ETH Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Diplomarbeit SS90
von
Stéphane Micheloud, Abt. IIIC

Oberon Compiler Back-End

für

Inmos Transputer

Zuständiger Professor: Prof. Niklaus Wirth
Betreuung: Cuno Pfister / Michael Franz

Ausgabe: 25.4.1990
Abgabe: 24.8.1990

Departement Informatik
Institut für Computersysteme
ETH Zentrum / 8092 Zürich

An meine Eltern und Grosseltern

Inhalt

Figuren

1. Einleitung

Am Institut für Computersysteme (Prof. Niklaus Wirth) ist ein portabler Oberon Compiler (OP2) entwickelt worden. Dieser wurde bereits auf mehrere Rechner mit Erfolg portiert. Dazu gehören die Ceres I, II und III (NS 32000-Familie), die IBM PC und PS/2 Modelle (Intel 80X86-Familie), die Apple Macintosh II (MC 680X0-Familie) und die Sun Sparcstation I (SPARC). Er wird auch bald auf der DECStation (MIPS R2000) verfügbar sein.

Der OP2 Compiler besteht aus zwei klar getrennten Teilen: dem Front-End und dem Back-End. Das erste ist auf alle Rechner gleich, da seine Aufgabe maschinenunabhängig ist. Nur das Back-End, das für die Speicher-Allozierung und die Code-Erzeugung zuständig ist, muss abgeändert werden. Dazu ist die entsprechende Runtime-Infrastruktur zu entwickeln.

Der im Rahmen dieser Arbeit zu entwickelnde Compiler ist ein Cross-Compiler. Er läuft auf der Ceres-Maschine, erzeugt aber Maschinencode für den Inmos Transputer ([Inmos 88]). Ein Transputerboard mit dem IMS T800 Prozessor und 1 MByte Speicher wurde zu diesem Zweck für die Ceres gebaut (siehe [Ingold 90]).

1.1. Analyse der Aufgabe

Diese Arbeit besteht aus den folgenden Aufgaben (siehe Aufgabenstellung im Anhang A):

- T800-Decoder
- T800-Compiler
- Runtime-Infrastruktur
- Testumgebung
- Bericht und Dokumentation

Die erste Aufgabe ist dem Kennenlernen des T800 Befehlssatzes gewidmet. Dabei ist ein Decoder für den T800 Transputer zu realisieren.

Das Handbuch *Compiler Writer's Guide* ([Inmos 88]) enthält nicht nur eine ausführliche Beschreibung des Befehlssatzes, sondern auch Richtlinien zur Uebersetzung hochsprachlicher Konstrukte.

Die zweite Aufgabe ist dem Kennenlernen des OP2 Compilers gewidmet. Als grösster Anteil der Arbeit ist ein Back-End für OP2 zu implementieren.

Eine Floppy-Disk mit dem OP2 Compiler für Ceres und der Source von OP2 in seiner "empty-Version" (Front-End mit Dump-Option, Back-End ohne Ceres-spezifische Teile) stellen zusammen mit dem *OP2 Technical Report* ([Crelier 90]) das Ausgangsmaterial dieser Arbeitsphase dar.

Die dritte Aufgabe befasst sich mit der Runtime-Infrastruktur (insbesondere mit der T800-Ceres Schnittstelle).

Die folgenden Punkte sollen hier auf möglichst einfache Weise gelöst werden.

- Das *Memory-Layout*
- Das Laden von Programmen und das Ausführen von Kommandos, auf dem Transputerboard, von der Ceres aus
- Das Linken von Modulen
- Die Behandlung von **HALT**(n), **NEW**(p) und **NIL**-Referenzen

Die vierte Aufgabe betrifft das Testen der Code-Erzeugung.

Die letzte Aufgabe ist der Dokumentation zusammen mit dem Schreiben dieses Berichts gewidmet. Eine kurze Benutzeranleitung soll die abzugebende Floppy-Disk begleiten.

1.2. Vorgehensweise

Für diese Arbeit wird der Zeitplan so vorgelegt, dass die ersten Versuche schon nach zwei Monaten stattfinden können. Fortschritte und Probleme werden einmal pro Woche zusammen mit dem zuständigen Assistenten besprochen.

In der ersten Arbeitsphase ist die Entscheidung früh gefallen, den Decoder zuerst auf einer Maschine zu implementieren, wo ein T800-Compiler schon vorhanden ist.

In der Fachgruppe für Systemtechnik von Prof. A. Kündig (Institut für technische Informatik und Kommunikationssysteme) ist vor einigen Jahren eine Transputerkarte für den Apple Macintosh II entwickelt worden. Der Autor selbst hat im Rahmen einer Semesterarbeit Erfahrungen mit dem *Transputer Development System* (TDS) - ebenfalls von Inmos entwickelt - gesammelt.

Mit dem T800-Decoder - und dem T800-Interpreter, beide mit MacMETH geschrieben - lässt sich der vom Occam Compiler erzeugte Code untersuchen. Dieser Einblick in die Code-Generierung von Occam bildet eine wertvolle Vorbereitung für die zweite Aufgabe.

Eine Ceres I Maschine (im Raum IFW E33) steht für die Implementation des Back-Ends zur Verfügung. Der OP2 Compiler wird zuerst genau untersucht, um den Rahmen der Arbeit klar zu definieren. Die Dump-Option von OP2 bietet zum Beispiel die Möglichkeit, das zu kompilierende Oberon Programm in seiner internen - vom Front-End aufgebauten - Darstellung zu analysieren. Dieses Hilfsmittel hat sich rasch als sehr nützlich erwiesen.

Folgende Teilaufgaben werden in dieser Phase durchgearbeitet: Speicherallozierung, Adressierung von Variablen, Felder und Parametern, Auswertung von Ausdrücken, Kontrollstrukturen, Prozeduraufrufe und externe Zugriffe. Das Back-End wird entsprechend abgeändert und die Code-Erzeugung konsequent nachgeprüft.

Im Gegensatz zum Back-End muss die Runtime-Infrastruktur vom *scratch* gestartet werden. Die Zielsetzung ist hier, eine einfache Testumgebung für das Testen und die Weiterentwicklung dieser Implementation zu realisieren. Bis zu einem gewissen Mass wird für die Realisierung das Oberon System von Ceres nachgebildet.

Für das Testen werden neben selbst geschriebenen Testprogrammen die Stanford Benchmarks ([Templ 90]) verwendet. Vergleiche zwischen der Ceres und der T800 Implementation sind im Abschnitt 6.4 zu finden.

2. Transputer Ueberblick

Der Begriff "Transputer" steht für Transistor Computer. Der europäische Hardware-Hersteller Inmos Ltd. (England) hat diesen Namen für seine Prozessoren gewählt. Sein Angebot besteht zur Zeit aus den T2 (16-bit), T4 (32-bit) und T8 (32-bit mit FPU) Prozessorfamilien (z.B. T801, T805, T414, T425, usw.). Die Transputer sind zusammen mit der Programmiersprache Occam entwickelt worden und werden vor allem in Multiprozessorsystemen eingesetzt.

Ein Transputer ist ein einfacher VLSI Bauteil mit Prozessor, Speicher (2 KBytes für den T414, 4 KBytes für den T800) und vier *communication links* (zwei für den T212) für eine direkte (serielle) Verbindung mit anderen Transputern. Seine Architektur bietet viele innovative Konzepte wie z.B. die Hardware-Prozessverwaltung und den Registersatz, der als Stack organisiert ist.

2.1. Systemarchitektur

Eine Adresse ist ein Datenwort, das ein Byte im Speicher identifiziert.

- Sie besteht aus zwei Teilen, einer Wortadresse und einem Byte-Selektor.
- Ihr Wert gehört zum Wertebereich vom Typ **`INTEGER`**, d.h. sie kann auch negativ sein.

Die Transputer der T4 und T8 Prozessorfamilien können bis 4 GBytes Speicher adressieren; die virtuelle Adressierung wird aber nicht unterstützt.

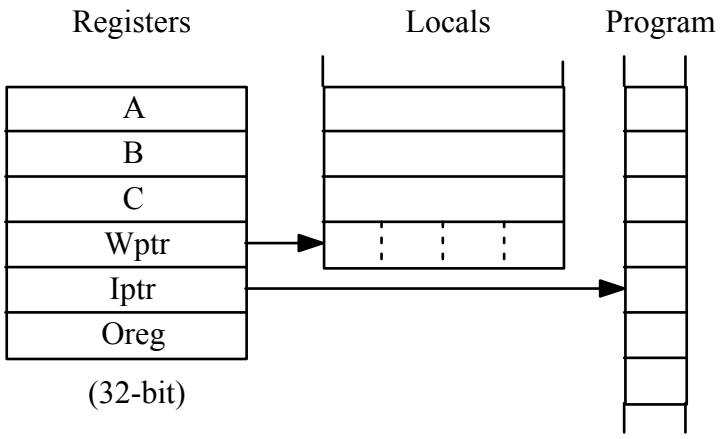
Der Transputer ist völlig *little-endian* , d.h. die *least significant* Information liegt immer an der niedrigsten Adresse. Dies gilt für Bits in Bytes, Bytes in Wörtern und Wörter im Speicher.

Der Prozessorzustand lässt sich durch die folgenden Register (jedes ist ein Wort lang) charakterisieren:

```
Iptr   pointer to next instruction to be executed
Wptr   contains pointer to current process workspace
Areg   evaluation stack (top of stack)
Breg   evaluation stack
Creg   evaluation stack
Oreg   operand register
```

Der Transputer besitzt einen relativ ungewöhnlichen Registersatz. So stehen keine der sonst oft vorhandenen *General Purpose Register* zur Verfügung. Statt dessen gibt es einen dreielementigen Datenstack bestehend aus den Registern `Areg`, `Breg` und `Creg`.

Diese Wahl stellt nach Inmos einen effektiven Kompromiss zwischen Codedichte und Komplexität der Implementation dar. Das Laden von mehr als drei Operanden auf den Stack wird durch keinen Hardware-Mechanismus geprüft; der Compiler ist dafür zuständig.



Figur 2-1: Registersatz.

Das `Wptr` Register - auf anderen Prozessoren auch *Stack Pointer* genannt - wird als Basisadresse für den Zugriff auf die lokalen Variablen verwendet. Der Byte-Selektor von `Wptr` soll immer Null sein.

Was den Befehlssatz betrifft ist der Transputer so konzipiert, dass Hochsprachen möglichst einfach und effizient übersetzt werden können. Die Anzahl Instruktionen für die Repräsentanten jeder Prozessorfamilie sind:

- T212 : 94 Instruktionen.
- T414 : 100 Instruktionen.
- T800 : 158 Instruktionen (davon 52 FPU-Instruktionen).

Zur Laufzeit können insgesamt sechzehn Instruktionen (fünfzehn für den T212) einen Fehler erzeugen, was durch ein einziges *Error Flag* ausgezeichnet wird. Das Setzen dieses Flags kann unterschiedliche Wirkungen haben, die durch besondere Instruktionen festgesetzt werden.

Der Befehlssatz des T800 Prozessors wird hier kurz beschrieben; mit Ausnahme der FPU besitzen die anderen Transputer ähnliche Charakteristiken.

2.2. Befehlssatz

Alle Instruktionen haben dasselbe Format. Der Befehlsatz ist damit von der Prozessorwortlänge (d.h. von der Anzahl Bytes in einem Wort) unabhängig.

Jede Instruktion ist ein Byte lang und besteht aus zwei 4-Bit Teilen.

- Die vier *most significant* Bits enthalten den Funktionscode.
- Die vier *least significant* Bits enthalten den Datenwert.

Mit dieser Aufteilung lassen sich sechzehn Funktionen darstellen, jede mit einem Datenwert von 0 bis 15.

- Zwölf davon, auch direkte Funktionen genannt, stehen für die wichtigsten Operationen, die beim Uebersetzen einer Hochsprache erzeugt werden (wie *jumps*, *calls* und Zugriffsinstruktionen).

- Eine davon, auch *operate function* genannt, bewirkt, dass ihr Operand als Operation-Code interpretiert wird (sogenannte indirekte Funktionen).
- Die zwei letzten Funktionen, auch *prefix functions* genannt, erlauben das Zusammensetzen von Operanden gewünschter Länge.

Durch die Benützung der *prefix functions* zusammen mit der *operate function* lässt sich der Befehlsatz fast beliebig erweitern !

Der Algorithmus zur Generierung eines konstanten Operands `val` für eine Funktion `op` kann durch die folgende rekursive Prozedur einfach formuliert werden.

```
PROCEDURE prefix(op, e: INTEGER);
    CONST nfix = 6H; pfix = 2H;
BEGIN
    IF e < 0 THEN prefix(nfix, -(e DIV 16)-1)
    ELSIF e >= 16 THEN prefix(pfix, e DIV 16)
    END;
    consume(16 * op + (e MOD 16))
END prefix;
```

wo `consume(..)` beispielsweise Code generiert. Zum Beispiel (**`ldc`** : "load constant") :

```
ldc         0       40H
ldc        15       4FH

ldc         1       41H
ldc        -1       60H 4FH

ldc       255       2FH 4FH
ldc      -255       6FH 41H
```

Alle Funktionen werden vom Prozessor gleich behandelt.

