P R O J E K T B E R I C H T

Entwurf und Implementierung eines Modula-2 - C Übersetzers

von
Klaus Preschern

Inhaltsübersicht

Abstract

In dieser Arbeit werden die wichtigsten Aspekte der automatischen Übersetzung von Modula-2 nach C diskutiert. Weiters wird der Entwurf und die Implementierung eines Modula-2-C Übersetzers beschrieben.

Die Entwicklung von Übersetzern, die von einer Hochsprache in eine andere Hochsprache übersetzen ist eine interessante Technologie, da dadurch die Portierbarkeit von Applikationen verbessert wird. Dies ist vor allem dann der Fall, wenn die Zielsprache eine weitere Verbreitung gefunden hat als die Quellsprache. Ein Übersetzer für die Übertragung von Modula-2 nach C ist daher ein praktisches Werkzeug, das in vielen Situationen sinnvoll angewendet werden kann.

Durch die Realisierung eines derartigen Übersetzers wird es möglich zwischen Modula-2 Applikationen und C-Bibliotheken bzw. C-Applikationen eine einfache und wirkungsvolle Verbindung zu schaffen. Es können dadurch Projekte teilweise in Modula-2 und teilweise in C realisiert werden. Weiters können Modula-2 Applikationen durch die Übersetzung nach C in Umgebungen übertragen werden, wo kein Modula-2 Compiler, jedoch ein C-Compiler vorhanden ist.

Bei der Entwicklung eines derartigen Übersetzers ist besonders darauf zu achten, daß der erzeugte Code so weit als möglich portabel ist. Ist ein solcher Übersetzer auch in der Lage sich selbst zu übersetzen, so ist dies nicht nur ein Qualitätsmerkmal, das die Erweiterung des Übersetzers erleichtert, sondern macht den Konverter auch selbst weitgehend portabel.

1 Einleitung

Die Produktion von Software erfordert sehr viel Zeit und damit Geld. Es ist heute kaum vorstellbar, daß man sämtliche Funktionen die eine Applikation verwendet auch selbst entwickelt. Statt dessen bedient man sich hier einseits der Fülle von Dienstleistungen die das Betriebssystem zur Verfügung stellt (Process Management, Input/Output, Memory Management, File System, etc.) und andererseits stützt man sich auf den Reichtum an Funktionen, die Programmbibliotheken offerieren.

Bietet nun das Werkzeug, das man zur Softwareentwicklung verwendet, nur eine eingeschränkte bzw. gar keine Verbindung zu den genannten Hilfsmitteln, so findet man sich plötzlich in die Lage von Robinson Crusoe [02] wieder, der auf seiner einsamen Insel gezwungen war, sich alles selbst anzufertigen.

Auch die Portabilität von Applikationen wird ein immer wichtigeres Kriterium für die Qualität der erzeugten Software. Betrachtet man die steigende Komplexität von Softwareprodukten, so ist es kaum vorstellbar, daß eine Applikation vollständig neu implementiert wird, wenn man sie auf ein anderes Computersystem überträgt. Die Verwendung von höheren Programmiersprachen für die Implementierung erleichtern die Portierung, da maschinenabhängige Dinge für den Programmierer weitgehend transparent gemacht werden. Im Idealfall, sollte es möglich sein, Applikationen ohne irgendwelche Änderungen von einer Umgebung in eine andere zu übertragen. Dies setzt allerdings voraus, daß auch in der neuen Umgebung ein Übersetzer für die verwendete Programmiersprache vorhanden ist.

Eine Möglichkeit, die angesprochenen Probleme in den Griff zu bekommen, ist die Übersetzung von einer höheren Programmiersprache in eine andere Hochsprache. Dies ist natürlich vor allem dann zwechmäßig, wenn die Zielsprache eine weitere Verbreitung gefunden hat als die Quellsprache.

Diese Arbeit beschreibt den Entwurf und die Implemetierung eines Übersetzers der Modula-2 nach C überträgt. Während die meisten Compiler absolut hardware-

spezifischen Objectcode produzieren, soll durch die Übersetzung von Modula-2 nach C diese Hardwareabhängigkeit durchbrochen werden und eine Verbindung zwischen der Modula-2 Welt und der C Welt auf möglichst hohem Niveau hergestellt werden.

Durch die Realisierung dieses Projektes soll es möglich werden:

a) Modula-2 Applikationen mit C-Bibliotheken bzw. C-Applikationen auf einfache Art und Weise zu verbinden. Dadurch ergibt sich auch die Möglichkeit Projekte für die Entwicklung von Software teilweise in Modula-2 und teilweise in C zu realisieren.

b) Modula-2 Applikationen durch die Übersetzung in C auch in Umgebungen zu übertragen, wo kein Modula-2 Compiler, jedoch ein C-Compiler vorhanden ist (dies trifft vor allem auf viele UNIX-Umgebungen zu).

Für Übersetzer, die von einer höheren Programmiersprache in eine Hochsprache übersetzen, haben sich verschiedene Begriffe entwickelt. So etwa Translator [16], Konverter [22] oder Transformator [05]. Definiert man einen Compiler als ein Programm, das von einer Quellsprache auf (irgendeine) Zielsprache übersetzt [12], so ist auch die Bezeichnung Compiler zulässig. Es ist jedoch kein Ziel dieser Arbeit hier eine Klarstellung oder gar Definition zu erreichen und so wird im folgenden, zum Zweck der Abgrenzung, der Begriff des Compilers für solche Programme benutzt, die als Output Assembler- oder Maschinencode produzieren. Für die in dieser Arbeit beschriebene Applikation wird meist der neutrale Begriff des "Übersetzers" oder eine der anderen angeführten Bezeichnungen verwendet.

Die Arbeit gliedert sich in zwei Hauptabschnitte. Im ersten wird ausführlich die Abbildung von Modula-2 nach C erörtert. Im zweiten Abschnitt wird die konkrete Implementierung eines Modula-2-C Übersetzers besprochen.

2 Die Abbildung von Modula-2 auf C

Modula-2 und C sind beides imperative Programmiersprachen, die beide eine abstrakte "Maschine" zur Verfügung stellen, um das Programmieren zu erleichtern. Die Entwicklung von Programmiersprachen ist zumeist ein evolutionärer Prozeß, wobei eine Sprache auf eine andere aufbaut oder zumindest die Sprachentstehung stark von anderen Programmierprachen beeinflußt wird. Will man nun eine Sprache in eine andere übersetzen, so spielt auch der "Grad der Verwandtschaft" eine Rolle. Daher wird im folgenden ein kurzer Überblick über die Beziehungen zwischen Modula-2 und C gegeben. Als Wurzel für die Entwicklung von Modula-2 und C kann die Algol Familie (Algol 60, Algol 68) angesehen werden [21] (siehe Abb. 1).

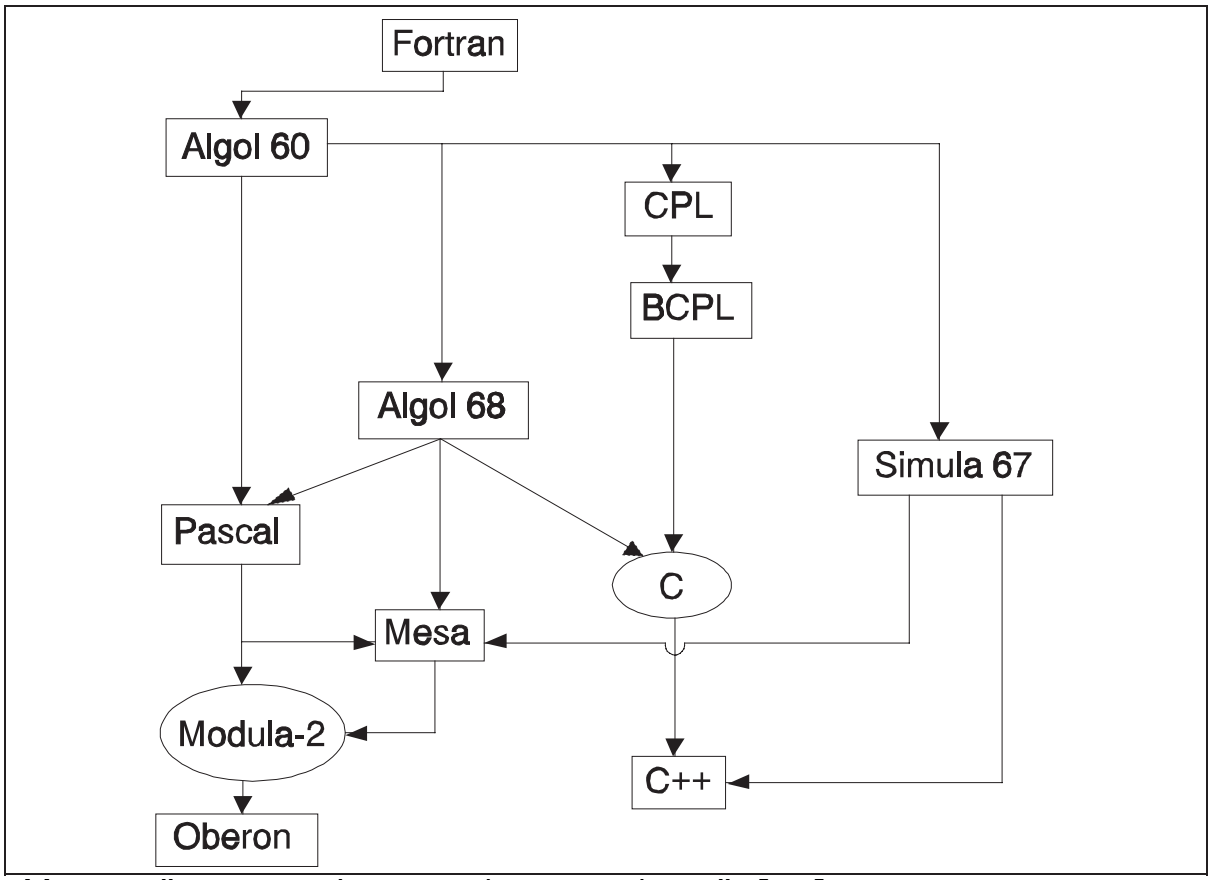

Abb. 1: "Programmiersprachenstammbaum" [21]

Ausgehend von der Algol Familie (1960) wurde dann von Niklaus Wirth die Sprache Pascal (1971) entwickelt und in weiterer Folge der Pascal-Nachfolger

Modula-2 (1983). Die Erweiterung gegenüber Pascal war hauptsächlich die Einführung des Modul-Konzepts und eine klarere und systematischere Syntax. Sowohl Pascal als auch Modula-2 verfügen über ein strenges Typkonzept.

Eine eher theoretische Schöpfung, da nie vollständig implementiert, war CPL (Combinded Programming Language) [21] (1966). Darauf aufbauend wurde BCPL (Basic CPL) (1969) als ein Werkzeug für die Compilerentwicklung geschaffen. Die Sprache C [03] wurde 1972 von Dennis Ritchie als Implementierungssprache für die Systemsoftware des UNIX Betriebssystems entworfen. Seit damals ist die weitere Entwicklung von C vor allem durch die Einführung eines strengeren Typkonzeptes geprägt, da dadurch die Portabilität von C-Software verbessert werden soll.

Modula-2 und C sind sich in ihren Strukturen und Auswertungsstrategien, nicht zuletzt durch die gemeinsamen entwicklungsgeschichtlichen Ursprünge, sehr ähnlich. Die Datentypen von Modula-2 und die darauf erlaubten Operationen können auf relativ einfache Art und Weise auf C abgebildet werden. Gleiches gilt für die Kontrollstrukturen, welche sehr direkt von Modula-2 nach C übersetzt werden können. Nur bei Moduln und Prozeduren (und Moduln innerhalb von Prozeduren) ergeben sich Schwierigkeiten, da Modula-2, hinsichtlich der statischen Verschachtelung von Moduln und Prozeduren, mehr Möglichkeiten bietet als C, das nur über eine flache Prozedurstruktur verfügt.

Die folgenden Abschnitte gehen von den Möglichkeiten die Modula-2 bietet aus und stellen diesen die entsprechenden Sprachelemente von C gegenüber. Wo keine direkte Abbildung möglich ist, werden Möglichkeiten für die Implementierung diskutiert. Da ein Modula-2-C Übersetzer implementiert wurde, wird in den einzelnen Abschnitten auch auf die konkrete Implementierung eingegangen. Dies erfolgt zumindest dort, wo die Abbildung von Sprachelementen nicht offensichtlich ist bzw. wo die vorliegende Implementierung Besonderheiten aufweist.

In fast jedem Abschnitt wird auch ein Beispiel für eine Übersetzung gezeigt, wobei diese mit dem Modula-2-C Übersetzer generiert wurden. Der Leser sei

jedoch darauf hingewiesen, daß die Beispiele für diese Arbeit von Hand nachbearbeitet wurden. Dies erfolgte mit der Absicht, in den Beispielen das für den jeweiligen Abschnitt wesentliche zu zeigen. Es handelt sich bei den Beispielen auch nicht um Programme, die irgendeine sinnvolle Funktion erfüllen. Es wurden folgende Änderungen des vom Übersetzer generierten Outputs vorgenommen:

- Objektnamen wurden auf die im Modula-2 Modul verwendete Form verkürzt. Der Übersetzer generiert etwas längere Namen, um Namenskonflikte zu vermeiden.

- Jener Code, der den Start des jeweiligen Hauptmoduls durchführt, wurde entfernt.

- Der Code, der für die Initialisierung von Moduln zuständig ist, bzw. mehrfache Initialisierungen verhindert, wurde entfernt.

- Es wurden Leerzeilen und Leerzeichen eingefügt bzw. gelöscht, um eine etwas bessere Strukturierung zu erreichen. Teilweise wurden Kommentare eingefügt.

Ein Beispiel für die vollständige Generierung von Namen wird im Abschnitt für Typ- und Variablendeklarationen gezeigt (siehe Abb. 11). Die vollständigen Start- und Initialisierungssequenzen von Moduln, werden im Abschnitt über die Übersetzung von Moduln gezeigt (siehe Abb. 16).

2.1 Deklarationen

Sowohl in Modula-2 als auch in C können Objekte deklariert werden. Dies können z.B. ganze Zahlen, gebrochene Zahlen, Zeichenketten, Datentypen und Variablen sein (Moduln und Prozeduren werden nicht in diesem, sondern in einem eigenen Abschnitt behandelt). Beiden Sprachen ist weiters gemeinsam, daß diese Objekte vor ihrer Verwendung deklariert werden müssen. Damit werden folgende Absichten verfolgt:

- Die Namen der Objekte werden mit einem Datentyp verbunden.

- Es können Konstante deklariert werden (Konstantendeklaration).

- Es ist möglich eigene Datentypen zu deklarieren (Typdeklaration).

- Da für die Datentypen jeweils entsprechende Operationen zugelassen sind, wird es möglich, die ordnungsgemäße Verwendung der Objekte sicherzustellen. Hier gibt es gewisse Unterschiede zwischen Modula-2 und C, da die letztere Sprache hier eine etwas "großzügigere" Handhabung dieser Überprüfungen zuläßt.

Da von Modula-2 nach C übersetzt wird und nicht umgekehrt, ergeben sich durch die strengeren Kriterien hinsichtlich erlaubter Operationen auf Datentypen in Modula-2 keine größeren Probleme. Im folgenden wird die Abbildung der Datentypen, der Konstanten- und Variablendeklarationen von Modula-2 auf C gezeigt.

2.1.1 Datentypen

Jeder Datentyp definiert eine Wertemenge und eine Menge von Operationen, die auf die Elemente der Wertemenge angewandt werden dürfen. Die Datentypen können weiters eingeteilt werden in vordeklarierte, einfache und strukturierte Datentypen.

2.1.1.1 Vordeklarierte Datentypen

Sowohl Modula-2 als auch C kennen eine Reihe von Datentypen, die in jedem Programm bekannt sind und nicht explizit deklariert werden müssen. Sie können daher auch als Standardtypen, Basistypen oder fundamentale Datentypen bezeichnet werden.

Stellt man die Modula-2 den C Standardtypen gegenüber, so erkennt man, daß es hier, neben der Namesgebung, noch weitere Unterschiede gibt. So bietet C einerseits eine größere Anzahl von Basistypen, aber andererseits gibt es in Modula-2 Datentypen denen kein unmittelbares Äquivalent in C gegenübersteht (siehe Abb. 2).

Wie bereits erwähnt, ist C auch hinsichtlich der Beschränkung der erlaubten Operationen (z.B. bezüglich Zuweisung untereinander) für die einzelnen Datentypen nicht so streng wie Modula-2.

Standardtypen in Modula-2	Standardtypen in C
CHAR	char unsigned char
BOOLEAN	?
INTEGER	short int int long int
CARDINAL	short unsigned int unsigned int long unsigned int
REAL	float double bzw. long float long double
BITSET	?

Abb. 2: Gegenüberstellung der Standardtypen von Modula-2 und C

Der Datentyp *CHAR* (*char*, *unsigned char*) wird für die Darstellung (in oft verschiedenen Codes - z.B. ASCII, EBCDIC) von Einzelzeichen verwendet. Jedes Zeichen wird durch eine im Code festgelegte Nummer dargestellt. In Modula-2 sind auf dem Typ *CHAR* nur Vergleichsoperationen, aber (im Gegensatz zu C) keine Rechenoperationen erlaubt.

Der Datentyp *BOOLEAN* beschreibt die Wahrheitswerte "wahr" und "falsch", für die in Modula-2 die vordeklarierten Konstanten *TRUE* und *FALSE* verwendet werden. Auf den ersten Blick hat *BOOLEAN* kein Äquivalent in C, jedoch wird hier der Wert 0 für "falsch" und jeder Wert ungleich 0 als "wahr" verwendet. Es kann also z.B. der Datentyp *short unsigned int* für Darstellung von *BOOLEAN* verwendet werden. Mit Objekten vom Typ *BOOLEAN* sind in Modula-2 neben Vergleichsoperationen (z.B. *FALSE* < *TRUE*) nur logische Operationen zulässig.

Der Datentyp *INTEGER* (*short int*, *int*, *long int*) umfaßt die positiven und negativen ganzen Zahlen, wobei der Wertebereich hardwareabhängig ist. In C werden die Schlüsselworte *short* bzw. *long* verwendet um unterschiedliche Wertebereiche zu spezifizieren. So kann etwa der Typ *int* den Bereich zwischen (-32768) und 32767 (16 bit Darstellung) umfassen, während *long int* im Bereich (-2147483648) und 2147483647 (32 bit Darstellung) liegt. Auf einem anderen Rechner könnte aber auch für beide Typen die 32 bit Darstellung und der gleiche Wertebereich verwendet werden. Auch für Modula-2 gibt es keine Spezifikation die den Wertebereich von *INTEGER* fix festlegen würde. In Ausdrücken sind neben Rechenoperationen für den Typ *INTEGER* auch Vergleichsoperationen zulässig.

Der Datentyp *CARDINAL* (*short unsigned int*, *unsigned int*, *long unsigned int*) wird für die Darstellung der positiven ganzen Zahlen (inklusive 0) verwendet. Hinsichtlich der Problematik der Wertebereiche gilt das gleiche wie für den Typ *INTEGER*. Weiters sind auch dieselben Rechen- und Vergleichsoperationen für den Typ *CARDINAL* erlaubt.

Der Datentyp *REAL* (*float*, *double*, *long double*) dient zur Darstellung der reellen Zahlen. Auch hier hängen der Wertebereich und weiters die Genauigkeit vom

jeweiligen Rechner ab. Weiters ist es vom verwendeten C-Compiler abhängig, ob der Typ *long double* unterstützt wird.

Auch für den Typ *REAL* sind die gleichen Rechen- und Vergleichsoperationen (nur für die Division wird in Modula-2, wie in C, der Operator "/" verwendet) zulässig, wie für die Typen *INTEGER* und *CARDINAL*.

Der Datentyp *BITSET* dient zur Darstellung von Mengen ganzer Zahlen, wobei diese Werte zwischen 0 und einer, wieder von der verwendeten Hardware abhängigen, Obergrenze (z.B. 15 für 16-Bit Rechner und 31 für 32-Bit-Rechner). Dieser Datentyp hat in C eigentlich keine Entsprechung. Zu seiner Realisierung können in einer entsprechenden Implementierung einerseits Bit-Felder oder andererseits der Typ (*long*) *unsigned int* mit den entsprechenden Bitmanipulationen, unterstützt vom Laufzeitsystem (run-time-system), verwendet werden.

Implementierung der Modula-2 Standardtypen: In jedem vom Modula-2-C Konverter übersetzten Modul wird im generierten C-Source-Modul der spezielle Header-File "M2rts.h" (siehe Listing von M2rts.h im Anhang G) importiert. In diesem Header-File sind die Modula-2 Standardtypen als symbolische Konstanten deklariert. Das heißt, daß der Modula-2-C Konverter bei der Übersetzung von Standardtypen, die in Modula-2 vorgesehenen Namen verwenden kann und diese dann vom C-Preprozessor durch die in M2rts.h definierten symbolischen Konstanten ersetzt werden.

Um die Verbindung von Modula-2 und C-Applikationen zu erleichtern, wurden die Modula-2 Basistypen um folgende Typen erweitert[1]: *SHORTINT*, *LONGINT*, *SHORTCARD*, *LONGCARD*, *SHORTREAL*, *LONGREAL*. Die gewählte Art der Implementierung hat den Vorteil, daß die konkrete Ausprägung der Typen leicht geändert werden kann. Da der Header-File von sämtlichen generierten C-Moduln importiert wird, wirkt sich eine derartige Änderung auf eine gesamte Applikation aus. Dies gilt auch für den Modula-2-C Konverter, da auch er denselben

[1] Einige dieser Typen werden auch von anderen Modula-2 Compilern unterstützt.

```
(*----------------------  Modula-2  ------------------------*)

MODULE StdTypes;
VAR
  i  : INTEGER;
  c  : CARDINAL;
  r  : REAL;
  b  : BOOLEAN;
  ch : CHAR;
  bi : BITSET;
BEGIN
  i  := -3;
  c  :=  3;
  r  := 3.0;
  b  := TRUE;
  ch := 'K';
  bi := {1, 2, 3};
END StdTypes.

/*----------------------     C      ------------------------*/

#include "M2rts.h"

/* MODULE */ void StdTypes();

static INTEGER i;
static CARDINAL c;
static REAL r;
static BOOLEAN b;
static CHAR ch;
static BITSET bi;

/* MODULE */ void StdTypes()
{
  i=(-3);
  c=3;
  r=(3.0E+000);
  b=TRUE;
  ch='K';
  bi=_MASK(1)|_MASK(2)|_MASK(3);
}
```

Abb. 3: Die Implementierung der Modula-2 Standardtypen

Mechanismus verwendet, sobald er sich selbst in C-Sourcecode übersetzt hat.

So kann z.B. bei einer Portierung der Typ *CARDINAL* einmal auf den C-Typ *unsigned int* (z.B. 16 Bit) und ein andermal auf den Typ *long unsigned int* (z.B. 32 Bit) abgebildet werden. Wird eine solche Änderung für eine Applikation durchgeführt, so setzt dies natürlich ein Wissen darüber voraus, welche Größe die

einzelnen Typen in dieser Anwendung mindestens haben müssen. Wenn die Applikation aber z.B. mit 16-Bit Typen entwickelt wurde, so sollten sich keine Probleme ergeben, wenn auf einem anderen Rechner stattdessen 32-Bit Typen verwendet werden. Allerdings ist Vorsicht geboten, falls eine Applikation in irgend einer Form (z.B. in Schleifen) auf die Grenzen der Standardtypen bezug nimmt (z.B. auch implizit über die Standardprozeduren *MIN()* oder *MAX()*).

Die durch diese Art der Implementierung erreichte Flexibilität kann vor allem der Übersetzer selbst nutzen. Wird er z.B. von einem 16-Bit-Rechner auf einen 32-Bit-Rechner portiert, so kann durch die entsprechende Änderung von "M2rts.h" eine einfache Anpassung an die neue Hardware vorgenommen werden. Es können in diesem Header-File auch Konstante definiert werden, die die Wertebereiche der einzelnen Typen angeben. Diese Konstanten können dann in den Range-Checks des Laufzeitsystems verwendet werden.

2.1.1.2 Einfache Datentypen

Sowohl in Modula-2 als auch in C können eigene Datentypen deklariert werden. Einfache Datentypen sind dabei in Modula-2 die Enumerations- und Subrange-Typen. In C gibt es zwar die Möglichkeit Enumerationstypen anzugeben, jedoch gibt es kein Äquivalent zu den Modula-2 Subrange-Typen. Diese müssen daher bei der Übersetzung auf ihren Basistyp abgebildet werden.

2.1.1.2.1 Enumerations-Typen

Mit Hilfe der Enumerations-Typen ist es auf einfache Weise möglich eine Gruppe von Konstanten zu deklarieren. Dabei werden den einzelnen Enumerationskonstanten in der Reihenfolge ihrer Deklaration konstante Werte zugewiesen, wobei die erste Konstante immer mit Null beginnt. Es sind auch Vergleiche zwischen den Konstanten erlaubt. Für die Abbildung dieses Typs auf C gibt es zwei Möglichkeiten: Man kann einerseits die von C unterstützten Aufzählungstypen (über das Schlüsselwort *enum*) verwenden, oder andererseits einen numerischen

Basistyp wählen, wobei der Modula-2-C Übersetzter den Konstanten ihre Werte zuweist.

```
(*--------------------  Modula-2  ----------------------*)

MODULE Enum;
TYPE
  Namen = (Andrea, Claudia, Elke, Wolfi, Michi, Frank,
           Klausi);
VAR
  Autor : Namen;
BEGIN
  Autor := Klausi;
END Enum.

/*--------------------     C      ----------------------*/

#include "M2rts.h"

/* MODULE */ void Enum();

static SHORTCARD  Autor;

/* MODULE */ void Enum()
{
  Autor=6;
}
```

Abb. 4: Beispiel für die Übersetzung von Enumerations-Typen

Implementierung von Enumerations-Typen: Der Modula-2-C Übersetzer verwendet letztere Methode und bildet die Enumerationstypen automatisch immer auf den Typ *SHORTCARD* (siehe "M2rts.h" im Anhang G) ab. Die Konstanten selbst werden dann direkt mit ihren Werten generiert (siehe Abb. 4).

2.1.1.2.2 Subrange-Typen

Die Subrange-Typen können in Modula-2 dafür verwendet werden, um den Wertebereich von Typen einzuschränken, deren Elemente aus Ordinalzahlen bestehen (*INTEGER*, *CARDINAL*, *BOOLEAN*, *CHAR*, Enumerationstypen). Von Modula-2 Compilern werden dann entsprechende Laufzeitprüfungen generiert,

um die Einhaltung der Wertebereiche sicherzustellen.

Implementierung von Subrange-Typen: In C gibt es zwar keine Art der Typdeklaration, die gleichwertige Eigenschaften wie die Modula-2 Subrange-Typen hätte, jedoch kann jeder Subrange-Typ auf seinen Basistyp abgebildet werden (siehe Abb. 5). Um die Einhaltung der Wertebereiche sicherzustellen, muß eine entsprechende Laufzeitunterstützung gegeben sein. In der derzeitigen Implementierung wird die Einhaltung der Wertebereiche allerdings nicht überprüft. Eine derartige Überprüfung müßte durch entsprechende Prozeduren im Laufzeitsystem durchgeführt werden und würde einen nicht unbeträchtlichen Aufwand zur Laufzeit erfordern. In Modula-2 Compilern kann hier normalerweise auf eine entsprechende Hardwareunterstützung zurückgegriffen werden.

```
(*----------------------- Modula-2 -------------------------*)

MODULE Subrange;
TYPE
  CharSubr = ["0".."9"];
  IntSubr  = [-1000..+1000];
VAR
  Card : [0..60000];
  Char : CharSubr;
  Int  : IntSubr;
BEGIN
  Card :=  3;
  Char := "3";
  Int  := -3;
END Subrange.

/*-----------------------    C     -------------------------*/

#include "M2rts.h"

/* MODULE */ void Subrange();

static CARDINAL  Card;
static CHAR  Char;
static INTEGER  Int;

/* MODULE */ void Subrange()
{
  Card=3;
  Char='3';
  Int=(-3);
}
```

Abb. 5: Beispiel für die Übersetzung von Subrange-Typen

2.1.1.3 Strukturierte Datentypen

Die strukturierten Datentypen setzen sich aus anderen Typen zusammen. In Modula-2 gibt es die Set-Typen, die Array-Typen, die Record-Typen und die Pointer-Typen. C bietet hinsichtlich der strukturierten Typen ähnliche Möglichkeiten wie Modula-2, wobei jedoch kein Set-Typ direkt unterstützt wird.

2.1.1.3.1 Set-Typen

In Modula-2 bestehen Mengen (Sets) aus Elementen desselben Datentyps, wobei dieser bei der Deklaration der Menge angegeben werden muß. Dieser Datentyp kann als Basistyp der Menge aufgefaßt werden und darf entweder ein Enumerations-Typ (*BOOLEAN* gilt als Enumerations-Typ) oder ein Subrange-Typ sein. Die Anzahl der Elemente eines Sets ist durch eine implementierungsabhängige Obergrenze beschränkt, wobei diese durch den Standardtyp *BITSET* festgelegt wird[2] [07]. Das heißt, daß jede Menge nur so viele Elemente haben darf, wie auch *BITSET* haben kann.

Implementierung von Set-Typen: Bei der Übersetzung von Modula-2 nach C werden alle Mengen auf den Basistyp *BITSET* abgebildet. Mit den Mengen sind somit auch alle Operationen erlaubt, die für diesen Typ gültig sind (siehe Abb. 6).

2.1.1.3.2 Array-Typen

Durch einen Array-Typ (Felder) können mehrere Objekte eines Typs zu einem Objekt zusammgengefaßt werden. In Modula-2 muß ein Feld aus einer festen Anzahl von Elementen bestehen (eine Ausnahme bilden ARRAY-Parameter, die im Abschnitt über die Prozeduren beschrieben werden), womit jedem Element ein

[2] Dies gilt jedoch nicht für alle Modula-2 Compiler. Einige erlauben Mengen mit bis zu 256 Elementen.

```
(*--------------------  Modula-2  ----------------------*)

MODULE Set;
TYPE
  Colors   = (blue, red, green);
  ColorSet = SET OF Colors;
VAR
  set : ColorSet;
BEGIN
  set := ColorSet{blue, green};
END Set.

/*--------------------     C      ----------------------*/

#include "M2rts.h"

/* MODULE */ void Set();

typedef BITSET ColorSet;
static ColorSet set;

/* MODULE */ void Set()
{
  set=_MASK(0)|_MASK(2);
}
```

Abb. 6: Beispiel für die Übersetzung von Mengen

Index zugeordnet werden kann. In C kann ein Array im Prinzip gleich aufgefaßt werden wie in Modula-2, wobei es jedoch eine enge Beziehung zwischen Zeigern und Feldern gibt. In C ist ein Zeiger eine Variable, die als Wert eine Adresse annehmen kann (wie in Modula-2). Ein Feld kann in C auch als konstanter Zeiger (der immer auf dieselbe Stelle zeigt) aufgefaßt werden.

In C beginnt ein Feld immer mit dem Index 0 und reicht bis zu einer angegebenen (positiven) *Grenze*. Die *Grenze* ist dabei gleich der Elementeanzahl des Feldes, womit die gültigen Indizes von 0 bis *Grenze* - 1 reichen. In Modula-2 dürfen beliebige (auch negative) Subranges [*Untergrenze*..*Obergrenze*] und Enumerationstypen als Indextypen angegeben werden, wobei die spezifizierten Grenzen ebenfalls als Indizes verwendet werden dürfen (es ist also beispielsweise auch die *Obergrenze* ein gültiger Index).

Als Elementtypen dürfen in beiden Sprachen beliebige Datentypen verwendet werden. Weiters gibt es in beiden Sprachen die Möglichkeit mehrdimensionale

Arrays zu deklarieren, wobei diese als Felder aufgefaßt werden, die wieder aus Feldern bestehen.

Um die Größe und die Indizes für Felder von Modula-2 in C wiederzugeben, muß es also eine Möglichkeit geben, um einen beliebigen (eventuell negativen bzw. teilweise negativen) Subrange [*Untergrenze*..*Obergrenze*] auf einen speziellen (positiven) Subrange von [0..*Grenze* - 1] abzubilden. Für Enumerationstypen als Indextyp ist die Aufgabe relativ einfach, da *Grenze* = (Anzahl der Elemente des Enumerationstyps) ist. Für positive Subranges gilt: *Grenze* = *Obergrenze* - *Untergrenze* + 1 und für negative Subranges (Ober- und Untergrenze sind negativ) bzw. teilweise negative (nur die Untergrenze ist negativ) Subranges gilt: *Grenze* = (-*Untergrenze*) + *Obergrenze* + 1. Dabei gilt implizit, daß immer *Untergrenze* < *Obergrenze* ist. Beispielsweise wird der Subrange von [-1000 .. +1000] auf [0..2000] abgebildet. So kann dann die Größe des Feldes für eine Deklaration in C bestimmt werden (im Beispiel ist *Grenze* = 2001).

Weiters muß dann beim Zugriff auf die einzelnen Elemente des Feldes wieder eine Korrektur in den speziellen Subrange vorgenommen werden. Der *Modula-2-Index* wird also in den entsprechenden *C-Index* umgewandelt. Für Enumerationstypen gilt, daß die Nummer des Elements gleich dem *C-Index* ist (Ein Enumerationstyp kann hier als Modula-2 Subrange mit *Untergrenze* = 0 und *Obergrenze* = Elementeanzahl - 1, aufgefaßt werden). Für Subranges muß wieder zwischen positiven und negativen Untergrenzen unterschieden werden. Für Subranges mit negativen Untergrenzen gilt: *C-Index* = *Modula-2-Index* + (-*Untergrenze*) und für positive (inklusive 0) Untergrenzen gilt: *C-Index* = *Modula-2-Index* - *Untergrenze*. Im Beispiel wird also (es wird die Deklaration eines Feldes mit dem Namen Array und dem oben angegebenene Beispiel-Subrange angenommen) Array[-1000] zu Array[0]. Dabei gilt implizit, daß immer *Untergrenze* <= *Modula-2-Index* <= *Obergrenze* ist. Für mehrdimensionale Felder gelten dieselben Regeln, aber eben für jeden Index einzeln.

Implementierung von Array-Typen: In der konkreten Implementierung wurden die Felder in Records "verpackt". Dies hat vor allem bezüglich der Implementierung von Referenzen durch Zeiger auf (eventuell mehrdimensionale)

```
(*--------------------  Modula-2  ----------------------*)

MODULE Array;
TYPE
  A1 = ARRAY [-1000..+1000] OF CARDINAL;
  A2 = ARRAY BOOLEAN, BOOLEAN OF BOOLEAN;
  A3 = ARRAY CHAR OF [0..60000];
VAR
  a1 : A1;
  a2 : A2;
  a3 : A3;
BEGIN
  a1[-100] := 3;
  a2[TRUE, FALSE] := TRUE;
  a3["a"]  := 9;
END Array.

(*--------------------     C      ----------------------*)

#include "M2rts.h"
/* MODULE */ void Array();

typedef struct A1 {
  CARDINAL  A[2001];
} _T_A1;
typedef struct A2 {
  struct PK1 {
    BOOLEAN  A[TRUE-FALSE+1];
  }  A[TRUE-FALSE+1];
} _T_A2;
typedef struct A3 {
  CARDINAL   A[256];
} _T_A3;
static struct A1 a1;
static struct A2 a2;
static struct A3 a3;

/* MODULE */ void Array()
{
  a1.A[_ARRAYCHK((-100),(-1000),1000)]=3;
  a2.A[_ARRAYCHK(TRUE,FALSE,TRUE)].
     A[_ARRAYCHK(FALSE,FALSE,TRUE)]=TRUE;
  a3.A[_ARRAYCHK('a','\0',255)]=9;
}
```

Abb. 7: Beispiel für die Übersetzung von Feldern

Felder Vorteile (siehe dazu den Abschnitt über Zeiger). Innerhalb eines solchen Records wird dann als einziges Recordelement das entsprechende Feld generiert (siehe Abb. 7).

2.1.1.3.3 Record-Typen

In einem Record (Struktur) können Elemente von verschiedenen Datentypen zusammengefaßt werden. Sowohl in Modula-2 als auch in C gibt es Strukturen, womit eine Abbildung recht einfach durchgeführt werden kann. Weiters gibt es in beiden Sprachen die Möglichkeit Varianten in Records anzugeben. In Modula-2 geschieht dies mit Hilfe der *CASE*-Anweisung, während in C Strukturen vom Typ *union* verwendet werden können. Dabei geht Modula-2 implizit und C explizit von der Annahme aus, daß sich die Varianten im Speicher überlappen.

