen Wert des Adreßzählers an dieser Stelle. Zusätzlich kann ein Name durch eine Definition explizit mit einem bestimmten Wert verknüpft werden.

```
Declaration  =  ident ':' | ident '=' Expression.
```

Ein Label kann auch zusammen mit einer Instruktion in derselben Zeile stehen.

1.3.5 Direktiven

Alle Direktiven sind Namen bestehend aus einem Punkt gefolgt von Kleinbuchstaben. Direktiven beeinflussen die Assemblierung.

Die folgenden alphabetisch aufgeführten Direktiven stehen zur Verfügung:

`.align` *bits* `,` *padding*
: Es wird der Adreßzähler auf eine bestimmte Adreßgrenze erhöht. Der Ausdruck *bits* gibt die (positive) Anzahl der niederwertigen Bits des Adreßzählers an, die nach der Versetzung auf 0 gesetzt sind. Der Ausdruck *padding* definiert den Wert, auf den die aufzufüllenden Bytes gesetzt werden.

`.asciz` *string* { `,` *string* }
: Es werden die Zeichen von einer Zeichenkette oder mehrerer durch Komma getrennter Zeichenketten abgesetzt. Nach jeder Zeichenkette wird ein Byte mit dem Wert 0 abgesetzt.

`.byte` *expression* { `,` *expression* }
: Der Wert jedes Ausdrucks modulo 256 wird in das jeweils nächste Byte abgesetzt. Von `as` wird eine Warnung ausgegeben, falls ein Wert größer als 255 ist.

`.comm` *ident* `,` *length*
: Es wird ein benannter Bereich im BSS-Abschnitt definiert. Der Ausdruck *length* gibt die (positive) Anzahl von Bytes an, die dieser Bereich umfaßt. Der Name *ident* steht für die Adresse des ersten Bytes dieses Bereichs und ist damit deklariert.

`.data`
: Für die Assemblierung der nachfolgenden Anweisungen wird der Datenabschnitt (data section) gewählt.

`.globl` *ident*
: Der Name *ident* wird „exportiert“, d.h. der Name wird für den Linker `ld` sichtbar gemacht, so daß in einem anderen Assemblerdokument auf diesen Namen Bezug genommen werden kann, obwohl es in diesem nicht deklariert ist.

`.long` *expression* { `,` *expression* }
: Der Wert jedes Ausdrucks wird in die jeweils nächsten vier Bytes als 32-Bit-Wert abgesetzt.

`.text`
: Für die Assemblierung der nachfolgenden Anweisungen wird der Codeabschnitt (text section) gewählt.

1.3.6 Instruktionen

Instruktionen sind alle Maschineninstruktionen des Intel 80386/387 in AT&T System V/386 Assemblersyntax. Diese Syntax unterscheidet sich von der sonst üblichen, u.a. auch von NELSON [1991] verwendeten Intel-Syntax in den folgenden Punkten:

- Unmittelbare Operanden werden durch ein vorangestelltes '`$`' gekennzeichnet.
  Beispiel: `pushl $4711`
- Registerbezeichnungen in Kleinbuchstaben werden mit dem Zeichen '`%`' präfiziert (z.B. `%ebp`).
- Der Quelloperand wird textuell vor dem Zieloperand aufgeführt.
  Beispiel: `movl $13,%eax`
- Die Operandengröße wird durch ein Op-code-Suffix explizit angegeben.

Suffix	*Größe*
`b`	8 Bit
`w`	16 Bit
`l`	32 Bit (oder 64 Bit bei Fließkomma-Op-codes)
`s`	32 Bit (bei Fließkomma-Op-codes)

- Bei den vorzeichenerweiternden bzw. nullerweiternden Op-codes `movs` bzw. `movz` werden die Größen des Quell- und des Zieloperanden durch je ein Op-code-Suffix ausgedrückt (z.B. `movsbl %al,%ebx`).
- Die allgemeine Adressierung eines Speicheroperanden erfolgt in der Form

  Versatz (*Basis* , *Index* , *Skalierung*)

  Die Adresse des Operanden ergibt sich aus

  $Basis + Index * Skalierung + Versatz$, mit
  $Basis, Index \in \{$`%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`, `%esp`$\}$,
  $Skalierung \in \{1, 2, 4, 8\}$

  Einzelne Bestandteile dieser Form können weggelassen werden.

  Beispiele: `8(%ebp)` `var(,%ebx,4)` `var+10(%eax,%ebx)`

1.3.7 Kommandozeilenoptionen

Der Aufruf von `as` erfolgt im allgemeinen in der Form:

`as -o` *ziel*`.o` *quelle*`.s` *[Optionen]*

Die wichtigsten Optionen sind:

`-a` Schreiben eines Assembler-Listings auf die Standardausgabe
`-L` Abschaltung der Spezialbehandlung von lokalen Labels[1]

[1] Labels, die mit dem Buchstaben `L` beginnen, werden als lokal bezeichnet und von `as` auch dann nicht in die Objektdatei aufgenommen und dem Linker bekannt gemacht, wenn diese global deklariert sind.

1.4 Speicherstrukturen

Die nachfolgenden Unterabschnitte dokumentieren die Speicherstrukturen eines übersetzten und ablauffähigen Oberon-2-Programms. Ausgehend von dem durch das Betriebssystem gegebenen Speicher-Layout von Prozessen und Programmen sowie der Definition von Konventionen für Labels und Typgrößen erfolgt eine Aufzählung verschiedener eigener Entwurfsentscheidungen bezüglich wichtiger Speicherstrukturen.

1.4.1 Prozesse

Der virtuelle Adreßraum eines Prozesses unter Linux besteht aus dem Kernel- und dem Nutzerbereich. Jeder solche Bereich umfaßt ein Code-, ein Daten- und ein Stacksegment, die alle denselben linearen Adreßraum überlagern und sich lediglich in ihren Zugriffsrechten unterscheiden. Hieraus folgt, daß für die Anwendungsprogrammierung in Assembler die Segmentregister des Intel 80386 ohne Bedeutung sind und Speicheroperanden über 32-Bit-Adressen angesprochen werden.

Die Aufteilung des Adreßraums des Nutzerbereichs im Nutzermodus ist in Abbildung 3 schematisch dargestellt. Dieser Speicherbereich wird durch das Laden einer Binärdatei im `a.out`-Format initialisiert und gliedert sich in verschiedene Abschnitte.

Der Nutzerbereich beginnt bei Adresse 0 mit dem Codeabschnitt, der den Programmcode und Konstanten enthält. Der sich daran anschließende Datenabschnitt dient zur Aufnahme von Variablen, die bereits durch das Laden der Binärdatei mit bestimmten Werten initialisiert werden. Der BSS-Abschnitt[1] enthält ebenfalls Variablen, die in der Binärdatei keinen „Platz verbrauchen“ und lediglich mit 0 initialisiert werden. Das Ende des BSS-Abschnitts wird durch die Betriebssystemvariable `brk` definiert und kann zur Laufzeit über einen entsprechenden Betriebssystemaufruf verändert werden. Die Vergrößerung des BSS-Abschnitts wird zur Aufnahme des Heaps benutzt, der in Richtung höherer Adressen wächst.

Ab der Adresse `0x60000000` sind Shared Libraries in den Adreßraum eingeblendet, deren Funktionalität von Anwendungsprogrammen genutzt wird. Shared Libraries sind für die weiteren Betrachtungen nur insofern von Belang, als daß das Wachstum von Heap (und Stack) durch sie begrenzt wird.

Am obereren Ende des Adreßraums unterhalb des Bereichs, der die Umgebungsvariablen und Kommandozeilenargumente enthält, befindet sich der Stack, der in Richtung niedriger Adressen wächst. Die Adresse des obersten Stack-Elements wird im Prozessorregister `esp` gehalten.

Während der Codeabschnitt im Nutzermodus nur lesbar und ausführbar ist, sind alle anderen Abschnitte lesbar und beschreibbar.

[1] Timar [1995] schreibt zum Stichwort BSS: „Dennis Ritchie says: Actually the acronym (in the sense we took it up; it may have other credible etymologies) is "Block Started by Symbol". It was a pseudo-op in FAP (Fortran Assembly [-er?] Program), an assembler for the IBM 704-709-7090-7094 machines. It defined its label and set aside space for a given number of words. There was another pseudo-op, BES, "Block Ended by Symbol" that did the same except that the label was defined by the last assigned word + 1. (On these machines Fortran arrays were stored backwards in storage and were 1-origin.)
The usage is reasonably appropriate, because just as with standard Unix loaders, the space assigned didn't have to be punched literally into the object deck but was represented by a count somewhere.“

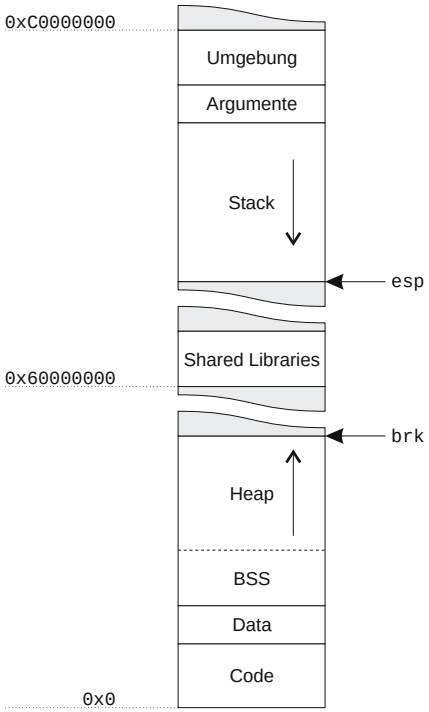

Abbildung 3: Speicherstruktur eines Prozesses, dessen Binärdatei vom Format `a.out` *ist, nach* Beck *[1994], S. 83*

1.4.2 Programme

Die Programmiersprache Oberon(-2) wurde als integraler Bestandteil des Oberon-Systems konzipiert. In diesem existiert jedoch der Begriff des Hauptprogramms nicht (vgl. Reiser [1991], S. 203) und damit auch nicht der des Hauptmoduls, wie z.B. in Modula-2. Statt dessen können im Oberon-System einzelne exportierte, parameterlose Prozeduren aktiviert werden, wodurch das dazugehörende Modul geladen wird, falls es noch nicht geladen ist. Nach Reiser und Wirth [1994] wird „die Anweisungsfolge eines Moduls, auch Modulrumpf genannt, ... ausgeführt, wenn das Modul zum ersten Mal benötigt und in den Speicher geladen wird“ (S. 75). Das im Oberon-System eingesetzte *dynamische Laden* von Modulen führt dazu, daß weitere importierte Module erst dann geladen werden, wenn sie erstmalig benutzt werden.

Demgegenüber ist ein Anwendungsprogramm unter Linux ein monolithisches Objekt, in welches alle benutzten Module vollständig eingebunden sind. Aus diesem Grund wird der Begriff des *Hauptmoduls* eingeführt. Ein Hauptmodul ist ein beliebiges Oberon-2-Modul, welches dem Compiler durch einen entsprechenden Aufruf als Hauptmodul kenntlich gemacht wird. Der Modulrumpf des Hauptmoduls entspricht dann dem Hauptprogramm. Desweiteren wird für jedes Anwendungsprogramm ein sogenanntes *Wurzelmodul* benötigt, welches die Modulrümpfe aller zur Anwendung gehörenden Module ausführt.

Als *Programm* wird eine ausführbare Binärdatei bezeichnet. Wenn `M` der Name eines Oberon-2-Hauptmoduls ist, das direkt oder indirekt die Module `M1`, `M2`, ..., `Mn` importiert, so wird vom Linker `ld` ein Programm aus den folgenden Dateien zusammengesetzt:

- Startcode `crt0.o`
- Laufzeitsystem `OB2RTS.o`
- Freispeicherverwaltung `Storage.o`
- Wurzelmodul `_M.o`
- Hauptmodul `M.o`
- Module `M1.o`, `M2.o`, ..., `Mn.o`
- C-Bibliothek `libc.a`

Der Startcode und die C-Bibliothek sind Bestandteil des Betriebssystems und üblicherweise im Verzeichnis `/usr/lib` zu finden. Das Laufzeitsystem und die Freispeicherverwaltung sind konstante Objektdateien und gehören zum Compiler-Paket. Das Wurzelmodul wird vom Compiler vor dem Aufruf des Linkers neu generiert. Alle restlichen Dateien sind jeweils das Resultat der Übersetzung des entsprechenden Oberon-2-Moduls.

Jede Objektdatei liegt im `a.out`-Format vor und besitzt deswegen Codedefinitionen für Code-, Daten- und BSS-Abschnitt.

Über alle Module, die zu einer ausführbaren Binärdatei gehören, existiert eine partielle Ordnung durch die Importierungsrelation. Mittels topologischer Sortierung wird daraus eine totale Ordnung ermittelt, die die sogenannte *Modulfolge* eines Programms definiert. Das Hauptmodul ist das erste Element der Modulfolge.

Programmbeispiel: `cat`

Im folgenden wird anhand eines kleinen Beispiels verdeutlicht, was der beschriebene Aufbau von Programmen unter Linux für die Programmierung in Oberon-2 bedeutet. Das gewählte Beispiel ist eine Implementierung der Kernfunktionalität des UNIX-Kommandos `cat`. Der hierfür notwendige Zugriff auf die Kommandozeilenparameter sowie auf die Dateioperationen des Betriebssystems geschieht über ein Fremdmodul. Die zur Modellierung der Schnittstelle zum Betriebssystem zur Verfügung stehenden Fremdmodule werden in Abschnitt 3.2.1, Änderungen und Ergänzungen des Frontends (S. 45), näher erläutert.

Das nachstehende Fremdmodul `SysLib` enthält ausschließlich die Deklarationen, welche für die Implementierung von `cat` benötigt werden. Die Variablen `argc` und `argv` repräsentieren zwei Variablen des Laufzeitsystems (`OB2RTS`) und enthalten die Anzahl der Kommandozeilenargumente sowie die Referenzen auf diese. Demgegenüber definieren die Prozedurdeklarationen die Schnittstelle zu den gleichnamigen Funktionen der C-Bibliothek, deren Funktionalität in den entsprechenden Dokumentationen beschrieben ist. Die Variablendeklaration für `argv` ist die Umsetzung der C-Deklaration „`char *argv[];`“ auf möglichst adäquate Weise.

```
FOREIGN MODULE SysLib;

   CONST stdin*  = 0;
         stdout* = 1;
         stderr* = 2;

         RDONLY* = 0;

   VAR   argc-   : LONGINT;
         argv-   : POINTER TO ARRAY 1000000 OF
                   POINTER TO ARRAY 1000000 OF CHAR;

   PROCEDURE open* (pathname,flags,mode:LONGINT):LONGINT; END open;
   PROCEDURE close*(fd                  :LONGINT):LONGINT; END close;
   PROCEDURE read* (fd,bufP,count       :LONGINT):LONGINT; END read;
   PROCEDURE write*(fd,bufP,count       :LONGINT):LONGINT; END write;

END SysLib.
```

Das im folgenden als Oberon-2-Modul `Cat` aufgeführte Programm besitzt die Funktionalität, die Inhalte aller Dateien, deren Namen als Parameter auf der Kommandozeile angegeben sind, an die Standardausgabe zu leiten.

```
MODULE Cat;
   IMPORT SysLib,SYSTEM;

   PROCEDURE CatFile(filename:ARRAY OF CHAR);
      VAR f,n:LONGINT; buf:ARRAY 4096 OF CHAR;
   BEGIN
      f:=SysLib.open(SYSTEM.ADR(filename),SysLib.RDONLY,0);
      IF f>=0 THEN
         LOOP
          n:=SysLib.read(f,SYSTEM.ADR(buf),LEN(buf));
          IF n<=0 THEN EXIT; END;
          n:=SysLib.write(SysLib.stdout,SYSTEM.ADR(buf),n);
         END;
         n:=SysLib.close(f);
      END;
   END CatFile;

   PROCEDURE AllFiles;
      VAR i:LONGINT;
   BEGIN
      FOR i:=1 TO SysLib.argc-1 DO
       CatFile(SysLib.argv[i]^);
      END;
   END AllFiles;

BEGIN
   AllFiles;
END Cat.
```

Die eingesetzte For-Schleife beginnt mit dem Wert 1, da der erste Kommandozeilenparameter immer den Dateinamen des Programms enthält. Die Übergabe der Parameter an die Prozedur `CatFile` als Val-Parameter führt trotz der statischen Elementgröße wegen einer speziellen Optimierung (vgl. S. 24) hier nicht zu einem Überlauf des Stacks.

1.4.3 Labels

Der hier konzeptionell verwendete Begriff des Labels dient der symbolischen Benennung von bestimmten Adressen. Durch die Verwendung eines Assemblers werden sie in der nachfolgend beschriebenen syntaktischen Gestalt auch tatsächlich innerhalb der Zielsprache eingesetzt. Es werden lokale und globale Labels unterschieden.

Ein lokales Label steht im allgemeinen für die Anfangsadresse eines nichtexportierten Oberon-2-Objekts und ist nur innerhalb eines übersetzten (noch nicht assemblierten) Moduls bekannt. Es besteht aus dem Buchstaben `L` gefolgt von einer Folge von Ziffern.

Ein globales Label steht für eine exportierte Prozedur, eine Speicherkonstante oder eine andere zu einem Objekt gehörende Adresse (s.u.). Wenn `M` der Name eines Moduls und `P` der Name einer Prozedur oder Speicherkonstanten ist, lautet das zugehörige globale Label `M_P`.

Zusätzliche mit einem Objekt assoziierte Adressen werden mit einem Label bezeichnet, das aus dem eigentlichen Objektnamen oder -Label und einem bestimmten Suffix besteht. Folgende Label-Suffixe werden verwendet:

Suffix	*Bedeutung*	*Objekt*
`$G`	BSS-Abschnitt	Modul
`$I`	Initialisierung	Modul oder Typ
`$D`	Deskriptor	Modul, Prozedur oder Typ
`$N`	Name	Modul, Prozedur oder Typ
`$S`	Skipper	Typ

Da ein Modulrumpf wie eine Prozedur behandelt wird (s.u.), existieren für ihn noch zwei weitere Labels mit den Suffixen für den Deskriptor (`$I$D`) und den Namen (`$I$N`). Ein *Skipper* dient im Rahmen der automatischen Freispeicherverwaltung zum „Überspringen“ eines im Heap angelegten Objekts.

Desweiteren werden sogenannte *implizite* Labels verwendet, um dem für einen anonymen Typ notwendigen Deskriptor eindeutig eine symbolische Adresse zuordnen zu können. Ein implizites Label besteht aus dem Prozedurnamen, gefolgt von dem Zeichen '`$`' und einem numerischen Suffix, dessen Wert pro Modul eindeutig ist.

1.4.4 Typgrößen

Die Größen der Oberon-2-Typen (in Byte) sind durch die nachstehende Tabelle gegeben. Die Größen der Standardtypen entsprechen den Empfehlungen von KIRK [1993]. Die beiden vom Modul `SYSTEM` exportierten Typen `BYTE` und `PTR` werden im weiteren ebenfalls als Standardtypen angesehen.

Typ	*Größe*
`CHAR`	1
`BOOLEAN`	1
`SHORTINT`	1
`INTEGER`	2
`LONGINT`	4
`REAL`	4
`LONGREAL`	8
`SET`	4
`POINTER`	4
`PROCEDURE`	4
`SYSTEM.BYTE`	1
`SYSTEM.PTR`	4

Ein Array-Typ hat die Größe seines Elementtyps multipliziert mit seiner Länge. Die Größe eines Record-Typs ergibt sich aus der Summe der Typgrößen seiner Felder. Eine besondere Ausrichtung von Record-Feldern erfolgt nicht. Dies überläßt dem Programmierer die volle Kontrolle über die Lage von Record-Feldern und garantiert die minimale Größe des Records.

Im folgenden werden 2 Bytes als ein Wort und 4 Bytes als ein Langwort bezeichnet.

1.4.5 Deskriptoren

Ein Deskriptor ist eine Speicherstruktur, die bestimmte Merkmale des Objekts, dem sie zugeordnet ist, beschreibt. Es werden die Deskriptoren für Module, Prozeduren, Record-Typen, statische Array-Typen und offene Array-Typen unterschieden. Deskriptoren werden im Rahmen der automatischen Freispeicherverwaltung benötigt. Die Deskriptoren von Record-Typen werden zusätzlich für Typtests und Typzusicherungen eingesetzt. Die Adresse eines Deskriptors ist durch ein Label mit dem Suffix `$D` definiert. Die einzelnen Elemente einer Deskriptorstruktur sind Langworte.

Wird die Adresse eines Deskriptors durch das Label `Label$D` beschrieben, so besitzen alle Deskriptoren die folgenden Elemente:

Adresse	*Element*
`Label$D-12`	Verweis auf den Objektnamen
`Label$D-8`	Zusätzliche Objektinformation
`Label$D-4`	Verweis auf die Skipper-Routine
ab `Label$D`	Offset-Tabelle

Ein Objektname dient ausschließlich dazu, beim Auftreten eines Fehlers zur Laufzeit eine informative Fehlermeldung ausgeben zu können. Der Objektname ist eine nullterminierte ASCII-Zeichenkette, welche unter `Label$N` abgelegt ist. Er besteht

- bei Modulen aus dem Modulnamen,
- bei Prozeduren aus dem *vollqualifizierten* Prozedurnamen,
- bei benannten Typen aus dem *vollqualifizierten* Typnamen,
- bei unbenannten Typen `T` aus dem *vollqualifizierten* Namen des Objekts `o` (Variable oder Typ), das `T` enthält, und dem „Weg“ von `o` nach `T`, bestehend aus Oberon-2-Selektoren (`'.'`, `'^'`, `'[]'`).

Über den *vollqualifizierten* Namen können alle Objekte eines Moduls trotz Prozedurverschachtelung eindeutig benannt werden. Ein Objekt (Konstante, Typ, Variable oder Prozedur), das global im Modul `M` mit dem Namen `x` deklariert ist, besitzt den *vollqualifizierten* Namen `M.x`. Ein Objekt, das lokal zu einer Prozedur `P` mit dem Namen `x` deklariert ist, besitzt als *vollqualifizierten* Namen den *vollqualifizierten* Namen von `P` mit dem Suffix `.x`.

Die zusätzliche Objektinformation gibt bei Records und statischen Arrays die Größe des Typs in Byte an. Bei offenen Arrays steht im Feld Objektinformation ein negativer Wert, dessen Betrag die Anzahl der offenen Dimensionen des Arrays angibt. In Modul- und Prozedurdeskriptoren hat das Feld den Wert 0.

Die Offset-Tabelle wird für die automatische Freispeicherverwaltung benötigt. Sie gibt die Lage aller Zeiger im Objekt an. Die Offset-Tabellen von Modul-, Prozedur- und Typdeskriptoren besitzen einen unterschiedlichen Aufbau.

Die Offset-Tabelle eines Moduldeskriptors umfaßt lediglich die (konstanten) Offsets aller modulglobalen Zeiger. Die Offset-Tabelle eines Prozedurdeskriptors kann neben den konstanten Zeiger-Offsets von Parametern und lokalen Variablen zusätzlich noch die Zeiger-Offsets besitzen, die für offene Array-Parameter relevant sind. Die Offset-Tabelle eines Typdeskriptors besteht für Record-Typen und nicht-offene Array-Typen lediglich aus den zum Typ gehörenden Zeiger-Offsets oder für offene Array-Typen aus den zum Elementtyp gehörenden Zeiger-Offsets. Der prinzipielle Aufbau ist durch nachfolgende Grammatikregeln skizziert. Alle Terminale sind Langworte, mit Ausnahme von `0`, welches lediglich ein Byte umfaßt. Weitere Details werden im Rahmen der Freispeicherverwaltung erläutert.

```
ModuleOffsetTab   =  Fixed .
ProcedureOffsetTab =  Fixed { Open } .
TypeOffsetTab     =  nullByte endMark | nullByte ( Open | Fixed ) descAddr .
Open              =  openMark offset elemSize { offset } endMark .
Fixed             =  offset { offset } endMark .
nullByte          =  0 .
openMark          =  -2 .
endMark           =  -1 .
offset            =  „Zeiger-Offset“.
descAddr          =  „Adresse des aktuellen Deskriptors“.
```

Einem Deskriptor für einen Record- oder (offenen) Array-Typ auf `Label$D` folgt auf `Label$I` immer eine Initialisierungsroutine. Diese dient nach dem Anlegen eines Objekts im Heap der Initialisierung von Zeiger- und Prozedurfeldern.

1.4.5.1 Deskriptoren von Record-Typen

Ein Deskriptor für einen Record-Typ `T` ist eine Erweiterung des allgemeinen Deskriptors. Er besitzt zusätzlich eine Tabelle der Basistypen von `T` und eine Tabelle der an `T` gebundenen Prozeduren.

Adresse	*Element*
`Label$D-48`	Prozedurtabelle
`Label$D-44`...`Label$D-16`	Basistyptabelle
`Label$D-12`	Verweis auf den Objektnamen
`Label$D-8`	Objektgröße
`Label$D-4`	Verweis auf die Skipper-Routine
`Label$D`	Offset-Tabelle

Die Tabelle der Basistypen besitzt eine feste Größe von 8 Elementen, welche mit $\texttt{type}_0$ bis $\texttt{type}_7$ bezeichnet werden. Die feste Größe dieser Tabelle ist eine Implementierungsbeschränkung. Die gewählte Größe wird jedoch von Pfister, Heeb und Templ [1991] als groß genug für alle praktischen Belange angesehen. Sei e die Erweiterungsstufe von `T` und $i \in [0, 7]$, so enthält $\texttt{type}_i$ die Deskriptoradresse des Basistyps von `T` mit der Erweiterungsstufe i, falls $i \leq e$ gilt und `NIL` sonst. Hieraus folgt insbesondere, daß $\texttt{type}_e =$ `Label$D` ist. Jeder Record-Typ, der ohne expliziten Basistyp deklariert ist, besitzt die Erweiterungsstufe 0.

Die Prozedurtabelle besitzt auf `L$D-48` ihr erstes Element. Sie wächst in Richtung niedriger Adressen. Die Elemente enthalten die Adressen der an `T` gebundenen Prozeduren.

Abbildung 4 zeigt die Deskriptoren der beiden Record-Typen `T1` und `T2` aus dem Modul `M`.

```
MODULE M;
   TYPE T1*= RECORD
               s: ARRAY 4 OF CHAR;
             END;
        T2*= RECORD(T1)
               p: POINTER TO T1;
               f: BOOLEAN;
               a: ARRAY 2,3 OF POINTER TO T1;
             END;
   PROCEDURE (VAR r:T1)P1*; END P1;
   PROCEDURE (VAR r:T1)P2*; END P2;
   PROCEDURE (VAR r:T1)P3*; END P3;
   PROCEDURE (VAR r:T2)P3*; END P3;
END M.
```

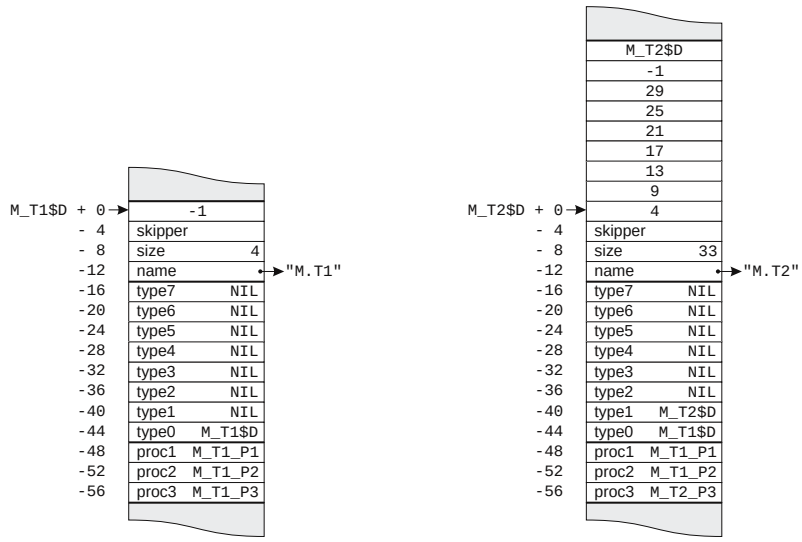

Abbildung 4: Deskriptoren von Record-Typen

1.4.5.2 Deskriptoren von offenen Array-Typen

Ein Deskriptor für einen offenen Array-Typ ist ebenfalls eine Erweiterung des allgemeinen Deskriptors. Die zusätzliche Objektinformation auf `Label$D-8` gibt die Anzahl der offenen Dimensionen des Array-Typs an. Zusätzlich besitzt der Deskriptor auf `Label$D-16` ein Feld, welches die Größe des Elementtyps in Byte enthält. Somit hat ein offener Array-Deskriptor folgenden Aufbau:

Adresse	*Element*
`L$D-16`	Elementgröße
`L$D-12`	Verweis auf den Objektnamen
`L$D-8`	Anzahl offener Dimensionen
`L$D-4`	Verweis auf die Skipper-Routine
`L$D`	Offset-Tabelle

1.4.6 Module

Ein übersetztes Oberon-2-Modul benutzt nur den Code- und BSS-Abschnitt. Die sogenannten *Speicherkonstanten* werden ausschließlich im Codeabschnitt abgelegt, da dieser zur Laufzeit schreibgeschützt ist. Der Datenabschnitt wird nicht verwendet, da die Initialisierung der Variablen explizit erfolgt.

Speicherkonstanten sind entweder Zeichenketten oder konstante Werte vom Typ `REAL` oder `LONGREAL`, die innerhalb von Ausdrücken als Operanden benutzt werden. Speicherkonstanten mit gleichem Wert werden pro Modul nur einmal abgelegt.

Wenn das Modul den Namen `M` hat, beginnt der BSS-Abschnitt mit der Adresse, die durch das globale Label `M$G` gekennzeichnet ist, und beinhaltet

- auf Adresse `M$G` einen Zeiger auf den Moduldeskriptor im Codeabschnitt,
- auf Adresse `M$G+4` einen Zeiger auf den BSS-Abschnitt des nachfolgenden Moduls,

- ab Adresse `M$G+8` alle globalen Variablen in der Reihenfolge ihrer Deklaration im Oberon-2-Quellmodul.

Die globalen Variablen sind in Abhängigkeit ihrer Größe auf bestimmte Adressen ausgerichtet.

Größe	*Ausrichtung*
1,3	keine
2	auf nächste gerade Adresse
$\geq$4	auf nächste durch 4 teilbare Adresse

Der Codeabschnitt enthält (in der Reihenfolge aufsteigender Adressen)

- den Moduldeskriptor ab Adresse `M$D`,
- den Code aller Prozeduren des Moduls,
- den Deskriptor des Modulrumpfs,
- die Anweisungen des Modulrumpfs ab Adresse `M$I`,
- die Speicherkonstanten.

Der Anfang jedes aufgeführten Objekts ist auf die nächste durch 4 teilbare Adresse ausgerichtet. Die Labels `M$D` und `M$I` sind global.

1.4.7 Wurzelmodul

Das Wurzelmodul wird vom Compiler automatisch erzeugt. Sein Name ergibt sich aus dem Namen des Hauptmoduls und dem Präfix '_'. Es besteht lediglich aus einem Codeabschnitt, in dem sich der Deskriptor seines Modulrumpfs sowie der Modulrumpf selbst befinden. Der Anfang des Modulrumpfs wird mit dem konstanten Label `_$I` bezeichnet, damit auf ihn aus dem Laufzeitsystem heraus Bezug genommen werden kann. Zusätzlich stellt es über die symbolische Adresse `_FirstModuleTDesc` die Referenz des Hauptmoduldeskriptors zur Verfügung, die ebenfalls vom Laufzeitsystem bei der Ausgabe von Laufzeitfehlermeldungen benutzt wird.

Im Modulrumpf des Wurzelmoduls wird für jedes Modul eines Programms die Adresse des Moduldeskriptors in den Anfang des zugehörigen BSS-Abschnitts eingetragen und die Verkettung der BSS-Abschnitte aller Module gemäß der Modulfolge hergestellt. Dies dient der automatischen Freispeicherverwaltung. Anschließend werden die Modulrümpfe aller Module in der erforderlichen Reihenfolge aufgerufen.

1.4.8 Prozeduren

Prozeduren sind im Codeabschnitt in der Reihenfolge des textuellen Erscheinens ihrer Rümpfe im Oberon-2-Quellmodul codiert. Diese Codierung besteht für eine Prozedur aus

- dem Prozedurdeskriptor ab Adresse `Label$D`,
- dem Prolog ab Adresse `Label`,
- dem Code der Anweisungen des Rumpfs,
- dem Epilog.

Für eine vom Modul `M` exportierte Prozedur `P` hat dabei `Label` die Form `M_P`. Nichtexportierte Prozeduren besitzen dagegen ein lokales Label.

Nach WILHELM und MAURER [1992] kann man die Folge der Prozeduraufrufe, die während der Ausführung eines Programms entstehen, als geordneten Baum ansehen (S. 31). Jeder Knoten bis auf den Wurzelknoten ist markiert mit einem Prozedurnamen `P`. Eine Markierung `P` kann mehrfach im Baum auftreten. Jedes Auftreten von `P` wird *Inkarnation* von `P` genannt. Eine Inkarnation ist *aktiv*, solange die Anweisungen des Prozedurrumpfs der zu dieser Inkarnation gehörenden Prozedur ausgeführt werden. Eine Inkarnation ist *lebendig*, wenn sie aktiv ist oder auf dem Weg von der Wurzel des Baums zur aktiven Inkarnation liegt.

Jede lebendige Inkarnation einer Prozedur besitzt im Stack ein *Aktivierungssegment*. Auf das Aktivierungssegment der aktiven Inkarnation verweist der *Basiszeiger*, dessen Wert im Register `ebp` gehalten wird. Die Einrichtung eines Aktivierungssegments wird vom Aufrufer einer Prozedur durch die Parameterübergabe und die Sicherung der Rücksprungadresse begonnen und vom Prolog der aufgerufenen Prozedur vervollständigt.

Abbildung 5 zeigt den prinzipiellen Aufbau eines Aktivierungssegments. Unterhalb der durch den Aufrufer abgelegten Parameter befindet sich die Rücksprungadresse, die durch die Call-Instruktion des Prozeduraufrufs dort abgelegt wurde. Der aktuelle Basiszeiger verweist auf den gesicherten Basiszeiger des Aufrufers. Das darunterliegende Stack-Element wird als *Prozedurmarke* (procedure tag) bezeichnet und enthält die Adresse des Prozedurdeskriptors. Danach folgt das Display zum Zugriff auf Objekte umfassender Prozeduren. Displays werden im folgenden Abschnitt näher erläutert. Temporäre Variablen werden für die For-Anweisung und für bestimmte Fließkommaausdrücke benötigt. Am unteren Ende des Aktivierungssegments befinden sich lokale Kopien von bestimmten Parametern.

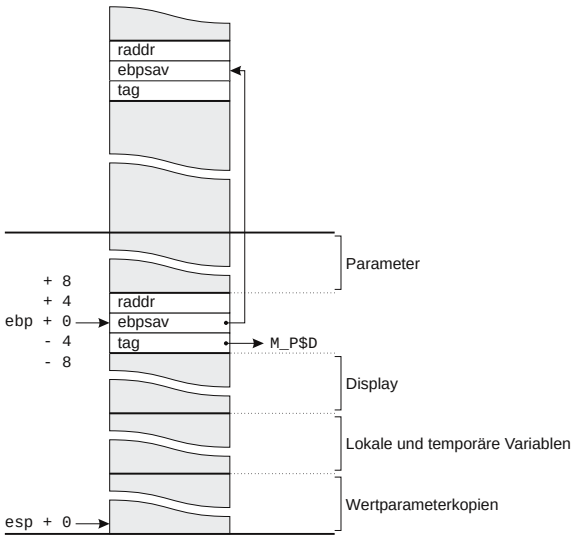

Abbildung 5: Prinzipieller Aufbau eines Aktivierungssegments

1.4.9 Displays

Displays ermöglichen den schnellen Zugriff auf lokale Variablen und Parameter umgebender Prozeduren. Die Inkarnation einer Prozedur der Schachtelungstiefe d besitzt ein Display der Größe d. Dieses Display ist ein Array, von d Zeigerelementen, die mit $\texttt{disp}_1$, ..., $\texttt{disp}_d$ bezeichnet werden. Die Schachtelungstiefe einer modulglobalen Prozedur ist 1. Die Adresse des Aktivierungssegments der zuletzt aktiven Inkarnation einer umgebenden Prozedur der Schachtelungstiefe i befindet sich in $\texttt{disp}_i$.

Der Aufbau des Displays erfolgt im Prolog der aufgerufenen Prozedur `P`. Sei d die Schachtelungstiefe von `P` und c die Schachtelungstiefe der aufrufenden Prozedur `Q` (mit $1 \leq d \leq c+1$). Dann werden die Display-Elemente $\texttt{disp}_1$, ..., $\texttt{disp}_{d-1}$ der zuletzt aktiven Inkarnation von `Q` kopiert, so daß für den Fall $d = c+1$ in $\texttt{disp}_c$ von `Q` bereits die Adresse des Aktivierungssegments von `Q` eingetragen sein sollte. Aus diesem Grund wird im Prolog einer Prozedur die Adresse ihres eigenen Aktivierungssegments im letzten Display-Element eingetragen, so daß dieses Element von einer lokalen Prozedur mit kopiert werden kann. Dagegen wird für Prozeduren, die keine lokalen Prozeduraufrufe enthalten, das letzte Display-Element nicht angelegt.

Um den Aufbau von Displays zu verdeutlichen, zeigt Abbildung 6 die Aktivierungssegmente bei Ausführung der nachstehenden Module `M1` und `M2` zur Laufzeit an den mit `(**)` markierten Stellen.

```
MODULE M1;
   TYPE T=LONGINT;

   PROCEDURE P*(Pp:T); VAR Pv:T;

      PROCEDURE S(Sp:T); VAR Sv:T;
      BEGIN (**)
      END S;

      PROCEDURE Q(Qp:T); VAR Qv:T;

         PROCEDURE R(Rp:T); VAR Rv:T;
         BEGIN S(4)
         END R;

      BEGIN R(3)
      END Q;

   BEGIN Q(2)
   END P;

BEGIN P(1)
END M1.
```

```
MODULE M2;
   TYPE T=LONGINT;

   PROCEDURE P*(Pp:T);

      PROCEDURE Q(Qp:T);

         PROCEDURE R(Rp:T);
         BEGIN
          IF Qp=2 THEN
             Q(4)
          ELSE (**)
          END
         END R;

      BEGIN R(3)
      END Q;

   BEGIN Q(2)
   END P;

BEGIN P(1)
END M2.
```

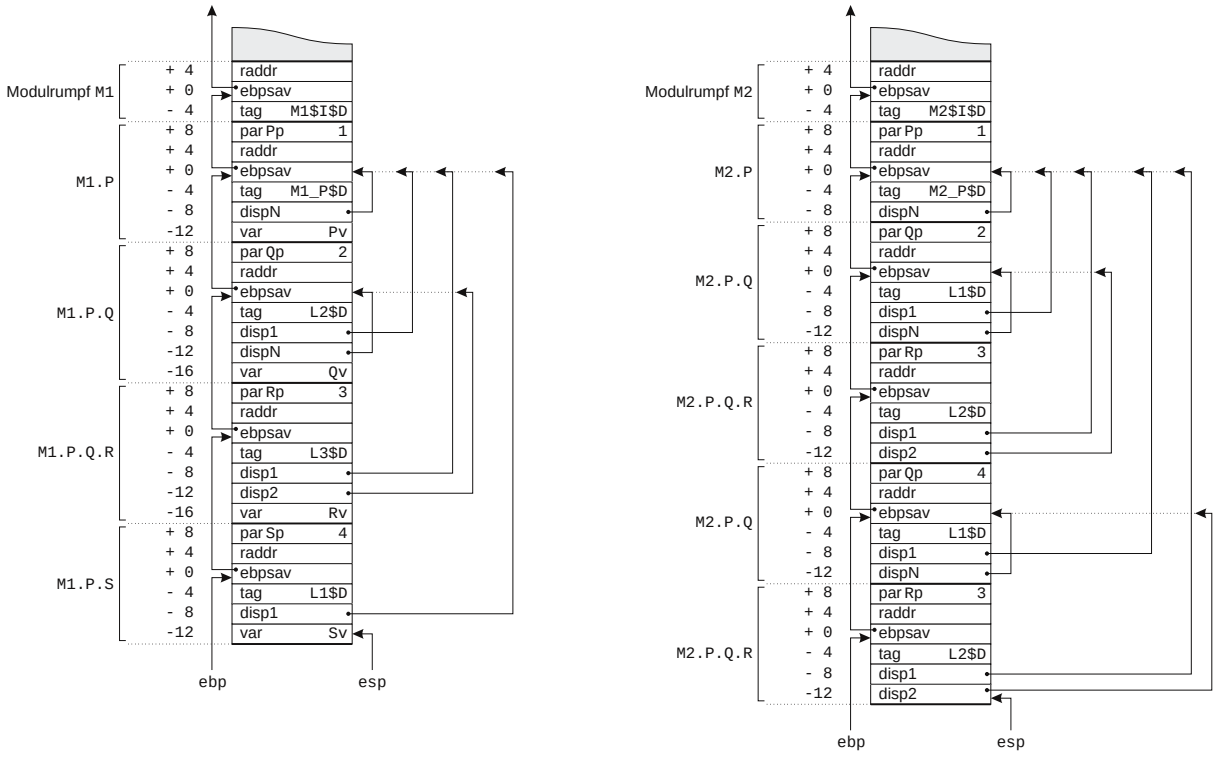

Abbildung 6: Display-Aufbau bei verschachtelten Prozeduren

1.4.10 Parameter

Die Behandlung der einzelnen aktuellen Parameter im Rahmen der Parameterübergabe geschieht in umgekehrter Reihenfolge der zugehörigen Deklarationen im Prozedurkopf, und es liegt in der Verantwortung des Aufrufers, den durch die Parameter belegten Speicher wieder freizugeben. Dies ist die bei der Programmiersprache C übliche Übergabekonvention wodurch die Betriebssystemanbindung erleichtert wird. Die Übergabe eines aktuellen Parameters erfolgt durch Ablage seines Werts oder seiner Adresse auf dem Stack.

