BOB+
Programmierhandbuch

Version 1.0
Oktober 2005

Ralf-Erik Ebert

BOB+ Programmierhandbuch

Version 1.0, Oktober 2005

1 Vorwort

Als ich im vergangenen Jahr (nicht ohne einen leisen Anflug von Nostalgie) einen ATARI Portfolio erstanden hatte, begab ich mich auf die Suche nach einer Möglichkeit, das gute Stück auch zu programmieren. Genauer: Das Programmieren für den Portfolio sollte auf dem Portfolio selbst möglich sein – schließlich ging das mit meinem Eigenbau-Homecomputer aus den achtziger Jahren, der noch viel ärmlicher ausgestattet war, ja auch!

Nun sind aber mit den Jahren Ansprüche und Erwartungen gewachsen, so dass das Ergebnis der Suche nach geeigneten Programmiersprachen/-systemen – zumindest für meinen Geschmack – etwas ernüchternd ausfiel. Prinzipiell möglich war die Programmiererei mit GoFolio, mit dessen Syntax ich mich aber nicht recht anfreunden konnte sowie dem doch recht steinzeitlichen PO-BASIC.

Und dann war da noch BOB. BOB fand sich mitsamt C-Quellcode und ein paar Beispielen in einem ZIP-Archiv [1], versprach Objektorientierung und sah auf den ersten Blick so ähnlich wie C++ aus. Leider war keinerlei Dokumentation dazu aufzutreiben. Eine längere Internet-Suche förderte immerhin BOBs Ursprung, einen 1991 entstandenen Artikel von David Michael Betz im Dr.Dobbs Journal [2] zutage. Allerdings ist dieser Artikel nicht hinreichend, um die Sprache tatsächlich verwenden zu können. Neben einer groben Erläuterung der hinter BOB stehenden Ideen, beschränkt sich der Autor auf die Aussage, dass es sich um keinen Ersatz für C oder C++ handele und es einfach sei, BOB um eigene Funktionen zu erweitern. Dass letzteres für eine sinnvolle Benutzung auch notwendig ist, zeigte dann ein erster Blick auf die Originalquellen und die darin bereitgestellten vordefinierten Funktionen.

Eine ausführlichere Recherche im Internet ergab, dass BOB von D. M. Betz selbst weiterentwickelt worden ist [3]. Die aktuelle Version lehnt sich ihrem Konzept nach (insbesondere was die Deklaration und Verwendung von Klassen betrifft) jedoch stärker an JavaScript als an C++ an und ist in ihrem Codeumfang bedeutend angewachsen, so dass ein Betrieb auf minimal ausgestatteten Rechnern – wie dem erwähnten Portfolio – nicht mehr möglich ist. Sie schied damit als einfache Alternative aus[1].
Also machte ich mich ans Werk, mit der Absicht, BOB etwas tiefer zu verstehen, die Sprache so zu dokumentieren, dass es möglich wird, sie sinnvoll einzusetzen und dabei um einige wichtige Funktionen zu erweitern. Dabei zeigte sich, dass es an einigen Stellen notwendig bzw. zweckmäßig war, die Sprache selbst zu erweitern und ihre Implementierung zu überarbeiten. Zur besseren Unterscheidung vom Original habe ich die neue Version **BOB+** getauft.

Wichtig war in diesem Zusammenhang, das ursprüngliche Ziel – die Benutzbarkeit auf Minimalsystemen, wie dem ATARI Portfolio – nicht aus den Augen zu verlieren, d.h. die Gesamtgröße von BOB+ weiterhin so klein zu halten, dass er auf solchen Systemen lauffähig bleibt und noch genügend Arbeitsspeicher für die eigentlichen Programme übrig lässt.

Der vorliegende Text ist keine Einführung in die Programmierung. Vielmehr wird versucht, die Sprache BOB+ in knapper Form darzustellen. Dabei müssen grundlegende Programmierkenntnisse in einer anderen – vorzugsweise C-ähnlichen – Sprache sowie Grundkenntnisse der objektorientierten Programmierung vorausgesetzt werden. Gleichzeitig werden die Gemeinsamkeiten und Unterschiede zu BOB dargestellt, so dass die Lektüre hoffentlich auch als Wegweiser für die Benutzung des Vorfahren von Nutzen ist.

[1] Wenn im weiteren Text von BOB die Rede ist, so ist damit immer die ursprüngliche Fassung nach [2] gemeint.

2 Grundlagen

2.1 Was ist BOB, was ist BOB+?

BOB ist eine Hybridsprache, d.h. eine ihrem Konzept nach prozedurale Sprache mit objektorientierten Erweiterungen. Syntaktisch lehnt sich BOB an C/C++ sowie Script-Sprachen wie JavaScript (D. Betz sieht eher eine Nähe zu Lisp) an. BOB ist eine sehr kleine Sprache mit nur 11 Schlüsselwörtern (vgl. Abschnitt 7), die schnell und einfach erlernbar ist. BOB ist zudem eine sehr gutmütige Sprache, die keine explizite Typisierung und keinen Deklarationszwang kennt – mit allen Vor- und Nachteilen, die das mit sich bringt.

BOB ist weder ein Compiler noch ein „echter“ Interpreter. Der Quellcode eines Programms wird zunächst in einen Bytecode übersetzt, der anschließend interpretiert wird.

BOB+ übernimmt das Konzept BOB, erweitert jedoch sowohl die Sprache als auch deren Implementierung um einige grundlegende Elemente, die für Anwendungen, die über Programmierexperimente hinausgehen, erforderlich sind. Neben einer Vielzahl zusätzlicher vordefinerter Funktionen (vgl. Abschnitt 11) gehören hierzu insbesondere:

- Unterstützung von Gleitpunkt-Arithmetik (Datentyp float)
- 32-Bit-Integer
- ganzzahlige numerische Literale auch in oktaler oder hexadezimaler Schreibweise
- variable Parameterlisten für Funktionen
- „echte“ polymorphe Klassen[2], automatische Destruktoren und initialisierte statische Member-Variablen
- Überladen von Operatoren für Objekte
- Typinformationen zur Laufzeit (RTTI)
- Zugriff auf systemnahe Funktionen
- integrierte Serialisierung beliebiger Daten
- Verwendung vorkompilierter Bytecode-Bibliotheken
- direkte Einbindung von BOB+-Quellen in Kommando-Scripts (Batch-Dateien)
- Ergebnisrückgabe an den Aufrufer im DOS-Errorlevel
- verbesserte Speicherverwaltung mit expliziter Freigabe nicht mehr benötigter Daten

Dabei bleibt BOB+ abwärtskompatibel zu BOB. D.h. für BOB geschriebene Programme können ohne Änderung mit BOB+ übersetzt und ausgeführt werden.

[2] BOB erlaubt zwar virtuelle Funktionen, kennt aber keine Möglichkeit, überschriebene Funktionen der Basisklasse aufzurufen.

2.2 Benutzung von BOB+

BOB+ besteht aus einer einzigen ausführbaren Datei (BP.EXE) und wird von der Kommandozeile gestartet. Um Aufrufe von beliebiger Stelle zu vereinfachen, sollte der Pfad zu dieser Datei in die PATH-Umgebungsvariable aufgenommen werden. Für den eigentlichen Aufruf gilt dann folgende Syntax:

```
bp [-i][-d][-t][-c] [sourcefile [..]] [-r objfile [..]]
   [-o outfile] [# userarg [..]]
```

Dabei haben die einzelnen Optionen folgende Bedeutung:

- -i (info) Anzeige von Copyright und Funktionsnamen[3]
- -d (debug) Ausgabe des erzeugten Byte-Codes beim Übersetzen
- -t (trace) Ablaufverfolgung des Byte-Codes bei der Ausführung
- -c (compile) Die angegebenen Quellen werden nicht ausgeführt sondern nur in Byte-Code übersetzt.
- -r (read / run) Lesen und Ausführen von vorkompilierten Byte-Code-Modulen
- -o (output) Name der auszugebenden Byte-Code-Datei. Die Option wird nur ausgewertet, wenn auch –c gesetzt ist. Fehlt die Angabe, so wird als Standardwert „out.bpm“ verwendet.

Quelldateien und einzulesende Module können ohne Dateiendungen angegeben werden. In solchen Fällen werden implizit die Erweiterungen „.bp“ bzw. „.bpm“ angenommen[4].

Bei Angabe mehrerer Quelldateien werden diese in der angegebenen Reihenfolge in Byte-Code übersetzt, bevor die eigentliche Programmausführung beginnt. Die zu ladenden vorkompilierten Module werden *vor* dem Kompilieren der Quellen (und ebenfalls in der angegebenen Reihenfolge) gelesen.

BOB+ erlaubt zusätzlich die Angabe einer Liste von Benutzer-Argumenten, die an das auszuführende Programm weitergegeben werden. Diese Liste wird durch ein Doppelkreuz (#) eingeleitet und bildet das Ende der Kommandozeile. Innerhalb eines BOB+-Programms kann mit den vordefinierten Funktionen getargs() bzw. getusrargs() auf die gesamte bzw. die Benutzer-Argumentliste zugegriffen werden (vgl. Abschnitt 11.9).

Für den Aufruf von Funktionen und die Zwischenspeicherung von Argumenten bzw. temporären Werten verwendet BOB/BOB+ einen Stack mit einer Kapazität von 500 Einträgen. BOB+ erlaubt die Modifikation dieses Wertes mit Hilfe der Umgebungsvariablen BPSTACK. So kann z.B. mit dem Eintrag `SET BPSTACK=1000` die Stack-Größe verdoppelt werden.

2.3 Weitergabe von BOB+-Programmen

Die Weitergabe eines BOB+-Programms an Dritte kann als Quelltext oder in vorkompilierter Form (also als Bytecode-Modul) erfolgen.

Für die Ausführung von Bytecode-Modulen steht die BOB+-Laufzeitungebung BPR.EXE zur Verfügung. Die Laufzeitumgebung darf frei kopiert und weitergegeben werden. Gegenüber der Vollversion von BOB+ besteht die Einschränkung, dass nur vorkompilierte Module ausgeführt –

[3] BOB verhält sich immer so, als sei –i gesetzt, die Optionen –d und –t sind in BOB ebenfalls verfügbar.
[4] In BOB *darf* keine Erweiterung angegeben werden und die Datei *muss* immer die Erweiterung „.bob“ haben.

also keine Quelltexte direkt übersetzt – werden können[5]. Dem entsprechend werden die Optionsschalter –c, -r, -o, -d und –t nicht unterstützt

Der Aufruf der Laufzeitumgebung erfolgt nach der Syntax

```
bpr [-i] [ objfile [..]] [# userarg [..]]
```

Dabei gibt die Option –i einen Copyright-Hinweis und die Aufrufsyntax aus. Für die zu ladenden vorkompilierten Module und die Benutzerargumente gelten sinngemäß die Aussagen aus Abschnitt 2.2. Mindestens ein Modul muss eine main-Funktion als Einsprungpunkt besitzen. Wie in der Vollversion besteht die Möglichkeit, die verfügbare Stack-Größe mit der Umgebungsvariablen BPSTACK einzustellen.

