Entwurf und Implementierung eines Übersetzers von Modula-2 nach C

Diplomarbeit

von

Matthias Martin

Universität Karlsruhe
Fakultät für Informatik

Februar 1990

Betreuer:

Prof. Dr. S. Jähnichen

Dr. J. Grosch

Übersicht

Diese Diplomarbeit beschreibt den Entwurf und die Implementierung des Übersetzers Mtc, der in Modula-2 geschriebene Programme in lesbaren C-Code umsetzt. Die Definition einer vollständigen Abbildung von Modula-2 nach C, die auch anhand von zahlreichen Beispielen illustriert wird, bildet den Hauptteil dieser Arbeit. Die Implementierung des Übersetzers erfolgte unter weitgehendem Einsatz von Übersetzerbauwerkzeugen. Die Beschreibung der Implementierung konzentriert sich daher besonders auf die Darstellung des Einsatzes dieser Werkzeuge. Ein abschließendes Kapitel über erste praktische Einsätze von Mtc demonstriert die Leistungsfähigkeit und Laufzeiteffizienz des Übersetzers sowie die Qualität des erzeugten C-Codes.

1. Einleitung

Diese Arbeit beschreibt den Entwurf und die Implementierung des Übersetzers *Mtc*, der in Modula-2 geschriebene Programme in lesbaren C-Code umsetzt.

1.1 Motivation und Zielsetzung

Modula-2 [Wirth 85] ist eine Weiterentwicklung der Programmiersprache Pascal, die im wesentlichen um ein Modulkonzept erweitert wurde. An der GMD Forschungsstelle Karlsruhe wurde der Modula-Übersetzer MOCKA [Engelmann 87] entwickelt, der auch vollständig in Modula-2 implementiert ist. Modula-2 und MOCKA werden zur Zeit an der GMD Forschungsstelle Karlsruhe als bevorzugte Werkzeuge zur Programmentwicklung eingesetzt. Insbesondere sind alle in den letzten drei Jahren am Institut entwickelten Übersetzerbauwerkzeuge in Modula-2 implementiert.

Die Sprache C [Kernighan 78] wurde ursprünglich entworfen und implementiert für die Entwicklung des UNIX-Betriebssystems [UNIX 79], welches abgesehen von wenigen Teilen, bei denen extrem hohe Effizienzanforderungen oder spezielle Maschineneigenschaften den Einsatz von Assembler notwendig machen, vollständig in C implementiert ist. Da UNIX nicht nur ein Betriebssystem ist, sondern, weniger eng betrachtet, unter dem Begriff UNIX eine Programmierumgebung mit einer Vielzahl von nützlichen Werkzeugen für die Programmentwicklung verstanden wird, hat UNIX in den letzten Jahren, besonders im wissenschaftlichen Bereich, eine sehr weite Verbreitung gefunden. Auch an der GMD Forschungsstelle Karlsruhe wird UNIX eingesetzt.

Aus den einleitenden Vorbemerkungen läßt sich nun direkt die hauptsächliche Zielsetzung und das geplante Einsatzgebiet von *Mtc* ableiten:

- Der Modula-Übersetzer MOCKA und die am Institut entwickelten Übersetzerbauwerkzeuge können mit *Mtc* schnell und relativ einfach auf neue Maschinen übertragen werden, auf denen das UNIX-System und ein Übersetzer für die Sprache C vorhanden sind.
- Die Übersetzerbauwerkzeuge werden auch für alle Interessenten zugänglich, die einen Übersetzer für die im Vergleich mit Modula-2 weit verbreitete Sprache C besitzen.
- Die Weiterentwicklung der Übersetzerbauwerkzeuge kann in der besser strukturierten, moderneren und relativ sicheren Sprache Modula-2 erfolgen, die Werkzeuge sind aber auch jederzeit, ohne den riesigen Aufwand einer Übersetzung von Hand, in C und unter UNIX verfügbar.

Da *Mtc* für die Übertragung von fertigen Modula-Programmen nach C gedacht ist und nicht für die Programmentwicklung eingesetzt werden soll, kann bei der Implementierung davon ausgegangen werden, daß nur korrekte Programme als Eingabe vorkommen. Insbesondere soll die semantische Korrektheit der Eingabeprogramme nicht überprüft werden. Weil die übersetzten Programme unter Umständen noch von Hand weiterbearbeitet werden sollen, ist eine wichtige Forderung an die erzeugten Programme, daß sie gut lesbar sind.

Ein weiteres Ziel dieser Arbeit ist der praktische Einsatz der bisher entwickelten Übersetzerbauwerkzeuge bei der Implementierung von *Mtc*, um dabei:

- Die praktische Einsetzbarkeit der Werkzeuge für die Entwicklung eines größeren Übersetzers zu zeigen.
- Die Leistungsfähigkeit der Werkzeuge und des (teilweise) generierten Übersetzers zu demonstrieren.
- Erfahrungen zu sammeln, die zu einer Verbesserung der Werkzeuge führen.

1.2 Randbedingungen

Die Implementierung des Übersetzers erfolgt auf UNIX-Workstations der Firmen PCS und SUN. Implementierungssprache ist Modula-2. Die folgenden an der GMD Forschungsstelle Karlsruhe entwickelten Übersetzerbauwerkzeuge standen für die Entwicklung von *Mtc* zur Verfügung:

- Der Generator für Symbolentschlüssler *Rex* [Grosch 87b].
- Die Zerteilergeneratoren *Lalr* und *Ell* [Vielsack 88].
- *Ast* [Grosch 89a], ein Werkzeug zur Spezifikation und Implementierung von abstrakten Syntaxbäumen.
- *Ag* [Grosch 89b], ein Generator für Attributauswerter.
- Das Werkzeug *Estra* [Vielsack 89] zur Spezifikation und Implementierung der Transformation attributierter Strukturbäume.

Neben diesen Werkzeugen wurden Modula-Moduln aus der umfangreichen Bibliothek *Reuse* [Grosch 87a] mit wiederverwendbarer Software verwendet.

2. Vergleichbare Arbeiten

In diesem Kapitel sollen kurz zwei mit *Mtc* vergleichbare Arbeiten vorgestellt werden, die sich mit der Übersetzung von Programmen von einer höheren Programmiersprache in eine andere höhere Programmiersprache beschäftigen.

2.1 PTC

Der Übersetzer PTC [PTC 87] war für die vorliegende Arbeit von besonderem Interesse, da die Zielsprache ebenfalls C und die Quellsprache Pascal der Vorgänger von Modula-2 ist. Er wurde von P. Bergsten bei der Firma Holistic Technology AB, Gothenburg, Schweden entwickelt.

PTC verarbeitet ein korrektes Pascal-Programm und erzeugt ein äquivalentes C-Programm. Der Übersetzer, der (fast) vollständiges Pascal implementiert, wurde entworfen, um fertig entwickelte Pascal-Programme auf Systeme zu übertragen, die keinen Pascal-Übersetzer besitzen. Er ist nicht für die Programmentwicklung gedacht. Daher wird die Korrektheit der

Eingabeprogramme auch nicht überprüft und beim ersten entdeckten Fehler bricht der Übersetzer mit einer entsprechenden Meldung ab. PTC besitzt daher keinerlei Fehlerbehandlung und fehlerhafte Programme können im schlimmsten Fall sogar zum Absturz des Übersetzers führen.

Auf die von PTC implementierte Abbildung von Pascal nach C soll hier nicht im Detail eingegangen werden. Da Modula-2 ein Nachfolger von Pascal ist und daher ein nicht unerheblicher Teil der beiden Sprachen, abgesehen von kleinen syntaktischen Unterschieden, nahezu identisch ist und darüberhinaus an vielen Stellen die Abbildung nach C auf der Hand liegt, stimmt die von PTC implementierte Abbildung für die gemeinsamen Konstrukte von Modula-2 und Pascal im Prinzip mit der in Kapitel 4 beschriebenen Abbildung von Modula-2 nach C überein. Dort wird an den entsprechenden Stellen darauf verwiesen, wo besonders elegante Lösungen von PTC übernommen wurden.

Hier soll nur kurz dargestellt werden, welche Probleme bei der Abbildung für Modula-2 im Vergleich zu Pascal entfallen und welche neu hinzukommen.

Da in Modula-2 die Ein-/Ausgabe, die Dateiverwaltung sowie die Speicherverwaltung nicht Teil der Sprachdefinition sind, entfällt die für Pascal notwendige und teilweise sehr aufwendige Abbildung dieser Konstrukte, die von PTC mit Hilfe der C-Standardbibliothek implementiert werden. Sprünge, insbesondere Sprünge, die eine Prozedur verlassen und die kein direktes Gegenstück in C haben, sind in Modula-2 ebenfalls nicht möglich. Ein weiteres, von PTC allerdings nicht gelöstes Problem, welches in Modula-2 entfällt, ist die in Pascal gegebene Möglichkeit lokal deklarierte Prozeduren als Prozedurparameter zu übergeben.

Verglichen mit Pascal kommen in Modula-2 insbesondere das Modulkonzept und die getrennte Übersetzung neu hinzu. Außerdem ist die Reihenfolge der Deklarationen und Anweisungen in Modula-2 nicht so stark eingeschränkt wie in Pascal und es sind allgemeine Ausdrücke in Deklarationen zulässig. Das Konzept der Übergabe von Funktionen bzw. Prozeduren als Parameter wurde in Modula-2 zu dem allgemeinen Konzept der Prozedurtypen und -variablen erweitert. Zusätzlich neu ist die Möglichkeit, Felder beliebiger Größe an Prozeduren zu übergeben sowie eine Reihe von Möglichkeiten zur „maschinennahen'' Programmierung.

PTC ist in Pascal implementiert. Das Programm ist im wesentlichen aus den drei Prozeduren *parse*, *transform* und *emit* aufgebaut.

Die Prozedur *parse* liest und zerteilt das Quellprogramm und baut einen abstrakten Strukturbaum auf. Die Zerteilung arbeitet nach dem Verfahren des rekursiven Abstiegs. Parallel zur Zerteilung wird eine Definitionstabelle aufgebaut und die Bezeichneridentifikation durchgeführt.

Die Prozedur *transform* führt eine Reihe von Baum-zu-Baum Transformationen durch, die notwendig sind, um das Pascal-Programm nach C zu übersetzen. Die wichtigste dieser Transformationen ist die Delokalisierung von in Pascal lokal deklarierten Prozeduren, da in C keine geschachtelten Funktionsdeklarationen zulässig sind. Zusätzlich zur Transformation des Strukturbaums werden bei Bedarf auch Bezeichner zur Vermeidung von Namenskonflikten umbenannt.

Die Prozedur *emit* schließlich traversiert den transformierten Baum und gibt die entsprechenden C-Konstrukte aus.

2.2 PascAda

Das an der University of California entworfenen *PascAda*-System [PascAda 80] besteht aus zwei Übersetzern; der eine transformiert Pascal-Programme in Ada-Programme, der zweite führt die entgegengesetzte Transformation durch. Beide Übersetzer sind in Pascal implementiert. Die wichtigsten Ziele dieses Projekts waren[1]:

- Da die beiden Übersetzer in Pascal implementiert sind, für das Implementierungen existieren, stellt der Übersetzer von Ada nach Pascal einen Übersetzer für (eine Teilmenge von) Ada dar.
- Möglichkeit der vorläufigen Entwicklung von Software in Ada.
- Umsetzen existierender, in Pascal implementierter Software nach Ada.
- Unterstützung der Übertragung von in Ada und/oder Pascal implementierter Software auf neue Systeme.
- Möglichkeit zu Experimenten mit Ada.
- Bootstrap eines Ada-Übersetzers.

Die *PascAda*-Gruppe ist bei diesem Projekt folgendermaßen vorgegangen. Zunächst wurden zwei Teilmengen *AdaP* und *PascalA* von Ada und Pascal definiert, für die eine direkte 1:1 Übersetzung in die jeweils andere Teilmenge existiert. Im nächsten Schritt wurden dann zwei erweiterte Teilsprachen *AdaPE* und *PascalAE* definiert, für welche Transformationsregeln gefunden werden konnten, wie Konstrukte dieser erweiterten Teilsprachen durch Transformation der Quellprogramme auf Konstrukte von *AdaP* bzw. *PascalA* abgebildet werden können. Da die Semantik von Ada eine weitgehende Obermenge von Pascal ist, ist *PascalAE* nahezu vollständiges Pascal. Eine Ausnahme sind z.B. Prozeduren und Funktionen als Parameter oder nicht lokale Sprünge. Da für viele Sprachkonzepte von Ada, wie z.B. Parallelverarbeitung und Ausnahmebehandlung, keine entsprechenden Konzepte in Pascal existieren und eine Simulation, wenn überhaupt, nur mit großem Aufwand möglich wäre, sind die Einschränkungen bei *AdaPE* natürlich größer.

Bei der Implementierung der beiden Übersetzer wurde als gemeinsame Zwischensprache die Baumstruktur aus einer formalen Definition von Ada [Donzeau 79] benutzt, die um einige spezielle Konstrukte für *PascalAE* erweitert wurde. Der Übersetzer von Pascal nach Ada besteht logisch aus 4 Komponenten:

PascalToTree:	Setzt das Quellprogramm in einen Baum der Zwischensprache um.
PascalAEtoA:	Setzt *PascalAE*-Konstrukte durch Baum-zu-Baum Transformationen in entsprechende *PascalA*-Konstrukte um.
PascalACheck:	Prüft, ob der Baum einem Programm der Teilsprache *PascalA* entspricht.
TreeToAda:	Setzt den Baum in ein entsprechendes Ada-Programm um.

Der Übersetzer von Ada nach Pascal besteht aus 4 analogen Komponenten.

Beide Übersetzer führen keine komplette semantische Analyse durch, sondern prüfen nur ob ein Programm in der jeweiligen Teilsprache *PascalAE* bzw. *AdaPE* enthalten ist.

---

1 Wobei man berücksichtigen muß, daß das Projekt zu einem Zeitpunkt durchgeführt wurde, als gerade eine erste Definition der Sprache Ada aber noch kein Ada-Übersetzer vorlag.

3. Die Sprachen

3.1 Die Quellsprache Modula-2

Die Programmiersprache Modula-2 [Wirth 85] wurde Ende der siebziger Jahre an der ETH Zürich unter der Leitung von Professor N. Wirth entwickelt. Modula-2 ist ein direkter Nachfolger der Programmiersprache Pascal. Das wichtigste neue Sprachkonzept im Vergleich zu Pascal ist ein Modulkonzept. Die folgende Aufzählung gibt einen kurzen Überblick über Modula-2:

- Ein Modula-Programm besteht aus einer Reihe von Übersetzungseinheiten, die getrennt übersetzt werden können. Eine solche Übersetzungseinheit ist entweder ein Programmodul, ein Definitionsmodul oder ein Implementierungsmodul. Ein Programmodul stellt ein Hauptprogramm dar. Definitions- und Implementierungsmoduln treten immer paarweise auf. Es handelt sich dabei jeweils um einen logisch zusammengehörenden Modul. Der Definitionsmodul stellt die Schnittstelle des Moduls dar und spezifiziert die vom Modul exportierten, d.h. die nach außen sichtbaren Objekte (Daten und Operationen). Der Implementierungsmodul enthält die Implementierung der nach außen sichtbaren Operationen und eventuell weitere lokale Objekte. Der Realisierung des Geheimnisprinzips dienen die sogenannten opaquen Typen, deren Struktur den sie benutzenden Moduln verborgen bleibt.
- Alle im Modula-Programm benutzten Objekte müssen durch eine Deklaration bekannt gemacht werden, es sei denn, es handelt sich um ein vordefiniertes Objekt. Die Objekte von Modula-2 sind: Konstanten, Typen, Variablen, Prozeduren und Moduln.
- Modula-2 ist eine blockstrukturierte Sprache. Zusätzlich zur Blockstruktur erlaubt das Modulkonzept eine explizite Kontrolle des Gültigkeitsbereichs bzw. der Sichtbarkeit von Objekten. Alle Objekte, die innerhalb von Deklarationen verwendet werden, müssen textuell vor ihrer Benutzung deklariert werden; ein Objekt kann aber innerhalb einer Anweisung benutzt werden, die der Deklaration des Objekts textuell vorangeht.
- Modula-2 ist eine streng typisierte Sprache. Die Grundtypen von Modula-2 sind: Zeichen, Wahrheitswerte sowie ganze und reelle Zahlen verschiedener Größe. Neben den Grundtypen besitzt Modula-2 einen umfangreichen Satz von Typkonstruktoren. Die möglichen Typen sind: Grundtypen, Aufzählungstypen, Unterbereichstypen, Felder, Verbunde, Verbunde mit Varianten, Mengen, Zeiger und Prozedurtypen.
- Prozedurdeklarationen bestehen aus einem Prozedurkopf, der angibt, welche formalen Parameter und welchen Resultattyp[1] die Prozedur besitzt und einem Rumpf, der lokale Deklarationen und Anweisungen enthält. Modula-2 kennt drei Arten von formalen Parametern: neben den von Pascal her bekannten Wert- und Referenzparametern gibt es noch sogenannte offene Felder. Offene Felder sind eindimensionale Felder, deren Feldgrenzen bei der Parameterdeklaration nicht festgelegt werden. Dadurch können Felder unterschiedlicher Größe von einer Prozedur bearbeitet werden. In Modula-2 können beliebige Objekte lokal zu einer Prozedur deklariert werden. Das bedeutet insbesondere, daß auch Prozedurdeklarationen geschachtelt werden können. Prozeduren können auch rekursiv aufgerufen werden.
- Ein (lokaler) Modul besteht im wesentlichen aus einer Reihe von Deklarationen und einer Anweisungsfolge. Die Anweisungsfolge dient der Initialisierung von lokalen Objekten. Grundsätzlich sind innerhalb des Moduls deklarierte Objekte außerhalb unsichtbar und umgekehrt sind außerhalb deklarierte Objekte innerhalb des Moduls unsichtbar. Durch Import-

1 Der Resultattyp wird nur bei Funktionsprozeduren angegeben.

bzw. Exportanweisungen kann der Programmierer die Sichtbarkeit von Objekten explizit kontrollieren und Objekte innerhalb/außerhalb des Moduls sichtbar werden lassen. Moduln ermöglichen es also, zusammengehörige Daten und Operationen zusammenzufassen und gegenüber anderen Programmteilen abzugrenzen. Die Import- bzw Exportanweisungen bilden dabei die Schnittstellen des Moduls nach außen. Ein Definitionsmodul ist im Prinzip nichts anderes, als die (erweiterte) Exportliste des zugehörigen Implementierungsmoduls.

- Neben den üblichen arithmetischen, logischen und relationalen Operatoren sowie Operatoren zum Zugriff auf strukturierte Variablen besitzt Modula-2 auch Operatoren zur Manipulation von Mengen.
- Die Anweisungen von Modula-2 sind: Zuweisung, bedingte Anweisung, Fallunterscheidung, verschiedene Schleifenkonstrukte, Schleifenausgangsanweisung, Rückkehranweisung und Prozeduraufruf. In Modula-2 können auch beliebig strukturiete Objekte als Ganzes zugewiesen werden. Eine Sprunganweisung existiert in Modula-2 nicht. Eine Spezialität von Modula-2 (und Pascal) ist die WITH-Anweisung, die es ermöglicht, auf Komponenten von Verbunden ohne Qualifikation durch den Namen des Verbundes zuzugreifen.
- Modula-2 bietet eine Reihe von Möglichkeiten, die für eine „maschinennahe'' Programmierung notwendig sind. Insbesondere können die Regeln der strengen Typisierung umgangen und die Adressen von beliebigen Variablen bestimmt werden. Die meisten dieser Möglichkeiten sind Datentypen und Prozeduren, die im vordefinierten Pseudomodul SYSTEM enthalten sind.
- Modula-2 erlaubt die Spezifikation von quasiparallelen Prozessen mit Hilfe von Koroutinen.
- Die Definition der Ein- und Ausgabe sowie der Datei- und Speicherverwaltung sind nicht Teil der Sprachdefinition von Modula-2.

3.2 Die Zielsprache C

Die Sprache C [Kernighan 78] wurde zu Beginn der siebziger Jahre von D. Ritchie für die Entwicklung des UNIX-Betriebssystems entworfen und implementiert. C ist eine relativ maschinennahe Sprache, die sich besonders für die Entwicklung von Betriebssystemen und anderen hardwarenahen Systemteilen eignet. Es wurden aber auch zahlreiche Übersetzer und umfangreiche Anwendungssoftware wie z.B. Datenbankanwendungen in C implementiert. Die folgende Aufzählung soll einen kurzen Überblick über die Sprache C geben, sofern sie für diese Arbeit relevant ist:

- Ein C-Programm besteht aus einer Reihe von Quelldateien (Moduln), die eine Folge von Typ-, Daten- und Funktionsdeklarationen enthalten und vom C-Übersetzer getrennt übersetzt werden können. Im Gegensatz zu Modula-2 werden vom C-Übersetzer aber keinerlei Prüfungen über die Grenzen der einzelnen Quelldateien hinweg vorgenommen.
- Der C-Präprozessor, welcher ein Teil der Sprachdefinition von C ist, ist ein Makroprozessor, der Textersatz vornehmen, Dateien in eine Quelldatei einfügen und Teile eines Programms von der Übersetzung ausschließen kann.
- In C müssen alle Objekte vor ihrer Anwendung deklariert werden. Es besteht jedoch die Möglichkeit von Vorwärtsdeklarationen und von sogenannten *extern*-Deklarationen, mit deren Hilfe in anderen Quelldateien deklarierte Objekte bekannt gemacht werden können.

- C kennt eine einfache Art von Blockstruktur: Variablen können innerhalb von Funktionen in Blöcken deklariert werden und verdecken globale Objekte bzw. Variablen in umfassenden Blöcken mit dem gleichen Namen. Funktionen können jedoch nicht lokal in einer anderen Funktion deklariert werden, dürfen also nicht statisch geschachtelt sein.
- Lebensdauer bzw. Sichtbarkeit von Objekten werden in C u.a. durch ihre Speicherklasse festgelegt. Lokale Variablen von Funktionen können dabei folgende Speicherklassen haben:

  static: Lebensdauer entspricht der Programmdauer. Dies entspricht den *own*-Variablen von Algol.

  auto: Variablen liegen auf dem Keller. Lebensdauer von Blockein- bis Blockaustritt.

  register: Lebensdauer wie *auto*-Objekte, diese sollen aber nach Möglichkeit vom Übersetzer in einem Register abgelegt werden.

  Die Lebensdauer von globalen Variablen ist die gesamte Programmdauer. Die Angabe der Speicherklasse *static* verhindert bei globalen Objekten, daß sie außerhalb der sie enthaltenden Quelldatei sichtbar sind.
- C ist keine streng typisierte Sprache. Für arithmetische Typen ist eine große Zahl von impliziten Typumwandlungen definiert. Neben impliziten Typumwandlungen sind auch explizite Konvertierungen möglich. Zuweisungen zwischen Zeigern verschiedenen Typs sind ebenfalls möglich. Die Grundtypen von C sind: Zeichen, ganze und reelle Zahlen mit verschiedenem Speicherbedarf. Die zusammengesetzten Datentypen von C sind: Zeiger, Vektoren (Felder), Strukturen (Verbunde)[1], Varianten und Aufzählungen. Eine Variante ist eine Struktur, bei der sich die Komponenten überlagern. Es handelt sich also um eine eingeschränkte Art von variantem Verbund.
- Eine Funktionsdeklaration besteht aus dem Ergebnistyp der Funktion, der Deklaration der Parameter und aus lokalen Deklarationen und Anweisungen in Form eines Blocks. Wie bereits erwähnt können Funktionsdeklarationen nicht geschachtelt werden. Als Hauptprogramm fungiert die Funktion *main*. Die Parameterübergabe erfolgt durch Wertübergabe (*call by value*). Bei Vektoren wird ein Zeiger auf den Anfang des Vektors übergeben (*call by reference*). Funktionen können auch rekursiv aufgerufen werden.
- C besitzt eine große Zahl von Operatoren. Die wichtigsten dieser Operatoren sind: Arithmetische, logische und relationale Operatoren, Operatoren zur Selektion von Komponenten strukturierter Variablen und Operatoren zur Bitmanipulation. Darüberhinaus gibt es noch eine Reihe von speziellen Operatoren wie bedingte Ausdrücke oder Inkrement- und Dekrementoperatoren.
- Die Anweisungen von C sind: Zuweisung, Sprunganweisung, bedingte Anweisung, Fallunterscheidung, verschiedene Schleifenkonstrukte, Schleifenausgangsanweisung, Rückkehranweisung und Prozeduraufruf. Die Zuweisung ist allerdings eingeschränkt: Vektoren können nicht als Ganzes zugewiesen werden und Strukturen nur bei neueren Implementierungen.
- C besitzt einen Adreßoperator, mit dem die Adressen sämtlicher Objekte bestimmt werden können. Insbesondere kann man auch die Adressen von Funktionen bestimmen und diese dann später über diese Adressen aufrufen.

---

1 In der deutschsprachigen Fachliteratur haben sich die Begriffe Vektor und Struktur für die entsprechenden C-Datenstrukturen eingebürgert.

3.3 Vergleich

Beim Vergleich von Modula-2 und C zeigt sich, daß C, trotz der auch in Modula-2 vorhandenen Möglichkeiten zur „maschinennahen'' Programmierung, die maschinennähere Sprache ist. Modula-2 besitzt ein deutlich höheres Abstraktionsniveau und an vielen Punkten mächtigere Ausdrucksmittel. Allerdings besteht, wenn man von den Koroutinen einmal absieht, die Möglichkeit in C nicht direkt vorhandene Modula-Konstrukte durch eine Kombination von primitiveren C-Konstrukten zu realisieren. So können z.B. die Mengen von Modula-2 in C mit Hilfe der Operatoren zur Bitmanipulation realisiert werden.

Die folgenden Punkte erfordern bei der Abbildung besonderen Aufwand:

- C kennt kein Modulkonzept, welches dem von Modula-2 vergleichbar wäre. Insbesondere existieren keine den opaquen Typen und den lokalen Moduln entsprechenden Konzepte.
- Die Regeln hinsichtlich Gültigkeit und Sichtbarkeit von Bezeichnern weisen eine Reihe von Unterschieden auf, z.B. verlangt C im Gegensatz zu Modula-2, daß alle Objekte immer vor ihrer Anwendung deklariert werden.
- Die zusammengesetzten Typen von C sind weniger mächtig als die zusammengesetzten Typen von Modula-2. Besondere Schwierigkeiten bereitet, daß sich die Behandlung von Vektoren von der Behandlung anderer Variablen unterscheidet. Auch die eingeschränkte Form von varianten Verbunden in C erfordert eine besondere Behandlung.
- Da C im Gegensatz zu Modula-2 keine geschachtelten Funktionsdeklarationen kennt, muß die in Modula-2 vom Übersetzer vorgenommene Verwaltung der Prozedurschachteln zum Zugriff auf die lokalen Variablen von statisch umfassenden Prozeduren (Verweis auf den statischen Vorgänger) auf geeignete Art und Weise mit Hilfe von C-Code vorgenommen werden.
- C kennt kein den offenen Feldern entsprechendes Konzept der Parameterübergabe.

Abgesehen von den oben genannten Punkten kann ein großer Teil der Deklarationen sowie die meisten Anweisungen und Ausdrücke ohne große Probleme direkt nach C übersetzt werden. Allerdings zeigt es sich bei der im folgenden Kapitel beschriebenen Abbildung, daß manchmal auch schon kleine semantische Unterschiede, bei sich direkt entsprechenden Sprachkonstrukten, die Abbildung deutlich verkomplizieren können.

4. Abbildung von Modula-2 nach C

4.1 Grundlagen der Abbildung

Bevor man eine Abbildung von Modula-2 nach C definieren kann, muß man zunächst klären, welche Definition der beiden Sprachen als Basis für diese Abbildung verwendet werden soll. Die wichtigsten Gründe dafür sind:

- Seit ihrer Entstehung unterlagen beide Sprachen einer gewissen Evolution: manche Sprachelemente wurden verändert, andere sind neu hinzugekommen.
- Verschiedene Implementierungen unterscheiden sich fast immer in einigen Details.

– Die vorhandenen Sprachdefinitionen sind nicht in allen Punkten völlig eindeutig.

Zur Zeit wird zwar für beide Sprachen an einer standardisierten Sprachversion gearbeitet, aber zum einen liegen diese Standards bisher noch nicht endgültig vor, zum anderen dürfte nach dem Vorliegen der Standards noch einige Zeit vergehen, bis Übersetzer verfügbar sind, die sie implementieren. Das Vorgehen in dieser Abbildung orientiert sich daher am geplanten Einsatz des Übersetzers: Übertragung von mit dem Karlsruher Modula-Übersetzer MOCKA unter dem UNIX-Betriebssystem entwickelten Programmen nach C.

MOCKA implementiert die Sprache Modula-2 im wesentlichen so, wie sie von N. Wirth [Wirth 85] definiert wurde. Diese Definition bildet, neben dem MOCKA-Benutzerhandbuch [Engelmann 87], die Ausgangsbasis der Abbildung, die darüberhinaus auch die meisten von MOCKA vorgenommenen Spracherweiterungen berücksichtigt. Insbesondere die Abbildung der Grundtypen, die zugrundegelegten Regeln hinsichtlich Typkompatibilität und die Behandlung von Mengen orientiert sich an der Implementierung von MOCKA.

Für die Sprache C wurde die deutsche Übersetzung der bereits etwas älteren Sprachbeschreibung von B.W. Kernighan und D. Ritchie [Kernighan 78, Kernighan 83] als Ausgangsbasis gewählt. Es werden jedoch auch einige Sprachelemente wie z.B. Zuweisung von Strukturen als Ganzes verwendet, die zum damaligen Zeitpunkt noch nicht in allen Implementierungen verfügbar waren. Die gestellte Mindestanforderung an die erzeugten C-Programme ist, daß sie von den an der GMD Forschungsstelle Karlsruhe verfügbaren C-Übersetzern der Firmen SUN und PCS akzeptiert werden.

In der folgenden Beschreibung der Abbildung wird an den entsprechenden Stellen auf möglicherweise implementierungsabhängige Annahmen besonders hingewiesen.

4.2 Programmrepräsentation

Die erzeugten C-Programme werden zur Erhöhung der Lesbarkeit durch geeignete Einrückungen und Verwendung von Zwischenräumen formatiert.

4.2.1 Bezeichner

Die möglichen Modula-Bezeichner bilden eine Teilmenge der möglichen C-Bezeichner, insbesondere werden in beiden Sprachen Groß- und Kleinbuchstaben unterschieden. Die Bezeichner können daher in der Regel direkt aus dem Modula-Programm in das C-Programm übernommen werden. Um Namenskonflikte im erzeugten C-Programm zu vermeiden, ist es jedoch in den folgenden Fällen notwendig, den Bezeichner durch das Hinzufügen eines Präfix umzubenennen:

– Der Bezeichner ist ein Schlüsselwort der Sprache C.
– Der Bezeichner wird bereits für ein vom Übersetzer vordefiniertes Objekt verwendet.
– Aufgrund der unterschiedlichen Regeln hinsichtlich des Gültigkeitsbereichs von Bezeichnern oder durch eine für die Abbildung nach C notwendige Transformation des Modula-Programms würde ein Namenskonflikt entstehen.

Der Präfix hat die Form *C_nnn_*, wobei *nnn* für eine eindeutige Nummer steht. Die Abkürzungen *nnn* und *xxx* werden im folgenden für die Bezeichnung von vom Übersetzer vergebenen Nummern benutzt.

Es sei an dieser Stelle darauf verwiesen, daß bei älteren C-Übersetzern zur Unterscheidung von Bezeichnern häufig nur wenige Zeichen verwendet werden. Da aber bei neueren C-Übersetzern hier meistens keine Restriktionen mehr bestehen, wird dies bei der Abbildung nicht berücksichtigt.

4.2.2 Numerische Konstanten

Die ganzzahligen Konstanten von Modula-2 werden gemäß der folgenden Tabelle mit regulären Ausdrücken nach C übersetzt:

Modula-2	C
`[0-9]+`	`[0-9]+`
`[0-7]+ B`	`0 [0-7]+`
`[0-7]+ C`	`(unsigned char) '\[0-7]+'`
`[0-9] [0-9A-F]* H`	`0X [0-9] [0-9A-F]*`

Tabelle 4.1: Abbildung von ganzzahligen Konstanten

Bei den dezimalen Konstanten müssen führende Nullen entfernt werden, da sie sonst in C als oktale Konstanten interpretiert werden.

Ganzzahlige Konstanten haben in C den Typ *int* oder *long*, d.h. sie werden als ganze Zahl mit Vorzeichen interpretiert. Modula-2 definiert, daß eine ganzzahlige Konstante n, deren Wert im Bereich $\mathrm{MAX(INTEGER)} < n \leq \mathrm{MAX(CARDINAL)}$ liegt, den Typ CARDINAL hat, d.h. n wird als ganze Zahl ohne Vorzeichen interpretiert. Bei einer direkten Übersetzung einer solchen Konstanten entsprechend der obigen Tabelle nach C, würde das dazugehörige Bitmuster vom C-Übersetzer als negative Zahl interpretiert. Daher wird einer solchen Konstanten in C eine explizite Typumwandlung in den Typ *unsigned long* vorangestellt, um eine korrekte Interpretation als ganze Zahl ohne Vorzeichen durch den C-Übersetzer zu erzwingen.

Für Zeichenkonstanten der Form [0-7]+ C existiert ein ähnliches Problem. MOCKA erlaubt Zeichenkonstanten c im Bereich $0 \leq c \leq 255$. In C haben Zeichenkonstanten den Typ *char*. Werden C-Zeichenkonstanten oder -variablen z.B. in relationalen Ausdrücken verwendet, so werden sie zunächst implizit in den Typ *int* umgewandelt. Diese Umwandlung ist maschinenabhängig. Eine Zeichenkonstante, deren Wert als oktales Bitmuster definiert ist, kann dabei negativ erscheinen. Der Modula-Ausdruck 0C < 377C ist immer wahr, der C-Ausdruck '\0' < '\377' ist dagegen, abhängig von der jeweiligen Maschine, wahr oder falsch. Um diese Fehlinterpretation zu vermeiden, wird, wie der obigen Tabelle zu entnehmen ist, solchen Zeichenkonstanten in C eine explizite Typumwandlung in den Typ *unsigned char* vorangestellt, was zur Folge hat, daß bei einer impliziten Typumwandlung in den Typ *int* kein negativer Wert mehr entstehen kann.

Im Modula-Programm enthaltene Gleitpunktkonstanten können textuell in das erzeugte C-Programm eingesetzt werden, da die in Modula-2 zulässigen Gleitpunktkonstanten eine Teilmenge der in C zulässigen Gleitpunktkonstanten darstellen.