- Die vier *least significant* Bits der Instruktion werden zuerst in die vier *least significant* Bits des Registers `Oreg` geladen, dessen Wert als Operand benutzt wird.
- Die Instruktion wird ausgeführt.
- Der Register `Oreg` wird gelöscht, ausser für die *prefix functions* .

Die Ausführung der zwei *prefix functions* läuft wie folgt ab.

- Die `pfix` Instruktion schiebt den Inhalt von `Oreg` viermal nach links.
- Die `nfix` Instruktion arbeitet wie `pfix`, komplementiert jedoch zuerst noch den Inhalt von `Oreg`.

Ausdrücke werden mit Hilfe des Evaluationsstacks ausgewertet. Operationen mit zwei Operanden benutzen die Werte von `Breg` und `Areg`. Das Resultat liegt in `Areg` und der Inhalt von Creg ist *popped* in `Breg` (`Creg` ist dann undefiniert).

Da der Stack auf drei Elemente begrenzt ist, muss auf die richtige Ausführungsreihenfolge grosser Wert gelegt werden. Manchmal muss

natürlich auf temporäre Variablen zurückgegriffen werden, falls die Auswertung mehr als drei Stackelemente benötigt.

Ein Ausdruck kann aber praktisch immer von innen nach aussen ausgewertet werden; bei Ausdrücken, die nur binäre Operationen enthalten, reicht mit dieser Strategie ein Stackbereich von drei Elementen in den meisten Fällen aus.

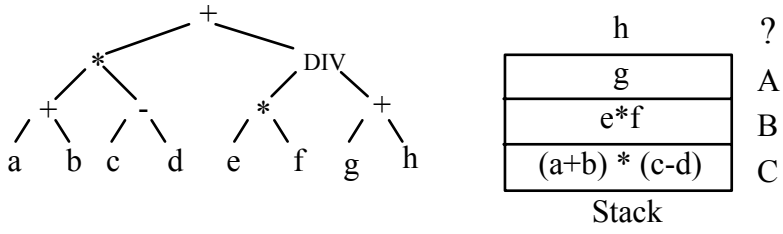
Sei der Ausdruck `e` durch einen Baum dargestellt, dann lässt sich die zu seiner Auswertung nötige Stackgrösse `depth(e)` folgendermassen bestimmen.

```
PROCEDURE depth(e: Node): INTEGER;
    VAR ed, ed1, ed2: INTEGER;
BEGIN
    IF "constant or variable" THEN ed:= 1
    ELSIF "function call" THEN ed:= "infinite"
    ELSE ed1:= depth(e^.left); ed2:= depth(e^.right);
        IF ed1 > ed2 THEN ed:= ed1
        ELSIF ed1 < ed2 THEN ed:= ed2
        ELSE ed:= ed1 + 1
        END
    END;
    RETURN ed
END depth;
```

Seien `e1` und `e2` zwei Ausdrücke, dann wird die Operation `e1 op e2` wie folgt ausgeführt (`"i1; i2; ..."` steht für eine Sequenz von Instruktionen) :

```
IF depth (e2) < stacksize THEN
    "e1; e2; op"
ELSIF depth (e1) < stacksize THEN
    "e2; e1"; IF NOT commute (op) THEN "rev" END; "op"
ELSE
    "e2; store temp; e1; load temp; op"
END;
```

Für die Auswertung von `(a + b) * (c - d) + (e * f) DIV (g + h)` wird zum Beispiel eine temporäre Variable benötigt (Figur 2-2). Temporäre Variablen werden auch dann benötigt, wenn Rückgabewerte von Funktionen in einem Ausdruck weiterverarbeitet werden müssen.



Figur 2-2: Auswertung mit Stacküberlauf.

Hier sind die vom TOP2 Compiler verwendeten T800-Instruktionen aufgelistet (siehe [Inmos 88]).

Abbreviation	Cycles		
ldc	1		load constant
mint	1		load most negative integer
adc	1	E	add constant
ldl	2		load local variable (relative to Wptr)
stl	1		store local variable
ldlp	1		load local pointer
ldnl	2		load non local variable (relative to Areg)
stnl	2		store non local variable
ldnlp	1		load pointer to non local variable
lb	5		load byte (relative to Areg)
sb	4		store byte
rev	1		reverse the contents of Areg and Breg
dup	1		duplicate top of evaluation stack
add	1	E	addition (Areg = Breg + Areg)
sub	1	E	subtraction
mul	38	E	multiplication
div	42	E	division
rem	37	E	remainder
sum	1		addition (Areg = Breg +unchecked Areg)
diff	1		subtraction
prod	b+4		multiplication (b = highest bit set in Areg)
and	1		bitwise and (Areg = Breg AND Areg)
or	1		bitwise or
xor	1		bitwise exclusive or
not	1		bitwise not
shl	n+2		shift left (n = #bits of a shift)
shr	n+2		shift right
lshl	n+3		long shift left
lshr	n+3		long shift right
ldpi	2		load pointer to instruction
mint	1		load most negative integer
bsub	1		byte subscript
wsub	2		word subscript
wsubdb	3		double word subscript
csub0	1	E	check subscript from 0
seterr	1	E	set error
move	2w+8		move message (w = #words)
in	2w+18		input message (communication procedes)
out	2w+20		output message

```
    xword   4         extend to word
    cword   5    E    check word
    xdble   2         extend to double
    csngl   3    E    check single

    eqc     2         equal to constant
    gt      2         greater than (Areg = Breg > Areg)
    j       3         jump
    cj      2/4  conditional jump (on false) (4 cycles if taken)
    call    7         call subroutine (relative to Wptr)
    gcall   4         general call (absolute)
    ajw     1         adjust workspace
    gajw    2         general adjust workspace
    ret     5         return
```

Für einen mit 20 MHz getakten Prozessor dauert ein Zyklus 50 Nanosekunden. Da die Befehle des Transputers durchschnittlich zwei Taktzyklen benötigen, liegt die Rechenleistung bei etwa 10 MIPS (*millions of instructions per second*).

Die mit E bezeichneten Instruktionen können das *Error Flag* setzen.

2.3. FPU-Befehlssatz

Im Vergleich zum T414 Prozessor ist der T800 Befehlssatz mit zusätzlichen Operationen erweitert worden, um die Gleitkomma-Arithmetik nach dem ANSI/IEEE 754-1985 Standard zu unterstützen. Diese Instruktionen werden vom Transputer so behandelt, als ob sie vom Hauptprozessor ausgeführt würden. In der Wirklichkeit werden sie aber von der FPU (*floating-point unit*) ausgeführt; beide Prozessoren arbeiten nämlich gleichzeitig.

In der FPU sind drei Register vorhanden, die ebenfalls als Stack organisiert sind. `FAreg`, `FBreg` und `FCreg` - wie sie auch gekennzeichnet werden - können reelle Zahlen im 32-bit (*single precision*) oder 64-bit (*double precision*) Format enthalten. Mit jedem Register ist ein internes *flag* assoziert, das die Präzision des Inhalts angibt.

Hier sind die vom TOP2 Compiler verwendeten FPU-Instruktionen aufgelistet.

```
Abbreviation  Cycles
    fpdup        1     duplicate top of fp stack
    fprev        1     reverse the contents of FAreg and FBreg

    fpldnlsn     3     load non local (single)
    fpldnldb     5     load non local (double)
    fpldzerosn   1     load zero single
    dpldzerodb   1     load zero double

    fpstnlsn     3     store non local (single)
    fpstnldb     5     store non local (double)

    fpadd        7   F fp addition
    fpsub        7   F fp subtraction
    fpmul       11   F fp multiplication (20 cycles for double)
    fpdiv       17   F fp division (32 cycles for double)
```

```
fpuabs        2   F  fp absolute value

fpgt          5   F  fp greater than
fpeq          4   F  fp equality
fpordered     4      fp orderability

fpchkerr      2      check fp error

fpur32tor64   3   F  convert from single to double length
fpur64tor32   5   F  convert from double to single length

fpint         6   F  round to 'floating integer'
fpstnli32     4      store 'floating integer' as integer
fprtoi32     10   F  round and check to INT32 range
```

Mit seiner integrierten FPU weist die 20 Mhz Version des T800 Prozessors eine Rechenleistung von 1.5 MFLOPS (*millions of floating-point operations per second*) auf.

Die mit F bezeichneten Instruktionen können das *FP Error Flag* setzen. Mit der Instruction fpchkerr wird dem *Error Flag* der Wert "*Error Flag* OR *FP Error Flag* " zugewiesen.

3. Uebersetzen von Oberon

Diese Implementation von Oberon für den T800 Transputer beachtet die meisten Richtlinien, die im Handbuch *compiler writer's guide* ([Inmos 88]) beschrieben sind. Hier werden nur wichtige Grundlagen und Unterschiede für das Uebersetzen von Oberon diskutiert.

3.1. Adressenbildung und Datenzugriffe

Adressenbildung

Folgende Befehle stehen für die Adressenbildung zur Verfügung:

```
ldpi      load pointer to instruction
bsub      byte subscript
wsub      word subscript
wsubdb    double word subscript
```

Die Adresse einer Datenstruktur, die sich im lokalen Workspace befindet, lässt sich mit dem Befehl **`ldlp`** ermitteln.

Die Adresse eines Speicherelements in einem Programm kann durch **`ldpi`** ermittelt werden (sogenannte PC-relative Adressierung). Damit lassen sich vollständig relozierbare Programme schreiben.

Die Befehle **`bsub`**, **`wsub`** und **`wsubdb`** interpretieren den Inhalt des Registers `Areg` als Zeiger auf den Beginn einer Datenstruktur.

Datenzugriffe

Die lokalen Zugriffsinstruktionen ldl, stl und ldlp arbeiten wortorientiert. Sie greifen auf Adressen relativ zum Register `Wptr` zu. Um nun auch einen Zugriff auf globale Variablen und auf Strukturen zu ermöglichen werden Befehle benötigt, die auf jede beliebige Adresse im Speicher zugreifen können.

Analog zu den lokalen Adressierungsarten stehen deshalb die sogenannten *non-local* -Befehle, die relativ zu `Areg` adressieren, zur Verfügung:

```
ldnl      load non local
stnl      store non local
ldnlp     load non local pointer
lb        load byte
sb        store byte
```

Angenommen, das Register `Areg` enthält den Wert `$8000`, so wird durch den Befehl `ldnl $100` der Inhalt der Speicherzelle `$8100` in `Areg` geladen. Dabei wird der Wert nicht auf den Evaluationsstack gebracht, sondern mit dem alten Inhalt von `Areg` ausgetauscht. Das bedeutet, dass die Basisadresse des Zugriffs bei dieser Operationen verloren geht.

Die zwei Befehle **lb** und **sb** geschehen byteorientiert und werden sowohl für lokale als auch für globale Zugriffe verwendet. Liegt der Wert auf dem Stack, dann muss der Inhalt von Areg zuerst zu einem Wort "expandiert" werden bevor irgendwelche Operationen auf dem Wert ausgeführt werden.

3.2. Vergleichs- und Sprungbefehle

Vergleiche und vergleichsabhängige Sprünge können mit den Befehlen

```
eqc     equal to constant
gt      greater than
```

zusammen mit

```
j       jump
cj      conditional jump
```

erreicht werden.

Die **eqc** Instruktion überschreibt das Register Areg mit dem Wahrheitswert 0 oder 1, abhängig vom Ausgang des Vergleichs von Areg mit dem konstanten Operanden. Entsprechend wird Areg nach dem Befehl **gt** gesetzt. In diesem Fall wird jedoch Breg mit Areg verglichen.

Der Sprungbefehl **j** berechnet das Sprungziel durch Addition seines Operanden mit dem PC-Wert der nächsten Instruktion. Der bedingte Sprungbefehl **cj** wirkt wie eine *jump on false* Instruktion; wird der Sprung ausgeführt, dann bleibt der Inhalt von Areg - d.h. Null -erhalten, andernfalls geht er verloren.

Das bedeutet, dass auch bei Sprungbefehlen konsequent relative Adressierung genutzt wird.

Boolesche Ausdrücke

Die **cj** Instruktion erlaubt die sogenannte "*short-circuit* " Auswertung von booleschen Ausdrücken. Die folgende Tabelle zeigt die Korrespondanz zwischen Oberon Ausdrücken und Instruktionen. X und Y sind Ausdrücke, und K ist ein konstanter Ausdruck.

```
TRUE      = ldc 1
FALSE     = ldc 0
NOT X     = ¬ (X)
X OR Y    = ¬ ((¬ X); cj L; (¬ Y); L:)
X & Y     = X; cj L; Y; L:
X = Y     = X; Y; diff; eqc 0
X <> Y    = ¬ (X = Y)
X = K     = X; eqc K
X # K     = ¬ (X = K)
X > Y     = X; Y; gt
X < Y     = Y; X; gt
X >= Y    = ¬ (X < Y)
X <= Y    = ¬ (X > Y)
```

wo ¬ (X) = (X; **eqc** 0).

