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

—

Analysephase des Compilers

Studienarbeit

November 1994

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 — Analysephase des Compilers

Studienarbeit an der
Technischen Universität Berlin
Fachbereich Informatik
Institut für Angewandte Informatik
FG Programmiersprachen und Compiler
im November 1994

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

Vorwort

Die Aufteilung der Studienarbeit im Rahmen der Gruppenarbeit:

Ralf Bauer befaßte sich mit der Aufbereitung der Sprachdefinition von Oberon-2 sowie der Erstellung des Scanners, Parsers und der Einbettung des Gesamtsystems. Er schrieb Kapitel 1 (Oberon-2) und die Abschnitte 3.1.1 (Überblick) bis 3.1.4 (Parser), 3.1.9 (Ausgabefunktionen) und 3.1.10 (Umgebende Zusatzmodule) sowie 3.2.1 (Sichtbarkeitsregeln bei Deklarationen) bis 3.2.5 (Vorausdeklarationen).

Maximilian Spring befaßte sich mit der Auswahl und Anwendung der Werkzeuge sowie der Erstellung des Evaluators mit seinen Zusatzmodulen und einer Testumgebung. Er schrieb Kapitel 2 (Werkzeuge) und die Abschnitte 3.1.5 (Evaluatorobjekte) bis 3.1.8 (Evaluatorhilfsfunktionen), 3.2.6 (Typerweiterungen) bis 3.2.10 (Vordeklarierte Prozeduren) sowie 3.3 (Testumgebung).

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

Die Arbeit wurde von Mario Kröplin betreut.

Inhaltsverzeichnis

Überblick

Diese Arbeit ist die Dokumentation der Implementierung eines Front-Ends für die Programmiersprache Oberon-2. Die Implementierung erfolgte mit der Compiler-Compiler-Toolbox von Dr. Josef Grosch (GMD Karlsruhe), einer Sammlung von Einzelwerkzeugen für die Generierung von Programmfragmenten für die verschiedenen Übersetzerphasen. Neben der Erstellung von Spezifikationstexten für diese Werkzeuge wurden umgebende Zusatzmodule in Modula-2 entworfen.

Ziel der Studienarbeit war die Untersuchung der praktischen Benutzbarkeit dieser Werkzeuge anhand ihres Einsatzes am Beispiel einer *echten* Programmiersprache.

Es wurde Oberon-2 als Quellsprache gewählt, da sie eine vom Sprachumfang kleine, aber durch ihren Typerweiterungsmechanismus mächtige Programmiersprache ist, die objektorientiertes Programmieren unterstützt.

Die Dokumentation gliedert sich in drei Kapitel. Die Sprache Oberon-2 wird im ersten Kapitel definiert. Die Definition besteht aus einer Übersetzung des englischsprachigen Oberon-2 Reports, die um einige kurze Passagen ergänzt wurde. Das zweite Kapitel beschreibt die bei der Implementierung eingesetzten Werkzeuge. Im dritten und letzten Kapitel wird auf die erstellte Implementierung eingegangen. Es besteht aus einer Beschreibung der einzelnen Spezifikationsdokumente und der Darstellung der gewählten Lösungen zu speziellen Problembereichen.

Obwohl die Dokumentation auf deutsch vorliegt, wurden die im Übersetzerbau 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 Dokumentation werden verschiedene Schriftarten benutzt, um dem Leser die Erkennung 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 Helvetica verwendet.

1. Oberon-2

1.1 Sprachdefinition von Oberon-2

Das vorliegende Kapitel stellt die Definition der Sprache Oberon-2 dar. Es handelt sich hierbei um eine möglichst wörtliche deutsche Übersetzung des englischsprachigen Oberon-2 Reports von MÖSSENBÖCK und WIRTH [1993]. Diese Übersetzung orientiert sich lediglich in den Begrifflichkeiten und in einzelnen Passagen an der deutschsprachigen Sprachdefinition von Oberon-2 von MÖSSENBÖCK [1993], da diese im Vergleich zum englischsprachigen Original Unterschiede aufweist. Gegenüber dem englischsprachigen Report wurden einzelne Ergänzungen und Präzisierungen vorgenommen. Diese Abweichungen vom Original stehen jeweils in den eckigen Klammern [und] und sind durch einen anderen Font zusätzlich kenntlich gemacht. Sie werden im Anschluß an die Sprachdefinition näher erläutert. Der Abschnitt 1.1.12.4 Die Oberon-Umgebung (S.28) ist lediglich aus Gründen der Vollständigkeit übernommen und besitzt für diese Arbeit keine Bedeutung.

1.1.1 Einleitung

Oberon-2 ist eine universelle Programmiersprache in der Tradition von Oberon und Modula-2. Ihre wichtigsten Merkmale sind Blockstruktur, Modulkonzept, getrennte Übersetzung, strenge Typenprüfung zur Übersetzungszeit (auch über Modulgrenzen hinweg) und Typerweiterung mit typgebundenen Prozeduren.

Die Typerweiterung macht Oberon-2 zu einer objektorientierten Programmiersprache. Ein Objekt ist eine Variable eines abstrakten Datentyps, bestehend aus privaten Daten (seinem Zustand) und Prozeduren, die auf diesen Daten operieren. Abstrakte Datentypen werden als erweiterbare Records deklariert. Oberon-2 deckt die meisten Ausdrücke objektorientierter Sprachen durch das gängige Vokabular von imperativen Sprachen ab, um die Anzahl von Begriffen ähnlicher Konzepte zu minimieren.

Dieser Report ist nicht als Programmierlehrbuch gedacht. Er ist bewußt knapp gehalten. Er soll als Referenz für Programmierer, Übersetzer-Implementierer und Autoren von Handbüchern dienen. Wenn etwas undefiniert bleibt, ist dies meist absichtlich so, da es aus den angegebenen Regeln der Sprache hervorgeht oder weil es die Sprachdefinition dort festlegen würde, wo eine allgemeine Festlegung nicht ratsam erscheint.

Abschnitt 1.1.12.1 definiert einige Begriffe, die zur Beschreibung der Kontextbedingungen von Oberon-2 benötigt werden. Wo sie im Text vorkommen, werden sie kursiv geschrieben, um ihre besondere Bedeutung hervorzuheben (z.B. *derselbe* Typ).

1.1.2 Syntax

Die Syntax von Oberon-2 wird in erweiterter Backus-Naur-Form (EBNF) beschrieben: Alternativen werden durch | getrennt. Eckige Klammern [und] umschließen Ausdrücke, die fehlen dürfen. Geschweifte Klammern { und } umschließen Ausdrücke, die (null- oder mehrmals) wiederholt werden dürfen. Nichtterminale beginnen mit einem Großbuchstaben (z.B. Statement). Terminale beginnen entweder mit einem Kleinbuchstaben (z.B. ident) oder werden ganz mit Großbuchstaben geschrieben (z.B. BEGIN) oder sind Zeichenketten (z.B. ":=").

1.1.3 Vokabular und Repräsentation

Die Repräsentation von Symbolen (Terminale) durch Zeichen wird über den ASCII-Zeichensatz definiert. Symbole sind Namen, Zahlen, Zeichenketten, Operatoren und Begrenzer. Die folgenden lexikalischen Regeln müssen beachtet werden: Leerzeichen und Zeilenumbrüche dürfen nicht in Symbolen vorkommen (außer in Kommentaren, und Leerzeichen in Zeichenketten). Sie werden ignoriert, es sei denn, sie haben eine Bedeutung für die Trennung zweier aufeinanderfolgender Symbole. Groß- und Kleinbuchstaben werden als unterschiedlich angesehen.

1. *Namen* sind Folgen von Buchstaben und Ziffern. Das erste Zeichen muß ein Buchstabe sein.

```
ident = letter { letter | digit }.
```

Beispiele: `x` `Scan` `Oberon2` `GetSymbol` `firstLetter`

2. *Zahlen* sind (vorzeichenlose) ganzzahlige oder reelle Konstanten. Der Typ einer ganzzahligen Konstanten ist der kleinste Typ, zu dem der Konstantenwert gehört (siehe 1.1.6.1). Endet eine ganzahlige Konstante mit dem Buchstaben `H`, so ist ihre Darstellung hexadezimal, sonst ist ihre Darstellung dezimal.

Eine reelle Zahl enthält immer einen Dezimalpunkt und wahlweise einen dezimalen Exponenten. Der Buchstabe `E` (oder `D`) bedeutet ,,mal zehn hoch“. Eine reelle Zahl ist vom Typ `REAL`, außer sie hat einen Exponenten, der den Buchstaben `D` enthält. In diesem Fall ist sie vom Typ `LONGREAL`.

```
number      =  integer | real.
integer     =  digit { digit } | digit { hexDigit } "H".
real        =  digit { digit } "." { digit } [ ScaleFactor ].
ScaleFactor =  ( "E" | "D" ) [ "+" | "-" ] digit { digit }.
hexDigit    =  digit | "A" | "B" | "C" | "D" | "E" | "F".
digit       =  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Beispiele:

Zahl	*Typ*	*Wert*
`1991`	`INTEGER`	`1991`
`0DH`	`SHORTINT`	`13`
`12.3`	`REAL`	`12.3`
`4.567E8`	`REAL`	`456700000`
`0.57712566D-6`	`LONGREAL`	`0.00000057712566`

3. *Zeichenkonstanten* werden durch die Ordnungszahl des Zeichens in hexadezimaler Notation angegeben, gefolgt vom Buchstaben `X`.

```
character = digit { hexdigit } "X".
```

4. *Zeichenketten* sind Folgen von Zeichen zwischen einfachen (`'`) oder doppelten (`"`) Hochkommas. Das öffnende Hochkomma muß das gleiche wie das schließende Hochkomma sein und darf in der Zeichenkette nicht vorkommen. Die Anzahl der Zeichen in einer Zeichenkette wird ihre *Länge* genannt. Eine Zeichenkette der Länge 1 kann überall dort benutzt werden, wo eine Zeichenkonstante erlaubt ist und umgekehrt. `[Die Zeichenkonstante 0X hat als Zeichenkette die Länge 0.]`

```
string = '"' { char } '"' | "'" { char } "'".
```

Beispiele: `"Oberon-2"` `"Don't worry"` `"x"`

5. *Operatoren* und *Begrenzer* sind die nachfolgend aufgeführten Sonderzeichen, Zeichenpaare und Schlüsselwörter. Schlüsselwörter bestehen ausschließlich aus Großbuchstaben und dürfen nicht als Namen verwendet werden.

```
+           :=          ARRAY       IMPORT      RETURN
-           ^           BEGIN       IN          THEN
*           =           BY          IS          TO
/           #           CASE        LOOP        TYPE
~           <           CONST       MOD         UNTIL
&           >           DIV         MODULE      VAR
.           <=          DO          NIL         WHILE
,           >=          ELSE        OF          WITH
;           ..          ELSIF       OR
|           :           END         POINTER
(           )           EXIT        PROCEDURE
[           ]           FOR         RECORD
{           }           IF          REPEAT
```

6. *Kommentare* können zwischen je zwei Symbolen in einem Programm eingefügt werden. Sie sind beliebige Zeichenfolgen, eingeleitet durch die Klammer `(*` und abgeschlossen durch `*)`. Kommentare dürfen geschachtelt werden. Sie beeinflussen die Bedeutung eines Programms nicht.

1.1.4 Deklarationen und Sichtbarkeitsbereichsregeln

Jeder in einem Programm vorkommende Name muß durch eine *Deklaration* eingeführt werden, mit Ausnahme von vordeklarierten Namen. Die Deklaration legt auch gewisse bleibende Eigenschaften eines Objekts fest, beispielsweise ob es eine Konstante, ein Typ, eine Variable oder eine Prozedur ist. Der Name wird dann dazu verwendet, um sich auf das betreffende Objekt zu beziehen.

Der *Sichtbarkeitsbereich* eines Objekts *x* erstreckt sich textuell von der Stelle seiner [Namensnennung bei der] Deklaration bis zum Ende des Blocks (Modul, Prozedur oder Record), zu dem die Deklaration gehört und zu dem das Objekt daher *lokal* ist. Er schließt die Sichtbarkeitsbereiche gleichnamiger Objekte aus, die in geschachtelten Blöcken deklariert werden. Es gelten die folgenden Regeln:

1. Kein Name darf innerhalb eines gegebenen Sichtbarkeitsbereichs mehr als ein Objekt bezeichnen (d.h. kein Name darf in einem Block doppelt deklariert werden);

2. Auf ein Objekt darf nur innerhalb seines Sichtbarkeitsbereichs Bezug genommen werden [, jedoch nicht innerhalb der eigenen Deklaration, falls es sich um eine Konstanten- oder Variablendeklaration handelt];

3. Ein Typ *T* der Form `POINTER TO T1` (siehe 1.1.6.4) kann an einer Stelle deklariert werden, an der *T1* noch unbekannt ist. Die Deklaration von *T1* muß im gleichen Block erfolgen, zu dem *T* lokal ist;

4. Namen von Record-Feldern (siehe 1.1.6.3) oder typgebundenen Prozeduren (siehe 1.1.10.2) sind nur innerhalb von Record-Bezeichnern gültig.

Einem Namen, der in einem Modulblock deklariert ist, darf eine Exportmarke ("*" oder "-") folgen, um anzuzeigen, daß er exportiert wird. Ein von einem Modul *M* exportierter Name *x* darf in anderen Modulen verwendet werden, falls sie *M* importieren (siehe 1.1.11). Er wird in diesen Modulen als *M.x* geschrieben und heißt *qualifizierter Name*. Namen, die bei ihrer De-

klaration mit "-" markiert werden, sind in importierenden Modulen schreibgeschützt. [Lediglich Namen von Variablen und Record-Feldern dürfen mit "-" markiert werden.]

```
Qualident   =  [ ident "." ] ident.
IdentDef    =  ident [ "*" | "-" ].
```

Die folgenden Namen sind vordeklariert; ihre Bedeutung wird in den angegebenen Abschnitten definiert:

ABS	(1.1.10.3)	LEN	(1.1.10.3)
ASSERT	(1.1.10.3)	LONG	(1.1.10.3)
ASH	(1.1.10.3)	LONGINT	(1.1.6.1)
BOOLEAN	(1.1.6.1)	LONGREAL	(1.1.6.1)
CAP	(1.1.10.3)	MAX	(1.1.10.3)
CHAR	(1.1.6.1)	MIN	(1.1.10.3)
CHR	(1.1.10.3)	NEW	(1.1.10.3)
COPY	(1.1.10.3)	ODD	(1.1.10.3)
DEC	(1.1.10.3)	ORD	(1.1.10.3)
ENTIER	(1.1.10.3)	REAL	(1.1.6.1)
EXCL	(1.1.10.3)	SET	(1.1.6.1)
FALSE	(1.1.6.1)	SHORT	(1.1.10.3)
HALT	(1.1.10.3)	SHORTINT	(1.1.6.1)
INC	(1.1.10.3)	SIZE	(1.1.10.3)
INCL	(1.1.10.3)	TRUE	(1.1.6.1)
INTEGER	(1.1.6.1)		

1.1.5 Konstantendeklarationen

Eine Konstantendeklaration verknüpft einen Namen mit einem Konstantenwert.

```
ConstantDeclaration  =  IdentDef "=" ConstExpression.
ConstExpression      =  Expression.
```

Ein Konstantenausdruck ist ein Ausdruck, der durch rein textuelle Betrachtung ausgewertet werden kann, ohne das Programm auszuführen. Seine Operanden sind Konstanten (1.1.8) oder vordeklarierte Funktionen (1.1.10.3), deren Wert zur Übersetzungszeit ausgewertet werden kann. [Der Typ eines ganzzahligen Konstantenausdrucks ist der kleinste Typ, der den Wert des Konstantenausdrucks einschließt.] Beispiele für Konstantendeklarationen sind:

```
N = 100
limit = 2*N-1
fullSet = {MIN(SET)..MAX(SET)}
```

1.1.6 Typdeklarationen

Ein Datentyp bestimmt die Menge der Werte, die Variablen dieses Typs annehmen können, sowie die anwendbaren Operatoren. Eine Typdeklaration verknüpft einen Namen mit einem Typ. Bei strukturierten Typen (Arrays und Records) definiert sie auch die Struktur von Variablen dieses Typs. [Ein Typ darf nicht in seiner eigenen Deklaration verwendet werden, außer als Zeigerbasistyp (siehe 1.1.6.4) oder Typ eines formalen Var-Parameters (siehe 1.1.10.1).]

```
TypeDeclaration  =  IdentDef "=" Type.
Type             =  Qualident | ArrayType | RecordType | PointerType | ProcedureType.
```

Beispiele:

```
Table = ARRAY N OF REAL
Tree = POINTER TO Node
Node = RECORD
  key: INTEGER;
  left, right: Tree
END
CenterTree = POINTER TO CenterNode
CenterNode = RECORD (Node)
  width: INTEGER;
  subnode: Tree
END
Function = PROCEDURE (x: INTEGER): INTEGER
```

1.1.6.1 Standardtypen

Die Standardtypen werden durch vordeklarierte Namen bezeichnet. Die auf einen Standardtyp anwendbaren Operatoren sind in Abschnitt 1.1.8.2 und die vordeklarierten Funktionsprozeduren in Abschnitt 1.1.10.3 beschrieben. Die Standardtypen haben folgende Wertebereiche:

1. BOOLEAN die Wahrheitswerte TRUE und FALSE
2. CHAR die Zeichen des erweiterten ASCII-Zeichensatzes (0X..0FFX)
3. SHORTINT die ganzen Zahlen zwischen MIN(SHORTINT) und MAX(SHORTINT)
4. INTEGER die ganzen Zahlen zwischen MIN(INTEGER) und MAX(INTEGER)
5. LONGINT die ganzen Zahlen zwischen MIN(LONGINT) und MAX(LONGINT)
6. REAL die reellen Zahlen zwischen MIN(REAL) und MAX(REAL)
7. LONGREAL die reellen Zahlen zwischen MIN(LONGREAL) und MAX(LONGREAL)
8. SET die Mengen der ganzen Zahlen zwischen MIN(SET) und MAX(SET)

Die Typen 3 bis 5 werden *ganzzahlige* Typen genannt, die Typen 6 und 7 *reelle* Typen; zusammen heißen sie *numerische Typen*. Sie bilden eine Hierarchie: der größere Typ *schließt* den kleineren Typ (seine Werte) *ein*.

```
LONGREAL ⊇ REAL ⊇ LONGINT ⊇ INTEGER ⊇ SHORTINT
```

1.1.6.2 Array-Typen

Ein Array ist eine Struktur, welche aus einer Anzahl von Elementen besteht, die alle vom gleichen Typ sind, der *Elementtyp* genannt wird. Die Anzahl der Elemente eines Arrays wird seine *Länge* genannt [und muß größer 0 sein]. Die Elemente werden durch Indizes bezeichnet, die ganze Zahlen zwischen 0 und der Länge minus 1 sind.

```
ArrayType  =  ARRAY [ Length { "," Length } ] OF Type.
Length     =  ConstExpression.
```

Ein Typ der Form

```
ARRAY L0, L1, ..., Ln OF T
```

ist eine Abkürzung für

```
ARRAY L0 OF
   ARRAY L1 OF
      ...
         ARRAY Ln OF T
```

Arrays, die ohne Längenangabe deklariert werden, nennt man *offene Arrays*. Sie dürfen nur als Zeigerbasistypen (siehe 1.1.6.4), Elementtypen von offenen Array-Typen und Typen formaler Parameter (siehe 1.1.10.1) verwendet werden.

Beispiele:

```
ARRAY 10, N OF INTEGER
ARRAY OF CHAR
```

1.1.6.3 Record-Typen

Ein Record-Typ ist eine Struktur, bestehend aus einer festen Anzahl von Elementen, genannt *Felder*, die möglicherweise verschiedene Typen besitzen. Die Deklaration eines Record-Typs definiert den Namen und den Typ jedes Felds. Der Sichtbarkeitsbereich der Feldnamen erstreckt sich von ihrer Deklarationsstelle bis zum Ende des Record-Typs; sie sind aber auch in Bezeichnern sichtbar, die sich auf Felder von Record-Variablen beziehen (siehe 1.1.8.1). Wird ein Record-Typ exportiert, müssen Felder, die außerhalb des deklarierenden Moduls sichtbar sein sollen, mit einer Exportmarke versehen werden. Diese Felder sind *öffentlich*; unmarkierte Felder sind *privat*.

```
RecordType =  RECORD [ "(" BaseType ")" ] FieldList { ";" FieldList } END.
BaseType   =  Qualident.
FieldList  =  [ IdentList ":" Type ].
```

Record-Typen sind erweiterbar, d.h. ein Record-Typ kann als *Erweiterung* eines anderen Record-Typs deklariert werden. Im Beispiel

```
T0 = RECORD x: INTEGER END
T1 = RECORD (T0) y: REAL END
```

ist *T1* eine (direkte) *Erweiterung* von *T0* und *T0* ist der (direkte) *Basistyp* von *T1* (siehe 1.1.12.1). Ein erweiterter Typ *T1* besteht aus den Feldern seines Basistyps und aus den in *T1* deklarierten Feldern (siehe 1.1.6). Alle Namen, die in dem erweiterten Record deklariert sind, müssen sich von den `[(sichtbaren)]` Namen unterscheiden, die in seinem (seinen) Basistyp-Record(s) deklariert sind.

Beispiele für Record-Deklarationen:

```
RECORD
  day, month, year: INTEGER
END
```

```
RECORD
  name, firstname: ARRAY 32 OF CHAR;
  age: INTEGER;
  salary: REAL
END
```

1.1.6.4 Zeigertypen

Variablen eines Zeigertyps *P* nehmen als Werte Zeiger auf Variablen eines gewissen Typs *T* an. *T* wird der Zeigerbasistyp von *P* genannt und muß ein Record- oder Array-Typ sein. Zeigertypen übernehmen die Erweiterungsbeziehung ihrer Zeigerbasistypen: Wenn ein Typ *T1* eine Erweiterung von *T* ist und *P1* vom Typ `POINTER TO` *T1* ist, dann ist *P1* auch eine Erweiterung von *P*.

```
PointerType = POINTER TO Type.
```

Wenn eine Variable *p* vom Typ *P* = POINTER TO *T* ist, bewirkt der Aufruf der vordeklarierten Prozedur *NEW(p)* (siehe 1.1.10.3), daß eine Variable vom Typ *T* im freien Speicher angelegt wird. Falls *T* ein Record-Typ oder ein Array-Typ mit fester Länge ist, muß die Speicheranforderung mit *NEW(p)* durchgeführt werden; falls *T* ein n-dimensionaler offener Array-Typ ist, muß die Speicheranforderung mit $NEW(p, e_0, \ldots, e_{n-1})$ erfolgen, wobei die Ausdrücke $e_0, \ldots, e_{n-1}$ die gewünschten Längen des Arrays angeben. In jedem Fall wird *p* ein Zeiger auf die angelegte Variable zugewiesen. *p* ist vom Typ *P*. Die *referenzierte* Variable *p^* (sprich *p-referenziert*) ist vom Typ *T*. Jede Zeigervariable kann den Wert NIL annehmen, der auf keine Variable verweist.

1.1.6.5 Prozedurtypen

Variablen eines Prozedurtyps *T* enthalten als Wert eine Prozedur oder NIL. Eine Prozedur *P* kann einer Variablen des Typs *T* zugewiesen werden, wenn die formalen Parameterlisten (siehe 1.1.10.1) von *T* und *P übereinstimmen* (siehe 1.1.12.1). *P* darf weder eine vordeklarierte oder typgebundene Prozedur sein, noch darf sie lokal zu einer anderen Prozedur sein.

ProcedureType = PROCEDURE [FormalParameters].

1.1.7 Variablendeklarationen

Variablendeklarationen führen Variablen ein, indem sie einen Namen und einen Datentyp für sie definieren.

VariableDeclaration = IdentList ":" Type.

Record- und Zeigervariablen haben sowohl einen *statischen Typ* (der Typ, mit dem sie deklariert sind – einfach ihr Typ genannt) als auch einen *dynamischen Typ* (der Typ ihres Werts zur Laufzeit). Bei Zeigern und Var-Parametern eines Record-Typs kann der dynamische Typ eine Erweiterung ihres statischen Typs sein. Der statische Typ bestimmt, welche Felder eines Records ansprechbar sind. Der dynamische Typ wird benutzt, um typgebundene Prozeduren aufzurufen (siehe 1.1.10.2).

Beispiele für Variablendeklarationen (siehe Beispiele in 1.1.6):

```
i, j, k: INTEGER
x, y: REAL
p, q: BOOLEAN
s: SET
F: Function
a: ARRAY 100 OF REAL
w: ARRAY 16 OF RECORD
     name: ARRAY 32 OF CHAR;
     count: INTEGER
   END
t, c: Tree
```

1.1.8 Ausdrücke

Ausdrücke sind Konstrukte zur Formulierung von Berechnungsregeln, in denen Konstanten und aktuelle Variablenwerte kombiniert werden, um andere Werte durch die Anwendung von Operatoren und Funktionsprozeduren zu berechnen. Ausdrücke bestehen aus Operanden und Operatoren. Um spezielle Assoziativitäten von Operatoren und Operanden auszudrücken, können Klammern verwendet werden.

1.1.8.1 Operanden

Mit Ausnahme von Mengenkonstruktoren und literalen Konstanten (Zahlen, Zeichenkonstanten oder Zeichenketten) werden Operanden durch *Bezeichner* ausgedrückt. Ein Bezeichner besteht aus einem Namen, der sich auf eine Konstante, Variable oder Prozedur bezieht. Dieser Name kann möglicherweise durch einen Modulnamen qualifiziert sein (siehe 1.1.4 und 1.1.11) und darf von *Selektoren* gefolgt werden, falls das bezeichnete Objekt ein Element einer Struktur ist.

```
Designator     =  Qualident { "." ident | "[" ExpressionList "]" | "^" | "(" Qualident ")" }.
ExpressionList =  Expression { "," Expression }.
```

Wenn *a* ein Array ist, bezeichnet *a[e]* dasjenige Element von *a*, dessen Index durch den Wert des Ausdrucks *e* gegeben ist. Der Typ von *e* muß *ganzzahlig* sein. Ein Bezeichner der Form $a[e_0,e_1,\ldots,e_n]$ steht für $a[e_0][e_1]\ldots[e_n]$. Wenn *r* ein Record ist, bezeichnet *r.f* das Feld *f* von *r* oder die Prozedur *f*, die an den dynamischen Typ von *r* gebunden ist (siehe 1.1.10.2). Wenn *p* ein Zeiger ist, bezeichnet *p^* die Variable auf die *p* verweist. Die Bezeichner *p^.f* und *p^[e]* können abgekürzt werden zu *p.f* und *p[e]*, d.h. Record- und Array-Selektoren implizieren Dereferenzierung. Wenn *a* oder *r* schreibgeschützt sind, dann sind es auch *a[e]* und *r.f*.

Eine *Typzusicherung* *v(T)* garantiert, daß *v* vom dynamischen Typ *T* (oder einer Erweiterung von *T*) ist, d.h. das Programm wird abgebrochen, falls der dynamische Typ von *v* nicht *T* (oder eine Erweiterung von *T*) ist. Innerhalb des Bezeichners wird dann *v* so betrachtet, als ob *v* vom statischen Typ *T* wäre. Die Zusicherung ist anwendbar, wenn

1. *v* ein Var-Parameter mit Record-Typ ist oder *v* ein Zeiger ist, und wenn

2. *T* eine Erweiterung des statischen Typs von *v* ist.

Ist das bezeichnete Objekt eine Konstante oder eine Variable, bezieht sich der Bezeichner auf dessen aktuellen Wert. Ist es eine Prozedur, bezieht sich der Bezeichner auf diese Prozedur, außer er wird von einer (möglicherweise leeren) Parameterliste gefolgt; er impliziert in diesem Fall einen Aufruf dieser Prozedur und steht für den resultierenden Wert ihrer Ausführung. Die aktuellen Parameter müssen mit den formalen Parametern genau wie bei gewöhnlichen Prozeduraufrufen korrespondieren (siehe 1.1.10.1).

Beispiele für Bezeichner (siehe Beispiele in 1.1.7):

```
i                       (INTEGER)
a[i]                    (REAL)
w[3].name[i]            (CHAR)
t.left.right            (Tree)
t(CenterTree).subnode   (Tree)
```

1.1.8.2 Operatoren

Vier Klassen von Operatoren mit verschiedenen Vorrangregeln (Bindungsstärken) werden in Ausdrücken syntaktisch unterschieden. Der Operator ~ hat den höchsten Vorrang, gefolgt von Multiplikations-, Additions- und Vergleichsoperatoren. Operatoren mit demselben Vorrang binden von links nach rechts: Beispielsweise steht *x - y - z* für *(x - y) - z*.

```
Expression       =  SimpleExpression [ Relation SimpleExpression ].
SimpleExpression =  [ "+" | "-" ] Term { AddOperator Term }.
Term             =  Factor { MulOperator Factor }.
```

```
Factor          = Designator [ ActualParameters ]
                | number | character | string | NIL | Set | "(" Expression ")" | "~" Factor.
Set             = "{" [ Element { "," Element } ] "}".
Element         = Expression [ ".." Expression ].
ActualParameters = "(" [ ExpressionList ] ")".
Relation        = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
AddOperator     = "+" | "-" | OR.
MulOperator     = "*" | "/" | DIV | MOD | "&".
```

Die verfügbaren Operatoren sind nachfolgend aufgeführt. Einige Operatoren können auf Operanden verschiedener Typen angewendet werden und bedeuten dann verschiedene Operationen. In diesen Fällen wird die tatsächliche Operation durch den Typ der Operanden bestimmt. Die Operanden müssen miteinander *ausdruckskompatibel* hinsichtlich des Operators sein (siehe 1.1.12.1).

Logische Operatoren

`OR`	logische Disjunktion	`p OR q`	≡	,,wenn p dann TRUE, sonst q“
`&`	logische Konjunktion	`p & q`	≡	,,wenn p dann q, sonst FALSE“
`~`	Negation	`~p`	≡	,,nicht p“

Diese Operatoren werden auf Operanden vom Typ `BOOLEAN` angewendet und liefern einen Wert vom Typ `BOOLEAN`.

Arithmetische Operatoren

`+`	Summe
`-`	Differenz
`*`	Produkt
`/`	reeller Quotient
`DIV`	ganzzahliger Quotient
`MOD`	Modulo-Operator

Die Operatoren `+`, `-`, `*` und `/` werden auf Operanden mit *numerischem* Typ angewendet. Der Ergebnistyp ist der Typ des Operanden, der den Typ des anderen Operanden *einschließt*, außer bei der Division (`/`), wo das Resultat der kleinste *reelle* Typ ist, der beide Operandentypen einschließt. Als monadischer Operator bedeutet `-` Vorzeichenumkehrung und `+` die Identitätsoperation. Die Operatoren `DIV` und `MOD` können nur auf Operanden mit *ganzzahligem* Typ angewendet werden. Sie stehen über folgende Formeln miteinander in Beziehung, die für ein beliebiges x und einen positiven Divisor y definiert sind:

```
x = (x DIV y) * y + (x MOD y)
0 ≤ (x MOD y) < y
```

Beispiele:

```
x      y      x DIV y   x MOD y
5      3         1         2
-5     3        -2         1
```

Mengenoperatoren

+	Vereinigung
-	Mengendifferenz (x - y = x * (-y))
*	Durchschnitt
/	symmetrische Mengendifferenz (x / y = (x - y) + (y - x))

Mengenoperatoren werden auf Operanden vom Typ SET angewendet und liefern ein Resultat vom Typ SET. Das monadische Minuszeichen bezeichnet das Komplement von x, d.h. -x bezeichnet die Menge aller ganzen Zahlen zwischen 0 und MAX(SET), die nicht Elemente von *x* sind. Mengenoperatoren sind nicht assoziativ ((a + b) - c # a + (b - c)).

Ein *Mengenkonstruktor* definiert den Wert einer Menge durch Aufzählung ihrer Elemente zwischen geschweiften Klammern. Die Elemente müssen ganze Zahlen im Bereich 0..MAX(SET) sein. Der Bereich *a*..*b* enthält alle ganzen Zahlen aus dem Intervall [*a*,*b*]. [Falls *a* > *b* ist, ist das Intervall leer.]

Vergleichsoperatoren

=	gleich
#	ungleich
<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich
IN	Mengenzugehörigkeit
IS	Typtest

Vergleiche liefern ein Resultat vom Typ BOOLEAN. Die Vergleiche =, #, <, <=, > und >= können auf *numerische* Typen, CHAR, Zeichenketten und Zeichen-Arrays, die 0X als Abschluß enthalten, angewendet werden. Die Vergleiche = und # können zusätzlich sowohl auf BOOLEAN und SET, als auch auf Zeiger- und Prozedurtypen (einschließlich des Werts NIL) angewendet werden. *x IN s* bedeutet „*x* ist ein Element von *s*“. *x* muß von einem *ganzzahligen* Typ sein und *s* vom Typ SET. *v IS T* bedeutet „der dynamische Typ von *v* ist *T* (oder eine Erweiterung von *T*)“ und wird *Typtest* genannt. Er ist anwendbar, wenn

1. *v* ein Var-Parameter mit Record-Typ ist oder *v* ein Zeiger ist, und wenn
2. *T* eine Erweiterung des statischen Typs von *v* ist.

Beispiele für Ausdrücke (siehe Beispiele in 1.1.7):

```
1991                  (INTEGER)
i DIV 3               (INTEGER)
~p OR q               (BOOLEAN)
(i+j) * (i-j)         (INTEGER)
s - {8, 9, 13}        (SET)
i + x                 (REAL)
a[i+j] * a[i-j]       (REAL)
(0<=i) & (i<100)      (BOOLEAN)
t.key = 0             (BOOLEAN)
k IN {i..j-1}         (BOOLEAN)
w[i].name <= "John"   (BOOLEAN)
t IS CenterTree       (BOOLEAN)
```

1.1.9 Anweisungen

Anweisungen bezeichnen Aktionen. Es gibt einfache und zusammengesetzte Anweisungen. Einfache Anweisungen enthalten keine Teile, die selbst Anweisungen sind. Dies sind die Zuweisung, der Prozeduraufruf, die Return- und die Exit-Anweisung. Zusammengesetzte Anweisungen enthalten Teile, die selbst Anweisungen sind. Sie dienen dazu, Folgen von Anweisungen sowie bedingte, ausgewählte und wiederholte Ausführung von Anweisungen auszudrücken. Eine Anweisung kann auch leer sein und bedeutet dann keine Aktion. Die leere Anweisung wurde hinzugefügt, um die Interpunktionsregeln in Anweisungsfolgen zu lockern.

```
Statement  = [  Assignment | ProcedureCall | IfStatement | CaseStatement
             |  WhileStatement | RepeatStatement |ForStatement | LoopStatement
             |  WithStatement | EXIT | RETURN [ Expression ] ].
```

1.1.9.1 Zuweisungen

Zuweisungen ersetzen den aktuellen Wert einer Variablen durch einen neuen Wert, der durch einen Ausdruck spezifiziert wird. Der Ausdruck muß mit der Variablen *zuweisungskompatibel* sein (siehe 1.1.12.1). Der Zuweisungsoperator wird ":=" geschrieben und ,,wird (zu)“ gesprochen.

```
Assignment = Designator ":=" Expression.
```

Wird ein Ausdruck e mit Typ T_e einer Variablen v mit Typ T_v zugewiesen, geschieht das folgende:

1. Wenn T_v und T_e Record-Typen sind, werden nur diejenigen Felder von T_e zugewiesen, die auch zu T_v gehören (*Projektion*); der dynamische Typ von v muß *derselbe* sein wie der statische Typ von v; er wird durch die Zuweisung nicht verändert.
2. Wenn T_v und T_e Zeigertypen sind, wird der dynamische Typ von v zum dynamischen Typ von e.
3. Wenn T_v ein Array-Typ `ARRAY` n `OF CHAR` ist und e eine Zeichenkette der Länge `m<n`, wird $v[i]$ zu $e[i]$ für $i=0..m-1$ und $v[m]$ wird zu `0X`.

Beispiele für Zuweisungen (siehe Beispiele in 1.1.7):

```
i := 0
p := i=j
x := i+1
k := Log2(i+j)
F := Log2                  (* siehe 1.1.10.1 *)
s := {2, 3, 5, 7, 11, 13}
a[i] := (x+y) * (x–y)
t.key := i
w[i+1].name := "John"
t := c
```

1.1.9.2 Prozeduraufrufe

Ein Prozeduraufruf aktiviert eine `[gewöhnliche]` Prozedur. Er kann eine Liste aktueller Parameter aufweisen, welche die korrespondierenden formalen Parameter der Prozedurdeklaration ersetzen (siehe 1.1.10). Die Zuordnung der Parameter erfolgt aufgrund ihrer Position in der aktuellen und formalen Parameterliste. Es gibt zwei Arten von Parametern: *Var-Parameter* und *Val-Parameter*.

Wenn ein formaler Parameter ein Var-Parameter ist, muß der entsprechende aktuelle Parameter ein Bezeichner sein, der für eine Variable steht. Falls er ein Element einer strukturierten Variablen bezeichnet, werden die Komponentenselektoren ausgewertet, wenn die Ersetzung des formalen durch den aktuellen Parameter erfolgt, d.h. vor der Ausführung der Prozedur. Wenn ein formaler Parameter ein Val-Parameter ist, muß der entsprechende aktuelle Parameter ein Ausdruck sein. Er wird vor der Ausführung der Prozedur ausgewertet und der resultierende Wert dem formalen Parameter zugewiesen (siehe 1.1.10.1).

ProcedureCall = Designator [ActualParameters].

Beispiele:

```
WriteInt(i*2+1)    (* siehe 1.1.10.1 *)
INC(w[k].count)
t.Insert("John")   (* siehe 1.1.11 *)
```

1.1.9.3 Anweisungsfolgen

Anweisungsfolgen stellen eine Folge von Aktionen dar. Sie bestehen aus den einzelnen Anweisungen, getrennt durch Semikolons.

StatementSequence = Statement { ";" Statement }.

1.1.9.4 If-Anweisungen

```
IfStatement =  IF Expression THEN StatementSequence
               { ELSIF Expression THEN StatementSequence }
               [ ELSE StatementSequence ]
               END.
```

If-Anweisungen drücken die bedingte Ausführung von Anweisungsfolgen aus. Der Boolesche Ausdruck vor einer Anweisungsfolge wird *Bedingung* genannt. Die Bedingungen werden in der Reihenfolge ihres Auftretens ausgewertet, bis eine von ihnen TRUE ergibt, worauf die zu ihr gehörende Anweisungsfolge ausgeführt wird. Wenn keine Bedingung erfüllt ist, wird die Anweisungsfolge nach dem ELSE-Symbol ausgeführt, falls dieses existiert.

Beispiel:

```
IF (ch >= "A") & (ch <= "Z") THEN ReadIdentifier
ELSIF (ch >= "0") & (ch <= "9") THEN ReadNumber
ELSIF (ch = "'") OR (ch = '"') THEN ReadString
ELSE SpecialCharacter
END
```

1.1.9.5 Case-Anweisungen

Case-Anweisungen beschreiben die Auswahl und Ausführung einer Anweisungsfolge in Abhängigkeit vom Wert eines Ausdrucks. Nach Auswertung des Case-Ausdrucks wird diejenige Anweisungsfolge ausgeführt, deren Case-Markenliste den berechneten Wert enthält. Der Case-Ausdruck muß entweder von einem *ganzzahligen* Typ sein, der die Typen aller Case-Marken *einschließt*, oder sowohl der Case-Ausdruck als auch die Case-Marken müssen vom Typ CHAR sein. Case-Marken sind Konstanten und kein Wert darf mehr als einmal auftreten. [Ein Case-Markenbereich *a..b* steht für alle Werte aus dem Intervall [*a,b*], wobei *a* kleiner oder gleich *b* sein muß.] Wenn der Wert des Ausdrucks in keiner Case-Marke auftritt, wird die Anweisungsfolge nach dem ELSE-Symbol ausgeführt, falls dieses vorhanden ist, ansonsten wird das Programm abgebrochen.

```
CaseStatement = CASE Expression OF Case { "|" Case }
                [ ELSE StatementSequence ] END.
Case          = [ CaseLabelList ":" StatementSequence ].
CaseLabelList = CaseLabels { "," CaseLabels }.
CaseLabels    = ConstExpression [ ".." ConstExpression ].
```

Beispiel:

```
CASE ch OF
  "A".."Z": ReadIdentifier
| "0".."9": ReadNumber
| "'",'"': ReadString
ELSE SpecialCharacter
END
```

1.1.9.6 While-Anweisungen

While-Anweisungen beschreiben die wiederholte Ausführung einer Anweisungsfolge, solange der Boolesche Ausdruck (ihre Bedingung) TRUE liefert. Die Bedingung wird vor jeder Ausführung der Anweisungsfolge geprüft.

```
WhileStatement = WHILE Expression DO StatementSequence END.
```

Beispiele:

```
WHILE i > 0 DO i := i DIV 2; k := k + 1 END
WHILE (t # NIL) & (t.key # i) DO t := t.left END
```

1.1.9.7 Repeat-Anweisungen

Eine Repeat-Anweisung beschreibt die wiederholte Ausführung einer Anweisungsfolge bis die durch einen Booleschen Ausdruck angegebene Bedingung erfüllt ist. Die Anweisungsfolge wird mindestens einmal ausgeführt.

```
RepeatStatement = REPEAT StatementSequence UNTIL Expression.
```

1.1.9.8 For-Anweisungen

Eine For-Anweisung beschreibt eine feste Anzahl von Ausführungen einer Anweisungsfolge, wobei eine *ganzzahlige* Variable (die sogenannte *Kontrollvariable*) bei jedem Durchlauf einen neuen Wert annimmt.

```
ForStatement = FOR ident ":=" Expression TO Expression
               [ BY ConstExpression ] DO StatementSequence END.
```

Die Anweisung

```
FOR v := beg TO end BY step DO statements END
```

ist gleichbedeutend mit

```
temp := end; v := beg;
IF step > 0 THEN
  WHILE v <= temp DO statements; v := v + step END
ELSE
  WHILE v >= temp DO statements; v := v + step END
END
```

temp hat den *gleichen* Typ wie *v*. *step* muß ein konstanter Ausdruck sein, dessen Wert nicht Null sein darf. Fehlt die Angabe der Schrittweite *step*, wird für sie der Wert 1 angenommen.

Beispiele:

```
FOR i := 0 TO 79 DO k := k + a[i] END
FOR i := 79 TO 1 BY –1 DO a[i] := a[i–1] END
```

1.1.9.9 Loop-Anweisungen

Eine Loop-Anweisung beschreibt die wiederholte Ausführung einer Anweisungsfolge. Sie terminiert durch Ausführung einer Exit-Anweisung innerhalb dieser Anweisungsfolge (siehe 1.1.9.10).

```
LoopStatement = LOOP StatementSequence END.
```

Beispiel:

```
LOOP
  ReadInt(i);
  IF i < 0 THEN EXIT END;
  WriteInt(i)
END
```

Loop-Anweisungen sind nützlich, um Schleifen mit mehreren Austrittspunkten darzustellen oder Schleifen, in denen sich die Austrittsbedingung in der Mitte der wiederholten Anweisungsfolge befindet.

1.1.9.10 Return- und Exit-Anweisungen

Eine Return-Anweisung bezeichnet die Termination einer Prozedur. Sie wird durch das Symbol RETURN ausgedrückt, gefolgt von einem Ausdruck, falls die Prozedur eine Funktionsprozedur ist. Der Typ des Ausdrucks muß mit dem im Prozedurkopf angegebenen Resultatstyp (siehe 1.1.10) *zuweisungskompatibel* sein (siehe 1.1.12.1).

Funktionsprozeduren erfordern das Vorhandensein einer Return-Anweisung, die den Resultatswert angibt. In gewöhnlichen Prozeduren wird eine Return-Anweisung durch das Ende des Prozedurrumpfs impliziert. Jede explizite Return-Anweisung ist daher eine zusätzliche (möglicherweise durch eine Ausnahme bedingte) Terminationsstelle.

Eine Exit-Anweisung wird durch das Symbol EXIT ausgedrückt. Sie bewirkt die Termination der unmittelbar umgebenden Loop-Anweisung und die Fortsetzung mit der der Loop-Anweisung folgenden Anweisung. Exit-Anweisungen sind kontextuell, wenn auch nicht syntaktisch, mit der Loop-Anweisung verbunden, die sie enthält.

1.1.9.11 With-Anweisungen

With-Anweisungen führen eine Anweisungsfolge in Abhängigkeit vom Resultat eines Typtests aus und wenden eine Typzusicherung auf jedes Vorkommen der getesteten Variable in der Anweisungsfolge an.

```
WithStatement =  WITH Guard DO StatementSequence
                 { "|" Guard DO StatementSequence }
                 [ ELSE StatementSequence ] END.
Guard         =  Qualident ":" Qualident.
```

Wenn *v* ein Var-Parameter mit Record-Typ oder eine Zeigervariable ist und wenn ihr statischer Typ *T0* ist, hat die Anweisung

```
WITH v: T1 DO S1
| v: T2 DO S2
ELSE S3
END
```

folgende Bedeutung: falls *v* vom dynamischen Typ *T1* ist, wird die Anweisungsfolge *S1* ausgeführt, wobei *v* wie eine Variable mit statischem Typ *T1* behandelt wird; andernfalls, wenn *v* vom dynamischen Typ *T2* ist, wird *S2* ausgeführt, wobei *v* wie eine Variable mit statischem Typ *T2* behandelt wird; andernfalls wird *S3* ausgeführt. *T1* und *T2* müssen Erweiterungen von *T0* sein. Wenn kein Typtest zutrifft und ein Else-Zweig fehlt, wird das Programm abgebrochen.

Beispiel:

```
WITH t: CenterTree DO i := t.width; c := t.subnode END
```

1.1.10 Prozedurdeklarationen

Eine Prozedurdeklaration besteht aus einem *Prozedurkopf* und einem *Prozedurrumpf*. Der Kopf definiert den Namen der Prozedur und ihre *formalen Parameter*, bei typgebundenen Prozeduren auch den Empfängerparameter. Der Rumpf enthält Deklarationen und Anweisungen. Der Prozedurname wird am Ende der Prozedurdeklaration wiederholt.

Es gibt zwei Arten von Prozeduren: *gewöhnliche Prozeduren* und *Funktionsprozeduren*. Letztere werden durch einen Funktionsbezeichner als Teil eines Ausdrucks aktiviert und liefern ein Resultat, das ein Operand des Ausdrucks ist. Gewöhnliche Prozeduren werden durch einen Prozeduraufruf aktiviert. Eine Prozedur ist eine Funktionsprozedur, wenn ihre formalen Parameter einen Resultatstyp angeben. Der Rumpf einer Funktionsprozedur muß eine Return-Anweisung enthalten, die ihr Resultat bestimmt.

Alle in einem Prozedurrumpf deklarierten Konstanten, Variablen, Typen und Prozeduren sind lokal zu dieser Prozedur. Da Prozeduren selbst als lokale Objekte deklariert werden können, dürfen Prozedurdeklarationen geschachtelt werden. Die Aktivierung einer Prozedur innerhalb ihrer Deklaration impliziert einen rekursiven Aufruf.

Objekte, die in den umgebenden Blöcken der Prozedur deklariert sind, sind auch in den Teilen der Prozedur sichtbar, in denen sie nicht durch lokal deklarierte Objekte mit gleichem Namen überdeckt werden.

```
ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading     = PROCEDURE [ Receiver ] IdentDef [ FormalParameters ].
ProcedureBody        = DeclarationSequence [ BEGIN StatementSequence ] END.
DeclarationSequence  = { CONST { ConstDeclaration ";" }
                       | TYPE { TypeDeclaration ";" }
                       | VAR { VariableDeclaration ";" } }
                       { ProcedureDeclaration ";" | ForwardDeclaration ";" }.
ForwardDeclaration   = PROCEDURE "^" [ Receiver ] IdentDef [ FormalParameters ].
```

Wird in einer Prozedurdeklaration ein *Empfängerparameter* deklariert, bedeutet das, daß die Prozedur an einen Typ gebunden ist (siehe 1.1.10.2). Eine *Vorausdeklaration* erlaubt, sich auf eine Prozedur zu beziehen, die erst später im Text deklariert wird. Die formalen Parameterlisten der Vorausdeklaration und der eigentlichen Deklaration müssen *übereinstimmen* (siehe

1.1.12.1). [Eine Prozedur wird exportiert, falls ihre Vorausdeklaration und/oder ihre eigentliche Deklaration mit einer Exportmarke versehen ist.]

1.1.10.1 Formale Parameter

Formale Parameter sind Namen, die in der formalen Parameterliste einer Prozedur deklariert sind. Sie entsprechen den beim Prozeduraufruf angegebenen aktuellen Parametern. Die Zuordnung zwischen formalen und aktuellen Parametern findet beim Prozeduraufruf statt. Es gibt zwei Arten von Parametern, *Val-* und *Var-Parameter*, die in der formalen Parameterliste durch das Fehlen oder Vorhandensein des Schlüsselworts VAR unterschieden werden. Val-Parameter sind lokale Variablen, denen der Wert des entsprechenden aktuellen Parameters als Anfangswert zugewiesen wird. Var-Parameter entsprechen aktuellen Parametern, die Variablen sind, und sie stehen für diese Variablen. Der Sichtbarkeitsbereich eines formalen Parameters erstreckt sich [textuell] von [der Stelle] seiner [Namensnennung bei der] Deklaration bis zum Ende des Prozedurblocks, in dem er deklariert ist. Eine parameterlose Funktionsprozedur muß eine leere Parameterliste aufweisen. Sie muß durch einen Funktionsbezeichner aufgerufen werden, dessen aktuelle Parameterliste ebenfalls leer ist. Der Resultatstyp einer Funktionsprozedur kann weder ein Record noch ein Array sein.