4.2.3 Zeichenketten

Die Abbildung von Zeichenketten hängt davon ab, wie sie im Modula-Programm verwendet werden. Dabei muß insbesondere folgendes beachtet werden:

- Eine Zeichenkette der Länge n ist in Modula-2 vom Typ ARRAY [0..n-1] OF CHAR.
- Eine Zeichenkette der Länge 1 ist in Modula-2 kompatibel mit dem Typ CHAR.
- In C wird eine Zeichenkette als initialisierter Vektor von Zeichen betrachtet, der am Ende ein zusätzliches NUL-Zeichen enthält.
- C unterscheidet zwischen Zeichenkonstanten und Zeichenketten der Länge 1.

Bei der Abbildung wird daher folgendermaßen vorgegangen:

- Eine Zeichenkette der Sprache Modula-2 mit Länge n > 1 wird auf eine Zeichenkette der Sprache C abgebildet.
- Falls eine Zeichenkette der Länge 1 in einem Kontext benutzt wird, in dem ein Ausdruck vom Typ CHAR erwartet wird, dann wird sie auf eine C-Zeichenkonstante abgebildet. Wird diese Zeichenkette jedoch in einem Kontext benutzt, in dem ein Ausdruck vom Typ ARRAY [0..0] OF CHAR erwartet wird, dann wird sie zu einer C-Zeichenkette.

Die folgenden Zeichen müssen in C innerhalb von Zeichenketten und -konstanten durch das Fluchtsymbol \ maskiert werden: \ ’ ".

4.2.4 Kommentare

Die Kommentare im Modula-Programm könnten auf C-Kommentare abgebildet werden, wobei berücksichtigt werden müßte, daß C keine geschachtelten Kommentare kennt. Da aber in Modula-2 Kommentare an jeder beliebigen Stelle des Programms stehen können und es keine allgemeinen Regeln bezüglich der Zuordnung von Kommentaren zu Programmstellen gibt, würde im erzeugten C-Programm — bei jeder für diese Zuordnung gewählten Lösung — ein erheblicher Teil der Kommentare an die falschen Programmstellen plaziert. Daher ist es sinnvoller auf die Kommentare im C-Programm völlig zu verzichten.

4.3 Gültigkeit und Sichtbarkeit

Die Regeln bezüglich Gültigkeit und Sichtbarkeit von Bezeichnern in Modula-2 und C weisen eine Reihe von Unterschieden auf, die bei der Abbildung berücksichtigt werden müssen.

Modula-2 ermöglicht es durch das Modulkonzept den Gültigkeitsbereich bzw. die Sichtbarkeit von in globalen oder lokalen Moduln[1] vereinbarten Bezeichnern mit Import- bzw. Exportanweisungen explizit zu kontrollieren. Außerdem kann man, insbesondere zur Vermeidung von Namenskonflikten, importierte Bezeichner durch den Namen des sie exportierenden Moduls qualifizieren. C hat kein vergleichbares Modulkonzept. Die einzige in C vorhandene Möglichkeit zur expliziten Kontrolle des Gültigkeitsbereichs von Bezeichnern ist, die Sichtbarkeit von

1 Ein globaler Modul ist ein Programmodul oder er besteht aus einem Definitions- und einem Implementierungsmodul.

globalen Objekten durch Angabe der Speicherklasse *static* auf die sie enthaltende Quelldatei zu beschränken. Daher wird bei der Abbildung nach C wie folgt vorgegangen:

- Alle von globalen Moduln exportierten Bezeichner werden in den C-Programmen in der qualifizierten Form *Modulname_Bezeichner* geschrieben, um Namenskonflikte zu vermeiden. Eine Qualifikation nur bei Namenskonflikten ist nicht mit einer getrennten Übersetzung von globalen Moduln nach C vereinbar.
- In lokalen Moduln deklarierte Bezeichner werden in unqualifizierter Form geschrieben. Sie müssen durch einen Präfix umbenannt werden, falls bei der Abbildung der lokalen Moduln nach C Namenskonflikte entstehen.
- Alle globalen Funktionen und Variablen, die nicht exportiert werden, werden in C in der Speicherklasse *static* vereinbart, um ihre Sichtbarkeit auf die sie enthaltende Quelldatei zu beschränken.

Modula-2 erlaubt, daß ein Bezeichner in einer Anweisung verwendet wird, die textuell der Deklaration des Bezeichners vorausgeht; dies ist in C nicht zulässig[1]. Dieses Problem wird folgendermaßen gelöst:

- Alle Prozedurdeklarationen werden an das Ende des sie enthaltenden Deklarationsteils verschoben. Abgesehen von dieser Verschiebung bleibt die ursprüngliche Reihenfolge der Deklarationen erhalten. Damit erfolgen alle Konstanten-, Variablen- und Typdeklarationen vor ihrer ersten Verwendung in einer Anweisung.
- Alle Prozeduren, die vor ihrer Vereinbarung verwendet werden, müssen im C-Programm vor ihrer ersten Benutzung durch eine Deklaration der Form

  ```
  Speicherklasse  Ergebnistyp  Prozedurname  ();
  ```

  bekannt gemacht werden. Eine solche Deklaration — die einer *forward*-Deklaration in Pascal entspricht— ist insbesondere dann notwendig, wenn sich Prozeduren gegenseitig rekursiv aufrufen. Die Speicherklasse ist *static*, falls die Prozedur nicht exportiert wird, und *extern*, falls sie exportiert wird.

Der folgende Ausschnitt eines Modula-Programms

```
PROCEDURE Expr;
BEGIN
  Term;
  WHILE Token = "+" DO
    Token := GetToken ();
    Term;
  END;
END Expr;

PROCEDURE Term;
BEGIN
  Factor;
  ...
```

1 Einzige Ausnahme: Funktionen die ein Resultat vom Typ *int* liefern.

```
PROCEDURE Factor;
BEGIN
  CASE Token OF
  | "(" : Token := GetToken (); Expr;
  ...
VAR  Token : CHAR;
```

wird in C zu

```
CHAR Token;

extern void Term   ();
extern void Factor ();

void Expr()
{
    Term();
    while (Token == '+') {
        Token = GetToken();
        Term();
    }
}

void Term()
{
    Factor();
  ...
void Factor()
{
    switch (Token) {
    case '(' :
        Token = GetToken(); Expr();
  ...
```

Modula-2 verlangt, daß in Deklarationen verwendete Bezeichner vor ihrer ersten Anwendung vereinbart werden. Um jedoch die Definition von „rekursiven'' Typen zu ermöglichen, kann bei der Deklaration von Zeigertypen der Name des Bezugstyps vor seiner Deklaration verwendet werden. Das folgende Beispiel zeigt eine solche „rekursive'' Definition:

```
TYPE
  tTree = POINTER TO tNode;
  tNode = RECORD
             Key   : INTEGER;
             Left  ,
             Right : tTree;
          END;
```

Da C die Verwendung von Strukturnamen erlaubt, bevor eine vollständige Definition der Struktur vorliegt, kann die obige Definition durch Einführung eines Strukturnamens nach C übersetzt werden:

```
typedef struct S_1 *tTree;
typedef struct S_1 {
    INTEGER Key;
    tTree   Left, Right;
} tNode;
```

Ist der Bezugstyp kein Verbund, so ist eine Abbildung nach C nur mit erheblichem Aufwand möglich. Eine allgemeine Lösung könnte folgendermaßen aussehen:

- Handelt es sich nicht um eine „rekursive'' Typdeklaration, dann werden die Deklarationen derart umgeordnet, daß alle Vereinbarungen der beteiligten Typen vor ihrer ersten Anwendung erfolgen[1].
- Handelt es sich um eine „rekursive'' Typdeklaration, dann müssen einige oder alle der beteiligten Typen in C zu einer Struktur gemacht werden. Die folgende Deklaration

  ```
  TYPE
    tPointer1 = POINTER TO tPointer2;
    tPointer2 = POINTER TO tPointer1;
  ```

  würde dann in C zu

  ```
  typedef struct S_2 *tPointer1;
  typedef struct S_2 {
      tPointer1 *X;
  } tPointer2;
  ```

Diese Lösung ist jedoch mit einem erheblichen Übersetzungsaufwand verbunden. Darüberhinaus muß man berücksichtigen, daß Zeigerdeklarationen, in denen der Bezugstyp erst nach dem Zeigertyp definiert wird, in der Praxis hauptsächlich zur Definition von Listen, Bäumen oder ähnlichen Datenstrukturen verwendet werden. In diesen Fällen ist der Bezugstyp aber fast immer ein Verbund. Aus diesen Gründen ist es sinnvoller, Modula-Programme, die solche Typdeklarationen enthalten, nur dann nach C zu übersetzen, wenn der Bezugstyp ein Verbund ist.

Symbolische Konstanten werden in C mit Hilfe der *#define*-Anweisung des C-Präprozessors vereinbart. Bei der Abbildung muß berücksichtigt werden, daß der Gültigkeitsbereich eines mit *#define* vereinbarten Makros der gesamte Rest der Quelldatei ist.

Für mit *typedef* vereinbarte Typnamen gilt laut C-Sprachbeschreibung die übliche Blockstruktur von C. Das heißt eine Deklaration der Art

```
typedef char tType;
  ...
{
    typedef int tType;
```

ist eigentlich zulässig. Viele C-Übersetzer sind jedoch nicht in der Lage, eine derartige

1 Dies entspricht einer topologischen Sortierung des gerichteten Graphen, der durch die Relation „wird verwendet zur Deklaration von'' definiert wird.

Deklaration zu verarbeiten.

Die oben genannten Einschränkungen für die mehrfache Verwendung von Konstanten- und Typnamen werden bei der Übersetzung dadurch gelöst, daß die Namen von Konstanten und Typen durch eine Umbenennung innerhalb einer Quelldatei eindeutig gemacht werden.

4.4 Deklarationen

4.4.1 Konstantendeklarationen

Die Abbildung von Konstantendeklarationen ist abhängig von der Art der Konstante.

Eine Konstante, deren Wert eine Zeichenkette mit Länge n > 1 ist, wird auf einen Vektor von Zeichen abgebildet, der mit der angegebenen Zeichenkette initialisiert wird. Ist der Wert der Konstante eine Zeichenkette der Länge 1, dann wird diese Konstante im C-Programm mit einer *#define*-Anweisung als Zeichenkonstante definiert. Wird diese Konstante jedoch in einem Kontext benutzt, in dem ein Ausdruck mit Typ ARRAY [0..0] OF CHAR erwartet wird, dann muß der Wert der Konstante als Zeichenkette direkt für den Konstantennamen eingesetzt werden.

Alle übrigen im Modula-Programm vereinbarten Konstanten werden im erzeugten C-Programm mit einer *#define*-Anweisung vereinbart. Konstante Ausdrücke werden dabei auf die entsprechenden C-Ausdrücke abgebildet, wenn der Ausdruck auch in C einen konstanten Ausdruck bildet. Konstante Ausdrücke, die logische Operatoren, einen in C als Funktion definierten Operator oder eine in C als Funktion definierte Standardfunktion enthalten, sind in C nicht mehr konstant. Solche Ausdrücke werden bei der Übersetzung ausgewertet und der Wert des Ausdrucks wird für den Ausdruck eingesetzt. Diese Unterscheidung ist notwendig, damit benannte Konstanten in C für Fallmarken von *switch*-Anweisungen und zur Dimensionierung von Vektoren verwendet werden können. Die Alternative, konstante Ausdrücke immer schon bei der Übersetzung auszuwerten, vermindert die Lesbarkeit der erzeugten Programme und wurde daher verworfen. Somit werden die folgenden Deklarationen

```
CONST
  Message       = "hello world";
  BitsPerBitset = SIZE (BITSET) * 8;
  EmptySet      = {};
  MaxToken      = 63;
```

in C zu

```
CHAR Message[] = "hello world";

#define BitsPerBitset   (sizeof(BITSET) * 8)
#define EmptySet        0XL
#define MaxToken        63
```

Eine Auswertung von konstanten Ausdrücken bei der Übersetzung nach C ist in den meisten Fällen problemlos möglich. Die von MOCKA gemachte Einschränkung, daß konstante Ausdrücke keine Standardfunktionen enthalten dürfen, wird dabei fallengelassen. Schwierigkeiten bereitet aber die Auswertung der Funktionen SIZE, TSIZE, MAX und MIN, da diese sowohl von der Abbildung der Typen nach C als auch von der Typabbildung des verwendeten C-Übersetzers

abhängen. Dieses Problem wird folgendermaßen gelöst:

- Der Übersetzer enthält eine Tabelle, in der vermerkt ist, welche Größe die Grund-, Standard-, Aufzählungs-, Mengen- und Zeigertypen in C haben. Die obigen Standardfunktionen können für diese Typen mit Hilfe der Tabelle ausgewertet werden.
- Feld- oder Verbundtypen als Argumente sind nicht zulässig, da dazu die genaue Typabbildung des C-Übersetzers, insbesondere die möglicherweise notwendige Ausrichtung von Strukturkomponenten, bekannt sein müßte.

Da der C-Präprozessor Textersatz vornimmt, wird der Wert aller mit *#define* vereinbarten Konstanten im C-Programm textuell für den Konstantennamen eingesetzt. Deshalb ist es nicht sinnvoll, Zeichenketten mit *#define* zu vereinbaren, denn in einem C-Programm sind alle Zeichenketten verschieden, selbst wenn sie aus der gleichen Zeichenfolge bestehen. Das bedeutet, daß das erzeugte Objektprogramm bei mehrfacher Anwendung eines mit *#define* vereinbarten Konstantennamens die zugehörige Zeichenkette ebenfalls mehrfach enthalten würde. Die Deklaration als initialisierter Vektor vermeidet dies.

Wird der Wert der Modula-Konstante durch einen konstanten Ausdruck spezifiziert, muß bei der Abbildung nach C noch folgendes beachtet werden:

- Der Ausdruck muß, wegen des vom C-Präprozessor vorgenommenen Textersatzes, geklammert werden, um eine korrekte Auswertungsreihenfolge zu garantieren.
- Falls der Ersatztext Namen enthält, die nicht selbst als Makros vereinbart sind, muß darauf geachtet werden, daß nicht auf subtile Art und Weise im C-Programm Namenskonflikte entstehen.
- Konstante Ausdrücke werden vom C-Übersetzer bereits bei der Übersetzung des C-Programms ausgewertet.

4.4.2 Typdeklarationen

Die im Modula-Programm vereinbarten Typen werden im erzeugten C-Programm mittels einer *typedef*-Anweisung vereinbart.

4.4.2.1 Typkompatibilität

Die von MOCKA implementierten Regeln hinsichtlich Typ- und Zuweisungskompatibilität, die weniger streng sind, als die in der Sprachbeschreibung von Modula-2 enthaltenen, bilden die Grundlage für diese Abbildung. Es muß besonders darauf geachtet werden, daß Typen, die in Modula-2 typ- bzw. zuweisungskompatibel sind, dies auch im erzeugten C-Programm sind. Dabei können folgende Eigenschaften von C vorteilhaft ausgenutzt werden:

- C definiert für arithmetische Typen[1] eine große Zahl von impliziten Typumwandlungen in Ausdrücken bei der Zuweisung und der Parameterübergabe. Arithmetische Typen können

1 Zu den arithmetischen Typen gehören in C die Typen: *char*, *int*, *float* und *double*. Der Typ *int* kann mit einer Größenangabe *short* oder *long* versehen sein. Außerdem kann man *char*- und *int*-Objekte auch als *unsigned*, d.h. als ganze Zahl ohne Vorzeichen, vereinbaren.

daher in C fast in beliebiger Kombination verwendet werden.

- Von den meisten C-Übersetzern werden Zeichen und ganzzahlige Werte, unabhängig von ihrem tatsächlichen Speicherbedarf, bei der Parameterübergabe mit 4 Byte übergeben.
- Der C-Übersetzer erlaubt Zuweisungen zwischen Zeigern verschiedenen Typs. Bei der Übergabe von Zeigern an eine Funktion darf sich der Typ des aktuellen Parameters vom Typ des formalen Parameters unterscheiden. Diese Zuweisungen bzw. Parameterübergaben erfolgen als reine Kopien, ohne jede Umwandlung. Ohne eine explizite Typumwandlung wird das Prüfprogramm *lint*, mit dem C-Programme auf semantische Fehler untersucht werden können, an diesen Stellen Typfehler melden, was aber nicht unbedingt bedeuten muß, daß die Programme wirklich fehlerhaft sind.
- Der C-Übersetzer nimmt praktisch keine Typprüfungen vor.

Bei der Abbildung der nicht zusammengesetzten Typen wird darauf geachtet, daß die Größe der Typen im C-Programm wenn möglich der von MOCKA für die entsprechenden Modula-Typen vorgesehenen Größe entspricht. Mit dieser Voraussetzung und aufgrund der oben geschilderten Eigenschaften von C sind im erzeugten C-Programm nur an einigen wenigen Stellen explizite Typumwandlungen notwendig, damit die C-Programme korrekt funktionieren. Durch die Angabe einer Option bei der Übersetzung von Modula-2 nach C kann man *Mtc* allerdings dazu veranlassen, weitere explizite Typumwandlungen zu erzeugen, die für das korrekte Funktionieren der erzeugten C-Programme nicht unbedingt notwendig sind und daher normalerweise wegen der damit verbundenen schlechteren Lesbarkeit der C-Programme weggelassen werden, die aber dazu führen, daß *lint* keine Typfehler mehr meldet.

4.4.2.2 Grundtypen

Die Menge der vordefinierten Grundtypen von Modula-2, die aus den Typen INTEGER, CARDINAL, BOOLEAN, CHAR, REAL, LONGREAL und LONGINT besteht, ist entsprechend der von MOCKA implementierten Spracherweiterung um die Grundtypen SHORTINT, SHORTCARD und LONGCARD erweitert. Die Definition dieser Grundtypen ist in der Definitionsdatei SYSTEM_.h enthalten, die mit Hilfe einer *#include*-Anweisung vom C-Präprozessor in jedes erzeugte C-Programm eingefügt wird. Die folgende Aufzählung zeigt die Definition der Grundtypen[1]:

```
typedef short           SHORTINT;
typedef long            LONGINT;
typedef LONGINT         INTEGER;

typedef unsigned short  SHORTCARD;
typedef unsigned long   LONGCARD;
typedef LONGCARD        CARDINAL;

typedef unsigned char   BOOLEAN;
#define FALSE           (BOOLEAN)0
#define TRUE            (BOOLEAN)1

typedef unsigned char   CHAR;
```

1 Da MOCKA die Typen INTEGER und LONGINT sowie CARDINAL und LONGCARD als synonym vereinbart, werden sie auch bei der Abbildung nach C als synonym vereinbart.

```
typedef float           REAL;
typedef double          LONGREAL;
```

Der Typ CHAR muß, wegen der Probleme bei der impliziten Umwandlung von Zeichenwerten in ganzzahlige Werte (s. Kap. 4.2.2), in C als *unsigned char* und nicht als *char* definiert werden.

4.4.2.3 Aufzählungstypen

Die auf den ersten Blick naheliegende Abbildung von Aufzählungstypen ist, die Aufzählungstypen von Modula-2 direkt auf die Aufzählungstypen von C abzubilden. Die Deklaration

```
TYPE  Color = (red, green, blue);
```

würde damit in C zu

```
typedef enum {red, green, blue} Color;
```

Die C-Sprachbeschreibung legt fest, daß die in der Aufzählung genannten Namen als Konstanten vereinbart sind und überall dort verwendet werden können, wo in C Konstanten zulässig sind. Die Konstanten erhalten aufeinanderfolgende ganzzahlige Werte, wobei die erste Konstante der Aufzählung den Wert 0 hat. Viele C-Übersetzer verbieten jedoch, Aufzählungskonstanten z.B. in Vergleichen oder zur Indizierung von Vektoren zu verwenden. Diese Einschränkung kann allerdings umgangen werden, indem Aufzählungskonstante in diesen Fällen mittels einer expliziten Typumwandlung in einen *int*-Wert umgewandelt werden:

```
a[(int)red] = 1;
```

Eine weitere Schwierigkeit ist, daß eine Reihe von C-Übersetzern es nicht verträgt, wenn eine Aufzählungskonstante innerhalb einer Funktion lokal neu deklariert wird. Solche Aufzählungskonstanten müßten daher umbenannt werden.

Um die oben genannten Probleme zu vermeiden, wurde folgende Abbildung gewählt: der Aufzählungstyp wird entsprechend der Anzahl von Aufzählungskonstanten in einen *unsigned*-Typ geeigneter Größe umgesetzt; die Aufzählungskonstanten werden mit *#define*-Anweisungen als symbolische Konstanten vereinbart. Für die Namen der Aufzählungskonstanten gilt wie für alle mit *#define* vereinbarten Konstantennamen, daß sie bei Bedarf durch eine Umbenennung innerhalb der sie enthaltenden Quelldatei eindeutig gemacht werden müssen (s. Kap. 4.3). Die obige Deklaration wird dann zu:

```
#define red     0
#define green   1
#define blue    2
typedef unsigned char Color;
```

Der Vorteil dieser Lösung ist, daß keine Typumwandlungen mehr notwendig sind; der Nachteil eine schlechtere Lesbarkeit der Typdeklaration.

4.4.2.4 Unterbereichstypen

Unterbereichstypen werden nach C abgebildet, indem mit Hilfe einer *typedef*-Anweisung der Name des Unterbereichstyps als synonym zum Namen des Basistyps vereinbart wird. Ist der Basistyp in der Deklaration des Unterbereichstyps nicht explizit angegeben, so muß er aus dem Typ der unteren und/oder oberen Grenze des Unterbereichstyps abgeleitet werden. Sind die untere/obere Grenze ganzzahlige Konstanten, dann legt die Sprachbeschreibung von Modula-2 fest, daß der Basistyp INTEGER ist, falls die untere Grenze eine negative Zahl ist, und ansonsten CARDINAL. Somit werden die folgenden Deklarationen

```
TYPE
  TrafficLight  = [red..green];
  tToken        = [0..MaxToken];
```

in C zu

```
typedef Color     TrafficLight;
typedef CARDINAL  tToken;
```

4.4.2.5 Felder

Eine direkte Abbildung der Felder der Sprache Modula-2 auf die Vektoren der Sprache C ist, wegen der ungewöhnlichen Semantik der Vektoren, nicht sinnvoll. Vektoren können nicht als Ganzes zugewiesen werden oder als Wertparameter übergeben werden. Bei der Zuweisung und der Parameterübergabe wird ein Vektorname immer als Zeiger auf das erste Element des Vektors interpretiert. Aus diesem Grund muß in C die Zuweisung eines Vektors elementweise erfolgen. Will man den Vektor als Wertparameter übergeben, muß der übergebene Vektor am Anfang der Funktion in einen lokal vereinbarten Vektor kopiert werden. Da neuere C-Übersetzer in der Lage sind Strukturen als Ganzes zuzuweisen oder sie als Wertparameter an Funktionen zu übergeben, wird diese Schwierigkeit umgangen, indem ein Feld auf eine Struktur mit einem Vektor *A* als einziger Komponente abgebildet wird [PTC 87]. Die Anzahl der Elemente von Vektoren wird in C durch einen konstanten Ausdruck angegeben, daher muß bei der Übersetzung für den Indextyp ein konstanter Ausdruck erzeugt werden, der angibt wieviele verschiedene Werte der Indexbereich umfaßt. Die folgenden Typdeklarationen

```
TYPE
  tString       = ARRAY [0..255] OF CHAR;
  tTokenSet     = ARRAY [0..1] OF BITSET;
```

werden in C also zu

```
typedef struct S_3 {
    CHAR A[255 + 1];
} tString;

typedef struct S_4 {
    BITSET A[1 + 1];
} tTokenSet;
```

Diese Lösung hat mehrere Vorteile: Zunächst wird damit die Zuweisung von Feldern bzw. ihre Übergabe als Wertparameter erheblich effizienter, weil kein C-Code zum Kopieren mehr notwendig ist, sondern direkt der Zuweisungsoperator bzw. der Parameterübergabemechanismus von C verwendet werden kann. Der vom C-Übersetzer dafür erzeugte Code ist in der Regel effizienter, da spezielle Assembler-Befehle verwendet werden können. Der zweite Vorteil ist eine Vereinfachung des Übersetzers, weil Felder jetzt bei der Zuweisung, der Parameterübergabe und der Adreßbestimmung wie alle anderen Datentypen behandelt werden können.

4.4.2.6 Verbunde

Modula-2 verlangt die Eindeutigkeit der Komponentennamen eines Verbundes lediglich innerhalb der Definition des Verbundtyps. Insbesondere kann man in einem Deklarationsteil den gleichen Komponentennamen in der Definition von zwei verschiedenen Verbunden benutzen. Die C-Sprachbeschreibung fordert, daß die Namen aller Komponenten von zwei im gleichen Gültigkeitsbereich vereinbarten Strukturen untereinander verschieden sind[1]. Die meisten neueren C-Übersetzer sind hier allerdings großzügiger und verlangen die Eindeutigkeit nur innerhalb der Definition einer Struktur. Diese Regel wird voraussichtlich auch im neuen C-Sprachstandard enthalten sein. Daher werden Verbunde direkt, ohne Berücksichtigung von möglichen Namenskonflikten aufgrund identischer Komponentennamen, auf die Strukturen der Sprache C abgebildet. Somit wird die Typdeklaration

```
TYPE  date = RECORD
               day   : [1..31];
               month : [1..12];
               year  : CARDINAL;
             END;
```

in C zu

```
typedef struct S_5 {
    CARDINAL day;
    CARDINAL month;
    CARDINAL year;
} date;
```

Die Strukturen, auf die die Felder und Verbunde der Sprache Modula-2 abgebildet werden, erhalten immer einen Strukturnamen. Diese Strukturnamen werden in den erzeugten C-Programmen an verschiedenen Stellen benötigt, z.B. bei der Übersetzung von Vorwärtsreferenzen in Zeigerdeklarationen (s. Kap. 4.3). Wegen der getrennten Übersetzung und zur Vermeidung von

1 Ausnahme: Zwei Strukturen beginnen mit der gleichen Folge von Komponenten.

Namenskonflikten haben die Strukturnamen der in einem Definitionsmodul definierten Verbunde und Felder in C die lange Form *Modulname_nnn*[1]; alle übrigen Strukturnamen haben die kürzere Form *S_nnn*.

4.4.2.7 Verbunde mit Varianten

Die Datenstruktur Variante der Sprache C entspricht einem varianten Teil eines Verbundes in Modula-2. Eine Variante in C kann allerdings — im Gegensatz zu einer Variante in Modula-2 — nur eine einzige Komponente zur gleichen Zeit enthalten[2].

Die Typdeklaration

```
TYPE  tRecord =    RECORD
                     x ,
                     y : CHAR;
        CASE      tag0 : Color OF
        | red   : a, b : CHAR;
        | green : c, d : CHAR;
        | blue  : e, f : CHAR;
        END;
                     z : CHAR;
        CASE      tag1 : BOOLEAN OF
        | TRUE  : u, v : INTEGER;
        | FALSE : r, s : INTEGER;
        END;
      END;
```

wird in C zu

```
typedef struct S_6 {
    CHAR    x, y;
    Color   tag0;
    union {
        struct {CHAR a, b;} V_1;
        struct {CHAR c, d;} V_2;
        struct {CHAR e, f;} V_3;
    } U_1;
    CHAR    z;
    BOOLEAN tag1;
    union {
        struct {INTEGER u, v;} V_1;
        struct {INTEGER r, s;} V_2;
    } U_2;
} tRecord;
```

---

1 Besteht der Modulname nur aus einem Buchstaben, so wird das Zeichen _ verdoppelt, um theoretisch mögliche Namenskonflikte mit anderen vom Übersetzer erzeugten Bezeichnern zu verhindern.

2 Die für die jeweiligen Datenstrukturen der beiden Sprachen gebräuchlichen deutschen Begriffe sind hier leider etwas verwirrend. Eine Komponente der C-Datenstruktur Variante (Englisch: Union) entspricht einer Variante eines varianten Teils eines Modula-Verbundes.

Jeder variante Teil des Modula-Verbundes wird in C zu einer Variante. In C muß die Variante allerdings, da sie eine Komponente der umfassenden Struktur ist, einen zusätzlichen Komponentennamen *U_nnn* bekommen, wobei *nnn* angibt, um den wievielten varianten Teil des Modula-Verbundes es sich handelt. Zur Umgehung der Einschränkung, daß Varianten in C nur genau eine Komponente zur gleichen Zeit enthalten können, muß jede Variante des Modula-Verbundes in C zusätzlich in eine Struktur geklammert werden. Diese Strukturen erhalten den Komponentennamen *V_nnn*, wobei *nnn* angibt, um die wievielte Variante innerhalb eines varianten Teils es sich jeweils handelt. Bei der Übersetzung des Zugriffs auf Varianten müssen diese zusätzlichen Komponentennamen dann entsprechend berücksichtigt werden.

4.4.2.8 Mengen

Mengen werden in C einheitlich auf den Typ *unsigned long* abgebildet. Damit kann eine Menge maximal 16 oder 32 Elemente enthalten[1] je nach Größe des Typs auf der Zielmaschine. Diese Abbildung vereinfacht die Behandlung von Mengen und steigert zusätzlich die Effizienz der Mengenverarbeitung, da jetzt:

- Konstante Mengen in C als Hexadezimalkonstante oder als konstante Ausdrücke dargestellt werden können.
- Die Mengenoperationen in C weitgehend ohne zusätzliche Funktionsaufrufe mit den Operatoren zur Bitmanipulation implementiert werden können.
- Mengen als Wertparameter übergeben und als Funktionsergebnis geliefert werden können.

Der Standardtyp BITSET ist definiert als:

```
typedef unsigned long BITSET;
```

4.4.2.9 Zeiger

Zeigertypen werden direkt auf die Zeigertypen der Sprache C abgebildet. Damit wird

```
TYPE  PtrToCardinal = POINTER TO CARDINAL;
```

in C zu

```
typedef CARDINAL *PtrToCardinal;
```

Wie oben bereits erläutert, werden die Zeigerdeklarationen, in denen der Bezugstyp erst nach dem Zeigertyp vereinbart wird, auf Fälle beschränkt, in denen der Bezugstyp ein Verbund oder ein Feld ist.

---

1 MOCKA macht die gleiche Einschränkung, daher dürfte dies in unserem Fall keine wirkliche Einschränkung bedeuten.

Die Konstante NIL ist in SYSTEM_.h als

```
#define NIL     0L
```

definiert.

4.4.2.10 Prozedurtypen

Ein Prozedurtyp wird in C zu einem Zeiger auf eine Funktion mit entsprechendem Ergebnistyp. Somit wird die Typdeklaration

```
TYPE  Function = PROCEDURE (CARDINAL): CARDINAL;
```

in folgende C-Deklaration umgesetzt:

```
typedef CARDINAL (*Function)();
```

Der Standardtyp PROC ist definiert als:

```
typedef void (*PROC)();
```

4.4.3 Variablendeklarationen

Variablendeklarationen werden auf entsprechende C-Variablendeklarationen abgebildet, die aus einem Typnamen und einer Liste von Deklaratoren [Kernighan 78] bestehen. Die folgenden Variablendeklarationen (s. Beispiele Kap. 4.3 und 4.4.2)

```
VAR
  x, y  : POINTER TO INTEGER;
  i, j  : CARDINAL;
  p, q  : BOOLEAN;
  s     : BITSET;
  F     : Function;
  S     : tString;
  t     : tTree;
  w, v  : ARRAY [0..7] OF
            RECORD
              ch    : CHAR;
              count : CARDINAL;
            END;
```

werden somit in C zu

```
INTEGER     *x, *y;
CARDINAL    i, j;
BOOLEAN     p, q;
BITSET      s;
Function    F;
```

```
tString     S;
tTree       t;

struct S_7 {
    struct S_8 {
        CHAR     ch;
        CARDINAL count;
    } A[7 + 1];
} w, v;
```

4.4.4 Prozedurdeklarationen

Die Deklaration einer Prozedur oder Funktion wird in C auf eine Funktionsdeklaration abgebildet. Eine Prozedur hat als C-Funktion den Ergebnistyp *void*, womit in C angedeutet wird, daß die Funktion kein Ergebnis liefert. Sowohl Modula-2 als auch C erlauben rekursive Aufrufe von Prozeduren bzw. Funktionen. Prozedurdeklarationen können in Modula-2 beliebig geschachtelt werden; C erlaubt keine geschachtelten Funktionsdeklarationen. Aus diesem Grund muß ein Modula-Programm, das solche geschachtelten Prozedurdeklarationen enthält, bei der Übersetzung nach C derart transformiert werden, daß keine geschachtelten Prozedurdeklarationen mehr vorhanden sind. Bei dieser Transformation müssen besonders die beiden folgenden Punkte beachtet werden:

- Bei der Deklaration einer lokalen Prozedur in Modula-2 können Konstanten und/oder Typen verwendet werden, die in einer statisch umfassenden Prozedur vereinbart sind.
- Eine lokale Prozedur kann auf eine lokale Variable einer statisch umfassenden Prozedur zugreifen. Um diesen Zugriff zur Laufzeit zu ermöglichen, enthält in Modula-2 jede Prozedurschachtel einen Verweis auf die Schachtel ihres statischen Vorgängers.

Geschachtelte Prozedurdeklarationen werden daher folgendermaßen behandelt:

- Ist eine Prozedur *Q* lokal in einer Prozedur *P* deklariert, dann steht die Funktion *Q* in C vor der Funktion *P*.
- Alle lokalen Konstanten- und Typdeklarationen der Prozedur *P* werden in C zu globalen Deklarationen, die vor die Funktion *Q* plaziert werden, damit sie auch für *Q* sichtbar sind. Im Prinzip würde es genügen, nur solche lokalen Konstanten- und Typen von *P* global zu vereinbaren, die in *Q* benutzt werden[1]. Zur Vereinfachung der Transformation werden aber alle lokalen Konstanten- und Typdeklarationen aus dem Modula-Programm in C zu globalen Deklarationen.
- Um in C der Funktion *Q* den Zugriff auf lokal in *P* vereinbarte Variablen zu ermöglichen, gibt es verschiedene Möglichkeiten. Eine Alternative ist, daß die Funktion *P* die Adressen der lokalen Variablen als zusätzliche Parameter an *Q* übergibt, eine zweite Alternative verwendet globale Zeigervariablen [PTC 87], um den Zugriff zu realisieren. Diese beiden Varianten werden unten näher beschrieben und miteinander verglichen.

---

1 Dies gilt dann natürlich auch rekursiv für alle lokalen Objekte, die in der Deklaration dieser Konstanten- und Typen verwendet werden.