Ist der formale Parameter ein Val-Parameter mit Standardtyp, Zeigertyp oder Prozedurtyp, erfolgt die Übergabe durch *call-by-value*. Ebenso bei einem strukturierten Typ, dessen Größe nicht mehr als 4 Byte beträgt und der kein offenes Array ist. Hierbei werden die Werte von Typen mit Größen kleiner als 4 Byte vorzeichenlos auf 4 Byte erweitert, damit alle Stack-Elemente immer auf einer durch 4 teilbaren Adresse liegen.

Ist der formale Parameter von einem strukturierten Typ, dessen Größe 4 Byte übersteigt, oder ein offenes Array, so erfolgt die Übergabe immer durch *call-by-reference*. Im Prolog der aufgerufenen Prozedur wird eine lokale Kopie des übergebenen Werts angelegt, falls es sich um einen formalen Val-Parameter handelt.

Var-Parameter werden stets durch *call-by-reference* übergeben.

Für einen formalen offenen Array-Parameter werden vor der Übergabe des aktuellen Parameters weitere implizite Parameter übergeben. Angefangen mit der innersten offenen Dimension wird für jede offene Dimension die entsprechende Länge des aktuellen Arrays übergeben. Für mehrdimensionale offene Arrays wird zusätzlich das Produkt aller übergebenen Längen als letzter impliziter Parameter auf den Stack gelegt. Hierdurch kann für das Anlegen einer lokalen Kopie durch den Prolog die Berechnung der Anzahl der zu kopierenden Bytes möglichst einfach erfolgen, und eine einheitliche Behandlung aller offenen eindimensionalen Arrays bleibt trotzdem gewährleistet.

Für einen formalen Var-Parameter mit Record-Typ wird nach der Übergabe des aktuellen Parameters zusätzlich eine *Typmarke* (type tag) als impliziter Parameter übergeben. Diese Typmarke verweist auf den Typdeskriptor des aktuellen Records.

Der Empfängerparameter von gebundenen Prozeduren wird als „normaler“ Parameter angesehen, der vor allen anderen Parametern deklariert ist, also zuletzt übergeben wird.

Bei Ausführung des nachstehenden Moduls `M` zeigt Abbildung 7 die Lage der Parameter im Aktivierungssegment der Prozedur `P` zur Laufzeit an der Stelle `(**)`.

```
MODULE M;
   TYPE T1= RECORD
             s: ARRAY 4 OF CHAR;
            END;
        T2= RECORD(T1)
             f: BOOLEAN;
            END;
   VAR  v1: T1;
        v2: T2;
        a2: ARRAY 2,5 OF CHAR;

   PROCEDURE P(    si: SHORTINT;
                   in: INTEGER;
                   li: LONGINT;
                   lr: LONGREAL;
                   p1: T1;
                   p2: T2;
                   s1: ARRAY OF CHAR;
                   s2: ARRAY OF ARRAY OF CHAR;
               VAR r1: T1);
      VAR l1,l2: SET;
   BEGIN
    (**)
   END P;

BEGIN
 P(1,2,3,4.0D0,v1,v2,"12345",a2,v2);
END M.
```

Bei formalen Val-Parametern, die durch *call-by-reference* übergeben werden, wird aus Optimierungsgründen keine lokale Kopie angelegt, falls der Parameter innerhalb der Prozedur nicht durch Zuweisungen oder durch Übergabe an eine weitere Prozedur verändert werden kann. Insbesondere wird bei Anwendung der Funktionsprozedur `SYSTEM.ADR` das Argument nicht verändert, so daß sich die Betriebssystemanbindung durch diese Optimierung vereinfacht.

1.4.11 Funktionsresultate

Da in Oberon-2 keine Record- und Array-Typen als Resultatstypen von Funktionsprozeduren zulässig sind, werden Resultate ausschließlich über ein Prozessorregister zurückgeliefert (vgl. 1.4.4 Typgrößen, S. 15):

Resultatstyp	*Register*
CHAR, BOOLEAN, SHORTINT, SYSTEM.BYTE	al
INTEGER	ax
LONGINT, SET, POINTER, PROCEDURE, SYSTEM.PTR	eax
REAL, LONGREAL	st(0)

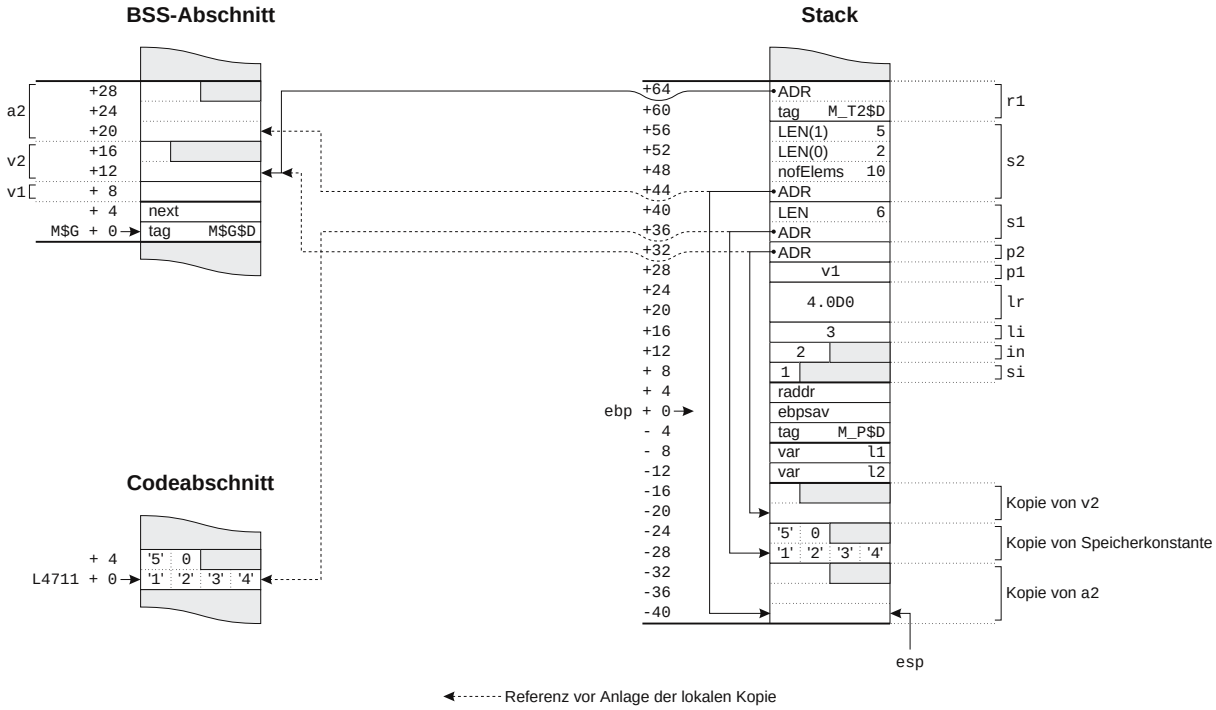

Abbildung 7: Parameter im Aktivierungssegment

1.4.12 Heap

Der Heap besteht aus einer Folge von unterschiedlich großen Speicherbereichen, welche unmittelbar hintereinander liegen und als Blöcke bezeichnet werden. Die Folge der Blöcke wird Blockfolge genannt. Die Größe der Blöcke beträgt 8 Byte oder ein Vielfaches von 8 Byte, damit beim Teilen von Blöcken niemals ein „Blockrest“ bleibt, dessen Größe weniger als 8 Byte beträgt. Der für den Heap benutzte Adreßbereich ist durch die Laufzeitvariablen `_HeapStart` (erstes Langwort) und `_HeapEnd` (letztes Langwort) gegeben. Die Anfangsadresse eines Blocks +4 muß durch 8 teilbar sein und wird im folgenden als *Blockanfang* bezeichnet. Es existieren unterschiedliche Arten von Blöcken, und zwar Verwaltungsblöcke, freie Blöcke und belegte Blöcke. Die Erkennung der Blockart geschieht über ein Verwaltungsfeld `tag` auf der Anfangsadresse, das auf einen Deskriptor verweist.

Verwaltungsblöcke dienen der einfacheren Verwaltung der Blockfolge, die durch den Ankerblock mit Blockanfang `_HeapStart+4` eingeleitet und durch den Wächterblock mit Blockanfang `_HeapEnd` abgeschlossen wird. Alle freien und belegten Blöcke befinden sich somit zwischen Anker- und Wächterblock. Die Größe beider Blöcke beträgt jeweils 8 Byte. Anker- und Wächterblock sind die einzigen Verwaltungsblöcke. Durch sie besitzen alle anderen Blöcke der Blockfolge immer einen Vorgänger- und einen Nachfolgerblock.

Freie Blöcke existieren in zwei verschiedenen Arten. Während der (normale) freie Block eine Größe von mehr als 8 Byte hat, die auf seinem Blockanfang +4 als Langwort eingetragen ist, besteht der kleine freie Block genau aus 8 Byte.

Der Ankerblock und jeder freie Block besitzt an seinem Blockanfang einen Verweis auf den Blockanfang des ersten bzw. nächsten freien Blocks der Blockfolge. Dadurch ergibt sich eine einfach verkettete Liste von freien Blöcken.

Belegte Blöcke sind das Ergebnis eines (erfolgreichen) Aufrufs der vordeklarierten Prozedur `NEW` oder der im Modul `SYSTEM` deklarierten Prozedur `NEW`. Ein belegter Block hat eine Größe, die sich aus der über (`SYSTEM.`)`NEW` angeforderten Größe und der Größe von zusätzlichen Verwaltungsfeldern zusammensetzt. Die Blockgröße wird auf den nächsten durch 8 teilbaren Wert erhöht. Es werden verschiedene Arten von belegten Blöcken unterschieden:

- Ein *statischer Block* ist das Ergebnis eines Aufrufs der vordeklarierten Prozedur `NEW(p)`, wobei die Zeigerbasis von `p` ein Record- oder ein statischer Array-Typ ist. Die Anfangsadresse des angelegten Objekts ist gleich dem Blockanfang.
- Ist der Zeigerbasistyp von `p` ein eindimensionales offenes Array, so erzeugt `NEW(p,x)` einen *offenen Block*, der auf Blockanfang das Verwaltungsfeld `LEN` mit Inhalt `x` besitzt. Die Anfangsadresse des angelegten Arrays ist Blockanfang +4.
- Für ein mehrdimensional offenes Array als Zeigerbasis von `p` führt der Aufruf von `NEW(p,`$\mathtt{x}_0,\ldots,\mathtt{x}_{n-1}$`)` zur Anlage eines *mehrfach-offenen Blocks*. Dieser enthält auf Blockanfang das Verwaltungsfeld `nofElems` mit dem Wert $\prod_{i=0}^{n-1} x_i$ und auf Blockanfang $+4*(i+1)$ die Verwaltungsfelder `LEN(`i`)` mit den Werten x_i, mit $0 \leq i < n$. Die Anfangsadresse des Arrays ergibt sich aus Blockanfang $+4*(n+1)$.

- Durch den Aufruf von `SYSTEM.NEW(p,x)` wird ein *Systemblock* angelegt. Dieser besitzt als letztes Langwort ein Verwaltungsfeld, das die Blockgröße enthält, und als vorletztes Langwort ebenfalls ein Verwaltungsfeld, welches die Adresse einer speziellen Skipper-Routine enthält, die nur Systemblöcken zugeordnet ist. Der Anfang des angeforderten Speichers ist identisch mit dem Blockanfang.

Die Tatsache, daß sich bei (mehrfach) offenen Blöcken die Anfangsadresse des jeweiligen Arrays vom Blockanfang unterscheidet, wird in der Codierung eines Bezeichners, über den der Zugriff erfolgt, entsprechend berücksichtigt.

Das Verwaltungsfeld `tag` verweist auf einen Deskriptor, der bei freien Blöcken lediglich aus der Adresse einer Skipper-Routine auf Deskriptoradresse -4 besteht. Bei statischen und (mehrfach) offenen Blöcken verweist `tag` auf den zugehörigen Typdeskriptor. Bei einem Systemblock enthält `tag` die Adresse des letzten Langworts des Blocks. Im Ankerblock hat `tag` den Wert 0.

Abbildung 8 zeigt eine beispielhafte Belegung des Heaps mit allen existierenden Blockarten nach dem Aufruf der Speicherbereinigung im Modul `M`.

```
MODULE M;
   IMPORT SYSTEM, Storage;
   VAR s:POINTER TO ARRAY OF CHAR;
       f:POINTER TO ARRAY 1 OF CHAR;
       v:SYSTEM.PTR;
       p:POINTER TO ARRAY 13 OF CHAR;
       a:POINTER TO ARRAY OF ARRAY OF INTEGER;
BEGIN
   NEW(s,60);
   SYSTEM.NEW(f,11);
   SYSTEM.NEW(v,5);
   SYSTEM.NEW(f,19);
   NEW(p);
   NEW(f);
   NEW(a,3,4);
   NEW(f);
   Storage.GC; (* Aufruf der Speicherbereinigung *)
END M.
```

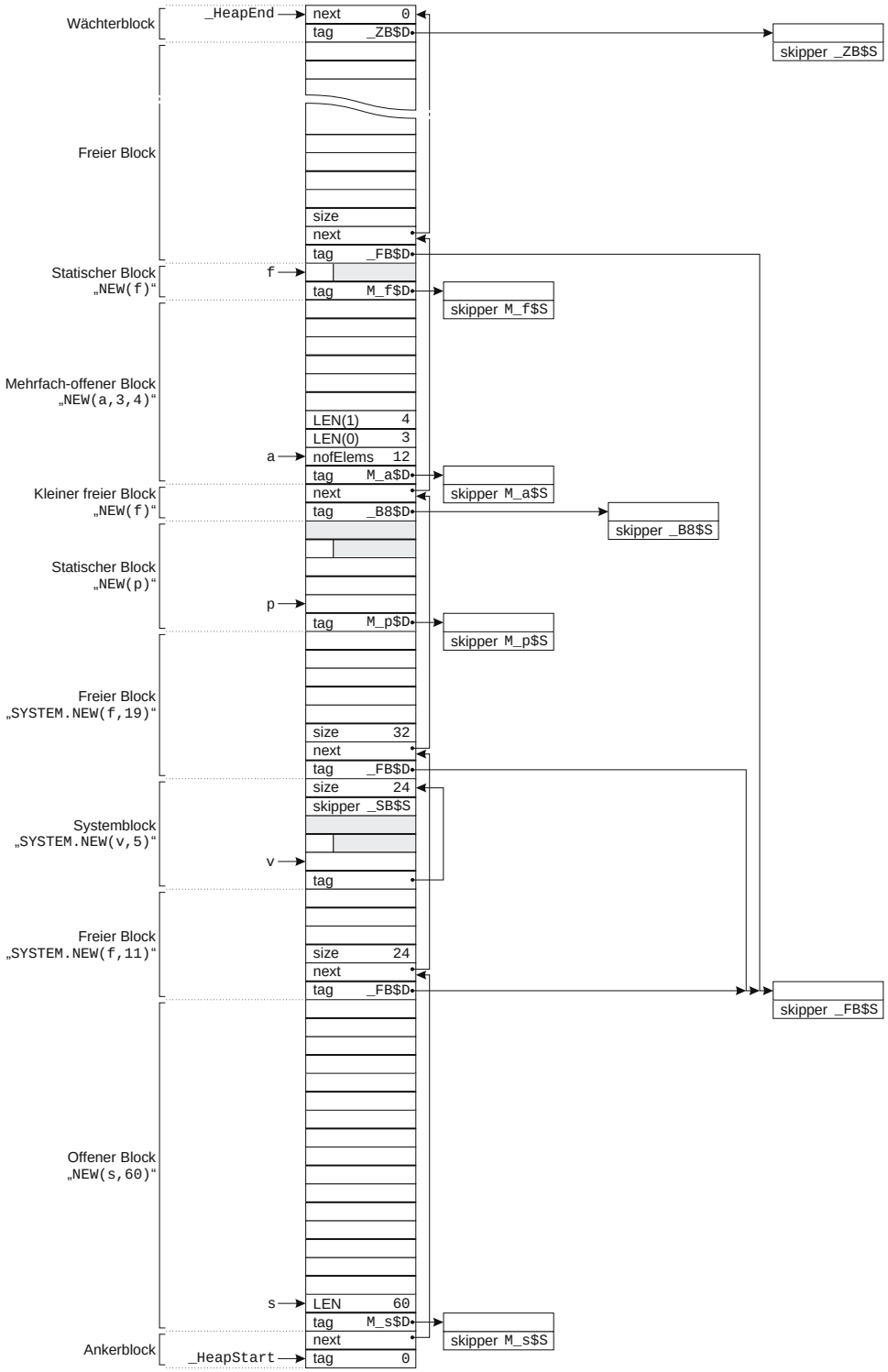

Abbildung 8: Beispielhafte Belegung des Heaps

2 BEG – Backend-Generator

Dieses Kapitel beschreibt das Wergzeug BEG in der Version 1.75 für das Betriebssystem Linux und stützt sich hierbei auf die in dieser Distribution enthaltene Dokumentation von Emmelmann [1989]. Diese Dokumentation bezieht sich jedoch leider auf eine wesentlich ältere Version von BEG, so daß sie nur in bezug auf die grundsätzliche Handhabung von BEG von Relevanz war. Das Fehlen einer aktuellen Dokumentation ist darin begründet, daß die eingesetzte Distribution von BEG verfügbar gemacht wurde, um den Codeerzeuger von Mocka generieren zu können. Da eine neuere Dokumentation nicht erhältlich war[1], lieferte die Backend-Beschreibung (CGD) von Mocka zusammen mit ihrer Dokumentation von Hopp [1993] für die konkrete Benutzung von BEG wertvolle Hinweise.

Aufgabe eines Backend-Generators ist es, aus einer Spezifikation einen Codeerzeuger zu generieren, welcher Ausdrucksbäume der Zwischensprache in Anweisungen der Zielsprache überführt. Die Codeerzeugung umfaßt dabei die Auswahl geeigneter Befehle der Zielsprache ebenso wie die Registerallokation.

Der Backend-Generator BEG generiert aus einer CGD genannten Codeerzeuger-Beschreibung (Code Generator Description) einen Codeerzeuger, der durch mehrere Module in der Werkzeugzielsprache (Modula-2) realisiert ist. Die CGD umfaßt unter anderem die Deklaration der Zwischensprache, in der die zu übergebenden Ausdrucksbäume formuliert sind sowie eine Reihe von Regeln. Eine Regel besteht im wesentlichen aus einem Muster, welches einen Ausschnitt des Ausdrucksbaums beschreibt, aus den Befehlen der Zielsprache, die für diesen Ausschnitt zu erzeugen sind und einer Kostenangabe, welches ein Maß für die Ausführungskosten der erzeugten Befehle darstellt. Die Muster definieren Bäume, in denen jeder Knoten mit einer Regel markiert ist. Ein solcher Baum ist ein Überdeckungsbaum für einen übergebenen Ausdrucksbaum, wenn durch Ersetzung jedes (Regel-)Knotens durch den in der Regel angegebenen Ausdrucksbaumausschnitt der Ausdrucksbaum entsteht. Die Muster der Regeln sind im allgemeinen so formuliert, daß es für einen Ausdrucksbaum verschiedene Überdeckungsbäume gibt. Die Auswahl des „besten“ Überdeckungsbaums erfolgt anhand seiner Kosten, die sich aus der Summe der Kosten seiner Knotenmarkierungen ergeben. Ein Überdeckungsbaum wird im folgenden auch *Überdeckung* genannt. Ein Überdeckungsbaum mit minimalen Kosten wird als *Minimalüberdeckung* bezeichnet.

Bei der Traversierung des attributierten abstrakten Syntaxbaums im Frontend wird eine Folge von Ausdrucksbäumen konstruiert. Ein Ausdrucksbaum umfaßt etwa eine Anweisung der Quellsprache und wird durch fortgesetzten Aufruf von Konstruktorprozeduren aus einem von BEG generierten Modul aufgebaut. Konzeptionell beginnt die Konstruktion einer Minimalüberdeckung, nachdem ein Ausdrucksbaum vollständig erzeugt wurde.

Ist eine Minimalüberdeckung gefunden, so erfolgt die Vergabe der konkreten Maschinenregister. Hierfür stellt BEG zwei verschiedene Methoden der Registerallokation zur Wahl, die *generelle* und die *on the fly* Registerallokation. Darüberhinaus kann die Registerallokation auch von Hand implementiert werden. In der vorliegenden Arbeit wurde die Methode

[1] Eine Beschreibung des aktuellen Stands der Entwicklung von BEG wird in einem sogenannten *BEG Status Report* festgehalten, der laut Aussage von H. Emmelmann ein Geheimdokument des COMPARE Esprit Projekts ist und daher nicht verschickt werden darf. Eine neuere Anleitung zu BEG 1.75 existiert nicht. Lediglich ein zugesandtes Papier von Emmelmann [1995] führt einige neuere Eigenschaften von BEG stichpunktartig auf.

der *generellen* Registerallokation verwendet, da nur diese die speziellen Eigenschaften des Registersatzes der Zielmaschine unterstützt. Bei der Spezifikation der Regeln wird von einer unendlichen Registermenge ausgegangen. Stehen während der Registerallokation nicht genügend Register zur Verfügung, muß der Code um Befehle zum Auslagern einiger Registerinhalte in den Speicher ergänzt werden. Die Kosten dieser Ergänzung können jedoch keine Berücksichtigung mehr hinsichtlich der intendierten Minimalüberdeckung finden.

Schließlich wird in der Ausgabephase eine postorder-Traversierung des Überdeckungsbaums durchgeführt, wobei die in den Regeln angegebenen Codefragmente abgesetzt werden. Eine schematische Darstellung der einzelnen Phasen eines von BEG generierten Codeerzeugers für die Bearbeitung eines Ausdrucksbaums zeigt Abbildung 9. Der Codeerzeuger arbeitet einen übergebenen Ausdrucksbaum vollständig ab und gibt den Speicher wieder frei, bevor der nächste Baum konstruiert werden kann. Dies bedeutet insbesondere, daß es nicht möglich ist, vor dem abschließenden Aufruf eines Top-level-Operators einen zusätzlichen Ausdrucksbaum aufzubauen und durch einen Top-level-Operator codieren zu lassen.

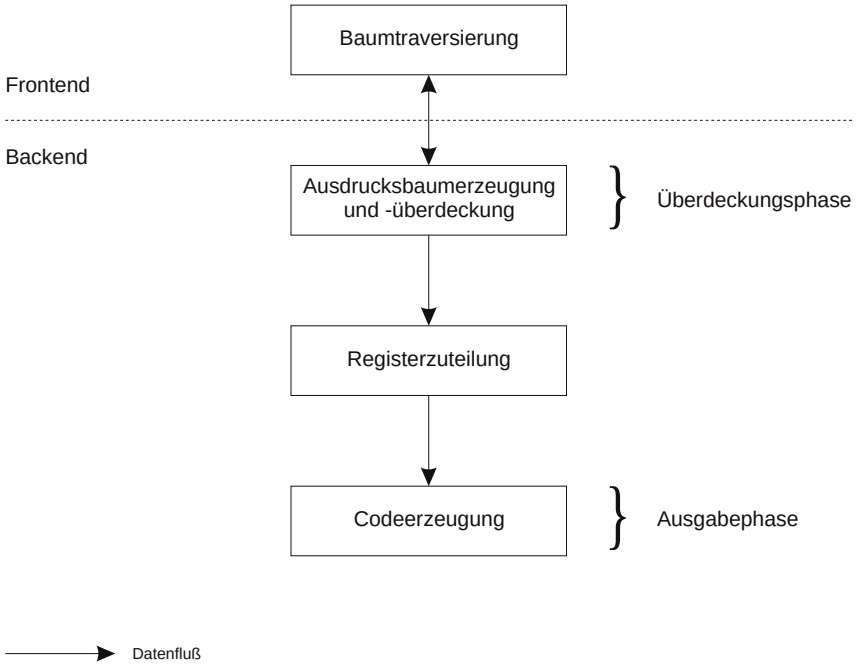

Abbildung 9: Phasen des von BEG generierten Codeerzeugers bei der Bearbeitung eines Ausdrucksbaums

2.1 Allgemeiner Aufbau einer CGD

Eine CGD wird in der werkzeugspezifischen Spezifikationssprache BEGL (BEG Language) geschrieben. Sie gliedert sich in mehrere Abschnitte, die sich an die dargestellten Phasen anlehnen. Sie beginnt mit der Beschreibung der Zwischensprache, welche die Schnittstelle des generierten Codeerzeugers zum Frontend bildet. Danach folgt eine Beschreibung der Register der Zielmaschine, falls die Registerallokation durch BEG generiert werden soll.

Es folgen die Nichtterminale und die Regeln. Die Nichtterminale repräsentieren u.a. die Adressierungsarten der Zielmaschine und werden an entsprechender Stelle näher erläutert.

Den Abschluß einer CGD bildet der *Einfügungsbereich*, in welchem Erweiterungen der von BEG generierten Module in Form von Code der Werkzeugzielsprache stehen. Hier müssen für die Registerallokation notwendige Deklarationen eingefügt werden.

Die allgemeine Form einer CGD ist:

```
CGD  =  CODE_GENERATOR_DESCRIPTION ident ';'
        IntermediateCode
        RegisterSet
        Nonterminals
        [ Rules ]
        [ Insertions ]
        END CODE_GENERATOR_DESCRIPTION ident '.'.
```

2.1.1 Zwischensprache

Die Definition der Zwischensprache erfolgt durch Aufzählung der in den Ausdrucksbäumen verwendeten Operatoren. Für jeden Operator muß dessen Signatur durch Angabe der Typen aller Operanden und des Ergebnistyps definiert werden. Die Namen der verwendeten Typen werden als Nichtterminale vor der Aufzählung der Operatoren in einem eigenen Abschnitt aufgelistet. Operatoren ohne Ergebnistyp werden *Top-level-Operatoren* genannt. Sie bilden die Wurzeln der konstruierten Ausdrucksbäume.

Eine Operatordefinition beginnt mit dem Namen des Operators. Dahinter können optional noch Attributdefinitionen folgen. Über diese Attribute werden Daten vom Frontend im Operatorknoten gespeichert. Innerhalb der Regeln kann dann ein Zugriff auf diese Daten erfolgen. Die Definition der Attribute entspricht der Parameterdeklaration in der Werkzeugzielsprache. Nach den Attributdefinitionen erfolgt die Angabe der Argumenttypen getrennt durch das Zeichen '*'. Die Anzahl der Argumenttypen bestimmt die Stelligkeit des Operators. Kommutative Operatoren können durch Angabe des Zeichens '+' zwischen den beiden Argumenttypen gekennzeichnet werden. Dies bewirkt eine Sonderbehandlung von Regeln, deren Muster diesen Operator enthalten (s.u.). Besitzt ein Operator einen Ergebnistyp, wird dieser hinter dem Symbol '->' angegeben.

```
IntermediateCode  =  INTERMEDIATE_REPRESENTATION
                     NONTERMINALS ident { ',' ident } ';'
                     OPERATORS { OperDefinition } .
OperDefinition    =  ident [ Attributes ] [ Operands ] [ '->' [ ident ] ] ';' .
Attributes        =  '(' ident ':' Type { ';' ident ':' Type } ')' .
Operands          =  ident '+' ident | ident {'*' ident } .
Type              =  „Modula-2 Datentyp“ .
```

Beispiel:

```
CODE_GENERATOR_DESCRIPTION Cons;
INTERMEDIATE_REPRESENTATION
NONTERMINALS
  Address, Data;
OPERATORS
  IntegerConst     (val:LONGINT)               -> Data   ;
  LocalVariable    (adr:LONGINT)               -> Address;
  SimpleAssignment              Address * Data           ;
...
END CODE_GENERATOR_DESCRIPTION  Cons.
```

BEG generiert aus der Spezifikation der Zwischensprache ein Modula-2-Modul, dessen Name der hinter dem Schlüsselwort `CODE_GENERATOR_DESCRIPTION` angegebene Name ist. Dieses Modul wird vom Frontend zum Aufbau der Ausdrucksbäume importiert. Die exportierten Datentypen und Prozeduren werden nach folgendem Schema aus der Spezifikation der Zwischensprache generiert:

- Für jeden Operandentyp wird ein gleichnamiger opaquer Datentyp generiert.
- Für jeden Operator wird eine Prozedur gleichen Namens generiert. Eventuelle Attribute werden zu Val-Parametern der Prozedur umgewandelt. Für sämtliche Operanden werden Val-Parameter vom angegebenen Operandentyp generiert sowie gegebenenfalls für das Operatorergebnis ein Var-Parameter von entsprechendem Typ.

Für die oben angegebene Beispielspezifikation der Zwischensprache generiert BEG folgendes Definitionsmodul:

```
DEFINITION MODULE Cons;
 TYPE
    Address;
    Data;
 PROCEDURE IntegerConst    (     Atval  : OT.oLONGINT ;
                           VAR result : Data        );
 PROCEDURE LocalVariable   (     Atadr  : LONGINT     ;
                           VAR result : Address     );
 PROCEDURE SimpleAssignment(     op1    : Address     ;
                                 op2    : Data        );
END Cons.
```

Zur Konstruktion eines Ausdrucksbaums müssen vom Frontend die Prozeduren der entsprechenden Operatoren, angefangen bei den Blättern des Ausdrucksbaums, aufgerufen werden. Die von den Operatorprozeduren gelieferten Werte werden als Argumente zur Konstruktion des Vaterknotens benutzt. Die Konstruktion endet mit dem Aufruf der Prozedur eines Top-level-Operators. Der generierte Codeerzeuger beginnt nunmehr mit der Überdeckung des erzeugten Ausdrucksbaums gemäß den in der CGD angegebenen Regeln (s.u.).

2.1.2 Register

Innerhalb einer CGD werden in einem eigenen Abschnitt die Register der Zielmaschine durch Aufzählung ihrer Namen angegeben. Diese Aufzählung ist für die Generierung der Registerallokation notwendig und ermöglicht darüberhinaus die einfache Angabe von Teilmengen aus dieser Registermenge.

Bei einigen Prozessoren kann auf Teile von Registern zugegriffen werden, oder Register können kombiniert werden; dies wird so berücksichtigt, daß hinter dem Namen eines Registers die konzeptionell enthaltenen Register (in spitzen Klammern) aufgeführt werden. In einem solchen Fall kann BEG jedoch nur die *generelle* Registerallokation verwenden.

```
RegisterSet  =  REGISTERS RegisterDef { ',' RegisterDef } ';' .
RegisterDef  =  ident [ '<' ident { ',' ident } '>' ] .
```

Beispiel:

```
REGISTERS
       al,ah ,       bl,bh ,       cl,ch ,       dl,dh ,
    ax<al,ah>,    bx<bl,bh>,    cx<cl,ch>,    dx<dl,dh>,    si ,    di,
eax<ax>      ,ebx<bx>      ,ecx<cx>      ,edx<dx>      ,esi<si>,edi<di>;
```

BEG generiert aus der Spezifikation der Register innerhalb des bereits erwähnten Definitionsmoduls für das Frontend einen Aufzählungstyp `BegRegister`, der genau die aufgezählten Registernamen präfiziert mit dem Kürzel `Reg` und den Wert `RegNil` enthält, welcher z.B. zur Initialisierung benutzerdeklarierter Variablen dieses Typs dienen kann.

2.1.3 Nichtterminale

Nichtterminale definieren die Typisierung der Knoten des Überdeckungsbaums. Sie werden u.a. zur Modellierung der verschiedenen Adressierungsarten der Zielmaschine eingesetzt und können Attribute besitzen, über die während der Ausgabephase *abgeleitete* (synthesized) Informationen weitergereicht werden können. Zusätzlich können spezielle *Bedingungsattribute* (Condition Attributes) definiert werden, die bereits während der Überdeckungsphase berechnet werden und deshalb in Bedingungsabschnitten von Regeln (s.u.) zur Steuerung der Regelauswahl von BEG benutzt werden können.

Grundsätzlich lassen sich die Nichtterminale durch Angabe entsprechender Schlüsselwörter bei ihrer Deklaration in die folgenden Klassen einteilen:

Register-Nichtterminale sind notwendig, damit BEG die Registerallokation generieren kann. Fehlen sie, so bedeutet dies, daß BEG keine Registerallokation generieren soll. Ein Register-Nichtterminal besitzt automatisch ein Attribut mit dem Namen `register` vom Typ `BegRegister`, in welches während der Registerallokation das ausgewählte Register gespeichert wird. Register-Nichtterminale werden durch das Schlüsselwort `REGISTERS` gekennzeichnet. Es ist möglich mehrere Register-Nichtterminale zu spezifizieren, um die Registerallokation zu steuern. Hinter dem Schlüsselwort `REGISTERS` werden die Register aufgezählt, die bei der Registerallokation für das Attribut `register` vergeben werden dürfen. Diese Register werden

im folgenden die *zulässigen* Register genannt. Dies ist eine Grundeinstellung, die für Vorkommen dieses Nichtterminals innerhalb einer Regel geändert werden kann.

Adressierungsart-Nichtterminale werden durch das Schlüsselwort `ADRMODE` gekennzeichnet. Sie dienen der Zusammenfassung von Adressierungsarten eines Operanden für eine konkrete Zielmaschineninstruktion. Meist werden in Attributen eines Adressierungsart-Nichtterminals die einzelnen Komponenten der Adressierungsarten gespeichert.

Speicher-Nichtterminale werden durch das Fehlen der beiden oben genannten Schlüsselwörter gekennzeichnet. Sie stehen für Operatorergebnisse, die nicht in einem Register gespeichert werden.

Nachstehendes Grammatikfragment definiert die Syntax des Nichtterminalabschnitts.

```
Nonterminals     =  NONTERMINALS NonterminalDef ';' { NonterminalDef ';' } .
NonterminalDef   =  ident
                    [ ADRMODE | REGISTERS '<' RegisterList '>' ]
                    [ COND_ATTRIBUTES Attributes ]
                    Attributes .
RegisterList     =  Regs { ',' Regs } .
Regs             =  ident | ident..ident .
```

Nachstehendes Beispiel zeigt die Deklaration von Nichtterminalen. Die Register-Nichtterminale `BReg` und `Reg` stehen für alle Register mit einer Breite von 8 und 16 Bit. Die Menge der zulässigen Register ist entsprechend angegeben. Alle Nichtterminale außer `BReg` besitzen ein Bedingungsattribut `size` vom Typ `tSize`, der ein Aufzählungstyp über `b`, `w` und `l` ist. Dem Register-Nichtterminal `BReg` kann, bedingt durch die angegebenen zulässigen Register, lediglich ein 8 Bit breites Register zugeordnet werden, so daß dieses Nichtterminal das Bedingungsattribut `size` nicht benötigt. Das Adressierungsart-Nichtterminal steht für einen Registeroperanden (Addressing mode REGister) oder einen unmittelbaren Operanden (Addressing mode IMMediate). Es besitzt die Attribute `reg` zur Aufnahme der Registerkodierung, falls es sich um einen Registeroperanden handelt, sowie `val`, zur Aufnahme eines konstanten Werts, falls es sich um einen unmittelbaren Operanden handelt, ausgedrückt durch den Wert `RegNil` im Attribut `reg`.

```
NONTERMINALS
  BReg     REGISTERS <al..dh>;
  Reg      REGISTERS <al..di> COND_ATTRIBUTES (size:tSize);
  ARegAImm ADRMODE            COND_ATTRIBUTES (size:tSize) (reg:BegRegister;
                                                            val:LONGINT);
  Constant                    COND_ATTRIBUTES (size:tSize;
                                               val:LONGINT);
```

2.1.4 Regeln

Der zentrale Abschnitt innerhalb einer CGD ist der Regelabschnitt. Hier wird durch Angabe einer Regelmenge die Übersetzung der Ausdrucksbäume spezifiziert. Das Muster einer

Regel besteht entweder aus einem Nichtterminal oder aus einem Operatornamen gefolgt von genau einem Muster für jeden Operanden des Operators gemäß der Definition des Operators in der Zwischensprache. Ist das erste Symbol eines Regelmusters ein Nichtterminal oder ein Operator, welcher kein Top-level-Operator ist, wird nach dem Muster ein Ergebnis-Nichtterminal angegeben.

Jedes Vorkommen eines Register-Nichtterminals innerhalb einer Regel kann mit einer Liste von zulässigen Registern versehen werden. Ist keine solche Liste angegeben, so gilt die bei der Deklaration des Register-Nichtterminals angegebene Liste. Eine solche Einschränkung der zulässigen Register dient vor allem der Beschreibung von Maschineninstruktionen, die Werte in bestimmten Registern erwarten oder ablegen.

Die Operatornamen und Nichtterminale können mit einer Abkürzung präfiziert werden, über die auf ihre Attribute innerhalb des „Ausgabeteils“ einer Regel Bezug genommen wird. Ein solcher Bezug hat die Form `Abkürzung:Attributname`. Fehlt eine Abkürzung, so muß der Attributname verwendet werden.

Nach dem Muster einer Regel folgt hinter dem Schlüsselwort `COST` die Angabe der Kosten dieser Regel. Hier kann eine Kardinalzahl angegeben werden. Falls keine Kosten für eine Regel definiert sind, so werden die Kosten dieser Regel als Null festgelegt. Auf die Angabe der Kosten folgt innerhalb einer Regel der „Ausgabeteil“. Er besteht aus einem optionalen Abschnitt mit Anweisungen (in Werkzeugzielsprache) zur Berechnung der Bedingungsattribute. Die Anweisungen in diesem durch das Schlüsselwort `EVAL` eingeleiteten Abschnitt werden bereits in der Überdeckungsphase ausgeführt. Wiederum optional folgt ein Abschnitt, in dem die Anweisungen (im folgenden Aktionen genannt) definiert sind, die während der Ausgabephase für diese Regel ausgeführt werden sollen. Er wird durch das Schlüsselwort `EMIT` eingeleitet. Aktionen werden in Werkzeugzielsprache geschrieben.

Enthält eine Regel einen kommutativen Operator, so wird diese Regel von BEG automatisch dupliziert, und die Muster für die Operanden dieses Operators werden in der duplizierten Regel vertauscht. Ergibt sich dadurch jedoch das gleiche Muster, so entfällt die Duplizierung.

```
Rules        =  { Rule | ...} .
Rule         =  RULE Pattern [ '->' Nonterminal ] ';'
                ...
                [ COST Integer ';' ]
                ...
                [ EVAL TargetCode ]
                [ EMIT TargetCode ] .
Pattern      =  [ Shortname ':' ] OperatorIdent { Pattern }
                | Nonterminal .
Nonterminal  =  [ Shortname ':' ] ident [ '<' RegisterList '>' ] .
Shortname    =  ident .
OperatorIdent =  ident .
TargetCode   =  '{' „Anweisungen der Werkzeugzielsprache“ }' .
```

Das folgende Beispiel zeigt zwei Regeln. Die zweite Regel besitzt in ihrem Muster das Speicher-Nichtterminal `Memory`, für das ein Attribut `loc` (location) zur Aufnahme der Adresse eines Speicheroperanden definiert ist. Zum Absetzen der Maschineninstruktionen wird das Assemblersubsystem `ASM` verwendet (siehe S. 54).

```
RULE op:IntegerConst -> c:Constant;
EVAL{ c.size:=w; c.val:=op.val; }

RULE SimpleAssignment m:Memory r:ARegAImm;
COST 1;
EMIT{ IF r.reg=RegNil THEN
         ASM.CS2( mov,r.size  ,  i(r.val),Loc(m.loc) );
      ELSE
         ASM.CS2( mov,r.size  ,  R(r.reg),Loc(m.loc) );
      END; }
```

2.1.4.1 Kettenregeln

Das Muster einer Regel kann auch aus einem einzigen Nichtterminal bestehen. In diesem Fall handelt es sich um eine Kettenregel. Eine Kettenregel überführt ein Nichtterminal in ein anderes Nichtterminal. Kann keine Überdeckung eines Ausdrucksbaums gefunden werden, weil an einer Stelle das Ergebnis-Nichtterminal und das Nichtterminal auf Operandenposition nicht übereinstimmt, wird versucht durch Einfügen von weiteren Knoten anhand der Kettenregeln zu einem Überdeckungsbaum zu gelangen.

Beispiel:

```
RULE src:BReg -> dst:Reg;
EVAL{ dst.size:=b; }
EMIT{ ASM.CS2( mov,b  ,  R(src),R(dst) ); }
```

Eine Regel kann zusätzliche Abschnitte enthalten, die jeweils durch ein Schlüsselwort eingeleitet werden und entweder eine Einschränkung für die Anwendbarkeit dieser Regel definieren oder Anweisungen für die Registerallokation enthalten.

2.1.4.2 Bedingungen

In manchen Fällen ist die Anwendbarkeit einer Regel an eine Bedingung geknüpft. Solche Bedingungen können innerhalb einer Regel hinter dem Schlüsselwort `COND` angegeben werden. Sie werden in Werkzeugzielsprache formuliert und müssen zu `TRUE` oder `FALSE` auswertbar sein. Innerhalb des Bedingungsausdrucks darf lediglich auf die Attribute des Operators und auf die Bedingungsattribute der Nichtterminale Bezug genommen werden, da die Attribute der Nichtterminale während der Überdeckungsphase nicht zur Verfügung stehen. Eine Regel mit einer Bedingung kann nur ausgewählt werden, wenn zusätzlich die Auswertung dieser Bedingung `TRUE` ergibt.