```
FormalParameters =  "(" [ FPSection { ";" FPSection }] ")" [ ":" Qualident ].
FPSection        =  [ VAR ] ident { "," ident } ":" Type.
```

Sei T_f der Typ eines formalen Parameters f (kein offenes Array) und T_a der Typ des entsprechenden aktuellen Parameters a. Bei Var-Parametern muß T_a *derselbe* Typ wie T_f sein, oder T_f muß ein Record-Typ sein und T_a eine Erweiterung von T_f. Bei Val-Parametern muß a *zuweisungskompatibel* mit f sein (siehe 1.1.12.1).

Wenn T_f ein offenes Array ist, muß a *array-kompatibel* mit f sein (siehe 1.1.12.1). Die Längen von f werden aus a übernommen.

Beispiele für Prozedurdeklarationen:

```
PROCEDURE ReadInt (VAR x: INTEGER);
  VAR i: INTEGER; ch: CHAR;
BEGIN
  i := 0; Read(ch);
  WHILE ("0" <= ch) & (ch <= "9") DO
    i := 10*i + (ORD(ch) – ORD("0")); Read(ch)
  END;
  x := i
END ReadInt;

PROCEDURE WriteInt (x: INTEGER);   (* 0 <= x < 100000 *)
  VAR i: INTEGER; buf: ARRAY 5 OF INTEGER;
BEGIN
  i := 0;
  REPEAT buf[i] := x MOD 10; x := x DIV 10; INC(i) UNTIL x = 0;
  REPEAT DEC(i); Write(CHR(buf[i] + ORD("0"))) UNTIL i = 0
END WriteInt;

PROCEDURE WriteString (s: ARRAY OF CHAR);
  VAR i: INTEGER;
BEGIN
  i := 0;
  WHILE (i < LEN(s)) & (s[i] # 0X) DO Write(s[i]); INC(i) END
END WriteString;
```

```
PROCEDURE log2 (x: INTEGER): INTEGER;
  VAR y: INTEGER; (* assume x > 0 *)
BEGIN
  y := 0; WHILE x > 1 DO x := x DIV 2; INC(y) END;
  RETURN y
END log2;
```

1.1.10.2 Typgebundene Prozeduren

Global deklarierte Prozeduren können einem im selben Modul deklarierten Record-Typ zugeordnet werden. Die Prozeduren werden als an den Record-Typ *gebunden* bezeichnet. Die Bindung wird durch den Typ des *Empfängers* im Kopf einer Prozedurdeklaration ausgedrückt. Der Empfänger kann entweder ein Var-Parameter mit Record-Typ *T* oder ein Val-Parameter mit Zeigertyp `POINTER TO` *T* sein (wobei *T* ein Record-Typ ist). Die Prozedur ist an den Typ *T* gebunden und wird als lokal zu diesem angesehen.

```
ProcedureHeading =  PROCEDURE [ Receiver ] IdentDef [ FormalParameters ].
Receiver         =  "(" [ VAR ] ident ":" ident ")".
```

Wenn eine Prozedur *P* an einen Typ *T0* gebunden ist, ist sie implizit auch an jeden Typ *T1* gebunden, der eine Erweiterung von *T0* ist. Jedoch darf eine Prozedur *P'* (mit dem gleichen Namen wie *P*) explizit an *T1* gebunden werden, wobei sie dann die Bindung von *P* überschreibt. *P'* wird als *Redefinition* von *P* für *T1* angesehen [und kann an einer Stelle deklariert werden, an der *P* noch unbekannt ist]. Die formalen Parameter von *P* und *P'* müssen *übereinstimmen* (siehe 1.1.12.1). Werden *P* und *T1* exportiert (siehe 1.1.4), muß auch *P'* exportiert werden.

Wenn *v* ein Bezeichner ist und *P* eine typgebundene Prozedur, bezeichnet *v.P* diejenige Prozedur *P*, die an den dynamischen Typ von *v* gebunden ist. Das kann eine andere Prozedur sein als die an den statischen Typ von *v* gebundene. *v* wird gemäß den in Kapitel 1.1.10.1 beschriebenen Parameterübergaberegeln an den Empfänger von *P* übergeben.

Wenn *r* ein Empfängerparameter deklariert mit Typ *T* ist, bezeichnet *r.P^* die (redefinierte) Prozedur *P*, die an den Basistyp von *T* gebunden ist.

In einer Vorausdeklaration einer typgebundenen Prozedur muß der Empfängerparameter *denselben* Typ haben wie in der eigentlichen Prozedurdeklaration. Die formalen Parameterlisten beider Deklarationen müssen *übereinstimmen* (siehe 1.1.12.1).

Beispiele:

```
PROCEDURE (t: Tree) Insert (node: Tree);
  VAR p, father: Tree;
BEGIN
  p := t;
  REPEAT father := p;
    IF node.key = p.key THEN RETURN END;
    IF node.key < p.key THEN p := p.left ELSE p := p.right END
  UNTIL p = NIL;
  IF node.key < father.key THEN father.left := node
  ELSE father.right := node
  END;
  node.left := NIL; node.right := NIL
END Insert;
```

```
PROCEDURE (t: CenterTree) Insert (node: Tree); (* redefinition *)
BEGIN
  WriteInt(node(CenterTree).width);
  t.Insert^(node) (* calls the Insert procedure bound to Tree *)
END Insert;
```

1.1.10.3 Vordeklarierte Prozeduren

Die folgenden Tabellen beschreiben vordeklarierte Prozeduren. Einige davon sind generisch, d.h. auf verschiedene Operandentypen anwendbar. *v* steht für eine Variable, *x* und *n* für Ausdrücke und *T* für einen Typ.

Funktionsprozeduren

Name	Argumenttyp	Resultatstyp	Bedeutung
ABS(*x*)	numerisch	Typ von *x*	Absolutwert
ASH(*x*,*n*)	*x*, *n*: ganzzahlig	LONGINT	arithmetische Shift-Operation ($x*2^n$)
CAP(*x*)	CHAR (Buchstabe)	CHAR	entsprechender Großbuchstabe
CHR(*x*)	ganzzahlig	CHAR	Zeichen mit Ordinalwert *x*
ENTIER(*x*)	reell	LONGINT	größte ganze Zahl nicht größer als *x*
LEN(*v*,*n*)	*v*: Array	LONGINT	Länge von *v* in der Dimension *n*
	n: ganzzahlige Konstante		(erste Dimension = 0)
LEN(*v*)	*v*: Array	LONGINT	äquivalent zu LEN(*v*,0)
LONG(*x*)	SHORTINT	INTEGER	Identität
	INTEGER	LONGINT	
	REAL	LONGREAL	
MAX(*T*)	*T* = Standardtyp	*T*	größter Wert des Typs *T*
	T = SET	INTEGER	größtes Element einer Menge
MIN(*T*)	*T* = Standardtyp	*T*	kleinster Wert des Typs *T*
	T = SET	INTEGER	0
ODD(*x*)	ganzzahlig	BOOLEAN	*x* MOD 2 = 1
ORD(*x*)	CHAR	INTEGER	Ordinalwert von *x*
SHORT(*x*)	LONGINT	INTEGER	Identität
	INTEGER	SHORTINT	Identität
	LONGREAL	REAL	Identität (mögl. Genauigkeitsverlust)
SIZE(*T*)	beliebiger Typ	ganzzahlig	Länge von *T* in Bytes

Gewöhnliche Prozeduren

Name	Argumenttyp	Bedeutung
ASSERT(*x*)	*x*: Boolescher Ausdruck	Programmabbruch, falls *x* = FALSE
ASSERT(*x*,*n*)	*x*: Boolescher Ausdruck	Programmabbruch, falls *x* = FALSE
	n: ganzzahlige Konstante	
COPY(*x*,*v*)	*x*: Zeichen-Array oder Zeichenkette	*v* := *x*
	v: Zeichen-Array	
DEC(*v*)	ganzzahlig	*v* := *v* - 1
DEC(*v*,*n*)	*v*, *n*: ganzzahlig	*v* := *v* - *n*
EXCL(*v*,*x*)	*v*: SET; *x*: ganzzahlig	*v* := *v* - {*x*}
HALT(*n*)	ganzzahlige Konstante	Programmabbruch
INC(*v*)	ganzzahlig	*v* := *v* + 1
INC(*v*,*n*)	*v*, *n*: ganzzahlig	*v* := *v* + *n*
INCL(*v*,*x*)	*v*: SET; *x*: ganzzahlig	*v* := *v* + {*x*}
NEW(*v*)	Zeiger auf Record oder Array fester Länge	Anlegen von *v*^
NEW(*v*,x_0,…,x_n)	*v*: Zeiger auf offenes Array; x_i: ganzzahlig	Anlegen von *v*^ mit Längen x_0…x_n

COPY erlaubt die Zuweisung einer Zeichenkette oder eines Zeichen-Arrays, welche ein abschließendes 0X enthalten, an ein weiteres Zeichen-Array. Falls nötig wird der zugewiesene Wert auf die Länge des Zieloperanden minus eins gekürzt. Der Zieloperand wird immer 0X als Abschluß enthalten. Bei ASSERT(*x*,*n*) und HALT(*n*) wird die Interpretation von *n* dem zugrundeliegenden Betriebssystem überlassen.

1.1.11 Module

Ein Modul ist eine Sammlung von Deklarationen von Konstanten, Typen, Variablen und Prozeduren, zusammen mit einer Anweisungsfolge, die vor allem dazu dient, den Variablen Anfangswerte zu geben. Ein Modul bildet einen Text, der als Einheit übersetzbar ist. Dieser Text muß in einer eigenen Datei vorliegen, deren Name (ohne Suffix) mit dem Modulnamen identisch ist. Der Modulname wird am Ende des Moduls wiederholt.

```
Module      =  MODULE ident ";" [ ImportList ] DeclarationSequence
               [ BEGIN StatementSequence ] END ident ".".
ImportList  =  IMPORT Import { "," Import } ";".
Import      =  [ ident ":=" ] ident.
```

Die Importliste spezifiziert die Namen der importierten Module. Wenn ein Modul *A* von einem Modul *M* importiert wird und *A* einen Namen *x* exportiert, wird *x* in *M* als *A.x* angesprochen. Wird *A* in der Form *B:=A* importiert, muß *x* als *B.x* angesprochen werden. Dies erlaubt die Verwendung von kurzen Alias-Namen in qualifizierten Namen. Ein Modul darf sich nicht selbst importieren. Namen, die exportiert werden sollen (d.h. die in anderen Modulen sichtbar sein sollen), müssen bei ihrer Deklaration mit einer Exportmarke versehen werden (siehe 1.1.4).

Die Anweisungsfolge nach dem Symbol BEGIN wird ausgeführt, wenn das Modul geladen wird. Vorher werden alle importierten Module geladen. Daraus folgt, daß zyklischer Import von Modulen verboten ist. Parameterlose, exportierte Prozeduren heißen *Kommandos* und können vom System aus aufgerufen werden (siehe 1.1.12.4).

```
MODULE Trees;
IMPORT Texts, Oberon;
(* exportiert: Tree, Node, Insert, Search, Write, Init *)
(* exportiert schreibgeschützt: Node.name *)

TYPE
  Tree* = POINTER TO Node;
  Node* = RECORD
    name-: POINTER TO ARRAY OF CHAR;
    left, right: Tree
  END;

VAR w: Texts.Writer;

PROCEDURE (t: Tree) Insert* (name: ARRAY OF CHAR);
  VAR p, father: Tree;
BEGIN
  p := t;
  REPEAT father := p;
    IF name = p.name^ THEN RETURN END;
    IF name < p.name^ THEN p := p.left ELSE p := p.right END
  UNTIL p = NIL;
  NEW(p); p.left := NIL; p.right := NIL;
  NEW(p.name, LEN(name)+1); COPY(name, p.name^);
  IF name < father.name^ THEN father.left := p ELSE father.right := p END
END Insert;
```

```
PROCEDURE (t: Tree) Search* (name: ARRAY OF CHAR): Tree;
  VAR p: Tree;
BEGIN
  p := t;
  WHILE (p # NIL) & (name # p.name^) DO
    IF name < p.name^ THEN p := p.left ELSE p := p.right END
  END;
  RETURN p
END Search;

PROCEDURE (t: Tree) Write*;
BEGIN
  IF t.left # NIL THEN t.left.Write END;
  Texts.WriteString(w, t.name^); Texts.WriteLn(w);
  Texts.Append(Oberon.Log, w.buf);
  IF t.right # NIL THEN t.right.Write END
END Write;

PROCEDURE Init* (t: Tree);
BEGIN
  NEW(t.name, 1); t.name[0] := 0X; t.left := NIL; t.right := NIL
END Init;

BEGIN
  Text.OpenWriter(w)
END Trees.
```

1.1.12 Anhänge zur Sprachdefinition

1.1.12.1 Begriffsdefinitionen

Ganzzahlige Typen	SHORTINT, INTEGER, LONGINT
Reelle Typen	REAL, LONGREAL
Numerische Typen	*Ganzzahlige* Typen und *reelle* Typen

Derselbe Typ Zwei Variablen a und b mit den Typen T_a und T_b haben *denselben* Typ, wenn

1. T_a und T_b durch den gleichen Typnamen bezeichnet werden, oder
2. T_a als zu T_b gleich deklariert ist in einer Typdeklaration der Form $T_a = T_b$, oder
3. a und b in der gleichen Namenliste einer Variablen-, formalen Parameter- oder Record-Felddeklaration auftreten und keine offenen Arrays sind.

Der gleiche Typ Zwei Typen T_a und T_b sind *gleich*, wenn

1. T_a und T_b *derselbe* Typ ist, oder
2. T_a und T_b offene Array-Typen sind, deren Elementtypen *gleich* sind, oder
3. T_a und T_b Prozedurtypen sind, deren formale Parameterlisten *übereinstimmen*.

Typeinschluß *Numerische* Typen *schließen* die Werte kleinerer *numerischer* Typen gemäß folgender Hierarchie *ein*:
LONGREAL ⊇ REAL ⊇ LONGINT ⊇ INTEGER ⊇ SHORTINT

Typerweiterung (Basistyp) In einer Typdeklaration T_b=RECORD(T_a)…END ist T_b eine *direkte Erweiterung* von T_a und T_a ist ein *direkter Basistyp* von T_b.
Ein Typ T_b ist eine *Erweiterung* eines Typs T_a (T_a ist ein *Basistyp* von T_b), wenn

1. T_a und T_b *derselbe* Typ ist, oder
2. T_b eine *direkte Erweiterung* einer *Erweiterung* von T_a ist.

Wenn P_a = POINTER TO T_a und P_b = POINTER TO T_b, so ist P_b eine Erweiterung von P_a (P_a ist ein Basistyp von P_b), falls T_b eine Erweiterung von T_a ist.

Zuweisungskompatibilität Ein Ausdruck e vom Typ T_e ist *zuweisungskompatibel* mit einer Variablen v vom Typ T_v, wenn eine der folgenden Bedingungen gilt:

1. T_e und T_v ist *derselbe* Typ.
2. T_e und T_v sind *numerische* Typen und T_v *schließt* T_e *ein*.
3. T_e und T_v sind Record-Typen, T_e eine *Erweiterung* von T_v und der dynamische Typ von v ist T_v.
4. T_e und T_v sind Zeigertypen und T_e ist eine *Erweiterung* von T_v.
5. T_v ist ein Zeiger- oder ein Prozedurtyp und e ist NIL.
6. T_v ist ARRAY n OF CHAR, e ist eine Zeichenkettenkonstante mit m Zeichen und $m < n$.
7. T_v ist ein Prozedurtyp und e ist der Name einer [globalen] Prozedur, deren formale Parameter mit denen von T_v *übereinstimmen*.

Array-Kompatibilität

Ein aktueller Parameter *a* vom Typ *Ta* ist mit einem formalen Parameter *f* vom Typ *Tf* *array-kompatibel*, wenn

1. *Tf* und *Ta* *derselbe* Typ ist, oder
2. *Tf* ein offenes Array ist, *Ta* ein beliebiges Array ist und ihre Elementtypen *array-kompatibel* sind, oder
3. *Tf* ein ARRAY OF CHAR und *a* eine Zeichenkette ist.

Ausdrucks-kompatibilität

Die Operandentypen eines gegebenen Operators sind *ausdrucks-kompatibel*, wenn sie folgender Tabelle entsprechen (die auch den Resultatstyp des Ausdrucks zeigt). Zeichen-Arrays, die verglichen werden sollen, müssen 0X als Abschluß enthalten. Der Typ *T1* muß eine *Erweiterung* von Typ *T0* sein.

Operator	Erster Operand	Zweiter Operand	Resultatstyp
+ - *	numerisch	numerisch	kleinster numerischer Typ, der beide Operandentypen einschließt.
/	numerisch	numerisch	kleinster reeller Typ, der beide Operandentypen einschließt.
+ - * /	SET	SET	SET
DIV MOD	ganzzahlig	ganzzahlig	kleinster ganzzahliger Typ, der beide Operandentypen einschließt.
OR & ~	BOOLEAN	BOOLEAN	BOOLEAN
= # < <= > >=	numerisch	numerisch	BOOLEAN
	CHAR	CHAR	BOOLEAN
	Zeichen-Array, -kette	Zeichen-Array, -kette	BOOLEAN
= #	BOOLEAN	BOOLEAN	BOOLEAN
	SET	SET	BOOLEAN
	NIL, Zeigertyp *T0* oder *T1*	NIL, Zeigertyp *T0* oder *T1*	BOOLEAN
	Prozedurtyp *T*, NIL	Prozedurtyp *T*, NIL	BOOLEAN
IN	ganzzahlig	SET	BOOLEAN
IS	Typ *T0*	Typ *T1*	BOOLEAN

Übereinstimmende Parameterlisten

Zwei formale Parameterlisten *stimmen überein*, falls

1. sie dieselbe Anzahl Parameter haben, und
2. sie *denselben* Funktionsresultatstyp oder keinen haben, und
3. Parameter an sich entsprechenden Positionen *gleiche* Typen haben, und
4. Parameter an sich entsprechenden Positionen beide entweder Val- oder Var-Parameter sind.

1.1.12.2 Syntax von Oberon-2

```
Module      = MODULE ident ";" [ImportList] DeclSeq [BEGIN StatementSeq] END ident ".".
ImportList  = IMPORT [ident ":="] ident {"," [ident ":="] ident} ";".
DeclSeq     = { CONST {ConstDecl ";"} | TYPE {TypeDecl ";"} | VAR {VarDecl ";"}}
              {ProcDecl ";" | ForwardDecl ";"}.
ConstDecl   = IdentDef "=" ConstExpr.
TypeDecl    = IdentDef "=" Type.
VarDecl     = IdentList ":" Type.
ProcDecl    = PROCEDURE [Receiver] IdentDef [FormalPars] ";" DeclSeq
              [BEGIN StatementSeq] END ident.
ForwardDecl = PROCEDURE "^" [Receiver] IdentDef [FormalPars].
FormalPars  = "(" [FPSection {";" FPSection}] ")" [":" Qualident].
FPSection   = [VAR] ident {"," ident} ":" Type.
Receiver    = "(" [VAR] ident ":" ident ")".
Type        = Qualident
            | ARRAY [ConstExpr {"," ConstExpr}] OF Type
            | RECORD ["("Qualident")"] FieldList {";" FieldList} END
            | POINTER TO Type
            | PROCEDURE [FormalPars].
FieldList   = [IdentList ":" Type].
StatementSeq = Statement {";" Statement}.
Statement   = [ Designator ":=" Expr
              | Designator ["(" [ExprList ]")"]
              | IF Expr THEN StatementSeq {ELSIF Expr THEN StatementSeq}
                [ELSE StatementSeq] END
              | CASE Expr OF Case {"|" Case} [ELSE StatementSeq] END
              | WHILE Expr DO StatementSeq END
              | REPEAT StatementSeq UNTIL Expr
              | FOR ident ":=" Expr TO Expr [BY ConstExpr] DO StatementSeq END
              | LOOP StatementSeq END
              | WITH Guard DO StatementSeq {"|" Guard DO StatementSeq}
                [ELSE StatementSeq] END
              | EXIT
              | RETURN [Expr]
              ].
Case        = [CaseLabels {"," CaseLabels} ":" StatementSeq].
CaseLabels  = ConstExpr [".." ConstExpr].
Guard       = Qualident ":" Qualident.
ConstExpr   = Expr.
Expr        = SimpleExpr [Relation SimpleExpr].
SimpleExpr  = ["+" | "-"] Term {AddOp Term}.
Term        = Factor {MulOp Factor}.
Factor      = Designator ["(" [ExprList] ")"] | number | character | string | NIL | Set
            | "(" Expr ")" | "~" Factor.
Set         = "{" [Element {"," Element}] "}".
Element     = Expr [".." Expr].
Relation    = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
AddOp       = "+" | "-" | OR.
MulOp       = "*" | "/" | DIV | MOD | "&".
Designator  = Qualident {"." ident | "[" ExprList "]" | "^" | "(" Qualident ")"}.
ExprList    = Expr {"," Expr}.
IdentList   = IdentDef {"," IdentDef}.
Qualident   = [ident "."] ident.
IdentDef    = ident ["*" | "-"].
```

1.1.12.3 Modul SYSTEM

Das Modul SYSTEM enthält Typen und Prozeduren, mit denen sich systemnahe Operationen für einen bestimmten Rechner oder ein bestimmtes Betriebssystem implementieren lassen. Darunter fallen Operationen zum Ansprechen peripherer Geräte oder zur Umgehung der Typregeln der Sprache. Es wird dringend empfohlen, SYSTEM nur in wenigen, systemnahen Modulen zu verwenden. Solche Module sind von Natur aus nichtportabel. Man erkennt sie daran, daß sie SYSTEM importieren. Die folgenden Angaben gelten für die Implementierung von Oberon-2 auf Ceres-Rechnern.

Das Modul SYSTEM exportiert einen Typ BYTE mit folgenden Eigenschaften: Variablen vom Typ CHAR oder SHORTINT können Variablen vom Typ BYTE zugewiesen werden. [Wenn ein formaler Var-Parameter vom Typ BYTE ist, darf der aktuelle Parameter vom Typ CHAR oder SHORTINT sein.] Wenn ein formaler Var-Parameter vom Typ ARRAY OF BYTE ist, darf der aktuelle Parameter von einem beliebigen Typ sein.

Ein anderer von SYSTEM exportierter Typ ist PTR. Einer Variablen vom Typ PTR kann eine Variable eines beliebigen Zeigertyps zugewiesen werden. Wenn ein formaler Var-Parameter vom Typ PTR ist, darf der aktuelle Parameter von einem beliebigen Zeigertyp sein.

Die von SYSTEM exportierten Prozeduren sind in den folgenden Tabellen beschrieben. Die meisten werden in einen einzigen Maschinenbefehl übersetzt, der anstelle ihres Aufrufs in das Programm eingefügt wird. *v* steht für eine Variable, *x*, *y*, *a* und *n* für Ausdrücke und *T* für einen Typ.

Funktionsprozeduren

Name	Argumenttyp	Resultatstyp	Bedeutung
ADR(*v*)	beliebig	LONGINT	Adresse der Variablen *v*
BIT(*a*,*n*)	*a*: LONGINT; *n*: ganzzahlig	BOOLEAN	Bit *n* von Mem[*a*]
CC(*n*)	ganzzahlige Konstante	BOOLEAN	Bedingungsbit *n* (0 ≤ *n* ≤ 15)
LSH(*x*,*n*)	*x*: ganzzahlig, CHAR, BYTE *n*: ganzzahlig	Typ von *x*	logisches Shift
ROT(*x*,*n*)	*x*: ganzzahlig, CHAR, BYTE *n*: ganzzahlig	Typ von *x*	Rotation
VAL(*T*,*x*)	*T*, *x*: beliebig	*T*	Typ von *x* wird in *T* umgewandelt

Gewöhnliche Prozeduren

Name	Argumenttyp	Bedeutung
GET(*a*,*v*)	*a*: LONGINT *v*: Standardtyp, Zeiger-, Prozedurtyp	*v* := Mem[*a*]
PUT(*a*,*x*)	*a*: LONGINT *x*: Standardtyp, Zeiger-, Prozedurtyp	Mem[*a*] := *x*
GETREG(*n*,*v*)	*n*: ganzzahlige Konstante *v*: Standardtyp, Zeiger-, Prozedurtyp	*v* := Register$_n$
PUTREG(*n*,*x*)	*n*: ganzzahlige Konstante *x*: Standardtyp, Zeiger-, Prozedurtyp	Register$_n$:= *x*
MOVE(a_0,a_1,*n*)	a_0, a_1: LONGINT; *n*: ganzzahlig	Mem[a_1…a_1+*n*−1] := Mem[a_0…a_0+*n*−1]
NEW(*v*,*n*)	*v*: beliebiger Zeiger *n*: ganzzahlig	Block mit *n* Bytes anlegen und seine Adresse *v* zuweisen

1.1.12.4 Oberon-Umgebung

Oberon-2 Programme laufen üblicherweise in einer Umgebung (hier Oberon-System genannt), die *Kommandoaktivierung*, *Speicherbereinigung* (garbage collection), *dynamisches Laden* von Modulen und gewisse *Laufzeitdatenstrukturen* zur Verfügung stellt. Obwohl diese Umgebung nicht Teil der Sprache ist, trägt sie wesentlich zur Mächtigkeit von Oberon-2 bei und wird zu einem gewissen Grad von der Sprache vorausgesetzt. Dieses Kapitel beschreibt die wesentlichen Eigenschaften einer typischen Oberon-Umgebung und gibt Hinweise für ihre Implementierung. Details können bei WIRTH und GUTKNECHT [1992], REISER [1991] und PFISTER, HEEB und TEMPL [1991] gefunden werden.

Kommandos

Ein Kommando ist eine parameterlose Prozedur *P*, die von einem Modul *M* exportiert wird. Es wird mit *M.P* bezeichnet und kann unter diesem Namen vom Benutzer im Dialog mit dem Betriebssystem aufgerufen werden. In Oberon ruft man Kommandos anstatt Programme oder Module auf. Das gibt dem Benutzer einen feineren Einfluß auf die Abläufe und erlaubt Programme mit mehreren Eintrittspunkten.

Beim Aufruf eines Kommandos *M.P* wird das Modul *M* geladen (falls es nicht bereits geladen ist) und die Prozedur *P* wird ausgeführt. Ist *P* beendet, wird die Kontrolle an das System zurückgegeben, *M* bleibt aber geladen. Alle globalen Variablen und alle Datenstrukturen, die über globale Zeiger in *M* erreicht werden können, behalten ihre Werte. Wird *P* oder ein anderes Kommando aus *M* erneut aufgerufen, kann es diese Wert weiterbenutzen.

Das folgende Modul zeigt, wie Kommandos benutzt werden. Es implementiert eine abstrakte Datenstruktur *Counter*, die einen Zähler kapselt und Kommandos zum Erhöhen und Drucken des Zählers zur Verfügung stellt.

```
MODULE Counter;
  IMPORT Texts, Oberon;

  VAR
    counter: LONGINT;
    w: Texts.Writer;

  PROCEDURE Add*; (* takes a numeric argument from the command line *)
    VAR s: Texts.Scanner;
  BEGIN
    Texts.OpenScanner(s, Oberon.Par.text, Oberon.Par.pos);
    Texts.Scan(s);
    IF s.class = Texts.Int THEN INC(counter, s.i) END
  END Add;

  PROCEDURE Write*;
  BEGIN
    Texts.WriteInt(w, counter, 5); Texts.WriteLn(w);
    Texts.Append(Oberon.Log, w.buf)
  END Write;

BEGIN counter := 0; Texts.OpenWriter(w)
END Counter.
```

Der Benutzer kann die folgenden zwei Kommandos aufrufen:

`Counter.Add` *n* addiert *n* zum Wert des Zählers

`Counter.Write` gibt den Wert des Zählers am Bildschirm aus

Da Kommandos parameterlos sind, müssen sie sich ihre Parameter über das Betriebssystem besorgen. Im allgemeinen können Kommandos ihre Parameter von beliebiger Stelle holen (zum Beispiel vom Text, der dem Kommando folgt, von der letzten Selektion oder von einem markierten Fenster). Das Kommando *`Add`* benutzt einen *Scanner* (einen vom Oberon-System zur Verfügung gestellten Datentyp), um die Zahl *`n`* im Text hinter dem Kommando zu lesen.

Wenn `Counter.Add` das erstemal aufgerufen wird, wird das Modul `Counter` geladen und sein Rumpf ausgeführt. Jeder Aufruf von `Counter.Add` *`n`* erhöht den Wert der Variablen `counter` um *`n`*. Jeder Aufruf von `Counter.Write` gibt den momentanen Wert von `counter` auf dem Bildschirm aus.

Da ein Modul auch nach Beendigung eines Kommandos geladen bleibt, stellt das Oberon-System ein Kommando zur Verfügung, das es erlaubt, ein Modul ausdrücklich zu entladen (zum Beispiel um die geladene Version durch eine neue zu ersetzen).

Dynamisches Laden von Modulen

Das Oberon-System erlaubt es, ein Kommando eines noch ungeladenen Moduls zu aktivieren, indem der Lader aufgerufen und ihm der Kommandoname als Zeichenkette übergeben wird. Das Modul wird dann dynamisch geladen und zu anderen, bereits geladenen Modulen gebunden; anschließend wird das gewünschte Kommando ausgeführt.

Dynamisches Laden ermöglicht es, ein Programm in einer Grundversion zu starten und es zur Laufzeit durch Hinzuladen neuer Module zu erweitern, falls sich die Notwendigkeit dafür ergibt.

Ein Modul *`M0`* kann das dynamische Laden eines Moduls *`M1`* bewirken, ohne daß *`M0`* dieses Modul zur Übersetzungszeit kennt und importiert. *`M1`* kann *`M0`* importieren und benutzen, umgekehrt muß *`M0`* aber nicht wissen, daß *`M1`* existiert. *`M1`* kann ein Modul sein, das lange nach *`M0`* entworfen und implementiert wurde.

Speicherbereinigung

In Oberon-2 wird die vordeklarierte Prozedur `NEW` benutzt, um Datenblöcke im freien Speicher anzulegen. Es gibt aber keine Möglichkeit, einen angelegten Block explizit wieder freizugeben. Stattdessen benutzt das Oberon-System eine automatische Speicherbereinigung (garbage collector), welche alle Blöcke findet, die nicht mehr benutzt werden, und wieder für die Speichervergabe verfügbar macht. Ein Block ist in Benutzung, solange er von einer globalen Zeigervariable durch eine Zeigerkette erreichbar ist. Eine Unterbrechung dieser Kette (z.B. das Setzen eines Zeigers auf `NIL`) gibt den Block für die Speicherbereinigung frei.

Die Speicherbereinigung befreit den Programmierer von der nichttrivialen Aufgabe der korrekten Freigabe von Datenstrukturen und hilft deswegen Fehler hierbei zu vermeiden. Sie benötigt Informationen über dynamische Datenstrukturen zur Laufzeit.

Browser

Die Schnittstelle eines Moduls (die Deklarationen der exportierten Objekte) wird aus dem Quelltext des Moduls durch einen sogenannten *Browser* gewonnen, welcher ein eigenes Werk-

zeug der Oberon-Umgebung ist. Aus dem Modul *Trees* (siehe 1.1.11) erzeugt der Browser zum Beispiel folgende Schnittstelle.

```
DEFINITION Trees;
  TYPE
    Tree = POINTER TO Node;
    Node = RECORD
      name: POINTER TO ARRAY OF CHAR;
      PROCEDURE (t: Tree) Insert (name: ARRAY OF CHAR);
      PROCEDURE (t: Tree) Search (name: ARRAY OF CHAR): Tree;
      PROCEDURE (t: Tree) Write;
    END;
  PROCEDURE Init (t: Tree);
END Trees.
```

Bei einem Record-Typ sammelt der Browser auch alle an diesen Typ gebundenen Prozeduren und zeigt ihre Deklaration in der Record-Deklaration an.

Datenstrukturen zur Laufzeit

Zur Laufzeit müssen gewisse Informationen über Records vorhanden sein: Für Typtests und Typzusicherungen wird der dynamische Typ von Record-Variablen benötigt. Für den Aufruf typgebundener Prozeduren wird eine Tabelle mit den Adressen dieser Prozeduren benötigt. Schließlich erfordert die Speicherbereinigung Informationen über die Position von Zeigern in dynamisch erzeugten Records. Alle diese Informationen werden in sogenannten *Typdeskriptoren* gespeichert, von denen es einen für jeden Record-Typ gibt.

Der dynamische Typ eines Records entspricht der Adresse seines Typdeskriptors. Bei dynamisch erzeugten Records wird diese Adresse in einer sogenannten *Typmarke* gespeichert, die vor den eigentlichen Record-Feldern liegt und für den Programmierer unsichtbar ist. Wenn *t* eine Variable vom Typ *CenterTree* ist (siehe Beispiel in 1.1.6) zeigt Abbildung 1 eine mögliche Implementierung der Datenstrukturen zur Laufzeit.

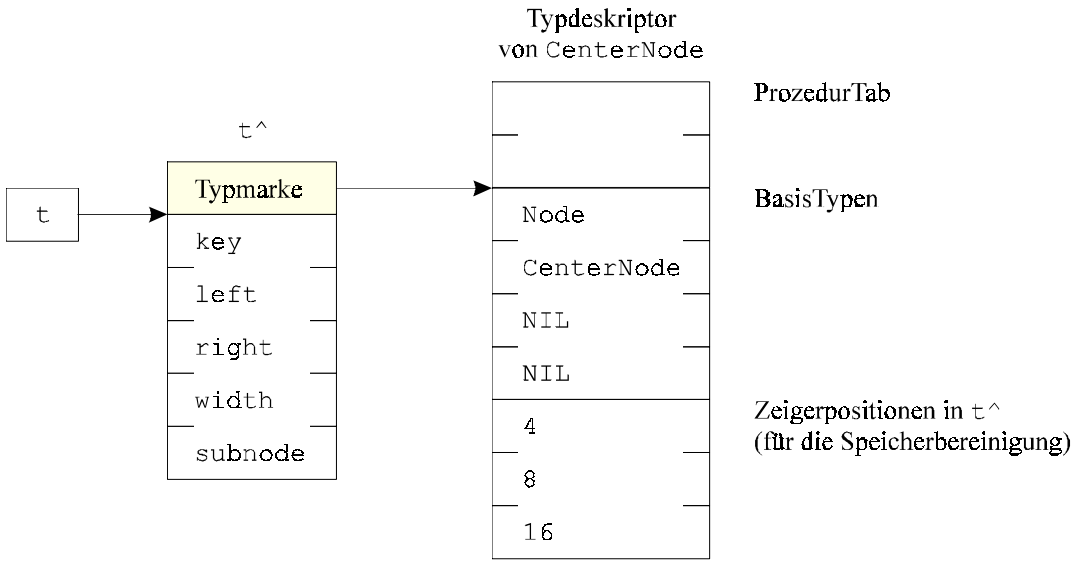

Abb. 1: Eine Variable t vom Typ CenterTree, das Record t^ vom Typ CenterNode, auf das sie zeigt und dessen Typdeskriptor.

Da sowohl die Tabelle der Prozeduradressen wie auch die der Zeigerpositionen eine feste Position innerhalb des Typdeskriptors haben müssen und da beide wachsen können, wenn der

Typ erweitert wird und weitere Prozeduren und Zeiger hinzugefügt werden, sind die Tabellen an den entgegengesetzten Enden des Typdeskriptors plaziert und wachsen in unterschiedliche Richtungen.

Eine typgebundene Prozedur *t.P* wird mit der Adresse

t^.Typmarke^.ProzedurTab[Index$_P$]

aufgerufen. Der Prozedurtabellenindex jeder typgebundenen Prozedur ist zur Übersetzungszeit bekannt.

Ein Typtest *v IS T* wird übersetzt als

v^.Typmarke^.BasisTypen[Erweiterungsstufe$_T$] = Typdeskriptoradresse$_T$

Sowohl die Erweiterungsstufe eines Record-Typs *T* als auch die Adresse seines Typdeskriptors sind zur Übersetzungzeit bekannt. Die Erweiterungsstufe von *Node* ist zum Beispiel 0 (*Node* hat keinen Basistyp), und die Erweiterungsstufe von *CenterNode* ist 1.

1.2 Anmerkungen zur Sprachdefinition

Die in der Sprachdefinition vorgenommenen Ergänzungen und Präzisierungen entstanden größtenteils nach ausführlicher elektronischer Korrespondenz mit dem Mitautor des Originalreports, Herrn Prof. Dr. Hanspeter Mössenböck, sowie anderen Angehörigen des Departement Informatik der Eidgenössischen Technischen Hochschule Zürich. Daneben dienten die folgenden von diesem Institut erstellten Oberon(-2) Übersetzer (als Bestandteil des Oberon-Systems) als Vergleichsimplementierungen.

Oberon-System	*Übersetzer*
Oberon (TM) System 3 Version 1.5 [for MS-DOS]	NW 1.8.91 / ARD 4.94
Oberon for Windows™ V4.0-1.31 on Windows 3.10	iOP2 RC/NM V1.3 23.6.94
Linux Oberon™ System 3 Version 1.5	iOP2 RC/NM V1.3 23.6.94

Die Übersetzer der beiden letztgenannten Portierungen des Oberon-Systems sind Oberon-2 Übersetzer und stützen sich auf die gleichen Quellen. Ihr Verhalten bezüglich der vorgenommenen Vergleiche ist deswegen gleich. Demgegenüber besitzt das erstgenannte Oberon-System lediglich einen Oberon Übersetzer (mit einigen kleinen Erweiterungen der Sprache, jedoch ohne typgebundene Prozeduren). Aus diesem Grund spielt sein Verhalten für den Vergleich nur eine untergeordnete Rolle.

Zusätzlich sind die ergänzten Passagen die Konsequenz aus der Auffassung, daß das Verhalten eines Übersetzers (vor allem in Grenzfällen) aus dem Dokument, welches die Sprache definiert, hervorgehen sollte.

Zu 1.1.3 Vokabular und Repräsentation, 4. *Zeichenketten*, S.4:
Die Verwendung des Zeichens `0X` zum Abschluß eines Zeichen-Arrays läßt diese Ergänzung ratsam erscheinen. Zusätzlich steht sie im Einklang mit der Definition der Zuweisung einer Zeichenkette an ein Zeichen-Array (siehe S.13), da sonst das Zeichen-Array, dem diese Zeichenkette zugewiesen wird, im ersten und im zweiten Element `0X` enthalten müßte.

Zu 1.1.4 Deklarationen und Sichtbarkeitsbereichsregeln S.5
und 1.1.10.1 Formale Parameter, S.18:
Die vorgenommene Präzisierung des Bereichsbeginns, ab dem ein Objekt sichtbar ist, ist notwendig, um beispielsweise die Legalität der Parameter- bzw. Variablendeklaration im Fragment

```
TYPE T1 = INTEGER; T2 = REAL;
PROCEDURE P(T1: T1);
   VAR T2: T2;
…
```

entscheiden zu können.

Zu 1.1.4 Deklarationen und Sichtbarkeitsbereichsregeln (2.), S.5:
Wegen der vorstehend beschriebenen Ergänzung bezüglich des Sichtbarkeitsbeginns eines Objekts mußte eine Beschränkung des Objektbezugs bei der Objektdeklaration eingeführt werden, um z.B. Deklarationen wie

```
CONST N=N+1;
VAR v: ARRAY 2,LEN(v,0) OF CHAR;
```

ausschließen zu können. Insbesondere verbietet die Originalsprachdefinition keine rekursive Bezugnahme über `LEN`. Für diesen Fall liefern die Oberon-2 Übersetzer eine Fehlermel-

dung (,,LEN not applied to array“) und der Oberon Übersetzer gerät in eine Ausnahmebehandlung (trap).

Zu 1.1.10 Prozedurdeklarationen, S.17:

Diese Anmerkung bezieht sich nicht auf eine Ergänzung der Sprachdefinition, sondern auf die Historie der Passage, welche die Gültigkeit globaler Objekte innerhalb von Prozeduren definiert.

In der diesbezüglichen Definition für Modula-2 bei WIRTH [1985] (Report on The Programming Language Modula-2, 10. Procedure declarations, S.160) heißt es: ,,In addition to its formal parameters and local objects, also the objects declared in the environment of the procedure are known and accessible in the procedure (with the exception of those objects that have the same name as objects declared locally).“ Das bedeutet, daß innerhalb einer Prozedur, auch textuell vor der Deklaration eines lokalen Objekts, ein Objekt gleichen Namens überdeckt ist. Dies steht in Übereinstimmung mit den allgemeinen Sichtbarkeitsregeln von Modula-2.

Die besagte Passage wurde sinngemäß in die Sprachdefinition von Oberon durch REISER und WIRTH [1992] (A.10 Procedure declarations, S.299) übernommen und lautet dort: ,,In addition to its formal parameters and locally declared objects, the objects declared in the environment of the procedure are also visible in the procedure (with the exception of those objects that have the same name as an object declared locally).“ Zusammen mit den anderen für Oberon geltenden Sichtbarkeitsregeln bedeutet dies jedoch, daß innerhalb einer Prozedur, textuell vor der Deklaration eines lokalen Objekts, ein globales Objekt gleichen Namens nicht mehr sichtbar und das lokale Objekt noch nicht sichtbar ist.

Während in einer früheren Version der Sprachdefinition von Oberon-2 von MÖSSENBÖCK [1992] dieser Absatz des Oberon Reports wörtlich übernommen wurde, heißt es in der aktuellen Version der Oberon-2 Sprachdefinition bei MÖSSENBÖCK [1993]: ,,Objects declared in the environment of the procedure are also visible in those parts of the procedure in which they are not concealed by a locally declared object with the same name.“ Dies bedeutet jedoch eine Erweiterung des Sichtbarkeitsbereichs eines globalen Objekts um den innerhalb einer Prozedur liegenden Bereich, der textuell vor der Deklaration eines lokalen Objekts gleichen Namens liegt. Alle Vergleichsimplementierungen handhaben die Sichtbarkeitsregeln demgemäß.

Zu 1.1.4 Deklarationen und Sichtbarkeitsbereichsregeln (,,Exportmarkierungen“), S.6
und 1.1.10 Prozedurdeklarationen, S.18:

Diese Präzisierungen bezüglich der Behandlung von Exportmarkierungen tragen der vorgenommen Implementierung Rechnung, da für diesen Sachverhalt das Original keine Aussagen enthält. Während die Oberon-2 Übersetzer der Vergleichsimplementierungen bei einer Vorausdeklaration und der eigentlichen Deklaration jeweils gleiche Exportmarkierungen erfordern, sieht der Oberon Übersetzer eine Prozedur bereits als exportiert an, wenn mindestens eine der Deklarationen eine Exportmarkierung besitzt. Die Oberon-2 Übersetzer akzeptieren zudem auch die Exportmarkierung - bei Konstanten-, Typ- und Prozedurdeklarationen.

Zu 1.1.5 Konstantendeklarationen, S.6:

Ohne diese Ergänzung bezüglich des Resultatstyps eines ganzzahligen Konstantenausdrucks wäre mit einer Festlegung der Repräsentation von `SHORTINT`-Werten als Zweierkomplement in 8 Bit im Fragment

```
VAR i: SHORTINT;
BEGIN
  CASE i OF
    128-1: ;
  END
END
```

der Case-Ausdruck fehlerhaft, da sein Typ (`SHORTINT`) nicht den Typ der Case-Marke (`INTEGER`) *einschließt* (vgl. S.14). Die Case-Marke ist ein Ausdruck, bestehend aus zwei Operanden mit numerischem Typ und dem Operator `-`. ,,Der Ergebnistyp [dieses arithmetischen Operators] ist der Typ des Operanden, der den Typ des anderen Operanden *einschließt*“ (vgl. S.11). Der erste Operand ist eine ganzzahlige Konstante, deren ,,Typ … der kleinste Typ [ist], zu dem der Konstantenwert gehört“ (vgl. S.4) und demnach `INTEGER`. Da der Typ des zweiten Operanden aus ebengenannten Gründen `SHORTINT` ist, ist deshalb der Typ des Case-Markenausdrucks `INTEGER`. Die Vergleichsimplementierungen erkennen hier keinen Fehler.

Zu 1.1.6 Typdeklarationen, S.6:

Diese Ergänzung bezüglich der Legalität von rekursiven Typdeklarationen ist die Ersetzung der Aussage: ,,A structured type cannot contain itself.“ (MÖSSENBÖCK und WIRTH [1993], 6. Type declarations). Nach Aussage eines der Autoren des Originalreports wird erwogen, diese Passage im Original durch den Satz ,,A type must not be used in its own declaration except as a pointer base type (see 6.4) or a type of a formal variable parameter (see 10.1).“ zu ersetzen. Ohne diese Präzisierung wären z.B. folgende rekursive Typdeklarationen möglich:

```
T0 = POINTER TO RECORD next: T0; END;
T1 = POINTER TO ARRAY SIZE(T1) OF CHAR;
T2 = PROCEDURE () : T2;
```

Die erwogene Ersetzung wird von den Vergleichsimplementierungen (noch) nicht berücksichtigt.

Zu 1.1.6.2 Array-Typen, S.7:

Diese Ergänzung entspricht der Realisierung in der erstellten Implementierung.

Zu 1.1.6.3 Record-Typen, S.8:

Die Ergänzung, daß sich alle in einem erweiterten Record deklarierten Namen lediglich von den sichtbaren Namen unterscheiden müssen, die in seinen Basistyp-Records deklariert sind, trägt dem Konzept der eingeschränkten Sichtbarkeit von Objekten über Modulgrenzen hinweg Rechnung und wird auch so von den Vergleichsimplementierungen gehandhabt. Da die Sprachdefinition hier von Namen und nicht von Feldern spricht, wird diese gelockerte Forderung auch auf typgebundene Prozeduren angewendet.

Zu 1.1.8.2 Operatoren, *Mengenoperatoren*, S.12 und 1.1.9.5 Case-Anweisungen, S.14:

Diese beiden Ergänzungen legen die Semantik eines Intervalls fest, dessen Untergrenze größer als die Obergrenze ist. Ein solches Intervall wird bei Mengen als leer und nicht als fehlerhaft angesehen, da eine Intervallgrenze nicht konstant sein muß und damit eine Laufzeitfehlerbehandlung erzeugt werden müßte. Demgegenüber kann bei einem Case-Marken-

bereich die gesamte Überprüfung zur Übersetzungzeit stattfinden, da die Case-Marken konstant sein müssen. Bei den Vergleichsimplementierungen findet keine Laufzeitüberprüfung statt, so daß die entsprechende Menge leer ist. Sind die Intervallgrenzen konstant, werden Fehler gemeldet.

Zu 1.1.9.2 Prozeduraufrufe, S.13
und 1.1.12.1 Begriffsdefinitionen, *Zuweisungskompatibilität* (7.), S.24:

Diese kleinen Ergänzungen unterstützten lediglich eine konsequente Verwendung der Begriffe. Auch ohne sie wäre die Sprachdefinition eindeutig, da diese Aussagen durch andere Stellen in der Definition impliziert werden.

Zu 1.1.10.2 Typgebundene Prozeduren, S.19:

Diese Präzisierung spiegelt die Tatsache wider, daß in der erstellten Implementierung eine Redefinition textuell vor der korrespondierenden Definition einer typgebundenen Prozedur deklariert werden kann. Obwohl der Originalreport dies nicht verbietet, wird es von den Vergleichsimplementierungen als Fehler gemeldet. Ein Verbot hätte in der erstellten Implementierung nur einen unwesentlich geringeren Aufwand bedeutet.

Zu 1.1.11 Module, S.22:

Diese Ergänzung drückt die konkrete Realisierung der Relation ,,Modul ↔ Datei“ aus.

Zu 1.1.12.3 Modul `SYSTEM`, S.27:

Aus Gründen der Orthogonalität vor allem zum Typ `PTR` wurde die Behandlung von aktuellen Parametern für formale Var-Parameter vom Typ `BYTE` in der vorgenommenen Implementierung so realisiert und deswegen in der Sprachdefinition entsprechend ergänzt. Die Oberon-2 Übersetzer gestatten hier neben `CHAR` und `SHORTINT` als aktuelle Parametertypen auch noch `LONGINT` und `LONGREAL`; der Oberon Übersetzer zusätzlich noch `REAL` und `SET`.

2. Werkzeuge

Für die verschiedenen Phasen eines Übersetzers existieren Werkzeuge, welche die Implementierung einer Phase unterstützen. Ein Werkzeug erzeugt aus einer Beschreibung in einer speziellen ,,Spezifikationssprache“ ein ,,Modul“ in einer höheren Programmiersprache, der sogenannten ,,Werkzeugzielsprache“. Die Beschreibung umfaßt die Spezifikation der konkreten Übersetzerphase, enthält jedoch auch Abschnitte mit Programmfragmenten der Werkzeugzielsprache. Aus den so erzeugten Modulen entsteht unter Benutzung einer (bereits bestehenden) Entwicklungsumgebung (Übersetzer/Binder) für eine konkrete Maschine ein ablauffähiger Übersetzer. Die Wahl einer höheren Programmiersprache als Zielsprache der Werkzeuge erhöht die Portabilität des erzeugten Übersetzers.

Bei der Erzeugung der einzelnen Module durch die Werkzeuge werden nur einige formalisierte Teilprobleme des Übersetzers mit adäquaten Methoden automatisch gelöst. Die zur Vervollständigung notwendigen (in der Werkzeugzielsprache formulierten) Programmfragmente müssen sich unter Kenntnis dieser Methoden korrekt in die erzeugten Module einfügen. Deshalb kann im Gegensatz zu *echten* Generatoren bei der Verwendung der Werkzeuge nicht von den Algorithmen, die den erzeugten Modulen zugrunde liegen, abstrahiert werden.

Die Compiler-Compiler-Toolbox beinhaltet eine Vielzahl von Werkzeugen. Als Werkzeugzielsprachen stehen Modula-2 und C zur Auswahl. Die Werkzeuge erzeugen voreingestellt Modula-2 Code. Für die verschiedenen Phasen in einem Übersetzer ist jeweils ein Werkzeug vorhanden. Zur Erzeugung des Parsers existiert sowohl ein Werkzeug für die Top-down- als auch für die Bottom-up-Analyse. Im vorliegenden Kapitel werden nur die Werkzeuge und ihre Einsatzmöglichkeiten beschrieben, welche in unserer Implementierung zur Anwendung kamen. Für eine vollständige Beschreibung der Werkzeuge wird deshalb auf die entsprechenden Werkzeugdokumentationen verwiesen. Da im vorliegenden Fall Modula-2 als Werkzeugzielsprache gewählt wurde, wird auf die Sprache C als Werkzeugzielsprache nicht weiter eingegangen.

Den Einsatz der gewählten Werkzeuge für die verschiedenen Phasen eines Übersetzers zur Generierungszeit sowie die Beziehung der generierten Module zur Übersetzungszeit zeigt Abbildung 2.

Generierungszeit

Aus einer Scanner-Spezifikation, basierend auf dem Konzept der regulären Ausdrücke, generiert das Werkzeug Rex ein Modul, welches eine Scanner-Funktion zur Verfügung stellt. Aus der Spezifikation einer kontextfreien Grammatik, angereichert um Angaben zur Abbildung der konkreten auf die abstrakte Syntax, erzeugt das Werkzeug Lalr das Parser-Modul. Das Werkzeug Ast generiert die Datenstruktur sowohl zur Repräsentation des Syntaxbaums aus einer Spezifikation der abstrakten Syntax als auch zur Repräsentation weiterer Objekte wie beispielsweise der Symboltabelle. In der Spezifikationssprache des Werkzeugs Ag wird eine Attributgrammatik formuliert, aus der das Modul zur semantischen Analyse generiert wird. Zur Formulierung von Hilfsfunktionen bei der semantischen Analyse stellt das Werkzeug Puma Funktionen zur Klassifikation und Transformation von Teilbäumen zur Verfügung, welche durch Ast spezifiziert wurden.

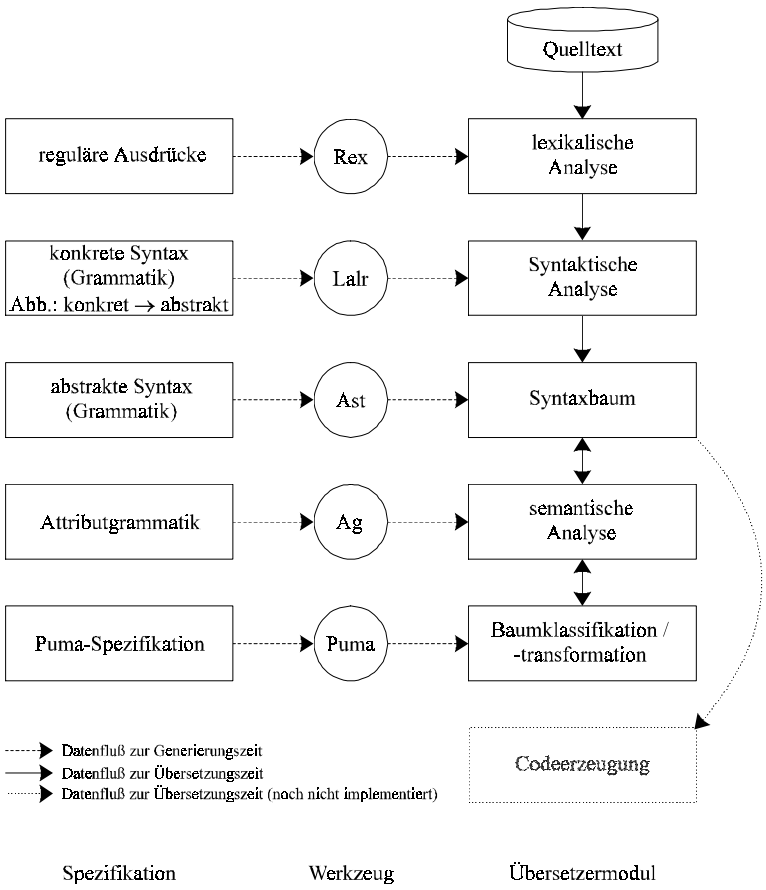

Abb. 2: Schematische Darstellung des Werkzeugeinsatzes für die verschiedenen Übersetzerphasen, nach GROSCH und EMMELMANN [1990].

Übersetzungszeit (Laufzeit des generierten Übersetzers)

Der Scanner liest zur lexikalischen Analyse den zu übersetzenden Quelltext zeichenweise aus einer Datei ein und liefert Symbole für die entsprechenden lexikalischen Einheiten an den Parser. Dieser baut während der syntaktischen Analyse den Syntaxbaum auf. Der Syntaxbaum wird durch die semantische Analyse mit Attributen dekoriert und einige seiner Teilbäume transformiert. Nach erfolgreicher semantischer Analyse wird anhand des Syntaxbaums die Codeerzeugung durchgeführt. Letzteres war nicht Bestandteil dieser Arbeit.

2.1 Rex – Generator für Scanner

Ein Scanner zerlegt einen Quellprogrammtext in Zeichenfolgen, die konzeptionell zusammengehören. Diese Zeichenfolgen werden aus Effizienzgründen auf abstrakte Symbole (Token) abgebildet oder überlesen. Die Abbildung wird häufig durch eine Scanner-Funktion `GetToken` realisiert, welche mit jedem Aufruf die nächste Zeichenfolge liest und das entsprechende Symbol in einer ganzzahligen Codierung liefert. Die Beschreibung der zu einem Symbol gehörenden Zeichenfolgen erfolgt im allgemeinen durch eine Menge von regulären Ausdrücken.

Rex (Regular EXpression tool) ist ein Scanner-Generator, dessen Eingabe eine Scanner-Spezifikation ist, die aus einzelnen Regeln besteht. Eine Regel beinhaltet eine Beschreibung von Zeichenfolgen durch (erweiterte) reguläre Ausdrücke sowie eine in der Werkzeugzielsprache formulierte Aktion. Diese Aktion besteht aus einer Anweisungsfolge, die mindestens die Berechnung des Resultats der generierten Scanner-Funktion beschreibt. Das Resultat muß von einem ganzzahligen Typ sein und wird im allgemeinen durch eine vom Spezifikateur festgelegte Konstante angegeben. Die Regeldefinitionen werden durch das Schlüsselwort `RULES` eingeleitet.

```
Specification  =  … Rules.
Rules          =  RULES {Rule}.
Rule           =  Pattern ":" TargetCode.
Pattern        =  RegExpr.
TargetCode     =  "{" {character} "}" .
```

Beispiel:

```
CONST IntegerToken = 2; … ArrayToken = 33;
…
RULES
"ARRAY" : { RETURN ArrayToken; }
{ 0-9 }+ : { RETURN IntegerToken; }
```

2.1.1 Spezifikationssprache

Reguläre Ausdrücke

Rex kennt den Beschreibungsmechanismus der regulären Ausdrücke. Reguläre Ausdrücke über einem Alphabet Σ (eine endliche, nichtleere Menge von Zeichen) und die von ihnen jeweils beschriebenen Sprachen können induktiv definiert werden (vgl. WILHELM und MAURER [1992], S.189):

- ∅ ist ein regulärer Ausdruck über Σ und beschreibt die reguläre Sprache ∅.
- ε ist ein regulärer Ausdruck über Σ und beschreibt die reguläre Sprache {ε}.
- a (für $a \in \Sigma$) ist ein regulärer Ausdruck über Σ und beschreibt die reguläre Sprache $\{a\}$.

Sind r_1 und r_2 reguläre Ausdrücke, welche die regulären Sprachen R_1 bzw. R_2 beschreiben, so ist

- $r_1|r_2$ ein regulärer Ausdruck über Σ und beschreibt die reguläre Sprache $R_1 \cup R_2$ (Vereinigung von Sprachen),
- $r_1 r_2$ ein regulärer Ausdruck über Σ und beschreibt die reguläre Sprache $R_1 R_2$ (Konkatenation von Sprachen),
- r_1* ein regulärer Ausdruck über Σ und beschreibt die reguläre Sprache R_1^* (Abschluß einer Sprache) und
- nichts sonst ein regulärer Ausdruck.

Die so beschriebenen regulären Ausdrücke lassen sich in ähnlicher Form innerhalb einer Scanner-Spezifikation angeben. Rex verwendet als Alphabet die Zeichen des ASCII-Zeichensatzes.

Einfache reguläre Ausdrücke

Zeichenkonstanten: Der einfachste reguläre Ausdruck besteht aus einem einzelnen Zeichen *a*. Dieser Ausdruck beschreibt die Sprache {*a*}, welche nur aus ebendiesem Zeichen *a* besteht. Die Zeichen, die Teil des Spezifikationsmechanismus sind, werden Metazeichen genannt und müssen mit dem sog. Escape-Zeichen "\" präfiziert werden. Alle Zeichen lassen sich auch durch die Angabe ihres ASCII-Werts mit vorangestelltem Escape-Zeichen darstellen. Für einige nichtdruckbare Zeichen stehen gesonderte Abkürzungen zur Verfügung.

```
Char = character | "\" digit {digit} | "\n" | "\t" | "\v" | "\b" | "\r" | "\v" | "\" character.
```

Beispiel:

Regulärer Ausdruck	Erkanntes Zeichen
`b`	`b`
`\{`	`{`
`\\`	`\`
`\65`	`A`
`\n`	`LF` (Zeilenende)
`\t`	`HT` (Horizontaltabulator)

Zeichenkettenkonstanten: Eine Zeichenkettenkonstante $a_1 \ldots a_n$, mit $a_i \in \Sigma$ für $1 \le i \le n$, beschreibt die reguläre Sprache $\{a_1 \ldots a_n\}$. Sie wird in Anführungszeichen (") geschrieben.

```
String = '"' {character} '"'.
```

Beispiel:

Regulärer Ausdruck	Erkannte Zeichenfolge
`"IMPORT"`	`IMPORT`
`":="`	`:=`

Namenskonstanten: Eine Namenskonstante ist eine Zeichenkettenkonstante ohne Anführungszeichen und darf nur aus Buchstaben und Ziffern bestehen.

```
Ident = letter { letter | digit }.
```

Zeichenmengen: Der in Rex formulierbare Ausdruck $\{a_1 \ldots a_n\}$, mit $a_i \in \Sigma$ und $1 \le i \le n$, ist eine abkürzende Schreibweise für den regulären Ausdruck $a_1|\ldots|a_n$ und beschreibt somit die gleiche Sprache. Innerhalb einer Zeichenmenge steht ein Ausdruck *a*-*b* (*a*, *b* ∈ Σ) für alle Zeichen, deren ASCII-Wert sich in dem Intervall [ASCII(*a*),ASCII(*b*)] befindet. Der vordefinierte Name `ANY` steht für die Zeichenmenge, die alle Zeichen enthält, bis auf das Zeilenendezeichen. Sei *s* eine Zeichenmenge, so beschreibt der Ausdruck –*s* die Sprache (`ANY` ∪ {`\n`}) – *s* (Komplement einer Sprache).

```
CharSet = "-" CharSet | "{" {Range} "}".
Range   = Char | Char "-" Char.
```

Beispiele:

Regulärer Ausdruck	Zeichen
`{ + \- * / }`	`+ - * /`
`{ 0-9 }`	alle Ziffern
`- { \0-\32 }`	alle Zeichen mit ASCII-Wert größer 32

Zusammengesetzte reguläre Ausdrücke

Reguläre Ausdrücke können mittels Operatoren aus einfacheren regulären Ausdrücken zusammengesetzt sein. Rex akzeptiert neben der bereits aus der formalen Definition bekannten Konkatenation, der Alternative und dem Abschluß noch weitere Operatoren.

Option: Der reguläre Ausdruck *r?* beschreibt die gleiche Sprache wie der Ausdruck $r|\varepsilon$.

Einschließende Iteration: Ein regulärer Ausdruck *r+* beschreibt die gleiche Sprache wie der Ausdruck $r\ r*$.

Fest definierte Iteration: Ein regulärer Ausdruck der Form *r[n]* entspricht dem Ausdruck $r_1 \ldots r_n$, mit $r_i = r$ für $1 \le i \le n$.

Variable Iteration: Ein regulärer Ausdruck der Form *r[m-n]* entspricht dem Ausdruck *r[m]|r[m+1]|…|r[n–1]|r[n]*, mit $m \le n$.

Gruppierung: Durch Klammerung eines regulären Ausdrucks der Form *(r)* kann der Vorrang der Operatoren umgangen werden.

Nachfolgend die vollständige Regel zur Bildung regulärer Ausdrücke in Rex.

```
RegExpr =  Char | String | Ident | CharSet
        |  "(" RegExpr ")"
        |  RegExpr "[" Number "-" Number "]"
        |  RegExpr "[" Number "]"
        |  RegExpr "?"
        |  RegExpr "*"
        |  RegExpr "+"
        |  RegExpr RegExpr
        |  RegExpr "|" RegExpr .
Number  =  digit {digit}.
```

Die Operatoren sind entsprechend ihres Vorrangs mit dem höchsten beginnend aufgeführt.

Benannte reguläre Ausdrücke

Reguläre Ausdrücke können zum Zwecke der Abkürzung benannt werden, so daß sie in anderen regulären Ausdrücken durch Angabe ihres Namens benutzt werden können. Diese Benennungen sind in einem speziellen Abschnitt der Spezifikation zusammenzufassen.

```
Define     =  DEFINE { Definition }.
Definition =  ident "=" RegExpr.
```

Beispiel

```
DEFINE
   Digit    = { 0-9     }.
   HexDigit = { 0-9 A-F }.
   Letter   = { a-z A-Z }.
```

Muster

Innerhalb einer Regel der Spezifikation werden die regulären Ausdrücke als sogenannte Muster definiert. Ein Muster ist ein regulärer Ausdruck, der nicht Bestandteil eines (umfassenden) regulären Ausdrucks ist. Sollen für verschiedene Muster dieselben Aktionen definiert werden, so darf als abkürzende Schreibweise eine Regel aus mehreren, durch Kommata getrennten

Mustern bestehen. Durch Voranstellen bzw. Anhängen spezieller Zeichen (<, >) an das Muster, läßt sich die Zugehörigkeit einer Zeichenfolge zu der durch das Muster beschriebenen Sprache in Abhängigkeit von der Position (Zeilenanfang/-ende) der Zeichenfolge einschränken. Die Definition einer Regel lautet also genauer:

```
Rule        =  PatternList ":" TargetCode.
PatternList =  Pattern { "," Pattern }.
Pattern     =  [ "<" ] RegExpr [ ">" ].
TargetCode  =  "{" {character} "}".
```

Regelauswahl und rechter Kontext

Als Erkennungsfolge eines Musters bezeichnen wir die längste Zeichenfolge, die Wort der Sprache des Musters ist und die Präfix der noch nicht gelesenen Eingabezeichen ist. Existieren Erkennungsfolgen von Mustern verschiedener Regeln, wird die Regel ausgewählt, deren Muster die längste Erkennungsfolge besitzt. Existieren maximal lange Erkennungsfolgen von Mustern mehrerer Regeln, wird die (textuell) erste Regel ausgewählt.

Die Auswahl der Regel, dessen Muster die längste Erkennungsfolge besitzt, ist dann problematisch, wenn eine von der Scanner-Funktion zu erkennende Zeichenfolge ein echtes Präfix einer Zeichenfolge ist, die Wort der Sprache eines Musters einer anderen Regel ist. Das Spezifikationsfragment

```
Digit+            : { RETURN IntegerToken; }
Digit+ "." Digit* : { RETURN RealToken; }
…
".."              : { RETURN DotDotToken; }
"."               : { RETURN PeriodToken; }
```

ergibt für die Eingabezeichen "`1..2`" die Symbole *`RealToken PeriodToken IntegerToken`*. Sollen diese Eingabezeichen jedoch z. B. für einen Ganzzahlbereich stehen und damit auf eine Folge von *`IntegerToken DotDotToken IntegerToken`* abgebildet werden, kann durch Angabe des Kontextoperators "`/`", gefolgt von einem regulären Ausdruck als rechter Kontext, die Länge der Erkennungsfolge des Musters vergrößert werden, ohne daß die zu dieser Erweiterung gehörenden Zeichen ,,gelesen“ werden.

```
Pattern  =  [ "<" ] RegExpr [ "/" RegExpr ] [ ">" ].
```

Beispiel:

```
Digit+ , Digit+ / ".." : { RETURN IntegerToken; }
```

Startzustände

Die Anwendbarkeit von Regeln kann eingeschränkt werden. Zu diesem Zweck lassen sich benannte Startzustände deklarieren. Diese können vor dem regulären Ausdruck eines Musters angegeben werden. Ein Muster wird nur berücksichtigt, wenn der Scanner sich in einem der im Muster angegebenen Startzustände befindet. Auf diese Weise kann zur Erkennung eines Symbols sein linker Kontext berücksichtigt werden. Ein Muster ohne angegebene Startzustände ist in jedem Startzustand des Scanners gültig. Während es innerhalb von Aktionen möglich ist, den Startzustand des Scanners durch Aufruf der Prozedur `yyStart` zu wechseln, kann der aktuelle Startzustand über die Variable `yyStartState` ermittelt werden. Der Scanner befindet sich initial im vordefinierten Startzustand `STD`.

```
Pattern     =  [ StartStates ] [ "<" ] RegExpr [ "/" RegExpr ] [ ">" ].
StartStates =  [ NOT ] "#" IdentList "#".
IdentList   =  ident { [","] ident }.
```

Wird der Liste der Startzustände das Schlüsselwort `NOT` vorangestellt, wird das nachfolgende Muster nur berücksichtigt, wenn sich der Scanner nicht in einem der angegebenen Startzustände befindet.

Durch die Definition verschiedener Startzustände lassen sich beispielsweise geschachtelte Kommentare elegant erkennen.

```
DEFINE CommentChar  = -{ * ( }.
…
#STD, Comment# "(*" : { INC(NestingLevel); yyStart(Comment); }
#Comment# "*)" : { DEC(NestingLevel);
                   IF NestingLevel=0 THEN yyStart(STD); END; }
#Comment# "(" | "*" | CommentChar + : { }
```

Die ganzzahlige Variable `NestingLevel` (initial 0) ist dabei an anderer Stelle zu deklarieren.

Aktionen

Aktionen sind Anweisungsfolgen, die in der Werkzeugzielsprache formuliert werden. Sie werden unverändert in den generierten Code eingefügt. Zur Programmierung der Aktionen stellt Rex verschiedene Prozeduren zur Verfügung: Neben der bereits erwähnten Prozedur `yyStart`, zum Wechseln des internen Scanner-Startzustands, liefert `GetWord(`*v*`)` über den Parameter *v* die erkannte Zeichenkette als Wert vom Typ `Strings.tString`.

Beispiel

```
#STD# Digit+ :{ GetWord(repr); val:=Strings.StringToInt(repr);
                IF val > MAX(SHORTINT) THEN Error; END;
                RETURN IntegerToken; }
```

Quelltextpositionen

Beim Lesen der Zeichenfolgen des Quellprogrammtexts wird die Position der Zeichen intern gespeichert. Für jede in einer Regel spezifizierte Zeichenfolge wird bei deren Erkennung die Position des ersten Zeichens dieser Zeichenfolge im Feld `Position` der globalen Variable `Attribute` (s.u.) abgelegt. Durch das Zeichen `-` vor den geschweiften Klammern einer Aktion wird die Aktualisierung dieses Felds bei der Regelanwendung unterbunden. Trotzdem wird auch in einem solchen Fall die Quelltextposition weitergerechnet.

Zur korrekten Behandlung der Quelltextpositionen nach Tabulatorzeichen existiert die generierte Prozedur `yyTab`, die in einer vordefinierten Regel die Spaltenposition auf die nächste Tabulatorposition setzt, falls eine Tabulatorzeichen erkannt wird.

2.1.2 Generierter Code

Die zentrale Scanner-Funktion `GetToken` liefert als Information nur das nächste erkannte Symbol in einer ganzzahligen Repräsentation. Das generierte Scanner-Modul stellt deshalb zusätzlich eine globale Variable `Attribute` vom Typ `tScanAttribute` zur Verfügung. In dieser Variable werden weitere Informationen zum gelieferten Symbol gespeichert, wie z. B.

die Quelltextposition. Der Typ `tScanAttribute` muß ein Record-Typ sein, der mindestens das Feld `Position`, wie in

```
TYPE tPosition      = RECORD
                       Line, Column : SHORTCARD;
                      END;
     tScanAttribute = RECORD
                       Position    : tPosition;
                       CASE : CARDINAL OF
                       |1: Ident   : Idents.tIdent;
                       |2: Integer : INTEGER;
                       |3: String  : Strings.tString;
                       END;
                      END;
```

besitzt. Der Record-Typ wird im allgemeinen vom Spezifikateur um weitere Felder ergänzt.

Daneben werden noch folgende Prozeduren generiert: `BeginScanner` / `CloseScanner` dient dem Auf- bzw. Abbau zusätzlicher Datenstrukturen. `BeginFile` / `CloseFile` öffnet bzw. schließt die zu lesende Eingabedatei, deren Name als Parameter `BeginFile` übergeben wird.

Eine Rex-Spezifikation wird durch Schlüsselwörter in verschiedene Abschnitte unterteilt und hat im allgemeinen folgenden Aufbau:

```
EXPORT  { export }
GLOBAL  { global }
LOCAL   { local }
BEGIN   { begin }
CLOSE   { close }
DEFAULT { default }
EOF     { eof }
DEFINE  …
START   …
RULES   rules
```

Hieraus erzeugt Rex folgenden Modula-2 Quelltext. Die in geschweiften Klammern stehenden Namen des vorherigen Abschnitts stellen Fragmente der Werkzeugzielsprache dar, die ungeprüft in den Modula-2 Quelltext eingefügt werden.

```
DEFINITION MODULE Scanner;
export …
PROCEDURE BeginScanner;
PROCEDURE BeginFile(FileName: ARRAY OF CHAR);
PROCEDURE GetToken(): INTEGER;
PROCEDURE GetWord(VAR Word: String.tString): INTEGER;
PROCEDURE CloseFile;
PROCEDURE CloseScanner; …
END Scanner.
```

```
IMPLEMENTATION MODULE Scanner;
global …
PROCEDURE BeginScanner; BEGIN begin END BeginScanner; …
PROCEDURE GetToken(): INTEGER;
  local
BEGIN
  Generierter Code aus rules, eof, default
END GetToken; …
PROCEDURE CloseScanner; BEGIN close END CloseScanner; …
END Scanner.
```

Im Abschnitt EXPORT werden üblicherweise die Importierungen der Datentypen für die benutzerdefinierten Felder des Scanner-Attributs vorgenommen sowie die beiden Datentypen tScanAttribute und tPosition definiert. Nur wenn in der Spezifikation kein Exportabschnitt definiert wird, generiert Rex die beiden Datentypen in einer minimalen Form. Der Abschnitt GLOBAL nimmt die Importierungen des Implementierungsmoduls sowie globale Variablen (z. B. NestingLevel) und Hilfsprozeduren auf. Der Abschnitt LOCAL enthält Deklarationen, die zur Scanner-Funktion lokal sind. In die bereits erwähnten Prozeduren BeginScanner und CloseScanner werden die Abschnitte BEGIN bzw. CLOSE eingefügt.

Im Gegensatz zu den bis hierher aufgeführten Spezifikationsabschnitten sind die abschließenden drei Abschnitte eng mit der eigentlichen Scanner-Funktionalität verbunden.

Die in DEFAULT angegebenen Aktionen werden ausgeführt, falls bei einem Aufruf von GetToken keine der spezifizierten Regeln anwendbar war. Dies kann dazu benutzt werden, entsprechende Fehlermeldungen zu erzeugen. Im Abschnitt EOF können Aktionen definiert werden, die ausgeführt werden, wenn das Ende des Eingabestroms erreicht wurde. Üblicherweise werden hier Fehlermeldungen erzeugt, falls z. B. das Ende eines Kommentars noch nicht erreicht worden ist.

Aus den Regelspezifikationen, die hinter dem Schlüsselwort RULES aufgeführt sind, generiert Rex zusammen mit den Anweisungen aus DEFAULT und EOF den Rumpf der Prozedur GetToken.

2.2 Ast – Generator für abstrakte Syntaxbäume

Ein abstrakter Syntaxbaum repräsentiert die Struktur eines zu übersetzenden Programms. Das Werkzeug Ast (Abstract Syntax Tree generator) ermöglicht eine kompakte Definition der Struktur von abstrakten Syntaxbäumen, indem Knotentypen und Elter-Kind-Beziehungen zwischen Knoten dieser Knotentypen angegeben werden. Aus einer Ast-Spezifikation erzeugt das Werkzeug für jeden Knotentyp einen Datentyp und eine Konstruktorprozedur in der Werkzeugzielsprache. Neben der Spezifikation von abstrakten Syntaxbäumen kann Ast auch zur Modellierung von Datentypen eingesetzt werden, die bei der Attributauswertung benötigt werden, wie z. B. Symboltabelleneinträge oder Repräsentationen der Quellsprachentypen.

2.2.1 Spezifikationssprache

Eingeleitet durch das Schlüsselwort `RULE` erfolgt die Definition der Knotentypen und ihrer Struktur im Regelabschnitt der Spezifikation.

```
Specification  =  … Spec … .
Spec           =  TREE [ ident ] … Rules … .
Rules          =  RULE { Type }.
Type           =  ident "=" { Field } "." .
Field          =  Child | Attr .
Child          =  [ ident ":" ] ident .
Attr           =  "[" ident [ ":" ident ] "]" .
```

Der hinter dem Schlüsselwort `TREE` aufgeführte Name bildet den Namen des zu erzeugenden Moduls. Fehlt dieser, so erhält das zu erzeugende Modul den Namen `Tree`.

Beispiel:

```
RULE …
Module = [ name    : tIdent ]
           imports : Imports
           decls   : Decls
           stmts   : Stmts .
```

Jeder Knotentyp besteht aus einer festen Anzahl von Feldern. Ein Feld besteht aus einem Selektornamen gefolgt von einem Doppelpunkt und einem Typ. Es existieren zwei Arten von Feldern: Kinder und Attribute. Während der Typ eines Kinds ein Knotentyp sein muß, sind Attribute in der Regel von einem Typ der Werkzeugzielsprache. Letztere dienen der Aufnahme zusätzlicher Information für Knoten. Ist bei einem Attribut kein Typ angegeben, hat es den Typ `INTEGER`. Die Selektornamen aller Felder eines Knotentyps müssen paarweise verschieden sein. Wird der Selektorname (zusammen mit dem Doppelpunkt) bei einem Kind weggelassen, so gilt der Knotentypname dieses Kinds als Selektorname. Folgendes Beispiel zeigt eine Anwendung dieses Abkürzungsmechanismus:

```
WhileStmt = Expr
            Stmts .
```

Erweiterungen

Ein Knotentyp *Ta* kann eine Erweiterung eines anderen (möglicherweise bereits erweiterten) Knotentyps *Tb* sein. *Tb* ist dann der Basisknotentyp von *Ta*. Ein erweiterter Knotentyp enthält alle Kinder und Attribute seines Basisknotentyps. Die Erweiterung von Knotentypen ermöglicht polymorphe Strukturen im Syntaxbaum: Überall wo ein bestimmter Knotentyp erfor-

derlich ist, kann auch eine Erweiterung dieses Knotentyps angegeben werden. Die Erweiterungsbeziehung ist reflexiv, d.h. ein Knotentyp ist seine eigene Erweiterung und sein eigener Basisknotentyp.

```
Type      =  ident "=" { Field } [ Extension ] "." .
Extension =  "<" { Type } ">" .
```

Beispiel:

```
Exprs        = [ pos   : tPosition ]
<  EmptyExpr = .
   DyExpr    = [ oper  : tOperator ]
                 expr1 : Exprs
                 expr2 : Exprs .
   IntConst  = [ int   : INTEGER   ] .
>.
```

Nachstehendes Spezifikationsfragment zeigt Knotendefinitionen zur Repräsentation von Quellsprachentypen, bei denen Erweiterungen zur Anwendung kommen.

```
RULE …
TypeReprs             =
<  EmptyTypeRepr      = .
   TypeRepr           = [ size         : INTEGER ]
   <  BooleanTypeRepr = .
      IntegerTypeRepr = .
      ArrayTypeRepr   = [ len          : INTEGER ]
                          elemTypeRepr : TypeReprs .
   >.
>.
```

Eigenschaften

Kindern und Attributen eines Knotentyps können bestimmte Eigenschaften (Properties) zugeordnet werden. Für die reine Baumerzeugung wird lediglich die Eigenschaft `INPUT` benötigt. Nur Kinder und Attribute mit dieser Eigenschaft erhalten ihren Wert bei der Knotenerzeugung, indem durch entsprechende formale Parameter im Kopf der Konstruktorprozedur die Angabe eines Werts bei der Knotenerzeugung erforderlich wird. Alle weiteren Eigenschaften sind nur im Zusammenhang mit dem Werkzeug Ag relevant, welches einen Evaluator generiert, der auf einem mittels Ast-Prozeduren erzeugten Baum arbeitet. Für die Angabe dieser Eigenschaften existieren prinzipiell zwei Möglichkeiten: die lokale und die globale Zuordnung. Bei der lokalen Zuordnung können die Eigenschaften direkt der Deklaration eines Felds folgen:

```
Type     =  ident "=" { Field } [ Extension ] "." .
Field    =  Child | Attr .
Child    =  [ ident ":" ] ident { Property }.
Attr     =  "[" ident ":" ident { Property } "]" .
Property =  INPUT | …
```

Die globale Zuordnung ermöglicht, die Felder aller Knotentypen mit Eigenschaften zu versehen. In dem durch das Schlüsselwort `PROPERTY` eingeleiteten Spezifikationsabschnitt müssen diese Eigenschaften angegeben werden.

```
Spec     =  … PropPart … Rules … .
PropPart =  PROPERTY { Property } .
```

Module

Die Definition von Knotentypen kann innerhalb eines Spezifikationsdokuments auf mehrere Module verteilt werden. Insbesondere können auch die Felder eines Knotentyps in verschiedenen Modulen definiert werden. Dies gestattet eine Zusammenfassung von thematisch zusammengehörenden Feldern von verschiedenen Knotentypen. Die globale Zuordnung von Eigenschaften bleibt auf die Felder der Knotentypen eines Moduls beschränkt.

```
Specification   =  Spec Modules | MODULE ident Spec END ident Modules .
Modules         =  { MODULE ident Spec END ident } .
```

Beispielsweise definiert die Spezifikation

```
MODULE Structure …
PROPERTY INPUT
RULE Node = left:Node right:Node .
END Structure

MODULE Data …
RULE Node = [ data:INTEGER ] .
END Data
```

einen Knotentyp `Node` bestehend aus zwei Kindern mit der Eigenschaft `INPUT` und einem Attribut ohne diese Eigenschaft.

2.2.2 Generierter Code

Die Benutzung des von Ast erzeugten Moduls erfolgt im allgemeinen innerhalb von Anweisungen der Werkzeugzielsprache. Deshalb ist die Kenntnis der Schnittstelle des generierten Codes hier von besonderem Interesse.

Anhand des folgenden Beispiels soll sowohl die Bedeutung bestimmter Spezifikationsabschnitte, als auch die Umsetzung einer Knotentypdefinition in Modula-2 Konstrukte erläutert werden.

```
MODULE Example TREE ThisTree
IMPORT   { import }
EXPORT   { export }
GLOBAL   { global }
BEGIN    { begin  }
CLOSE    { close  }
PROPERTY INPUT
RULE     Slice    = [ c:CHAR ]
         <  Empty = .
            Other = n:Slice .
         >.
END Example
```

Aus jedem Knotentyp erzeugt Ast einen Record-Typ, eine ganzahlige Konstante, die den Knotentyp identifiziert (Knotentypkonstante) und eine Konstruktorprozedur, deren Name aus dem Knotentypnamen, präfiziert mit dem Buchstaben `m`, gebildet wird. Alle Konstruktorprozeduren liefern Werte desselben Zeigertyps (des Baumtyps), der auf ein variantes Record verweist. So entsteht aus obiger Spezifikation das Modul:

```
DEFINITION MODULE ThisTree;
import
CONST NoThisTree = NIL;
      Slice      = 1;
      Empty      = 2;
      Othe       = 3;
TYPE  tThisTree  = POINTER TO yyNode;
export
TYPE  ySlice     = RECORD yyKind:SHORTCARD; c:CHAR; END;
      yEmpty     = RECORD yyKind:SHORTCARD; c:CHAR; END;
      yOther     = RECORD yyKind:SHORTCARD; c:CHAR; n:tThisTree; END;
      yyNode     = RECORD
                    CASE :SHORTCARD OF
                    |0    : Kind :SHORTCARD;
                    |Slice: Slice:ySlice;
                    |Empty: Empty:yEmpty;
                    |Other: Other:yOther;
                    END;
                   END;
VAR   ThisTreeRoot : tThisTree;
PROCEDURE mSlice(pc:CHAR): tThisTree;
PROCEDURE mEmpty(pc:CHAR): tThisTree;
PROCEDURE mOther(pc:CHAR; pn:tThisTree): tThisTree;
PROCEDURE IsEqualThisTree(Tree1, Tree2: tThisTree): BOOLEAN;
PROCEDURE BeginThisTree;
PROCEDURE CloseThisTree;
END ThisTree.

IMPLEMENTATION MODULE ThisTree;
global …
PROCEDURE mSlice(pc:CHAR): tThisTree; BEGIN … END mSlice;
PROCEDURE mEmpty(pc:CHAR): tThisTree; BEGIN … END mEmpty;
PROCEDURE mOther(pc:CHAR; pn:tThisTree): tThisTree;
  BEGIN … END mOther;
PROCEDURE IsEqualThisTree(Tree1, Tree2: tThisTree): BOOLEAN;
  BEGIN … END IsEqualThisTree;
PROCEDURE BeginThisTree; BEGIN begin END BeginThisTree;
PROCEDURE CloseThisTree; BEGIN close END CloseThisTree; …
END ThisTree.
```

Die Variable `ThisTreeRoot` kann zur (globalen) Speicherung einer Baumwurzel benutzt werden. Eine Konstruktorprozedur reserviert im Freispeicher den für den zu konstruierenden Knoten benötigten Platz und weist dem Feld `yyKind` (bzw. `Kind`) die entsprechende Knotentypkonstante zu. Das Feld `Kind` im varianten Teil des Record-Typs `yyNode` ermöglicht den Zugriff auf das Feld `yyKind` eines konkreten Baumknotens, ohne den jeweiligen Knotentyp zu kennen, da jeder Knotentyp-Record `yyKind` als erstes Feld besitzt.

Die Prozedur `IsEqualThisTree` kann zum strukturellen Vergleich zweier (Teil-)Bäume benutzt werden: Ein Teilbaum ist strukturell gleich einem anderen Teilbaum, wenn beide vom gleichen Knotentyp sind, die korrespondierenden Attribute beider Wurzelknoten gleiche Werte haben und die korrespondierenden Kinder beider Wurzelknoten strukturell gleich sind.

2.3 Lalr – Generator für LALR(1)-Parser

Die Aufgabe eines Parsers ist es, aus der vom Scanner gelieferten Symbolfolge gemäß einer zugrundeliegenden Grammatik einen abstrakten Syntaxbaum zu erzeugen. Dieser Syntaxbaum stellt eine Abstraktion des einfachen Ableitungsbaums, der sich prinzipiell auch automatisch erzeugen ließe, dar. Für den Fall, daß die Eingabesymbolfolge kein Satz der durch die Grammatik erzeugten Sprache ist, werden entsprechende Fehlermeldungen ausgegeben.

Das Werkzeug Lalr erzeugt einen Bottom-up-Parser aus einer kontextfreien Grammatik, welche die LALR(1)-Eigenschaft erfüllt und deren Regeln die Aktionen zur Konstruktion des abstrakten Syntaxbaums zugeordnet sind. Die Grammatik ist der formale Anteil der Parserspezifikation und wird durch kontextfreie Regeln angegeben. Die Aktion einer Regel wird ausgeführt, wenn der Parser die rechte Seite der zugeordneten Regel zum Nichtterminal auf der linken Regelseite reduziert. Diese Aktionen müssen vom Spezifikateur in der Werkzeugzielsprache formuliert werden.

Ist der Einsatz einer Bottom-up-Strategie nicht angezeigt, kann das Werkzeug Ell (Extenden LL(1)) zur Erzeugung eines Parsers mit Top-down-Strategie verwendet werden.

2.3.1 Spezifikationssprache

Den wesentlichen Abschnitt der Spezifikation stellt die kontextfreie Grammatik dar. Formal ist eine kontextfreie Grammatik ein System $G = (V_N,V_T,P,S)$, wobei V_N, V_T disjunkte Alphabete sind, V_N die Menge der Nichtterminale, V_T die Menge der Terminale, $P \subseteq V_N \times (V_N \cup V_T)^*$ die endliche Menge der Regeln ist und $S \in V_N$ das Startsymbol ist (vgl. WILHELM und MAURER [1992], S.222).

Terminale

Die Terminale der Grammatik sind die vom Scanner gelieferten Symbole in ganzzahliger Kodierung. Sie werden in einem eigenen durch das Schlüsselwort `TOKEN` eingeleiteten Abschnitt durch Namen bzw. Zeichenketten definiert. Die Namen müssen sich von den Schlüsselwörtern der Spezifikationssprache unterscheiden. Der Spezifikateur muß sicherstellen, daß die Symbolkodierungen von Scanner und Parser übereinstimmen!

```
Tokens      =  TOKEN Declaration { Declaration }.
Declaration =  Terminal [ "=" number ].
Terminal    =  ident | string.
```

Wird für die Symbolkodierung keine Nummer angegeben, so weist das Werkzeug diesem Symbol nach einem festen Schema eine noch nicht benutzte Nummer zu.

Beispiel:

```
TOKEN integer = 2   ...   '+' = 7   ...   ARRAY = 33
```

Nichtterminale und Startsymbol

Durch die linken Regelseiten der Grammatik sind implizit die Nichtterminale definiert. Dabei ist das Nichtterminal auf der linken Seite der (textuell) zuerst aufgeführten Regel das Startsymbol. Nichtterminale können nur durch Namen bezeichnet werden.

Regeln

Die Regeln der Grammatik werden hinter dem Schlüsselwort RULE aufgeführt. Sie sind im wesentlichen durch das folgende Grammatikfragment beschrieben:

```
Specification  =  … Rules.
Rules          =  RULE Rule { Rule }.
Rule           =  ident ":" RightSide ".".
RightSide      =  { ident | string } [ TargetCode ].
TargetCode     =  "{" {character} "}".
```

Beispiel:

```
TOKEN ident = 1   integer = 2 … '.' = 13
      ARRAY = 33 … OF = 53 … POINTER = 55 … TO = 61
…
RULE Type      : Qualident.
     Type      : ARRAY integer OF Type.
     Type      : POINTER TO Type.
     Qualident : ident.
     Qualident : ident '.' ident.
```

S-Attributierung

Lalr stellt für jedes Regelsymbol (Terminal und Nichtterminal) ein Attribut zur Verfügung, über das eine abgeleitete (synthesized) Information während der Parsierung bei jedem Reduktionsschritt weitergegeben werden kann. Die Attribute sind alle vom gleichen Typ `tParsAttribute`, der vom Spezifikateur im globalen Deklarationsabschnitt definiert wird:

```
TYPE tParsAttribute = RECORD
                       CASE :INTEGER OF
                       |1: Scan: Scanner.tScanAttribute;
                       |2: Tree: Tree.tTree;
                       END;
                      END;
```

Dieser Typ muß mindestens das Feld `Scan` vom Typ des Scanner-Attributs enthalten, da während der Parsierung dort für jedes Terminalsymbol die vom Scanner gelieferte Attributinformation automatisch abgelegt wird. Er kann um weitere Felder ergänzt werden, in denen z. B. bereits konstruierte Teilbäume des abstrakten Syntaxbaums gespeichert werden können.

Aktionen

Die in der Werkzeugzielsprache formulierten Anweisungen stellen die Aktionen dar, die bei der Reduktion einer rechten Regelseite zum Nichtterminal auf der linken Regelseite ausgeführt werden. Aktionen müssen grundsätzlich am Ende einer Regel stehen. Innerhalb dieser Aktionen kann auf das Attribut des *i*-ten Regelsymbols der rechten Seite mittels der Pseudovariable $*i* zugegriffen werden. Das Attribut des Nichtterminals auf der linken Regelseite wird mit $$ angesprochen.

Beispiel:

```
RULE
  Type      : Qualident
            { $$.Tree:=MakeNamedType($1.Tree) }.
  Type      : ARRAY integer OF Type
            { $$.Tree:=MakeArrayType($2.Scan.Integer,$4.Tree) }.
  Qualident : ident '.' ident
            { $$.Tree:=MakeQualident($1.Scan.Ident,$3.Scan.Ident) }.
```

Mehrdeutige Grammatiken

Das Werkzeug erzeugt auch im Fall vom Grammatiken, welche die LALR(1)-Eigenschaft nicht erfüllen, stets einen *Parser*. So kommt es z.B. bei mehrdeutigen Grammatiken zu Shift-Reduce- und/oder Reduce-Reduce-Konflikten. Diese werden schon bei der Generierung des Parsers gemeldet und von Lalr nach definierten Standardregeln gelöst. Zur Behandlung der genannten Konflikte können Terminale als Operatoren definiert werden. Sie werden hinter dem Schlüsselwort `OPER` aufgeführt und können Vorränge und Assoziativitäten besitzen. Beim Lösen der genannten Konflikte orientiert sich der Generator an diesen Operatoren, falls sie in den problematischen Regeln vorkommen. Führt dies zu keiner Lösung, greift Lalr auf die eben erwähnten Standardregeln zur Konfliktauflösung zurück. Gemäß diesen wird bei einem Shift-Reduce-Konflikt immer gelesen und bei einem Reduce-Reduce-Konflikt immer nach der (textuell) ersten Regel reduziert.

2.3.2 Generierter Code

Eine Lalr-Spezifikation wird durch Schlüsselwörter in verschiedene Abschnitte unterteilt und hat im allgemeinen den Aufbau:

```
EXPORT { export }
GLOBAL { global }
BEGIN  { begin }
CLOSE  { close }
TOKEN  tokendef
OPER   operdef
RULES  rules
```

Der daraus erzeugte Modula-2 Code hat folgendes Aussehen:

```
DEFINITION MODULE Parser;
export …
PROCEDURE Parse(): CARDINAL;
PROCEDURE CloseParser;
END Parser.

IMPLEMENTATION MODULE Parser;
global …
PROCEDURE Parse(): CARDINAL; BEGIN
  BeginParser;
  Generierter Code aus rules, operdef
END Parse; …
PROCEDURE BeginParser; BEGIN begin END BeginParser;
PROCEDURE CloseParser; BEGIN close END CloseParser; …
END Parser.
```

Die zentrale Funktion `Parse` liefert als Resultat nur die Anzahl der bei der Parsierung entdeckten Syntaxfehler, so daß weitere aus der Parsierung resultierende Werte über globale Variablen zurückgegeben werden müssen.

2.3.3 Präprozessor Bnf

Durch Aufruf von Lalr mit einer bestimmten Option wird vor der Analyse der Parser-Spezifikation das Werkzeug Bnf aufgerufen, welches eine EBNF-Grammatik in eine äquivalente BNF-Grammatik transformiert. Bnf akzeptiert die Spezifikationssprache von Lalr und wandelt Regeln, die in EBNF vorliegen, nach einem festen Schema in BNF-Regeln um, indem zusätzliche Nichtterminale erzeugt werden. Dabei wird sichergestellt, daß Aktionen immer am Ende einer Regel stehen. Die EBNF-Konstrukte zur Iteration werden in linksrekursive BNF-Regeln umgewandelt.

Sei X ∈ V_N, X_1 und X_2 „neue" Nichtterminale sowie α, β, γ, δ beliebige EBNF-Satzformen, geschieht die Umwandlung nach folgendem Schema:

EBNF	*BNF-Umsetzung des Konstrukts*			
X: α [β] γ.	X: α X_1 γ.	X_1: .	X_1: β.	
X: α (β) γ.	X: α X_1 γ.	X_1: β.		
X: α β+ γ.	X: α X_1 γ.	X_1: X_2.	X_1: X_1 X_2.	X_2: β.
X: α β* γ.	X: α X_1 γ.	X_1: .	X_1: X_1 β.	
X: α | β.	X: α.	X: β.		
X: α β || γ δ.	X: α X_1 X_2 δ.	X_1: β.	X_2: .	X_2: X_2 γ X_1.
X: α { … } β.	X: α X_1 β.	X_1: { … }.		

Bei der Bezugnahme auf ein Attribut mittels der Pseudovariable $*i* in einer Aktion muß die Stellung des entsprechenden Regelsymbols innerhalb der umgewandelten BNF-Regel beachtet werden. Aus diesem Grunde eignen sich die EBNF-Konstrukte bis auf die Alternative (|) für den praktischen Einsatz zum Aufbau eines Syntaxbaums nicht. Bei der Verwendung des Alternativenkonstrukts sind die Ordinalitäten der Regelsymbole einer (EBNF-)Alternative und die Ordinalitäten der Regelsymbole der rechten Seite der erzeugten BNF-Regel identisch.

Beispiel (mit X ∈ V_N, a*i*, b*i* ∈ V_T und A, B Aktionen in Werkzeugzielsprache):

EBNF	*BNF*
X: a1 a2 a3 {A} | b1 b2 b3 {B} .	X: a1 a2 a3 {A} .
	X: b1 b2 b3 {B} .

2.4 Ag – Generator für Attributevaluatoren

Zur Beschreibung der Analyse der statischen Semantik werden zumeist Attributgrammatiken verwendet. Das Werkzeug Ag akzeptiert sowohl geordnete Attributgrammatiken (OATGs), als auch Attributgrammatiken höherer Ordnung (HATGs). Der erzeugte Evaluator entspricht einem OATG-Treewalk-Evaluator. Bei einer Attributgrammatik werden Attribute mit den Symbolen einer kontextfreien Grammatik, der *zugrundeliegenden Grammatik*, verbunden. Beim vorliegenden Werkzeug ist die zugrundeliegende Grammatik durch die Spezifikation des abstrakten Syntaxbaums gegeben, indem die Produktionen den Knotentypdefinitionen entsprechen. Den Knotentypen werden somit Attribute zugeordnet. Die Attributvorkommen zu einem Knotentyp umfassen die Attribute dieses Knotentyps und die Attribute aller seiner Kinder. Attributexemplare (oder auch Attributinstanzen) sind die Attribute eines Syntaxbaumknotens. Sie erhalten zur Evaluationszeit die statische, semantische Information, indem der Attributevaluator gemäß den Attributauswertungsregeln ihre Werte berechnet.

Da die Baumsichtweise der abstrakten Syntax beibehalten wird, soll die folgende Gegenüberstellung den Bezug zwischen zugrundeliegender Grammatik und der beim Werkzeug Ast eingeführten Baumterminologie verdeutlichen.

Baumspezifikation	***Kontextfreie Grammatik***
Namen der Knotentypen mit Kindern	Nichtterminale
Namen der Knotentypen ohne Kinder (Blattknoten)	Terminale
Name des Wurzelknotentyps	Startsymbol
Knotentypdefinitionen	Produktionen

Das Werkzeug Ag (Attribute evaluator Generator) erzeugt aus einer Attributgrammatik einen Attributevaluator. Die Spezifikation der zugrundeliegenden kontextfreien Grammatik erfolgt mittels der vom Werkzeug Ast benutzten Spezifikationssprache, in der den Knotentypen bereits Attribute zugeordnet werden können. Als Erweiterung der Ast-Spezifikationssprache können in einer Ag-Spezifikation Attributauswertungsregeln definiert werden.

2.4.1 Spezifikationssprache

Die Spezifikationssprache von Ag ist eine Erweiterung der Sprache des Werkzeugs Ast. Die Erweiterungen dienen im wesentlichen zur Formulierung der Attributauswertungsregeln (Action).