– Bei dieser Transformation können eventuell Namenskonflikte entstehen. Diese werden in C durch Umbenennungen gelöst.

Zur Erläuterung der obigen Transformation und der beiden Alternativen, den Zugriff auf lokale Variablen zu ermöglichen, dient der folgende Ausschnitt eines Modula-Programms:

```
PROCEDURE p;
TYPE
  tCard = CARDINAL;
VAR
  i, j  : tCard;
  PROCEDURE q;
  BEGIN
    i := 1;
  END q;
  PROCEDURE r;
  BEGIN
    j := 1;
    q;
  END r;
BEGIN
  r;
END p;
```

Die erste Alternative einer Funktion *Q*, den Zugriff auf lokal in einer Funktion *P* vereinbarte Variablen zu ermöglichen, geht von folgender Überlegung aus: Da die Prozedur *Q* im Modula-Programm lokal zur Prozedur *P* vereinbart ist, ruft die Prozedur *P* irgendwann, möglicherweise indirekt über den Aufruf weiterer lokaler Prozeduren, die Prozedur *Q* auf. Ist dies nicht der Fall, dann ist die Prozedur *Q* „nutzlos'' und kann aus dem Programm gestrichen werden. Um in C der Funktion *Q* den Zugriff auf ihre lokalen Variablen zu ermöglichen, übergibt *P* deren Adressen als zusätzliche Parameter an *Q*. Wegen des möglicherweise indirekten Aufrufs von *Q* muß zur Bestimmung der zusätzlichen Parameter der Aufrufgraph der lokalen Prozeduren betrachtet werden. Einige der zusätzlichen Parameter haben eventuell nur die Funktion, die Adressen der Variablen entlang des Aufrufgraphen weiterzureichen. Prozedurvariablen müssen dabei nicht berücksichtigt werden, da in Modula-2 nur global auf Ebene 0 vereinbarte Prozeduren als Wert von Prozedurvariablen zulässig sind.

Der folgende Datenflußalgorithmus berechnet die Mengen *Param(p)*, die angeben, um welche Parameter die Parameterlisten der lokalen Prozeduren bei der Abbildung nach C erweitert werden müssen:

```
BEGIN
  LocalProc := { p | p ist eine auf Ebene k ≥ 1 vereinbarte Prozedur }
  GlobalVar := { v | v ist eine auf Ebene 0 vereinbarte Variable }
```

```
  FORALL p ∈ LocalProc DO
    Local(p) := { v | v ist eine lokale Variable von p }
    Call(p)  := { q | q ∈ LocalProc  p ruft q auf }
    Use(p)   := { v | v wird in p benutzt  v ∉ Local(p)  v ∉ GlobalVar }
    Param(p) := Use(p)
  END
  REPEAT
    Changed := FALSE
    FORALL p ∈ LocalProc DO
      Old := Param(p)
      Param(p) ∪:= ∪ Param(q) \ Local(p)
                  q ∈ Call(p)
      Changed := Changed OR (Param(p) ≠ Old)
    END
  UNTIL NOT Changed
END
```

Die REPEAT-Schleife im obigen Algorithmus ist notwendig, weil der Aufrufgraph der lokalen Prozeduren möglicherweise aufgrund von Rekursion Zyklen enthält. Die Effizienz des Algorithmus ist stark abhängig von der Reihenfolge, in der die Prozeduren der Menge *LocalProc* bearbeitet werden. Nach Möglichkeit sollte jede Prozedur erst bearbeitet werden, wenn alle ihre Nachfolger im Aufrufgraph bereits bearbeitet wurden. Enthält der Aufrufgraph keine Zyklen, dann sind bei dieser Bearbeitungsreihenfolge höchstens zwei Iterationen notwendig. Mit diesem Algorithmus wird das obige Beispiel in C zu:

```
typedef CARDINAL tCard;

static void q(i)
tCard *i;
{
    *i = 1;
}

static void r(i, j)
tCard *i, *j;
{
    *j = 1;
    q(i);
}

void p()
{
    tCard i, j;

    r(&i, &j);
}
```

Die zweite Alternative einer Funktion *Q*, den Zugriff auf lokal in *P* vereinbarte Variablen zu ermöglichen ist, für jede dieser Variablen eine globale Zeigervariable zu vereinbaren. Am Anfang der Funktion *P* werden die Adressen dieser lokalen Variablen an die globalen Zeigervariablen zugewiesen. Damit dies auch funktioniert, wenn die Funktion *P* rekursiv aufgerufen wird, werden die ursprünglichen Werte der globalen Zeigervariablen am Anfang von *P* in lokalen

Zeigervariablen gesichert und am Ende von *P* wieder zurückgeschrieben. Die Funktion *Q* kann jetzt über die globalen Zeigervariablen auf die lokalen Variablen von *P* zugreifen. Damit wird das obige Beispiel in C zu:

```
typedef CARDINAL tCard;

static tCard *G_1_i;
static tCard *G_2_j;

static void q()
{
    *G_1_i = 1;
}

static void r()
{
    *G_2_j = 1;
    q();
}

void p()
{
    tCard i, j;
    tCard *L_1, *L_2;

    L_1 = G_1_i; G_1_i = &i;
    L_2 = G_2_j; G_2_j = &j;

    r();

    G_1_i = L_1;
    G_2_j = L_2;
}
```

Eine mögliche Optimierung wäre, zu bestimmen, ob die Funktion *P* rekursiv aufgerufen wird und nur in diesem Fall den Wert der globalen Zeigervariablen lokal in *P* zu sichern.

Beide oben vorgestellten Varianten funktionieren in Modula-2 auch im Zusammenhang mit Prozedurvariablen, da in Modula-2 als Wert von Prozedurvariablen nur global auf Ebene 0 vereinbarte Prozeduren zulässig sind. Damit entfällt die z.B. in Pascal vorhandene Schwierigkeit, beim Aufruf einer lokalen Prozedur die bei der Zuweisung an die Prozedurvariable gültige Umgebung zu bestimmen.

Ein Vergleich der beiden oben vorgestellten Alternativen ergibt folgendes:

- Bei der Übersetzung nach C ist der Aufwand für die erste Alternative, wegen der Notwendigkeit die Aufrufabhängigkeiten der lokalen Prozeduren zu analysieren, erheblich höher als der Aufwand für die zweite Alternative.
- Wird innerhalb einer Prozedur *P* eine lokale Prozedur *Q* mehrfach aufgerufen, müssen die Adressen der lokalen Variablen von *P* bei jedem Aufruf von *Q* als Parameter übergeben werden. Außerdem haben die Parameter manchmal nur die Funktion die Adressen der Variablen entlang des Aufrufgraphen weiterzureichen. Im Gegensatz dazu werden die Adressen der lokalen Variablen nur einmal an die globalen Zeigervariablen zugewiesen, jedoch verursacht die lokale Sicherung der Werte der globalen Zeigervariablen zusätzlichen Aufwand.

– Der Speicheraufwand für beide Alternativen ist in der Regel sehr gering.

Da der Berechnungsaufwand für die erste Alternative bei der Übersetzung verhältnismäßig hoch ist und der Aufwand zur Laufzeit bei der zweiten Alternative in den meisten Fällen geringer ist, wurde die zweite Alternative für den Übersetzer ausgewählt.

Ein besonderes Problem entsteht, wenn in der Deklaration einer lokalen Variablen ein anonymer strukturierter Typ verwendet wird. Übersetzt man diese Variablendeklaration wie in Kapitel 4.4.3 beschrieben nach C, dann ist der für den strukturierten Typ erzeugte Strukturnamen nur lokal in der betreffenden Funktion sichtbar und kann daher nicht in der Deklaration einer globalen Zeigervariablen verwendet werden. Aus diesem Grund wird bei der Übersetzung nach C für solche anonymen strukturierten Typen zunächst eine globale Strukturdeklaration erzeugt, die den Typ beschreibt. Der zu dieser Strukturdeklaration dazugehörige Strukturnamen kann dann in der Deklaration der globalen Zeigervariablen benutzt werden. Damit wird folgender Ausschnitt eines Modula-Programms

```
PROCEDURE p;
  VAR a : ARRAY [0..127] OF CHAR;
  PROCEDURE q;
  BEGIN
    (* Zugriff auf a *)
  END q;
  ...
```

in C zu

```
struct S_9 {
    CHAR A [127 + 1];
};
  ...
struct S_9 *G_3_a;
  ...
void p()
{
    struct S_9 a;
    struct S_9 *L_3;
  ...
```

4.4.4.1 Formale Parameter

Bei der Abbildung der formalen Parameter müssen drei Fälle unterschieden werden: offene Felder, Wertparameter und Referenzparameter. Die offenen Felder müssen bei der Abbildung getrennt behandelt werden, da C kein vergleichbares Konzept kennt und daher, insbesondere wenn das offene Feld als Wertparameter übergeben wird, in C eine Simulation des gewünschten Effekts notwendig wird. Die Abbildung von „normalen'' Parametern macht hingegen wenig Mühe.

4.4.4.1.1 Wert- und Referenzparameter

In C werden alle Parameter — mit Ausnahme der Vektoren — als Wertparameter übergeben. Daher kann die Deklaration eines Wertparameters, mit der oben beschriebenen Abbildung der Felder, direkt auf die entsprechende Parameterdeklaration in C abgebildet werden. Hierbei wird allerdings vorausgesetzt, daß der C-Übersetzer in der Lage ist auch beliebige Strukturen als Wertparameter zu übergeben.

Die semantische Bedeutung eines Referenzparameters in Modula-2 ist, daß dem Unterprogramm die Adresse des aktuellen Parameters, der in diesem Fall eine Variable sein muß, übergeben wird und das Unterprogramm dann über die Adresse direkt auf diese Variable zugreift. Da in C die Adresse jeder Variablen mit dem Operator & bestimmt werden kann, erreicht man in C den Effekt eines Referenzparameters, indem man den formalen Parameter als Zeiger vereinbart und bei der Parameterübergabe die Adresse der Variablen an das Unterprogramm übergibt. Innerhalb des Unterprogramms wird dann — genau wie in Modula-2 — über die Adresse auf diese Variable zugegriffen. Der einzige Unterschied ist, daß in C die Operatoren zur Adreßbestimmung und zur Dereferenzierung explizit aufgeschrieben werden müssen. Semantisch besteht jedoch kein Unterschied.

Der folgende Ausschnitt eines Modula-Programms, der die Verwendung von Wert- und Referenzparametern zeigt,

```
VAR  GlobalRestartSet, LocalRestartSet : tTokenSet;
  ...
PROCEDURE Union (VAR t1: tTokenSet; t2: tTokenSet);
BEGIN
  t1[0] := t1[0] + t2[0];
  t1[1] := t1[1] + t2[1];
END Union;
   ...
Union (GlobalRestartSet, LocalRestartSet);
```

wird in C zu

```
tTokenSet GlobalRestartSet, LocalRestartSet;
  ...
void Union(t1, t2)
tTokenSet *t1;
tTokenSet t2;
{
    t1->A[0] = t1->A[0] | t2.A[0];
    t1->A[1] = t1->A[1] | t2.A[1];
}
  ...
Union(&GlobalRestartSet, LocalRestartSet);
```

4.4.4.1.2 Offene Felder

Offene Felder erfordern in C eine spezielle Behandlung, die aber dadurch vereinfacht wird, daß in Modula-2 für sie nur elementweiser Zugriff und die Übergabe als aktueller Parameter an ein Unterprogramm definiert ist, dessen formaler Parameter ebenfalls ein offenes Feld ist.

Die Übergabe der offenen Felder wird in C dadurch realisiert, daß als aktueller Parameter ein Zeiger auf den Feldanfang übergeben wird. Da diese Art der Übergabe in C allgemein für die Übergabe von Vektoren benutzt wird, wird die Deklaration eines offenen Feldes in eine Parameterdeklaration der Sprache C umgesetzt, die angibt, daß der Parameter ein Vektor ist[1]. Dabei spielt es bei der Übergabe zunächst keine Rolle, ob das offene Feld im Modula-Programm als Wert- oder Referenzparameter vereinbart wurde (siehe unten).

Ein zusätzlicher Parameter enthält die aktuelle Anzahl von Feldelementen. Dieser Parameter wird benötigt, um innerhalb des Unterprogramms für ein als Wertparameter vereinbartes offenes Feld einen lokalen Vektor entsprechender Größe anlegen zu können. Außerdem ist der Parameter zur Übersetzung der Standardfunktionen HIGH und SIZE notwendig, mit denen die obere Grenze bzw. die Größe des offenen Feldes bestimmt werden kann.

Erfolgt aus einer Prozedur *Q*, die lokal zu einer Prozedur *P* deklariert ist, ein Zugriff auf ein offenes Feld *a* von *P*, dann muß auch der zusätzliche Parameter mit der aktuellen Anzahl von Elementen des Feldes *a*, genau wie die Adresse von *a* (s. Kap. 4.4.4), an eine globale Variable zugewiesen werden, da auch in *Q* ein Aufruf HIGH(*a*) bzw. SIZE(*a*) möglich ist und *Q* somit auch Zugriff auf den aktuellen Wert des zusätzlichen Parameters von P haben muß.

Beim Zugriff auf ein Element des offenen Feldes, bei der Bestimmung seiner Adresse und seiner Übergabe als aktueller Parameter an eine Prozedur muß entsprechend berücksichtigt werden, daß die Abbildung der offenen Felder sich von der Abbildung der „normalen'' Felder unterscheidet.

Die Deklaration

```
PROCEDURE Sort (VAR a : ARRAY OF CARDINAL);
```

wird in C abgebildet auf

```
void Sort(a, O_1)
CARDINAL a[]; LONGCARD O_1;
```

Der erste Parameter *a* enthält den Zeiger auf den Feldanfang, der zweite Parameter *O_1* die aktuelle Anzahl von Feldelementen.

Die Behandlung der Übergabe eines offenen Feldes als Wertparameter wird zunächst an einem Beispiel demonstriert:

---

1 Zur Umgehung der damit verbundenen Einschränkungen werden die „normalen'' Felder auf Strukturen mit einem Vektor als Komponente abgebildet (s. Kap. 4.4.2.5). Für offene Felder sind diese Einschränkungen aber unproblematisch, da für sie in Modula-2 ähnliche Einschränkungen gelten wie für die Vektoren in C.

```
PROCEDURE WriteS (s : ARRAY OF CHAR);
VAR  i : CARDINAL;
BEGIN
  FOR i := 0 TO HIGH (s) DO WriteC (s[i]); END;
END WriteS;
```

Diese Prozedur wird in C zu

```
void WriteS(s, O_2)
CHAR s[]; LONGCARD O_2;
{
    CARDINAL i;
    OPEN_ARRAY_LOCALS
    ALLOC_OPEN_ARRAYS(O_2 * sizeof(CHAR), 1)
    COPY_OPEN_ARRAY(s, O_2, CHAR)
    {
        CARDINAL B_1 = 0, B_2 = (O_2 - 1);
        if (B_1 <= B_2)
          for (i = B_1;; i += 1) {
              WriteC(s[i]);
              if (i >= B_2) break;
          }
    }
    FREE_OPEN_ARRAYS
}
```

Der zweite Parameter von *WriteS O_2* enthält wieder die aktuelle Anzahl von Feldelementen. Der erste Parameter *s* enthält beim Prozeduraufruf zunächst die Anfangsadresse des übergebenen Feldes. OPEN_ARRAY_LOCALS, ALLOC_OPEN_ARRAYS, COPY_OPEN_ARRAY und FREE_OPEN_ARRAYS sind in SYSTEM_.h definierte Makros. OPEN_ARRAY_LOCALS ist die Definition von lokalen Hilfsvariablen, die in den übrigen Makros benötigt werden. Die Aufgabe von ALLOC_OPEN_ARRAYS ist, einen Speicherblock ausreichender Größe für das als Wertparameter übergebene Feld zu beschaffen. COPY_OPEN_ARRAY bestimmt zunächst die Anfangsadresse eines lokalen Vektors geeigneter Größe und Ausrichtung im von ALLOC_OPEN_ARRAYS beschafften Speicherblock, kopiert das übergebene Feld in diesen lokalen Vektor und weist schließlich die Anfangsadresse des lokalen Vektors an den Parameter *s* zu. FREE_OPEN_ARRAYS gibt den Speicherplatz für den von ALLOC_OPEN_ARRAYS beschafften Speicherblock wieder frei. FREE_OPEN_ARRAYS muß am Ende des Unterprogramms sowie vor jeder RETURN-Anweisung innerhalb des Unterprogramms stehen. Die genaue Definition dieser Makros hängt von den auf der Zielmaschine verfügbaren Bibliotheksfunktionen ab. Eine Möglichkeit wäre folgende:

```
#define SYSTEM_ALIGN            8
#define SYSTEM_MASK             (~(SYSTEM_ALIGN - 1))
#define OPEN_ARRAY_LOCALS       char *BLOCK_POINTER, *FREE_POINTER;
```

```
#define ALLOC_OPEN_ARRAYS(size, arrays) \
    BLOCK_POINTER = FREE_POINTER = \
        malloc((unsigned)((size) + (arrays) * (SYSTEM_ALIGN - 1)));

#define FREE_OPEN_ARRAYS        free(BLOCK_POINTER);

#define COPY_OPEN_ARRAY(array, elems, type) \
{ \
    int ARRAY_SIZE = elems * sizeof(type); \
    \
    array = (type *)memcpy(FREE_POINTER, (char *)array, ARRAY_SIZE); \
    FREE_POINTER += (ARRAY_SIZE + (SYSTEM_ALIGN - 1)) & SYSTEM_MASK; \
}
```

Die Konstante SYSTEM_ALIGN ist maschinenabhängig und sorgt dafür, daß die Anfangsadressen der lokalen Vektoren korrekt ausgerichtet sind. Die Bibliotheksfunktion *malloc* beschafft Speicherplatz auf der Halde, *free* gibt diesen Speicherplatz wieder frei. Die Bibliotheksfunktion *memcpy(s1, s2, n)* kopiert *n* Zeichen von *s2* nach *s1* und liefert den Zeiger *s1* als Ergebnis.

An einigen Maschinen stehen auch Bibliotheksfunktionen zur Verfügung, mit denen man den Speicherplatz erheblich effizienter in der Aufrufschachtel des Unterprogramms beschaffen kann[1]. Bei der Rückkehr wird dieser Speicherplatz dann automatisch freigegeben. Das Makro FREE_OPEN_ARRAYS wäre in diesem Fall leer.

Da der zuletzt für die Speicherung von offenen Feldern beschaffte Speicherblock immer als erster wieder freigegeben wird, könnte man die Aufrufe von *malloc* und *free* auch durch Aufrufe von selbstprogrammierten Funktionen zur Speicherverwaltung ersetzen, die den angeforderten Speicher kellerartig verwalten und daher wahrscheinlich effizienter programmiert werden können als *malloc* und *free*, die die Anforderung und Freigabe von Speicherplatz in beliebiger Reihenfolge unterstützen.

4.4.4.2 Standardprozeduren und -funktionen

Dieser Abschnitt enthält die Definition der Standardprozeduren und -funktionen in C. Alle diese Definitionen sind in SYSTEM_.h bzw. SYSTEM_.c enthalten.

Da ABS eine überladene Funktion ist, existieren in C verschiedene Definitionen von ABS. Abhängig vom Typ und der Art des aktuellen Parameters, wird in C ein Aufruf der entsprechenden Funktion bzw. des entsprechenden Makros eingesetzt. Die verschiedenen Definitionen von ABS haben folgende Form:

```
#define ABS(x)          ((x) < 0 ? -(x) : (x))
#define ABSSC(x)        ((SHORTCARD) (x))
#define ABSLC(x)        ((LONGCARD) (x))
#define ABSSI(x)        ((SHORTINT) ABSLI((LONGINT) (x)))
#define ABSR(x)         ((REAL) ABSLR(x))

extern LONGINT ABSLI();
extern LONGREAL ABSLR();
```

1 An der SUN z.B. existiert eine solche Bibliotheksfunktion mit dem Namen *alloca*.

```
LONGINT ABSLI(x)
register LONGINT x;
{
    return (x < 0 ? -x : x);
}
LONGREAL ABSLR(x)
register LONGREAL x;
{
    return (x < 0 ? -x : x);
}
```

Das Makro ABS, das nur benutzt wird, wenn der Parameter *x* ein konstanter Ausdruck ist, kann nicht in allen Fällen verwendet werden, da *x* möglicherweise ein Funktionsaufruf mit Seiteneffekten ist und in diesem Fall die doppelte Auswertung von *x* zu Fehlern führen würde.

Die Funktion CAP ist definiert als:

```
extern CHAR CAP();
CHAR CAP(ch)
register CHAR ch;
{
    return (ch >= 'a' && ch <= 'z' ? ch - 'a' + 'A' : ch);
}
```

CAP ist nicht als Makro definiert, um eine mehrfache Auswertung von *ch* zu verhindern.

Die Funktionen für Typumwandlungen CHR, FLOAT, ORD, TRUNC und VAL sind wie folgt definiert:

```
#define CHR(x)          ((CHAR) (x))
#define FLOAT(x)        ((REAL) (x))
#define ORD(x)          ((CARDINAL) (x))
#define TRUNC(x)        ((CARDINAL) (x))
#define VAL(T,x)        ((T) (x))
```

Die Umwandlung von Gleitkommawerten in ganzzahlige Werte ist in C nicht exakt definiert. In der Regel wird aber mit der obigen Definition von TRUNC der gewünschte Effekt erzielt. Es ist jedoch nicht völlig ausgeschlossen, daß in manchen Implementierungen gerundet wird.

Die Standardfunktion SIZE, zur Bestimmung der Anzahl der von einer Variablen oder einem Typ benötigten Speichereinheiten, wird normalerweise direkt auf den C-Operator *sizeof* abgebildet. Ist der aktuelle Parameter von SIZE jedoch ein offenes Feld, dann ergibt sich die von diesem Feld benötigte Anzahl von Speichereinheiten aus: *sizeof(Elementtyp) * aktueller Anzahl von Feldelementen*.

Die Behandlung der Funktionen MAX und MIN hängt von der Art des an sie als Argument übergebenen Typs ab. Für einen Aufzählungs- oder einen Unterbereichstyp *T* ist der Wert von MIN(*T*) bzw. MAX(*T*) aus der Typdefinition bekannt und wird daher textuell in das erzeugte C-Programm eingesetzt. Für die vordefinierten Standardtypen, auf die diese Funktionen anwendbar sind, sind in SYSTEM_.h Definitionen von symbolischen Konstanten der Form

```
#define MIN_CARDINAL    ...
#define MAX_CARDINAL    ...
#define MIN_BOOLEAN     FALSE
#define MAX_BOOLEAN     TRUE
  ...
```

enthalten. Diese symbolischen Konstanten werden im C-Programm für die Aufrufe von MIN bzw. MAX eingesetzt.

Ist der aktuelle Parameter der Funktion HIGH kein offenes Feld, dann ist der Wert von HIGH(*a*) bei der Übersetzung bekannt und wird in das C-Programm als Konstante eingesetzt. Ist der aktuelle Parameter jedoch ein offenes Feld, dann wird HIGH(*a*) im C-Programm zum Ausdruck (*O_nnn* - 1). Der Parameter *O_nnn* der Funktion, die *a* als formalen Parameter hat, gibt — wie oben erwähnt — die aktuelle Anzahl der Feldelemente des offenen Feldes an.

Die Standarfunktion ODD ist folgendermaßen definiert:

```
#define ODD(x)          ((BOOLEAN)((x) & 01))
```

Die Standardprozeduren DEC, INC, EXCL und INCL haben folgende Definition:

```
#define DEC(x)          (x)--
#define DEC1(x,n)       x -= n
#define INC(x)          (x)++
#define INC1(x,n)       x += n
#define EXCL(s,i)       s &= ~(0X1L << (i))
#define INCL(s,i)       s |= 0X1L << (i)
```

Ein Aufruf der Standardprozedur HALT wird in C zu einem Aufruf der Bibliotheksfunktion *exit*. Da HALT in der Regel in Fehlerfällen benutzt wird, wird *exit*, entsprechend der in C üblichen Konvention, mit dem Argument 1 aufgerufen, um der Umgebung den Fehler anzuzeigen.

Modula-2 definiert, daß Aufrufe NEW(*p*) bzw. DISPOSE(*p*) der Standardprozeduren NEW bzw. DISPOSE ersetzt werden durch Aufrufe ALLOCATE(*p*, TSIZE(*T*)) bzw. DEALLOCATE(*p*, TSIZE(*T*)), wobei *p* vom Typ POINTER TO *T* sein muß und ALLOCATE bzw. DEALLOCATE vom Benutzer definierte und an der Aufrufstelle sichtbare Prozeduren mit folgender Definition sein müssen:

```
PROCEDURE ALLOCATE      (VAR a: ADDRESS; size: CARDINAL);
PROCEDURE DEALLOCATE    (VAR a: ADDRESS; size: CARDINAL);
```

Daher wird folgendes Modula-Programmfragment

```
FROM Storage    IMPORT ALLOCATE, DEALLOCATE;
  ...
VAR p: POINTER TO INTEGER;
  ...
NEW (p); p^ := 1; DISPOSE (p);
```

in C zu

```
  ...
INTEGER *p;
  ...
Storage_ALLOCATE(&p, sizeof(INTEGER));
*p = 1;
Storage_DEALLOCATE(&p, sizeof(INTEGER));
```

4.4.5 Lokale Moduln

C kennt kein den lokalen Moduln entsprechendes Konzept. Daher muß ein Modula-Programm, das lokale Moduln enthält derart transformiert werden, daß keine lokalen Moduln mehr vorhanden sind. Dies wird mit folgender Transformation erreicht:

- Alle Objekte, die in einem lokalen Modul deklariert sind, werden zu lokalen Objekten der Prozedur bzw. Übersetzungseinheit, die das lokale Modul in ihrem Deklarationsteil enthält.
- Der Rumpf des lokalen Moduls wird zu einer parameterlosen Prozedur. Diese Prozedur wird am Anfang des Anweisungsteils der Prozedur oder Übersetzungseinheit aufgerufen, die den lokalen Modul in ihrem Deklarationsteil enthält. Sind mehrere lokale Moduln vorhanden, dann erfolgen diese Aufrufe entsprechend der Reihenfolge der lokalen Moduln im Quelltext. Dies entspricht der in der Sprachdefinition von Modula-2 geforderten Reihenfolge.
- Da lokale Moduln mit Hilfe von Import- bzw. Exportanweisungen eine explizite Kontrolle der Sichtbarkeit von Bezeichnern erlauben, können bei dieser Transformation möglicherweise Namenskonflikte entstehen. Diese werden in C durch eine Umbenennung von Bezeichnern gelöst.

Diese Transformation muß durchgeführt werden, bevor die Reihenfolge der Deklarationen umgeordnet wird und bevor geschachtelte Prozedurdeklarationen transformiert werden.

Die Übersetzung des Moduls *Global* nach C soll die oben beschriebene Transformation nochmals verdeutlichen:

```
MODULE Global;

VAR  a : INTEGER;

MODULE Local;
EXPORT QUALIFIED a;
VAR
  a : INTEGER;
BEGIN
  a := 1;
END Local;

BEGIN
  a := Local.a;
END Global.
```

In C wird *Global* zu:

```
#include "SYSTEM_.h"

static INTEGER a;
static INTEGER C_1_a;

static void Local()
{
    C_1_a = 1;
}

void Global()
{
    Local();
    a = C_1_a;
}
```

4.5 Ausdrücke

Bei der Übersetzung von Ausdrücken muß darauf geachtet werden, daß die Auswertungsreihenfolge im erzeugten C-Programm der Auswertungsreihenfolge im Modula-Programm entspricht. Dabei ist insbesondere von Bedeutung, daß sich der Vorrang der Operatoren in Modula-2 und C teilweise deutlich unterscheidet. Eine korrekte Auswertungsreihenfolge muß daher bei Bedarf durch das Setzen von Klammern erzwungen werden.

4.5.1 Operanden

Die Abbildung von Zeichenketten und Zahlen wurde bereits in Kapitel 4.2 besprochen.

Mengen werden in einen C-Ausdruck umgesetzt, der die gewünschte Menge liefert[1]. Die folgende Menge

```
{1..2, i..j, j + 2}
```

wird in C zu

```
SET_cRNG(1, 2) | SET_RANGE(i, j) | SET_ELEM(j + 2)
```

SET_ELEM, SET_cRNG und SET_RANGE sind in SYSTEM_.h bzw. SYSTEM_.c folgendermaßen definiert:

```
#define SYSTEM_MaxSet   (sizeof(unsigned long) * 8 - 1)

#define SET_ELEM(el)    (0X1L << (el))
#define SET_cRNG(lo,hi) \
    ((lo) <= (hi) ? ~0XL >> (lo) << (lo) + SYSTEM_MaxSet - (hi) \
     >> SYSTEM_MaxSet - (hi) : 0XL)
```

1 So definierte Mengenkonstanten und konstante Elemente werden vom C-Übersetzer bereits bei der Übersetzung ausgewertet.

```
#define SET_RANGE(lo, hi) \
    SET_RANGE1((CARDINAL)(lo), (CARDINAL)(hi))

extern unsigned long SET_RANGE1();

unsigned long SET_RANGE1(lo, hi)
register CARDINAL lo, hi;
{
    return (lo <= hi ? ~0XL >> lo << lo + SYSTEM_MaxSet - hi
            >> SYSTEM_MaxSet - hi : 0XL);
}
```

Die Funktion SET_RANGE1 wird für nicht konstante Elemente verwendet, um eine mehrfache Auswertung von *lo* und *hi* zu verhindern.

Alle von globalen Moduln exportierten Bezeichner werden im C-Programm in der qualifizierten Form *Modulname_Bezeichner* geschrieben. Nicht von globalen Moduln exportierte Bezeichner werden unverändert nach C übernommen, sind aber bei Bedarf zur Lösung eines Namenskonfliktes mit einem Präfix versehen.

Die Selektion einer Verbundkomponente

```
R.f
```

wird in C zu

```
R.f
```

wenn die Komponente nicht in einem varianten Teil des Verbundes enthalten ist. Ist die Komponente in einem varianten Teil enthalten, so erfolgt die Selektion entsprechend der oben vorgestellten Abbildung von varianten Verbunden mit:

```
R.U_xxx.V_nnn.f
```

Der Zugriff auf ein Feldelement

```
A[E]
```

wird entsprechend der oben beschriebenen Abbildung von Feldern in C zu

```
A.A[E - Lwb]
```

wenn *A* kein offenes Feld ist. Da in C alle Vektoren die untere Grenze 0 besitzen, muß beim Zugriff auf ein Feldelement die in Modula-2 angegebene untere Grenze *Lwb* des Feldes abgezogen werden, sofern diese ungleich 0 ist. Ist *A* ein offenes Feld, so erfolgt der Zugriff entsprechend der oben vorgestellten Abbildung von offenen Feldern auf Vektoren mit:

```
A[E]
```

Die Dereferenzierung von Zeigervariablen

```
P^
```

wird in C zu

```
*P
```

Eine Dereferenzierung ist, wegen der oben beschriebenen Abbildung von Referenzparametern auf Zeiger, auch notwendig, wenn *P* ein Referenzparameter ist.

Der Funktionsaufruf

```
F()
```

wird in C zu

```
F()
```

wenn *F* keine Prozedurvariable ist. Ist *F* eine Prozedurvariable, so ist, da Prozedurtypen in C als Zeiger auf Funktionen repräsentiert werden, eine zusätzliche Dereferenzierung notwendig:

```
(*F)()
```

Die Klammerung von **F* ist notwendig, weil in C der Vorrang von () größer ist als der Vorrang von *.

Eine Dereferenzierung gefolgt von der Selektion einer Verbundkomponente wird auf den in C speziell für diese Operationsfolge vorgesehenen Operator –> abgebildet.

4.5.2 Operatoren

Die arithmetischen Operatoren werden entsprechend der folgenden Tabelle nach C abgebildet:

Modula-2	C
+	+
–	–
*	*
/	/
DIV	/
MOD	%

Tabelle 4.2: Abbildung der arithmetischen Operatoren

Im Gegensatz zu Modula-2 ist allerdings in der C-Sprachbeschreibung nicht eindeutig definiert, welches Ergebnis die Operatoren / und % liefern, falls einer der Operanden eine negative Zahl ist. Bei der Abbildung werden aber solche möglichen Unterschiede in der Arithmetik nicht berücksichtigt.

Da sowohl Modula-2 als auch C die Kurzauswertung logischer Ausdrücke verlangen, können die logischen Operatoren NOT, AND und OR direkt auf die C-Operatoren !, && und || abgebildet werden.

Mit der vereinfachenden Annahme, daß alle Mengen nur Wortgröße (*unsigned long*) haben, können die Mengenoperatoren in C leicht mit Hilfe der Operatoren zur Bitmanipulation realisiert werden. Die folgende Tabelle zeigt die Übersetzung der Mengenoperatoren:

Modula-2	C
+	|
-	SET_DIFF
*	&
/	^
=	==
#	!=
<=	SET_IS_SUBSET1
>=	SET_IS_SUBSET2
IN	IN

Tabelle 4.3: Abbildung der Mengenoperatoren

Die Makros SET_DIFF, IN, SET_IS_SUBSET1 und SET_IS_SUBSET2 sind in SYSTEM_.h wie folgt definiert:

```
#define SET_DIFF(s1,s2)          ((s1) & ~(s2))
#define IN(x,s)                  ((BOOLEAN)((s) >> (x) & 0X1L))
#define SET_IS_SUBSET1(s1,s2)    ((BOOLEAN)!((s1) & ~(s2)))
#define SET_IS_SUBSET2(s1,s2)    ((BOOLEAN)!((s2) & ~(s1)))
```

Aufgrund der oben angegebenen Abbildung der Modula-Typen nach C, können die relationalen Operatoren von Modula-2 immer direkt auf die relationalen Operatoren von C abgebildet werden.