3 Programmstruktur

Jedes BOB/BOB+-Programm besteht – wie in C – aus einer Menge von Funktionen (siehe Kapitel 9). Einsprungpunkt ist die parameterlose Funktion *„main()“*. Neben reinen Funktionen können auf der obersten Strukturebene zusätzlich Klassendeklarationen stehen (vgl. Kapitel 10). Anders als in C erfolgt die Definition von Variablen stets innerhalb einer Funktion oder einer Klasse. Dennoch können in Funktionen definierte Variablen einen globalen Gültigkeitsbereich besitzen. Wie in C und C++ ist es *nicht* möglich, Funktionen *lokal*, d.h. innerhalb anderer Funktionen zu definieren.

Das nun folgende unvermeidliche „Hello World“-Beispiel zeigt den typischen Programmaufbau:

```
/* BOB+ Hello World example */

// write a string to console
printString(str; i, n)
{
   n = strlen(str);
   for (i=0; i<n; i++)
      putc(str[i], stdout);
}

main()
{
    printString(„Hello World – this is BOB+\n“);
    return 0;
}
```

Wie man sieht, besteht eine große syntaktische Ähnlichkeit zu C/C++.

Zunächst ist zu sagen, dass BOB+ (wie auch BOB) formatfrei ist. Das bedeutet, dass Einrückungen, Leerzeilen, Zeilenumbrüche usw. keine syntaktische Bedeutung haben. Sie dienen lediglich der besseren Lesbarkeit des Quelltextes. Die Länge einer einzelnen Quelltextzeile darf 300 Zeichen (in BOB sind es 80) nicht überschreiten.

Zeilen 1 und 3 in obigem Beispiel enthalten Kommentare. Danach wird eine Funktion *printString* definiert (siehe Abschnitt 9), welche eine Zeichenkette im Parameter *str* übernimmt und deren Inhalt zeichenweise auf die Konsole ausgibt. Hierbei verwendet sie die vordefinierten Funktionen[6] *strlen* (Länge der Zeichenkette bestimmen) und *putc* (Dateiausgabe eines Zeichens) sowie die ebenfalls vordefinierte Variable *stdout*, die einen Verweis auf die Standardausgabe – üblicherweise die Konsole – darstellt.
Die im Funktionskopf hinter dem Semikolon aufgeführte Liste (i, n) definiert lokal innerhalb der Funktion gültige Variablen.

[5] Das dynamische Einbinden von Quelltextabschnitten mit der compile-Funktion ist möglich.
[6] siehe hierzu Abschnitt 11

Die Einsprungfunktion *main* ruft nun *printString* mit der auszugebenden Zeichenkette als Argument auf und gibt anschließend mit Hilfe der return-Anweisung den Wert 0 zurück.
In diesem Zusammenhang ist auf die besondere Bedeutung des Rückgabewertes der *main*-Funktion hinzuweisen. Da es innerhalb des Programmes keinen Aufrufer dieser Funktion gibt, wird der Wert an das Betriebssystem selbst (unter DOS als Errorlevel) zurückgegeben[7]. Daraus ergibt sich die Einschränkung, dass der Rückgabewert nur eine Ganzzahl sein kann. Aus diesem Grunde prüft BOB+ nach Beendigung der *main*-Funktion zunächst den Typ des Rückgabewertes[8]. Handelt es sich um einen Integer-Wert, so wird dieser, andernfalls 0 zurückgegeben.

4 Syntaktische Grundelemente

In diesem Abschnitt werden eine Reihe syntaktischer Grundelemente erklärt, die für das Schreiben von BOB+-Programmen von Bedeutung sind. Bitte beachten Sie, dass es sich hierbei um keine vollständige Syntaxbeschreibung handelt. Vielmehr sollen die wichtigsten Sprachelemente im Vergleich zu C/C++ dargestellt werden.

4.1 Kommentare

BOB+ kennt drei Arten von Kommentaren:

- Kommentare in C-Stil:
  Solche Kommentare haben die Form `/* Kommentartext */`.
  Sie können sich über mehrere Zeilen erstrecken, dürfen jedoch nicht verschachtelt werden.
- Kommentare im C++-Stil:
  Diese Kommentare werden in der Form `// Kommentartext`
  notiert. Sie reichen jeweils bis zum Ende der aktuellen Quelltextzeile.
- Mit @ eingeleitete Kommentare:
  Diese Form des Kommentars dient der Einbettung von BOB+-Quellen in Komando-Scripts (Batch-Dateien) unter DOS. Für BOB+ selbst ist sie identisch mit den C++-Kommentaren. Da das Zeichen @ in Batch-Scripts aber eine gültige Kommandozeile einleitet, deren Ausgabe lediglich unterdrückt wird, lässt sich diese Art des Kommentars verwenden, um BOB+-Code in Stapeldateien einzubinden. Ein Beispiel hierzu findet sich in Abschnitt 4.5.

4.2 Bezeichner

Für Bezeichner gelten die gleichen Regeln wie in C bzw. C++:
Bezeichner bestehen aus einer Folge von Buchstaben (ohne Umlaute und ß) und Ziffern bzw. dem Unterstrich (_). Sie dürfen nicht mit einer Ziffer beginnen und eine Gesamtlänge von 200 Zeichen[9] nicht überschreiten. Die Groß- bzw. Kleinschreibung ist signifikant.

[7] Das gilt nur für BOB+. In BOB ist der Rückgabewert undefiniert.
[8] Genau genommen wird nicht der Rückgabewert sondern der letzte auf dem internen Stack liegende Wert verwendet. Wird *main* nicht mit einer *return*-Anweisung verlassen, so ist der an das Betriebssystem übergebene Wert mehr oder weniger zufällig.
[9] BOB gestattet maximal 50 Zeichen.

Beipiele für Bezeichner:

```
MyClass
myValue
_anInternalVariable
p1
```

Generell sollten Bezeichner so gewählt werden, dass sie das spätere Lesen eines Quelltextes erleichtern. Das bedeutet insbesondere, dass Bezeichner möglichst einen direkten Schluss auf ihren Verwendungszweck zulassen sollten.

4.3 Literale

Literale sind direkt im Quellcode notierte Werte. BOB+ kennt hiervon vier Typen:

Zeichenliterale

Ein Zeichenliteral ist ein einzelnes Zeichen mit einem Zeichencode zwischen 0 und 255, das in Hochkommata geklammert ist, z.B. `‘A‘,‘x‘,‘ä‘,‘7‘,‘\n‘.`

Das letzte Zeichen weist dabei eine Besonderheit auf. Es repräsentiert kein druckbares Zeichen sondern ein Steuerzeichen. Die Darstellung spezieller Zeichen wird stets mit einem Backslash (\ , oft auch Escape-Zeichen genannt) eingeleitet. Tabelle 1 listet die von BOB+ interpretierten Escape-Codes mit ihrer Bedeutung auf.

Escape-Code	Bedeutung	Zeichencode (Hex)
\a	Alarmton	07
\b	Backspace	08
\f	Seitenvorschub	0C
\n	Zeilenvorschub *	10
\r	Wagenrücklauf	0D
\t	Tabulator horizontal *	09
\v	Tabulator vertikal	0B
\\	Backslash *	5C
\‘	Apostroph *	27
\“	Anführungszeichen *	22

Tabelle 1 Von BOB+ unterstützte Escape-Codes[10]

Achtung: Anders als in C/C++ interpretiert BOB+ keine hexadezimal oder oktal angegebenen Zeichencodes in Escape-Sequenzen. Ein Konstrukt wie ‘\0xFF‘ ist also nicht das Zeichen mit dem Code 255 sondern ein ungültiger Ausdruck.

Intern werden Zeichenliterale wie Ganze Zahlen behandelt, wobei der Zeichencode dem Zahlenwert entspricht. Dementsprechend können sie im Quelltext überall dort verwendet werden, wo auch eine Ganze Zahl stehen kann.

Zeichenkettenliterale

Zeichenkettenliterale sind Folgen beliebiger druckbarer Zeichen oder Escape-Codes (vgl. Tabelle 1), die in Anführungszeichen eingeschlossen sind. Kommt das Anführungszeichen selbst in einer

[10] Die mit * gekennzeichneten Codes werden auch von BOB unterstützt.

Zeichenkette vor, so muss es als Escape-Code angegeben werden. Implementierungsbedingt dürfen Zeichenkettenliterale eine Länge von 200 Zeichen (bei BOB 50 Zeichen) nicht überschreiten.

Beispiele für Zeichenkettenliterale:

```
“Hello World!“
“Er sagte \“Ich bin müde\“.“
“Adresse:\n\tName:\n\tStrasse:\n\tOrt:\n“
“c:\\prog\\bobplus\\bp –c hello –o hello.bpm“
```

Ganzzahlige Literale:

Ganzzahlige Literale sind Darstellungen Ganzer Zahlen in dezimaler, oktaler oder hexadezimaler Schreibweise, denen optional ein Vorzeichen (+ oder -) vorangestellt sein kann. Dabei werden Oktalzahlen – wie in C – dadurch gekennzeichnet, dass die erste Ziffer eine Null ist und die zweite Ziffer im Bereich zwischen 1 und 7 liegt. Hexadezimalzahlen beginnen mit 0x. Alle anderen Zahlen sind Dezimalzahlen. Wegen der internen Darstellung als 32-Bit-Integer[11] haben ganzzahlige Literale einen Wertebereich von –2^31 bis +2^31-1.

Beispiele für ganzzahlige Literale:

```
17
-234567
0x7FFFFF
021
0x2dff0
```

Fließkommaliterale

Fließkommaliterale sind Darstellungen rationaler Zahlen nach der Syntax
`[+|-] digit[.digit {digit}][e|E [+|-] digit{digit}]`.

Dabei ist *digit* eine Dezimalziffer. Die interne Darstellung erfolgt mit einfacher Genauigkeit (4 Byte, wie Typ float in C), wodurch die Genauigkeit auf 7 Stellen der Mantisse begrenzt ist und der Wertebereich zwischen -3.4E38 und 3.4E+38 liegt.

Beispiele für Fließkommaliterale:

```
3.141593
-123.4567E23
123e-4
-1.234567e-2
2.5E+6
3E14
```

4.4 Definitionen, Anweisungen und Blöcke

Wie bereits im Abschnitt 3 erwähnt, besteht ein BOB+-Programm auf der obersten Quelltextebene im wesentlichen aus Klassen- und Funktionsdefinitionen (hinzu kommen noch Verarbeitungsanweisungen – siehe 4.5).

Jede dieser Definitionen besteht aus einem *Kopf*, der u.a. den Namen der jeweiligen Codeobjekts beinhaltet sowie einem *Block*, welcher den eigentlichen Inhalt einschließt.
Ein Block ist eine Zusammenfassung von aufeinander folgenden *Anweisungen* und (optional) *Variablendeklarationen*. In BOB+ wird ein Block – wie u.a. in C/C++ - durch geschwungene Klammern ({ und }) begrenzt.