Implementierung von Record-Typen: Ein Modula-2 Record wird einfach auf eine entsprechende C-Struktur abgebildet. Eine Modula-2 *CASE*-Anweisung wird auf eine C-*union* abgebildet, wobei jede Modula-2 Variante in einer eigenen C-Struktur "verpackt" wird. Für die *union* und die einzelnen Varianten-Strukturen sind dabei vom Übersetzer entsprechende Namen zu erzeugen, wobei diese natürlich auch bei den Zugriffen auf die einzelnen Elemente zu generieren sind (siehe Abb. 8).

2.1.1.3.4 Pointer-Typen

Sowohl in Modula-2 als auch in C lassen sich dynamische Datenstrukturen mit Hilfe von Zeigern realisieren. Dabei ist Modula-2 insofern strenger als C, da Zeiger an einen bestimmten Datentyp gebunden werden (und nicht als Zeiger auf Variablen anderer Typen verwendet werden dürfen) und keine arithmetischen Operationen mit Zeigern erlaubt sind (nur Prüfung auf Gleich- und Ungleichheit, eine Ausnahme bildet *ADDRESS*). Weiters gibt es in Modula-2 die spezielle Konstante *NIL*, die verwendet werden kann, um Zeiger als unbenutzt (Ende von Listen, usw.) zu kennzeichnen.

Sowohl in Modula-2 als auch in C darf ein Zeiger auf einen beliebigen Typ zeigen, womit die Abbildung der vordeklarierten und einfachen Datentypen unproblematisch ist. Etwas problematischer für die Abbildung von Zeigern ist die

```
(*--------------------  Modula-2  ----------------------*)

MODULE Record;
TYPE
  R = RECORD
        a, b : CARDINAL;
        CASE c : BOOLEAN OF
          TRUE  : d : CHAR;
        | FALSE : e : INTEGER;
        ELSE
          f : CARDINAL;
        END (* case *);
      END (* record *);
VAR
  r : R;
BEGIN
  r.a := 1;
  r.c := TRUE;
  r.d := 'K';
END Record.

/*--------------------     C      ----------------------*/

#include "M2rts.h"

/* MODULE */ void Record();
typedef struct R {
  CARDINAL a;
  CARDINAL b;
  BOOLEAN c;
  union {
    struct {
      CHAR d;
    } R1;
    struct {
      INTEGER e;
    } R2;
    struct {
      CARDINAL f;
    } R3;
  } U1;
} _T_R;
static struct R r;

/* MODULE */ void Record()
{
  r.a=1;
  r.c=TRUE;
  r.U1.R1.d='K';
}
```

Abb. 8: Beispiel für die Übersetzung von Records

Möglichkeit der Vorausdeklaration (foreward declaration) in Modula-2. Dabei darf

ein Zeiger auf eine Typ zeigen, der noch nicht explizit deklariert wurde bzw. erst später deklariert wird.

Bei C-Compilern hingegen müssen alle Objekte, auf die sie während der Übersetzung stoßen, bereits deklariert sein. Es ist also bei der Übersetzung von Modula-2 nach C dafür zu sorgen, daß solche Typen vor den auf sie weisenden Zeigern generiert werden. Dies scheint jedoch auf den ersten Blick nicht für Typen möglich zu sein, die es erlauben in sich einen Zeiger auf sich selbst (rekursive Strukturen) zu erzeugen (Man müßte den Typ vor sich selbst erzeugen!?). Solche rekursiven Strukturen sind in Modula-2 nur mit strukturierten Datentypen (Pointer-, Array- und Record-Typen) möglich.

Natürlich können auch in C rekursive Datenstrukturen erzeugt werden. Also zum Beispiel:

```
typedef struct Liste {
        unsigned            daten;
        struct Liste *      next;
} ListTyp;
```

Implementierung von Pointer-Typen: Bei dieser Deklaration wird also dem Typ zuerst, noch vor dessen expliziter Deklaration, ein Name zugewiesen, auf den dann in der Struktur ein Zeiger (und nur ein Zeiger) bezug nehmen kann. Die Abbildung von rekursiven Strukturen für Records ist also relativ einfach. Um den gleichen Mechanismus auch für Felder verwenden zu können, werden diese in Records "verpackt". Dadurch wird es möglich "straight foreward" Code zu generieren, ohne daß vorher die Strukturen, auf die ein Zeiger weist, aufwendig analysiert werden müßten (siehe Abb. 9). Als letztes zu lösenden Problem, bleiben Zeiger auf Zeigerreferenzen übrig.

Man kann hier weiters zwischen sinnvollen und meiner Ansicht nach eher absurden Fällen unterscheiden. Bei der sinnvollen Variante zeigt eine Folge von Zeigertypen (kann auch nur ein Zeiger sein) irgendwann einmal auf eine Struktur, die kein Pointer-Typ mehr ist. Die absurde Variante besteht nur aus

Pointer-Typen. Man kann sich überlegen, daß letzteres immer ein Zeigerzyklus sein muß. Also zum Beispiel:

```
TYPE
        Willi  = POINTER TO Otto;
        Otto   = POINTER TO Willi;
```

Da es auch in C Zeiger auf Zeiger gibt, ist die Abbildung von "sinnvollen Pointern" kein Problem. Will man allerdings die "absurden Pointer" ebenfalls abbilden, so ergibt sich sofort wieder das oben beschriebene Problem, daß ein Typ vor sich selbst generiert werden muß. Eine Lösung wäre etwa, daß auch Pointer (ebenso wie Felder) in Records "verpackt" werden könnten, jedoch rechtfertigt der vorliegende Sonderfall meiner Ansicht nach nicht den Implementierungsaufwand. Es ist leichter das Problem bei der Codegenerierung zu erkennen (Pointer auf Pointerzyklen untersuchen) und für solche "absurden Pointer" einfach Zeiger vom Typ *void* zu erzeugen.

2.1.2 Konstantendeklarationen

Sowohl in Modula-2 als auch in C ist die Deklaration von Konstanten möglich. Problematisch für die Übersetzung ist jedoch, daß Ausdrücke mit Konstanten in Modula-2 auch für die Deklaration von Subranges verwendet werden können. Diese Subranges können dann verwendet werden um z.B. die Größe von Feldern festzulegen. Um nun den Typ von einem solchen Subrange exakt bestimmen zu können muß der Konstantenausdruck vom Übersetzer ausgewertet werden. Der Typ wird dann verwendet, um sicherzustellen, daß der Subrange-Typ in Ausdrücken oder bei der Indizierung von Feldern korrekt verwendet wird. Weiters muß bei Subrange-Typen gewährleistet sein, daß der linke Ausdruck (Untergrenze) immer kleiner ist als der rechte Ausdruck (Obergrenze), was auch nur durch eine Auswertung der Konstanten sichergestellt werden kann. Die Auswertung von Konstanten erfordert, daß diese vom Übersetzer in ihre binäre Repräsentation umgewandelt werden.

```
(*--------------------  Modula-2  ----------------------*)

MODULE Pointer;
TYPE
  ListPtr = POINTER TO List;
  List    = RECORD
              data : CARDINAL;
              next : ListPtr;
            END (* record *);
  FieldPtr = POINTER TO Field;
  Field    = ARRAY BOOLEAN OF FieldPtr;
VAR
  LRoot : ListPtr;
  FRoot : FieldPtr;
  CPtr  : POINTER TO CARDINAL;
BEGIN
  LRoot^.next^.next := NIL;
  FRoot^[TRUE] := NIL;
  CPtr^:= 3;
END Pointer.

/*--------------------     C      ----------------------*/

#include "M2rts.h"

/* MODULE */ void Pointer();
typedef struct List {
  CARDINAL data;
  struct List   *  next;
}   *  ListPtr;

typedef struct Field {
  struct Field   *   A[TRUE-FALSE+1];
}   *  FieldPtr;

static struct List   *  LRoot;
static struct Field   *  FRoot;
static CARDINAL   *  CPtr;

/* MODULE */ void Pointer()
{
  LRoot->next->next=NIL;
  FRoot->A[_ARRAYCHK(TRUE,FALSE,TRUE)]=NIL;
  (*CPtr)=3;
}
```

Abb. 9: Beispiel für die Übersetzung von Pointern

Der Übersetzer erzeugt dann in der Codegenerierung keine symbolischen Konstantendeklarationen mehr. Werden Konstante in Deklarationen (z.B. für die Angabe von Feldgrößen) verwendet, so wird die Feldgröße berechnet und in

```
(*----------------------- Modula-2 -------------------------*)

MODULE Constants;
CONST
  MAXNAM = 30 + 3;
  STRLEN = MAXNAM + 10;
VAR
  String : ARRAY [0..STRLEN - 1] OF CHAR;
  namlen : CARDINAL;
BEGIN
  namlen := MAXNAM + 1 - 1;
END Constants.

/*-----------------------    C     -------------------------*/

#include "M2rts.h"

/* MODULE */ void Constants();

static struct PK1 {
  CHAR  A[43];
} String;
static CARDINAL namlen;

/* MODULE */ void Constants()
{
  namlen=33 + 1 - 1;
}
```

Abb. 10: Beispiel für die Behandlung von Konstanten

numerischer Form generiert (siehe Abb. 10). Symbolische Konstante, die in Anweisungen verwendet werden, werden in der Codegenerierung ebenfalls durch ihre numerischen Werte ersetzt (Konstante Ausdrücke in den Anweisungsteilen werden in der derzeitigen Implementierung allerdings nicht vom Modula-2-C Übersetzer ausgewertet).

2.1.3 Typ- und Variablendeklarationen

Sowohl in Modula-2, als auch in C, können vom Benutzer für Datentypen eigene Namen vergeben werden. Dies ist mit allen bisher angeführten Datentypen (einfache Datentypen, strukturierte Datentypen und Prozedurtypen - diese werden im Abschnitt über Prozeduren behandelt) möglich. Für die Übersetzung

von Modula-2 nach C hat dies eigentlich keine Auswirkungen. Wird einer der vordeklarierten Datentypen einem Typnamen zugewiesen, so ist die Generierung einer Typdeklaration nicht unbedingt notwendig und es kann der ursprüngliche Name verwendet werden.

Setzt sich ein Typ aus anderen Typen zusammen, so ist zu prüfen, ob sämtliche dieser Typen (zuvor) korrekt deklariert wurden. Dies gilt eingeschränkt für Pointer-Typen, da für diese die Möglichkeit besteht, Vorausdeklarationen anzugeben (siehe dazu den Abschnitt über Pointer-Typen).

Etwas problematischer sind Namensprobleme, die sich für globale Variablen, für Prozeduren und für Moduln ergeben können. So ist es in Modula-2 möglich, einen Modul zu importieren und Objekte aus diesem Modul durch die Voranstellung des Modulnamens zu verwendet. Dabei dürfen durchaus Objekte gleichen Namens in der lokalen Umgebung deklariert sein. Durch die Verwendung des Modulpräfix ist klargestellt, welches Objekt, in welcher Umgebung gemeint ist. Ähnliche Probleme treten auf, wenn verschachtelte Modula-2 Prozeduren oder innere Moduln auf die flache C-Prozedurstruktur abgebildet werden. Weiters können Objektnamen, die in Modula-2 absolut gültig sind, in C reserviert sein (z.B. eine Variable mit dem Namen *char*).

Um diese Namensprobleme in den Griff zu bekommen, gibt es mehrere Möglichkeiten. Die einfachste davon ist, das Problem überhaupt zu ignorieren und auf den Programmierer abzuwälzen. Dieser soll gefälligst eindeutige Namen verwenden und dabei berücksichtigen, daß sein Modula-2 Programm (eventuell) in C-Sourcecode umgewandelt wird.

Eine bessere Möglichkeit (die auch in der beschriebenen Form implementiert wurde), ist die Generierung von eindeutigen Namen. Man kann sich überlegen, daß der Name eines Objektes eindeutig durch die Umgebungshierarchie bestimmt ist, in der es deklariert ist. Eine Variable in einem(r) Modul (Prozedur) also durch ihren Namen selbst, den Namen des(r) umgebenden Moduls (Prozedur), den Namen der Umgebung des(r) Moduls (Prozedur), usw., bis zur äußersten Umgebungsebene (Hauptmodul oder Implementierungsmodul). Der sich ergebende

Namensstring bezeichnet eindeutig ein Modula-2 Objekt (siehe Abb. 11). Der Nachteil dieser Methode ist, daß die Namen der Objekte länger werden (manche C-Compiler schneiden dann die Namen, ab einer gewissen Länge, erst wieder ab) und dadurch die Lesbarkeit des erzeugen Outputs etwas leidet.

```
(*--------------------  Modula-2  ----------------------*)

MODULE Names;
VAR
  name : CARDINAL;
  MODULE Nested;
  VAR
    name : CARDINAL;
  BEGIN
    name := 3;
  END Nested;
BEGIN
  name := 1;
END Names.

/*--------------------     C      ----------------------*/

#include "M2rts.h"

/* MODULE */ void Names_M2();
static CARDINAL name_Names_M2;
/* MODULE */ static void Nested_Names_M2();
static CARDINAL name_Nested_Names_M2;

/* MODULE */ void Names_M2()
{
  Nested_Names_M2();
  name_Names_M2=1;
}
static /* MODULE */ void Nested_Names_M2()
{
  name_Nested_Names_M2=3;
}
```

Abb. 11: Beispiel für die Lösung des Namensproblems

Für Objekte die lokal in Prozeduren oder Records deklariert sind, ist die beschriebene Namensgenerierung nicht notwendig, da diese Objekte eben nur lokal gelten (für Prozeduren die verschachtelte Prozeduren besitzen, wird dies durch die gewählte Abbildungsmethode - siehe Abschnitt über Prozeduren - sichergestellt) bzw. auch in C ein entsprechender Record-Präfix vorangestellt werden muß. Um eine Abgrenzung gegenüber reservierten C-Schlüsselwörtern zu

erreichen kann ein entsprechender Namenssuffix generiert werden.

Eine weitere Möglichkeit, eindeutige Namen zu erhalten, ist die Abbildung des oben beschriebenen Namensstrings auf einen kurzen und eindeutigen (alpha-) numerischen Schlüssel. Neben der Problematik, einen entsprechenden Algorithmus (in [5] wird ein empirischer Ansatz beschrieben) zu finden, wird dadurch der erzeugte C-Output absolut unlesbar.

2.2 Die Übersetzung von Prozeduren

Die Abbildung der Modula-2 Datentypen und der darauf erlaubten Operationen auf C ist, wie wir gesehen haben, relativ direkt möglich. Gleiches gilt, wie noch zu zeigen ist, für die Kontrollstrukturen. Was Prozeduren betrifft, so offeriert Modula-2 reichhaltigere Möglichkeiten hinsichtlich der statischen Verschachtelung. Im Gegensatz dazu bietet C nur eine flache Prozedurstruktur.

2.2.1 Parameter

In Modula-2 gibt es die Unterscheidung zwischen Variablen-Parametern (call-by-reference) und Wert-Parametern (call-by-value). In C gibt es eigentlich nur Wert-Parameter, wobei für Referenzen explizit Zeiger (Adressen) übergeben werden. Die Abbildung der Modula-2 Parameterkonzepte stellt also kein allzu großes Problem dar. Auch die Nachbildung von (eventuell offenen) Array-Parametern ist möglich, wobei jedoch für Wert-Parameter dieser Art besondere Vorkehrungen getroffen werden müssen.

In C wird für ein Feld immer nur ein Zeiger übergeben und es ist daher notwendig Wert-Parameter dieser Art zu kopieren, um das "Orginal" vor Veränderungen zu schützen. Dafür ist es notwendig Platz zur Verfügung zu stellen, wobei jedoch bei offenen Arrays die Größe des Parameters im voraus nicht bekannt ist. Als Ausweg kann beim Eintritt in eine Prozedur Speicherplatz auf dem Heap allozieren werden, um dann das Feld dorthin zu kopieren. Die Größe eines offenen Feldes wird als zusätzlicher Paramter übergeben. Vor dem Verlassen der Prozedur wird dieser Speicher dann wieder freigegeben. Ein Beispiel für die Übersetzung von Prozeduren mit offenen Feldern befindet sich, aus Platzgründen, im Anhang B.

Weiters werden offene Arrays (wie alle Felder) von der Standardprozedur *HIGH()* unterstützt, wofür ebenfalls vorzusorgen ist. Es wird dafür der zusätzliche Parameter verwendet, der die Größe des übergebenen Feldes enthält.

2.2.2 Verschachtelte Prozeduren

Modula-2 Prozeduren können innerhalb anderer Prozeduren deklariert (verschachtelt) werden und haben dadurch Zugriff auf die Umgebungen der äußeren Prozeduren. Die Lebensdauer einer Prozedurumgebung ist gleich der Zeit, in der die Prozedur aktiv ist. Es gelten dabei folgende Sichtbarkeitsregeln (scope rules) [07]:

- Ein Name ist in der Prozedur sichtbar, in der er deklariert wird und weiters in allen von dieser Prozedur eingeschlossenen Prozeduren (Vorbehaltlich die folgende Regel).

- Wenn ein in Prozedur P deklarierter Name i in einer von P eingeschlossenen Prozedur Q redeklariert wird, dann sind die Prozedur Q und alle von Q eingeschlossenen Prozeduren aus dem Sichtbarkeitsbereich des in P deklarierten Namens i ausgeschlossen.

- Die Standardnamen von Modula-2 sind überall sichtbar.

Im Gegensatz zu Modula-2 dürfen in C innerhalb von Prozeduren keine weiteren Prozeduren mehr deklariert werden. Alle Objekte die (vor der Prozedur selbst) außerhalb und innerhalb einer C Prozedur deklariert wurden sind auch in der Prozedur sichtbar.

Um nun verschachtelte Modula-2 Prozeduren korrekt auf C abzubilden, gibt es mehrere Möglichkeiten, von denen einige im folgenden kurz diskutiert werden. Der erste Ansatz faßt die Umgebung einer Prozedur in einer Struktur zusammen und stellt deren Adresse, als Link-Parameter, für Zugriffe auf die Umgebung zur Verfügung (Structurization approach [15] oder im folgenden als Strukturisierung bezeichnet). Eine andere Möglichkeit ist die Übergabe von Rerferenzen auf Objekte der Umgebung als Parameter (Extended Parameter list approach [15] oder im folgenden als erweiterte Parameterlisten bezeichnet). Im letzten Teil, dieses Abschnitts über verschachtelte Prozeduren, wird dann der konkret implementierte Ansatz geschildert. Dieser hat zwar gewisse Ähnlichkeit mit der

Strukturisierung, jedoch wird nicht mit Link-Parametern gearbeitet, sondern mit globalen Zeigern (ähnlich einem "Display" [12]; diese Art der Implementierung wird daher im folgenden als displayorientierte Strukturisierung bezeichnet).

Es sei darauf hingewiesen, daß der gesamte Aufwand nur für Prozeduren betrieben wird, in denen noch weitere Prozeduren deklariert sind. Prozeduren für die dies nicht gilt, können direkt auf C abgebildet werden.

Ein ausführliches Beispiel für die Übersetzung von verschachtelten Modula-2 Prozeduren befindet sich, aus Platzgründen, im Anhang C. Dort wird die Übersetzung eines Modula-2 Moduls mit verschachtelten Prozeduren mit der Technik der Strukturisierung, erweiterten Parameterlisten und displayorientierter Strukturen gezeigt. In diesem Abschnitt wird ein kurzes Beispiel gezeigt, um die wesentlichsten Aspekte und Prinzipien der einzelnen Lösungen zu zeigen (siehe Abb. 12).

```
MODULE Nested;

  PROCEDURE Proc(param : INTEGER);
  VAR
    local : INTEGER;

    PROCEDURE Proc1(param1 : INTEGER);
    VAR
      local1 : INTEGER;
    BEGIN
      local1 := 1;
      local  := 1;
      param  := 1;
    END Proc1;
  BEGIN
  END Proc;

BEGIN
END Nested.
```

Abb. 12: Beispiel für die Übersetzung von verschachtelten Prozeduren

2.2.2.1 Strukturisierung

Bei der Strukturisierung wird die Umgebung einer Prozedur P, in der eine Prozedur Q deklariert ist, in einer Struktur (vergleichbar einem activation record) zusammengefaßt. Die Adresse dieser Struktur wird dann als Parameter an die Prozedur Q übergeben. Über diese Adresse (static link parameter [15]; vergleichbar mit dem access link in [12]) kann die Prozedur Q dann auf die Umgebung von P zugreifen.

In der in [15] vorgeschlagenen Implementierung werden jeweils die Parameterliste und die lokalen Variablen in getrennte Strukturen zusammengefaßt, wobei in beiden Strukturen ein extra Feld generiert wird, um eine Adresse aufzunehmen. Dieser Zeiger wird in der Struktur für die Parameter verwendet, um die Verbindungsinformation (static link) aufzunehmen, die von der rufenden Prozedur übergeben wird. Der Zeiger in der Struktur für die lokalen Variablen, dient für die Übergabe der Verbindung zur eigenen Umgebung an gerufene Prozeduren (siehe Abb. 13 und Anhang C). Dabei gilt folgendes [15]:

- Eine Prozedur, in der weitere Prozeduren verschachtelt sind, erfordert eine Strukturisierung ihrer Umgebung. Liegt ein solcher Fall vor, so gilt:

- Die äußerste Prozedur greift auf keine lokalen Objekte anderer Prozeduren zu (Moduln sind hier ausgenommen) und benötigt daher keinen static-link-pointer in ihrer Parameterstruktur. Allerdings ist eine Strukturisierung ihrer lokalen Umgebung (falls vorhanden) erforderlich, da diese Information an die inneren Prozeduren weitergegeben werden muß.

- Die innerste Prozedur benötigt einen static-link-pointer in ihrer Parameterstruktur, um auf nicht lokale Objekte zugreifen zu können. Da jedoch auf ihre lokale Umgebung keine äußeren Prozeduren zugreifen können, kann eine Strukturisierung der lokalen Objekte entfallen.

- Für Prozeduren die sowohl äußere als auch innere Prozeduren besitzen (die also "dazwischen" liegen) ist neben einem static-link-pointer in ihrer

Parameterstruktur, auch eine Strukturisierung der lokalen Umgebung notwendig.

- Eine Prozedur am äußersten Niveau, die keine verschachtelten Prozeduren besitzt (sie ist sowohl eine äußerste, als auch eine innerste Prozedur), erfordert weder einen static link Parameter, noch eine Strukturisierung der lokalen Umgebung.

Die Art und Weise, wie die Strukturisierung die Sichtbarkeitsregeln auf die flache C Prozedurstruktur abbildet, ist der Vorgangsweise von Modula-2 Compilern sehr ähnlich. Die Nachteile dieser Vorgangsweise sind:

- Die lokale Umgebung einer Prozedur muß in Strukturen verpackt werden.

- Die lokalen Referenzen der Prozedur müssen ebenfalls die erzeugte Struktur verwenden, um auf lokale Objekte zuzugreifen.

- Nicht lokale Referenzen erfordern, in Abhängigkeit von der Verschachtelungstiefe der zugreifenden Prozedur und der (nicht lokalen) Umgebung auf die zugegriffen wird, eine Folge von Zeigerdereferenzierungen. Dies erfordert einen gewissen Laufzeitaufwand und auch der erzeugte Code wird dadurch nicht unbedingt lesbarer.

- Die Art und Weise, wie die in [15] vorgeschlagene Implementierung Parameter übergibt, enthält ein implizites Wissen darüber, wie in C Strukturelemente abgelegt werden. Arrangiert ein C-Compiler Elemente von Strukturen nicht auf diese Art, so ist der von diesem Compiler erzeugte Code nicht lauffähig. Somit ist die vorgeschlagene Lösung nur bedingt portabel.

2.2.2.2 Erweiterte Parameterlisten

Eine Alternative zur Strukturisierung, ist die Übergabe von zusätzlichen

```
/*------------------------  Proc  --------------------------*/

typedef struct {
  int param;
} Ft_Proc;

typedef struct {
  int *pL;
  int local;
} Lt_Proc;

void Proc(fV)
  Ft_Proc fV;
{
  Lt_Proc lV;

  lV.pL = (int *) &fV;
}

/*------------------------  Proc1 --------------------------*/

typedef struct {
  int *sL;                /* static link */
  int param1;
} Ft_Proc1;

typedef struct {
  int *pL;                /* parameter list */
  int local1;
} Lt_Proc1;

void Proc1(fV)
  Ft_Proc1 fV;
{
  Lt_Proc1 lV;

  lV.pL = (int *) &fV;    /* save address of parameter list */
  lV.local1 = 1;
  ((Lt_Proc *)*(int *)&fV)->local = 1;
  ((Ft_Proc *)**(int **)&fV)->param = 1;
}

/*------------------------ Nested --------------------------*/

void Nested()
{
}
```

Abb. 13: Beispiel für die Strukturisierung

Referenzparametern (Var-Parametern) an Prozeduren, die Zugriffe auf die Umgebung anderer Prozeduren tätigen. Diese Parameter sind die Adressen der

nicht lokalen Objekte, auf die die inneren Prozeduren zugreifen. Dabei kann die Erweiterung der Parameterliste auf jene Objekte beschränkt werden, auf die auch tatsächlich zugegriffen wird (siehe Abb. 14 und Anhang C). Es gilt dabei folgendes [15]:

- Für Prozeduren auf der äußersten Ebene sind keine besonderen Vorkehrungen notwendig.

- Eine innerste Prozedur muß in ihrer Parameterliste sämtliche Referenzen auf die von ihr benötigten nicht lokale Objekte übergeben bekommen.

- Eine Prozedur, die keine äußerste Prozedur ist, aber auch keine innerste Prozedur, muß in ihrer Parameterliste sämtliche nicht lokalen Referenzen übergeben bekommen, die sie selbst und die in ihr verschachtelten Prozeduren benötigen. Diese Referenzen werden dann, erweitert um die eventuell benötigten Referenzen auf die lokale Umgebung der Prozedur selbst, an die tiefer verschachtelten Prozeduren weitergegeben.

Die Art und Weise, wie hier die Sichtbarkeitsregeln von Modula-2 auf C abgebildet werden, ist eher mit der Vorgangsweise von Programmierern vergleichbar, die eben lokale Objekte einfach als Parameter an andere Prozeduren übergeben, um dort einen Zugriff zu ermöglichen. Vergleicht man die Methode der erweiterten Parameterlisten mit der Strukturisierung, so erkennt man:

- Der Zugriff auf nicht lokale Objekte erfolgt direkt und ohne "Umwege" (nicht über viele Zeigerdereferenzierungen).

- Es müssen mehr Parameter übergeben werden (um dabei die wirklich benötigten herauszufinden, dürfte ein etwas größerer Implementierungsaufwand erforderlich sein).

- Die Lösung sollte absolut portabel sein.

```
/*------------------------  Proc  --------------------------*/

void Proc(param)
  int param;
{
  int  local;
}

/*------------------------ Proc1  --------------------------*/

void Proc1(param1, p_local, p_param)
  int param1;
  int * p_local;
  int * p_param;
{
  int local1;

  local1 = 1;
  *p_local = 1;
  *p_param = 1;
}

/*------------------------ Nested --------------------------*/

void Nested()
{
}
```

Abb. 14: Beispiel für erweiterte Parameterlisten

2.2.2.3 Displayorientierte Strukturisierung

Ein Display [12] ist ein Feld von Zeigern auf activation-records (Strukturen mit Parametern und lokalen Objekten) von Prozeduren. Über diese Zeiger wird dann auf nicht lokale Objekte anderer Prozeduren zugegriffen. Bei der klassischen Implementierung von Displays (bei von Compilern generiertem Code) ist die Größe des Feldes durch die maximale Verschachtelungstiefe der Prozeduren vorgegeben. Somit hat jede Verschachtelungsebene einen Eintrag zur Verfügung. Da die Größe des Displays begrenzt ist, aber Prozeduren auf der gleichen Ebene natürlich mehrmals aktiviert werden können, muß das Display entsprechend verwaltetet werden.

Um diese Verwaltung zu vereinfachen, wird in der vorliegenden Implementierung vom begrenzten Display abgegangen und einfach für jede Prozedur, die verschachtelte Prozeduren besitzt, ein Zeiger generiert. Diese Zeiger werden nicht

in einem Feld verwaltet, sondern sind einfach global und zeigen auf Strukturen, die sowohl die Parameter als auch sämtliche lokalen Variablen einer Prozedur umfassen (siehe Abb. 15 und Anhang C). Dabei gilt folgendes:

- Für Prozeduren, die keine inneren Prozeduren besitzen, sind keine besonderen Vorkehrungen zu treffen.

- Für Prozeduren, die in sie verschachtelte Prozeduren aufweisen, wird die Umgebung strukturisiert und ein globaler Zeiger mit dem Typ dieser Umgebungsstruktur generiert. Beim Aufruf der Prozedur wird der Zeiger entsprechend auf die lokale Umgebung gesetzt und die inneren Prozeduren können über diesen Zeiger auf die, aus ihrer Sicht, nicht lokale Umgebung zugreifen.

- Für Prozeduren auf innerster Ebene sind keine besonderen Vorkehrungen zu treffen. Sie greifen über die globalen Zeiger auf nicht lokale Objekte zu.

Der Speicherplatz, für die Aufnahme der lokalen Umgebung der Prozedur, wird in der Prozedur beim Aufruf generiert. Weiters befindet sich dort ein lokaler Zeiger, der dazu dient den aktuellen Wert des globalen Zeigers zu sichern. Nachdem der globale Zeiger gesichert ist, wird er auf den lokalen Record gesetzt, der die Umgebung der Prozedur enthält. Um die Portabilität der Lösung sicherzustellen, werden Parameter normal an die Prozedur übergeben und in den activation record kopiert. Für die lokalen Objekte ist natürlich keine Initialisierung notwendig.

Vor dem Verlassen der Prozedur wird der Wert des lokalen Zeigers wieder in den globalen Zeiger übertragen, womit dieser wieder auf den activation record der vorhergehenden Aktivierung der Prozedur zeigt.

Die Implementierung ist im Prinzip eine Form der Strukturisierung, versucht aber einige Nachteile zu vermeiden:

```
#include "M2rts.h"

/* MODULE */ void Nested();
static /* PROCEDURE */ void Proc();
typedef struct {
  INTEGER param;
  INTEGER local;
} _G_Proc;
static _G_Proc * _GP_Proc;

/* PROCEDURE */ void Proc1();

/*------------------------- Nested --------------------------*/

/* MODULE */ void Nested()
{
}

/*------------------------- Proc ----------------------------*/

static void Proc(param)
  INTEGER param;
{
  _G_Proc _L_Proc, * _LP_Proc = _GP_Proc;

  _GP_Proc = &_L_Proc;
  _GP_Proc->param = param;
  _GP_Proc = _LP_Proc;

}

/*------------------------- Proc1 ---------------------------*/

static void Proc1(param1)
  INTEGER param1;
{
  INTEGER local1;

  local1=1;
  _GP_Proc->local=1;
  _GP_Proc->param=1;
}
```

Abb. 15: Beispiel für displayorientierte Strukturen

- Die Umgebung von Prozeduren muß in Strukturen verpackt werden.

- Die lokalen Referenzen der Prozedur müssen ebenfalls die erzeugte Struktur verwenden, um auf lokale Objekte zuzugreifen.

- Im Gegensatz zur Strukturisierung ist maximal eine

Zeigerdereferenzierung notwendig, um auf nicht lokale Objekte zuzugreifen. Die Lesbarkeit des erzeugten Codes wird nur unwesentlich verschlechtert.

- Der größte Nachteil dieser Form der Implementierung ist, daß beim Eintritt in die Prozedur deren Parameter kopiert werden müssen. Als Vorteil bleibt dadurch allerdings die Portabilität des erzeugten Codes gewährleistet.

- Die Lösung ist ohne allzu großen Aufwand zu implementieren.

2.2.3 Standardprozeduren

In Modula-2 werden eine Reihe von Standardprozeduren unterstützt. Für die Implementierung empfiehlt sich, diese (ähnlich den Standardtypen) im Laufzeitsystem zusammenzufassen, oder direkt bei der Übersetzung entsprechenden Code zu erzeugen.

Die Standardprozeduren sind alle in einem C-Header-File (siehe Anhang G) als Makros deklariert. Dieser Header-File wird von allen vom Übersetzer erzeugten C-Source-Moduln importiert. Dabei stellen einige der deklarierten Makros Operationen zur Verfügung, einige rufen Prozeduren des Laufzeitsystems auf und wieder andere dienen einfach als Platzhalter, wobei für die Argumente vom Übersetzer ein geeigneter Code generiert wird.

2.2.4 Prozedurtypen und Prozedurvariablen

Sowohl in C, als auch in Modula-2 gibt es die Möglichkeit eine Prozedur einer Variablen zuzuweisen bzw. als Parameter zu übergeben. Somit stellt die Abbildung dieses Konzeptes kein großes Problem dar. Als Einschränkung gilt bei Modula-2, daß Prozedurvariablen bzw. Parametern, die einen Prozedurtyp verkörpern, keine Standardprozeduren zugewiesen werden dürfen. Weiters gilt,

daß keine Prozeduren, die lokal in anderen Prozeduren deklariert sind, solchen Variablen zugewiesen bzw. als Parameter übergeben werden dürfen.

Würde letztere Einschränkung nicht gelten, so müßte dafür vorgesorgt werden, daß eine als Parameter übergebene innere Prozedur auf ihre Umgebung (d.h. auch auf die Umgebungen der sie umschließenden Prozeduren) zugreifen kann. Bei der vorliegenden Implementierung, mit Hilfe eines "Displayeintrags" für jede Prozedur, sind alle Umgebungen der derzeit aktiven Prozeduren global bekannt, womit die Übergabe von Prozeduren, die lokal in anderen Prozeduren deklariert sind, kein Problem darstellen würde (man erhält die Lösung quasi "automatisch").

2.3 Die Übersetzung von Moduln

Das Modulkonzept ist wohl die herausragendste Eigenschaft von Modula-2. In C gibt es eigentlich kein wirklich vergleichbares Konzept, sondern ein C-Programm besteht im Prinzip aus einer Reihe von Prozeduren, von denen eine insofern ausgezeichnet ist, als dort das Programm zu laufen beginnt (*main()*). Während in Modula-2 der Austausch von Objekten zwischen verschiedenen Moduln von der Sprache genau geregelt ist und vom Compiler überprüft wird, findet in C keine derartige Überprüfung statt. Die einzelnen Prozeduren können in C wohl auf verschiedene Dateien verteilt werden, jedoch werden Prüfungen, ob auch alles deklariert ist, auf den Linker weitergeschoben. Eine Prüfung, ob etwa Prozedurparameter entsprechend der Prozedurdeklaration verwendet werden, findet in C nicht statt. Allerdings versuchen neue C-Standards (ANSI-C) dieses Manko von C zu beheben.

In Modula-2 kann ein Programm auf mehrere Moduln aufgeteilt werden, wobei jeder Modul eigene Konstanten, Typen, Variablen, Prozeduren und weitere Moduln enthalten kann. Durch entsprechende Import- bzw. Export-Listen können dann Objekte von verschiedenen Moduln verwendet bzw. zur Verfügung gestellt werden. Man kann in Modula-2 weiters zwischen "inneren" Moduln (lokale Moduln) und "äußeren" Moduln (Definitionsmoduln, Implementierungsmoduln, Hauptmodul) unterscheiden.

Da es in C nur Prozeduren gibt, bleibt eigentlich keine Wahl, als Moduln auf Prozeduren abzubilden. Dabei müssen jedoch Gültigkeitsbereiche und Lebensdauer von Objekten, "innere" und "äußere" Moduln, sowie die Initialisierung von Moduln berücksichtigt werden.

2.3.1 Lebensdauer und Gültigkeitsbereiche

In Modula-2 können Moduln durch *IMPORT*- und *EXPORT*- Anweisungen Objekte untereinander austauschen. Von zentraler Bedeutung für die Abbildung auf C ist dabei die Lebensdauer und der Gültigkeitsbereich solcher Objekte.

Im Gegensatz zu Prozeduren, deren lokale Objekte nur solange existieren, wie die Prozedur aktiv ist, existieren Objekte in einem Modul genauso lange wie die Umgebung des Moduls. Dabei gelten folgende Regeln [08]:

- Die lokalen Objekte eines Moduls (einschließlich der exportieren) werden "angelegt", bevor die Ausführung des Blocks der Umgebung beginnt.

- Nach dem Erzeugen der Variablen wird der Rumpf des Moduls ausgeführt (Initialisierung).

- Die Lebensdauer der Objekte eines Moduls endet, sobald die letzte Anweisung des Umgebungsblocks ausgeführt ist.

Da Moduln ineinander verschachtelt werden können und auf eine flache C-Prozedurstruktur abgebildet werden müssen, ergeben sich ähnliche Probleme wie bei Prozeduren. Die Verschachtelung muß aufgebrochen werden, wobei jedoch die Regeln bezüglich der Lebensdauer von Objekten völlig andere sind.