Die folgenden Ausdrücke (s. Beispiele Kap. 4.4.3)

```
i DIV 3
NOT p OR q
(i + j) * (i - j)
s - {8, 9, 13}
(1 <= i) AND (i < 100)
t^.Key = 0
{13..15} <= s
i IN {0, 5..8, 15}
```

werden in C zu

```
i / 3
!p || q
(i + j) * (i - j)
SET_DIFF(s, SET_ELEM(8) | SET_ELEM(9) | SET_ELEM(13))
1 <= i && i < 100
```

```
t->Key == 0
SET_IS_SUBSET1(SET_cRNG(13, 15), s)
IN(i, SET_ELEM(0) | SET_cRNG (5, 8) | SET_ELEM(15))
```

4.6 Anweisungen

4.6.1 Zuweisung

Aufgrund der Abbildung der Typen nach C kann — mit einer Ausnahme — die Zuweisung direkt auf den C-Zuweisungsoperator = abgebildet werden. Die einzige Ausnahme ist die Zuweisung von Zeichenketten, die mit der Bibliotheksfunktion *strncpy* durchgeführt wird. Die Funktion *strncpy(s, t, n)* kopiert eine Zeichenkette, die — wie in C üblich — durch ein NUL-Zeichen beendet wird von *t* nach *s*. Es werden jedoch maximal die als drittes Argument *n* übergebene Anzahl von Zeichen kopiert. Damit kann gerade die in Modula-2 für die Zuweisung von Zeichenketten geltende Regel „Zeichenketten der Länge *n1* können auch an Felder von Zeichen der Länge *n2 > n1* zugewiesen werden. Der Wert der Zeichenkette wird in diesem Fall um ein NUL-Zeichen verlängert.'' implementiert werden. Die folgenden Zuweisungen (s. Beispiele Kap. 4.4.3)

```
w := v;
F := log2;
s := {2, 3, 5..7, 11};
t^.Key := j;
w[i+1].ch := "a";
S := "hello world";
```

werden somit in C zu

```
w = v;
F = log2;
s = SET_ELEM(2) | SET_ELEM(3) | SET_cRNG(5, 7) | SET_ELEM(11);
t->Key = j;
w.A[i+1].ch = 'a';
(void)strncpy(S.A, "hello world", sizeof(S.A));
```

Es sei an dieser Stelle noch einmal ausdrücklich darauf hingewiesen, daß auch das Feld *v*, wegen der Abbildung von Feldern auf Strukturen mit einem Vektor als Komponente, an das Feld *w* als Ganzes zugewiesen werden kann. Wie bereits erwähnt, wird dabei allerdings vorausgesetzt, daß der C-Übersetzer Strukturen als Ganzes zuweisen kann.

Der Name einer Funktion wird in C, wenn er nicht zum Aufruf der Funktion verwendet wird, als Zeiger auf die Funktion verstanden.

4.6.2 Prozeduraufruf

Der Aufruf einer Prozedur wird in einen Aufruf der entsprechenden C-Funktion übersetzt. Handelt es sich beim Bezeichner der Prozedur um eine Prozedurvariable, dann ist, wegen der Abbildung der Prozedurtypen auf Zeiger auf Funktionen, eine Dereferenzierung beim Aufruf notwendig.

Die Abbildung der aktuellen Parameter ist abhängig von der Art der formalen Parameter:

Ist der formale Parameter ein Wertparameter (kein offenes Feld), dann wird der aktuelle Parameter in den entsprechenden C-Ausdruck übersetzt. Die Übergabe von Zeichenketten macht dabei aber aus zwei Gründen Schwierigkeiten:

- Zeichenketten werden in C auf Zeichenketten oder einen initialisierten Vektor von Zeichen abgebildet. Felder von Zeichen werden auf eine Struktur mit einem Vektor von Zeichen als Komponente abgebildet. Zeichenketten bzw. Vektoren werden in C aber im Gegensatz zu Strukturen nicht als Ganzes an eine Funktion übergeben, sondern es wird nur ein Zeiger auf den Anfang der Zeichenkette bzw. des Vektors übergeben.
- Bei der Parameterübergabe gelten die gleichen Regeln wie bei der Zuweisung. Das bedeutet, daß die Länge der Zeichenkette kleiner sein darf, als die Länge des als formaler Parameter angegebenen Feldes.

Diese beiden Probleme können gelöst werden, indem die Zeichenkette vor dem Aufruf der Funktion an eine temporäre Feldvariable zugewiesen und dann diese Feldvariable als aktueller Parameter übergeben wird. Die Lösung ist allerdings wegen der zusätzlichen Zuweisung nicht besonders effizient. Der Aufruf der Prozedur *WriteString* aus dem folgenden Ausschnitt eines Modula-Programms

```
PROCEDURE WriteString (s : tString);
  ...
WriteString ("hello world");
```

wird in C dann zu

```
tString X_1;
  ...
(void)strncpy(X_1.A, "hello world", sizeof(X_1.A));
  ...
WriteString(X_1);
```

Ist der formale Parameter ein Referenzparameter (kein offenes Feld), dann muß der aktuelle Parameter eine Variable sein. Da Referenzparameter in C auf Zeiger abgebildet werden, muß die Adresse dieser Variablen in C bei der Parameterübergabe mit dem Operator & bestimmt und übergeben werden.

Ist der formale Parameter ein offenes Feld, dann muß in C die Adresse des aktuellen Parameters übergeben werden. Dies gilt auch dann, wenn das als formaler Parameter angegebene offene Feld ein Wertparameter ist (vgl. Kap. 4.4.4.1.2). Dabei müssen, entsprechend der Art des aktuellen Parameters, folgende Fälle unterschieden werden:

- Wenn der aktuelle Parameter eine Zeichenkette ist, dann wird in C automatisch ein Zeiger auf das erste Zeichen dieser Zeichenkette übergeben.
- Ist der aktuelle Parameter ein offenes Feld, dann wird in C wegen der Abbildung der offenen Felder auf Vektoren automatisch ein Zeiger auf den Feldanfang übergeben.
- Falls der aktuelle Parameter in C weder eine Zeichenkette noch ein Vektor ist, dann muß die Adresse des aktuellen Parameters wie üblich mit dem Operator & bestimmt werden.

Zusätzlich zur Adresse des Feldes muß in C ein Parameter mit der Anzahl von Feldelementen übergeben werden. Falls der aktuelle Parameter ebenfalls ein offenes Feld ist, wird der zu diesem Feld dazugehörige Parameter mit der Anzahl von Feldelementen übergeben. Sonst ist die Anzahl von Feldelementen bei der Übersetzung bekannt und wird als Konstante in das C-Programm eingesetzt.

MOCKA erlaubt — im Gegensatz zur Sprachbeschreibung von Modula-2 — auch Zeichenketten als aktuelle Parameter, wenn das offene Feld ein Referenzparameter ist. Mit der oben beschriebenen Abbildung wird auch dieser Fall abgedeckt.

Die Übersetzung des folgenden Ausschnitts eines Modula-Programms demonstriert die Übergabe von offenen Feldern als Parameter noch einmal (s. Beispiele Kap. 4.4.3):

```
PROCEDURE p (VAR s1: ARRAY OF CHAR; s2: ARRAY OF CHAR);
BEGIN
  q (s1, s2);
END p;
  ...
p (S, "hello world");
```

Dieser Ausschnitt wird in C zu

```
void p(s1, O_3, s2, O_4)
CHAR s1[]; LONGCARD O_3;
CHAR s2[]; LONGCARD O_4;
{
    OPEN_ARRAY_LOCALS
    ALLOC_OPEN_ARRAYS(O_4 * sizeof(CHAR), 1);
    COPY_OPEN_ARRAY(s2, O_4, CHAR);
    q(s1, O_3, s2, O_4);
    FREE_OPEN_ARRAYS
}
  ...
p(S.A, 256L, "hello world", 11L);
```

4.6.3 IF-Anweisung

Die *if*-Anweisung in C entspricht genau der IF-Anweisung in Modula-2, wenn man bedenkt, daß der ELSIF-Teil von Modula-2 nur eine abkürzende Schreibweise für Ketten von Bedingungen ist. Damit wird

```
IF (ch >= "A") AND (ch <= "Z") THEN
  ReadIdentifier;
ELSIF (ch >= "0") AND (ch <= "9") THEN
  ReadNumber;
ELSE
  SpecialCharacter;
END
```

in C zu

```
if (ch >= 'A' && ch <= 'Z') {
    ReadIdentifier();
} else if (ch >= '0' && ch <= '9') {
    ReadNumber();
} else {
    SpecialCharacter();
}
```

4.6.4 CASE-Anweisung

Die CASE-Anweisung wird in eine *switch*-Anweisung übersetzt. Alle in Modula-2 für die Fallmarken zulässigen Typen sind, mit der oben beschriebenen Abbildung der Typen nach C, auch in C für den *switch*-Ausdruck und die Fallmarken zulässig. Die Listen mit Fallmarken und Bereichsangaben werden in eine Folge von C-Fallmarken der Form *case constant :* umgesetzt. Konstante Ausdrücke, die in Modula-2 als Fallmarken zulässig sind, müssen dabei bereits während der Übersetzung nach C ausgewertet werden, wenn der entsprechende C-Ausdruck kein konstanter Ausdruck ist. Die *default*-Marke der *switch*-Anweisung entspricht dem ELSE-Teil der CASE-Anweisung.

Die Anweisung

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

wird in C zu

```
switch (ch) {
case 'A' :
case 'B' :
  ...
case 'Z' :
    ReadIdentifier();
    break;
case '0' :
  ...
```

```
case '9' :
    ReadNumber();
    break;
default :
    SpecialCharacter();
    break;
}
```

In C muß nach jeder Anweisungsfolge, die zu einer Folge von Fallmarken gehört, eine *break*-Anweisung stehen, damit die *switch*-Anweisung verlassen wird.

4.6.5 Schleifen

Die WHILE-, REPEAT- und LOOP-Schleife werden folgendermaßen nach C übersetzt:

```
while (...) {          do {                    for (;;) {
  ...                    ...                     ...
}                      } while (!(...))        } EXIT_xxx:;
```

Die Marke *EXIT_xxx* dient als Sprungziel für die *goto*-Anweisungen, in welche die EXIT-Anweisungen innerhalb der LOOP-Schleife übersetzt werden.

Eine direkte Übersetzung der FOR-Schleife

```
FOR v := a TO b BY c DO ... END;
```

in die *for*-Schleife der Sprache C

```
for (v = a; v <= b; v += c) { ... };    /* falls c >= 0 */
for (v = a; v >= b; v += c) { ... };    /* falls c <  0 */
```

ist wegen der unterschiedlichen Semantik der beiden Schleifenkonstrukte in den meisten Fällen nicht möglich. Modula-2 verlangt, daß Anfangs- und Endwert der Schleife genau einmal vor dem ersten Schleifendurchlauf ausgewertet werden. Die Schrittweite c muß ein konstanter Ausdruck sein. In C wird der zweite Ausdruck der *for*-Schleife vor jedem Schleifendurchlauf ausgewertet. Ist außerdem der Typ T der Schleifenkontrollvariablen v ein arithmetischer Typ und gilt für den Endwert der Schleife b, daß $b \geq \mathrm{MAX}(T) - c + 1$ (falls $c > 0$) bzw. $b \leq \mathrm{MIN}(T) - c - 1$ (falls $c < 0$), dann terminiert die Schleife in C nicht, da die Schleifenkontrollvariable am Schleifenende vor der erneuten Auswertung der Abbruchbedingung inkrementiert wird und ein unbemerkter Über- bzw. Unterlauf stattfindet.

Daher wird die obige FOR-Schleife bei positiver Schrittweite c in die folgende zusammengesetzte C-Anweisung übersetzt:

```
{
    T B_xxx = a, B_nnn = b;
```

```
    if (B_xxx <= B_nnn)
      for (v = B_xxx, B_nnn = FOR_LIMIT_UP(B_nnn, c, MIN(T));; v += c) {
          ...
          if (v >= B_nnn) break;
      }
}
```

Bei negativer Schrittweite *c* wird die FOR-Schleife in C zu:

```
{
    T B_xxx = a, B_nnn = b;
    if (B_xxx >= B_nnn)
      for (v = B_xxx, B_nnn = FOR_LIMIT_DOWN(B_nnn, c, MAX(T));; v += c) {
          ...
          if (v <= B_nnn) break;
      }
}
```

Diese Übersetzung der FOR-Schleife vermeidet die oben angesprochenen Schwierigkeiten, da Anfangs- und Endwert der Schleife nur einmal ausgewertet werden und die Abbruchbedingung am Schleifenende geprüft wird, bevor die Schleifenkontrollvariable inkrementiert wird. Durch den Aufruf der in SYSTEM_.h definierten Makros FOR_LIMIT_UP und FOR_LIMIT_DOWN wird der Schleifenendwert vor der ersten Ausführung des Schleifenrumpfes geeignet korrigiert, um einen Über- bzw. Unterlauf der Schleifenkontrollvariablen zu verhindern. Die beiden Makros haben folgende Definition:

```
#define FOR_LIMIT_UP(last, step, min) \
    ((last) < (min) + ((step) - 1) ? (min) : (last) - ((step) - 1))
#define FOR_LIMIT_DOWN(last, step, max) \
    ((last) > (max) + ((step) + 1) ? (max) : (last) - ((step) + 1))
```

Ist der Wert der Schrittweite $c \in \{-1, 0, 1\}$, dann ist die Korrektur des Schleifenendwertes überflüssig und wird weggelassen.

Wegen der damit verbundenen besseren Lesbarkeit der C-Programme, wird die FOR-Schleife in die „normale'' *for*-Schleife der Sprache C übersetzt, wenn der Endwert der Schleife eine Konstante ist und ein Über- bzw. Unterlauf der Schleifenkontrollvariablen nicht auftreten kann.

4.6.6 WITH-Anweisung

Die WITH-Anweisung hat kein direktes Gegenstück in C. Sie wird in eine zusammengesetzte Anweisung übersetzt, die am Anfang die Deklaration und Initialisierung einer Zeigervariablen enthält. Diese Zeigervariable wird mit der Adresse der Struktur initialisiert, die dem in der WITH-Anweisung angesprochenen Verbund des Modula-Programms entspricht. Innerhalb der WITH-Anweisung wird in C dann über diese Zeigervariable auf die Strukturkomponenten zugegriffen. Diese Übersetzung erfüllt die Forderung der Sprachdefinition von Modula-2, daß der Selektor, welcher den in der WITH-Anweisung angesprochenen Verbund festlegt, nur einmal ausgewertet wird. Damit wird die folgende Anweisung

```
WITH t^ DO
  Key := 0; Left := NIL; Right := NIL;
END
```

in C zu

```
{
    register tNode *W_1 = t;
    W_1->Key = 0; W_1->Left = NIL; W_1->Right = NIL;
}
```

4.6.7 RETURN-Anweisung

Die *return*-Anweisung in C entspricht genau der RETURN-Anweisung in Modula-2, wobei allerdings die — aus der Sprachdefinition von Modula-2 nicht eindeutig hervorgehende — Einschränkung gemacht wird, daß der Ergebnistyp einer Funktion kein Feld- oder Verbundtyp sein darf[1].

Wie oben erwähnt müssen vor jeder RETURN-Anweisung die vor dem Aufruf der betreffenden Prozedur bzw. Funktion *P* gültigen Werte der globalen Zeigervariablen, die zur Übersetzung des Zugriffs auf lokale Variablen von *P* benötigt werden, wiederhergestellt werden. Außerdem muß der Speicherplatz wieder freigegeben werden, der für die als Wertparameter übergebenen offenen Felder angefordert wurde. Enthält bei einer Funktion der RETURN-Ausdruck aber einen Zugriff auf ein offenes Feld oder den Aufruf einer (lokalen) Funktion, dann dürfte das Zurückschreiben der alten Werte der globalen Zeigervariablen und die Freigabe des Speichers eigentlich erst nach der RETURN-Anweisung erfolgen, was nicht möglich ist. In diesem Fall wird daher die RETURN-Anweisung zu einer zusammengesetzten Anweisung, die am Anfang die Deklaration einer lokalen Variablen enthält, welche mit dem RETURN-Ausdruck initialisiert wird. Danach erst werden die alten Werte der globalen Zeigervariablen zurückgeschrieben, der Speicher für offene Felder freigegeben und der Wert des RETURN-Ausdrucks an die aufrufende Prozedur zurückgeliefert. Damit wird der folgende Ausschnitt eines Modula-Prgramms

```
PROCEDURE p (a : ARRAY OF CHAR): CHAR;
BEGIN
  ...
  RETURN a[0];
END p;
```

in C zu

```
CHAR p(a, O_5)
CHAR a[]; LONGCARD O_5;
{
    ...
```

1 MOCKA macht diese Einschränkung auch.

```
    {
        CHAR R_1 = a[0];
        FREE_OPEN_ARRAYS
        RETURN R_1;
    }
}
```

4.6.8 Laufzeitprüfungen

Durch die Angabe einer Option bei der Übersetzung kann man *Mtc* dazu veranlassen, Laufzeitprüfungen zu erzeugen, die die Abgabe einer entsprechenden Fehlermeldung mit Angabe der Position im C-Quellprogramm und den Abbruch des C-Programms zur Folge haben, falls in einer CASE-Anweisung durch den Selektorausdruck keine Fallmarke ausgewählt wird und der ELSE-Teil fehlt oder falls eine Funktion kein Ergebnis liefert.

4.7 Modul SYSTEM

C ist eine relativ „maschinennahe'' Sprache. Daher steht ein großer Teil der in Modula-2 vom Modul SYSTEM zur Verfügung gestellten Hilfsmittel für die maschinennahe Programmierung auch in C zur Verfügung.

Da C allerdings keine Koroutinen kennt, müßte man diese in C auf geeignete Weise simulieren. Da aber eine Simulation sowohl aufwendig als auch ineffizient wäre und außerdem die Verwendung von Koroutinen, wenn es sich nicht um spezielle Systemprogramme handelt, relativ selten ist, unterstützt der Übersetzer die Abbildung von NEWPROCESS, TRANSFER und IOTRANSFER nicht.

Die vom Übersetzer unterstützten Objekte aus dem Modul SYSTEM sind: die Typen WORD und ADDRESS, die Funktionen ADR und TSIZE sowie die Funktionen für den Typtransfer. Außerdem wird noch die von MOCKA implementierte Spracherweiterung um den Typ BYTE, der synonym zum Typ WORD ist, berücksichtigt.

Eine Typtransferfunktion wird in C als eine explizite Umwandlung in den angegebenen Typ realisiert, die dadurch ausgedrückt wird, daß dem betreffenden Wert der Name des gewünschten Datentyps in Klammern vorangestellt wird.

Die Typen BYTE und WORD sind in SYSTEM_.h folgendermaßen definiert:

```
typedef unsigned char WORD;
typedef WORD          BYTE;
```

Damit die Übergabe eines aktuellen Parameters an eine Prozedur, in welcher der entsprechende formale Parameter den Typ WORD bzw. BYTE hat, auch in C korrekt funktioniert, müssen alle Typen, die bei der Typabbildung von MOCKA die gleiche Größe wie WORD bzw. BYTE zugewiesen bekommen, auch in C die gleiche Größe haben.

Ist der formale Parameter einer Prozedur vom Typ ARRAY OF WORD bzw. ARRAY OF BYTE, dann hat die entsprechende C-Funktion — wie bei allen offenen Feldern — einen zusätzlichen Parameter, der die aktuelle Anzahl von Feldelementen enthält. Bei der Parameterübergabe muß bestimmt werden, welche Größe in Worten bzw. Bytes der aktuelle Parameter hat. Abgesehen davon wird ein offenes Feld mit Elementen vom Typ WORD bzw. BYTE genauso behandelt wie jedes andere offene Feld auch. Der folgende Ausschnitt eines Modula-Programms

```
PROCEDURE WriteBytes (VAR a: ARRAY OF WORD);
  ...
WriteBytes (w);
```

wird in C zu

```
void WriteBytes(a, O_6)
WORD a[]; LONGCARD O_6;
  ...
WriteBytes(&w, sizeof(w));
```

Bei dieser Abbildung wird angenommen, daß *sizeof*(WORD) bzw. *sizeof*(BYTE) den Wert 1 haben. Ist der aktuelle Parameter ein offenes Feld, dann ist die Größe des aktuellen Parameters in Worten bzw. Bytes: *sizeof(Elementtyp) * aktuelle Anzahl von Feldelementen*.

Der Typ ADDRESS ist definiert als:

```
typedef unsigned char *ADDRESS;
```

In Modula-2 ist der Typ ADDRESS kompatibel mit allen anderen Zeigertypen. Wegen der in C vorhandenen Möglichkeit Zeiger verschiedenen Typs einander zuzuweisen oder einen Zeiger an eine Funktion zu übergeben, dessen Typ sich vom Typ des formalen Parameters unterscheidet, erfordert diese in Modula-2 definierte Typkompatibilität bei der Abbildung nach C keine spezielle Behandlung. Allerdings wird bei der Zuweisung in C eine explizite Typumwandlung erzeugt, um sonst vom C-Übersetzer ausgegebene Warnungen zu unterdrücken.

In Modula-2 können alle arithmetischen Operatoren auf Operanden mit Typ ADDRESS angewandt werden, da ADDRESS auch kompatibel mit dem Typ CARDINAL ist. Damit können beliebige Adreßberechnungen durchgeführt werden. In C kann nur ein ganzzahliger Wert zu einem Zeiger addiert oder von einem Zeiger subtrahiert werden und zwei Zeiger können subtrahiert werden. Außerdem ist in C Zeigerarithmetik so definiert, daß der Ausdruck *p+1* den Wert von *p*, erhöht um die Größe des Objekts auf welches *p* zeigt, liefert. Hat *sizeof(ADDRESS)* den Wert 1, dann ergibt sich hier allerdings kein Unterschied zur Adreßarithmetik in Modula-2. Wird daher im Modula-Programm der Typ ADDRESS innerhalb von arithmetischen Ausdrücken in einer in C nicht zulässigen Kombination verwendet, dann wird der betreffende Operand in C zuerst in den Typ CARDINAL umgewandelt und falls notwendig der Wert des gesamten Ausdrucks in den Typ ADDRESS zurückgewandelt[1].

1 MOCKA definiert, daß ein arithmetischer Ausdruck in dem mindestens ein Operand den Typ ADDRESS hat den Resultattyp ADRRESS hat.

Die Funktion TSIZE wird, wie die Standardfunktion SIZE, auf den C-Operator *sizeof* abgebildet.

ADR ist in SYSTEM_.h mit Hilfe des C-Operators & zur Adreßbestimmung als folgendes Makro definiert:

```
#define ADR(x)          ((ADDRESS) &(x))
```

Da offene Felder auf Vektoren abgebildet werden und ein Vektorname als Ausdruck einen Zeiger auf das erste Element des Vektors liefert, wird für die Bestimmung der Adresse eines offenen Feldes das Makro ADR1 anstelle von ADR benutzt:

```
#define ADR1(x)         ((ADDRESS) x)
```

4.8 Übersetzungseinheiten

4.8.1 Definitionsmoduln

Ein Definitionsmodul *M.md* wird in eine Definitionsdatei *M.h* übersetzt, die die vom Modul *M* exportierten Größen beschreibt. Diese Definitionsdatei wird mit einer *#include*-Anweisung in alle Moduln, die *M* importieren und in das C-Programm *M.c*, welches die Übersetzung des zu *M.md* dazugehörigen Implementierungsmoduls ist, eingefügt.

Alle im Definitionsmodul *M.md* vereinbarten Objekte werden in C in der qualifizierten Form *M_Objektname* geschrieben[1].

Konstanten- und Typdefinitionen werden wie im Kapitel über Deklarationen besprochen nach C umgesetzt.

Die Definitionsdatei enthält eine *extern*-Deklaration für jede exportierte Variable. Diese Deklaration legt die Eigenschaften wie Datentyp und Speicherbedarf der Variablen fest, damit sie in allen Moduln, die die Variable benutzen, bekannt sind. Mit der *extern*-Deklaration wird aber kein Speicherplatz reserviert. Die eigentliche Vereinbarung der Variablen, die dafür sorgt, daß Speicherplatz reserviert wird, ist in *M.c* enthalten.

Wird in der Definition einer exportierten Variablen ein anonymer strukturierter Typ verwendet, so enthält in C nur die *extern*-Deklaration in der Definitionsdatei die vollständige Strukturdeklaration dieses Typs. In der eigentlichen Vereinbarung in *M.c* wird der in der *extern*-Deklaration eingeführte Strukturname zur Beschreibung des Typs verwendet. Diese Unterscheidung ist notwendig, da eine doppelt vorhandene vollständige Strukturdeklaration vom C-Übersetzer als unzulässige Redeklaration der betreffenden Struktur zurückgewiesen wird. Somit wird der folgende Ausschnitt des Definitionsmoduls *Parser.md*

```
VAR ParsTab : ARRAY [0..127] OF CHAR;
```

in der Definitionsdatei *Parser.h* zu

1 Ausnahme: Die Komponentennamen einer Struktur werden in unqualifizierter Form geschrieben, da sie immer bereits durch den Namen der Struktur, die sie enthält, qualifiziert sind.

```
extern struct Parser_1 {
    CHAR A[127 + 1];
} Parser_ParsTab;
```

und im Programm *Parser.c* zu

```
struct Parser_1 Parser_ParsTab;
```

Da Zeichenketten der Länge n > 1 auf einen initialisierten Zeichenvektor abgebildet werden, muß eine in einem Definitionsmodul als Konstante vereinbarte Zeichenkette bei der Übersetzung nach C wie eine exportierte Variable behandelt werden. Das bedeutet, daß in der Definitionsdatei *M.h* eine *extern*-Deklaration und im Programm *M.c* die eigentliche Vereinbarung und Initialisierung der Konstante enthalten ist.

Für jede von *M* exportierte Prozedur ist ebenfalls eine *extern*-Deklaration vorhanden, die den Namen und den Ergebnistyp der für die Prozedur erzeugten C-Funktion in allen Moduln bekannt macht, die die Prozedur importieren.

Der Definitionsmodul *Tree.md*

```
DEFINITION MODULE Tree;

CONST  NoTree   = NIL;

TYPE
  tTree         = POINTER TO tNode;
  tNode         = RECORD
                    Key   : INTEGER;
                    Left  ,
                    Right : tTree;
                  END;

VAR  Root       : tTree;

PROCEDURE MakeNode (Key: INTEGER; Left, Right: tTree): tTree;

END Tree.
```

wird in C zur Definitionsdatei *Tree.h*

```
#define Tree_NoTree     NIL

typedef struct Tree_1 *Tree_tTree;
typedef struct Tree_1 {
    INTEGER    Key;
    Tree_tTree Left, Right;
} Tree_tNode;

extern  Tree_tTree Tree_Root;

extern  Tree_tTree Tree_MakeNode();
```

4.8.2 FOREIGN-Moduln

MOCKA erlaubt es, daß in C geschriebene Prozeduren im Modula-Programm aufgerufen werden. Diese Prozeduren müssen in sogenannten FOREIGN-Modul deklariert sein, die als Definitionsmoduln fungieren[1]. Bei der Übersetzung von Modula-2 nach C wird ein FOREIGN-Modul im Prinzip wie ein Definitionsmodul behandelt. Der einzige Unterschied ist, daß alle dort definierten Prozeduren im erzeugten C-Programm nicht mit dem Modulnamen qualifiziert werden.

4.8.3 Implementierungs- und Programmoduln

Ein Implementierungs- oder Programmodul *M.mi* wird in ein C-Programm *M.c* übersetzt. *M.c* enthält neben der Übersetzung der im Implementierungs- oder Programmodul enthaltenen Konstanten-, Typ-, Variablen- und Prozedurdeklarationen auch Vereinbarungen für die in der Definitionsdatei *M.h* in *extern*-Deklarationen erwähnten exportierten Variablen.

SYSTEM_.h mit der Definition der Standardtypen, -prozeduren und -funktionen wird in jedes erzeugte C-Programm als erste Definitionsdatei eingefügt.

Da auch in C eine getrennte Übersetzung möglich ist, können alle vom Übersetzer für Implementierungs- bzw. Programmoduln erzeugten C-Programme vom C-Übersetzer getrennt übersetzt werden.

4.8.4 IMPORT-Anweisungen

IMPORT-Anweisungen werden in eine Folge von *#include*-Anweisungen übersetzt. Damit eine Definitionsdatei nicht mehrfach in ein C-Programm eingefügt wird[2], wird am Anfang jeder Definitionsdatei *M.h* der Makroname DEFINITION_*M* mit einer *#define*-Anweisung definiert und die IMPORT-Anweisung

```
FROM M IMPORT ...
```

in die Präprozessor-Anweisungen

```
#ifndef DEFINITION_M
#include "M.h"
#endif
```

übersetzt. Damit wird erreicht, daß die Definitionsdatei *M.h* nur genau einmal in eine Quelldatei eingefügt wird.

---

1 FOREIGN-Moduln können auch Typ- und Konstantendeklarationen enthalten.

2 Mehrfach vorhandene identische Typ- bzw. Konstantendeklarationen innerhalb einer Quelldatei sind in C nicht zulässig.

4.8.5 Opaque Typen

In Modula-2 ist es erlaubt im Definitionsmodul nur den Namen eines Typs zu vereinbaren, dessen vollständige Definition dann im Implementierungsmodul enthalten sein muß. Diese Typen, deren Struktur den sie importierenden Moduln unbekannt ist, werden als opaque Typen bezeichnet. Opaque Typen sind immer Zeigertypen.

Die Abbildung nach C wird folgendermaßen gehandhabt: In der Definitionsdatei wird ein opaquer Typ als

```
typedef OPAQUE Modulname_Typname;
```

vereinbart. Der Typ OPAQUE ist in SYSTEM_.h als

```
typedef ADDRESS OPAQUE;
```

definiert.

Da auf opaque Typen in Modula-2 außerhalb des Moduls in dem sie definiert sind, nur Zuweisung und Vergleich anwendbar sind, ist bei der Übersetzung von Moduln, die diese Typen importieren keine spezielle Behandlung von opaquen Typen notwendig. Die im Implementierungsmodul enthaltene vollständige Definition des opaquen Typs wird in eine zweite *typedef*-Anweisung umgesetzt. Da der Typname in der Definitionsdatei qualifiziert wird, ergeben sich dabei keine Namenskonflikte. Auch innerhalb des nach C übersetzten Implementierungsmoduls wird die Typdefinition aus der Definitionsdatei in allen Deklarationen benutzt. Um jedoch eine korrekte Anwendung des opaquen Typs in Anweisungen zu erzwingen, wird vor jeder Dereferenzierung eine explizite Typumwandlung in den vollständigen Typ eingefügt.

Der Definitionsmodul *Lists.md*

```
DEFINITION MODULE Lists;
FROM SYSTEM     IMPORT ADDRESS;
TYPE  tList;
PROCEDURE Insert (VAR List: tList; Elmt: ADDRESS);
  ...
END Lists.
```

wird in C zur Definitionsdatei *Lists.h*

```
#define DEFINITION_Lists
typedef OPAQUE Lists_tList;
extern  void Lists_Insert();
  ...
```

Der Implementierungsmodul *Lists.mi*

```
IMPLEMENTATION MODULE Lists;
FROM SYSTEM     IMPORT ADDRESS;
TYPE
  tList = POINTER TO tElmt;
  tElmt = RECORD
            Elmt : ADDRESS;
            Next : tList;
          END;
PROCEDURE Insert (VAR List: tList; Elmt: ADDRESS);
VAR  Head : tList;
BEGIN
  Head       := MakeElmt (Elmt);
  Head^.Next := List;
  List       := Head;
END Insert;
  ...
END Lists.
```

wird in C zum Programm *Lists.c*

```
#include "SYSTEM_.h"
#ifndef DEFINITION_Lists
#include "Lists.h"
#endif
typedef struct S_1 *tList;
typedef struct S_1 {
    ADDRESS     Elmt;
    Lists_tList Next;
} tElmt;
void Lists_Insert(List, Elmt)
Lists_tList *List; ADDRESS Elmt;
{
    Lists_tList Head;
    Head = MakeElmt(Elmt);
    ((tList)Head)->Next = List;
    *List = Head;
}
  ...
```

4.8.6 Modulinitialisierung

Die Modulinitialisierung wird in C mit folgendem Schema behandelt: Für jeden Implementierungsmodul wird in C eine parameterlose Initialisierungsroutine erzeugt. Diese ruft zunächst die Initialisierungsroutinen aller direkt importierten Moduln in der textuellen Reihenfolge der zugehörigen IMPORT-Anweisungen auf und enthält dann die Anweisungen aus dem Rumpf des Moduls. Um eine korrekte und vollständige Initialisierung aller Moduln zu erreichen, müssen dabei auch die IMPORT-Anweisungen aus dem zum Implementierungsmodul dazugehörigen Definitionsmodul berücksichtigt werden. Diese Reihenfolge der Initialisierung entspricht genau der in der Sprachdefinition von Modula-2 geforderten Reihenfolge. Damit die

Initialisierungsroutine nicht mehrfach ausgeführt wird, enthält sie eine boolesche Variable, die angibt, ob die Routine bereits aufgerufen wurde. Nur beim ersten Aufruf wird sie ausgeführt. Somit haben die Initialisierungsroutinen folgende Form:

```
void BEGIN_Modulname()
{
    static BOOLEAN has_been_called = FALSE;

    if (!has_been_called) {
        has_been_called = TRUE;

        /*
         *  Initialisierungsroutinen der
         *  importierten Moduln aufrufen.
         *
         *  Anweisungen des Modulrumpfs
         *  ausführen.
         */
    }
}
```

Diese Behandlung der Modulinitialisierung erfordert zur Laufzeit einen etwas erhöhten Aufwand, da ein Teil der Initialisierungsroutinen möglicherweise mehrfach aufgerufen wird. Der große Vorteil dieser Lösung ist aber, daß alle Moduln korrekt initialisiert werden, ohne daß bei der Übersetzung des Programmoduls bestimmt werden muß, welche Moduln in einem Programm benutzt werden.

4.8.7 Hauptprogramm

Der Rumpf des Programmoduls bildet in Modula-2 das Hauptprogramm. In C wird er zu der Funktion BEGIN_MODULE. Diese ruft zunächst die Initialisierungsroutinen aller direkt importierten Moduln auf und enthält dann die Anweisungen aus dem Rumpf des Programmoduls.

Die Datei SYSTEM_.c enthält eine feste Definition der Funktion *main*, die in C das Hauptprogramm bildet. Diese Funktion hat folgendes Aussehen:

```
int    SYSTEM_argc;
char **SYSTEM_argv;
char **SYSTEM_envp;

main(argc, argv, envp)
int argc;
char *argv[], *envp[];
{
  SYSTEM_argc = argc;
  SYSTEM_argv = argv;
  SYSTEM_envp = envp;

  BEGIN_MODULE();

  exit(0);
}
```

Vor dem Aufruf von BEGIN_MODULE werden in *main* die Programmparameter behandelt. In C und unter UNIX können beim Aufruf eines Programms Argumente aus der Kommandozeile an das Programm übergeben werden. Das erste Argument *argc* ist die Anzahl der Argumente, das zweite Argument *argv* ist ein Zeiger auf einen Vektor mit Zeichenketten, die die Argumente enthalten. Das dritte Argument *envp* schließlich ermöglicht den Zugriff auf UNIX-Variablen der Aufrufumgebung.