[11] BOB arbeitet mit 16-Bit-Integer-Werten. Dementsprechend geht der Wertebereich von –2^15 bis + 2^15-1. Außerdem sind nur dezimale Zahldarstellungen zulässig.

Jede in einem Block enthaltene Anweisung oder Definition muss – ebenfalls wie in C/C++ - mit einem Semikolon (;) abgeschlossen[12] werden. Überall dort, wo syntaktisch eine Anweisung vorgesehen ist, kann wiederum ein Block stehen. Dies ist besonders im Zusammenhang mit den Steuerstrukturen (siehe Abschnitt 8) von Bedeutung.

Nachfolgend werden die beschriebenen Elemente anhand eines etwas ausführlicheren Beispiels veranschaulicht. Es handelt sich dabei um ein Programm, welches die zeilenweise Eingabe von Zeichenketten (z.B. Namen) erwartet, sie in einer sortierten Liste speichert und am Ende den Listeninhalt wieder ausgibt. Das Beispiel geht über das bisher Besprochene hinaus, indem Gebrauch von Elementen (Klassen, Funktionen, Steuerstrukturen) Gebrauch gemacht wird, deren Erläuterung Gegenstand späterer Abschnitte ist. Es ist jedoch ein „typisches“ BOB+-Programm und sollte – einige Programmierkenntnisse in anderen Sprachen vorausgesetzt – leicht nachvollziehbar sein.

```
/* sorted stringlist example */

// ================================================================
// ListEntry class declaration
class ListEntry
{
   ListEntry(str);      // constructor
   ~ListEntry();        // destructor
   getValue();          // method for reading the value
   getNext();           // get pointer to next entry
   setNext(entry);      // set pointer to next entry

   _strValue;           // value of the entry
   _next;               // pointer to next entry
}
// ================================================================
// Implementation of ListEntry class

// constructor
ListEntry::ListEntry(str)
{
   _strValue = str;      // initialize value
   _next = null;         // initially _next points to nothing
}

// get value method
ListEntry::getValue()
{
   return _strValue;             // simply return current value
}

// destructor
ListEntry::~ListEntry()
{
   if (_next)
      delete _next;              // destroy next entry
}

// get next method
ListEntry::getNext() { return _next; }
// set next method
ListEntry::setNext(entry) { _next = entry; }
```

[12] Das ist ein feiner aber wichtiger Unterschied zu Sprachen wie z.B. PASCAL, wo das Semikolon die Anweisungen nicht abschließt sondern *trennt*. In BOB/BOB+ muss das Semikolon auch hinter der letzten Anweisung eines Blocks oder vor einem Schlüsselwort geschrieben werden.

```
// ============================================================
// Declaration of List class
class List
{
   List();                      // constructor
   ~List();                     // destructor
   addEntry(str);               // add entry to list
   printList();                 // print entries to console
   _first;                      // pointer to first ListEntry object
}

// ============================================================
// Implementation of List class

// constructor
List::List()
{
   _first = null;               // new list is empty
}

// destructor
List::~List()
{
   if (_first)
      delete _first;                // delete contents
}

// add new entry to sorted list
List::addEntry(str;newentry,p)       // addEntry uses local variables
                                     // newentry and p
{
   newentry = new ListEntry(str);    // create new entry
   if (!_first)                      // nothing to compare in empty list
      _first = newentry;             // single statement
   else
   {                                 // statement block
      if (strcmp(_first->getValue(),str) > 0)
      {
         newentry->setNext(_first);  // -> operator references a
                                     // member function
         _first = newentry;
      }
      else
      {
         // Walk through the list until correct position found.
         // Variable p is used like a pointer.
         for(p=_first;p->getNext();p=p->getNext())
         {
            if (strcmp(p->getNext()->getValue(),str) > 0)
            {
               newentry->setNext(p->getNext());
               break;               // insertion is done so return
            }
         }
         p->setNext(newentry);       // close chain of entries
      }
   }
}

// print sorted values to console
List::printList(;p)                  //p is a local variable
{
   p=_first;
   while ( p != null)
```

```
   {
      print(p->getValue(),"\n");

      p = p->getNext();
   }
}

// ============================================================
// programs entrypoint
main(;s,nameList)
{
   nameList = new List();
   // read text lines from console until empty line was read
   while (1)       // loops forever
   {
      print("Name: ");
      s = gets();
      if (strlen(s) == 0)
         break;    // break un empty string
      nameList->addEntry(s);
   }
   print("---- sorted -----\n");
   nameList->printList();
   delete nameList             // free list
   return 0;                   // return code 0 to OS
}
```

Bitte beachten Sie, dass das Beispiel Spracherweiterungen und vordefinierte Funktionen verwendet, die in BOB nicht zur Verfügung stehen.

4.5 Verarbeitungsanweisungen

Verarbeitungsanweisungen sind eine mit BOB+ eingeführte Spracherweiterung. Obgleich BOB+ keinen Präprozessor besitzt, haben sie eine gewisse Ähnlichkeit zu Präprozessorbefehlen in C/C++ und werden auch so notiert.

Verarbeitungsanweisungen werden – im Gegensatz zu allen anderen Anweisungen – vom Compiler während der Übersetzung des Quellcodes und nicht vom Bytecode-Interpreter ausgeführt. Sie sind syntaktisch außerdem nur auf der äußersten Quelltextebene – also außerhalb aller Klassen- und Funktionsdefinitionen – zugelassen und werden ***nicht*** mit einem Semikolon abgeschlossen.

Die aktuelle Version von BOB+ kennt folgende Verarbeitungsanweisungen:

#use

Die #use-Anweisung bindet eine vorkompilierte Bytecode-Bibliothek in das aktuelle Programm ein. Die in der Bibliothek enthaltenen Symbole werden damit in das Programm übernommen. Das bedeutet, dass beim Übersetzen eines Programmes, welches mit #use eine Bibliothek einbindet, Bytecode entsteht, der den kompletten Inhalt der Bibliothek mit enthält.

Verwendung: `#use “filename.ext“`

Dem Dateinamen kann ein relativer oder absoluter Pfad vorangestellt sein. BOB+ sucht die angegebene Datei zunächst im aktuellen Verzeichnis. Hat die Suche dort keinen Erfolg, so werden zusätzlich die in der Umgebungsvariablen PATH angegebenen Verzeichnisse durchsucht.

Anwendungsbeispiel:

Eine Bibliothek soll den Namen tstfunc.bp haben und eine Funktion mit dem Namen testfunc exportieren.

```
/* using #use example – module tstfunc.bp */
testfunc(n)
{
   print(“testfunc(“,n,“) from tstfunc.bp\n“);
}
```

Die Bibliothek wird nun in Bytecode übersetzt:

```
bp –c tstfunc –o tstfunc.bpm
```

... und von nachfolgendem Programm benutzt:

```
/* using #use example – main module */

#use “tstfunc.bpm“

main()
{
   testfunc(“Hello World“);
}
```

#defvar

Mit #defvar können initialisierte globale Variablen außerhalb von Funktionen definiert werden.

Verwendung: `#defvar varname value`

Dabei ist *varname* ein gültiger Bezeichner und *value* der zugehörige Wert. Bitte beachten Sie, dass die Initialisierung des Variablenwerts zur Übersetzungs- und nicht zur Laufzeit erfolgt. Deshalb können für *value* nur Literale angegeben werden.

Beispiele:

```
#defvar PI 3.141593
#defvar Version “2.1.0“
#defvar maxSize 2048
```

#endsrc

Die #endsrc Anweisung kennzeichnet das (vorzeitige) Ende eines BOB+-Quelltextes, oder anders ausgedrückt, jeglicher Text hinter #endsrc wird vom Compiler ignoriert. Der Anwendungszweck besteht vor allem in der Einbindung von BOB+-Quellen in Batch-Scripts (vgl. 4.1). Auch hierzu ein Beispiel. Das folgende Programm ist ein Batch-Script mit dem Namen graph.bat, das BOB+-Quelltext enthält:

```
@bp graph.bat
@goto end

// global settings
#defvar sizeX 240
#defvar sizeY 64
#defvar graphMode 4
#defvar textMode 7

main(;px,py,ms,x,n,x2,y2,y,r)
{
   px=sizeX;
   py=sizeY;
   setscrmode(graphMode);              // switch to graph mode
   ms = timer();
   n=0;
```

```
  for (x=0;x<px;x+=2)
  {
     n++;
     x2= px -x;
     line(x,0,x2,py-1,1);
  }
  for (y=0;y<py;y+=2)
  {
     n++;
     y2= py-1 -y;
     line(0,y,px-1,y2,2);
  }
  ms = timer() - ms;
  r = n*1000 / ms;
   gets();
   setscrmode(textMode);                // switch back to textmode
   print(r, " lines per second\n");
}
#endsrc
:end
```

Das Programm füllt den Bildschirm mit Linien und misst die dafür benötigte Zeit. Am Ende wird die Zeichengeschwindigkeit (Linien je Sekunde) ausgegeben.

Die ersten beiden Zeilen enthalten Anweisungen für den DOS-Kommandoprozessor. Das vorangestellte @-Zeichen unterdrückt unter DOS das Bildschirmecho und kennzeichnet die Zeilen unter BOB+ als Kommentare. Der BOB+-Quelltext endet mit der #endsrc Anweisung.

Die letzte Zeile gehört wieder dem Kommandoprozessor – es ist die das Programmende kennzeichnende Sprungmarke.

5 Variablen und Datentypen

BOB+ (und auch BOB) ist eine Sprache, die gewöhnlich als *nicht* oder *schwach* typisiert bezeichnet wird. Etwas genauer gilt folgendes:

In BOB+ haben nicht die Variablen sondern allein die Werte einen Datentyp.

5.1 Variablen

Variablen sind demnach einfach Ablageorte für einen beliebigen Wert. Da sie selbst über keine Typeigenschaften verfügen, ist es i.a. auch nicht erforderlich, sie explizit[13] zu deklarieren. Eine Variable entsteht im einfachsten Fall automatisch durch Zuweisung eines Wertes an einen frei wählbaren Bezeichner.

Der Gültigkeitsbereich einer Variablen ist – sofern keine besonderen Vorkehrungen getroffen werden – *global*, d.h. auf eine Variable kann, gleichgültig wo sie definiert wurde, von jeder Stelle des Programmes zugegriffen werden. Dies kann bei größeren Programmen zu unerwünschten Seiteneffekten führen, wenn man versehentlich an verschiedenen Stellen ein- und dieselbe Variable zu unterschiedlichen Zwecken benutzt. Hierzu ein Beispiel:

```
// calculate n!
fac(n)
{
   if (n==0)
     return 1;
   for(f=n, i=n-1; i>1; i--)
      f*=i;
   return f;
}
```

[13] Etwas anders verhält es sich mit Member-Variablen von Klassen – siehe hierzu Abschnitt 10.1.

```
main()
{
   for (j=1;j<=10;j++)
     print("fac(",j,")=",fac(j),"\n");
}

/* Variable i in this version of main will be modified by fac.
main()
{
   for (i=1;i<=10;i++)
     print("fac(",i,")=",fac(i),"\n");
}
*/
```

Wenn Sie die hier auskommentierte Variante der main-Funktion aktivieren, berechnet das Programm bis in alle Ewigkeit die Fakultät von 1. Der Grund dafür ist, dass sowohl fac als auch main die globale Variable i verwenden (ohne etwas von einander zu „wissen“).