```
Specification   =  … Spec … .
Spec            =  TREE [ ident ] EVAL … Rules .
Rules           =  RULE { Type }.
Type            =  ident "=" { Field } … "." .
Field           =  … | "{" { Action } "}" .
```

Module

Ag erlaubt ebenfalls die Aufteilung der Spezifikation in mehrere Module, die auch auf verschiedene Dateien verteilt werden können. Auf diese Weise ist es möglich, die Spezifikation für das Werkzeug Ast unverändert zu übernehmen und zusammen mit weiteren Modulen für die Attributauswertung als Ag-Spezifikation zu nutzen.

```
Specification   =  Spec Modules | MODULE ident Spec END ident Modules .
Modules         =  { MODULE ident Spec END ident } .
```

Attribute

Die Attribute der Attributgrammatik sind genau die im Abschnitt über das Werkzeug Ast beschriebenen Attribute.

```
Rules    =  RULE { Type } .
Type     =  ident "=" { Field } … "." .
Field    =  … | Attr | … .
Attr     =  "[" ident ":" ident … "]" .
```

Soll ein Attribut mehreren Knotentypen, die nicht in einer Erweiterungsrelation stehen, zugeordnet werden, kann in einem Deklarationsabschnitt hinter dem Schlüsselwort DECLARE dieses Attribut einer Menge von Knotentypen zugeordnet werden.

```
Spec     =  TreeCodes EvalCodes PropPart DeclPart Rules .
DeclPart =  DECLARE { Decl } .
Decl     =  { ident } "=" { Field } "." .
```

Beispiel:

```
DECLARE
   Stmts Cases GuardedStmts = [ IsInLoop : BOOLEAN INHERITED ].
```

Eigenschaften

Attribute können Eigenschaften besitzen, die diesen entweder lokal (bei der Knotentypdeklaration) oder global (im Spezifikationsabschnitt PROPERTY) zugewiesen werden (siehe Ast). Für den Attribut-Evaluator wichtig sind die Eigenschaften SYNTHESIZED, INHERITED, THREAD und VIRTUAL.

```
Field    =  … | Attr | … .
Attr     =  "[" ident ":" ident { Property } "]" .
Property =  … | SYNTHESIZED | INHERITED | THREAD | VIRTUAL | … .
```

Die Attributeigenschaften INHERITED und SYNTHESIZED entsprechen den bei Attributgrammatiken üblichen Eigenschaften *ererbt* bzw. *abgeleitet*. Die Deklaration eines Attributs mit der Eigenschaft THREAD ist eine Kurzform für die Definition von zwei Attributen, deren Namen sich aus dem angegebenen Namen und dem Suffix In bzw. Out zusammensetzt. Das Attribut mit dem Suffix In besitzt dabei die Eigenschaft INHERITED, dasjenige mit dem Suffix Out die Eigenschaft SYNTHESIZED. Bei der Applikation eines solchen Attributs ist der generierte volle Name anzugeben. Eine Deklaration der Art

```
Imports = [ Table : tOB THREAD ] .
```

entspricht also der Definition

```
Imports = [ TableIn : tOB INHERITED   ]
          [ TableOut: tOB SYNTHESIZED ] .
```

Attribute mit der zusätzlichen Eigenschaft VIRTUAL dienen ausschließlich zur Beeinflussung der Auswertungsreihenfolge von Attributen in den Attributauswertungsregeln (s.u.). Für sie wird weder Speicherplatz belegt noch werden evtl. angegebene Auswertungsregeln vom Evaluator ausgeführt.

Attributauswertungsregeln

Für alle definierenden Vorkommen von Attributen eines Knotentyps (Produktion der zugrundeliegenden Grammatik) müssen Attributauswertungsregeln angegeben werden. Sie werden, eingerahmt durch geschweifte Klammern {}, in die Liste der Felder eines Knotentyps eingefügt.

```
Type  =  ident "=" { Field } … .
Field =  … Attr | "{" { Action } "}" .
```

Die Attributauswertungsregeln bestimmen die funktionalen Abhängigkeiten zwischen den Attributvorkommen eines Knotentyps. Die Reihenfolge der textuellen Angabe der Attributauswertungsregeln hat keinerlei Einfluß auf die Reihenfolge ihrer Auswertung zur Evaluationszeit. Die Auswertungsreihenfolge ergibt sich aus den funktionalen Abhängigkeiten der in den Attributsauswertungsregeln applizierten Attributvorkommen.

Innerhalb einer Attributauswertungsregel wird über Attributbezeichner auf ein Attributvorkommen (bzw. seinen Wert) Bezug genommen. Dieser Attributbezeichner besteht einfach aus dem Attributnamen, falls es ein Attributvorkommen des Knotentyps ist, dem die Attributauswertungsregel zugeordnet ist. Für den Bezug auf ein Attributvorkommen eines Kinds des Knotentyps muß der Attributname mit dem Selektornamen (gefolgt von einem Doppelpunkt) präfiziert werden. Zur Unterscheidung eines Objekts (z. B. einer vom Spezifikateur definierten Prozedur) von einem gleichnamigen Attributvorkommen (oder Kind) in einer Attributauswertungsregel dient das vor den Objektnamen geschriebene Escape-Zeichen "\".

```
Attribute  =  [ ident ":" ] ident .
```

Für die Formulierung von Attributauswertungsregeln stehen mehrere Regelarten zur Verfügung:

```
Action     =  Assignment | Copy | AssignCode | Condition | After | Before .
```

Zuweisung: Diese Regelart ermöglicht die Übergabe eines Werts, der aus der Auswertung eines Ausdrucks resultiert, an ein Attribut. Der Ausdruck wird in der Werkzeugzielsprache formuliert und darf Attributbezeichner enthalten.

```
Assignment =  Attribute ":=" TargetExpr ";" .
TargetExpr =  „Ausdruck der Werkzeugzielsprache“ .
```

Beispiel:

```
Exprs        =                [ v:INTEGER SYNTHESIZED ]
<  Sum       =   x:Exprs
                 y:Exprs      { v := x:v + y:v; }.
   IntConst  = [ c:INTEGER ]  { v := c;         }.
>.
```

Kopie: Das Kopieren ist eine spezielle Form des Zuweisens: Als Ausdruck ist nur ein Attributbezeichner erlaubt. Sie ermöglicht Ag die Generierung ,,effizienteren“ Codes.

```
Copy  =  Attribute ": -" Attribute ";" .
```

Beispiel:

```
Exprs       =        [ v:INTEGER SYNTHESIZED ]
<  …
   Identity = Exprs  { v :- Exprs:v; }.
>.
```

Zuweisungscode: Es gibt Situationen, in denen es nicht möglich ist, die Berechnung eines Attributs ausschließlich durch einen einzelnen Ausdruck der Werkzeugzielsprache zu beschreiben. Beispielsweise kann die Verwendung einer Fallunterscheidung notwendig sein, oder der strukturierte Typ des zu berechnenden Werts kann es verbieten, den Wert als Resultat einer Funktionsprozedur zu übergeben. Aus diesem Grund kann die Berechnung im Zuweisungscode durch eine Anweisungsfolge der Werkzeugzielsprache beschrieben werden.

```
AssignCode =  Attributes ":=" "{" TargetStmts "}" ";" .
Attributes    =  Attribute { "," Attribute }.
TargetStmts =  „Anweisungsfolge der Werkzeugzielsprache“ .
```

Der Zuweisungscode kann auch die Berechnung mehrerer Attribute beschreiben. Die eigentliche Zuweisung des Werts wird nicht generiert. Sie muß im Zuweisungscode explizit aufgeführt werden.

Beispiel:

```
Exprs          =          [ v:INTEGER SYNTHESIZED ]
<  …
   Conditional = e:Exprs
                 x:Exprs
                 y:Exprs  { v := { IF e:v#0 THEN v:=x:v;
                                            ELSE v:=y:v; END; };
                          }.
>.
```

Bedingung: Die eigentliche Überpüfung der statischen Semantik erfolgt in den Bedingungen, indem der Wert eines Ausdrucks getestet wird und in Abhängigkeit davon eine Anweisung oder Anweisungsfolge ausgeführt wird.

```
Condition    =  CHECK TargetExpr Statement .
Statement    =  "==>" TargetStmt
             |  "==>" "{" TargetStmts "}" .
TargetStmt   =  „Anweisung der Werkzeugzielsprache“ .
```

Die Anweisung oder Anweisungsfolge wird ausgeführt, falls der Ausdruck *FALSE* ergibt.

Beispiel:

```
Exprs        =          [ t:tTYPE SYNTHESIZED ]
<  Dyadic    = x:Exprs
               y:Exprs  { t :- x:t;
                          CHECK x:t = y:t
                          ==> WriteError("Not compatible"); }.
   IntConst  =          { t := IntType;  }.
   BoolConst =          { t := BoolType; }.
>.
```

Künstliche Abhängigkeiten: Die Reihenfolge der Attributberechnungen kann explizit angegeben werden.

```
After  = Attributes AFTER Attributes ";" .
Before = Attributes BEFORE Attributes ";" .
```

Erweiterungen

Der bereits beim Werkzeug Ast beschriebene Erweiterungsmechanismus betrifft auch die Auswertungsregeln: Ein erweiterter Knotentyp enthält (erbt) alle Auswertungsregeln (sowie Kinder und Attribute) seines Basisknotentyps. Eine geerbte Auswertungsregel kann jedoch im erweiterten Knotentyp redefiniert werden.

Beispiel:

```
Exprs         =                  [ t:tTYPE SYNTHESIZED ]
<  Dyadic     = x:Exprs y:Exprs  { t :- x:t; }
   <  Add     = .
      Sub     = .
      Compare =                  { t := BoolType; }.
   >.
   IntConst   =                  { t := IntType;  }.
   BoolConst  =                  { t := BoolType; }.
>.
```

Hier erben die vom Knotentyp `Dyadic` erweiterten Knotentypen `Add` und `Sub` die Auswertungsregel für das Attribut `t`, die im Knotentyp `Compare` redefiniert wird.

Fehlende Auswertungsregeln

Ist für ein Attribut *a* in einem Knotentyp *n* keine Auswertungsregel angegeben und wird keine Auswertungsregel für *a* von einem Basisknotentyp geerbt, generiert Ag eine Kopierregel gemäß folgendem Schema:

- Hat ein Kind *c* das Attribut *a* mit der Eigenschaft `INHERITED` und nicht `THREAD` und besitzt der Knotentyp ein Attribut *a*, wird die Regel *c:a :- a* hinzugefügt.
  Bsp.: `c = [ a INHERITED ].  n = [ a ] … c { c:a :- a; }.`
- Hat der Knotentyp das Attribut *a* mit der Eigenschaft `SYTHESIZED` und nicht `THREAD` und existiert ein Kind *c* mit dem Attribut *a*, wird die Regel *a :- c:a* hinzugefügt. Falls mehrere Kinder existieren, die das Attribut *a* besitzen, wird das letzte Kind gewählt.
  Bsp.: `c = [ a ].  n = [ a SYNTHESIZED ] … c1:c … cn:c { a :- cn:a; }.`
- Hat der Knotentyp das Attribut *a* mit der Eigenschaft `THREAD` und existiert kein Kind mit dem Attribut *a*, das die Eigenschaft `THREAD` hat, wird die Regel *aOut :- aIn* hinzugefügt.
  Bsp: `n = [ a THREAD ] { aOut :- aIn; }.`
- Hat ein Kind *c* das Attribut *a* mit der Eigenschaft `THREAD` und existiert links von ihm kein anderes Kind, das das Attribut *a* mit der Eigenschaft `THREAD` besitzt, und besitzt der Knotentyp ebenfalls das Attribut *a* mit der Eigenschaft `THREAD`, wird die Regel *c:aIn :- aIn* hinzugefügt.
  Bsp.: `c = [ a THREAD ].  n = [ a THREAD ] … c1:c … cn:c { c1:aIn :- aIn; }.`
- Hat ein Kind c das Attribut *a* mit der Eigenschaft `THREAD` und gibt es links von ihm ein Kind *b*, das ebenfalls das Attribut *a* mit der Eigenschaft `THREAD` besitzt, wird die Regel *c:aIn :- b:aOut* hinzugefügt. Falls mehrere solcher Kinder links von Kind *c* existieren, wird das letzte von ihnen gewählt.
  Bsp.: `c = [ a THREAD ].  n = [ a THREAD ] … b:c … c:c { c:aIn :- b:aOut; }.`
- Hat der Knotentyp das Attribut *a* mit der Eigenschaft `THREAD` und existiert ein Kind *c*, das ebenfalls das Attribut *a* mit der Eigenschaft `THREAD` besitzt, wird die Regel *aOut :- c:aOut* hinzugefügt. Falls mehrere Kinder existieren, die das Attribut *a* mit der Eigenschaft `THREAD` besitzen, wird das letzte Kind gewählt.
  Bsp.: `c = [ a THREAD ].  n = [ a THREAD ] … c1:c … cn:c { aOut :- cn:aOut; }.`

Bei der Generierung des Evaluators gibt das Werkzeug Ag die Anzahl der eingefügten Kopierregeln jeweils für die Attributeigenschaften SYNTHESIZED, INHERITED und THREAD aus.

2.4.2 Generierter Code

Eine Ag-Spezifikation umfaßt üblicherweise mindestens zwei in einem Dokument zusammengefaßte Module: Die Spezifikation der abstrakten Syntax als zugrundeliegende Grammatik und die Spezifikation des Evaluators.

```
MODULE AbstractSyntax TREE ThisTree
…
END AbstractSyntax

MODULE Evaluator TREE ThisTree EVAL
IMPORT  { import }
EXPORT  { export }
GLOBAL  { global }
LOCAL   { local  }
BEGIN   { begin  }
CLOSE   { close  }
DECLARE …
RULE    rules
END Evaluator
```

Hieraus erzeugt Ag ein Modula-2 Modul, bestehend aus Definitions- und Implementationsteil:

```
DEFINITION MODULE Eval; IMPORT ThisTree;
import
export
PROCEDURE Eval(yyt: ThisTree.tThisTree);
PROCEDURE BeginEval;
PROCEDURE CloseEval;
END Eval.
```

Eval ist die eigentliche Auswertungsprozedur. Sie bekommt die Wurzel des Syntaxbaums, dessen Attributexemplare ausgewertet werden. BeginEval bzw. CloseEval dient dem Auf- bzw. Abbau zusätzlicher, durch den Spezifikateur in den Abschnitten EXPORT und GLOBAL definierter Datenstrukturen.

```
IMPLEMENTATION MODULE Eval; …
global
PROCEDURE Eval(yyt: ThisTree.tThisTree);
  BEGIN yyVisit1(yyt); END Eval;
PROCEDURE yyVisit1(yyt: ThisTree.tThisTree);
  local
BEGIN
  Generierter Code aus rules
END yyVisit1;
PROCEDURE BeginEval; BEGIN begin END BeginEval;
PROCEDURE CloseEval; BEGIN close END CloseEval;
END Eval.
```

Entsprechend der maximalen Anzahl von Besuchen eines Knotens erzeugt Ag mehrere Visit-Prozeduren (yyVisit1, yyVisit2, …), die alle den im Abschnitt LOCAL angegebenen Code enthalten.

2.5 Puma – Generator für Funktionen auf Bäumen

Das Werkzeug Puma (Pattern MAtching and Unification) ermöglicht eine kompakte Spezifikation von Funktionen, die (Teil-)Bäume in Abhängigkeit ihrer Struktur klassifizieren. Diese Funktionen werden nachfolgend Puma-Funktionen genannt. Die Klassifikation der Bäume erfolgt über Mustererkennung. Für eine Klasse werden Aktionen bzw. Resultatswerte angegeben, die ausgeführt bzw. geliefert werden, falls die als Argumente übergebenen Teilbäume in die zugehörige Klasse fallen. Funktionsresultate sind wiederum Teilbäume oder von einem Typ der Werkzeugzielsprache. Teilbäume sind durch eine Ast-Spezifikation definiert.

Puma führt eine Überprüfung der Typisierung der Puma-Funktionen durch. Als Typen stehen Knotentypen und Typen der Werkzeugzielsprache zur Verfügung. Ein Typ *Ta* wird als kompatibel zu einem Typ *Tf* bezeichnet, falls *Ta* und *Tf* Werkzeugzielsprachentypen sind und *Ta* und *Tf dieselben* Typen sind, oder falls *Ta* und *Tf* Knotentypen sind und *Ta* eine Erweiterung von *Tf* ist.

2.5.1 Spezifikationssprache

Eine Puma-Spezifikation umfaßt mehrere verschiedene Abschnitte. Im Kopf der Spezifikation werden neben der Angabe des Namens des zu generierenden Moduls (hinter dem Schlüsselwort TRAFO), dem Schlüsselwort TREE folgend, die Namen der Baumspezifikationen aufgeführt, deren Teilbäume als Argumente verwendet werden. Der Exportabschnitt (EXPORTS) listet die Namen der exportierten Puma-Funktionen auf.

```
Specification  =  TrafoName TreePart ExportPart … Functions .
TrafoName      =  [ TRAFO ident ].
TreePart       =  [ TREE ident { "," ident } ].
ExportPart     =  [ EXPORTS { ident } ].
TargetCode     =  "{" {character} "}" .
```

Dem Kopf der Spezifikation folgt die Definition der Puma-Funktionen. Diese bestehen aus einer oder mehreren Regeln, die Muster, Ausdrücke und Anweisungen beinhalten können. Puma-Funktion können Eingabe- und Ausgabeparameter besitzen.

```
Functions      =  … { Rule }.
Rule           =  Patterns … "." .
```

Muster

Der einer Puma-Funktion zugrundeliegende Mechanismus ist die Mustererkennung. Ein Muster definiert eine Klasse von Bäumen oder Werten eines Werkzeugzielsprachentyps. Im weiteren werden Werte eines Werkzeugzielsprachentyps einfach nur als Werte bezeichnet; Teilbäume und Werte werden Objekte genannt. Ein Muster *trifft* ein Objekt, falls das Objekt Element der durch das Muster definierten Klasse ist. Die Mustererkennung stellt dies fest.

Ein Muster ist entweder ein Ausdruck (s.u.), das Symbol _, eine Marke oder ein Knotenkonstruktor, der mit einer *ungebundenen* Marke präfiziert sein kann. Eine Marke besteht lediglich aus einem Namen und wird nur bei ihrem erstmaligen Auftreten (innerhalb einer Regel, s.u.) als ungebunden bezeichnet, ansonsten als gebunden. Der Knotenkonstruktor wird durch Angabe des Knotentypnamens geschrieben. Ihm muß eine in runden Klammern stehende Liste von (Unter-)Mustern folgen. Die Anzahl der Listenelemente muß gleich der Anzahl der Felder des Knotentyps sein. Die Elementtypen müssen kompatibel zu den entsprechenden Feldtypen sein.

```
Patterns =  [ Pattern { "," Pattern } ].
Pattern  =  Expr | "_" | Label
         |  [ Label ":" ] ident "(" [Patterns] ")" .
Label    =  ident .
```

Beispiele für Muster für Knoten vom Baumtyp `Objects` (siehe Beispiel in 2.2.1, S. 47):

```
TypeRepr(_)
IntegerTypeRepr(_)
ArrayTypeRepr(s,0,_)
ArrayTypeRepr(_,_,Te:TypeReprs)
```

Das *Treffen* eines Objekts durch ein Muster kann wie folgt definiert werden:

- Das Muster, das aus einem Ausdruck besteht, trifft genau den Wert, der sich durch die Auswertung des Ausdrucks ergibt.
- Das Muster _ trifft jedes beliebige Objekt.
- Das Muster, das aus einer ungebundenen Marke besteht, trifft jedes beliebige Objekt.
- Das Muster, das aus einer gebundenen Marke *m* besteht, trifft genau das Objekt, welches (strukturell) gleich (vgl. S.49) dem Objekt ist, das durch die ungebundene Marke *m* getroffen wurde.
- Das Muster, das aus einem Konstruktor für den Knotentyp *Tk* besteht, trifft einen Teilbaum mit dem Wurzelknoten *w* vom Knotentyp *Tw*, falls *Tw* eine Erweiterung von *Tk* ist und alle Untermuster des Konstruktors die korrespondierenden Felder von *w* treffen.

Für die Formulierung von Mustern stehen Abkürzungen zur Vefügung: Mehrere am Ende einer Musterliste aufgeführte Muster, die nur aus dem Symbol _ bestehen, können durch das Symbol `..` ersetzt werden. Hat ein Konstruktor eine Liste von Untermustern mit der Form `(..)`, kann diese weggelassen werden.

```
Patterns =  Pattern { "," Pattern } [ "," ".." ] | ".." .
Pattern  =  Expr | "_" | Label
         |  [ Label ":" ] ident [ "(" [Patterns] ")" ].
```

Eine Liste von Mustern *trifft* genau dann, wenn alle Muster der Liste *treffen*.

Ausdrücke

Ein Ausdruck beschreibt die Berechnung eines Objekts. Er orientiert sich syntaktisch an der Syntax der Ausdrücke der Werkzeugzielsprache, kann aber auch Knotenkonstruktoren, Applizierungen von Puma-Funktionen, Marken und Code der Werkzeugzielsprache beinhalten.

```
Expr       =  [Expr] Operator Expr | "(" Expr ")" | number | string
           |  ident "(" [Exprs] ")"
           |  FuncAppl
           |  Label
           |  "_"
           |  TargetCode .
FuncAppl   =  ident "(" [Exprs] ["=>" Patterns] ")"
Exprs      =  Expr { "," Expr }.
Operator   =  „Operatoren der Werkzeugzielsprache“.
```

Der Ausdruck bestehend aus dem Symbol _ liefert ein undefiniertes Ergebnis und kann benutzt werden, falls keine Berechnung erfolgen soll.

Ein Knotenkonstruktor wird durch Angabe des Knotentypnamens, gefolgt von einer in runden Klammern stehenden Liste von Unterausdrücken, formuliert. Sein Ergebnis ist der erzeugte

Baumknoten, dessen Felder mit den Objekten der Unterausdrücke besetzt werden, woraus folgt, daß die Typen der Unterausdrücke kompatibel zu den entsprechenden Feldtypen sein müssen. Bei der Applizierung einer Puma-Funktion muß für jeden formalen Eingabeparameter ein Ausdruck und für jeden formalen Ausgabeparameter ein Muster angegeben werden. Die Ausgabeparameter werden von den Eingabeparametern durch das Symbol `=>` getrennt. Die Typen der Ausdrücke bzw. Muster müssen kompatibel zu den Typen der entsprechenden formalen Parameter sein. Das Ergebnis der Applizierung ist das Resultat der Puma-Funktion. Das Ergebnis einer Marke *m* ist das Objekt, das durch die ungebundene Marke *m getroffen* wurde.

Für die Formulierung von Ausdrücken stehen ebenfalls Abkürzungen zur Vefügung: Mehrere am Ende einer Ausdrucksliste aufgeführte Ausdrücke, die nur aus dem Symbol _ bestehen, können durch das Symbol `..` ersetzt werden. Hat ein Konstruktor eine Liste von Unterausdrücken mit der Form `(..)`, kann diese weggelassen werden.

```
Exprs =  Exprs { "," Expr } [ "," ".." ] | ".." .
Expr  =  … | ident [ "(" [Exprs] ")" ] | … .
```

Ein Ausdruck *trifft* genau dann, wenn alle Applizierungen von Puma-Funktionen, die der Ausdruck beinhaltet, *treffen*. Eine Puma-Funktionsapplizierung *trifft* genau dann, wenn alle seine Muster die korrespondierenden, von der Puma-Funktion über die Ausgabeparameter gelieferten Objekte *treffen*.

Anweisungen

Anweisungen dienen der Angabe von Bedingungen, dem Aufruf weiterer Puma-Prozeduren (s.u.), der Zuweisung von Objekten an Felder eines Knotens, der Steuerung des Kontrollflusses in einer Puma-Funktion und der Formulierung beliebigen anderen Verhaltens durch Anweisungen der Werkzeugzielsprache.

```
Functions  =  … Statement … .
Statement  =  Expr
           |  ident "(" [Exprs] ["=>" Patterns] ")"
           |  ( ident | Label ) ":=" Expr
           |  REJECT
           |  FAIL
           |  TargetCode .
```

Eine Bedingung muß ein Ausdruck sein, der ein Ergebnis vom Typ `BOOLEAN` liefert. Für einen Prozeduraufruf gelten bezüglich der Parameter die gleichen Regeln, wie bei der Applikation einer Puma-Funktion.

Auch eine Anweisung kann *treffen* oder nicht. Eine Bedingung *trifft* nur, falls ihre Auswertung `TRUE` liefert. Ein Prozeduraufruf *trifft*, wie bei einer Puma-Funktionsapplikation, falls die Muster die Ausgabeparameterwerte *treffen*. Das *Treffen* einer Zuweisung ergibt sich aus dem *Treffen* ihres Ausdrucks. `REJECT` *trifft* nie. Die Anweisung `FAIL` führt zur unmittelbaren Termination der gesamten Puma-Funktion und darf nicht in *echten Funktionen* (s.u.) verwendet werden. Code der Werkzeugzielsprache *trifft* immer!

Puma-Funktionen

Die Spezifikationssprache gestattet es, drei verschiedene Arten von Puma-Funktionen zu formulieren: *Echte Funktionen* liefern einen Wert und werden als Bestandteil eines Ausdrucks

verwendet. *Prädikate* sind Funktionen mit dem Resultatstyp BOOLEAN. *Prozeduren* liefern keinen Wert und werden als Anweisungen benutzt.

```
Functions  =  Header [ LOCAL TargetCode ] { Rule }.
Header     =  PROCEDURE ident "(" Parameters ")"
           |  FUNCTION  ident "(" Parameters ")" Type
           |  PREDICATE ident "(" Parameters ")".
Parameters =  [ Param {"," Param} ] [ "=>" Param {"," Param} ].
Param      =  [ ident ":" ] Type .
Type       =  ident .
```

Die Parameter einer Puma-Funktion werden durch das Symbol => in Eingabe- und Ausgabeparameter unterteilt. Sie müssen nicht benannt werden. Als Typen sind Knotentypen oder Typen der Werkzeugzielsprache erlaubt, wobei für die Wahl des Resultatstyps einer Funktion die hierfür geltenden Regeln der Werkzeugzielsprache beachtet werden müssen.

Regeln

Eine Regel besteht aus einer Liste von Mustern und Aktionen. Die Aktionen können sich aus Ausdrücken, einem Return-Ausdruck und einer Folge von Anweisungen zusammensetzen.

```
Rule     =  Patterns Actions "." .
Patterns =  [ Pattern { "," Pattern } ].
Exprs =  Exprs { "," Expr } [ "," ".." ] | ".." .
Actions  =  [ "=>" Exprs ] [ RETURN Expr ] "?" { Statement ";" }.
```

Für jeden formalen Eingabeparameter muß ein Muster und für jeden formalen Ausgabeparameter ein Ausdruck vorhanden sein. Ein Muster bzw. Ausdruck ist über seine Position in der Liste dem entsprechenden formalen Parameter zugeordnet. Der Typ des Musters bzw. Ausdrucks muß kompatibel zum Typ des formalen Parameters sein. Ebenso muß der Typ des Return-Ausdrucks kompatibel zum Resultatstyp sein.

Eine Regel *trifft* genau dann, wenn alle ihre Muster, Ausdrücke und Anweisungen *treffen* und keine textuell vor ihr stehende Regel *trifft*. Wenn alle Muster der Musterliste der Regel *treffen*, werden die Anweisungen ausgeführt und die Ausdrücke ausgewertet. Erst wenn alle Anweisungen und Ausdrücke (einschließlich des Return-Ausdrucks bei einer echten Funktion) *getroffen* haben, terminiert die Puma-Funktion. Bei Termination einer echten Funktion wird der durch den Return-Ausdruck der Regel berechnete Wert zurückgeliefert. Für den Fall, daß keine Regel einer Funktion *trifft*, ist das Verhalten abhängig von der Art der Funktion: Eine Prozedur terminiert, eine echte Funktion erzeugt einen Laufzeitfehler und ein Prädikat liefert FALSE.

Beispiele für Puma-Funktionen:

```
FUNCTION DimOfArrayType ( TypeReprs ) INTEGER
  ArrayTypeRepr(_,_,e) RETURN 1+DimOfArrayType(e) ?.
  _                    RETURN 0                   ?.

PREDICATE IsCompatible ( TypeReprs , TypeReprs )
  BooleanTypeRepr      , BooleanTypeRepr      ?.
  IntegerTypeRepr      , IntegerTypeRepr      ?.
  ArrayTypeRepr(_,n,a) , ArrayTypeRepr(_,n,b) ? IsCompatible(a,b);.
```

```
PROCEDURE WrType ( TypeReprs )
  EmptyTypeRepr        ? { WrStr("<emptytype>");      };.
  BooleanTypeRepr      ? { WrStr("BOOLEAN");          };.
  IntegerTypeRepr      ? { WrStr("INTEGER");          };.
  ArrayTypeRepr(_,n,e) ? { WrStr("ARRAY "); WrInt(n);
                           WrStr(" OF "); WrType(e);  };.
```

Das Werkzeug Puma eignet sich insbesondere auch zur Spezifikation von Baumtransformationsfunktionen. Das nachfolgende Beispiel zeigt eine Puma-Funktion zur Konstantenfaltung (siehe Beispiel in 2.2.1, S. 47) .

```
FUNCTION FoldExpr ( e:Exprs ) Exprs
         DyExpr ( p,op,IntConst(_,c1),IntConst(_c2) )
  RETURN IntConst ( p,Evaluate(op,c1,c2) ) ?.
         _
  RETURN e ?.

FUNCTION Evaluate ( op:tOperator , v1:INTEGER , v2:INTEGER ) INTEGER
  {opAdd} , .. RETURN v1+v2 ?.
  {opSub} , .. RETURN v1-v2 ?.
```

2.5.2 Generierter Code

Das Werkzeug Puma erzeugt aus einer Spezifikation ein Modula-2 Modul, bestehend aus Definitions- und Implementationsteil. Für jede definierte Puma-Funktion wird eine Modula-2 Prozedur generiert.

Folgendes Beispiel zeigt, wie die verschiedenen Abschnitte der Spezifikation in den generierten Code einfließen.

```
TRAFO Beispiel TREE Baum EXPORTS Funktion
IMPORT { import }
EXPORT { export }
GLOBAL { global }
BEGIN  { begin  }
CLOSE  { close  }
FUNCTION Funktion ( a:Knoten => b:Anderer ) INTEGER
  LOCAL { local }
  rules
```

Hieraus wird folgender Modula-2 Code erzeugt:

```
DEFINITION MODULE Beispiel; IMPORT Baum;
import
export
PROCEDURE Funktion(a: Baum.tBaum; VAR b: Baum.tBaum): INTEGER;
PROCEDURE BeginBeispiel;
PROCEDURE CloseBeispiel;
END Beispiel.
```

```
IMPLEMENTATION MODULE Beispiel; IMPORT Baum;
global …
PROCEDURE Funktion(a: Baum.tBaum; VAR b: Baum.tBaum): INTEGER;
  local
BEGIN
  Generierter Code aus rules
END Funktion;
PROCEDURE BeginBeispiel; BEGIN begin END BeginBeispiel;
PROCEDURE CloseBeispiel; BEGIN close END CloseBeispiel;
BEGIN
  BeginBeispiel;
END Beispiel.
```

2.6 Reuse – Bibliothek wiederverwendbarer Funktionen

Neben den bisher vorgestellten Werkzeugen enthält die Compiler-Compiler-Toolbox noch eine Sammlung von Modula-2 Modulen, genannt Reuse (REUSablE software). Einige dieser Module werden von den verschiedenen Werkzeugen benötigt, d.h. die generierten Modula-2 Module importieren aus einigen der Reuse-Module. Im folgenden werden alle Reuse-Module zusammen mit einer Kurzbeschreibung aufgeführt. Dabei bilden die innerhalb von Code der Werkzeugzielsprache verwendeten Module den ersten Abschnitt der Aufstellung, gefolgt von den restlichen Modulen. Eine detaillierte Beschreibung der Modulschnittstellen kann der Dokumentation von GROSCH [1987] entnommen werden.

Modul	*Beschreibung*
`Strings`	Zeichenketten-Datentyp
`StringMem`	Zeichenketten-Speicherung
`Idents`	Namenstabelle – eindeutige Codierung von Zeichenketten
`Sets`	Ganzzahlenmengen (ohne Laufzeitüberprüfungen)
`IO`	Gepufferte Ein- und Ausgabe
`System`	Schnittstelle für Systemaufrufe
`SysCalls`	Zugriff auf Systemaufrufe
`Memory`	Dynamische Speicherverwaltung mit Freispeicherlisten
`Heap`	Dynamische Speicherverwaltung ohne Freispeicherlisten
`DynArray`	Dynamische und größenveränderliche Arrays
`Lists`	Listen beliebiger Objekte
`Texts`	Listen von Zeichenketten
`SetsC`	Ganzzahlenmengen (mit Laufzeitüberprüfungen)
`Relations`	Zweistellige Relationen über Ganzzahlen
`SystemIO`	Ungepufferte, binäre Ein- und Ausgabe
`StdIO`	Gepufferte Ein-/Ausgabe auf Standarddateien
`Layout`	Weitere Routinen für die Ein- und Ausgabe
`General`	Diverse Funktionen
`Checks`	Überprüfung der Systemaufrufe
`Times`	Zugriff auf die Prozessorzeit

2.7 Bewertung

Die Benutzung der Werkzeuge Rex und Lalr zur Spezifikation von Scanner und Parser gestaltete sich weitgehend problemlos, zumal sie sich in Syntax und Handhabung prinzipiell nicht wesentlich von anderen verbreiteten Werkzeugen unterscheiden. Demgegenüber fallen die Werkzeuge Ast und Puma doch etwas aus dem Rahmen der üblichen Übersetzerbauwerkzeuge. Jedoch können mit ihnen nach einer entsprechenden Eingewöhnungsphase relativ übersichtliche Spezifikationen erstellt werden. Zudem erlaubt das enge Zusammenspiel von Ast, Ag und Puma eine abstrakte Formulierung der semantischen Analyse.

Verständlicherweise werden die Möglichkeiten und Vorzüge der Werkzeuge der Compiler-Compiler-Toolbox in der einführenden Werkzeugbeschreibung von GROSCH und EMMELMANN [1990] teilweise übertrieben dargestellt. Der praktische Einsatz zeigte jedoch einige Punkte auf, die einer wertenden Anmerkung bedürfen.

Der erste Kritikpunkt betrifft die Beschreibung der Werkzeuge selbst. In jeder Werkzeugbeschreibung findet sich im Anhang eine Grammatik der Spezifikationssprache. Diese Grammatik ist wiederum in der Spezifikationssprache formuliert bzw. nutzt Konstrukte der Spezifikationssprache. Während die Grammatik der Sprache des Werkzeugs Rex trotz Verwendung der regulären Ausdrucksoperatoren `*` und `+` für die (einschließende) Iteration noch halbwegs verständlich ist, birgt die Formulierung der Grammatik zum Werkzeug Ast mittels der durch sie definierten Sprache immense Schwierigkeiten.

Die Einfügung von Kommentaren in das Spezifikationsdokument erfordert größte Sorgfalt. Kommentare werden normalerweise durch Kommentarklammern im C-Stil `/* */` eingeschlossen und dürfen nicht geschachtelt werden. Innerhalb von Code der Werkzeugzielsprache (`TargetCode`) muß jedoch die Kommentarkonvention der Werkzeugzielsprache beachtet werden, was im Fall von Modula-2 bedeutet, daß die Klammerung `(* *)` benutzt werden muß. Trotz dieser generellen Regel erlauben einige Werkzeuge (entgegen anderslautender Aussage in der Werkzeugbeschreibung) auch die Kommentierung in C-Notation innerhalb der besagten Abschnitte. Diese Kommentare werden zur Generierungszeit entfernt. Da dies nicht für alle Werkzeuge gilt, ist hier zusätzlich Vorsicht geboten.

Zur Benutzung eines Reuse-Moduls ist es erforderlich, den Namen des betreffenden Moduls in die Importliste des Spezifikationsdokuments einzufügen. In den erzeugten Modula-2 Quelltexten befinden sich jedoch bereits generierte Importierungen einiger Reuse-Module, die für die Realisierung der entsprechenden Funktionalität benötigt werden und zusätzlich nach eingestellter Werkzeugoption noch variieren. Somit ist die Kenntnis der generierten Importierungen unabdingbar, um Doppelimportierungen zu vermeiden, welche der Modula-2 Übersetzer der eingesetzten Entwicklungsumgebung als Fehler meldet. Eine ähnliche Problematik ergibt sich bei den Importierungen in einer auf mehrere (Ast-)Module verteilten Ast-Spezifikation: Trotz einer möglichen Benutzung in mehreren (Ast-)Modulen darf ein Modula-2 Modul nur in einem der entsprechenden `IMPORT`-Abschnitte auftauchen.

Obwohl der Scanner und der Parser die gleiche Kodierung für die ausgetauschten Symbole verwenden müssen, besitzen die entsprechenden Werkzeuge keinen Mechanismus, um dies automatisch sicherzustellen. Es liegt ausschließlich in der Verantwortung des Spezifikateurs, dem Rechnung zu tragen.

Wie bereits erwähnt, besitzt der Parser-Generator die Möglichkeit EBNF-Konstrukte zur Beschreibung der konkreten Syntax zu verwenden. Von diesen Konstrukten ist jedoch zum Aufbau des abstrakten Syntaxbaums nur die Alternative (|) sinnvoll einsetzbar (vgl. S.53).

Beim Werkzeug Puma ist die Bindung von Untermustern zu den entsprechenden Feldern ausschließlich über die Position realisiert, was besonders bei einer großen Anzahl von (möglicherweise über mehrere Module verteilten) Feldern eines Knotentyps schwer zu handhaben und dadurch fehleranfällig ist. Ein zusätzlicher Bindungsmechanismus, der die Angabe eines Feldnames zur direkten Bindung benutzt, wäre hierbei eleganter.

Die Wahl des Namens ,,Modul“ für einen speziellen Abschnitt in einer Ast-Spezifikation sorgt, wegen der Kollision mit dem Begriff für eine Übersetzungseinheit der Werkzeugzielsprache Modula-2, an manchen Stellen der Werkzeugbeschreibung für Verwirrung. Es besteht der Verdacht, daß die Sprache C gegenüber Modula-2 als Werkzeugzielsprache bevorzugt wird. Dieser Eindruck wird zudem noch durch die bereits erwähnte Konvention bezüglich der Kommentarklammerung verstärkt.

3. Implementierung

Die Implementierung wurde unter UNIX (SunOS 4.1.3) auf einer SparcStation erstellt. Als Entwicklungsumgebung wurde der Modula-2 Übersetzer Mocka (MOdula-2 Compiler KArlsruhe, Version 9101) eingesetzt.

3.1 Beschreibung der Spezifikationsdateien

Das vorliegende Kapitel besteht aus Erläuterungen der einzelnen Spezifikationen. Zu ihrem Verständnis ist die Kenntnis der Werkzeuge und der Spezifikationstexte notwendig. Die Erläuterungen beziehen sich lediglich auf Sachverhalte, die weder einer ,,üblichen“ Benutzung der Werkzeuge entsprechen noch durch Spezifikationskommentare erwähnt sind. Jeder Abschnitt gliedert sich in allgemeine Kommentare bzw. Konventionen, nur lose zusammenhängende Einzelpunkte und einer Beschreibung der Werkzeugeinstellung bei Aufruf mit der Spezifikationsdatei.

3.1.1 Überblick

Die Analysephase eines Übersetzers realisiert funktionell die Transformation eines Zeichenstroms in eine interne Repräsentation, die in aller Regel einem attributierten, abstrakten Syntaxbaum entspricht. Abbildung 3 zeigt schematisch den Datenfluß der Transformation im generierten Übersetzer. 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. Aus dem so entstandenen Syntaxbaum wird dann in der Synthesephase Code der Zielsprache erzeugt. Die Codeerzeugung war nicht Bestandteil der Aufgabenstellung.

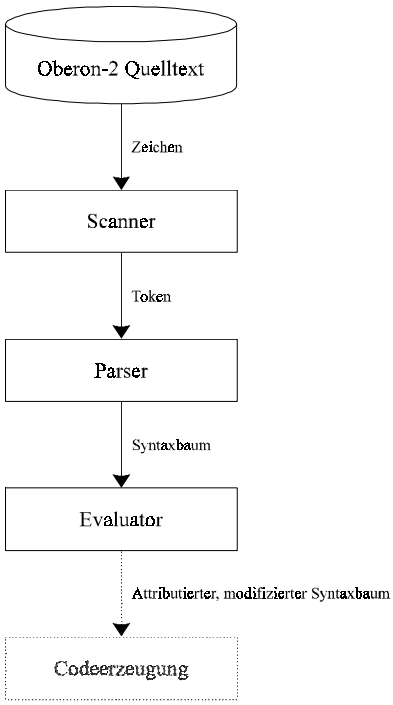

Abb. 3: Datenfluß im generierten Übersetzer.

Einen Überblick über die Komponenten der Übersetzerspezifikation und ihren funktionellen Zusammenhang gibt Abbildung 4. Das Hauptmodul analysiert die Kommandozeilenparameter und ruft das Treibermodul auf. Dieses stößt den Parser an und übergibt dem Evaluator nach erfolgreicher Parsierung den Syntaxbaum. Der Parser baut mittels Konstruktorfunktionen des Syntaxbaummoduls aus den Symbolen, die fortlaufend durch Scanner-Aufrufe geliefert werden, den Syntaxbaum auf. Der Scanner benutzt Datei-Lesefunktionen, um aus dem Oberon-2 Quelltext Symbole zu erzeugen. Der Evaluator besucht die Knoten des Syntaxbaums, transformiert bestimmte Teilbäume und berechnet Attribute mit Hilfe der Evaluatorhilfsfunktionen. Komplexere Attributtypen, wie z.B. Symboltabelleneinträge, und ihre Manipulationsfunktionen werden durch das Evaluatorobjektmodul zur Verfügung gestellt.

Lexikalische, syntaktische und semantische Fehler im Oberon-2 Quelltext führen zum Aufruf von Fehlerausgaberoutinen. Der Treiber aktiviert den Evaluator nur, wenn Scanner und Parser keine Fehler erkannt haben.

Die Bibliothek ,,Reuse“, das Modul für die Oberon-2 Datentypen und die Module der allgemeinen Hilfsfunktionen werden von vielen Modulen der beschriebenen Komponenten verwendet. Das Modul der Oberon-2 Datentypen kapselt die konkrete Repräsentation der zur Quellsprache gehörenden Basistypen und -operationen. Die allgemeinen Hilfsfunktionen ergänzen die Bibliothek um weitere Module grundlegender Funktionalität.

Folgende Konvention gilt für alle Spezifikationsdateien und erstellte Modula-2 Zusatzmodule: Es wurde weitgehend nur die Importierungsform verwendet, die auch in Oberon-2 zur Verfügung steht (,,`FROM`-lose“ Importierung). Durch die somit erzwungene Qualifizierung benutzter Objekte mit dem Modulnamen ließen sich Namenskollisionen leichter vermeiden. Aus diesem Grunde wurden kurze, nur aus Großbuchstaben bestehende Modulnamen gewählt. Da jedoch an einigen Stellen innerhalb der Spezifikationen keine qualifizierten Namen erlaubt sind (z.B. bei Ast die Feldtypen in Knotentypdefinitionen), wurden in diesen Spezifikationen die benötigten, importierten Objekte mit dem gleichen Namen neu deklariert.

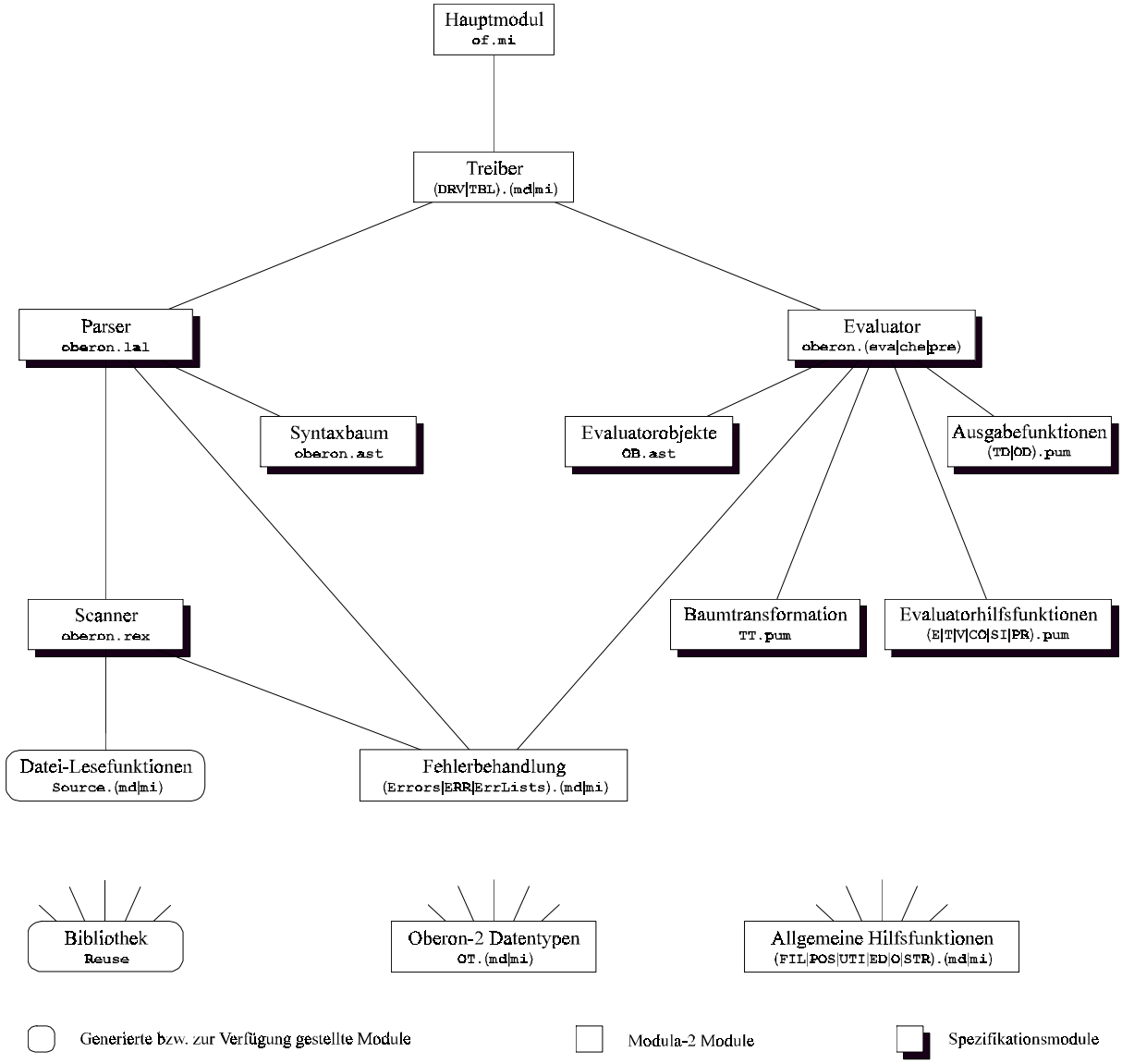

Abb. 4: Funktioneller Zusammenhang der Übersetzerkomponenten.

3.1.2 Scanner (oberon.rex)

Die vom Scanner zu liefernden Symbole sind in der Spezifikation als ganzzahlige Konstanten mit paarweise verschiedenen Werten im EXPORT-Abschnitt deklariert. Konzeptionell werden zwei Kategorien von Symbolen unterschieden: Symbole, welche einer konstanten Zeichenfolge entsprechen (Operatoren und Begrenzer) werden *einfache Symbole* genannt, und Symbole, die für eine ganze Klasse von Zeichenfolgen stehen (Zahlen, Zeichen, Zeichenketten und Namen) werden als *Klassensymbole* bezeichnet.

Scanner-Attribut

Für jedes Klassensymbol ist der Record-Typ tScanAttribute um eine Variante erweitert, deren Feld zur Aufnahme eines Werts dient, der sich aus der zum Symbol gehörenden Zeichenfolge ergibt. Dabei dient das Feld Integer zur Aufnahme aller ganzzahligen Oberon-2 Typen. Zeichenketten werden im Feld String gespeichert, falls ihre Länge ungleich 1 ist. Zeichenketten der Länge 1 werden als Zeichenkonstanten im Feld Char abgelegt. Dies erleichtert die Handhabung der Gleichsetzung von Zeichenkonstanten und Zeichenketten der Länge 1 (vgl. Sprachdefinition von Oberon-2, S.4).

Exportierte Prozeduren

Da bei der Parsierung im Fehlerfall u.U. Symbole vom Parser erzeugt werden, dient die Prozedur ErrorAttribute dazu, daß bei Klassensymbolen auch das entsprechende Feld des Scanner-Attributs einen definierten Wert erhält. Die Prozedur TokenNum2TokenName wird lediglich bei einer speziellen Option des erzeugten Übersetzers aus dem Treibermodul DRV.mi aufgerufen. Diese Option erlaubt das Testen des Scanners.

Startzustände

Neben dem implizit definierten Startzustand STD existieren die Startzustände Comment, StrSQ (STRing Single Quoted) und StrDQ (STRing Double Quoted) für die Handhabung von Quelltextkommentaren und Zeichenketten in einfachen bzw. doppelten Hochkommas. Da innerhalb eines Quelltextkommentars oder einer Zeichenkette die Quelltextdatei nicht zu Ende sein darf, werden im EOF-Abschnitt Fehlermeldungen erzeugt, falls der Scanner sich im entsprechenden Startzustand befindet. Der Wechsel der Startzustände erfolgt gemäß des in Abbildung 5 skizzierten (Keller-)Automaten. Die Knoten entsprechen den Startzuständen, die Kanten den Übergängen zwischen diesen durch Aufruf der Prozedur yyStart. Die Markierungen der Kanten stellen die Bedingungen dar, unter denen ein Zustandswechsel stattfindet (?…) sowie die bei diesem Zustandswechsel auszuführenden Aktionen (!…).

Einfache Symbole

Für die Symbole mit konstanter Zeichenfolge sind reguläre Ausdrücke spezifiziert, die aus eben jener Zeichenfolge bestehen und bei deren Erkennung das entsprechende Symbol zurückgeliefert wird.

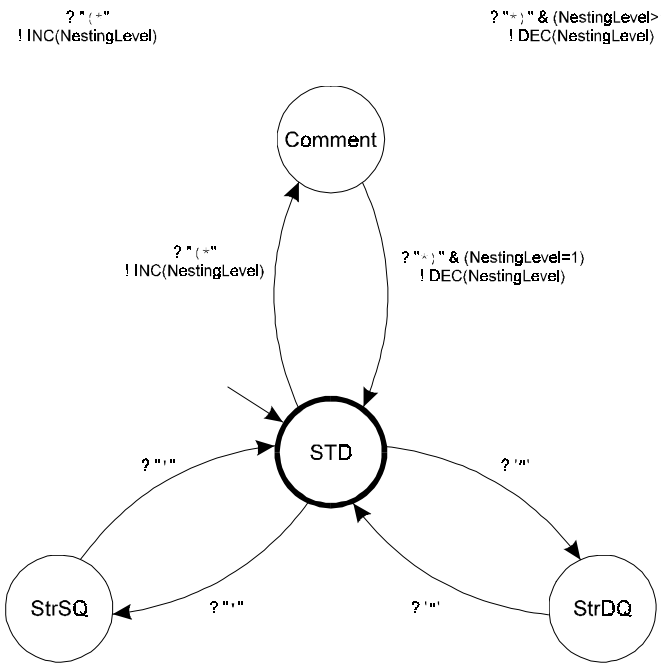

Abb. 5: Übergänge der Startzustände im Scanner.

Klassensymbole

Bei Erkennung eines Klassensymbols wird die zugehörige Zeichenfolge über die Prozedur `GetWord` in die `tString`-Variable `Repr` übertragen. Anschließend findet eine Umwandlung dieser Zeichenfolge in die entsprechende interne Repräsentation statt. Dies erfolgt durch Aufruf von Prozeduren, die vom Modul `UTI` und `OT` zur Verfügung gestellt werden. War die Umwandlung nicht erfolgreich, wird eine Fehlermeldung erzeugt. Diese Vorgehensweise gilt für die Symbole `IntegerToken`, `RealToken`, `LongrealToken` und `CharToken`, deren Erkennung jeweils in einer Regel (pro Darstellungsart) spezifiziert ist.

Demgegenüber werden für die Behandlung von Zeichenketten (`StringToken`) mehrere Regeln sowie jeweils ein eigener Startzustand eingesetzt. Dies erlaubt die Weiterführung der korrekten Quelltextposition durch Aufruf der generierten Prozedur `yyTab` bei Auftreten eines Tabulatorzeichens in einer Zeichenkette.

Um als Quelltextposition einer Zeichenkette die Position des einleitenden (einfachen oder doppelten) Hochkommas zu erhalten, ist bei allen weiteren Regeln für diese Zeichenkette die Positionsweiterrechnung durch ein angehängtes Minuszeichen `"-"` hinter dem Doppelpunkt unterbunden.

Die Regel für die Erkennung von Namen ist textuell nach den Regeln zur Erkennung der Schlüsselwortsymbole aufgeführt, da sonst gemäß den Kriterien zur Regelauswahl die Schlüsselwörter als Namen erkannt würden. Die letzte spezifizierte Regel dient der Erkennung genau derjenigen Zeichen, welche zum erlaubten Zeichensatz (s.u.) gehören, jedoch nicht Bestandteil eines gültigen Oberon-2 Quelltexts sind. Wird ein solches Zeichen erkannt, so wird eine Fehlermeldung ausgegeben.

Textuell hinter den vom Spezifikateur angegebenen Regeln werden vordefinierte Regeln automatisch angehängt. Diese Regeln lauten