Bei einer reinen Modulhierarchie (keine Moduln in Prozeduren), leben sämtliche Objekte in den Moduln solange wie der äußerste Modul. Die Objekte in diesen Moduln werden also in C global deklariert. Dabei ist dafür zu sorgen, daß keine Namensprobleme auftreten (siehe Abschnitt über Typ- und Variablendeklarationen und Abb. 11).

Die lokalen Objekte eines Moduls (und die Objekte von darin verschachtelten Moduln) innerhalb einer Prozedur leben solange, wie die lokalen Objekte der Prozedur und werden daher in der Umgebung der Prozedur deklariert. Der Zugriff auf diese Objekte erfolgt genauso, wie verschachtelte Prozeduren auf Objekte ihrer Umgebung zugreifen (siehe Abschnitt über verschachtelte Prozeduren und Anhang I).

2.3.2 Definitions- und Implementierungsmoduln

Äußere Moduln können in Modula-2 getrennt übersetzt werden. Die Kommunikation dieser Moduln erfolgt über Definitionsmoduln. Auch in C ist es möglich, Schnittstellenbeschreibungen (z.B. *extern* Deklarationen) in sogenannten Header-Files (*.h* Dateien) abzulegen. Diese Beschreibungen werden dann vom C-Präprozessor in jene Datei inkludiert, die vom C-Compiler übersetzt werden soll.

Man kann sich für die Abbildung von Definitionsmoduln überlegen, ob man dafür Header-Dateien generiert. Der Vorteil ist, daß statt der Deklaration der importierten Objekte in der C-Source-Datei nur mehr ein *#include* Statement generiert werden muß. Der Nachteil ist, daß sich die Anzahl der Dateien verdoppelt. Eine andere Möglichkeit ist, daß die Arbeit des C-Präprozessors vom Modula-2-C Übersetzer übernommen wird und dieser in den erzeugten C-Sourcefiles die entsprechenden (externen) Deklarationen generiert. Der Vorteil ist, daß keine Header-Dateien generiert werden müssen und der Nachteil ist, daß der erzeugte Output etwas größer wird. In jenem Implementierungsmodul, zu dem ein Defintionsmodul gehört, müssen alle Objekte die dort exportiert werden gobal deklariert werden. Im Gegensatz dazu werden rein lokale Objekte in C als *static* deklariert.

Bei Implementierungs-, Haupt- und lokalen Moduln ist neben dem Aufbrechen von Verschachtelungen (Moduln und Prozeduren) für den korrekten Durchlauf der Initialisierungsteile zu sorgen. Für jeden importieren Modul muß dessen Initialisierungsteil (einmal) aufgerufen werden. Bei Moduln in Prozeduren muß dies bei jeder Prozeduraktivierung geschehen.

2.3.3 Hauptmoduln

Ein Programm beginnt in C immer mit der Prozedur *main()* (diese wird zumeist vom C-Laufzeitsystem aufgerufen, wo das Programm wirklich zu laufen beginnt). Dabei ist es in C möglich dem Programm Argumente mitzugeben, die als Parameter der Prozedur *main()* aufgefaßt werden. In Modula-2 beginnt ein

Programm immer mit dem Block des Hauptmoduls. Eine offensichtliche Möglichkeit der Abbildung ist nun, den Hauptmodulblock einfach auf die Prozedur *main()* abzubilden. Oft ist es jedoch notwendig vor dem Start des Hauptmoduln noch Initialisierungen (z.B. Kommandozeilenparameter sichern, eventuell durchzuführende Betriebssystemaufrufe, Heapinitialisierungen, usw.) im Laufzeitsystem vorzunehmen.

Es empfiehlt sich somit, die Prozedur *main()* im Laufzeitsystem zu plazieren, womit jedes übersetzte Modula-2 Programm den gleichen, genau definierten, Startpunkt hat. Von dort kann dann eine spezielle Startprozedur (die vom Übersetzer bei der Übersetzung des Hauptmoduls generiert wird) aufgerufen werden, die wiederum die Prozedur, die den Hautpmodul verkörpert, aktiviert. Dort beginnt dann die Initialisierung der vom Hauptmodul importierten und im Hauptmodul deklarierten Moduln, wobei Mehrfachinitialisierungen verhindert werden müssen (siehe Abb. 16). Letzteres gilt auch für jeden Implementierungsmodul.

```
(*--------------------  Modula-2  ----------------------*)

MODULE Hello;
  FROM InOut IMPORT WriteString, WriteLn;
BEGIN
  WriteLn;
  WriteString("Hello world !");
  WriteLn;
END Hello.

/*--------------------     C      ----------------------*/

#include "M2rts.h"

BOOLEAN  _KEY_Hello_M2 = FALSE;
extern /* PROCEDURE */ void WriteString_InOut_M2();
extern /* PROCEDURE */ void WriteLn_InOut_M2();

/* MODULE */ void Hello_M2();
void _START_MODULA()            /* here we start */
{
  Hello_M2();
}

/* MODULE */ void Hello_M2()
{
  if (_KEY_Hello_M2) return;    /* already initialized ? */
  _KEY_Hello_M2 = TRUE;         /* now initialized */
  InOut_M2();                   /* initialize InOut */
  WriteLn_InOut_M2();
  WriteString_InOut_M2("Hello world !\0", 13*sizeof(CHAR ));
  WriteLn_InOut_M2();
}
```

Abb. 16: (Ungekürztes) Beispiel für die Übersetzung und den Start eines Hauptmoduls

2.4 Die Abbildung von Ausdrücken

In Modula-2 gibt es verschiedene Klassen von Ausdrücken. Fast alle dieser Klassen sind auch in C vorhanden, sodaß eine Übersetzung kein großes Problem darstellt. Diese Klassen sind im einzelnen [08]:

- Arithmetische Ausdrücke (Datentypen *CARDINAL*, *INTEGER*, *REAL*) für Berechnungen im engeren Sinn.

- Logische Ausdrücke (Datentyp *BOOLEAN*) für die Beschreibung von

Vergleichen und logischen Verknüpfungen.

- Mengen-Ausdrücke (Datentypen *BITSET* und *SET OF* ...) für Mengenoperationen. Bei dieser Klasse von Ausdrücken ist, für die Übersetzung von Modula-2 nach C, teilweise eine spezielle Unterstützung durch das Laufzeitsystem vorzusehen.

- Allgemeine Ausdrücke (*RECORD*s, *ARRAY*s, *POINTER*) sind solche, die nicht oder nur eingeschränkt (für Zeiger sind nur Vergleiche erlaubt) durch Operationen miteinander verknüpft werden können.

Ein Ausdruck kann sowohl in Modula-2 als auch in C Konstanten, Variablen oder Aufrufe von Funktionsprozeduren als Operanden enthalten. Für jede Klasse von Ausdrücken gibt es auch die entsprechenden arithmetischen und logischen Operatoren, sowie Mengen- und Vergleichsoperatoren.

2.4.1 Arithmetische Operatoren

Alle arithmetischen Operatoren (siehe Abb. 17) sind auf Operanden des Typs *INTEGER* und *CARDINAL* (außer "/"), sowie auf Unterbereiche dieser Typen anwendbar. Auf Operanden vom Typ *REAL* sind sämtliche Operationen, außer *DIV* und *MOD* anwendbar. Die Vorzeichenumkehr ist nur für die Typen *INTEGER* und *REAL* erlaubt.

2.4.2 Logische Operatoren

Die logischen Operatoren (siehe Abb. 18) sind in Modula-2 nur auf Operanden des Typs *BOOLEAN* anwendbar. Die Abbildung der logischen Operatoren auf C stellt kein Problem dar, da auch hier die entsprechenden Operationen vorhanden sind. Allerdings gibt es bei den Operanden insofern Unterschiede, als in C jeder Wert ungleich "0" als wahr gilt, während in Modula-2 dafür die Konstante *TRUE* vorgesehen ist. Der vordeklarierte Datentyp *BOOLEAN* kann als

Rechenoperationen	Modula-2	C
Addition	a + b	a + b
Subtraktion und Vorzeichenumkehr	a - b - a	a - b - a
Multiplikation	a * b	a * b
Ganzzahlige Division	a DIV b	a / b
Reelle Division	a / b	a / b
Divisionsrestbildung	a MOD b	a % b

Abb. 17: Gegenüberstellung der Rechenoperationen von Modula-2 und C

Enumerationstyp mit den Elementen (*FALSE*, *TRUE*) aufgefaßt werden. Sowohl in C, als auch in Modula-2 steht somit der Wert "0" für den Wahrheitswert "falsch", allerdings ist in Modula-2 hierfür die vordefinierte Konstante *FALSE* vorgesehen.

2.4.3 Vergleichsoperatoren

Vergleichsoperationen (Relationen) sind auf auf alle Standardtypen, sowie auf Aufzählungs- und Unterbereichstypen anwendbar (siehe Abb. 19). Vergleichsoperationen liefern immer ein Resultat vom Typ *BOOLEAN*. Die Vergleichsoperatoren =, # und <> sind auf Mengen, Zeiger und abstrakte Datentypen anwendbar. Die Relationen <= und >= können auch auf Mengen angewendet werden, um festzustellen, ob eine Menge eine Teilmenge einer anderen Menge ist.

Logische Operationen	Modula-2	C
Und-Verknüpfung	a AND b a & b	a && b
Oder-Verknüpfung	a OR b	a || b
Negation	NOT a ~ a	! a

Abb. 18: Gegenüberstellung der logischen Operationen von Modula-2 und C

Mit dem Relationenoperator *IN* kann überprüft werden, ob sich ein angegebenes Element in einer Menge befindet oder nicht.

2.4.4 Mengenoperatoren

Mengen werden in Modula-2 meist auf ein Speicherwort abgebildet, wobei die einzelnen Elemente durch die einzelnen Bits verkörpert werden. In C gibt es eigene Operatoren für die bitweise Verknüpfung von Operanden, die für die Abbildung der Mengenoperationen (siehe Abb. 20) verwendet werden können (siehe auch Abb. 21).

Relationen	Modula-2	C
Gleich	a = b	a == b
Ungleich	a <> b oder a # b	a != b
Kleiner	a < b	a < b
Kleiner gleich und Teilmenge	a <= b a <= b	a <= b (a & b) == a
Größer	a > b	a > b
Größer gleich und Teilmenge	a >= b a >= b	a >= b (a & b) == b
Element von	a IN b	Laufzeitsystem

Abb. 19: Gegenüberstellung der Vergleichsoperationen von Modula-2 und C

2.5 Die Abbildung der Anweisungen

Modula-2 kennt elf verschiedene Anweisungsarten, die allesamt ohne größere Probleme auf C abgebildet werden können. Einige verlangen allerdings eine spezielle Behandlung, wie z.B. die Zuweisung von Feldern, die Initialisierung von Mengen oder die in C unbekannte *WITH*-Anweisung. Weiters ist bei der Abbildung auf die Portabilität des erzeugten C-Sourcecodes zu achten (dies betrifft hauptsächlich die *FOR*-Anweisung). Folgen von Anweisungen werden sowohl in Modula-2 als auch in C durch Strichpunkte voneinander getrennt.

2.5.1 Wertzuweisungen und Prozeduraufrufe

Die meisten Modula-2 Zuweisungen können direkt auf C abgebildet werden. Eine Sonderbehandlung erfordern dabei Felder. In Modula-2 ist es möglich einen

Mengenoperationen	Modula-2	C
Vereinigung	a + b	a ¦ b
Differenz	a - b	a & ~b
Durchschnitt	a * b	a & b
symmetrische Differenz	a / b	(a¦b) & ~(a&b)

Abb. 20: Abbildung der Modula-2 Mengenoperationen auf C

String der Länge n_1 an eine Stringvariable der Länge n_2 zuzuweisen, wenn $n_2 >= n_1$ gilt. In C ist hierfür eine explizite Kopieroperation zu erzeugen. Einen weiteren Sonderfall für die Abbildung der Zuweisungen bildet die Initialisierung von Mengen.

Bei Prozeduraufrufen bzw. bei Funktionsprozeduren ist auf die Unterscheidung zwischen Variablen- und Wert-Parametern zu achten. Für Variable-Parameter sind in C entsprechende Adressoperatoren zu generieren, wobei Felder wieder eine Ausnahme bilden.

Die mit den Prozeduraufrufen verknüpfte *RETURN*-Anweisung kann ebenfalls direkt auf C abgebildet werden. Hier sind allerdings, in Abhängigkeit von der Implementierung von Prozeduren und deren Parametern, eventuell spezielle Vorkehrungen zu treffen. So kann es notwendig sein, vor dem Verlassen einer Prozedur entsprechende Aktionen vorzunehmen. In der vorliegenden Implementierung muß etwa der Speicherplatz für Felder, die als Wert-Parameter übergeben

```
(*--------------------  Modula-2  ----------------------*)

MODULE SetOps;
VAR
  u, v, w : BITSET;
  b       : BOOLEAN;
BEGIN
  u := {1, 2, 5, 6};
  v := {1, 3, 5, 7};
  w := u + v;
  w := u - v;
  w := u * v;
  w := u / v;
  b := u <= v;
  b := u >= v;
  b := 3 IN v;
END SetOps.

/*--------------------     C      ----------------------*/

#include "M2rts.h"

/* MODULE */ void SetOps_M2();
static BITSET u;
static BITSET v;
static BITSET w;
static BOOLEAN b;

/* MODULE */ void SetOps_M2()
{
  u=_MASK(1)|_MASK(2)|_MASK(5)|_MASK(6);
  v=_MASK(1)|_MASK(3)|_MASK(5)|_MASK(7);
  w=u | v;
  w=u & ~ v;
  w=u & v;
  w=(u | v) & ~(u  & v);
  b=(u & v) == u;
  b=(u & v) == v;
  b=(_MASK(3) & v) == _MASK(3);
}
```

Abb. 21: Beispiel für die Übersetzung der Mengenoperationen

wurden, auf dem Heap freigegeben werden. Ist nun z.B. ein Element eines solchen Feldes der Rückgabewert einer Funktionsprozedur, so muß dieser Wert, vor der Deallozierung des Speicherplatzes, in eine temporäre Variable übertragen werden. Diese Variable wird dann von der *return*-Anweisung zurückgegeben.

2.5.2 Die WITH-Anweisung

Eine *WITH*-Anweisung spezifiziert in Modula-2 eine Rekordvariable und eine Anweisungsfolge. Innerhalb der Anweisungsfolge kann dann die Qualifizierung der Rekordfelder weggelassen werden. Der *WITH*-Selektor wird nur einmal (vor der Anweisungsfolge) ausgewertet. Bei der Abbildung der *WITH*-Anweisung nach C kann auf Zeiger zurückgegriffen werden, da der *WITH*-Selektor im Prinzip die Adresse eines Rekords darstellt. Dieser Zeiger wird in einem für die *WITH*-Anweisung erzeugten Block generiert und mit der Adresse des Records initialisiert. Innerhalb der Anweisungsfolge wird dann über diesen Zeiger auf die Feldelemente zugegriffen (siehe Abb. 22).

2.5.3 Verzweigungen

In Modula-2 gibt es für Verzweigungen die *IF*- und die *CASE*-Anweisung. In C stehen diesen Anweisungen die *if*- und die *switch*-Anweisung gegenüber. Als Besonderheit gibt es in Modula-2 innerhalb von *IF*-Anweisungen noch die *ELSIF*-Anweisung, wobei diese in C auf entsprechende *else if* Folgen abzubilden ist. Problematisch bei der Abbildung der *CASE*-Anweisung auf die *switch*-Anweisung ist, daß die Case-Label-Listen in Modula-2 auch Wertebreiche enthalten können. Um einen solchen Wertebereich innerhalb einer *switch*-Anweisung auszudrücken, müssen für alle Elemente des Wertebereiches entsprechende *case*-Marken erzeugt werden (das können sehr viele sein). Es empfiehlt sich daher auch die *CASE*-Anweisung auf die *if*-Anweisung abzubilden (ein Beispiel für die Übersetzung von Verzweigungen befindet sich im Anhang D).

2.5.4 Schleifen

In Modula-2 gibt es vier verschiedene Anweisungen für die Bildung von Schleifen. Dies sind im einzelnen die *WHILE*-, *REPEAT*-, *FOR*- und *LOOP*-Anweisung. Diesen Modula-2 Schleifenkonstrukten stehen in C die *while*-, *do*- und *for*-Anweisungen gegenüber.

```
(*--------------------  Modula-2  ----------------------*)

MODULE With;
VAR
  R : RECORD
        a, b, c : CARDINAL;
      END (* record *);
BEGIN
  WITH R DO
    a := 1;
    b := 2;
    c := 3;
  END (* with *);
END With.

/*--------------------     C      ----------------------*/

#include "M2rts.h"

/* MODULE */ void With();
static struct PK1 {
  CARDINAL a;
  CARDINAL b;
  CARDINAL c;
} R;

/* MODULE */ void With()
{
  {
    struct PK1  *  _W_PK1 = &R;

    _W_PK1->a=1;
    _W_PK1->b=2;
    _W_PK1->c=3;
  };
}
```

Abb. 22: Beispiel für die Übersetzung einer *WITH*-Anweisung

Die Modula-2 *WHILE*-Anweisung kann direkt auf die C *while*-Anweisung abgebildet werden. Die *REPEAT*-Anweisung wird auf die *do*-Anweisung abgebildet, wobei die Terminierungsbedingung einfach negiert (Bildung des Komplements) wird.

Der *FOR*-Anweisung in Modula-2 steht auch in C eine *for*-Anweisung gegenüber. In Modula-2 wird allerdings der Anfangswert und der Endwert des "Schleifenlaufbereichs" vor dem betreten der Schleife einmal ausgewertet und ist danach festgelegt. In C wird der Endwert bei jedem Schleifendurchlauf neu ausgewertet

und kann daher in der Schleife verändert werden. Es empfiehlt sich daher die Modula-2 *FOR*-Anweisung auf die C *while*-Anweisung abzubilden, wobei für die Anfangs- und Endwerte der Schleife, in einem die Schleife umgebenden Block, entsprechende Hilfsvariablen zu erzeugen sind. Dadurch ist sichergestellt, daß die Modula-2 Semantik bei der Übersetzung der *FOR*-Anweisung auf alle Fälle erhalten bleibt, selbst wenn Variablen, die den Anfangs- oder Endwert des "Schleifenlaufbereichs" angeben, innerhalb der Schleife verändert werden.

Die *LOOP*-Anweisung stellt im Prinzip eine Endlosschleife dar, die durch eine *EXIT*-Anweisung verlassen wird. Diese Schleife kann daher in C einfach durch eine endlose *while*-Schleife nachgebildet werden. Für die Modula-2 *EXIT*-Anweisung wird in C die *break*-Anweisung generiert (ein Beispiel für die Übersetzung von Schleifen befindet sich im Anhang E).

2.6 Systemabhängige Spracheigenschaften

Modula-2 wurde als Allzwecksprache entworfen und sollte daher auch Möglichkeiten für die maschinennahe Programmierung bieten. Da es hier kaum möglich ist, ein allgemeine Schnittstelle zur Hardware anzubieten, sind die entsprechenden Sprachelemente und deren Umfang von der jeweiligen Modula-2 Implementierung abhängig [07, 08]. Da C für die Entwicklung des Betriebssystems UNIX geschaffen und eingesetzt wurde, sind auch hier genügend Möglichkeiten für die maschinennahe Programmierung gegeben. Allerdings gilt auch für C, daß diese Sprachelemente und deren Wirkung vollständig von der jeweiligen C-Implementierung abhängig sind. Dies bedeutet, daß die Problematik der Verwendung maschinennaher Sprachelemente auch nach der Abbildung von Modula-2 auf C (soweit die Abbildung überhaupt möglich ist) erhalten bleibt.

Die Abbildung der maschinennahen Sprachelemente von Modula-2 auf C kann in drei Bereiche gegliedert werden. Der erste Bereich umfaßt diejenigen Sprachelemente, die direkt abgebildet werden können. Dies sind die meisten Typtransferfunktionen und Teile des Moduls SYSTEM. Der zweite Bereich kann ebenfalls abgebildet werden, benötigt allerdings entsprechende Laufzeitunterstützung. Dies betrifft einige Typtransferfunktionen sowie Coroutinen. Der dritte Bereich kann nicht auf C abgebildet werden und betrifft die absolute Adressierung von Variablen.

2.6.1 Typtransferfunktionen

In C ist die Toleranz bzw. Strenge bei der Überprüfung der Typkompatibilität vom jeweiligen C-Compiler abhängig. Einige generieren automatisch Typtransferfunktionen, andere geben Warnungen aus und wieder andere ignorieren das Problem teilweise oder gar vollständig. In Modula-2 sind hingegen Typtransferfunktionen im allgemeinen (auf einige Ausnahmen z.B. bezüglich der Ausdruckskompatibilität von Subranges mit Basistyp *INTEGER* oder *CARDINAL* wird hier nicht näher eingegangen) explizit erforderlich.

Die Abbildung von Basistypkonvertierungen (eine Ausnahme bildet *REAL*) bzw. Zeigern stellt kein großes Problem dar, da dies auch in C erlaubt ist und unterstützt wird. Für Floating-Point-Typen ist in Modula-2 die Verwendung entsprechender Standardfunktionen vorgesehen, sodaß hier durch das Laufzeitsystem eine Unterstützung (aufbauend auf den (Pseudo-) Standard der C-Bibliothek) geboten werden kann. Soll auch die Konvertierung von strukturierten Typen (betrifft z. B. die Zuweisung von *RECORD*s mit unterschiedlichem Typ) möglich sein, so kann auch dies mit entsprechenden Kopieroperationen implementiert werden.

2.6.2 Der Modul *SYSTEM*

In jeder Implementierung von Modula-2 gibt es einen fiktiven Modul *SYSTEM*, der systemabhängige Datentypen und Prozeduren exportiert. Dabei "weiß" der jeweilige Modula-2 Compiler, daß hier ein Sonderfall vorliegt und diese Objekte eine Sonderbehandlung erfordern. Soweit die Abbildung auf C nicht direkt durch den Übersetzer möglich ist, kann auch hier das Laufzeitsystem eine entsprechende Unterstützung bieten. Direkt abbildbar sind die Typen *WORD* und *ADDRESS*, sowie die Funktionen *ADR()*, *SIZE()* und *TSIZE()*, während für den Typ *PROCESS* und die Prozeduren *NEWPROCESS()* und *TRANSFER()* eine Laufzeitunterstützung erforderlich ist.

Soll auch das Prioritäten- und Monitorkonzept, sowie die Prozedur *IOTRANSFER()* auf C abgebildet werden, so muß vom Übersetzer entsprechender Code generiert werden. Da dieser Code absolut von der jeweiligen Hardware und teilweise vom Betriebssystem abhängig ist, können hier keine genaueren Richtlinien für die Abbildung angegeben werden.

In der vorliegenden Implementierung wurde auf die Einführung von Prioritäten und Monitoren überhaupt verzichtet, da dies die Portabilität des Übersetzers stark beeinträchtigen würde. Die Implementierung von Coroutinen wurde nicht direkt im Übersetzer, sondern in einem getrennten Modul realisiert (siehe Anhang H), sodaß auch die Portierung unabhängig voneinander erfolgen kann.

2.6.3 Absolute Adressierung von Variablen

Es ist in einigen Modula-2 Implementierungen möglich, für globale Variablen bereits bei der Deklaration zu bestimmen, auf welchen Speicherstellen sie bei der Ausführung des Programms liegen sollen. Es sei darauf hingewiesen, daß diese Möglichkeit der absoluten Adressierung von Variablen nicht zur Sprachdefinition von Modula-2 gehört [08]. Die direkte Abbildung der, bereits bei der Variablendeklaration festgelegten, absoluten Adressierung auf C ist nicht möglich. Es können in C nur Zeigern absolute Adressen zugewiesen werden, um dann damit absolute Speicherstellen zu adressieren. Das Format der Adressen ist dabei natürlich wieder absolut hardwarespezifisch.

2.7 Kapitelzusammenfassung

In diesem Abschnitt wurde die Abbildung von Modula-2 auf C besprochen. Da Modula-2 und C in ihren Strukturen und Auswertungsstrategien sehr ähnlich sind, ist die Abbildung in vielen Fällen relativ einfach. Trotzdem gibt es einige Fälle, in denen eine direkte Übersetzung nicht möglich ist.

Der wesentlichste Unterschied ist wohl das Modulkonzept von Modula-2, dem in C kein entsprechendes Sprachkonstrukt gegenübersteht. Während eine Modula-2 Applikation aus einer Reihe von Moduln bestehen, ist ein C-Programm einfach eine Folge von Prozeduren. Das Modulkonzept zeichnet sich dabei durch eine Reihe von exakten Vorschriften aus, die die Lebensdauer und Sichtbarkeit von Objekten in Moduln, die Initialisierung von Moduln und den Datenaustausch zwischen Moduln regeln.

Ein weiterer wesentlicher Strukturunterschied ist, daß in Modula-2 Prozeduren ineinander verschachtelt werden können, während C nur über eine "flache" Prozedurstruktur verfügt. Hier wurden in diesem Abschnitt drei unterschiedliche Ansätze für die Übersetzung diskutiert, nämlich die Strukturisierung, erweiterte Parameterlisten und die displayorientierte Strukturisierung.

Auch bei den Datentypen gibt es zwischen den beiden Sprachen eine Reihe von Unterschieden. So bietet etwa C ein reicheres Angebot an Standardtypen als der Modula-2 Sprachstandard. Weiters gibt es Differenzen bei der Deklaration von rekursiven Strukturen, da die Möglichkeiten für Vorausdeklarationen unterschiedlich sind. In Modula-2 gibt es weiters das Konzept des Subrange-Typs, das auch zur Laufzeit die Einhaltung eines angegebenen Wertebereichs garantiert, während es in C kein derartiges Prinzip gibt. Da es in C keine Subranges gibt, können Felder in C auch immer nur positive und bei 0 beginnende Indizes besitzen.

Bei den arithmetischen Ausdrücken gibt es hinsichtlich der Syntax und der Auswertungsstrategien keine wesentlichen Unterschiede. Allerdings verfügt Modula-2 über ein weit strengeres Typkonzept als C. Da hier jedoch die Abbildung von

Modula-2 nach C das Ziel ist und nicht umgekehrt, stellt diese Tatsache kein Problem dar. Bei den Anweisungen offeriert zwar Modula-2 mehr Möglichkeiten als C, allerdings können diese "zusätzlichen" Konstrukte (*REPEAT*-, *LOOP*-, *EXIT*- und *WITH*-Anweisung) mit den in C vorhandenen Sprachelementen realisiert werden. Vorsicht erfordert auch die *FOR*-Anweisung, da hier die Semantik zwischen Modula-2 und C etwas unterschiedlich ist.

Probleme ergeben sich auch bei einigen der systemabhängigen Sprachelemente von Modula-2, die auch teilweise von der jeweiligen Übersetzerimplementierung abhängig sind. Dabei stellen etwa die Typtransferfunktionen im wesentlichen kein Problem dar, während die absolute Adressierung von Variablen nicht direkt abgebildet werden kann.

Das Prioritäten und Monitorkonzept ist mit C allein nicht zu realisieren. Hier muß eine Unterstützung durch die jeweilige Umgebung (Übersetzer, Betriebssystem, Hardware) vorhanden sein. Auch das Coroutinenkonzept ist nur umgebungsspezifisch in C implementierbar (siehe Anhang H).

3 Beschreibung des Modula-2-C Übersetzers

In diesem Abschnitt wird die Entwicklung eines Modula-2-C Übersetzers beschrieben. Soll ein solcher Übersetzer nicht nur zur reinen Transformation von Modula-2 nach C dienen, sondern auch zur Softwareentwicklung eingesetzt werden können, so sind dieselben Aufgaben zu erfüllen, wie sie normalerweise auch Modula-2 Compiler wahrnehmen. Dies umfaßt die Überprüfung der Syntax, die Analyse der Deklarationen und Anweisungen, sowie die Generierung von entsprechendem Code. Der "Unterschied" besteht also darin, daß hier eben C-Sourcecode anstatt von Maschinencode oder Assemblercode erzeugt wird.

Der Modula-2-C Übersetzer wurde in Modula-2 mit Hilfe eines Modula-2 Compilers implementiert. Soll der Übersetzer von diesem unabhängig werden, so genügt die Implementierung der reinen Übersetzeraufgaben nicht, sondern es muß auch eine entsprechende Bibliothek entwickelt werden. Im folgenden Abschnitt liegt jedoch das Hauptgewicht auf der Beschreibung der zentralen Übersetzerkomponenten.

Im folgenden werden zuerst die wichtigsten Anforderungen und Entwicklungskriterien beschrieben. Danach wird ein grober Überblick über die Struktur des Übersetzers gegeben. Weiters folgt eine Beschreibung der wesentlichsten Datenstrukturen, ein Überblick über die Architektur des Übersetzers und die Erklärung der Funktionen der einzelnen Komponenten. Zum Abschluß folgt ein Überblick über den Projektverlauf.

3.1 Anforderungen und Entwicklungskriterien

Aus den Möglichkeiten, die ein Modula-2-C Übersetzer bieten soll, lassen sich auch die wesentlichsten Anforderungen ableiten.

Um die Möglichkeit zu erhalten, den produzierten C-Sourcecode in unterschiedliche Umgebungen zu übertragen, muß dieser Output portabel sein. Dabei ist klar, daß es in der Praxis die universelle C-Maschine nicht gibt, jedoch soll einerseits durch die Einhaltung von (Pseudo-) Standards versucht werden einen gemeinsamen Nenner zu finden und andererseits kann durch die Übergabe entsprechender Parameter an den Übersetzer eine gewisse Flexibilität erreicht werden.

Um als Werkzeug für die Übertragung von Modula-2 nach C sinnvoll zu sein, ist es notwendig Modula-2 möglichst vollständig auf C abzubilden. Als Entwicklungssprache für den Übersetzer wurde Modula-2 gewählt, wobei bei der Implementierung soweit wie möglich der Modula-2 Standard eingehalten wurde. Ist die Abbildung von Modula-2 nach C weitgehend vollständig und der Übersetzer selbst in Modula-2 implementiert, so ergibt sich dadurch die Möglichkeit, daß der Übersetzer sich selbst in C-Sourcecode transformieren kann.

Ist nun der produzierte C-Code auch portabel, so ergibt sich eine zweifache Portabilität: Einerseits ist es möglich, den in Modula-2 implementierten Übersetzer auf relativ einfache Weise in verschiedene Modula-2 Umgebungen zu portieren und andererseits kann auch der, durch die Übersetzung mit sich selbst, gewonnene C-Code in unterschiedliche C-Umgebungen portiert werden.

Man kann die Anforderungen wie folgt zusammenfassen:

a) der erzeugte Code soll so weit als möglich portabel sein.

b) Modula-2 soll möglichst vollständig nach C abgebildet werden (zumindest soweit dabei nicht gegen Anforderung a) verstoßen wird).

c) der Übersetzer soll in der Lage sein, sich selbst in C zu transformieren.

Die Entwicklung soll in zwei Phasen erfolgen. Da die Selbstübersetzung einen wesentlichen Meilenstein bei der Implementierung darstellt, sollen zuerst alle Kräfte eingesetzt werden, um dies zu erreichen. Weil der Übersetzer mit einem Modula-2 Compiler entwickelt wird, kann man davon ausgehen, daß er (der Modula-2-C Übersetzer) aus fehlerfreiem Modula-2 Code besteht.

In der ersten Phase soll der Übersetzer daher so konstruiert werden, daß von einem relativ fehlerfreiem Input ausgegangen wird. Dadurch wird ein schneller Entwicklungsfortschritt erreicht und man erhält relativ schnell eine ausführbare Version des Übersetzers. Dieser kann dann einerseits bereits zum Testen verwendet werden und andererseits zum Bootstrap des Übersetzers dienen.

Dies bedeutet jedoch nicht, daß es keine Fehlerbehandlung im Übersetzer gibt, sondern es werden bestimmte Dinge (z.B. Schalterkomponenten in Variantenrecords) einfach etwas "nachlässig" geprüft bzw. keine aufwendigen Recovery-Maßnahmen (z.B. perfektes Aufsetzen des Parsers nach Syntaxfehlern) implementiert. In der zweiten Phase soll der Übersetzer dann soweit vervollständigt werden, daß ihn auch ein fehlerbehafteter Input nicht mehr aus dem Gleichgewicht bringen kann.

Um Phase zwei nicht unnötig zu komplizieren und allgemein die Wartbarkeit zu erleichtern werden die einzelnen Funktionen (Syntaxanalyse, Deklarationsanalyse, Blockanalyse, Codegenerierung) des Übersetzers streng voneinander getrennt und orientieren sich stark an der Syntax von Modula-2. Durch die starke Syntaxorientierung ergibt sich trotz unterschiedlicher Funktionalität eine einheitliche Struktur der einzelenen Komponeneten. Es ist dafür jedoch erforderlich, daß sich die interne Darstellung des zu transformierenden Inputs im Laufe der Übersetzung nicht zu stark verändert.

Durch die Trennung der einzelnen Funktionen läßt es sich kaum vermeiden, daß der Input mehrmals gelesen werden muß. Um jedoch den Transformationsvorgang durch das Lesen von Interpassfiles nicht unnötig zu verlängern und da sich die

Struktur des Inputs zwischen den einzelnen Funktionen kaum verändert, soll der Modula-2 Sourcefile nur einmal gelesen und in ein entsprechendes internes Format konvertiert werden. Dieses wird dann von den einzelnen Funktionen des Übersetzers bearbeitet und zu guter letzt in C-Sourcecode transformiert.

Da der Übersetzer einerseits auch in Umgebungen mit beschränktem Speicherangebot lauffähig sein soll und andererseits die einzelnen Aufgabenbereiche des Konverters zwar logisch voneinander getrennt, jedoch dieser physisch zu einem (relativ großen) monolithischen Programm gebunden wird, ist eine möglichst kompakte interne Darstellung des Inputs notwendig. Sollten sich dennoch in einer Umgebung Speicherprobleme ergeben, so kann die Implementierung ohne großen Aufwand geändert werden, um jene Strukturen, die für die interne Darstellung des Modula-2 Inputs verwendet werden, auf ein externes Speichermedium auszulagern.

3.2 Übersicht über den Übersetzer

Dieser Abschnitt gibt einen groben Überblick über die vertikale und horizontale Struktur des Modula-2-C Konverters. Die vertikale Struktur zeigt dabei, wie der Übersetzer und auch jede von ihm übersetzte Applikation, über die Modula-2 Bibliothek, auf die jeweilige C-Bibliothek aufsetzt. Die horizontale Struktur zeigt die einzelnen Phasen, die ein Implementierungs- oder Hauptmodul während des Übersetzungsprozesses durchläuft. Dies sind im einzelnen die Syntaxanalyse, die Deklarationsanalyse, die Blockanalyse und zum Schluß die Codegenerierung. Für Definitionsmoduln werden nur die ersten beiden Phasen durchlaufen.

3.2.1 Die Schnittstellen zur Umgebung

Ein Ziel bei der Implementierung des Modula-2-C Übersetzers war, die Abhängigkeit von der jeweiligen Umgebung so gering wie möglich zu halten. Vor allem sollten Zugriffe auf Funktionen des Betriebssystems und Funktionen für die Datenkonvertierung nur auf einige wenige Moduln konzentriert werden. Dadurch sollen eventuell notwendige Änderungen bei einer Portierung möglichst auf diese Moduln beschränkt bleiben. Im Gegensatz zu der in [01] beschriebenene Implementierung wurde jedoch kein unabhängiges Interface zwischen dem Übersetzer und dem Betriebssystem konstruiert. Vielmehr stützt sich der Konverter auf den in [07] und [08] beschriebenen Pseudo-Standard einer Modula-2 Bibliothek, um die Übertragung in unterschiedliche Modula-2 Umgebungen zu erleichtern. Leider wurden die Library-Moduln (FileSystem, InOut, Terminal, Conversions, RealConversion, RealIO) nicht mit derselben Restriktivität definiert, wie die Sprache Modula-2. Daher gibt es keine eindeutig festgelegte Syntax oder Semantik, was zu unterschiedlichen Vorschlägen [09] und Implementierungen geführt hat.

Diese Library-Moduln sind selbstverständlich ebenfalls in Modula-2 implementiert und können daher vom Modula-2-C Konverter in C umgewandelt werden. Das besondere an diesen Moduln ist, daß sie von sogenannten *FOREIGN DEFINITION* Moduln Prozeduren importieren. Dies sind spezielle Moduln, die

nicht zu einem korrespondierenden Modula-2 *IMPLEMENTATION* Modul gehören, sondern zu einem in C implementierten Modul (Modula-2-C Interface Moduln oder kurz M2CIMs). Die von solchen Moduln importieren Prozeduren können auf ganz normale Weise von Modula-2 Moduln genutzt werden. Der einzige Nachteil ist, daß der Übersetzer keine Möglichkeit hat zu überprüfen, ob die in Modula-2 definierten Prozedurheader mit der tatsächlichen C Implementierung übereinstimmen. Auch bei der Implementierung dieser M2CIMs wurde versucht, so weit als möglich C Konventionen und (Pseudo-) Standards einzuhalten, um die Notwendigkeit von Änderungen bei Portierungen auf ein Minimum zu reduzieren.