Beispiel:

```
RULE i:AMemAImm -> r:BReg;
COND{ i.size=b }
EMIT{ ASM.CS2( mov,b  ,  Operand(i.oper),R(r) ); }
```

2.1.4.3 Zielregister

Zielregister werden verwendet, wenn Maschineninstruktionen ein Operandenregister als Register für das Ergebnis benutzen. Dies ist bei zweistelligen arithmetischen Operationen meist der Fall. Durch die Angabe eines Operandenregisters als Zielregister über das Schlüsselwort `TARGET` wird sichergestellt, daß bei der Registerallokation für das Register-Nichtterminal dieses Operanden und für das Ergebnis-Nichtterminal das gleiche Register ausgewählt wird. Bei Verwendung der *on the fly* Registerallokation müßte die Menge der zulässigen Register beider Register-Nichtterminale gleich sein. Bei der *generellen* Registerallokation reicht es, wenn für jedes Register aus der einen Menge ein Register aus der anderen Menge existiert, welches das erste beinhaltet.

Beispiel:

```
RULE Sub a1:Reg a2:AMemARegAImm -> r:Reg;
COST 1;
TARGET a1;
EVAL{ r.size:=a1.size; }
EMIT{ ASM.CS2( sub,a1.size  ,  Operand(a2.oper),R(a1) ); }
```

2.1.4.4 Registerveränderungen

Bei manchen Instruktionen der Zielmaschine können als Seiteneffekt die Inhalte bestimmter Register verändert werden. Dies muß bei der Registerallokation berücksichtigt werden, damit keine Registerinhalte verlorengehen. Innerhalb von Regeln, die solche Instruktionen absetzen, kann hinter dem Schlüsselwort `CHANGE` die Menge der veränderten Register angegeben werden. Dabei darf diese Menge keines der zulässigen Register der Register-Nichtterminale dieser Regel enthalten. Dies wird von BEG nicht überprüft.

Beispiel:

```
RULE op:MemCopy dst:Memory src:Memory;
CHANGE <ecx,esi,edi>;
EMIT{ ASM.CS2( lea,l  ,  Loc(src.loc),R(esi) );
      ASM.CS2( lea,l  ,  Loc(dst.loc),R(edi) );
      ASM.CS2( mov,l  ,  i(op.len),R(ecx)    );
      ASM.C0 ( cld                           );
      ASM.C0 ( rep                           );
      ASM.C0 ( movs,b                        ); }
```

2.1.4.5 Explizite Registerallokation

Um bei der Registerallokation ein temporäres Register für eine Regel zu erhalten, kann hinter dem Schlüsselwort `SCRATCH` ein Name und eine Registermenge angegeben werden. Während der Name innerhalb der Regel dem Zugriff auf das Register dient, wird durch die Registermenge festgelegt, welche Register in Frage kommen.

Beispiel:

```
RULE SwapByte o1:Memory o2:Memory;
SCRATCH tmp <al..dh>;
EMIT{ ASM.CS2( mov,b   ,  Loc(o1.loc),R(tmp) );
      ASM.CS2( xchg,b  ,  R(tmp),Loc(o2.loc) );
      ASM.CS2( mov,b   ,  R(tmp),Loc(o1.loc) ); }
```

Die Möglichkeit der expliziten Registerallokation ist bei Regeln mit einem Adressierungsart-Nichtterminal als Ergebnis-Nichtterminal nicht erlaubt.

2.1.4.6 Routinen

Zur direkten Implementierung eines Operators durch Anweisungen in Werkzeugzielsprache kann eine sogenannte *Routine* verwendet werden. Eine solche Routine wird durch das Schlüsselwort `ROUTINE` eingeleitet, gefolgt vom Operatornamen und Anweisungen in Werkzeugzielsprache. Innerhalb der Anweisungen kann auf Operanden und Attribute des Operators zugegriffen werden. In der vorliegenden Arbeit werden keine Routinen verwendet.

Die vollständige Grammatik der BEG-Regeln lautet:

```
Rules        =  { Rule | Routine } .
Rule         =  RULE Pattern [ '->' Nonterminal ] ';'
                [ COST Integer ';' ]
                [ COND TargetCode ]
                [ CHANGE RegisterList ';' ]
                [ SCRATCH ident RegisterList ';' ]
                [ TARGET Shortname ';' ]
                [ EVAL TargetCode ]
                [ EMIT TargetCode ] .
Pattern      =  [ Shortname ':' ] OperatorIdent { Pattern }
                | Nonterminal .
Nonterminal  =  [ Shortname ':' ] ident [ RegisterList ] .
Shortname    =  ident .
OperatorIdent=  ident .
Routine      =  ROUTINE OperatorIdent TargetCode ';' .
TargetCode   =  '{' „Anweisungen der Werkzeugzielsprache“ '}' .
```

2.1.4.7 Wohlgeformtheitsbedingungen

In Abhängigkeit der *Art* des Ergebnis-Nichtterminals einer Regel gelten bestimmte Bedingungen für die Aktionen dieser Regel. Diese Bedingungen müssen bei der Spezifikation der Regeln eingehalten werden, damit eine korrekte Registerallokation erfolgen kann. Folgende Bedingungen sind mit dem Ergebnis-Nichtterminal verknüpft:

Adressierungsart-Nichtterminal: Die Regelaktionen dienen dem Aufbau von Adressierungen. Operandenregister sind nach Anwendung der Regel weiterhin nicht verfügbar.

Register-Nichtterminal: Der abgesetzte Code erzeugt einen Wert, der in einem Register gespeichert wird. Operandenregister sind nach Anwendung der Regel wieder verfügbar.

Speicher-Nichtterminal: Der abgesetzte Code erzeugt einen Wert, der im Speicher abgelegt wird. Operandenregister sind nach Anwendung der Regel wieder verfügbar.

2.2 Endlichkeit der Registermenge

Aufgrund der Endlichkeit der Registermenge konkreter Prozessoren, muß der Fall gehandhabt werden, daß bei der Registerallokation kein passendes, freies Register für ein Vorkommen eines Register-Nichtterminals gefunden werden kann. Um ein belegtes Register in einem solchen Fall wieder verwenden zu können, muß dessen Inhalt temporär zwischengespeichert werden. Bei Bedarf wird der gespeicherte Wert wieder in das Register zurückgeladen. Für diese beiden Situationen, werden Aufrufe der Prozeduren `Spill` und `Restore` generiert, die innerhalb der Spezifikation deklariert sein müssen. Sie werden in einem bestimmten Abschnitt für die Anpassung von BEG erwartet (siehe 2.4). Dort muß zusätzlich eine Prozedur `LR` (Load Register) deklariert sein, welche der Registerallokation zum Absetzen von Instruktionen für das Kopieren des Inhalts eines Registers in ein anderes Register dient. Die Schnittstellen der Prozeduren sind von der Dokumentation zu BEG vorgegeben. Sie lauten:

```
PROCEDURE LR(to, from: Register);

TYPE Spilllocation = INTEGER;
PROCEDURE Spill(reg: Register; loc: Spilllocation);
PROCEDURE Restore(reg: Register; loc: Spilllocation);
```

Die Prozeduren `Spill` und `Restore` werden gemäß LIFO-Prinzip aufgerufen, so daß der Zielmaschinen-Stack als temporärer Speicher verwendet werden kann. In einem solchen Fall kann der Parameter `loc` in der Implementierung der Prozeduren ignoriert werden. Ansonsten kann durch die in `loc` übergebene Information ein Stack simuliert werden.

Beispiel:

```
PROCEDURE LR(to,from:Register);
BEGIN
 ASM.CS2( mov,ASM.RegSizeTab[to]  ,  R(from),R(to) );
END LR;

PROCEDURE Spill(reg:Register; loc:Spilllocation);
BEGIN
 ASM.C1 ( pushl  ,  R(ASM.SizedRegTab[reg,l]) );
END Spill;

PROCEDURE Restore(reg:Register; loc:Spilllocation);
BEGIN
 ASM.C1 ( popl  ,  R(ASM.SizedRegTab[reg,l]) );
END Restore;
```

2.3 Generierter Codeerzeuger

Der gesamte Codeerzeuger ist auf die von BEG generierten Module verteilt. Abbildung 10 gibt einen Überblick über diese Module, ihre Importierungsrelationen, den Datenfluß und ihre Einbettung. Das Modul `Cons` enthält die Datentypen und Prozeduren zum Aufbau des Ausdrucksbaums und stützt sich dabei auf das Basismodul `IR` (Internal Representation), welches interne Datenstrukturen enthält. Die *generelle* Registerallokation ist im Modul `RegAlloc` implementiert. Es benutzt u.a. das Modul `GcgTab` (Generated Code Generator TABles), welches für die Registerallokation benötigte Tabellen enthält. Die *on the fly* Registerallokation wäre dagegen direkt im Modul `Emit` implementiert, in dem auch die Aktionen der Regeln stehen.

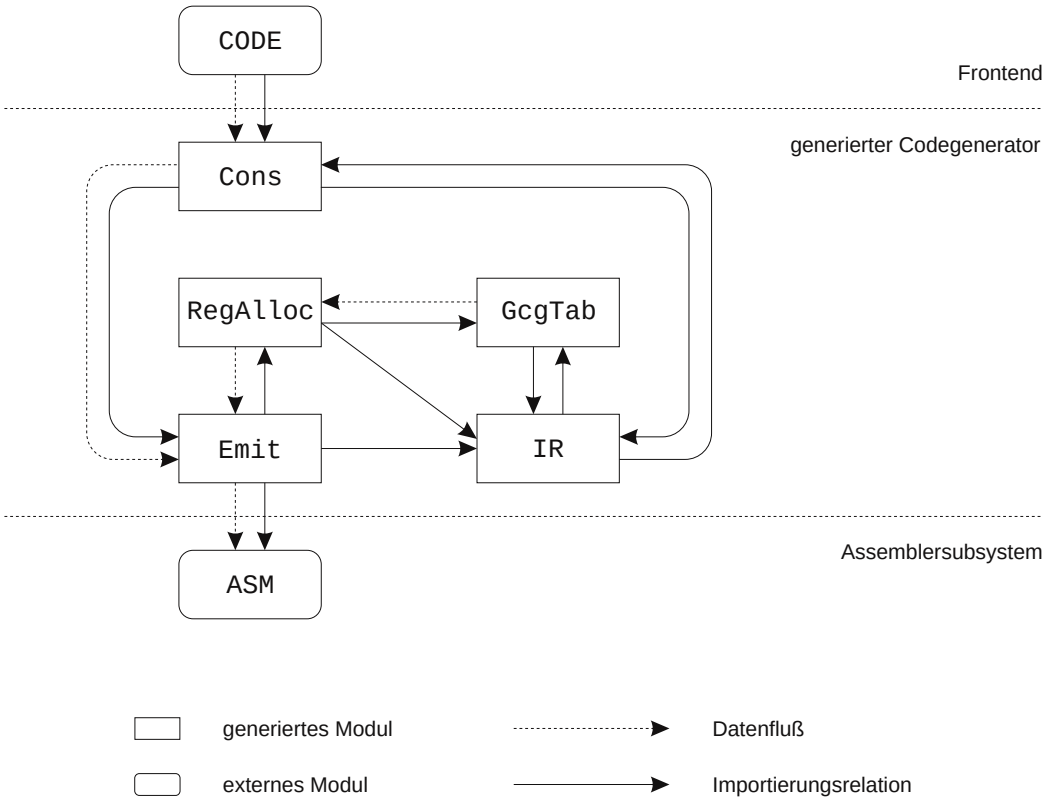

Abbildung 10: Schematische Modulstruktur des generierten Codeerzeugers

2.4 Benutzerspezifische Anpassung

An vielen Stellen innerhalb einer BEG-Spezifikation wird Code in der Werkzeugzielsprache eingefügt. Dieser Code wird in die von BEG generierten Werkzeugzielsprachen-Module kopiert. Damit der Benutzer eigene Datentypen und Prozeduren verwenden kann, muß er die Möglichkeit haben, diese innerhalb der generierten Module zu deklarieren bzw. zu importieren. Zu diesem Zweck existiert in BEGL ein Einfügungsmechanismus. Hinter dem Schlüsselwort `INSERTS` können benannte Abschnitte definiert werden, in denen Werkzeugzielsprachencode steht. Der Name eines Abschnitts definiert einen sogenannten *Einfügungspunkt*, an dem der Abschnitt eingefügt werden soll. Ein solcher Name hat die

Form `Ip`*`Modulname`*`_d` oder `Ip`*`Modulname`*`_i` für das Definitions- bzw. Implementationsmodul. Die Abschnitte werden direkt hinter die letzte generierte Importierungsanweisung des jeweiligen Moduls eingefügt, so daß sowohl weitere Importierungen angegeben werden können als auch Deklarationen von Datentypen und Prozeduren.

Die drei für die Registerallokation notwendigen Prozeduren `LR`, `Spill` und `Restore` werden im Implementierungsmodul `Emit` benötigt. Sie müssen deshalb über den Einfügungspunkt `IpEmit_i` deklariert werden.

Neben den allgemeinen Einfügungspunkten existieren noch weitere spezielle Einfügungspunkte, über die Werkzeugzielsprachen-Abschnitte an anderen Stellen der generierten Module eingefügt werden können. Die beiden wichtigsten Einfügungspunkte sind `IpIR-ConsInit` zur Initialisierung benutzerdefinierter Datenstrukturen und `IpTestImport` für die Importierung von Ausgabeprozeduren, die von BEG verwendet werden, wenn die Option für Testausgabe beim Aufruf von BEG aktiviert wurde (s.u.). In letzterem Abschnitt muß der Benutzer für alle in Attributsdefinitionen verwendeten Typen Ausgabeprozeduren importieren. Diese Prozeduren müssen in der Form `Print`*`Typname`*`(p:`*`Typname`*`)` deklariert sein. Für die Modula-2 Basistypen `INTEGER`, `CARDINAL` und `BOOLEAN` werden diese Prozeduren bereits von BEG zur Verfügung gestellt.

```
Insertions  =  INSERTS { ident TargetCode } .
```

Beispiel:

```
INSERTS
  IpCons_d    { IMPORT ASMOP,Idents,LAB,OT;
                TYPE   tSize = (NoSize,b,w,l,s); }
  IpIR_d      { IMPORT ASM,ASMOP,Idents,LAB,OT; }
  IpIRConsInit{ BooleanTab[FALSE]:=0; BooleanTab[TRUE]:=1; }
  IpTestImport{ FROM   BETO IMPORT PrinttSize,PrinttRelation; }
```

2.5 Werkzeugaufruf und Optionen

Der Aufruf von BEG erfolgt mit dem Namen der Datei, welche die CGD enthält. Hinter dem Dateinamen können noch Optionen angegeben werden, die den Generierungsprozeß beeinflussen. Diese Optionen können auch an den Anfang der CGD (präfiziert mit dem Zeichen '`%`') geschrieben werden und haben dadurch Vorrang vor den mit dem Aufruf angegebenen Optionen. Durch Angabe des Optionsnamens ist die Option aktiviert. Will man eine Option deaktivieren, wird der Optionsname mit dem Präfix `no` angegeben. Alle Optionen sind voreingestellt entweder aktiviert oder deaktiviert.
Folgende Optionen stehen zur Verfügung:

`onthefly` (deaktiviert): Es wird die *on the fly* Registerallokation generiert. Ist diese Option deaktiviert, wird die *generelle* Registerallokation generiert. Falls kein Register-Nichtterminal spezifiziert wurde, wird keine Registerallokation generiert.

`IRConsCheck` (aktiviert): Ist diese Option aktiviert, findet beim Aufbau der Ausdrucksbäume eine statische Typüberprüfung anhand der für die Zwischensprache definierten Operandentypen statt. Ist sie deaktiviert, werden alle definierten Typen auf den Modula-2 Datentyp `ADDRESS` abgebildet.

`test` (deaktiviert): Wird diese Option aktiviert, generiert BEG den Codeerzeuger mit zusätzlichen Routinen, die der Ausgabe von Informationen während der Bearbeitung eines Ausdrucksbaums dienen.

3 Implementierung

Die Generierung des Oberon-2-Compilers aus der Spezifikation erfolgte unter dem Betriebssystem Linux (Kernel-Version 1.0.8) auf einem *IBM PC*-kompatiblen Rechner mit einem Intel 80486-DX2 Prozessor. Eine Hauptspeichergröße von 20 MByte gestattete ein „flüssiges“ Arbeiten mit den Werkzeugen unter der graphischen Oberfläche X11R5. Als Entwicklungswerkzeuge kamen der GNU Assembler `as` (Version 2.2 / i486-linux), der Modula-2 Compiler Mocka (Version 9409), die Compiler-Compiler-Toolbox Cocktail (Version 9209) sowie der Backend-Generator BEG (Version 1.75) zum Einsatz. Für die Freispeicherverwaltung wurde der GNU C Compiler `gcc` (Version 2.5.8) verwendet. Zusätzlich wurde der C-Präprozessor `cpp` eingesetzt, um umfangreiche Spezifikationstexte auf mehrere Dateien verteilen zu können und um über den Makromechanismus des Präprozessors eine Form von „bedingter Kompilierung“ für die Fehlersuche einsetzen zu können.

Der generierte Compiler läuft ebenfalls unter dem Betriebssystem Linux auf einem *IBM PC*-kompatiblen Rechner mit einem Intel 80386 Prozessor (oder höher) und mindestens 8 MByte Hauptspeicher. Er benötigt den GNU Assembler `as` und den Linker `ld`.

Das in der Studienarbeit von Bauer und Spring [1994] spezifizierte Frontend wurde auf einer SparcStation unter SunOS 4.1.3, mittels der dort verfügbaren, älteren Versionen von Mocka und Cocktail entwickelt. Im Rahmen der Diplomarbeit wurden zunächst diese Spezifikationen an die neue Entwicklungsumgebung sowie an die neueren Versionen von Mocka und Cocktail angepaßt. Diese Anpassung bestand im wesentlichen aus der Änderung eines Schlüsselworts und eines Werkzeugaufrufs. Zusätzlich wurde das Frontend im Verlauf der Arbeit am Backend um wenige, weitere Aspekte ergänzt.

Das vorliegende Kapitel beschreibt den implementierten Compiler in seiner Gesamtheit. In einem Überblick werden die Komponenten und ihr Zusammenwirken vorgestellt. Die anschließende Beschreibung listet alle im Frontend vorgenommenen Änderungen auf und gibt zudem eine zusammenfassende Darstellung der bereits in der Studienarbeit ausführlich beschriebenen Frontend-Bestandteile. Es folgt eine ausführliche Beschreibung der durch das Backend realisierten Synthesephase. In einem weiteren Unterkapitel werden die umgebenden Zusatzmodule beschrieben. Ein eigenes Unterkapitel befaßt sich mit der erforderlichen Funktionalität zur Laufzeit eines übersetzten Oberon-2-Programms. Abgeschlossen wird das Implementierungskapitel mit der Darstellung spezieller Problembereiche und deren Lösungen, die sich über verschiedene Phasen bzw. Komponenten des Compilers erstrecken.

3.1 Überblick

Ein Compiler realisiert konzeptionell die Transformation eines Satzes aus der Quellsprache in einen Satz aus der Zielsprache. Abbildung 11 zeigt für den generierten Compiler schematisch den Datenfluß im Rahmen dieser Transformation. Der Scanner liest einen Oberon-2-Quelltext zeichenweise aus einer Datei ein und liefert einen Symbolstrom an den Parser. Dieser baut daraus einen abstrakten Syntaxbaum auf, der vom Evaluator modifiziert und mit Attributen dekoriert wird. Diese Phasen der Analyse werden dem Frontend zugerechnet und waren Bestandteil der Studienarbeit. Die nachfolgenden Phasen gehören zum Backend. Aus dem attributierten Syntaxbaum werden einzelne Aus-

drucksbäume für den Codeerzeuger geknüpft, während der Syntaxbaum traversiert wird. Der Codeerzeuger wandelt einen Ausdrucksbaum in symbolisch codierte Assembleranweisungen um, welche durch das Assemblersubsystem in die entsprechenden textuell repräsentierten Assembleranweisungen transformiert und als Assemblerquelltext gespeichert werden. Der Assemblerquelltext wird durch einen externen Assembler in binär codierte Maschineninstruktionen umgewandelt, welche in einer Objektdatei gespeichert werden.

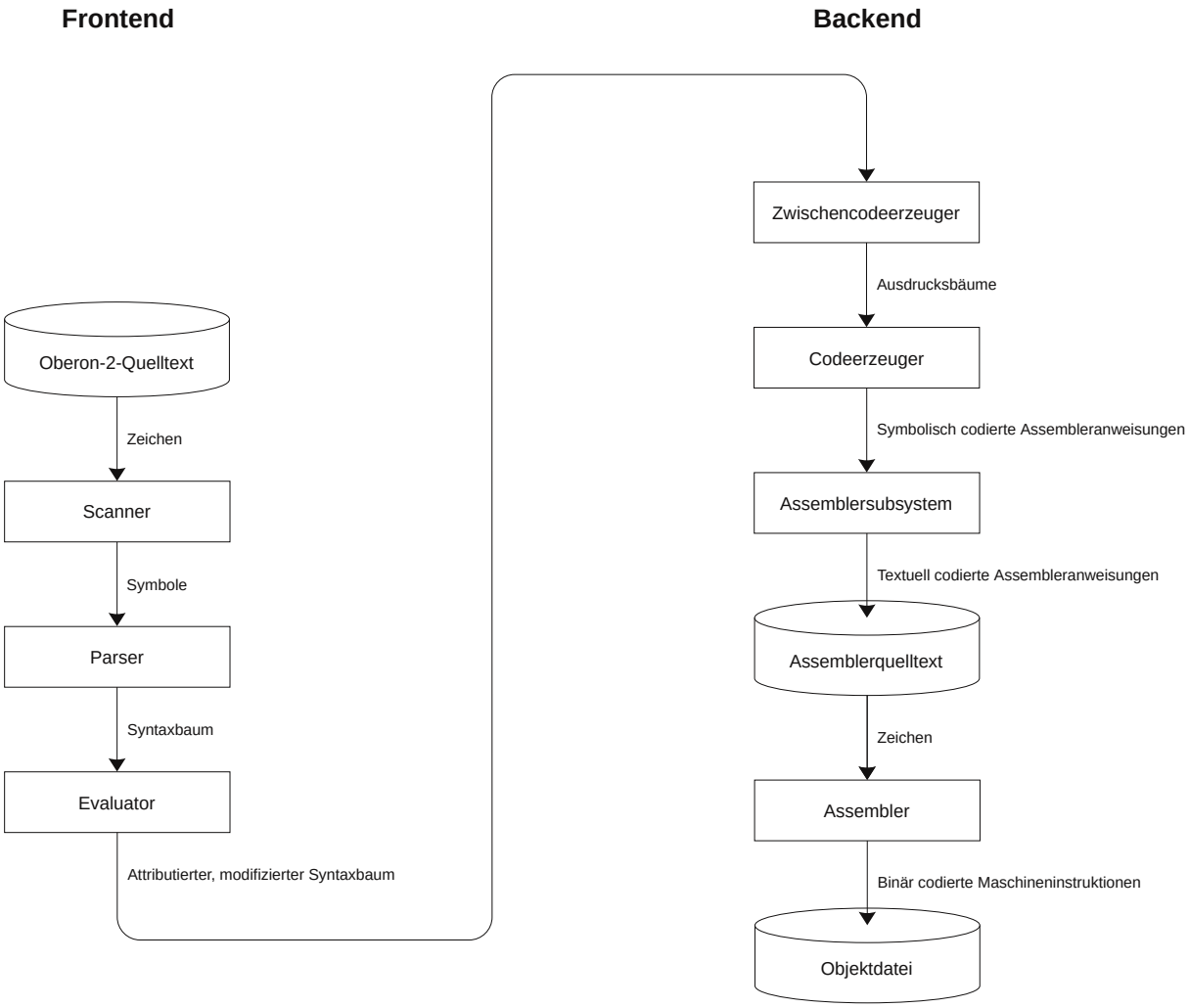

Abbildung 11: Datenfluß im generierten Compiler

Einen Überblick über die Komponenten des Compilers und ihren funktionellen Zusammenhang gibt Abbildung 12. Eine detailliertere Darstellung der Komponenten aus dem Frontend ist Bauer und Spring [1994] zu entnehmen (S. 70f.). Das Hauptmodul analysiert die Kommandozeilenparameter und ruft das Treibermodul auf. Dieses stößt für die Analyse die Komponenten aus dem Frontend an. Der Treiber ruft nach erfolgreicher Analyse die Baumtraversierung auf und übergibt dieser den attributierten Syntaxbaum. Nach der Traversierung aktiviert er zudem das Zielsystem zur Erzeugung der Objektdatei. Das Zielsystem kapselt zusätzlich die Funktionalität, die notwendig ist, damit die Objektdateien übersetzter Oberon-2-Module zu ausführbaren Programmen durch den Linker zusammengesetzt werden können. Hierzu zählt das Absetzen von Assembleranweisungen zur Realisierung modulübergreifender Aspekte, wie beispielsweise den Aufrufen der Rümpfe

aller Module der Modulfolge durch das Wurzelmodul. Das Zielsystem leistet zudem die Abstraktion vom konkreten Aufruf der Shell-Skripts zur Aktivierung des externen Assemblers und Linkers.

Der für ein Oberon-2-Modul zu erzeugende Code läßt sich einteilen in relativ einfachen Code, beispielsweise für Kontrollstrukturen, in etwas komplexeren Code, der aber dennoch gemäß einem festen Schema entstehen muß, z.B. Code für den Prolog von Prozeduren und in komplexen Code, z.B. für Zuweisungen. Lediglich für letztere Kategorie wurde das Werkzeug BEG eingesetzt, da nur bei ihr Mechanismen zur Codeselektion und Registerallokation benötigt werden. Da die Traversierung des abstrakten Syntaxbaums in einem Puma-Modul spezifiziert ist, wird für die ersten beiden Kategorien ebenfalls das Werkzeug Puma eingesetzt.

Die Baumtraversierung stützt sich zudem auf ein Kommentierungsmodul, welches den Assemblerquelltext optional um entsprechende Kommentierungen anreichert, damit dieser für einen Leser besser nachzuvollziehen ist. Schließlich kapselt das Speicherkonstantenmodul die notwendige Funktionalität, damit identische Speicherkonstanten nur einmal abgelegt werden müssen. Die Module für Fließkomma- und Assembleranweisungen werden dem Assemblersubsystem zugerechnet. Sie leisten die Transformation der symbolisch codierten Assembleranweisungen in textuell codierte Assembleranweisungen. Das Modul für Fließkommaanweisungen handhabt zusätzlich die Schwierigkeiten, die sich aus der kellerartigen Verwaltung der Register des numerischen Koprozessors im Zusammenhang mit der Registerallokation ergeben.

Die restlichen vier allgemeinen Module werden an vielen Stellen im Compiler, u.a. auch im Frontend, benutzt. Die Analyse der Kommandozeile des Compiler-Aufrufs sowie die daraus resultierenden Werte der Optionen werden vom Modul für Kommandozeilenargumente zur Verfügung gestellt. Alle für die Adreßrechnung notwendigen Aspekte sind ebenfalls in einem eigenen Modul gekapselt. Ebenso wird die Berechnung von Labels durch ein eigenes Modul realisiert. Die implementierungstechnischen Grenzen des Compilers, z.B. maximale Schachtelungstiefe von Prozeduren, sind im Modul für Begrenzungen zentral definiert.

3.2 Frontend

3.2.1 Änderungen und Ergänzungen

Nachfolgend wird eine Zusammenfassung der im Frontend vorgenommenen Änderungen und Ergänzungen gegeben. Diese betreffen zum einen Änderungen der Sprachdefinition von Oberon-2, zum anderen die Benutzung der neueren Version der Compiler-Compiler-Toolbox und außerdem Ergänzungen der Spezifikation zur Bereitstellung von Werten für die Codeerzeugung.

- Seit der Fertigstellung des Frontends erfolgte eine Präzisierung des Sprachreports bezüglich der Termination von Funktionsprozeduren. Während Bauer und Spring [1994] lediglich das Vorhandensein einer Return-Anweisung innerhalb von Funktionsprozeduren fordern, legt Mössenböck [1995] fest, daß Funktionsprozeduren immer über eine Return-Anweisung verlassen werden müssen.

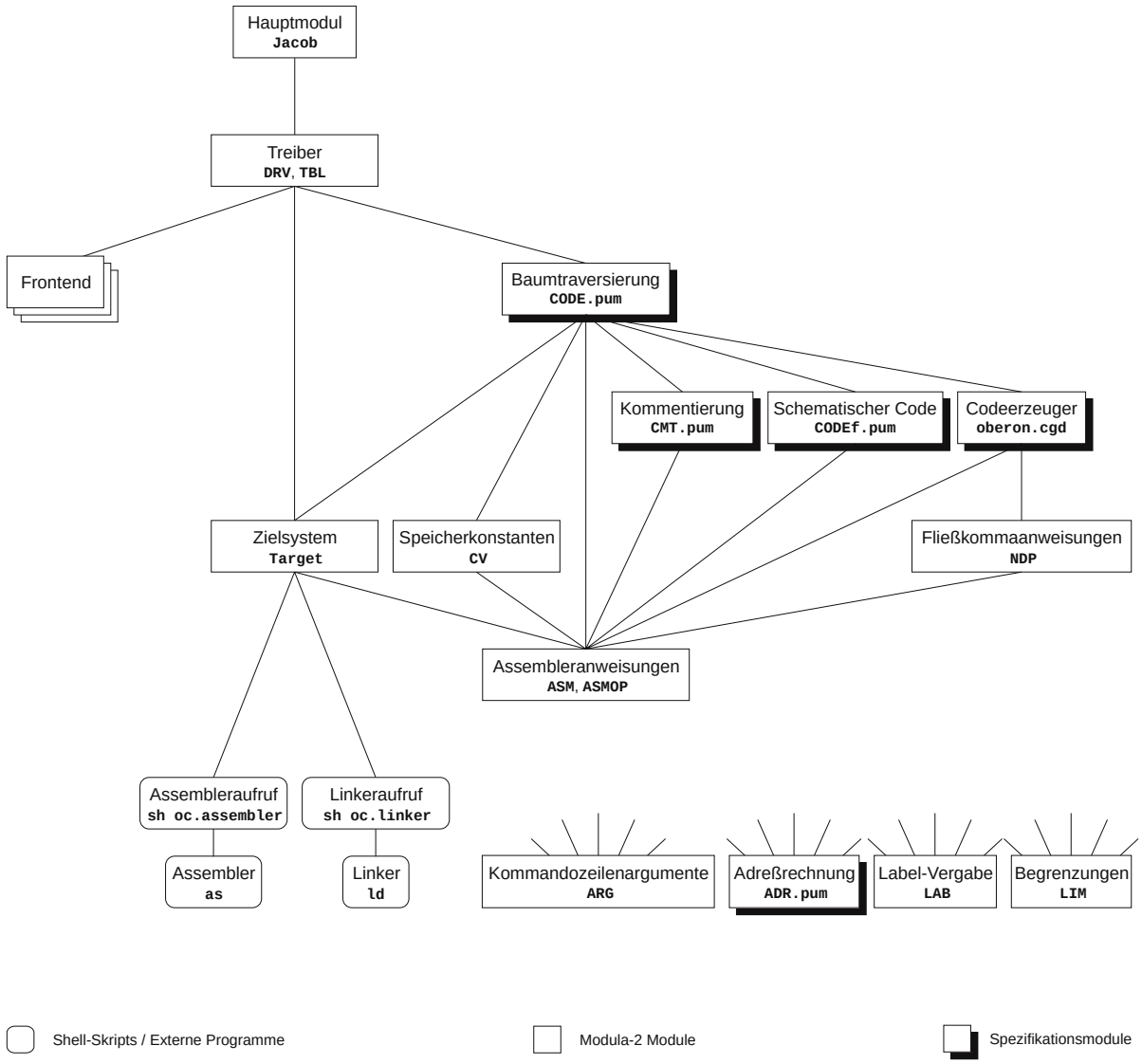

Abbildung 12: Funktioneller Zusammenhang der Backend-Komponenten

Obwohl die Prozedur `VAL` aus dem Modul `SYSTEM` eine Funktionsprozedur ist, darf sie auf einer Var-Parameterposition verwendet werden, um den aktuellen Parameter umtypisieren zu können. Dies wird von N. Wirth und J. Gutknecht [1992] so auch eingesetzt (z.B. S. 211).

Die im Sprachreport in der Version von Juli 1993 fallengelassene Forderung, daß Zeigervariablen mit dem Wert `NIL` initialisiert werden müssen, wird in der verwendeten Sprachdefinition wieder aufgenommen und auf Prozedurvariablen erweitert.

Es gelten Einschränkungen bezüglich der Schachtelungstiefe von Prozeduren (maximal 30) aus Gründen der einfachen Implementierung und der Erweiterung von Record-Typen (höchstens 8 Stufen) als implementierungstechnische Restriktionen (vgl. S. 17).

Als Erweiterung der Sprachdefinition, vor allem zur Betriebssystemanbindung eines Oberon-2-Programms, wird der Zugriff auf Funktionen benötigt, die nicht in Oberon-2 implementiert sind. Hierfür kann mit sogenannten *Fremdmodulen* die

Modellierung einer „sauberen“ Schnittstelle erfolgen, über welche eine bestimmte Fremdfunktionalität für Oberon-2 verfügbar wird. Ein Fremdmodul ist ein durch das neueingeführte Schlüsselwort `FOREIGN` gekennzeichnetes Oberon-2-Modul. Dieses Konzept entspricht der von EMMELMANN und VOLLMER [1994] für Mocka beschriebenen Methode.

```
Module = [ FOREIGN ] MODULE ident ';' ... .
```

Ein Fremdmodul enthält neben exportierten Konstanten, Typen und Variablen im allgemeinen exportierte Prozeduren mit leerem Rumpf. Da in Funktionsprozeduren eine Return-Anweisung vorhanden sein muß, ist diese Kontextbedingung dahingehend geändert, daß allgemein bei fehlendem `BEGIN` eine Return-Anweisung nicht erforderlich ist. Dies gestattet zudem die Formulierung von Oberon-2-Modulen mit Schnittstellencharakter. Für ein Fremdmodul erzeugt der Compiler keinen Code. Bei der Applikation einer Variable oder Prozedur, die in einem Fremdmodul deklariert ist, wird ihr Name, präfiziert mit dem Zeichen '_', als Label für den Zugriff verwendet. Dies erleichtert die Einbindung von Funktionen der C-Bibliothek. Die Auflösung dieses Labels in eine konkrete Adresse bleibt dem Linker überlassen.

Da im Zusammenhang mit der Betriebssystemanbindung über Fremdmodule auf Funktionen zugegriffen werden muß, in denen das Zeichen '_' vorkommt, sollten Namen in Oberon-2 ebenfalls dieses Zeichen enthalten können.

```
ident              = letterOrUnderscore { letterOrUnderscore | digit }.
letterOrUnderscore = letter | '_'.
```

Um der durch die Verwendung der Prozedur `SYSTEM.NEW` zur undisziplinierten Speicherallokation entstehenden Gefahr für die automatische Freispeicherverwaltung (inkonsistenter Heap) entgegenzuwirken, gelten für den Typ T des ersten Parameters Einschränkungen. In einem mit `SYSTEM.NEW` angelegten Speicherblock ist kein Verwaltungsfeld zur Aufnahme einer Typdeskriptorreferenz vorgesehen. Deshalb darf T keine weiteren Zeiger enthalten. Diese Forderung wird von WIRTH und GUTKNECHT [1992] ebenfalls aufgestellt (S. 252). Für den Zugriff auf ein Element eines offenen Arrays innerhalb eines Bezeichners wird ebenfalls vom Vorhandensein bestimmter Verwaltungsfelder ausgegangen. Aus diesem Grund darf T kein offenes Array sein.

- Die Benutzung der neueren Version der Compiler-Compiler-Toolbox hatte zur Folge, daß bei allen Puma-Spezifikationen das Schlüsselwort `EXPORTS` durch `PUBLIC` ersetzt werden mußte. Ansonsten mußte lediglich beim Aufruf des Werkzeugs Puma eine Kommandozeilenoption angepaßt werden.

- Für Werte, die für die Codeerzeugung benötigt werden, sind in einem eigenen (Ast-)Modul innerhalb der Spezifikation des abstrakten Syntaxbaums Attribute definiert. Es handelt sich bei ihnen zum Teil auch um Attribute, die bisher in der Evaluatorspezifikation lokal definiert waren und jetzt durch ihre Verlagerung für die Codeerzeugung „sichtbar“ gemacht sind, da sie nun die Eigenschaft `OUTPUT` besitzen. Im Rahmen der Ergänzung der Attributauswertungsregeln zur Berechnung dieser Attribute wurden der Evaluatorspezifikation weitere (Evaluator-)Module hinzugefügt. Die wichtigsten durch den Evaluator zusätzlich für die Codeerzeugung berechneten Werte sind die folgenden:

  - Adressen und Offsets von Variablen, Parametern und Record-Feldern liegen in Form von ganzzahligen Werten vor.
  - Labels bezeichnen die symbolischen Adressen von Oberon-2-Objekten.
  - Die Konkretisierung des Typs von `NIL` ermöglicht eine unterschiedliche Behandlung der Zuweisung an einen Zeiger oder eine Prozedurvariable.
  - Eine Liste von Typrepräsentationen definiert die zu erzeugenden Deskriptoren.
  - Die Numerierung der gebundenen Prozeduren gibt die Positionen der Prozeduren in der Prozedurtabelle an.

  Neben diesen notwendigen Berechnungen dient die Bereitstellung der nachfolgenden Werte lediglich Optimierungszwecken.

  - Sogenannte Blocklisten sind Datenstrukturen zur Beschreibung der Lage von Zeigern und Prozedurvariablen. Sie gestatten die Erzeugung von kompaktem Code zu ihrer Initialisierung.
  - Die Menge der Schachtelungstiefen von Prozeduraufrufen innerhalb einer Prozedur gestattet einen effizienteren Aufbau von Displays, wird hier aber nur zur Optimierung des letzten Eintrags verwendet.
  - Die Ermittlung, ob ein Val-Parameter innerhalb einer Prozedur verändert wird bzw. verändert werden kann, ermöglicht die eventuelle Unterdrückung der Anlage einer lokalen Kopie im Prozedurprolog.

3.2.2 Scanner (`oberon.rex`)

Die Datei `oberon.rex` beinhaltet die Spezifikation des Scanners. In dieser Spezifikation ist die Zerlegung des Quellprogrammtexts in die textuellen Repräsentationen der Oberon-2-Symbole durch reguläre Ausdrücke innerhalb von Regeln definiert. Das Werkzeug Rex erzeugt aus ihr das Scanner-Modul, welches im wesentlichen eine Prozedur `GetToken` zur Verfügung stellt, die mit jedem Aufruf das nächste Symbol des Quellprogrammtexts liefert.

Die Erweiterungen der Sprachdefinition von Oberon-2 führte in der Scanner-Spezifikation zu folgenden Änderungen:

Zur Einführung des Schlüsselworts `FOREIGN` wurde die neue Konstante `ForeignToken` in die Liste der Symbolkodierungen aufgenommen. Da diese Liste alphabetisch sortiert

bleiben sollte, war eine neue Numerierung notwendig. Ebenso wurde eine neue Regel zur Erkennung der textuellen Repräsentation von `FOREIGN` in die Liste der Oberon-2-Schlüsselwörter eingefügt.

Die Aufnahme des Unterstrichs in die Menge der erlaubten Zeichen von Namen läßt sich am einfachsten durch das Hinzufügen von '_' zur Zeichenmenge `Letter` sowie durch das Entfernen dieses Zeichens aus dem regulären Ausdruck der illegaler Zeichen realisieren.

3.2.3 Abstrakter Syntaxbaum (`oberon.ast`)

Die Spezifikation des abstrakten Syntaxbaums ist in der Datei `oberon.ast` enthalten. Die Struktur des abstrakten Syntaxbaums ist über Knotentypen und ihre Beziehung untereinander definiert. Das Werkzeug Ast erzeugt aus dieser Spezifikation ein Modul, welches für jeden Knotentyp einen Datentyp und eine Konstruktorprozedur zur Verfügung stellt.

Nach der bereits erwähnten Ergänzung um ein (Ast-)Modul besteht diese Spezifikation nun aus

- dem Modul `SyntaxTree`, in welchem der abstrakte Syntaxbaum definiert ist, der durch den Parser geknüpft wird,
- dem Modul `ExtendedSyntaxTree`, in dem die zusätzlichen Knotentypen, Kinder und Attribute definiert sind, die für die Modifikation des abstrakten Syntaxbaums durch den Evaluator benötigt werden,
- dem neuen Modul `CoderInformation`, welches die Definitionen von Attributen enthält, die für die Codeerzeugung sichtbar sind.

Während sich alle Attributdefinitionen des Moduls `ExtendedSyntaxTree` nun im Modul `CoderInformation` befinden, wurde das Modul `SyntaxTree` lediglich um die nachfolgende Definition ergänzt. Die Einführung von Fremdmodulen spiegelt sich im neuen Booleschen Attribut `IsForeign` des Knotentyps `Module` wider. Die aus der Pragmatik von Fremdmodulen abgeleitete geänderte Behandlung von Return-Anweisungen bei Funktionsprozeduren erfordert im abstrakten Syntaxbaum die Unterscheidung der leeren Anweisungsfolge von der fehlenden Anweisungsfolge. Dies wird durch die vom Knotentyp `Stmts` abgeleiteten Knotentypen `mtStmts` und `NoStmts` modelliert.

Im allgemeinen ist es zulässig, daß die Codierung einer Typzusicherung aus Gründen der Effizienz unterbleibt, falls der zugesicherte Typ gleich dem statischen Typ ist. Diese Typengleichheit tritt jedoch innerhalb einer With-Anweisung für die dortigen impliziten Typzusicherungen notwendigerweise auf. Daher dient das neue Boolesche Attribut `IsImplicit` im Knotentyp `Guarding` dazu, explizit im Oberon-2-Quelltext angegebene Typzusicherungen von eingefügten, impliziten Typzusicherungen zu unterscheiden, so daß letztere immer codiert werden können.