Um dieses Problem zu beheben, besteht die Möglichkeit, Variablen mit lokaler Gültigkeit bezüglich einer Funktion zu verwenden. Solche Variablen *müssen* am Ende des Funktionskopfes, durch ein Semikolon von der Parameterliste getrennt, *deklariert werden*. Nachfolgend wird obiges Beispiel entsprechend modifiziert:

```
/* using function-level variables */
// calculate n!
fac(n;i,f,n)
{
   if (n==0)
     return 1;
   for(f=n,i=n-1; i>1; i--)
      f*=i;
   return f;
}

main(;i)
{
   for (i=1;i<=10;i++)
     print("fac(",i,")=",fac(i),"\n");
}
```

Wie man sieht, werden hier ausschließlich lokale Variablen verwendet. Wenn Sie wiederverwendbare und gut wartbare Programme schreiben wollen, sollten Sie globale Variablen so weit wie möglich vermeiden – und wenn sie schon unbedingt notwendig sind, gut dokumentieren..

5.2 Datentypen

In BOB+ werden alle in einem Programm vorkommenden Werte in sogenannten Value-Objekten abgelegt. Ein solches Objekt besitzt im wesentlichen eine Typinformation sowie den jeweils *zugeordneten Wert selbst* oder eine *Referenz* darauf.. Dementsprechend kann man zwischen Werttypen und Referenztypen unterscheiden[14]. Diese Unterscheidung ist für die Verwendung in zweierlei Hinsicht von Bedeutung:

- Bei der Zuweisung an Variablen werden Werttypen vollständig kopiert, bei Referenztypen dagegen nur der Verweis auf den Datenbereich. Deshalb betreffen hier Datenänderungen *alle* Variablen, die auf den jeweiligen Wert verweisen.

[14] Technisch gesehen sind die Typen, deren Wert nicht mehr als 4 Bytes benötigt Wert-, alle anderen Referenztypen. Hier wird der eigentliche Datenbereich dynamisch erzeugt und im Value-Objekt nur ein Zeiger darauf abgelegt.

Ein Beispiel:

```
main(;s1,s2)
{
   s1 = “Hello“;              // initialize s1 as string
   s2 = s1;                   // assign variable s1 to s2
   s2[1] = ‘a‘;               // modify second character in s2
   print(s1,“\n“);            // print value of s1
}
```

Hier wird „Hallo“ und nicht – wie man vielleicht vermuten könnte – „Hello“ ausgegeben.

- Die erwähnten Value-Objekte und mit ihnen die Daten der Werttypen werden auf dem Stack angelegt und mit demselben aufgeräumt. Anders verhält es sich mit den Referenztypen. Deren Datenbereich muß (bzw. sollte) – sofern sie zur Laufzeit des Programms erzeugt werden – vom Benutzer wieder freigegeben werden.

5.2.1 Werttypen

Werttypen sind in BOB+ die numerischen Typen sowie der Typ null.

Datentyp int:

Der Datentyp int repräsentiert eine Ganze Zahl mit 32 Bit Breite[15]. Daraus ergibt sich ein Wertebereich von -2^31 bis +2^31-1. Werte dieses Typs werden auch zur Darstellung einzelner Zeichen und boolescher Werte verwendet.

Zulässige Operationen für den Typ int sind:

- Zuweisung an eine Variable
- Verwendung in numerischen Ausdrücken
- Verwendung in string-Ausdrücken
- Bitoperationen
- numerische Vergleiche
- Verwendung als Funktionsparameter

Werte vom Typ int können in numerischen Ausdrücken und Vergleichen ohne explizite Typkonvertierung mit Werten des Typs float kombiniert werden. In diesem Fall erfolgt eine implizite Umwandlung des gesamten Ausdrucks nach float.
Bei Verwendung von int-Werten in string-Ausdrücken (Verkettung) wird der int-Wert als Zeichencode interpretiert.

Datentyp float

Der Datentyp float ist nur in BOB+ (nicht in BOB) definiert. Er repräsentiert eine Rationale Zahl mit einfacher Genauigkeit (32 Bit). Der Wertebereich liegt zwischen -3.4E38 und 3.4E+38 bei einer numerischen Genauigkeit von 7 Stellen.

Zulässige Operationen für den Typ float sind:

- Zuweisung an eine Variable
- Verwendung in numerischen Ausdrücken
- numerische Vergleiche
- Verwendung als Funktionsparameter

[15] In BOB sind int-Werte nur 16 Bit breit.

Werte vom Typ float können in numerischen Ausdrücken und Vergleichen ohne explizite Typkonvertierung mit Werten des Typs int kombiniert werden. In diesem Fall erfolgt eine implizite Umwandlung des gesamten Ausdrucks nach float.

Datentyp null

Der Datentyp null kennzeichnet einen nicht initialisierten Wert.. Im Quelltext eines BOB+-Programmes werden Daten dieses Typs durch das Schlüsselwort **null**[16] beschrieben, das man auch als Konstante und einzigen möglichen Wert des Typs null auffassen kann.. Außerdem können eine Reihe vordefinierter Funktionen null zurückgeben.

Zulässige Operationen für den Typ null sind:

- Zuweisung an eine Variable
- Verwendung in numerischen Ausdrücken
- Verwendung in Vergleichen
- Verwendung als Funktionsparameter

Werte vom Typ null können in Vergleichen ohne explizite Typkonvertierung mit allen anderen Typen kombiniert werden. Dabei ist der Wert **null** kleiner als jeder andere Wert.

5.2.2 Referenztypen

Alle Datentypen in BOB+, die keine Werttypen sind, sind Referenztypen..

Datentyp string

Der Datentyp string repräsentiert eine Zeichenkette. Werte dieses Typs können mit Hilfe von Zeichenkettenliteralen (vgl. 4.3) oder durch die vordefinierte Funktion *newstring* (vgl. 11.1) explizit erzeugt werden. Darüber hinaus entstehen sie bei der Anwendung des Verkettungsoperators (+) sowie als Rückgabewert einiger weiterer vordefinierter Funktionen. Bitte beachten Sie, dass alle Werte des Typs string, die nicht durch Literale erzeugt wurden, mit Hilfe der Funktion *free* wieder freigegeben werden sollten, sobald sie nicht mehr benötigt werden.

Zulässige Operationen für den Typ string sind:

- Zuweisung an eine Variable
- Verwendung in string-Ausdrücken
- Verwendung in Vergleichen
- Verwendung als Funktionsparameter
- Zugriff auf einzelne Zeichen mit Hilfe des Operators []

Datentyp FILE

Der Datentyp FILE ist ein Verweis auf eine geöffnete Datei. Instanzen dieses Typs werden mit der vordefinierten Funktion *fopen* erzeugt und mit *fclose* wieder freigegeben. In BOB+ (und auch in

[16] Zur Wahrung der Abwärtskompatibilität zu BOB existiert außerdem das Sclüsselwort **nil**, das gleichbedeutend mit **null** ist.

BOB) existieren – wie in C – drei vordefinierte Konstanten des FILE-Typs, die auf die Standardein- und –ausgabedateien des Systems verweisen:

- *stdin*: Standard-Eingabe
- *stdout*: Standard-Ausgabe
- *stderr*: Standard-Fehlerausgabe

Diese Standarddateien können *nicht* mit *fopen* bzw. *fclose* geöffnet oder geschlossen werden.

Zulässige Operationen für den Datentyp FILE sind:

- Zuweisung an eine Variable
- Verwendung in Vergleichen
- Verwendung als Funktionsparameter (insbesondere bei vordefinierten I/O-Funktionen (siehe Abschnitt 11.3)

Datentyp vector

Der Datentyp vector repräsentiert ein Datenfeld fester (d.h. bei der Erzeugung festgelegter) Größe, dessen Elemente beliebige (auch typverschiedene) Werte sein können. Instanzen dieses Typs werden mit den vordefinierten Funktionen *newvector* oder *Vec* bzw. *T* erzeugt und mit *free* wieder freigegeben.

Zulässige Operationen für den Datentyp vector sind:

- Zuweisung an eine Variable
- Verwendung als Funktionsparameter
- Zugriff auf einzelne Elemente mit Hilfe des Operators []

Datentyp function

Der Datentyp function repräsentiert eine innerhalb des Programmes aufrufbare Funktion. Intern werden vordefinierte und benutzerdefinierte (d.h. selbst in BOB+ geschriebene) Funktionen nochmals unterschieden[17]. Aus Benutzersicht ist diese Unterscheidung nicht notwendig. Eine explizite Instanzenbildung von Funktionen ist nicht möglich. Von einer Funktion existiert stets genau eine Instanz, die für vordefinierte Funktionen bei der Initialisierung von BOB+ und für benutzerdefinierte Funktionen während der Übersetzung der Funktionsdeklaration (siehe Abschnitt 9) erzeugt wird.

Zulässige Operationen für den Datentyp function sind:

- Zuweisung an eine Variable (vgl. Abschnitt 9.4)
- Aufruf
- Verwendung als Funktionsparameter

Datentyp class

Der Datentyp class repräsentiert eine innerhalb des Programmes verfügbare Klasse (vgl. 10.1). Klassen sind Vorlagen für konkrete Objekte, Deshalb ist eine explizite Instanzenbildung von

[17] Deshalb liefert die Funktion gettypename (siehe 11.2) auxh den Typnamen „code“ für vordefinierte und „bytecode“ für benutzerdefinierte Funktionen.

Klassen ist nicht möglich. Von einer Klasse existiert stets genau eine Instanz, während der Übersetzung der Klassendeklaration erzeugt wird.

Zulässige Operationen für den Datentyp class sind:

- Aufruf in der Klasse definierter statischer Funktionen
- Verwendung als Funktionsparameter in den RTTI-Funktionen (siehe Abschnitt 11.2)
- Verwendung als Vorlage für die Bildung von Objektinstanzen mit dem Operator *new*

Datentyp object

Der Datentyp object repräsentiert konkrete Instanzen einer Klasse (vgl 10.2). Objekte werden mittels des Operators *new* aus Vorlagen (Typ class) erzeugt und müssen mit dem Operator *delete* oder der vordefinierten Funktion *free* explizit wieder freigegeben werden[18].

Zulässige Operationen für den Datentyp object sind:

- Aufruf von Member-Funktionen
- Zuweisung an eine Variable
- Verwendung als Funktionsparameter

6 Operatoren

Das Konzept der Operatoren in BOB+ (und auch BOB) entspricht weitgehend dem von C. Dieser Abschnitt beschränkt sich daher auf eine kurze Referenz der in BOB+ verfügbaren Operatoren und vorhandener Unterschiede zu C. Für Klassen (bzw. Objekte) können einige der Operatoren neu- bzw. umdefiniert werden – siehe hierzu Abschnitt 10.7.