Damit Sprünge möglichst früh ausgeführt werden können, sollte man folgende Regeln beachten:

```
¬ (X & Y)       = ¬ (X) OR ¬ (Y)
¬ (X OR Y)      = ¬ (X) & ¬ (Y)
(X OR Y); cj L= ¬ (X); cj M; Y; cj L; M:
(X & Y); cj L = X; cj L; Y; cj L
X = Y; cj L    = X; Y; diff; cj L
X = 0; cj L    = X; cj L
```

Kontrollstrukturen

Die Kontrollstrukturen von Oberon lassen sich in drei Typen von Konstrukten unterteilen, die unmittelbar durch einfache Code-Sequenzen übersetzt werden.

1) Auswahl-Konstrukte : **IF**- und **CASE**-Anweisungen

```
IF expr0 THEN statseq0                  expr0; cj ELSIFi; statseq0
ELSIF expri THEN statseqi} ELSIFi:      expri; cj ELSE; statseqi
[ELSE statseqF]            ELSE:        statseqF
END;                       END:

CASE expr OF                            expr; ldc low; diff
                                        dup; ldc high-low
                                        gt; eqc 0; cj ELSE
                                        dup; ldc -1; gt; cj ELSE
                                        ldc 3; prod; ldc 2
                                        ldpi; bsub; gcall
  casei { "|" casei }                   j CASEi
[ELSE statseqF]            ELSE:        statseqF; j END
CASEi:                                  statseqi
END;                       END:
```

2) Schleifen-Konstrukte : **WHILE**-, **REPEAT**- und **LOOP**-Anweisungen

```
WHILE expr DO              WHILE:   expr; cj END
  statseq                           statseq; j WHILE
END;                       END:

REPEAT statseq             REPEAT:  statseq
UNTIL expr;                         expr; cj REPEAT

LOOP statseq END;          LOOP:    statseq; j LOOP
```

3) Sprung-Konstrukte : **EXIT**- und **RETURN**-Anweisungen

```
LOOP statseq0;             LOOP:    statseq0;
  IF expr THEN EXIT END;            expr; cj L; j END
  statseq1                 L:       statseq1; j LOOP
END;                       END:

PROCEDURE P;
BEGIN statseq0;                     statseq0;
  IF expr THEN RETURN END;          expr; cj L; j END
  statseq1                 L:       statseq1
END P;                     END:
```

```
PROCEDURE Q(): INTEGER;
BEGIN RETURN expr                    expr; j END
   (* function trap *)               ldc 17; seterr
END Q;                          END:
```

3.3. Prozeduren

Die folgenden Befehle dienen zur Implementation von Prozeduraufrufen.

```
call    call subroutine
gcall   general call
ajw     adjust workspace
gajw    general adjust workspace
ret     return
```

Die **ajw** Instruktion erhöht das Register Wptr um die im Operand angegebene Anzahl von Wörtern. Wptr wird durch eine positive Zahl erhöht, und durch eine negative Zahl erniedrigt.

Die **call** Instruktion erniedrigt das Register Wptr implizit um vier Wörter. Ihr Operand wird zum aktuellen Iptr addiert. Das ergibt einen Sprung in die Unterroutine.

Die Instruktionen **gajw** und **gcall** laden die Register Wptr bzw. Iptr mit dem Inhalt von Areg; ihr alter Wert muss also meistens noch gerettet werden.

Die **ret** Instruktion setzt Iptr auf den Wert, der in die Adresse Wptr+0 gerettet wurde. Danach wird Wptr wieder um vier Wörter erhöht; der Inhalt des Evalutationstacks sowie dieser vier Speicherwörter bleibt dabei erhalten. Der Wert einer Funktion wird z.B. einfach in Areg zurückgegeben.

Mit **ajw** und **gajw** kann Speicherplatz für die lokalen Variablen sowie für die aktuellen Parameter reserviert werden. Vor dem Verlassen einer Prozedur wird dieser Platz wieder freigegeben.

Prozeduren in Oberon können durch den Prozedurnamen selbst oder über eine Prozedur-Variable aufgerufen werden. Prozeduraufrufe lassen sich also durch die Instruktionen call und **gcall** einfach übersetzen. Bei externen Aufrufen wird ihren Operanden erst beim Laden des Programms ein endgültiger Wert zugewiesen.

Diese zwei Fälle - direkter Prozeduraufruf und Aufruf über eine Prozedur-Variable - werden wie folgt behandelt:

```
VAR p: PROCEDURE;

PROCEDURE P;              ajw  -4    (entry address for p call)
   VAR x, y: INTEGER;     stl   0
BEGIN                     ajw  -2    (entry address for P call)
   (* ... *)              ...
END P;                    ret

BEGIN
```

```
  P;                        call  P
  (* p:= P; *)  p           ldc   @P    (absolute address of P)
                            adc   -3
                            gcall
END ...
```

Bei Prozedur-Variablen müssen also explizit 4 Wörter auf dem Stack alloziert werden; ebenso wird die Rücksprungadresse an die Adresse `Wptr+0` gerettet.

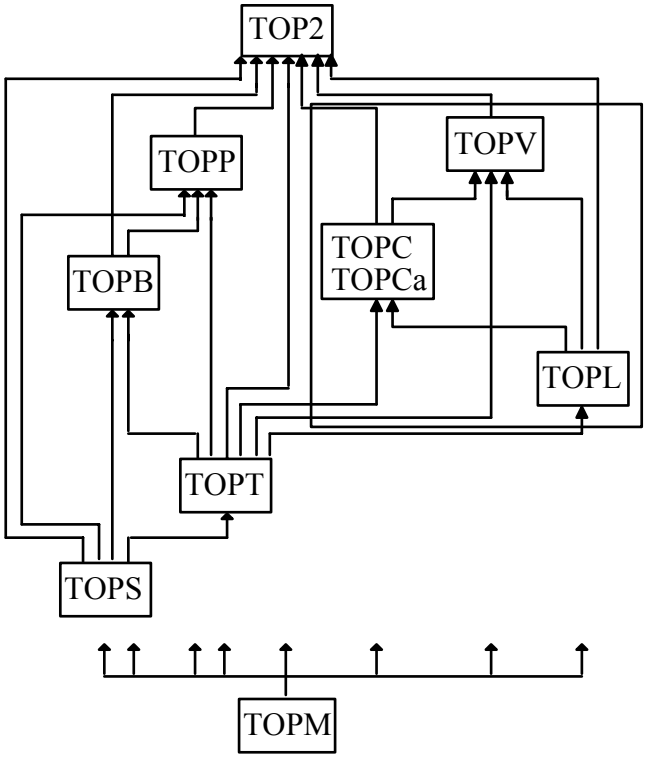
4. Back-End Beschreibung

Für diese Implementation werden die zehn Module von OP2 mit dem Prefix "T" für Transputer umbenannt.

Der TOP2 Compiler besteht aus folgenden Teilen:

- Der Hauptmodul TOP2 enthält das Kommando Compile.
- Die Module TOPP, TOPB, TOPT und TOPS bilden das Front-End.
- Die Module TOPV, TOPC, TOPCa und TOPL bilden das Back-End.
- Der Modul TOPM fasst Dienstroutinen zusammen, wie E/A-Prozeduren.

Die Modul-Struktur sieht wie folgt aus.



Figur 4-1: Modul-Struktur vom TOP2 Compiler.

In [Crelier 90] (Abschnitt 4.5) sind die zwei Phasen des Back-Ends ausführlich beschrieben, nämlich die Speicherallozierung und die Code-Erzeugung. Ihre praktische Realisierung wird in diesem Abschnitt präsentiert.

Drei Aenderungen in der Modularisierung wurden im Rahmen dieser Arbeit vorgenommen:

- Die Standardprozeduren von Oberon (CAP, ODD, LEN, COPY, ...) sind in TOPC zusammengefasst worden (bisher in TOPC und TOPCa aufgeteilt).
- Die elf Prozeduren (ADR, BIT, LSH, ROT,...) des Moduls SYSTEM sind in TOPCa zusammengefasst worden (bisher in TOPC und TOPCa aufgeteilt).

- Die sechs Prozeduren für die Sprungbehandlung (inklusiv CASE) sind in TOPL zusammengefasst worden (diese werden zum Teil in TOPC und TOPCa benutzt).

4.1. Speicher-Allozierung

Speicherplatz wird nicht nur für Variablen, sondern auch für String- und Real-Konstanten und für Hilfsvariablen (*type anchors* und *scope anchors*) alloziert. Temporäre Variablen werden übrigens auf besondere Art behandelt.

Die *type anchors* - zusammen mit den entsprechenden *type descriptors* - werden in Oberon für die Type-Tests/Guards gebraucht. In [Pfister 90] werden zwei mögliche Implementationen für die *type descriptors* besprochen; auf ihre Realisierung wird jedoch im Rahmen dieser Arbeit verzichtet (u.a. für den *bootstrap* -Prozess nicht notwendig). Gegenwärtig wird von den Type-Tests/Guards ein **NIL**-Trap ausgelöst.

Für Zugriffe auf sogenannte *intermediate variables* stellen die *scope anchors* eine interessante Alternative zu den traditionellen *static link* und *display area* Methoden dar ([Heeb 90]). Diese Zeiger werden einfach als globale Variablen alloziert und enthalten die *stack frame* Adresse der entsprechenden "*intermediate*" Prozedur.

Basistypen

In dieser Implementation von Oberon werden die Basistypen intern wie folgt dargestellt (ihre interne Darstellung wird im Modul TOPM festgelegt):

Datentyp	interne Darstellung
SHORTINT	1 Byte (mit Vorzeichen)
CHAR, **BYTE**	1 Byte
BOOLEAN	1 Byte (**FALSE** = 0, **TRUE** = 1)
INTEGER, **LONGINT**	4 Bytes (mit Vorzeichen)
SET	4 Bytes ({0} = 1)
POINTER-Typ	4 Bytes (**NIL** = 0)
Prozedur-Typ	4 Bytes
REAL	4 Bytes IEEE-Format (754-1985)
LONGREAL	8 Bytes IEEE-Format (754-1985)

Auf dem Transputer wird byte- oder wortweise auf den Speicher zugegriffen; Zugriffe auf Daten, die 2 Bytes gross sind - wie es für den Typ **INTEGER** oft der Fall ist -, sind damit nicht unterstützt. Aus diesem Grund werden die Typen **INTEGER** und **LONGINT** intern gleich dargestellt.

Eine Variante dazu wäre die folgende interne Darstellung: 4 Bytes für den Typ **INTEGER** und 8 Bytes für den Typ **LONGINT**. Diese Lösung wird zwar vom Transputer unterstützt, ihre Realisierung führt aber zusammen mit dem Evaluationsstack zu praktischen Problemen.

Der Transputer und der NS32X32 Prozessor von Ceres haben folgende zwei Charakteristiken gemeinsam: einerseits sind sie *little-endian*, anderseits unterstützen sie beide die Gleitkomma-Arithmetik nach dem IEEE-754

Standard. Die Allozierung von reellen Konstanten in den *constant frame* und die Implementierung des Moduls `Terminal` (siehe Abschnitt 6-2) werden damit sehr erleichtert.

Speicherallozierung

Für die Speicherallozierung unterscheidet man folgende Objektklassen:

- Felder in `RECORD`-Typen/Variablen
- globale Variablen
- lokale Variablen
- Parameter in **`PROCEDURE`**-Typen/Variablen oder in Prozedurkopfen

Globale und lokale Variablen werden nach der gleichen Strategie wie Felder von Rekordtypen alloziert. Im Einzelnen werden folgende Regeln angewendet:

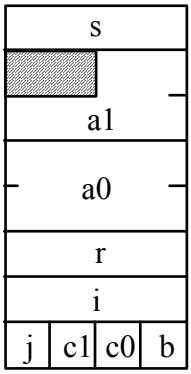
- Ueberschreitet die Feld-(Variablen-)Grösse ein Byte, dann wird das Feld, respektiv die Variable, immer an eine Wortadresse alloziert.
- Sonst wird das Feld, respektiv die Variable, an die erste Adresse der grössten Speicherlücke (es können nämlich mehrere sein) alloziert.
- Den globalen Variablen werden negative Adressen, den lokalen Variablen (und den Feldern) positive Adressen zugewiesen.

Die Prozeduren `AllocFld` (für Felder und lokale Variablen) und `AllocVar` (für globale Variablen) vom Modul `TOPV` sind hier zusammen mit einem Beispiel angegeben.