3.2.4 Parser (`oberon.lal`)

Die Spezifikation in der Datei `oberon.lal` definiert einen *Bottom-up*-Parser. Sie besteht aus einer kontextfreien Grammatik, deren Regeln um Aktionen ergänzt sind. Die Aktionen definieren die Konstruktion des abstrakten Syntaxbaums. Das Werkzeug Lalr erzeugt

ein Modul, das über eine Prozedur `Parser` die gesamte Parsierungsfunktionalität in einem eigenen Pass zur Verfügung stellt. Das Resultat dieser Prozedur ist ein vollständiger abstrakter Syntaxbaum, falls der entsprechende Quelltext ein Satz der kontextfreien Grammatik ist.

Wegen der Einführung von Fremdmodulen mußte die Grammatikregel zum Startsymbol `Module` um eine Alternative ergänzt werden. Die Regel zur Handhabung von Prozedurrümpfen mußte lediglich im Aktionsteil einer Alternative dahingehend geändert werden, daß bei fehlendem `BEGIN` ein Knoten vom Knotentyp `NoStmts` geknüpft wird.

3.2.5 Evaluatorobjekte (`OB.ast`)

Die Ast-Spezifikation in der Datei `OB.ast` definiert Datentypen, die während der semantischen Analyse benötigt werden. Die spezifizierten Knotentypen dienen vor allem der Modellierung von Symboltabelleneinträgen, Typrepräsentationen, Wertrepräsentationen und Signaturen.

Innerhalb dieser Knotentypen wurden die folgenden Änderungen vorgenommen:

- Zur Speicherung von Label-Werten wurden die Knotentypen `ConstEntry`, `ProcedureEntry`, `BoundProcEntry` und `TypeRepr` um das Attribut `label` ergänzt.
- Zur Repräsentation von `NIL`-Werten bei einer Zuweisung `v:=NIL` wird in Abhängigkeit des Typs von `v` (Zeiger oder Prozedurtyp) der neue Knotentyp `NilValue` sowie seine Erweiterungen `NilPointerValue` und `NilProcedureValue` eingeführt.
- Für ein externes Objekt werden häufig Informationen zum exportierenden Modul benötigt. Daher besitzt für die Symboltabelle der Knotentyp `DataEntry` das Kind `module`, welches auf einen Knoten vom neuen Knotentyp `ModuleEntry` verweist, in dem diese Informationen einmal pro Modul gespeichert sind. Dieses Kind ersetzt u.a. das Attribut `ModuleIdent`.
- Das Attribut `isWithed` im Knotentyp `VarEntry` wird im Rahmen der Behandlung einer Typzusicherung `v(T)` dazu verwendet, für diese Typzusicherung Code zu erzeugen, obwohl der statische Typ von `v` gleich `T` ist (vgl. S. 49).
- Während das Attribut `parMode` des Knotentyps `VarEntry` zur Unterscheidung von Val- und Var-Parametern dient, repräsentiert der Wert des neuen Attributs `refMode` den tatsächlich anzuwendenden Parameterübergabemechanismus *call-by-value* bzw. *call-by-reference* gemäß Objektgröße und -art.
- Der Redundanzeliminierung dient die Änderung im Knotentyp `Signature`: Die Felder dieses Knotentyps entsprachen den gleichnamigen Feldern im Knotentyp `VarEntry` und wurden deshalb durch ein Kind des Knotentyps `VarEntry` ersetzt.
- Die vom Knotentyp `TypeRepr` ausgehende Knotentyphierarchie wurde modifiziert, indem die Knotentypen zur Repräsentation von ganzzahligen Oberon-2-Typen einen neuen gemeinsamen Basisknotentyp `IntTypeRepr` haben, der an Stelle der drei Knotentypen in der Hierarchie steht. Entsprechend wurden die beiden Knotentypen zur Repräsentation von `REAL` und `LONGREAL` unter dem neuen Knotentyp `FloatTypeRepr` zusammengefaßt.

- Das Einfügen des Knotentyps `MemValueRepr` dient der einfacheren Behandlung der Wertrepräsentationen von Speicherkonstanten durch das Werkzeug Puma.
- Das Attribut `nofBoundProcs` des Knotentyps `RecordTypeRepr` dient bei der Berechnung der Numerierung der gebundenen Prozeduren als Zähler. Die Nummer einer gebundenen Prozedur wird im Attribut `procNum` des Knotentyps `BoundProcEntry` gespeichert.
- Das neue Kind `redefinedProc` des Knotentyps `BoundProcEntry` wird für die Codierung des Aufrufs einer redefinierten gebundenen Prozedur benötigt.

Hinzugekommen ist im wesentlichen der Knotentyp `Blocklists` im Zusammenhang mit der kompakten Codierung von Prozedurvariablen- oder Zeigerinitialisierungen sowie der Knotentyp `NamePaths` zur Repräsentation von vollqualifizierten Namen (vgl. S. 16).

3.2.6 Evaluator (`oberon.eva`, `oberon.che`, `oberon.pre`)

Die Spezifikation des Evaluators ist auf die Dateien `oberon.eva`, `oberon.che` und `oberon.pre` verteilt. Sie erweitert die Spezifikation des abstrakten Syntaxbaums um Attributauswertungsregeln und weitere Attribute. Das Wergzeug Ag erzeugt aus ihr das Modul `Eval` das eine Prozedur `Eval` zur Verfügung stellt, welche im wesentlichen die semantische Analyse beschreibt.

Lediglich in der Datei `oberon.eva` wurden Änderungen vorgenommen, indem das vorhandene Evaluatormodul modifiziert und um neue Module ergänzt wurde.

- Das Modul `Nil` dient der Berechnung des konkreten Typs von `NIL`.
- In das Modul `Levels` wurden aus Gründen des thematischen Zusammenhangs die Attributauswertungsregeln verlagert, welche im bereits bestehenden Modul `Evaluator` der Berechnung der Prozedurschachtelungstiefen dienen.
- Im Modul `SizesAndAddrs` sind die Attributauswertungsregeln zur Berechnung von Typgrößen sowie Variablen- und Parameteradressen gekapselt.
- Das Modul `Temporaries` dient der Adreßberechnung von temporären Variablen, z.B. für For-Anweisungen.
- Im Modul `Labels` befindet sich die Label-Vergabe für Konstanten, Typen und Prozeduren sowie für die Loop-Anweisung.
- Im Modul `NamePaths` ist der Aufbau von vollqualifizierten Namen spezifiziert.
- Das Modul `TypeDescriptors` dient dem Aufbau der Liste von Typrepräsentationen, für die Typdeskriptoren erzeugt werden müssen.
- Die Module `OptLaccess` und `OptDisplay` kapseln die Berechnung von Werten, anhand derer im Zusammenhang mit lokalen Parameterkopien sowie mit Displays „optimierter“ Code erzeugt werden kann.

Die aufgeführten Änderungen bereiteten keine besonderen Schwierigkeiten, da sie sich innerhalb des konzeptionellen Rahmens der Evaluator-Spezifikation gut einfügten.

3.2.7 Baumtransformation (TT.pum)

Das Puma-Modul TT stellt Funktionen zur Verfügung, die bei der semantischen Analyse durch den Evaluator zur Transformation von Teilbäumen des abstrakten Syntaxbaums eingesetzt werden.

Im Rahmen der Behandlung von Bezeichnern bei With-Anweisungen wurde die Funktion DesignorToDesignation um eine Funktion DesignorToGuardedDesignation ergänzt, welche zusätzlich die Erzeugung von Knoten zur Repräsentation der impliziten Typzusicherung realisiert.

3.2.8 Evaluatorhilfsfunktionen

Die Evaluatorhilfsfunktionen werden vor allem im Rahmen der semantischen Analyse eingesetzt. Sie sind in Abhängigkeit ihres Aufgabenbereichs auf die nachfolgend aufgeführten Puma-Spezifikationen verteilt. Die neu hinzugekommenen Puma-Spezifikationen dienen der Handhabung von Blocklisten sowie der Adreßrechnung. Die Änderung einiger bereits bestehender Puma-Spezifikationen bestand lediglich aus der Hinzunahme von einfachen Prädikaten und Selektorfunktionen auf Evaluatorobjekten. Auf diese Ergänzungen wird im folgenden nicht näher eingegangen.

E – Funktionen zur Handhabung von Symboltabelleneinträgen
T – Typrepräsentationsbezogene Funktionen
V – Funktionen für die Konstantenauswertung zur Übersetzungszeit
CO – Typanpassungen
SI – Funktionen zur Handhabung von formalen Parameterlisten
PR – Konstruktion der Tabelle der vordeklarierten Objekte und der vom Modul SYSTEM exportieren Objekte (wurde vereinfacht)

Die folgenden Puma-Spezifikationen sind neu hinzugekommen:

- Die Spezifikation BL dient der Berechnung der für Typrepräsentationen, globale Variablen und Aktivierungssegmente von Prozeduren benötigten Blocklisten. Anhand dieser Blocklisten vereinfacht sich die Erzeugung von Code zur Initialisierung von Zeigern und Prozedurvariablen (vgl. 3.6.1, S. 112).
- Die Spezifikation ADR stellt diverse Puma-Funktionen und Modula-2-Funktionsprozeduren zur Verfügung, die für die Adreßrechnung zum Einsatz kommen.

3.2.9 Ausgabefunktionen (TD.pum, OD.pum)

Die in den Puma-Spezifikationen TD und OD definierten Ausgabefunktionen zur Unterstützung der Fehlersuche im Compiler wurden entsprechend den obengenannten Änderungen angepaßt.

3.2.10 Umgebende Zusatzmodule

Um die Spezifikationstexte übersichtlich zu halten, wurden die Deklarationen der Prozeduren, die der Vergrößerung der algorithmischen Basis dienen, aus den Werkzeugzielsprachenabschnitten der Spezifikationsdokumente in eigenständige Modula-2-Module verlagert. Die nachstehend aufgeführten Module wurden seit ihrer Erstellung für das Frontend nicht wesentlich geändert und werden deshalb an dieser Stelle lediglich tabellarisch aufgeführt.

`Errors`	– Fehlermeldungen generierter Module
`ERR`	– Sortierung aufgetretener Fehlermeldungen
`ErrLists`	– Abstrakter Datentyp „Fehlermeldungen“
`TBL`	– Verwaltung aufgebauter Symboltabellen
`OT`	– Kapselung der Oberon-2-Datentypen und der Prozeduren auf diesen
`FIL`	– Behandlung verschachtelter (Eingabe-)Dateien
`POS`	– Repräsentation von Positionen im Quelltext
`O`	– Prozeduren zur Konsolenausgabe
`STR`	– Funktionen auf Zeichenketten
`UTI`	– Weitere Hilfsprozeduren
`ED`	– Abstrakter Datentyp „Editor“ für formatierte Ausgabe eines Syntaxbaums

3.3 Backend

Die Beschreibung des Backends erfolgt in einer Art „Bottom-up-Strategie“. Dabei erfolgt zunächst die Beschreibung des Assemblersubsystems, welches aus Sicht des Codeerzeugers einen abstrakten Assembler darstellt. Dem schließt sich die Darstellung der Codeerzeugung an, in der zuerst eine Erläuterung der Codegeneratorbeschreibung gegeben wird, so daß danach auf die sich darauf stützende Baumtraversierung eingegangen werden kann.

3.3.1 Assemblersubsystem

3.3.1.1 Zielsystem (`Target`, `oc.assembler`, `oc.linker`)

Das Zielsystem stellt die Schnittstelle zum Betriebssystem für die Aktivierung des Assembler- und Linker-Shell-Skripts dar. Das Modul `Target` kapselt hierbei die Funktionalität, die zur Erstellung eines ausführbaren Programms aus Oberon-2-Modulen notwendig ist. Die konzeptionelle Benutzung der von `Target` zur Verfügung gestellten Prozeduren wird durch folgendes Schema dargestellt:

```
Target.ClearLinkList
forall M ∈ Modulfolge do
      Target.Module(M)
      Codeerzeugung für Deklarationen von M
      Target.BeginModule
      Codeerzeugung für Modulrumpf von M
      Target.EndModule
      Target.Assemble
      Target.AddToLinkList
end
Target.Link(Hauptmodul)
```

Die Prozedur `Target.Module` öffnet eine neue Assemblerdatei, in welche die über das Modul `ASM` abgesetzten Assembleranweisungen geschrieben werden. Die Assembleranweisungen für den Deskriptor und den Prolog des Modulrumpfs werden durch die Prozedur `Target.BeginModule` abgesetzt. Durch `Target.EndModule` wird der Epilog des Modulrumpfs erzeugt, die Erzeugung der Speicherkonstanten (s.u.) veranlaßt und die Assemblerdatei geschlossen. In `Target.Assemble` erfolgt der Aufruf des Assemblers durch das Shell-Skript `oc.assembler`. Über `Target.AddToLinkList` wird das soeben assemblierte Modul in die sogenannte Link-Liste aufgenommen. Die Link-Liste enthält die Namen aller Objektdateien der zu einem Programm gehörenden Module. Anhand dieser Liste erzeugt `Target.Link` das Wurzelmodul, initiiert dessen Assemblierung und ruft den Linker über das Shell-Skript `oc.linker` unter Angabe des Namens der zu erzeugenden ausführbaren Datei und der Link-Liste auf.

Die Abstraktion vom konkreten Aufruf des Assemblers erfolgt über das Shell-Skript `oc.assembler`, dem als einziges Argument der Name der zu assemblierenden Datei übergeben wird.

```
#!/bin/sh
as -L -o $1.o $1.s
```

Der Linker wird durch das Shell-Skript `oc.linker` aufgerufen, dem als erstes Argument der Name des zu erzeugenden Programms und als weitere Argumente die Liste der Namen aller dazu notwendigen Objektdateien übergeben wird. Es definiert beim Aufruf des Linkers die Objektdateien des Laufzeitsystems und der Freispeicherverwaltung sowie die für die Betriebssystemanbindung notwendige Bibliothek.

```
#!/bin/sh
PROGRAM=$1 ; shift ; OBJECTS=$*
SYSDIR=/usr/jacob/sys
ld -o $PROGRAM /usr/lib/crt0.o $SYSDIR/OB2RTS.o $SYSDIR/Storage.o \
   $OBJECTS -lc
```

Die Benutzung von Shell-Skripts gestattet hier eine einfache Anpassung der Arbeitsweise des Compilers, indem beispielsweise der Aufruf des Assemblers in

```
as -L -o $1.o $1.s -a >$1.lst
```

geändert werden kann, wodurch für jedes Modul ein Assembler-Listing erzeugt wird, dem die exakte Assemblierung zu entnehmen ist. Falls eine weitere Bibliothek für Oberon-2 verfügbar gemacht werden soll, ist es erforderlich, im Shell-Skript `oc.linker` den Bibliotheksnamen als weiteres Linker-Argument (`-l...`) aufzuführen. (Die Erstellung eines entsprechenden `FOREIGN`-Moduls zur Schnittstellenbeschreibung ist für diesen Fall ebenfalls erforderlich, um die Funktionalität der Bibliothek in Oberon-2 nutzen zu können.)

3.3.1.2 Assembleranweisungen (`ASM`, `ASMOP`)

Die Spezifizierung von abzusetzenden Assembleranweisungen in einer Notation, deren Syntax der von einem Assembler akzeptierten Sprache möglichst nahe kommt, wird vom Werkzeug BEG nicht unterstützt. BEG gestattet an den entsprechenden Stellen lediglich Modula-2-Code. Die von Emmelmann [1989] für diese Aufgabe vorgeschlagene Benutzung des Präprozessors *Dot Tool*, der bei Hopp [1993] auch zum Einsatz kam, erfordert jedoch zum einen die Einhaltung einer weiteren speziellen Syntax und intendiert zum anderen lediglich die Verwendung von Prozeduraufrufen `Write...` zum Schreiben der textuellen Repräsentationen der Assembleranweisungen in eine Datei, so daß weitergehende Manipulationen von einmal abgesetzten Assembleranweisungen effizient nicht mehr durchführbar sind. Daher wurde im Rahmen dieser Arbeit eine andere Vorgehensweise gewählt, die es gestattet, mit Hilfe von Prozeduren und Konstanten in Modula-2 Assembleranweisungen absetzen zu können, wodurch in Verbindung mit einem entsprechenden Layout nicht nur die Lesbarkeit der Codeerzeugung erhöht wird, sondern es auch möglich wird, in den Prozeduren lokale Optimierungen vorzunehmen, welche die Codeerzeugung entlasten.

Im Modul ASM ist die Realisierung dieser Vorgehensweise gekapselt. Somit kann beispielsweise die Assembleranweisung

```
cmpl   $1,%eax
```

durch den Prozeduraufruf

```
ASM.CS2( cmp,l   ,  i(1),R(eax) );
```

in Modula-2 abgesetzt werden.

Wegen der Vermeidung eines Importierungszyklus über die Definitionsmodule von ASM und dem von BEG generierten Modul Cons gibt es ein eigenes Modul ASMOP. Das Modul Cons stellt einen Datentyp zur Repräsentation der Prozessorregister (BegRegister) zur Verfügung, der von ASM importiert wird, um die bei der Registerallokation vergebenen Register verwenden zu können. Bestimmte Attribute von Operatoren der CGD dienen der Repräsentation von Assembleroperationen und sind deswegen vom einem eigenen Typ (tOper), der im Definitionsmodul von Cons importiert werden muß, aber nicht vom Modul ASM exportiert werden kann. Daher ist der Aufzählungstyp tOper zusammen mit weiteren im Rahmen der Repräsentation von Assembleroperationen relevanten Objekten in einem eigenen Modul ASMOP deklariert, das keine Module importiert.

Das Absetzen von Assembleranweisungen, im folgenden kurz Anweisungen genannt, muß durch einen Aufruf der Prozedur ASM.Begin eingeleitet und durch einen Aufruf von ASM.End abgeschlossen werden.

3.3.1.2.1 Operanden

Ein Operand einer Anweisung wird über eine Funktionsprozedur konstruiert, die ein Resultat vom Typ ASM.tOp liefert, welches innerhalb des Aufbaus von Anweisungen (s.u.) benutzt wird. Der opake Typ ASM.tOp dient somit der Repräsentation aller möglichen Operandenarten und kann als Funktionsresultatstyp verwendet werden. Entsprechend den unterschiedlichen Arten von Maschinenoperanden existieren drei Klassen von Konstruktorfunktionen:

- Registeroperanden werden durch Aufruf der Funktionsprozedur R konstruiert. Der Typ ASM.tReg ist deklariert als Cons.BegRegister. Er wird zusammen mit seinen Elementen vom Modul ASM reexportiert.

  ```
  PROCEDURE R(r:tReg):tOp;
  ```

- Für die Konstruktion von unmittelbaren Operanden stehen verschiedene Funktionsprozeduren zur Verfügung. Die Funktion i wird für dezimale Repräsentationen von ganzzahligen Werten eingesetzt; die Funktion x für hexadezimale Repräsentationen zur Erhöhung der Lesbarkeit des Assemblercodes (z.B. für SET-Konstanten). Die Funktionen iL und ioL dienen der unmittelbaren Angabe eines Labels mit und ohne Offset. Zusätzlich wird die Funktion S für die Konstruktion eines Zeichenkettenoperanden verwendet, was nur in Verbindung mit der Direktive asciz sinnvoll ist.

```
PROCEDURE i  (v:LONGINT          ):tOp;
PROCEDURE x  (v:LONGCARD         ):tOp;
PROCEDURE iL (           l:tLabel):tOp;
PROCEDURE ioL(o:LONGINT; l:tLabel):tOp;
PROCEDURE S  (s:ARRAY OF CHAR    ):tOp;
```

- Speicheroperanden können aus unterschiedlichen Bestandteilen aufgebaut sein. Für jede (erlaubte) Kombination dieser Bestandteile existiert eine eigene Funktionsprozedur mit entsprechenden Parametern. Die Signatur einer solchen Prozedur ist in ihrem Namen durch einzelne Buchstaben in der Reihenfolge der nachstehenden Tabelle codiert.

Buchstabe	*Bedeutung*	*Parameter*
o	Offset	o:LONGINT
L	Label	l:tLabel
B	Basisregister	br:tReg
I	Indexregister	ir:tReg
f	Skalierungsfaktor	f:LONGINT

Beispiele: `oL(30,labelVar)` `BIf(ebx,edx,4)`

Darüber hinaus werden vom Modul `ASM` zur Zusammenfassung von Operandenbestandteilen folgende Record-Typen zur Verfügung gestellt.

```
TYPE tLocation = RECORD
                  label            : tLabel;
                  ofs              : LONGINT;
                  breg, ireg       : tReg;
                  factor           : LONGINT;
                 END;
     tOperand  = RECORD
                  CASE kind:tOperKind OF
                  |okImmediate: val : LONGINT;
                  |okRegister : reg : tReg;
                  |okMemory   : loc : tLocation;
                  END;
                 END;
```

Beispielsweise werden bei der Ausführung von Aktionen einer CGD-Regel, deren Resultat ein Adressierungsart-Nichtterminal ist, die einzelnen Bestandteile des (Speicher-)Operanden in den Feldern eines solchen Record-Typs gehalten und/oder modifiziert. Die Funktionsprozeduren `Loc` und `Operand` konstruieren dann aus diesem Record die entsprechende Repräsentation vom Typ `tOp`, dessen interne Struktur aus konzeptionellen Gründen außerhalb von `ASM` nicht sichtbar ist. Das Feld `factor` des Typs `tLocation`, welches den Skalierungsfaktor einer Indexadressierung repräsentiert, kann hierbei jeden beliebigen (positiven) Wert enthalten; für den Fall das er kein gültiger Skalierungsfaktor (1, 2, 4 oder 8) ist, wird vor der zugehörigen Instruktion eine möglichst effiziente Instruktion abgesetzt, welche das entsprechende Indexregister so multipliziert, daß ein gültiger Faktor verbleibt. Diese Multiplikation wird von der Prozedur `MultR`, die vom Modul `ASM` nicht exportiert wird, unter Berücksichtigung etwaiger Optimierungsmöglichkeiten (s.u.) abgesetzt und in der Funktionsprozedur `Loc` veranlaßt.

3.3.1.2.2 Anweisungen

Anweisungen lassen sich nach der Zusammenfassung der unterschiedlichen Arten von Operanden anhand der Anzahl ihrer Operanden und dem Vorhandensein einer Größenangabe klassifizieren. Zum Absetzen einer Anweisung existiert daher für jede Klasse eine eigene Prozedur mit einem Kopf der Form (mit $i \in [0,3]$):[1]

```
PROCEDURE Ci   ( oper:ASMOP.tOper;            op1,...,opi:tOp );
PROCEDURE CSi  ( oper:ASMOP.tOper;  s:tSize;  op1,...,opi:tOp );
```

Zusätzlich steht die Prozedur `ASM.Ln` zum Absetzen einer leeren Anweisung (in Form einer leeren Zeile) zur Verfügung.

Im Aufzählungstyp `ASMOP.tOper` gibt es Konstanten für die Op-codes des Prozessors und für die (verwendeten) Direktiven des GNU Assemblers. Die Namen dieser Konstanten sind identisch mit den textuellen Repräsentationen der Op-codes des GNU Assemblers bzw. seiner Direktiven ohne führenden Punkt. Der Typ `ASM.tSize` ist ebenfalls ein Aufzählungstyp mit den Werten `NoSize`, `b`, `w`, `l` und `s`.

Beispiel für das Absetzen von Anweisungen:

```
ASM.C0 ( text                               ); (* .text                    *)
ASM.CS1( inc,w  ,  R(dx)                    ); (* incw   %dx                *)
ASM.CS2( mov,l  ,  R(eax),oBIf(23,ebp,ecx,4) ); (* movl   %eax,23(%ebp,%ecx,4) *)
```

Zum Absetzen von Direktiven und Label-Deklarationen stellt das Modul `ASM` weitere Prozeduren zur Verfügung. Diese haben Makro-Charakter, indem sie mehrere Anweisungen zusammenfassen und/oder zusätzliche Funktionalität besitzen.

```
PROCEDURE Comm     (l:tLabel; v:LONGINT);   (* .comm  l,v    *)
PROCEDURE Align    (a:LONGINT          );   (* .align a,0x90 *)
PROCEDURE Data;                             (* .data         *)
PROCEDURE Text;                             (* .text         *)
PROCEDURE Asciz    (s:ARRAY OF CHAR    );   (* .asciz "s"    *)
PROCEDURE ByteI    (i:LONGINT          );   (* .byte  i      *)
PROCEDURE LongI    (i:LONGINT          );   (* .long  i      *)
PROCEDURE LongI2   (i,j:LONGINT        );   (* .long  i,j    *)
PROCEDURE LongL    (l:tLabel           );   (* .long  l      *)
PROCEDURE Globl    (l:tLabel           );   (* .globl l      *)
PROCEDURE Label    (l:tLabel           );   (* l:            *)
PROCEDURE GLabel   (l:tLabel           );   (* .globl l; l:  *)
PROCEDURE LabelDef (l:tLabel; v:LONGINT);   (* l = v         *)
```

Daneben existieren weitere „Makro-Prozeduren“, die jeweils ein größeres Codefragment absetzen.

[1] Der Maschinenbefehl `imul` ist der einzige, bei dem drei Operanden angegeben werden können. Er erfordert jedoch immer die explizite Angabe der Operandengröße. Daher gibt es lediglich die Prozedur `CS3` und nicht `C3`.

`PROCEDURE RegCopy(src,dst:tReg);`
Kopiert das Register `src` in das Register `dst`. Stimmen die Größen der Register nicht überein, wird der Wert abgeschnitten bzw. mit 0 erweitert. Wird lediglich zur Codierung von `SYSTEM.VAL` verwendet.

`PROCEDURE MemCopy(src,dst:tOp; tmp:tReg; size:LONGINT);`
Kopiert `size` Bytes ab der Adresse `src` nach Adresse `dst`. Benötigt ein temporäres Register.

`PROCEDURE FillZ(dst:tOp; tmp:tReg; size:LONGINT);`
Füllt `size` Bytes ab der Adresse `dst` mit 0. Benötigt ein temporäres Register.

`PROCEDURE ConstStringCompare(VAR str:OT.oSTRING; reg:tReg);`
Vergleicht das Zeichen-Array ab der Adresse in Register `reg` mit der Zeichenkette `str`.

Die zuletzt über eine der obengenannten Möglichkeiten abgesetzte Anweisung kann mit einem Kommentar versehen werden. Dafür stehen verschiedene Prozeduren zur Verfügung, die jeweils die textuelle Repräsentation des Werts eines bestimmten Typs an den bereits aufgebauten Kommentar anhängen.

3.3.1.2.3 Optimierungen

Eine Anweisung wird im folgenden als *obsolet* bezeichnet, wenn sie im Rahmen einer Optimierung entfällt. Eine obsolete Anweisung ist lediglich noch in einer Assemblerdatei bei gesetzter Option `ARG.OptionCommentsInAsm` als Kommentar vorhanden.

Die symbolische Repräsentation einer Assembleranweisung ermöglicht eine einfache und effiziente Durchführung von lokalen Optimierungen. Folgende Optimierungen werden hierbei durchgeführt:

- Eine Addition oder Subtraktion mit den Werten 0, 1 oder -1 wird entweder obsolet oder durch eine `inc`- oder `dec`-Instruktion ersetzt.
- Eine Multiplikation mit einem konstanten Faktor wird in Abhängigkeit des Faktors entweder obsolet oder durch eine schnellere Instruktion ersetzt.[1] Dies geschieht für Zweierpotenzen (> 8) unter Benutzung der `shl`-Instruktion oder bei Langwort-Multiplikationen durch die `lea`-Instruktion gemäß folgender Tabelle:

Instruktion	*Semantik*
`lea` $(r,2),r$	$r := 2 \times r$
`lea` $(r,r,2),r$	$r := 3 \times r$
`lea` $(r,4),r$	$r := 4 \times r$
`lea` $(r,r,4),r$	$r := 5 \times r$
`lea` $(r,8),r$	$r := 8 \times r$
`lea` $(r,r,8),r$	$r := 9 \times r$

- In einer Folge von Anweisungen, die keine Label-Deklarationen und keine `call`-Instruktionen enthält, sind alle, bis auf das erste, Vorkommen einer `cld`-Instruktion[2] obsolet. Dies geschieht in Verbindung mit dem modulglobalen Flag `is_cld`.

[1] Die Division kann nicht in der gleichen Weise lokal optimiert werden, da die `idiv`-Instruktion keinen unmittelbaren Operanden erlaubt.

[2] „Clear Direction Flag“ führt dazu, daß nachfolgende Maschineninstruktionen zur String-Bearbeitung mit aufsteigenden Adressen arbeiten.

Die textuelle Repräsentation der über die genannten Prozeduren abgesetzten Anweisungen wird vom Modul `ASM` nicht unmittelbar in die Assemblerdatei geschrieben. Es erfolgt vielmehr eine Zwischenspeicherung der aufgebauten symbolischen Repräsentationen in einem Puffer mit beschränkter Größe (1024). Dies gestattet die nachträgliche Manipulation bereits abgesetzter Anweisungen zum Zwecke weiterer Optimierungen. Erst wenn der Puffer vollständig belegt ist und eine weitere Anweisung abgesetzt werden soll, wird die textuelle Repräsentation der „ältesten“ zwischengespeicherten Anweisung in die Assemblerdatei geschrieben und die symbolische Repräsentation aus dem Puffer entfernt. Durch `ASM.End` wird der gesamte Puffer entsprechend „geleert“. Dies muß ebenfalls bei der Verwendung von Prozeduren `ASM.Wr...` zum direkten Schreiben in die Assemblerdatei geschehen, da hier keine symbolische Codierung zur Zwischenspeicherung erfolgt.

Im Rahmen der durch diese Zwischenspeicherung ermöglichten Peephole-Optimierung sind zwei Optimierungen im Modul `ASM` implementiert, welche durch folgendes Schema beschrieben sind:

```
    jmp     Lx          ⟹      #   jmp     Lx
Lx:                             Lx:

    jcond   Lx                  #   jcond   Lx
    jmp     Ly          ⟹      #   jmp     Ly
Lx:                             Lx: jcond^-1 Ly
```

Diese Optimierungen stellen u.a. eine wesentliche Entlastung bei der Codeerzeugung für Boolesche Ausdrücke dar.

Die Funktionalität, eine Anweisung als obsolet kennzeichnen zu können, wird vom Modul `ASM` auch exportiert. Über die Prozedur `ASM.GetLastOperId` wird ein Wert vom Typ `ASM.tOperId` geliefert, welcher die zuletzt abgesetzte Anweisung identifiziert. Anhand der Prozedur `ASM.MakeObsolete` und dieser Identifikation kann zu einem späteren Zeitpunkt die entsprechende Anweisung als obsolet gekennzeichnet werden, sofern sich diese Anweisung noch im Puffer befindet. Dies wird z.B. eingesetzt, um die Reservierung von lokalem Speicher im Aktivierungssegment durch den Prozedurprolog zu optimieren, da die Größe des lokalen Speichers wegen der Art der Fließkommaarithmetik erst nach der Codeerzeugung für den Prozedurrumpf vollständig bekannt ist (siehe nächster Unterabschnitt). Dies begründet die gewählte Größe des Zwischenpuffers.

3.3.1.3 Fließkommaanweisungen (`NDP`)

Ein kellerartig organisierter Registersatz, wie er im Koprozessor 80387 realisiert ist, wird durch die von BEG bereitgestellten Methoden zur Registerallokation nicht unterstützt. Zur Lösung dieses Problems wird von einem prinzipiell unendlichen Registerkeller ausgegangen, der durch das Modul `NDP` realisiert wird. Zum Absetzen von Fließkommaanweisungen exportiert es drei Prozeduren `C0`, `C1` und `CS1`, die jeweils die gleiche Signatur und konzeptionell gleiche Semantik wie die entsprechende Prozedur des Moduls `ASM` besitzen. Diese Prozeduren können aufgrund der Registerbeschränkung zu einer Modifikation der abzusetzenden Anweisung führen.

Hierbei wird von folgender Benutzung des Registerkellers ausgegangen: Der Registerkeller wächst durch Ladeanweisungen und schrumpft durch zweistellige Arithmetikanweisungen bis sich nur noch ein Element im Keller befindet, welches durch eine Speicheranweisung „entfernt“ wird. Zweistellige Arithmetikanweisungen haben als Operanden entweder die obersten beiden Kellerelemente oder das oberste Kellerelement und einen (Haupt-)Speicheroperanden. Eine explizite Bezugnahme auf andere Kellerelemente wird vom Modul `NDP` nicht unterstützt.

Zur Realisierung eines prinzipiell unendlichen Registerkellers erfolgt eine Auslagerung des zuerst gekellerten Elements, falls der Registerkeller des Koprozessors vollständig belegt ist und ein weiterer Wert durch eine Anweisung geladen werden soll. Für diese Auslagerung gibt es innerhalb des Aktivierungssegments jeder Prozedur einen speziellen Bereich `FTemp`. Ein einmal ausgelagertes Element wird nicht wieder in den Koprozessorkeller zurückkopiert, sondern fließt innerhalb einer zweistelligen Arithmetikanweisung, die den ausgelagerten Wert benötigt, direkt als Speicheroperand in die Operation ein. Die Vorschrift zur Benutzung des Registerkellers verhindert es, daß beide Operanden ausgelagert sind.

Vor dem Absetzen von Fließkommaanweisungen über `NDP` muß dem Modul über die Prozedur `NDP.Init` der Offset des `FTemp`-Bereichs mitgeteilt werden. Nachdem alle Fließkommanweisungen abgesetzt sind, liefert die Funktionsprozedur `NDP.UsedTempSize` die benötigte Größe des `FTemp`-Bereichs. Aus diesem Grund erfolgt die Reservierung von lokalem Speicher im Aktivierungssegment einer Prozedur gemäß folgendem Schema, so daß die notwendige Vorwärtsreferenz durch den Assembler aufgelöst werden muß:

```
subl $localVariableSpace+L4711,%esp
Code für Prozedurrumpf
ret
L4711 = UsedTempSize
```

Im Rahmen der Codeerzeugung von Funktionsaufrufen innerhalb von Oberon-2-Ausdrükken werden durch die Prozedur `NDP.Save` die Werte aller im Koprozessor belegten Kellerelemente in den `FTemp`-Bereich ausgelagert, damit der Registerkeller für die aufgerufene Funktion vollständig zur Verfügung steht. Da Fließkommaresultate von Funktionsprozeduren im obersten Kellerelement übergeben werden, gibt es eine Prozedur `SetTop`, über welche dem Modul `NDP` mitgeteilt wird, das sich nach dem Aufruf der Funktionsprozedur ein Element im Registerkeller befindet.

3.3.1.4 Speicherkonstanten (`CV`)

Speicherkonstanten werden im Codeabschnitt abgelegt und sind somit bereits durch die Speicherverwaltung des Betriebssystems vor Veränderung geschützt. Würde eine Speicherkonstante an der Stelle abgelegt werden, an der sie benötigt wird, müßte im allgemeinen eine `jmp`-Instruktion zusätzlich abgesetzt werden, um die Speicherkonstante zu überspringen. Deshalb werden alle in einem Oberon-2-Modul applizierten Speicherkonstanten gesammelt und erst nach allen abgesetzten Anweisungen des Moduls in die Assemblerdatei geschrieben. Dabei werden identische Speicherkonstanten nur einmal abgesetzt.

Die hierfür notwendige Funktionalität ist im Modul CV gekapselt. Es stellt für Zeichenkettenkonstanten und Konstanten vom Typ REAL und LONGREAL jeweils zwei Prozeduren zur Verfügung, mit denen benannte und unbenannte Konstanten „abgesetzt“ werden können.

```
PROCEDURE Named Typename(v:Type; name:LAB.T; isExported:BOOLEAN);
PROCEDURE Typename(v:Type):LAB.T;
```

Die für die Repräsentation einer Speicherkonstanten notwendigen Assembleranweisungen werden jedoch nicht sofort erzeugt. Stattdessen werden alle relevanten Informationen zwischengespeichert, und erst durch Aufruf der Prozedur CV.Code wird das Codefragment für die Konstanten abgesetzt. Bei der Traversierung des abstrakten Syntaxbaums wird für jede deklarierte Speicherkonstante über Named*Typename* der Wert, das berechnete Label und die Exportmarkierung der Konstante in die Konstantentabelle des aktuell bearbeiteten Quellmoduls aufgenommen. Bei der Applikation einer Speicherkonstanten innerhalb eines Ausdrucks wird nicht zwischen einer benannten und unbenannten Konstante unterschieden; das hierbei benötigte Label wird dann von der Funktionsprozedur *Typename* in Abhängigkeit des übergebenen Werts geliefert. Befindet sich dieser Wert noch nicht in der Konstantentabelle, wird er in diese zusammen mit einem neuen lokalen Label aufgenommen.

3.3.2 Codeerzeugung

Die Realisierung der Codeerzeugung erstreckt sich über eine BEG- und drei Puma-Spezifikationen. In jeder Spezifikation (außer der für Kommentierungen) sind bestimmte Abschnitte zur einfacheren Handhabung in eigene Dateien ausgelagert und durch #include-Direktiven in der Spezifikationsdatei ersetzt. Der C-Präprozessor cpp erzeugt nun aus jeder Spezifikationsdatei die Spezifikation als Eingabe für das jeweilige Werkzeug. Der Name jeder *Einfügungsdatei* besteht aus dem Namen der Spezifikationsdatei und einem Suffix, das den ausgelagerten Abschnitt charakterisiert.

3.3.2.1 Kommentierungen (CMT.pum)

Die gewünschte Ausführlichkeit der Kommentierung einer Assemblerdatei legte die Entscheidung nahe, diese Funktionalität in ein eigenes Modul zu kapseln. Dazu stellt das Puma-Modul CMT im wesentlichen drei Prozeduren zur Verfügung, mittels derer die textuelle Repräsentation einer Oberon-2-Anweisung oder ausführliche Informationen zu Deklarationen von Prozeduren bzw. lokalen Variablen als Kommentar in die Assemblerdatei geschrieben werden kann. Das Werkzeug Puma erleichtert hierbei die Spezifikation dieser aus Symboltabelleneinträgen und Syntaxbaumfragmenten generierten Kommentare.

3.3.2.2 Schematische Codeerzeugung (CODEf.pum)

Das Modul CODEf.pum (code fragments) stellt Prozeduren zur Verfügung, welche den Code für den Prozedurprolog und (Typ-)Deskriptoren absetzen. Da für diesen Code die Codeselektion und Registerallokation gemäß einem einfachen Schema in Abhängigkeit von

Evaluatorobjekten erfolgt, welche durch eine Ast-Spezifikation beschrieben sind, ist seine Erzeugung in einem Puma-Modul spezifiziert.

Die Erzeugung des Codes zum Anlegen eines Displays (Prozedur `StackFrameLinks`) und lokaler Kopien von entsprechenden Parametern (Prozedur `RefdValParamsCopy`) ist in der Spezifikationsdatei selbst definiert. Demgegenüber befindet sich die Codeerzeugung für Variableninitialisierungen (Prozedur `VarInitializing`) in der Einfügungsdatei `CODEf.pum.Init`. Die Initialisierung wird in Abschnitt 3.6.1 ausführlicher dargestellt (S. 112). Die Datei `CODEf.pum.TDesc` enthält die Prozeduren zum Anlegen eines Moduldeskriptors (Prozedur `GlobalTDesc`), eines Prozedurdeskriptors (Prozedur `LocalTDesc`) sowie zum Anlegen von Typdeskriptoren (Prozedur `TDescList`).

3.3.2.3 Codegeneratorbeschreibung (`oberon.cgd`)

Die Transformation von Ausdrucksbäumen in symbolisch codierte Assembleranweisungen ist in der Codegeneratorbeschreibung spezifiziert. Die Spezifikationsdatei `oberon.cgd` enthält die Beschreibung der Zwischensprache, die Register- und Nichtterminaldefinitionen, die wesentlichen Kettenregeln sowie die Abschnitte zur benutzerspezifischen Anpassung. Die Regeln, welche die Transformation beschreiben, sind auf mehrere Dateien verteilt. Die Datei `oberon.cgd.min` enthält die Regeln, die benötigt werden, um eine vollständige Überdeckung aller zu konstruierenden Ausdrucksbäume zu gewährleisten. Der hieraus resultierende Code besitzt jedoch an vielen Stellen einen äußerst ungünstigen Aufbau. Daher gibt es in der Datei `oberon.cgd.opt` zusätzliche sogenannte *optimierende* Regeln, mit denen für bestimmte Ausdrucksbäume besserer Code erzeugt wird. Entsprechend sind alle Regeln zur Behandlung der Fließkommaarithmetik auf die beiden Dateien `oberon.cgd.floats.min` und `oberon.cgd.floats.opt` verteilt.

Bedingt durch die Auswahl der Teile eines Quellprogramms, deren Codeerzeugung durch BEG generiert wird, stehen Register und Operandenadressen im Mittelpunkt der Spezifikation. Daher sind die nachfolgenden Unterabschnitte so aufgebaut, daß zunächst eine Darstellung der Registermodellierung und der für die Adreßrechnung modellierten Nichtterminale sowie ihr Zusammenhang erfolgt. Daran anschließend wird ein Überblick über die Nichtterminale gegeben, welche der Repräsentation von Operanden dienen. Nach der Darstellung der Realisierung von Speicherzugriffen wird der Einsatz der spezifizierten Konzepte anhand eines typischen Beispiels einer Zuweisung vorgestellt. Zuletzt werden einzelne Aspekte der CGD dargestellt, die bestimmte Bereiche der Codeerzeugung betreffen und deren jeweilige Realisierung sich über mehrere Regeln erstreckt.