6.1 Arithmetische Operatoren

Die arithmetischen Operatoren realisieren in Verbindung mit numerischen Operanden (Datentypen *int* und *float*) die vier Grundrechenarten. Der Additionsoperator ist darüber hinaus auch auf Zeichenketten (Datentyp string) anwendbar. In diesem Fall dient er als Verkettungsoperator.

BOB+ kennt folgende arithmetischen Operatoren:

Operator	Bedeutung	Beispiel
+ (unär)	positives Vorzeichen	+a
- (unär)	negatives Vorzeichen	-a
*	Multiplikation	a * b
/	Division	a / b
%	Divisionsrest (Modulo)	a % b
+	Addition	a + b
-	Subtraktion	a – b

Tabelle 2 Arithmetische Operatoren in BOB+

[18] Die Möglichkeit der Freigabe besteht nur in BOB+. Bei Freigabe mit *delete* wird – im Gegensatz zu *free* – ein ggf. vorhandener Destruktor automatisch aufgerufen.

Hinweise:

- Das Verhalten des Divisionsoperators hängt von den Datentypen der Operanden ab. Haben beide Operanden den Typ *int*, so wird eine ganzzahlige Division durchgeführt, hat mindestens ein Operand den Typ *float*, eine Fließpunkt-Division.
- Der Modulo-Operator kann nur auf Operanden des Typs *int* angewendet werden.
- Ist ein Operand des Additionsoperators (+) vom Typ *string*, so wirkt der Operator als Verkettungsoperator. Der andere Operand muss dann entweder auch vom Typ *strring* oder vom Typ *int* sein. Im letztgenannten Fall wird das niederwertigste Byte des *int*-Operanden als Code eines einzelnen Zeichens interpretiert.

6.2 Vergleichsoperatoren

Die Vergleichsoperatoren – auch als relationale Operatoren bezeichnet – dienen dem Größenvergleich ihrer (stets zwei) Operanden.

BOB+ kennt folgende Vergleichsoperatoren:

Operator	Bedeutung	Beispiel
==	gleich	a == b
!=	ungleich	a != b
>	größer als	a > b
>=	größer oder gleich	a >=b
<	kleiner als	a < b
<=	kleiner oder gleich	a <= b

Tabelle 3 Vergleichsoperatoren in BOB+

Hinweise:

- Ist ein Operand vom Typ *int* und der andere vom Typ *float*, so wird der *int*-Wert vor dem Vergleich implizit nach *float* konvertiert. Dies kann zu Problemen insbesondere bei den Operatoren == und != führen.
- Ist einer der Operanden numerisch (*int* oder *float*), so muss auch der andere Operand numerisch sein.
- Zeichenketten (*string*) können mit Zeichenketten oder mit *null* verglichen werden.
- Werte aller anderen Referenztypen können mit Werten des selben Typs oder mit *null* verglichen werden. Sind beide Operanden vom Referenztyp, so werden intern die Adressen des Datenbereichs verglichen[19].
- Werte des Typs *null* sind kleiner als alle anderen Werte.

[19] Deshalb ist für Referenztypen i.a. nur die Verwendung der Operatoren == und != sinnvoll.

6.3 Logische Operatoren

Die logischen Operatoren verknüpfen Wahrheitswerte miteinander. BOB+ besitzt keinen eigenen Datentyp für Wahrheitswerte, statt dessen gelten folgende Regeln:

- Werte vom Typ *null* haben den Wahrheitswert *false.*
- Numerische Werte haben den Wahrheitswert *false*, wenn ihr Betrag 0 ist, sonst *true*.
- Alle anderen Werte haben den Wahrheitswert *true*.

Folgende logische Operatoren sind in BOB+ definiert:

Operator	Bedeutung	Beispiel
!	NOT	!a
&&	AND	a && b
||	OR	a || b

Tabelle 4 Logische Operatoren in BOB+

Hinweise:

- Der Operator ! liefert als Resultat 1 für *true* und 0 für *false*.
- Ausdrücke mit den binären Operatoren && bzw. || werden von links nach rechts ausgewertet. Die Auswertung bricht ab, sobald das Ergebnis des Ausdrucks feststeht. Der Rückgabewert ist dann derjenige Operand, der zum Abbruch der Auswertung führte. Deshalb dürfen logische Werte *nicht* implizit als *Integer*-Werte angesehen werden.

6.4 Bitoperatoren

Die Bitoperatoren sind nur auf Operanden des Typs *int* anwendbar. Sie realisieren Operationen auf den einzelnen Bits der Operanden.

Folgende Bitoperatoren sind in BOB+ definiert:

Operator	Bedeutung	Beispiel
~	bitweises NOT	~a
&	bitweises AND	a & b
|	bitweises OR	a | b
^	bitweises XOR	a ^ b
<<	bitweise Linksverschiebung	a << 3
>>	bitweise Rechtsverschiebung	a >>3

Tabelle 5 Bitoperatoren in BOB+

6.5 Zuweisungsoperatoren

Die Zuweisungsoperatoren dienen der Zuweisung eines Wertes an eine Variable. Wie in C/C++ wird die Zuweisung mit dem Operator = realisiert, der mit einigen arithmetischen Operatoren kombiniert werden kann. Nachfolgende Tabelle zeigt die Möglichkeiten:

Operator	Bedeutung	Beispiel
=	einfache Zuweisung	a = b
+=	Zuweisung mit Addition	a += b
-=	Zuweisung mit Subtraktion	a -= b
*=	Zuweisung mit Multiplikation	a *= b
/=	Zuweisung mit Division	a /= b
%=	Zuweisung mit Divisionsrest[20]	a %= b

Tabelle 6 Zuweisungsoperatoren in BOB+

Hinweise:

- Der einfache Zuweisungsoperator ist mit allen Datentypen verwendbar. Bitte beachten Sie, dass bei der Zuweisung von Werten der Referenztypen keine Kopie des Wertes sondern lediglich ein Verweis darauf zugewiesen wird.
- Die zusammengesetzten Zuweisungsoperatoren sind Kombinationen aus dem einfachen Zuweisungsoperator und einem arithmetischen Operator (vgl. 6.1), wobei das Zuweisungsziel als linker Operand der arithmetischen Operation dient. So ist etwa der Ausdruck `a += b` identisch mit `a = a + b`.
- Für die Operanden der zusammengesetzten Zuweisungsoperatoren gelten die gleichen Regeln wie für die der entsprechenden arithmetischen Operatoren.

6.6 Inkrement- und Dekrement-Operatoren

Die Inkrement- und Dekrement-Operatoren sind unär. Sie erhöhen (Operator ++) bzw. erniedrigen (Operator --) ihren Operanden jeweils um den Wert 1. Der Operand muss dabei eine mit dem Datentyp *int* initialisierte Variable sein.

Beide Operatoren können in Präfix- oder Postfix-Notation verwendet werden. Ein Unterschied zwischen diesen Notationen besteht bei der Verwendung innerhalb von zusammengesetzten Ausdrücken. Bei der Präfix-Notation ist der Wert des Teilausdrucks gleich dem Wert des Operanden nach dem Inkrement/Dekrement während bei der Postfix-Notation der Wert des Operanden vor dem Inkrement/Dekrement zurückgegeben wird.

Nachfolgende Tabelle soll die Varianten veranschaulichen:

Operator	Bedeutung	Beispiel	Entsprechung
++x	Inkrement (Präfix)	b = ++a	a = a+1, b = a
x++	Inkrement (Postfix)	b = a++	b = a, a = a+1
--x	Dekrement (Präfix)	b = --a * 7	a = a-1, b = a * 7
x--	Dekrement (Postfix)	b = a-- * 7	b = a * 7, a = a-1

Tabelle 7 Inkrement und Dekrement

[20] Nur in BOB+ definiert.

Hinweise:

- Die Inkrement- bzw. Dekrement-Operatoren erzeugen kürzeren und schnelleren Bytecode als ihre „ausgeschriebenen“ Entsprechungen (siehe Tabelle 7). Sie sind deshalb besonders für die Modifikation von Zählvariablen in Schleifen (vgl. Abschnitt 8) geeignet.
- In zusammengesetzten Ausdrücken sollten diese Operatoren mit Vorsicht und nur dann verwendet werden, wenn die etwas höhere Effizienz wirklich notwendig ist, da neben der schlechteren Lesbarkeit u.U. unerwünschte Seiteneffekte auftreten können. So wird beispielsweise in einem Ausdruck `c = a || (--b == 1)` die Variable b niemals dekrementiert, solange a *true* ergibt.

6.7 Sonstige Operatoren

Zugriffsoperator

Der Zugriffsoperator wird durch eckige Klammern [] ausgedrückt. Er dient dem indexbasierten Zugriff auf einzelne Elemente eines Vektors oder einer Zeichenkette.

Beispiel:

```
/* operator [] example */
main()
{
   vec = T(1,2,3,4);           // create a vector
   vec[1] = 7;                 // assign 7 to second element of vec
   print(vec[1],“\n“);         // prints 7 to console
   free(vec);
}
```

Der Operator ist in seiner vordefinierten Form nur auf Variablen der Typen *vector* und *string* anwendbar. Der Operand in der Klammer muss vom Typ *int* sein.

Klammern

Runde Klammern zählen ebenfalls zu den Operatoren. Sie werden in zwei Zusammenhängen verwendet:

- als Begrenzung der Argumentliste von Funktionen bei der Deklaration und beim Aufruf
  `myFunc(a, b, c)`
- zur Steuerung der Auswertungsreihenfolge von Ausdrücken – „Klammern rechnen wir zuerst aus!“
  `x = (a + b) * c`

Bedingungsoperator

Der Bedingungsoperator hat die allgemeine Form `condexpr ? expr1 : expr2`.

Dabei wird zunächst der Ausdruck *condexpr* (eine Bedingung) ausgewertet. Sein Resultat wird als Wahrheitswert interpretiert. Ist das Ergebnis *true*, so wird anschließend *expr1*, sonst *expr2* ausgewertet. Das jeweilige Resultat ist das Ergebnis des gesamten Ausdrucks.

Beispiel:

```
/* get minimum */
min(a, b)
{
   return a < b ? a : b;
}
```

Hinweis:

Der Typ des Rückgabewerts der Funktion in obigem Beispiel ist nicht unbedingt bekannt. Ist beispielsweise a vom Typ *int* und b vom Typ *float*, so wird der Bedingungsausdruck (`a<b`) temporär nach *float* konvertiert. Abhängig vom Ergebnis ist der Rückgabewert der Funktion vom Typ *int* oder *float*.

Komma-Operator

Der Komma-Operator gestattet die Verkettung von mehreren Ausdrücken zu einer Ausdrucksliste. Anders ausgedrückt: Mit Hilfe dieses Operators lassen sich mehrere Ausdrücke dort notieren, wo syntaktisch nur einer vorgesehen ist. Dabei ist das Ergebnis des Gesamtausdrucks (also der Liste) das des letzten Teilausdrucks.