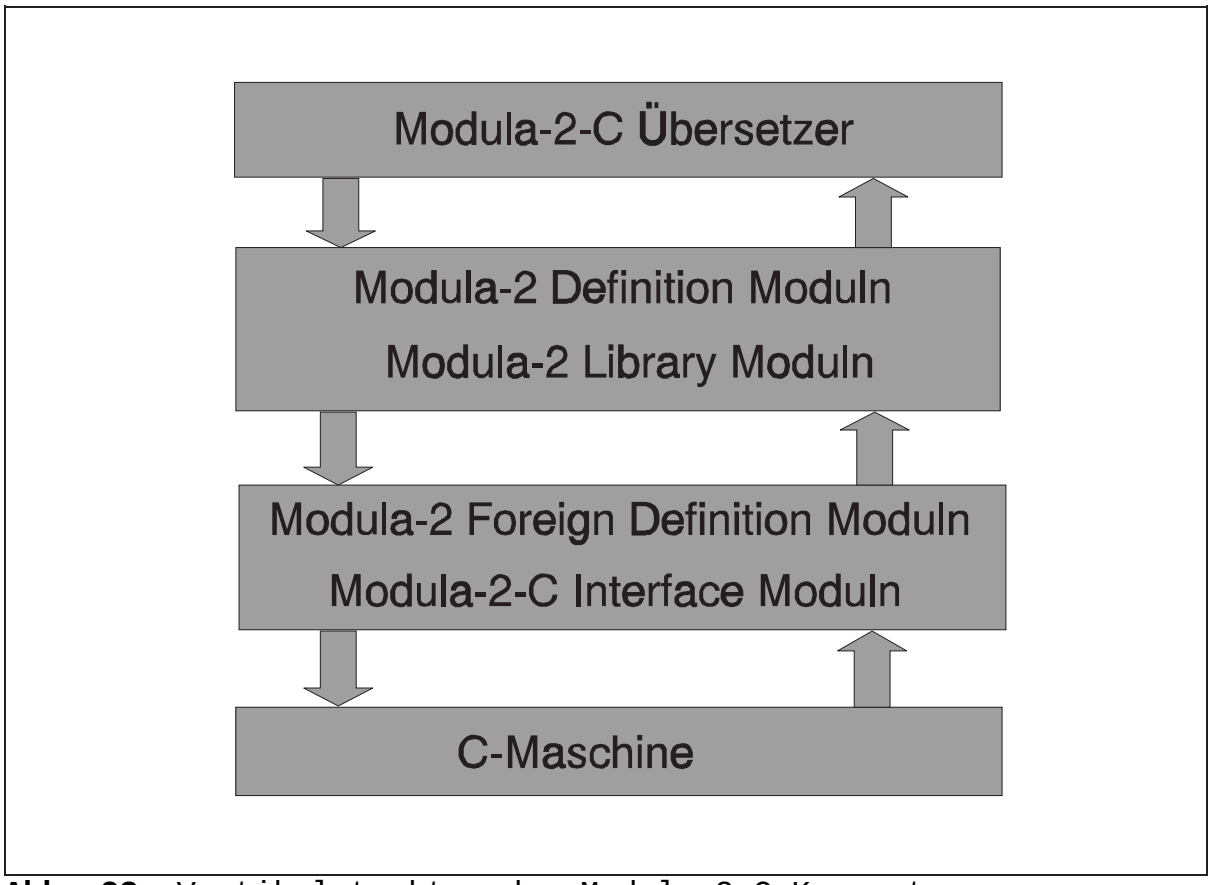

Abb. 23: Vertikalstruktur des Modula-2-C Konverters

Bei einer Portierung, bei der Änderungen notwendig sind, wird man von unten nach oben vorgehen (siehe Abb. 23) und versuchen, die Änderungen auf die M2CIMs zu beschränken. Die oben abgebildete Struktur gilt dabei nicht nur für den Modula-2-C Übersetzer, sondern auch für jede von ihm übersetzte Modula-2 Applikation.

3.2.2 Die Grobstruktur des Übersetzers

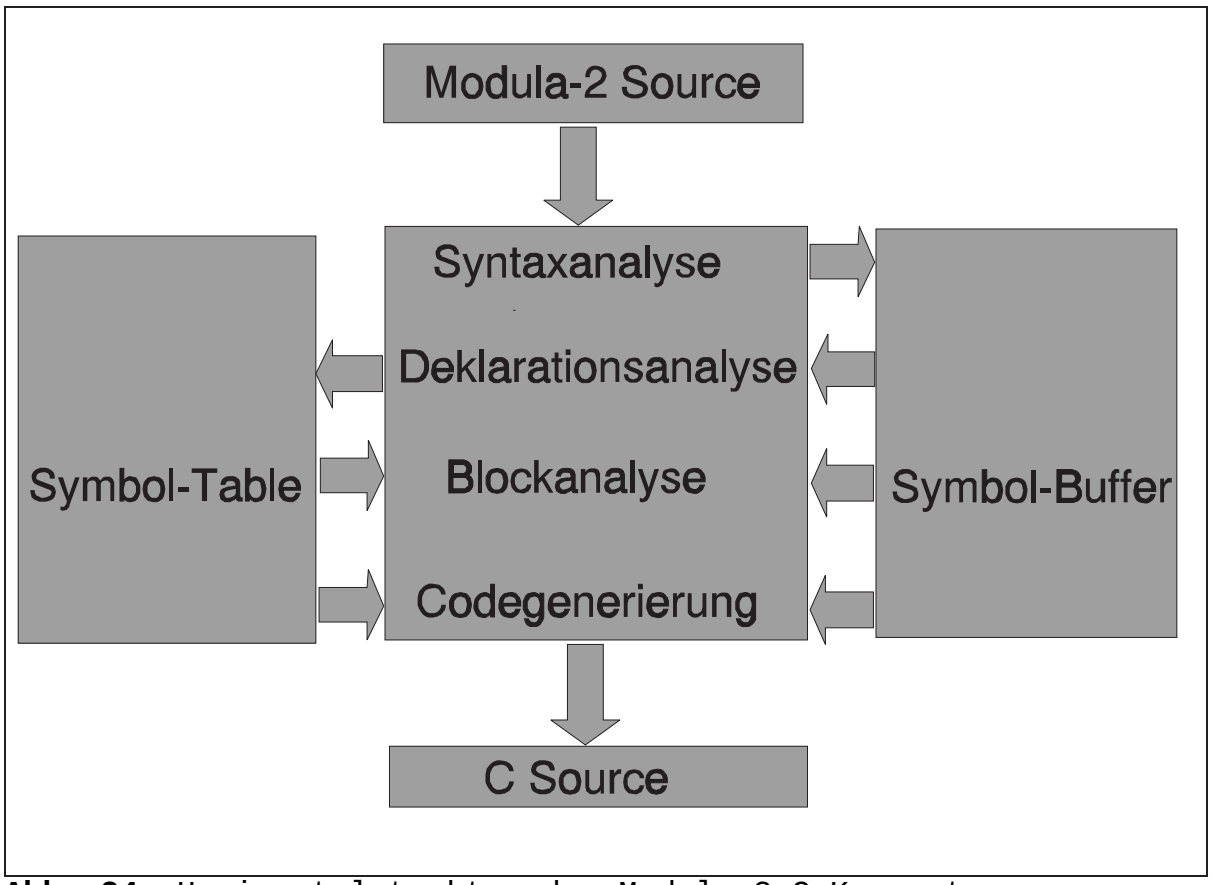

Abb. 24: Horizontalstruktur des Modula-2-C Konverters

Durch die Trennung der einzelnen Übersetzerfunktionen ergeben sich die in Abb. 24 dargestellten Hauptkomponenten.

Der Modula-2 Sourcecode wird von der Syntaxanalyse (Pass 1) vollständig gelesen und im sogenannten Symbol-Buffer für die weitere Bearbeitung abgelegt. Identifier und Konstante werden dabei in einer getrennten Struktur deponiert. Außerdem obliegt es ebenfalls der Syntaxanalyse alle benötigten Definitionsmoduln einzulesen, da keine eigenen Symbolfiles generiert werden. Diese werden ebenfalls auf ihre syntaktische Korrektheit überprüft und im Symbol-Buffer abgelegt.

Von der Deklarationsanalyse (Pass 2) werden dann in einer ersten Phase die Deklarationsteile sämtlicher Moduln und Prozeduren im Symbol-Buffer durchge-

gangen, um daraus die Symbol-Table aufzubauen. In einem zweiten Schritt werden dann die unterschiedlichen Referenzen der Deklarationen aufgelöst und auf ihre Korrektheit überprüft. Danach stehen für die weitere Bearbeitung zwei Informationsquellen zur Verfügung: Einerseits die im Symbol-Buffer abgelegten Informationen über die Anweisungsteile der einzelnen Moduln und Prozeduren und andererseits die Symbol-Table mit den Deklatationsteilen.

In der Blockanalyse (Pass 3) werden dann, ausgehend von den Verweisen der Symbol-Table in den Symbol-Buffer, die Anweisungsteile der zu übersetzenden Moduln und Prozeduren auf ihre Korrektheit hin untersucht.

Zu guter letzt wird dann von der Codegenerierung (Pass 4) der Input aus dem internen Format (Symbol-Table, Symbol-Buffer) in C-Sourcecode umgewandelt, wobei auch hier eine Trennung zwischen der Generierung der Deklarationsteile und der Anweisungsteile vorgenommen wird.

Neben den beschriebenen vier Hauptfunktionsblöcken gibt es noch eine Reihe von Peripheriefunktionen, die Dienstleistungen anbieten, um die Hauptaufgaben zu unterstützen. Zu erwähnen wären hier die Dialogkomponente für das Bearbeiten der an den Übersetzer übergebenen Parameter, das Symbol-Table- und Symbol-Buffer-Management, der Initialisierungsteil für die Initialisierung der einzelnen Übersetzerkomponenten, sowie die Komponenten für das Errorhandling und das Generieren von (Error-) Listings. Des weiteren gibt es auch eine Komponente für das Debugging der vom Übersetzer erzeugten Datenstrukturen, die jedoch nur für die Entwicklung des Übersetzers eingesetzt wird.

3.3 Datenstrukturen des Übersetzers

Der Entwurf der Datenstrukturen des Modula-2-C Konverters richtet sich hauptsächlich nach der Syntax und Semantik von Modula-2. Während jedoch andere Modula-2 Compiler [01, 04] ihre Datenstrukturen teilwiese auf sogenannte Inter-Pass-Files oder Work-Files auslagern, wurde hier versucht darauf zu verzichten, um die File I/O Operationen einzuschränken. Die wesentlichen Informationen über das zu übersetzende Modula-2 Programm werden in der String-Table, in der Symbol-Table und im Symbol-Buffer abgelegt.

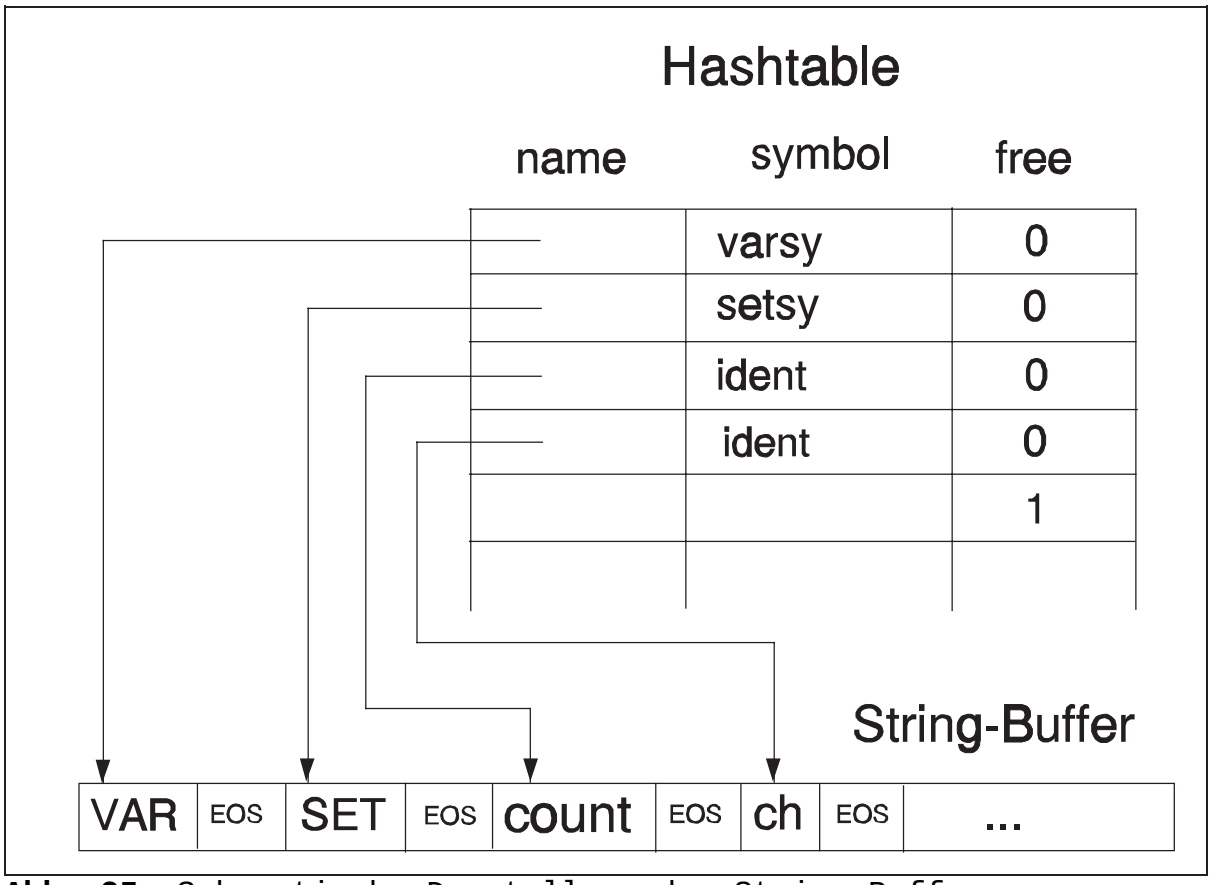

Abb. 25: Schematische Darstellung des String-Buffers

3.3.1 String-Table

Eine Eigenheit von Modula-2 ist, daß Identifier (theoretisch) eine beliebige Länge haben können, was den Entwurf einer geeigneten Datenstruktur nicht unbedingt

erleichtert. Weiters ist die Behandlung von Namen weit effizienter und einfacher, wenn diese in eine numerische Darstellungsform umgewandelt werden. Daher wurde auf eine in [12] vorgeschlagene Art der Ablage von Identifiern zurückgegriffen, wobei dieses Speichern aus Effizienzgründen mit Hilfe von Schlüsseltransformationen [19] erfolgt (siehe Abb. 25). Ein Zugriff auf die String-Table ist jedoch nur über eigene Prozeduren möglich, sodaß die Art der Implementierung jederzeit leicht geändert werden kann, ohne irgendwelche Änderungen in anderen Teilen des Konverters zu verursachen.

Neben Identifiern werden auch normale Zeichenketten (Strings) in dieser Datenstruktur abgelegt, wobei jedoch die Schlüsseltransformation entfällt. Die numerische Repräsentation der Identifier und Strings wird in der Symbol-Table abgelegt und dient als Index in die beschriebene Datenstruktur, um die entsprechenden Zeichenketten für die Codegenerierung wiederzufinden.

3.3.2 Symbol-Table

Die wohl wichtigste Datenstruktur des Modula-2-C Übersetzers ist die Symbol-Table. In ihr werden sämtliche Informationen über die Deklarationsteile des zu übersetzenden Moduls und über alle benötigten Definitionsmoduln abgelegt. Weiters wird sie für die Überprüfung der kontext-sensitiven Regeln von Modula-2 und für die Codegenerierung verwendet.

Im Gegensatz zu anderen Modula-2 Comilern [01, 04] wurde die Symbol-Table nicht als inhaltlich global bekannte Typstruktur implementiert, sondern als abstrakter Datentyp (siehe Abb. 26). Dies hat zwar zur Folge, daß die Zugriffe auf die Symbol-Table etwas ineffizienter werden, hat jedoch den Vorteil, daß eine Änderung der Implementierung (z.B. Auslagern der Daten auf externe Speichermedien, falls dies durch extreme Speicherbeschränkungen notwendig sein sollte) leicht durchgeführt werden kann und somit die Flexibilität der Anpassung des Übersetzers an unterschiedliche Umgebungen erhöht wird. Wichtig ist auch, daß dadurch sämtliche Zugriffe auf die Datenstrukturen korrekt erfolgen und kein "Wissen" über die interne Darstellung der Daten enthalten (z.B. welche Felder

sich bei Variantenrecords überlappen, binäre Darstellung von Konstanten, etc.), um die Portabilität nicht zu gefährden.

```
DEFINITION MODULE M2Sym;

TYPE
  SymTbl;
  Object;
  Struct;
  Const;

  Symbol =      (eop, andsy, divsy, times, slash, modsy, notsy,
                plus, minus, orsy, eql, neq, grt, geq, lss, leq,
                insy, lparent, rparent, lbrack, rbrack, lconbr,
                rconbr, comma, semicolon, period, colon, range,
                constsy,  typesy,  varsy,  arraysy,  recordsy,
                variant,   setsy,   pointersy,   tosy,   arrow,
                importsy,   exportsy,   fromsy,   qualifiedsy,
                beginsy, casesy, ofsy, ifsy, thensy, elsifsy,
                elsesy,  loopsy,  exitsy,  repeatsy,  untilsy,
                whilesy, dosy, withsy, forsy, bysy, returnsy,
                becomes, endsy, definitionsy, implementationsy,
                foreignsy, proceduresy, modulesy, ident, const,
                endblock, option, errorsy, eol, field);

  ModKind =     (MainMod,  ImpMod,  MainDefMod,  Foreign,  Mod,
                DefMod);

  VarKind =     (noparam, varparam, valparam);

  State =       (imported, exported, innerimport, innerexport,
                local);

  Stpures =     (incp, decp, newp, disp, inlp, exlp, hltp, ilnp,
                absf, adrf, capf, chrf, fltf, higf, maxf, minf,
                oddf, ordf, sizf, trcf, tszf, valf);

(*********************************************************)

 (* Prozeduren für Zugriffe auf die deklarierten Strukturen *)
                                ...
END M2Sym.
```

Abb. 26: Definition-Modul für den Symbol-Table

Im folgenden wird die konkrete Implementierung der wesentlichsten Datentypen der Symbol-Table (siehe Abb. 27) näher erläutert. Der Typ *Symbol* reflektiert die Syntax von Modula-2 und wird für deren interne Darstellung verwendet (vor allem im Symbol-Buffer). Der Typ *ModKind* dient zur Unterscheidung der

verschiedenen Modularten (Hauptmodul, Implementierungsmodul, Hauptdefinitionsmodul (dies ist der Definitionsmodul des Implementierungsmoduls, der gerade übersetzt wird), Definitionsmodul, Foreign-Definition-Modul, Modul) und *VarKind* gibt die möglichen Arten von Variablen (normale Variablen, Parameter by Value, Parameter by Reference) an.

Der mögliche Status von Objekten (siehe abstrakter Datentyp *Object* in Abb. 28) wird durch *State* dargestellt, wobei zwischen Objekten unterschieden wird, die von äußeren oder inneren Moduln importiert bzw. exportiert werden und solchen die nur lokal verwendet werden. Der Aufzähltyp *Stpures* repräsentiert die implementierten Standardprozeduren und Standardfunktionen.

Im folgenden wird die Implementierung der einzelnen abstrakten Datentypen näher erläutert, wobei es Datenstrukturen für die Symbol-Tables, Objekte (Moduln, Prozeduren, Variablen, ...), Typstrukturen (*ARRAY*s, *RECORD*s, ...) und Konstanten gibt.

```
TYPE
  SymTbl  = POINTER TO SymRec;
  ObjList = POINTER TO ListRec;

  ListRec = RECORD
              object : Object;
              next   : ObjList;
            END (* record *);

  SymRec =  RECORD
              root,
              current : ObjList;
            END (* record *);
```

Abb. 27: Beschreibung von *SymTbl*

Der Datentyp *SymTbl* (siehe Abb. 27 und Beispiel in Abb. 33) dient zur Verwaltung von Symbol-Tables. Die Recordvariable *root* ist dabei ein Zeiger auf die Wurzel einer Objektliste, während *current* auf das zu bearbeitende Objekt in der Liste zeigt. Der Typ *ObjList* ist nur im Implementierungsteil des Moduls für

die Symbol-Table-Verwaltung (**M2Sym**) deklariert und somit für andere Moduln nicht zugänglich.

Die einzelnen Objekte (Datentyp *Object*, siehe Abb. 28) werden in linearen Listen abgelegt, wobei die Möglichkeit besteht, diese nach deren Namen, oder der Reihenfolge der Deklaration der einzelnen Objekte innerhalb eines Moduls oder einer Prozedur, zu sortieren, um dadurch die Zugriffsalgorithmen etwas zu optimieren. Die Informationen, die für die Verwaltung der Listen notwendig sind, wurden nicht direkt in *Object* integriert, um dadurch ein Objekt auch in unterschiedlichen Listen ablegen zu können, ohne es kopieren zu müssen. Somit repräsentiert der Zeiger auf ein Objekt auch das Objekt selbst, womit Vergleiche recht einfach durchzuführen sind.

Der Datentyp *Object* (siehe Abb. 28) dient für die interne Repräsentation von Konstanten, Datentypen, Variablen, Elemente von Records, Prozeduren und Moduln (siehe Datentyp *Class*) [01]. Außerdem gibt es noch Objekte, deren Klassifizierung nicht sofort möglich ist und die daher temporär als unbekannte Objekte gelten. Dies gilt vor allem für Objekte die von anderen Moduln importiert werden und deren Klassifizierung erst von der Deklarationsanalyse vorgenommen wird. Die Datenstruktur besteht dabei aus einem allgemeinen Teil und klassenspezifischen Varianten.

Die wesentlichsten Felder des allgemeinen Teiles eines Objektes sind der Name (*name* ist ein Index in den String-Buffer), die Beschreibung des Datentyps des Objektes (*idtyp*, siehe Datentyp *Struct* in Abb. 29) und ein Zeiger auf das Objekt, das die Umgebung (Modul oder Prozedur) des dargestellten Objektes bildet (*envp*). Weiters gibt es Felder für die Position des Objektes im Modula-2 Sourcefile (*line*, *col*) und für die Position im Symbol-Buffer (*pos*). Das Feld *seq* dient für das festhalten der Reihenfolge von Deklarationen, *state* beschreibt den Status eines Objekts (siehe Typ *State* in Abb. 26) und *used* gibt an ob ein deklariertes Objekt auch im Anweisungsteil tatsächlich verwendet wird.

Konstante Ausdrücke werden in mehreren Stufen ausgewertet. Daher werden zuerst die Elemente eines arithmetischen Ausdruckes in einer eigenen Symbol-

```
TYPE
  Object  = POINTER TO ObjRec;

  Class = (consts, types, vars, fields, pures, mods, unknown);

  ObjRec = RECORD
             seq,                    (* sequence number *)
             pos,                    (* buffer position *)
             name      : CARDINAL; (* object name *)
             line, col : CARDINAL; (* file position *)
             envp      : Object;   (* object environment *)
             idtyp     : Struct;   (* type of object *)
             state     : State;    (* object state *)
             used      : BOOLEAN;  (* object use *)
             CASE class : Class OF
               consts  : (* constants *)
                         cids   : SymTbl;
                         cvalue : Const;
             | types   : (* types     *)
             | vars    : (* variables *)
                         vkind  : VarKind;
             | fields  : (* record fields *)
             | pures   : (* procedures *)
                         CASE isstandard : BOOLEAN OF
                           TRUE  : pname : Stpures;
                         | FALSE : pcode : CARDINAL;
                                   hasnested : BOOLEAN;
                                   pimpp,
                                   pextp,
                                   pcodeid,
                                   plocp : SymTbl;
                         END (* case *)
             | mods    : (* modules *)
                         mimpp,
                         mextp,
                         mexpp,
                         mcodeid,
                         mlocp   : SymTbl;
                         mkind   : ModKind;
                         mcode   : CARDINAL;
                         qualexp : BOOLEAN;
             | unknown :
             END (* case *)
           END (* record *);
```

Abb. 28: Beschreibung von *Object*

Table (*cids*) abgelegt und nach der Auswertung wird das Ergebnis in *cvalue* (siehe Typ *Const* in Abb. 30) abgelegt.

Für Typen, Elemente innerhalb von Records und temporär unbekannte Objekte ist keine weiter Information notwendig, während für Variablen die jeweilige Art in *vkind* (siehe Typ *VarKind* in Abb. 26) gespeichert wird.

Bei Prozeduren wird zwischen Standardprozeduren (*pname*, siehe auch Typ *Stpures* in Abb. 26) und normal deklarierten Prozeduren unterschieden. Des weiteren gibt es eine Variable, die anzeigt ob eine Prozedur noch weitere in ihr deklarierte (verschachtelte) Prozeduren enthält. Die restlichen Felder für Prozeduren sind in ihrer Bedeutung mit denen für Moduln identisch und werden daher gemeinsam beschrieben. Die Felder *pcode* bzw. *mcode* sind Indizes in den Symbol-Buffer zu den entsprechenden Anweisungsteilen. Die Felder *pimpp* und *mimpp* dienen für das Ablegen von Objekten, die in die jeweilige Umgebung (für Prozeduren implizit durch Exporte von innern Moduln) importiert werden. Die Variablen *pextp* und *mextp* dienen für das Ablegen von Elementen von Aufzähltypen, um mehrfache Deklarationen bzw. Importe derartiger Elemente zu erkennen. Die Felder *pcodeid* und *mcodeid* dienen für das Speichern von Objekten der jeweiligen Anweisungsteile, während in *plocp* und *mlocp* die lokal deklarierten Objekte abgelegt werden.

Weitere Felder, die nur für Moduln verwendet werden, sind *mexpp* für exportierte Objekte, *qualexp* um qualifizierten Export anzuzeigen und *mkind* bestimmt die Art (siehe Typ *ModKind* in Abb. 26) des beschriebenen Moduls.

Der Datentyp *Struct* (siehe Abb. 29) ist ein Record (oder genauer gesagt, ein Zeiger auf einen Record) der für die Darstellung der verschiedenen möglichen Typstrukturen (siehe Typ *StructForm* in Abb. 29) von Modula-2 dient. Der allgemeine Teil dieses Records besteht aus den Feldern *tcheck*, um mehrmaliges Überprüfen bzw. Generieren von solchen Strukturen zu vermeiden, und *typeid*, welches eine Verbindung zum Namen der Struktur herstellt (falls ein solcher existiert).

Für Enumerationstypen wird die Anzahl der Elemente (*count*) ermittelt, sowie eine Liste mit den einzelnen Elementen angelegt (*list*). Bei Subrangetypen (darunter fallen hier der Einfachheit halber sämtliche Standardtypen, inkl. der

```
TYPE
  Struct  = POINTER TO StructRec;

  StructForm = (enums, subranges, pointers, sets, arrays,
                records, proctypes, hiddens, opens);

  StructRec = RECORD
                tcheck : BOOLEAN;
                typeid : Object;
                CASE form : StructForm OF
                  enums     : count : CARDINAL;
                              list  : SymTbl;
                | subranges : type  : Struct;
                              min, max : Object;
                | pointers  : elemp : Struct;
                | sets      : basep : Struct;
                | arrays    : elp,
                              ixp : Struct;
                              dyn : BOOLEAN;
                | records   : fieldp,
                              casep  : SymTbl;
                | proctypes : params : SymTbl;
                              funcp  : Struct;
                | hiddens   : (* hidden type  *)
                | opens     : (* unknown type *)
                END (* case *)
              END (* record *);
```

Abb. 29: Beschreibung von *Struct*

Typ *REAL* als Sonderfall) wird einerseits die untere (*min*) und die obere Grenze (*max*) und andererseits der diesen Grenzen zugrundeliegende Basistyp (*type*) ermittelt (die Standardtypen zeigen hier auf sich selbst). Für Zeigertypen wird der Datentyp des Elements (*elemp*), auf das gezeigt wird, abgelegt und bei Mengen der entsprechende Basistyp (*basep*) des Sets. Für Felder wird neben dem Elementtyp (*elp*) und dem Typ des Index (*ixp*) auch noch gespeichert, ob es sich um ein dynamisches (*dyn*) oder normales Feld handelt. Bei Records ist es notwendig neben den einzelnen Recordfeldern (*fieldp*) auch etwaige Variantenstrukturen (*casep*) in der Datenstruktur abzulegen. Für Prozeduren werden die Parameter (*params*) und im Fall einer Funktionsprozedur auch der Funktionstyp (*funcp*) gespeichert. Für abstrakte Datentypen und offene Typen (dies sind Datentypen deren Struktur temporär unbekannt ist) sind keine weiteren Informationen notwendig.

Der Datentyp *Const* (siehe Abb. 30) dient für die interne Darstellung von

Konstanten. Konstante Ausdrücke werden ausgewertet und in der angegebenen Datenstruktur abgelegt. Eine Auswertung ist vor allem deshalb notwendig, um den Typ von Subranges exakt bestimmen zu können. Stringkonstante werden im String-Buffer abgelegt und durch den entsprechenden Bufferindex (*string*) repräsentiert.

```
TYPE
  Const   = POINTER TO ConstRec;

  ConstRec = RECORD
               CASE CARDINAL OF
                 0 : bool   : BOOLEAN;
               | 1 : char   : CHAR;
               | 2 : int    : INTEGER;
               | 3 : card   : CARDINAL;
               | 4 : real   : REAL;
               | 5 : bitset : BITSET;
               | 6 : string : CARDINAL; (* string table ptr *)
               END (* case *)
             END (* record *);
```

Abb. 30: Beschreibung von *Const*

3.3.3 Symbol-Buffer

Im Symbol-Buffer wird der zu übersetzende Modula-2 Input in symbolischer Form abgelegt. Während die Deklarationsteile später in die Symbol-Table übertragen werden, bleiben die Anweisungsteile der zu übersetzenden Moduln und Prozeduren im Symbol-Buffer und werden von der Blockanalyse und der Codegenerierung verarbeitet.

Der Symbol-Buffer ist dabei ein Feld vom Typ *Symbol* (siehe Abb. 26), womit im Vergleich zu anderen Darstellungsformen [10, 18], eine relativ kompakte Repräsentation des Inputs erreicht wird. Identifier und Konstante werden von der Syntaxanalyse einfach in linearen Listen (vom Typ *SymTbl*) abgelegt und später von der Deklarationsanalyse in die Symbol-Table übernommen (siehe Abb. 35).

Dieser Abschnitt beschreibt die Implementierung der einzelnen Übersetzerkomponenten und deren Aufteilung auf unterschiedliche Moduln. Der Hauptmodul des Übersetzers hat nur die Aufgabe die Ausführung der anderen Übersetzerteile zu veranlassen. Sämtliche von ihm kontrollierten Teile (mit gewissen Einschränkungen bei der Dialogkomponente) wurden so entwickelt, daß ein mehrmaliger Aufruf möglich ist. Dies bedeutet, daß dem Übersetzer mehrere Modula-2 Sourcefiles auf einmal zur Bearbeitung übergeben werden können. Tritt während der Übersetzung in einer Phase ein Fehler auf, so werden die nachfolgenden Phasen nicht mehr durchlaufen, sondern es wird ein Fehlerlisting generiert.

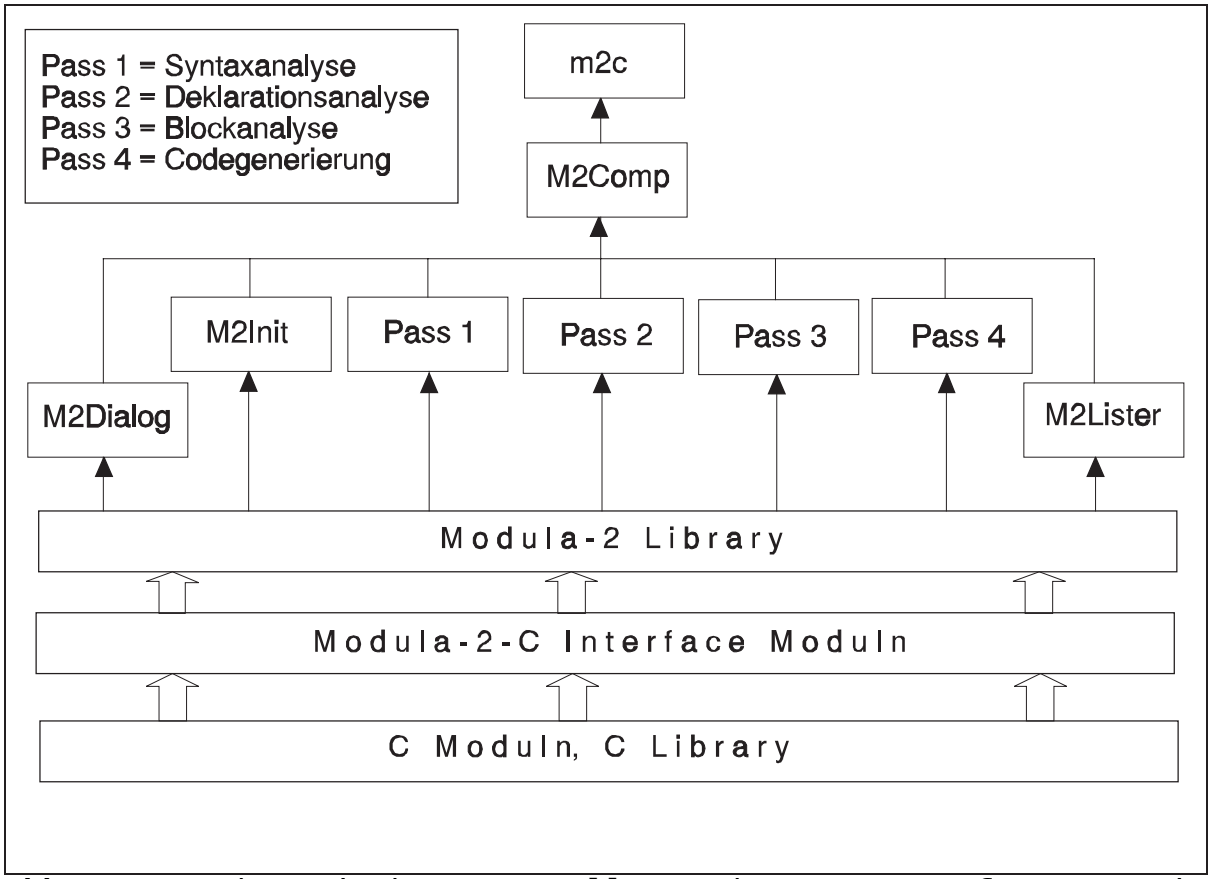

Abb. 31: Schematische Darstellung der Importreferenzen im Modula-2-C Übersetzer

In Abb. 31 werden schematisch die Importreferenzen im Übersetzer gezeigt (ein Pfeil von Modul A nach Modul B bedeutet, daß B von A Objekte importiert). Der

(sonst leere) Hauptmodul **m2c** importiert **M2Comp**, der der eigentliche Kontrollmodul für den Übersetzungsprozeß ist. In **M2Comp** wird im Prinzip folgende Schleife durchlaufen (die Namen in den Klammern sind die Moduln, die die entsprechenden Funktionen zur Verfügung stellen):

```
SCHEMATIC MODULE M2Comp;
BEGIN (* M2Comp *)
        Lies die Kommandozeilenparameter (M2Dialog);
        WHILE (Moduln vorhanden (M2Dialog)) DO
                Initialisierung (M2Init);
                Pass 1 (M2P1);          (* Syntaxanalyse        *)
                Pass 2 (M2P2);          (* Deklarationsanalyse *)
                Pass 3 (M2P3);          (* Blockanalyse         *)
                Pass 4 (M2P4);          (* Codegenerierung      *)
                Lister (M2Lister);      (* eventuell Listings   *)
        END (* while *);
END M2Comp.
```

Die Parameter (Namen von Modula-2 Source-Dateien, Optionen), die dem Übersetzer übergeben werden, werden in **M2Dialog** gelesen und dort aufbewahrt. Bei jedem Schleifendurchlauf wird in weiterer Folge geprüft, ob noch Moduln zum übersetzen vorhanden sind. Ist dies der Fall, so werden diese von **M2Dialog** geöffnet, sonst terminiert die Schleife in **M2Comp**. Wird ein Definitionsmodul übergeben, so wird im Prinzip nur Pass 1 und Pass 2 durchlaufen. Für die restlichen Phasen, wird zwar der jeweilige Kontrollmodul betreten (**M2P3**, **M2P4**), jedoch sofort wieder verlassen, wenn festgestellt wird, daß ein Definitionsmodul vorliegt. Ein Listing wird nur im Fehlerfall generiert bzw. bei Übergabe einer entsprechenden Option.