Während der praktischen Benutzung des generierten Codeerzeugers über die Schnittstelle der Operatoren der Zwischensprache haben sich zwei vereinfachende Sichtweisen als hilfreich erwiesen, bei der von vielen Mechanismen des Codeerzeugers abstrahiert wird. Aus der „äußeren“ Sicht werden die Operatoren als Funktionen angesehen, welche als Argumente Code erhalten und als Resultat Code liefern. Der jeweilige Typename in der Operatordefinition charakterisiert hierbei den Code des zugehörigen Arguments bzw. des Resultats. Über die Operatoren werden Ausdrucksbäume geknüpft, welche durch Regeln überdeckt werden, so daß eine Zuordnung von Nichtterminalen einer Regel auf die Argumente bzw. auf das Resultat eines Operators erfolgt. Aus diesem Grund wird aus der „inneren“ Sicht auch davon gesprochen, daß ein Argument bzw. das Resultat eines Operators ein Nichtterminal ist. Im folgenden wird häufiger zwischen diesen Sichtweisen gewechselt,

um die für die Erläuterung einer Lösung jeweils anschaulichste Terminologie verwenden zu können.

3.3.2.3.1 Register

Die Modellierung des Registersatzes der Zielsprache ist im Registerabschnitt der CGD angesiedelt. Dieser hat folgendes Aussehen:

```
REGISTERS
       al,ah ,       bl,bh ,       cl,ch ,       dl,dh ,
    ax<al,ah>,    bx<bl,bh>,    cx<cl,ch>,    dx<dl,dh>,    si ,    di ,
eax<ax,al,ah>,ebx<bx,bl,bh>,ecx<cx,cl,ch>,edx<dx,dl,dh>,esi<si>,edi<di>,ebp,esp,
st,st1,st2,st3,st4,st5,st6,st7;
```

Die Register `ebp` und `esp` sollen durch die Registerallokation nicht vergeben werden. Das gleiche gilt für Fließkommaregister, die alle über ein Nichtterminal repräsentiert werden, welches gesondert erläutert wird. Das allgemeine Register-Nichtterminal lautet somit:

```
Reg REGISTERS <al..edi> COND_ATTRIBUTES (size:tSize);
```

Da bei der Registerallokation unterschiedliche Registergrößen berücksichtigt werden sollen, sind für die Registerklassen jeweils eigene Register-Nichtterminale spezifiziert.

```
BReg REGISTERS < al.. dh>;
WReg REGISTERS < ax.. di>;
LReg REGISTERS <eax..edi>;
```

Das Nichtterminal `Reg` besitzt das Bedingungsattribut `size`, das die Registergröße repräsentiert und das für die Übergänge zwischen den Register-Nichtterminalen in den folgenden Kettenregeln benötigt wird.

```
RULE BReg -> Reg; TARGET BReg; EVAL{ Reg.size:=b; }
RULE WReg -> Reg; TARGET WReg; EVAL{ Reg.size:=w; }
RULE LReg -> Reg; TARGET LReg; EVAL{ Reg.size:=l; }

RULE Reg -> BReg; TARGET Reg; COND{ Reg.size=b }
RULE Reg -> WReg; TARGET Reg; COND{ Reg.size=w }
RULE Reg -> LReg; TARGET Reg; COND{ Reg.size=l }
```

3.3.2.3.2 Adreßrechnung

Die erste nicht triviale Aufgabe innerhalb der Entwicklung einer CGD besteht aus der Modellierung der Berechnung von Operandenadressen. Für diese Modellierung soll ausgehend vom strukturellen Aufbau von Bezeichnern der Quellsprache die Möglichkeiten der Adressierungsarten der Zielsprache weitestgehend ausgenutzt werden, um möglichst wenig Instruktionen absetzen zu müssen.

Der Anfang eines jeden Bezeichners für Variablen kann in zwei Fälle unterteilt werden. Eine globale Variable ist durch einen Offset relativ zu einem Label definiert. Eine lokale Variable auf aktueller Schachtelungstiefe ist durch einen Offset relativ zum Basiszeiger (Register `ebp`) gegeben. Dagegen finden eine lokale Variable, die nicht zur aktuellen Schachtelungstiefe gehört, hier keine Berücksichtigung, da sie konzeptionell als Feld eines Records angesehen wird, der dem jeweiligen Aktivierungssegment entspricht. Aus diesem Grund existieren die beiden folgenden Operatoren der Zwischensprache, die Adreßcode liefern.

```
GlobalVariable (label:LAB.T; adr:LONGINT) -> Address;
LocalVariable  (             adr:LONGINT) -> Address;
```

Zur Modellierung dieser Adressen wurden Nichtterminale definiert, welche den Möglichkeiten der Adressierung in der Zielsprache entsprechen. Es existieren somit die Adressierungsart-Nichtterminale `Gv` (globale Variable über direkte Adressierung), `Breg` (Basisadressierung), `Ireg` (Indexadressierung) und `BregIreg` (Basis- und Indexadressierung). Zur Repräsentation der entsprechenden Komponenten einer Adressierungsart besitzen diese Nichtterminale alle ein Attribut `loc` vom Modula-2-Typ `ASM.tLocation` (vgl. 3.3.1.2.1, S. 57).

Für die Adreßrechnung ist innerhalb eines Bezeichners die Selektion eines Record-Felds, die Indizierung des Elements eines (nicht offenen) Arrays und die Dereferenzierung eines Zeigers zu berücksichtigen. Die Indizierung von offenen Array-Dimensionen erfordert eine gesonderte Behandlung.

Abbildung 13 stellt den Zusammhang der für die Adreßrechnung eingesetzten Nichtterminale über die relevanten Regeln dar. Die Knoten des abgebildeten Graphen sind die Nichtterminale bzw. die Anfänge von Variablenbezeichnern. Jede Kante ist mit einer Regel markiert, welche die Bildung einer neuen Adreßrepräsentation in Abhängkeit des Oberon-2-Selektors definiert.

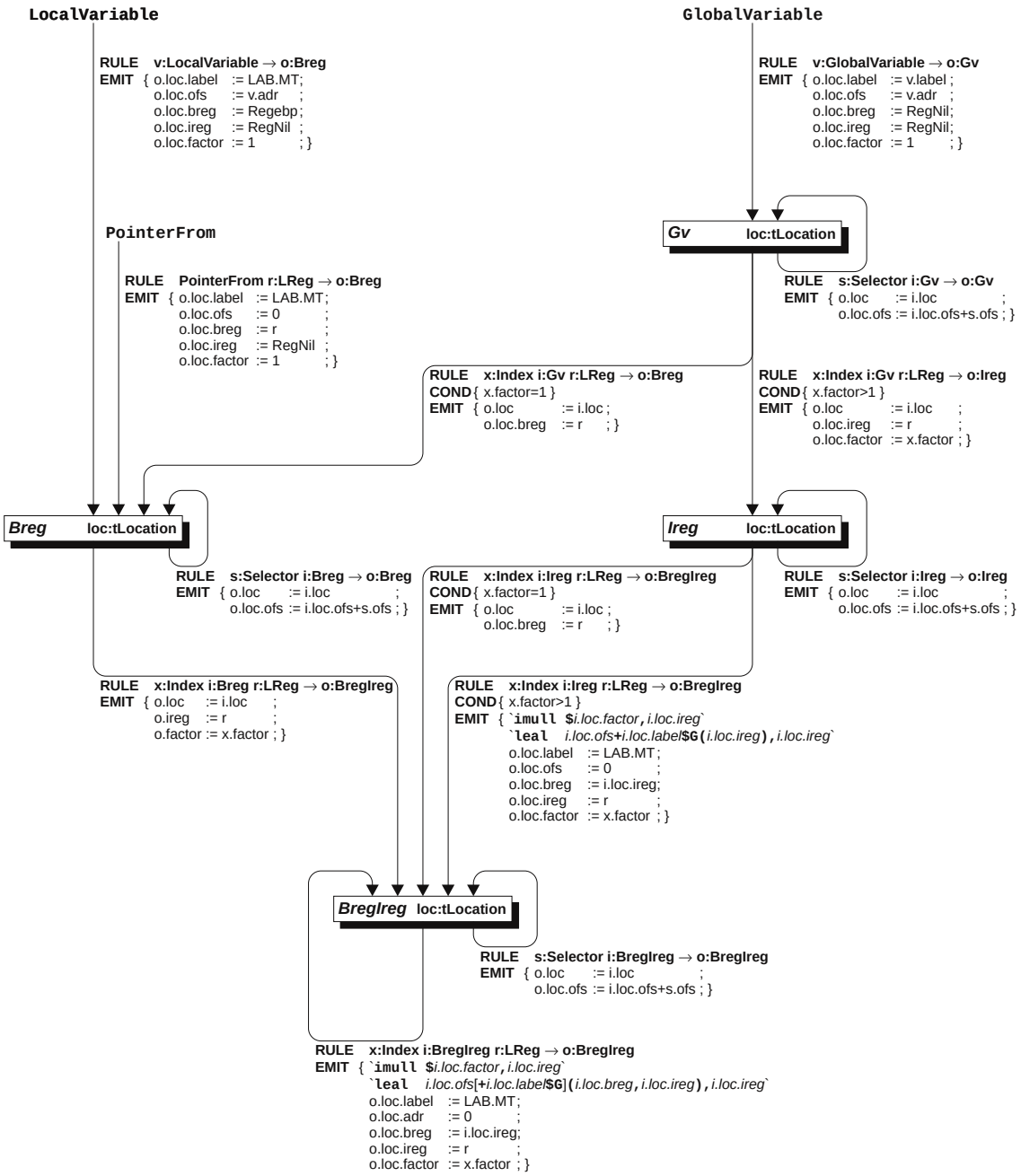

Abbildung 13: Nichtterminale und Regeln zur Adreßrechnung

Der Operator `LocalVariable` liefert das Nichtterminal `Breg`, in dessen Attribut die Adressierung über den Offset `adr` relativ zum Basisregister `ebp` repräsentiert ist. Der Operator `GlobalVariable` liefert demgegenüber das Nichtterminal `Gv`, in dessen Attribut die Felder `loc.label` und `loc.ofs` die Adresse einer globalen Variable definieren.

Die Codierung eines Record-Feldselektors wird durch den Operator

```
Selector (ofs:LONGINT) Address -> Address;
```

vorgenommen. Das resultierende Nichtterminal ist immer das gleiche wie das Argument-Nichtterminal. Die Attributbelegung des resultierenden Nichtterminals unterscheidet sich lediglich im Offset-Feld.

Für die Behandlung von Array-Indizierungen ist es notwendig, daß der Wert des Indexausdrucks für die weitere Adreßrechnung in einem (Langwort-)Register vorliegt. Daher existiert ein weiterer Typ `Data`, welcher sogenannten *Datencode* repräsentiert.

Ein durch einen Operator mit Resultatstyp `Data` konstruierter Knoten des Ausdrucksbaums wird immer von einer Regel überdeckt, welche ein Register-Nichtterminal als Resultats-Nichtterminal besitzt. Über den Operator

```
Index (factor:LONGINT) Address * Data -> Address;
```

erfolgt die Codierung einer Array-Indizierung. Das Attribut `factor` enthält hierbei die Größe des indizierten Array-Elements. In Abhängigkeit des Argument-Nichtterminals kann möglicherweise das Register r, welches den Indexwert enthält, in die Repräsentation der Adressierungsart mit aufgenommen werden. Für den Fall, daß die Größe des Array-Elements 1 beträgt und in der Adreßrepräsentation noch kein Basisregister verwendet wurde, kann r direkt als Basisregister zum Einsatz kommen. Ist das Indexregister in der Adreßrepräsentation noch nicht belegt, kann r als Indexregister eingesetzt werden. Das Feld `factor` der Adreßrepräsentation bekommt dann die Größe des Array-Elements zugewiesen. Läßt sich r in der Adreßrepräsentation nicht mehr „unterbringen“, wird dessen Indexregister für r verfügbar gemacht, indem die durch die Adreßrepräsentation definierte Adresse ausgerechnet und das Resultat als Basisregister der Adreßrepräsentation gehalten wird.

Für die Codierung einer Zeigerdereferenzierung durch den Operator

```
PointerFrom Data -> Address;
```

ist lediglich das Register, welches den Adreßwert enthält, als Basisregister in der Adreßrepräsentation des Resultats zu vermerken.

Da alle diese vier Nichtterminale eine Adressierung von Speicheroperanden repräsentieren, gibt es ein weiteres Nichtterminal `Memory`, ebenfalls mit einem Attribut `loc`, sowie vier entsprechende Kettenregeln.

```
RULE i:Gv       -> o:Memory; EMIT{ o.loc:=i.loc; }
RULE i:Ireg     -> o:Memory; EMIT{ o.loc:=i.loc; }
RULE i:Breg     -> o:Memory; EMIT{ o.loc:=i.loc; }
RULE i:BregIreg -> o:Memory; EMIT{ o.loc:=i.loc; }
```

Somit kann der Zugriff auf Speicheroperanden innerhalb von Regeln über das Nichtterminal `Memory` spezifiziert werden.

3.3.2.3.3 Operanden

Viele Instruktionen der Zielsprache gestatten es, auf einer Operandenposition unterschiedliche Arten von Operanden zu verwenden. Genauso wie alle Adressierungsarten eines Speicheroperanden unter dem Nichtterminal `Memory` zusammengefaßt sind, gibt es Nichtterminale, welche die verschiedenen Kombinationen von Operandenarten repräsentieren. Ausgehend von Registeroperanden, unmittelbaren Operanden und Speicheroperanden, für

die es jeweils ein eigenes Nichtterminal `AReg`, `AImm` und `AMem` gibt, sind für alle Kombinationen dieser drei Operandenarten weitere Nichtterminale spezifiziert.

Abbildung 14 zeigt diese Nichtterminale, die Nichtterminale im Rahmen der Adreßrechnung, die Register-Nichtterminale und das Nichtterminal zur Repräsentation von Konstanten sowie die durch Regeln spezifizierten Übergänge.

Alle Nichtterminale für Operandenarten besitzen ein Attribut `oper` vom Typ `ASM.tOperand`. Dieses Attribut dient als Argument der Funktionsprozedur `ASM.Operand` zur Bildung eines Instruktionsoperanden (vgl. 3.3.1.2.1, S. 57).

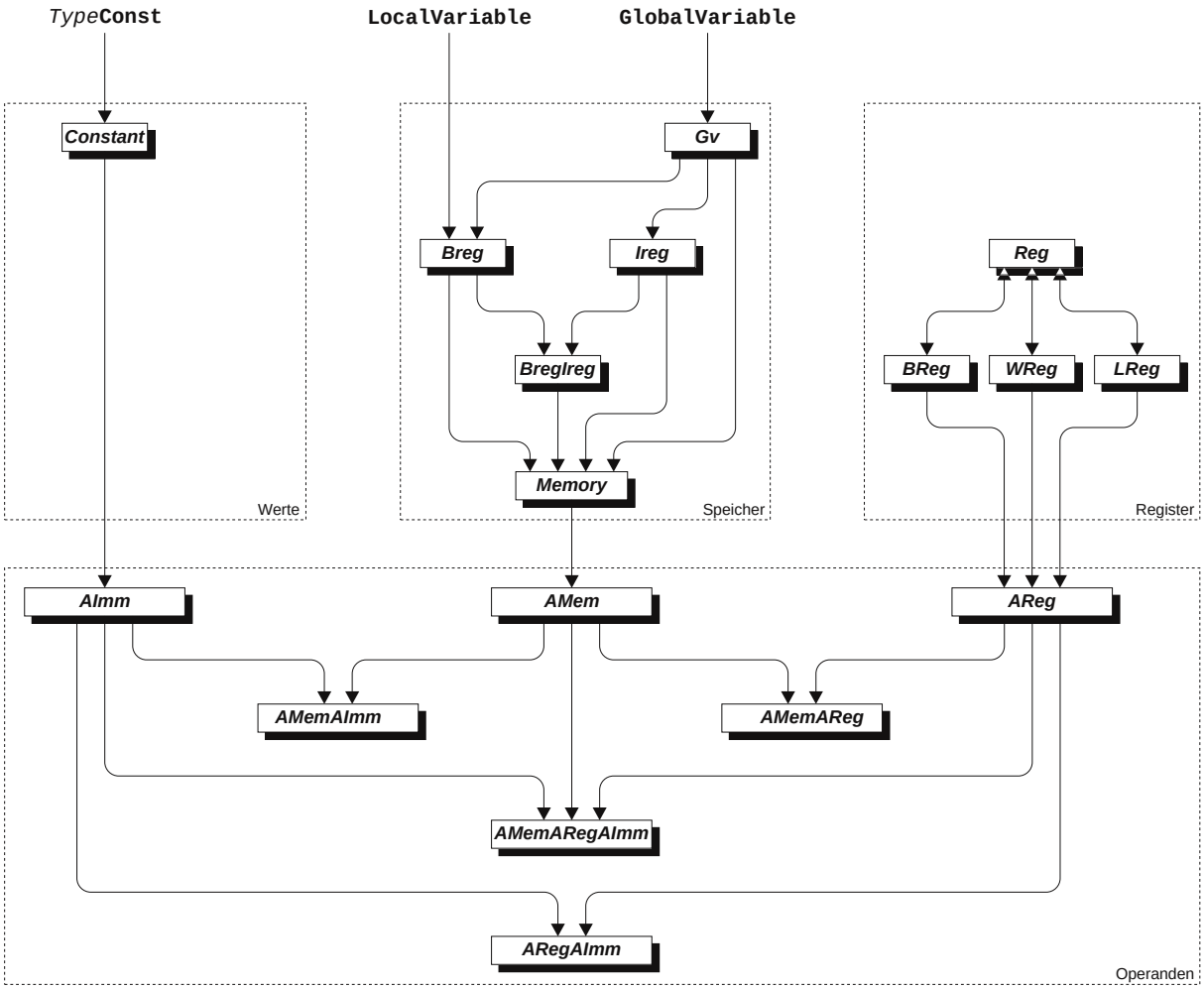

Abbildung 14: Übersicht über die eingesetzten Nichtterminale

3.3.2.3.4 Speicherzugriffe

Das Lesen eines Werts mit Standardtyp (ohne `REAL` und `LONGREAL`) aus dem Hauptspeicher wird mittels des Operators

```
ContentOf (size:tSize) Address -> Data;
```

vorgenommen. Der Operator liefert ein Register-Nichtterminal, welches das Register repräsentiert, das den gelesenen Wert enthält. Das Operatorattribut `size` definiert die Größe

des Werts (`b`, `w` oder `l`). Für die Überdeckung dieses Operators gibt es eine Regel, welche die „Verpackung“ der Operandenrepräsentation in das Speicheroperanden-Nichtterminal vornimmt.

```
RULE s:ContentOf m:Memory -> o:AMem;
EVAL{ o.size:=s.size; }
EMIT{ o.oper.kind := ASM.okMemory;
      o.oper.loc  := m.loc; }
```

Damit nun ein Speicheroperand tatsächlich in ein Register gelangt, gibt es drei Kettenregeln. Der Einsatz einer einzelnen Regel ist wegen der Berücksichtigung der unterschiedlichen Registergrößen durch die Registerallokation hier nicht möglich. Diese Regeln können auch für unmittelbare Operanden angewendet werden.

```
RULE i:AMemAImm -> r:BReg;
COND{ i.size=b }
EMIT{ ASM.CS2( mov,b  ,  Operand(i.oper),R(r) ); }

RULE i:AMemAImm -> r:WReg;
COND{ i.size=w }
EMIT{ ASM.CS2( mov,w  ,  Operand(i.oper),R(r) ); }

RULE i:AMemAImm -> r:LReg;
COND{ i.size=l }
EMIT{ ASM.CS2( mov,l  ,  Operand(i.oper),R(r) ); }
```

Demgegenüber ist für die Überdeckung des Operators

```
AddressOf Address -> Data;
```

zur Bildung des Adreßwerts einer Adreßrepräsentation eine Regel ausreichend, da Adreßwerte immer in Langwortregistern gespeichert sind.

```
RULE AddressOf m:Memory -> r:LReg;
EMIT{ ASM.CS2( lea,l  ,  Loc(m.loc),R(r) ); }
```

3.3.2.3.5 Beispiel: Zuweisung

Der Einsatz der bisher erläuterten Möglichkeiten der Adreßrechnung und des Operandenzugriffs wird nun anhand eines Beispiels dargestellt. Für die Zuweisung innerhalb des Oberon-2-Moduls

```
MODULE M;
  VAR a:ARRAY 10 OF LONGINT;

  PROCEDURE P;
    VAR b:LONGINT;
    PROCEDURE Q;
      VAR i:LONGINT;
    BEGIN
      a[i]:=4711*b;
    END Q;
  BEGIN
  END P;
END M.
```

wird die Konstruktion des Ausdrucksbaums durch nachstehende Folge von Operatoraufrufen vorgenommen.

```
VAR a_acode,b_acode,i_acode,acode: Cons.Address;
    b_dcode,c_dcode,i_dcode,dcode: Cons.Data;
...
Cons.GlobalVariable  ("M$G",8                    ,a_acode);
Cons.LocalVariable   (-16                        ,i_acode);
Cons.ContentOf       (l                 ,i_acode,i_dcode);
Cons.IndexCheck      (10                ,i_dcode,i_dcode);
Cons.Index           (4         ,a_acode,i_dcode,  acode);
Cons.LongintConst    (4711                       ,c_dcode);
Cons.LocalVariable   (-8                         ,b_acode);
Cons.ContentOf       (l                 ,b_acode,b_dcode);
Cons.PointerFrom     (                   b_dcode,b_acode);
Cons.Selector        (-12               ,b_acode,b_acode);
Cons.ContentOf       (l                 ,b_acode,b_dcode);
Cons.SymDyOper       (ASMOP.imul,c_dcode,b_dcode,  dcode);
Cons.SimpleAssignment(                     acode,  dcode);
```

Den Überdeckungsbaum zu dem so entstandenen Ausdrucksbaum (mit beispielhafter Registerbelegung) zeigt Abbildung 15. Der von einer Regel abgesetzte Code ist hierbei neben dem jeweiligen Knoten abgebildet. Die Konstruktion stützt sich zusätzlich auf die folgenden noch nicht beschriebenen Operatoren.

```
LongintConst (val:OT.oLONGINT)              -> Data;
IndexCheck   (len:LONGINT)            Data -> Data;
SymDyOper    (code:ASMOP.tOper) Data + Data -> Data;
```

Der Operator `LongintConst` liefert das Speicher-Nichtterminal `Constant`, welches zwei Bedingungsattribute zur Repräsentation des Werts und der Größe der Konstanten besitzt. Dem Operator `IndexCheck` wird über das Attribut `len` die Länge des entsprechenden Arrays übergeben. Die für die Überdeckung des von ihm gelieferten Knotens spezifizierte Regel

```
RULE op:IndexCheck r:LReg -> o:LReg;
TARGET r;
EMIT{ ASM.CS2( cmp,l   ,  i(op.len),R(r)    );
      ASM.C1 ( jae     ,  L(LAB.IndexFault) ); }
```

setzt innerhalb ihrer Aktionen Code ab, der einen vorzeichenlosen Vergleich zwischen dieser Länge und dem Wert des Indexausdrucks durchführt und eventuell zu einer Fehlerroutine verzweigt. Für die Überdeckung eines vom Operator `SymDyOper` geknüpften Knotens gibt es die folgende Regel:

```
RULE op:SymDyOper a1:Reg a2:AMemARegAImm -> r:Reg;
COST 1;
TARGET a1;
EVAL{ r.size:=a1.size; }
EMIT{ ASM.CS2( op.code,a1.size  ,  Operand(a2.oper),R(a1) ); }
```

Über das Operatorattribut `code` wird der Regel die zu codierende Instruktion übergeben.

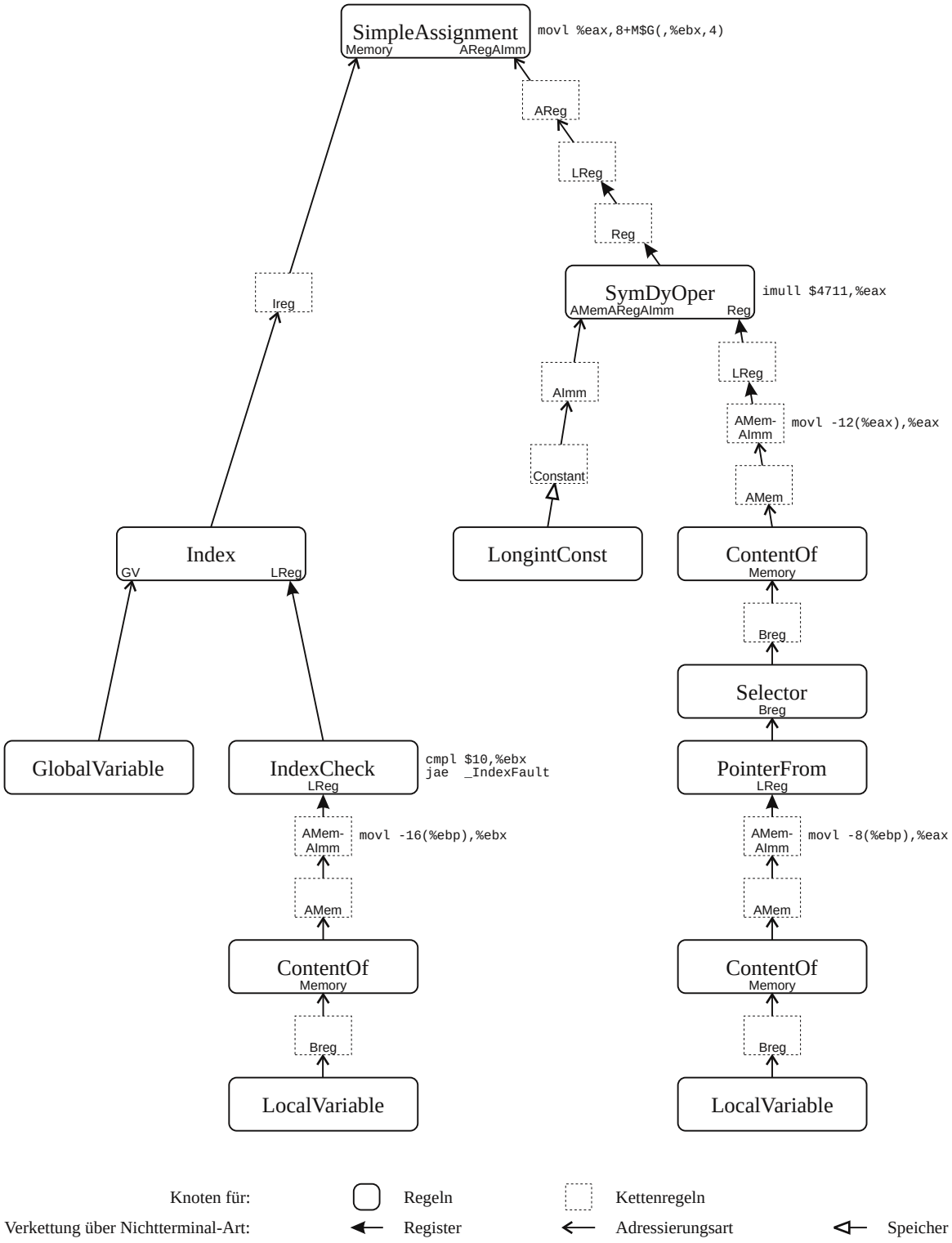

Abbildung 15: Überdeckungsbaum für eine Zuweisung

3.3.2.3.6 Indizierung offener Arrays

Die Adresse eines Array-Elements ergibt sich aus der Anfangsadresse des Arrays, den Werten aller Indexausdrücke sowie den Längen aller zugehöriger Dimensionen. Da bei offenen Arrays diese Längen nicht konstant sind, kann die notwendige Adreßrechnung effizient nur nach dem Horner-Schema erfolgen.

Indexausdrücke zu offenen Dimensionen werden im folgenden als *offene Indizierungen* bezeichnet. Desweiteren wird davon ausgegangen, daß der Typ eines Bezeichners v ein offenes Array ist, das den Aufbau

$$\underbrace{\texttt{ARRAY OF ... ARRAY OF}}_{n\text{-mal}}\ \texttt{T}$$

hat (mit $n > 0$), und daß ein Bezeichner der Form

$$v[e_0]\ldots[e_{m-1}]$$

vorliegt (mit $0 < m \leq n$). Falls mehr Indexausdrücke als offene Array-Dimensionen vorhanden sind, wird die Auswertung der „überzähligen“ Indizierungen durch die bereits beschriebene, einfache Adreßrechnung vorgenommen, nachdem die Adresse des Objekts vom Typ `T` gemäß den offenen Indizierungen ermittelt wurde.

Die Berechnung der Elementadresse kann somit gemäß folgender Formel durchgeführt werden (mit $L_i = \texttt{LEN}(v,i)$):

$$\begin{aligned}&\Big(\Big(\ldots\big(\big(\quad e_0 \quad * L_1 + e_1\big) \quad * L_2 + e_2\big)\ldots\Big) \quad * L_{m-1} + e_{m-1}\Big)\\ &\qquad * \quad L_m * \cdots * L_{n-1}\\ &\qquad * \quad \texttt{SIZE(T)}\\ &\qquad + \quad \texttt{ADR}(v)\end{aligned}$$

Die Realisierung dieser Berechnung erfolgt mittels einer Variablen *disp* (displacement), welche mit dem Wert von e_0 initialisiert und gemäß obiger Formel für jede offene Indizierung schrittweise erhöht wird. Der hieraus resultierende Wert von *disp* wird anschließend mit der Größe des Elementtyps multipliziert, der nach den offenen Indizierungen gilt. Für den Fall, daß $m < n$ ist, sind hierbei die „überzähligen“ Längen zu berücksichtigen. Der Zugriff auf die Längen, welche für die Berechnung (und die Indexüberprüfung) notwendig sind, ist im allgemeinen nicht über eine einzige Adreßdereferenzierung möglich. Daher wird in der Variablen *base* die Basisadresse der Verwaltungsfelder des Arrays gespeichert, so daß mit einem Offset relativ zu *base* der Wert der jeweiligen Länge zur Verfügung steht.

Aus Effizienzgründen müßten die Werte von *disp* und *base* während der gesamten Berechnung in jeweils einem Register gehalten werden. Daher würde bei der Realisierung mit BEG als Regelresultat ein Adressierungsart-Nichtterminal eingesetzt werden. Dies hätte jedoch zur Folge, daß ein Register, welches durch den Wert eines Indexausdrucks belegt ist, für die Auswertung nachfolgender Indexausdrücke nicht wieder zur Verfügung

steht. Aus diesem Grund wird ein Register-Nichtterminal als Resultat eingesetzt, dessen Register die Variable *disp* repräsentiert. Die Werte von *base* werden in einer temporären Variable (im Aktivierungssegment) gespeichert, deren Offset jedem Operator als Attribut übergeben wird.

Für die erste offene Indizierung steht der folgende Operator zur Verfügung:

```
OpenIndexFirst ( tmpOfs     :   LONGINT ;
                 lenOfs     :   LONGINT )
               (*headerBase :*) Address
           *   (*x          :*) Data    -> (*disp:*)Data;
```

Seine Operanden sind die Basisadresse der Verwaltungsfelder des Arrays sowie das Register, welches den Wert von e_0 enthält. Zusätzlich erhält der Operator über das Attribut `lenOfs` den Offset der ersten Länge (`LEN(`v`,0)`). Der in der zugehörigen Regel spezifizierte Code kann somit die Indexüberprüfung durchführen und die temporäre Variable mit dem Wert der Basisadresse der Verwaltungsfelder laden. Die Regel liefert das Register, welches e_0 enthält, als *disp* zurück.

Für alle weiteren offenen Indizierungen wird der Operator `OpenIndex` verwendet:

```
OpenIndex ( tmpOfs :   LONGINT ;
            lenOfs :   LONGINT )
          (*disp   :*) Data
       *  (*x      :*) Data    -> (*disp:*)Data;
```

Er setzt den Code für einen Berechnungsschritt ab. Seine Operanden sind daher das Register *disp* sowie das Register, welches den Wert des Indexausdrucks e_i enthält. Über das Attribut `lenOfs` erhält der Operator den Offset der relevanten Länge (`LEN(`v`,`i`)`).

Die Berechnung wird abgeschlossen, indem die Größe des Elementtyps durch die beiden folgenden Operatoren in die Berechnung einfließt.

```
OpenIndexOpenBase ( tmpOfs     :   LONGINT ;
                    lenOfs     :   LONGINT )
                  (*disp       :*) Data    -> (*disp:*)Data;

OpenIndexBase     ( tmpOfs     :   LONGINT ;
                    staticSize :   LONGINT )
                  (*disp       :*) Data
           *      (*newReg     :*) Data    -> OpenIndexedElem;
```

Für jede „überzählige“ Länge erfolgt eine Anwendung von `OpenIndexOpenBase` jeweils unter Übergabe ihres Offsets. Dem Operator `OpenIndexBase` wird die statische Elementgröße über das Attribut `staticSize` übergeben. Er liefert das Adressierungsart-Nichtterminal

```
aOpenIndexedElem ADRMODE (headerBaseReg:Register; displacementReg:Register);
```

als Resultat, welches in seinen Attributen die Register für *base* und *disp* enthält. Diese ergeben sich aus den Operanden `newReg` und `disp` des Operators.

Der Einsatz eines Adressierungsart-Nichtterminals an dieser Stelle ermöglicht die Formulierung optimierender Regeln, in denen *base* tatsächlich in einem Register repräsentiert werden kann. Darüber hinaus dient ein Wert dieses Nichtterminals einer effizienten Codierung an den Stellen, an denen zu der Adresse des indizierten Array-Elements die „überzähligen" Längen benötigt werden (z.B. bei der Parameterübergabe). Da jedoch eine Regel, deren Resultat ein Adressierungsart-Nichtterminal ist, keine explizite Registerallokation (`SCRATCH`) erlaubt, muß das benötigte Register durch den Operator

```
ConjureRegister -> Data;
```

„herbeigezaubert" werden. Die für seine Überdeckung vorgesehene Regel lautet einfach:

```
RULE ConjureRegister -> LReg;
```

Zur Anwendung des Ergebnisses dieser Berechnung für offene Indizierungen innerhalb einer Adreßrechnung dient der folgende Operator:

```
OpenIndexApplication ( objOfs     : LONGINT ;
                       isStackObj : BOOLEAN )
                       OpenIndexedElem      -> Address;
```

Die Anfangsadresse des Arrays ergibt sich in Abhängigkeit des Attributs `isStackObj` aus der Basisadresse der Verwaltungsfelder des Arrays. Während sich für ein dynamisch angelegtes offenes Array sein Anfang bereits auf Offset `objOfs` relativ zur Basisadresse befindet, steht für offene Array-Parameter auf diesem Offset lediglich die Referenz auf den Array-Anfang.

Zur Vermeidung der temporären Variablen bei Indizierungen eindimensionaler offener Arrays sind optimierende Regeln spezifiziert, welche den an dieser Stelle erwarteten Code erzeugen. Für mehrdimensionale offene Indizierungen lassen sich in ähnlicher Weise optimierende Regeln angeben, die lediglich durch die Endlichkeit der Registermenge eingeschränkt sind.

3.3.2.3.7 Prozeduraufrufe und Parameterübergabe

Eine naheliegende Vorgehensweise bei der Spezifizierung von Parameterübergabe und Prozeduraufrufen wäre, die Codierung durch eine Folge von Top-level-Operatoren für jeden aktuellen Parameter sowie den Prozeduraufruf vorzunehmen. Da jedoch Funktionsprozeduren im Kontext von Ausdrücken vorkommen und BEG den verschachtelten Aufbau von Ausdrucksbäumen nicht gestattet (vgl. S. 30), muß der Prozeduraufruf zusammen mit seinen Parametern in einem Ausdrucksbaum vorliegen. Diese Modellierung erleichtert zudem, den Code des Prozedurbezeichners vor dem Code der aktuellen Parameter abzusetzen, wodurch eine im Prozedurbezeichner eventuell eingesetzte Funktionsprozedur mit Seiteneffekt an der vom Programmierer erwarteten Stelle aufgerufen wird.

Die aktuellen Parameter eines Prozeduraufrufs werden daher über entsprechende Operatoren zu einer Liste des Typs `Arguments` verkettet. Diese Liste wird dann zum Operanden des Operators für die Codierung des Prozeduraufrufs.

Aufruf

Der Operator zur Codierung des Prozeduraufrufs ist ebenfalls kein Top-level-Operator, sondern liefert einen Wert vom Typ `Address`, der lediglich zur Anbindung an die Operatoren

```
NoFuncResult Address;
FuncResultOf (size:tSize) Address -> Data;
```

dient. Während der Top-level-Operator `NoFuncResult` schließlich zum Absetzen des Codes des gesamten Prozeduraufrufs führt, wird das Resultat des Operators `FuncResultOf` als Bestandteil eines Ausdrucks weiterverwendet. Zur Überdeckung von `FuncResultOf` sind drei Regeln spezifiziert, so daß die Registerallokation in Abhängigkeit der als Attribut übergebenen Typgröße `size` erfolgt (vgl. 1.4.11 Funktionsresultate, S. 24).

```
RULE op:FuncResultOf Memory -> r:BReg<al>;
COND{ op.size=b }
CHANGE <ah,ebx,ecx,edx,esi,edi>;

RULE op:FuncResultOf Memory -> r:WReg<ax>;
COND{ op.size=w }
CHANGE <ebx,ecx,edx,esi,edi>;

RULE op:FuncResultOf Memory -> r:LReg<eax>;
COND{ op.size=l }
CHANGE <ebx,ecx,edx,esi,edi>;
```

Die bei jeder Regel angegebene `CHANGE`-Klausel zeigt BEG an, daß Registerinhalte über Prozeduraufrufe hinweg nicht erhalten bleiben. Für die Codierung eines direkten Prozeduraufrufs bzw. eines Aufrufs über eine Prozedurvariable stehen die Operatoren

```
DirectCall   ( paramSpace:LONGINT ; label:LAB.T ) Arguments -> Address;
IndirectCall ( paramSpace:LONGINT ) Data        * Arguments -> Address;
```

zur Verfügung. Zur Stack-Bereinigung wird über das Attribut `paramSpace` die Größe des von den Parametern auf dem Stack belegten Speicherbereichs übergeben. Während dem Operator `DirectCall` über das Attribut `label` die symbolische Adresse der aufzurufenden Routine mitgeteilt wird, benötigt der Operator `IndirectCall` das Register, welches die Zieladresse enthält.

Beim Aufruf einer gebundenen Prozedur muß die Typmarke des Receivers als Parameter übergeben werden und im allgemeinen muß aus dieser Typmarke die Zieladresse des Aufrufs berechnet werden. Da hierbei ein möglicherweise komplexer Bezeichner des aktuellen Receivers nicht doppelt codiert werden darf, erfolgt die Codierung des Aufrufs zusammen mit der Übergabe des Receivers über einen Operator. In Abhängigkeit vom formalen und vom aktuellen Receiver stehen die folgenden Operatoren zur Verfügung.

Formaler Receiver	*Aktueller Receiver*	*Operator*
Zeiger	Zeiger	`BoundCall_FPtr_APtr`
Record	Zeiger	`BoundCall_FRec_APtr`
Zeiger	Record	Verboten
Record	Record	`BoundCall_FRec_ARec`

Die Signatur dieser drei Operatoren hat jeweils die Form:

```
(bprocLab:LAB.T; procOfs:LONGINT; paramSpace:LONGINT)
Data * Arguments -> Address;
```

Für den herkömmlichen Aufruf einer gebundenen Prozedur (`bprocLab=LAB.MT`) erfolgt die Codierung einer indirekten `call`-Instruktion über den Adreßeintrag in der Prozedurtabelle des Deskriptors, der durch die jeweilige Typmarke referenziert wird, auf Offset `procOfs`. Beim Aufruf einer an den Basistyp gebundenen (redefinierten) Prozedur („`r.P^`“) steht die Zieladresse bereits zur Übersetzungszeit fest. Für diesen Fall wird diese (symbolische) Adresse über das Attribut `bprocLab` übergeben, so daß die direkte `call`-Instruktion abgesetzt werden kann.

Das Register `Data` muß für die beiden Operatoren, die bei einem Zeiger als aktuellem Receiver zur Anwendung kommen, den Wert des Zeigers enthalten. Demgegenüber muß bei einem Record als aktuellem Receiver im Register `Data` der Wert der Typmarke angegeben sein. Die Übergabe der Referenz auf den aktuellen Receiver-Record muß in diesem Fall wie ein „normaler“ Parameter bereits innerhalb von `Arguments` erfolgt sein.

Parameter

Für die Übergabe von Parametern sind mehrere Operatoren spezifiziert, die zur Erläuterung in drei Gruppen eingeteilt werden: Adressen und Werte mit Standardtyp werden als „einfache“ Parameter übergeben; „komplexe“ Parameter beinhalten zusätzlich implizit zu übergebende Werte, wie Typmarken und Array-Längen; Parameter, deren Größe zur Übersetzungszeit nicht bekannt ist, werden formalen offenen Array-Parametern mit Elementtyp `SYSTEM.BYTE` als „untypisierte“ Parameter einschließlich ihrer Größe in Byte übergeben. Für die Überdeckung jedes Operators gibt es jeweils genau eine Regel.

`NoParam -> Arguments;`
Konstruktor für eine leere Parameterliste.

`Param Arguments * Data -> Arguments;`
Übergabe von Byte-, Wort- oder Langwort-Werten.

`Param0 Arguments * Data -> Arguments;`
„Übergabe“ eines leeren Records als Val-Parameter.