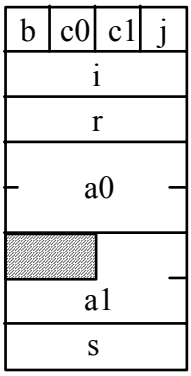
```
    PROCEDURE IncAdr(VAR adr: LONGINT; s: LONGINT);
    BEGIN IF adr <= MaxAdr - s THEN INC(adr, s) ELSE (*error*) END
    END IncAdr;


    PROCEDURE AllocFld(VAR offset, this, mark, n: LONGINT; size:
LONGINT);
    BEGIN (* size = #bytes *)
      IF (size = 1) & (n > 0) THEN
        this := mark; INC(mark); DEC(n)
      ELSE
        IncAdr(offset, (-offset) MOD 4);
        this:= offset; IncAdr(offset, size);
        n1:= (-offset) MOD 4;
        IF n1 >= n THEN mark:= offset; n:= n1 END
      END
    END AllocFld;


    PROCEDURE AllocVar(VAR offset, this, mark, n: LONGINT; size:
LONGINT);
      VAR align: LONGINT;
    BEGIN
      AllocFld(offset, this, mark, n, size);
      this:= -this - size; align:= (-size) MOD 4;
      IF (size > 1) & (align > 0) THEN DEC(this, align) END
    END AllocVar;
```

Beispiel

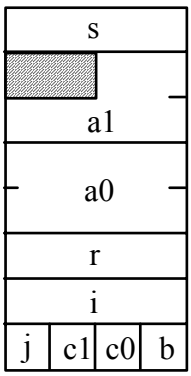
```
MODULE MemoryAllocation;

 TYPE Rec = RECORD (* fields *)
         b: BYTE;
         i: INTEGER;
         c0, c1: CHAR;
         r: REAL;
         a0: ARRAY 8 OF CHAR;
         j: SHORTINT;
         a1: ARRAY 6 OF CHAR;
         s: SET
        END;

 VAR (* global variables *)
   b: BYTE;
   i: INTEGER;
   c0, c1: CHAR;
   r: REAL;
   a0: ARRAY 8 OF CHAR;
   j: SHORTINT;
   a1: ARRAY 6 OF CHAR;
   s: SET;

 PROCEDURE P;
  VAR (* local variables *)
    b: BYTE;
    i: INTEGER;
    c0, c1: CHAR;
    r: REAL;
    a0: ARRAY 8 OF CHAR;
    j: SHORTINT;
    a1: ARRAY 6 OF CHAR;
    s: SET
 END P;

END MemoryAllocation.
```



off = 28
off = 20
off = 12
off = 8
off = 4
off = 3, 2, 1, 0



adr = -1, -2, -3, -4
adr = -8
adr = -12
adr = -20
adr = -28
adr = -32



adr = 28
adr = 20
adr = 12
adr = 8
adr = 4
adr = 3, 2, 1, 0

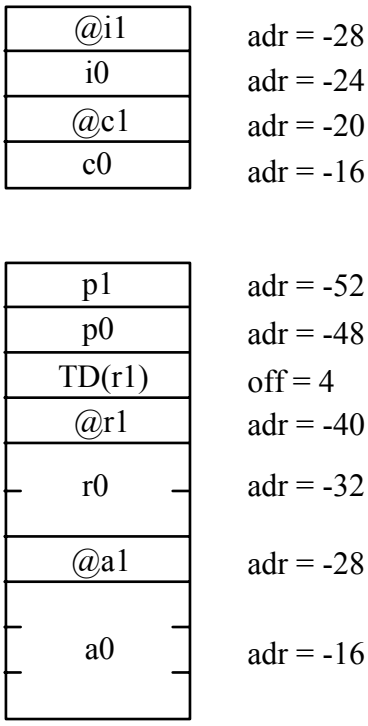
Prozedur-Parameter werden folgendermassen behandelt:

- Für alle Parameter (inklusiv **CHAR**-Typ, usw.) werden ein oder mehrere Wörter alloziert, was einen effizienteren Zugriff erlaubt.
- Ist ein (**VAR**-)Parameter ein *open* (or *dynamic) array* , dann werden dim+1 Wörter reserviert, wobei dim die Anzahl Dimensionen darstellt.
- Zu den Parametern werden negative Adressen zugewiesen, die ab den Wert -16 anfangen, wobei 16 die von der call Instruktion reservierte Anzahl Bytes darstellt.

Das folgende Beispiel fasst einige typische Fälle zusammen.

Beispiel

```
MODULE Parameters;


 TYPE
   A = ARRAY 3 OF INTEGER;
   R = RECORD x, y: INTEGER END;
   P = POINTER TO R;


 PROCEDURE P0(c0: CHAR;
          VAR c1: CHAR;
          i0: INTEGER;
          VAR i1: INTEGER);
 END P0;


 (* structured parameters *)

 PROCEDURE P1(a0: A;
          VAR a1: A;
          r0: R;
          VAR r1: R;
          p0: P;
          VAR p1: P);
 END P1;


 (* dynamic array parameters *)

 PROCEDURE P2(a0: ARRAY OF INTEGER;
          VAR a1: ARRAY OF INTEGER);


 END Parameters.
```



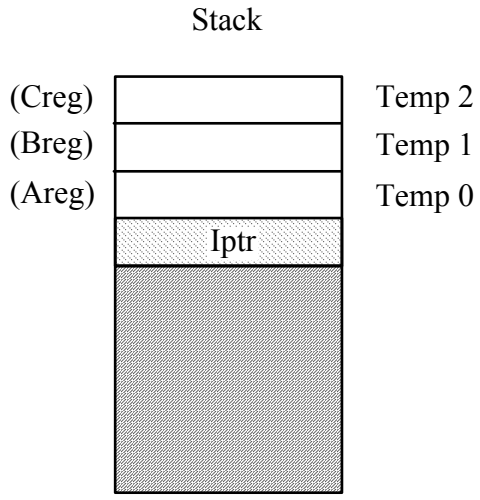
Temporäre Variablen

Auf temporäre Variablen muss dann zurückgegriffen werden, wenn die Auswertung von Ausdrücken mehr als drei Stackelemente benötigt (siehe Abschnitt 2.2). Praktisch kann eine solche Situation in drei Fällen auftreten, nämlich:

- innerhalb von "komplexen" Ausdrücken
- wenn Ausdrücke auf beiden "Seiten" einer Zuweisung auftreten
- bei einigen arithmetischen Operationen (**`MOD`**, **`ASH`**)

Bei normalen Prozeduraufrufen werden von der **`call`** Instruktion die drei Register und die Rücksprungadresse auf dem *stack* gerettet. Mit diesem Mechanismus können bis drei (aktuelle) Parameter auf effiziente Art auf den *stack* gebracht werden. Die Parameterbehandlung wird aber durch die nötigen Fallunterscheidungen - z.B. wenn es mehr als drei Parameter sind - komplizierter.

In dieser Implementation werden diese drei "freien" Speicherplätze für temporäre Variablen verwendet. Diese reichen nämlich für die drei oben erwähnten Fälle. Es werden also keine weitere temporäre Variablen gebraucht.



Figur 4-2: Die **`call`** Instruktion erniedrigt den *stack* um vier Wörter.

4.2. Code-Erzeugung

Die Code-Erzeugung erfolgt wie in [Crelier 90] beschrieben. Das Back-End kann nämlich an die Systemarchitektur des Transputers mit relativ wenig Aufwand angepasst werden. In dieser Implementation sind z.B. die Module `TOPC` und `TOPCa` 30% kürzer als die von Ceres.

Im diesem Abschnitt werden einige Punkte besprochen, die zu Schwierigkeiten geführt haben (Auswertung von Ausdrücken und Parameterbehandlung) oder zu dieser Implementation besonders sind (Behandlung von *forward jumps /calls* und von externen Zugriffen).

Auswertung von Ausdrücken

Mit Ausnahme der boolschen Ausdrücken wird ein Ausdruck der Form `e1 op e2` (wobei `op` für eine arithmetische Operation oder eine Relation steht) folgendermassen ausgewertet:

```
PROCEDURE expr(n: OPT.Node; VAR x: OPL.Item);
   VAR (*...*)
BEGIN
   CASE n^.class OF
      (*..*)
   | Ndop:
        IF n^.subcl IN {and, or} THEN
           expr(n^.left, x);
           IF n^.subcl = and THEN OPC.CondAnd(x)
           ELSE CondOr(x)
           END;
           expr(n^.right, y)
        ELSE
           IF ExprDepth(n^.right) < StkSize THEN
           expr(n^.left, x); expr(n^.right, y)
           ELSIF ExprDepth(n^.left) < StkSize THEN
           expr(n^.right, y); expr(n^.left, x);
           IF x.mode > Stk THEN OPL.Gen1(rev)
           ELSIF (*..*)
           END
           ELSE f:= n^.left^.typ^.form;
           expr(n^.right, y);
           IF y.mode = Ind THEN f := LInt END;
           OPL.SaveAreg(0, f);
           expr(n^.left, x); OPL.RestoreAreg(0, f)
              END
      (*..*)
   END
END expr;
```

Diese Auswertung wird im Modul `TOPV` durchgeführt; der Code für die Operation `op` wird aber im Modul `TOPC` (bzw. `TOPCa`) erzeugt. Liegen beide Operanden (genauer die Adresse oder der Wert dieser Operanden) auf dem Evaluationsstack, dann entspricht immer der *top-of-stack* dem zweiten Operanden.

Prozeduraufrufe

Prozeduren sind durch eine `Enter`- und eine `Exit`-Phase abgegrenzt, nämlich:

```
PROCEDURE Enter(proc: OPT.Object);
BEGIN
   IF proc = NIL THEN  (* enter module *)
      OPL.Gen0(stl, 0);  (* save return address to boot program *)
      (*..*)
   ELSE
      (* special prologue for procedure variables *)
      OPL.Gen0(ajw, -4); OPL.Gen0(stl, 0);
      (* external procedure *)
      IF proc^.mode = XProc THEN OPL.SetEntry(proc^.adr) END;
      (* forward call *)
      IF proc^.linkadr < 0 THEN OPL.FixProc(-proc^.linkadr) END;
      proc^.linkadr:= OPL.pc;
      "copy dynamic array value parameters if any"
      (* allocate space for local variables on the stack *)
      IF lsize > 0 THEN OPL.Gen0(ajw, -lsize) END;
      (*..*)
      IF "some intermediate access" THEN
         "save old anchor; update anchor"
      END
   END
END Enter;

PROCEDURE Exit(proc: OPT.Object);
BEGIN
   IF proc # NIL THEN  (* procedure *)
      (*..*)
      IF "some intermediate access" THEN
         "restore old anchor"
      END;
      (* release space for local variables from the stack *)
      IF lsize > 0 THEN OPL.Gen0(ajw, lsize) END;
      "release space for dynamic array copies if any"
   END;
   OPL.Gen1(ret)
END Exit;
```

Die *scope anchors* (für Zugriffe auf *intermediate variables*) und die als Wertparameter übergebenen *dynamic arrays* (siehe weiter) werden im Modul `TOPCa` von diesen zwei Prozeduren behandelt.

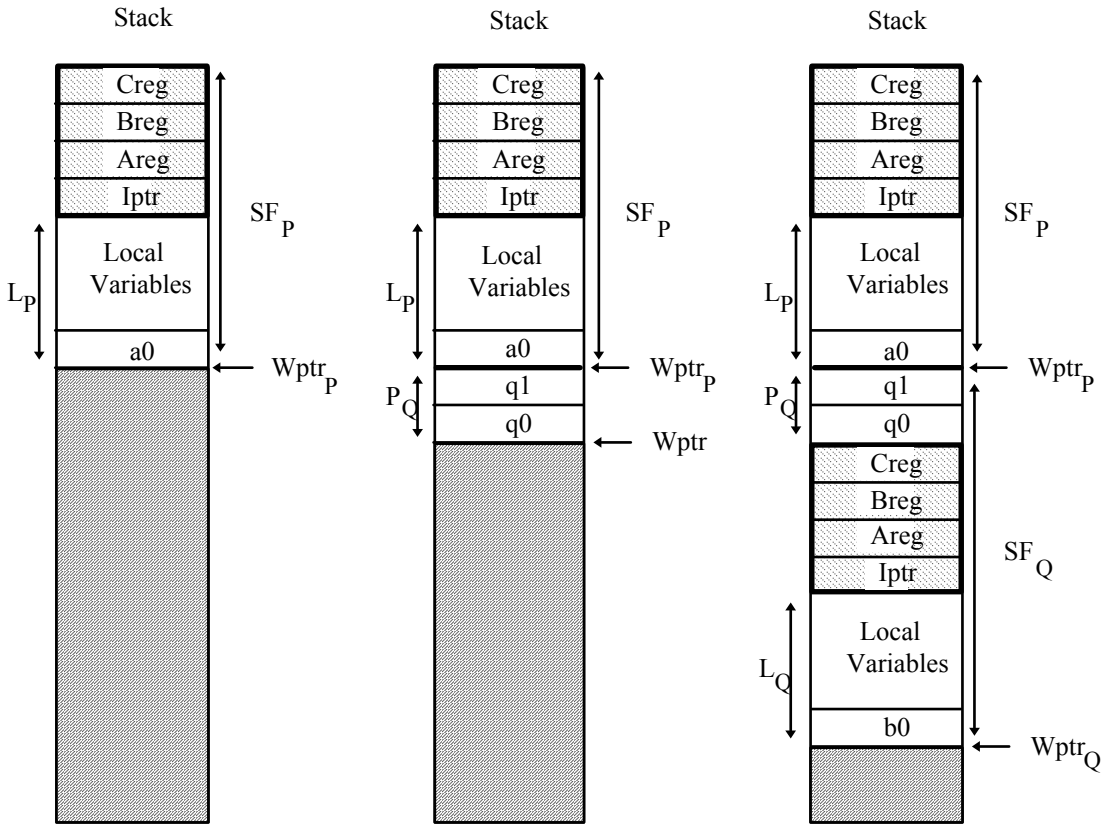
Bei Prozeduraufrufen muss aber vor allem der *stack*-Zustand konsistent gehalten werden, damit Variablen und Parameter richtig zugegriffen werden. Dieser Mechanismus lässt sich am besten durch ein Beispiel illustrieren.