In Abb. 31 wird nur das Grundschema des Übersetzers, mit den wichtigsten Funktionsblöcken gezeigt. Moduln, die von fast jedem dieser Funktionsblöcke in Anspruch genommen werden, wurden weggelassen, um die Abbildung nicht zu überladen. Dies sind z.B. Moduln wie **M2Sym** (symbol-table-handling), **M2Buffer** (symbol-buffer-handling), **M2Hash** (string-table-handling) und **M2Error** (error-

handling).

Auch alle weiteren Abbildungen, die Importreferenzen zwischen Moduln zeigen, beschränken sich auf die wesentlichsten Modulverbindungen. So werden etwa Verbindungen zur Modula-2 Bibliothek und zu Moduln, die im jeweiligen Fall nur eine unwesentliche Funktion zur Verfügung stellen, weggelassen. Das Ziel der Abbildungen ist nicht, die Importreferenzen vollständig zu dokumentieren, sondern einen Überblick über die Architektur des Übersetzers zu geben.

Für die einzelnen Moduln gilt folgendes Namensschema: Moduln deren Namen mit **M2?*** ("?" steht für einen Buchstabe ungleich "P" und "*" bezeichnet eine beliebige Zeichenkette) beginnen, gehören zum Übersetzer und bieten allgemeine Dienstleistungen (z.B. Verwalten von Datenstrukturen, Auswerten von Konstantenausdrücken, usw.) an. Eine Ausnahme bildet **M2Comp** der im Prinzip als der eigentliche Hauptmodul angesehen werden kann. Moduln deren Namen mit **M2Px*** beginnen, gehören zum Übersetzer und beinhalten phasenspezifische Funktionen, wobei x die Nummer der Übersetzerphase angibt (z.B. **M2P1*** sind alle phasenspezifischen Moduln von Pass 1). Ein **M2Px*** Modul importiert keine Objekte von einem **M2Py*** Modul (für x ungleich y). Moduln deren Namen nicht mit **M2*** beginnt gehören zur Modula-2 Bibliothek.

3.4.1 Dialog und Initialisierung

Die Dialogkomponente hat die Aufgabe, die an den Übersetzer übergebenen Parameter (Files und Optionen) einzulesen und zu speichern. Die Dialogkomponente hat auch die Aufgabe, die übergebenen Dateien (sobald sie benötigt werden) zu suchen und zu öffnen. Dies gilt nicht nur für die übergebenen Dateien, sondern in weiterer Folge auch für die von diesen importierten Definitionsmoduln. Entsprechende Anforderungen werden später von Pass 1 an **M2Dialog** übergeben.

Von der Dialogkomponente wird auch eine sogenannte "Konfigurationsdatei" eingelesen, welche Informationen über die Umgebung des Übersetzers enthält

(siehe Abb. 32). Dies sind im einzelnen Informationen über die maximale Länge von Dateinamen, Angaben über jene Verzeichnisse wo Definition-Moduln und Moduln zu finden sind, bzw. wo der erzeugte C-Sourcefile oder (Error-) Listings abzulegen sind. Dadurch wird es einerseits erleichtert Ordnung zu schaffen und andererseits ist es dadurch möglich die Syntax der Pfadangaben leicht an unterschiedliche Umgebungen anzupassen.

```
MOD = C:\M2C\MOD\
DEF = C:\M2C\DEF\
C__ = C:\M2C\C\
LST = C:\M2C\LST\
```

Abb. 32: Beispiel einer Konfigurationsdatei

Der Initialisierungsteil hat die Aufgabe die restlichen Übersetzerkomponenten zu intitialisieren. Sollen mit einem Aufruf des Übersetzers mehrere Modula-2 Sourcefiles übersetzt werden, so wird nur einmal, beim ersten Modul, eine vollständige Initialisierung durchgeführt, während bei den restlichen Moduln eine verkürzte Initialisierung vorgenommen wird.

Die Initialisierung umfaßt dabei im einzelnen, das Eintragen der in Modula-2 reservierten Wörter (*MODULE*, *VAR*, ...) in die String-Table, sowie die Erzeugung der Symbol-Table (siehe Abb. 33) mit der Eintragung der Standardtypen (*CHAR*, *CARDINAL*, *INTEGER*, ...), der Standardkonstanten (*TRUE, FALSE*, *NIL*) , der Standardprozeduren (*CHR*, *ABS*, ...) und die Erzeugung des Moduls *SYSTEM* und sämtlicher von diesem exportierten Objekten (*BYTE*, *WORD*, *ADDRESS*, ...).

Der Dialog- und Initialisierungsteil besteht hauptsächlich aus den Moduln:

M2Dialog	Dialogkomponente
M2Init	Initialisierung

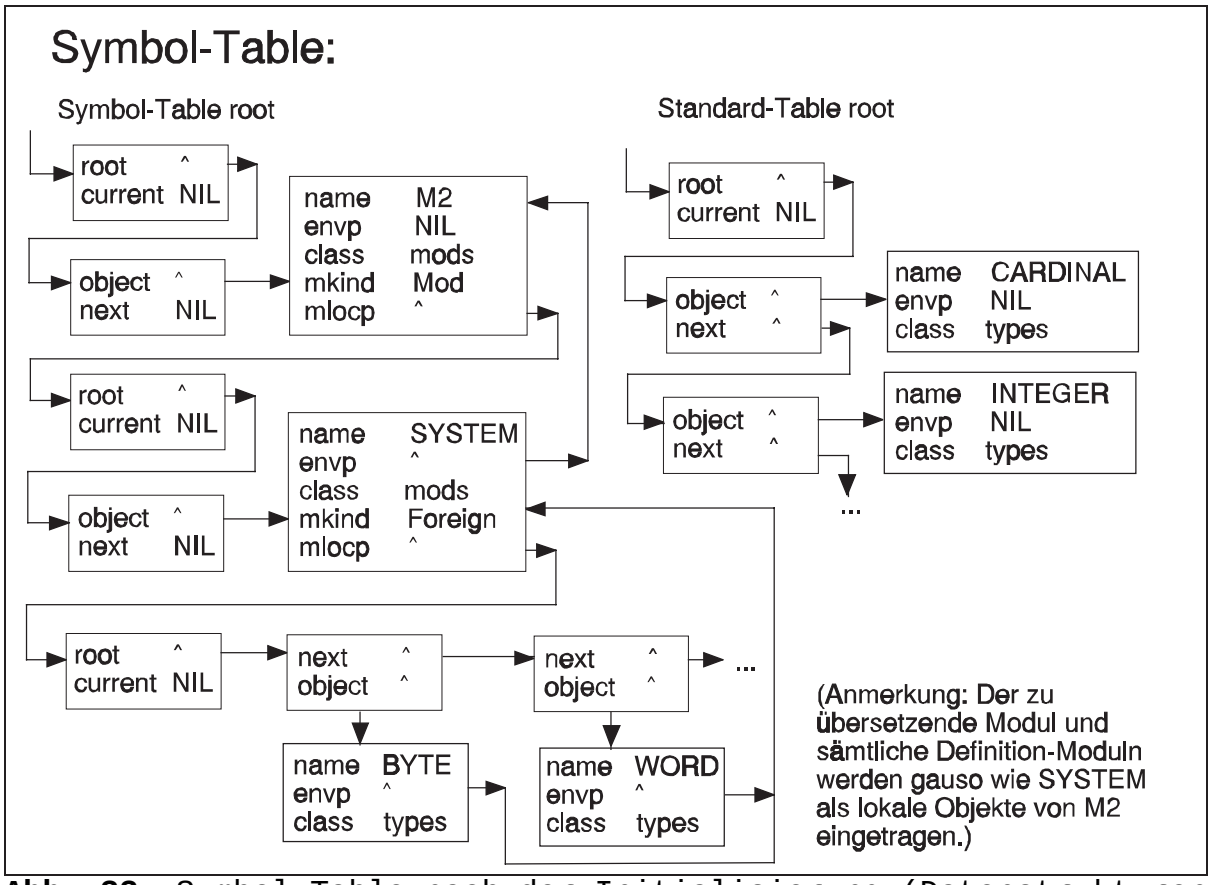

Abb. 33: Symbol-Table nach der Initialisierung (Datenstrukturen vereinfacht dargestellt)

3.4.2 Syntaxanalyse

Während der Syntaxanalyse wird die syntaktische Korrektheit des zu übersetzenden Modul-2 Programms überprüft und es wird in seiner symbolischen Form im Symbol-Buffer abgelegt (siehe Abb. 35). Die Syntaxanalyse besteht im wesentlichen aus insgesamt drei Moduln:

M2P1	Kontrollmodul für die Syntaxanalyse
M2P1Scan	Lexikalische Analyse (Scanner)
M2P1Pars	Syntaktische Analyse (Parser)

Der Kontrollmodul (siehe Abb. 34) initialisiert den Scanner und den Parser. Danach wird die Kontrolle an den Parser übergeben, der die syntaktische Korrektheit des Inputs überprüft und diesen im Symbol-Buffer (**M2Buffer**) ablegt. Dabei werden vom Parser immer wieder Prozeduren im Scanner

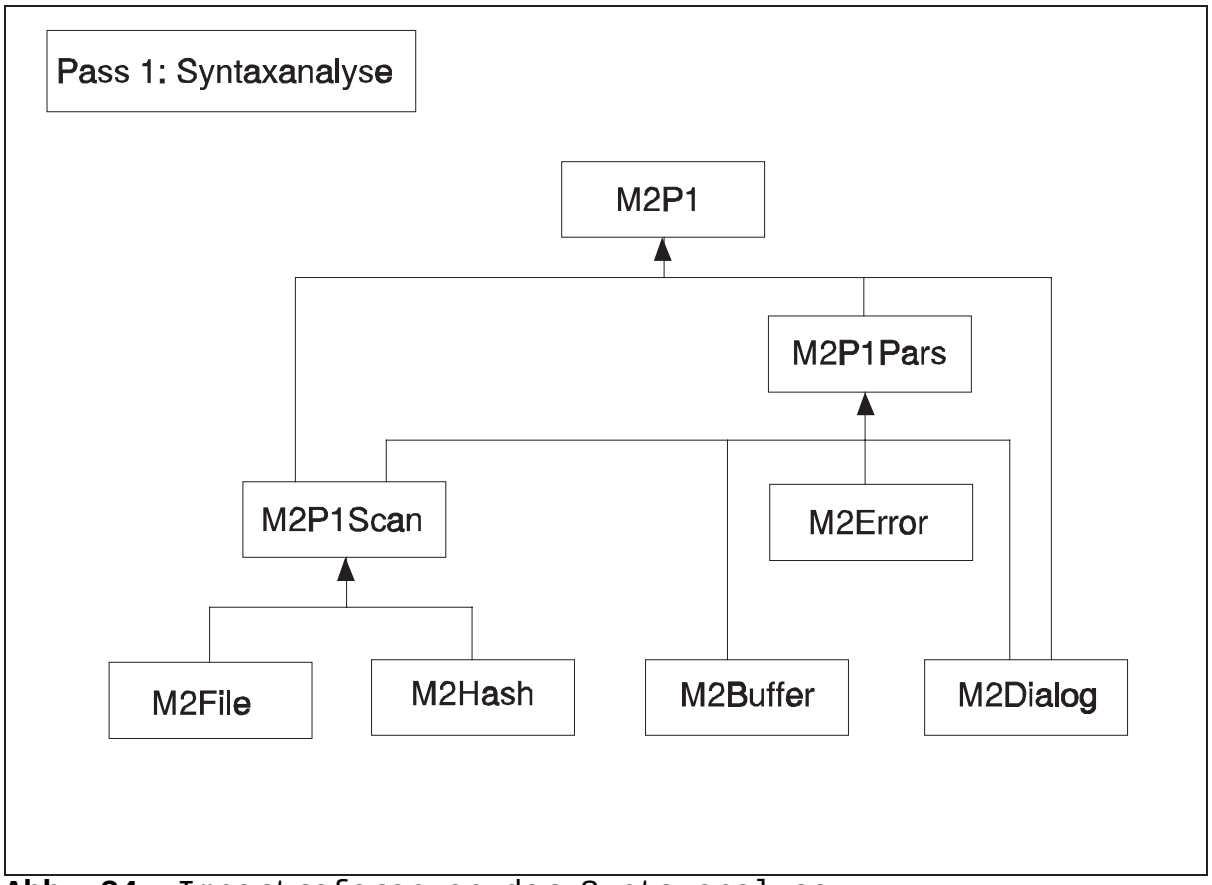

Abb. 34: Importreferenzen der Syntaxanalyse

aufgerufen, wenn ein neues Symbol (Token) benötigt wird. Nachdem ein Symbol für die Analyse der Syntax verwendet wurde, wird es vom Parser im Symbol-Buffer abgelegt.

Der Scanner hat die Aufgabe den Input von "unwesentlichen" Dingen (Blanks, Zeilenstruktur, Kommentare) zu "säubern" und diesen in eine leichter handhabbare Form umzuwandeln. Dies umfaßt das Erkennen von syntaktischen Symbolen (Terminalsymbolen), das Ablegen von Bezeichnern (Identifieren) in der String-Table (**M2Hash**), sowie die Konvertierung von Konstanten. Der Scanner bedient sich des Moduls **M2File**, um Zeichen aus einer Datei zu lesen.

Die Namen von in Importanweisungen spezifizierten Moduln werden vom Parser an **M2Dialog** weitergegeben und dort aufbewahrt. Dadurch können dann später die benötigten Definitionsmoduln eingelesen werden. Der Kontrollmodul **M2P1** hat folgende Struktur:

```
SCHEMATIC MODULE M2P1;
BEGIN (* M2P1 *)
        Initialisiere den Scanner (M2P1Scan);
        Aufruf des Parsers für die Syntaxanalyse  (M2P1Pars);
        WHILE (Definition Moduln einzulesen (M2Dialog)) DO
                Initialisiere den Scanner (M2P1Scan);
                Aufruf des Parsers für die Syntaxanalyse  (M2P1Pars);
        END (* while *);
END M2P1.
```

Nachdem der Parser einen Modul verarbeitet hat, überprüft der Kontrollmodul (mit Hilfe von **M2Dialog**), ob es notwendig ist Definitionsmoduln zu lesen. Ist dies der Fall, so wird der Scanner und der Parser erneut aufgerufen, um diese Moduln ebenfalls auf deren syntaktische Korrektheit zu überprüfen. Der Symbol-Buffer enthält nach der Syntaxanalyse hintereinander sämtliche für die Übersetzung benötigten Moduln in symbolischer Form. Identifier und Konstante werden in eigenen Strukturen abgelegt (siehe Abb. 35).

Viele Modula-2 Compiler [01, 04] generieren aus den Definitionsmoduln sogenannte "Symbolfiles". Diese enthalten im Prinzip den überprüften Definitionsmodul und auch sämtliche von diesem importierte Objekte bzw. Moduln. Werden in weiterer Folge Objekte aus einem solchen Definitionsmodul bzw. der Modul selbst von einem anderen Modul importiert, so wird der Symbolfile dieses Definitionsmoduls eingelesen. Der Vorteil dieser Methode ist, daß nur mehr ein Symbolfile gelesen werden muß (auch wenn der Definitionsmodul Objekte aus anderen Definitionsmoduln importiert) und dieser nicht mehr auf seine Korrektheit überprüft werden muß. Der Nachteil ist, daß bei Änderungen in einem Definitionsmodul alle betroffenen Symbolfiles recompiliert werden müssen.

Der Modula-2-C Übersetzer erzeugt keine Symbolfiles, sondern es werden einfach die entsprechenden Definitionsmoduln gelesen. Dadurch ist allerdings auch immer wieder deren syntaktische und semantische Überprüfung notwendig.

Treten während der Syntaxanalyse Fehler auf, so werden diese in **M2Error** abgelegt.

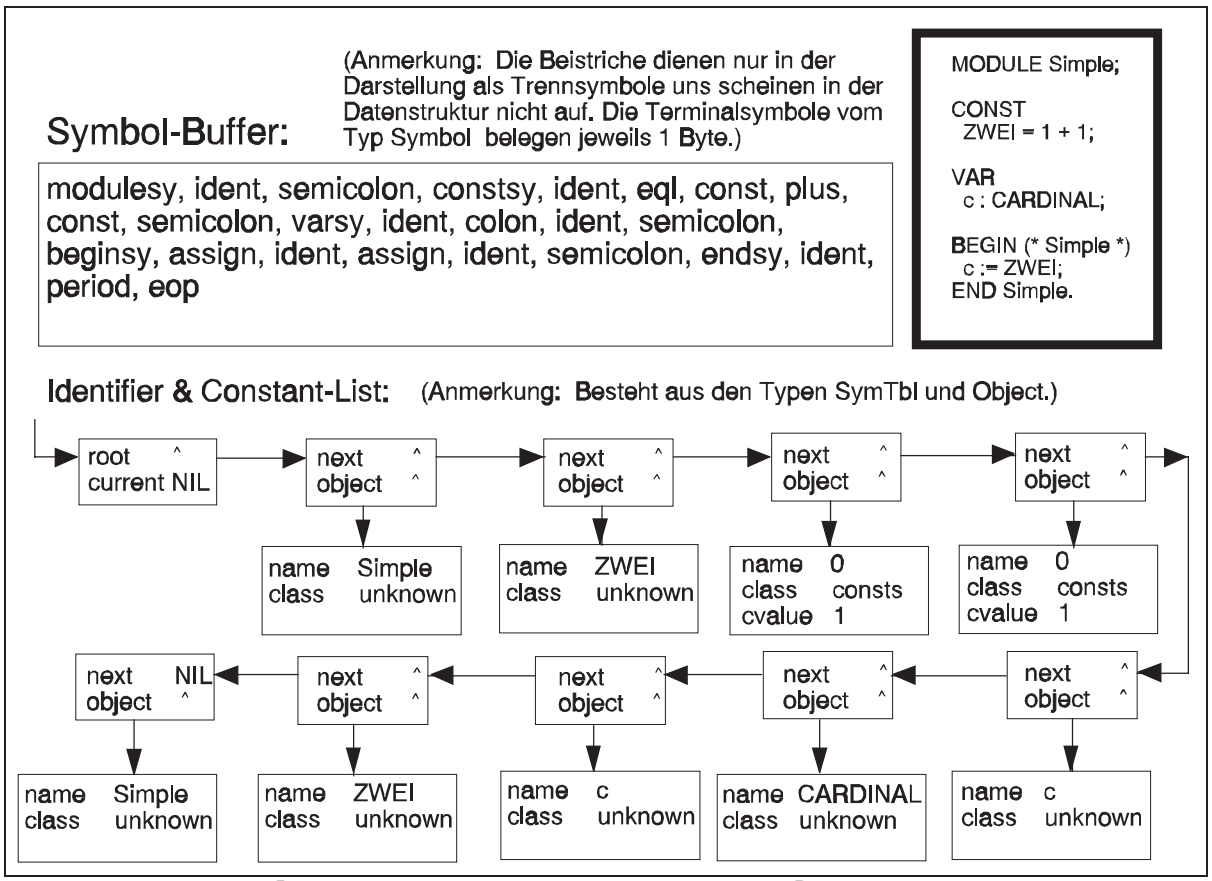

Abb. 35: Symbol-Buffer nach der Syntaxanalyse

Als Parsing Methode findet Recursive-Descent-Parsing [12] Verwendung, wobei diese Art der Strukturierung auch in den späteren Phasen (Deklarationsanalyse, Blockanalyse, Codegenerierung) verwendet wird. Somit ergibt sich in den einzelnen Komponenten ein einheitliches (syntaxorientiertes) Skelett, in dem sich nur die Aktionen (Analyse der Syntax, Aufbau der Symbol-Table, Semantische Analyse der Anweisungsteile, Generierung von C-Sourcecode) ändern, die durchzuführen sind.

Ein Vorteil der Beschränkung auf die reine Syntaxanalyse in Pass 1 ist, daß der Übersetzungsvorgang im Fall von Syntaxfehlern sofort abgebrochen werden kann und keine, in diesem Fall, überflüssigen Definitionsmoduln mehr eingelesen bzw. Datenstrukturen (Symbol-Table) aufgebaut werden. Des weiteren ist dadurch auch die Struktur der Moduln für Pass 1 einfacher und übersichtlicher.

3.4.3 Deklarationsanalyse

Die Hauptaufgaben der Deklarationsanalyse sind der Aufbau der Symbol-Table und das Prüfen und Auflösen der unterschiedlichen Referenzen (Typdeklarationen, Importe, Exporte) der Modula-2 Deklarationen (einschließliche aller Deklarationen in den eingelesenen Definitionsmoduln). Die Deklarationsanalyse bezieht dabei ihren Input aus dem Symbol-Buffer, wobei die Anweisungsteile (Definitionsmodulen verfügen über keine Anweisungen) von Moduln und Prozeduren überlesen werden und aus den Deklarationsteilen die Symbol-Table aufgebaut wird.

Die Deklarationsanalyse kann dabei davon ausgehen, daß der Inhalt des Symbol-Buffers syntaktisch korrekt ist. Wird der Anfang eines Anweisungsteils (über ein *BEGIN*-Symbol) erkannt, so können alle Symbole bis zum Ende des Anweisungsteils überlesen werden. Um das Ende eines Anweisungsteils zu erkennen wird von der Syntaxanalyse ein eigenes Block-Ende-Symbol im Symbol-Buffer abgelegt. Dann wird dort im Symbol-Buffer wieder "aufgesetzt", um (falls das Ende des Symbol-Buffers noch nicht erreicht wurde) den nächsten Deklarationsteil zu lesen. Verschachtelte Moduln bzw. Prozeduren können dadurch erkannt werden, daß sie in den Deklarationsteilen anderer Moduln oder Prozeduren beginnen müssen. Der Beginn der Anweisungsteile wird in der Symbol-Table als Index in den Symbol-Buffer abgelegt, um sie in der Blockanalyse direkt ansprechen zu können. Die Deklarationsanalyse gliedert sich hauptsächlich in folgende Moduln:

M2P2	Kontrollmodul der Deklarationsanalyse
M2P2Pars	Aufbau der Symbol-Table
M2P2Decl	Analyse der Deklarationen

Die Aufgaben der Deklarationsanalyse sind im einzelnen:

- Eintragen sämtlicher Deklarationen in die Symbol-Table

- Prüfen und Auflösen aller Exporte und Importe

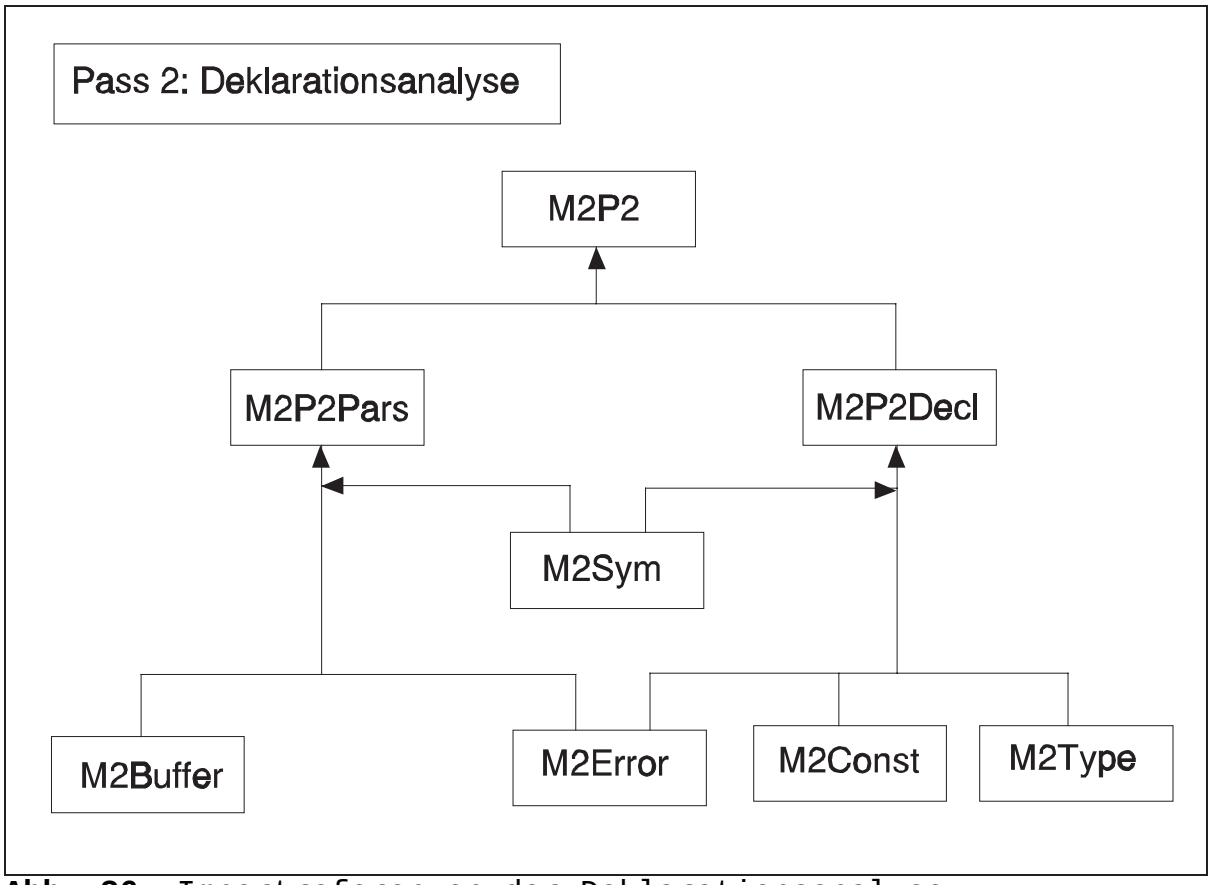

Abb. 36: Importreferenzen der Deklarationsanalyse

- Prüfen und Auflösen sämtlicher Typdeklarationen

- Prüfen von Deklartionen im Definitionsmodul und im zugehörigen Implementierungsmodul

- Auswertung von Konstantenausdrücken

Der Kontrollmodul **M2P2** (siehe Abb. 36) veranlaßt zuerst den Aufbau der Symbol-Table für die Übersetzungseinheit (Compilation-Unit) und sämtlicher Definitionsmoduln. Der für den Aufbau der Symbol-Table-Strukturen verantwortliche Modul **M2P2Pars** hat dabei wiederum eine ähnliche Struktur wie der Recursive-Descent-Parser für die Syntaxanalyse. Die einzelnen Symbole werden aus dem Symbol-Buffer (**M2Buffer**) gelesen und daraus wird dann die Symbol-Table (**M2Sym**) konstruiert. Danach veranlaßt **M2P2** die Überprüfung der zuvor aufgebauten Strukturen durch **M2P2Decl**.

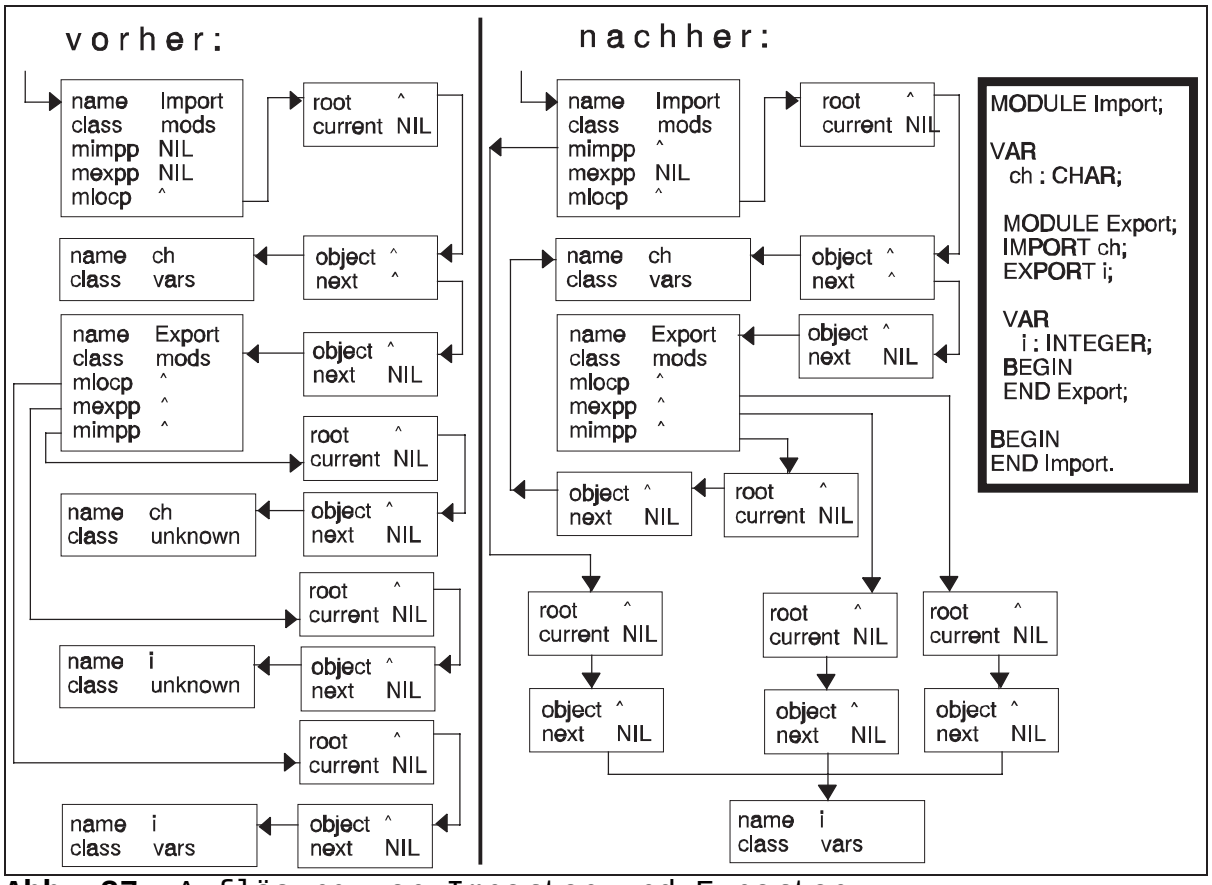

Abb. 37: Auflösung von Importen und Exporten

Dabei werden zuerst sämtliche Exporte und danach sämtliche Importe überprüft und aufgelöst. Wird ein Implementierungsmodul übersetzt, so ist es auch notwendig im dazugehörenden Definitionsmodul deklarierte Konstanten, Variablen und Typen (außer abstrakte Datentypen) zu analysieren und in den Implementationsmodul zu übernehmen. Danach enthält jeder Modul sämtliche Objekte die in ihm verwendet werden. Somit können die Typdeklarationen und Konstantenausdrücke in Deklarationen innerhalb der einzelnen Moduln überprüft und aufgelöst werden. Dafür verwendet **M2P2Decl** Funktionen von **M2Type** und **M2Const**. Handelt es sich um einen Implementierungsmodul, so muß auch die Übereinstimmung mit im Definitionsmodul deklarierten Prozeduren und abstrakten Datentypen überprüft werden.

3.4.4 Blockanalyse

Von der Blockanalyse werden die Anweisungsteile der Moduln und Prozeduren

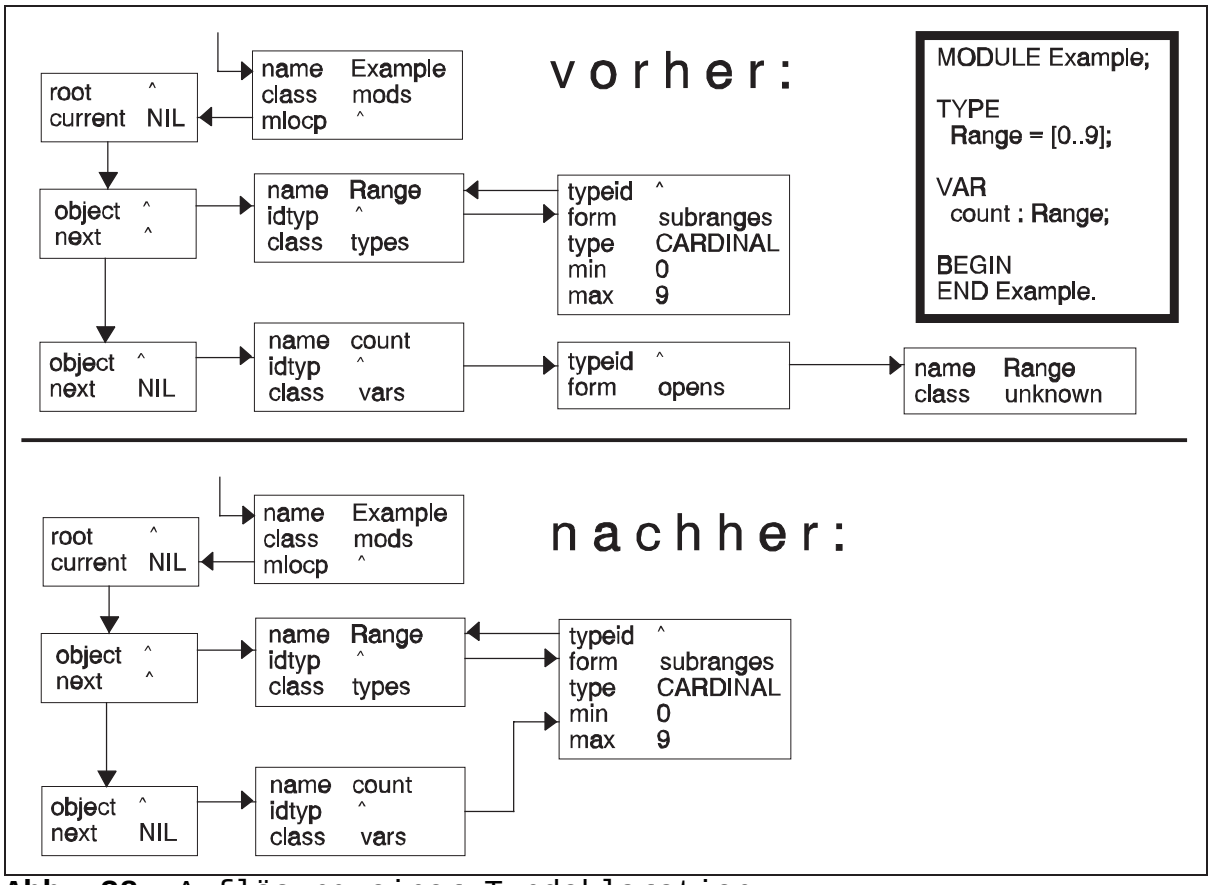

Abb. 38: Auflösung einer Typdeklaration

auf ihre Korrektheit untersucht. Die Hauptaufgabe ist dabei die Überprüfung der Typkompatibilität von Objekten in Ausdrücken, Zuweisungen und Prozeduraufrufen. Des weiteren ist zu überprüfen, ob sämtliche Objekte die in Anweisungen verwendet werden auch deklariert sind. Dafür werden sowohl Informationen aus der Symbol-Table (enthält die Deklarationen) als auch aus dem Symbol-Buffer (enthält die Anweisungen) herangezogen.