Zur Übergabe eines komplexen Parameters werden seine einzelnen Bestandteile in einer eigenen Liste mit Typ `Implicits` repräsentiert. Die vollständig aufgebaute Liste wird dann als eigenständiger Parameter in die Parameterliste (`Arguments`) eingefügt. Bereits den beiden initialen Konstruktoren der Liste werden jeweils über entsprechende Attribute alle für die Codierung notwendigen Werte übergeben, so daß die sich anschließenden Operatoren keine weitere Informationen mehr benötigen. Bei einem Record, dessen dynamischer Typ zur Übersetzungszeit bekannt ist, muß die Übergabe der Typdeskriptorreferenz explizit als einfacher Parameter codiert werden.

```
ImplicitArg
(implOfs:LONGINT; objOfs:LONGINT; isStackObj:BOOLEAN)
Address * (*newReg:*)Data -> Implicits;
```
Start von impliziten Parametern für ein Objekt, dessen Verwaltungsfelder auf `Address` beginnen. Das Objekt befindet sich auf `objOfs`, das letzte Verwaltungsfeld auf `implOfs`, jeweils relativ zum Anfang der Verwaltungsfelder im Stack (`isStackObj`). Das Register `newReg` muß „herbeigezaubert“ werden.

```
ImplicitOpenIndexedArg
(implOfs:LONGINT; objOfs:LONGINT; isStackObj:BOOLEAN)
OpenIndexedElem * (*newReg:*)Data-> Implicits;
```
Start von impliziten Parametern für ein Element eines offenen Arrays. `OpenIndexedElem` bekommt den Code der offenen Indizierung. `implOfs`, `objOfs`, `isStackObj` und `newReg`: s.o.

```
ImplConstLen (len:LONGINT) Implicits -> Implicits;
```
Übergabe einer konstanten Länge.

```
ImplOpenLen Implicits -> Implicits;
```
Übergabe einer nichtkonstanten Länge.

```
ImplNewNofElems (useActNofElems:BOOLEAN) Implicits -> Implicits;
```
Übergabe der Gesamtanzahl von Array-Elementen (`nofElems`). In `useActNofElems` wird angezeigt, daß die Gesamtanzahl nicht neuberechnet werden muß, sondern daß sie als Wert im entsprechenden Verwaltungsfeld des aktuellen Parameters zur Verfügung steht.

```
ImplObjReference Implicits -> Implicits;
```
Übergabe der Referenz des Objekts.

```
ImplTypedesc Implicits -> Implicits;
```
Übergabe des Typdeskriptors aus der Typmarke des Objekts.

```
Param_Implicit Arguments * Implicits -> Arguments;
```
Einfügen der Liste der impliziten Parameter in die Parameterliste.

Bei formalen Var-Parametern vom Typ `ARRAY OF SYSTEM.BYTE` muß die zu übergebende Anzahl von Bytes, für den Fall, daß die Größe des aktuellen Parameters zur Übersetzungszeit bekannt ist, als einfacher Parameter explizit übergeben werden (Operator `Param`). Für den Fall, daß es sich um einen indirekt referenzierten Record oder um ein offenes Array handelt, ergibt sich diese Größe aus dem Typdeskriptor bzw. aus den Verwaltungsfeldern. Wird ein offenes Array hier vollständig übergeben, ist die Größe das Produkt aus der statischen Elementgröße und der zur Verfügung stehenden Gesamtanzahl der Array-Elemente. Wird ein Element eines offenen Arrays übergeben, welches selbst ein offenes Array ist, muß die zu übergebende Größe aus den „überzähligen“ Längen und der statische Elementgröße berechnet werden.

```
Param_RecordSizeAndAddr Arguments * Data -> Arguments;
```
Übergabe der Größe und der Referenz eines Records anhand seiner Adresse (`Data`).

```
Param_OArrSizeAndAddr
(objOfs:LONGINT; elemSize:LONGINT)
Arguments * (*objHeaderAddr:*)Data -> Arguments;
```
Übergabe der Größe und der Referenz eines dynamisch angelegten offenen Arrays anhand der Anfangsadresse seiner Verwaltungsfelder (`Data`), dem Offset des Array-Anfangs (`objOfs`) relativ dazu und seiner statischen Elementgröße (`elemSize`).

```
Param_PartialOArrSizeAndAddr
(nofUnindexedLens:LONGINT; ofsOfLastLen:LONGINT;
staticSize:LONGINT; objOfs:LONGINT; isStackObj:BOOLEAN)
Arguments * OpenIndexedElem -> Arguments;
```
Übergabe der Größe und der Referenz des Elements (`OpenIndexedElem`) eines offenen Arrays, welches selbst ein offenes Array ist. Dabei gibt `nofUnindexedLens` die Anzahl der „überzähligen" Längen, `ofsOfLastLen` den Offset der letzten Länge und `staticSize` die statische Elementgröße an.

3.3.2.3.8 Mengenkonstruktoren

Die Codierung von Mengenkonstruktoren erfolgt gemäß folgender Vorgehensweise. Die Bestandteile eines Mengenkonstruktors sind entweder einzelne Zahlen oder Zahlenbereiche. Aus den konstanten Bestandteilen wurde bereits im Frontend die konstante Teilmenge der zu konstruierenden Menge berechnet. Diese konstante Teilmenge wird nun um die Elemente erweitert, welche durch die übrigen Bestandteile gegeben sind. Ist der Bestandteil eine einzelne Zahl, wird diese über den Operator

```
SetExtendByElem (*set:*)Data * (*element:*)Data -> Data;
```

in die Menge aufgenommen. Ein Zahlenbereich wird durch den Operator

```
SetExtendByRange (*set:*)Data * (*a:*)Data * (*b:*)Data -> Data;
```

aufgenommen. Diese Operatoren setzen jeweils zusätzlichen Code für die Bereichsüberprüfung (und die Meldung eines Laufzeitfehlers) ab. Der für Zahlenbereiche abgesetzte Code stützt sich (zur Vermeidung einer Schleife) auf eine im Laufzeitsystem vorhandene Tabelle

$$\begin{aligned} \texttt{BitRangeTab}_0 &= \{\} \\ \texttt{BitRangeTab}_n &= \{0,\ldots,n-1\}, \quad \text{für } 0 < n \leq 32, \end{aligned}$$

wodurch die Aufnahme eines Bestandteils $a\,..\,b$ in die Menge S gemäß

$$S := S \quad \cup \quad \Big(\texttt{BitRangeTab}_{b+1} \quad \cap \quad (\{0,\ldots,31\} \setminus \texttt{BitRangeTab}_a)\Big)$$

kompakt codiert werden kann. Die durch die Tabelle verursachte Vergrößerung des Laufzeitsystems um lediglich 132 Bytes kann für diese „kleine" Optimierung in Kauf genommen werden, da mit ihr eine Mengenerweiterung durch fünf Instruktionen codiert werden kann. Falls a oder b konstant ist und/oder S bisher konstant ist, kommen in naheliegender Weise optimierende Regeln zum Einsatz.

3.3.2.3.9 Boolesche Ausdrücke

Damit die vom Report geforderte Kurzschlußauswertung realisiert wird, erfolgt die Behandlung Boolescher Ausdrücke nach der von AHO, SETHI und ULLMAN [1988] beschriebene Methode zur Übersetzung in Kontrollflußanweisungen (S. 605ff.). Zunächst wird die eingesetzte Methode anhand eines Beispiels skizziert, und anschließend wird die Realisierung beschrieben.

Die Bedingung der If-Anweisung des nachstehenden Oberon-2-Moduls ist ein Boolescher Ausdruck, welcher aus zwei *Booleschen Basisausdrücken* besteht. Ein Boolescher Basisausdruck ist die Applikationen einer Variablen vom Typ `BOOLEAN`, die Anwendung eines Vergleichs oder der Aufruf einer Funktionsprozedur mit Resultatstyp `BOOLEAN`. Für die If-Anweisung werden die rechts daneben stehenden Assembleranweisungen abgesetzt.

```
MODULE M;
   VAR f:BOOLEAN;                          f = 8+M$G
       a:LONGINT;                          a = 12+M$G
       b:LONGINT;                          b = 16+M$G
BEGIN                                      ...
   IF  f                                   testb   $1,f
                                           jz      L3
                                           jmp     L2
   OR                                  L3:
       (a=b)                               movl    b,%eax
                                           cmpl    %eax,a
                                           jnz     L1
                                       #   jmp     L2
   THEN                                L2:
       a:=1;                               movl    $1,a
                                           jmp     L4
   ELSE                                L1:
       a:=2;                               movl    $2,a
   END;                                L4:
END M.                                     ...
```

Jeder Boolesche Basisausdruck appliziert ein *True-Label* und ein *False-Label*, welche die symbolischen Adressen darstellen, an denen in Abhängigkeit des Werts des Ausdrucks die Programmausführung fortgesetzt wird. Zusätzlich deklariert jeder Boolesche Basisausdruck ein *Expression-Label*, welches die symbolische Adresse darstellt, an der die Auswertung des Ausdrucks codiert ist. Das Expression-Label des ersten Booleschen Basisausdrucks eines Booleschen Ausdrucks ist leer. Die Vergabe der benötigten Labels erfolgt nicht innerhalb von Regeln der CGD, da dies praktisch nur über *ererbte* Attribute durchführbar ist. Vom Werkzeug BEG werden jedoch nur *abgeleitete* Attribute unterstützt, so daß die Vergabe bei der Baumtraversierung durchgeführt werden muß. So erfolgt die Berechnung der Labels, die in obigem Beispiel notwendig sind, in der Prozedur, welche zur Codierung der Bestandteile einer If-Anweisung in weitere Prozeduren verzweigt, unmittelbar vor deren Aktivierung.

Die Ausführung eines übersetzten Booleschen Basisausdrucks erfolgt in zwei Schritten: Eine Maschineninstruktion hinterläßt einen bestimmten Zustand in den Bits des Statusregisters `EFLAGS`. Anschließend erfolgt in Abhängigkeit dieses Zustands immer eine Änderung des Kontrollflusses durch eine Folge von einem bedingten Sprung für den Wert

FALSE und einem unbedingten Sprung für den Wert TRUE. Dieser „Zweisprung“ erleichtert die Codierung Boolescher Ausdrücke und führt erst in Verbindung mit der vom Assemblersubsystem durchgeführten Optimierung (vgl. S. 60) zur Erzeugung des erwarteten Codes.

Aufgrund der konzeptionellen Zerlegung in zwei Schritte wird die Codierung Boolescher Basisausdrücke in zwei Operatorgruppen aufgeteilt. Innerhalb der ersten Gruppe gibt es Operatoren für die Applikation einer Booleschen Variablen und für die Anwendung eines Vergleichs. Das Resultat einer Booleschen Funktionsprozedur wird wie eine Boolesche Variable behandelt, da Funktionsergebnisse in einem Register zur Verfügung gestellt werden.

```
Flag          (rel:tRelation) Label *                    Data -> Condition;
Compare       (rel:tRelation) Label * (*a:*)Data * (*b:*)Data -> Condition;
StringCompare (rel:tRelation) Label * (*a:*)Data * (*b:*)Data -> Condition;
```

Diese Operatoren haben den Resultatstyp Condition, der anzeigt, daß der von ihnen abgesetzte Code einen bestimmten Zustand im Statusregister hinterlassen hat. Somit kann der Operator der zweiten „Gruppe“ den obengenannten „Zweisprung“ codieren.

```
Branch
(isSigned:BOOLEAN; trueLabel:LAB.T; falseLabel:LAB.T) Condition -> Boolean;
```

Damit die Deklaration eines Expression-Labels vor allen Instruktionen, die zu einem Booleschen Basisausdruck gehören, abgesetzt wird, besitzen die Operatoren der ersten Gruppe einen ersten Operanden (Label), auf dessen Position das Ergebnis des Operators LabelDef steht, der diese Deklaration absetzt.

```
LabelDef (label:LAB.T) -> Label;
```

Die Relation zum durchzuführenden Vergleich wird bereits den Operatoren der ersten Gruppe durch das Attribut rel angezeigt und über das Nichtterminal

```
aCondition (rel:tRelation);
```

dem Operator zur Codierung der Verzweigung übermittelt. Diese Vorgehensweise ist nötig, da die zur Codierung verwendete cmp-Instruktion als zweiten Operanden keinen unmittelbaren Operanden gestattet. In einem solchen Fall erfolgt durch die entsprechende Regel eine Vertauschung der Operanden der Vergleichsinstruktion mit einer Umkehrung der Relation. Die vom Operator Flag für einen numerisch codierten Booleschen Wert *f* abgesetzte Instruktion

```
testb $1,f
```

entspricht einem Vergleich mit FALSE. Daher muß für das Attribut rel des Operators die Repräsentation der Relation # angegeben werden, damit der nachfolgende bedingte Sprung durch eine jz-Instruktion richtig codiert wird.

Das Attribut `isSigned` zeigt beim Operator `Branch` an, ob der Vergleich vorzeichenbehaftet (für *numerische* Typen) oder vorzeichenlos (für `CHAR`) durchgeführt werden soll. Sein Resultatstyp `Boolean` sowie das zugehörige Speicher-Nichtterminal `aBoolean` dienen der Repräsentation von „Booleschem Kontrollfluß".

Die folgenden Operatoren repräsentieren die Booleschen Oberon-2-Operatoren. Ihnen ist jeweils eine Regel zugeordnet, welche aber keinen Code absetzt. Sie werden lediglich eingesetzt, damit die Struktur eines Booleschen Ausdrucks für den Codeerzeuger zur Spezifizierung optimierender Regeln sichtbar bleibt.

```
Not Boolean           -> Boolean;
And Boolean * Boolean -> Boolean;
Or  Boolean * Boolean -> Boolean;
```

In der Quellsprache kann ein Boolescher Ausdruck entweder als Bestandteil einer Anweisung zur Änderung des Kontrollflusses oder innerhalb einer Wertzuweisung (Zuweisung und Parameterübergabe) vorkommen. Die Zuweisung des Werts eines Booleschen Ausdrucks an eine Variable f erfolgt im allgemeinen explizit gemäß folgendem Schema:

```
IF boolexpr THEN f:=TRUE ELSE f:=FALSE END
```

Zur Codierung der beiden Möglichkeiten von Anwendungen Boolescher Ausdrücke sind zwei Operatoren vorgesehen:

```
NoBoolVal Boolean;
BoolVal (trueLabel:LAB.T; falseLabel:LAB.T) Boolean -> Data;
```

Beim Top-level-Operator `NoBoolVal` werden die Ziel-Labels als symbolische Anfangsadressen der jeweiligen Anweisungsfolge deklariert. Der Operator `BoolVal` erhält die gleichen Labels, die auch der zugehörige Boolesche Ausdruck als Ziel-Labels bekommen hat, und setzt die beiden Zuweisungen jeweils zusammen mit der Deklaration ihres Labels ab.

Die Booleschen Operatoren `IN` und `IS`, die vordeklarierte Funktionsprozedur `ODD` sowie die im Modul `SYSTEM` deklarierten Funktionsprozeduren `BIT` und `CC` lassen sich nicht auf die gleiche Art codieren. Daher gibt es für ihre Codierung jeweils einen eigenen Operator, der direkt den Resultatstyp `Boolean` besitzt.

Die somit ermöglichte Codeerzeugung für Boolesche Ausdrücke ist vor allem in denjenigen Fällen nicht besonders effizient, in denen die ganzzahlige Codierung Boolescher Werte benötigt wird (z.B. `f:=(a>b)`). Daher ist für diese Fälle eine optimierende Regel vorgesehen, welche unter Einsatz der `set`*cc*-Instruktion[1] wesentlich besseren Code absetzt. Die vorgenommene Aufteilung über `Condition` und `Boolean` gestattet es, diese Optimierung besonders elegant zu spezifizieren.

```
RULE BoolVal
       op:Branch cc:aCondition
     -> o:BReg;
COST 1;
EMIT{ oper:=ASM.FlagSetOperTab[cc.rel,op.isSigned];
      ASM.C1 ( oper  ,  R(o) ); }
```

[1] Die `set`*cc*-Instruktion setzt das vom (einzigen) Operanden adressierte Byte in Abhängigkeit von *cc* auf den Wert 1 oder 0.

Der Einsatz dieser „Flag-setzenden“ Instruktion muß demgegenüber bei den Operatoren, die sich nicht über `Condition` und `Boolean` aufteilen ließen, jeweils durch eine eigene optimierende Regel vorgenommen werden. Zusätzlich sind Regeln spezifiziert, die für konstante Operanden beim Oberon-2-Operator `IN` besseren Code absetzen.

3.3.2.3.10 Vordeklarierte Prozedur `COPY`

Beim Aufruf der vordeklarierten Prozedur `COPY` wird der für den Kopiervorgang notwendige Code direkt an der Aufrufstelle abgesetzt (*Inline*-Codierung). Der eigentliche Kopiervorgang ist durch die nachfolgende Prozedur beschrieben.

```
PROCEDURE COPY(src:ARRAY OF CHAR; VAR dst:ARRAY OF CHAR);
VAR i,n:LONGINT;
BEGIN
 IF LEN(src)<LEN(dst) THEN n:=LEN(src); ELSE n:=LEN(dst); END;
 FOR i:=0 TO n-2 DO
  dst[i]:=src[i];
  IF dst[i]=0X THEN RETURN; END;
 END;
 dst[n-1]:=0X;
END COPY;
```

Da somit für jedes Argument von `COPY` sowohl seine Adresse als auch seine Länge vorliegen muß, ist für die Modellierung der Argumentreferenzen die Verwendung von Adressierungsart-Nichtterminalen nicht ausreichend. Daher werden lediglich die Adressen jeweils als Wert über ein Register-Nichtterminal an die Regel zum Absetzen des `COPY`-Codes übermittelt. Die Längen der Argumente werden über den Stack übergeben.

Hierbei werden Zeichen-Arrays mit fester Länge, offene Zeichen-Arrays und offene Zeichen-Arrays nach offener Indizierung unterschieden. Für jede dieser Arten gibt es jeweils einen eigenen Operator. Ausgehend von der Deklaration dreier Variablen

```
VAR s : ARRAY n OF CHAR;
    t : POINTER TO ARRAY OF CHAR;
    u : POINTER TO ARRAY OF ARRAY OF CHAR;
```

sind die Bezeichner `s`, `t^` und `u[`i`]` jeweils der einfachste Repräsentant einer Art. Zeichenkettenkonstanten werden hierbei wie Zeichen-Arrays mit fester Länge behandelt.

Argumentart	*Bezeichner*	*Operator*
Zeichen-Array mit fester Länge	`s`	`ImplicifyConst`
Offenes Zeichen-Array	`t^`	`Implicify`
Offenes Zeichen-Array nach offener Indizierung	`u[`i`]`	`ImplicifyOpenIndexed`

Diese Operatoren liefern jeweils Datencode, welcher bei seiner Ausführung zur Folge hat, daß die Array-Adresse in einem Register vorliegt und die Länge des Arrays auf dem Stack abgelegt ist.

```
ImplicifyConst       ( len           :   LONGINT )
                     (*objectLoc     :*) Address -> Data;

Implicify            ( lenOfs        :   LONGINT ;
                       isStackObject :   BOOLEAN ;
                       objOfs        :   LONGINT )
                     (*objHeaderLoc  :*) Address -> Data;

ImplicifyOpenIndexed ( lenOfs        :   LONGINT ;
                       isStackObject :   BOOLEAN ;
                       objOfs        :   LONGINT )
                       OpenIndexedElem           -> Data;
```

Den Operatoren für offene Zeichen-Arrays wird hierbei durch `lenOfs` der Offset der relevanten Länge übergeben. Das Attribut `isStackObject` zeigt an, ob es sich bei dem Array um einen offenen Array-Parameter oder ein dynamisch angelegtes offenes Array handelt. Die Codierung der Berechnung der Array-Adresse erfolgt aufgrund des unterschiedlichen Speicher-Layouts in Abhängigkeit dieses Attributs. Der von diesen Operatoren erzeugte Datencode für die beiden Argumente von `COPY` wird dem Operator

```
StrCopyArguments (*srcReg:*)Data * (*dstReg:*)Data -> StrCopyArgs;
```

übergeben. Die spezifizierte Regel zur Überdeckung dieses Operators stellt sicher, daß das Register `edi` für die Zieladresse und `esi` für die Quelladresse bei der Registerallokation ausgewählt werden, indem die zulässigen Register bei dieser Regel entsprechend eingeschränkt werden. Der von dieser Regel abgesetzte Code legt das Minimum der beiden auf dem Stack übergebenen Argumentlängen im Register `ecx` ab. Der Resultatstyp `StrCopyArgs` dient dazu, daß auf das Resultat nur der Operator

```
StrCopy StrCopyArgs;
```

anwendbar ist. Durch die Regel zur Überdeckung dieses Operators wird somit unter Verwendung der `lodsb`- und `stosb`-Instruktion[1] eine effiziente Kopierroutine abgesetzt.

Für Zeichenketten der Länge 0 oder 1 als Quellargument erfolgt über den Operator `ShortConstStrCopy` eine kompaktere Codierung von `COPY`.

Die „Weitergabe“ der Argumentlängen über den Stack wirkt sich natürlich ungünstig auf die Laufzeit aus. Daher gibt es optimierende Regeln für die gebräuchlichsten Argumentkombinationen, bei denen die Längen leicht zu bestimmen sind. Falls das Minimum zur Übersetzungszeit bereits bekannt und klein ist (≤ 4), erfolgt als weitere Optimierung eine „Entrollung“ der Kopierschleife.

[1] „Load String“ und „Store String“ haben bei gelöschtem Direction Flag die Semantik `al:=MEM(esi); esi:=esi+1` bzw. `MEM(edi):=al; edi:=edi+1`.

3.3.2.3.11 Funktionsprozedur SYSTEM.VAL

Die Anwendung der im Modul SYSTEM deklarierten Funktionsprozedur VAL(T,x) liefert ein Objekt vom Typ T, dessen binäre Darstellung identisch mit dem Wert von x ist. Der Ausdruck x hat den Typ T_x. Bei der Codierung dieser Funktionsprozedur ist zu beachten, daß eventuelle Größenunterschiede durch Abschneiden bzw. Auffüllen mit Nullbytes ausgeglichen werden. Zudem besteht das Problem, daß sowohl das Argument als auch das resultierende Objekt jeweils in Abhängigkeit seines Typs unterschiedlich angesprochen wird: Während bei einem strukturierten Typ lediglich die Referenz auf das Objekt gehandhabt wird, muß sich bei einem Standardtyp der Wert in einem Register befinden.

Die Vielzahl der zu betrachtenden Fälle, die sich aus diesen Kombinationen ergeben, wird zur Lösung dadurch eingeschränkt, daß mit Hilfe einer temporären Variablen als Zwischenziel konzeptionell die Realisierung in zwei einfachen Schritten erfolgt. Die temporäre Variable liegt auf dem Stack und ihre Größe ergibt sich aus dem Maximum über SIZE(T) und SIZE(T_x).

Im ersten Schritt wird die temporäre Variable mit dem Wert von x geladen und mit Nullbytes aufgefüllt, falls der Größenunterschied dies anzeigt. Entsprechend der obigen Unterscheidung stehen zur Codierung der Argumentanbindung die beiden folgenden Operatoren zur Verfügung.

```
Addr2Retype (srcLen:LONGINT; dstLen:LONGINT; tmpOfs:LONGINT) Address -> Retype;
Data2Retype (srcLen:LONGINT; dstLen:LONGINT; tmpOfs:LONGINT) Data    -> Retype;
```

Ihnen wird der Wert SIZE(T_x) und SIZE(T) sowie der Offset der temporären Variablen relativ zum Register ebp übergeben. Das zu ihrem Resultat gehörende Speicher-Nichtterminal

```
aRetype COND_ATTRIBUTES (dstLen:LONGINT; tmpOfs:LONGINT);
```

übermittelt über seine Attribute den „Folgeoperatoren“ die für den zweiten Schritt notwendigen Werte. Deshalb benötigen diese Operatoren keine Attribute:

```
Retype2Data Retype -> Data;
Retype2Addr Retype -> Address;
```

Für die Fälle, in denen die Umtypisierung durch SYSTEM.VAL direkt erfolgen kann, sind optimierende Regeln spezifiziert, welche die (ineffiziente) Benutzung der temporären Variablen vermeiden.

3.3.2.3.12 Optimierung zuweisender Operationen

Die meisten arithmetischen Instruktionen der Zielmaschine gestatten einen Speicheroperanden als Zieloperanden. Dies ermöglicht eine effiziente Codierung von *zuweisenden Operationen*, also Zuweisungen, deren Zielvariable im Zuweisungsausdruck vorkommt (z.B. `v:=v*8`). Hierbei zulässige Variablen sind globale Variablen und lokale Variablen auf beliebiger Schachtelungstiefe sowie Record-Felder von zulässigen Variablen. Für die Zusammenfassung der drei Arten zulässiger Variablen ist das Nichtterminal

```
AVar ADRMODE COND_ATTRIBUTES (var:ASM.tVariable);
```

spezifiziert, welches aufgrund der eingeschränkten Syntax der Spezifikationssprache BEGL nicht zusammen mit den optimierenden Regeln angegeben werden kann. Das Attribut zur Repräsentation einer konkreten zulässigen Variable ist von folgendem Typ:

```
tVariable = RECORD
             label      : tLabel;
             frame, ofs : LONGINT;
             tmpreg     : tReg;
             log2       : LONGINT;
            END;
```

Globale Variablen werden durch Werte in den Feldern `label` und `ofs` definiert. Der Bezug auf lokale Variablen ergibt sich aus den Werten der Felder `frame` und `ofs`. Das durch das Feld `tmpreg` repräsentierte Register wird bei umgebenden lokalen Variablen zum Halten der Aktivierungssegmentadresse benötigt. Aus einem Wert vom Typ `tVariable` konstruiert die Funktionsprozedur `ASM.Variable` einen Operanden vom Typ `ASM.tOp` zum Absetzen von Instruktionen.

Die folgenden Regeln beschreiben jeweils den Übergang einer zulässigen Variablen zum Nichtterminal `AVar`.

```
RULE gv:GlobalVariable -> v:AVar;
EVAL{ v.var.label    := gv.label;
      v.var.frame    := 0;
      v.var.ofs      := gv.adr;
      v.var.tmpreg   := RegNil; }

RULE lv:LocalVariable -> v:AVar;
EVAL{ v.var.label    := LAB.MT;
      v.var.frame    := 0;
      v.var.ofs      := lv.adr;
      v.var.tmpreg   := RegNil; }

RULE sel:Selector
        PointerFrom
           ContentOf lv:LocalVariable
     -> v:AVar;
EVAL{ v.var.label    := LAB.MT;
      v.var.frame    := sel.ofs;
      v.var.ofs      := lv.adr;
      v.var.tmpreg   := RegNil; }
```

```
RULE sel:Selector i:AVar -> o:AVar;
EVAL{ o.var:=i.var; INC(o.var.ofs,sel.ofs); }
```

Zur Behandlung von monadischen und dyadischen zuweisenden Operationen gibt es nun mehrere optimierende Regeln. Da die Erkennung zuweisender Operationen über Muster nicht vollständig definiert werden kann, besitzen die zugehörigen Regeln Bedingungsklauseln, zum Test der Gleichheit der relevanten Variablen über das Prädikat `ASM.AreEqualVariables`. Die wesentliche Regel zur Optimierung monadischer zuweisender Operationen lautet dann:

```
RULE SimpleAssignment
        dst:AVar
        op:MonOper
           s:ContentOf src:AVar;
COST 1;
SCRATCH tmp <eax..edi>;
COND{ ASM.AreEqualVariables(dst.var,src.var) }
EMIT{ src.var.tmpreg:=tmp;
      ASM.CS1( op.code,s.size  ,  Variable(src.var) ); }
```

Bei der entsprechenden Regel zur Optimierung von dyadischen zuweisenden Operationen ist die Multiplikation durch einen Zusatz in der Bedingungsklausel explizit ausgenommen, da die `imul`-Instruktion einen Speicheroperanden als Zieloperanden nicht unterstützt. Lediglich die Multiplikation einer Variablen mit einer Konstanten, die eine Zweierpotenz ist, kann durch eine eigene Regel spezifiziert werden, da hier die `shl`-Instruktion eingesetzt werden kann. Entsprechendes gilt für die Division.

Der Test, ob eine Zahl x eine Zweierpotenz ist, erfolgt durch Aufruf der Funktionsprozedur `ASM.IntLog2`, wobei gleichzeitig der Wert $y = \log_2 x$ effizient berechnet wird. Da der Wert y, der innerhalb der Bedingungsklausel berechnet wird, für das Absetzen der Shift-Instruktion benötigt wird, von BEG jedoch keine „regellokalen“ Variablen unterstützt werden, dient das zusätzliche Feld `log2` des Typs `ASM.tVariable` der Übermittlung dieses Werts.

3.3.2.3.13 `Spill` und `Restore`

Obwohl im generierten Codeerzeuger die 8 Bit breiten Teilregister des Prozessors verwendet werden, gilt die Konvention, daß nur Langwortwerte auf dem Stack abgelegt werden. Dies erfordert innerhalb der für die Registerallokation benötigten Prozeduren `Spill` und `Restore` eine Fallunterscheidung in Abhängigkeit des zu sichernden bzw. zu restaurierenden Registers.

```
PROCEDURE Spill(reg:Register; loc:Spilllocation);
BEGIN
 CASE reg OF
 |Regah,Regbh,Regch,Regdh:       ASM.CS2( sub,l  ,  ASM.i(4),R(esp)           );
                                 ASM.CS2( mov,b  ,  R(reg),B(esp)             );
 |Regal,Regbl,Regcl,Regdl,Regax..Regedi:
                                 ASM.C1 ( pushl  ,  R(ASM.SizedRegTab[reg,l]) );
 ELSE
                                 ASM.C1 ( pushl  ,  R(RegNil)                 );
 END;
END Spill;

PROCEDURE Restore(reg:Register; loc:Spilllocation);
BEGIN
 CASE reg OF
 |Regal..Regdh:                  ASM.CS2( mov,b  ,  B(esp),R(reg)             );
                                 ASM.CS2( add,l  ,  ASM.i(4),R(esp)           );
 |Regax..Regedi:
                                 ASM.C1 ( popl   ,  R(ASM.SizedRegTab[reg,l]) );
 ELSE
                                 ASM.C1 ( popl   ,  R(RegNil)                 );
 END;
END Restore;
```

3.3.2.3.14 Fließkommaarithmetik

Da BEG im Rahmen der Registerallokation eine kellerartige Organisation von Registern nicht unterstützt, erfolgt die Modellierung von Fließkommawerten des mathematischen Koprozessors nicht über Register-Nichtterminale. Statt dessen werden diese Register explizit verwaltet (siehe Abschnitt 3.3.1.3, S. 60), so daß zur Modellierung der Werte lediglich ein Speicher-Nichtterminal `Float` eingesetzt wird.

Das Lesen eines Fließkommawerts aus dem Hauptspeicher wird durch den Operator

```
FloatContentOf (size:tSize) Address -> Data;
```

realisiert, dessen Attribut `size` anzeigt, ob es sich hierbei um einen Wert vom Typ `REAL` (`s`) oder vom Typ `LONGREAL` (`l`) handelt. Der zu diesem Operator gehörende Knoten wird im allgemeinen von zwei Regeln überdeckt. Diese Aufteilung auf eine Regel und eine Kettenregel gestattet, innerhalb optimierender Regeln, bei dyadischen Fließkommaanweisungen den Einsatz der Speicheradressierung für den zweiten Operanden.

```
RULE op:FloatContentOf m:Memory -> o:AMem;
COST 1;
EVAL{ o.size:=op.size; }
EMIT{ o.oper.kind := ASM.okMemory;
      o.oper.loc  := m.loc; }

RULE i:AMem -> Float;
COST 1;
EMIT{ NDP.CS1( fld,i.size  ,  Operand(i.oper) ); }
```

In den nichtoptimierenden Regeln sind die Operanden dyadischer Anweisungen immer die obersten beiden Kellerelemente. Für dyadische (nichtkommutative) Fließkommaoperationen steht der Operator

```
FloatDyOper (code:ASMOP.tOper) Data * Data -> Data;
```

zur Verfügung, dessen Attribut `code` die auszuführende Maschineninstruktion anzeigt (`fsub` oder `fdiv`).

Einige Instruktionen des mathematischen Koprozessors gibt es in verschiedenen Versionen. Während die „Grundversion“ einer zweistelligen Fließkommainstruktion lediglich das Kellerelement des ersten Operanden durch das Resultat ersetzt, nimmt die „`p`-Version“ dieses Element vom Keller und schreibt das Resultat in das Kellerelement des zweiten Operanden (z.B. `faddp`). Zusätzlich gibt es die Instruktionen für die Subtraktion und Division jeweils in einer „`r`-Version“, welche die reziproke Operation durchführt (z.B. `fdivrp`). Die Abbildung einer Instruktion auf die jeweilige Version ist über die Tabellen `PopFloatTab` und `RevFloatTab` im Modul `ASMOP` implementiert.

Die zum Operator `FloatDyOper` gehörende nichtoptimierende Regel lautet:

```
RULE op:FloatDyOper Float Float -> Float;
COST 1;
EMIT{ NDP.C0( ASMOP.RevFloatTab[ASMOP.PopFloatTab[op.code]] ); }
```

Da sich hier beide Operanden auf dem Registerkeller befinden, muß die Instruktion in der `p`-Version durchgeführt werden. Bedingt durch die Ausführungsreihenfolge der Regelaktionen (von Unterbäumen) liegt der erste Operand der dyadischen Operation im Registerkeller unterhalb des zweiten Operanden. Die dyadischen Instruktionen der Grundversion erwarten jedoch die Operanden in umgekehrter Reihenfolge, so daß zusätzlich die Instruktion auf ihre `r`-Version abgebildet werden muß.

Für den Einsatz der Speicheradressierung gibt es optimierende Regeln, die über das Nichtterminal `AMem` einen Speicheroperanden für dyadische Fließkommaoperationen erlauben.

```
RULE op:FloatDyOper a:AMem Float -> Float;
COST 1;
EMIT{ NDP.CS1( ASMOP.RevFloatTab[op.code],a.size  ,  Operand(a.oper) ); }

RULE op:FloatDyOper Float a:AMem -> Float;
COST 1;
EMIT{ NDP.CS1( op.code,a.size                     ,  Operand(a.oper) ); }
```

Für kommutative Fließkommaoperationen, welche durch den kommutativen Operator

```
FloatSymDyOper (code:ASMOP.tOper) Data + Data -> Data;
```

konstruiert werden, genügt demgegenüber die Angabe einer einzigen optimierenden Regel, da BEG in diesem Fall die „gespiegelte“ Regel hinzufügt. Dieser Mechanismus ist deshalb anwendbar, da in beiden Fällen derselbe Code erzeugt werden kann. Die Regel lautet:

```
RULE op:FloatSymDyOper a:AMem Float -> Float;
COST 1;
EMIT{ NDP.CS1( op.code,a.size  ,  Operand(a.oper) ); }
```

Das Schreiben von Fließkommawerten in den Hauptspeicher erfolgt bei einer Zuweisung oder einer Parameterübergabe durch die folgenden Operatoren:

```
FloatAssignment (size:tSize) Address   * Data;
FloatParam      (size:tSize) Arguments * Data -> Arguments;
```

Die erzeugten Knoten werden durch die folgenden Regeln überdeckt.

```
RULE op:FloatAssignment m:Memory Float;
COST 1;
EMIT{ NDP.CS1( fstp,op.size  ,  Loc(m.loc) ); }

RULE op:FloatParam Args Float -> Args;
COST 2;
EMIT{ ASM.CS2( sub,l         ,  i(ASM.FloatByteSizeTab[op.size]),R(esp) );
      NDP.CS1( fstp,op.size  ,  B(esp)                                  ); }
```

3.3.2.3.15 Kosten für Regeln

Bei der schrittweisen Erstellung der Spezifikation ist zunächst eine vollständige Überdeckung aller relevanten Ausdrucksbäume zu erreichen. Die dafür spezifizierten Regeln weisen keine Redundanz auf, so daß die Angabe von Kosten in diesem Schritt entfallen kann. Erst beim Hinzufügen von optimierenden Regeln, die jeweils eine kürzere Folge von Instruktionen für eine Zusammenfassung von mehreren Knoten des Ausdrucksbaums absetzen, wird die Kostenangabe relevant. Es hat sich gezeigt, daß eine Kostenfunktion, welche lediglich die Anzahl der abgesetzten Instruktionen angibt, ausreichend ist, um BEG die spezifizierten optimierenden Regeln auswählen zu lassen.

3.3.2.3.16 Testausgabe (`BETO`)

In der Codegeneratorbeschreibung ist die Option `test` vermerkt. Deshalb generiert BEG den Codeerzeuger einschließlich entsprechender Routinen zur Testausgabe (vgl. S. 42). Diese stützen sich für sämtliche in Attributdefinitionen verwendeten Typen auf jeweils eine Prozedur zur Ausgabe eines Werts dieses Typs. Die hierfür zu deklarierenden Prozeduren sind im Modula-2-Modul `BETO` (Back End Test Output) versammelt.

Das von BEG generierte Modul `IR` exportiert die Booleschen Variablen `OptEmitIR`, `OptEmitMatch` und `OptEmitAlloc`. Über diese Flags kann zur Laufzeit die Ausgabe eines aufgebauten Ausdrucksbaums, die Ausgabe der ausgewählten Regeln bei der Überdeckung sowie die Ausgabe der relevanten Register bei der Registerallokation aktiviert werden. Diese Flags werden über Kommandozeilenoptionen gesetzt.

3.3.2.3.17 Nachträgliche Umbenennung

Aufgrund der ungünstigen Namenswahl einer Variablen innerhalb des von BEG generierten Modula-2-Moduls `Emit` wird in diesem Modul nach seiner Generierung eine automatische Umbenennung der Variable vorgenommen.

Es handelt sich bei der Variable mit dem Namen `i` um eine lokale Variable der Prozedur `EmitStatement`. In diese Prozedur werden die innerhalb der BEG-Regeln in den `EMIT`-Abschnitten angegebenen Anweisungen kopiert. Der Name `i` überdeckt dabei die gleichnamige Funktionsprozedur zur Konstruktion unmittelbarer Operanden mit ganzzahligem Wert aus dem Assemblersubsystem (vgl. S. 56). Zur besseren Lesbarkeit des abgesetzten Assemblercodes innerhalb der BEG-Regeln werden diese Funktionsprozeduren zur Konstruktion von Operanden immer unqualifiziert verwendet.

Da die Variable lediglich an einer Stelle innerhalb der Prozedur verwendet wird, um einen Programmabbruch zu erzeugen, wird ihr Name nach erfolgreicher Generierung in `yyDIV0var` geändert. Der generierte Code hat nach dieser Manipulation an der betreffenden Stelle folgendes Aussehen:

```
...
VAR yyDIV0var:INTEGER;
BEGIN
   ...
   IR.Error ('internal error');
   yyDIV0var:=0; yyDIV0var:=1 DIV yyDIV0var; HALT;
   ...
```

3.3.2.4 Baumtraversierung (`CODE.pum`)

Zur Übersetzung muß der attributierte abstrakte Syntaxbaum traversiert werden. Dabei müssen die für die Codeerzeugung zur Verfügung stehenden Prozeduren aufgerufen werden. Die Baumtraversierung stützt sich hierbei hauptsächlich auf die Ausdrucksbaumkonstruktoren des generierten Codeerzeugers. An bestimmten Stellen wird zusätzlich das Zielsystem zur Erzeugung des Modulskeletts sowie die schematische Codeerzeugung für Deskriptoren und den Prolog von Prozeduren eingesetzt. An manchen „einfachen“ Stellen wird direkt Assemblercode abgesetzt.

Sowohl die Definition des abstrakten Syntaxbaums als auch die Definition der Evaluatorobjekte ist als Ast-Spezifikation gegeben. Aus diesem Grund wird die Traversierung mit dem Werkzeug Puma vollständig rekursiv beschrieben. Dadurch kann von der internen Repräsentation der Knotentypen in Modula-2 abstrahiert werden.

Wegen ihrer Vielzahl werden die zu den einzelnen Abschnitten im Syntaxbaum gehörenden Puma-Prozeduren gemäß ihres thematischen Zusammenhangs in Einfügungsdateien

ausgelagert. Die Spezifikationsdatei `CODE.pum` beschreibt die Behandlung eines Moduls und seiner Konstanten- und Prozedurdeklarationen. Die Einfügungsdateien mit dem Namenspräfix `CODE.pum` enthalten die Puma-Prozeduren für die Traversierung von

- Anweisungen (`.Stmts`),
- Prozeduraufrufen und Parametern (`.ProcCalls`),
- Zuweisungen (`.Assignments`),
- Ausdrücken (`.Exprs`),
- Booleschen Ausdrücken (`.BooleanExprs`),
- Bezeichnern (`.Designators`),
- Vordeklarierten Prozeduren (`.Predecls`).

Der Präprozessor `cpp` wird neben der Verwendung zum Einfügen dieser Dateien in die Spezifikationsdatei auch zur Expandierung zweier Makros benutzt, die es ermöglichen, zu Testzwecken während der Traversierung die Namen der besuchten Knoten ausgeben zu können.

Puma-Prozeduren, in denen über Regeln alle möglichen Fälle vollständig unterschieden sein müssen, enthalten zusätzlich eine letzte Regel, welche immer anwendbar ist und die über die Prozedur `ERR.Fatal` einen möglichen Fehler in der Spezifikation signalisiert.

Der „äußeren“ Sicht der Zwischensprachoperatoren, als Funktionen, welche Code erhalten und Code liefern, wird bei der Baumtraversierung durch eine Namenskonvention Rechnung getragen (vgl. S. 63). Dazu werden gemäß folgender Tabelle Zwischensprachtypen durch Redeklaration umbenannt, und für Variablen werden entsprechende Namen gewählt.

Zwischensprachtyp	*Neuer Typname*	*Variable (Suffix)*
`Cons.Address`	`ACode`	`acode (Acode)`
`Cons.Data`	`DCode`	`dcode (Dcode)`
`Cons.Condition`	`CondCode`	`condcode`
`Cons.Boolean`	`BoolCode`	`boolcode`
`Cons.Arguments`	`ArgCode`	`argcode`
`Cons.Implicits`	`ImplCode`	`implcode`
`Cons.Retype`	`RetypeCode`	`retypecode`

Nachfolgend wird ein Überblick über die zur Baumtraversierung gehörenden Dateien jeweils in einem eigenen Abschnitt gegeben. An den Stellen, an denen die Codeerzeugung nicht über Operatoren spezifiziert ist, wird explizit darauf eingegangen. Wegen der großen Anzahl der spezifizierten Puma-Prozeduren in den einzelnen Dateien gibt es in jedem Unterabschnitt eine Abbildung, aus welcher für jede Prozedur der Datei jeweils die „oberen“ und „unteren“ Kontexte innerhalb der Aufrufhierarchie ersichtlich sind.