Die MOCKA-Bibliothek enthält ein FOREIGN-Modul, welches die Definition von Prozeduren enthält, welche Modula-Programmen den Zugriff auf die Programmparameter ermöglichen. Die Implementierung dieses FOREIGN-Moduls ist bei MOCKA ein Teil des Laufzeitsystems. Die obige Behandlung der Programmparameter in der Funktion *main* ermöglicht es, dieses FOREIGN-Modul in C zu implementieren.

Die Funktion *main* wird mit *exit(0)* beendet, um der Aufrufumgebung anzuzeigen, daß die Ausführung des Programms erfolgreich war.

5. Implementierung des Übersetzers

Bild 5.1 gibt einen kurzen Überblick über die einzelnen Phasen des Übersetzers *Mtc*, die für die Implementierung der Phasen verwendeten Übersetzerbauwerkzeuge (in Klammern hinter den Namen der Phasen) sowie über die wichtigsten intern auftretenden Datenstrukturen, die der Kommunikation der einzelnen Übersetzerphasen dienen.

Die folgenden Kapitel enthalten eine genauere Besprechung der Aufgaben und Implementierung der einzelnen Phasen des Übersetzers, wobei besonderer Wert auf die Darstellung des Einsatzes der Werkzeuge gelegt wird. Grundlegende Begriffe und Techniken des Übersetzerbaus [Aho 86, Waite 84] werden dabei als weitgehend bekannt vorausgesetzt.

5.1 Lexikalische Analyse

Der Symbolentschlüssler liest das Eingabeprogramm zeichenweise und zerlegt es in eine Folge von Grundsymbolen, die an den Zerteiler weitergereicht werden. Seine Aufgaben sind im einzelnen:

- Erkennen der möglichen Grundsymbole der Sprache Modula-2: Bezeichner, ganze Zahlen (oktal, dezimal oder hexadezimal), reelle Zahlen, Zeichenkonstanten, Zeichenketten, Schlüsselwörter, Operatoren und Trennsymbole.
- Berechnung der Attribute der Grundsymbole:
  - Quelltextposition
  - semantische Information
- Überlesen von Kommentaren, Zwischenräumen und Zeilenwechseln.
- Melden von illegalen Zeichen und nicht geschlossenen Kommentaren und Zeichenketten.

Die Kodierung der erkannten Grundsymbole und deren Attribute werden an den Zerteiler weitergereicht.

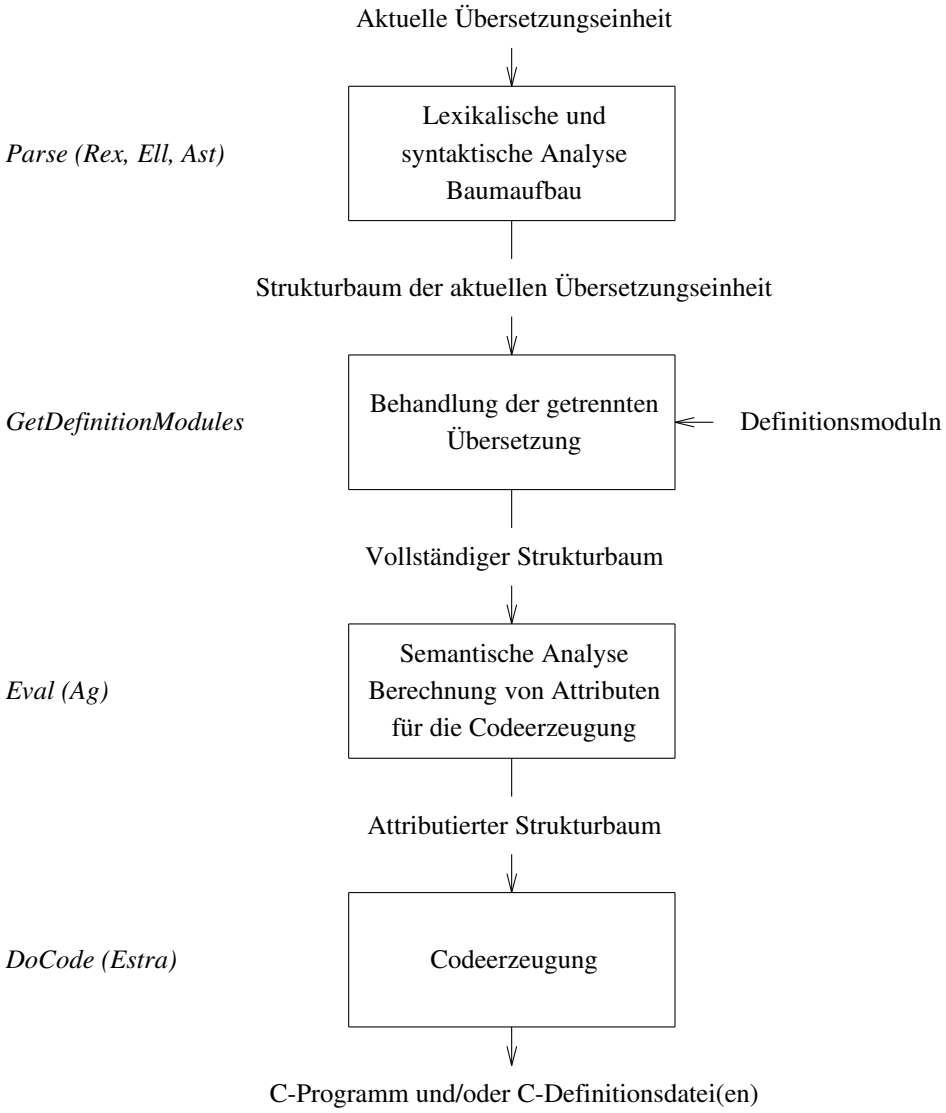

Bild 5.1: Phasen des Übersetzers *Mtc*

5.1.1 Attribute der Grundsymbole

Die Quelltextposition der Grundsymbole wird für spätere Fehlermeldungen benötigt. Sie setzt sich zusammen aus dem Namen der Eingabedatei sowie aus Zeile und Spalte des Grundsymbols innerhalb der Eingabedatei. Warum auch der Name der Eingabedatei mit in die Quelltextposition aufgenommen wurde, wird im Zusammenhang mit der Behandlung der getrennten Übersetzung näher erläutert.

Für spätere Übersetzerphasen wird zusätzliche sematische Information für solche Grundsymbole benötigt, die durch die zugehörige Zeichenkette nicht eindeutig festgelegt sind. Die Tabelle 5.2 beschreibt diese Grundsymbole und die Bedeutung der zugehörigen semantischen Information.

Grundsymbol	Semantische Information
Bezeichner	Eindeutige Abbildung der zugehörigen Zeichenkette auf eine ganze Zahl
Dezimalzahl	Wert der Zahl
Oktalzahl	Wert der Zahl
Hexadezimalzahl	Wert der Zahl
Zeichenkonstante	Zugehöriges Zeichen
Reelle Zahl	Verweis in die Konstantentabelle
Zeichenkette	Verweis in die Konstantentabelle

Tabelle 5.2: Semantische Information der Grundsymbole

5.1.2 Symbol- und Konstantentabelle

Der Symboltabellenmodul *Idents* zur eindeutigen Abbildung von Zeichenketten auf ganze Zahlen, der Konstantentabellenmodul *StringMem* zur Speicherung von Zeichenketten variabler Länge sowie der von beiden verwendete Modul *Strings* zur Manipulation von Zeichenketten und zur Konvertierung von Zeichenketten in interne Darstellungen, wie z.B. ganze Zahlen, wurden aus der Bibliothek *Reuse* [Grosch 87a] entnommen.

5.1.3 Spezifikation des Symbolentschlüsslers

Der Symbolentschlüssler wurde mit Hilfe des Generators *Rex* [Grosch 87b] aus einer Beschreibung der Grundsymbole von Modula-2 erzeugt. Eine solche Beschreibung besteht aus einer Reihe von regulären Ausdrücken, welche die Grundsymbole beschreiben und Aktionen, die ausgeführt werden, wenn die zugehörigen regulären Ausdrücke in der Eingabe erkannt werden. *Rex* übersetzt diese Beschreibung in einen tabellengesteuerten Symbolentschlüssler, der im Modul *Scanner* enthalten ist.

Die Aktionen bestehen in den meisten Fällen aus der Berechnung der Attribute des zugehörigen Grundsymbols und einer RETURN-Anweisung, mit der die Kodierung des Grundsymbols an den Zerteiler zurückgeliefert wird. Zeile und Spalte der Grundsymbole im Quelltext werden vom generierten Symbolentschlüssler automatisch berechnet.

Das folgende Beispiel zeigt einige Ausschnitte aus der Spezifikation des Symbolentschlüsslers:

```
#STD#      "(*"          :- {NestingLevel := 1; yyStart (Comment);}
#Comment#  "(*"          :- {INC (NestingLevel);}
#Comment#  "*)"          :- {DEC (NestingLevel);
                             IF NestingLevel = 0 THEN yyStart (STD); END;}
#Comment#  "(" | "*" | - {*(\t\n} + :- {}
#STD# ARRAY              : {Attribute.Pos.File   := SourceFileName;
                            Attribute.Pos.Line   := Line;
                            Attribute.Pos.Column := Column;
                            RETURN TokArray;}
#STD# letter (letter | digit) * : {
                            GetWord (Word);
```

```
Attribute.Ident      := MakeIdent (Word);
Attribute.Pos.File   := SourceFileName;
Attribute.Pos.Line   := Line;
Attribute.Pos.Column := Column;
RETURN TokIdent;}
```

Für komplexe Aufgaben und insbesondere für die Behandlung von Linkskontext stehen sogenannte Startzustände zur Verfügung. In der Spezifikation kann festgelegt werden, daß bestimmte Muster nur in bestimmten Startzuständen erkannt werden. Zu Beginn ist der Symbolentschlüssler immer im Startzustand STD. Startzustände können mit speziellen Anweisungen in den Aktionen gewechselt werden. Diese Möglichkeit wird im obigen Beispiel für die Bearbeitung von Kommentaren benutzt. Die Variable *NestingLevel* ist notwendig, da Kommentare in Modula-2 geschachtelt sein können und geschachtelte Kommentare nicht durch einen regulären Ausdruck beschrieben werden können. Die Funktion *MakeIdent* aus dem Modul *Idents* bildet die vom Symbolentschlüssler erkannte Zeichenfolge eines Bezeichners eindeutig auf eine ganze Zahl ab.

5.2 Syntaktische Analyse und Baumaufbau

Die Aufgabe des Zerteilers ist es, die syntaktische Struktur des Quellprogramms zu analysieren, dessen syntaktische Korrektheit zu überprüfen und einen abstrakten Strukturbaum aufzubauen.

5.2.1 Zerteilerspezifikation

Für die Erstellung eines Zerteilers standen die beiden Zerteilergeneratoren *Ell* und *Lalr* [Vielsack 88] zur Verfügung. Verglichen mit *Lalr* hat *Ell* die folgenden Vor- und Nachteile:

Vorteile:

- Die von *Ell* unterstützte L-Attributierung[1], mit der parallel zur Zerteilung Attributwerte berechnet werden können, ist mächtiger als die von *Lalr* unterstützte S-Attributierung[2]. Außerdem kann die Attributierung bei *Ell* im Gegensatz zu *Lalr* [3] symbolisch durchgeführt werden und ist damit weniger fehleranfällig.
- Ein von *Ell* erzeugter Zerteiler ist etwa doppelt so schnell wie ein von *Lalr* erzeugter.

Nachteile:

- Der gravierendste Nachteil von *Ell* ist, daß die von *Ell* akzeptierte Sprachklasse ELL(1) weniger mächtig ist als die von *Lalr* akzeptierte Sprachklasse LALR(1).

---

1 Bei der L-Attributierung sind sowohl vererbte als auch synthetisierte Attribute möglich. Dabei darf innerhalb einer Grammatikregel $X_0 : X_1 \ldots X_n$. ein Attribut des Symbols X_i ($1 \leq i \leq n$) nur von vererbten Attributen des Symbols X_0 oder von Attributen der Symbole X_1 bis X_{i-1} abhängen.

2 Bei der S-Attributierung sind nur synthetisierte Attribute zulässig.

3 *Lalr* benutzt die Pseudovariablen $$ bzw. $n, um auf das Nichtterminal der linken Seite bzw. das n-te Symbol der rechten Seite einer Produktion zuzugreifen.

– Von *Lalr* erzeugte Zerteiler sind in der Regel kleiner als von *Ell* erzeugte.

Da bereits eine ELL(1)-Grammatik für Modula-2 vorlag, gaben die oben genannten Vorteile von *Ell* den Ausschlag für die Entscheidung *Ell* zu verwenden.

Die Eingabe für *Ell* besteht aus einer kontextfreien Grammatik in erweiterter BNF, die die Struktur der Eingabesprache beschreibt. Jede Produktion der Grammatik kann eine Reihe von semantischen Aktionen enthalten, welche ausgeführt werden, sobald der Zerteiler die zugehörigen Grammatikregeln analysiert.

Ell erzeugt aus der kontextfreien Grammatik einen Zerteiler (enthalten im Modul *Parser*), der nach dem Verfahren des rekursiven Abstiegs arbeitet. Die semantischen Aktionen werden an die entsprechenden Stellen in den Quelltext des Zerteilers kopiert. Der erzeugte Zerteiler besitzt eine automatische Fehlerbehandlung, die mit einem rücksetzungsfreien Fehlerkorrekturalgorithmus arbeitet. Ist die vom Symbolentschlüssler gelieferte Folge von Grundsymbolen nicht syntaktisch korrekt, setzt der Zerteiler nach Abgabe einer Fehlermeldung die Zerteilung fort, indem er Symbole überliest und/oder einfügt, so daß immer ein syntaktisch korrekter Strukturbaum erzeugt wird.

5.2.2 Spezifikation des Strukturbaums

Eine Spezifikation der abstrakten Syntax von Modula-2 wurde mit der Spezifikationssprache des Generators *Ast* [Grosch 89a] erstellt. *Ast* erzeugt aus der Spezifikation der abstrakten Syntax den Modul *Tree*, der Typdeklarationen zur Implementierung des Strukturbaums und Prozeduren zum Aufbau von Baumknoten und zur Umkehr von Knotenlisten enthält.

Die Struktur und die Eigenschaften der Knoten, aus denen ein abstrakter Strukturbaum besteht, werden in einer *Ast*-Spezifikation mit sogenannten Knotentypen beschrieben. Jeder Knoten gehört zu einem bestimmten Knotentyp. Der Typ beschreibt die Kindknoten (Unterbäume) und die Attribute des Knotens. Knotentypen können als Erweiterung von anderen Knotentypen definiert werden; letztere werden als Basistypen bezeichnet, erstere als abgeleitete Typen. Die abgeleiteten Typen erben die Kindknoten und Attribute des Basistyps und können wiederum erweitert werden. Überall dort, wo ein Knoten eines bestimmten Basistyps erwartet wird, ist auch ein Knoten zulässig, dessen Typ aus dem Basistyp abgeleitet wurde.

Folgender Ausschnitt der *Ast*-Spezifikation beschreibt die abstrakte Syntax eines Teils der Anweisungen von Modula-2:

```
Stmts          = <
  Stmts0       = .
  Stmt         = Next: Stmts REVERSE <
    Assign     = Designator Expr .
    Call       = Designator Actuals .
    If         = Cond: Expr Then: Stmts Elsifs Else: Stmts .
    Case       = Expr Cases Else: Stmts [Default: BOOLEAN] .
    While      = Cond: Expr Stmts .
    ...
  >.
>.
```

Die Knotentypen *Assign*, *Call*, usf. sind Erweiterungen des Knotentyps *Stmt*, welcher wiederum eine Erweiterung des Knotentyps *Stmts* ist. Attribute sind in eckigen Klammern eingeschlossen. Die Spezifikation von Kindknoten besteht aus dem zugehörigen Knotentyp und einem optionalen Selektornamen. Die Knotentypen *Assign*, *Call*, usf. erben den Kindknoten mit Namen *Next* von ihrem Basistyp *Stmt*. Die abstrakte Syntax enthält im Gegensatz zur konkreten Syntax keine eigenen Knotentypen für Listen. Stattdessen enthält jeder Knotentyp für Listenelemente einen Kindknoten *Next*, der auf das nächste Element der Liste verweist. Die Angabe REVERSE wird von *Ast* zur Erzeugung der Prozedur für die Umkehr von Knotenlisten benötigt.

Beim Entwurf der abstrakten Syntax wurde darauf geachtet, daß die Strukturbäume möglichst einfach und kompakt aufgebaut sind. Dadurch wird nicht nur der Speicherbedarf der Strukturbäume reduziert, sondern auch deren Verarbeitung beschleunigt, da bei der Traversierung weniger Knoten besucht werden müssen. Anhang A enthält die vollständige *Ast*-Spezifikation der abstrakten Syntax von Modula-2, wie sie für den Übersetzer verwendet wurde.

5.2.3 Baumaufbau

Zum Aufbau des Strukturbaums während der Zerteilung wird der von *Ell* angebotene Mechanismus zur L-Attributierung, die von *Ast* generierten Prozeduren für den Baumaufbau und die Prozedur *ReverseTree* zur Umkehr von Knotenlisten benutzt.

Das folgende Beispiel zeigt einen Ausschnitt der kontextfreien Grammatik für Fallmarken mit semantischen Aktionen für den Baumaufbau:

```
Labels          /* Tree: synthesized                    */
                :
  { Label1.Tree := mLabels0 (); }
                   Label || ','
  { Labels0.Tree := ReverseTree (Label1.Tree); }
                .
Label           /* Tree: inherited and synthesized      */
                : Expr
                    ( '..' Expr
    { Label0.Tree := mLabelRange (Label0.Tree, Expr1.Tree, Expr2.Tree); }
                   |
    { Label0.Tree := mLabel (Label0.Tree, Expr1.Tree); }
                    )
                .
```

Baumknoten werden aufgebaut durch Aufrufe der von *Ast* für diesen Zweck generierten Prozeduren. *Ast* generiert für jeden Knotentyp eine solche Prozedur. Die Kindknoten und Attribute des Knotens werden als Parameter übergeben und ein Zeiger auf den neu angelegten Knoten wird zurückgeliefert. Die Nichtterminale der kontextfreien Grammatik besitzen ein Attribut *Tree*, welches den Teilbaum beschreibt, dessen Wurzel das Nichtterminal ist. Die kontextfreie Grammatik enthält noch verschiedene Nichtterminale für Listen und Listenelemente. Im Baum existieren aber nur noch Knotentypen für die Listenelemente mit einem Verweis auf das jeweils nächste Element. Daher wird bei der Zerteilung einer Liste der bisher aufgebaute Strukturbaum für die Liste, der aus verketteten Listenelementen besteht, als vererbtes Attribut an das Nichtterminal für die Listenelemente übergeben und dort beim Aufbau eines neuen Listenelements als

Verweis auf das nächste Listenelement verwendet. Dabei wird die Liste zunächst in der verkehrten Reihenfolge aufgebaut. Dies wird aber in der Grammtikregel für die Liste nach der Zerteilung der Liste durch einen Aufruf der von *Ast* generierten Prozedur *ReverseTree* korrigiert.

5.3 Behandlung der getrennten Übersetzung

Die getrennte Übersetzung ermöglicht es, Programme in kleinere Teile zu zerlegen, die vom Übersetzer einzeln übersetzt werden können. Die wichtigsten Ziele einer solchen Zerlegung sind:

- Aufteilung eines Programmsystems in überschaubare und möglichst abgeschlossene Einheiten (Moduln), die eine definierte Schnittstelle nach außen besitzen und deren interne Implementierung nach außen verborgen bleibt.
- Arbeitsteilige Erstellung von Programmsystemen.
- Reduktion des Übersetzungsaufwands.
- Entwicklung von Programmbibliotheken mit wiederverwendbarer Software.

Bei einer getrennten Übersetzung sollte der Übersetzer zur Erzielung höherer Zuverlässigkeit nach Möglichkeit Korrektheitsprüfungen über die Grenzen von Übersetzungseinheiten hinweg vornehmen[1].

Modula-2 unterstützt die oben genannten Forderungen durch sein Modulkonzept, insbesondere durch die Trennung von globalen Moduln in Schnittstelle (Definitionsmodul) und Implementierung (Implementierungsmodul). Soll in Modula-2 eine Übersetzungseinheit übersetzt werden, dann müssen dem Übersetzer die Schnittstellen aller von der Übersetzungseinheit importierten Moduln bekannt sein. Auch bei der Übersetzung von Modula-2 nach C müssen die Schnittstellen der importierten Moduln bekannt sein, da für eine korrekte Übersetzung semantische Informationen über die importierten Objekte notwendig sind.

Eine Lösung für dieses Problem, die auch von MOCKA implementiert wird, ist, bei der Übersetzung eines Definitionsmoduls eine sogenannte Symboldatei anzulegen, die einen Ausschnitt der Definitionstabelle des Übersetzers darstellt und eine Beschreibung aller vom zugehörigen Modul exportierten Objekte enthält. Diese Symboldatei wird dann bei der Übersetzung von abhängigen Moduln benutzt, indem die dort enthaltene Information wieder in die Definitionstabelle eingetragen wird.

Eine alternative Lösung ist, bei jedem Übersetzungsvorgang den Quelltext aller Schnittstellen von (transitiv) importierten Moduln von neuem einzulesen und zu analysieren.

Vergleicht man die beiden Lösungen ergibt sich folgendes:

- Ein Übersetzer, welcher die Lösung mit den Symboldateien verwendet, ist schneller und der gesamte Übersetzungsaufwand reduziert sich, da eine Schnittstelle nur einmal übersetzt und nur einmal eine (binäre) Symboldatei angelegt werden muß; bei der Übersetzung aller abhängigen Übersetzungseinheiten muß dann nur jeweils die einmal angelegte Symboldatei neu eingelesen werden, was sicher weniger Aufwand verursacht als den Quelltext der Schnittstelle jedesmal von neuem einzulesen und zu analysieren.

1 Eine häufige Fehlerquelle bei der Entwicklung von C-Programmen ist, daß in C solche Prüfungen vom Übersetzer nicht automatisch vorgenommen werden.

– Der Programmieraufwand für die Lösung mit den Symboldateien ist deutlich höher, da Dateiformate für die Symboldateien definiert und Operationen programmiert werden müssen, die a) die Information aus der Definitionstabelle lesen und auf die Symboldatei schreiben und b) die Symboldatei lesen und die Information in die Definitionstabelle eintragen. Dieser zusätzliche Programmieraufwand entfällt für die zweite Lösung, da Einlesen und Analyse der Schnittstellen sowieso ein Teil des Übersetzers sind.
– Zur Unterstützung des Benutzers sollte ein Übersetzer, der die erste Lösung realisiert, überprüfen, ob alle für die Übersetzung einer bestimmten Übersetzungseinheit benötigten Symboldateien vorhanden und auf dem neuesten Stand, d.h. konsistent mit dem Quelltext des zugehörigen Definitionsmoduls, sind und bei Bedarf zuerst die (Neu-)Übersetzung der zugehörigen Definitionsmoduln in eine Symboldatei anstoßen[1]. Dies entfällt bei der zweiten Lösung automatisch, da sowieso alle benötigten Definitionsmoduln eingelesen werden[2].

Da *Mtc* nicht für die Programmentwicklung gedacht ist und folglich ein fertig entwickeltes Modula-Programm auch nur einmal nach C übersetzt werden muß, ist es nicht kritisch, wenn der Übersetzer aufgrund eines wiederholten Einlesens und einer wiederholten Analyse von Schnittstellen etwas langsamer ist. Daher gab der deutlich geringere Aufwand für die Implementierung der zweiten Lösung den Ausschlag, diese im Übersetzer *Mtc* zu realisieren.

Zur Realisierung der oben skizzierten Lösung im Übersetzer *Mtc* wurde zunächst die Definition des Strukturbaums erweitert. Ein Strukturbaum besteht nicht nur aus einer Übersetzungseinheit, sondern aus einer Liste von Übersetzungseinheiten:

```
ROOT            = CompUnits .
CompUnits       = <
  CompUnits0    = .
  CompUnit      = [Kind: SHORTCARD] [Ident: tIdent] [Pos: tPosition]
                  Next: CompUnits REVERSE <
    DefMod      = Import Decls .
    ProgMod     = Import Decls Stmts .
  >.
>.
```

Da die Übersetzungseinheiten aus verschiedenen Quelldateien stammen, wird jetzt auch klar, warum, wie im Kapitel über die lexikalische Analyse bereits erwähnt, die Quelltextposition auch den Namen der Quelldatei enthalten muß.

Der Aufbau dieser Liste von Übersetzungseinheiten, d.h. der Aufbau des Strukturbaums in seiner endgültigen Form und somit die Behandlung der getrennten Übersetzung, übernimmt der Modul *DefMods*. Der vom Zerteiler aufgebaute Strukturbaum für die aktuelle Übersetzungseinheit bildet die Eingabe für diesen Modul. Innerhalb von *DefMods* werden die Importlisten aller bereits eingelesenen Übersetzungseinheiten traversiert und für alle importierten und bisher noch nicht eingelesenen (Definitions-)Moduln der Zerteiler aufgerufen, welcher das Einlesen der

1 Ist eine derartige automatische Unterstützungsfunktion nicht implementiert, muß der Benutzer selbst die Modulabhängigkeiten analysieren, muß sich selbst um die Übersetzungsreihenfolge und die Konsistenz der erzeugten Symboldateien bzw. Objektprogramme kümmern. Eine u.U. recht aufwendige und fehleranfällige Arbeit, die unbedingt vom Übersetzer übernommen werden sollte.

2 Allerdings wird dadurch nicht verhindert, daß die erzeugten Objekt- bzw C-Programme möglicherweise gegenseitig inkonsistent sind!

Definitionsmoduln steuert und Strukturbäume für sie aufbaut[1]. Ist die aktuelle Übersetzungseinheit ein Implementierungsmodul, muß auch der zugehörige Definitionsmodul eingelesen werden. Aus dem Teilbaum der aktuellen Übersetzungseinheit und aus den vom Zerteiler gelieferten Teilbäumen der benötigten Definitionsmoduln wird schließlich der vollständige Strukturbaum aufgebaut und an die semantische Analyse weitergereicht.

Vergleicht man den Umfang der für die Behandlung der getrennten Übersetzung notwendigen Programmteile von *Mtc* und MOCKA, bestätigen sich die oben gemachten Aussagen des Vergleichs hinsichtlich des Programmieraufwands der beiden Lösungen eindrucksvoll. Allein der Modul *DfFiles* von MOCKA, der die Symboldateien liest und beschreibt sowie deren Daten aus der Definitionstabelle liest und dort wieder einträgt, umfaßt ca. 1450 Zeilen Modula-Code. Dazu kommt dann noch zusätzlich die von MOCKA implementierte Steuerung der Nachübersetzung bzw. der Übersetzungsreihenfolge. Im Gegensatz dazu umfaßt der Modul *DefMods* von *Mtc*, der die gesamte getrennte Übersetzung behandelt, nur ca. 250 Zeilen Modula-Code.

5.4 Semantische Analyse

Normalerweise ist es die Aufgabe der semantischen Analyse, die Bedeutung bzw. die Eigenschaften des Quellprogramms, welche zur statischen Semantik[2] der Sprache gehören, zu bestimmen und zu überprüfen, ob diese Eigenschaften den Regeln der statischen Semantik der Sprache, wie sie von der Sprachdefinition festgelegt werden, genügen. Wie in der Einleitung bereits näher erläutert und begründet, soll *Mtc* die semantische Korrektheit der Eingabeprogramme nicht überprüfen. Daher kann die semantische Analyse im Übersetzer *Mtc* darauf beschränkt werden, die Eigenschaften des Modula-Programms zu bestimmen, soweit sie für eine korrekte Übersetzung nach C bekannt sein müssen. Aus diesem Grund sind die zentralen Aufgaben der semantischen Analyse im Übersetzer *Mtc* der Aufbau einer Definitionstabelle, die Bezeichneridentifikation und die Typbestimmung in Ausdrücken.

5.4.1 Spezifikation der semantischen Analyse mit einer Attributgrammatik und abstrakten Datentypen

Die semantische Analyse wurde mit einer Attributgrammatik spezifiziert. Der Generator *Ag* [Grosch 89b] erzeugt aus dieser Attributgrammatik einen Attributauswerter (enthalten im Modul *Semantics*), der mit rekursiven Besuchsprozeduren, welche den Strukturbaum traversieren und die Attribute berechnen, implementiert ist.

Ag arbeitet eng mit *Ast* zusammen. Die in Kapitel 5.2.2 beschriebene Spezifikation der abstrakten Syntax ist nur der erste Teil einer Attributgrammatik, deren zweiter Teil aus der Deklaration von Attributen und aus Attributierungsregeln besteht, die die Werte dieser Attribute festlegen.

---

1 Entsprechend der von MOCKA implementierten Konvention wird ein Definitionsmodul *M* in einer Datei mit Namen *M.md* erwartet. Beim Aufruf von *Mtc* können auch Bibliotheken angegeben werden, die dann nach den benötigten Definitionsmoduln durchsucht werden.

2 Zur statischen Semantik einer Sprache gehören alle Eigenschaften eines Programms, die ohne eine Ausführung des Programms abgeleitet werden können. Die dynamische Semantik eines Programms dagegen umfaßt alle Eigenschaften, die nur durch seine Ausführung bestimmt werden können.

Attribute und Attributierungsregeln werden den Knotentypen der abstrakten Syntax zugeordnet. Als Typen für die Attribute sind alle Typen der Zielsprache Modula-2, in welcher der Attributauswerter implementiert ist, zulässig. Die Berechnung der Attribute wird ebenfalls mit Anweisungen der Zielsprache Modula-2, wie z.B. Zuweisung oder bedingter Anweisung, ausgedrückt, aus denen *Ag* die Attributabhängigkeiten und eine geeignete Auswertungsreihenfolge ableitet. Da die Anweisungen auch Aufrufe von externen Funktionen enthalten können, wird eine Verwendung von in der Zielsprache Modula-2 implementierten abstrakten Datentypen in der Attributgrammatik möglich, die in getrennt übersetzbaren Moduln definiert sind. Die Attributberechnungen können auch Seiteneffekte enthalten. Es ist außerdem möglich künstliche Attributabhängigkeiten zu definieren, um eine bestimmte Auswertungsreihenfolge zu erzwingen, oder um bei einer Attributgrammatik die von *Ag* geforderte Eigenschaft OAG [Kastens 80] herbeizuführen, falls sie diese noch nicht besitzt.

Der in Kapitel 5.2.2 beschriebene Erweiterungsmechanismus für die Attribute und Kindknoten von Knotentypen gilt auch für die Attributberechnungen. Abgeleitete Knotentypen erben die Attributberechnungen ihrer Basistypen. Durch Angabe einer speziellen Attributberechnung für den abgeleiteten Knotentyp kann die vererbte Attributberechnung aber überschrieben werden.

5.4.2 Abstrakte Datentypen

Dieses Kapitel enthält eine kurze Beschreibung der wichtigsten abstrakten Datentypen, die in der Attributgrammatik verwendet wurden.

5.4.2.1 Spezifikation der Definitionstabelle und Bezeichneridentifikation

Ast wurde im Übersetzer *Mtc* nicht nur für die Implementierung des Strukturbaums verwendet, sondern auch für die Implementierung des Definitionstabellenmoduls *Defs*.

Für jedes im Modula-Programm deklarierte Objekt sowie für alle vordefinierten Objekte wird eine Objektbeschreibung angelegt, die alle vorhandenen Informationen über das Objekt enthält. Der Knotentyp *Object*[1] definiert die Struktur der Objektbeschreibungen:

```
Object          = [Ident: tIdent] <
  Const1        = -> [Value: tValue] .
  EnumLiteral1  = Type [Index: SHORTCARD] .
  Field1        = Type .
  Module1       = ExportList: Objects -> Objects Locals: Objects .
  Proc1         = Type -> [IsExported: BOOLEAN] Locals: Objects .
  ProcHead1     = Type .
  TypeDecl1     = -> Type .
  Var1          = Type [IsVAR: BOOLEAN] .
  ...
>.
```

1 In der abgebildeten Definition der Objektbeschreibungen fehlen die Attribute für die Codeerzeugung (s. Kap. 5.5)

Die Kindknoten bzw. Attribute, die rechts des Symbols -> stehen, werden als „non-input'' Attribute bezeichnet. Die Werte dieser Attribute werden beim Aufbau der Objektbeschreibungen noch nicht festgelegt; es wird jedoch Speicherplatz für sie reserviert, so daß ihr Wert zu einem späteren Zeitpunkt nachgetragen werden kann. Das Attribut *ExportList* ist ein Ausschnitt der vollständigen Exportliste *Objects* eines Moduls und enthält nur die von diesem Modul exportierten Typnamen. Die Behandlung einiger Attribute als „non-input'' Attribute und die „doppelte'' Exportliste sind notwendig, da beim Aufbau der Objekt- und Typbeschreibungen zyklische Abhängigkeiten auftreten, die durch einen schrittweisen Aufbau dieser Beschreibungen behandelt werden.

Betrachtet man den folgenden Ausschnitt eines Modula-Programms, dann sieht man sofort, daß die Typbäume der abstrakten Syntax in Modula-2 für eine Beschreibung der Typen innerhalb der Definitionstabelle nur wenig geeignet sind:

```
TYPE t1 = INTEGER; t2 = t1;
VAR  v1 : t1; v2 : t2;
  ...
v1 := v2;
```

Die Typen der Variablen *v1* und *v2* sind identisch; aus den Typbäumen für *t1* und *t2* ist dies aber nicht direkt ablesbar. Daher werden für die Typen Typbeschreibungen aufgebaut, deren Struktur — abgesehen von einigen Vereinfachungen — mit der Struktur der Typbäume übereinstimmt, in denen aber die Typnamen durch die Typbeschreibung dieser Typen ersetzt sind. Neben Knotentypen für die Typkonstruktoren von Modula-2 existieren auch Knotentypen für die Repräsentation von Grund- und Standardtypen sowie für eine Reihe von speziellen Typen, die nur intern im Übersetzer verwendet werden und die keine direkte Entsprechung in der Sprachdefinition von Modula-2 haben.