```
" " :- { }
\t  :- { yyTab; }
\n  :- { yyEol(0); }
```

und ergänzen die Spezifikation um die korrekte Handhabung von Leerzeichen, Tabulatoren und Zeilenendezeichen.

Werkzeugaufruf

Die verschiedenen Einstellungen zur Konfiguration des Generierungsprozesses werden beim Aufruf von Rex über die Kommandozeile eingegeben. Von den bei GROSCH [1991a] beschriebenen Optionen wurden die folgenden benutzt:

`d`	Generierung eines Definitionsmoduls `Scanner.md`
`i`	Als erlaubter Zeichensatz des Quelltexts wird ISO 8-Bit Code verwendet

Ferner wurden bei der erstmaligen Generierung über die Option `s` zwei Unterstützungsmodule generiert, die Funktionalitäten zum Einlesen des Quelltexts sowie ein Treiberprogramm zur Verfügung stellen. Von diesen wurde jedoch nur das Modul zum Einlesen des Quelltexts (`Source.(md|mi)`) verwendet.

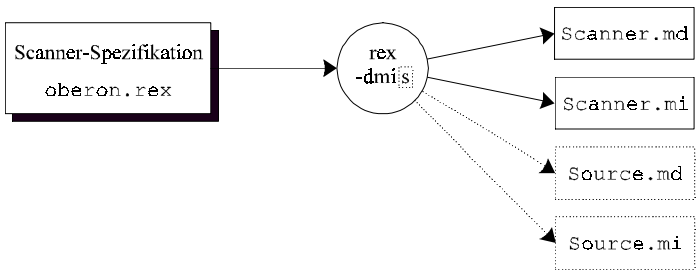

Abb. 6: Einsatz des Werkzeugs Rex zur Scanner-Erzeugung.

3.1.3 Abstrakter Syntaxbaum (oberon.ast)

Die Definition der Knotentypen des abstrakten Syntaxbaums ist im Spezifikationsdokument auf die zwei Module SyntaxTree und ExtendedSyntaxTree verteilt, damit die Eigenschaft INPUT für alle Knotentypfelder im Modul SyntaxTree global angegeben werden kann. Im Modul ExtendedSyntaxTree sind demgegenüber weitere Knotentypfelder ohne diese Eigenschaft aufgeführt.

Die gesamte Definition des abstrakten Syntaxbaums umfaßt eigentlich auch Teile der Spezifikationsdokumente oberon.eva, oberon.che und oberon.pre. In diesen Dokumenten sind einigen Knotentypen weitere Attribute zugeordnet, die jedoch für den Aufbau des abstrakten Syntaxbaums während der Parsierung ohne Relevanz sind. Sie müssen aber bei der Generierung in die jeweiligen Record-Deklarationen aufgenommen werden. Diesem Umstand wird durch den Werkzeugaufruf Rechnung getragen.

Der verstärkte Einsatz des Erweiterungsmechanismus ermöglicht eine Klassenbildung, die es erlaubt, die Anzahl der Attributauswertungsregeln zu minimieren, da erweiterte Knotentypen die Attributauswertungsregeln ihrer Basisknotentypen erben.

Listenbildung

Die Namensgebung von Knotentypen, deren Knoten als Elemente einer einfach verketteten Liste verwendet werden, erfolgt nach einem festen Schema:

```
Nodes     =
<  mtNode = .
   Node   = Next : Nodes
            …
          .
>.
```

Die eigentlichen Listenelemente sind vom Knotentyp Node (Singular). Das Ende der Liste ist ein Knoten vom Knotentyp mtNode (eMpTy node). Beide Knotentypen sind Ableitungen des Knotentyps Nodes (Plural), von welchem nie ein Knoten angelegt wird (abstrakter Knotentyp). Die Verkettung der Listenelemente erfolgt über das Kind Next vom Knotentyp Nodes, so daß ein nachfolgendes Element vom Knotentyp mtNode oder Node sein kann. Die Nutzinformation in einem Element wird in weiteren Feldern im Knotentyp Node gehalten.

Quelltextpositionen

Die in vielen Knotentypen definierten Attribute vom Typ tPosition werden während der Attributauswertung zur Positionsangabe bei Fehlermeldungen benötigt. Sie erhalten deswegen bereits bei der Parsierung ihren Wert.

Deklarationen

Die Repräsentation der Deklarationsabschnitte im abstrakten Syntaxbaum erfolgt als eine Liste von Deklarationseinheiten (DeclUnits), deren Elemente Listen von Deklarationen (Decls) sind. Ein Deklarationsabschnitt kann wegen der eingesetzten Parsierungstechnik nicht durch eine einfache Liste von Deklarationen repräsentiert werden.

Formale Parameter

Das Kind `Type` des Knotentyps `FormalPars`, welcher Bestandteil aller vom Knotentyp `Proc` abgeleiteten Knotentypen ist, repräsentiert den Resultatstyp einer Funktionsprozedur bzw. hat den Wert `mtType`, falls es sich um eine gewöhnliche Prozedur handelt. Dabei bedeutet ,,ein Kind *k* hat den Wert *n*“, daß das Kind *k* eine Referenz auf einen Knoten vom Knotentyp *n* besitzt.

Zeigertypen

Die Erweiterung von `PointerType` auf die Knotentypen `PointerToIdType`, `PointerToQualIdType` und `PointerToStructType` erleichtert die Behandlung von Vorwärtszeigern und rekursiven Typdefinitionen im Evaluator, indem hier von syntaktischer Information nicht abstrahiert wird, so daß die notwendigen Attributauswertungsregeln differenzierter angegeben werden können.

Qualifizierte Namen

Die Knotentypen `mtQualident` und `ErrorQualident` als Erweiterungen des Knotentyps `Qualidents` werden zur feineren Unterscheidung von qualifizierten Namen bei der Baumtransformation (`TT.pum`) der Knoten von aktuellen Parametern von vordeklarierten Prozeduren benötigt.

Typtest

Die Modellierung des Operators `IS` erfolgte **nicht** mittels eines `DyExpr`-Knotens mit `IsOper` als Wert des Kinds `DyOperator`. Vielmehr wurde ein eigener Knotentyp `IsExpr` definiert, der als ersten Operanden nur einen Bezeichner sowie als zweiten Operanden ausschließlich einen (qualifizierten) Namen erlaubt. Dies erleichtert im Evaluator die Handhabung von Oberon-2 Ausdrücken, die für einen Typ stehen.

Dyadische Operatoren

Das Attribut `Operator` im Knotentyp `DyOperator` bekommt bei der Knotenerzeugung als Wert die Knotentypkonstante des entsprechenden Knotentyps. Dieser Wert ist im Feld `Kind` des erzeugten Knoten-Records ebenfalls enthalten. Diese redundante Speicherung ist notwendig, um in den Attributauswertungsregeln zur Behandlung von Konstantenauswertungen in Ausdrücken zur Übersetzungszeit über dieses Attribut auf den Operator zugreifen zu können (siehe 3.2.8 Ausdrücke, S.121). Ein direkter Zugriff innerhalb einer Attributauswertungsregel auf das Feld `Kind` des zugehörigen Knotens wäre nur mit Kenntnis der internen, generierten Struktur des Evaluators möglich.

Selektoren

Die kontextfreie Grammatik von Oberon-2 unterscheidet nicht zwischen Prozeduraufrufen und Typzusicherungen sowie zwischen Namensqualifizierung und Record-Selektion. Diese Unterscheidung läßt sich ohne Zuhilfenahme von Kontextinformation bei der Parsierung nicht treffen. Deshalb wird bei der Parsierung an diesen Stellen nur ein Teilbaum aufgebaut, welcher die syntaktische Struktur der Selektoren durch entsprechend benannte Knotentypen repräsentiert. Diese Knotentypen werden in Anlehnung an den Begriff ,,Operator“ als ,,Designoren“ bezeichnet und sind Erweiterungen des Knotentyps `Designor`. Die Selektoren `"."`,

"[...]", "^" und "(...)" werden durch die Knotentypen `Selector`, `Indexor`, `Dereferencor` und `Argumentor` repräsentiert.

Der Teilbaum wird bei der Attributauswertung unter Heranziehung von Kontextinformation durch wiederholten Aufruf einer Puma-Funktion (`TT.DesignorToDesignation`) schrittweise in einen anderen Teilbaum transformiert, der dann die eigentliche Struktur der Selektoren repräsentiert. Die Knotentypen, die im transformierten Teilbaum benutzt werden, sind Erweiterungen des Knotentyps `Designation` (in Anlehnung an ,,Operation"). Das Kind des Knotentyps `Designator` bzw. `Designation` mit dem Knotentyp `Designations` ist im Modul `ExtendedSyntaxTree` definiert, da es seinen Wert nicht zum Zeitpunkt der Knotenerzeugung sondern erst durch die Transformationsfunktion erhalten kann.

Vordeklarierte und SYSTEM-Prozeduren

Die Hierarchie der Knotentyperweiterungen vom Knotentyp `PredeclArgumenting` dient der einfacheren Handhabung der aktuellen Parameter von vordeklarierten Prozeduren und Prozeduren, die vom Modul `SYSTEM` zur Verfügung gestellt werden, bei der Attributauswertung. Die hierfür geltenden Kontextbedingungen sind wesentlich komplexer als diejenigen, die für die aktuellen Parameter von Prozeduren gelten, die vom Sprachbenutzer definiert werden können. Die Knoten, deren Knotentypen Erweiterungen vom Knotentyp `PredeclArgumenting` sind, werden ebenfalls durch die bereits erwähnte Puma-Funktion zur Transformation des abstrakten Syntaxbaums angelegt. Deshalb ist der Knotentyp `PredeclArgumenting` eine Erweiterung des Knotentyps `Designation`.

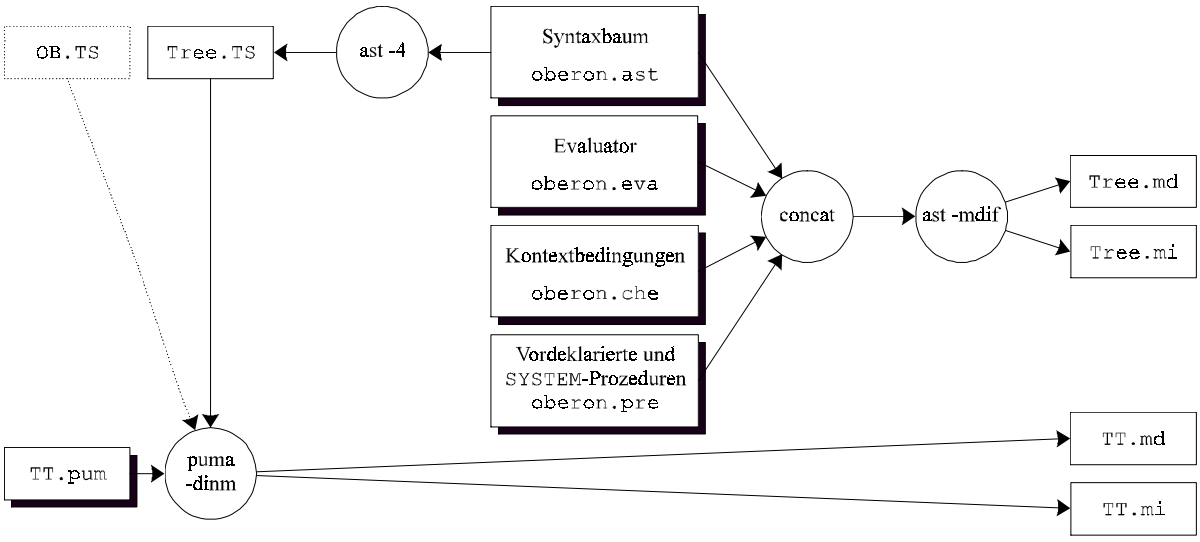

Abb. 7: Einsatz des Werkzeugs Ast zur Erzeugung eines Syntaxbaummoduls.

Werkzeugaufruf

Die Spezifikationsdatei `oberon.ast` dient sowohl zur Erzeugung der Datenstruktur des abstrakten Syntaxbaums (zusammen mit den Spezifikationsdateien `oberon.eva`, `oberon.che` und `oberon.pre`), als auch zum Aufbau einer Baumbeschreibungsdatei (`Tree.TS`), welche vom Werkzeug Puma benutzt wird, um Kenntnis über die Felder der Knotentypen zu erhalten (siehe Abbildung 7). Die vier vorstehend genannten Spezifikationsdateien werden beim Aufruf des Werkzeugs Ast konkateniert. Ast erzeugt daraus das Modu-

la-2 Modul Tree.(md|mi) welches die Datenstrukturen für den abstrakten Syntaxbaum enthält. Von den bei GROSCH [1991c] beschriebenen Optionen für Ast werden hierbei die folgenden eingesetzten:

d	Generierung des Definitionsmoduls Tree.md
i	Generierung des Implementationsmoduls Tree.mi
m	Generierung von Knotenkonstruktorprozeduren mit formalen Parametern für alle Attribute des Knotentyps mit der Eigenschaft INPUT
f	Generierung der Baumdestruktorprozedur ReleaseTREE

Zur Generierung von Funktionen zur Klassifikation von Teilbäumen benötigt das Werkzeug Puma Informationen über die Anzahl und den Typ der Felder derjenigen Knotentypen, deren Knoten den Teilbaum bilden. Diese Information wird in einer Datei bereitgestellt, welche das Werkzeug Ast bei Aufruf mit der Option 4 erzeugt.

Für die Puma-Funktionen zur Baumtransformation ist die Kenntnis über die in der Spezifikationsdatei oberon.ast definierten Felder ausreichend. Eine Hinzunahme der restlichen, für den abstrakten Syntaxbaum relevanten Spezifikationsdateien würde die Anzahl der für Puma ,,sichtbaren“ Felder der Knotentypen und damit die Anzahl der anzugebenden Untermuster in den Mustern von Puma-Funktionen unnötig vergrößern.

3.1.4 Parser (oberon.lal)

Das für die Parsierung verwendete, abgeleitete Attribut vom Typ tParsAttribute umfaßt neben dem obligatorischen Feld Scan vom Typ des Scanner-Attributs nur noch das Feld Tree, welches als Typ den Baumtyp des abstrakten Syntaxbaums besitzt. Bei der notwendigen Wiederholung der Symbolkodierung im Abschnitt TOKEN werden die Symbolnamen weitgehend wieder in ihre textuelle Repräsentation überführt (z.B. '&' statt AmpersandToken). Dies erhöht die Lesbarkeit der kontextfreien Grammatik in der Spezifikationssprache.

Transformation der Grammatik

Die durch die Sprachdefinition gegebene Grammatik ist für die Parsierung mit einer Top-down-Strategie besonders geeignet. Trotzdem wurde das Werkzeug Lalr und damit eine Bottom-up-Strategie gewählt, weil so eine stärkere (aus didaktischen Gründen gewollte) Trennung der Attributauswertung von der Parsierung natürlicher erscheint.

Wegen der Wahl einer Bottom-up-Strategie müssen diverse Transformationen der Grammatik durchgeführt werden. Sie beziehen sich vor allem auf die Eliminierung der EBNF-Konstrukte, welche bis auf die Alternative in einer Lalr-Spezifikation nicht sinnvoll einsetzbar sind (vgl. S.53). Bei der Eliminierung von EBNF-Iterationskonstrukten wurden die entsprechenden BNF-Regeln rechtsrekursiv formuliert. Rechtsrekursive Regeln bedeuten zwar für einen Bottom-up-Parser einen erhöhten Speicheraufwand bei der Parsierung, erleichtern jedoch die Konstruktion von Listen, in denen die Reihenfolge ihrer Elemente dem Auftreten im Quelltext entspricht. Daneben dienen die Transformationen der notwendigen Eliminierung der Mehrdeutigkeit. Weitere vorgenommene Transformationen, die nicht zwingend erforderlich sind, werden im folgenden im entsprechenden Abschnitt näher erläutert.

Die einzelnen Transformationen werden nachfolgend durch Gegenüberstellung der Originalgrammatikregeln und der transformierten Regeln tabellarisch aufgeführt. Hierbei sind zunächst die Transformationen aufgeführt, welche lediglich der Eliminierung von EBNF-Konstrukten gelten.

Originalgrammatik		*Transformierte Grammatik*	
Module	= … [ImportList] … .	Module	= … ImportList … .
ImportList	= IMPORT [ident ":="] ident {"," [ident ":="] ident} ";".	ImportList	= IMPORT Import Imports ";" | .
		Imports	= "," Import Imports | .
		Import	= ident | ident ":=" ident .
Module	= … [BEGIN StatementSeq] … .	Module	= … BeginStmts … .
ProcDecl	= … [BEGIN StatementSeq] … .	ProcDecl	= … BeginStmts … .
		BeginStmts	= BEGIN StatementSeq | .
Module	= … DeclSeq … .	Module	= … DeclSection … .
DeclSeq	= { CONST {ConstDecl ";"} | TYPE {TypeDecl ";" } | VAR {VarDecl ";" }} { ProcDecl ";"| ForwardDecl ";"}.	DeclSection	= DeclUnits ProcDecls .
		DeclUnits	= DeclUnit DeclUnits | .
		DeclUnit	= CONST ConstDecls | TYPE TypeDecls | VAR VarDecls .
		ConstDecls	= ConstDecl ";" ConstDecls | .
		TypeDecls	= TypeDecl ";" TypeDecls | .
		VarDecls	= VarDecl ";" VarDecls | .
		ProcDecls	= ProcDecl ";" ProcDecls | .
ProcDecl	= … .		
ForwardDecl	= … .		
ProcDecl	= PROCEDURE [Receiver] IdentDef [FormalPars] … .	ProcDecl	= PROCEDURE IdentDef FormalPars … . | PROCEDURE "^" IdentDef FormalPars | PROCEDURE Receiver IdentDef FormalPars … . | PROCEDURE "^" Receiver IdentDef FormalPars .
ForwardDecl	= PROCEDURE "^" [Receiver] IdentDef [FormalPars] .		

```
ProcDecl      = … [FormalPars] … .
ForwardDecl   = … [FormalPars] … .
Type          = … PROCEDURE [FormalPars] .
FormalPars    = "(" [FPSection {";" FPSection}] ")"
                [ ":" Qualident] .

FPSection     = [VAR] ident {"," ident} ":" Type .

Receiver      = "(" [VAR] ident ":" ident ")" .

StatementSeq  = Statement {";" Statement} .
Statement     = [ Designator ":=" Expr
                | Designator …
                | IF … | CASE … | WHILE …
                | REPEAT … | FOR … | LOOP …
                | WITH … | EXIT | RETURN …
                ].

Statement     = … | IF Expr THEN StatementSeq
                    {ELSIF Expr THEN StatementSeq}
                    [ELSE StatementSeq] END | … .

Statement     = … | CASE Expr OF Case {"|" Case}
                    [ELSE StatementSeq] END | … .
Case          = [CaseLabels {"," CaseLabels}
                ":" StatementSeq] .

CaseLabels    = ConstExpr [".." ConstExpr] .

Statement     = … | FOR ident ":=" Expr TO Expr
                    [BY ConstExpr] DO StatementSeq
                    END | … .

Statement     = … | WITH Guard DO StatementSeq
                    {"|" Guard DO StatementSeq}
                    [ELSE StatementSeq] END | … .

Statement     = … | RETURN [Expr] … .

SimpleExpr    = ["+"|"-"] Term {AddOp Term}.

Term          = Factor {MulOp Factor}.

Set           = "{" [Element {"," Element}] "}" .
Element       = Expr [".." Expr] .

ExprList      = Expr {"," Expr}.

IdentList     = IdentDef {"," IdentDef}.
```

```
ProcDecl      = … FormalPars … .
Type          = … ProcedureType .
ProcedureType = PROCEDURE FormalPars .
FormalPars    = "(" ")" FormalResult
              | "(" FPSections ")" FormalResult | .
FormalResult  = ":" Qualident | .
FPSections    = FPSection ";" FPSections | FPSection .

FPSection     = ParIds ":" Type | VAR ParIds ":" Type .
ParIds        = ident "," ParIds | ident .

Receiver      = "(" ident ":" ident ")"
              | "(" VAR ident ":" ident ")" .

StatementSeq  = Statement ";" StatementSeq
              | ";" StatementSeq | Statement | .
Statement     = AssignStmt | CallStmt | IfStmt | CaseStmt
              | WhileStmt | RepeatStmt | ForStmt
              | LoopStmt | WithStmt
              | ExitStmt | ReturnStmt .
AssignStmt    = Designator ":=" Expr .
CallStmt      = Designator .
IfStmt        = IF … .
CaseStmt      = CASE … .
WhileStmt     = WHILE … .
RepeatStmt    = REPEAT … .
ForStmt       = FOR … .
LoopStmt      = LOOP … .
WithStmt      = WITH … .
ExitStmt      = EXIT .
ReturnStmt    = RETURN … .

Statement     = … | IfStmt | … .
IfStmt        = IF Expr THEN StatementSeq ElsIfs .
ElsIfs        = ELSIF Expr THEN StatementSeq ElsIfs
              | ELSE StatementSeq  END | END .

Statement     = … | CaseStmt | … .
CaseStmt      = CASE Expr OF Cases
                ELSE StatementSeq END
              | CASE Expr OF Cases END .
Cases         = Case "|" Cases | "|" Cases | Case | .
Case          = CaseLabelList ":" StatementSeq .
CaseLabelList = CaseLabels "," CaseLabelList
              | CaseLabels .

CaseLabels    = ConstExpr ".." ConstExpr | ConstExpr .

Statement     = … | ForStmt | … .
ForStmt       = FOR ident ":=" Expr TO Expr
                BY ConstExpr DO StatementSeq END
              | FOR  ident ":=" Expr TO Expr
                DO StatementSeq END .

Statement     = … | WithStmt | … .
WithStmt      = WITH Guard DO StatementSeq
                Guards ELSE StatementSeq END
              | WITH Guard DO StatementSeq
                Guards END .
Guards        = "|" Guard DO StatementSeq Guards | .

Statement     = … | ReturnStmt .
ReturnStmt    = RETURN Expr | RETURN .

SimpleExpr    = SimpleExpr AddOp Term
              | "+" Term | "-" Term | Term .

Term          = Term MulOp Factor | Factor .

Set           = "{" Elements "}" | "{" "}" .
Elements      = Element "," Elements | Element .
Element       = Expr ".." Expr | Expr .

ExprList      = Expr "," ExprList | Expr .

IdentList     = IdentDef "," IdentList | IdentDef .
```

```
IdentDef = ident ["*"|"-"].
```

```
IdentDef = ident | ident "*" | ident "-" .
```

Desweiteren wurde die Regel für das Nichtterminal Factor dahingehend geändert, daß statt number die vom Scanner erkannten Terminale integer, real und longreal verwendet werden. Die Transformation ist wie folgt:

Originalgrammatik

```
Factor = … | number | character | … .
```

Transformierte Grammatik

```
Factor = … | integer | real | longreal | character … .
```

Eine weitere Transformation betrifft die Regel für Zeigertypen. Durch die Einführung eines Nichtterminals PointerBaseType für die beiden legalen Zeigerbasistypen (ArrayType und RecordType) und eine entsprechende Änderung der Regel für PointerType ist es möglich, die Kontextbedingung, welche die legalen Zeigerbasistypen einschränkt, bereits kontextfrei zu formulieren. Zudem sind hierbei anstatt einer Alternative ,,POINTER TO Qualident“ zwei Alternativen für die beiden möglichen Formen benannter Typen angegeben. Dies gestattet eine einfache Konstruktion unterschiedlicher Knoten zur gezielten Behandlung von Vorwärtszeigern. Die restlichen Transformationen betreffen wiederum die Umsetzung in BNF und sind der Vollständigkeit halber an dieser Stelle mit aufgeführt.

Originalgrammatik

```
Type = Qualident | ARRAY … | RECORD …
     | POINTER … | PROCEDURE … .
```

Transformierte Grammatik

```
Type          = Qualident | PointerBaseType
              | PointerType | ProcedureType .
PointerBaseType = ArrayType | RecordType .
ArrayType     = ARRAY … .
RecordType    = RECORD … .
PointerType   = POINTER … .
ProcedureType = PROCEDURE … .
```

Originalgrammatik

```
Type = … | ARRAY [ConstExpr {"," ConstExpr}]
       OF Type | … .
```

Transformierte Grammatik

```
Type          = … | PointerBaseType | … .
PointerBaseType = ArrayType | … .
ArrayType     = ARRAY ArrayExprList OF Type
              | ARRAY OF Type .
ArrayExprList = ConstExpr "," ArrayExprList | ConstExpr .
```

Originalgrammatik

```
Type      = … | RECORD ["(" Qualident ")"] FieldList
            {";" FieldList} END | … .
FieldList = [ IdentList ":" Type ].
```

Transformierte Grammatik

```
Type          = … | PointerBaseType | … .
PointerBaseType = … | RecordType .
RecordType    = RECORD "(" Qualident ")" FieldLists END
              | RECORD FieldLists END .
FieldLists    = FieldList ";" FieldLists
              | ";" FieldLists | FieldList | .
FieldList     = IdentList ":" Type .
```

Originalgrammatik

```
Type = … | POINTER TO Type | … .
```

Transformierte Grammatik

```
Type          = … | PointerType | … .
PointerType   = POINTER TO ident
              | POINTER TO ident "." ident
              | POINTER TO PointerBaseType .
PointerBaseType = ArrayType | RecordType .
```

Eine weitere einschränkende Transformation betrifft den Operator IS, welcher in der transformierten Grammatik nicht mehr bei den Vergleichsoperatoren (Relation) angegeben ist, sondern bei den Ausdrücken (vgl. Typtest S.76).

Originalgrammatik

```
Expr     = SimpleExpr [Relation SimpleExpr].
Relation = "=" | ... | IN | IS .
```

Transformierte Grammatik

```
Expr     = SimpleExpr Relation SimpleExpr
         | Designator IS Qualident | SimpleExpr .
Relation = "=" | ... | IN .
```

Die letzte Transformation betrifft die Auflösung der Mehrdeutigkeit der kontextfreien Originalgrammatik. Sowohl für die Symbolfolge ident(ident) als auch für ident.ident lassen sich zwei verschiedene Ableitungsbäume konstruieren. Abbildung 8 zeigt beispielsweise zwei verschiedene Ableitungsbaumfragmente für einen Prozeduraufruf ident(ident). In Abbildung 9 ist

das Ableitungsbaumfragment für diese Symbolfolge gemäß der transformierten eindeutigen Grammatik dargestellt. Diese Transformation bedeutet eine Erweiterung der durch die kontextfreie Grammatik definierten Sprache bezüglich der erlaubten Bezeichner (Designations), die durch zusätzliche Kontextbedingungen in den Attributauswertungsregeln wieder eingeschränkt wird.

Originalgrammatik

```
Statement   = … | Designator ["(" [ExprList] ")"] | … .
Designator  = Qualident … .
Qualident   = [ident "."] ident .
```

```
Designator  = Qualident {"." ident | "[" ExprList "]" | "^"
                        | "(" Qualident ")" }.
Qualident   = [ident "."] ident .
```

Transformierte Grammatik

```
Statement   = … | CallStmt | … .
CallStmt    = Designator .
Designator  = ident Designations .
Designations = "." ident Designations | …
             | "(" ExprList ")" Designations
             | "(" ")" Designations | .
```

```
Designator  = ident Designations .
Designations = "." ident Designations
             | "[" ExprList "]" Designations
             | "^" Designations
             | "(" ExprList ")" Designations
             | "(" ")" Designations | .
Qualident   = ident | ident "." ident .
```

Rückgabe des erzeugten abstrakten Syntaxbaums

Der Wert des Wurzelattributs, welcher die Wurzel des abstrakten Syntaxbaums darstellt, wird nicht in der Variable `TreeRoot` aus dem von Ast erzeugten Modul gespeichert, sondern in einer eigenen Datenstruktur aus dem Modula-2 Modul `FIL` abgelegt, neben anderen Werten, die ebenfalls zu dem zu übersetzenden Modul gehören.

Listenkonstruktion

Der Aufbau von Listen des abstrakten Syntaxbaums erfolgt grundsätzlich nach folgendem Schema (vgl. Listenbildung S.75):

```
Symbols:
  Symbol
  Symbols { $1.Tree^.Node.Next := $2.Tree;
            $$.Tree            := $1.Tree; }
|         { $$.Tree            := Tree.mmtNode(); }
.
Symbol:
  SYMBOL  { $$.Tree            := Tree.mNode(Tree.NoTree,…); }
.
```

Die rechtsrekursive Formulierung der Symbols-Regel gestattet eine einfache Verkettung der einzelnen Listenelemente unter Beibehaltung ihrer Reihenfolge im Quelltext. Bei der Verkettung wird der Verweis auf die Restliste direkt dem `Next`-Feld eines Listenelements zugewiesen. Zur Realisierung dieser Verkettungsmethode ist die Kenntnis der konkreten Modellierung der Ast-Datentypen durch Modula-2 Typen allerdings erforderlich.

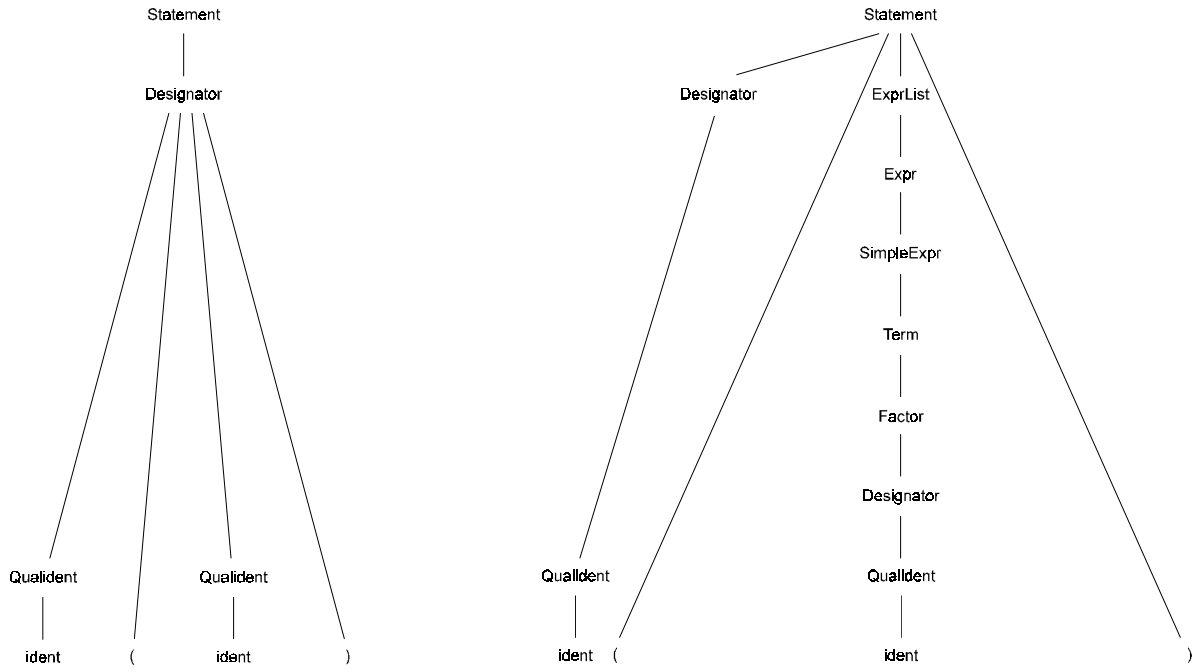

Abb. 8: Zwei verschiedene Ableitungsbäume für ident (ident) *nach der Originalgrammatik.*

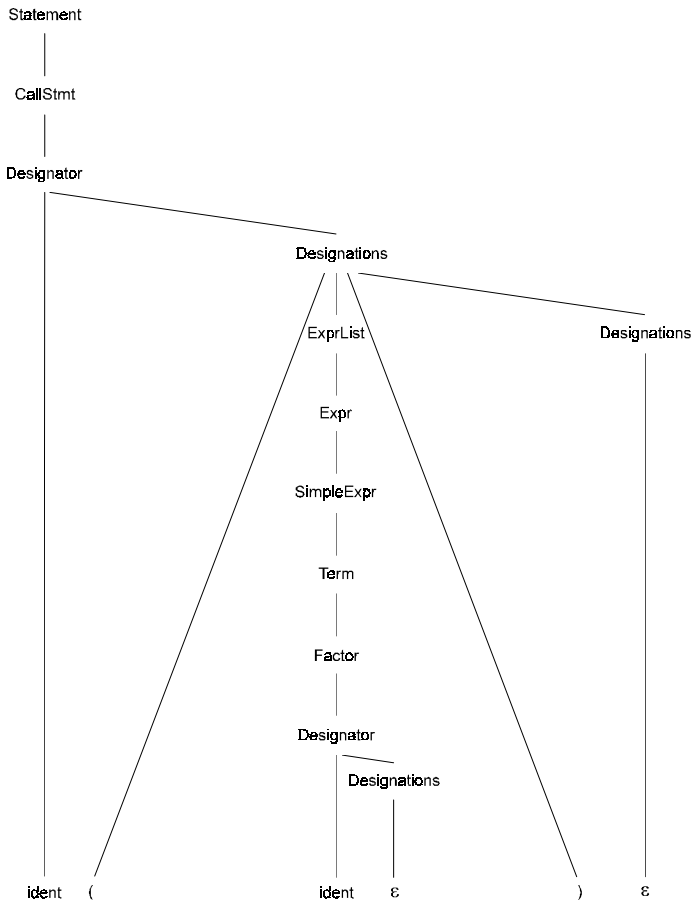

Abb. 9: Ableitungsbaum für ident (ident) *nach der transformierten Grammatik.*

Fehlende Else-Zweige

Die Regel CaseStmt bzw. WithStmt definiert einen optionalen Else-Zweig. Im abstrakten Syntaxbaum wird jedoch immer ein Knoten mit vorhandener Else-Anweisungsfolge erzeugt. Diese besteht bei fehlendem Else-Zweig lediglich aus einem Prozeduraufruf der (vordeklarierten) Prozedur _CASEFAULT bzw. _WITHFAULT.

Case-Marken und Mengenelemente

Bei der Angabe von Case-Marken bzw. Mengenelementen besteht die Möglichkeit, entweder nur einen einzelnen Ausdruck oder zwei (einen Bereich definierende) Ausdrücke anzugeben. Hier wird in beiden Fällen immer ein Knoten mit zwei Ausdrücken erzeugt. Für einen fehlenden zweiten Ausdruck wird bei einer Case-Marke ein Knoten vom Knotentyp `ConstExpr` mit leerem Ausdruck (`mtExpr`) und bei einem Mengenelement ein Knoten vom Knotentyp `mtExpr` angelegt.

Schrittweite in For-Anweisung

Eine fehlende Schrittweitenangabe („BY-Ausdruck“) wird durch einen Unterbaum ergänzt, der einen Konstantenausdruck mit dem ganzzahligen Wert 1 repräsentiert.

Position von geklammerten Ausdrücken

Bei der Erzeugung des abstrakten Syntaxbaums für Ausdrücke wird von eventuellen Klammerungen abstrahiert. Für die Positionsangabe von Fehlermeldungen, die sich auf einen geklammerten Ausdruck beziehen, wird die Position der äußersten öffnenden Klammer als Position des Ausdrucks benutzt.

Werkzeugaufruf

Von den bei GROSCH und VIELSACK [1991] beschriebenen Optionen für das Werkzeug Lalr wurden die folgenden benutzt:

`b`	Aufruf des Präprozessors Bnf
`d`	Generierung des Definitionsmoduls `Parser.md` in Modula-2

Die Benutzung der Option `b` führt dazu, daß die Eingabedatei `oberon.lal` vom Präprozessor Bnf eingelesen und dessen Ausgabe direkt vom Werkzeug Lalr weiterverarbeitet wird. Bei der erstmaligen Generierung wurde zusätzlich über die Option `e` der Prototyp eines Moduls zur Fehlerbehandlung bei der Parsierung erzeugt (`Errors.(md|mi)`).

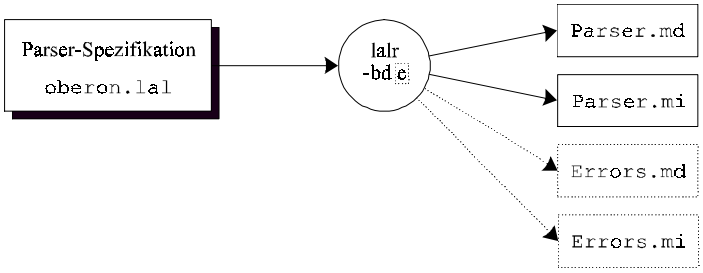

Abb. 10: Einsatz des Werkzeugs Lalr zur Parser-Erzeugung.

3.1.5 Evaluatorobjekte (OB.ast)

Zur semantischen Analyse werden diverse Datentypen benötigt. Werte dieser Datentypen werden daher Evaluatorobjekte genannt. Diese Datentypen, wie z.B. Symboltabelle oder Typ- und Wertrepräsentationen, sind unter Zuhilfenahme des Werkzeugs Ast modelliert. Dies gestattet die Formulierung von Puma-Funktionen zur Operation auf den Evaluatorobjekten. Die Definitionen der Evaluatorobjekte sind in dem Spezifikationsdokument `OB.ast` (evaluator OBjects) zusammengefaßt.

Das Modul `OB` stellt neben den Datentypen noch einige ausgezeichnete Werte zur Verfügung, die zur Belegung bestimmter Attribute bei der semantischen Analyse eingesetzt werden. Wegen einer Restriktion[1] der verwendeten Entwicklungsumgebung müssen diese Werte in Variablen gehalten werden. Die Namen dieser ,,Konstanten“ bestehen ausschließlich aus Großbuchstaben.

An einigen Stellen bei der semantischen Analyse werden sog. *konstante* Objekte bzw. Knoten als Ersatz für sonst immer wieder neu zu erzeugende Objekte bzw. Knoten eingesetzt. Für diese sind Variablen deklariert, die mit einem Verweis auf einen entsprechenden Knoten initialisiert werden. Um den konstanten Charakter dieser Variablen zu verdeutlichen beginnen ihre Namen alle mit einem `C`.

3.1.5.1 Symboltabelleneinträge

Alle Objekte (Konstanten, Typen, Variablen, Prozeduren und importierte Module), die in einem Modul deklariert sind, werden als Einträge (Knotentyp `Entry`) in der Symboltabelle repräsentiert.

Die Symboltabelle hat eine baumartige Struktur, da die zu den verschiedenen Sichtbarkeitsbereichen gehörenden Listen jeweils eine gemeinsame Restliste haben, in der die gemeinsamen Anteile dieser Sichtbarkeitsbereiche gehalten werden. Die Verkettung erfolgt in umgekehrter Weise von den Blättern zur Wurzel über das Kind `prevEntry` und die Verankerung des ,,Baums“ über Verweise auf seine Blätter.

In Abbildung 11 ist beispielhaft der Symboltabellenbaum zu den nachstehenden Modulen skizziert.

```
MODULE M; IMPORT X;
VAR V:… ;
PROCEDURE P(p:… );
  VAR V:… ;
  BEGIN END P;
PROCEDURE Q(p:… );
  CONST C=… ;
  BEGIN END Q;
BEGIN END M.
```

1 Das Werkzeug Puma erzeugt zur Realisierung des *Treffens* eines (Wert-)Ausdrucks den Aufruf einer Testprozedur `yyIsEqual`, falls der Typ des Ausdrucks vom Spezifikateur definiert ist. Die formalen Parameter dieser Testprozedur sind vom Typ `ARRAY OF SYSTEM.BYTE`, für den jedoch der verwendete Modula-2 Übersetzer Mocka als aktuelle Parameter keine skalaren Konstanten gestattet.