Beispiel

```
MODULE Calls;

   PROCEDURE P;
      VAR
         a0: INTEGER;
         (* other local variables *)
```

```
    PROCEDURE Q(q0,q1:INTEGER);
      VAR b0: INTEGER;
    BEGIN (*3*)
    END Q;

  BEGIN (*1*) Q(0,1(*2*))
  END P;

BEGIN P
END Calls.
```

Die Figur 4-2 zeigt den Stack vor dem Aufruf der Prozedur Q (1), nach dem Laden der Parametern q0 und q1 (2) und beim Prozedureintritt (3).



Figur 4-3: Stack-Frame

Der *stack frame* besteht beim Aufruf der Prozedur Q (SF_Q) aus drei Teilen:

- Die *call area* enthält eine Kopie der drei Register `Areg`, `Breg` und `Creg` und die Rücksprungadresse (*return address*). Sie ist 4-Wörter gross (sei R = 4).
- Die *local area* enthält die lokalen Variablen, deren relative Adressen im positiven Wertebereich 0 bis (L_Q-1) liegen, wobei L_Q die gesamte Wortgrösse ist.
- Die *parameter area* enthält die Prozedurparameter, deren relative Adressen im negativen Wertebereich (0-R) bis (1-P_Q-R) liegen, wobei P_Q die Wortgrösse ist.

Der Zugriff auf die verschiedenen Variablen sieht dann so aus (alle effektiven Adressen sind positiv !):

1) Vor dem Prozeduraufruf (*procedure call*) :

   @ $a_i = i$

2) Nach dem Prozeduraufruf, aber vor dem Prozeduranfang:

   @ $a_i = i + P_Q$
   @ $q_i = - (i + R)$

3) Nach dem Prozeduranfang :

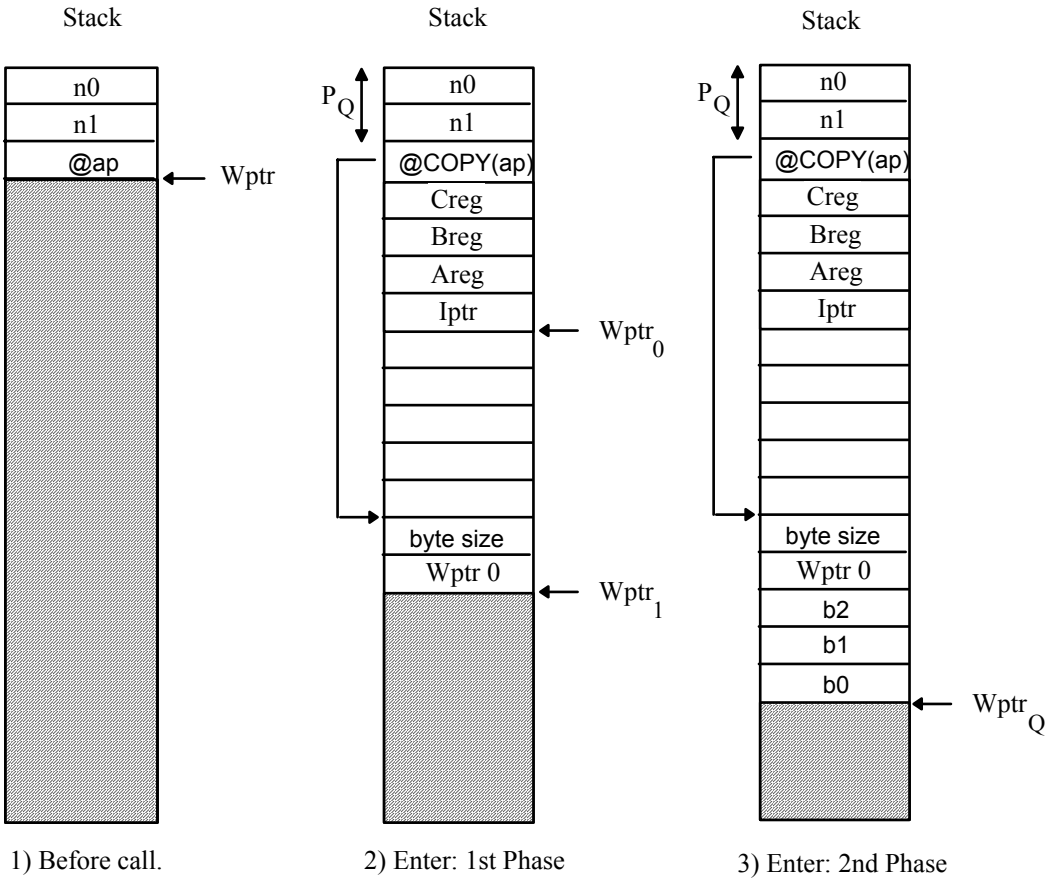
   @ $a_i = i + SF_Q$ (wo $SF_Q = P_Q + L_Q + R$)
   @ $q_i = L_Q - i$
   @ $b_i = i$

Als Wertparameter übergebene *dynamic arrays* werden auf dem Stack kopiert. Da ihr Speicherbedarf erst zur Laufzeit bekannt wird, muss der alte Wert von `Wptr` auch im *stack frame* gerettet werden.

Als Illustration wird das folgende Beispiel zusammen mit der Figur 4-4 vorgeführt.

Beispiel

```
MODULE DynArr;

  VAR a: ARRAY 3,6 OF CHAR;

  PROCEDURE Q(q: ARRAY OF ARRAY OF CHAR);
    VAR b0, b1, b2 : INTEGER;
  BEGIN (*...*)
  END Q;

BEGIN Q(a)
END DynArr.
```



Figur 4-4: *Dynamic arrays* als Wertparameter.

Im Gegensatz zu internen Aufrufen sind Compiler und Lader an der Behandlung von externen Aufrufen beteiligt, nämlich:

- Für interne Aufrufe ist der PC-relative Offset zur Kompilationszeit bekannt; der Compiler kann also entsprechende Code erzeugen.
- Für externe Aufrufe stellt der Compiler die für den Loader nötigen Informationen bereit (Fixup-Tabelle im Object-File). Die endgültigen *offsets* werden erst während dem Laden des Programms im Code ge-*patched*. Für jede importierte Prozedur wird ein einziger Eintrag in der Fixup-Tabelle gemacht; weitere Aufrufe werden im Code verkettet (siehe [Templ 90]).

Behandlung von forward jumps/calls

Die *forward jumps/calls* werden im Modul `TOPL` behandelt. Die Sprungdistanz wird von dieser Implementation auf die folgenden Werte festgelegt:

- 3 Bytes für *forward jumps* (diese Distanz entspricht 4 KBytes Code)
- 4 Bytes für *forward calls* (diese Distanz entspricht 32 KBytes Code)

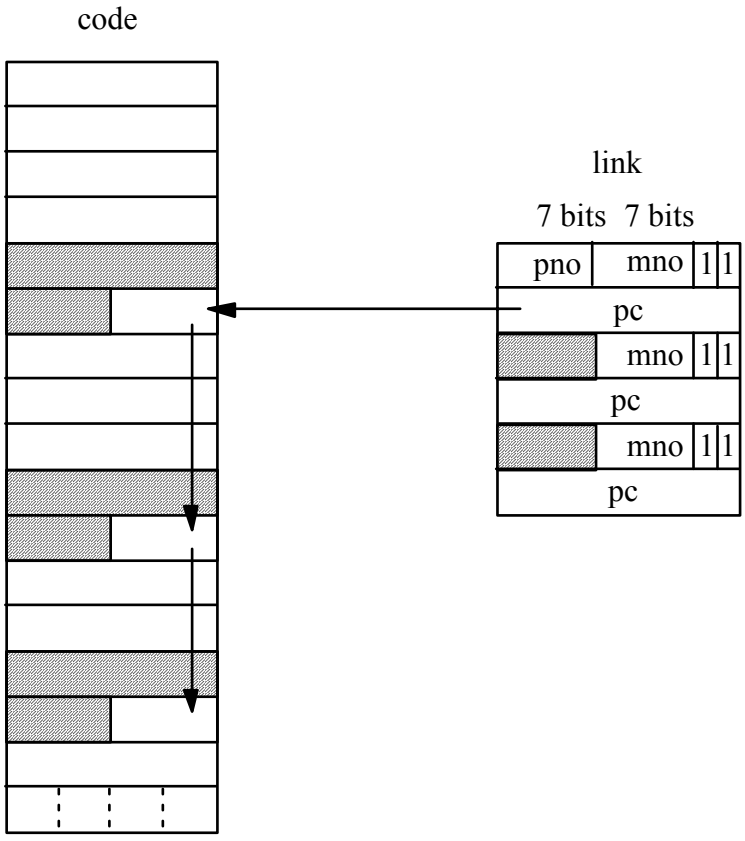
Externe Zugriffe

Es gibt drei Arten von externen Zugriffen:
- Zugriff auf externe Variablen
- Aufruf von externen Prozeduren
- Zuweisung von Prozeduren an Prozedur-Variablen

In der Fixup-Tabelle werden diese drei Fälle durch 2 Bits dargestellt (Figur 4-5). Im ersten Fall genügt ein Eintrag pro Modul; sonst ein Eintrag pro Prozedur. Weitere Zugriffe werden im Code verkettet.

Ein Eintrag besteht aus:
- einer Prozedur-Nummer (nur für externe Prozeduren)
- einer Modul-Nummer
- 2 Bits für die Zugriffsart
- einem PC-Wert (2 Bytes)

Für jeden externen Zugriff werden im Code 8 Bytes reserviert, wobei die zwei niedrigsten auf den nächsten Zugriff zeigen. Die Prozeduren `XRefVar` und `XRefProc` vom Modul `TOPL` sind für diese Aufgabe zuständig.



Figur 4-5: Externe Zugriffe werden im Code verkettet.

4.3. Object-File Format

Das Object-File Format ist mit demjenigen von Ceres identisch; in der Fixup-Tabelle werden jedoch die ersten zwei Bytes jedes Eintrags anders interpretiert (*LinkID*).

In [Crelier 90] (Abschnitt 4.4) ist eine allgemeine Beschreibung des Object-Files zu finden. Dieser wird nach dem folgenden Format von Lader und Decoder geladen, respektiv dekodiert.

EBNF Syntax

```
ObjFile     =  OFtag  HeaderBlk  EntryBlk  CommandBlk
               PointerBlk  ImportBlk  LinkBlk ConstBlk  CodeBlk
               TypeBlk  RefBlk.
OFtag       =  0F8X 36X.
HeaderBlk   =  refsize4  nofentries2  nofcoms2  nofptrs2
               nofrecs2  nofgmod2 nofixups2  datasize4
               consize2  codesize2  key4  modname20.
EntryBlk    =  82X {pc2}.
CommandBlk  =  83X {name pc2}.
PointerBlk  =  84X {off2}.
ImportBlk   =  85X {key4 name}.
LinkBlk     =  86X {LinkID fixup2}.
LinkID      =  pno7 mno7 mode2.    (#bits)
ConstBlk    =  87X {con1}.
CodeBlk     =  88X {inst1}.
TypeBlk     =  89X {tdsize2 tdadr2 {byte1} }.
RefBlk      =  8AX {0F8X procend name {Mode Form adr name}}.
Mode        =  Var | VarPar.
Form        =  Byte | Bool | Char | SInt | Int | LInt | Real |
               LReal | Set | Pointer | String.

Var = 1. VarPar = 3. Byte = 1. Bool = 2. Char = 3. SInt = 4.
                  Int = 5. LInt = 6.
Real = 7. LReal = 8. Set = 9. Pointer = 13. String = 15.
```

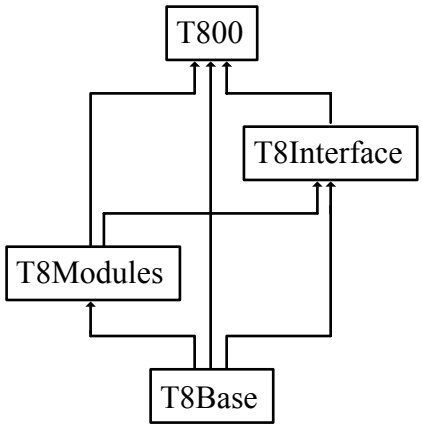
Die Prozedur `Outcode` von Modul `TOPL` schreibt das Object-File mit dem Suffix `.ObJ` (`.Obj` für Ceres) auf die Festplatte.