Die Aufgabe der Bezeichneridentifikation ist es, jedem angewandten Auftreten eines Bezeichners das zugehörige Objekt bzw. die zugehörige Objektbeschreibung zuzuordnen. Die Sprachdefinition von Modula-2 legt durch ihre Gültigkeitsbereichsregeln fest, wie die Zuordnung zu treffen ist. Modula-2 ist eine blockstrukturierte Sprache in der die Gültigkeitsbereiche der Bezeichner geschachtelt sind. Eine Definition eines Bezeichners in einem inneren Block verdeckt die Definition dieses Bezeichners in einem äußeren Block. In Modula-2 kommt zur Blockstruktur noch das Modulkonzept hinzu, welches eine explizite Kontrolle der Gültigkeitsbereiche von Bezeichnern mit Hilfe von Import- und Exportanweisungen ermöglicht. Innerhalb der Attributgrammatik werden den Knotentypen der abstrakten Syntax bei Bedarf sogenannte Umgebungsattribute zugeordnet, welche die an der jeweiligen Stelle gültigen Definitionen repräsentieren. Die Struktur der Umgebungsattribute wird in der *Ast*-Spezifikation durch den Knotentyp *Env* beschrieben:

```
Env = Objects HiddenEnv: Env .
```

Objects sind die im aktuellen Block definierten Objekte, *HiddenEnv* enthält die in äußeren Blöcken definierten Objekte.

Die Identifikation in der Attributgrammatik erfolgt durch den Aufruf der ebenfalls im Modul *Defs* definierten Funktion

```
Identify : Ident × Env → Object .
```

die jedem Bezeichner mit Hilfe des gültigen Umgebungsattributs das zugehörige Objekt zuordnet.

5.4.2.2 Auswertung konstanter Ausdrücke und Repräsentation ihrer Werte

Wie in Kapitel 4 erläutert, müssen für die Übersetzung nach C die Werte einer Reihe von konstanten Ausdrücken bekannt sein. Zu diesem Zweck existiert der Modul *Values*, welcher Typdeklarationen zur Repräsentation der Werte von konstanten Ausdrücken und eine Operation für ihre Auswertung enthält. Den Objektbeschreibungen der Konstanten in der Definitionstabelle und einigen Baumknoten wie z.B. den Fallmarken wird ein Attribut zugeordnet, welches den Wert des zugehörigen konstanten Ausdrucks repräsentiert. Die Funktion

```
CompConst : Tree × Env → Value .
```

die in der Attributgrammatik aufgerufen wird, berechnet aus dem Baum für den konstanten Ausdruck und der aktuell gültigen Umgebung, die für den Zugriff auf die Werte von benannten Konstanten benötigt wird, den Wert des konstanten Ausdrucks.

Um den Aufwand für die Implementierung des Moduls *Values* zu reduzieren, arbeitet *Values* nicht interpretativ, sondern mit einer direkten Ausführung der Operationen. Die meisten möglichen Fehler, wie etwa Division durch Null, werden aber durch entsprechende Abfragen abgefangen. Arithmetische Überläufe werden allerdings nicht erkannt. Eine Auswertung von Ausdrücken, die ganzzahlige Konstanten im Bereich MAX(INTEGER) + 1 .. MAX(CARDINAL) enthalten, ist nicht möglich. *CompConst* liefert in diesem Fall aber einen definierten Wert zurück, der dies anzeigt.

5.4.2.3 Operationen auf Typen

Die Definition der Typbeschreibungen und die Prozeduren für ihren Aufbau sowie Prozeduren für den Zugriff auf in den Typbeschreibungen enthaltene Informationen sind Teil des Moduls *Defs*.

Der Modul *Types* enthält alle weiteren Informationen über die Typen von Modula-2. Dabei handelt es sich insbesondere um Operationen für die Typbestimmung in Ausdrücken.

Außerdem enthält *Types* auch die meisten Details über die Abbildung der Typen von Modula-2 nach C, wie sie für die Codeerzeugung, insbesondere im Zusammenhang mit der Erzeugung von expliziten Typumwandlungen, benötigt werden. Die in Kapitel 4.4.1 erwähnte Tabelle, die für eine Auswertung der Standardfunktionen SIZE, TSIZE, MIN und MAX benötigt wird, ist ebenfalls Teil des Moduls *Types*.

5.4.3 Die Attributgrammatik

Die folgende Beschreibung der Attributgrammatik ist nur eine auszugsweise Darstellung einiger interessanter Aspekte dieser Attributgrammatik. Eine ausführliche Darstellung der Spezifikation der semantischen Analyse mit Attributgrammatiken und die Lösung von typischen

Problemen in diesem Zusammenhang kann in [Waite 84] gefunden werden.

5.4.3.1 Aufbau der Definitionstabelle und Berechnung von Umgebungsattributen

In Modula-2 existieren eine Reihe von Regeln, die den Aufbau von Objekt- und Typbeschreibungen und die Berechnung von Umgebungsattributen erschweren und die dazu führen, daß die Objekt- und Typbeschreibungen nicht in einem einzigen Schritt aufgebaut werden können, da dies zu zyklischen Abhängigkeiten in der Attributgrammatik führen würde.

Neben den ,,rekursiven'' Typdeklarationen (s. Kap. 4.3) ist hier insbesondere die Tatsache zu nennen, daß in den Deklarationen eines Blocks Typnamen verwendet werden können, die im gleichen Block textuell später deklariert sind. Der Aufbau von Objektbeschreibungen für die in einem Block deklarierten Objekte erfordert daher ein Umgebungsattribut zur Identifikation dieser Typnamen. Für die Berechnung dieses Umgebungsattributs werden aber umgekehrt die Objektbeschreibungen der in diesem Block deklarierten Objekte benötigt.

Die Lösung dieses Problems besteht darin, die Objekt- bzw. Typbeschreibungen schrittweise aufzubauen und für die Deklarationen eine Reihe von aufeinanderfolgenden Umgebungsattributen zu berechnen, die jeweils etwas mehr Informationen über die deklarierten Objekte enthalten.

Da die meisten der oben genannten Probleme durch die Typdeklarationen entstehen, werden zuerst vorläufige Objektbeschreibungen für die benannten Typen und vollständige Typbeschreibungen für alle Typknoten der abstrakten Syntax aufgebaut, die in den Objektbeschreibungen verwendet werden können.

Zunächst wird für die Deklarationen aller Blöcke ein Attribut *Objects1* berechnet, welches eine Liste von Objektbeschreibungen für Typen darstellt, die im Prinzip als einzige Information die im entsprechenden Block deklarierten Typnamen enthält. In diese Objektliste werden für die Behandlung des Modulkonzepts noch zusätzlich Objektbeschreibungen der Moduln mit einer vorläufigen Exportliste aufgenommen, die alle vom Modul exportierten Typnamen umfaßt. Die Behandlung des Modulkonzepts wird in einem der folgenden Abschnitte noch näher erläutert. Diese Objektlisten werden zu einem ersten Umgebungsattribut *Env1* kombiniert, welches in Typdeklarationen die Zuordnung von Objektbeschreibungen zu den Typnamen ermöglicht. Mit der Berechnung der beiden Attribute *Objects2* und *Env2* wird der Prozess des Aufbaus von Typbeschreibungen vervollständigt. Wie das im Detail funktioniert, wird im nächsten Kapitel beschrieben.

Jetzt erst kann eine Objektliste *Objects3* aufgebaut werden, die die vollständigen Objektbeschreibungen der Typen, Variablen, Prozeduren und Moduln mit der kompletten Exportliste enthält. Für die Konstanten fehlt in *Objects3* noch der Wert der Konstanten, da hier ein ähnliches Problem wie für die Typen auftritt. Zur Auswertung der Konstanten *a* in der folgenden Deklaration

```
CONST b = 1;
CONST a = b * 2;
```

wird der Wert der Konstanten *b* benötigt. Hier sind in Modula-2 die Probleme im Gegensatz zu den Typen nicht so groß, da für die in Konstantendeklarationen verwendeten Konstantennamen immer gelten muß, daß diese Namen textuell vor dieser Deklaration definiert sein müssen. Zyklische Abhängigkeiten der Konstantendeklarationen können daher in Modula-2 nicht auftreten.

Mit der Objekliste *Objects3* wird die Umgebung *Env3* aufgebaut, die bis auf die Werte der Konstanten vollständig ist.

Mit Hilfe der beiden Attribute *Objects4In* und *Objects4Out* wird schließlich für die Deklarationen aller Blöcke eine weitere Objektliste aufgebaut, wobei die zu den Konstantendeklarationen dazugehörigen konstanten Ausdrücke in der textuellen Reihenfolge der Deklarationen mit Hilfe der Umgebung *Env3* und der Funktion *CompConst* ausgewertet werden. Die Werte der Konstanten werden dabei in deren Objektbeschreibungen übernommen.

Mit der Objektliste *Objects4Out* kann jetzt die vollständige Umgebung *Env4* berechnet werden, die in den Anweisungen für die Bezeichneridentifikation verwendet wird.

Vorstehend war die Rede von schrittweisem Aufbau der Objekt- und Typbeschreibungen. In der Attributgrammatik werden selbstverständlich jeweils neue Attribute berechnet, die den bei diesem schrittweisen Aufbau entstehenden Teilergebnissen entsprechen. Durch die Möglichkeit von kontrollierten Seiteneffekten und dem Einsatz von Zeigertypen (abstrakte Datentypen) für die Attributwerte, können aber vorhandene Beschreibungen erweitert werden. Damit entfällt die Notwendigkeit mehrere Objekt- bzw. Typbeschreibungen für das gleiche Objekt bzw. den gleichen Typ aufbauen und eventuell noch eine gemeinsame Repräsentation für diese finden zu müssen. Außerdem wird die Implementierung hinsichtlich Laufzeit und Speicherbedarf effizienter. Der Nachteil dieser Lösung ist allerdings, daß man sich wegen der Seiteneffekte über die Attributabhängigkeiten und die Auswertungsreihenfolge Gedanken machen muß und eventuell geeignete Attribute und Attributabhängigkeiten einführen muß, die nur die Aufgabe haben, eine bestimmte Auswertungsreihenfolge der Attribute und damit eine korrekte Reihenfolge der Seiteneffekte zu erzwingen.

5.4.3.2 Aufbau von Typbeschreibungen

Allen Typknoten der abstrakten Syntax wird in der Attributgrammatik eine Typbeschreibung als abgeleitetes Attribut zugeordnet. Um die oben genannten zyklischen Abhängigkeiten der Typbeschreibungen behandeln zu können, erfolgt deren Aufbau in zwei Schritten. Das Attribut *Type2* ist die vollständige Typbeschreibung; ein weiteres Attribut *Type1* ist eine vorläufige und unvollständige Typbeschreibung, welche während des Aufbaus der vollständigen Typbeschreibungen als Zwischenergebnis auftritt.

Folgender Ausschnitt der Attributgrammatik demonstriert den Prozeß des Aufbaus von Typbeschreibungen (vgl. Anh. A):

```
Type    = { Type2 AFTER Env2;                                  } .
Pointer = { Type1  := mPointer1 ();
            Type2  := mPointer2 (Type1, TargetType:Type2); } .
TypeId0 = { Object := Identify (Ident, Env1);
            Type1  := mQualident1 (Object);
            Type2  := GroundType (Type1);                  } .
```

Die oben beschriebene Umgebung *Env1* ermöglicht es, den Typnamen die zugehörigen Objektbeschreibungen zuzuordnen, die allerdings noch keine Typbeschreibung für den Typ enthalten. *Qualident1* bildet eine vorläufige Typbeschreibung für Typnamen, die einen Verweis auf die Objektbeschreibung des Typnamens enthält. Für andere Typknoten wird eine vorläufige Typbeschreibung angelegt, die zwar der endgültigen Typbeschreibung entspricht, die aber nur

die Information enthält, um welche Art von Typ es sich handelt. Die übrigen Informationen über den Typ, wie z.B. bei Zeigertypen der Bezugstyp, werden als „non-input'' Attribute behandelt.

In einem vollständigen Durchlauf durch alle Deklarationen des Programms werden die vorläufigen Typbeschreibungen *Type1* in die Objektbeschreibungen der benannten Typen aufgenommen (Berechnung des Attributs *Objects2*). *Objects2* wird für die Berechnung eines zweiten „Umgebungsattributs'' *Env2* verwendet, dessen einzige Funktion es ist, Attributabhängigkeiten zu definieren, die garantieren, daß die vollständigen Typbeschreibungen *Type2* erst berechnet werden, wenn alle vorläufigen Typbeschreibungen in die Objektbeschreibungen der benannten Typen eingetragen wurden. Wegen der künstlichen Attributabhängigkeit *Type2* AFTER *Env2* wird *Type2* erst nach *Env2* berechnet.

Mit Hilfe der im Modul *Defs* definierten Funktion *GroundType* kann man dann aus der vorläufigen Typbeschreibung eines Typnamens die endgültige Typbeschreibung dieses Typs bestimmen. *GroundType* ist folgendermaßen definiert (vereinfacht ohne Fehlerbehandlung):

```
PROCEDURE GroundType    (Type: tType): tType;
BEGIN
  IF Type^.Kind = Qualident1 THEN
    RETURN GroundType (Type^.Qualident1.Object^.TypeDecl1.Type);
  END;
  RETURN Type;
END GroundType;
```

Durch rekursive Aufrufe werden die Typ- bzw. Objektbeschreibungen solange traversiert bis eine Typbeschreibung für einen Typnamen gefunden wird, die keine vorläufige Typbeschreibung der Form *Qualident1* ist (Typgleichsetzungen können über eine beliebige Anzahl von Stufen geschrieben werden). *GroundType* liefert die so gefundene Typbeschreibung für den Typnamen an den Aufrufer zurück.

Der Aufbau der vollständigen Typbeschreibung *Type2* erfolgt dann schließlich dadurch, daß die noch fehlenden Informationen, wie z.B. bei Zeigertypen die Typbeschreibung des Bezugstyps *TargetType*:*Type2*, in die vorläufige Typbeschreibung *Type1* eingetragen wird. Diese vollständigen Typbeschreibungen können dann bei der Berechnung des Attributs *Objects3* in die Objektbeschreibungen der benannten Typen aufgenommen und als Typbeschreibung in den Objektbeschreibungen der Variablen, Prozeduren, usf. verwendet werden.

5.4.3.3 Behandlung des Modulkonzepts

Zusätzlich zur Blockstruktur ermöglicht Modula-2 mit dem Modulkonzept eine explizite Kontrolle des Gültigkeitsbereichs von Bezeichnern durch Import- bzw. Exportanweisungen. Eine wichtige Sonderregel in diesem Zusammenhang ist, daß mit dem Import- bzw. Export eines Aufzählungstyps auch automatisch die zugehörigen Aufzählungsliterale importiert bzw. exportiert werden. Die vordefinierten Objekte von Modula-2 müssen nicht explizit importiert werden, sondern sind automatisch in jedem Modul sichtbar. Bei der Behandlung von Implementierungsmoduln muß außerdem beachtet werden, daß alle im zugehörigen Definitionsmodul definierten Konstanten, Typen und Variablen im Implementierungsmodul ebenfalls automatisch sichtbar sind.

Der folgende Ausschnitt der Attributgrammatik zeigt, wie die oben genannten Regeln in der Attributgrammatik behandelt werden (vgl. Anh. A):

```
ProgMod = { Import:Env2 := Env3;
            DefObjects3 := Filter (GetExport2 (Identify (Ident, Env3)));
            Decls:Env3  :=
              mEnv (UNION (UNION (UNION (Predefs, Import:Objects2),
                    Decls:Objects3), DefObjects3), NoEnv);            } .
From    = { Object2     := Identify (Ident, Env2);
            ImpIds:Env2 := mEnv (GetExport2 (Object2), NoEnv);
            Objects2    := UNION (ImpIds:Objects2, Next:Objects2);    } .
ImpIds1 = { Object2     := Identify (Ident, Env2);
            Type        := GetType (Object2);
            Objects2    := {
          IF (Object2^.Kind = TypeDecl1   ) AND
             (Type^.Kind    = Enumeration1) THEN
            Objects2    :=
              mElmt (Ident, Object2, UNION (Type^.Enumeration1.Objects,
                     Next:Objects2));
          ELSE
            Objects2    := mElmt (Ident, Object2, Next:Objects2);
          END;          };                                            } .
```

Das Attribut *DefObjects3* ist eine Objektliste, welche die im zum Implementierungsmodul dazugehörigen Definitionsmodul definierten Konstanten, Typen und Variablen enthält. Die Berechnung erfolgt dadurch, daß in der Umgebung des Implementierungsmoduls zunächst durch einen Aufruf von *Identify* die Objektbeschreibung des (Definitions-)Moduls bestimmt und aus dieser dann die Liste der exportierten Objekte entnommen wird. Da diese Exportliste zunächst noch die Definition von Prozedurköpfen und opaquen Typen enthält, die im Implementierungsmodul redeklariert werden müssen, wird mit Hilfe der im Modul *Defs* definierten Funktion *Filter* eine neue Objektliste aufgebaut, die diese nicht mehr enthält.

In einer Importanweisung der Form

```
FROM Module IMPORT Object1, ... , Objectn;
```

ist die für den Bezeichner *Module* gültige Umgebung, die Umgebung des Moduls, der die Importanweisung enthält. Die Umgebung der Bezeichner *Object1* bis *Objectn* ergibt sich aus der Liste der Objekte, die von *Module* exportiert werden. Diese Tatsache wird im obigen Ausschnitt der Attributgrammatik durch die Attributierungsregeln zur Berechnung der Umgebungsattribute *Import*:*Env2* und *ImpIds*:*Env2* reflektiert.

Die importierten Objekte werden in der Objektliste *Objects2* gesammelt. Die Attributierungsregeln für den Knotentyp *ImpIds1* zeigen dabei, wie der implizite Import von Aufzählungsliteralen behandelt wird.

Die Umgebung *Env3* der Deklarationen des Implementierungsmoduls ergibt sich schließlich aus einer Vereinigung der vordefinierten Objekte *Predefs*, der explizit importierten Objekte *Import*:*Objects2*, der Objekte aus dem Definitionsmodul *DefObjects3* und aus den lokal deklarierten Objekten *Decls*:*Objects3*. Die Tatsache, daß die äußere Umgebung eines Moduls vollständig verdeckt wird, ist aus der Verwendung der leeren Umgebung *NoEnv* als äußere Umgebung des Moduls im Aufruf von *mEnv* zu entnehmen.

5.4.3.4 Typbestimmung in Ausdrücken

Den Ausdrücken wird ein abgeleitetes Attribut *Type* zugeordnet, welches den Typ des Ausdrucks beschreibt. Die Berechnung dieses Attributs erfolgt, wie der folgende Beispielausschnitt der Attributgrammatik zeigt (vgl. Anh. A), mit Hilfe der in *Types* definierten Operationen auf Typen und den in *Defs* enthaltenen Operationen für den Zugriff auf Objekt- und Typbeschreibungen.

```
Binary    = { Type := ResultType (Operator, Lop:Type, Rop:Type);  } .
RealConst = { Type := TypeREAL;                                   } .
Qualid0   = { Type := GetType (Object);                           } .
Subscript = { Type := GetElemType (Designator:Type);              } .
Deref     = { Type := GetTargetType (Designator:Type);            } .
FuncCall  = { Type := {
              IF Designator:Type^.Kind = StdProcType1 THEN
                Type := StdResultType (Designator:Type, Actuals:Types);
              ELSIF Designator:Type^.Kind = ProcType1 THEN
                Type := GetResultType (Designator:Type);
              ELSE /* may be a type transfer function  */
                Type := Designator:Type;
              END;   };                                           } .
```

Die in *Types* definierte Funktion *StdResultType* ermittelt aus dem Typ einer Standardprozedur und den Typen der aktuellen Parameter den Resultattyp des Aufrufs dieser Standardprozedur. Die Typen der aktuellen Parameter sind notwendig, da ein Teil der Standardprozeduren wie z.B. ABS, MIN und MAX in Modula-2 überladen sind.

Die Typen der Ausdrücke werden für die Codeerzeugung benötigt, um z.B. den korrekten C-Operator für überladene Modula-Operatoren einsetzen zu können.

5.5 Berechnung von Attributen für die Codeerzeugung

Für die Durchführung der Codeerzeugung müssen weitere Informationen über das Quellprogramm abgeleitet werden, die über die in der semantischen Analyse berechneten Attribute hinausgehen und die für eine korrekte Übersetzung nach C benötigt werden. Diese Attributberechnungen sind ebenfalls Teil der im vorigen Kapitel beschriebenen Attributgrammatik.

Zur besseren Gliederung der Attributgrammatik ermöglicht *Ag* deren Unterteilung in sogenannte Moduln, die es erlauben, logisch zusammengehörige Attributdeklarationen und die zugehörigen Attributberechnungen auch textuell zusammenzufassen. Daher sind die Attributberechnungen für die semantische Analyse und die Berechnung von Attributen für die Codeerzeugung textuell eindeutig getrennt. Diese Trennung erleichtert eine eventuelle Wiederverwendung des Front-Ends von *Mtc*.

Die Trennung der Attributberechnungen in semantische Analyse und Berechnung von Attributen für die Codeerzeugung ist auf den ersten Blick teilweise etwas willkürlich, da z.B. die in der semantischen Analyse bestimmten Typen der Ausdrücke auch für die Codeerzeugung benötigt werden. Die Unterscheidung wurde aber auf folgender Basis getroffen: Alle Attribute, die auch benötigt würden, falls man die semantische Korrektheit der Modula-Programme überprüfen wollte, werden zur semantischen Analyse gezählt. Alle anderen Attribute, die ausschließlich für

eine Übersetzung nach C notwendig sind, zählen zu den Attributen für die Codeerzeugung. Im Folgenden sollen die wichtigsten dieser Attribute übersichtsartig vorgestellt werden.

Für die Operatoren der Sprache Modula-2 ist eine Operatoridentifikation notwendig, da eine Reihe von Operatoren in Modula-2 überladen sind und daher auf unterschiedliche C-Operatoren abgebildet werden müssen. Die Knotentypen, welche die Operatoren als Attribute enthalten (vgl. Anh. A), erhalten daher ein Attribut *COperator*, welches den zugehörigen C-Operator beschreibt. Die Berechnung dieses Attributs erfolgt mit Hilfe des Modula-Operators und den Typen der Operanden des zugehörigen Ausdrucks.

Wie in Kapitel 4 erläutert, muß für eine Reihe von Modula-Ausdrücken bekannt sein, ob diese Ausdrücke in C konstant sind. Sie erhalten daher ein boolesches Attribut *IsCConst*.

Für die Übersetzung des Zugriffs auf lokale Variablen von statisch umfassenden Prozeduren nach C (s. Kap. 4.4.4) existieren die folgenden Attribute:

- Das boolesche Attribut *IsGlobalPtr* wird jedem Bezeichner in Ausdrücken oder Anweisungen zugeordnet. Das Attribut ist wahr, falls der Bezeichner eine lokale Variable einer statisch umfassenden Prozedur bezeichnet und daher der Zugriff in C mit Hilfe einer globalen Zeigervariablen realisiert werden muß.
- Das Attribut *LocalPtrs*, welches jeder Deklaration einer Prozedur *p* zugeordnet wird, ist eine Liste aller lokalen Variablen von *p*, die in lokal zu *p* deklarierten Prozeduren benutzt werden. Es wird benötigt, um für die nach C übersetzte Prozedur *p* den entsprechenden Code für eine Realisierung des Zugriffs auf diese lokalen Variablen zu erzeugen.
- Programm- und Implementierungsmoduln wird ein Attribut *GlobalPtrs* zugeordnet, welches eine Vereinigung der Attribute *LocalPtrs* ist und für die Erzeugung der oben erwähnten globalen Zeigervariablen benutzt wird.

Die Übersetzung der varianten Teile der Verbunde nach C erfordert die Berechnung von zusätzlichen Komponentenselektoren (s. Kap. 4.4.2.7). Die entsprechenden Selektoren werden den varianten Teilen der Verbunde im Strukturbaum zugeordnet und auch in die Objektbeschreibungen der zugehörigen Verbundkomponenten übernommen, um Zugriffe auf diese Komponenten korrekt übersetzen zu können.

5.5.1 Umbenennung von Bezeichnern

In Kapitel 4 wurde ausführlich dargelegt, welche Bezeichner im C-Programm für die Modula-Objekte verwendet werden. Es wurden im wesentlichen 2 Fälle unterschieden:

- Der Bezeichner eines von einem globalen Modul exportierten Objekts wird in C in der qualifizierten Form *Modulname_Bezeichner* geschrieben.
- Alle übrigen Bezeichner werden direkt aus dem Modula-Programm übernommen, werden aber bei Bedarf zur Vermeidung von Namenskonflikten mit einem Präfix *C_nnn_* versehen.

In der Attributgrammatik wird für jedes im Modula-Programm deklarierte Objekt ein Attribut *CIdent* berechnet, welches den C-Bezeichner dieses Objekts darstellt. Dieser Bezeichner muß auch in die Objektbeschreibung übernommen werden, um bei jedem angewandten Auftreten des Bezeichners den richtigen C-Bezeichner einsetzen zu können.

Zur Entdeckung und Vermeidung von möglichen Namenskonflikten existiert der Modul *UniqueIds*, welcher die Verwaltung der im C-Programm verwendeten unqualifizierten Bezeichner entsprechend den Gültigkeitsbereichsregeln der Sprache C übernimmt. Qualifizierte Bezeichner der Form *Modulname_Bezeichner* brauchen nicht von *UniqueIds* verwaltet zu werden, da für sie auf Grund der Eindeutigkeit von globalen Modulnamen keine Namenskonflikte entstehen können. Die wichtigsten Operationen des Moduls *UniqueIds* sind:

```
NameConflict : Ident × IdentClass × Idents → BOOLEAN .
DeclareIdent : Ident × IdentClass × Idents → Idents .
EnterProc    : Idents → Idents .
LeaveProc    : Idents → Idents .
```

NameConflict überprüft, ob die Verwendung eines bestimmten Bezeichners zu einem Namenskonflikt mit einem bereits verwendeten Bezeichner führen würde. *DeclareIdent* deklariert einen Bezeichner als im C-Programm verwendet. *EnterProc* und *LeaveProc* werden zu Beginn und am Ende der Behandlung der lokalen Deklarationen einer Prozedur aufgerufen.

Die Bezeichner werden entsprechend ihrer Art bzw. der Art des zugehörigen Objekts in folgende Klassen eingeteilt (vgl. Kap. 4):

- Schlüsselwörter: *UniqueIds* behandelt eine Reihe von Bezeichnern als Schlüsselwörter (Bezeichner für vom Übersetzer vordefinierte Objekte und Schlüsselwörter der Sprache C). *NameConflict* liefert für einen Schlüsselwortbezeichner immer wahr, was dazu führt, daß der Bezeichner auf jeden Fall durch einen Präfix umbenannt wird.
- Konstanten- und Typbezeichner, die im C-Programm innerhalb einer Quelldatei nur genau einmal auftreten dürfen.
- Prozedurbezeichner: alle Prozedurbezeichner sind in C globale Bezeichner.
- Bezeichner von Verbundkomponenten.
- Variablenbezeichner, die nicht in lokalen Moduln deklariert sind.
- Variablenbezeichner, die in lokalen Moduln deklariert sind.

Die Variablen, die in einem in Modula-2 lokal zu einer Prozedur deklarierten Modul enthalten sind, werden in C zu lokalen Variablen dieser Prozedur. Im folgenden Beispiel

```
PROCEDURE p;
  ...
PROCEDURE q;
  MODULE l;
    ...
    VAR p: INTEGER;
    ...
  END l;
BEGIN
  p;
END q;
```

muß bei der Übersetzung nach C die lokale Variable *p* (in C lokal zu *q* deklariert) umbenannt werden, da sonst der globale Prozedurbezeichner *p* im Rumpf von *q* verdeckt würde. Aus diesem Grund werden in lokalen Moduln deklarierte Variablen von *UniqueIds* immer behandelt, als würde in C für diese Variablen sowohl eine lokale Deklaration in der entsprechenden Prozedur

als auch eine globale Deklaration existieren. Diese Behandlung führt dazu, daß im obigen Beispiel die Variable *p* umbenannt wird, da im gleichen Gültigkeitsbereich bereits der Prozedurbezeichner *p* deklariert wurde. Bei einer Behandlung von *p* ausschließlich als lokale Variable von *q* würde kein Namenskonflikt entdeckt, da *UniqueIds* nur Informationen über die Deklaration der Bezeichner und nicht auch über deren Anwendung enthält. Für in Modula-2 lokal in einer Prozedur deklarierte Variablen kann das Problem nicht auftreten, da sowohl in Modula-2 als auch in C identische globale Bezeichner durch diese Deklaration verdeckt werden.

Die meisten Operationen des Moduls *UniqueIds* haben einen Seiteneffekt. In der Attributgrammatik garantieren die beiden Attribute *IdsIn* und *IdsOut*, welche die jeweils aktuelle Menge von C-Bezeichnern repräsentieren, daß diese Seiteneffekte in der richtigen Reihenfolge ausgeführt werden und insbesondere, daß der C-Bezeichner *CIdent* zum richtigen Zeitpunkt berechnet wird. Der große Vorteil dieser Lösung mit Seiteneffekten ist, daß die Operationen des Moduls *UniqueIds* sehr zeit- und speichereffizient programmiert werden können. Insbesondere ist der Aufwand für *NameConflict* und *DeclareIdent* in der Größenordnung O(1).

5.6 Codeerzeugung

Eingabe für die Codeerzeugung ist der attributierte Strukturbaum des Modula-Programms, der alle für die Übersetzung nach C notwendigen semantischen Informationen enthält. Die Ausgabe der Codeerzeugung ist ein C-Programm bzw. eine C-Definitionsdatei für die aktuelle Übersetzungseinheit.

Durch Angabe einer Option kann man *Mtc* dazu veranlassen, auch Definitionsdateien für alle transitiv importierten Definitionsmoduln zu erzeugen. Diese Option ist insbesondere dann nützlich, wenn man — z.B. für Testzwecke — zunächst nur einen einzelnen Implementierungs- oder Programmodul nach C übersetzen und diesen dann vom C-Übersetzer in ein Objektprogramm übersetzen lassen möchte. Für diese Übersetzung benötigt der C-Übersetzer die Definitionsdateien aller transitiv importierten Definitionsmoduln, da diese mit *#include*-Anweisungen in das C-Programm eingefügt werden. Durch die oben genannte Option wird es überflüssig diese Definitionsmoduln einzeln nach C zu übersetzen; insbesondere braucht man nicht mühsam abzuleiten, welche (Definitions-)Moduln denn tatsächlich importiert werden.

5.6.1 Spezifikation des Codegenerators

Der Codegenerator, der im Modul *Code* enthalten ist, wurde mit Hilfe des Generators *Estra* [Vielsack 89] aus einer formalen Spezifikation der Codeerzeugung generiert. *Estra* ist ein Werkzeug für die Spezifikation und Implementierung der Transformation von attributierten Bäumen. Im Folgenden wird kurz darauf eingegangen, wie eine solche Spezifikation und die daraus abgeleitete Implementierung im Prinzip aussehen. Die Spezifikation der Codeerzeugung wird nicht im Detail erörtert, da es sich dabei lediglich um eine Umsetzung der in Kapitel 4 verbal beschriebenen Abbildung von Modula-2 nach C in eine formale Spezifikation für *Estra* handelt.

Die Beschreibung der Transformation besteht im wesentlichen aus zwei Teilen: Einer Baumgrammatik, welche die Struktur der zu transformierenden Bäume beschreibt und einer oder mehreren Funktionen, die die Abbildung der Bäume beschreiben. Eine Funktion besteht aus der Festlegung des Definitionsbereichs, aus einer Angabe von synthetisierten und vererbten Attributen

sowie aus Vorschriften wie die Transformation durchgeführt werden soll. Der Definitionsbereich legt fest, auf welche Knotentypen die Funktion anwendbar ist. Die Vorschriften, die die Abbildung im Einzelnen festlegen, bestehen aus einem Muster, welches angibt auf welche Teilbäume die betreffende Vorschrift angewandt werden kann und aus Anweisungen, die festlegen wie der Teilbaum behandelt werden soll. Diese Anweisungen können insbesondere Aufrufe von Funktionen für die Transformation von Unterbäumen enthalten. Die Anwendbarkeit bestimmter Vorschriften kann durch Bedingungen eingeschränkt werden. Damit wird es möglich, die vom Attributauswerter berechneten semantischen Informationen für die Festlegung der Transformation zu berücksichtigen. Für eine Auflösung von Mehrdeutigkeiten dient die Angabe von Kosten für die Anwendung von Vorschriften. Werden diese Kosten durch eine Konstante festgelegt, so ergeben sich die Kosten der Anwendung einer Vorschrift aus der Summe dieser Konstanten und den Kosten der Funktionsaufrufe für die Transformation der Unterbäume, die in den Anweisungen enthalten sind. Wird dies nicht gewünscht, so besteht auch die Möglichkeit die Kosten durch einen Ausdruck der Quellsprache (in geschweiften Klammern) direkt festzulegen. Bei der Transformation wird immer die kostengünstigste Vorschrift angewandt.

Folgender Ausschnitt der Funktion für die Spezifikation der Codeerzeugung für Ausdrücke soll obenerwähntes noch einmal erläutern:

```
FUNCTION CodeExpr  Prec: SHORTCARD  ->  /Expr, Elems/
  ...
Subscript   (Qualid0 (), Index: Expr)
            CONDITION { IsOpenArray (Qualid0.Object) }
            COSTS { 1 }
{
    CodeExpr (Qualid0, pSubscript); @[@ CodeExpr (Index, pMinPrec); @]@
}
  ...
```

Das vererbte Attribut *Prec* ist der C-Vorrang des Operators im Vaterausdruck und wird benötigt, um die Ausdrücke in C korrekt zu klammern. Da die Indexoperation in C den höchsten Vorrang hat, ist in obigem Beispiel eine Klammerung nicht notwendig. Die Bedingung in der Vorschrift schränkt ihre Anwendbarkeit auf offene Felder ein (vgl. Kap. 4.5.1). Die Anweisungen legen die Abbildung des Zugriffs auf offene Felder fest und enthalten Funktionsaufrufe für die Transformation der Unterbäume. @[@ ist eine Anweisung für einen Präprozessor, der für eine bessere Lesbarkeit der Spezifikation und zur Vereinfachung der Schreibweise für Anweisungen zur Ausgabe von C-Programmtext mit Hilfe des Zeileneditors *sed* implementiert wurde, die in WriteC (f, '['); umgesetzt wird.

Da die meisten Attribute bereits vom Attributauswerter berechnet werden, wird von der von *Estra* angebotenen Möglichkeit für die Attributierung nur wenig Gebrauch gemacht. Eine unterschiedliche Transformation bestimmter Knotentypen, abhängig vom Ort des zugehörigen Teilbaums, wie z.B. die unterschiedliche Abbildung von Deklarationen in Definitionsmoduln bzw. Implementierungs- und Programmoduln (s. Kap. 4.8) oder auch die mehrfache Transformation eines Teilbaums auf unterschiedliche Art und Weise, wie sie z.B. für die Umordnung der Deklarationen (s. Kap. 4.3) benötigt wird, wird dadurch erreicht, daß für jede dieser unterschiedlichen Transformationen eine eigene Funktion existiert, die dann an den entsprechenden Stellen aufgerufen wird.