```
MODULE X;
TYPE T=...;
END X.
```

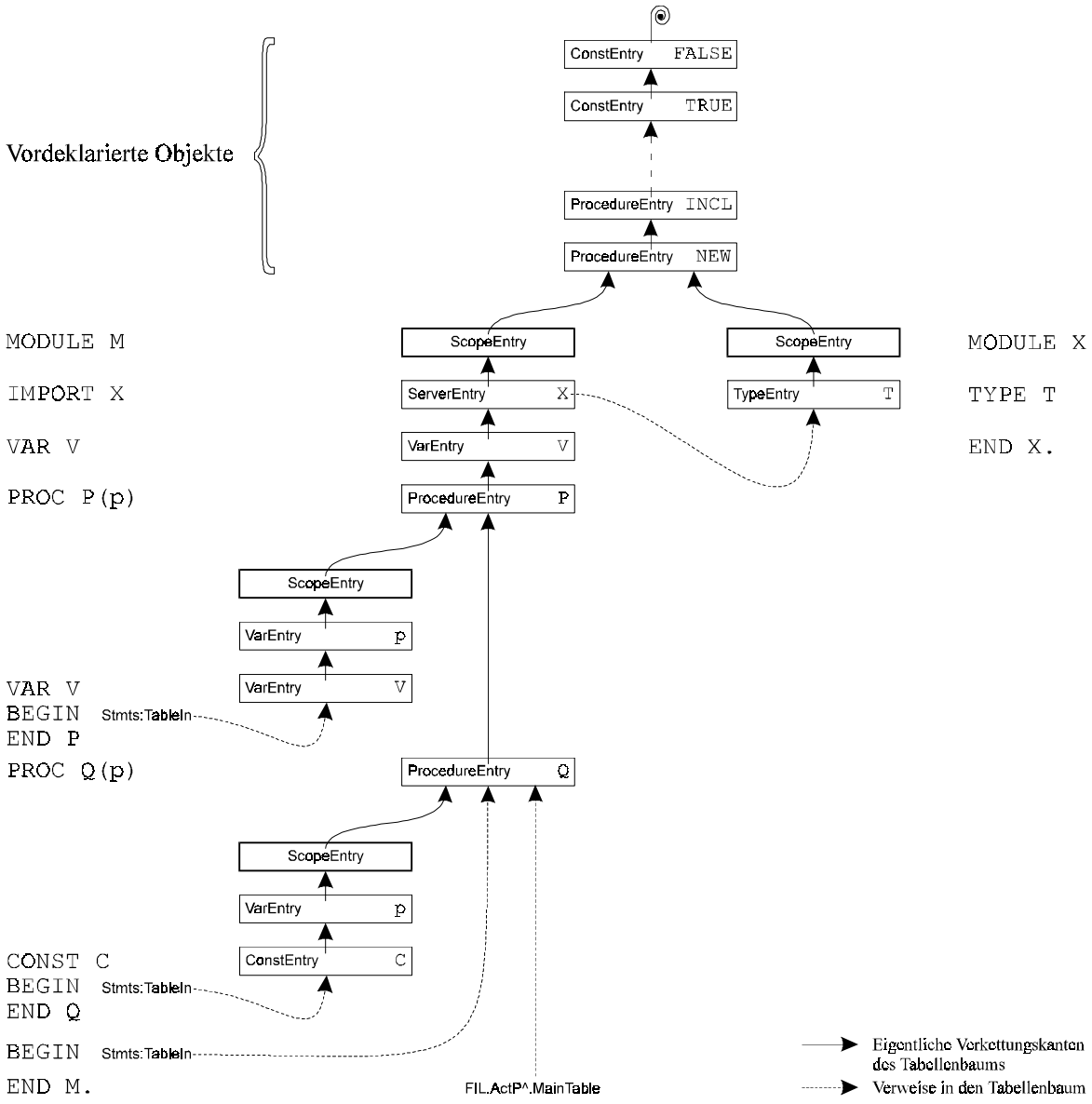

Abb. 11: Prinzipieller Aufbau der Symboltabelle an einem Beispiel.

Der Knotentyp `Entry` ist eine Erweiterung des allgemeinen Knotentyps `Entries`. Zusätzliche Erweiterungen von `Entries` sind die Knotentypen `mtEntry` und `ErrorEntry`, die aber nicht in die Symboltabelle aufgenommen werden, sondern lediglich zur Repräsentation spezieller Zustände von entsprechenden Puma-Funktionen geliefert werden.

Grundsätzlich werden zwei Arten von Symboltabelleneinträgen als Erweiterungen des Knotentyps `Entry` unterschieden. Der Knotentyp `ScopeEntry` dient hierbei ausschließlich zur Anzeige einer Sichtbarkeitsgrenze, wodurch die Suche in einem lokalen Sichtbarkeitsbereich vereinfacht wird. Die zweite Erweiterung von `Entry` ist der Knotentyp `DataEntry`, in welchem die gemeinsamen Attribute der verschiedenen Symboltabelleneinträge definiert sind. Im Attribut `declStatus` wird hierbei ein sog. Deklarationszustand eines Eintrags gehalten, um

den Sichtbarkeitsregeln bei der semantischen Analyse Rechnung tragen zu können (siehe S.106).

Der Knotentyp `ServerEntry` ist eine Erweiterung des Knotentyps `DataEntry` und enthält ein Kind `serverTable`, das auf den letzten Eintrag der Symboltabelle eines importierten Moduls verweist. Diese Symboltabelle enthält alle in dem Modul deklarierten Objekte, also auch die nichtexportierten Objekte.

Die formalen Parameter einer (gebundenen) Prozedur und ihr (evtl. vorhandener) Resultatstyp werden bei den Knotentypen `ProcedureEntry` und `BoundProcEntry` im Kind `typeRepr` über einen Prozedurtyp repräsentiert.

Die beiden Knotentypen `BoundProcEntry` und `InheritedProcEntry`, die ebenfalls eine Erweiterung des Knotentyps `DataEntry` sind, treten nie als Einträge in einer Symboltabelle auf. Sie werden lediglich als Einträge für gebundene Prozeduren in der Feldliste einer Record-Typrepräsentation verwendet. Hierbei wird `BoundProcEntry` für eine durch Deklaration an einen Record-Typ gebundene Prozedur bzw. `InheritedProcEntry` für eine durch Typerweiterung an einen Record-Typ gebundene Prozedur eingesetzt.

3.1.5.2 Typrepräsentationen

Ähnlich wie bei den Symboltabelleneinträgen existieren vom Knotentyp `TypeReprs`, neben der Erweiterung `TypeRepr` zur Repräsentation von Typen, noch die Erweiterungen `mtTypeRepr` und `ErrorTypeRepr` zur Kodierung spezieller Zustände bei der semantischen Analyse.

Der Knotentyp `TypeRepr` umfaßt die gemeinsamen Attribute aller zur Typrepräsentation eingesetzten Knotentypen, die Erweiterungen des Knotentyps `TypeRepr` sind. Hierbei verweist das Kind `entry` bei allen benannten Typen immer auf den Symboltabelleneintrag der entsprechenden Typdeklaration bzw. auf den *konstanten* Knoten `cNonameEntry` bei unbenannten Typen.

Die Repräsentation von Basistypen erfolgt über konstante Knoten des entsprechenden Knotentyps. Dies erleichtert die Realisierung der Typkompatibilitätsregeln (zwei Typen sind *dieselben*, falls die Verweise auf die entsprechenden Typrepräsentationsknoten gleich sind). Der Knotentyp `NilTypeRepr` als Repräsentation des Typs von `NIL` wurde aus dem gleichen Grund eingeführt. Zur Typrepräsentation von Zeichenkonstanten und Zeichenketten der Länge 1 wird der Knotentyp `CharStringTypeRepr` verwendet.

Die Repräsentation von Array-Typen erfolgt über den Knotentyp `ArrayTypeRepr`, der eine Längenangabe `len` und eine Angabe des Elementtyps `elemTypeRepr` umfaßt. Der Typ von offenen Arrays wird ebenfalls über diesen Knotentyp repräsentiert. Er hat als Länge jedoch den ausgezeichneten Wert `OPENARRAYLEN`.

Die Felder eines Record-Typs werden mittels einer Liste von Einträgen (`Entries`) im Knotentyp `RecordTypeRepr` repräsentiert. Sein Kind `fields` verweist auf den letzten Eintrag dieser Liste, die durch einen `ScopeEntry` eingeleitet wird. Abbildung 12 verdeutlicht dies anhand der Deklaration eines benannten Record-Typs. Das Attribut `extLevel`, die restlichen Kinder `baseTypeRepr` und `extTypeReprList` sowie der Objektknotentyp `TypeReprLists` dienen der Verwaltung des Typerweiterungsmechanismus, die im Abschnitt 3.2.6 Typerweiterungen (S.115) näher ausgeführt wird.

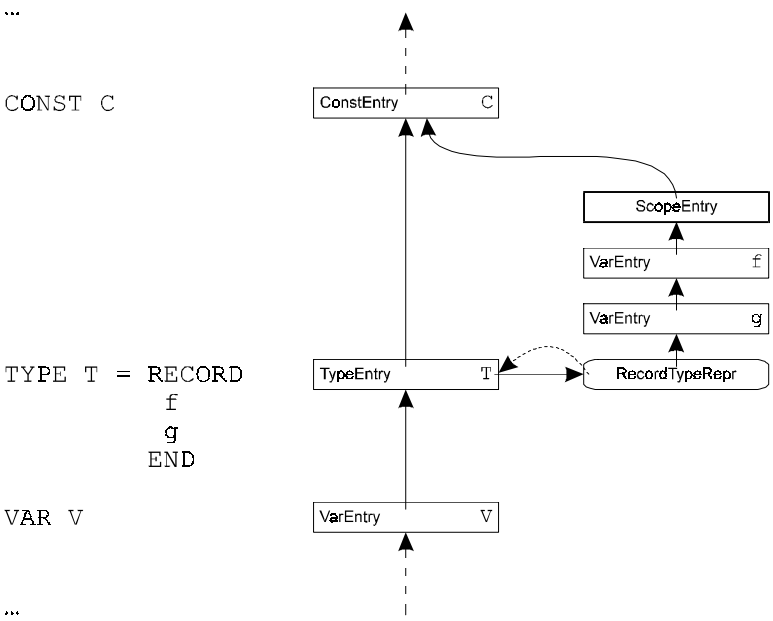

Abb. 12: Repräsentation eines Record-Typs.

Das Kind, das im Knotentyp `PointerTypeRepr` zur Repräsentation des Zeigerbasistyps dient (`baseTypeEntry`), ist vom Knotentyp `Entries` und nicht vom Knotentyp `TypeReprs`. Dies ermöglicht eine einfache Handhabung von Vorwärtszeigern, indem das Kind `baseTypeEntry` auf einen `TypeEntry` verweist, dessen Kind `typeRepr` dann den entsprechenden Zeigerbasistyp repräsentiert (siehe 3.2.4 Vorwärtszeiger, S.111).

Der Knotentyp `ProcedureTypeRepr` besitzt ein Kind `signatureRepr` mit Knotentyp `SignatureRepr`, welches zur Repräsentation der *Signatur* einer Prozedur dient. Mit Signatur wird ausschließlich die Liste der formalen Parameter einer Prozedur bezeichnet. Eine gebundene Prozedur besitzt für den Empfängerparameter eine eigene Signatur.

Wegen der gesonderten Behandlung der vordeklarierten Prozeduren und Prozeduren aus dem Modul `SYSTEM` besitzen diese jeweils eine eigene (Prozedur-)Typrepräsentation. Für diese wird der Knotentyp `GenericSignature` verwendet, der eine Erweiterung des Knotentyps `SignatureRepr` ist.

3.1.5.3 Wertrepräsentationen

Die Werte, die zur Übersetzungszeit aus Konstanten berechnet werden können, werden durch Knoten des Knotentyps `ValueReprs` bzw. seiner Erweiterungen repräsentiert. Das Schema der Erweiterungshierarchie entspricht hier ebenfalls dem der Knotentypen `Entries` und `TypeReprs`.

Für die drei möglichen ganzzahligen Typen `SHORTINT`, `INTEGER` und `LONGINT` ist nur ein Knotentyp `IntegerValue` definiert, der ein Attribut `v` vom Typ `oLONGINT` besitzt.

Der Knotentyp `ProcedureValue` dient der Repräsentation von Prozedurwerten, also dem Wert eines Prozedurbezeichners ohne Parameterliste innerhalb eines Ausdrucks.

Die Behandlung der Kontextbedingungen bezüglich der Case-Marken bei Case-Anweisungen erfordert die Repräsentation der durch sie definierten Werte(-bereiche). Hierzu wird der Knotentyp `LabelRanges` mit seinen Erweiterungen eingesetzt.

Werkzeugaufruf

Zur Erzeugung der in OB.ast spezifizierten Datentypen wurde das Werkzeug Ast mit den nachfolgend aufgeführten Optionen benutzt:

d	Generierung des Definitionsmoduls OB.md
i	Generierung des Implementationsmoduls OB.mi
m	Generierung von Knotenkonstruktorprozeduren mit formalen Parametern für alle Attribute des Knotentyps mit der Eigenschaft INPUT
=	Generierung der Vergleichsprozedur IsEqualOB

Die Datei OB.TS, welche die Informationen über die Knotentypen für das Werkzeug Puma bereitstellt, wurde wiederum durch Aufruf des Werkzeugs Ast mit der Option 4 erzeugt.

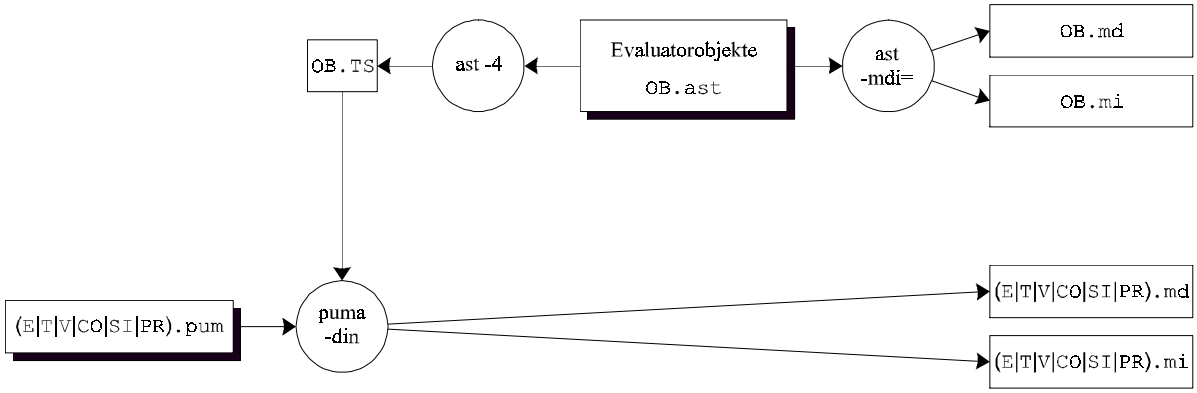

Abb. 13: Einsatz des Werkzeugs Ast zur Erzeugung eines Moduls für Evaluatorobjekte.

3.1.6 Evaluator (oberon.eva, oberon.che, oberon.pre)

Die in den Dateien oberon.eva, oberon.che und oberon.pre angegebenen Module Evaluator, Checks und Predeclareds erweitern die Spezifikation des abstrakten Syntaxbaums (oberon.ast) um Attribute und Attributauswertungsregeln, so daß in der Gesamtheit dieser Dokumente die Evaluatorspezifikation vorliegt. Abbildung 14 skizziert die Aufteilung dieser Definitionen auf die verschiedenen Module bzw. Dateien.

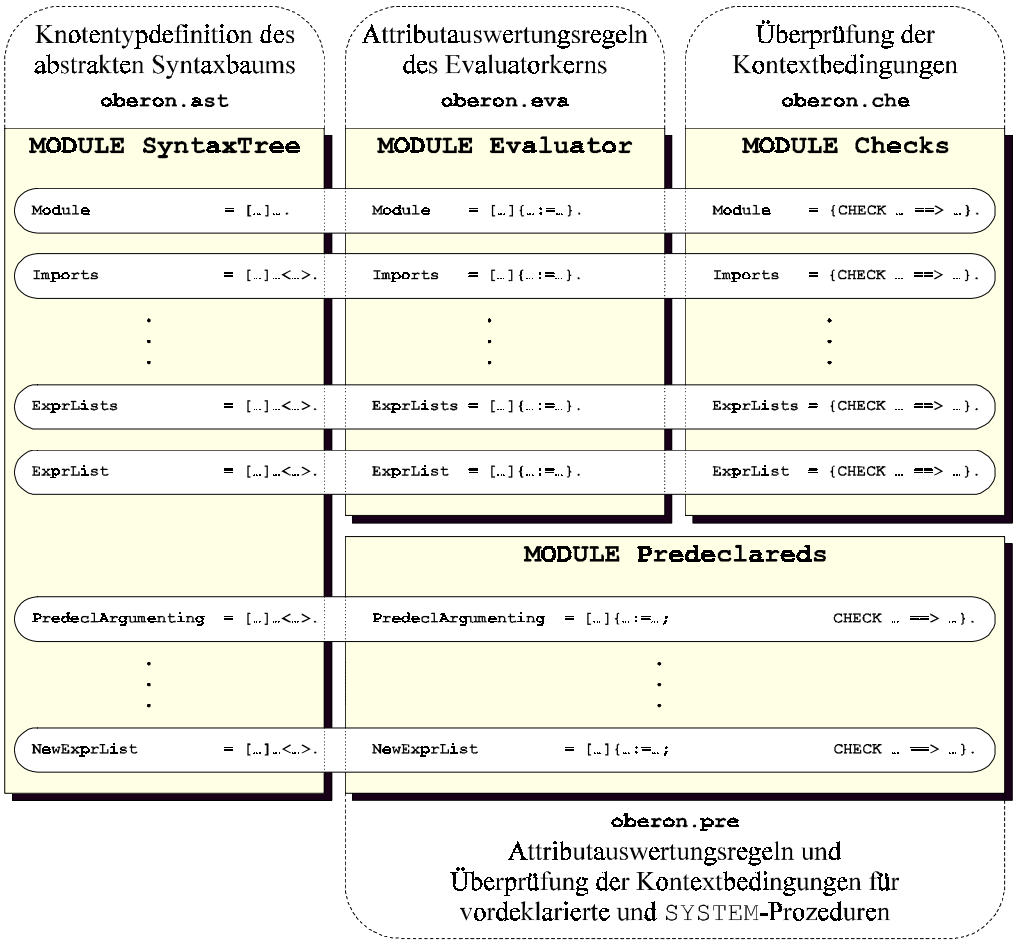

Abb. 14: Aufteilung der Evaluatorspezifikation auf verschiedene Module.

Innerhalb der Attributauswertungsregeln wurde versucht, so wenig Code wie möglich in der Werkzeugzielsprache zu formulieren und statt dessen die benötigte Funktionalität in Puma-Funktionen zu implementieren.

Der Evaluator führt im Rahmen der semantischen Analyse folgende Aufgaben durch:

- Aufbau der Symboltabelle aus den Deklarationen
- Aufbau der Typrepräsentationen aus den definierten Typen
- Auswertung von konstanten Ausdrücken
- Transformation provisorischer Teilbäume
- Überprüfung der Kontextbedingungen

Das Modul Evaluator enthält den Evaluatorkern, das Modul Checks die eigentlichen Überprüfungen der Kontextbedingungen und das Modul Predeclareds die Attributaus-

wertungsregeln im Zusammenhang mit den vordeklarierten Objekten sowie SYSTEM-Objekten. In der Datei oberon.eva ist zusätzlich das Modul SuperfluousApplications angegeben. Um Warnungen des Werkzeugs Ag für unbenutzte Attribute (,,attribute never used“) zu unterdrücken, werden in diesem Modul diese Attribute innerhalb von Bedingungsregeln als Parameter der Funktion superfluous_application_due_to_warning-_suppression übergeben. Diese konstante Funktion liefert den Wert TRUE. Diese Unterdrückung ungewollter Warnungen erleichtert bei der Entwicklung der Evaluatorspezifikation die Reaktion auf ,,begründete“ Warnungen.

Bei der Benennung von Attributen wurde folgende Konvention eingehalten: Die Namen von Attributen mit der Eigenschaft INHERITED bzw. SYNTHESIZED sind immer um das Suffix In bzw. Out erweitert. Dies erleichtert ihre Unterscheidung untereinander und von Attributen mit der Eigenschaft INPUT bzw. von Attributen ohne Eigenschaft, den sog. (Regel-)lokalen Attributen.

Es wurden Attribute definiert, die für die semantische Analyse nicht gebraucht werden, die aber für die spätere Codeerzeugung relevant sind. Dazu zählen u.a. die Attribute zur Berechnung der Objektgröße und für die Speicherung der Erweiterungsstufe einer Typerweiterung.

Kurzschlußauswertung

Die Auswertung eines logischen Ausdrucks ist als sog. Kurzschlußauswertung definiert. Attributexemplare von ValDontCareIn erhalten genau dann den Wert TRUE, falls sie sich in Knoten von Teilausdrücken eines Konstantenausdrucks befinden, die nicht weiter ausgewertet werden müssen. Da es jedoch angemessener ist, Konstantenausdrücke zur Übersetzungszeit stets vollständig auszuwerten, dient der Wert von ValDontCareIn lediglich der Unterdrückung etwaiger Fehlermeldungen (z.B. Division durch 0).

Kontextabhängige Fehlermeldung

Der Knotentyp IdentLists wird für die Repräsentation von Namenslisten sowohl bei Variablendeklarationen als auch bei Felddeklarationen eines Record-Typs benutzt. Um in Abhängigkeit dieses Kontexts unterschiedliche Fehlermeldungen ausgeben zu können, falls eine Deklaration zu einer Überschreitung einer bestimmten Objektgröße führt, wird die (Kodierung der) Fehlermeldung über das Attribut TooBigMsgIn weitergereicht.

Künstliche Abhängigkeiten

Im Knotentyp TypeDecl kann der Aufruf der Puma-Funktion E.DefineTypeEntry dazu führen, daß die Position des erstmaligen Auftretens des Basistyps eines Vorwärtszeigers (siehe 3.2.4 Vorwärtszeiger, S.111) wegen eines Seiteneffekts (aus Effizienzgründen) von E.DefineTypeEntry nicht mehr zur Verfügung steht. Da diese Position jedoch bei einer etwaigen Fehlermeldung benötigt wird, muß sie innerhalb der Attributauswertungsregel für Next:TableIn vor dem Aufruf von E.DefineTypeEntry ermittelt werden. Die künstliche Abhängigkeit

```
ForwardedPos BEFORE Next:TableIn
```

stellt dies sicher. Analog existiert im Knotentyp ProcDecl die künstliche Abhängigkeit

```
IsUndeclared BEFORE DeclSection:TableIn
```

wegen eines Seiteneffekts von E.DefineProcEntry (siehe 3.2.5 Vorausdeklaration, S.113).

Formale Parameterlisten

Formale Parameter werden im abstrakten Syntaxbaum gemäß folgendem Ausschnitt einer Ast-Spezifikation als Liste der ParId-Listen repräsentiert:

```
FPSections     =
<  mtFPSection = .
   FPSection   = Next : FPSections … ParIds .
>.
ParIds         =
<  mtParId     = .
   ParId       = Next : ParIds … .
>.
```

Die Repräsentation der durch die formalen Parameter gegebenen Signatur soll jedoch als einfache Liste erfolgen, in der die Reihenfolge der Signaturelemente der textuellen Reihenfolge der formalen Parameter entspricht:

```
SignatureRepr =
< mtSignature = .
  Signature   = next : SignatureRepr … .
>.
```

Deswegen erfolgt die Knüpfung dieser Liste beim Wiederaufstieg einer rechts-links Traversierung:

```
FPSections  = [ SignatureReprOut : tOB SYNTHESIZED ].
ParIds      = [ SignatureRepr    : tOB THREAD      ].

FPSections  = { SignatureReprOut       := cmtSignature;            }.
FPSection   = { ParIds:SignatureReprIn := Next:SignatureReprOut;
                SignatureReprOut       := ParIds:SignatureReprOut; }.
ParId       = { Next:SignatureReprIn   := SignatureReprIn;
                SignatureReprOut       := mSignature
                                          (Next:SignatureReprOut,…);
}.
```

Hierbei ist zu beachten, daß u.a. die folgenden beiden Auswertungsregeln implizit gelten:

```
mtFPSection = { SignatureReprOut := cmtSignature; }.
mtParId     = { SignatureReprOut := SignatureReprIn; }.
```

Nachfolgende Abbildung veranschaulicht den Aufbau einer Signatur für die formalen Parameter des Prozedurkopfs „PROCEDURE P(a, b: T1; c, d: T2)“.

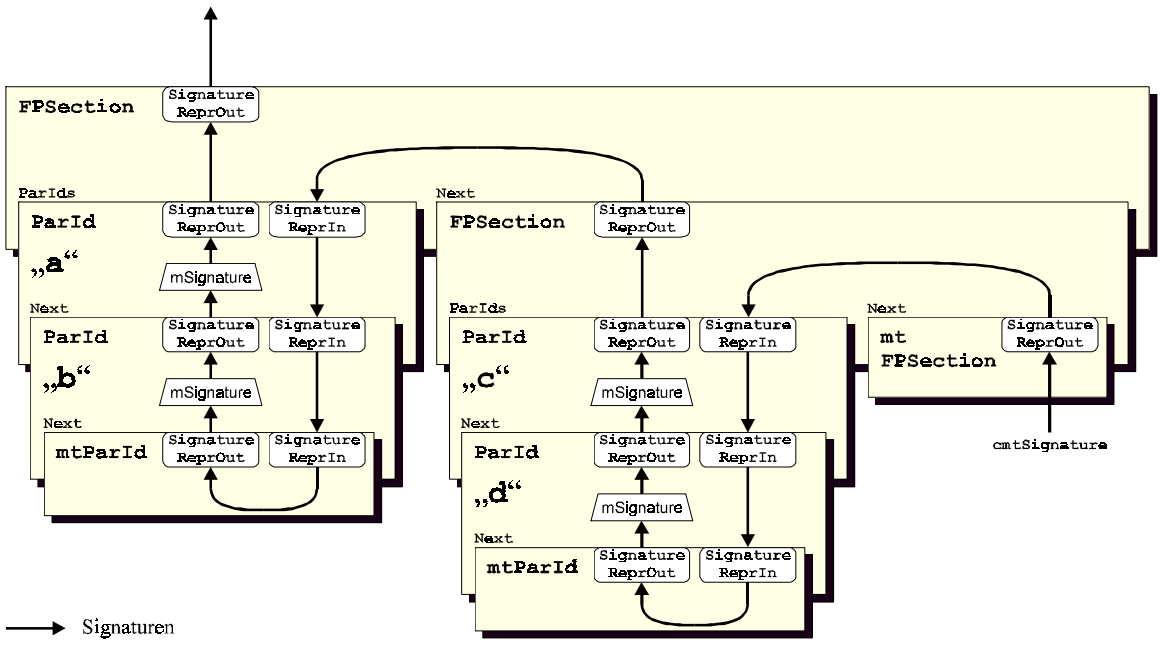

Abb. 15: Aufbau einer Signatur für formale Parameter.

Array-Typkonstruktion

In ähnlicher Weise wie bei den formalen Parametern erfolgt die Knüpfung der Repräsentation eines Array-Typs beim Wiederaufstieg aus der Ausdrucksliste. Die hierbei relevanten abstrakten (Teil-)Syntaxbäume sind definiert durch:

```
Type              = …
< …
  ArrayType       = ArrayExprLists Type .
>.
ArrayExprLists    =
< mtArrayExprList = .
  ArrayExprList   = Next : ArrayExprLists … .
>.
```

Die zu erzeugende Typrepräsentation ist jedoch strukturell nur eine einfache Liste von `ArrayTypeRepr`-Knoten:

```
TypeReprs       =
< …
  ArrayTypeRepr = … elemTypeRepr : TypeReprs .
>.
```

Anders als bei den formalen Parametern werden jedoch hier zwei verschieden benannte Attribute, mit der Eigenschaft `INHERITED` bzw. `SYNTHESIZED`, eingesetzt, um die unterschiedliche Bedeutung auszudrücken.

```
ArrayType       = [ TypeReprOut    : tOB SYNTHESIZED ].
ArrayExprLists  = [ ElemTypeReprIn : tOB INHERITED   ]
                  [ TypeReprOut    : tOB SYNTHESIZED ].
```

```
ArrayType       = { ArrayExprLists:ElemTypeReprIn
                                  := Type:TypeReprOut;
                    TypeReprOut := ArrayExprLists:TypeReprOut; }.
mtArrayExprList = { TypeReprOut := ElemTypeReprIn;             }.
ArrayExprList   = { TypeReprOut := mArrayTypeRepr
                                   (…,Next:TypeReprOut);       }.
```

Behandlung von Typfehlern

Die semantische Analyse wird nach Auftreten eines Typfehlers fortgesetzt, so daß es vorkommen kann, daß mit einer falschen bzw. unvollständigen Typrepräsentation weitergearbeitet werden muß. Um die Ausgabe von Folgefehlermeldungen zu unterdrücken, existiert die Typrepräsentation `ErrorTypeRepr`, die alle entsprechenden Kontextbedingungen erfüllt.

Eine Ausnahme von dieser allgemeinen Vorgehensweise bildet die Attributauswertungsregel für `TypeReprOut` des Knotentyps `DesignExpr`, welcher einen Bezeichner-Ausdruck repräsentiert. Für den Fall, daß es sich bei dem Bezeichner um einen Aufruf einer gewöhnlichen Prozedur handelt, erhält `Designator:TypeReprOut` die Typrepräsentation `mtTypeRepr`. Dieser Umstand wird im Modul `Checks` durch entsprechende Attributauswertungsregeln des Knotentyps `DesignExpr` als Fehler gemeldet.

Deshalb erfordert die Attributauswertungsregel für `TypeReprOut` im Knotentyp `DesignExpr` den Aufruf der Funktion `T.EmptyTypeToErrorType`, welche nur bei `mtTypeRepr` als Argument die Typrepräsentation `ErrorTypeRepr` liefert und damit der oben beschriebenen, generellen Vorgehensweise Rechnung trägt.

Konstante Mengenelemente

Die Behandlung des Mengenkonstruktors folgt der Methode, die Werte aller konstanten Mengenelemente in einem einzigen Mengenwert zusammenzufassen, wobei hier unter Mengenelement auch ein Elementbereich (a..b) verstanden wird. Dieser Mengenwert wird bei Auftreten eines nichtkonstanten Mengenelements wieder verworfen, d.h. zu `mtValue`. Dadurch läßt sich dann direkt feststellen, ob eine Menge insgesamt einen konstanten Wert besitzt.

Zur Unterscheidung eines einfachen konstanten Mengenelements (1) von einem Elementbereich mit konstantem ersten Wert und nichtkonstantem zweiten Wert (2) wird die Puma-Funktion `TT.ElementCorrection` eingesetzt, die in Abhängigkeit des Vorhandenseins des zweiten Ausdrucks die Wertrepräsentation des ersten Ausdrucks (für 1) oder die des zweiten Ausdrucks (für 2) liefert.

Werkzeugaufruf

Analog zum Aufruf des Werkzeugs Ast werden beim Aufruf des Werkzeugs Ag die relevanten Spezifikationsdateien (oberon.ast, oberon.eva, oberon.che und oberon.pre) konkateniert. Ag erzeugt aus diesen das Modula-2 Modul Eval.(md|mi), welches die Funktionalität des Evaluators zur Verfügung stellt (siehe Abbildung 16). Von den bei GROSCH [1991d] beschriebenen Optionen für Ag werden die folgenden verwendet:[1]

- D Generierung des Definitionsmoduls
- I Generierung des Implementationsmoduls

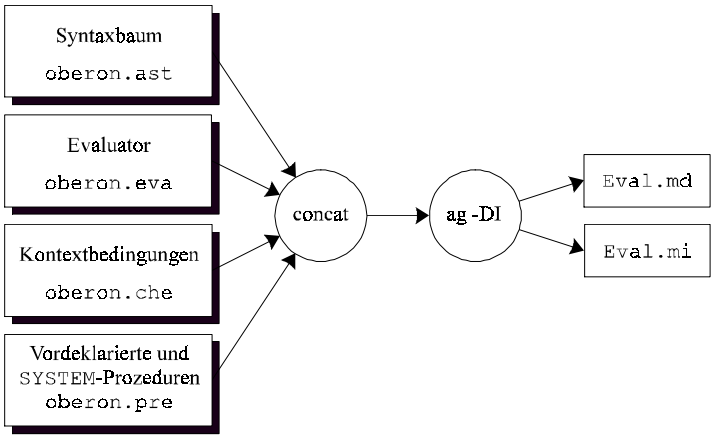

Abb. 16: Einsatz des Werkzeugs Ag zur Erzeugung eines Evaluator-Moduls.

1 In der eingesetzten Version der Werkzeuge liegen die Einzelwerkzeuge Ast und Ag als Teil des Programms cg vor. Es existieren zwei Shell-Skriptdateien mit Namen ast und ag mit identischem Inhalt, die das Programm cg mit ihren Kommandozeilenparametern aufrufen. Zur Zuordnung der Parameter zum konkreten Werkzeug werden deshalb Kleinbuchstaben für Ast und Großbuchstaben für Ag verwendet.

3.1.7 Baumtransformation (TT.pum)

Das Modul TT (Tree Transformation) stellt im wesentlichen Funktionen zur Transformation von Teilbäumen des abstrakten Syntaxbaums zur Verfügung. Darüberhinaus exportiert das Modul noch einfache Prädikate und Selektorfunktionen, die ebenfalls alle auf dem abstrakten Syntaxbaum arbeiten. Daneben besitzen einige Funktionen zusätzlich Argumente, deren Typen Evaluatorobjekttypen sind. Deswegen sind im Spezifikationskopf hinter dem Schlüsselwort TREE die beiden Namen Tree und OB aufgeführt.

Argumentkorrektur

Die Überprüfung der Übereinstimmung einer aktuellen Parameterliste mit der entsprechenden formalen Parameterliste ist in den zum Knotentyp ExprLists gehörenden Attributauswertungsregeln definiert. Damit diese auch für den Aufruf einer gewöhnlichen Prozedur, die keine aktuelle Parameterliste besitzt, vorgenommen wird, erzeugt die Puma-Funktion TT.ArgumentCorrection für diesen Fall eine leere Parameterliste.

Bei der Evaluation der Bezeichnerknoten werden schrittweise alle zu einem Bezeichner gehörenden Selektoren besucht. Vor dem Besuch des nächsten Selektorknotens wird dieser über den Parameter next der Funktion TT.ArgumentCorrection übergeben. Diese liefert für den besagten Fall dann einen neu erzeugten Argumenting-Selektorknoten, der eine leere Ausdrucksliste (mtExprList) besitzt, ansonsten liefert sie den übergebenen Selektorknoten unverändert zurück. Die Fallunterscheidung stützt sich hierbei auf die Argumente isCall (Bezeichner steht in einem Prozeduraufruf-Kontext) und type (bis zum aktuell besuchten Selektorknoten ermittelter Selektortyp).

Umwandlung von Selektoren

Die wegen der bereits erwähnten Mehrdeutigkeit der Originalgrammatik von Oberon-2 geknüpften Designor-Knoten (siehe 3.1.3 Abstrakter Syntaxbaum: Selektoren, S.76) werden durch die Funktion TT.DesignorToDesignation in die Knoten transformiert, welche die eigentlichen (Oberon-2) Selektoren repräsentieren. Hierbei werden unter Heranziehung von entsprechenden Kontextinformationen die Abbildungen

- "." (Selector) auf Record-Feldselektion (Selecting)
  oder Importierung (Importing),
- "(ExprLists)" (Argumentor) auf aktuelle Parameterliste (Argumenting)
  oder Typzusicherung (Guarding), oder
- "^" (Dereferencor) auf Zeigerdereferenzierung (Dereferencing)
  oder Aufruf einer an einen Basistyp gebundenen Prozedur (Supering)

unterschieden. Daneben erzeugt die Funktion für die impliziten Dereferenzierungen vor Record- und Array-Selektoren einen Dereferencing-Knoten. Die Ausdrucksliste eines Indexor-Knotens wird in eine Folge von Indexing-Knoten mit jeweils einem Ausdruck umgewandelt. Die Funktion behandelt ebenfalls die Extrahierung der Ausdrücke aus Argumentor-Knoten von vordeklarierten Prozeduren und SYSTEM-Prozeduren gemäß der Definition ihrer formalen Parameter (siehe 3.2.10 Vordeklarierte Prozeduren, S.126). TT.Designor-

ToDesignation stützt sich hierbei auf die nichtexportierten Puma-Funktionen, deren Namen alle das Präfix ExtractExpr besitzen.

Zu Testzwecken besteht die Möglichkeit, den Syntaxbaum vor bzw. nach der Evaluation in Quelltextdarstellung auszugeben. Für eine bessere Unterscheidung der verschiedenen Selektoren bei dieser Ausgabe werden für die Selektoren folgende Zeichen(-folgen) verwendet, die auch in den Quelltextkommentaren innerhalb von TT.DesignorToDesignation eingesetzt werden:

Knotentyp	*Zeichendarstellung*
Selector	\.
Indexor	\[ExprLists \]
Dereferencor	\^
Argumentor	\(ExprLists \)
Importing	$
Selecting	.
Indexing	[Exprs]
Dereferencing	^
Supering	!
Argumenting	(ExprLists)
Guarding	‘(Qualidents)

Werkzeugaufruf

Die Generierung des Baumtransformationsmoduls erfolgt durch den Aufruf des Werkzeugs Puma mit den Optionen:

- d Generierung des Definitionsmoduls
- i Generierung des Implementationsmoduls
- m Benutzung der Prozedur MakeTREE zur Knotenkonstruktion
- n Überprüfung der Parameter auf NoTREE (NIL) und entsprechende Fehlerbehandlung

Die letzte Option dient vor allem in der Entwicklungsphase zur Vermeidung einer Dereferenzierung von NIL während der Laufzeit des erzeugten Übersetzers. Neben der Spezifikationsdatei TT.pum benötigt Puma zur Generierung die beiden Dateien OB.TS und Tree.TS (vgl. Abb. 7, S.77).

3.1.8 Evaluatorhilfsfunktionen

Die Evaluatorhilfsfunktionen sind gemäß ihrer Zugehörigkeit zu den verschiedenen Aufgabenbereichen auf sechs verschiedene Puma-Spezifikationen verteilt:

- `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` exportierten Objekte

Die in diesen Spezifikationen definierten Puma-Funktionen (im folgenden nur noch Funktionen genannt) dienen im Evaluator der Durchführung der semantischen Analyse, indem sie die Manipulation der Evaluatorobjekte erlauben.

Zu einzelnen Aufgabenbereichen existiert in der entsprechenden Puma-Spezifikation eine Reihe von einfachen Prädikaten und (einfachen) Selektorfunktionen. Die einfachen Prädikate (`Is`…) werden dazu verwendet, bestimmte einfache Eigenschaften von Objekten zu überprüfen. Die Selektorfunktionen dienen dazu, bestimmte Knotenfelder eines Objekts zu selektieren, d.h. die Werte eines solchen Knotenfelds zu liefern. Dies geschieht bei manchen Funktionen nur unter der Voraussetzung, daß das übergebene Objekt bestimmte Eigenschaften besitzt.

Es existieren Paare von einfachen Prädikaten, die neben ihrer direkten Form (`Is`…) auch zusätzlich in ihrer negierten Form (`IsNot`…) definiert sind. Ein solches Prädikat liefert für ein fehlerrepräsentierendes Objekt (z.B. `ErrorEntry` oder `ErrorTypeRepr`) als Argument immer den Wert `TRUE` und kann somit nicht durch logische Negation seines Pendants eingespart werden.

Werkzeugaufruf

Die Generierung der Module mit den Evaluatorhilfsfunktionen erfolgt durch den bei GROSCH [1991b] beschriebenen Aufruf des Werkzeugs Puma mit den für alle Module gleichen Optionen:

`d`	Generierung des Definitionsmoduls
`i`	Generierung des Implementationsmoduls
`n`	Überprüfung der Parameter auf `NoTREE` (`NIL`) und entsprechende Fehlerbehandlung

Neben der jeweiligen Spezifikationsdatei benötigt Puma zur Generierung die Datei `OB.TS` (vgl. Abb. 13, S.89).

3.1.8.1 Symboltabelleneinträge (`E.pum`)

Das Modul `E` (Entry related functions) stellt Puma-Funktionen zur Verfügung, die alle im wesentlichen auf Symboltabelleneinträgen operieren.

Deklarationsbezogene Funktionen

Die Funktionen `Define(Const|Type|Var|Procedure)Entry` komplettieren jeweils einen Symboltabelleneintrag, der eine entsprechende Deklaration repräsentiert, die noch nicht vollständig und deswegen mit dem Deklarationszustand `TOBEDECLARED` in der Tabelle eingetragen ist (vgl. 3.2.1 Sichtbarkeitsregeln bei Deklarationen, S.106). Sie liefern diesen Sym-

boltabelleneintrag bzw. die weiter zu verwendende Symboltabelle zurück. Die Funktion `DefineTypeEntry` handhabt auch den Fall, daß diese Typdeklaration durch die Deklaration eines ,,Vorwärtszeigers“ bereits als Zeigerbasistyp in die Symboltabelle eingetragen worden ist (vgl. 3.2.4 Vorwärtszeiger, S.111). Ähnlich dazu berücksichtigt die Funktion `DefineProcedureEntry` auch Vorausdeklarationen. Die beiden letzteren Funktionen verwenden die nichtexportierte Funktion `LinkTypeToEntry`, um den Verweis der (Prozedur-)Typrepräsentation auf ihren Deklarationseintrag zu realisieren. Die Funktion `ChooseProcedureEntry` dient ebenfalls der Behandlung von Vorausdeklarationen.

Suchfunktionen

Die Funktionen `Lookup[0|Extern|Forward]` dienen der Suche in der Symboltabelle nach einem Eintrag mit einem bestimmten Namen. Existieren mehrere Einträge gleichen Namens, wird der zuletzt eingetragene Eintrag geliefert. Sie liefern `cErrorEntry`, falls die Suche erfolglos ist. Dabei sucht `Lookup` (möglicherweise) in der gesamten Symboltabelle, `Lookup0` nur bis zur nächsten Sichtbarkeitsbereichsgrenze, `LookupExtern` in der Symboltabelle eines importierten Moduls, und `LookupForward` sucht im lokalen Sichtbarkeitsbereich nur nach einer Vorwärtsdeklaration einer Zeigerbasis. Letztere liefert im Gegensatz zu den anderen Suchfunktionen bei einer erfolglosen Suche einen Knoten vom Knotentyp `mtEntry`.

Prüffunktionen

Die Funktionen `CheckUnresolvedForward(Pointers|[Bound]Procs)` dienen zur Überprüfung der Symboltabelle auf nicht aufgelöste Vorwärts- oder Vorausdeklarationen. Für jeden gefundenen Eintrag wird eine entsprechende Fehlermeldung ausgegeben. Sie liefern die übergebene Symboltabelle unverändert zurück.

Sonstige Funktionen

Die Funktion `ForwardProcOnly` ,,filtert“ Einträge, die Vorausdeklarationen von Prozeduren repräsentieren. Nur diese Einträge werden unverändert zurückgeliefert. Während für alle anderen Einträge `cErrorEntry` zurückgegeben wird.

Die Funktion `MaxExportMode` berechnet die Maximumsfunktion über Exportmarkierungen bezüglich der Ordnung `PRIVATE < READONLY < PUBLIC`. Sie wird bei der Bestimmung der Exportmarkierung von Prozeduren eingesetzt, für die eine Vorausdeklaration existiert (vgl. 1.1.10 Sprachdefinition von Oberon-2: Prozedurdeklarationen, S.18).

Die Funktion `EntryWithed` wird im Zusammenhang mit dem Typerweiterungsmechanismus eingesetzt und wird in Abschnitt 3.2.6 Typerweiterungen, S.115, näher beschrieben.

Funktion zur Handhabung von Zeigerbasistypen

Die Funktion `ApplyPointerBaseIdent` handhabt die Definition eines Zeigertyps der Form `POINTER TO` *T*, wobei *T* ein unqualifizierter Name ist und möglicherweise noch nicht deklariert wurde (Vorwärtsdeklaration). Sie liefert für alle legalen Fälle als Resultat den entsprechenden Eintrag, der die (Typ-)Deklaration des Basistyps repräsentiert, im Fehlerfall `cErrorEntry`. Es werden hierbei die folgenden legalen Fälle unterschieden:

- Eine Vorwärtsdeklaration ist erlaubt und *T* ist nicht deklariert.
- Eine Vorwärtsdeklaration ist erlaubt und *T* ist nicht deklariert, aber bereits in einer Typdefinition als Zeigerbasis benutzt.
- Die Definition des Zeigertyps steht innerhalb der Deklaration von *T* (erlaubte rekursive Typdeklaration).
- *T* ist deklariert.

Eine detaillierte Beschreibung erfolgt in Abschnitt 3.2.4 Vorwärtszeiger, S.111.

3.1.8.2 Typen (`T.pum`)

Das Modul `T` (Type related functions) stellt Funktionen zur Verfügung, deren inhaltlicher Zusammenhang die Bearbeitung von Typrepräsentationen ist.

Zuweisungsfunktionen

Im Zusammenhang mit der Behandlung von rekursiven Typdeklarationen (vgl. S.108) dienen die Funktionen `Complete(OpenArray|Record|Pointer|Procedure)TypeRepr` der Komplettierung einer bereits provisorisch angelegten Typrepräsentation. Jede dieser Funktionen belegt die Knotenfelder der übergebenen entsprechenden Typrepräsentation mit den übergebenen Werten und liefert die so vervollständigte Typrepräsentation zurück. Die Funktion `CopyArrayTypeRepr` dient ebenfalls dieser Komplettierung. Im Gegensatz zu den anderen Komplettierungsfunktionen selektiert sie aber die Werte aus einer ebenfalls bereits angelegten Typrepräsentation.

Typklassenprädikate

Es existieren Funktionen, die die Zugehörigkeit einer Typrepräsentation zu einer Klasse überprüfen. Neben den durch die Sprachdefinition eingeführten (z.B. `IsIntegerType` für *ganzzahlige* Typen) wurden die neuen Typklassen *shiftable types* und *registerable types* definiert. Diese werden im Zusammenhang mit den formalen Parametern von `SYSTEM`-Prozeduren verwendet.

Prädikate für Kontextbedingungen

Diese Prädikate modellieren einige durch die Sprachdefinition eingeführte formale Begriffe (z.B. `AreSameTypes` für *derselbe* Typ) und werden für die Überprüfung der Kontextbedingungen eingesetzt.

Funktionen für die Ausdruckskompatibilität

Der Modellierung des in der Sprachdefinition eingeführten Begriffs der Ausdruckskompatibilität dienen diese Funktionen. Ihre Funktionalität ist durch ihre Namen bereits ausgedrückt. Das Präfix `SetOr` im Namen zweier Funktionen zeigt an, daß sie auch die Behandlung der Typrepräsentation von `SET` mit übernehmen. Die Funktion `RelationInputType` liefert die Repräsentation des Typs, auf dem eine Vergleichsoperation durchgeführt werden soll, an welchen also die Typen der Operanden angepaßt werden müssen (Typanpassung). Das Prädikat `IsLegalOrderRelationInputType` testet für eine solche Typrepräsentation, ob sie als Operandentyp einer Ordnungsrelation (<, <=, > und >=) erlaubt ist.

Weitere Funktionen für Kontextbedingungen

Die hierunter aufgeführten Funktionen dienen der Sicherstellung konkreter Kontextbedingungen, wie beispielsweise dem Test auf erlaubte Resultatstypen von Funktionsprozeduren. Neben Prädikaten sind hierbei auch Filterfunktionen definiert, welche das übergebene Objekt unverändert zurückliefern, falls das Objekt bestimmte Eigenschaften besitzt. Besitzt es diese Eigenschaften nicht, wird ein ausgezeichnetes fehlerrepräsentierendes Objekt zurückgegeben.

Funktionen für Typerweiterung und gebundene Prozeduren

Für den Mechanismus der Typerweiterung (siehe S.115) sowie der typgebundenen Prozeduren (siehe S.118) werden diese Funktionen zur Manipulation der beteiligten Typrepräsentationen eingesetzt.

Typoperationen

In dieser Gruppe sind einerseits Funktionen definiert, die innerhalb der Selektoren bei Bezeichnern für die dort notwendige Typberechnung eingesetzt werden. Andererseits dienen einige Funktionen der Behandlung von vordeklarierten Prozeduren, deren Argumente Typen sind.

3.1.8.3 Werte (`V.pum`)

Neben den einfachen Prädikaten und Selektorfunktionen stellt das Modul `V` (Value related functions) Funktionen auf Wertrepräsentationen zur Verfügung. Eine Gruppe von Funktionen ermöglicht die Behandlung von Mengenwerten und Markenbereichen, bei der Überprüfung von Case-Marken. Daneben existieren Funktionen, die im Rahmen der Konstantenauswertung eingesetzt werden (vgl. 3.2.8 Ausdrücke, S.121). Sie gliedern sich in Funktionen für die Auswertung von einstelligen und zweistelligen Operationen sowie von vordeklarierten Prozeduren, die zur Übersetzungszeit auswertbar sind.

3.1.8.4 Typanpassungen (`CO.pum`)

Das Modul `CO` (COercion related functions) stellt lediglich zwei Funktionen zur Verfügung. Während die Funktion `GetCoercion` die Typanpassung bezüglich eines Ursprungs- und Zieltyps liefert, realisiert die Funktion `DoCoercion` die gemäß einer Typanpassung notwendige Umwandlung einer Wertrepräsentation.

3.1.8.5 Formale Parameterlisten (`SI.pum`)

Zur Handhabung von formalen Parameterlisten stellt das Modul `SI` (SIgnature related functions) neben einfachen Prädikaten und Selektorfunktionen die Funktion `AreMatchingSig-`

natures und das Prädikat IsCompatibleParam zur Verfügung, deren Funktionalität durch ihre Namensgebung ausreichend erklärt ist.

3.1.8.6 Vordeklarierte Tabelle und SYSTEM-Tabelle (PR.pum)

Die Puma-Spezifikation in der Datei PR.pum stellt lediglich die beiden Puma-Funktionen GetTablePREDECL und GetTableSYSTEM zur Verfügung. Diese Puma-Funktionen liefern jeweils einen Verweis auf die Symboltabelle, welche die Einträge der vordeklarierten Objekte bzw. der vom Modul SYSTEM exportierten Objekte enthält.

Lediglich der erstmalige Aufruf einer dieser Puma-Funktionen führt zum Aufbau sowohl der vordeklarierten als auch der SYSTEM-Tabelle. Jeder erneute Aufruf einer GetTable-Funktion liefert einen Verweis auf die entsprechende, beim erstmaligen Aufruf erzeugte Tabelle.

Zum Aufbau dieser Tabellen existiert für jedes vordeklarierte Objekt eine eigene Puma-Funktion, die im wesentlichen das Ergebnis des entsprechenden Konstruktors zur Erzeugung des Symboltabelleneintrags liefert. Die für Prozedureinträge hier verwendete Typrepräsentation wird in 3.2.10 Vordeklarierte Prozeduren, S.126, näher beschrieben. Der eigentliche Tabellenaufbau erfolgt in der Puma-Funktion BuildTables durch wiederholten Aufruf der Puma-Funktion Enter, der u.a. jeweils das Ergebnis der entsprechenden Konstruktionsfunktion übergeben wird.

3.1.9 Ausgabefunktionen (TD.pum, OD.pum)

Zur Unterstützung der Fehlersuche im erzeugten Front-End des Übersetzers wurde das Hauptmodul of so gestaltet, daß die Möglichkeit besteht, das Durchlaufen der einzelnen Phasen des Front-Ends zu steuern und Zwischenergebnisse in geeigneter Form auszugeben. Deshalb wurden Puma-Funktionen erstellt, welche die Ausgabe des geknüpften Syntaxbaums in Quelltextdarstellung (TD.pum) sowie der Symboltabellen (OD.pum) ermöglichen. Die zur Verfügung gestellte Funktionalität stützt sich im wesentlichen auf das Modula-2 Modul ED.

3.1.10 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. Diese Aufteilung ist zudem der Klarheit des Gesamtentwurfs dienlich.

3.1.10.1 Fehlerbehandlung (`Errors`, `ERR`, `ErrLists`, `Errors.Tab`)

Das vom Werkzeug Lalr generierte Modul `Errors` wurde dahingehend ergänzt, daß es statt der eigenen Ausgabe von Fehlermeldungen, die Prozedur `MsgI` aus dem Modul `ERR` dazu benutzt. Letzteres stellt zusätzlich, neben zwei weiteren Ausgabeprozeduren, für die verschiedenen Fehlermeldungen entsprechende Konstanten zur Verfügung. Die Fehlermeldungstexte werden aus der Textdatei `Errors.Tab` gelesen. Die bei der Übersetzung auftretenden Fehlermeldungen werden zusammen mit ihren Quelltextpositionen in einer vom Modul `ErrLists` zur Verfügung gestellten Datenstruktur gesammelt. Nach abgeschlossener Übersetzung erfolgt die Ausgabe dieser Fehlermeldungen in der Reihenfolge ihrer Quelltextpositionen.

3.1.10.2 Oberon-2 Datentypen (`OT`)

Das Modul `OT` (Oberon-2 data Types) kapselt die Datentypen, welche zur Repräsentation der Oberon-2 Typen verwendet werden. Neben den Typen (`oBOOLEAN`, `oCHAR`, …)[1] exportiert es Konstanten, welche die Anzahl von Bytes repräsentieren, die ein Typ bei der Speicherung einnimmt (`SIZEoBOOLEAN`, `SIZEoCHAR`, …) und welche die kleinsten bzw. größten Werte dieser Typen darstellen (`MINoBOOLEAN`, `MAXoBOOLEAN`, …).

Desweiteren sind in `OT` Prozeduren definiert, die die Umwandlung von Oberon-2 Typen in Modula-2 Typen (und umgekehrt) sowie in eine textuelle Repräsentation realisieren. Für die Attributauswertung existieren Prozeduren, welche u.a. für die Behandlung von `SET` und Case-Marken sowie zur Ermittlung von Typgrößen eingesetzt werden. Schließlich sind im Modul `OT` die Operationen und vordeklarierten Prozeduren implementiert, die im Rahmen der Konstantenauswertungen benötigt werden.

3.1.10.3 Hauptmodul und Treiber (`of`, `DRV`, `TBL`)

Das Hauptmodul `of` (Oberon-2 Front-end) dient der Analyse der Kommandozeilenparameter und -optionen sowie dem Anstoßen der entsprechenden Prozeduren aus `DRV` in Abhängigkeit der Kommandozeilenparameter (s.u.). Das Modul `DRV` (front-end DRiVer) stellt die eigentliche Funktionalität des implementierten Front-Ends zur Verfügung: Die Prozeduren `DRV.Compile` und `DRV.Import` steuern jeweils die ,,Übersetzung“ eines Oberon-2 Moduls. Während `DRV.Compile` hierbei lediglich für die Bearbeitung des Oberon-2 Hauptmo-

1 Der Grund für die Modellierung des Oberon-2 Typs `LONGREAL` als Record-Typ und nicht durch den (Mocka-)Modula-2 Typ `LONGREAL` liegt in der Umgehung eines Fehlers in der vom Werkzeug Ast erzeugten eigenen Freispeicherverwaltung. Diese ist im erzeugten Baummodul realisiert und wird fehlerhaft so initialisiert, daß eine durch die Rechnerarchitektur erforderte Ausrichtungsbedingung (Alignment) für doppelt lange Fließkommawerte nicht eingehalten wird. Dieser Umstand führt bei einer Zuweisung zur Laufzeit zu einem Busfehler (bus error) während der Speicherung eines solchen Werts, da der Transport über ein Fließkommaregisterpaar durchgeführt wird. Durch ,,Verpacken“ der Variablen in einem Record erfolgt die Zuweisung durch eine byteweise arbeitende Kopierroutine.

duls zuständig ist, liefert die Funktionsprozedur `DRV.Import` zusätzlich die Symboltabelle des bearbeiteten Oberon-2 Moduls. `DRV.Import` wird innerhalb der Attributauswertung einer Importierung aufgerufen. Die von ihr gelieferte Symboltabelle umfaßt alle Deklarationen, die im globalen Sichtbarkeitsbereich des entsprechenden Moduls stehen, einschließlich der Objekte ohne Exportmarkierungen. Das Modul `TBL` (symbol TaBLe storage) stellt die beiden Prozeduren `TBL.Store` und `TBL.Retrieve` zur Speicherung und Wiedererlangung einer Symboltabelle zur Verfügung. Diese Prozeduren werden von `DRV.Import` eingesetzt, damit eine zu importierende Symboltabelle nur einmal aus dem entsprechenden Oberon-2 Quelltextmodul aufgebaut werden muß, obwohl dieses mehrfach von verschiedenen Oberon-2 Modulen importiert wird.

Kommandozeilenparameter

Das erzeugte Programm kann gemäß folgendem Schema aufgerufen und mit Parametern versehen werden:

`of` {[`-`*option(en)*] [*datei(en)*]}

Die Angabe der zu *übersetzenden* Dateien erfolgt durch Nennung ihrer Namen. Hierbei kann die Angabe des Dateisuffixes `.ob2` unterbleiben. Es existieren die folgenden Optionen:

`h`	Ausgabe von Information zur Benutzung (usage)
`Y`	Durchführung der Scanner-Phase und Ausgabe der textuellen Repräsentation der gelieferten Symbole
`b`	Ausgabe der Modulschnittstelle (unvollständig implementiert)
`d`	Durchführung der Parsierung und Ausgabe des abstrakten Syntaxbaums
`D`	Ausgabe des abstrakten Syntaxbaums nach der Attributauswertung
`Tp`	Ausgabe der Symboltabelle, welche die vordeklarierten Objekte enthält
`Ti`	Ausgabe der importierten Symboltabellen
`Ts`	Ausgabe der für Anweisungsfolgen gültigen (lokalen) Symboltabellen
`Tw`	Ausgabe der (lokalen) Symboltabellen, welche für Anweisungsfolgen gelten, die innerhalb von With-Anweisungen stehen
`Ta`	Angabe der Adressen von Typrepräsentationen bei der Tabellenausgabe
`Ta`	Angabe der Typgrößen bei der Tabellenausgabe
`Ta`	Angabe der (relativen) Adressen von Variablen bei der Tabellenausgabe
`E`	Ausgabe der Prozedurnamen und -parameter bei Aufrufen von Prozeduren aus dem Modul `ED` (Fehlersuche)
`l`*dir*	`of` sucht die benötigten Tabellen (`Scan.Tab`, `Pars.Tab` und `Errors.Tab`) im Verzeichnisses *dir*

3.1.10.4 Allgemeine Hilfsfunktionen (`FIL`, `POS`, `ED`, `O`, `STR`, `UTI`)

Die hier aufgeführten Module sind unter einer Überschrift zusammengefaßt, da jedes für sich nur eine geringe Funktionalität besitzt und/oder lediglich zu Ausgabezwecken in der Entwicklungsphase eingesetzt wurde.

- `FIL` (FILe operations) kapselt die für die Behandlung von verschachtelten (Eingabe-)Dateien benötigte Funktionalität.
- `POS` (source POSition handling) stellt einen abstrakten Datentyp zur Verfügung, der zur Repräsentation von Quelltextpositionen benutzt wird.[1]

1 Eine neuere Version der Compiler-Compiler-Toolbox beinhaltet ein solches Modul bereits.

- `ED` (simple text EDitor functionality) stellt einen abstrakten Datentyp ,,Editor“ für den leichteren Aufbau einer formatierten Ausgabe (Einrückstruktur) zur Verfügung. Diese Funktionalität wird vor allem für die Ausgabe eines abstrakten Syntaxbaums in Quelltextdarstellung benötigt.
- `O` (Output routines) exportiert einen ganzen Satz von Ausgabeprozeduren, deren Funktionalität an jene des Moduls `InOut` angelehnt ist. Die Modulschnittstelle von `O` ist im Gegensatz zu `InOut` einfacher gestaltet.
- `STR` (STRing functions) realisiert diverse Funktionen auf Zeichenketten (Zeichen-Arrays).
- `UTI` (miscellaneous UTIlities) beinhaltet verschiedenste Prozeduren, vor allem zur Umwandlung von Basistypwerten, welche wegen ihrer geringen Anzahl ein jeweils eigenes Modul nicht gerechtfertigt hätten.

3.2 Spezielle Problembereiche

3.2.1 Sichtbarkeitsregeln bei Deklarationen

Die Sprachdefinition von Oberon-2 legt fest, daß der Sichtbarkeitsbereich eines Objekts textuell an der Stelle seiner Namensnennung bei der Deklaration beginnt. Daraus folgt, daß ein Objekt bereits ab seiner Namensnennung gleichnamige Objekte umfassender Blöcke ,,überdeckt“ (vgl. Anmerkung zur Sprachdefinition von Oberon-2, S.33). Die Konstantendeklaration innerhalb der Prozedur `P` im Programmfragment