Hinweis:

Der *Komma-Operator* ist nicht zu verwechseln mit dem *Trennzeichen Komma*, das in Parameter- oder Variablenlisten verwendet wird.

Operatoren new, delete, :: und ->

Diese Operatoren haben nur im Zusammenhang mit Klassen und Objekten Bedeutung. Sie werden in den Abschnitten 10.2 bis 10.4 ausführlicher erläutert und sind in nachfolgender Tabelle nur der Vollständigkeit halber aufgeführt.

Operator	Bedeutung	Beispiel
new	Objektinstanz aus Klasse erzeugen	a = new A()
delete[21]	Löschen einer Objektinstanz	delete a
::	Definition von Methoden, Aufruf von Klassenmethoden	A::A() { ... } i = A::getMaxId()
->	Aufruf von Instanzmethoden	name = a->getName()

Tabelle 8 Objektbezogene Operatoren in BOB+

[21] nur in BOB+ verfügbar

6.8 Hierarchie der Operatoren

Werden mehrere Operatoren innerhalb eines Ausdrucks verwendet, so ergibt sich die Frage nach der Reihenfolge ihrer Auswertung. Um eine solche Reihenfolge festlegen zu können, wird eine Rangfolge (Hierarchie) der Operatoren definiert. BOB/BOB+ hält sich dabei weitgehend an die Regeln von C/C++. In Tabelle 9 ist die Hierarchie der Operatoren in BOB+ aufgeführt. Dabei werden Operatoren höherer Priorität vor jenen niederer Priorität ausgewertet. Bei Operatoren der gleichen Prioritätsstufe erfolgt die Auswertung von links nach recht entsprechend der Notation.

Priorität	Operatoren
0	,
1	= += -= *= /= %=
2	?:
3	||
4	&&
5	|
6	^
7	&
8	= !=
9	< <= >= >
10	<< >>
11	+ -
12	* / %
13	! ~ ++x –-x new delete
14	() [] x++ x– ->

Tabelle 9 Operatorhierarchie in BOB+

7 Schlüsselwörter und reservierte Bezeichner

Schlüsselwörter sind im syntaktischen Sinne Terminalsymbole. Das bedeutet, dass sie bereits bei der syntaktischen Analyse identifiziert und speziellen internen Symbolen zugeordnet werden. Innerhalb eines Programms ist den Schlüsselwörtern eine feste (zentrale) Bedeutung zugeordnet. Deshalb können keine benutzerdefinierten Bezeichner definiert werden die den Schlüsselwörtern entsprechen.

Die Schlüsselwörter in BOB/BOB+ sind im wesentlichen eine Teilmenge der aus C/C++ bekannten. Nachfolgende Tabelle gibt eine Übersicht. Die darin mit (*) gekennzeichneten Schlüsselwörter sind nur in BOB+ definiert.

Schlüsselwort	Kategorie	Bedeutung
null (*)	Typkonstante	Bezeichner für den Datentyp *null* und dessen einziger Wert
nil (*)	Typkonstante	identisch mit **null**, zur Kompatibilität mit BOB
do	Steuerung	Konstruktion von Wiederholungsanweisungen
while	Steuerung	Konstruktion von Wiederholungsanweisungen
for	Steuerung	Konstruktion von Wiederholungsanweisungen
break	Steuerung	Abbruch einer Wiederholungsanweisung
continue	Steuerung	vorzeitiger Sprung zum Anfang einer Wiederholungs-anweisung
if	Steuerung	Konstruktion bedingter Anweisungen
else	Steuerung	Konstruktion bedingter Anweisungen
return	Steuerung	(vorzeitige) Rückkehr aus einer Funktion
class	OOP	Beginn einer Klassendeklaration
static	OOP	Deklaration eines Klassen-Members
new	OOP	Erzeugen einer Objektinstanz
delete (*)	OOP	Zerstören einer Objektinstanz

Tabelle 10 Schlüsselwörter in BOB+

Im Zusammenhang mit der objektorientierten Programmierung kennt BOB+ noch folgende reservierte Bezeichner (mitunter auch "*magic names*" genannt):

- **this** Verweis auf das aktuelle Objekt in Member-Funktionen[22]
- **dtor** Alias-Bezeichner für den Destruktor einer Klasse (siehe 10.2)
- **OP_CALL, OP_VREF, OP_VSET, OP_ADD, OP_SUB, OP_MUL, OP_DIV, OP_REM, OP_BOR, OP_BAND, OP_XOR, OP_SHL, OP_SHR**
  spezielle Funktionsnamen zur Definition von Operatoren (siehe 10.7)

Die reservierten Bezeichner sind *keine Schlüsselwörter* im oben beschriebenen Sinne. Sie besitzen jedoch implizit eine besondere Bedeutung im Programm und sollten deshalb nicht als allgemeine Bezeichner verwendet werden.

[22] auch in BOB

8 Steuerstrukturen

Für jedes Programm, das aus mehr als einer bloßen Aneinanderreihung von Befehlen bestehen soll, werden Möglichkeiten zur Steuerung des Programmablaufs benötigt. Grundsätzlich unterscheidet man hier zwischen *Verzweigungen des Programmablaufes auf Grund von Bedingungen* und *Wiederholungen von Programmabschnitten*.

Viele Programmiersprachen erlauben zudem noch unbedingte Sprunganweisungen. Solche Anweisungen sind in BOB/BOB+ nicht vorgesehen[23] und werden deshalb nicht weiter behandelt.

8.1 Bedingungen

Zur Steuerung der Programmabarbeitung in Abhängigkeit von einer Bedingung dient die ***if-else-Anweisung***, die in Syntax und Semantik dem gleichnamigen Konstrukt in C/C++ entspricht:
`if ( bedingung ) anweisung1 [ else anweisung2 ] .`

Ergibt der Ausdruck *bedingung* den Wahrheitswert *true*, so wird *anweisung1*, andernfalls *anweisung2* ausgeführt. Die auszuführenden Anweisungen können einfache Anweisungen oder Anweisungsblöcke sein. Insbesondere kann hierfür die *if-else-Anweisung* selbst verwendet werden, wodurch die Formulierung geschachtelter Bedingungen bzw. Fallunterscheidungen[24] möglich ist.

Beispiel:

```
main(;fp)
{
   fp = fopen(“testfile.txt“,“rt“);
   if (fp)                          // same as fp != null
   {
      // do anything
      fclose(fp);
   }
   else
     print(“opening testfile.txt failed\n“);
}
```

8.2 Wiederholungen (Schleifen)

Für die Formulierung von Wiederholungen von Codeabschnitten kennt BOB/BOB+ die Anweisungen *while*, *do-while* und *for*. Syntax und Semantik dieser Anweisungen entsprechen dabei den gleichnamigen Konstrukten in C/C++.

8.2.1 Die while-Anweisung

Die while-Anweisung hat die allgemeine Form

`while ( bedingung ) anweisung`

Dabei wird *anweisung* so lange (wiederholt) ausgeführt, wie *bedingung* den Wert *true* ergibt.

[23] Sie sind auch in keinem BOB+-Programm zwingend erforderlich.
[24] BOB/BOB+ kennt im Unterschied zu C/C++ keine switch-case-Anweisung. Deshalb müssen Fallunterscheidungen mit Hilfe geschachtelter if-else-Anweisungen formuliert werden.

Beispiel:

```
/* type contents of file to console */
main(;fp,s)
{
   fp = fopen(“testfile.txt“,“rt“);
   if (fp)                          // same as fp != null
   {
      while(!feof(fp))
      {
          s = fgets(fp);
          print(s);
          free(s);
      }
      fclose(fp);
   }
   else
     print(“opening testfile.txt failed\n“);
}
```

8.2.2 Die do-while-Anweisung

Die do-while-Anweisung hat die allgemeine Form

`do` *`anweisung`* `while (` *`bedingung`* `)`

Sie ist der while-Anweisung sehr ähnlich. Auch hier wird *anweisung* so lange ausgeführt, wie *bedingung* den Wert *true* ergibt. Der wesentliche Unterschied besteht darin, dass die Bedingung erst am Ende geprüft und damit *anweisung* stets mindestens einmal ausgeführt wird.

Beispiel:

```
/* prompt user for input until empty line was entered */
main(;s)
{
   s = null;
   do
   {
      if (s != null) free(s);
      print(“enter line: “);
      s = gets();
      // do anything
   }
   while (strlen(s) > 0);
   free(s);
}
```

8.2.3 Die for-Anweisung

Die for-Anweisung ist die mächtigste Form der Formulierung einer Wiederholung. Wie in C/C++ (und im Gegensatz zu Sprachen wie BASIC oder PASCAL) dient sie nicht zwingend der Abarbeitung einer festgelegten Anzahl von Schleifendurchläufen sondern ist eher eine Verallgemeinerung bzw. Erweiterung der *while*-Anweisung. Die for-Anweisung hat die allgemeine Form

`for (` *`init`* `;` *`bedingung`* `;` *`reinit`*`)` *`anweisung`*

Vor dem Beginn der Abarbeitung einer for-Schleife wird zunächst einmalig die Initialisierungsanweisung *init*[25] ausgeführt. Vor jedem Schleifendurchlauf wird der Ausdruck *bedingung* ausgewertet. Ergibt er *true*, so wird *anweisung* ausgeführt, andernfalls ist die Schleife

[25] Bitte beachten Sie, dass auch hier an Stelle einer einzelnen Anweisung jeweils eine kommaseparierte Anweisungsliste oder ein Anweisungsblock stehen kann.

beendet. Die Anweisung *reinit* wird am Ende jedes Schleifendurchlaufs (d.h. nach Ausführung von *anweisung*) ausgeführt.

Beispiel:

```
/* list all arguments of commandline to console */
main(;argvec,n,i)
{
   argvec = getargs();       // get argument vector
   n = vecsize(argvec);      // get number of elements
   for(i=0; i<n; i++)        // print each element to console
      print(argvec[i],"\n");
   free(argvec);             // free allocated memory of argvec
}
```

8.2.4 Geschachtelte Schleifen

Schleifen können ineinander geschachtelt werden. Ein typischer Fall für die Anwendung geschachtelter Schleifen ist das Füllen eines mehrdimensionalen Feldes, wie in nachfolgendem Beispiel gezeigt:

```
/* create and fill a 2-dim array */
main()
{
   rows = 8;
   cols = 8;
   /* create a table */
   array = newvector(rows);
   for (i=0; i<rows; i++) array[i] = newvector(cols);
   /* intialize table */
   for (i=0; i<rows; i++)
      for (j=0; j<cols; j++)
         array[i][j] = rows * i + j;
   // do anything
   // ...
   /* free array */
   for (i=0; i<rows; i++) free(array[i]);
   free(array);
}
```

Die maximale Schachtelungstiefe ist dabei in BOB+ auf 20, in BOB auf 10 Ebenen begrenzt.