Die Spezifikation des Codegenerators wird von *Estra* in eine Implementierung umgesetzt. Diese Implementierung führt die Transformation in zwei Schritten durch. Zunächst wird in einem vorbereitenden Schritt festgelegt, welche Vorschriften für die Transformation des vorhandenen Baums anzuwenden sind. Dazu wird bei einem Bottom-Up-Baumdurchlauf geprüft, welche Muster auf welche Knoten (Teilbäume) passen und ob die zugehörigen Bedingungen erfüllt sind. Für jede existierende Funktion wird aus den anwendbaren Vorschriften diejenige mit den geringsten Kosten ausgewählt und im Knoten zusammen mit ihren Kosten festgehalten. Die eigentliche Durchführung der Transformation erfolgt im zweiten Schritt durch Anwendung der ersten Funktion auf die Wurzel des Baums. Die Transformation erfolgt dann unter Berücksichtigung der im ersten Schritt festgelegten Vorschriften durch Ausführung der in den Anweisungen dieser Vorschriften enthaltenen Funktionsaufrufe für die Teilbäume.

5.6.2 Nachoptimierung des Codegenerators

Estra ist ein im Rahmen einer Diplomarbeit entwickelter Prototyp. Die Spezifikation des Codegenerators von *Mtc* war die erste größere Anwendung von *Estra* und es lagen aus diesem Grund bisher noch keine praktischen Erfahrungen mit großen Anwendungen vor. Nachdem eine erste (Teil-)Spezifikation der Codeerzeugung vorlag, ergaben Tests, daß der von *Estra* aus dieser Spezifikation erzeugte Codegenerator einen extrem hohen Speicherbedarf besaß, der trotz der heutigen relativ großen Hauptspeichergrößen für große Eingabeprogramme einen Trashing-Effekt zur Folge hatte. Der hohe Hauptspeicherbedarf ergibt sich daraus, daß der Codegenerator bei der Vorbereitung der Transformation (s. Kap. 5.6.1) in jedem Knoten für jede existierende Funktion die anwendbare Vorschrift (4 Byte) und die Kosten für diese Anwendung (4 Byte) ablegt. Die erste Version der Spezifikation der Codeerzeugung bestand aus ca. 40 Funktionen, d.h. für jeden Baumknoten wurden etwa 320 Byte dynamischer Speicher angefordert.

Um die Codeerzeugung nicht doch noch aus praktischen Gründen ,,von Hand'' programmieren zu müssen, war es notwendig, diesen Speicherbedarf durch eine Veränderung der Spezifikation und durch eine automatische Nachoptimierung des von *Estra* erzeugten Codegenerators zu reduzieren.

Ein erster Schritt zur Reduktion des Speicherbedarfs war, die Anzahl der Funktionen in der Spezifikation und damit die in jedem Knoten abgelegte Menge von Informationen zu reduzieren. Dies ist aber aus den folgenden Gründen nicht unbegrenzt möglich bzw. sinnvoll:

- Für bestimmte Knotentypen müssen verschiedene Funktionen existieren, da sie wie oben beschrieben mehrfach auf unterschiedliche Art und Weise transformiert werden müssen.
- Eine gewünschte Modularisierung der Spezifikation für Zwecke der Verständlichkeit und Wartbarkeit läßt es nicht sinnvoll erscheinen, eine Spezifikation von mehreren tausend Zeilen mit einer einzigen Funktion festzulegen, selbst wenn dies theoretisch möglich wäre.
- Beim Einsatz von Attributen sind in der Regel für verschiedene Knotentypen auch unterschiedliche Attribute und somit verschiedene Funktionen notwendig.

Soweit dies logisch vertretbar erschien, wurden einzelne Funktionen zusammengefaßt, so daß die endgültige Spezifikation der Codeerzeugung nur noch aus 22 Funktionen besteht. Allerdings hat sich die Lesbarkeit der Spezifikation durch diese Zusammenfassung verschlechtert.

Ein Ansatz für eine automatische Nachoptimierung ergab sich aus folgender Überlegung: Die Kosten werden von *Estra* dazu benutzt, die kostengünstigste Vorschrift auszuwählen. Die Kosten für die Transformation eines Baumknotens hängen dabei im allgemeinen — wie es z.B. in Codegeneratoren für Maschinensprache sinnvoll ist — von den Kosten für die Transformation beliebiger Unterbäume ab. Daher müssen die Kosten für die Anwendung der einzelnen Vorschriften im Baum abgelegt werden. Für eine Abbildung des attributierten Strukturbaums nach C kann man aber immer bereits lokal an einem bestimmten Knoten entscheiden, wie dieser nach C abzubilden ist, ohne dabei die Abbildung der Unterbäume bzw. deren Kosten berücksichtigen zu müssen. Die Kosten werden für eine Abbildung nach C nur benötigt, um bei der Vorbereitung der Transformation für jeden Knoten lokal die kostengünstigste Vorschrift auswählen zu können. Aus diesem Grund würde es völlig ausreichen, die Kosten, anstatt sie im Baum abzuspeichern, in einer lokalen Variablen der Besuchsprozedur abzulegen, die die Transformation vorbereitet. Nur die Vorschrift selbst müßte dann noch im Baum gespeichert werden, um im zweiten Schritt die Transformation durchführen zu können.

In der Spezifikation wurden alle Kosten so festgelegt, daß die Kosten der Transformation eines bestimmten Knotens nicht mehr von den Kosten der Transformation der Kindknoten (Unterbäume) abhängen. Mit Hilfe des Zeileneditors *sed* wurde der von *Estra* erzeugte Codegenerator — wie oben angedeutet — automatisch nachoptimiert. Das dafür verwendete *Sed*-Skript ist relativ einfach und umfaßt nur ca. 40 Zeilen. Besonders wichtig ist, daß diese Nachoptimierung ohne Eingreifen des Benutzers automatisch durchgeführt werden kann, da es keinesfalls sinnvoll wäre, ein von einem Übersetzerbauwerkzeug generiertes Programm noch nachträglich „von Hand'' nachzuoptimieren, was dann selbstverständlich bei jeder noch so kleinen Änderung der Spezifikation durchgeführt werden müßte.

Durch die oben beschriebenen Maßnahmen konnte der Speicherbedarf der endgültigen Version des Codegenerators auf etwa 1/4 des Speicherbedarfs der ersten Version reduziert werden. Der Trashing-Effekt tritt daher für Modula-Programme in praktisch vorkommenden Größen nicht mehr (so stark) in Erscheinung. Im nächsten Kapitel wird bei der Bewertung der Werkzeuge ein Vorschlag gemacht, wie *Estra* unter Ausnutzung von in der Spezifikation enthaltenen Informationen den Speicherbedarf auf allgemeine Weise drastisch reduzieren könnte.

5.7 Fehlerbehandlung

Lexikalische Fehler wurden in der Spezifikation des Symbolentschlüsslers berücksichtigt und werden daher vom Übersetzer behandelt und gemeldet.

Wie in Kapitel 5.2.1 besprochen, besitzt der generierte Zerteiler eine automatische Fehlerbehandlung. Dem Benutzer werden daher syntaktische Fehler und die vom Zerteiler durchgeführte Fehlerreparatur gemeldet.

Bei der Behandlung der getrennten Übersetzung können zwei Fehler auftreten, die vom Übersetzer gemeldet werden:

- Ein für die Übersetzung der aktuellen Übersetzungseinheit benötigter Definitionsmodul kann nicht gefunden werden.
- Es existieren — in Modula-2 verbotene — zyklische Abhängigkeiten zwischen den Definitionsmoduln.

Tritt während der lexikalischen und syntaktischen Analyse oder bei der Behandlung der getrennten Übersetzung ein Fehler auf, dann wird die Übersetzung nach der entsprechenden Übersetzerphase abgebrochen. Da die aktuelle Übersetzungseinheit und die Definitionsmoduln aus verschiedenen Quelldateien stammen, enthalten die Fehlermeldungen neben der Angabe von Zeile und Spalte im Quelltext auch immer den Namen der zugehörigen Quelldatei.

Wie bereits besprochen, soll die semantische Korrektheit der Eingabeprogramme vom Übersetzer nicht überprüft werden. Natürlich ist es trotzdem möglich, daß ein semantisch falsches Programm als Eingabe auftritt. Daher müssen mögliche semantische Fehler, wie z.B. eine bei der Bezeichneridentifikation entdeckte fehlende Deklaration für einen Bezeichner, zumindest intern behandelt werden, auch wenn solche Fehler dem Benutzer nicht gemeldet werden. Für die Behandlung dieser Fehler wird eine für Attributgrammatiken übliche Technik angewandt: Für die Attributtypen werden spezielle Fehlerwerte eingeführt, welche im Fehlerfall von den entsprechenden Operationen zurückgeliefert werden. Außerdem erfolgt der Zugriff auf semantische Informationen wie z.B. die Typ- und Objektbeschreibungen nicht direkt, sondern durch spezielle Zugriffsoperationen, die überprüfen, ob die Beschreibungen bestimmte semantische Bedingungen erfüllen und die bei Nichterfüllung der Bedingungen entsprechende Fehlerwerte zurückliefern.

Für semantisch falsche Programme wird aber in jedem Fall C-Code erzeugt. Allerdings sind diese C-Programme in 95% der Fälle entweder semantisch oder meist sogar syntaktisch fehlerhaft und daher wird in der Regel der C-Übersetzer entdecken, daß die ursprünglichen Modula-Programme fehlerhaft waren.

Falls eines der vom Übersetzer nicht unterstützten Modula-Konstrukte im Quellprogramm auftritt, wird dies von der Codeerzeugung gemeldet.

5.8 Umfang der Implementierung des Übersetzers

Tabelle 5.3 zeigt den Umfang der für die Implementierung des Übersetzers *Mtc* verwendeten Spezifikationen und den Umfang der Quellmoduln, die aus diesen Spezifikationen erzeugt wurden.

Übersetzerteil	Spezifikation			Erzeugter Quellmodul		
	Formaler Teil	Quellcode	Summe	Def.-Modul	Impl.-Modul	Summe
Symbolentschlüssler	392	133	525	56	1320	1376
Zerteiler	934	87	1021	80	2918	2998
Strukturbaum	189	51	240	579	3234	3813
Definitionstabelle	118	985	1103	417	1549	1966
Attributauswerter	2071	159	2230	9	3591	3600
Codegenerator	2775	1030	3805	50	7448	7498
Summe	6479	2445	8924	1191	20060	21251

Tabelle 5.3: Umfang der Spezifikationen und der daraus erzeugten Quellmoduln in Zeilen

Die Zahlen für den Codegenerator beziehen sich auf die nachoptimierte Version. Vor der Nachoptimierung beträgt seine Größe 7571 Zeilen.

Tabelle 5.4 zeigt Umfang (in Zeilen) und Anzahl der Moduln, aus denen *Mtc* besteht, wobei unterschieden wird zwischen aus Spezifikationen erzeugten Moduln, Bibliotheksmoduln und „von Hand'' programmierten Moduln.

Übersetzermoduln	Anzahl	Umfang
Aus Spezifikationen erzeugte Moduln	6	21251
Bibliotheksmoduln	18	3541
„Von Hand'' programmierte Moduln	11	3871
Summe	35	28663

Tabelle 5.4: Umfang und Anzahl der Moduln von *Mtc*

Wie die beiden obigen Tabellen zeigen, besteht der überwiegende Teil des Übersetzers *Mtc* aus Moduln, die mit Hilfe von Werkzeugen aus Spezifikationen erzeugt wurden und aus wiederverwendbaren Bibliotheksmoduln. Nur ein relativ kleiner Teil des Übersetzers besteht aus Moduln, die ausschließlich „von Hand'' programmiert sind. Betrachtet man den gesamten Umfang von *Mtc*, dann wird klar, daß die Implementierung eines Programms dieses Umfangs und dieser Komplexität ohne den Einsatz von Werkzeugen und wiederverwendbarer Software kaum im Rahmen einer Diplomarbeit möglich gewesen wäre.

6. Praktische Ergebnisse

6.1 Test und erste Einsätze des Übersetzers

Da schon seit mehreren Jahren an der GMD Forschungsstelle Karlsruhe die Sprache Modula-2 und der Übersetzer MOCKA eingesetzt werden, bestand an Testprogrammen für den Übersetzer *Mtc* kein Mangel. Anfänglich noch vorhandene Schwächen in der Abbildung von Modula-2 nach C, insbesondere solche ursprünglich nicht berücksichtigten Spezialfälle, wie die in Kapitel 4.2.2 beschriebenen Probleme bei der Abbildung von großen ganzzahligen Konstanten oder von Zeichenkonstanten, konnten daher relativ schnell entdeckt und beseitigt werden.

Der erste große Test des Übersetzers *Mtc*, der auch mit dem Betreuer zu Beginn der Diplomarbeit als Abnahmetest vereinbart worden war, war die Übersetzung von *Mtc* selbst nach C. Dieser Test wurde von *Mtc* erfolgreich absolviert, so daß *Mtc* jetzt sowohl in der ursprünglichen Modula-Version als auch in einer daraus erzeugten C-Version vorliegt.

Wichtigster und anspruchsvollster Test von *Mtc* war die Übersetzung des Modula-Übersetzers MOCKA nach C.

Das erste dabei auftretende Problem war, daß MOCKA eine Möglichkeit für eine bedingte Übersetzung mit Hilfe von Übersetzerschaltern besitzt, die in den Quellen von MOCKA verwendet wird, um die verschiedenen MOCKA-Versionen für unterschiedliche Zielmaschinen zu verwalten. Diese bedingte Übersetzung wird von *Mtc* nicht unterstützt. Die MOCKA-Versionen für die SUN- und PCS-Workstations mußten daher erst aus den Quellen mit den Übersetzerschaltern erzeugt werden. Dies konnte allerdings automatisch mit Hilfe des Zeileneditors *sed* durchgeführt werden.

Das zweite auftretende Problem war, daß von MOCKA eine im Benutzerhandbuch nicht dokumentierte, aber in den Quellen benutzte Spracherweiterung vorgenommen wird: MOCKA erlaubt es das Zeichen _ in Bezeichnern wie einen Buchstaben zu verwenden. Daraufhin wurde die Spezifikation des Symbolentschlüsslers von *Mtc* so erweitert, daß *Mtc* diese Spracherweiterung auch akzeptiert. Allerdings kann es jetzt, da die Umbenennung von Bezeichnern davon ausgeht, daß dieses Zeichen in Modula-Bezeichnern nicht vorkommt, in ungünstigen Fällen zu Namenskonflikten in den erzeugten C-Programmen kommen. Bei Verwendung des Zeichens _ in Bezeichnern gibt *Mtc* daher eine entsprechende Warnung aus.

Ein weiteres Problem trat an den PCS-Workstations auf. Der Modul *SuValues* enthält Konstantendeklarationen der Form

```
CONST UPBdiv8 = MaxLongCard DIV 8;
CONST UPBmod8 = MaxLongCard MOD 8;
```

wobei *MaxLongCard* den Wert MAX(LONGCARD) hat. Trotz der in Kapitel 4.2.2 besprochenen Typumwandlungen ist der C-Übersetzer nicht in der Lage, diese Konstanten korrekt auszuwerten, da offensichtlich intern nur mit *long*-Werten gerechnet und Über- bzw. Unterlauf nicht erkannt wird. Diese Konstanten mußten daher an den PCS-Workstations in den C-Programmen „von Hand'' ausgewertet und eingesetzt werden. Der C-Übersetzer der Firma SUN ist dagegen in der Lage, diese konstanten Ausdrücke korrekt auszuwerten. *Mtc* gibt jetzt bei der Verwendung einer Konstanten im Bereich von MAX(INTEGER) + 1 .. MAX(CARDINAL) eine entsprechende Warnung ab.

Das letzte auftretende Problem war folgender Typtransfer im Modul *CgMobil*:

```
GenLongIntMode (SHORTINT (0FFFFH), LowWordMaskOp);
```

Der erste formale Parameter von *GenLongIntMode* hat den Typ LONGINT. Der Wert des aktuellen Parameters nach der Übergabe ist in Modula-2 65535. Die Übersetzung nach C liefert

```
GenLongIntMode((SHORTINT)0XFFFF, &LowWordMaskOp);
```

In C ist der Wert des aktuellen Parameters nach der Übergabe -1. In den erzeugten C-Programmen mußte daher die Typumwandlung entfernt werden, damit MOCKA auch in C korrekt funktioniert. Da der Wert 0FFFFH (=MAX(SHORTCARD)) eigentlich als SHORTINT-Wert überhaupt nicht darstellbar ist und außerdem der Typtransfer keine erkennbare Funktion hat, scheint die obige Konstruktion aber prinzipiell fragwürdig zu sein und sollte aus den Modula-Quellen von MOCKA entfernt werden.

Nach der Beseitigung der obengenannten Probleme lag auch der Modula-Übersetzer MOCKA sowohl auf den PCS- als auch auf den SUN-Workstations in einer C-Version vor und wurde auch bereits für die Übersetzung von zahlreichen Modula-Programmen wie z.B. *Mtc* und MOCKA eingesetzt.

Neben den beiden obengenannten großen Testfällen wurden noch folgende Modula-Programme mit *Mtc* nach C übersetzt: Der Minilax-Übersetzer aus dem Übersetzerbaupraktikum, die komplette Bibliothek *Reuse*, die Standardbibliothek des Übersetzers MOCKA sowie ein mit dem Zerteilergenerator PGS [Klein 86] erzeugter Zerteiler, der starken Gebrauch vom Zugriff auf lokale Variablen von statisch umfassenden Prozeduren macht und daher als Testfall ausgewählt wurde. Insgesamt wurden vom Verfasser dieser Diplomarbeit Modula-Programme mit einem Gesamtumfang von etwa 80000 Zeilen erfolgreich nach C übertragen.

Vom Betreuer dieser Diplomarbeit Dr. J. Grosch wurden außerdem während der Anfertigung dieser schriftlichen Ausarbeitung die Übersetzerbauwerkzeuge *Rex*, *Lalr* und *Ell* erfolgreich nach C übertragen. Diese Programme haben einen Umfang von etwa 35000 Zeilen Modula-Code.

Die erzeugten C-Programme wurden bisher an den folgenden Maschinen getestet: PCS- und SUN-Workstations (MC68020-Prozessor), DEC VAX 8530 und DEC 3100 (MIPS-Prozessor) und sind daher als gut portabel anzusehen.

6.2 Größe, Laufzeit und Speicherbedarf des Übersetzers

Sämtliche Messungen in diesem und den folgenden Abschnitten wurden an einer SUN-Workstation (MC68020-Prozessor, 20 MHZ Taktfrequenz) durchgeführt. Alle Angaben für den Übersetzer *Mtc* beziehen sich auf die Modula-Version des Übersetzers.

Die Größe des ausführbaren Objektprogramms von *Mtc*, gemessen mit dem *size*-Kommando [UNIX 79], kann Tabelle 6.1 entnommen werden.

Text	Data	Bss	Summe
294912	8192	6344	309448

Tabelle 6.1: Größe des Objektprogramms von *Mtc* in Byte

Tabelle 6.2 enthält Laufzeit und Leistung von *Mtc* für die Übersetzung des Modula-Übersetzers MOCKA, der aus 35 Moduln besteht, die insgesamt 37792 Zeilen Modula-Code umfassen. Die Laufzeit wurde gemessen mit dem *time*-Kommando [UNIX 79] und ist die Summe aus User- und System-Zeit.

Messung	Laufzeit [s]	Leistung [Zeilen/s]	Leistung [Grundsymbole/s]
A	303	125	496
B	255	148	589

Tabelle 6.2: Laufzeit und Leistung von *Mtc* für die Übersetzung von MOCKA

Bei der Messung A wurde jeder Definitions-, Implementierungs- und Programmodul einzeln nach C übersetzt. Bei der Messung B wurde *Mtc* nur für Implementierungs- und Programmoduln aufgerufen und bei jedem Übersetzungsvorgang mit Hilfe der in Kapitel 5.6 erwähnten Option die Definitionsdateien aller transitiv importierten Definitionsmoduln ausgegeben. Die Laufzeit (Messung A) verteilt sich folgendermaßen auf die einzelnen Übersetzerphasen:

Parse	19 %
GetDefinitionModules	29 %
Eval	31 %
DoCode	21 %

Dabei ist noch erwähnenswert, daß — neben dem Aufruf des Symbolentschlüsslers — die mit Abstand aufwendigste Einzeloperation, die Operation *Identify* für die Bezeichneridentifikation ist, die ca. 10–20 % der gesamten Übersetzungszeit beansprucht.

Im Vergleich zu *Mtc* ist die Laufzeit bzw. Leistung von MOCKA bei der Übersetzung von MOCKA 178 Sekunden bzw. 212 Zeilen pro Sekunde und die Laufzeit bzw. Leistung des C-Übersetzers für die Übersetzung der C-Version von MOCKA (36665 Zeilen C-Code) 408 Sekunden bzw. 90 Zeilen pro Sekunde.

Alle Leistungsangaben wurden berechnet aus: Gesamtumfang von MOCKA geteilt durch die Übersetzungszeit. Dabei wird nicht berücksichtigt, daß *Mtc* bei jedem Übersetzungsvorgang auch noch alle transitiv importierten Definitionsmoduln einliest bzw., daß der C-Übersetzer die Definitionsdateien dieser Definitionsmoduln in das C-Quellprogramm einfügt.

Berücksichtigt man, daß *Mtc* im Gegensatz zu MOCKA bei jedem Übersetzungsvorgang alle transitiv importierten Definitionsmoduln von neuem einliest und analysiert und daß die Bezeichneridentifikation wegen der Verwendung einer Attributgrammatik über Listen erfolgt, dann ist die Leistung von *Mtc* auch im Vergleich mit MOCKA sehr gut. Insbesondere scheint der Preis für eine Vereinfachung der Behandlung der getrennten Übersetzung durchaus vertretbar zu sein.

Tabelle 6.3 enthält den Speicherbedarf von *Mtc* (dynamisch angeforderter Speicher) für die Übersetzung der größten vorhandenen Übersetzungseinheit, dem Implementierungsmodul des Codegenerators von *Mtc*, der 7448 Zeilen Modula-Code umfaßt. Bei der Übersetzung dieses Moduls werden zusätzlich noch 18 Definitionsmoduln mit einem Gesamtumfang von 1992 Zeilen eingelesen.

Zweck	Dynamischer Speicherbedarf
Baum und Attribute	2243
Definitionstabelle	328
Vorbereitung der Codeerzeugung	4751
Gesamter Speicherbedarf	7484

Tabelle 6.3: Dynamischer Speicherbedarf von *Mtc*
für die Übersetzung des Codegenerators von *Mtc* in Kilobyte

Der Speicherbedarf ist mit 793 Kilobyte pro 1000 Zeilen Quellcode relativ hoch. Der größte Anteil entfällt allerdings auf den Codegenerator. Ohne die in Kapitel 5.6.2 besprochene automatische Nachoptimierung würde dieser 9155 Kilobyte Speicher benötigen. Bei der Angabe des Speicherbedarfs für Baum und Attribute ist zu berücksichtigen, daß der von *Ag* erzeugte Attributauswerter zur Zeit keine Optimierung der Attributspeicherung enthält und daher sämtliche Attribute im Baum gespeichert werden. Daß der Speicherbedarf sich trotzdem in vertretbaren Grenzen hält, liegt im wesentlichen daran, daß die meisten Attribute als abstrakte Datentypen in der Zielsprache realisiert werden und im Baum nur jeweils Zeiger auf die eigentlichen Attributwerte gespeichert werden. Eine entsprechende Optimierung der Attributspeicherung durch *Ag* und eine bessere Darstellung der in den Knoten des Strukturbaums bei der Vorbereitung der Codeerzeugung abgelegten Informationen durch *Estra* würde den Speicherbedarf des Übersetzers deutlich reduzieren.

6.3 Qualität des erzeugten C-Codes

Für eine Bewertung der Qualität des von *Mtc* erzeugten C-Codes wurden die Modula- und die C-Version von MOCKA miteinander verglichen. Um eine Verfälschung der Meßergebnisse zu vermeiden, wurde die Modula-Version von MOCKA ohne Laufzeitprüfungen übersetzt, d.h. auch die Modula-Version enthält — wie die C-Version — keine Überprüfung von Bereichs- und Feldgrenzen.

Tabelle 6.4 enthält die Größe der ausführbaren Objektprogramme der beiden MOCKA-Versionen.

MOCKA-Version	Text	Data	Bss	Summe
Modula-2	385024	8192	62680	455896
C	327680	32768	66288	426736

Tabelle 6.4: Größe der beiden MOCKA-Versionen in Byte

Tabelle 6.5 enthält Laufzeit und Leistung der beiden Versionen für die Übersetzung von MOCKA (37792 Zeilen Modula-Code).

MOCKA-Version	Laufzeit [s]	Leistung [Zeilen/s]	Leistung [Grundsymbole/s]
Modula-2	178	212	844
C	165	229	910

Tabelle 6.5: Laufzeit und Leistung der MOCKA-Versionen für die Übersetzung von MOCKA

Wie den beiden Tabellen entnommen werden kann, ist die C-Version von MOCKA sowohl kleiner als auch schneller wie die Modula-Version.

Eine leicht durchführbare Möglichkeit für eine Nachoptimierung der erzeugten C-Programme auf Quellsprachebene, die in Modula-2 nicht vorhanden ist, ist die Möglichkeit in C bestimmte Variablen in der Speicherklasse *register* (s. Kap. 3.2) zu deklarieren. Mit einigen wenigen solcher Deklarationen für häufig benutzte Variablen kann u.U. noch eine deutliche Leistungssteigerung der C-Programme erreicht werden.

6.4 Implementierung eines Makefile-Generators

Mit Hilfe der Werkzeuge *Rex* und *Ell* und des Interpreters für eine Sprache zur Textmusterverarbeitung *awk* [UNIX 79] wurde der Makefile-Generator *makemake* implementiert, der aus den Quellen eines Modulaprogramms durch Analyse der Importanweisungen eine Beschreibung der Abhängigkeiten zwischen den nach C übersetzten Quellen erzeugt, wie sie vom UNIX-Kommando *make* [UNIX 79] verarbeitet werden kann. Diese Beschreibung enthält außerdem Kommandos, die folgende Schritte veranlassen:

- Übersetzung der Modula-Quellen durch *Mtc* nach C.
- Übersetzung der erzeugten C-Quellen durch den C-Übersetzer in Objektprogramme.
- Binden der Objektprogramme zu einem ausführbaren Programm.

Durch einen Aufruf von *make*, welches die obige Beschreibungsdatei (Makefile) als Eingabe erhält, werden die Kommandos in der richtigen Reihenfolge ausgeführt und eine ausführbare C-Version des jeweiligen Modula-Programms erzeugt.

Das Programm für den Interpreter *awk* konnte aus einem ähnlichen Makefile-Generator, der für eine alte Version des Modula-Übersetzers MOCKA ohne automatische Nachübersetzung implementiert worden war, übernommen und entsprechend angepaßt werden.

Der Makefile-Generator war für die Testphase und die ersten praktischen Einsätze von *Mtc* eine wichtige Hilfe, da z.B. für die Erzeugung der C-Version von MOCKA ca. 150 Aufrufe von *Mtc* und des C-Übersetzers mit einer Vielzahl von Optionen und Parametern notwendig sind, die außerdem noch in der richtigen Reihenfolge erfolgen müssen. Durch die automatisch erzeugte Beschreibungsdatei reduziert sich dieser Aufwand auf einen Aufruf des Kommandos *make*.

Die Beschreibungdatei enthält übrigens auch genügend Abhängigkeitsinformationen für eine Verwaltung der C-Quellen: Bei einer Modifikation einer oder mehrerer C-Quellen werden durch einen Aufruf von *make* alle abhängigen C-Programme nachübersetzt. Die Beschreibung reicht allerdings nicht aus, um bei einer Modifikation der Modula-Quellen eine vollständige Nachübersetzung nach C zu veranlassen, da für eine Vereinfachung der Beschreibung in dieser nur vermerkt ist, daß *M.c* von *M.mi* und *M.h* von *M.md* abhängt. Es wäre allerdings — wenn dies benötigt werden sollte — nicht besonders aufwendig, *makemake* so zu erweitern, daß auch die Abhängigkeit einer Übersetzungseinheit von allen transitiv importierten (Definitions-)Moduln beschrieben wird.

6.5 Bewertung der Übersetzerbauwerkzeuge

Dieser Abschnitt enthält einige spezielle Anmerkungen zu den Übersetzerbauwerkzeugen sowie eine Reihe von Verbesserungswünschen und -vorschlägen.

Rex und *Ell* ermöglichen es, in kürzester Zeit einen Symbolentschlüssler bzw. einen Zerteiler mit automatischer Fehlerbehandlung aus einer knappen und leicht verständlichen formalen Spezifikation zu erstellen. Ein wichtiges Merkmal der erzeugten Übersetzerteile für den praktischen Einsatz ist ihre hohe Laufzeiteffizienz.

Auch der Einsatz von *Ast* ist eine enorme Arbeitserleichterung für den Übersetzerbauer: Aus einer sehr kurzen formalen Spezifikation der abstrakten Syntax (vgl. Anh. A) kann ein zwar relativ leicht „von Hand'' programmierbarer aber umfangreicher Modul für die Implementierung des Strukturbaums erzeugt werden.

Ein interessanter Aspekt in diesem Zusammenhang wäre, ob aus einer eventuell gemeinsamen Spezifikation des Zerteilers und der abstrakten Syntax nicht auch die semantischen Aktionen für den Baumaufbau automatisch erzeugt werden könnten, die bei der erstellten Zerteilerspezifikation den meisten Aufwand erforderten.

Insgesamt nahm die Implementierung der ersten beiden Phasen von *Mtc* (lexikalische und syntaktische Analyse, Baumaufbau und Behandlung der getrennten Übersetzung) weniger als 1/7 der gesamten Implementierungszeit in Anspruch.

6.5.1 Ag

Die zur Zeit vorliegende Version von *Ag* ist ein noch in der Entwicklung befindlicher Prototyp.

Die aus der Sicht des Verfassers wichtigste Verbesserung der Spezifikationssprache wäre, eine Möglichkeit innerhalb der Attributierungsregeln auf Attribute von Vorgängerknoten im Strukturbaum zugreifen zu können (analog INCLUDING des GAG-Systems [Kastens 82]). Durch eine solche Möglichkeit könnte die Attributgrammatik deutlich verkleinert werden und eine nicht unerhebliche Anzahl von Attributen und Attributberechnungen, die nur für Attributtransfers benötigt werden, könnte entfallen.

Während der Entwicklung der Attributgrammatik trat häufig das Problem auf, daß diese nicht mehr der Klasse OAG angehörte. In diesem Fall war es notwendig, geeignete künstliche Attributabhängigkeiten einzuführen, die wieder die Eigenschaft OAG erzwangen. Da auf der einen

Seite die von *Ag* ausgedruckten Informationen über die Attributabhängigkeiten sehr umfangreich sind und außerdem die Eigenschaft OAG nicht besonders leicht nachvollziehbar ist, war es jedesmal nicht ganz einfach, geeignete künstliche Attributabhängigkeiten zu finden. Es wäre daher wünschenswert, wenn *Ag* — ähnlich wie das GAG-System — in der Lage wäre, diese künstlichen Abhängigkeiten zur Erzielung der OAG-Eigenschaft ohne Einwirkung des Benutzers automatisch einzuführen.

Für eine formale Spezifikation der statischen Semantik einer Sprache im Rahmen der Sprachdefinition ist eine volle Funktionalität der Attributgrammatik sicher unerläßlich. Für eine praktische Implementierung der semantischen Analyse, die hinsichtlich Laufzeit und Speicherbedarf des Attributauswerters möglichst effizient sein soll, ist der kontrollierte Einsatz von Seiteneffekten manchmal nahezu unerläßlich. Außerdem lassen sich zyklische Abhängigkeiten innerhalb von Objekt- und Typbeschreibungen, wie sie in den meisten in der Praxis verwendeten Sprachen einfach auftreten, durch einen schrittweisen Aufbau dieser Beschreibungen noch am einfachsten auflösen. Ein Problem, welches durch den Einsatz von kontrollierten Seiteneffekten natürlich hinzukommt ist, daß man sich in diesem Zusammenhang Gedanken über die Reihenfolge der Seiteneffekte machen und diese Reihenfolge eventuell durch die Einführung von geeigneten Attributen bzw. Attributabhängigkeiten erzwingen muß.

Durch den Einsatz von in Modula-2 implementierten abstrakten Datentypen, durch kontrollierte Seiteneffekte und durch die von *Ag* verwendete direkte Implementierung der Besuchssequenzen mit rekursiven Prozeduren ist es gelungen, einen Attributauswerter aus einer Attributgrammatik zu erzeugen, der insbesondere im Hinblick auf Laufzeiteffizienz mit „von Hand'' implementierten Übersetzern wie MOCKA durchaus mithalten kann.

Die Entwicklung einer Attributgrammatik für die semantische Analyse ist eine relativ komplexe und aufwendige Aufgabe und der Teil der Arbeit, der dabei von einem Generator wie *Ag* übernommen werden kann, ist im Vergleich z.B. zur Arbeitserleichterung bei der automatischen Erzeugung eines Zerteilers aus einer kontextfreien Grammatik relativ klein. Die für die semantische Analyse von *Mtc* entwickelte Attributgrammatik ist sicher nur ein erster Ansatz im Hinblick auf Kombination von abstrakten Datentypen und Attributgrammatiken einschließlich dem Einsatz von kontrollierten Seiteneffekten und könnte in mancherlei Hinsicht verbessert werden. Die Entwicklung einer kompletten Attributgrammatik für die semantische Analyse von Modula-2, die sowohl Forderungen hinsichtlich Lesbarkeit, Verständlichkeit und Vollständigkeit als auch hinsichtlich Effizienz des daraus erzeugten Attributauswerters erfüllt, ist sicher ein interessantes Forschungs- und/oder Diplomarbeitsthema.