Die Blockanalyse gliedert sich im wesentlichen in folgende Moduln:

M2P3	Kontrollmodul der Blockanalyse
M2P3Pars	Analyse der Anweisungsteile

Der Modul **M2P3** (siehe Abb. 39) hat die Aufgabe die eigentliche Blockanalyse (**M2P3Pars**) aufzurufen. Die Symbol-Table (**M2Sym**) liefert dort die Referenzen in den Symbol-Buffer (**M2Buffer**), um die Anweisungsteile der Moduln und Prozeduren zu finden. Prozeduren von **M2Type** und **M2Const** werden verwendet,

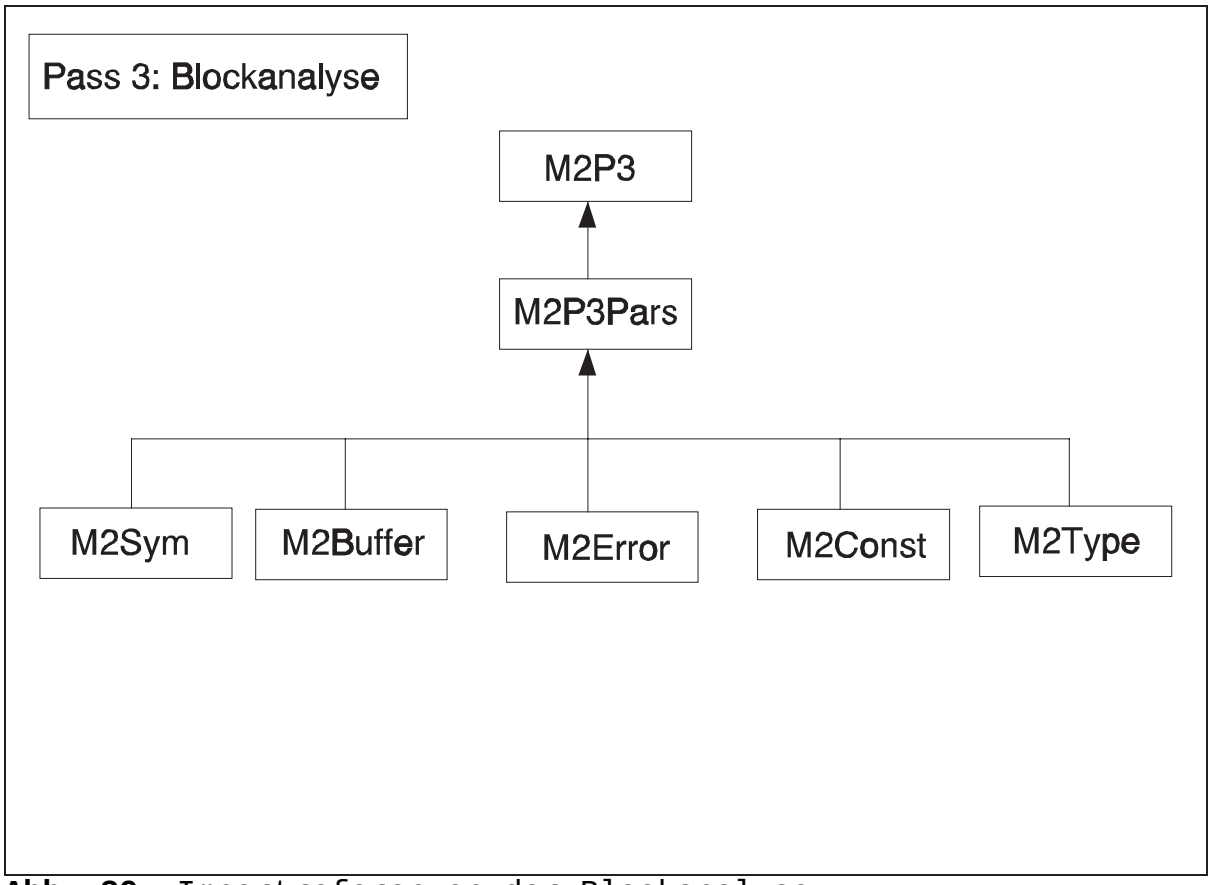

Abb. 39: Importreferenzen der Blockanalyse

um Anweisungen und Ausdrücke auf deren Korrektheit zu prüfen.

Jeder Identifier in einem Anweisungsteil wird in der Symbol-Table gesucht, um seine ordnungsgemäße Deklaration zu überprüfen und seinen Typ zu bestimmen. Auch für Konstante wird der jeweilige Typ bestimmt. Die Typinformationen werden dann verwendet, um die Typkompatibilität von Ausdrücken, Teilausdrücken (Ausdruckskompatibilität) und Zuweisungen (Zuweisungskompatibilität) zu überprüfen. Wird auf einen Ausdruck eine Typkonversion angewendet, so wird der Typ des Ausdrucks durch den Typ der Konvertierungsfunktion ersetzt. Bei Prozdeduraufrufen (auch von Standardprozeduren) wird die Typkompatibilität und die Übereinstimmung der Anzahl von formalen und aktuellen Parametern überprüft. Bei der Verwendung von Prozdurvariablen muß auch die Prozedurkompatibilität überprüft werden.

3.4.5 Codegenerierung

Die Codegenerierung erzeugt mit Hilfe der Symbol-Table und des Symbol-Buffers den gewünschten C-Sourcecode. Sie ist hauptsächlich in folgende Moduln gegliedert:

M2P4	Kontrollmodul der Codegenerierung
M2P4Decl	Generierung der Deklarationen
M2P4Body	Generierung der Anweisungen
M2P4Gen	Output von C-Sourcecode

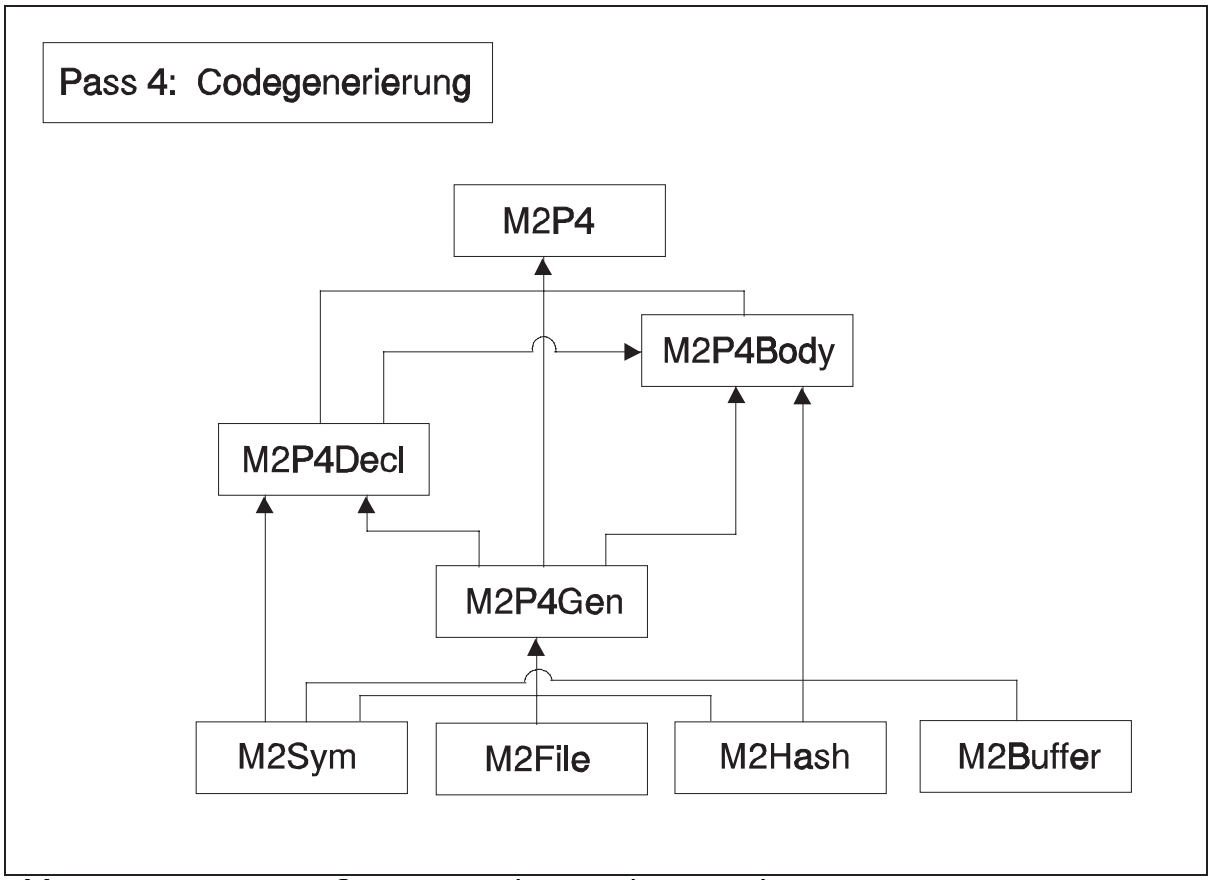

Abb. 40: Importreferenzen der Codegenerierung

Der Modul **M2P4** (siehe Abb. 40) hat die Aufgabe, die Generierung der Deklarationen und Anweisungen aufzurufen. Die Erzeugung der Deklarationen (durch **M2P4Decl**) stützt sich hauptsächlich auf die Symbol-Table (**M2Sym**) als Informationsquelle, während die Anweisungen (durch **M2P4Body**) aus dem Symbol-Buffer (**M2Buffer**) generiert werden. Sowohl **M2P4Decl** als auch

M2P4Body bedienen sich des Moduls **M2P4Gen** für die Emittierung von C-Sourcecode.

Die Symbol-Table wird zweimal traversiert. Zuerst werden sämtliche Deklarationen erzeugt. Dies umfaßt Objekte, die von anderen Moduln importiert werden, Objekte die lokal deklariert sind, sowie Vorausdeklarationen von Prozeduren und Initialisierungsteilen von Moduln (die Initialisierungsteile von Moduln werden auf Prozeduren abgebildet). Für jeden Modul wird weiters eine globale Variable erzeugt, die später verwendet wird, um Mehrfachinitialisierungen zu verhindern. Für Prozeduren, die verschachtelte Prozeduren aufweisen, wird die Umgebung strukturisiert und ein entsprechender "Displayzeiger" generiert. Bei Prozeduren, die keine verschachtelten Prozeduren besitzen, erfolgt keine Strukturisierung der Umgebung.

Beim zweiten Durchlauf durch die Symbol-Table werden die Zeiger in den Symbol-Buffer von Prozedur- und Modulobjekte verwendet, um die entsprechenden Anweisungsteile zu finden. Bei Moduln muß, vor der eigentlichen Initialisierungssequenz, Code generiert werden, der Mehrfachinitialisierungen verhindert (eine Ausnahme bilden Moduln in Prozeduren). Bei Prozeduren, die verschachtelte Prozeduren besitzen, müssen neben den Anweisungsteilen, entsprechende Eintritts- und Austrittssequenzen erzeugt werden (siehe Anhang C). Gleiches gilt für Prozeduren, die Felder als Wertparameter aufweisen (siehe Anhang B).

3.4.6 Fehlerbehandlung

Treten in einer der angeführten Phasen Fehler auf, so wird dieser Abschnitt (möglichst ordnungsgemäß) bis zu dessen Ende fortgesetzt und der Übersetzungsvorgang danach abgebrochen. Die Fehler werden in **M2Error** mit Fehlernummer und Fehlerposition (Zeile, Spalte im Source) gesammelt und es wird ein Fehlerlisting generiert, wobei dafür der Modula-2 Sourcecode nochmals gelesen wird. Um die Position eines Fehlers anzugeben, werden während der Syntaxanalyse Positionsangaben vom Scanner generiert, während die Fehlerbehandlung für

spätere Phasen solche Informationen aus der Symbol-Table bezieht (für jedes Objekt wird dessen Position im Source gespeichert). Die Fehlerbehandlung gliedert sich im wesentlichen in folgende Moduln:

M2Error	Sammeln von Fehlern
M2Lister	Generierung des Fehlerlistings

Der Modul **M2Error** verwaltet eine nach Fehlerpositionen sortierte Liste in die auftretende Fehler mit ihrer Fehlernummer und der Fehlerposition abgelegt werden. Diese Liste wird dann von dem Modul **M2Lister** für die Generierung des Fehlerlistings herangezogen. Dieses Listing besteht im Prinzip aus dem Modula-2 Sourcecode mit Zeilennummern, wobei jedoch nach Zeilen in denen Fehler gefunden wurden zusätzliche Zeilen eingefügt werden, die die Spaltenposition und eine kurze Beschreibung des Fehlers angeben.

3.5 Projektübersicht

Dieser Abschnitt gibt eine Übersicht über den Ablauf des Projektes. Der Modula-2-C Übersetzer wurde auf einem PC unter MS-DOS mit Hilfe eines Modula-2 Compilers entwickelt. Um schon bei der Entwicklung die Portabilität des erzeugten C-Codes soweit als möglich sicherzustellen, wurden weiters zwei C-Compiler verwendet. Einer davon diente allerdings nur für diese "Portabilitätskontrolle", während der andere tatsächlich für die Erstellung einer ausführbaren Übersetzerversion verwendet wurde.

Im folgenden wird zuerst der zeitliche Ablauf des Projektes geschildert, wobei hier zu berücksichtigen ist, daß der Autor allein den Entwurf und die Implementierung durchführte und "nebenbei" auch noch anderen Verpflichtungen nachzukommen hatte. Es wurde somit nicht ständig am Projekt gearbeitet und es ergaben sich sogar längere Unterbrechungen, während denen die Implementierungsarbeiten ruhten.

Ein wichtiger Meilenstein des Projektes war die Selbstübersetzung des Modula-2-C Übersetzers, die in einem der folgenden Abschnitte detailliert beschrieben wird. Danach kam es zu einer Reihe von "Portierungstests", um die Portabilität des Übersetzers so früh wie möglich zu Überprüfen. Weiters werden Probleme und Komplikationen die sich im Verlauf des Projektes ergaben erörtert und am Schluß diese Abschnitts werden Restriktionen und geplante Erweiterungen des Übersetzers diskutiert.

3.5.1 Projektverlauf

Dieser Abschnitt gibt einen Überblick über den zeitlichen Verlauf des Projektes. Über die Entwurfsphase existieren keine detaillierten Zeitaufzeichnungen. Die ersten Entwurfsideen entstanden, als an der Änderung der Codegenerierung eines Modula-2 Compilers gearbeitet wurde, um nicht mehr Maschinencode sondern C-Sourcecode zu erzeugen. Es ging dabei um einen Modula-2 Compiler, der Maschinencode für die Motorola 680x0 Familie erzeugt [04] und unter UNIX implemen-

tiert ist. Allerdings gelang es bei diesem Projekt nicht einen einfach zu portierenden Modula-2-C Transformator [05] zu erzeugen, der auch in der Lage ist sich selbst zu übersetzen.

Vor allem der Entwurf der Codegenerierung hat von den Erfahrungen, die in dem vorangegangenen Projekt gewonnen wurden, profitiert. Der Entwurf für den, in dieser Arbeit beschriebenen, Modula-2-C Übersetzer entstand in der Zeit von Ende Juni 1990 bis Mitte Juli.

1990 - 07: Start der Implementierung (1990-07-15). Festlegen der wesentlichsten Definitionsmoduln.

Implementierung der Hash-Table und des String-Buffers.

Implementierung des Hauptmoduls und (soweit möglich) von M2Comp. Implementierung der Kontrollmoduln (M2Init, M2P1, M2P2, M2P3, M2P4) der einzelnen Übersetzerphasen (Pass 2 bis 4 nur als Skelett). Implementierung des (vorerst primitiven) Error-Handlings (M2Error).

Implementierung der Dialogkomponente und des Scanners für Pass 1. Implementierung des Listers (M2Lister).

Test und Debugging des Scanners.

Implementierung des Symbol-Buffers (M2Buffer). Implementierung des Parsers (M2P1Pars). Verbesserung des Error-Handlings.

Test und Debugging der Syntaxanalyse.

1990 - 08: Implementierung des Einlesens von Definitionsmoduln. Verbesserungen des Error-Handlings und der Dialogkomponente.

Beginn der Implementierung von Pass 2 (zuerst M2P2Pars).

Implementierung von M2Debug, für das Debugging der vom Übersetzer erzeugten Strukturen (Symbol-Table). Integration von M2P2Pars in den Übersetzer. Test und Debugging von M2P2Pars.

Implementierung von M2Type (Hilfsfunktionen für das Type-Checking). Implementierung von M2Const für die Auswertung von Ausdrücken mit Konstanten.

Test der Konstantenauswertung. Implementierung von M2P2Decl.

Test und Debugging der Deklarationsanalyse (vor allem M2P2Decl).

Beginn Implementierung der Blockanalyse (M2P3Pars).

1990 - 09: Abschluß Implementierung der Blockanalyse (Test und Debugging von M3P3Pars). Beginn der Implementierung der Codegenerierung (M2P4Gen).

Implementierung der Generierung von Deklarationen (M2P4Decl, M3P4Body - vorerst leere Implementierung)

Test und Debugging von M2P4Decl. Implementierung von M2P4Body.

Von 1990-09-10 bis 1990-10-10 Unterbrechung des Projektes.

1990 - 10: Implementierung des Laufzeitsystems. Verbesserungen in M2P4Gen.

Erste (noch nicht ernst gemeinte) Versuche der Selbstübersetzung des Übersetzers. Debugging. Debugging. Debugging.

Implementierung der Modula-2 Library. Test der Library.

1990 - 11: Forsetzung des Debuggings des Übersetzers und der Library.

Wechsel des C-Compilers, da es, wegen der Programmgröße, mit der Entwicklungsumgebung Schwierigkeiten gab. Fortsetzung der Implementierung der Modula-2 Library.

Erste (ernst gemeinte) Versuche den Compiler zu übersetzen (einige Library Implementation-Moduln fehlen noch). Vervollständigung der Library.

Erstes Übersetzen und Linken des Compilers (1990-11-14) mit dem C-Compiler (Anmerkung: Der C-Output wird dabei von dem mit dem Modula-2 Compiler erzeugten Übersetzer generiert). Debugging. Anpassungen in der Library. Speicherprobleme bei der Übersetzung großer Moduln (Heap overflow).

Implementierung von Variantenrecords mit unions. Debugging. Verbesserung des Laufzeitsystems.

1990 - 12: Probleme mit Stack overflows. Zurückwechseln zum ursprünglichen C-Compiler. Speicherpobleme (Heap overflow). Umschreiben des Übersetzers für die Nutzung von EMS-Memory für die Symbol-Table.

Übersetzer hat sich selbst übersetzt (1990-12-06).

Test und Debugging des Übersetzers. Optimierungen in der Codegenerierung.

1991 - 01: Test des Übersetzers. Abschluß von Phase 1 (Meilenstein war die Selbstübersetzung) der Entwicklung und Übergang zu Phase 2.

Umschreiben der Symbol-Table-Verwaltung (M2Sym) des Übersetzers, um ohne die Nutzung vom EMS-Memory auszukommen. Test des Übersetzers.

Erster Portierungstest des Übersetzers auf UNIX (CISC Architektur

mit Motorola 68020). Änderungen in der Codegenerierung, um portableren (der UNIX-C-Compiler ist relativ empfindlich) Code zu erzeugen.

Implementierung von Moduln in Prozeduren.

Von 1991-01-24 bis 1991-04-18 Unterbrechung des Projektes.

1991 - 04: Änderungen in der Library. Änderung der Implementierung der Konfigurationsdatei (M2Config) um die Anpassung des Übersetzers an unterschiedliche Umgebungen zu erleichtern.

Testen und verbessern des Übersetzers.

1991 - 05: Testen und verbessern des Übersetzers.

Weiterer Portierungstest auf UNIX (RISC-Architektur).

Portierungstest auf VMS (Micro-VAX).

3.5.2 Die Selbstübersetzung

Die Entwicklung des Übersetzers wird mit Hilfe sogenannter T-Diagramme erläutert. Abb. 41 gibt einige Erläuterungen über die Bedeutung von T-Diagrammen. Weitere Details über die Verwendung und Notation von T-Diagrammen finden sich in [20].

T-Diagramme sind eine hilfreiche Notation, um die Entwicklung von Compilern zu illustrieren, haben aber den Nachteil, daß man vor allem bei Selbstübersetzungen leicht den Überblick verliert. Diese Problematik wird dadurch verursacht, daß oft Boxen mit gleichem Inhalt entstehen, allerdings trotzdem unterschiedliche Compiler gemeint sind. Um hier eine Unterscheidung

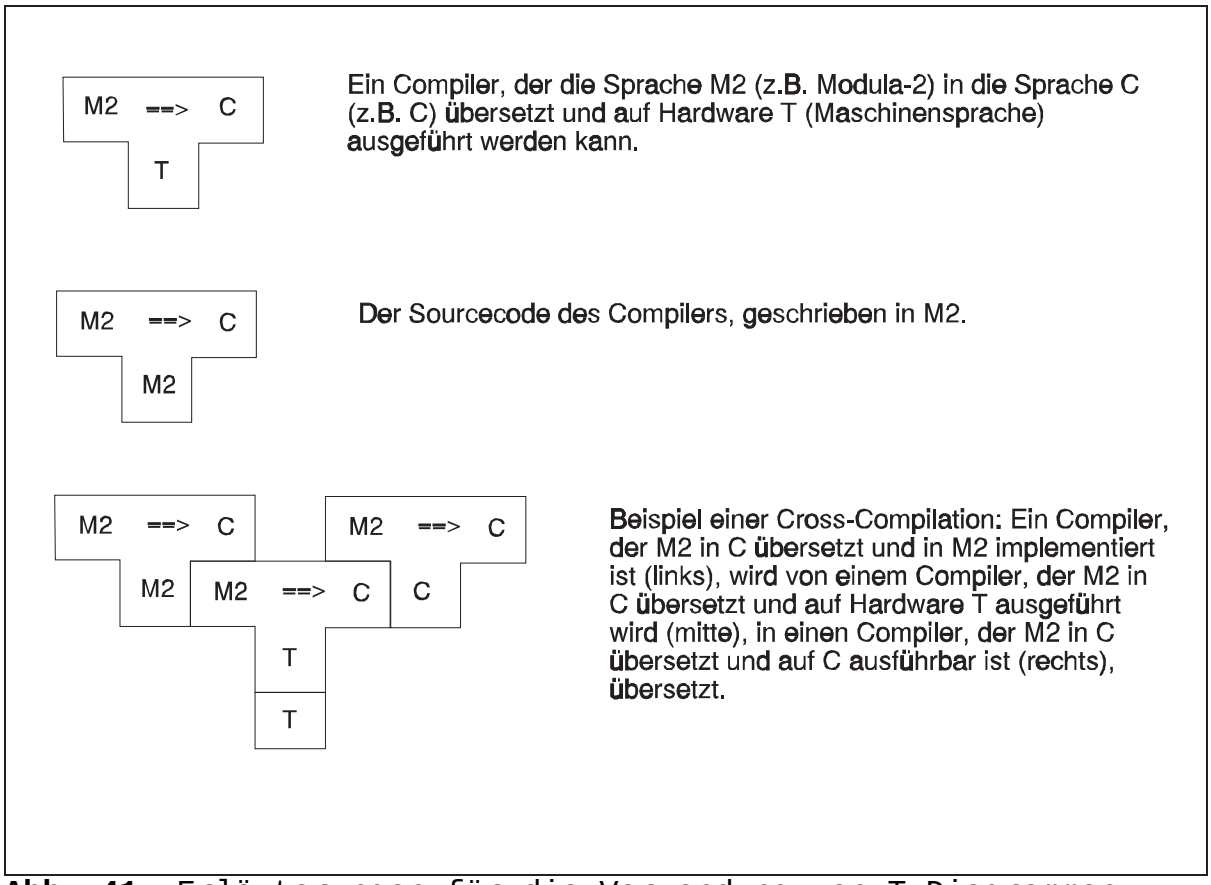

Abb. 41: Erläuterungen für die Verwendung von T-Diagrammen

zu ermöglichen, werden in den folgenden Abbildungen geklammerte Nummern über die Boxen geschrieben.

Die wichtigsten Entwicklungsschritte des Modula-2-C Übersetzers werden in Abb. 42 dargestellt.

Die Entwicklung (siehe Abb. 42) beginnt mit dem (fertigen) Compiler (0)[3]. Es handelt sich dabei um einen fehlerfreien und auf Hardware T ablauffähigen Modula-2 Compiler, der Modula-2 Input in den entsprechenden Output in Maschinensprache für T umwandelt.

Mit diesem Compiler wurde der Modula-2-C Übersetzer (1) in Modula-2 implementiert. Dies sagt sich relativ einfach, ist jedoch die eigentliche Arbeit. Der

[3] Der Übersetzer wurde mit dem LOGITECH Modula-2/86 Compiler entwickelt.

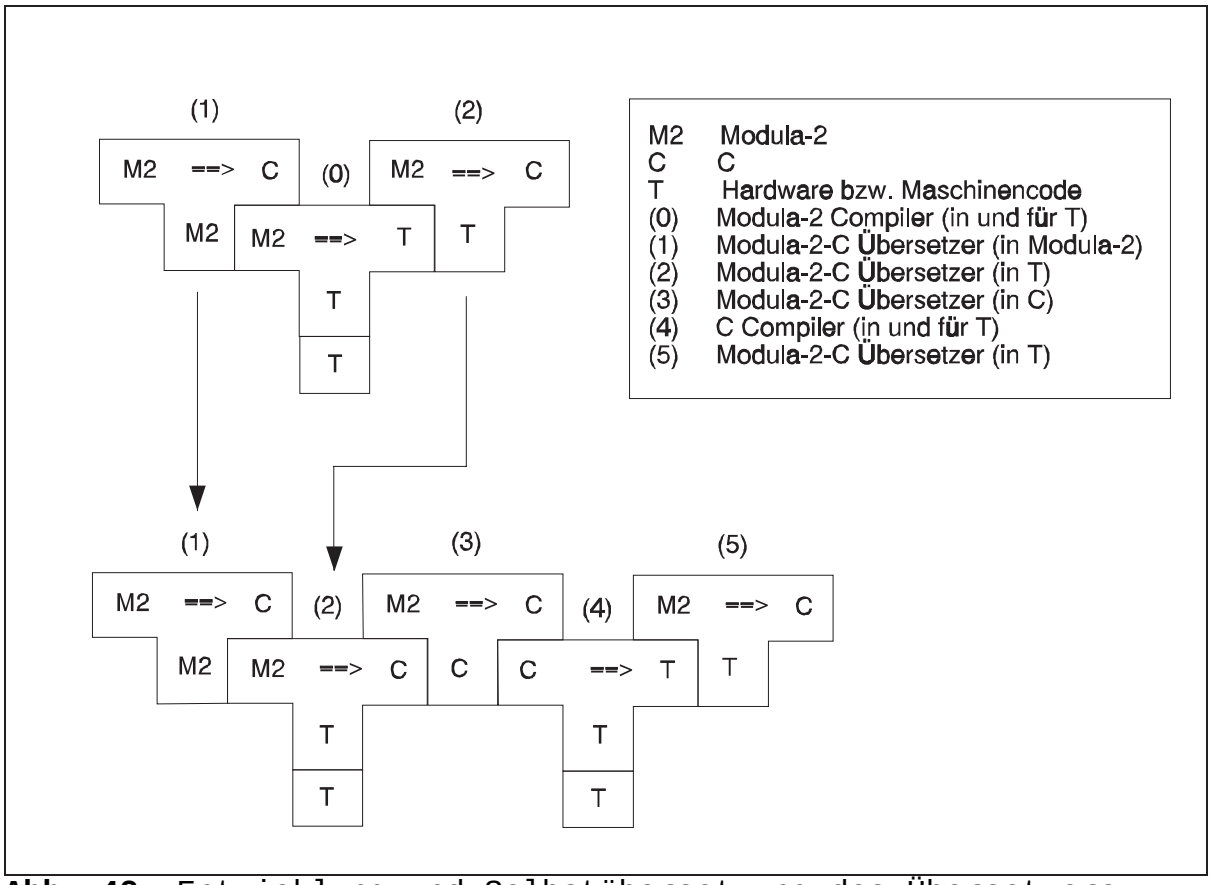

Abb. 42: Entwicklung und Selbstübersetzung des Übersetzers

Modula-2-C Übersetzer (1) wird mit dem Modula-2 Compiler (0) übersetzt und man erhält einen Modula-2-C Übersetzer (2), der auf Hardware T ausführbar ist.

Dies ist zwar schon recht schön, genügt aber noch nicht unseren Ansprüchen. Ein wesentlicher Nachteil ist, daß der Übersetzer (2) vollständig von der Bibliothek von (0) abhängig ist. Die Bibliothek kann allerdings jetzt mit Hilfe von (2) neu implementiert werden, wobei darauf zu achten ist, daß sich die Schnittstellen nicht ändern. Dies hat den Vorteil, daß der Übersetzer (1) nach wie vor von Compiler (0) übersetzt werden kann. In den Implementierungsmoduln der Bibliothek darf hingegen alles verwendet werden, was der Modula-2-C Übersetzer (1) dem Modula-2 Compiler (0) im Hinblick auf Modula-2 voraus hat (z.B. *FOREIGN* Definitionsmoduln, neue Typen, usw.). Was die Bibliothek betrifft, findet hier im Prinzip ein Bootstrap statt, wobei eventuelle Vorteile der neuen Bibliothek (z.B. effizientere Implementierung) beim nächsten Übersetzungsschritt zum tragen kommen.

Man nimmt nun den Modula-2-C Übersetzer (1) und übersetzt ihn mit sich selbst (2). Dadurch erhält man (ohne große Arbeit) einen Modula-2-C Übersetzer (3), der in C implementiert ist. Man benötigt nun einen C-Compiler (4), um den Übersetzer (3) wieder in eine auf Hardware T ausführbare Version (5) zu übertragen. Man beachte, daß der Übersetzer (2) NICHT mit dem Übersetzer (5) identisch ist. Der Übersetzer (2) wurde mit dem Modula-2 Compiler (0) übersetzt und baut außerdem vollständig auf dessen Bibliothek auf, während der Übersetzer (5) mit Hilfe eines C-Compilers (4) erzeugt wurde und seine eigene Bibliothek verwendet. Diese setzt beim Übersetzer (5) nicht auf der Hardware oder beim Betriebssystem auf, sondern beruht im Prinzip (über *FOREIGN* Definitionsmoduln) auf der Bibliothek des C-Compilers. Dies kann man auch bei den folgenden Übersetzertests feststellen (siehe Abb. 43).

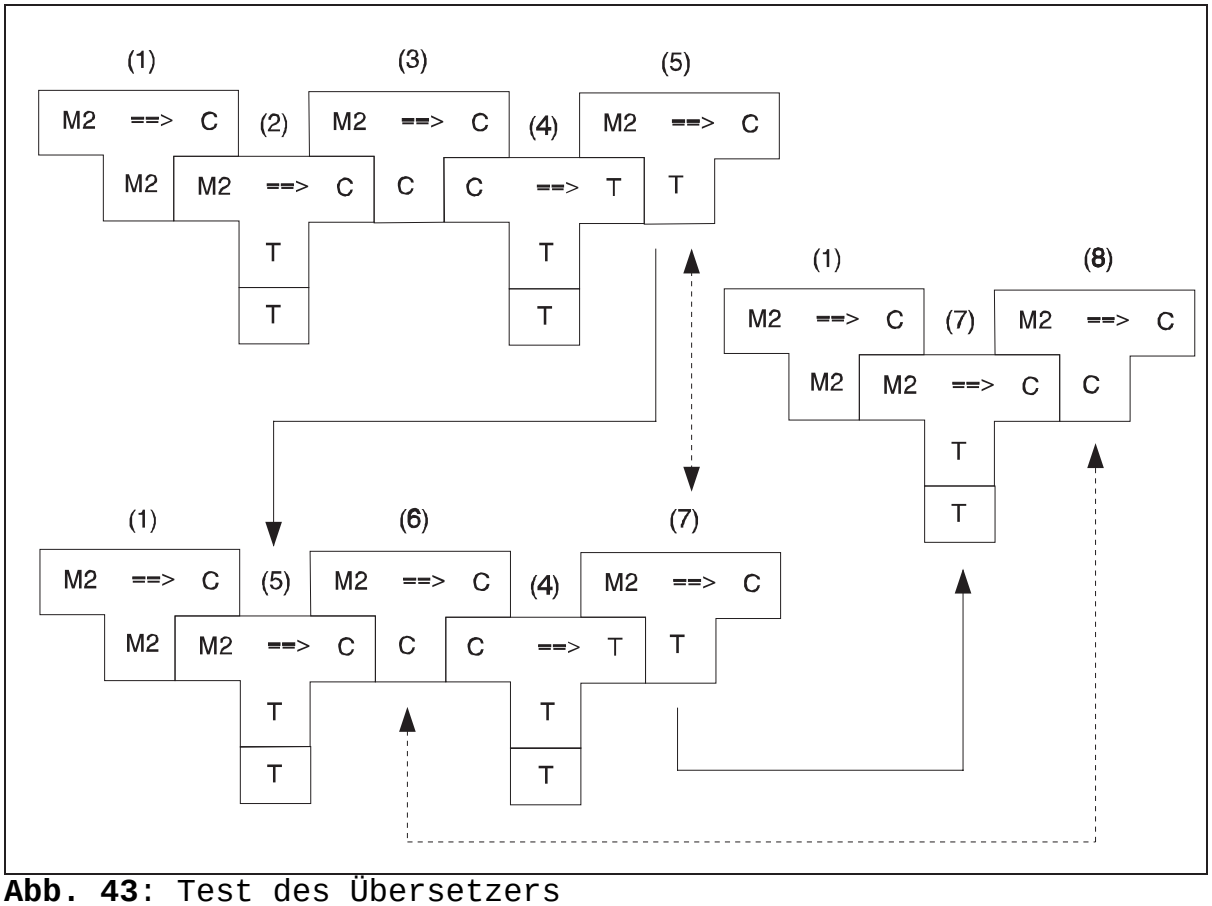

Abb. 43: Test des Übersetzers

Man nimmt nun erneut den Modula-2-C Übersetzer (1) der in Modula-2 implementiert ist, und übersetzt ihn mit dem vorher gewonnenen Übersetzer (5). Man erhält dadurch wieder einen Modula-2-Übersetzer (6), in Form von C-Sourcecode. Vergleicht man nun den Übersetzer (3) mit dem Übersetzer (6)

Zeichen für Zeichen (beide liegen als C-Sourcecode vor), so können sich Unterschiede ergeben. Diese liegen in den unterschiedlichen Bibliotheken begründet. So kann etwa die Darstellung von Floating-Point-Konstanten unterschiedlich sein, obwohl derselbe Wert dargestellt wird. Es sei darauf hingewiesen, daß die Semantik der Übersetzer (3) und (6) dieselbe sein muß.

Dies kann auf folgende Weise festgestellt werden: Man nimmt erneut den C-Compiler (4) und übersetzt damit den Modula-2-C Übersetzer (6) und erhält erneut eine auf Hardware T ausführbare Übersetzerversion (7). Geht man davon aus, daß der C-Compiler (4) korrekt ist, so müssen die Übersetzer (5) und (7) identisch sein. Beide bauen nun auch auf dieselbe Bibliothek auf.

Will man C-Sourcecode vergleichen, so kann man noch einen Übersetzungsschritt anhängen. Man übersetzt nochmals den Übersetzer (1) und zwar mit dem zuvor gewonnen Übersetzer (7) und erhält den C-Sourcecode von Übersetzer (8). Da bereits (5) und (7) identisch sind, müssen nun auch Übersetzer (6) und (8) zeichenweise identisch sein.

3.5.3 Die Portierung des Übersetzers

Ein Ziel des Projektes war die einfache Portierbarkeit des Übersetzers. Einerseits sollte der in Modula-2 vorliegende Übersetzer portabel sein und andererseits ebenso der durch die Selbstübersetzung gewonnene C-Sourcecode des Übersetzers.

Man stößt in diesem Zusammenhang auf die (hier etwas unangenehme) Frage: Was ist eigentlich "Portabilität" ? Auf diese Frage gibt es wohl mehrere Antworten. Im Folgenden, soll kurz dargestellt werden was, im Zusammenhang mit diesem Projekt der Implementierung eines Modula-2-C Übersetzers, unter dem Begriff "Portabilität" verstanden wird.

3.5.3.1 Erörterung des Portabilitätsbegriffs

Betrachten wird zuerst das Ideal: Die Sprachen Modula-2 und C sind weitgehend standardisiert und es sollte dadurch möglich sein, den Sourcecode von Applikationen von einer Entwicklungsumgebung in eine andere Modula-2 bzw. C Umgebung zu übertragen. Unter Entwicklungsumgebung wird hier einfach eine entsprechende Hardware- und Software- (Compiler, Library, Betriebssystem) Kombination verstanden. Dort wird die Applikation (ohne Änderungen im Sourcecode) übersetzt und sollte dann, bei Ausführung in der neuen Umgebung, die gleichen Leistungsmerkmale aufweisen, wie in der alten Umgebung.

Der Modula-2-C Übersetzer ist nun eine solche Applikation, allerdings ist man in der Praxis mit einigen Schwierigkeiten (die folgende Beschreibung erhebt keinen Anspruch auf Vollständigkeit) konfrontiert. Diese betreffen einerseits die Sprachstandards und andererseits die Verschiedenheit der Entwicklungsumgebungen.

Bei beiden Sprachen wurden Anstrengungen für die Standardisierung unternommen, allerdings gab es hier auch Änderungen und (dadurch) teilweise auch (in C mehr als in Modula-2) gewisse Graubereiche. Schlimmer ist allerdings die Situation bei den Entwicklungsumgebungen. Hier gibt es Compiler, die sich nicht an die Sprachstandards halten, sowie Unterschiede in den jeweiligen Bibliotheken. Höhere Programmiersprachen haben zwar die Aufgabe, die jeweilige Umgebung weitgehend transparent zu machen, allerdings gibt es meist trotzdem gewisse Abhängigkeiten (z.B. Wortgrößen, Zeichensätze). Aus dieser Sicht scheint es also (die oben gemeinte) "Portabilität" nicht zu geben.