8.2.5 break und continue

Die Anweisungen *break* und *continue* können innerhalb von Schleifen (und nur dort) als zusätzliche Hilfsmittel zur Ablaufsteuerung eingesetzt werden. Mit break kann eine Schleife vorzeitig (d.h. unabhängig von der eigentlichen Abbruchbedingung) abgebrochen werden. Mit continue wird der Rest des aktuellen Anweisungsblocks einer Schleife übersprungen und (sofern die Abbruchbedingung nicht erfüllt ist) direkt zum nächsten Schleifendurchlauf übergegangen.

Beispiel:

```
/* write all printable characters of testfile.txt to console */
main()
{
   fp = fopen(“testfile.txt“,“rt“);
   if (!fp)
   {
      print(“error opening file\n“);
      return 1;
   }
   while(1)            // loops forever
   {
      if (feof(fp))
         break;        // abort loop if end of file reached
      c = getc(fp);
      if ((c<‘ ‘) && (c != ‘\n‘))
         continue;     // ignore control characters other than EOL
      putc(c,stdout);  // write characrter to console
   }
   fclose(fp);
   return 0;
}
```

9 Funktionen

Wie bereits in Abschnitt 3 angedeutet, sind Funktionen die zentralen Strukturierungselemente eines jeden BOB+-Programmes. Grundsätzlich ist dabei zwischen zwei Typen von Funktionen, den *vordefinierten* und den *benutzerdefinierten* zu unterscheiden. Vordefinierte Funktionen (siehe Abschnitt 11) sind fester Bestandteil des BOB/BOB+-Laufzeitsystems. Sie sind in jedem Programm verfügbar und können vom Benutzer (d.h. Programmierer) normalerweise[26] nicht geändert werden.

Das eigentliche Verhalten eines Programms wird mit Hilfe der benutzerdefinierten Funktionen festgelegt (der Einsprungpunkt *main* – siehe Abschnitt 3 – ist selbst eine solche Funktion).

9.1 Funktionsdeklaration

Eine benutzerdefinierte Funktion hat die allgemeine Form

funcname (*parameterlist* ; *localvarlist*) *body*

Dabei ist *funcname* der Funktionsname, der ein gültiger Bezeichner (siehe 4.2) sein muss. *parameterlist* ist eine kommaseparierte Liste benannter Funktionsparameter und *localvarlist* definiert die Namen der innerhalb der Funktion gültigen weiteren lokalen Variablen (ebenfalls als kommaseparierte Liste). *body* schließlich ist ein Anweisungsblock, der den Funktionsrumpf enthält. Sowohl *parameterlist* als auch *localvarlist* sind optional. Fehlt *localvarlist*, so kann auch das als Trennzeichen zwischen den Listen verwendete Semikolon entfallen.

Benannte Parameter werden innerhalb des Funktionsrumpfes wie lokale Variablen behandelt. Alle Variablen, die innerhalb des Funktionsrumpfes verwendet werden und nicht in *parameterlist* oder *localvarlist* enthalten sind, haben – auch wenn sie nicht außerhalb der Funktion explizit deklariert wurden – globale Gültigkeit.

Anders als in C/C++ geben Funktionen in BOB/BOB+ *immer* einen Funktionswert zurück. Wird eine Funktion mit return, gefolgt von einem Argument, verlassen, so ist der Wert dieses Arguments der Funktionswert. Ansonsten ist der Wert undefiniert – genau genommen der Wert, der gerade zuoberst auf dem Stack liegt.

[26] Prinzipiell ist es möglich, vordefinierte Funktionen durch benutzerdefinierte zu ersetzen und unter Nutzung von Funktionszeigern (siehe Abschnitt 9.4) sogar zu erweitern.

Beispiele für Funktionsdeklarationen:

```
/* parameterless function without defined return value */
sayHello()
{
   print(“Text from function body\n“);
}

/* named parameter, local variables and return value */
factorial(n; i, result)
{
   for(result=1, i=n; i>1; i++)
     result *= i;
   return result;
}

/* no named parameters but local variable and return value */
queryName(;name)
{
   name =““;
   while (strlen(name) == 0)
   {
      print(“Tell me your name: “);
      name = fgets(stdin);
   }
   return name;
}
```

9.2 Aufruf von Funktionen

Der Aufruf einer Funktion erfolgt durch Nennung des Funktionsnamens, gefolgt von einer geklammerten kommaseparierten Liste der Parameter. Syntaktisch kann ein Funktionsaufruf als einzelne Anweisung oder an Stelle eines Wertes stehen.

Beispiel:

```
main()
{
   sayHello();          // simple statement – return value is ignored
   f = factorial(17);   // assign return value to variable
   g = sqrt(2.0) * 0.5; // use function in expression
   print(queryName());  // use function as parameter of another one
}
```

Hinweis:
Beim Aufruf einer Funktion müssen *mindestens* so viele Parameter angegeben werden, wie in der Funktionsdeklaration angegeben. Werden mehr Parameter angegeben, so werden den benannten Parameternamen die *letzten* (d.h. am weitesten rechts stehenden) Parameterwerte zugewiesen.

Wie C/C++ unterstützt BOB/BOB++ den rekursiven Aufruf von Funktionen.

Beispiel:

```
factorial(n)          // recursive version of factorial function
{
   if (n > 1)
     return n * factorial(n-1);
   return 1;
}
```

Die mögliche Rekursionstiefe wird dabei durch die Größe des Stacks begrenzt. Für einen Funktionsaufruf werden mindestens vier Stack-Einträge benötigt. Hinzu kommt jeweils ein Eintrag für jeden übergebenen Parameter und jede lokale Variable. Demnach berechnet sich die maximale Rekursionstiefe (Rmax) wie folgt:

```
Rmax = Stackgröße / (Parameterzahl + Anzahl lokaler Variablen + 4)
```

In BOB+ (nicht in BOB) besteht die Möglichkeit, die Größe des Stacks (und damit die mögliche Rekursionstiefe) mit Hilfe der Umgebungsvariablen BPSTACK zu beeinflussen (vgl. 2.2).

9.3 Variable Parameterlisten

Wie im vorigen Abschnitt erwähnt, ist es möglich, einer Funktion beim Aufruf mehr Parameter zu übergeben als in der Deklaration angegeben. BOB+ erlaubt unter Ausnutzung dieser Eigenschaft die Konstruktion von Funktionen mit einer beliebigen Anzahl von Parametern. Für den Zugriff auf die übergebenen Parameter wird dann an Stelle von Parameternamen die vordefinierte Funktion *arg* verwendet. Die Bestimmung der Parameteranzahl erfolgt mit *argcnt* (siehe 11.9).

Beispiel:

```
sum(;i,result)
{
    result = 0;
    for (i=0; i<argcnt(); i++)
        result += arg(i);
    return result;
}

main()
{
   print(sum(0,1,2,3,4,5,6,7,8,9),“\n“);
}
```

9.4 Funktionszeiger

In Abschnitt 5.2.2 wurde der Datentyp *function* erklärt, der einem Wert vom Funktionstyp zugeordnet ist. Tatsächlich lässt sich eine Funktion in gewissen Grenzen wie ein „normaler“ Wert behandeln, d.h. insbesondere an eine Variable zuweisen oder anderen Funktionen als Parameter übergeben. Da Werte vom Funktionstyp nichts anderes als einen Zeiger auf den ausführbaren Code der jeweiligen Funktion speichern, sind sie mit den Funktionszeigern in C/C++ vergleichbar und können in analoger Weise verwendet werden.

Beispiel:

```
myFunc()
{
   print(“myFunc called\n“);
}

main(;f)
{
   f = myFunc;          // assign myFunc to variable f
   f();                 // call function
}
```

9.5 Überladen von Funktionen

BOB/BOB+ identifiziert eine Funktion ausschließlich anhand ihres Namens und nicht (wie z.B. C++) anhand der vollständigen Signatur. Damit ist ein Überladen (d.h. die Definition mehrerer Funktionen gleichen Namens) im eigentlichen Sinne nicht möglich. Die Neudefinition einer bereits vorhandenen Funktion führt zwar nicht zu einem Fehler, ersetzt die vorher definierte gleichnamige Funktion aber vollständig.

Unter Ausnutzung der in den vorangegangenen Abschnitten beschriebenen variablen Parameterlisten und Funktionszeigern ist in BOB+ jedoch ein Pseudo-Überladen möglich. Betrachten wir dazu nachfolgendes Beispiel:

```
myFunc(i)
{
   print("myFunc(i) called\n");
}

myFunc2()
{
   if (argcnt() != 2)
      myFunc1(arg(0));
   else
      print("myFunc(i,j) called\n");
}


main()
{
   myFunc1 = myFunc;
   myFunc = myFunc2;
   myFunc(1,2);
   myFunc(1);
}
```

Ziel ist es hier, zwei Funktionen zu definieren, die beide über den Namen myFunc aufrufbar sind, wobei die eine einen Parameter und die andere zwei Parameter verarbeitet.

Zuerst wird unter dem Namen myFunc die Variante mit einem Parameter definiert. Die Variante für zwei Parameter erhält zunächst einen anderen Namen (myFunc2) und wird so implementiert, dass sie eine variable Anzahl von Parametern verarbeiten kann. Ist die Anzahl der übergebenen Parameter nicht zwei, so ruft sie eine Funktion myFunc1 auf, andernfalls wird der eigene Code ausgeführt.
Die Funktion main weist nun der (globalen) Variablen myFunc1 den Code von myFunc und myFunc den Code von myFunc2 zu.

10 Objektorientierte Programmierung

Das Konzept der Objektorientierung in BOB/BOB++ ist trotz syntaktischer Ähnlichkeit wesentlich einfacher als das von C++. Dennoch werden die grundlegenden Paradigmen objektorientierter Programmierung – Datenkapselung (zumindest ansatzweise), Vererbung und Polymorphie – berücksichtigt. Darüber hinaus bestehen zusätzliche Möglichkeiten, wie die Konstruktion partieller Klassen, nachträgliche Redefinition von Member-Funktionen sowie (in gewissen Grenzen und nur bei BOB+) die Definition von Operatoren für Objekte. In den nachfolgenden Abschnitten werden die Prinzipien und Möglichkeiten beschrieben.