5. Runtime Organisation

Das T800 Runtime System (RTS) besteht aus folgenden Teilen:

- Der Hauptmodul `T800` enthält den Loader und den Kommando-Interpreter.
- Der Modul `T8Interface` funktioniert als Server für den T800.
- Der Modul `T8Modules` ist für die Modul-Verwaltung im Speicher zuständig.
- Der Modul `T8Base` enthält Low-level Routinen für die Kommunikation mit dem Transputerboard.
- Ein Boot-Programm für das Transputerboard.

Die Modul-Struktur sieht wie folgt aus:



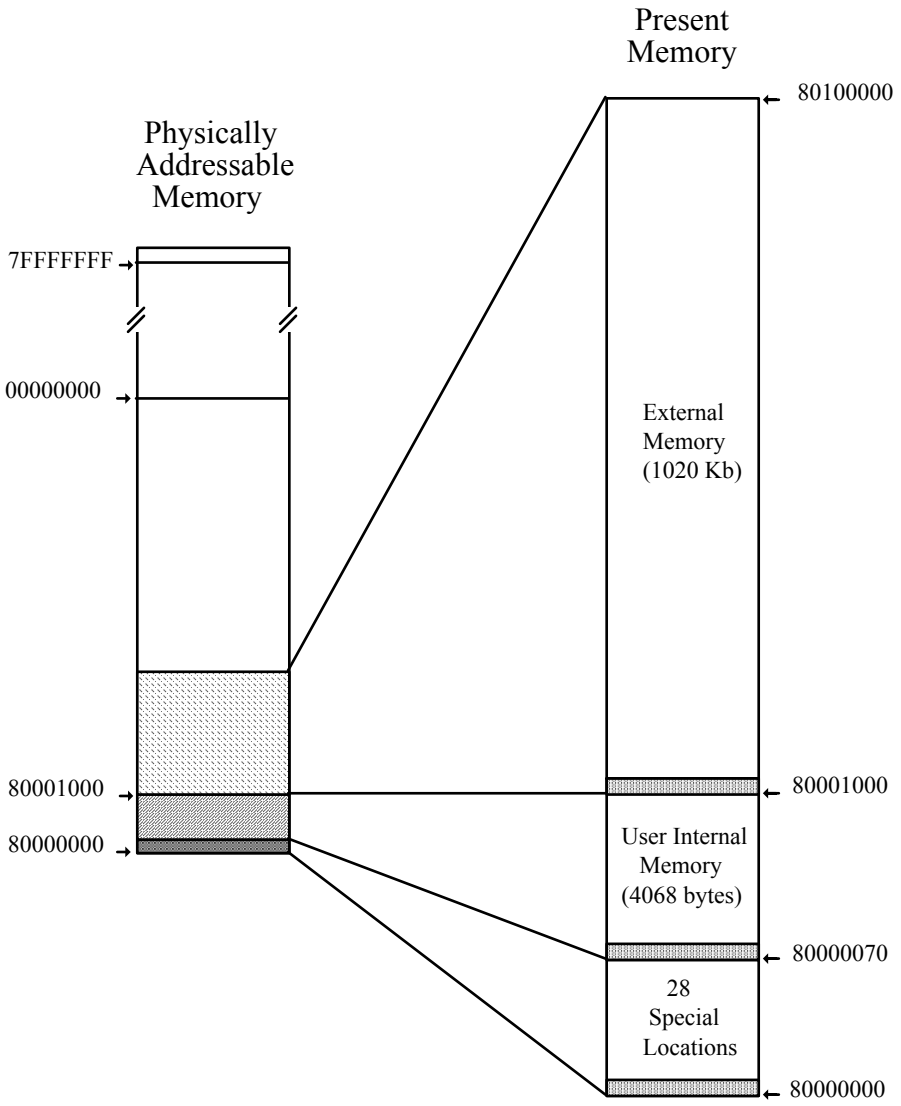
Figur 5-1: Modul-Struktur vom T800 RTS.

5.1. Memory-Layout

Die Hardware-Charakteristiken der Transputerkarte sind in [Ingold 90] beschrieben.

Bei Zugriffen auf den physischen Speicher gilt es folgendes zu beachten:

- Alle positiven Adressen, d.h. Adressen von `00000000H` bis `7FFFFFFFH`, erzeugen einen **`NIL`**-Trap.
- Für alle negativen Adressen wird der Betrag durch Modulo `10000H` auf den Speicherbereich `80000000H` bis `800FFFFFH` abgebildet.



Figur 5-2: Memory-Layout

Bei der benutzten Transputerkarte ist der externe Speicher (1 MByte RAM) etwa fünfmal langsamer als der interne Speicher (4 KBytes *on-chip Memory*).

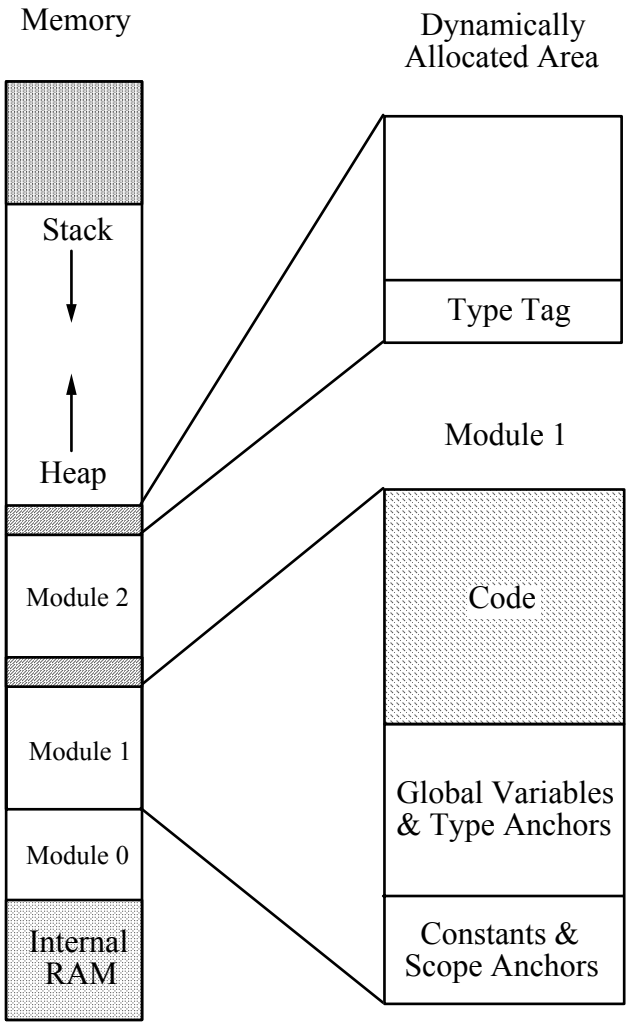
Zur Laufzeit wachsen sich der *Stack* und der *Heap* im Speicher entgegen; das Zusammenstossen der beiden Speicherbereiche wird nicht getestet.

In dieser Implementation wird der Speicher des T800-Boards wie folgt verwaltet:

- Der *Stack* wächst von oben nach unten. Das Register `Wptr` enthält die *top-of-stack* Adresse (Stack Pointer).
- Der *Heap* -Zustand wird auf Ceres vom Modul `T8Modules` verwaltet.

Diese Lösung lässt sich einfach realisieren und stellt im Rahmen dieser Arbeit einen guten Kompromiss dar zwischen Effizienz und Komplexität der Implementation. Ihr Hauptnachteil betrifft die dynamische Allozierung (mit

`NEW` und `SYSTEM.New`); in diesem Fall müssen nämlich zwischen Transputer und Ceres Informationen über den seriellen Link ausgetauscht werden.



Figur 5-3: Heap-Allozierung

5.2. Laden und Linken von Modulen

Ein Modul im Speicher besteht typischerweise aus drei Bereichen - auch *frames* genannt -, die nebeneinander auf dem *Heap* alloziert werden (Figur 5-3) :

- Dem *constant frame* für reelle und *string* Konstanten zusammen mit den *scope anchors*.
- Dem *code frame* für den gepatchten Programmcode.
- Dem *data frame* für die globalen Daten und die *type anchors*; dieser Speicherbereich wird zur Ladezeit mit Nullen initialisiert.

Ein Modul wird dann geladen wenn er:

- noch nicht im Speicher vorhanden ist und ein Kommando aus diesem Modul vom Kommando-Interpreter ausgeführt wird.
- von einem anderen Modul importiert wird, der selbst geladen wird.

Das Laden eines Moduls erfolgt in drei Schritten:
- die im Modul vorhandenen externen Zugriffe werden vom Linker ge-*patched*.
- der Modul wird über den seriellen Link in den Speicher des T800-Boards geladen.
- der Modulkörper wird ausgeführt.

Das Linken eines Moduls erfolgt in zwei Schritten:
- anhand der *entry* Tabelle wird die absolute Adresse von Prozeduren, Kommandos und Prozedur-Variablen berechnet.
- anhand der *link* Tabelle wird im Code jeder externe Zugriff ge-*patched*.

Boot-Programm

Beim Systemstarten wird ein kleines Boot-Programm (148 Bytes gross) in den internen Speicher des Transputerboards ausgeführt; dieses Programm ist entweder in einem ROM eingebrannt oder kann über den seriellen Link von der *host*-Maschine (hier der Ceres) geladen werden. Dieses Programm bleibt dann im Speicher resident.

Das Boot-Programm bringt nicht nur den T800 Prozessor in seinen Initialzustand, sondern erledigt auch zwei weitere Aufgaben zur Laufzeit:
- die Behandlung von **NIL**-Referenzen
- das Laden von Programmen und das Ausführen von Kommandos

Zu diesem Zweck werden auf dem Transputer zwei Prozesse gestartet:
- der *high priority* Prozess wartet einfach auf ein Hardware-Event, das durch eine **NIL**-Referenz ausgelöst wird
- der *low priority* Prozess besteht aus einer endlosen Schleife, die an das Oberon Loop vom Oberon System erinnert

Die hier als Main Loop bezeichnete Schleife sieht etwa wie folgt aus:

```
(* VAR len, adr: LONGINT; block: ARRAY len OF BYTE *)
LOOP
   LOOP
      Receive(len)              (* get block length *)
      Receive(adr)              (* get target address *)
      IF len = 0 THEN EXIT END
      Receive(block)            (* block move *)
   END
   Call(adr)                    (*execute program code *)
END
```

5.3. Schnittstelle Transputerboard-Ceres

Der Austausch von Informationen zwischen den Transputerboard und Ceres erfolgt über einen Oberon Task im Modul `T8Interface`. Dieser Task funktioniert als Server für den T800; seine Dienstleistungen werden in drei Fällen gebraucht, nämlich:

- Für alle Operationen, die von Prozeduren aus den Library Moduln Terminal und Files (siehe Abschnitt 6.2) durchgeführt werden.
- Für die dynamische Allozierung mit **`NEW`**`(p)` und `SYSTEM.New(p, size)`.
- Für das Abfangen von Laufzeitfehlern.

Zur Illustration wird hier eine einfache Transaktion zwischen den T800 und die Ceres durchgespielt. Die aufgerufene Prozedur `Terminal.Write(ch)` wird zur Laufzeit vom Oberon Task folgendermassen behandelt :

T800 :

```
PROCEDURE -Write(ch: CHAR)  (* code procedure in Terminal *)
   "send ID-tag"
   "send parameter ch"
```

Ceres :

```
PROCEDURE Poll;  (* Oberon task in module T8Interface *)
BEGIN
   IF T8Base.ErrFlgSet() THEN T8Modules.TrapHandler(FALSE)
   ELSIF "input expected" THEN
      IF "input done" THEN "get input" END
   ELSE
      WHILE "new message" & "no input expected" DO
         "get ID-tag"
         IF "NEW operation" THEN
            "heap allocation"
         ELSIF "output operation" THEN
            CASE "operation" OF
               "Write" :
                  T8Base.Read(ch);     (* get ch from T800 *)
                  Texts.Write(W, ch)  (* display ch in T800.Log *)
            (* ... *)
            END
         ELSIF "input operation" THEN
            CASE "operation" OF
            (* ... *)
            END
         ELSIF "file operation" THEN
            CASE "operation" OF
            (* ... *)
            END
         END
      END
   END
END Poll;
```

Figur 5-4: Einfache Transaktion zwischen den T800 und Ceres.