```
CONST N=1;
PROCEDURE P;
  CONST N=N+1;
…
```

wäre beispielsweise dadurch verboten.

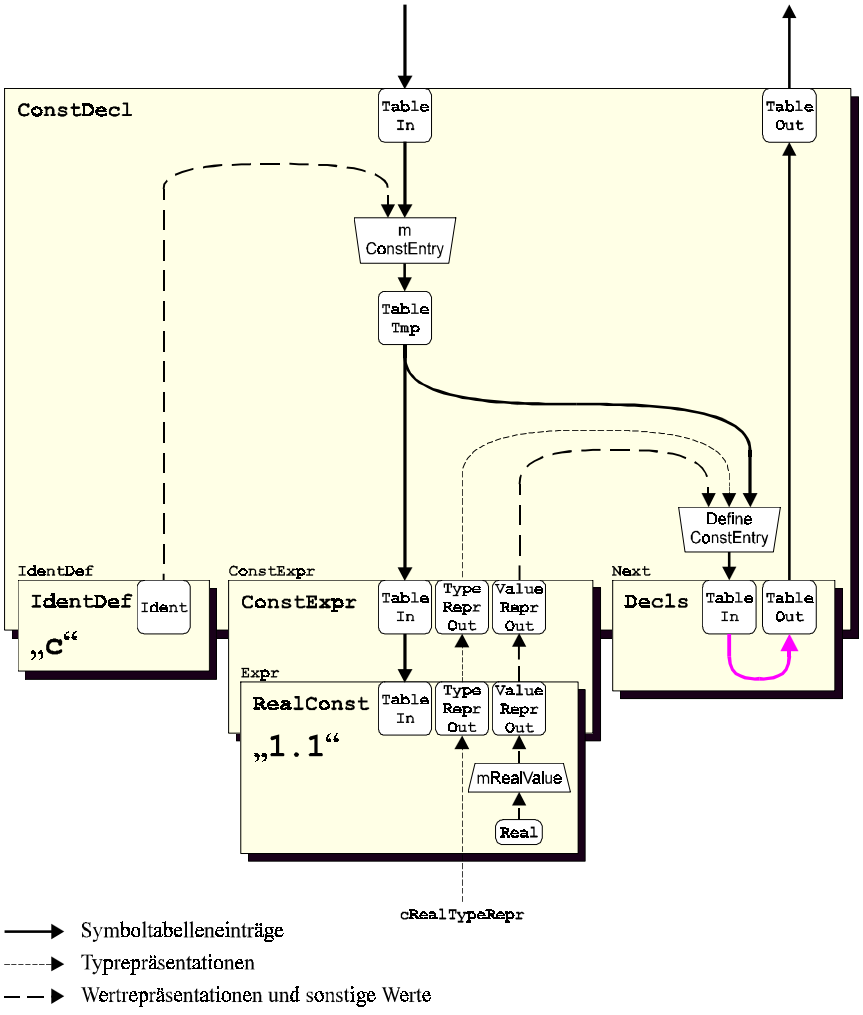

Abb. 17: Aufbau der Symboltabelle bei einer Konstantendeklaration.

Die Realisierung dieser Sichtbarkeitsregeln erfolgt dadurch, daß nach dem Besuch des `IdentDef`-Knotens bereits ein entsprechender Eintrag für das Objekt in der Symboltabelle angelegt wird, bevor diese an den Ausdrucksknoten weitergereicht wird. Dieser Eintrag ist

noch nicht vollständig und sein Deklarationszustand wird im Attribut `declStatus` als `TOBEDECLARED` eingetragen. Bevor die Symboltabelle schließlich an die nächste Deklaration weitergereicht wird, erfolgt eine Komplettierung des Eintrags über eine Puma-Funktion. Diese ändert den Deklarationsstatus in `DECLARED` und ergänzt die fehlenden Werte. Den Datenfluß skizziert Abbildung 17 für das Beispiel ,,`CONST c=1.1`“.

Diese Vorgehensweise erfolgt grundsätzlich bei allen Deklarationen von Objekten.

Die Einhaltung dieses Schemas bei Variablendeklarationen erfordert die Spezifizierung von zwei Attributen für den Aufbau der Symboltabelle innerhalb des Knotentyps `IdentLists`. Dies sind die Attribute `Table1` und `Table2`, die beide die Eigenschaft `THREAD` besitzen. In `Table1` werden die einzelnen Variablendeklarationen wiederum unvollständig eingetragen. Nach der Ermittlung des Typs werden in `Table2` die Einträge durch die Puma-Funktion `E.DefineVarEntry` komplettiert. Für die Berechnung der Attribute `Table1` und `Table2` durch den Evaluator sind zwei Pässe über Knoten vom Typ `IdentLists` erforderlich. Abbildung 18 veranschaulicht diesen Sachverhalt am Beispiel einer Variablendeklaration der Form ,,`VAR v, w: T`“.

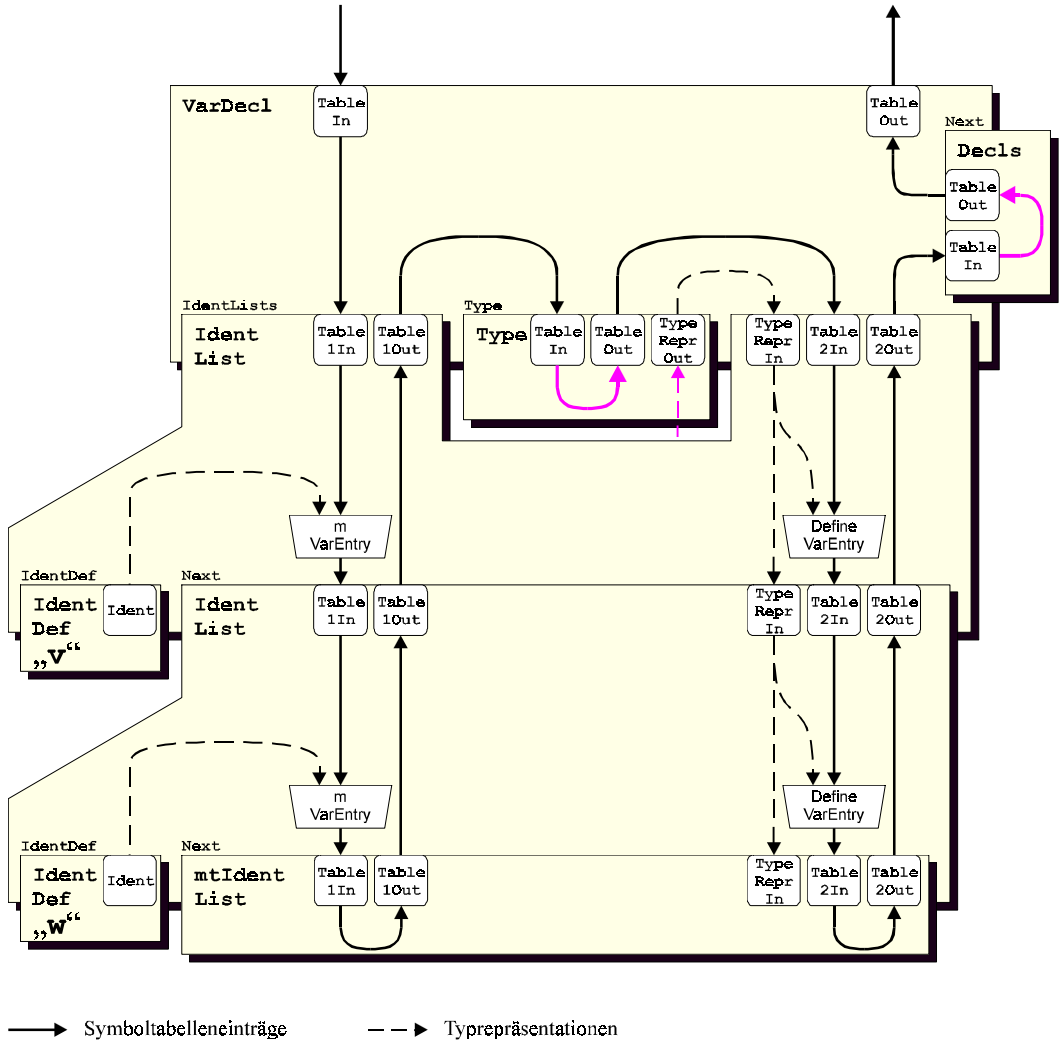

Abb. 18: Aufbau der Symboltabelle bei einer Variablendeklaration.

3.2.2 Rekursive Typdeklarationen

Eine Typdeklaration bindet einen Typ *T* an einen Namen *N*, indem die Typrepräsentation für *T* in der Symboltabelle in einem Typeintrag mit dem Namen *N* gespeichert wird. Für den Fall, daß *T* ein noch unbenannter strukturierter Typ, Zeigertyp oder Prozedurtyp ist, besteht die Möglichkeit einer rekursiven Bezugnahme auf *T* innerhalb dieser Deklaration. Die Sprachdefinition führt Fälle auf, in denen eine solche Bezugnahme gestattet ist (vgl. 1.1.6 Typdeklarationen, S.6).

Im Knotentyp `NamedType` wird daher zur Feststellung des Vorliegens einer rekursiven Typdeklaration der Deklarationszustand des applizierten Typeintrags über das Puma-Prädikat `E.IsNotToBeDeclared` abgefragt. Das Boolesche Attribut `IsVarParTypeIn` definiert dann die Legalität dieser Rekursion. Das Attribut `IsVarParTypeIn` zeigt in einem `Type`-Knoten an, daß sein Vaterknoten vom Typ `FPSection` mit `REFPAR` als Parametermodus ist und er somit den Typ eines formalen Var-Parameters repräsentiert. Hierbei ist zu beachten, daß ein benannter Zeigerbasistyp im Knotentyp `PointerToIdType` eigenständig gehandhabt wird und deshalb in der Legalitätsprüfung im Knotentyp `NamedType` nicht berücksichtigt werden muß.

Die Repräsentation eines strukturierten Typs *T'* bzw. eines Prozedurtyps *T'* ist zur Vereinfachung durch **direkte** Verweise auf `TypeRepr`-Knoten implementiert, welche die Untertypen von *T'* repräsentieren. Lediglich bei einem Zeigertyp erfolgt eine Indirektion über einen `TypeEntry`-Knoten für den Zeigerbasistyp (vgl. 3.2.4 Vorwärtszeiger, S.111)). Aus diesem Grund ist der in der Symboltabelle eingetragene, unvollständige Typeintrag zum Aufbau einer (rekursiven) Typrepräsentation nicht ausreichend. Für den Knotentyp `Type` wird daher über das Attribut `FirstTypeReprOut` im voraus der Wurzelknoten der zu erstellenden Typrepräsentation für *T* geliefert, für dessen Erzeugung keine weiteren Informationen aus den Kindern von `Type` benötigt werden. Dieser Wurzelknoten wird nach vollständiger Auswertung der Untertypen um deren Typrepräsentationen durch die Puma-Funktionen `Complete*TypeRepr` komplettiert. Abbildung 19 verdeutlicht den Gesamtzusammenhang am Beispiel der folgenden rekursiven Typdeklaration:

```
TYPE T = PROCEDURE (VAR p: T);
```

3.2.3 Elimination von Typ-Bezeichnern in Ausdrücken

Die Sprachdefinition gestattet die Applizierung eines Typbezeichners als Ausdruck, der als zweiter Operand eines `IS`-Operators aufgeführt ist. Zusätzlich muß der aktuelle Parameter der vordeklarierten Prozeduren `MAX`, `MIN`, `SIZE` sowie der erste Parameter der Prozedur `VAL` aus dem Modul `SYSTEM` ebenfalls ein Typausdruck sein. Die Behandlung von Typausdrücken als Ausdrücke erfordert jedoch einen gewissen Mehraufwand bei der Spezifizierung. Aus diesem Grund wurden Typausdrücke separat behandelt.

Für den `IS`-Operator wurde dies durch Einschränkung der kontextfreien Grammatik auf `Designator` für den ersten Operanden und `Qualident` für den zweiten Operanden realisiert (siehe Grammatiktransformation, S.81).

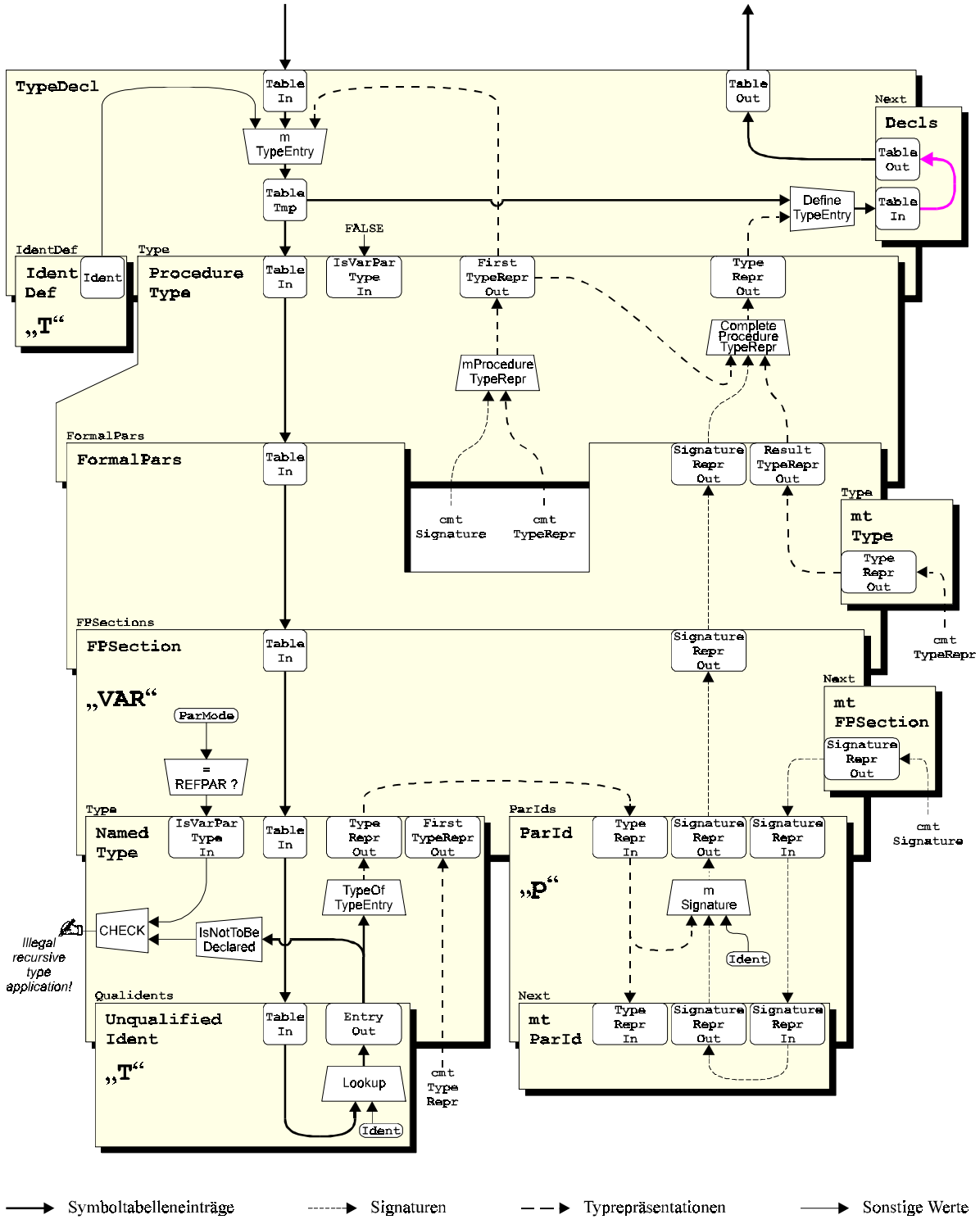

Abb. 19: Datenfluß bei einer rekursiven Typdeklaration.

Für die aktuellen Parameter der obengenannten Prozeduren wurde dies erreicht, indem es für jede vordeklarierte Prozedur und `SYSTEM`-Prozedur einen eigenen Knotentyp gibt, der zur Repräsentation der Liste der aktuellen Parameter herangezogen wird (vgl. 3.2.10 Vordeklarierte Prozeduren, S.126). Ein Knoten dieser Knotentypen wird durch die Puma-Funktion `TT.DesignorToDesignation` (siehe S.96) aus der Liste der Ausdrücke, welche die aktuellen Parameter repräsentieren, erzeugt. Handelt es sich um eine typparametrisierte Prozedur und besteht der erste Ausdruck der Liste genau aus einem (qualifizierten) Namen, wird dieser aus dem Ausdruck extrahiert und im erzeugten Knoten selbst gespeichert.

In beiden Fällen ist somit die Handhabung von Typausdrücken trivial, da der typrepräsentierende Unterbaum im abstrakten Syntaxbaum nun vom Knotentyp `Qualidents` ist, der bei der Attributauswertung direkt den entsprechenden Symboltabelleneintrag liefert.

3.2.4 Vorwärtszeiger

Bei der Deklaration eines Zeigertyps der Form `TP = POINTER TO T` erlauben die Sichtbarkeitsregeln von Oberon-2, daß die Deklaration des Zeigerbasistyps `T` textuell hinter der Zeigertypdeklaration erfolgt. Der Zeigertyp `TP` wird (ähnlich dem Begriff ,,Vorausdeklaration" bei Prozeduren) als *Vorwärtszeiger* bezeichnet. Die Deklaration eines Vorwärtszeigers führt zu einer sog. *Vorwärtsdeklaration* von `T`. Diese besitzt im Gegensatz zu ,,normalen" Deklarationen spezielle Eigenschaften, die ab der *eigentlichen* Deklaration von `T` ihre Gültigkeit verlieren.

Da bei einem anonymen sowie bei einem in einem anderen Modul deklarierten Typ als Zeigerbasistyp keine Möglichkeit zu einer Vorwärtsdeklaration besteht und diese Typen zudem bereits kontextfrei erkannt werden können, wurden (bezüglich der Zeigerbasistypen) die im Abschnitt 3.1.4, S.81 beschriebenen Transformationen der kontextfreien Grammatik bezüglich der Zeigerbasistypen vorgenommen. Somit kann in jeweils einer Alternative des Nichterminals PointerType genau der entsprechende Knoten vom Knotentyp `PointerToIdType`, `PointerToQualIdType` oder `PointerToStructType` erzeugt werden. Neben der Einschränkung auf die legalen Zeigerbasistypen durch Formulierung entsprechender Regeln der kontextfreien Grammatik ermöglicht dieser Ansatz die gezielte Behandlung von Vorwärtszeigern genau für den Knotentyp `PointerToIdType`.

Abbildung 20 zeigt die Symboltabelle nach Bearbeitung der Deklarationen

```
TYPE P1 = POINTER TO T;
     P2 = POINTER TO T;
     T  = RECORD END;
     P3 = POINTER TO T;
```

bei der Attributauswertung, auf die sich alle für die Behandlung von Vorwärtszeigern relevanten Fälle reduzieren lassen. Die Entstehung der abgebildeten Symboltabelle wird im folgenden schrittweise erläutert.

- Vor der ersten Deklaration sei der Name `T` noch unbekannt. Die Typdeklaration für `P1` führt vor dem Besuch des Typunterbaums im `TypeDecl`-Knoten zur Erzeugung eines unvollständigen Typeintrags für `P1` in der Symboltabelle.
- Durch die Attributauswertungsregeln des Knotentyps `PointerToIdType` wird mittels der Puma-Funktion `E.ApplyPointerBaseIdent` ein neuer Typeintrag α mit Namen `T` angelegt, da in der Symboltabelle bisher kein Eintrag für `T` vorhanden ist. Das Kind `typeRepr` von α verweist dabei auf einen neuangelegten `ForwardTypeRepr`-Knoten. Wie bei allen Zeigertypknoten wird die Typrepräsentation `PointerTypeRepr`, die zur Repräsentation des Basistyps von `P1` einen Verweis auf den Eintrag α besitzt, angelegt und über das Attribut `TypeReprOut` an `TypeDecl` zurückgegeben. Der Typeintrag `P1` wird mit dieser Typrepräsentation komplettiert.
- Die Typdeklaration für `P2` führt wiederum zur Erzeugung eines entsprechenden Typeintrags. In `E.ApplyPointerBaseIdent` wird nun jedoch α als zu verwendender Typeintrag der Zeigerbasis zurückgeliefert. Die zur Vervollständigung des Typeintrags `P2` notwendigen Schritte entsprechen den oben für `P1` beschriebenen.
- Die *eigentliche* Deklaration von `T` führt im `TypeDecl`-Knoten mittels der Puma-Funktion `E.DefineTypeEntry` zur gewünschten Vervollständigung der Typrepräsentation der Vorwärtszeiger. Auch diese Deklaration hat zur Folge, daß ein unvollständiger Typeintrag β mit Namen `T` angelegt wird. `E.DefineTypeEntry` bekommt neben dem Typeintrag β (Parameter `entry`) und der zu deklarierenden Typrepräsentation (Parameter `type`) den

bereits angelegten Typeintrag α (Parameter `forwardEntry`), dessen Typrepräsentation noch `ForwardTypeRepr` ist. Die Puma-Funktion setzt nun das `typeRepr`-Feld von Typeintrag α und Typeintrag β auf die deklarierte Typrepräsentation.

- Bei der Typdeklaration für `P3` wird somit als Typeintrag der Zeigerbasis der Typeintrag β verwendet.

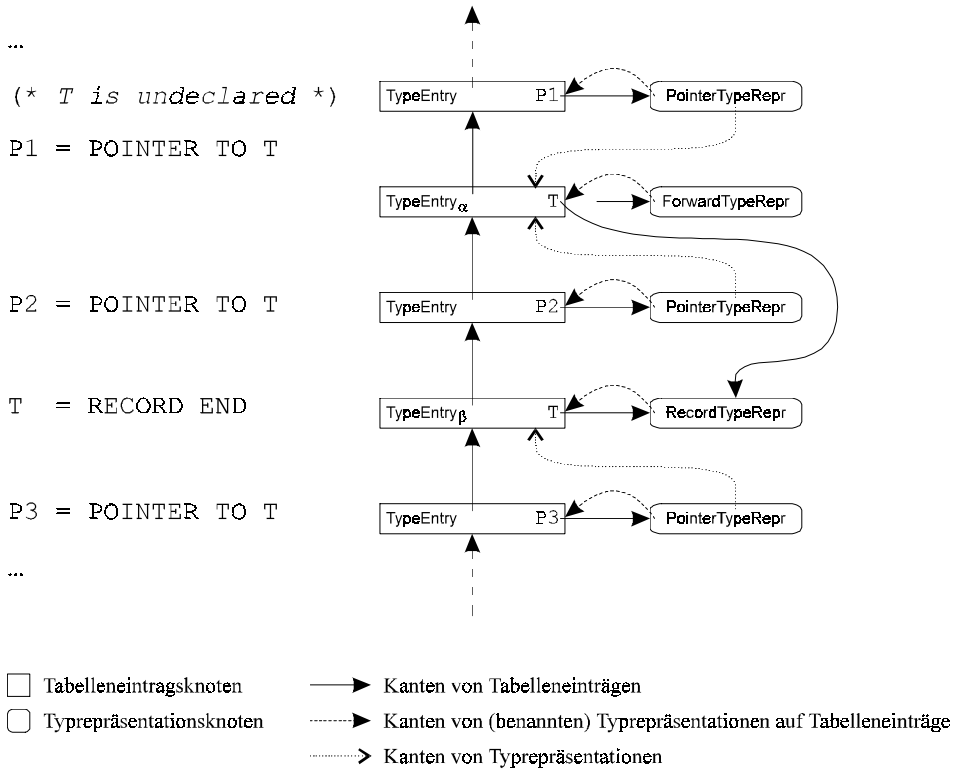

Abb. 20: Symboltabellenaufbau bei Vorwärtszeiger.

Dieses Beispiel motiviert auch die Entscheidung, die Repräsentation eines Zeigertyps durch eine Indirektion über einen Typeintrag zu modellieren. Dadurch kann selbst bei mehreren Vorwärtszeigern mit gleicher Zeigerbasis durch **eine** Zuweisung die korrekte Typrepräsentation realisiert werden. Aus Gründen der Vereinfachung der Attributauswertungsregeln im Knotentyp `TypeDecl` wird der Eintrag einer Vorwärtsdeklaration durch einen neu angelegten Typeintrag überdeckt und damit auf die weitere Nutzung des überdeckten Eintrags verzichtet.

Die Prüfung der Symboltabelle auf noch existierende ,,unaufgelöste“ Vorwärtsdeklarationen geschieht bei der Übergabe der Tabelle an die `Procs`-Knotenliste innerhalb des Knotentyps `DeclSection`.

3.2.5 Vorausdeklarationen

Eine Prozedurdeklaration kann durch eine Vorausdeklaration bereits vor ihrer eigentlichen Deklaration eingeführt werden, wodurch ein Bezug auf diese Prozedur vor der eigentlichen Deklaration ermöglich wird (vgl. 1.1.10 Prozedurdeklarationen, S.17).

Eine Vorausdeklaration führt durch Aktionen zum Knotentyp `ForwardDecl` zum Anlegen eines Prozedureintrags in der Symboltabelle, welcher durch Belegung des Attributs `complete` mit dem Wert `FALSE` als Vorausdeklaration kenntlich gemacht wird. Dieser Prozedureintrag (einschließlich des zugehörigen Prozedurtyps) wird auch nach der Behandlung der *eigentlichen* Deklaration zur Repräsentation der Prozedur weiter verwendet.

Die Zusammenhänge bei der Komplettierung einer Vorausdeklaration während der Behandlung einer Prozedurdeklaration durch die Aktionen zum Knotentyp `ProcDecl` skizziert Abbildung 21. Diese Abbildung zeigt den Knotentyp `ProcDecl` zusammen mit den relevanten Attributen und Operationen zur Ermittlung der Symboltabelleneinträge. Neben der Darstellung des Knotentyps sind Symboltabellen für drei verschiedene Deklarationsfragmente abgebildet. Der Abbildung können für jedes einzelne Deklarationsfragment die Werte der Attribute `TableIn`, `AlreadyDeclEntry`, `ForwardedEntry`, `TableTmp` und `FormalPars:TableIn` des Knotentyps `ProcDecl` entnommen werden.

Dem lokalen Attribut `AlreadyDeclEntry` wird ein (möglicherweise bereits vorhandener) Eintrag der Symboltabelle zugewiesen, der im selben Sichtbarkeitsbereich wie die Prozedurdeklaration liegt und den gleichen Namen besitzt. Der Wert des Attributs wird zur Überprüfung unerlaubter Mehrfachdeklarationen benötigt. Zusätzlich erhält das Attribut `ForwardedProcEntry` diesen Wert, falls es sich dabei um einen Prozedureintrag handelt, der durch eine Vorausdeklaration angelegt wurde. Das bedeutet insbesondere, daß dieses Attribut entscheidend dafür ist, welches Aussehen die Symboltabelle nach der Deklaration besitzt: Existiert keine entsprechende Vorausdeklaration, muß in der Symboltabelle ein neuer Prozedureintrag angelegt werden; Andernfalls muß der bereits durch die Vorausdeklaration angelegte Prozedureintrag verwendet werden. Dazu realisiert die Puma-Funktion `E.ChooseProcedureEntry` anhand des Werts von `ForwardedProcEntry` die Auswahl zwischen der bisherigen Symboltabelle (`TableIn`) und der Symboltabelle, in der ein neuer Prozedureintrag angelegt wurde. Die zu verwendende Symboltabelle wird dem lokalen Attribut `TableTmp` zugewiesen.

Wie bei allen Deklarationen üblich, wird der anfangs unvollständige Eintrag beim Abschluß der Deklaration komplettiert. Die Puma-Funktion `E.DefineProcedureEntry` unterscheidet hierzu zwischen dem Fehlen und dem Vorliegen einer Vorausdeklaration. Beim Fehlen einer Vorausdeklaration wird der Prozedureintrag aus `TableTmp` um den Prozedurtyp ergänzt. Demgegenüber muß beim Vorliegen einer Vorausdeklaration der neuangelegte Prozedureintrag, auf den `ForwardedProcEntry` verweist, komplettiert werden. Seine endgültige Exportmarkierung ist durch die Funktion `E.MaxExportMode` aus den Exportmarkierungen der Vorausdeklaration und der eigentlichen Deklaration zu ermitteln. `E.DefineProcedureEntry` liefert den Wert des Parameters `table` unverändert als Resultat. Diese Identitätsfunktionalität dient lediglich der Festlegung des Zeitpunkts, zu dem der Seiteneffekt der Vervollständigung des Eintrags erfolgen soll: Eine vollständige Prozedurdeklaration ist erst nach dem Besuch des Kinds `FormalPars` zu erstellen und wird für den Besuch des Kinds `DeclSection` benötigt.

Die dem aktuellen `ProcDecl`-Knoten folgende Prozedurdeklaration erhält schließlich die Symboltabelle, auf die `TableTmp` verweist. Diese Symboltabelle enthält den durch den Seiteneffekt von `DefineProcedureEntry` vervollständigten Prozedureintrag.

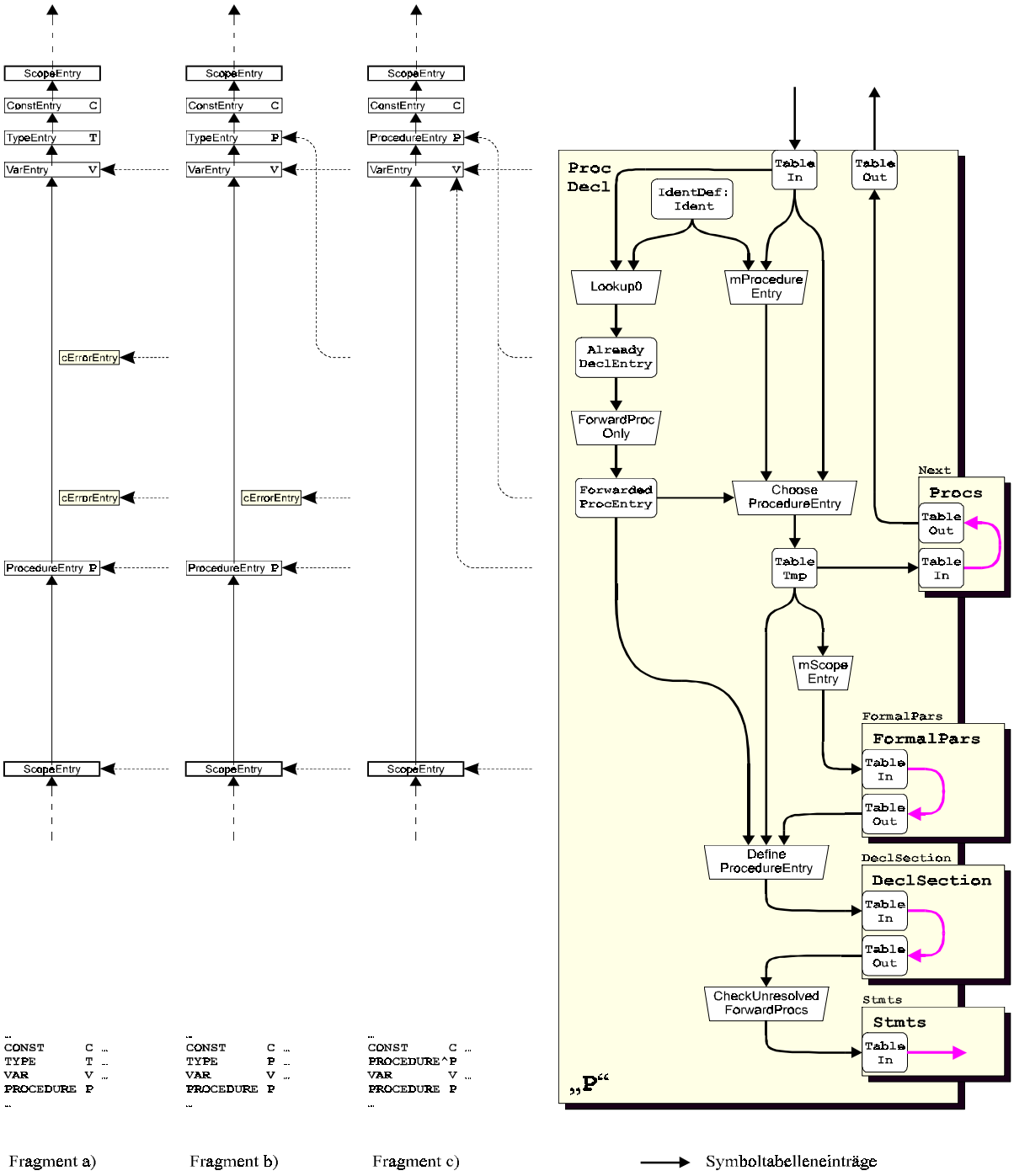

Abb. 21: Datenfluß bei einer Prozedurdeklaration.

3.2.6 Typerweiterungen

Einer der zentralen Mechanismen von Oberon-2 ist die Erweiterung von Record-Typen. Ein erweiterter Record-Typ umfaßt alle Felder seines Basistyps. Der Modellierung der Typerweiterung dienen im Knotentyp `RecordTypeRepr` das Attribut `extLevel` sowie die Kinder `baseTypeRepr` und `extTypeReprList`.

Das Attribut `extLevel` enthält die Erweiterungsstufe eines Record-Typs. Es wird durch das erstellte Front-End lediglich verwaltet, und wird erst möglicherweise bei der Codeerzeugung benötigt (vgl. Typinformationen zur Laufzeit, S.30). Ein Record-Typ *T*, dessen Basistyp *T* ist (d.h. *T* ist keine Erweiterung), besitzt die Erweiterungsstufe `ROOTEXTLEVEL` (=`0`).

In der Typrepräsentation eines Record-Typs *T* dient das Kind `baseTypeRepr` als Verweis auf die Repräsentation des direkten Basistyps von *T*. Ist *T* keine Erweiterung, enthält `baseTypeRepr` den Wert `cmtObject`.

Für die im nachfolgenden beschriebene Handhabung von typgebundenen Prozeduren wird das Kind `extTypeReprList` benötigt. Es enthält in der Typrepräsentation eines Record-Typs *T* eine Liste von Verweisen auf die Typrepräsentationen aller direkten Erweiterungen von *T*.

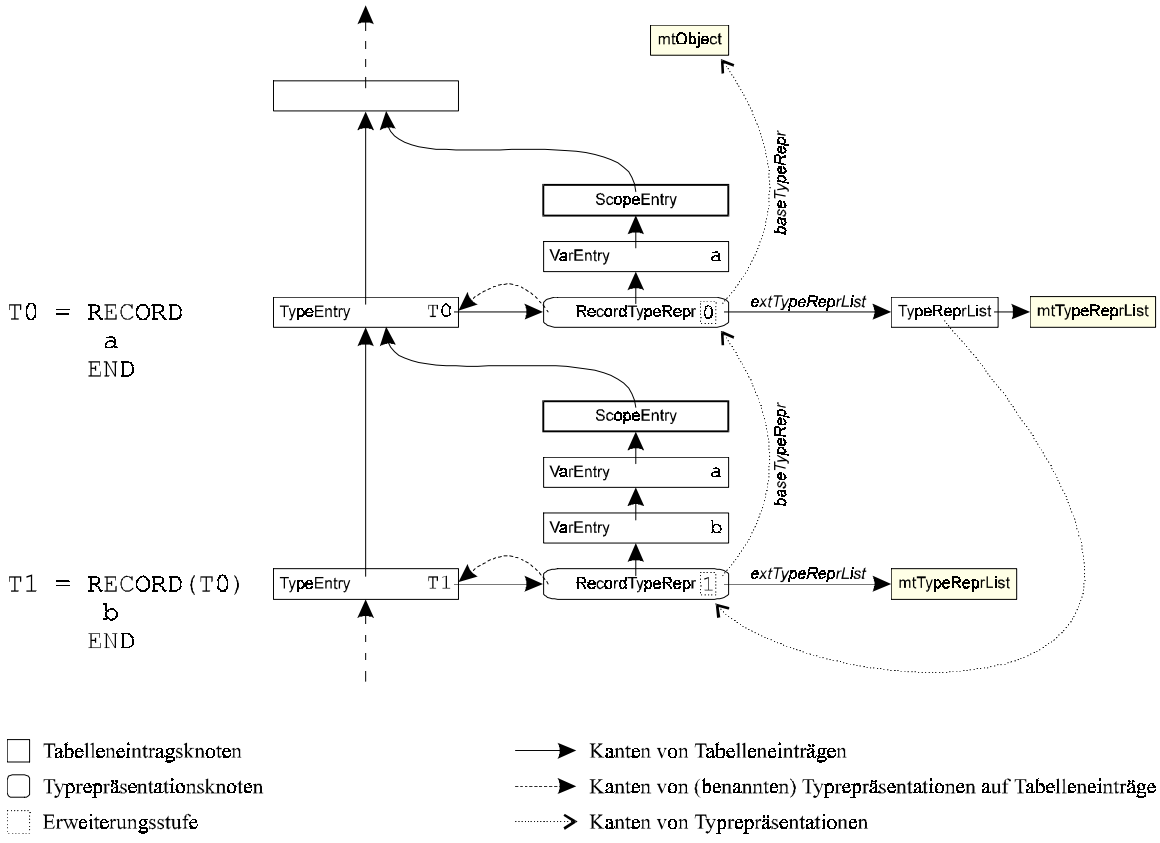

Abb. 22: Repräsentation eines erweiterten Record-Typs und seines Basistyps.

Abbildung 22 skizziert für die beiden Typdeklarationen

```
T0 = RECORD     a: … END;
T1 = RECORD(T0) b: … END;
```

den Einsatz dieser Knotentypfelder.

Die Erzeugung der Repräsentation einer Typerweiterung durch die Attributauswertungsregeln des Knotentyps `ExtendedType` stützt sich auf die Puma-Funktionen `T.CloneRecord` und `T.AppendExtension`.

Bevor die Symboltabelle zum Aufbau der Record-Felder dem `FieldLists`-Knoten übergeben wird, erfolgt durch `T.CloneRecord` eine Duplizierung der Record-Felder des Basistyps. `T.CloneRecord` benutzt hierzu die lokal zum Modul `T` spezifizierte Puma-Funktion `T.DupAllFields`, die rekursiv die Einträge aller Record-Felder bis zum nächsten `ScopeEntry` kopiert. Diese Duplizierung, als Alternative zu einer Mehrfachnutzung der Basistypfelder, erleichtert die im nächsten Abschnitt beschriebene Behandlung von typgebundenen Prozeduren wie Record-Felder.

Die Herstellung des Verweises auf den Basistyp sowie das Eintragen der entsprechenden Erweiterungsstufe erfolgt direkt beim Aufruf von `T.CompleteRecordTypeRepr` zur Vervollständigung des zu deklarierenden erweiterten Record-Typs. Demgegenüber wird die Aktualisierung der Erweiterungsliste `extTypeReprList` des Basistyps durch die Puma-Prozedur `T.AppendExtension` vorgenommen.

Neben der Deklaration von erweiterten (Record-)Typen sind die Stellen von Interesse, an denen der Typerweiterungsmechanismus zur Anwendung gelangt. Während der Typtest durch die gewählte Modellierung von Typerweiterungen einfach ist, erfordert die Behandlung der Typen bei With-Anweisungen einen etwas größeren Aufwand.

Für eine With-Anweisung der Form