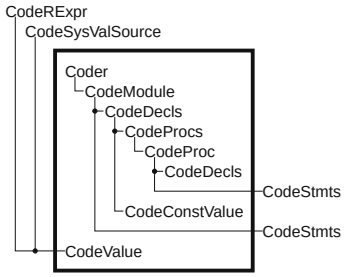

Abbildung 16: Aufrufhierarchie zur Traversierung eines Moduls

3.3.2.4.1 Modul und Deklarationen

Die Spezifikationsdatei `CODE.pum` enthält die einzige vom Puma-Modul `CODE` exportierte Prozedur `Coder`, welche die Traversierung des Syntaxbaums (`FIL.ActP^.TreeRoot`) veranlaßt. In den Prozeduren dieses Puma-Moduls ist die Traversierung der Deklarationen aller Konstanten und Prozeduren eines Moduls spezifiziert.

Die Puma-Prozedur `CodeModule` erzeugt den globalen Rahmen eines Moduls und stützt sich dabei ausschließlich auf das Zielsystem und die schematische Codeerzeugung. Letztere wird ebenfalls von der Puma-Prozedur `CodeProc` eingesetzt, welche zusätzlich die Assembleranweisungen für den Epilog direkt absetzt. In der Prozedur `CodeConstValue` werden die Werte der deklarierten Speicherkonstanten über das Modul `CV` codiert.

3.3.2.4.2 Anweisungen (`.Stmts`)

In dieser Einfügungsdatei ist die Traversierung der Oberon-2-Anweisungen spezifiziert, mit Ausnahme des Prozeduraufrufs und der Return-Anweisung sowie der Zuweisung. Die Prozedur `CodeStmt` delegiert die Codierung einer Anweisung an weitere Puma-Prozeduren. Für eine Return-Anweisung ohne Ausdruck stützt sie sich auf den generierten Codeerzeuger. Lediglich die Codierung von Loop- und Exit-Anweisungen wird durch direktes Absetzen von Assembleranweisungen bereits in der Prozedur `CodeStmt` vorgenommen.

Für eine If-, While- und Repeat-Anweisung mit konstanter Bedingung findet eine Elimination von unerreichbarem Code in naheliegender Weise statt. Darüberhinaus erfolgt bei einer If-Anweisung eine Optimierung der Sprünge, falls die auszuführenden Anweisungsfolgen leer sind oder mit einer Exit-Anweisung beginnen.

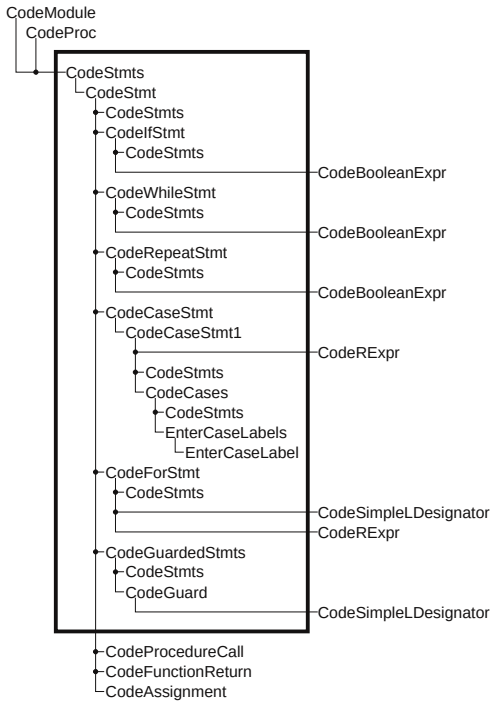

Abbildung 17: Aufrufhierarchie zur Traversierung von Anweisungen

Innerhalb der Traversierung von Case-Anweisungen dient die Prozedur `CodeCaseStmt1` lediglich der Zusammenfassung der beiden möglichen Typen des Case-Ausdrucks, so daß in ihr die eigentliche Codierung einer Case-Anweisung erfolgt. Der Code einer Case-Anweisung besitzt folgenden schematischen Aufbau, wobei *min* den Wert der kleinsten und *max* den Wert der größten Case-Marke darstellt.

```
        index ← „Wert des Case-Ausdrucks“
        if index ∉ [min,max] then jmp Else end
        jmp (Table_index-min)

L1:     „Anweisungen des ersten Falls“; jmp End
        ...
Ln:     „Anweisungen des letzten Falls“; jmp End
Table:  .long L0
        ...
        .long L_max-min
Else:   „Anweisungen des Else-Falls“
End:
```

Die Auswertung des Case-Ausdrucks und die Verzweigung auf die einzelnen Case-Fälle wird über den vom generierten Codeerzeuger zur Verfügung gestellten Operator `CaseExpr` codiert.

Daher muß nur noch die Codierung der einzelnen Case-Fälle mittels `CodeCases` veranlaßt werden, wobei die Sprungtabelle ermittelt und anschließend abgesetzt wird. Hierzu besitzt `CodeCaseStmt1` das Array `LabelTab` (4096 Elemente) zur Aufnahme der Labels für die einzelnen Fälle, das mit dem Label des Else-Falls initialisiert wird. Bei der Traversierung eines Case-Falls wird dann das Label dieses Falls für alle seine Case-Marken durch `EnterCaseLabels` eingetragen.

In der Prozedur `CodeForStmt` erfolgt die Codierung einer For-Anweisung der Form

`FOR` *v* `:=` *beg* `TO` *end* `BY` *step* `DO` „Anweisungen“ `END`

bei positiver Schrittweite gemäß folgendem Schema:

temp ← *end*; *v* ← *beg*
`jmp Cond`
`Loop:` „Anweisungen“

v ← *v* + *step*
`Cond:` **if** *v* ≤ *temp* **then** `jmp Loop` **end**

Die Erhöhung der Kontrollvariablen sowie der Test werden dabei durch den Operator `ForStmt` codiert.

Die Übersetzung von With-Anweisungen beschränkt sich auf die Codierung der Folge von Verzweigungen durch `CodeGuardedStmts` jeweils in Abhängigkeit des Ergebnisses eines Typtests, codiert durch `CodeGuard`.

3.3.2.4.3 Prozeduraufrufe und Parameter (`.ProcCalls`)

Die Puma-Prozedur `CodeProcedureCall` nimmt lediglich die Unterscheidung zwischen vordeklarierten Prozeduren bzw. Prozeduren des Moduls `SYSTEM` und sonstigen Prozeduren vor. Während im ersten Fall die weitere Traversierung an `CodePredeclProcs` delegiert wird, stützt sie sich im anderen Fall auf die Prozedur `CodeLDesignator`, die den Bezeichner des Prozeduraufrufs codiert und `CodeArguments` aufruft.

Das Absetzen der zum Aufruf gehörenden `call`-Instruktion durch den entsprechenden Operator erfolgt während der Traversierung des Bezeichners in `CodeLDesignator` falls Argumente auftreten.

Da diese Puma-Prozedur offensichtlich Adreßcode liefert (L-value eines Bezeichners), müssen diese Operatoren ebenfalls ein Ergebnis dieses Typs liefern. Funktionsprozeduren kommen nur in Ausdrücken als Bezeichner vor und müssen jeweils einen Wert ergeben. Daher erfolgt die Einbindung des Ergebnisses des Prozeduraufrufs in die Wertberechnung in der Puma-Prozedur `CodeRDesignator`.

Innerhalb der Puma-Prozedur `CodeArguments` erfolgt die postorder-Traversierung einer Parameterliste, wobei die Behandlung der Parameter in Abhängigkeit der Übergabeart an weitere Prozeduren delegiert wird (vgl. S. 23).

In `CodeRefdValParam` wird zur Codierung der Übergabe an formale offene Array-Parameter auch die Prozedur `CodeOpenArrayParam` verwendet. Zur Übergabe einer Zeichenkette erfolgt die Behandlung in `CodeOpenArrayStringParam`.

Bei der Übergabe an ein offenes Array müssen die Längen des aktuellen Parameters als implizite Parameter zusätzlich übergeben werden. Ist der aktuelle Parameter ebenfalls ein offenes Array, so sind die Längen äußerer Dimensionen nicht konstant. Über eine postorder-Traversierung aller offenen Dimensionen des formalen Parametertyps werden alle impliziten Parameter durch die Prozedur `CodeImplicitConstLens` codiert.

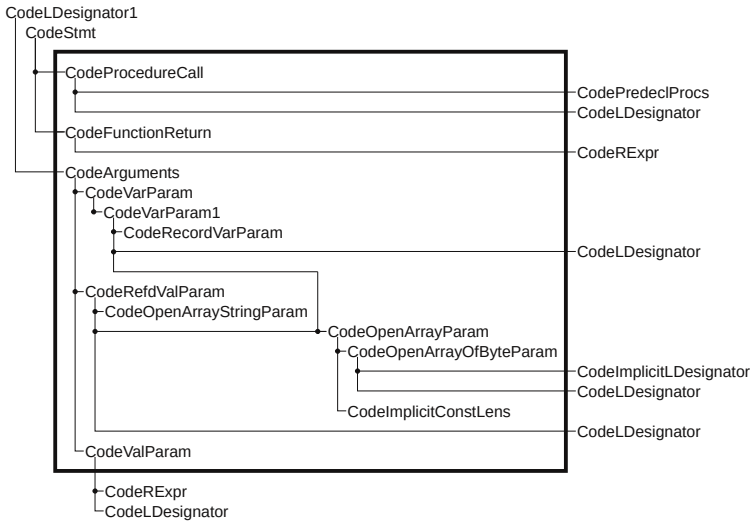

Abbildung 18: Aufrufhierarchie zur Traversierung von Prozeduraufrufen, Parametern und Funktionsresultaten

3.3.2.4.4 Zuweisungen (`.Assignments`)

Die Anwendung der vordeklarierten Prozedur `COPY` zum Kopieren einer Zeichenkette in ein statisches Array ist identisch mit einer entsprechenden Zuweisung. Ebenso hat der Einsatz von `SYSTEM.PUT` den Charakter einer Zuweisung. Daher ist die Behandlung einer Zuweisung so auf drei Puma-Prozeduren verteilt, daß auch die beiden Spezialfälle mit abgedeckt werden.

Innerhalb der Puma-Prozedur `CodeAssignment` wird somit lediglich aus dem Bezeichner des Zuweisungsziels der Adreßcode berechnet und dieser zusammen mit dem Ausdruck an `CodeAssign` übergeben.

Die im Report aufgestellte Forderung, daß bei einer Zuweisung an eine Variable v der dynamische Typ von v derselbe sein muß wie der statische Typ von v, wird durch die Prozedur `CodeImplicitAssignmentGuard` codiert, falls es sich bei v um einen indirekt referenzierten Record handelt.

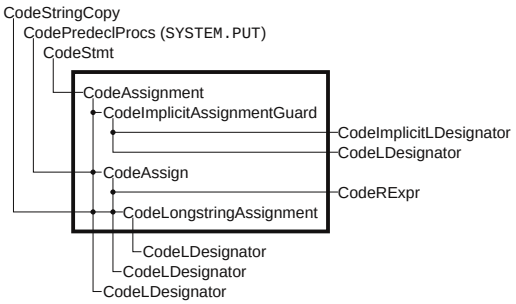

Abbildung 19: Aufrufhierarchie zur Traversierung von Zuweisungen

3.3.2.4.5 Ausdrücke (`.Exprs`)

Die Puma-Prozedur `CodeRExpr` bekommt den zu traversierenden Ausdruck, dessen Typrepräsentation sowie die Typanpassung, die auf den Wert des Ausdrucks angewendet werden muß, und liefert den daraus berechneten Datencode.

Weil es keine `imul`-Instruktion auf Byte-Operanden gibt, erfordert eine `SHORTINT`-Multiplikation entsprechende Umwandlungen.

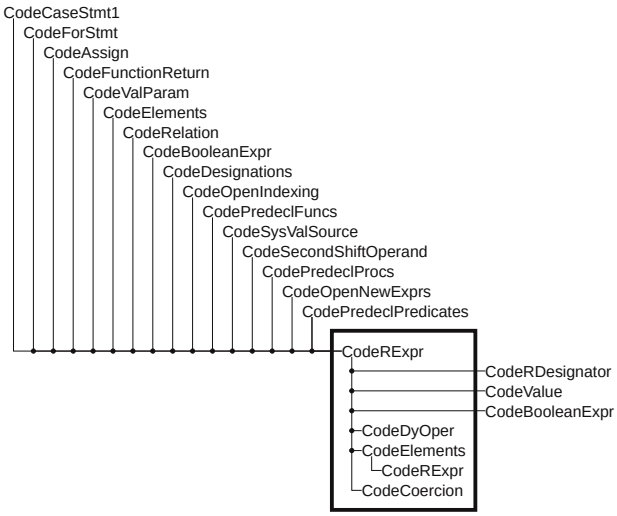

Abbildung 20: Aufrufhierarchie zur Traversierung von Ausdrücken

3.3.2.4.6 Boolesche Ausdrücke (`.BooleanExprs`)

Der Puma-Prozedur `CodeBooleanExpr` wird der Boolesche Ausdruck, sein eventuell konstanter Wert sowie das Expression-, True- und False-Label übergeben. Aus diesen Parametern wird unter Zuhilfenahme von `CodeRelation` und `CodeIsExpr` der zugehörige Code erzeugt.

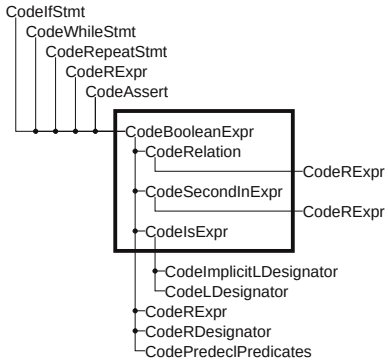

Abbildung 21: Aufrufhierarchie zur Traversierung von Booleschen Ausdrücken

3.3.2.4.7 Bezeichner (.Designators)

Innerhalb der Behandlung von Bezeichnern steht die Puma-Prozedur `CodeLDesignator` an zentraler Stelle. Sie erzeugt aus einem Bezeichner den Adreßcode, der die Adresse des bezeichneten Objekts berechnet.

Dabei wird innerhalb der Prozedur selbst lediglich der Fall behandelt, daß der Bezeichner eine Applikation von `SYSTEM.VAL` darstellt. Alle anderen Fälle werden an `CodeImplicitLDesignator` delegiert. Diese Prozedur liefert ebenfalls Adreßcode zu einem Bezeichner, jedoch zusätzlich bestimmte Werte, die für einen anschließenden Zugriff auf implizite Größen zum Bezeichner eingesetzt werden. Sie werden im folgenden implizite Werte des Bezeichners genannt. Der hierfür eingesetzte Parameter hat den folgenden Typ:

```
tImplicitDesignationData = RECORD
                             typeOfObject          : OB.tOB;
                             isStackObject         : BOOLEAN;
                             acodeToObjHeader      : ACode;
                             ofsOfObjHeader        : LONGINT;
                             ofsOfObject           : LONGINT;

                             ofsOfLEN0             : LONGINT;
                             ofsOfLastLEN          : LONGINT;
                             nofOpenIndexings      : LONGINT;
                             nofUnindexedLens      : LONGINT;
                             staticSize            : LONGINT;
                             codeToOpenIndexedElem : Cons.OpenIndexedElem;
                             codeToObjBaseReg      : DCode;
                           END;
```

Die Prozedur `CodeImplicitLDesignator` führt selbst lediglich die Zerlegung eines Bezeichners in seinen Symboltabelleneintrag und seine Selektoren (`Designations`) durch und übergibt diese Werte zusammen mit seinen impliziten Werten zur weiteren Traversierung an die Puma-Prozedur `CodeLDesignator1`.

Hier wird anhand des Symboltabelleneintrags die Unterscheidung von Konstanten, Prozeduren und Variablen vorgenommen. Für Prozeduren erfolgt die Traversierung der Parameterliste durch `CodeArguments` aus technischen Gründen über die Puma-Prozedur `CodeProcArgs`. Für Variablen erfolgt die Erzeugung des initialen Adreßcodes in Abhängigkeit des Orts der Variablen. Dieser Adreßcode wird durch die Prozedur `CodeDesignations` gemäß der Selektoren des Bezeichners entsprechend erweitert, außer es handelt sich um einen Array-Elementselektor für eine offene Dimension oder um eine Typzusicherung für einen Var-Parameter mit Record-Typ.

Da zur Behandlung dieser Selektoren der Zugriff auf die entsprechenden impliziten Größen (Typmarke oder Längen) benötigt wird und dieser Zugriff abhängig vom Ort der Variablen ist, die der Bezeichner referenziert, gibt es die Puma-Prozeduren `CodeLocalImplicitDesignations` für Variablen auf aktueller Schachtelungstiefe, die Prozedur `CodeStackImplicitDesignations` für Variablen auf anderer Schachtelungstiefe sowie `CodeHeapImplicitDesignations` für dynamisch angelegte Variablen.

Diese Prozeduren werden vorsorglich immer aufgerufen, wenn ein solcher Selektor auftreten könnte, d.h. falls die Variable ein durch *call-by-reference* übergebener Parameter ist

(`CodeLDesignator1`) oder nach einer Zeigerdereferenzierung (`CodeDesignations`). Die obigen drei Prozeduren vermerken immer die vorhandenen impliziten Größen der Variable, die durch den Bezeichner referenziert wird und führen eventuell die Codierung einer Typzusicherung oder „offenen“ Indizierung durch. Der oben erwähnte Aufruf dieser Prozeduren erfolgt deshalb vorsorglich, damit die benötigten impliziten Werte verfügbar sind.

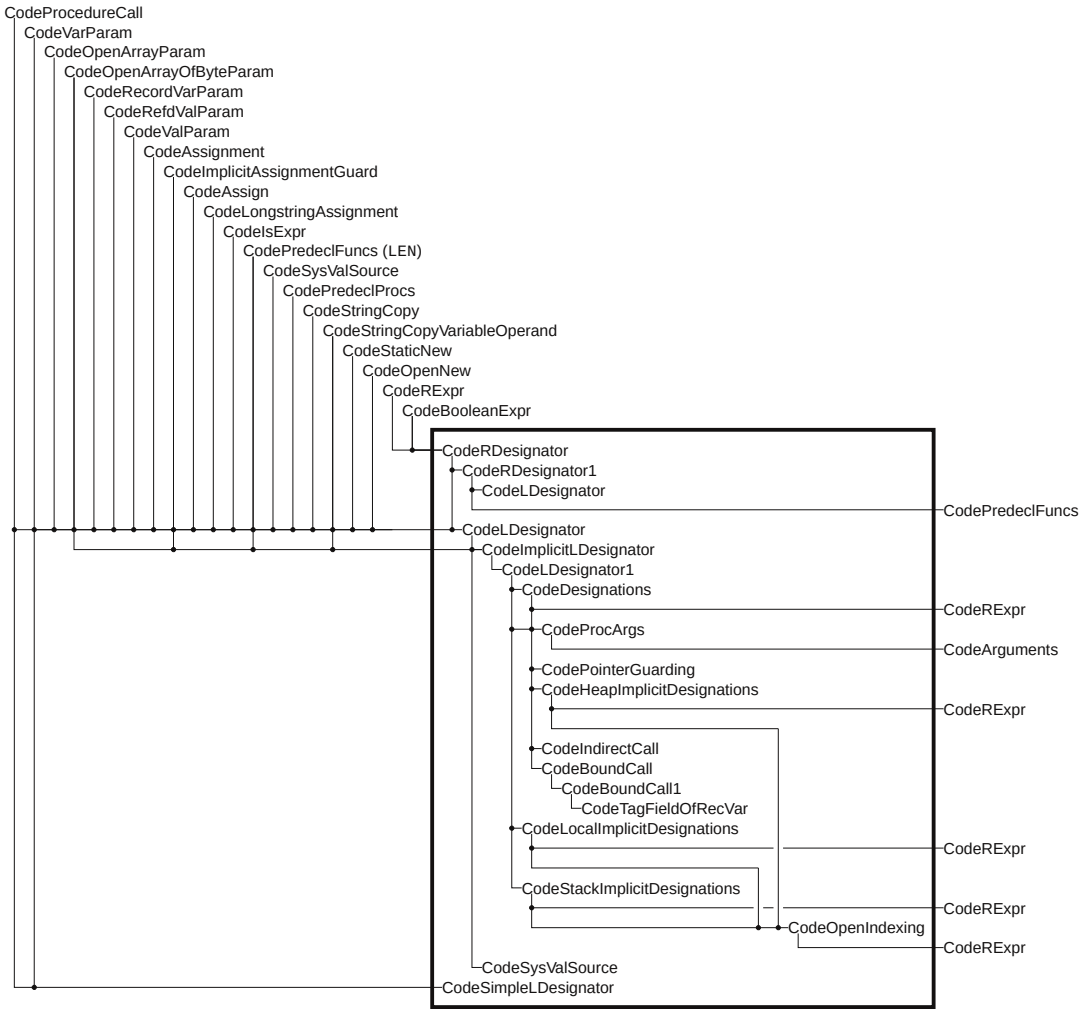

Abbildung 22: Aufrufhierarchie zur Traversierung von Bezeichnern

In Übereinstimmung mit der bereits beschriebenen Methode zur Codierung von Indizierungen offener Arrays (vgl. S. 73) findet die Behandlung des ersten Indexausdrucks in den Prozeduren selbst statt. Zur Durchführung der restlichen Berechnung dient die Prozedur `CodeOpenIndexing`.

Die Unterhierarchie ausgehend von der Prozedur `CodeBoundCall` dient lediglich der Auswahl der Werte, die zur Codierung der drei möglichen Aufrufarten gebundener Prozeduren benötigt werden.

Die Puma-Prozedur `CodeSimpleLDesignator` erzeugt den Adreßcode zum Zugriff auf eine Variable, deren Bezeichner keine Selektoren enthalten kann und die deshalb im abstrakten Syntaxbaum nicht durch `Designator` repräsentiert ist. Dies gilt beispielsweise für die Kontrollvariable der For-Schleife oder die getestete Variable innerhalb der With-Anweisung.

3.3.2.4.8 Vordeklarierte Prozeduren (`.Predecls`)

Für die vordeklarierte Prozeduren und die Prozeduren des Moduls `SYSTEM` gibt es eine Einteilung in gewöhnliche Prozeduren, Funktionsprozeduren mit Resultatstyp `BOOLEAN`. sowie sonstige Funktionsprozeduren. Boolesche Resultate müssen gesondert behandelt werden, da ihr Resultat als Kontrollfluß in den umgebenden Ausdruck oder in die umgebende Anweisung einfließt. Da `SYSTEM.VAL` auch eine Umwandlung auf den Typ `BOOLEAN` gestattet, müssen die zur Codierung von `SYSTEM.VAL` spezifizierten Prozeduren ebenfalls bei der Puma-Prozedur für Prädikate verwendet werden.

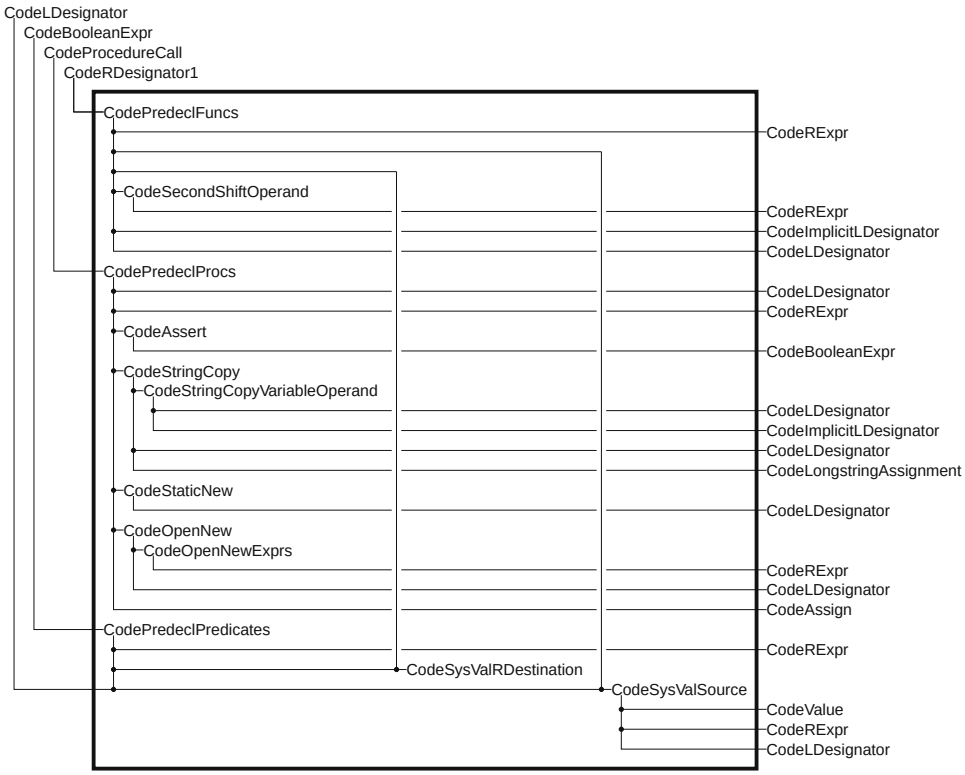

Abbildung 23: Aufrufhierarchie zur Traversierung von vordeklarierten Prozeduren

3.4 Umgebende Zusatzmodule

3.4.1 Oberon-2-Restriktionen (LIM)

Die Konstanten-Definitionen im Modul `LIM` definieren die Restriktionen, die an den verschiedenen Stellen innerhalb der Implementierung gelten. Sie definieren insbesondere die maximale Schachtelungstiefe von Prozeduren (30) und die maximale Erweiterungsstufe von Record-Typen (8).

3.4.2 Hauptmodul und Treiber (Jacob, ARG, DRV)

Das Hauptmodul `Jacob` (Just A Compiler for OBeron-2) ruft die entsprechenden Prozeduren aus `DRV` in Abhängigkeit der Kommandozeilenparameter (s.u.) auf, deren Analyse im Modul `ARG` (command line ARGuments) beschrieben ist. Alle aus dieser Analyse resultierenden Werte werden von `ARG` über exportierte Variablen zur Verfügung gestellt. Das Modul `DRV` (compiler DRiVer) stellt die eigentliche Funktionalität des implementierten Compilers zur Verfügung, die darin besteht, daß ausgehend von einem Hauptmodul jedes durch Importierung „erreichbare“ Modul gelesen und analysiert wird. Bei einem *Make*-Lauf wird jedes dieser Module übersetzt und assembliert, falls entweder keine Objektdatei existiert oder sie älter ist als irgendeine Quelldatei, von der das Modul abhängt (einschließlich seiner eigenen). Zusätzlich wird der Linker zur Erzeugung des Programmdatei aufgerufen, falls entweder keine Programmdatei existiert oder sie älter ist als irgendeine Quelldatei, von der das Hauptmodul abhängt. Der *Make*-Lauf kann durch Kommandozeilenparameter eingeschränkt werden.

Kommandozeilenparameter

Der Compiler kann gemäß folgendem Schema aufgerufen und mit Parametern versehen werden:

`Jacob` { `-`*Kommando* | `-`*Option* | *datei*[`.ob2`] }

Die Angabe der zu übersetzenden Quelldateien erfolgt durch Nennung ihrer Namen, wobei das Suffixes `.ob2` fehlen darf. Alle im folgenden beschriebenen Kommandos und Optionen gelten für jede angegebene Datei, welche zusätzlich als Hauptmodul angesehen wird, falls es sich um einen *Make*-Lauf handelt.

Kommandos steuern die Arbeitsweise des Compilers. Es wird üblicherweise höchstens eines der nachfolgenden Kommandos (auf beliebiger Position) angegeben. Bei mehreren Kommandos gilt das angegebene aufgeführte.

`v`	Ausgabe der Versionsnummer und des Versionsdatums des Compilers
`h`	Ausgabe von Information zur Benutzung
`cg`	Codeerzeugung wird unterdrückt
`S`	Assemblierung wird unterdrückt
`c`	Einfache Übersetzung der angegebenen Module
`m`	Übersetzung aller Module ohne Linkeraufruf

Ist kein Kommando explizit angegeben, wird ein vollständiger *Make*-Lauf einschließlich Linkeraufruf für jede angegebene Datei durchgeführt.

Optionen beeinflussen die Art der Übersetzung und dienen zur Angabe zusätzlicher Informationen, die für den Compiler notwendig sind.

`it`	Ignorieren der Zeitmarken der relevanten Dateien
`dp`	Abschalten der Prozedurzählerausgabe
`ic`	Abschalten der Erzeugung von Indexüberprüfungen
`rc`	Abschalten der Erzeugung von Bereichsüberprüfungen
`nc`	Abschalten der Erzeugung von `NIL`-Überprüfungen
`ac`	Abschalten der Codeerzeugung für `ASSERT`
`op`	Abschalten der Optimierung bei Val-Parametern
`tab` *dir*	Die vom Compiler benötigten benötigten Tabellen (`Scanner.Tab`, `Parser.Tab` und `Errors.Tab`) werden im Verzeichnis *dir* gesucht
`I`*dir*	Zu importierende Module sollen zusätzlich im Verzeichnis *dir* gesucht werden (Angabe mehrfach möglich)
`asm` *cmd*	Angabe des Shell-Skripts für den Assembleraufruf
`link` *cmd*	Angabe des Shell-Skripts für den Linkeraufruf

Da für die korrekte Arbeitsweise des Compilers die Angaben zu `tab`, `asm` und `link` immer notwendig sind, bietet es sich an, den eigentlichen Aufruf des Compilers in einem Shell-Skript zu kapseln:

```
#!/bin/sh
JACOBROOT=/usr/jacob
$JACOBROOT/sys/Jacob \
   -tab  $JACOBROOT/sys \
   -link $JACOBROOT/sys/oc.linker \
   -asm  $JACOBROOT/sys/oc.assembler \
   -I$JACOBROOT/lib \
   $*
```

Zur einfacheren Behebung von Fehlern im Compiler, zu Testzwecken oder falls die Arbeitsweise des Compilers detaillierter nachvollzogen werden soll, können die nachfolgend aufgeführten Kommandos und Optionen eingesetzt werden.

Kommandos

`t`	Ausgabe der textuellen Repräsentationen der gelieferten Symbole während der Scanner-Phase
`pt`	Ausgabe des Syntaxbaums nach der Parsierung
`dt`	Ausgabe des dekorierten Syntaxbaums nach der Attributauswertung
`read`	Abbruch nachdem die angegebenen Dateien eingelesen wurden
`scan`	Abbruch nach der Scanner-Phase
`parse`	Abbruch nach der Parsierung

Optionen

`kt`	Erzeugte Assemblerdateien (`*.s`) werden nicht gelöscht
`cmt`	Assemblerdateien werden mit ausführlichen Kommentaren erzeugt
`Ts`	Ausgabe der aktuell durchgeführten Compiler-Phase

`Dp`	Ausgabe der Symboltabelle, welche die vordeklarierten Objekte enthält
`Di`	Ausgabe der importierten Symboltabellen
`Ds`	Ausgabe der (lokalen) Symboltabellen
`Dw`	Ausgabe der (lokalen) Symboltabellen, welche für Anweisungsfolgen innerhalb von With-Anweisungen gelten
`DA`	Angabe der Adressen von Typrepräsentationen bei Symboltabellenausgabe
`BL`	Angabe der Blocklisten bei der Symboltabellenausgabe
`PN`	Angabe der Numerierung gebundener Prozeduren bei Symboltabellenausgabe
`DS`	Angabe der Typgrößen bei Symboltabellenausgabe
`DV`	Angabe der (relativen) Adressen von Variablen bei Symboltabellenausgabe
`Ss`	Ausgabe von statistischen Informationen zur Übersetzung
`E`	Ausgabe der Prozedurnamen und -parameter bei Aufrufen von Prozeduren aus dem Modul `ED`
`ee`	Fehlermeldungen werden bei ihrer Erzeugung sofort auf der Konsole ausgegeben
`ctei`	Ausgabe der für den Codeerzeuger aufgebauten Ausdrucksbäume
`ctem`	Ausgabe der vom Codeerzeuger zur Überdeckung ausgewählten Regeln
`ctra`	Ausgabe der bei der Registerallokation relevanten Register

3.4.3 Labelverwaltung (`LAB`)

Im Modul `LAB` wird die Funktionalität der Labelvergabe gekapselt. Es stehen Prozeduren zur Verfügung, mittels derer alle eingesetzten Formen von Labels aufgebaut werden können. Darüber hinaus sind die konstanten Labels bereits definiert.

Insbesondere liefert die Funktionsprozedur `LAB.New` ein neues lokales Label sowohl als Resultat als auch zusätzlich über ihren Var-Parameter. Dadurch kann die Label-Erzeugung beim Absetzen von Assembleranweisungen elegant erfolgen:

```
ASM.C1 ( jmp , LAB.New(label) );
...
ASM.Label( label );
...
```

3.5 Funktionalität zur Laufzeit

Die Implementierung der zur Laufzeit benötigten Funktionalität ist aus konzeptionellen Gründen auf mehrere Module verteilt. Die Basisfunktionalität befindet sich im Modul `OB2RTS` (OBeron-2 Run-Time System). Die automatische Freispeicherverwaltung ist in einem eigenen Modul `Storage` implementiert, wodurch diese Funktionalität leichter ausgetauscht werden kann. Zu Testzwecken während der Entwicklung stützt sich `Storage` auf Formatierungs- und Ausgaberoutinen des Moduls `UTIS`.

3.5.1 Laufzeitsystem (`OB2RTS.as`)

Das Laufzeitsystem dient der Einbettung eines Oberon-2-Programms in das Betriebssystem. Es wird nicht verwendet, um Funktionalität des Betriebssystems in Oberon-2 verfügbar zu machen, sondern lediglich dazu, einen „ordnungsgemäßen“ Programmstart sowie eine einheitliche Programmtermination zu realisieren. Es stellt ferner spezielle Laufzeitkonstanten zur Verfügung, damit diese in einem Programm nur einmal codiert sind. Da der spezifizierte Compiler Assemblercode erzeugt und das Laufzeitsystem den konstanten Anteil eines Programms darstellt, ist es in Assembler implementiert.

3.5.1.1 Initialisierung

Innerhalb des Startcodes in der Objektdatei `crt0.o` wird davon ausgegangen, daß eine aufzurufende Funktion gemäß der Deklaration des folgenden C-Prototyps exisiert:

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

Dabei wird die Anzahl der Kommandozeilenparameter (`argc`), die Referenz auf diese (`argv`) sowie die Referenz auf die Umgebungsvariablen (`env`) übergeben. Aus diesem Grund wird die Programminitialisierung im Laufzeitsystem durch eine Routine mit dem globalen Label `_main` vorgenommen.

Diese Routine kopiert die Werte der übergebenen Parameter in die globalen Variablen `_argc`, `_argv` und `_env`, so daß diese Werte über die Schnittstellenbeschreibung eines Fremdmoduls in Oberon-2 verwendet werden können. Anschließend erfolgt die Initialisierung des mathematischen Koprozessors, indem lediglich dessen Rundungssteuerung auf „Aufrunden in Richtung negativer Unendlichkeiten“ (vgl. NELSON [1991], S. 64) gesetzt wird, so daß die Codierung der vordeklarierten Funktionsprozedur `ENTIER` die geforderte Semantik erfüllt. Zur Initialisierung der automatischen Freispeicherverwaltung erfolgt der „Aufruf“ des entsprechenden Modulrumpfs (`_Storage$I`). Durch die Aktivierung des Wurzelmodulrumpfs wird das eigentliche Oberon-2-Programm gestartet. Bei *impliziter* Termination des Programms wird durch den anschließenden Aufruf von `HALT(0)` die Finalisierung eingeleitet.

3.5.1.2 Finalisierung

Die Finalisierung eines Programms besteht darin, die Kontrolle an das Betriebssystem über die von der C-Bibliothek zur Verfügung gestellte Funktion

```
void exit(int status);
```

wieder abzugeben. Dabei wird der Funktion über den Parameter ein ganzzahliger *Terminationscode* übergeben. Dieser Wert steht dann für das Betriebssystem als sogenannter *exit-Status* zur Verfügung (z.B. zur Abfrage über `$?` in einem Shell-Skript). Die lokale Routine `StdExit` des Laufzeitsystems führt den Aufruf der Funktion `exit` mit dem Wert der lokalen Variablen `ExitCode` als Terminationscode durch.

Eine implizite Termination, eine explizite Termination durch die vordeklarierte Prozedur `HALT` und eine Termination als Reaktion auf einen Laufzeitfehler führen dazu, daß der Kontrollfluß bei der Routine mit dem Label `_Halt` fortgesetzt wird. Diese Routine speichert den über ihren Parameter erhaltenen Terminationscode in `ExitCode` und ruft über die Variable `_ExitProc` normalerweise die Routine `StdExit` auf.

Auf die Variable `_ExitProc` kann aber auch in Oberon-2 über eine Fremdmodulschnittstelle zugegriffen werden, so daß eine Prozedur eines Moduls als sogenannte *Exit-Prozedur* vor der Programmtermination noch die Möglichkeit bekommt, bestimmte Aufgaben durchzuführen, wie z.B. das Sichern von Dateipuffern und Schließen von Dateien. Von einem Modul wird eine Exit-Prozedur üblicherweise im Modulrumpf installiert, indem der alte Wert von `_ExitProc` gesichert und durch die neue Prozedur(-adresse) ersetzt wird. Die vom alten Wert referenzierte Prozedur sollte am Ende der Exit-Prozedur aufgerufen werden, damit eine Kette von Exit-Prozeduren nicht unterbrochen wird. Der Aufruf der Routine `StdExit` ist in jedem Fall garantiert, da `_Halt` nach dem Aufruf der Exit-Prozeduren immer zu `StdExit` verzweigt.

3.5.1.3 Laufzeitfehlermeldungen

Laufzeitfehler werden wie folgt behandelt: Die Erkennung eines Zustands, der zur Laufzeit als fehlerhaft gilt, erfolgt durch den vom Codeerzeuger abgesetzten Code. Als Reaktion auf einen Fehlerzustand wird in Abhängigkeit der Fehlerart zu einer entsprechenden Routine im Laufzeitsystem verzweigt. Diese Routinen geben jeweils eine Fehlermeldung sowie die Aufrufhierarchie an der Fehlerstelle aus und führen anschließend die Programmtermination über `HALT(1)` herbei. Die folgenden Laufzeitfehlerarten werden unterschieden:

- Der Rumpf einer Funktionsprozedur wurde vollständig ausgeführt, ohne daß eine Return-Anweisung zu ihrer Termination führte.
- Der Wert eines Case-Ausdrucks kommt in keiner Case-Marke einer Case-Anweisung ohne `ELSE`-Zweig vor.
- Alle Typtests einer With-Anweisung ohne `ELSE`-Zweig schlagen fehl.
- Eine Typzusicherung schlägt fehl. (Die vollqualifizierten Namen (vgl. S. 16) der beteiligten Typen werden ausgegeben.)
- Der Wert eines Indexausdrucks liegt außerhalb der Array-Grenzen.

- Bei einer Zeigerdereferenzierung, einem Typtest oder einer Typzusicherung hat ein Zeiger den Wert `NIL`[1]
- Ein ganzzahliger Wert liegt außerhalb des zulässigen Bereichs einer bestimmten Operation. Die folgende Tabelle definiert hierbei die einzelnen Fehlerzustände, wobei x und y den ganzzahligen Typ T besitzen, v ein offenes Array bezeichnet und s eine Variable vom Typ `SET` ist.

Operation	*Bedingung*
`ABS(`x`)`	$x =$ `MIN(`T`)`
`CHR(`x`)`	$x \notin \{0, \ldots,$ `ORD(MAX(CHAR))`$\}$
`SHORT(`x`)`	`LONG(SHORT(`x`))`$\neq x$
`NEW(`v`,...,`x`,...)`	$x <= 0$
y `DIV` x, y `MOD` x	$x <= 0$
`INCL(`s`,`x`)`, `EXCL(`s`,`x`)`, `{...,`x`,...}`	$x \notin \{0, \ldots,$ `MAX(SET)`$\}$

- Eine Prozedurvariable wird mit dem Wert `NIL` aufgerufen.
- Die vordeklarierte Prozedur `ASSERT` „schlägt fehl“. (Der Terminationscode wird ausgegeben. Falls der optionale Parameter vorhanden ist, bestimmt dieser den Terminationscode.)

Der „Test“ von Prozedurvariablen auf den Wert `NIL` ist dadurch realisiert, daß dieser Wert durch die Adresse der Routine `_NILPROC` repräsentiert wird. Demgegenüber müßte bei Verwendung des Werts 0 vor jedem Aufruf über eine Prozedurvariable ein expliziter Test durchgeführt werden, ohne den in diesem Fall ein kompletter Neustart des Programms erfolgen würde.

Die Ausgabe der Aufrufhierarchie läßt sich wegen der Strukturen, die für die automatische Freispeicherverwaltung angelegt werden, gemäß folgendem Schema einfach realisieren (vgl. S. 19):

```
frame ← %ebp
while MEM(frame - 4) ≠ _FirstModuleTDesc do
        WriteStringAtAdress(MEM(frame - 4) - 12)
        frame ← MEM(frame)
end
```

3.5.1.4 Laufzeitkonstanten

Neben der bereits erwähnten konstanten Tabelle (auf der Adresse `_BitRangeTab`) zur effizienten Codierung von Mengenkonstruktoren (vgl. S. 79) wird vom Laufzeitsystem eine Konstante über das globale Label `_NullChar` zur Verfügung gestellt. Diese Konstante besteht lediglich aus einem Nullbyte und wird benutzt, wenn eine Referenz auf eine leere Zeichenkette benötigt wird.

[1] Die Deaktivierung dieser Laufzeitfehlerüberprüfung durch die Kommandozeilenoption `nc` führt bei Anwendung eines `NIL`-Zeigers zu einem Abbruch des Programms mit der Meldung „Segmentation fault“. Insbesondere wird die Finalisierung nicht durchgeführt.