5.4. Fehlerbehandlung

Laufzeitfehler werden von der Prozedur `TrapHandler` im Modul `T8Modules` behandelt und im Viewer `T800.Trap` angezeigt. In der Benutzeranleitung (Anhang B) ist ein kurzes Beispiel zu finden.

Man unterscheidet drei Kategorien von Fehlern, nämlich:

- **`NIL`**-Referenzen werden vom Boot-Programm abgefangen
- ein *Heap*-Ueberlauf wird auf Ceres selbst behandelt
- die übrigen Fehler (sowie die **`HALT`**-Anweisung) werden von Transputer Instruktionen erzeugt

Hier sind die vom TOP2 Compiler verwendeten Instruktionen aufgelistet, die das *Error Flag* setzen können und damit zu einem Laufzeitfehler führen.

adc	add constant	**add**	addition
csngl	check single	**csub0**	check subscript from 0
cword	check word	**div**	division
mul	multiplication	**rem**	remainder
seterr	set error flag	**sub**	subtraction
fpchkerr	fp check error		

Die Anweisung **`HALT`**`(n)` von Oberon lässt sich z.B. einfach durch die Code-Sequenz **`ldc`** `n`, **`seterr`** übersetzen. Eine entsprechende Meldung soll aber für jede Art von Fehlern dem Benutzer mitgeteilt werden. Der Transputer bietet hier die Möglichkeit, die betroffene Instruktion anhand ihres PC-Wertes zu bestimmen.

Wird der Transputer auf seinen Initialzustand zurückgesetzt, dann wird der alte Inhalt der Register `Iptr` und `Wptr` auf den Evaluationsstack gerettet. Die ganze Fehler-behandlung beruht also auf diese zwei Werte.

Die Prozedur `TrapHandler` sieht folgendermassen aus.

```
PROCEDURE TrapHandler(Ceres: BOOLEAN);
BEGIN
   "open viewer T800.Trap"
   T8Base.Reset;                        (* reset transputer *)
   "get old value of Iptr and Wptr"
   GetMod(Iptr, mod);                   (* mod^ : module descriptor *)
   IF Ceres THEN (*..*) trapno := 1     (* out of heap space *)
   ELSE
      T8Base.Reset;
      data := T8Base.Peek(Iptr-3);      (* peek in program code *)
      "identify instruction from data"
      CASE "instruction" OF
         "add": trapno:= 13             (* integer overflow *)
      |  "div": trapno:= 5              (* integer division by zero *)
         (*..*)
      END
   END;
   "display error message in viewer T800.Trap"
   "reload boot file"                   (* reboot transputer *)
END TrapHandler;
```

6. Testumgebung

Zur Testumgebung des T800-Boards gehören folgende Tools:

- `TOP2`, der T800 Oberon Cross-Compiler
- `T800`, das T800 Runtime System
- `TDecoder`, der T800 Object-File Decoder
- `TDec`, der T800 Boot-File Decoder
- `Browser`, der Oberon Symbol-File "Decoder"

6.1. T800 Tools

Die von jedem Tool angebotenen Kommandos sind hier aufgelistet.

Tools	Kommandos	Parameter	Beispiele
TOP2	Compile	{name}~	Compile Test.Mod~
	Compile	*	
T800	Reset	-	
	Execute	name	Execute Test.Do
	Inspect	name	Inspect Test
	LoadMap	-	
	Watch	-	
	EraseLog	-	
	Close	-	
TDecoder	Decode	{name}~	Decode Test~
TDec	Prefix	{number}~	Prefix 10 0FFH -2~
	Generate	{name}~	Generate T800~
	Decode	{name}~	Decode T800~
Browser	ShowDef	{name}~	ShowDef Terminal~
	SetExtension	string	SetExtension ".SyM"

Der Transputer Tool kann mit `System.Open T800.Tool` geöffnet werden. Eine genauere Beschreibung ist in der Benutzeranleitung (Anhang B) zu finden.

6.2. Library Module

Einige Library Moduln sind zu Testzwecken implementiert worden:

- `Terminal` Ein-/Ausgabe mit Tastatur und Bildschirm
- `Files` Standard-Operationen mit der Festplatte
- `Math` / `MathL` Mathematische Standard-Prozeduren
- `Clock` Zeitabfragen

Die Moduln `Terminal` und `Files` sind *host*-abhängige Module, indem sie gewisse Ressourcen von Ceres dem Transputer zur Verfügung stellen. Diese enthalten Ceres ähnliche Operationen zur Ein-/Ausgabe von Daten auf den

Bildschirm und auf die Festplatte; sie sind in Form von Code-Prozeduren implementiert.

Bei aufgerufenen Prozeduren aus dem Modul `Terminal` erfolgt die Ein/Ausgabe im Viewer `T800.Log`. Eine Eingabe wird durch *carriage return* beendet.

Der Modul `Files` stellt ähnliche Operationen wie auf Ceres zur Verfügung. In der gegenwärtigen Implementation ist die maximale Anzahl der geöffneten Files auf vier beschränkt.

Aufgerufene Prozeduren senden einfach eine Meldung mit Identifikation (ein Byte) und Parametern über den seriellen Link (10 Mbits/sec Uebertragungsrate). Auf Ceres wird das Ankommen von Daten durch einen Oberon Task ständig überprüft. Die gewünschten Operationen werden dann ausgeführt und eventuelle Resultate dem Transputer übermittelt.

Die übrigen Module, nämlich `Math`, `MathL` und `Clock` brauchen keine Ceres Unterstützung; sie sind zum Teil auch als Code-Prozeduren implementiert.

Mit dem Modul `Clock` besteht die Möglichkeit, Zeitmessungen von Programmabläufen zu vergleichen. Für die Ausführungszeit von Codeblöcken, die Aufrufe zu den Moduln `Terminal` und `Files` enthalten, ist mit einem zusätzlichen Zeitaufwand zu rechnen.

Der Modul `Clock` ist hier als Beispiel angegeben.

```
MODULE Clock;  (* SM 17.6.90 *)

  CONST
    oneSec* = 15625;
    oneMin* = 60*oneSec;

  PROCEDURE -Time*(): LONGINT
    22H, 0F2H;          (* load timer *)

  PROCEDURE -Delay*(d: LONGINT)
    22H, 0F2H, 70H,     (* ldtimer, ldl 0 *)
    25H, 0F2H,          (* sum *)
    22H, 0FBH;          (* tin *)

END Clock.
```

6.3. Beispiele von dekodierten Programmstücken

Nachfolgend sind zwei Programmstücke und zwei kurze dekodierte Programme wiedergegeben; sie wurden mit `TDecoder.Decode` erzeugt. Der vollständige Output für das Programm `Factorial` ist im Anhang C zu finden.

Programmstück 1: Ausdrücke

x, y und z sind lokale **`INTEGER`** Variablen.

```
x:= 10 * x + (y + z) DIV 3
```

Dekodiertes Programm

```
ldc     10
ldl      0        [load x]
mul
ldl      1        [load y]
ldl      2        [load z]
add
ldc      3
div
add
stl      0        [store x]
```

Programmstück 2: Einfache Zuweisungen

`i: INTEGER` und `a: ARRAY 4 OF INTEGER` sind lokale Variablen.

```
i := a[3]; a[3] := a[i]
```

Dekodiertes Programm

```
ldlp     4
ldnl     0
stl      0          [store i]
ldl      0
ldc      4
csub     0          [check index i]
ldlp     1
wsub
ldlp     4
rev
ldnl     0          [load a[i] ]
rev
stnl     0          [store a[3] ]
```

Programm 1: Funktionen, Iteration und Rekursion

```
MODULE Factorial;

  CONST n = 8;

  PROCEDURE IterFact(x: INTEGER): INTEGER;
    VAR f: INTEGER;
  BEGIN f:= 1;
    WHILE x > 1 DO f:= f * x; DEC(x) END;
    RETURN f
  END IterFact;

  PROCEDURE RekFact(x: INTEGER): INTEGER;
  BEGIN
    IF x <= 0 THEN RETURN 1
    ELSE RETURN x * RekFact(x-1)
    END
  END RekFact;

BEGIN
  IF IterFact(n) # RekFact (n) THEN HALT(99) END
END Factorial.
```

Dekodiertes Programm

```
Factorial  stl      0          MODULE Factorial;
           ajw     -1          BEGIN
           ldc      8            IF
           stl      0               [load param. 8]
           call IterFact            IterFactorial(8)
           ajw      1
           stl      1               [store temp] #
           ajw     -1
           ldc      8
           stl      0               [load param. 8]
           call RekFact             RekFactorial(8)
           ajw      1
           ldl      1               [load temp]
           diff
           cj     END             THEN
           ldc     99
           seterr                    HALT(99)
END:       ret                   END Factorial.
```

```
IterFact    ajw     -4          PROCEDURE IterFact(x: INTEGER);
            stl      0          BEGIN
            ajw     -1            VAR f: INTEGER;
            ldc      1
            stl      0            f:= 1;
WHILE:      ldl      5
            ldc      1
            gt
            cj ENDWHILE           WHILE x > 1 DO
            ldl      0
            ldl      5
            mul
            stl      0              f:= f * x;
            ldl      5
            adc     -1
            stl      5              DEC(x)
            j    WHILE            END;
ENDWHILE:   ldl      0
            j      END            RETURN f
            ldc     17
            seterr                (* function trap *)
END:        ajw      1
            ret                 END IterFact;

RekFact     ajw     -4          PROCEDURE RekFact(x: INTEGER);
            stl      0          BEGIN
            ldl      4
            ldc      0
            gt
            eqc      0
            cj    ELSE              IF x <= 0 THEN
            ldc      1
            j      END              RETURN 1
            j     TRAP
ELSE:       ajw     -1            ELSE
            ldl      5
            adc     -1
            stl      0              [load param. x-1]
            call RekFact
            ajw      1
            ldl      4
            mul
            j      END              RETURN x*RekFact(x-1)
TRAP:       ldc     17            END
            seterr                (* function trap *)
END:        ret                 END RekFact;
```

Programm 2a: Prozedur-Variablen und externe Zugriffe

```
MODULE Export;

   TYPE PROC* = PROCEDURE (VAR x: INTEGER);
   VAR
      i*: INTEGER;
      p*: PROC;

   PROCEDURE Inc*(VAR x: INTEGER);
   BEGIN INC(x)
   END Inc;

BEGIN
   i:= 0; p:= Inc
END Export.
```

Dekodiertes Programm

```
Export      stl      0          MODULE Export;
            ldc      0          BEGIN
            ldc      3
            ldpi
            stnl    -3            i:= 0;
            ldc   @Inc
            ldc      2
            ldpi
            stnl    -7            p:= Inc
            ret                 END Export.

Inc         ajw     -4          PROCEDURE Inc(VAR x: INTEGER);
            stl      0          BEGIN
            ldl      4
            ldnl     0
            adc      1
            ldl      4
            stnl     0             INC(x)
            ret                 END Inc;
```

Programm 2b: Prozedur-Variablen und externe Zugriffe

Die mit `@` markierten Bezeichner stehen für absolute Adressen, die erst zur Ladezeit ausgelöst werden. Hier stellen `@Export` die Basisadresse des Moduls `Export` und `@Inc` die absolute Adresse der Prozedur `Dec` dar.

```
MODULE Import;

   IMPORT Export;

   TYPE PROC = PROCEDURE (VAR x: INTEGER);
   VAR
      i: INTEGER;
      p: PROC;
      q: Export.PROC;

   PROCEDURE Dec(VAR x: INTEGER);
   BEGIN DEC(x)
   END Dec;

BEGIN
   i:= 0; Export.p(i); Export.Inc(Export.i);
   IF i # Export.i THEN HALT(98) END;

   p:= Dec; q:= Export.p; p(i); q(i);
   IF i # 1 THEN HALT(99) END
END Import.
```

Dekodiertes Programm

```
Import      stl      0          MODULE Import;
            ldc      0          BEGIN
            ldc      3
            ldpi
            stnl    -3             i:= 0;
            ajw     -1
            ldc      0
            ldpi
            ldnlp   -4
            stl      0             [load param. @i]
            ldc @Export
            ldnl    -2
            adc     -3
            gcall                  Export.p(i);
            ajw      1
            ajw     -1
            ldc @Export
            ldnlp   -1
            stl      0             [load param. @Export.i]
            call Export.Inc        Export.Inc(Export.i);
            ajw      1
            ldc      2
            ldpi
            ldnl   -15
            ldc @Export
            ldnl    -1
            diff
            cj   ENDIF             IF i # Export.i THEN
            ldc     98
```

```
                seterr                    HALT(98)
ENDIF:          ldc @Dec                  END;
                ldc     3
                ldpi
                stnl   -24                p:= Dec;
                ldc @Export
                ldnl    -2
                ldc      0
                ldpi
                stnl   -28                q:= Export.p;
                ajw     -1
                ldc      1
                ldpi
                ldnlp  -28
                stl      0                [load param. @i]
                ldc      3
                ldpi
                ldnl   -31
                adc     -3
                gcall                     p(i);
                ajw      1
                ajw     -1
                ldc      0
                ldpi
                ldnlp  -32
                stl      0                [load param. @i]
                ldc      2
                ldpi
                ldnl   -36
                adc     -3
                gcall                     q(i);
                ajw      1
                ldc      1
                ldpi
                ldnl   -36
                ldc      1
                diff
                cj     END                IF i # 1 THEN
                ldc     99
                seterr                      HALT(99)
END:            ret                       END Import.

Dec             ajw     -4                PROCEDURE Dec(VAR x: INTEGER);
                stl      0                BEGIN
                ldl      4
                ldnl     0
                adc     -1
                ldl      4
                stnl     0                  DEC(x)
                ret                       END Dec;
```

6.4. Messungen

In [Templ 90] werden anhand der Stanford Benchmarks die SPARC und die Ceres Implementation von Oberon verglichen; ursprünglich handelt es sich bei der Stanford Benschmarks um C-Testroutinen, also um keine typische Oberon Programme (z.B. keine externe Zugriffe). Das Vergleichen von Zeitmessungen führt dennoch zu interessanten Feststellungen.