```
WITH v: T DO S END
```

liefert die Puma-Funktion `E.EntryWithed` die für *S* geltende Symboltabelle, in der zusätzlich ein neuer Eintrag für die Variable *v* existiert. Dieser Eintrag besitzt im Feld `typeRepr` die Typrepräsentation von *T*. Für den Fall, daß *v* nicht im aktuellen Modul deklariert ist, wird in der Symboltabelle ein neuer `ServerEntry` angelegt, dessen Symboltabelle um einen entsprechenden Variableneintrag ergänzt ist. Die Abbildungen 23 und 24 skizzieren jeweils die veränderte Symboltabelle.

Ähnlich der With-Anweisung hat auch die Typzusicherung den Effekt, daß der statische Typ einer Variablen (oder eines Parameters) innerhalb eines lokalen Bereichs geändert wird. Eine Typzusicherung der Form *v*(*T*) hat zur Folge, daß für den Selektor, der dem Bezeichner *v*(*T*) unmittelbar folgt, *v* so betrachtet wird, als ob *v* vom statischen Typ *T* wäre. Dies wird realisiert, indem im `Guarding`-Knoten, der die Typzusicherung repräsentiert, dem nachfolgenden `Designations`-Knoten die Typrepräsentation von *T* über das Attribut `TypeReprIn` übergeben wird.

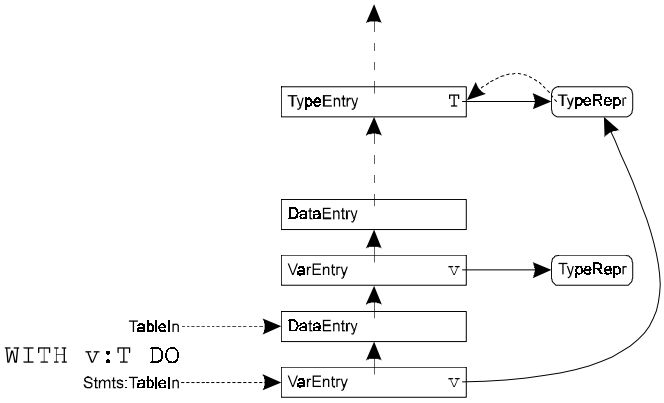

Abb. 23: Veränderte Symboltabelle bei With-Anweisung mit lokaler Variable.

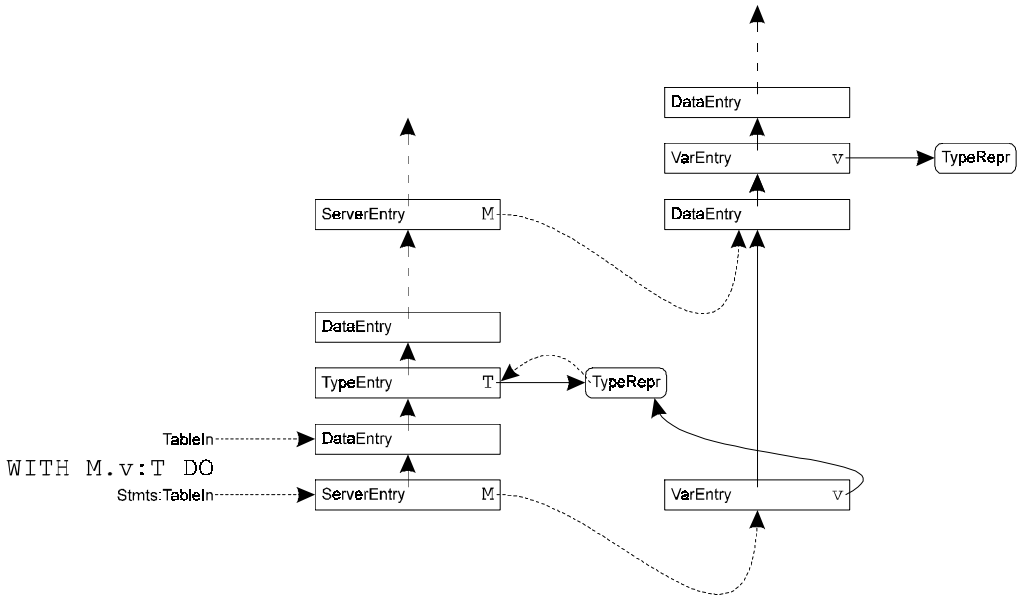

Abb. 24: Veränderte Symboltabelle bei With-Anweisung mit externer Variable.

3.2.7 Typgebundene Prozeduren

Bei der Betrachtung der Erweiterung von Record-Typen in Oberon-2 spielen die typgebundenen Prozeduren, als zentrale Neuerung gegenüber Oberon, eine große Rolle. Ihre Implementierung ist mit einem erheblichen Aufwand verbunden.

Typgebundene Prozeduren werden wie Record-Felder angesehen und sind auch wie solche implementiert. Dies bedeutet, daß Deklarationen von Prozeduren, die an einen Typ *T* gebunden sind, durch Einträge vom Knotentyp `BoundProcEntry` innerhalb der Feldliste von *T* repräsentiert werden. Der Empfängerparameter wird durch eine einelementige Signatur als Kind (`receiverSig`) repräsentiert. Für Prozeduren, die durch Erweiterung eines Record-Typs *geerbt* werden, wird der Knotentyp `InheritedProcEntry` zur Repräsentation innerhalb der Feldliste des erweiterten Typs eingesetzt. Dieser Knotentyp besitzt das Kind `boundProcEntry`, welches auf die Repräsentation der *originär* gebundenen Prozedur verweist. Die bereits im vorhergehenden Abschnitt erwähnte Duplizierung der Record-Typrepräsentation bei einer Typerweiterung durch die Puma-Funktion `T.CloneRecord` führt zusätzlich die Umwandlung von `BoundProcEntry`-Knoten der Feldliste des Basistyps zu `InheritedProcEntry`-Knoten in der Feldliste des erweiterten Typs durch.

Überprüfung der Kontextbedingungen

Bei der Bindung einer Prozedur mit Namen *P* an einen Record-Typ *T* durch eine entsprechende (Voraus-) Deklaration sind unter anderem folgende Fälle zu beachten: In *T* existiert

- ☺ kein Feld und keine gebundene Prozedur mit Namen *P*,
- ☹ ein Feld mit Namen *P*,
- 😐 eine vorausdeklarierte gebundene Prozedur mit Namen *P*,
- ☹ eine originär gebundene Prozedur mit Namen *P*,
- ☺ eine geerbt gebundene Prozedur mit Namen *P*.

Zusätzlich ist zu beachten, daß durch die Bindung von *P* an *T* eine bereits an eine Erweiterung von *T* originär gebundene Prozedur *P′* gleichen Namens zu einer Redefinition von *P* wird. *P′* wird im folgenden als *Preredefinition* bezeichnet.

Zur Behandlung der Bindung einer Prozedur *P* an einen Typ *T* dient innerhalb der Attributauswertungsregeln zum Knotentyp `BoundProcDecl` der Aufruf der Puma-Funktion `T.CheckBoundProc` der Legalitätsprüfung der aufgezählten Fälle. Diese Funktion liefert eine entsprechende Fehlermeldung bzw. `NoErrorMsg`. Ihre Argumente sind u.a. das Ergebnis der Suche eines Eintrags *e* mit Namen *P* innerhalb der Record-Feldliste von *T*, die bereits angelegte Typrepräsentation von *P* sowie die Typrepräsentation von *T*. Anhand des Knotentyps von *e* und dem Inhalt seiner Knotentypfelder können in `T.CheckBoundProc` diese Fälle vollständig unterschieden werden. Insbesondere wird für den Fall, daß in *T* kein Feld und keine gebundene Prozedur mit Namen *P* existiert, die Überprüfung der Preredefinitionen mittels der Puma-Funktion `T.CheckPreRedefinitions` durchgeführt.

`T.CheckPreRedefinitions` dient lediglich der Extraktion des Kinds `extTypeReprList` (Liste der direkten Erweiterungen von *T*) aus der Typrepräsentation von *T*. Über diese Listen ist die gesamte Erweiterungshierarchie von *T* als Baum definiert. Dieser Baum wird durch die Puma-Funktion `T.CheckTypeReprListsPreRedefs` traversiert. Jede Feldliste der Record-Typen dieser Erweiterungshierarchie wird dabei durch Aufruf von `T.CheckFieldsPreRedefs` auf eventuelle Preredefinitionen durchsucht, und jede gefun-

dene Preredefinition wird auf Übereinstimmung der formalen Parameter mit denen von *P* überprüft.

Aktualisierung der Repräsentation

Der Vorgang der Bindung einer Prozedur *P* an einen Typ *T* erfordert eine Aktualisierung der beteiligten Typrepräsentation(en).

Diese Aktualisierung leistet die Puma-Funktion `T.BindProcedureToRecord`, die beim Aufruf u.a. die Typrepräsentation von *T*, den neu erzeugten Eintrag zur Repräsentation von *P* sowie den (möglicherweise) bereits existierenden Feldeintrag in *T* mit gleichem Namen wie *P* erhält. `T.BindProcedureToRecord` unterscheidet hierfür lediglich zwischen der Komplettierung einer durch Vorausdeklaration bereits gebundenen Prozedur und der Bindung durch eine eigentliche Prozedurdeklaration. Im letzteren Fall wird aus der Feldliste von *T* über die Puma-Funktion `T.DeleteBoundProc` ein möglicherweise vorhandener Eintrag für eine geerbte Prozedur mit gleichem Namen wie *P* entfernt und der übergebene Prozedureintrag für *P* angehängt.

Schließlich muß noch dafür gesorgt werden, daß *P* auch an alle Erweiterungen von *T* (geerbt) gebunden wird. Dies leistet die Puma-Funktion `T.BindProcedureToExtensions`. Sie erhält unter anderem die Liste der Erweiterungen von *T* und die zu bindende Prozedur *P*. `T.BindProcedureToExtensions` dient lediglich der Rekursion über die Elemente *T′* dieser Liste. Für jedes *T′* erfolgt die Bindung von *P* durch erneuten Aufruf der Prozedur `T.BindProcedureToRecord`. Neben *T′* wird jedoch nicht die zu bindende Prozedur *P* übergeben, sondern ein neu angelegter `InheritedProcEntry`-Knoten, der einen Verweis auf *P* enthält.

Nur für diesen verschränkt rekursiven Aufruf existiert in `T.BindProcedureToRecord` ein dritter Fall. Falls an den zu bindenden Record-Typ *T′* nicht bereits eine Prozedur *P* originär gebunden war, führt `T.BindProcedureToRecord` hier die Bindung von *P* an *T′* durch und ruft `T.BindProcedureToExtensions` zur Bindung von *P* an alle Erweiterungen von *T′* erneut auf.

Abbildung 25 zeigt den Datenfluß bei der Evaluation der Attribute des Knotentyps `BoundProcDecl`, die für eine Deklaration einer typgebunden Prozedur relevant sind.

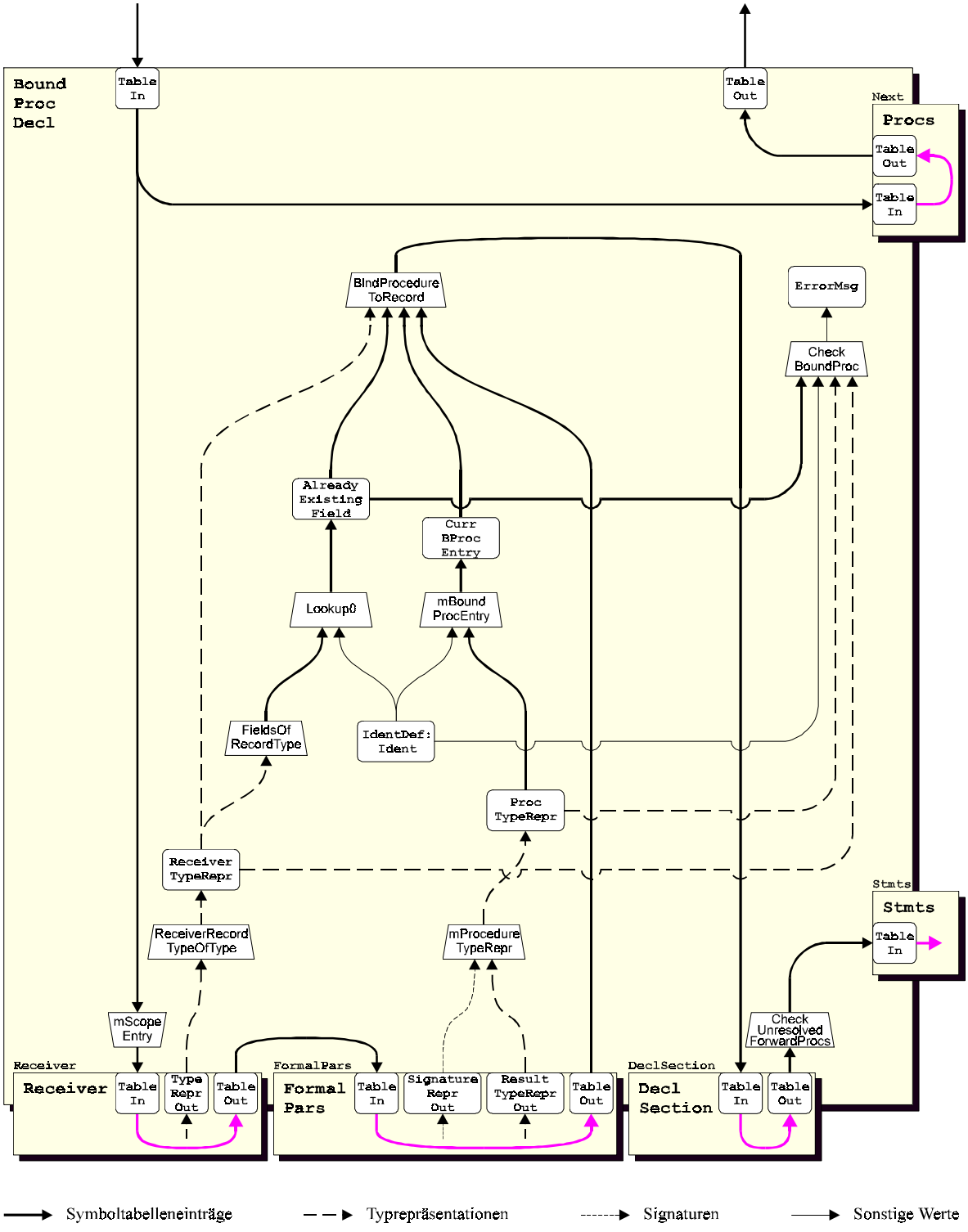

Abb. 25: Datenfluß bei der Deklaration einer gebundenen Prozedur.

3.2.8 Ausdrücke

Die Vorränge der (zweistelligen) Operatoren innerhalb von Ausdrücken sind in der Sprachbeschreibung explizit durch die entsprechende Formulierung der Regeln der kontextfreien Grammatik festgelegt. Demgegenüber sind im abstrakten Syntaxbaum die Vorränge nur noch implizit durch die Art der Verknüpfung der relevanten Knoten gegeben. Die hierfür verwendeten Knotentypen sind alle Erweiterungen des Knotentyps `Exprs`. Für die Repräsentation von zweistelligen Operatoren wird jeweils ein eigener Knoten eines Knotentyps der Erweiterungshierarchie, die in Abbildung 26 dargestellt ist, eingesetzt.

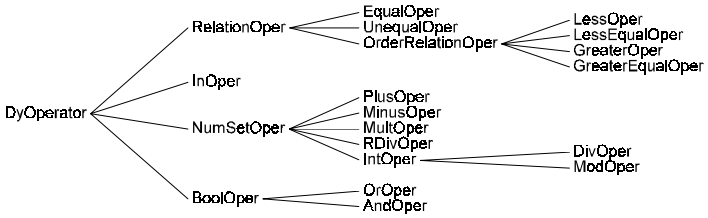

Abb. 26: Erweiterungshierarchie der Knotentypen für zweistellige Operatoren.

In den Attributauswertungsregeln für Ausdrücke ist die Typermittlung und -überprüfung sowie die Berechnung von Werten im Rahmen der Konstantenauswertung definiert. Diese Attributauswertungsregeln sind alle bei den jeweiligen Erweiterungen des Knotentyps `Exprs` angegeben. Lediglich die für Ausdrücke mit zwei Operanden geltenden Attributauswertungsregeln sind auf die Knotentypen aus der Erweiterungshierarchie von `DyOperator` verteilt. Aus diesem Grunde besitzt der Knotentyp `DyOperator` Attribute zur Übergabe der Typ- und Wertrepräsentationen beider Unterausdrücke und des Gesamtausdrucks vom bzw. zum Elterknoten vom Knotentyp `DyExpr`. Die Erweiterungshierarchie der `DyOperator`-Knoten ist so angelegt, daß die Attributauswertungsregeln nur jeweils einmal für eine ganze Klasse von Operatoren angegeben werden müssen.

Die für Ausdrücke relevanten Attributauswertungsregeln stützen sich im wesentlichen auf Puma-Funktionen aus dem Modul `T`, zur Ermittlung der neuen Typrepräsentation und zur Typüberprüfung, sowie aus dem Modul `V`, für die Konstantenauswertung zur Übersetzungszeit. Die Berechnung von Werten im Rahmen dieser Konstantenauswertung erfolgt dreistufig:

- Die Attributauswertungsregeln benutzen Puma-Funktionen aus dem Modul `V`, denen sie die zur Berechnung notwendigen Wertrepräsentationen und die Operatorkodierung übergeben. Die Puma-Funktionen liefern eine entsprechende Wertrepräsentation des Ergebnisses oder `ErrorValue`, falls ein Berechnungsfehler aufgetreten ist (z.B. Division durch 0).
- Die Puma-Funktionen zur Wertberechnung wählen in Abhängigkeit des konkreten Knotentyps der Wertrepräsentationen die entsprechende Berechnungsprozedur aus dem Modul `OT` aus, selektieren die Binärwerte aus den Wertrepräsentationsknoten und übergeben diese zusammen mit der Operatorkodierung an die Berechnungsprozedur. Das binär repräsentierte Ergebnis der Berechnungsprozedur wird dann in einen neu angelegten Wertrepräsentationsknoten ,,verpackt“ und zurückgegeben.
- In den Berechnungsprozeduren im Modul `OT` erfolgt die Durchführung der eigentlichen Wertberechnung in Abhängigkeit der übergebenen Operatorkodierung.

Abbildung 27 skizziert die in diesem Zusammenhang relevanten Attributauswertungen für den Ausdruck `1+2`. In der Abbildung sind zusätzlich die Attribute und die Puma-Funktionen zur Typanpassung mit dargestellt.

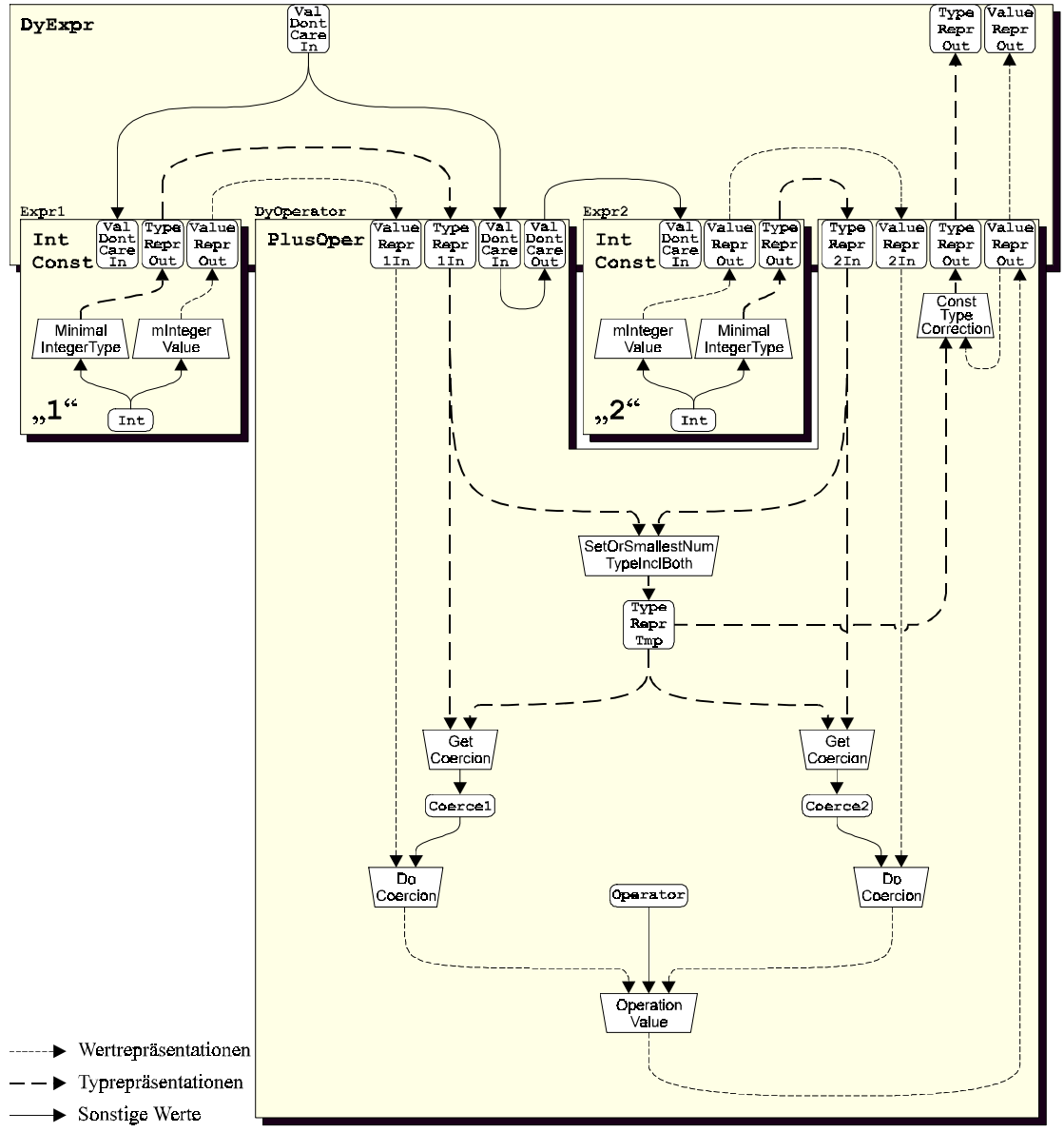

Abb. 27: Datenfluß bei einem zweistelligen Ausdruck.

Künstliche Attributabhängigkeit

Im Knotentyp `DyOperator` ist eine künstliche Attributabhängigkeit definiert, die dem Werkzeug Ast vorschreibt, daß das Attribut `ValDontCareOut` vor den Attributen `TypeRepr2In` und `ValueRepr2In` berechnet werden soll. Ohne die Angabe dieser Abhängigkeit werden vom Werkzeug Ast zwei ,,Zyklen“ im Attributabhängigkeitsgraph gemeldet, die zur Folge haben, daß kein OATG-Evaluator aus der Spezifikation konstruiert werden kann. Diese Zyklen resultieren aus der notwendigen Einfügung von Abhängigkeiten zwischen Attributen eines Symbols der zugrundeliegenden Grammatik, die das Werkzeug bei der Konstruk-

tion der Besuchsfolge für dieses Symbol erstellt. Beim Einfügen solcher Abhängigkeiten zur Erlangung einer Ordnung über den Attributen kann es passieren, daß Ag ,,schlecht rät“ und Abhängigkeiten einfügt, die tatsächlich in keinem Syntaxbaum auftreten. So würde Ag ohne die oben erwähnte künstliche Attributabhängigkeit das Attribut `ValDontCareOut` von den Attributen `TypeRepr2In` und `ValueRepr2In` abhängig machen. Durch diese (und weitere) eingefügten Abhängigkeiten ergeben sich zusammen mit den durch die Attributauswertungsregeln definierten Abhängigkeiten zwei Zyklen. Tatsächlich hängt das Attribut `ValDontCareOut` lediglich vom Wert des ersten Ausdrucks ab. Es entscheidet bei den konkreten Knotentypen `OrOper` und `AndOper`, zusammen mit dem Attribut `ValDontCareIn`, ob eine Kurzschlußauswertung vorgenommen werden könnte. Somit ist vor dem Besuch des Knotens für den zweiten Ausdruck und damit auch vor der Auswertung der Attribute `TypeRepr2In` und `ValueRepr2In`, die ebenfalls von Attributen des Knotens des zweiten Ausdrucks abhängen, das Attribut `ValDontCareOut` berechenbar.

3.2.9 Bezeichner

Die Behandlung von Bezeichnern stellt innerhalb der Spezifikation den komplexesten Einsatz der benutzten Werkzeuge dar, da nur hier der durch den Evaluator traversierte abstrakte Syntaxbaum während der Evaluation mittels Puma-Funktionen modifiziert wird.

Durch die bereits beschriebene Mehrdeutigkeit der Grammatik für Bezeichner (einschließlich des Prozeduraufrufs) kann ohne Zuhilfenahme von Kontextinformation bei der Parsierung kein adäquater abstrakter Syntaxbaum aufgebaut werden (vgl. S.81). Dies trifft insbesondere auf das Einfügen von Knoten zur Repräsentation einer etwaigen (impliziten) Dereferenzierung vor Array- und Record-Selektoren zu.

Zur Lösung dieser Probleme wird bei der Parsierung eines Bezeichners eine Liste von `Designor`-Knoten, die lediglich die syntaktische Gestalt der existierenden Selektoren repräsentieren, erzeugt und im `Designator`-Knoten gehalten (siehe 3.1.3 Abstrakter Syntaxbaum: Selektoren, S.76).

Diese `Designor`-Liste wird durch den Evaluator in eine Liste von `Designations`-Knoten umgewandelt, die dann die eigentlichen Selektoren repräsentieren. Diese Umwandlung geschieht schrittweise vor dem Besuch des jeweiligen Knotens. Abbildung 28 skizziert hierzu eine Folge von Knotenbesuchen für einen Bezeichner mit zwei Selektoren α und β (beispielsweise `a.b(c)`). Dargestellt ist jeweils das Aussehen des abstrakten Syntaxbaums zum Zeitpunkt des Besuchs des hervorgehobenen Knotens, vor Anwendung seiner Attributauswertungsregeln.

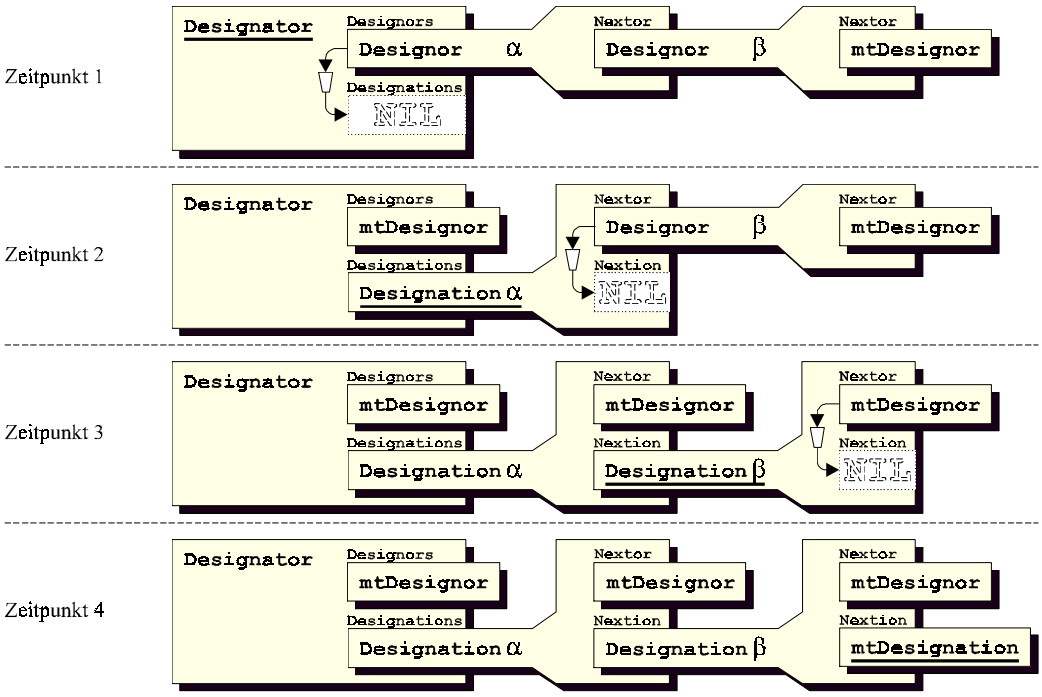

Abb. 28: Transformation der Selektorknoten eines Bezeichners.

Beim Besuch des `Designator`-Knotens wird der erste `Designations`-Knoten über die Puma-Funktion `TT.DesignorToDesignation` aus dem ersten Element der `Designor`-Liste erzeugt (siehe 3.1.7 Baumtransformation: Umwandlung von Selektoren, S.96).

Die Erzeugung des nächsten `Designations`-Knotens durch die Transformationsfunktion stützt sich – neben dem zu transformierenden `Designor`-Knoten – auf den bis zum aktuellen `Designations`-Knoten ermittelten Symboltabelleneintrag und die bis hierhin ermittelte Typrepräsentation. Dieser `Designations`-Knoten repräsentiert den nächsten eigentlichen Selektor und verweist über das Kind `Nextor` auf die noch zu bearbeitende Restliste der `Designor`-Knoten. Sein Kind `Nextion` hat nach der Umwandlung noch den Wert `NIL`. Das Kind `Nextor` des aktuellen `Designations`-Knotens, welches vor der Umwandlung auf die `Designor`-Liste verwies, besitzt nach der Umwandlung einen Verweis auf einen Knoten vom Typ `mtDesignor`. Nun kann der gerade erzeugte `Designations`-Knoten besucht werden, wo nach der gleichen Methode die Umwandlung der restlichen `Designor`-Knoten stattfindet. Enthält das Kind `Nextor` eines `Designations`-Knotens den Wert `mtDesignor`, welcher das Ende der `Designor`-Liste anzeigt, wird ein Knoten vom Knotentyp `mtDesignation` erzeugt. Mit dessen Besuch ist der Syntaxbaum für den Bezeichner komplett umgewandelt.

Bei dieser schrittweisen Umwandlung der Selektoren ist jedoch zu beachten, daß es auch Fälle gibt, bei denen die `Designor`-Liste durch einen Aufruf der Transformationsfunktion nicht verkürzt wird. Falls die bis zum aktuellen `Designations`-Knoten ermittelte Typrepräsentation für einen Zeigertyp steht und der zu transformierende `Designor`-Knoten vom Knotentyp `Selector` oder `Indexor` ist, wird lediglich ein `Dereferencing`-Knoten, der die implizite Dereferenzierung repräsentiert, samt unveränderter `Designor`-Liste zurückgeliefert. In ähnlicher Weise bleibt die `Designor`-Liste fast unverändert, falls ihr Kopf aus einem `Indexor`-Knoten besteht, dessen Kind `ExprList` auf eine mindestens zweielementige Ausdrucksliste verweist. Die Transformationsfunktion liefert hier einen `Indexing`-Knoten, der nur den ersten Ausdruck der Ausdrucksliste besitzt. Die Ausdrucksliste im `Indexor`-Kopf der `Designor`-Liste ist aber um diesen Ausdruck verkürzt.

3.2.10 Vordeklarierte Prozeduren

Der Begriff ,,vordeklarierte Prozeduren“ schließt im folgenden auch die im Modul `SYSTEM` deklarierten Prozeduren ein. Da einige der vordeklarierten Prozeduren generisch sind, ist der Mechanismus zur Repräsentation von benutzerdeklarierten Prozeduren für sie nicht ausreichend. Da zudem die echt vordeklarierten Prozeduren bereits zur Übersetzungszeit bei Vorliegen von konstanten Argumenten auszuwerten sind, ist eine gesonderte Behandlung naheliegend.

Parsierung

Für die Parsierung sind die vordeklarierten Prozeduren noch ohne Bedeutung. Der Einsatz der neu eingeführten ,,vordeklarierten“ Prozeduren `_CASEFAULT` bzw. `_WITHFAULT` bei fehlendem Else-Zweig (vgl. S.84) bildet hiervon eine Ausnahme.

Symboltabelle

Eine vordeklarierte Prozedur wird in der Symboltabelle als Prozedureintrag gehalten, dessen Prozedurtyp als Erweiterung des Knotentyps `PreDeclProcTypeRepr` modelliert ist. Dieser Knotentyp ist wiederum eine Erweiterung von `ProcedureTypeRepr`. Für jede vordeklarierte Prozedur existiert genau ein Knotentyp zur Repräsentation des Prozedurtyp. Das Kind `signatureRepr` besitzt bei allen vordeklarierten Prozeduren einen Verweis auf einen Knoten vom Knotentyp `GenericSignature`. Das Kind `resultType` verweist immer auf einen Knoten vom Knotentyp `mtObject`.

Der Aufbau der Symboltabellen findet durch Aufruf von `PR.GetTablePREDECL` bzw. `PR.GetTableSYSTEM` statt. In den Attributauswertungsregeln des Knotentyps `Module` wird als initiale Symboltabelle eines Moduls das Ergebnis von `PR.GetTablePREDECL` an den Knotentyp `Imports` weitergegeben. Demgegenüber wird `PR.GetTableSYSTEM` innerhalb der Prozedur `DRV.Import` aus dem Modula-2 Modul `DRV` aufgerufen, falls das durch diese Prozedur zu importierende Modul den Namen `"SYSTEM"` besitzt. Die Prozedur `DRV.Import` wird in den Attributauswertungsregeln des Knotentyps `Import` benutzt, um die Symboltabelle eines zu importierenden Moduls zu ermitteln.

Abstrakter Syntaxbaum

Die bei der Applikation einer vordeklarierten Prozedur im Knotentyp `Designator` ermittelte Typrepräsentation (eine Erweiterung von `PreDeclProcTypeRepr`) wird an die bereits beschriebene Puma-Funktion `TT.DesignorToDesignation` übergeben. Diese Typrepräsentation dient dort der Erzeugung eines zur jeweiligen vordeklarierten Prozedur spezifischen Knotens, dessen Knotentyp eine Erweiterung des Knotentyps `PredeclArgumenting` ist. `PredeclArgumenting` ist wiederum eine Erweiterung des Knotentyps `Designation`. Die in Abbildung 29 dargestellte Erweiterungshierarchie strukturiert die benötigten Knotentypen zur Repräsentation der aktuellen Parameter von vordeklarierten Prozeduren bezüglich der Gestalt der durch die Sprachdefinition gegebenen Signatur.

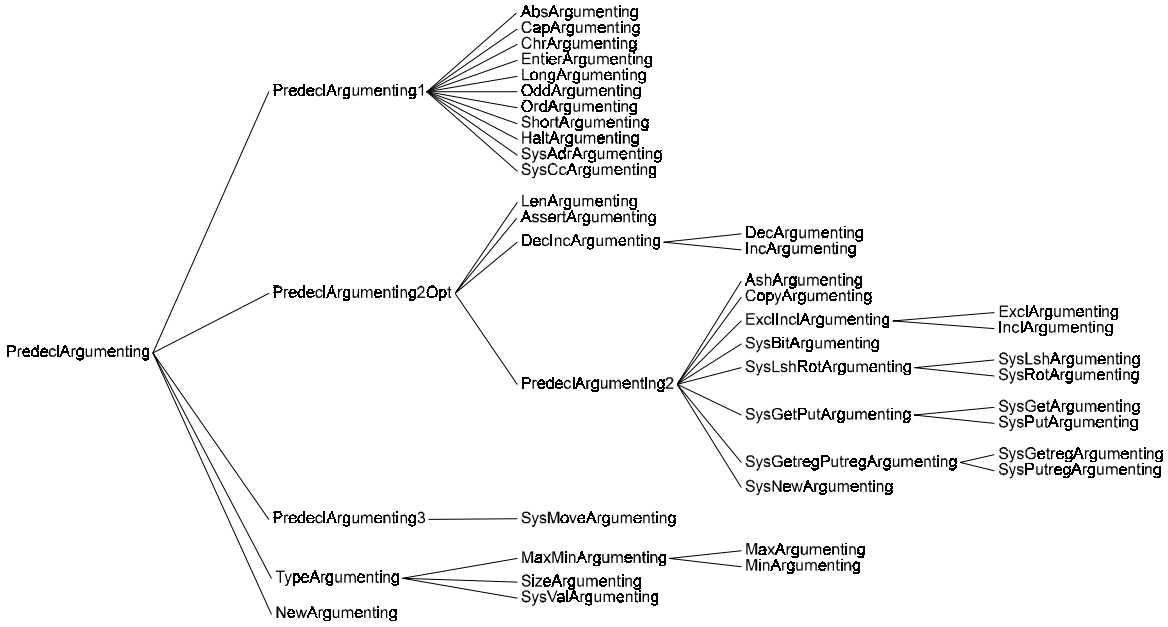

Abb. 29: Erweiterungshierarchie der Knotentypen für vordeklarierte Prozeduren.

Die Transformation durch `TT.DesignorToDesignation` bedient sich der vom Modul `TT` nicht exportierten Puma-Funktionen, deren Namen alle das Präfix `ExtractExpr` besitzen. Beim Aufruf einer solchen Puma-Funktion wird die zu transformierende `Designor`-Liste und der entsprechende, durch einen parameterlosen Knotenkonstruktor (!) neu erzeugte `PredeclArgumenting`-Knoten übergeben. Sie liefert diesen neuen Knoten, nachdem dessen Knotenfelder aus den Werten der Knotenfelder des ersten Elements der `Designor`-Liste ermittelt wurden. Die Funktionen `ExtractExpr*` extrahieren aus der Ausdrucksliste des `Designor`-Listenkopfs jeweils die Anzahl von Ausdrücken, die der Anzahl der formalen Parameter der jeweiligen vordeklarierten Prozedur entsprechen. Zusätzlich wird die Restliste der Ausdrücke im neu erzeugten `PredeclArgumenting`-Knoten gehalten.

Die Funktionen `ExtractExprType` und `ExtractExprVal` führen diese Extrahierung nur unter der Bedingung durch, daß der zu extrahierende Ausdruck die Form eines (qualifizierten) Namens hat, indem sie sich auf die Funktion `ExtractFirstQualident` stützen. Für den Fall, daß besagter Ausdruck nicht diese Form besitzt, wird das entsprechende Knotenfeld mit `ErrorQualident` belegt.

Für die Behandlung der aktuellen Parameter der vordeklarierten Prozedur `NEW` wandelt die Funktion `ExtractExprNew`, nach Extraktion des ersten Ausdrucks, die Restliste der Knoten vom Knotentyp `ExprLists` in eine Liste von Knoten des Knotentyps `NewExprLists` um. Diesen Knotentypen sind spezifische Attributauswertungsregeln zugeordnet, über welche die spezielle Semantik der Parameter von `NEW` implementiert ist. Hierzu zählt insbesondere die Handhabung von übergebenen Array-Längen bei der Speicheranforderung für ein mehrdimensionales, offenes Array.

Evaluation

Die durch diese Transformation geleistete „Vorarbeit“ ermöglicht nun die Formulierung von Attributauswertungsregeln zur Behandlung der aktuellen Parameter von vordeklarierten Pro-

zeduren auf einfachste Weise. Die hierfür angegebenen Attributauswertungsregeln befinden sich alle im Evaluatormodul `Predeclareds` (`oberon.pre`).

Die vom Knotentyp `PredeclArgumenting` ausgehende Erweiterungshierarchie gestattet die Überprüfung der Anzahl der aktuellen Parameter durch Angabe der notwendigen Attributauswertungsregeln bei den abstrakten Knotentypen für jeweils eine ganze Klasse von vordeklarierten Prozeduren. Die Attributauswertungsregeln zur Überprüfung der für die aktuellen Parameter geltenden Kontextbedingungen sowie derjenigen zur Berechnung von Werten im Rahmen der Konstantenauswertung sind bei den konkreten Knotentypen angegeben.

3.3 Testumgebung

Zur Überprüfung der Korrektheit des spezifizierten Front-Ends wurde die Methode des Testens angewandt. Da diese Methode lediglich die Korrektheit bezüglich einer endlichen Teilmenge der Eingabemenge des zu testenden Programms zeigt, wurde versucht diese Teilmenge ausreichend mächtig zu gestalten und zusätzlich ein repräsentatives Spektrum der Eingabemenge zu schaffen. Für die erstellten Testprogramme bedeutet dies, daß sie sowohl quantitativ als auch in qualitativ diesem Anspruch genügen müssen.

Das spezifizierte Front-End des Übersetzers (im folgenden abkürzend nur noch Übersetzer genannt) liegt nach der Generierung in Form eines ausführbaren Programms mit dem Namen `of` vor. Durch die Angabe spezieller Kommandozeilenparameter lassen sich die verschiedenen Komponenten des Front-Ends einzeln testen (vgl. S.104). Es existieren deshalb grundsätzlich drei verschiedene Sätze von Testprogrammen, mit denen jeweils spezifische Komponenten des Übersetzers getestet werden können. Dies sind die Testprogramme für den Scanner (`scan*.ob2`), für den Parser (`pars*.ob2`) sowie für den Evaluator (`e*.ob2`). Diese Sätze lassen sich jeweils wiederum unterteilen in die Menge der (bezüglich der Komponente) korrekten und der fehlerhaften Testprogramme. Die gewählten Namen aller Testprogramme sollen eine leichte Zuordnung zu diesen Mengen ermöglichen. Es wurde deshalb die folgende Namenskonvention verwendet:

Alle Testprogramme besitzen Namen aus acht Zeichen, gefolgt von einem Punkt und der Namenserweiterung `ob2`. Das achte Zeichen ist bei einem (bezüglich der zugehörigen Komponente) korrekten Programm ein `t` (True), bei einem fehlerhaften Programm ein `f` (False). Die drei Zeichen vor diesem achten Zeichen sind Ziffern, die jeweils zusammen eine fortlaufende Nummer für die korrekten und fehlerhaften Programme bilden. Die ersten vier Buchstaben sind bei Scanner-Testprogrammen alle `scan`, bei den Parser-Testprogrammen `pars` und bei den Evaluator-Testprogrammen ein vierstelliger Buchstabencode, welcher mit einem `e` beginnt und dessen letzte drei Buchstaben den Abschnitt der Sprachdefinition kodieren, in dem die Kontextbedingungen beschrieben sind, welche mit diesem Testprogramm geprüft werden. Es wurde die nachfolgend dargestellte Kodierung verwendet. Für Abschnitte ohne Kodierungskürzel existieren entweder keine Kontextbedingungen oder die Kontextbedingungen werden in einem anderen Abschnitt mit geprüft. Die aufgeführten Abschnittsnummerierungen sowie -titel sind aus der englischsprachigen Sprachdefinition von MÖSSENBÖCK und WIRTH [1993] übernommen, da sich die gewählte Kürzelkodierung an ihren Titeln orientiert.

4.	Declarations and scope rules	`dec`
5.	Constant declarations	`con`
6.	Type declarations	`typ`
6.1	Basic types	`tba`
6.2	Array types	`tar`
6.3	Record types	`tre`
6.4	Pointer types	`tpo`
6.5	Procedure types	`tpr`
7.	Variable declarations	`var`
8.	Expressions	
8.1	Operands	`eop`
8.2	Operators	
8.2.1	Logical operators	`elo`
8.2.2	Arithmetic operators	`eao`
8.2.3	Set Operators	`eso`
8.2.4	Relations	`ere`
9.	Statements	

9.1	Assignments	
9.2	Procedure calls	`spr`
9.3	Statement sequences	
9.4	If statements	`sif`
9.5	Case statements	`sca`
9.6	While statements	`swh`
9.7	Repeat statements	`sre`
9.8	For statements	`sfo`
9.9	Loop statements	
9.10	Return and exit statements	`srt`
9.11	With statements	`swi`
10.	Procedure declarations	`pro`
10.1	Formal parameters	`pfp`
10.2	Type-bound procedures	`ptb`
10.3	Predeclared procedures	`ppr`
11.	Modules	`mod`
Appendix A:	Definition of terms	`apa`
	Same Types	`ast`
	Equal Types	`aet`
	Type Inclusion	`ati`
	Type extension	`ate`
	Assignment compatible	`aas`
	Array compatible	`aar`
	Expression compatible	`aex`
	Matching formal parameter lists	`amf`
Appendix C:	The module `SYSTEM`	`apc`

Testprogramme für den Scanner

Mittels der Kommandozeilenoption `Y` wird der Übersetzer veranlaßt, lediglich die vom Scanner gelieferten Symbole in ihrer textuellen Repräsentation auszugeben. Die Ausgabe stützt sich dabei auf die vom Scanner-Modul exportierte Prozedur `TokenNum2TokenName`. Die Testprogramme für den Scanner beinhalten die in Oberon-2 verwendeten Symbole sowie Fehler, die bereits von der Scanner-Komponente erkannt werden.

Testprogramme für den Parser

Die Kommandozeilenoption `d` führt zur Ausgabe des erzeugten abstrakten Syntaxbaums in Quelltextdarstellung nach der Parsierung. Zur Ausgabe wird die Puma-Funktion `TD.Dump` verwendet. Die Parser-Testprogramme beinhalten ein breites Spektrum von kontextfrei korrekten und fehlerhaften Oberon-2 Modulen.

Testprogramme für den Evaluator

Mit der Kommandozeilenoption `Ts` ist es möglich, nach der Evaluation die globale und alle lokalen Symboltabellen ausgeben zu lassen. Diese Ausgabe stützt sich auf die Puma-Funktion `OD.DumpTable` bzw. `OD.DumpTable0`. Ausgegeben werden jeweils die Objekte und ihre Typrepräsentationen.

Die Evaluator-Testprogramme orientieren sich sehr stark an den entsprechenden Abschnitten der Sprachdefinition. Damit das lokale Verständnis erhöht wird, enthalten die Testprogramme diese Passagen noch einmal als Quelltextkommentar. Ebenfalls in der Form eines Quelltextkommentars enthalten die fehlerhaften Programme einen Verweis auf die Fehlerstelle und die genaue Fehlermeldung.

Testlauf

Testläufe dienten während der Entwicklungsphase der Untersuchung, ob sich das Verhalten des Übersetzers, vor allem nach größeren Designänderungen, bezüglich bereits implementierter Eigenschaften verändert hat. Dazu besitzen alle Testprogramme eine Referenzdatei, in der die Ausgabe des Übersetzers für dieses Testprogramm festgehalten ist. Je nach Satz, zu dem das Testprogramm gehört, werden alle Symbolrepräsentationen, der abstrakte Syntaxbaum (in Quelltextdarstellung) oder die Symboltabellen sowie alle Fehlermeldungen ausgegeben. Ein Testlauf besteht nun aus der ,,Übersetzung“ aller Testprogramme, dem (zeichenweisen) Vergleich der dabei erzeugten Ausgabe mit der Referenzausgabe und der Meldung des Vergleichsergebnisses.

Abbildungsverzeichnis

Literaturverzeichnis

J. GROSCH [1987]:
Reusable Software – A Collection of MODULA-Modules ,
Compiler Generation Report No. 4, GMD Forschungsstelle an der Universität Karlsruhe, September 1987.

J. GROSCH [1991a]:
Rex – A Scanner Generator ,
Compiler Generation Report No. 5, GMD Forschungsstelle an der Universität Karlsruhe, März 1991.

J. GROSCH [1991b]:
Puma – A Generator for the Transformation of Attributed Trees ,
Compiler Generation Report No. 26, GMD Forschungsstelle an der Universität Karlsruhe, Juli 1991.

J. GROSCH [1991c]:
Ast – A Generator for Abstract Syntax Trees ,
Compiler Generation Report No. 15, GMD Forschungsstelle an der Universität Karlsruhe, September 1991.

J. GROSCH [1991d]:
Ag – An Attribute Evaluator Generator ,
Compiler Generation Report No. 16, GMD Forschungsstelle an der Universität Karlsruhe, September 1991.

J. GROSCH und H. EMMELMANN [1990]:
A Tool Box for Compiler Construction ,
Compiler Generation Report No. 20, GMD Forschungsstelle an der Universität Karlsruhe, Januar 1990.

J. GROSCH und B. VIELSACK [1991]:
The Parser Generators Lalr and Ell ,
Compiler Generation Report No. 8, GMD Forschungsstelle an der Universität Karlsruhe, September 1991.

H. MÖSSENBÖCK [1992]:
The Programming Language Oberon-2 ,
Report 160, ETH Zürich, Institut für Computersysteme, Mai 1992 (überarbeitet).

H. MÖSSENBÖCK [1993]:
Objektorientierte Programmierung in Oberon-2 ,
Springer-Verlag, Berlin, 1993.

H. MÖSSENBÖCK und N. WIRTH [1993]:
The Programming Language Oberon-2 ,
ETH Zürich, Institut für Computersysteme, Oktober 1993,
ftp://neptune.inf.ethz.ch/pub/Oberon/Docu/Oberon2.Report.ps.Z .

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: Users Guide and Programmers Manual ,
Addison-Wesley, New York.

M. REISER und N. WIRTH [1992]:
Programming in Oberon, Steps beyond Pascal and Modula ,
Addison-Wesley, New York.

R. WILHELM und D. MAURER [1992]:
Übersetzerbau: Theorie, Konstruktion, Generierung ,
Springer-Verlag, Berlin.

N. WIRTH [1985]:
Programming in Modula-2 ,
Springer-Verlag, New York, Third, Corrected Edition.

N. WIRTH und J. GUTKNECHT [1992]:
Project Oberon: The Design of an Operating System and Compiler ,
Addison-Wesley.