3.5.2 Freispeicherverwaltung (`Storage.c`)

Die Aufgabe der Freispeicherverwaltung ist es, Anforderungen zur Anlage von Datenblöcken im Heap über die vordeklarierte Prozedur `NEW` (oder `SYSTEM.NEW`) zu erfüllen. Da es in Oberon-2 keine Möglichkeit gibt, einen angelegten Block explizit wieder freizugeben, wird zudem eine automatische Speicherbereinigung (garbage collection) eingesetzt, welche alle Datenblöcke wieder verfügbar macht, die nicht mehr benutzt werden können.

Die hierzu notwendigen Algorithmen besitzen im allgemeinen einen nicht unerheblichen Laufzeitaufwand, so daß sie sehr effizient implementiert sein sollten. Dies motiviert die direkte Implementierung in Assembler, zumal es sich hierbei um die Handhabung von Datenblöcken beliebigen Typs handelt. Andererseits werden während der Entwicklung einer Freispeicherverwaltung diverse Ausgabe- und Testroutinen benötigt, deren Formulierung in Assembler zu umständlich wäre. Daher wurde zur Implementierung die Sprache C herangezogen, da der zur Verfügung stehende Compiler `gcc` auch gestattet, als Anweisungen Assemblerinstruktionen angeben zu können. Die beiden innerhalb der Speicherbereinigung verwendeten Funktionen sind vollständig in Assembler implementiert. Der durch die Verwendung des `gcc` ebenfalls zur Verfügung stehende C-Präprozessor `cpp` erlaubt zudem die Definition von Makros, über die der Zugriff auf die zur Verwaltung des Heaps notwendigen Elemente elegant formuliert werden kann.

3.5.2.1 Initialisierung

Die Einrichtung des initialen Heaps (Größe: 2 MByte) erfolgt in der Funktion `Storage$I`, die sich hierzu auf den Systemaufruf `sbrk` zum Lesen und Setzen der Systemvariablen `brk` stützt (vgl. S. 10). Falls dies fehlschlägt, wird eine entsprechende Meldung ausgegeben, und es erfolgt ein Programmabbruch mit dem Terminationscode 1.

3.5.2.2 Speicheranforderungen

Für die Codierung von Speicheranforderungen durch den Codeerzeuger stellt das Modul `Storage` drei Funktionen zur Verfügung. Die Verwendung des Zeichens '`$`' in den Funktionsnamen verhindert, daß die Funktionen über eine Fremdmodulschnittstelle in Oberon-2 aufgerufen werden können, und muß über eine entsprechende Kommandozeilenoption des C-Compilers erlaubt werden.

Die Codierung des Aufrufs der vordeklarierten Prozedur `NEW` mit einem Zeiger auf einen Record oder ein Array fester Länge erfolgt in Form eines Aufrufs der folgenden Funktion:

```
void $staticNEW(unsigned *var
                ,unsigned size
                ,unsigned tdescAddr
                ,unsigned initAddr);
```

Dieser Funktion muß die Adresse der Zeigervariablen, die Größe des Zeigerbasistyps, die Adresse des zugehörigen Typdeskriptors sowie die Adresse der zugehörigen Initialisierungsroutine übergeben werden (vgl. S. 17).

Für einen Aufruf der vordeklarierten Prozedur `NEW` mit einem Zeiger auf ein offenes Array wird eine andere Funktion verwendet.

```
void $openNEW(unsigned *var
              ,unsigned elemSize
              ,unsigned tdescAddr
              ,unsigned initAddr
              ,unsigned nofLens
              ,...);
```

Ihr wird als Größe die Anzahl der Bytes übergeben, die der statische Elementtyp belegt. Zusätzlich erhält sie die Anzahl der übergebenen Längen sowie die Längen selbst.

Die für die Prozedur `NEW` aus dem Modul `SYSTEM` eingesetzte Funktion benötigt lediglich die Variablenadresse und die Anzahl der angeforderten Bytes.

```
void SYSTEM$NEW(unsigned *var, unsigned size);
```

Diese drei Funktionen berechnen jeweils die für die Anforderung benötigte Blockgröße unter Berücksichtigung der Granularität des Heaps (8 Byte). Mit diesem Wert wird die lokale Funktion `AllocAligned` aufgerufen, welche als Resultat die Adresse des angelegten Blocks liefert. Die Initialisierung der entsprechenden Verwaltungsfelder in diesem Block schließt dann die Oberon-2-Speicheranforderung ab.

In der Funktion `AllocAligned` wird gemäß der First-fit-Strategie ein passender Block (eventuell unter Aufteilung eines freien Blocks) gesucht. Gelingt dies nicht, wird die lokale Funktion `AllocFail` aufgerufen, um Abhilfe zu schaffen. Da hierbei die Speicherbereinigung sowie die Vergrößerung des Heaps einzeln oder in Kombination erfolgreich sein können, wird die Suche solange wiederholt, bis ein passender Block gefunden wurde.

Um einem Anwendungsprogramm größere Flexibilität zu erlauben, verzweigt die Funktion `AllocFail` über eine Zeigervariable `AllocFailHandlerProc` zu einer entsprechenden Routine. Die Zeigervariable ist mit der Adresse der Funktion `DefaultAllocFailHandler` initialisiert, welche lediglich bei ihrem ersten Aufruf die Speicherbereinigung anstößt und bei allen weiteren Aufrufen den Heap um seinen initialen Wert vergrößert. Ist selbst dies nicht (mehr) durchführbar, wird ein Programmabbruch wie `HALT(1)` durchgeführt.

3.5.2.3 Speicherbereinigung

Die Speicherbereinigung erfolgt nach der Mark-and-sweep-Methode. Bei dieser werden zwei grundsätzliche Phasen unterschieden. In der Mark-Phase erfolgt die Markierung der Blöcke, die von aktiven Zeigern referenziert werden. Dabei wird einmal die sogenannte *Wurzelmenge* (root set) bestimmt, welche genau die globalen Zeiger und die lokalen Zeiger aller Prozedurinkarnationen enthält. Zum anderen muß ausgehend von jedem Element der Wurzelmenge eine Traversierung aller erreichbaren Blöcke durchgeführt werden. In der Sweep-Phase werden dann alle nicht markierten Blöcke für nachfolgende Speicheranforderungen wieder verfügbar gemacht. Die beiden Phasen sind in den lokalen Funktionen `Mark` und `Sweep` implementiert und werden in der globalen Funktion `GC` aufgerufen.

3.5.2.3.1 Mark-Phase

Zur einfachen Ermittlung der Wurzelmenge sind alle Aktivierungssegmente und die BSS-Abschnitte aller Module auf folgende Weise miteinander verbunden:

Ausgehend vom aktuellen Wert des Registers `ebp` ist die Verkettung aller Aktivierungssegmente bereits durch die Sicherung dieses Registers im Prolog jeder Prozedur gegeben. Im Aktivierungssegment des Hauptmodulrumpfs gibt es einen Verweis auf den BSS-Abschnitt des ersten Moduls der Modulfolge, von dem aus die Verkettung aller BSS-Abschnitte bis zum Aktivierungssegment des Wurzelmodulrumpfs fortgesetzt ist. Dieses besitzt statt eines Verweises in `ebpsav` den Wert 0. Damit sind sowohl die Aktivierungssegmente als auch alle BSS-Abschnitte in einer Liste verkettet.

Damit der Ausgangspunkt der Verkettung für die Mark-Phase verfügbar ist, wird in allen globalen Funktionen des Moduls `Storage` der aktuelle Wert des Registers `ebp` in der globalen Variable `_root_ebp` gesichert.

Abbildung 24 zeigt die Verkettung der Aktivierungssegmente und BSS-Abschnitte für die folgenden Oberon-2-Module zum Zeitpunkt des Aufrufs von `Storage.GC`.

```
MODULE O;
   VAR Og:LONGINT;
END O.

MODULE N;
   VAR Ng:LONGINT;
END N.

MODULE M;
   IMPORT N,O,Storage;
   VAR Mg:LONGINT;
   PROCEDURE S*; BEGIN Storage.GC; END S;
   PROCEDURE R*; BEGIN S;          END R;
   PROCEDURE Q*; BEGIN R;          END Q;
   PROCEDURE P*; BEGIN Q;          END P;
BEGIN                  P;          END M.
```

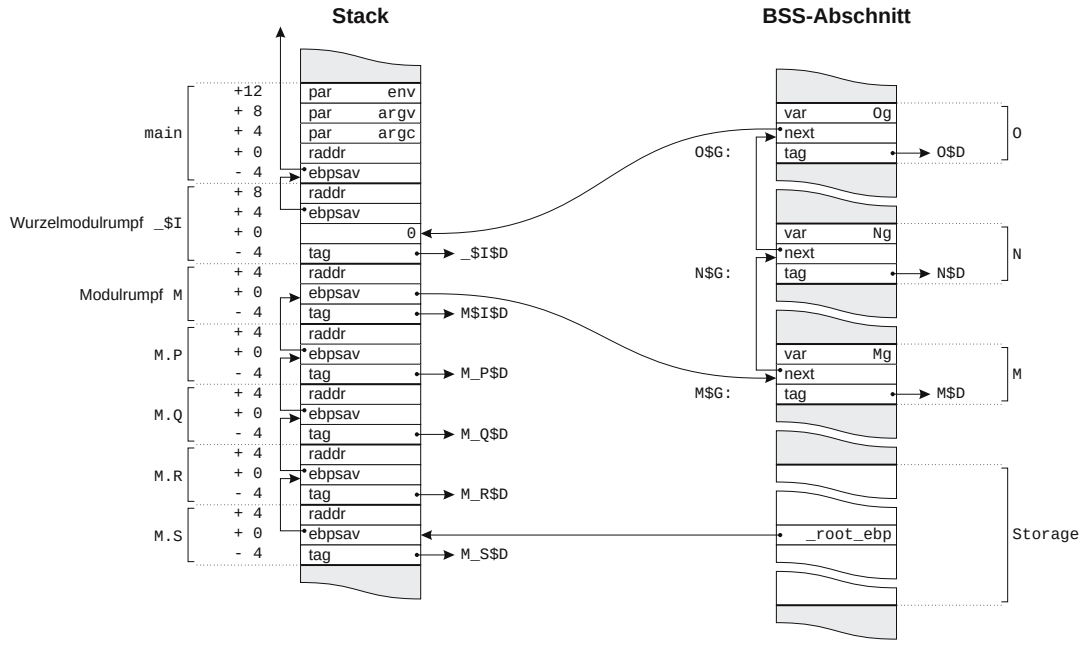

Abbildung 24: Beispielhafte Verkettung zu einer Wurzelmenge

Alle Aktivierungssegmente sowie die BSS-Abschnitte aller Module besitzen jeweils eine „Typ"-Marke (tag), welche auf den zugehörigen Deskriptor verweist. Über die Typmarke erfolgt nun in der Mark-Phase die Traversierung innerhalb des folgenden Rahmens:

```
last ← NIL
act  ← _root_ebp
repeat
    ...
    act ← MEM(act)
until act = 0
```

Somit liegt als Induktionsvoraussetzung die Situation vor, daß der Zeiger *act* den zu besuchenden Speicherblock referenziert und der Zeiger *last* auf den Speicherblock verweist, zu dem nach dem Besuch zurückgekehrt werden muß. Zusätzlich enthält `MEM(`*act*`-4)` die Adresse des zugehörigen Deskriptors und damit eine Referenz auf dessen Offset-Tabelle.

Der Induktionsschritt besteht nun darin, die *Verfolgung* aller Zeiger eines Speicherblocks über eine Parsierung seiner Offset-Tabelle gemäß deren „Grammatik" (vgl. S. 16) durchzuführen. Dies geschieht in Anlehnung an die von Pfister, Heeb und Templ [1991] beschriebene Methode der automatischen Freispeicherverwaltung des Oberon-Systems.

Die Verfolgung eines Zeigers bedeutet, daß der von ihm referenzierte Speicherblock für den Fall, daß er noch nicht markiert ist, markiert und anschließend besucht wird. Zur Minimierung der Belastung des Stacks durch lokale Variablen wird hierbei der „Rückweg" in den aktuell verfolgten Zeigern gespeichert. Die Speicherung der Offsets der aktuell verfolgten Zeiger erfolgt zudem in der Typmarke des jeweiligen Speicherblocks. Folgendes Schema zeigt den verwendeten Algorithmus:

```
MarkBlock: desc ← MEM(act-4)
           ofs  ← MEM(desc)
           while ofs ≠ -1 do
               new ← MEM(act+ofs)
               if (new ≠ NIL) ∧ ISNOTMARKED(new-4) then
                   MEM(act+ofs) ← last
                   MEM(act-4)   ← desc
                   last         ← act
                   act          ← new
                   call MarkBlock
                   new          ← act
                   act          ← last
                   desc         ← MEM(act-4)
                   ofs          ← MEM(desc)
                   last         ← MEM(act+ofs)
                   MEM(act+ofs) ← new
               end
               desc ← desc+4
               ofs  ← MEM(desc)
           end
```

Bei jedem Speicherblock, der im Heap liegt, dient das niederwertigste Bit der Typmarke zu seiner Markierung. Aus diesem Grund, und da ein solcher Speicherblock zuerst markiert und dann besucht wird, beginnt seine zugehörige Offset-Tabelle mit einem Byte, so daß der Zugriff über die Typmarke den ersten zu betrachtenden Offset liefert.

Das Prädikat ISNOTMARKED beinhaltet den Test und das Setzen der Markierung eines Speicherblocks. Es kann kompakt durch zwei Maschineninstruktionen codiert werden (mit `eax`=*new*):

```
btsb   $0,-4(%eax)
jc     BlockNotMarked
```

Der rekursive Aufruf ist notwendig, da zur Handhabung von offenen Arrays, insbesondere von offenen Array-Parametern, eine andere Routine benutzt wird, so daß bei einer Rückkehr zum zuletzt besuchten Speicherblock die Ermittlung des nächsten zu verfolgenden Zeigers nicht eindeutig ist.

3.5.2.3.2 Sweep-Phase

In dieser Phase wird der Heap durchlaufen und aus allen unmarkierten Blöcken wird eine neue Liste von freien Blöcke aufgebaut. Dabei werden alle Markierungen wieder zurückgenommen. Um unabhängig von der Art des aktuellen Blocks den Blockanfang des unmittelbar folgenden Blocks zu berechnen, besitzt jeder Deskriptor eine Skipper-Routine.

3.6 Spezielle Problembereiche

3.6.1 Initialisierung von Zeigern und Prozedurvariablen

Damit die korrekte Arbeitsweise der Freispeicherverwaltung gewährleistet ist, müssen alle Zeigervariablen mit dem Wert `NIL` initialisiert werden. Zudem kann der Aufruf einer uninitialisierten Prozedurvariablen zu unvorhersehbarem Verhalten führen, so daß ebenso die Initialisierung aller Prozedurvariablen mit dem Wert `_NILPROC` erfolgt.

Da lediglich die Position dieser Variablen innerhalb eines Speicherblocks für ihre Initialisierung von Interesse ist, gibt es sogenannte *Blocklisten*, über die gerade diese Positionen kompakt verwaltet werden. Die weiteren Ausführungen beschränken sich auf Blocklisten für Zeiger und gelten sinngemäß für Prozedurvariablen. Die Funktionen zur Handhabung solcher Blocklisten arbeiten ausschließlich auf Typrepräsentationen und Symboltabelleneinträgen und sind daher als Puma-Prozeduren implementiert.

Eine Blockliste B ist entweder leer ($\perp$) oder ein 5-Tupel

$$B = (previous, sublist, offset, count, increment),$$

wobei *previous* und *sublist* jeweils Blocklisten sind und *offset*, *count* sowie *increment* ganzzahlige Werte annehmen können. Eine nichtleere Blockliste definiert ab Position *offset* im zugehörigen Speicherblock eine Folge von *count* Positionen; der Abstand einer Position zur nächsten beträgt immer *increment* Bytes. Ist die Unterliste (*sublist*) leer, bezeichnen die Positionen direkt die Zeiger des Speicherblocks. Andernfalls bezeichnen die Positionen eine Folge von Unterspeicherblöcken, deren Zeigerpositionen durch *sublist* bezeichnet werden.

Handelt es sich bei dem Speicherblock um ein (mehrdimensionales) offenes Array mit Elementtyp T, enthält *count* die Anzahl der offenen Dimensionen des Arrays mit negativem Vorzeichen, *increment* die Größe von T und *sublist* die Blockliste von T. Die restlichen Bestandteile des Tupels sind leer.

Die Grundlage der Ermittlung von Zeigerpositionen ist die Funktion `PtrBlocklistOfType`, welche einen Typ auf eine Blockliste abbildet. Diese (rekursive) Funktion liefert in ihrer Verankerung für einen Zeigertyp $(\perp, \perp, 0, 1, 4)$ und für alle anderen nichtstrukturierten Typen $\perp$. Für Array- und Record-Typen muß eine Konstruktion des Resultats aus den Blocklisten der Elementtypen durchgeführt werden. Hierbei werden naheliegende Zusammenfassungen durchgeführt, so daß eine möglichst „kompakte“ Blockliste konstruiert wird.

Beispielsweise wird zum Typ `ARRAY 5 OF POINTER TO ...` die Blockliste $(\perp, \perp, 0, 5, 4)$ konstruiert; für einen Record-Typ mit zwei Elementen, deren Blocklisten $(p, s_1, 20, 3, 8)$ und $(\perp, s_2, 44, 2, 8)$ lauten, wird die Blockliste $(p, s_1, 20, 5, 8)$ geliefert, falls s_1 strukturell äquivalent mit s_2 ist.

Entsprechend zu der Berechnung der Blockliste eines Records erfolgt die Berechnung der Blockliste aus der Deklarationsliste für globale Variablen eines Moduls oder lokale Variablen einer Prozedur über jeweils eine eigene Puma-Prozedur.

Blocklisten werden auch im Rahmen der Erzeugung von Offset-Tabellen für Deskriptoren verwendet. Hierbei sind lediglich Zeiger-Blocklisten von Relevanz. Da eine Offset-Tabelle

für die Speicherbereinigung auch die Parameter einer Prozedur enthalten muß, gibt es eine eigene Puma-Funktion für die Berechnung der dafür notwendigen Blockliste.

Ausgehend von einer Blockliste kann somit sehr effizient kompakter Code zur Initialisierung abgesetzt werden.

3.6.2 Numerierung gebundener Prozeduren

Die Nummer einer gebundenen Prozedur P definiert eindeutig die Position der Prozeduradresse in der Prozedurtabelle des Deskriptors zu dem Typ, an den P gebunden ist. Hierbei muß insbesondere sichergestellt sein, daß eine Redefinition P' die gleiche Nummer hat, wie die redefinierte Prozedur P. Da es gestattet ist, P' textuell vor P zu deklarieren, kann die Nummernvergabe erst erfolgen, nachdem alle Deklarationen eines Moduls bearbeitet worden sind.

Die Berechnung der Prozedurnummern ist im Modul `T` ausgehend von der Puma-Prozedur `CalcProcNumsOfEntries` spezifiziert. Hierbei wird zu jedem deklarierten Record-Typ, an den Prozeduren gebunden werden können, die Anzahl der Prozeduren gezählt, die er entweder von seinem Basistyp „erbt“ oder redefiniert. Die nächste für die Numerierung zur Verfügung stehende Zahl ergibt sich dann aus dieser Summe, so daß jede an den Record-Typ explizit gebundene Prozedur P eine neue Nummer bekommen kann, falls P keine Redefinition ist. Andernfalls wird die Nummer der durch P redefinierten Prozedur übernommen.

Eine „geerbte“, nicht redefinierte Prozedur muß in diesem Zusammenhang nicht explizit mit einer Nummer versehen werden, da sie lediglich durch einen Verweis auf die Repräsentation der expliziten Bindung angesprochen wird.

4 Abschließende Bewertung

Die Bewertung des im Rahmen dieser Arbeit spezifizierten Oberon-2-Compilers Jacob erfolgt anhand eines Vergleichs mit bestehenden Compilern. Der für die Implementierung verwendete Modula-2-Compiler Mocka ist hierfür aufgrund seines ähnlichen Aufbaus besonders geeignet: Er besitzt ebenfalls ein Backend, das vom Werkzeug BEG generiert wird und das mittels des GNU Assemblers `as` und des Linkers `ld` monolithische Programme erzeugt.

Für einen direkten Vergleich von Mocka und Jacob wurden die Quelltexte von Jacob, die von den eingesetzten Werkzeugen generiert wurden und die zusätzlich „von Hand geschriebenen" Module manuell nach Oberon-2 „übersetzt". Aus den so transformierten Quelltexten sollte Jacob lediglich eine Programmdatei (executable) erzeugen können, ohne das diese sinnvoll ablauffähig sein mußte. Die notwendige Auflösung der With-Anweisung erfolgte bereits in den Modula-2-Quelltexten, um die Vergleichbarkeit nicht zu beeinträchtigen. Tabelle 1 zeigt das Ergebnis der Transformation von 86 Modulen.

Art der Datei	*Zeilenanzahl*	*Größe*
Modula-2-Definitionsmodule	7.885	383.671
Modula-2-Implementationsmodule	84.671	2.935.357
Oberon-2-Module	87.257	3.188.018

Tabelle 1: Größen der Testquelltexte

Für den Vergleich beider Compiler wurde der bereits für die Entwicklung eingesetzte und in Kapitel 3 (S. 43) beschriebene Rechner verwendet, wobei dieser auf eine Hauptspeichergröße von 48 MByte erweitert wurde, damit keine Nutzung der Hauptspeicherauslagerung (swapping) die Meßergebnisse beeinflußt. Dies ist insbesondere bei der Anwendung von Jacob auf große Quelltexte nötig, da bei der Spezifizierung von Jacob eine möglichst effiziente Speichernutzung nicht im Vordergrund stand.

4.1 Übersetzungszeit

Die in Tabelle 2 und Tabelle 3 aufgeführten Übersetzungszeiten dieser Quelltexte werden in Abbildung 25 dargestellt. Während für Jacob dabei die Zeiten in den einzelnen Phasen berücksichtigt werden, beschränkt sich die Abbildung bei Mocka lediglich auf die Unterscheidung der Übersetzungszeiten von Definitions- und Implementationsmodulen. Die Zeiten aller Phasen bis einschließlich des Evaluators konnten sich bei Jacob über Kommandozeilenparameter ermitteln lassen. Für die Ermittlung der Zeiten der ersten drei Phasen des Backends mußten demgegenüber die von diesen Phasen aufgerufenen Prozeduren der jeweils nachfolgenden Phase durch Prozeduren mit leerem Rumpf ersetzt werden.

Trotz des intensiven Einsatzes von Generierungswerkzeugen hält sich der damit verbundene Mehraufwand an Übersetzungszeit in vertretbaren Grenzen. Der hohe Zeitaufwand der Attributauswertungsphase läßt sich vor allem damit begründen, daß sämtliche dabei zu verarbeitende Informationen durch Attributzuweisungen von Knoten zu Knoten innerhalb des abstrakten Syntaxbaums weitergereicht werden müssen. Zusätzlich ist sicherlich die große Anzahl von eingesetzten Puma-Funktionen zur Attributberechnung der Laufzeit nicht zuträglich.

Die Verteilung der Laufzeiten in den Phasen des Backends ist überraschend, wobei insbesondere der hohe Anteil des Assemblersubsystems einer näheren Untersuchung bedarf.

„Phase“	*Zeit*	*Anteil*
Reader	1,2s	0,4%
Scanner	9,9s	3,6%
Parser	10,6s	3,9%
Evaluator	74,7s	27,5%
Zwischencodeerzeuger	8,9s	3,3%
Codeerzeuger	26,3s	9,7%
Assemblersubsystem	46,9s	17,2%
Assembler	85,5s	31,4%
Linker	8,1s	3,0%
Gesamt	272,2s	

Tabelle 2: Übersetzungszeit (Jacob)

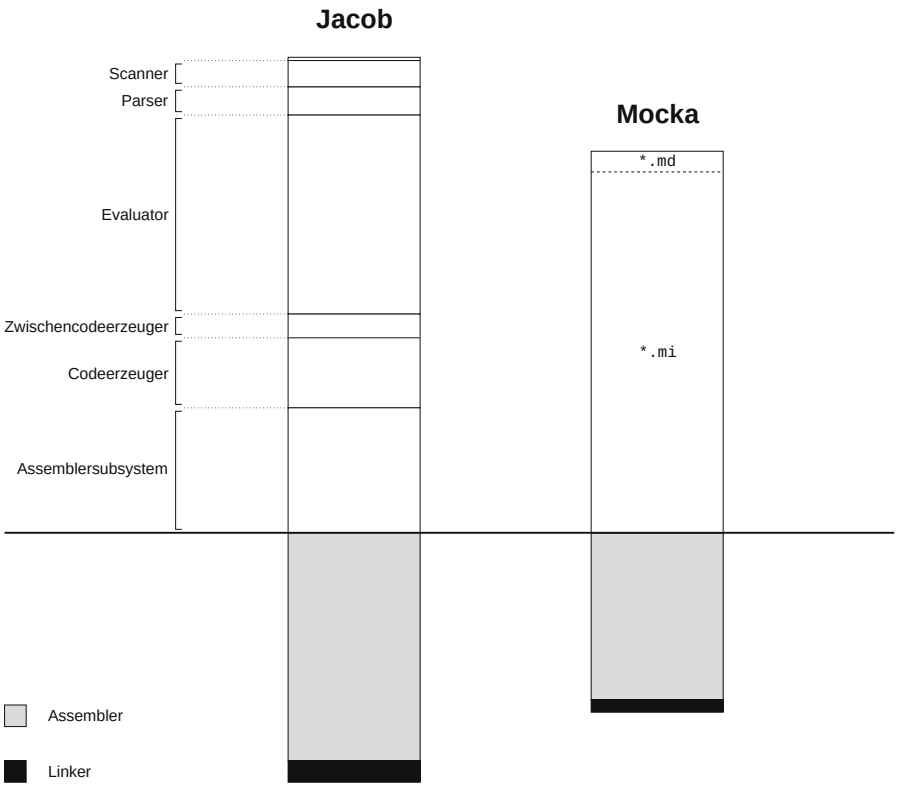

Abbildung 25: Vergleich der Übersetzungszeiten zwischen Jacob und Mocka

Art	*Zeit*	*Anteil*
Definitionsmodule	7,8s	3,7%
Implementationsmodule	135,4s	64,3%
Assembler	62,6s	29,7%
Linker	4,7s	2,2%
Gesamt	210,5s	

Tabelle 3: Übersetzungszeit (Mocka)

4.2 Codegröße

Die Tabellen 4 und 5 zeigen die Größen der aus der Übersetzung resultierenden Programme und geben über deren Bestandteile Auskunft.

Art	*Größe*	*Anteil*
Moduldeskriptoren	49.152	2,0%
davon: - Namen	499	
- Offsets	48.653	
Prozedurdeskriptoren	102.400	4,1%
davon: - Namen	32.768	
- Offsets	45.056	
Typdeskriptoren	508.009	20,6%
davon: - Namen	8.192	
- Skipper	4.096	
- Prozedurtabelle	0	
- Basistypen	12.288	
- Offsets	458.752	
- Initialisierung	16.489	
Variableninitialisierungen	10.149	0,4%
sonstiger Code	1.412.719	57,2%
Speicherkonstanten	28.672	1,2%
Wurzelmodul	21.553	0,9%
Laufzeitsystem	51.173	2,1%
Indexüberprüfungen	24.576	1,0%
Bereichsüberprüfungen	16.384	0,7%
Überprüfungen auf `NIL`	245.760	9,9%
überflüssige Parameterkopien	8.192	0,3%
Gesamt	2.470.547	

Tabelle 4: Codegrößen (Jacob)

Daß sich die Codegröße der von Jacob übersetzten Programme gegenüber Mocka mehr als verdoppelt hat, relativiert sich etwas durch die Größe der zugehörigen Typdeskriptoren und insbesondere deren Offset-Tabelle. Die Tatsache, daß der sonstige Code bei Jacob immer noch um den Faktor 1,7 größer ist als der von Mocka erzeugte Code, hat hauptsächlich folgende Ursache: In der Abwägung zwischen Codegröße und Laufzeit wurde bei Jacob im Gegensatz zu Mocka konsequent zugunsten der Laufzeit entschieden. Dies zeigt sich beispielsweise bei der „Entrollung“ von Schleifen zum Kopieren von kleineren Speicherbereichen und beim Epilog von Prozeduren.

Der hohe Anteil der Laufzeitüberprüfung von `NIL`-Zeigern an der Gesamtgröße ist durch die Anzahl der Zeigerdereferenzierungen innerhalb von Puma-Prozeduren erklärt. Eine

Deaktivierung dieser Überprüfung bedeutet jedoch keine Sicherheitseinbuße, da der Speicher vor illegalem Zugriff durch das Betriebssystem geschützt ist. Allerdings dient die Überprüfung vor allem dem schnellen Auffinden solcher Speicherschutzverletzungen. Das Fehlen einer solchen Option beim Modula-2-Compiler Mocka wurde während der Entwicklung von Jacob des öfteren mit längerer Fehlersuche bezahlt. Das größere Laufzeitsystem von Jacob geht auf die Integration der Freispeicherverwaltung zurück.

Art	*Größe*	*Anteil*
sonstiger Code	813.260	91,0%
Wurzelmodul	3.441	0,4%
Laufzeitsystem	11.030	1,2%
Indexüberprüfungen	32.768	3,7%
Bereichsüberprüfungen	32.768	3,7%
Gesamt	893.267	

Tabelle 5: Codegrößen (Mocka)

4.3 Laufzeit

Zur Einordnung des Laufzeitverhaltens wurde der von WEICKER [1988] in der Version 2 vorgestellte Dhrystone-Benchmark verwendet, welcher in seiner Struktur statistische Untersuchungen über die Verteilung von Programmiersprachkonstrukten bei der Systemprogrammierung berücksichtigt.

Die Abbildung 26 zeigt die Ergebnisse dieses Benchmarks für einige ausgewählte Compiler auf dem oben genannten Rechner. Bei iOP2 handelt es sich um den Oberon-2-Compiler iOP2 RC/NM V1.3 23.6.94 des Oberon-Systems Linux Oberon (TM) System 3 Version 1.5. Die Quelltexte dieses Benchmarks befinden sich (für die drei eingesetzten Sprachen) im Anhangsband zu dieser Arbeit. Alle Laufzeitüberprüfungen wurden soweit deaktiviert, wie dies der jeweilige Compiler gestattete. Der GNU C-Compiler wurde mit und ohne Optimierungen getestet.

Der eingesetzte Benchmark benutzt zwei Bibliotheks-Prozeduren für die String-Zuweisung und den String-Vergleich. In C wurden dafür die Funktionen `strcpy` und `strcmp` eingesetzt. Demgegenüber wurden für die Modula-2- und die Oberon-2-Version jeweils zwei Prozeduren `Assign` und `Compare` formuliert und zum jeweiligen Quelltext hinzugefügt. Da Oberon-2 für diese Funktionalität die vordeklarierte Prozedur `COPY` sowie eine entsprechende Relation zur Verfügung stellt, wurden auch diese in einem eigenen Test eingesetzt. Zusätzlich wurde Jacob noch mit der Optimierung bezüglich lokaler Kopien von Val-Parametern getestet.

4.4 Werkzeugbewertung

Das eingesetzte Werkzeug BEG erwies sich als durchaus brauchbar, um eine effiziente Codeerzeugung zu spezifizieren. Neben der bereits erwähnten Unwägbarkeit zusätzlicher Kosten durch eingefügte `Spill`-Instruktionen (vgl. S. 2) haben sich während der praktischen Benutzung noch einige Schwierigkeiten ergeben, die Verbesserungen wünschenswert erscheinen lassen.

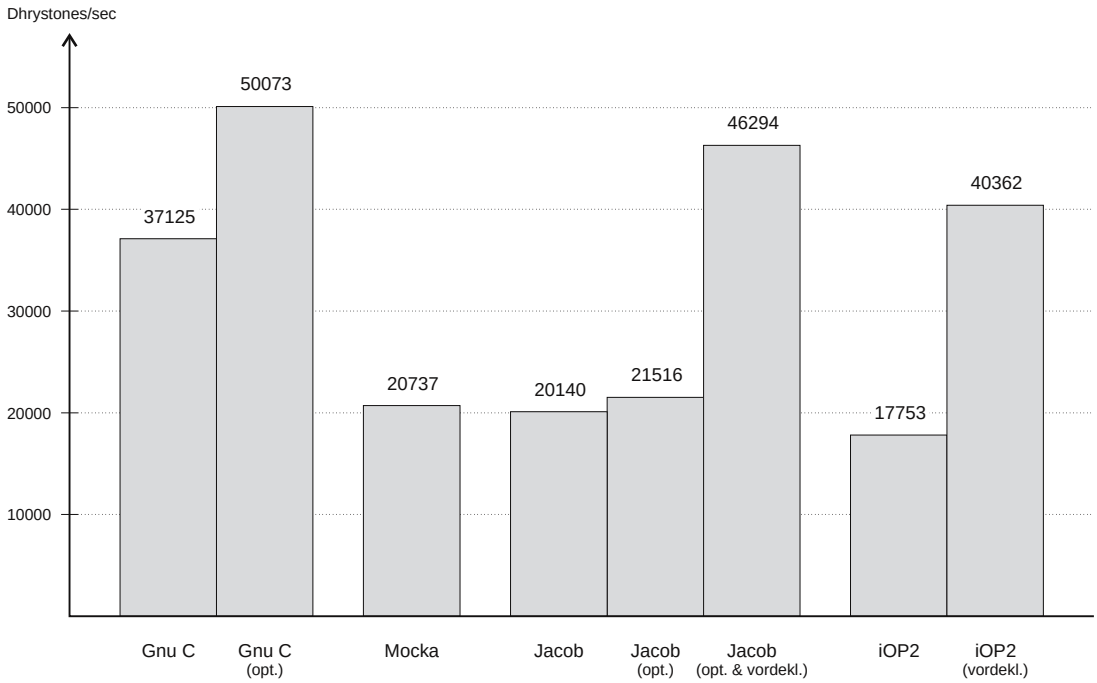

Abbildung 26: Dhrystone-Benchmark für einen Laufzeitvergleich

- Die Syntax von BEGL hat einige Defizite. Vor allem die fehlende Modularisierbarkeit erschwert die Spezifizierung größerer Codeerzeuger.
- Die Forderung, daß sich die Namen der Zwischensprachtypen von den Nichtterminalnamen unterscheiden müssen, führt zu einer unnötigen Anreicherung des Namensraums.
- Der Typ `BegRegister` sollte in einem eigenen Modul definierbar sein, damit eine Realisierung mit dem Assemblersubsystem leichter möglich ist.
- Es sollte die Möglichkeit bestehen, als Resultat einer Regel eine Registermenge spezifizieren zu können.
- Es gibt keine Möglichkeit, die Größe eines Registers zu definieren.
- Regellokale Variablen wären eine Erleichterung für den Benutzer
- Das Verbot der expliziten Registerallokation innerhalb von Regeln mit einem Adressierungsart-Nichtterminal als Resultat ist nicht nachvollziehbar.
- Die Möglichkeit der verschränkten Konstruktion von Ausdrucksbäumen wäre hilfreich.

4.5 Abschluß

Die Wahl der Sprache Oberon-2 als Quellsprache führte gegenüber Oberon an manchen Stellen der Spezifikation zu einem nicht unerheblichen Mehraufwand. Insbesondere sind mehrdimensionale offene-Arrays als Zeigerbasistypen ungleich schwieriger zu behandeln,

als offene Array-Parameter. Insgesamt war die Wahl von Oberon-2 jedoch richtig, da die dadurch gewonnenen Erfahrungen den zusätzlichen Aufwand rechtfertigen.

Die Codeerzeugung für Intel Prozessoren bereitete keine Schwierigkeiten. Zum einen macht das „flache“ Speichermodell des Betriebssystems die Benutzung der Segmentregister überflüssig. Zum anderen erleichtert die Verwendung der von BEG generierten *generellen* Registerallokation den Umgang mit vorhandenen Einschränkungen bezüglich der Anwendbarkeit von Registern bei bestimmten Instruktionen.

Das eingesetzte Betriebssystem Linux sowie der dort verfügbare Assembler `as` erwiesen sich als äußerst stabile Basis für die Durchführung eines solchen Projekts.

Zur Beurteilung des Werkzeugeinsatzes bei der Spezifikation von Jacob stellt die Betrachtung der Gesamtheit der erstellten Quelltexte einen geeigneten Ausgangspunkt dar. Dazu gibt Tabelle 6 einen Überblick über die Verteilung der Spezifikation auf die verschiedenen Quelltextarten. Die angegebene Gesamtgröße ist insbesondere im Vergleich mit der bei WIRTH und GUTKNECHT [1992] angegebenen Größe der Quelltexte des Oberon-Compilers zum Oberon-System von 4000 Zeilen beachtlich (S. 21).

Art der Dateien	*Zeilenanzahl*	*Anteil*	*Größe*	*Anteil*
Scanner-Spezifikation	392	1,2%	22049	1,5%
Parser-Spezifikation	1382	4,4%	73308	5,1%
Evaluator-Spezifikation	3960	12,5%	185236	12,9%
Ast-Spezifikationen	1236	3,9%	74885	5,2%
Puma-Spezifikationen	12401	39,2%	570293	39,6%
Codeerzeuger-Spezifikation	3919	12,4%	214768	14,9%
Modula-2-Quelltexte	8376	26,5%	298554	20,7%
Gesamt	31666		1439093	

Tabelle 6: Größen der Quelltexte

Der Versuch, den Umfang von Modula-2-Code in den Zielsprachenabschnitten der Spezifikationen zu minimieren, führte zu dem hohen Anteil an Puma-Spezifikationen, was sich auch negativ zur Laufzeit von Jacob bemerkbar macht. Bei einem erneuten Einsatz der Compiler-Compiler-Toolbox würde gerade der Einsatz des Werkzeugs Puma nicht so weit gehen. Wenn eine möglichst intensive Benutzung der Werkzeuge nicht im Vordergrund stünde, würde zur Spezifikation eines Oberon(-2)-Compilers das Werkzeug Ell zur Generierung eines LL(1)-Parsers das Werkzeug Lalr samt Evaluatorgenerator Ag ersetzen. Trotz der genannten Schwierigkeiten soll das Werkzeug BEG für eine eventuell weitere, dann in Oberon-2 „handgeschriebene“ Version von Jacob eingesetzt werden.

Abbildungsverzeichnis

Literaturverzeichnis

A.V. Aho, R. Sethi und J.D. Ullman [1988]:
Compilerbau,
Addison-Wesley.

R. Bauer und M. Spring [1994]:
Entwicklung eines Oberon-2-Compilers mit Hilfe einer Compiler-Compiler-Toolbox – Analysephase des Compilers,
Studienarbeit an der Technischen Universität Berlin, Fachbereich Informatik, November 1994.

M. Beck [1994]:
Linux-Kernel-Programmierung: Algorithmen und Strukturen der Version 1.0,
Addison-Wesley.

D. Elsner und J. Fenlason [1993]:
Using as – The GNU Assembler,
Free Software Foundation Inc., März 1993.

H. Emmelmann [1989]:
BEG – a Back End Generator, User Manual,
Arbeitspapiere der GMD 420, Gesellschaft für Mathematik und Datenverarbeitung mbh, Dezember 1989.

H. Emmelmann [1995]:
BEG Extensions Version 1.7,
Zweiseitiges Arbeitspapier, Januar 1995.

H. Emmelmann und J. Vollmer [1994]:
GMD MODULA SYSTEM MOCKA: User Manual,
Universität Karlsruhe, Institut für Programm- und Datenstrukturen, April 1994.

S. Hetze, D. Hohndel, M. Müller und O. Kirch [1994]:
LinuX Anwenderhandbuch und Leitfaden für die Systemverwaltung,
LunetIX Softfair, 4. erweiterte und aktualisierte Auflage.

H. Hopp [1993]:
Entwicklung eines Codegenerators für den Modula-2-Compiler Mocka mit BEG auf dem Intel 80386/387,
Studienarbeit, GMD Forschungsstelle an der Universität Karlsruhe, Oktober 1993.

B. Kirk (Hrsg.) [1993]:
The Oakwood Guidelines for Oberon-2 Compiler Developers,
Robinson Associates, Rev.1A, Dezember 1993.

B. Miller, D. Koski, C. Pheow Lee,
V. Maganty, R. Murthy, A. Natarajan und J. Steidl [1995]:
Nichts dazugelernt: Empirische Studie zur Zuverlässigkeit von Unix-Utilities,
iX Multiuser Multitasking Magazin, September 1995, S. 108ff.

H. Mössenböck und N. Wirth [1995]:
The Programming Language Oberon-2,
ETH Zürich, Institut für Computersysteme, März 1995,
ftp://ftp.inf.ethz.ch/pub/Oberon/Docu/Oberon2.Report.ps.Z

R.P. Nelson [1991]:
80386/486 – Handbuch für Programmierer,
Microsoft Press Deutschland, München.

C. Pfister, B. Heeb und J. Templ [1991]:
Oberon Technical Notes,
Report 156, ETH Zürich, Institut für Computersysteme, März 1991.

M. Reiser [1991]:
The Oberon System: User Guide and Programmer's Manual,
Addison-Wesley, New York.

M. Reiser und N. Wirth [1994]:
Programmieren in Oberon: Das neue Pascal,
Addison-Wesley, New York.

T. Timar [1995]:
Unix – Frequently Asked Questions,
comp.unix.questions, März 1995.

R.P. Weicker [1988]:
Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules,
SIGPLAN Notices, Vol. 23, No. 8, S. 49–62

R. Wilhelm und D. Maurer [1992]:
Übersetzerbau: Theorie, Konstruktion, Generierung,
Springer-Verlag, Berlin.

N. Wirth und J. Gutknecht [1992]:
Project Oberon: The Design of an Operating System and Compiler,
Addison-Wesley.