Alle Zeitangaben sind in Millisekunden, alle Codegrössen in Bytes angegeben.

Ceres-1 ist mit 10 MHz, Ceres-2 mit 25 MHz und der T800 mit 20 MHz getakt.

(msec)	Ceres-1	Ceres-2	T800	Ceres-1	Ceres-2	T800	T800(1)
	(with index-check)			(without index-check)			
Perm	2640	383	658	2370	323	619	459
Towers	3340	483	793	2576	323	724	568
Queens	1910	230	450	1506	163	406	283
IntMm	4530	600	751	3726	423	673	537
Mm (2)	5390	820	650	4566	646	584	448
Puzzle	28426	4113	5330	21010	2597	4840	4111
Quick	2376	336	566	1892	233	479	357
Bubble	5098	653	1267	3270	316	1061	871
FFT	8240	1264	1227	6490	936	1088	763
Average	6883	987	1299	5267	662	1182	933
	10.4	**1.5**	**2.0**	**8**	**1**	**1.8**	**1.4**
Tree (3)	2016	310	909	1952	300	896	806

(1): Stack im 4KBytes internen RAM des T800 Prozessors.
(2): Matrix-Multiplikation mit reellen 40x40-Matrizen.
(3): Dynamische Allozierung erfolgt auf Ceres.

Tabelle 6-1.

Nach diesen Messungen sind Programme auf Ceres-2 etwa 1.5-mal schneller als auf dem T800. Zwei Punkte sind aber zu berücksichtigen: einerseits ist ihre Taktfrequenz 25% schneller als die von T800, anderseits sind auf dieser Maschine zwei kleinen *Memory Caches* vorhanden (512 Bytes für Instruktionen und 1 KByte für Daten).

Mit Index-Prüfung verschlechtert sich die mittlere Ablaufszeit (Tabelle 6-1) um 30% für Ceres-1, 49% für Ceres-2 und 10% für den T800; damit sind

"*index-checks* " für den Transputer etwa dreimal billiger als für NS32X32 Prozessoren. Dieses Resultat ist zum Teil auf die Präsenz eines *memory cache* (Ceres-2) und eines internen Speichers (T800 mit internen Stack) zurückzuführen.

Die Stanford Benchmarks sind auf Ceres mit dem NOP2 Compiler übersetzt worden. In der Tabelle 6-2 ist die Codegrösse für beide Implementationen angegeben.

(# Bytes)	Ceres	T800	Ceres/x	T800/x	T800/x(1)
Code	7292	7596	6648	6900	6686
Data	73056	73068			
	1.1	**1.15**	**1**	**1.04**	**1**

(1): mit Sprung-Optimierung (214 überflüssige Prefix-Instruktionen bei *forward jumps*).

Tabelle 6-2.

Durch kleine Optimierungen (besonders bei Sprüngen) kann die Codegrösse um etwa 5% reduziert werden; damit sollte die Codedichte von Ceres erreicht werden und es kann mit besseren Zeitmessungen gerechnet werden.

Das Handbuch *Compiler Writer's Guide* ([Inmos 88]) enthält einige *hints* bezüglich kleiner Optimierungen, die zusammen mit Sprüngen oder FPU-Operationen durchgeführt werden können.

7. Zusammenfassung

Mit der Portierung des OP2 Compilers auf den Inmos Transputer ist eine weitere Implementation von Oberon entstanden. Die Anpassung des Compilers konnte einfach durch das Ersetzen des Back-Ends erfolgen. Mit der Realisierung eines einfachen Runtime-Systems wird zusätzlich eine kleine Testumgebung für die Weiterentwicklung dieses Projekts angeboten.

Im Rahmen dieser Arbeit erfolgte die Entwicklung des Back-Ends schrittweise, indem jede Aenderung in der Code-Erzeugung systematisch überprüft wurde. Dieses Vorgehen ist aber nur dann erfolgversprechend wenn jede Erweiterung gut überlegt wird. Die einfachen Konzepte, die im OP2 Compiler sowie im T800 Prozessor vorliegen, sind in dieser Hinsicht eine hilfreiche Voraussetzung gewesen.

Konkret stellten die Auswertung von Ausdrücken und die Parameterbehandlung die zwei grössten Schwierigkeiten bei der Implementation des Back-Ends dar. In beiden Fällen hätten Anpassungen im Front-End zu einer einfacheren Lösung geführt; damit wäre aber sein maschinenunabhängiger Charakter verloren gegangen.

Das Back-End wurde in drei Monaten implementiert. Nach zwei Monaten konnten die ersten Tests durchgeführt werden. Mittels Benchmarks-Routinen wurde später die Ceres mit der T800 Implementation verglichen.

Im letzten Monat wurde das Runtime System zusammen mit den Library Moduln implementiert. Hier waren die zwei wichtigsten Probleme die Implementation eines Oberon Tasks (Schnittstelle T800-Ceres) auf der Ceres und die Behandlung von **`NIL`**-Referenzen auf dem T800.

Während diesen vier Monaten habe ich mit dem Transputer sowie mit dem OP2 Compiler gute Erfahrungen gemacht. Diese Arbeit bildet den ersten Schritt eines Projekts, das mit der Portierung des ganzen Oberon Systems seinen Höchstpunkt erreichen wird. Ich wünsche meinem Nachfolger viel Erfolg für die Weiterentwicklung dieser Implementation.

Schliesslich möchte ich Martin Holzherr (Diplomand) für das Durchlesen dieser Arbeit und für sein Tool `Project`, Joseph Templ für sein Tool `Browser`, Beat Heeb für das Mitteilen seiner Erfahrungen mit dem T800-Board und die von ihm vorgenommene Hardware-Modifikation zur Behandlung der Laufzeitfehler, Régis Crelier für seine Erklärungen über OP2 und seine guten Tips, Michael Franz und Cuno Pfister für ihre Betreuung danken.

Zürich, August 1990

Referenzen

[**Crelier 90**]
Crelier R., OP2 : A Portable Oberon Compiler.
Report 125 (Februar 1990), Departement Informatik, ETH Zürich.

[Gutknecht 89]
Gutknecht J., The Oberon Guide.
Report 119 (Dezember 1989), Departement Informatik, ETH Zürich.

[**Heeb 90**]
Heeb B. & Pfister C., On Intermediate Variables and Local Procedures as Parameters.
Internal paper, Departement Informatik, ETH Zürich.

[Homewood 87]
Homewood M. & al., The IMS T800 Transputer.
IEEE Computer, Vol. 7(9), Oktober 1987, S. 10-26(17).

[Ingold 90]
Ingold T., Transputer Board für Ceres.
Semesterarbeit (März-Juni 1990), Departement Informatik, ETH Zürich.

[**Inmos 88**]
Inmos Limited, Transputer Instruction Set : A Compiler Writer's Guide.
Prentice Hall 1988 (167 S.).

[Inmos 89]
Inmos Limited, The Transputer Databook (2. Edition).
Inmos Ltd. (UK) 1989 (582 S.).

[**Pfister 90**]
Pfister C. & Crelier R., Oberon Technical Notes.
Internal paper, Departement Informatik, ETH Zürich.

[Templ 90]
Templ J., SPARC-Oberon: User's Guide and Implemenation.
Report 133 (Juni 1990), Departement Informatik, ETH Zürich.

[Wirth 88]
Wirth N. & Gutknecht J., The Oberon System.
Report 88 (Juli 1988), Departement Informatik, ETH Zürich.

[Wirth 89]
Wirth N., The Programming Language Oberon (Revised Edition).
Report 111 (September 1989), Departement Informatik, ETH Zürich.

Anhang

A. Aufgabenstellung

ETH ZÜRICH
Departement Informatik
Institut für Computersysteme

25.4.90

Oberon Compiler Back-End für Inmos Transputer

Diplomarbeit für Herrn Stéphane Micheloud, Abt. IIIC

Thema

Höhere Programmiersprachen sollen es ermöglichen, von einzelnen Rechnern und ihren Besonderheiten zu abstrahieren. Somit ist es erwüunscht, dieselbe Sprache für möglichst vielen verschiedenen Rechnern verwenden zu können. Dies dedingt für jeden Rechnertyp einen eigenen Compiler.

Auf diesem Grund ist am Institut für Computersysteme ein portabler Oberon Compiler entwickelt worden. Dieser besteht aus zwei klar getrennten Teilen: dem Front-End und dem Back-End. Der erste liest den zu kompilierenden Text, macht die lexikalische Analyse und Satzzerlegung, überprüft die kontextabhängige Syntax, wie z.B. Typenkompatibilitätsregeln, und baut eine Symbol-Tabelle auf, sowie einen Baum, der alle Anweisungen des Moduls repräsentiert. Diese Aufgabe des Front-Ends ist maschinenunabhängig, sodas eine Anpassung des Compilers auf einen anderen Rechner mit relativ wenig Aufwand erfolgen kann. Nur das Back-End, das diese Zwischendatenstruktur in Code umwandelt, muss ersetzt werden.

Aufgage

Entwickeln Sie ein Back-End für den vorhandenen OP2 Compiler, das Code für de Inmos Transputer erzeugt. Kompilierte Oberon-Module benötigen ein Runtime-Insfrastruktur, um ausgeführt werden zu können (Modul-Lader/Linker, Command-Interpreter, Speicherverwaltung, Garbage Collector, …). Ferner soll eine einfache Testumgebung geschaffen werden, welche es erlaubt, Programme von der Ceres auf ein Transputerboard zu laden und dort auszuführen. Die Art der Testprogramme ist mit dem betreuenden Assistenten abzusprechen.

Im Rahmen einer Semesterarbeit werden ein Transputerboard für die Ceres entwickelt. Dieses Board kann für die Tests verwendet werden. Ausserdem steht ein in Oberon geschriebener Interpreter zur Verfügung. Als erstes soll ein Decoder beschrieben werden.

Der zu entwickelnde Compiler ist ein Cross-Compiler. Er läuft auf der Ceres, der erzeugte Binärcode wird mittels einer Art Terminalprogramm über die seriellen Transputerlinks auf das Transputerboard geladen.

Bemerkungen

- Legen Sie eine Zeitplan vor.
- Die Arbeit soll jede Woche mit dem betreuenden Assistenten besprochen werden.
- Der Schlussbericht soll folgende Teil enthalten:
  - Analyse der Aufgabe
  - Beschreibung des gewählten Object-File-Format
  - Beschreibung der Runtime-Organisation
  - Beschreibung der Testumgebung
  - Beispiele von dekodierten Programmstücken
  - Zusammenfassung
- Am Ende der Arbeit ist ein schriftlicher Bericht, sowie eine Floppy-Disk mit den Programmlisten und der Dokumentation abzugeben.

Zuständiger Professor:	Prof. N. Wirth
Zuständiger Assistant:	C. Pfister / M. Franz
Ausgabe:	25.4.90
Abgabe:	24.08.90

B. T800 Tool - Anleitung

C. Listings

- TOP2.Mod
- TOPV.Mod
- TOPC.Mod
- TOPCa.Mod
- TOPL.Mod
- TOPM.Mod

- T800.Mod
- T8Interface.Mod
- T8Modules.Mod
- T8Base.Mod

- Terminal.Mod
- Files.Mod
- Clock.Mod
- Math.Mod
- MathL.Mod

- TDecoder.Mod
- Factorial.Mod
- Factorial.DeC

- TDec.Mod
- T800.HEX
- T800.DEC