Man hat also keine andere Wahl, als gewisse Kompromisse zu schließen (z.B. auch Änderungen im Sourcecode erlauben), um eben das Beste aus der Situation zu machen. Es wurden daher bei der Entwicklung des Übersetzers gewisse Vorkehrungen für spätere Portierungen getroffen. Dies betrifft einerseits die Modula-2 Version des Übersetzers und andererseits die Codegenerierung und die Bibliothek.

Bei der Implementierung des Übersetzers wurde darauf geachtet, daß keine

maschinennahen Sprachelemente verwendet werden. Eine Ausnahme stellen hier einige (wenige) Typkonvertierungen dar, die allerdings zwischen Typen ablaufen, deren Konvertierung, nach Meinung des Autors, auf jedem Rechner möglich sein sollten (z.B. zwischen *INTEGER* und *CARDINAL*). Für die Variablen im Übersetzer gilt, daß davon ausgegangen wird, daß mindestens 16 bit für die Darstellung von Werten verwendet werden. Der Übersetzer kann natürlich auch ausgeführt werden, wenn stattdessen 32 bit verwendet werden. Um Fehler bei der Auswertung von Konstanten zu vermeiden, müssen die erlaubten Wertebereiche der einzelnen Typen, für die Konstante deklariert werden dürfen, bekannt sein. Solche, ebenfalls von der jeweiligen Hardware und in weiterer Folge vom C-Compiler abhängigen Dinge wurden in einem Modul konzentriert, um eventuell notwendige Änderungen so weit wie möglich zu beschränken.

3.5.3.2 Die Portabilität der Bibliotheken

Für Modula-2 Bibliotheken hat sich ein gewisser (Pseudo-) Standard herausgebildet und es wurde versucht, diesen einzuhalten. Hier liegt meiner Ansicht nach, was die Modula-2 Portabilität betrifft, das größte Problem. Wird der Übersetzer von einem Modula-2 Compiler übersetzt, so verwendet er dessen Bibliothek und nicht seine eigene, da diese als Schnittstelle zu C dient. Stimmen die Bibliotheken nicht überein, so wird man es natürlich vermeiden, im Modula-2-C Übersetzer oder in dessen Bibliothek Änderungen vorzunehmen, sondern die Library des jeweiligen Modula-2 Compilers erweitern bzw. anpassen. Problematisch sind hier vor allem Funktionen wie File I/O, Terminal I/O und Konvertierungen, die allerdings, wenn auch möglicherweise in etwas unterschiedlicher Form, in jeder Modula-2 Bibliothek vorhanden sein sollten. Weiters wurde durch die Einführung einer Konfigurationsdatei versucht, die Anpassung an verschiedene Betriebssystemoberflächen (betrifft hauptsächlich das File-System) zu erleichtern.

Für C gilt im Prinzip dieselbe Problematik (vor allem auch das Problem der verschiedenen Bibliotheken) wie für Modula-2. Für die Übersetzung von Modula-2 auf C kommt weiters hinzu, daß die Modula-2 Typen entsprechend auf C abgebildet werden müssen. Auch hier wurde versucht, durch Einhaltung von

(Pseudo-) Standards (betrifft hautptsächlich wieder die Library) dem Problem die Schärfe zu nehmen. Weiters wurden kritische Funktionen auf wenige C-Moduln konzentriert (z.B. die Laufzeitunterstützung, C-Moduln für Terminal I/O und Typkonvertierungen). Die Abbildung der Modula-2 Typen kann auf sehr einfache Weise geändert werden (siehe Anhang G über die Laufzeitunterstützung).

3.5.3.3 Portierung auf unterschiedliche Systeme

Um zu überprüfen, wie sich die oben beschriebenen Maßnahmen in der Praxis bewähren, wurden einige "Portierungstests" durchgeführt. Diese Portierungen haben insofern Testcharakter, als keine "optimalen" Anpassungen an die neuen Umgebungen durchgeführt wurden. Insbesondere wurden keine Änderungen der Abbildungen der Standardtypen, sowie nur minimale Änderungen in der Bibliothek vorgenommen. Der Testablauf war dabei der folgende:

- Kopieren der Modula-2 Sourcen und C-Sourcen in die neue Umgebung.

- Übersetzen der C-Sourcen mit dem C-Compiler der neuen Umgebung. Linken zu einer ausführbaren Version.

- Übersetzen der Modula-2 Sourcen mit dem Modula-2-C Übersetzer. Vergleichen der C-Sourcen (diese müssen nicht zeichenweise identisch sein, allerdings muß die Semantik gleich sein).

- Ein Test galt als erfolgreich, wenn sich der Übersetzer in der neuen Umgebung selbst übersetzen konnte.

Der erste Portierungstest war der Schritt von MS-DOS auf UNIX (CISC-Architektur mit einem Motorola 68020). Die allerersten Versuche scheiterten, da der C-Sourcecode nicht vom C-Compiler unter UNIX übersetzt werden konnte. Dies lag daran, daß die C-Compiler auf dem PC und jener unter UNIX etwas unterschiedliche Auffassungen über "korrektes" C hatten. Der UNIX C-Compiler war dabei "etwas genauer" als seine PC-Kollegen. Die Meinungsverschiedenheiten der

C-Compiler konnten durch Änderungen in der Codegenerierung beseitigt werden. Dies hatte keine Auswirkungen auf die PC-Version des Übersetzers (es wurde im Gegenteil "korrekteres" C generiert).

Nach diesen Änderungen konnte der Übersetzer problemlos unter UNIX übersetzt werden. Danach waren kleinere Änderung in der Modula-2 Bibliothek notwendig, da z.B. Terminal I/O unter UNIX etwas anders ist, als auf einem PC (z.B. bezüglich "echo" von Zeichen, end-of-line Zeichen, ...). Weiters waren kleine Änderungen im Laufzeitsystem notwendig (z.B. Änderung von *#include* Anweisungen). In weiterer Folge gelang die Selbstübersetzung. Als besonders nützlich erwies sich bei der Portierung (und allen folgenden) die Konfigurationsdatei, die eine Anpassung an die geänderte Syntax der Pfadangaben und die Namenslänge von Dateien in neuen Umgebungen erleichtert. Die Hauptarbeit bei der Übertragung bestand hauptsächlich im Kopieren der Sourcen und im Erstellen von Kommandodateien für die Übersetzung und das Linken des Übersetzers.

Der nächste Portierungstest war von UNIX (siehe oben) auf UNIX (RISC-Architektur). Es wurde im Prinzip einfach die gesamte Übersetzerumgebung vom ersten UNIX-System auf das zweite kopiert (sämtliche Kommandodateien konnten natürlich weiterverwendet werden). Die Übersetzung der C-Sourcen und in weiterer Folge die Selbstübersetzung gelang sofort und ohne Probleme.

Der bislang letzte Portierungstest war (kurz vor dem Abschluß dieser Arbeit) von MS-DOS auf VMS (Micro-VAX). Nicht unerwartet ergaben sich wieder Probleme mit dem Terminal I/O (siehe oben) und mit den Wertebereichen von floating-point Konstanten (der Übersetzer kennt die maximale und die minimale floating-point Konstante). Weiters mußte auch wieder eine Änderung der *#include* Dateien im Laufzeitsystem vorgenommen werden. In weiterer Folge gelang dann die Selbstübersetzung der Modula-2 Sourcen. Allerdings konnte der erzeugte C-Output teilweise (betraf nur einige C-Moduln) nicht mehr vom VAX-C-Compiler gelesen werden. Wie sich herausstellte, lag dies daran, wie UNIX Stream I/O auf das VMS-RMS abgebildet wird. Ein UNIX Stream wird dabei nach bestimmten Kriterien (z.B. end-of-line Zeichen) in sogenannte "logische Records" eingeteilt. Diese dürfen eine bestimmte Länge nicht überschreiten und werden in weiterer

Folge auf RMS-Records abgebildet. Werden die vorgegebenen Längenbeschränkungen nicht eingehalten, so ist die erzeugte Datei nicht mehr lesbar. Das Problem wurde einfach dadurch gelöst, daß während der Codegenerierung allzu lange Zeilen (diese können sich durch komplizierte Ausdrücke ergeben) abgeteilt und auf mehrere Zeilen verteilt werden.

3.5.4 Probleme und Komplikationen

Während der Entwicklung des Übersetzers ergaben sich naturgemäß einige Probleme und Komplikationen. Die, nach Meinung des Autors, wesentlichsten davon werden in diesem Abschnitt kurz behandelt.

3.5.4.1 Speicherprobleme

Das größte Problem während der Entwicklung des Modula-2-C Übersetzers waren die Speicherbeschränkungen der PC-Entwicklungsumgebung unter MS-DOS. Nachdem der Übersetzer mit dem C-Compiler übersetzt worden war, ergaben sich bei der Selbstübersetzung Probleme (heap-overflow) mit großen Modula-2 Moduln. In dieser Phase erwies sich die Implementierung der Symbol-Table durch einen abstrakten Datentyp als Segen. Die Symbol-Table konnte teilweise in den sogenannten EMS-Memory ausgelagert werden. Es handelt es sich dabei im Prinzip um eine einfache Speichererweiterung, wobei Speicherseiten "händisch" (d.h. explizit) vom Hauptspeicher in den erweiterten Speicher ausgelagert werden können und umgekehrt. Nach dieser Umstellung der Symbol-Table-Implementierung gelang die Selbstübersetzung.

Wie sich später herausstellte wurde das Problem durch eine Modula-2 Spracheigenschaft verursacht, die nicht komplett auf C abgebildet worden war. In Modula-2 ist es möglich bei der dynamischen Speicherallozierung von Variantenrecords nur für bestimmte Varianten Speicher zu reservieren. Also z.B. durch:

NEW(RecordPointer, Schalterkomponente_1, Schalterkomponente_2, ...)

Eine solche Anweisung wird derzeit einfach auf *NEW*(RecordPointer) abgebildet, womit immer für die größte Variante Speicher alloziert wird. Dies führte bei der Symbol-Table (nach der Abbildung auf C) dazu, daß zu viel Speicher verbraucht wurde. Nachdem die Ursache für die Speicherproblem bekannt war, wurde die Implementierung der Symbol-Table erneut geändert, um die Abhängigkeit vom EMS-Memory los zu werden. Die derzeitige Implementierung entspricht zwar semantisch nach wie vor der in dieser Arbeit vorgestellten, allerdings wurden die großen Varianten des Datentyps *Object* (siehe Abb. 28) auf einzelne Records verteilt. In *Object* wird dann nur mehr ein Pointer auf den jeweils benötigten Record abgelegt, womit bezüglich Speicherallozierung im Prinzip derselbe Effekt erzielt wird, wie in Modula-2 (es wird nur mehr für die wirklich benötigte Variante Speicher alloziert).

3.5.4.2 Bootstrap Probleme

Der Übersetzer bietet gegenüber dem Modula-2 Standardtypen einige Typerweiterungen (*SHORTCARD*, *LONGCARD*, *SHORTINT*, *LONGINT*, *SHORTREAL*, *LONGREAL*). Diese dienen derzeit vor allem dazu, die Verbindung zu C in der Bibliothek zu erleichtern. Im Übersetzer selbst finden diese Typen keine Verwendung. Die Einführung dieser Typen in den Übersetzer würde eine klassische Bootstrap Situation darstellen, jedoch würde dies bedeuten, daß der Übersetzer von anderen Modula-2 Compilern nicht mehr übersetzt werden kann. Es ist daher derzeit nicht möglich, Konstante der oben genannten Typen in Modula-2 Applikationen anzugeben, da diese im Übersetzer intern nicht (korrekt) dargestellt werden können.

Das Problem könnte allerdings dadurch gelöst werden, daß sich der Übersetzer bei der Selbstübersetzung "quasi selbst erkennt" und automatisch den oben gemeinten Bootstrap durchführt, während "gewöhnliche" Modula-2 Compiler eine normale Modula-2 Applikation "sehen". Dies kann durch die Einführung spezieller Übersetzerdirektiven erreicht werden. Diese könnten z.B. folgende Gestalt aufweisen:

```
(* $R- *)
        TYPE
                SHORTCARD = CARDINAL;
                LONGCARD  = CARDINAL;
                SHORTINT  = INTEGER;
                 ...
(* $R+ *)
```

Übersetzt nun ein "normaler" Modula-2 Compiler den Übersetzer, so "sieht" er nur die normalen Modula-2 Standardtypen, während der Modula-2-C Übersetzer bei der Selbstübersetzung die angegebene Deklaration einfach überspringt. Dabei muß allerdings sichergestellt sein, daß im Übersetzer für die Selbstübersetzung die Wertebereiche der Modula-2 Standardtypen ausreichend sind (da hierfür ja z.B. *LONGCARD = CARDINAL* gilt).

3.5.4.3 Portierungsprobleme

Eine Reihe von Problemen, die bei den Portierungstests auftraten, wurden bereits im vorigen Abschnitt erörtert. Probleme, die dort nicht besprochen wurden, weil sie nicht akut auftraten bzw. hauptsächlich für vom Übersetzer erzeugte Applikationen gelten, werden in diesem Abschnitt diskutiert.

Es wurde erwähnt, daß bei der Übertragung zumeist Änderungen in den I/O Bibliotheken auftraten. Der Grund dafür ist, daß bestimmte Sonderzeichen, in verschiedenen Systemen teilweise anders im Zeichensatz (ASCII, EBCDIC) dargestellt werden. Absolut problematisch war jedoch, daß der Übersetzer für einzelne Zeichen einfach den ASCII-Code generierte. Verwendete ein System nicht diesen Code, so kann der Übersetzer (in Form von C-Sourcecode) oder eine von ihm übersetzte Applikation nicht übertragen werden.

Um dieses Problem zu beseitigen, müssen druckbare Zeichen einfach als entsprechende Zeichenkonstante (also z.B. 'A' und nicht ASCII-Code 65) generiert werden. Dies gilt auch für die im Übersetzer verwendeten Sonderzeichen (also

z.B. '\n' für end-of-line). Für den Übersetzer gilt dabei jedoch, daß auch in anderen Zeichensätzen Vergleiche der Form: "('A' <= Zeichen) && (Zeichen <= 'Z')" möglich sein müssen (was allerdings gewährleistet sein sollte).

Ein weiteres allgemeines Problem ist, daß bei Übertragung von Applikationen die Wertebreiche der verwendeten Typen beachtet werden müssen (eine Applikation, die von 32 bit Werten ausgeht, kann nicht einfache in eine 16 bit Umgebung übertragen werden). Selbst bei ordnungsgemäßer Verwendung, kann es z.B. bei floating-point Typen, durch Unterschiede in verschiedenen C-Umgebungen, zu Ungenauigkeiten kommen.

3.5.5 Restriktionen und weitere geplante Entwicklungen

In der derzeit vorliegenden Version des Modula-2-C Übersetzers gibt es, bezüglich der vollständigen Übersetzung von Modula-2, einige Einschränkungen.
So wurde das Prioritäten und Monitorkonzept nicht implementiert. Der Grund dafür ist, daß dadurch die relativ einfache Portierbarkeit des Übersetzers stark leidet, da absolut systemspezifischer Code generiert werden müßte. Die Einschränkung ist, nach Meinung des Autors, allerdings nicht allzu gravierend, da diese Sprachelemente doch eher selten in Modula-2 Applikationen Verwendung finden.

Weiters wurden Coroutinen (*PROCESS*, *NEWPROCESS*, *TRANSFER*) nicht im Modul *SYSTEM* implementiert, sondern in der Modula-2 Bibliothek, da es sich hier ebenfalls um sehr systemabhängige Sprachelemente handelt (siehe Anhang H).

Eine weitere Einschränkung ist, daß (zumindest derzeit) keine Range-Checks durchgeführt werden (eine Ausnahme sind Feldindizes). Derartige Prüfungen werden normalerweise von der Hardware unterstützt, müßten allerdings beim Modula-2-C Übersetzer von der Software (Laufzeitunterstützung) durchgeführt werden. Weiters ergeben sich hier teilweise wiederum Probleme mit der Portabilität.

Die Verwendung des Modula-2-C Übersetzers ist zur Zeit noch relativ unbequem. Der C-Compiler muß gegenwärtig noch "händisch" aktiviert werden. Weiters gibt es keine Unterstützung für das Linken von Applikationen und es wird gegenwärtig noch keine Versionskontrolle durchgeführt. Diese Dinge lassen sich jedoch relativ einfach durch eigene Applikationen implementieren.

Der Übersetzer kann für jeden übersetzten Modul eine Datei erzeugen, die die Namen sämtlicher importierten Moduln enthält. Diese Datei kann dann von einer Applikation gelesen werden, die dann den C-Compiler (der Dateiname ergibt den Namen des zu übersetztenden C-Moduls) aktiviert. Weiters kann in dieser Datei zu jedem Modul das Erzeugungsdatum des entsprechenden Definitionsmoduls abgelegt werden. Diese Dateien können dann von einem "Linker" gelesen werden, der eine Versionskontrolle durchführt und den Systemlinker aktiviert. Diesem werden die Namen sämtlicher Moduln einer Applikation übergeben, damit er eine ausführbare Version erzeugen kann.

3.6 Kapitelzusammenfassung

In diesem Abschnitt wurde die Struktur und Implementierung eines Modula-2-C Übersetzers beschrieben. Am Anfang stand die Darstellung der Anforderungs- und Entwicklungskriterien, wobei die wichtigsten Punkte die Fähigkeit zu Selbstübersetzung, sowie die Portabilität des Übersetzers waren. Vor allem letzteres wird dadurch erleichtert, daß der Übersetzer nicht für eine konkrete Hardware Code erzeugt, sondern für eine abstrakte "C-Maschine".

Weiters wurde erläutert, welche Datenstrukturen für die Übersetzung benötigt wurden. Dies waren im einzelnen Strukturen für die Aufbewahrung von Bezeichnern (String-Table), Strukturen für die Überprüfung der kontext-sensitiven Modula-2 Konstrukte (Symbol-Table) sowie eine Struktur für das Ablegen der Programmstruktur (Symbol-Buffer).

Danach wurde die Struktur und die Funktion der einzelnen Übersetzerkomponenten erläutert. Die wichtigsten davon waren die Initialisierung, die Syntaxanalyse, die Blockanalyse, die Codegenerierung und die Fehlerbehandlung.

Den Abschluß dieses Kapitels bildete schließlich eine Übersicht über den Projektverlauf, die Schritte der Selbstübersetzung, die Probleme, mit denen während der Entwicklung gekämpft wurde und ein Ausblick auf zukünftige Entwicklungsmöglichkeiten.

4 Resümee

Diese Arbeit hatte zum Ziel, die wichtigsten Aspekte der automatischen Übersetzung von Modula-2 nach C darzustellen und zu diskutieren. Es wurde nicht nur "graue Theorie" geboten, sondern der Entwurf und die Implementierung eines Modula-2-C Übersetzers durchgeführt und beschrieben.

Die Zielsetzungen des Projektes waren dabei die folgenden:

a) Die Schaffung einer Verbindung zwischen Modula-2 und C-Applikationen bzw. Bibliotheken.

b) Die möglichst hohe Portabilität des erzeugten C-Sourcecodes.

c) Die möglichst vollständige Abbildung von Modula-2 auf C.

d) Der Modula-2-C Übersetzer sollte in der Lage sein sich selbst zu übersetzen.

e) Der Modula-2-C Übersetzer sollte selbst relativ einfach zu portieren sein.

Die genannten Ziele sind dabei nicht unabhängig voneinander zu sehen, sondern bedingen und beeinflussen sich gegenseitig. So ist die Erreichung von c) notwendig um d) zu erreichen, wenn man sich bei der Entwicklung, hinsichtlich der Verwendung von Modula-2, nicht selbst allzu stark beschränken will. Erreicht man die Ziele b) und d), so ist der Weg zu e) nicht mehr allzu weit, wobei auch a) dem Übersetzer selbst nützt.

Der Übersetzer wurde in zwei Phasen entwickelt. In einem ersten Schritt sollte die Fähigkeit zur Selbstübersetzung erreicht werden, wobei natürlich bei der Entwicklung die Portabilität des Übersetzers und des von ihm erzeugten Outputs berücksichtigt bzw. angestrebt wurde. Erst in einem zweiten Schritt wurde dann begonnen die "Standfestigkeit" des Übersetzers, bezüglich des Verhaltens gegenüber fehlerbehaftetem Input, zu verbessern. Das Projekt befindet sich, nach

Einschätzung des Autors, zum gegenwärtigen Zeitpunkt (d.h. beim Abschluß dieser Arbeit) ungefähr in der Mitte der zweiten Phase, allerdings ist der Übersetzer für Softwareentwicklungsprojekte (nicht unbedingt für solche von Modula-2 Anfängern) einsatzfähig.

Der gewählte Entwicklungsmodus hat sich insofern bewährt, als bereits sehr früh Portierungen auf unterschiedliche Systeme durchgeführt werden konnten, wobei bei jeder Portierung wertvolle Erkenntnisse gewonnen werden konnten und Impulse für Verbesserungen gesetzt wurden. Der Übersetzer läuft daher bereits im derzeitigen Entwicklungsstadium auf vier, teilweise sehr unterschiedlichen, Systemen (MS-DOS, UNIX (CISC, RISC), VMS) bzw. ist ohne großen Aufwand auf solche Systeme zu portieren.

Als Wermutstropfen ist jedoch zu betrachten, daß der Übersetzer bis jetzt vor allem für die Selbstübersetzung und vom Autor selbst verwendet wurde. Es wurden zwar zahlreiche kleinere Moduln zu Testzwecken erzeugt und übersetzt, allerdings keine größeren Applikationen. Es mangelt also derzeit noch am Einsatz für andere Projekte und durch andere Benutzer. Dies wäre in der derzeitigen Entwicklungsphase notwendig, um den notwendigen "Druck" und neue Impulse, für die Verfeinerung der Handhabung und die weitere Verbesserung der Qualität des Übersetzers, zu erzeugen. Trotz dieses Mankos kann man sagen, daß die gesetzten Ziele erreicht wurden.

Hinsichtlich der Unterteilung des Übersetzers in vier Phasen ist zu bemerken, daß diese Aufgabenteilung und die Unabhängigkeit der Komponenten für die Entwicklung und deren Fortschritt recht positiv war. Man kann sich jedoch Überlegen, ob in Zukunft nicht manche Aufgaben, aus Effizienzgründen, zusammengelegt werden. So könnte etwa bereits während der Syntaxanalyse mit dem Aufbau der Symbol-Table begonnen werden. Weiters könnte dann in dieser Phase (zumindest teilweise) auch eine Deklarationsanalyse durchgeführt werden.

Die Abbildung von Modula-2 nach C war in weiten Bereichen keine allzu schwierige Aufgabe. Nur das Modulkonzept, sowie die verschachtelten Prozeduren von Modula-2 erforderten einen größeren Aufwand für die Übertragung nach C.

Allerdings wurde auf die Implementierung des Monitor- und Prioritätenkonzeptes von Modula-2 verzichtet, um die Portabilität des Übersetzers nicht einzuschränken und zu erschweren.

Der Bau von Konvertern, die zwischen Hochsprachen übersetzen und somit Code für "abstrakte Maschinen" produzieren ist eine interessante Tätigkeit, die die Portabilität von Applikationen und damit letztlich ihren Wert beträchtlich erhöhen kann. Allerdings ist es dafür auch notwendig, die Architektur solcher "abstrakten Maschinen" bzw. die Syntax und Semantik von Programmiersprachen sehr exakt zu formulieren und entsprechende Compiler zu implementieren. Nur dann kann die Übersetzung zwischen höheren Programmiersprachen erfolgreich durchgeführt werden. Ist dies allerdings gewährleistet, dann wird, mit hoher Wahrscheinlichkeit, diese Art der Übersetzungstechnik für die Softwareentwicklung in Zukunft eine immer wichtigere Rolle spielen.

Literaturverzeichnis

[01] L. B. Geissmann:
Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith,
ETH Diss. Nr. 7286, Zürich, 1983

[02] D. Defoe:
Die seltsamen und erstaunlichen Abenteuer des Robinson Crusoe, Bertelsmann, Gütersloh, 1973

[03] B. W. Kernighan, D. M. Ritchie:
The C programming language,
Prentice Hall, New Jersey, 1988

[04] L. Böszörményi, A.Ercsényi, M.Szabó:
MOMO: A MODULA-2 system for the MOTOROLA 680x0 family,
Computer & Automation Institute, Hungarian Academy of Sciences, Budapest, 1988

[05] H. Frank, C. Starzacher:
Modula-2 to C transformator,
Institut für Informatik, Universität Klagenfurt, 1990

[06] N. Wirth:
The personal computer Lilith,
in Tutorial: Software Development Environments, IEEE Computer Society, Hrsg.: Anthony I. Wasserman, University of California, San Francisco, 1981

[07] N. Wirth:
Programmieren in Modula-2,
Springer Verlag (Übers. d. third corrected edition), 1985

[08] G. Blaschek, G. Pomberger, F. Ritzinger:
Einführung in die Programmierung mit Modula-2,
Springer Verlag, 1986

[09] M. Odersky:
MINOS: A New Approach to the Design of an Input/Output Library for Modula-2,
ETH Report Nr. 86 des Instituts für Informatik, Fachgruppe Computer-Systeme, Zürich, 1988

[10] R. Crelier:
OP2: A Portable Oberon Compiler,
ETH Report Nr. 125 des Instituts für Informatik, Fachgruppe Computer-Systeme, Zürich, 1990

[11] C. Jacobi:
Code Generation and the Lilith Architecture,
ETH Diss. Nr. 7195, Zürich, 1982

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

[13] L. Böszörményi:
Program Execution on the LILIPUTH Workstation, Design and Implementation,
Institut für Informatik, Klagenfurt, 1990

[14] K. Louden:
P-Code and Compiler Portability: Experience with a Modula-2 Optimizing Compiler,
ACM Sigplan Notices 25:5, S. 53-60, 1990

[15] N. Sundaresan:
Translation of Nested Pascal Routines to C,
ACM Sigplan Notices 25:5, S. 69-82, 1990

[16] K. Bothe, B. Hohberg, Ch. Horn, O. Wikarski:
A Portable High-Speed PASCAL to C Translator,
ACM Sigplan Notices 24:9, S. 60-66, 1989

[17] A.S. Tanenbaum, M.F. Kaashoek, K.G. Langendoen, C.J.H. Jacobs:
The Design of Very Fast Portable Compilers,
ACM Sigplan Notices 24:11, S. 125-132, 1989

[18] V.E. Waddle:
Production Trees: A Compact Representation of Parsed Programs,
ACM Transactions on Programming Languages and Systems 12:1, S. 61-84, 1990

[19] N. Wirth:
Algorithmen und Datenstrukturen mit Modula-2,
Teubner, Stuttgart, 1986

[20] N. Wirth:
Compilerbau, Eine Einführung,
Teubner, Stuttgart, 1984

[21] R. Sethi:
Programming Languages, Concepts and Constructs,
Addison Wesley, Reading, 1989

[22] Logitech:
Program Converter from Turbo-Pascal to Modula-2/86,
Documentation, Logitech, 1986

Abbildungsverzeichnis

Anhang A: Abbildung der Syntax von Modula-2 nach C

In diesem Abschnitt wird *eine mögliche* (nicht die) Abbildung der Modula-2 Syntax auf die Syntax von C gezeigt. Dabei wurde der Syntax von C insofern "Gewalt angetan", als von der Modula-2 Syntax ausgegangen wird und die Syntax von C entsprechend aufgespalten wird (Ich hoffe, daß C darunter nicht allzu sehr gelitten hat bzw. die Syntax korrekt geblieben ist). Dabei werden von C nur jene Teile gezeigt, die für die Abbildung benötigt werden.

Die rein syntaktische Abbildung ist dabei natürlich insofern problematisch, als gewisse Sprachelemente vom Modula-2 nicht rein syntaktisch auf C abgebildet werden können. Hier spielt die Semantik eine entscheidende Rolle, die mit der EBNF-Notation nicht dargestellt werden kann. Dieser Abschnitt muß daher in Zusammenhang mit dem 2. Kapitel dieser Arbeit gesehen werden, wo entsprechende Beschreibungen und Begründungen für die gewählte Form der Übersetzung geboten werden. Einige "kritische" Sprachelement wurden, auch in diesem Abschnitt, noch zusätzlich mit Anmerkungen versehen.

Die Nicht-Terminalsymbole, die zu Modula-2 gehören, beginnen alle mit "M_", während die zu C gehörenden mit "C_" beginnen. Nicht-Terminalsymbole, die nicht mit "M_" oder "C_" beginnen, gelten für beide Sprachen. Die Schlüsselwörter von Modula-2 werden durch Großschreibung dargestellt, während jene von C einfach als Strings (normale Terminalsybole) beschrieben werden.

```
M_ident ::= letter {letter | digit}.
C_ident ::= (letter | "_") {letter | digit | "_"}.

M_number ::=      M_integer | real.
C_number ::=      C_integer | real.
```

Anmerkung zu C_integer:
Bei der Übersetzung werden octale und hexadezimale Ziffern in dezimale Darstellung übertragen.

```
M_integer ::=     digit {digit} | octalDigit {octalDigit} ("B" | "C") | digit
                  {hexDigit} "H".
C_integer ::=     digit {digit} | "0" octalDigit {octalDigit} | "0" ("x" |
                  "X") {hexDigit}.

real ::=    digit {digit} "." {digit} [ScaleFactor].

ScaleFactor ::=   ("e" | "E") ["+" | "-"] digit {digit}.

hexDigit ::=      digit | "A" | "B" | "C" | "D" | "E" | "F".

digit ::=   octalDigit | "8" | "9".

octalDigit ::=    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7".
```

```
M_string ::=      "'" {character} "'" | '"' {character} '"'.
C_string ::=      '"' {character} '"'.
```

Anmerkung zu C_qualident:
M_qualident muß entsprechend auf C_qualident übertragen werden.

```
M_qualident ::=   M_ident {"." M_ident}.
C_qualident ::=   C_ident.
```

Anmerkung zu M_ConstantDeclaration:
In C werden Konstante normalerweise mit der #define Direktive deklariert. Diese werden dann vom C-Präprozessor aufgelöst. Für die Übersetzung von Modula-2 auf C können Konstante vom Übersetzer ausgewertet werden. Daher wird hier keine Gegenübestellung von Modula-2 und C Konstantendeklarationen durchgeführt.

```
M_ConstantDeclaration ::=     M_ident "=" M_ConstExpression.

M_ConstExpression ::=   M_SimpleConstExpr [M_relation M_SimpleConstExpr].
```

Anmerkung zu C_relation:
Für Mengenoperationen muß vom Übersetzer entsprechender Code generiert werden.

```
M_relation ::=    "=" | "#" | "<>" | "<" | "<=" | ">" | ">=" | IN.
C_relation ::=    "==" | "!=" | "<" | "<=" | ">" | ">=".

M_SimpleConstExpr ::=   ["+" | "-"] M_ConstTerm {M_AddOperator M_ConstTerm}.

M_AddOperator ::= "+" | "-" | OR.
C_AddOperator ::= "+" | "-" | "||".

M_ConstTerm ::=   M_ConstFactor {M_MulOperator M_ConstFactor}.

M_MulOperator ::= "*" | "/" | DIV | MOD | AND | "&".
C_MulOperator ::= "*" | "/" | "%" | "&&".

M_ConstFactor ::= M_qualident | M_number | M_string | M_ConstSet | "("
                  M_ConstExpression ")" | NOT M_ConstFactor.
```

Anmerkung zu M_ConstSet:
Wie ein Menge in einer Konstantendeklaration ausgewertet wird, ist von der gewählten Repräsentation von Mengen (*BITSET*) abhängig.

```
M_ConstSet ::=    [M_qualident] "{" [M_ConstElement {"," M_ConstElement}]
                  "}".

M_ConstElement ::=      M_ConstExpression [".." M_ConstExpression].
```

```
M_TypeDeclaration ::=   M_ident "=" M_type.
C_TypeDeclaration ::= "typedef" C_typedecl.

Anmerkung zu C_typedecl:
Hier erfolgt für C eine Aufspaltung in "normale" Typen und Prozedurtypen.

C_typedecl ::=    C_type C_ident | Ctype "(*" C_ident ")()".

M_type ::=  M_SimpleType  |  M_ArrayType  |  M_RecordType  |  M_SetType  |
            M_PointerType | M_ProcedureType.
C_type ::=  C_SimpleType  |  C_ArrayType  |  C_RecordType  |  C_SetType  |
            C_PointerType.

M_SimpleType ::=  M_qualident | M_enumeration | M_SubrangeType.
C_SimpleType ::= C_standard  | C_enumeration | C_SubrangeType.

C_standard ::=    "void" | "char" | C_int | "float" | ["long"] "double".

C_int ::=   ["unsigned"] ["long" | "short"] "int".

M_enumeration ::= "(" M_IdentList ")".
C_enumeration ::= C_int.

M_IdentList ::=   M_ident {"," M_ident}.
C_IdentList ::=   C_ident {"," C_ident}.

M_SubrangeType ::=      [M_ident]    "["    M_ConstExpression    ".."
                        M_ConstExpression "]".
C_SubrangeType ::=      C_int.

Anmerkung zu C_ArrayType:
Felder werden in Records "verpackt".

M_ArrayType ::=   ARRAY M_SimpleType {"," M_SimpleType} OF M_type.
C_ArrayType ::=   "struct" [C_ident] "{" C_type "A"" [" C_expression "]" "}".

M_RecordType ::=  RECORD M_FieldListSequence END.
C_RecordType ::=  "struct" [C_ident] "{" C_FieldListSequence "}".

M_FieldListSequence ::= M_FieldList {";" M_FieldList}.
C_FieldListSequence ::= C_FieldList {";" C_FieldList}.

Anmerkung zu C_FieldList:
Variantenrecords werden auf unions abgebildet. Ist eine Schalterkomponente
gegeben, so muß diese vor der union generiert werden.

M_FieldList ::=   [M_IdentList ":" M_type | CASE [M_ident] ":" M_qualident OF
                  M_variant {"|" M_variant} [ELSE M_FieldListSequence] END].
```

```
C_FieldList ::=   [C_type C_IdentList | [C_type C_ident ";"] "union" "{"
                  C_variant {C_variant} "}".

M_variant ::=     [M_CaseLabelList ":" M_FieldListSequence].
C_variant ::=     "struct" "{" C_FieldListSequence "}".

M_CaseLabelList ::=     M_CaseLabels {"," M_CaseLabels}.

M_CaseLabels ::=  M_ConstExpression [".." M_ConstExpression].

M_SetType   ::=   SET OF SimpleType.
C_SetType ::=     C_int.

M_PointerType ::= POINTER TO M_type.
C_PointerType ::= C_type "*".

M_ProcedureType ::=     PROCEDURE [M_FormalTypeList].

M_FormalTypeList ::=    "(" [[VAR] M_FormalType {"," [VAR] M_FormalType}]
                        ")" [":" M_qualident].

M_VariableDeclaration ::=     M_IdentList ":" M_type.
C_VariableDeclaration ::=     ["extern" | "static"] C_type C_IdentList.
```

Anmerkung zu C_designator:
Die vorliegende Abbildung ist etwas problematisch. In C sind hier bei der Codegenerierung die entsprechenden Rangfolgen der Operatoren (z.B. durch Klammerung) zu beachten. Für Felder wird hier der Zugriff auf ein Element entsprechend der Felddeklaration gezeigt.

```
M_designator ::=  M_qualident {"." M_ident | "[" M_ExpList "]" | "^"}.
C_designator ::=  (C_qualident | "(" {"*"} C_qualident ")") {"." C_ident |
                  ("." | "->") "A" "[" C_expression "]" | "->"}.

M_ExpList ::= M_expression {"," M_expression}.
C_ExpList ::= ["&"] C_expression {"," ["&"] C_expression}.

M_expression ::=  M_SimpleExpression [M_relation M_SimpleExpression].
C_expression ::=  C_SimpleExpression [C_relation C_SimpleExpression].

M_SimpleExpression ::=  ["+" | "-"] M_term {M_AddOperator M_term}.
M_SimpleExpression ::=  ["+" | "-"] C_term {C_AddOperator C_term}.

M_term ::=  M_factor {M_MulOperator M_factor}.
C_term ::=  C_factor {C_MulOperator C_factor}.
```