Eine letzte Bemerkung zu *Ag* gilt der Speicherung der Attributwerte. In der bisherigen Version von *Ag* werden sämtliche Attribute im Baum gespeichert. Wie die Messungen in Kapitel 6.2 zeigen ist dies — im Gegensatz zu anderslautenden Feststellungen in der Literatur — heute aufgrund der gewachsenen Hauptspeichergrößen durchaus möglich. Allerdings sollten dabei die folgenden Punkte berücksichtigt werden:

- Der Übersetzer *Mtc* enthält keine vollständige semantische Analyse. Eine solche vollständige semantische Analyse würde zu einer höheren Anzahl von Attributen und damit zu einem höheren Attributspeicherbedarf führen.
- Bei der Implementierung wurde wegen der fehlenden Speicheroptimierung stark auf den Speicherbedarf der Attributwerte geachtet und für die Attributtypen wann immer möglich Typen der Zielsprache mit möglichst geringem Speicherbedarf verwendet.

– Durch den Einsatz von in der Zielsprache implementierten abstrakten Datentypen, die mit Hilfe von Zeigertypen realisiert wurden, wird erreicht, daß die meisten im Baum gespeicherten Attribute nur Zeiger auf die eigentlichen Attributwerte (Objekt- und Typbeschreibungen, Umgebungsattribute, usf.) sind.

Somit wurde ein wesentlicher Teil der Optimierung des Attributspeicherbedarfs bereits ,,von Hand'' vorweggenommen. Es ist daher fraglich, ob es nicht doch sinnvoll wäre trotz der heutigen Hauptspeichergrößen, eine Optimierung der Attributspeicherung in *Ag* einzubauen. Eine solche eventuelle Optimierung wurde in der Attributgrammatik bereits vorbereitet, indem alle Attribute, die noch in der Codeerzeugung benötigt werden, als ,,output'' Attribute [Grosch 89a] gekennzeichnet wurden.

6.5.2 Estra

Die Erfahrungen des Verfassers bei diesem Einsatz von *Estra* sind sehr positiv. Der Übergang von einer direkten Implementierung zu einer Spezifikation des Codegenerators mit *Estra* hat eine Reihe von Vorteilen. Der Benutzer wird durch die Spezifikation von implementierungstechnischen Details wie der Traversierung des Baums entlastet. Die Auswahl bestimmter Transformationsvorschriften durch den Codegenerator kann auf einem höheren Abstraktionsniveau mit Mustern, Bedingungen und Kosten beschrieben werden. Außerdem werden die Zugriffe auf in den Mustern vorkommende Teilbäume und deren Attribute von *Estra* unterstützt und vereinfacht. Insgesamt ergibt sich eine bessere Lesbarkeit und Verständlichkeit der Codeerzeugung aufgrund des höheren Abstraktionsniveaus der Spezifikation. Außerdem wird eine leichtere Änderbarkeit und Wartbarkeit des Codegenerators erreicht.

Die Laufzeiteffizienz des erzeugten Codegenerators ist, wie die Messungen aus Kapitel 6.2 zeigen, ebenfalls gut. Allerdings fällt der Codegenerator mit insgesamt 7571 Zeilen recht groß aus. Hier wäre vielleicht eine Möglichkeit zur Modularisierung sowohl der Spezifikation als auch der daraus erzeugten Implementierung angebracht.

Hauptkritikpunkt an *Estra* ist der bereits in Kapitel 5.6.2 erwähnte und in den Messungen in Kapitel 6.2 noch einmal deutlich dokumentierte extrem hohe Speicherbedarf des von *Estra* erzeugten Codegenerators. Dieser Speicherbedarf resultiert daraus, daß *Estra* bei der Vorbereitung der Transformation für jeden Knoten des Strukturbaums einen Informationsblock anlegt, der für jede Funktion der Spezifikation die auf diesen Knoten anwendbare Vorschrift und die Kosten dieser Anwendung enthält (vgl. Kap. 5.6.2). Es muß aber bereits in der Spezifikation für jede Funktion angegeben werden, auf welche Knotentypen diese Funktion anwendbar ist. Betrachtet man die Spezifikation des Codegenerators von *Mtc*, die 22 Funktionen umfaßt, dann stellt man fest, daß im Mittel nur ca. 2 der 22 Funktionen auf jeden Knotentyp anwendbar sind. Das bedeutet aber, daß über 90 % des dynamisch angeforderten Speichers völlig umsonst angefordert werden, da dort nur die bereits aus der Spezifikation bekannte Tatsache festgehalten wird, daß die meisten Funktionen auf den jeweiligen Knoten des Strukturbaums überhaupt nicht anwendbar sind. Würde *Estra* für jeden Knoten nur einen Informationsblock anlegen, der Informationen über die auf diesen Knoten anwendbaren Funktionen enthält, könnte der Speicherbedarf des Codegenerators auf unter 1/10 reduziert werden.

7. Zusammenfassung und Ausblick

Zentraler Teil dieser Diplomarbeit bildet die Definition einer vollständigen Abbildung von Modula-2 nach C. Insbesondere wurde auch gezeigt wie statisch geschachtelte Prozedurdeklarationen und das Modulkonzept von Modula-2 nach C abgebildet werden können. Modula-2 besitzt zwar ein deutlich höheres Abstraktionsniveau, aber durch eine Kombination von primitiveren C-Konstrukten können auch in C nicht direkt vorhandene Modula-Konstrukte realisiert werden. Einzige Ausnahme hierbei bilden die Koroutinen für deren Abbildung kein geeignetes C-Konstrukt existiert.

Im Rahmen dieser Diplomarbeit wurde ebenfalls der Übersetzer *Mtc* entwickelt, der die oben beschriebene Abbildung implementiert und die Modula-Programme in lesbaren C-Code umsetzt. Etwa 3/4 des Quellcodes von *Mtc* besteht aus Moduln, die mit den an der GMD Forschungsstelle Karlsruhe entwickelten Übersetzerbauwerkzeugen *Rex*, *Ell*, *Ast*, *Ag* und *Estra* aus Spezifikationen erzeugt wurden. Da *Mtc* nicht für die Programmentwicklung, sondern für die Übertragung von fertig entwickelten Modula-Programmen nach C gedacht ist, wird die semantische Korrektheit der Eingabeprogramme nicht überprüft. Die Übersetzungsleistung von *Mtc* ist mit ca. 150 Zeilen in der Sekunde an einer SUN-Workstation (MC68020-Prozessor) sehr gut; allerdings ist der Speicherbedarf mit bis zu 800 Kilobyte je 1000 Zeilen Quellprogramm relativ hoch.

Da an der GMD Forschungsstelle Karlsruhe Modula-2 seit mehreren Jahren für die Programmentwicklung eingesetzt wird, bestand an geeigneten Testprogrammen für *Mtc* kein Mangel und anfänglich noch vorhandene Schwächen der Abbildung konnten schnell aufgedeckt und beseitigt werden. Bisher wurden u.a. folgende Programme mit *Mtc* erfolgreich nach C übertragen: der Übersetzer *Mtc* selbst, der Modula-Übersetzer MOCKA sowie die Übersetzerbauwerkzeuge *Rex*, *Ell* und *Lalr*. Wie erste Messungen zeigen ist die Qualität des erzeugten C-Codes ebenfalls gut: Die erzeugten C-Programme sind kleiner und schneller als die ursprünglichen Modula-Programme.

Erste Benutzerwünsche legen den Schluß nahe, daß die in Kapitel 4.2.4 begründete Entscheidung, Kommentare nicht von Modula-2 nach C zu übersetzen, doch noch einmal überdacht werden sollte. Durch eine Berücksichtigung von ,,typischen'' Konventionen, die für die Kommentierung in der Regel unbewußt eingehalten werden, wäre es vielleicht möglich, eine Lösung zu implementieren, die die Kommentare wenigstens in den meisten Fällen im C-Programm richtig plaziert und nur in wenigen Fällen eine manuelle Verschiebung durch den Benutzer erfordert. Allerdings sollte aus diesem Grund die Übersetzung der Kommentare optional sein und nur auf ausdrücklichen Wunsch des Benutzers stattfinden.

Die Implementierung von *Mtc* ließe sich sicher noch in mancher Hinsicht verbessern. Wichtigster Punkt wäre eine Reduktion des in Kapitel 6.2 dokumentierten hohen Speicherbedarfs von *Mtc*. Eine bessere Darstellung der in den Knoten des Strukturbaums bei der Vorbereitung der Codeerzeugung abgelegten Informationen durch *Estra* und eine Optimierung der Attributspeicherung durch *Ag* würde den Speicherbedarf des Übersetzers deutlich reduzieren.

Ein weiterer Punkt wäre eine interpretative Auswertung von konstanten Ausdrücken, die es auch ermöglicht, Überläufe zu erkennen und konstante Ausdrücke mit Operanden im Bereich MAX(INTEGER) + 1 .. MAX(CARDINAL) auswerten zu können.

Der hohe Aufwand für die Bezeichneridentifikation, der auf eine Implementierung der Umgebungsattribute mit Listen zurückzuführen ist (für große Programme ca. 20 % der gesamten Übersetzungszeit), könnte möglicherweise durch eine Implementierung der Umgebungsattribute

mit Hilfe von Suchbäumen reduziert werden.

Die Übersetzerbauwerkzeuge *Rex*, *Ell* und *Ast* sind weitgehend ausgereift und eignen sich wegen der enormen Arbeitserleichterung für den Übersetzerbauer, aber auch wegen der hohen Laufzeiteffizienz der erzeugten Übersetzerteile, für einen Einsatz auch in der Konstruktion von Übersetzern mit Produktionsqualität.

Die Prototypen *Ag* und *Estra* bieten erste interessante Ansätze für eine Spezifikation der semantischen Analyse und der (Zwischen-)Codeerzeugung, insbesondere sind die erzeugten Übersetzerteile sehr laufzeiteffizient. Allerdings besteht nach Ansicht des Verfassers dieser Diplomarbeit noch ein Entwicklungsbedarf sowohl hinsichtlich Ausdruckskraft der Spezifikationen als auch hinsichtlich des Speicherbedarfs der erzeugten Übersetzerteile bevor ein Einsatz dieser Werkzeuge auch außerhalb von Forschungsprojekten möglich ist.

Anhang A: Abstrakte Syntax von Modula-2

```
TREE

IMPORT  {
FROM StringMem  IMPORT tStringRef;
FROM Idents     IMPORT tIdent;
FROM Errors     IMPORT tPosition;
}

EXPORT  {
CONST
  Definition            =  1;           (* compilation unit kind        *)
  Foreign               =  2;
  Implementation        =  3;
  Program               =  4;

  NotEqual              =  1;           (* operators                    *)
  Times                 =  2;
  Plus                  =  3;
  Minus                 =  4;
  Divide                =  5;
  Less                  =  6;
  LessEqual             =  7;
  Equal                 =  8;
  Greater               =  9;
  GreaterEqual          = 10;
  And                   = 11;
  Div                   = 12;
  In                    = 13;
  Mod                   = 14;
  Not                   = 15;
  Or                    = 16;

  Decimal               =  1;           (* integer constant kind        *)
  Octal                 =  2;
  Hexadecimal           =  3;
}

GLOBAL  {
FROM StringMem  IMPORT tStringRef;
FROM Idents     IMPORT tIdent;
FROM Errors     IMPORT tPosition;
}

RULE

ROOT            = CompUnits .

CompUnits       = <
  CompUnits0    = .
  CompUnit      = [Kind: SHORTCARD] [Ident: tIdent] [Pos: tPosition]
                  Next: CompUnits REVERSE <
    DefMod      = Import Decls .
    ProgMod     = Import Decls Stmts .
  >.
>.

Import          = <
  Import0       = .
  Import1       = Next: Import REVERSE <
```

```
    From        = [Ident: tIdent] [Pos: tPosition] ImpIds .
    Objects     = ImpIds .
  >.
>.
ImpIds          = <
  ImpIds0       = .
  ImpIds1       = [Ident: tIdent] [Pos: tPosition] Next: ImpIds REVERSE .
>.
Export          = <
  Export0       = .
  Export1       = [Qualified: BOOLEAN] ExpIds .
>.
ExpIds          = <
  ExpIds0       = .
  ExpIds1       = [Ident: tIdent] Next: ExpIds REVERSE .
>.
Decls           = <
  Decls0        = .
  Decl          = Next: Decls REVERSE <
    Var         = VarIds Type .
    Object      = [Ident: tIdent] <
      Const     = Expr .
      TypeDecl  = Type [Pos: tPosition] .
      Proc      = Formals ResultType: PrimaryType Decls Stmts .
      ProcHead  = Formals ResultType: PrimaryType [Pos: tPosition] .
      Module    = Import Export Decls Stmts .
      Opaque    = .
    >.
  >.
>.
VarIds          = <
  VarIds0       = .
  VarIds1       = [Ident: tIdent] Next: VarIds REVERSE .
>.
Formals         = <
  Formals0      = .
  Formals1      = [IsVAR: BOOLEAN] ParIds Type Next: Formals REVERSE .
>.
ParIds          = <
  ParIds0       = .
  ParIds1       = [Ident: tIdent] Next: ParIds REVERSE .
>.
Type            = <
  Array         = [IsOpen: BOOLEAN] IndexType: SimpleType ElemType: Type .
  Record        = Fields .
  SetType       = BaseType: SimpleType .
  Pointer       = TargetType: Type .
  ProcType      = FormalTypes ResultType: PrimaryType .
  SimpleType    = <
    Enumeration = EnumIds .
    Subrange    = BaseType: PrimaryType Lwb: Expr Upb: Expr .
    PrimaryType = <
      Void      = .
      TypeId    = [Ident: tIdent] [Pos: tPosition] <
```

```
        TypeId0 = .
        TypeId1 = TypeId .
      >.
    >.
  >.
>.
Fields          = <
  Fields0       = .
  Fields1       = Next: Fields REVERSE <
    RecordSect  = FieldIds Type .
    VariantSect = TagField Variants Else: Fields.
  >.
>.
FieldIds        = <
  FieldIds0     = .
  FieldIds1     = [Ident: tIdent] Next: FieldIds REVERSE .
>.
TagField        = Type: TypeId <
  TagField0     = .
  TagField1     = [Ident: tIdent] .
>.
Variants        = <
  Variants0     = .
  Variant       = Labels Variant: Fields Next: Variants REVERSE .
>.
FormalTypes     = <
  FormalTypes0  = .
  FormalType    = [IsVAR: BOOLEAN] Type Next: FormalTypes REVERSE .
>.
EnumIds         = <
  EnumIds0      = .
  EnumIds1      = [Ident: tIdent] Next: EnumIds REVERSE .
>.
Expr            = <
  Binary        = [Operator: SHORTCARD] Lop: Expr Rop: Expr .
  Unary         = [Operator: SHORTCARD] Mop: Expr .
  IntConst      = [Kind: SHORTCARD] [IntVal: CARDINAL] [Pos: tPosition] .
  RealConst     = [RealVal: tStringRef] .
  StringConst   = [StringVal: tStringRef] .
  CharConst     = [CharVal: CHAR] .
  FuncCall      = Designator Actuals .
  Set           = BaseType: Qualid Elems .
  BitSet        = Elems .
  Designator    = [Pos: tPosition] <
    Qualid      = [Ident: tIdent] <
      Qualid0   = .
      Qualid1   = Qualid .
    >.
    Subscript   = Designator Index: Expr .
    Deref       = Designator .
    Select      = Designator [Field: tIdent] .
  >.
>.
Elems           = <
```

```
  Elems0        = .
  Elems1        = Next: Elems REVERSE <
    Elem        = Elem: Expr .
    ElemRange   = Lwb: Expr Upb: Expr .
  >.
>.

Actuals         = <
  Actuals0      = .
  Actual        = Expr Next: Actuals REVERSE .
>.

Stmts           = <
  Stmts0        = .
  Stmt          = Next: Stmts REVERSE <
    Assign      = Designator Expr .
    Call        = Designator Actuals .
    If          = Cond: Expr Then: Stmts Elsifs Else: Stmts .
    Case        = Expr Cases Else: Stmts [Default: BOOLEAN] .
    While       = Cond: Expr Stmts .
    Repeat      = Stmts Cond: Expr .
    Loop        = Stmts .
    For         = Qualid From: Expr To: Expr By: Expr Stmts .
    With        = Designator Stmts .
    Exit        = .
    Return1     = .
    Return2     = Result: Expr .
  >.
>.

Elsifs          = <
  Elsifs0       = .
  Elsifs1       = Cond: Expr Stmts Next: Elsifs REVERSE .
>.

Cases           = <
  Cases0        = .
  Cases1        = Labels Stmts Next: Cases REVERSE .
>.

Labels          = <
  Labels0       = .
  Labels1       = Next: Labels REVERSE <
    Label       = Label: Expr .
    LabelRange  = Lwb: Expr Upb: Expr .
  >.
>.
```

Anhang B: Beispiel für den erzeugten C-Code

```
(*====== Modula-Programm ================================================*)
(*------ Tree.md ---------------------------------------------------------*)
DEFINITION MODULE Tree;

CONST NoTree    = NIL;

TYPE
  tTree         = POINTER TO tNode;
  tNode         = RECORD
                    Key         : INTEGER;
                    Count       : CARDINAL;
                    Left, Right : tTree;
                  END;
  tProcOfNode   = PROCEDURE (INTEGER, CARDINAL);

VAR Root        : tTree;

PROCEDURE Insert        (x: INTEGER; VAR t: tTree);
PROCEDURE Delete        (x: INTEGER; VAR t: tTree);
PROCEDURE InOrder       (t: tTree; p: tProcOfNode);

END Tree.
(*------ Tree.mi ---------------------------------------------------------*)
IMPLEMENTATION MODULE Tree;

FROM Storage    IMPORT ALLOCATE, DEALLOCATE;

PROCEDURE Insert (x: INTEGER; VAR t: tTree);
BEGIN
  IF t = NoTree THEN
    NEW (t);
    WITH t^ DO
      Key := x; Count := 1;
      Left := NoTree; Right := NoTree;
    END;
  ELSIF x < t^.Key THEN Insert (x, t^.Left);
  ELSIF x > t^.Key THEN Insert (x, t^.Right);
  ELSE
    INC (t^.Count);
  END;
END Insert;

PROCEDURE Delete (x: INTEGER; VAR t: tTree);
  VAR s : tTree;

  PROCEDURE del (VAR t: tTree);
  BEGIN
    IF t^.Right # NoTree THEN
      del (t^.Right);
    ELSE
      s^.Key := t^.Key; s^.Count := t^.Count;
      s := t; t := t^.Left;
    END;
  END del;

BEGIN
  IF t # NoTree THEN
    IF x < t^.Key THEN Delete (x, t^.Left);
```

```
    ELSIF x > t^.Key THEN Delete (x, t^.Right);
    ELSE
      s := t;
      IF s^.Right = NoTree THEN t := s^.Left;
      ELSIF s^.Left = NoTree THEN t := s^.Right;
      ELSE del (s^.Left);
      END;
      DISPOSE (s);
    END;
  END;
END Delete;

PROCEDURE InOrder (t: tTree; p: tProcOfNode);
BEGIN
  IF t # NoTree THEN
    InOrder (t^.Left, p);
    p (t^.Key, t^.Count);
    InOrder (t^.Right, p);
  END;
END InOrder;

BEGIN
  Root := NoTree;
END Tree.

(*------ Main.mi ------------------------------------------------------*)

MODULE Main;

FROM StdIO      IMPORT ReadI, ReadNl, WriteI, WriteNl, CloseIO;
FROM Tree       IMPORT tTree, NoTree, Root, Insert, Delete, InOrder;

CONST cMax = 4;

VAR
  i : SHORTCARD;
  x : INTEGER;
  a : ARRAY [1..cMax] OF INTEGER;

MODULE TestOutput;

IMPORT WriteI; EXPORT WriteArray, WriteNode;

  VAR i : CARDINAL;

  PROCEDURE WriteArray (VAR a: ARRAY OF INTEGER);
  BEGIN
    FOR i := 0 TO HIGH (a) DO WriteI (a[i], 5); END;
  END WriteArray;

  PROCEDURE WriteNode (Key: INTEGER; Count: CARDINAL);
  BEGIN
    FOR i := Count TO 1 BY -1 DO WriteI (Key, 5); END;
  END WriteNode;

END TestOutput;

BEGIN
  FOR i := 1 TO cMax DO a[i] := ReadI (); Insert (a[i], Root); END; ReadNl;
  WriteArray (a); WriteNl;
  REPEAT
    InOrder (Root, WriteNode); WriteNl;
    x := ReadI (); ReadNl;
    Delete (x, Root);
  UNTIL Root = NoTree;
  CloseIO;
```

```
END Main.
(*====== Von Mtc erzeugtes C-Programm ==================================*)
/*------ Tree.h ------------------------------------------------------*/
#define DEFINITION_Tree
#define Tree_NoTree    NIL
typedef struct Tree_1 *Tree_tTree;
typedef struct Tree_1 {
    INTEGER Key;
    CARDINAL Count;
    Tree_tTree Left, Right;
} Tree_tNode;
typedef void (*Tree_tProcOfNode)();
extern Tree_tTree Tree_Root;
extern void Tree_Insert();
extern void Tree_Delete();
extern void Tree_InOrder();
extern void BEGIN_Tree();
/*------ Tree.c ------------------------------------------------------*/
#include "SYSTEM_.h"
#ifndef DEFINITION_Storage
#include "Storage.h"
#endif
#ifndef DEFINITION_Tree
#include "Tree.h"
#endif
Tree_tTree Tree_Root;
static void del();
static Tree_tTree *G_1_s;
void
Tree_Insert(x, t)
INTEGER x;
Tree_tTree *t;
{
  if (*t == Tree_NoTree) {
    Storage_ALLOCATE(t, sizeof(Tree_tNode));
    {
      register Tree_tNode *W_1 = *t;
      W_1->Key = x;
      W_1->Count = 1;
      W_1->Left = Tree_NoTree;
      W_1->Right = Tree_NoTree;
    }
  } else if (x < (*t)->Key) {
    Tree_Insert(x, &(*t)->Left);
  } else if (x > (*t)->Key) {
    Tree_Insert(x, &(*t)->Right);
  } else {
    INC((*t)->Count);
  }
}
static void
del(t)
```

```
Tree_tTree *t;
{
  if ((*t)->Right != Tree_NoTree) {
    del(&(*t)->Right);
  } else {
    (*G_1_s)->Key = (*t)->Key;
    (*G_1_s)->Count = (*t)->Count;
    *G_1_s = *t;
    *t = (*t)->Left;
  }
}

void
Tree_Delete(x, t)
INTEGER x;
Tree_tTree *t;
{
  Tree_tTree s;
  Tree_tTree *L_1;

  L_1 = G_1_s;
  G_1_s = &s;
  if (*t != Tree_NoTree) {
    if (x < (*t)->Key) {
      Tree_Delete(x, &(*t)->Left);
    } else if (x > (*t)->Key) {
      Tree_Delete(x, &(*t)->Right);
    } else {
      s = *t;
      if (s->Right == Tree_NoTree) {
        *t = s->Left;
      } else if (s->Left == Tree_NoTree) {
        *t = s->Right;
      } else {
        del(&s->Left);
      }
      Storage_DEALLOCATE(&s, sizeof(Tree_tNode));
    }
  }
  G_1_s = L_1;
}

void
Tree_InOrder(t, p)
Tree_tTree t;
Tree_tProcOfNode p;
{
  if (t != Tree_NoTree) {
    Tree_InOrder(t->Left, p);
    (*p)(t->Key, t->Count);
    Tree_InOrder(t->Right, p);
  }
}

void BEGIN_Tree()
{
  static BOOLEAN has_been_called = FALSE;

  if (!has_been_called) {
    has_been_called = TRUE;
```

```
    BEGIN_Storage();
    Tree_Root = Tree_NoTree;
  }
}
/*------ Main.c ------------------------------------------------------*/
#include "SYSTEM_.h"
#ifndef DEFINITION_StdIO
#include "StdIO.h"
#endif
#ifndef DEFINITION_Tree
#include "Tree.h"
#endif
#define cMax   4
static SHORTCARD i;
static INTEGER x;
static struct S_1 {
    INTEGER A[cMax - 1 + 1];
} a;
static CARDINAL C_1_i;
static void WriteArray();
static void WriteNode();
static void
WriteArray(a, O_1)
INTEGER a[];
LONGCARD O_1;
{
  {
    LONGCARD B_1 = 0, B_2 = (O_1 - 1);
    if (B_1 <= B_2)
      for (C_1_i = B_1;; C_1_i += 1) {
        StdIO_WriteI(a[C_1_i], 5);
        if (C_1_i >= B_2) break;
      }
  }
}
static void
WriteNode(Key, Count)
INTEGER Key;
CARDINAL Count;
{
  for (C_1_i = Count; C_1_i >= 1; C_1_i += -1) {
    StdIO_WriteI(Key, 5);
  }
}
static void TestOutput()
{}
void BEGIN_MODULE()
{
  BEGIN_StdIO();
  BEGIN_Tree();
  TestOutput();
  for (i = 1; i <= cMax; i += 1) {
```

```
    a.A[i - 1] = StdIO_ReadI();
    Tree_Insert(a.A[i - 1], &Tree_Root);
  }
  StdIO_ReadNl();
  WriteArray(a.A, 4L);
  StdIO_WriteNl();
  do {
    Tree_InOrder(Tree_Root, WriteNode);
    StdIO_WriteNl();
    x = StdIO_ReadI();
    StdIO_ReadNl();
    Tree_Delete(x, &Tree_Root);
  } while (!(Tree_Root == Tree_NoTree));
  StdIO_CloseIO();
}
```

Anhang C: UNIX-Manualseite für Mtc

NAME

mtc - Modula-2 to C Translator

SYNOPSIS

mtc [options] [file]

DESCRIPTION

Mtc translates Modula-2 programs into readable C code. *Mtc* implements the language Modula-2 as defined in N. Wirth's report (3rd edition) with a few minor restrictions (see below) and most language extensions implemented by *MOCKA*, the Modula-2 Compiler Karlsruhe. It produces K&R (not ANSI) C code with a few very common extensions like passing structures as value parameters.

A definition or foreign module *module*.md is translated into a C header file *module*.h. An implementation or program module *module*.mi is translated into the corresponding C source file *module*.c. Separate compilation is handled by reprocessing all transitively imported definition modules when translating a compilation unit. If *file* is omitted *mtc* reads from standard input.

Mtc is intended as a tool for translating finished programs from Modula-2 to C and not as a tool for program development. Therefore, the translator does not check the semantic correctness of the Modula-2 programs.

For each foreign module an empty implementation module corresponding to it has to be translated to C, because the initialization routine produced for the dummy implementation module is used within the modules resp. C programs which import the foreign module.

If the library function *alloca* is available and the C programs are compiled with the flag -D*StackAlloc*, then the memory space for open array value parameters will be allocated in the stack frame of the corresponding procedure. This temporary space will be freed automatically when the procedure returns. Otherwise, *malloc* and *free* will be used to allocate and deallocate memory space for open array value parameters.

OPTIONS

-w	Suppress warning diagnostics.
-i	Generate header files for imported modules.
-c	Generate type casts to make the C programs *lint* free.
-r	Generate runtime checks.
-h	Print help information.
-t	Print test output (time).
-m	Print test output (memory).
-d*dir*	Allow import from modules in library *dir*.
-l*dir*	Specify directory where *mtc* finds its tables.

FILES

module.md	Source file of definition or foreign module *module*.
module.mi	Source file of implementation or program module *module*.

module.h	C header file produced for *module*.md.
module.c	C source file produced for *module*.mi.
SYSTEM_.h	Definition of standard constants, types, functions, and macros, which are used in the generated C programs.
SYSTEM_.c	Main program and implementation of standard functions.

SEE ALSO

Entwurf und Implementierung eines Übersetzers von Modula-2 nach C by M. Martin.

Programming in Modula-2 (3rd edition) by N. Wirth.

MOCKA User Manual by F. Engelmann.

The C Programming Language by B. W. Kernighan and D. M. Ritchie.

DIAGNOSTICS

The translator reports lexical and syntactic errors, errors detected during the handling of separate compilation, and restrictions of the code generation. The translator does not check the semantic correctness of the Modula-2 programs. The diagnostics produced by *mtc* are intended to be self-explanatory.

BUGS

Coroutines are not supported.

Forward references within pointer declarations are limited to structured types.

Comments are not translated from Modula-2 to C.

In some very rare cases the translator has to evaluate constant expressions, because a literal translation is not possible. The translator will fail to do this, if the expression or one of its operands is not in the range MIN(INTEGER) .. MAX(INTEGER).

Anhang D: Verzeichnis der vom Übersetzer erzeugten Bezeichner

Die folgende Tabelle gibt einen Überblick über Bezeichner und Bedeutung der vom Übersetzer *Mtc* erzeugten C-Objekte. Die Abkürzung *nnn* steht dabei für eine eindeutige, vom Übersetzer vergebene Nummer.

Bezeichner	Bedeutung
A	Abbildung von Feldern auf Strukturen mit einem Vektor A als einziger Komponente (s. Kap. 4.4.2.5)
BEGIN_MODULE	Rumpf von Programmoduln (s. Kap. 4.8.7)
BEGIN_*Modulname*	Initialisierungsroutine (Rumpf) des (Implementierungs-)Moduls *Modulname* (s. Kap. 4.8.6)
B_*nnn*	Anfangs- bzw. Endwert von FOR-Schleifen (s. Kap. 4.6.5)
DEFINITION_*M*	Makroname für die Steuerung des Einfügens der Definitionsdatei des Moduls *M* mit #include-Anweisungen in alle Moduln, die *M* importieren (s. Kap. 4.8.1 u. 4.8.4)
EXIT_*nnn*	Sprungziel für EXIT-Anweisungen (s. Kap. 4.6.5)
G_*nnn*_*Name*	Abbildung von statisch geschachtelten Prozedurdeklarationen: Globaler Zeiger auf lokale Variable *Name* (s. Kap. 4.4.4)
L_*nnn*	Lokale Zeigervariable für die Kellerung der Werte von G_*nnn_Name* bei Rekursion (s. Kap. 4.4.4)
*Modulname*_*nnn*	Strukturnamen für vom globalen Modul *Modulname* exportierte Verbunde oder Felder (s. Kap. 4.4.2.6)
O_*nnn*	Abbildung offener Felder: Zusätzlicher Parameter mit aktueller Anzahl von Feldelementen (s. Kap. 4.4.4.1.2)
R_*nnn*	Wert von RETURN-Ausdrücken (s. Kap. 4.6.7)
S_*nnn*	Strukturnamen für nicht exportierte Verbunde oder Felder (s. Kap. 4.4.2.6)
U_*nnn*, V_*nnn*	Abbildung varianter Verbunde durch Einführung zusätzlicher Strukturkomponenten (s. Kap. 4.4.2.7)
W_*nnn*	Zeiger auf Verbunde für Abbildung von WITH-Anweisungen (s. Kap. 4.6.6)
X_*nnn*	Feldvariable für die Parameterübergabe von Zeichenketten (s. Kap. 4.6.2)
dummy	Strukturkomponente für in Modula-2 leere Verbunde

Als C-Bezeichner für die Modula-Objekte werden die Bezeichner aus dem Modula-Programm verwendet. Dabei werden 2 Fälle unterschieden:

- Der Bezeichner eines von einem globalen Modul exportierten Objekts wird in C in der qualifizierten Form *Modulname_Bezeichner* geschrieben.
- Alle übrigen Bezeichner werden direkt aus dem Modula-Programm übernommen, werden aber bei Bedarf zur Vermeidung von Namenskonflikten mit einem Präfix C_*nnn*_ versehen.

Alle anderen Bezeichner, die in den C-Programmen verwendet werden, sind Bezeichner für vom Übersetzer *Mtc* vordefinierte Objekte, wie z.B. die C-Definitionen der Standardtypen und -prozeduren der Sprache Modula-2. Alle diese Objekte sind in SYSTEM_.h bzw. SYSTEM_.c definiert und ihre Bedeutung ist aus ihrem Bezeichner ablesbar.

Literaturverzeichnis

[Aho 86]
A. V. Aho, R. Sethi, J. D. Ullman: Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading. Ma, 1986.

[Donzeau 79]
V. Donzeau-Gouge, G. Kahn, B. Krieg-Brückner, B. Lang: Formal Definition of Ada (preliminary draft). CII Honeywell Bull, Okt. 1979.

[Engelmann 87]
F. Engelmann: GMD MODULA SYSTEM MOCKA – User Manual. GMD Forschungsstelle an der Universität Karlsruhe, Dez. 1987.

[Grosch 87a]
J. Grosch: Reusable Software – A Collection of MODULA-Modules. Compiler Generation Report No. 4, GMD Forschungsstelle an der Universität Karlsruhe, Sept. 1987.

[Grosch 87b]
J. Grosch: Rex – A Scanner Generator. Compiler Generation Report No. 5, GMD Forschungsstelle an der Universität Karlsruhe, Dez. 1987.

[Grosch 89a]
J. Grosch: Ast – A Generator for Abstract Syntax Trees (Revised Version). Compiler Generation Report No. 15, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Grosch 89b]
J. Grosch: Ag – An Attribute Evaluator Generator. Compiler Generation Report No. 16, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Kastens 80]
U. Kastens: Ordered Attributed Grammars. Acta Informatica 13, 229–256, 1980.

[Kastens 82]
U. Kastens, B. Hutt, E. Zimmermann: GAG: A Practical Compiler Generator. Springer Verlag, Heidelberg, 1982.

[Kernighan 78]
B. W. Kernighan, D. M. Ritchie: The C Programming Language. Prentice-Hall, Englewood Cliffs, N.J., 1978.

[Kernighan 83]
B. W. Kernighan, D. M. Ritchie: Programmieren in C. Carl Hanser Verlag, München, Wien, 1983.

[Klein 86]
E. Klein, J. Grosch: User Manual for the PGS-System. GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1986.

[PascAda 80]
P. F. Albrecht, P. E. Garrison, S. L. Graham, R. H. Hyerle, P. Ip, B. Krieg-Brückner: Source-to-Source Translation: Ada to Pascal and Pascal to Ada. Symposium on the Ada Programming Language, ACM-SIGPLAN, 1980.

[PTC 87]
P. Bergsten: PTC implementation note. Holistic Technology AB, Grona Gatan 59, 41454 Gothenburg, Sweden, 1987.

[UNIX 79]
The UNIX Programmers Manual. Volume 1, 2a, 2b, Bell Laboratories, 1979.

[Vielsack 88]
B. Vielsack: The Parser Generators Lalr and Ell. Compiler Generation Report No. 8, GMD Forschungsstelle an der Universität Karlsruhe, April 1988.

[Vielsack 89]
B. Vielsack: Spezifikation und Implementierung der Transformation attributierter Bäume. Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik, Juni 1989.

[Waite 84]
W. M. Waite, G. Goos: Compiler Construction. Springer Verlag, New York, 1984.

[Wirth 85]
N. Wirth: Programming in Modula-2 (3rd edition). Springer Verlag, Berlin, Heidelberg, New York, Tokyo, 1985.

Inhaltsverzeichnis