Anmerkung zu C_factor:
Für Mengen (M_set), muß vom Übersetzer entsprechender Code erzeugt werden.

```
M_factor ::=      M_number    |    M_string    |    M_set    |    M_designator
                  [M_ActualParameters] | "(" M_expression ")" | NOT M_factor.
C_factor ::=      C_number | C_string | C_designator [C_ActualParameters] |
                  "(" C_expression ")" | "!" C_factor.

M_set ::=   [M_qualident] "{" [M_element {"," M_element}]"}".

M_element ::=     M_expression [".." M_expression].

M_ActualParameters ::= "(" [M_ExpList] ")".
C_ActualParameters ::= "(" [C_ExpList] ")".

M_statement ::=   [M_assignment   |   M_ProcedureCall   |   M_IfStatement   |
                  M_CaseStatement | M_WhileStatement | M_RepeatStatement |
                  M_LoopStatement | M_ForStatement | M_WithStatement | EXIT
                  | RETURN [M_expression]].
C_statement ::=   [C_assignment   |   C_ProcedureCall   |   C_IfStatement   |
                  C_CaseStatement | C_WhileStatment | C_RepeatStatement |
                  C_LoopStatement  |  C_ForStatement  |  C_WithStatement  |
                  "break" | "return" [C_expression]].

Anmerkung zu C_assignment:
Für die Zuweisung von Zeichenketten muß vom Übersetzer entsprechender Code
(Kopieroperation) erzeugt werden.

M_assignment ::=  M_designator ":=" M_expression.
C_assignment ::=  C_designator "=" C_expression.

M_ProcedureCall ::=     M_designator [M_ActualParameters].
C_ProcedureCall ::=     C_designator [C_ActualParameters].

M_StatementSequence ::= M_statement {";" M_statement}.
C_StatementSequence ::= C_statement {";" C_statement}.

M_IfStatement ::= IF    M_expression    THEN    M_StatementSequence    {ELSIF
                  M_expression    THEN    M_StatementSequence}    [ELSE
                  M_StatementSequence] END.
C_IfStatement ::= "if (" C_expression ") {" C_StatementSequence ["} else {"
                  C_StatementSequence] "}".

Anmerkung zu C_CaseStatement:
CASE-Anweisungen können entsprechend auf IF-Anweisungen abgebildet werden.

M_CaseStatement ::=     CASE  M_expression  OF  M_case  {"|"  M_case}  [ELSE
                        M_StatementSequence] END.
C_CaseStatement ::=     C_IfStatement.

M_case ::=  [M_CaseLabelList ":" M_StatementSequence].
```

```
M_WhileStatement ::=    WHILE M_expression DO M_StatementSequence END.
C_WhileStatement ::=    "while (" C_expression ") {" C_StatementSequence
                        "}".

M_RepeatStatement ::=   REPEAT M_StatementSequence UNTIL M_expression.
C_RepeatStatement ::=   "do {" C_StatementSequence "} while "(!("
                        C_expression "))".
```

Anmerkung zu C_ForStatement:
Für FOR-Anweisungen wird (bei Abbildung auf while-Anweisungen) ein entsprechender C_block mit lokalen Variablen erzeugt.

```
M_ForStatement ::=      FOR M_ident ":=" M_expression TO M_expression [BY
                        M_ConstExpression] DO M_StatementSequence END.
C_ForStatement ::=      C_block.

M_LoopStatement ::=     LOOP M_StatementSequence END.
C_LoopStatement ::=     "while (1) {" C_StatementSequence "}".
```

Anmerkung zu C_WithStatement:
WITH-Anweisungen werden auf einen C_block abgebildet.

```
M_WithStatement ::=     WITH M_designator DO M_StatementSequence END.
C_WithStatement ::=     C_block.

M_ProcedureDeclaration ::=    M_ProcedureHeading ";" M_block M_ident.
C_ProcedureDeclaration ::=    ["extern" | "static"] C_type
                              C_ProcedureHeading C_block.

M_ProcedureHeading ::=  PROCEDURE M_ident [M_FormalParameters].
C_ProcedureHeading ::=  C_ident [C_FormalParameters].

M_block ::= {M_declaration} [BEGIN M_StatementSequence] END.
C_block ::= "{" {C_TypeDeclaration | C_VariableDeclaration}
            C_StatementSequence "}".

M_declaration ::= CONST {M_ConstantDeclaration ";"} | TYPE {M_TypeDeclaration
                  ";"} | VAR {M_VariableDeclaration ";"} |
                  M_ProcedureDeclaration ";" | M_ModuleDeclaration ";".
C_declaration ::= {C_TypeDeclaration ";"} | {C_VariableDeclaration ";"} |
                  {C_ProcedureDeclaration ";"} | {C_ModuleDeclaration ";"}.

M_FormalParameters ::=  "(" [M_FPSection {";" M_FPSection}] ")" [":"
                        M_qualident].
C_FormalParameters ::=  "(" [C_IdentList] ")" {C_typedecl}.

M_FPSection ::=   [VAR] M_IdentList ":" M_FormalType.
```

```
M_FormalType ::=  [ARRAY OF] M_qualident.

M_ModuleDeclaration ::= MODULE M_ident [priority] ";" {M_import} [M_export]
                        M_block M_ident.
C_ModuleDeclaration ::= C_ProcedureDeclaration.
```

Anmerkung zu M_priority:
Prioritäten werden in der vorliegenden Implementierung ignoriert.

```
M_priority ::=    "[" M_ConstExpression "]".
```

Anmerkung zu M_export:
Objekte die exportiert werden, müssen in C global deklariert werden (nicht "static").

```
M_export ::=      EXPORT [QUALIFIED] M_IdentList ";".
```

Anmerkung zu M_import:
Für Objekte die importiert werden, müssen in C entsprechende "extern" Deklarationen erzeugt werden.

```
M_import ::=      [FROM M_ident] IMPORT IdentList ";".

M_DefinitionModule ::=  DEFINITION   MODULE   M_ident   ";"   {M_import}
                        {M_definition} END M_ident ".".

M_definition ::=  CONST {M_ConstantDeclaration ";"} | TYPE {M_ident ["="
                  M_type]  ";"}  |  VAR  {M_VariableDeclaration  ";"}  |
                  M_ProcedureHeading ";".

M_ProgramModule ::=     MODULE M_ident [M_priority] ";" {M_import} M_block
                        M_ident ".".
C_ProgramModule ::=     C_declaration.

M_CompilationUnit ::=   M_DefinitionModule   |   [IMPLEMENTATION]
                        M_ProgramModule.
C_CompilationUnit ::=   C_ProgramModule.
```

Anhang B: Übersetzung von Prozeduren mit ARRAY-Parametern

```
(*-----------------------  Modula-2  ------------------------*)

MODULE Procedure;
TYPE
  String = ARRAY [0..30] OF CHAR;
VAR
  str      : String;

  PROCEDURE StrCopy(VAR str1 : ARRAY OF CHAR;
                        str2 : ARRAY OF CHAR);
  VAR
    i : CARDINAL;
  BEGIN
    i := 0;
    WHILE (i <= HIGH(str1)) AND (i <= HIGH(str2)) AND
          (str2[i] # 0C) DO
      str1[i] := str2[i];
      INC(i);
    END (* while *);
    IF (i <= HIGH(str1)) THEN
      str1[i] := 0C;
    END (* if *);
  END StrCopy;

BEGIN
  StrCopy(str, "Dies ist ein String");
END Procedure.

/*-----------------------     C      ------------------------*/

#include "M2rts.h"

/* MODULE */ void Procedure();
typedef struct String {
  CHAR  A[31];
} _T_String;

static struct String str;
static /* PROCEDURE */ void StrCopy();

/* MODULE */ void Procedure()
{
  StrCopy(str.A, 31, "Dies ist ein String\0", 19*sizeof(CHAR));
}

static void StrCopy(A_str1, _str1, A_str2, _str2)
```

```
  CHAR   * A_str1;
  CARDINAL  _str1;
  CHAR   * A_str2;
  CARDINAL  _str2;
{
  CARDINAL i;
  struct {
    CHAR   * A;
  } str1;
  struct {
    CHAR   * A;
  } str2;
  str1.A = A_str1;
  str2.A = (CHAR  *) _ALLOCATE(_str2*sizeof(CHAR));
  _COPY(str2.A,A_str2,_str2*sizeof(CHAR),_str2,TRUE);
  i=0;
  while ((i <= HIGH(_str1*sizeof(CHAR)-1)) &&
         (i <= HIGH(_str2*sizeof(CHAR)-1)) &&
         (str2.A[_ARRAYCHK(i,0,_str2*sizeof(CHAR ))] != 0)) {
    str1.A[_ARRAYCHK(i,0,_str1*sizeof(CHAR))]=
    str2.A[_ARRAYCHK(i,0,_str2*sizeof(CHAR))];
    INC(i,1);
  };
  if ((i <= HIGH(_str1*sizeof(CHAR)-1))) {
    str1.A[_ARRAYCHK(i,0,_str1*sizeof(CHAR))]=0;
  } ;
  _DEALLOCATE(str2.A);
}
```

Anhang C: Übersetzung verschachtelter Prozeduren

Im folgenden wird ein Beispiel für die Übersetzung von verschachtelten Prozeduren gezeigt. Das Beispiel wurde dabei in veränderter Form aus der Arbeit von N. Sundaresan [15] übernommen. Zuerst wird das zu übersetzende Modula-2 Beispiel gezeigt. Dieses wurde dann von Hand (in Anlehnung an [15]) übersetzt, um die Prinzipien der Technik der Strukturisierung und der erweiterten Parameterlisten zu zeigen. Der Output, der die Technik der displayorientierten Strukturen zeigt, wurde vom Modula-2-C Übersetzer erzeugt.

Ein Modula-2 Beispiel mit verschachtelten Prozeduren:

```
MODULE Nested;

  PROCEDURE Proc(param : INTEGER);
  VAR
    local : INTEGER;

    PROCEDURE Proc1(param1 : INTEGER);
    VAR
      local1 : INTEGER;

      PROCEDURE Proc11();
      VAR
        local11 : INTEGER;
      BEGIN
        local11 := 1;
      local1  := 1;
      Proc2(local11);
      END Proc11;

    BEGIN
      local1 := 1;
      local  := 1;
      param  := 1;
    END Proc1;

    PROCEDURE Proc2(param2 : INTEGER);
    VAR
      local2 : INTEGER;

      PROCEDURE Proc21();
      BEGIN
      END Proc21;

    BEGIN
      Proc1(param2);
```

```
    END Proc2;

  BEGIN
    Proc2(10);
  END Proc;

BEGIN
  Proc(25);
END Nested.
```

Die Übersetzung von verschachtelten Modula-2 Prozeduren mit Hilfe der Strukturisierungs-Technik [15]:

```
/*-------------------------  Proc  --------------------------*/

typedef struct {
  int param;
} Ft_Proc;

typedef struct {
  int *pL;
  int local;
} Lt_Proc;

void Proc(fV)
  Ft_Proc fV;
{
  void Proc2();
  void Proc1();
  Lt_Proc lV;

  lV.pL = (int *) &fV;
  Proc2((int *) &lV, 10);
}

/*-------------------------  Proc1 -------------------------*/

void Proc2();  /* foreward */

typedef struct {
  int *sL;
  int param1;
} Ft_Proc1;

typedef struct {
  int *pL;
  int local1;
} Lt_Proc1;

void Proc1(fV)
  Ft_Proc1 fV;
{
  void Proc11();
  Lt_Proc1 lV;

  lV.pL = (int *) &fV;
  lV.local1 = 1;
  ((Lt_Proc *)*(int *)&fV)->local = 1;
```

```
  ((Ft_Proc *)**(int **)&fV)->param = 1;
}

/*------------------------- Proc11 --------------------------*/

void Proc11(sL)
  int *sL;
{
  int local11;

  local11 = 1;
  ((Lt_Proc1 *) sL)->local1 = 1;
  Proc2(**(int **) sL, local11);
}

/*------------------------- Proc2 --------------------------*/

typedef struct {
  int *sL;
  int param2;
} Ft_Proc2;

typedef struct {
  int *pL;
} Lt_Proc2;

void Proc2(fV)
  Ft_Proc2 fV;
{
  void d21();
  Lt_Proc2 lV;

  lV.pL = (int *) &fV;
  Proc1(*(int *) &fV, fV.param2);
}

/*------------------------- Proc21 --------------------------*/

void Proc21(sL)
  int * sL;
{
}
/*------------------------- Nested --------------------------*/

void Nested()
{
  Proc(25);
}
```

Die Übersetzung von verschachtelten Modula-2 Prozeduren mit Hilfe von erweiterten Parameterlisten [15]:

```
/*-------------------------  Proc  --------------------------*/

void Proc(param)
  int param;
{
  void Proc2();
  void Proc1();
  int  local;

  Proc2(10, &local, &param);
}

/*------------------------- Proc1  --------------------------*/

void Proc2();  /* foreward */

void Proc1(param1, p_local, p_param)
  int param1;
  int * p_local;
  int * p_param;
{
  void Proc11();
  int local1;

  local1 = 1;
  *p_local = 1;
  *p_param = 1;
}

/*------------------------- Proc11 -------------------------*/

void Proc11(p_local1, p_local, p_param)
  int * p_local1;
  int * p_local;
  int * p_param;
{
  int local11;

  local11 = 1;
  *p_local1 = 1;
  Proc2(local11, p_local, p_param);
}
```

```
/*------------------------- Proc2 ---------------------------*/

void Proc2(param2, p_local, p_param)
  int param2;
  int * p_local;
  int * p_param;
{
  void Proc21();

  Proc1(param2, p_local, p_param);
}

/*------------------------- Proc21 --------------------------*/

void Proc21()
{
}

/*------------------------- Nested --------------------------*/

void Nested()
{
  Proc(25);
}
```

Die Übersetzung von verschachtelten Modula-2 Prozeduren mit Hilfe von displayorientierten Strukturen:

```
#include "M2rts.h"

/* MODULE */ void Nested();
static /* PROCEDURE */ void Proc();
typedef struct {
  INTEGER param;
  INTEGER local;
} _G_Proc;
static _G_Proc  * _GP_Proc;

/* PROCEDURE */ void Proc1();
  typedef struct {
    INTEGER param1;
    INTEGER local1;
  } _G_Proc1;
  static _G_Proc1  * _GP_Proc1;

/* PROCEDURE */ void Proc11();
/* PROCEDURE */ void Proc2();
  typedef struct {
    INTEGER param2;
    INTEGER local2;
  } _G_Proc2;
  static _G_Proc2  * _GP_Proc2;

/* PROCEDURE */ void Proc21();

/*------------------------- Nested --------------------------*/

/* MODULE */ void Nested()
{
  Proc(25);
}

/*------------------------- Proc ----------------------------*/


static void Proc(param)
  INTEGER param;
{
  _G_Proc _L_Proc,  * _LP_Proc = _GP_Proc;

  _GP_Proc = &_L_Proc;
  _GP_Proc->param = param;
  Proc2(10);
```

```
  _GP_Proc = _LP_Proc;

}

/*------------------------- Proc1 ---------------------------*/

static void Proc1(param1)
  INTEGER param1;
{
  _G_Proc1 _L_Proc1, * _LP_Proc1 = GP_Proc1;

  _GP_Proc1 = &_L_Proc1;
  _GP_Proc1->param1 = param1;
  _GP_Proc1->local1=1;
  _GP_Proc->local=1;
  _GP_Proc->param=1;
  _GP_Proc1 = _LP_Proc1;

}

/*------------------------- Proc11 ---------------------------*/

static void Proc11()
{
  INTEGER local11;
  local11=1;
  _GP_Proc1->local1=1;
  Proc2(local11);
}

/*------------------------- Proc2 ---------------------------*/

static void Proc2(param2)
  INTEGER param2;
{
  _G_Proc2 _L_Proc2,  * _LP_Proc2 = _GP_Proc2;

  _GP_Proc2 = &_L_Proc2;
  _GP_Proc2->param2 = param2;
  Proc1(_GP_Proc2->param2);
  _GP_Proc2 = _LP_Proc2;

}
/*------------------------- Proc21 ---------------------------*/

static void Proc21()
{
}
```

```
(*-----------------------  Modula-2  ------------------------*)

MODULE Branches;
VAR
  c : CARDINAL;
BEGIN
  IF (c >= 0) AND (c <= 3) THEN
    c := 1;
  ELSIF (c = 4) THEN
    c := 2;
  ELSE
    c := 3;
  END (* if *);

  CASE c OF
    0..3 : c := 1;
  | 4    : c := 2;
  ELSE
    c := 3;
  END (* case *);
END Branches.

/*-----------------------     C      ------------------------*/

#include "M2rts.h"
/* MODULE */ void Branches();
static CARDINAL c;

/* MODULE */ void Branches()
{
  if ((c >= 0) && (c <= 3)) {
    c=1;
  } else if ((c == 4)) {
    c=2;
  } else {
    c=3;
  } ;

  if ((0 <= c) && (c <= 3)) {
    c=1;
  } else if ((4 == c)) {
    c=2;
  } else {
    c=3;
  };
}
```

Angang E: Die Übersetzung von Schleifen

```
(*-----------------------  Modula-2  ------------------------*)

MODULE Loops;
VAR
  c, i : CARDINAL;
BEGIN
  c := 0;
  WHILE (c < 3) DO
    INC(c);
  END (* while *);

  REPEAT
    INC(c);
  UNTIL (c > 6);

  FOR i := 0 TO 3 DO
    INC(c);
  END (* for *);

  LOOP
    IF (c = 33) THEN
      EXIT;
    END (* if *);
    INC(c);
  END (* loop *);
END Loops.

/*-----------------------     C      ------------------------*/

#include "M2rts.h"

/* MODULE */ void Loops();
static CARDINAL c;
static CARDINAL i;

/* MODULE */ void Loops()
{
  c=0;
  while ((c < 3)) {
    INC(c,1);
  };

  do {
    INC(c,1);
  } while (!((c > 6)));
```

```
  {
    CARDINAL _FROM=0, _TO=3;
    i=0;
    while ((_FROM <= i) && (i <=_TO)) {
      INC(c,1);
      i+=1;
    }
  };

  while (TRUE) {
    if ((c == 33)) {
      break;
    } ;
    INC(c,1);
  };
}
```

Anhang F: Modulübersicht des Modula-2-C Entwicklungssystems

Der folgende Abschnitt zeigt eine Liste sämtlicher Moduln, aus denen der Modula-2-C Übersetzer besteht. Danach folgt eine Liste sämtlicher Bibliotheksmoduln. Die Zeilen- und Wortzahlen wurden mit einem einfachen "word count" Programm ermittelt und schließen Kommentare sowie Leerzeilen ein.

Moduln des Modula-2-C Übersetzers:

Zeilen	Worte	Modulname	Kurzbeschreibung
64 L	179 W	M2Buffer.def	Symbol-Buffer
63 L	171 W	M2Comp.def	Hauptkontrollmodul
35 L	105 W	M2Config.def	Konfigurationsdatei
39 L	110 W	M2Const.def	Konstantenauswertung
41 L	113 W	M2Debug.def	Datenstrukturdebugging
47 L	132 W	M2Dialog.def	Dialogkomponente
40 L	122 W	M2Error.def	Errorhandling
44 L	136 W	M2File.def	Übersetzer File I/O
41 L	130 W	M2Hash.def	String-Table
33 L	90 W	M2Init.def	Initialisierung
29 L	81 W	M2Lister.def	Listinggenerator
29 L	84 W	M2P1.def	Pass 1 Kontrollmodul
31 L	87 W	M2P1Pars.def	Parser
65 L	147 W	M2P1Scan.def	Scanner
29 L	84 W	M2P2.def	Pass 2 Kontrollmodul
29 L	84 W	M2P2Decl.def	Deklarationsanalyse
36 L	96 W	M2P2Pars.def	Symbol-Table Aufbau
29 L	84 W	M2P3.def	Pass 3 Kontrollmodul
32 L	90 W	M2P3Pars.def	Blockanalyse
29 L	84 W	M2P4.def	Pass 4 Kontrollmodul
32 L	89 W	M2P4Body.def	Generierung Anweisungen
36 L	101 W	M2P4Decl.def	Generierung Deklarationen
101 L	280 W	M2P4Gen.def	C-Sourcecode Output
514 L	1474 W	M2Sym.def	Symbol-Table
89 L	256 W	M2Type.def	Type-Checking
1557 L	4409 W	total	

```
 Zeilen   Worte   Modulname         Kurzbeschreibung
-------------------------------------------------------------
  296 L    785 W   M2Buffer.mod     siehe Definitionsmodul
   29 L     81 W   m2c.mod          Hauptmodul
  285 L    752 W   M2Comp.mod       siehe Definitionsmodul
  178 L    540 W   M2Config.mod     "
 1251 L   4418 W   M2Const.mod      "
 1478 L   3946 W   M2Debug.mod      "
  508 L   1534 W   M2Dialog.mod     "
  426 L   1671 W   M2Error.mod      "
  136 L    376 W   M2File.mod       "
  323 L    894 W   M2Hash.mod       "
  852 L   2124 W   M2Init.mod       "
  272 L    706 W   M2Lister.mod     "
   66 L    165 W   M2P1.mod         "
 1531 L   5025 W   M2P1Pars.mod     "
  587 L   1979 W   M2P1Scan.mod     "
   85 L    242 W   M2P2.mod         "
 1247 L   3810 W   M2P2Decl.mod     "
 1502 L   4757 W   M2P2Pars.mod     "
   77 L    220 W   M2P3.mod         "
 1479 L   5526 W   M2P3Pars.mod     "
   83 L    232 W   M2P4.mod         "
 1998 L   6066 W   M2P4Body.mod     "
  980 L   2998 W   M2P4Decl.mod     "
  784 L   2359 W   M2P4Gen.mod      "
 2423 L   6817 W   M2Sym.mod        "
  716 L   1972 W   M2Type.mod       "
-------------------------------------------------------------
19592 L  59995 W   total
```

Moduln der Modula-2-C Bibliothek: Definitionsmoduln ohne korrespondierenden Implementierungsmodul sind *FOREIGEN* Definitionsmoduln und durch F: gekennzeichnet. Eine Ausnahme bildet der Modul **M2rts.def** (siehe Anhang G), der eine Zwischenstellung zwischen Übersetzer und Bibliothek einnimmt und als Interface zum Laufzeitsystem einer Applikation dient.

```
 Zeilen    Worte   Modulname        Kurzbeschreibung
-------------------------------------------------------------
   33 L    183 W   ASCII.def        ASCII-Codes
   73 L    229 W   conio.def        F: low level console I/O
   68 L    228 W   Conersions.def   Konvertierungen
   56 L    185 W   convert.def      F: low level Konvertierung
   36 L    109 W   Coroutines.def   F: Coroutinen
   64 L    228 W   FileSystem.def   File I/O
  204 L    696 W   InOut.def        Terminal I/O
   90 L    311 W   io.def           F: low level I/O
   83 L    257 W   Limits.def       Std.-Typenwertebereiche
   39 L    124 W   M2rts.def        F: run-time-system
  145 L    420 W   malloc.def       F: Speicherallozierung
  160 L    541 W   NumberConversion.def   Konvertierungen
   52 L    151 W   RealConversions.def    Konvertierungen
   74 L    254 W   RealInOut.def    Real I/O
  234 L    857 W   stdio.def        F: Stream I/O
  212 L    909 W   stdlib.def       F: Std. C-Library
   10 L     29 W   Storage.def      Speicherallozierung
  135 L    444 W   Strings.def      String Handling
   56 L    133 W   Terminal.def     low level Terminal I/O
-------------------------------------------------------------
 1824 L   6288 W   total

  Zeilen   Worte   Modulname        Kurzbeschreibung
-------------------------------------------------------------
    5 L      9 W   ASCII.mod        siehe Definitionsmodul
   39 L    164 W   Conversions.mod  "
  213 L    597 W   FileSystem.mod   "
  353 L   1228 W   InOut.mod        "
  485 L    905 W   Limits.mod       "
  201 L    671 W   NumberConversion.mod   "
  130 L    461 W   RealConversions.mod    "
   91 L    280 W   RealInOut.mod    "
   29 L     70 W   Storage.mod      "
  175 L    617 W   Strings.mod      "
  194 L    607 W   Terminal.mod     "
-------------------------------------------------------------
 1915 L   5609 W   total
```

Anhang G: Modula-2-C Laufzeitunterstützung

Listing des Header-Files des Modula-2-C Laufzeitsystems (run-time-system).

```
/*************************************************************/
/*                          id: M2rts.h                      */
/*************************************************************/

#define BOOLEAN         unsigned short int
#define CHAR            unsigned char

#define SHORTCARD       unsigned short int
#define CARDINAL        unsigned int
#define LONGCARD        unsigned long int

#define SHORTINT        short int
#define INTEGER         int
#define LONGINT         long int

#define SHORTREAL       float
#define REAL            double
#define LONGREAL        long double

#define _SHORTINTCARD   SHORTINT
#define _INTCARD        INTEGER
#define _LONGINTCARD    LONGINT

#define BITSET          unsigned long int

#define BYTE            unsigned short int
#define WORD            unsigned int
#define LONGWORD        unsigned long int

#define ADDRESS         void *
#define NIL             (void *) 0

#define FALSE           (unsigned short int) 0
#define TRUE            (unsigned short int) 1

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

#ifndef _MODULA_RTS

extern void          _COPY();
extern BITSET        _MASK();
extern WORD *        _ALLOCATE();
extern void          _DEALLOCATE();
extern void          _HALT();
```

```
extern CHAR          _CAP();
extern CHAR *        _CHAR2STR();

#endif

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

#define _MASK(x)        __MASK((BITSET) (x))

#define INC(x,i)        x += i
#define DEC(x,i)        x -= i
#define NEW(x)          x
#define DISPOSE(x)      x
#define INCL(s,x)       (s = s | _MASK(x))
#define EXCL(s,x)       (s = s & ~_MASK(x))
#define HALT()          _HALT()
#define INLINE(x)       x
#define ABS(x)          (((x) > 0) ? (x) : (-x))
#define ADR(x)          &x
#define CAP(x)          _CAP(x)
#define CHR(x)          x
#define FLOAT(x)        ((float) x)
#define HIGH(x)         x
#define MAX(x)          x
#define MIN(x)          x
#define ODD(x)          ((x % 2) == 0)
#define ORD(x)          x
#define SIZE(x)         sizeof(x)
#define TRUNC(x)        ((unsigned int) x)
#define TSIZE(x)        sizeof(x)
#define VAL(x,y)        y

/*************************************************************/
```

Foreign-Definitionsmodul des Modula-2-C Laufzeitsystems:

```
(*************************************************************)
(*                          id: M2rts.def                    *)
(*************************************************************)

FOREIGN DEFINITION MODULE M2rts;

FROM SYSTEM IMPORT ADDRESS;

PROCEDURE _ALLOCATE(size : CARDINAL) : ADDRESS;

PROCEDURE _DEALLOCATE(ptr : ADDRESS);

PROCEDURE _Get_argc() : INTEGER;

PROCEDURE _Get_argv(i : CARDINAL; VAR arg : ARRAY OF CHAR) : BOOLEAN;

PROCEDURE _Get_envp(i : CARDINAL; VAR env : ARRAY OF CHAR) : BOOLEAN;

END M2rts.
```

Modula-2-C Laufzeitsystem (C-Hauptmodul):

```
/*****************************************************************************/
/*                            id: M2rts.c                                    */
/*****************************************************************************/

#define  _MODULA_RTS

#include <malloc.h>
#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include "M2rts.h"

extern void _START_MODULA();

static CHAR *  chptr;
static CHAR    glob_str[2];
static INTEGER glob_argc;
static CHAR ** glob_argv;
static CHAR ** glob_envp;

/*****************************************************************************/
/*                            id: _CAP()                                     */
/*****************************************************************************/

CHAR _CAP(ch)
  CHAR ch;
{
  return toupper(ch);
}

/*****************************************************************************/
/*                            id: _HALT()                                    */
/*****************************************************************************/

void _HALT()
{
  printf("HALT called\n");
  exit(-1);
}

/*****************************************************************************/
/*                            id: _ALLOCATE()                                */
/*****************************************************************************/

WORD * _ALLOCATE(size)
  CARDINAL size;
```

```
{
  WORD * ptr;

  ptr = (WORD *) malloc(size);
  if (ptr == NIL) {
    printf("ERROR: _ALLOCATE - heap overflow\n");
    _HALT();
  }
  return ptr;
}

/*****************************************************************************/
/*                            id: _DEALLOCATE()                              */
/*****************************************************************************/

void _DEALLOCATE(ptr)
  WORD * ptr;
{
  if (ptr != NIL) free((void *) ptr);
  else printf("ERROR: _DEALLOCATE - pointer was NIL\n");
}

/*****************************************************************************/
/*                            id: _COPY()                                    */
/*****************************************************************************/

void _COPY(target, source, t_size, s_size, string)
  CHAR *    target;
  CHAR *    source;
  CARDINAL  t_size;
  CARDINAL  s_size;
  SHORTCARD string;
{
  if ((t_size > s_size) && string) s_size++;   /* copy '\0' also */

  if (t_size >= s_size) {
    strncpy(target, source, s_size);
  } else {
    printf("Run-time-error: illegal array assignment\n");
    _HALT();
  }
}

/*****************************************************************************/
/*                            id: __MASK()                                   */
/*****************************************************************************/

BITSET __MASK(x)
```

```
  BITSET x;
{
  BITSET i, result = 1;

  for (i = 1; i <= x; i++) {
    result <<= 1;
  };
  return result;
}

/*****************************************************************************/
/*                             id: _ARRAYCHK()                               */
/*****************************************************************************/

CARDINAL _ARRAYCHK(index, low, high)
  INTEGER index, low, high;
{
  if ((low <= high) && (low <= index) && (index <= high)) {
    if (low < 0) {
      index = index + (-low);
    } else if (low > 0) {
      index = index - low;
    }
  } else {
    printf("Run-time-error: array index out of range\n");
    _HALT();
  }
  return index;
}

/*****************************************************************************/
/*                             id: _CHAR2STR()                               */
/*****************************************************************************/

CHAR * _CHAR2STR(ch)
  CHAR ch;
{
  glob_str[0] = ch;
  glob_str[1] = '\0';
  chptr = glob_str;
  return chptr;
}

/*****************************************************************************/
/*                             id: _Get_argc()                               */
/*****************************************************************************/

INTEGER _Get_argc()
```

```
{
  return glob_argc;
}

/*************************************************************************/
/*                            id: _Get_argv()                            */
/*************************************************************************/

BOOLEAN _Get_argv(i, arg_i)
  CARDINAL i;
  CHAR * arg_i;
{
  if (i >= _Get_argc()) {
    strcpy(arg_i, "\0");
    return FALSE;
  } else {
    strncpy(arg_i, glob_argv[i], strlen(glob_argv[i])+1);
    return TRUE;
  }
}

/*************************************************************************/
/*                            id: _Get_envp()                            */
/*************************************************************************/

BOOLEAN _Get_envp(i, env_i)
  CARDINAL i;
  CHAR * env_i;
{
  if (glob_envp[i] == NULL) {
    strcpy(env_i, "\0");
    return FALSE;
  } else {
    strncpy(env_i, glob_envp[i], strlen(glob_envp[i])+1);
    return TRUE;
  }
}

/*************************************************************************/
/*                            id: main()                                 */
/*************************************************************************/

void main(argc, argv, envp)
  INTEGER argc;
  CHAR *  argv[];
  CHAR *  envp[];
{
   glob_argc = argc;
```

```
   glob_argv = argv;
   glob_envp = envp;

  _START_MODULA();
}

/*****************************************************************************/
```

Anhang H: Implementierung von Coroutinen

Der folgende Abschnitt zeigt den Prototyp einer Implementierung von Coroutinen in der Modula-2 Bibliothek. Der Code wurde für einen PC unter MS-DOS entwickelt und ist weitgehend systemspezifisch.

```
(*************************************************************************)
(*                           id: Coroutines.def                          *)
(*************************************************************************)

FOREIGN DEFINITION MODULE Coroutines;

FROM SYSTEM IMPORT ADDRESS;

TYPE
  PROCESS;

PROCEDURE NEWPROCESS(    P : PROC; a : ADDRESS; n : CARDINAL;
                     VAR p : PROCESS);

PROCEDURE TRANSFER(VAR from, to : PROCESS);

END Coroutines.
```

C-Modul für Coroutinen:

```
/*************************************************************************/
/*                           id: Coroutines.c                            */
/*************************************************************************/

#include <dos.h>
#include <malloc.h>
#include "setjmp.h"
#include "m2rts.h"

typedef struct {
  jmp_buf   state;            /* processor state */
} PROCESS;
unsigned trans_count = 0;     /* counter for TRANSFER() calls */
PROCESS  first_process;       /* descriptor for first process */


/*************************************************************************/
/*                           id: NEWPROCESS()                            */
/*************************************************************************/

void NEWPROCESS(proc, adr, n, process)
  void     (*proc)();
  ADDRESS  adr;
  CARDINAL n;
  PROCESS  **process;
{
  unsigned long stack;
  unsigned ss;

  stack = ((long) FP_SEG(adr) * (long) 0x10) + (long) FP_OFF(adr);
  while (!(stack % (long) 0x10)) {              /* compute stack segment */
    stack++;
  }
  ss = (unsigned) (stack >> 4);                    /* stack segment */
  (*process) = (PROCESS *) malloc(sizeof(PROCESS)); /* create PROCESS item */
  setjmp((*process)->state);                       /* init descriptor */
  (*process)->state[0].j_cs = FP_SEG(proc);        /* code segement */
  (*process)->state[0].j_ip = FP_OFF(proc);       /* instruction pointer */
  (*process)->state[0].j_ss = ss;                  /* stack segment */
  (*process)->state[0].j_sp = n;                   /* stack pointer */
}
```

```
/*************************************************************************/
/*                          id: TRANSFER()                               */
/*************************************************************************/

void TRANSFER(from, to)
  PROCESS **from, **to;
{
  if (*from == *to) return;
  if (!trans_count) {                /* first call of TRANSFER() ? */
    *from = &first_process;
    trans_count++;
  }
  if (!setjmp((*from)->state)) {     /* save state of current process */
    longjmp((*to)->state, 1);        /* context switch */
  }
}

/*************************************************************************/
```

Anhang I: Beispiel für die Übersetzung von Moduln in Prozeduren

In diesem Abschnitt wird ein Beispiel für die Übersetzung von Moduln in Prozeduren gezeigt. Da die Objekte, die in den Moduln deklariert sind, so lange "leben" müssen (man denke z.B. an einen *EXPORT* solcher Objekte in die Prozedurumgebung), wie die Prozedurumgebung, werden diese Deklarationen in diese Umgebung übertragen. Weiters ist dafür zu sorgen, daß die Moduln bei jeder Aktivierung der Prozedur neu initialisiert werden.

```
(*-----------------------  Modula-2  ------------------------*)

MODULE ProcMod;
VAR
  c : CARDINAL;
  PROCEDURE WithModule;
  VAR
    c : CARDINAL;
    MODULE InProc;
    VAR
      c : CARDINAL;
      MODULE InProcMod;
      VAR
        c : CARDINAL;
      BEGIN
        c := 3;
      END InProcMod;
    BEGIN
      c := 2;
    END InProc;
  BEGIN
    c := 1;
  END WithModule;
BEGIN
  c := 0;
END ProcMod.

/*-----------------------     C      ------------------------*/

#include "M2rts.h"

BOOLEAN  _KEY_ProcMod_M2 = FALSE;

/* MODULE */ void ProcMod_M2();

static CARDINAL c_ProcMod_M2;
static /* PROCEDURE */ void WithModule_ProcMod_M2();
typedef struct {
```

```
  CARDINAL c_M2;
  CARDINAL c_InProc_WithModule_ProcMod_M2;
  CARDINAL c_InProcMod_InProc_WithModule_ProcMod_M2;
} _G_WithModule_ProcMod_M2;
static _G_WithModule_ProcMod_M2 * _GP_WithModule_ProcMod_M2;

/* MODULE */ static void InProc_WithModule_ProcMod_M2();
/* MODULE */ static void InProcMod_InProc_WithModule_ProcMod_M2();

void _START_MODULA()
{
  ProcMod_M2();
}

/* MODULE */ void ProcMod_M2()
{
  if (_KEY_ProcMod_M2) return;
  _KEY_ProcMod_M2 = TRUE;
  c_ProcMod_M2=0;
}

static void WithModule_ProcMod_M2()
{
  _G_WithModule_ProcMod_M2 _L_WithModule_ProcMod_M2,
   * _LP_WithModule_ProcMod_M2 = _GP_WithModule_ProcMod_M2;

  _GP_WithModule_ProcMod_M2 = &_L_WithModule_ProcMod_M2;
  InProc_WithModule_ProcMod_M2();
  _GP_WithModule_ProcMod_M2->c_M2=1;
  _GP_WithModule_ProcMod_M2 = _LP_WithModule_ProcMod_M2;

}

static /* MODULE */ void InProc_WithModule_ProcMod_M2()
{
  InProcMod_InProc_WithModule_ProcMod_M2();
  _GP_WithModule_ProcMod_M2->c_InProc_WithModule_ProcMod_M2=2;
}

static /* MODULE */ void InProcMod_InProc_WithModule_ProcMod_M2()
{
  _GP_WithModule_ProcMod_M2->c_InProcMod_InProc_WithModule_ProcMod_M2=3;
}
```