10.1 Aufbau einer Klasse

Zur Erläuterung des Aufbaus einer Klasse in BOB/BOB+ soll nachfolgendes Beispiel dienen.
Es stellt den Anfang einer fiktiven Klassenbibliothek mit der Wurzel MyBaseClass und einer davon abgeleiteten Klasse MyDerivedClass dar. Alle Objektinstanzen dieser Bibliothek sollen einen eindeutigen Identifikator (_ID) haben, dessen Vergabe die einzige Aufgabe der Basisklasse ist.

```
// declaration of MyBaseClass
class MyBaseClass
{
   MyBaseClass();
   getID();
   _ID;
   static createID();
   static _maxID = 0;   // initialization works only for BOB+
}
```

```
// implementation of MyBaseClass
MyBaseClass::MyBaseClass() { _ID = createID; }
MyBaseClass::getID() { return _ID; }
MyBaseClass::createID() { return ++_maxID; }

// declaration of MyDerivedClass
class MyDerivedClass : MyBaseClass
{
   MyDerivedClass(name, value);
   getName();
   setName(name);
   getValue();
   setValue(value);
   _name, _value;
}

// implementation of MyDerivedClass
MyDerivedClass::MyDerivedClass(name, value)
{
   MyBaseClass();
   _name = name;
   _value = value;
   // following line is needed for BOB (but not BOB+)
   // return this;
}
MyDerivedClass::getName() { return _name; }
MyDerivedClass::setName(name) { _name = name; }
MyDerivedClass::getValue() { return _value; }
MyDerivedClass::setValue() { _value = value; }
```

Die Klassendeklarationen wie auch die Implementierungen ähneln auf den ersten Blick dem von C++ gewohnten. Allerdings gibt es eine Reihe von wichtigen Unterschieden:

Keine Zugriffsspezifizierer

In BOB/BOB+ sind Member- und Klassenfunktionen sowie Klassenvariablen (static) implizit öffentlich (public im Sinne von C++).

Member-Variablen sind implizit geschützt (protected im Sinne von C++), also nur in der deklarierenden Klasse und ihren Ableitungen sichtbar. Für den Zugriff von außen müssen entsprechende Zugriffsmethoden implementiert werden.

Keine Mehrfachvererbung

Anders als in C++ hat eine BOB/BOB+-Klasse höchstens eine Basisklasse. Auch ein Interface-Konzept (wie etwa von Java oder C# bekannt) gibt es nicht. Die Vererbung selbst ist immer öffentlich, d.h. alle öffentlichen Elemente der Basisklasse werden auch öffentliche Elemente der abgeleiteten Klassen.

Deklaration von Variablen und Methoden

In BOB/BOB+-Klassen ist die Nennung von Member-Funktionen in der Klassendeklaration optional. Etwa hätte man MyDerivedClass in obigem Beispiel auch so deklarieren können[27]:

```
MyDerivedClass : MyBaseClass
{
   _name, _value;
}
```

Diese Eigenschaft erlaubt es, Klassen bei Bedarf nachträglich um Methoden zu ergänzen (siehe hierzu auch 10.6).

Member-Variablen und alle statischen Elemente *müssen* deklariert werden.

[27] Normalerweise ist von dieser Kurzform eher abzuraten, da sie die Lesbarkeit des Quelltextes sicher nicht verbessert.

Steht der static-Modifizierer vor einer kommaseparierten Variablenliste, so gilt er für alle ihre Elemente.

Eingeschränkte Initialisierung von Klassenvariablen

In BOB können Klassenvariablen nicht explizit initialisiert werden[28]. In BOB+ ist dies möglich, jedoch erfolgt die Initialisierung bereits zur Kompilier- und nicht erst zur Laufzeit. Deshalb sind zur Initialisierung nur Literale (und keine Funktionsaufrufe) verwendbar. Die Initialisierung muss stets innerhalb der Klassendeklaration notiert werden (siehe _maxID in MyBaseClass).

Objekte werden immer dynamisch erzeugt

Alle Instanzen von Klassen (Objekte) haben den internen Typ *object* und sind damit Referenztypen (siehe 5.2.2). Der einzige Weg, sie zu erzeugen führt über den Operator *new*:

```
myObj = new MyDerivedClass(“anumber“,1000);
```

Deshalb ist es nicht möglich, wie in C++ direkt im Anschluss an eine Klassendeklaration Variablen dieses Typs anzulegen, was auch erklärt, weshalb hinter der schließenden Klammer der Klassendeklaration kein Semikolon steht.

10.2 Konstruktoren und Destruktoren

Ein *Konstruktor* ist eine spezielle Member-Funktion eines Objekts, deren Aufgabe hauptsächlich darin besteht, bei der Objekterzeugung die Member-Variablen mit geeigneten Anfangswerten zu initialisieren.

In BOB/BOB+ hat ein Konstruktor – wie in C++ – den selben Namen wie die Klasse selbst. Da ein Überladen von Funktionen im eigentlichen Sinne nicht möglich ist (siehe 9.5), kann eine Klasse nur einen Konstruktor besitzen. Dieser Konstruktor muss einen Verweis auf das erzeugte Objekt als Funktionswert zurückgeben. In BOB bedeutet dies, dass ein Konstruktor stets mit der Anweisung

```
return this;
```

verlassen werden muss. Der Compiler von BOB+ fügt den entsprechenden Bytecode automatisch an das Ende jeder Konstruktorfunktion an, so dass die Anweisung hier nur dann notiert werden muss, wenn die Funktion vorzeitig (auf Grund einer Abruchbedingung) verlassen wird. Anders als z.B. in C++ ruft ein Konstruktor in BOB/BOB++ nicht automatisch auch den Konstruktor einer ggf. vorhandenen Basisklasse auf. Ein solcher Aufruf muss explizit kodiert werden. Ein Beispiel zur Verdeutlichung:

[28] Sie erhalten hier implizit den Wert **nil**.

```
class Base
{
   Base();
}

Base::Base()
{
   print(“ctor of Base called\n“);
   // return this;    // uncomment when using BOB
}

class Derived : Base
{
   Derived();
}

Derived::Derived
{
   Base();
   print(“ctor of Derived called\n“);
   // return this;    // uncomment when using BOB
}

main()
{
   obj = new Derived();
   // do anything
   obj = delete obj;  // comment out when using BOB
}
```

Anmerkungen:

- Ein Konstruktor lässt sich in BOB/BOB+ bei Bedarf auch wie eine „normale“ Member-Funktion aufrufen.
- Von einer Klasse, die keinen eigenen Konstruktor besitzt, können keine Instanzen erzeugt werden. Diese Eigenschaft lässt sich zur Definition *abstrakter Klassen* nutzen.

Ein *Destruktor* ist gewissermaßen das Gegenstück zum Konstruktor. Er dient dazu, beim Zerstören eines Objekts ggf. von ihm belegte externe Ressourcen (zusätzlicher Speicher, offene Dateien etc.) wieder freizugeben.

BOB hat keine Möglichkeit, ein erzeugtes Objekt vor dem Programende explizit freizugeben, womit sich hier die Frage nach einem Destruktor erübrigt. BOB+ implementiert eine Objektfreigabe mit Hilfe des Operators *delete*[29] und übernimmt im wesentlichen das Konzept der Destruktoren aus C++. Konkret heißt dies:

- Ein Destruktor wird als parameterlose Member-Funktion deklariert, die den Klassennamen mit vorangestellter Tilde (~) trägt.
- Der Destruktor wird unmittelbar vor der Zerstörung des Objekts automatisch aufgerufen, d.h. innerhalb des Destruktors sind alle Elemente (einschließlich des *this*-Verweises) noch gültig.
- Am Ende der Destruktorfunktion werden rekursiv auch die Destruktoren aller Basisklassen aufgerufen, d.h. der Destruktorcode der abgeleiteten Klasse wird immer vor dem der Basisklasse ausgeführt. Klassen ohne Destruktor werden dabei innerhalb der Klassenhierarchie übersprungen.

[29] Die Objektfreigabe ist prinzipiell auch mit der vordefinierten Funktion *free* möglich, dann erfolgt aber kein Destruktoraufruf.

Zur Verdeutlichung erweitern wir obiges Beispiel:

```
class Base
{
   Base();
   ~Base();
   getClassName();
}

Base::Base()
{
   print(“ctor of Base called\n“);
}

Base::~Base()
{
   print(“dtor of Base called\n“);
}

Base::getClassName() { return “Base“; }

class Derived : Base
{
   Derived();
   ~Derived();
   getClassName();
}

Derived::Derived
{
   Base();
   print(“ctor of Derived called\n“);
}

Derived::~Derived
{
   print(“dtor of Derived called\n“);
}

Derived::getClassName() { return “Derived“; }

main()
{
   obj = new Derived();
   // do anything
   obj = delete obj;
}
```

Anmerkungen:

- Der Operator *delete* gibt in BOB+ immer **null** zurück. Er kann deshalb – wie hier in der Funktion *main* – als rechter Ausdruck einer Zuweisung verwendet werden, um die freigegebene variable als ungültig zu markieren.
- Implementierungsbedingt bildet BOB+ den Destruktor auf eine Member-Funktion mit dem reservierten Bezeichner ***dtor*** ab. Dieser Bezeichner kann alternativ zur C++-konformen Schreibweise auch für die Deklaration/Definition des Destruktors verwendet werden. Außerdem lässt sich der Destruktor über diesen Namen wie eine normale Member-Funktion aufrufen[30].

[30] Vorsicht: Auch hier werden die Destruktoren der Basisklassen mit aufgerufen!

10.3 Zugriff auf Member-Funktionen und -Variablen innerhalb einer Klasse

Enthält die Implementierung einer Memberfunktion Funktionsaufrufe oder Variablenzugriffe, so wird zunächst in der aktuellen Klasse und ggf. deren Basisklassen nach einem passenden Member gesucht. Verläuft die Suche ergebnislos, so wird der jeweilige Funktions- oder Variablenbezeichner als globaler Bezeichner angenommen. Wird vor dem Bezeichner das Schlüsselwort ***this*** mit nachfolgendem Dereferenzierungsoperator (`->`) angegeben, so wird der Bezeichner nur im jeweiligen Objekt gesucht.

Für statische Member ist das Verhalten ähnlich, jedoch wird hier zur Spezifikation einer konkreten Klasse vor dem eigentlichen Bezeichner der Klassenname und der Bereichsauflösungsoperator (::) angegeben.

In diesem Zusammenhang entsteht das Problem, dass unter Umständen in der Implementierung einer Member-Funktion auf ein globales Symbol (Variable oder Funktion) zugegriffen werden muss, dass namensgleich mit einem Element der aktuellen Klasse ist. BOB bietet dafür keine Lösung, d.h., der Pprogrammierer muss dafür sorgen, dass solche Situationen nicht entstehen.
In BOB+ kann – wie in C++ – der Zugriff auf die globalen Symbole mit Hilfe des Bbereichsauflösungsoperators erzwungen werden.

Beispiel:

```
// globally defined function
message() { print(“global function message called\n“) };

class MyClass
{
   MyClass();
   message();    // member-function message
   doAction();
}

MyClass() {}
MyClass::message() { print(“function MyClass::message\n“);}
MyClass::doAction(;obj)
{
   message();          // callinternal message method
   this->message();    // callinternal message method,
                       // search only in current class
   ::message()         // call global function message
}
```

10.4 Virtuelle Methoden

Da in BOB/BOB+ nicht die Variablen sondern lediglich die Werte einen Typ besitzen, gibt es keine „frühe Bindung“. Das bedeutet, dass alle Methoden einer Klasse – auch die statischen – *virtuell* sind. Betrachten wir dazu nochmals das Destruktor-Beispiel aus Abschnitt 10.2 und ändern dessen *main*-Funktion wie folgt:

```
main()
{
   obj = new Base();
   print(obj->getClassName(),“\n“);
   obj = delete obj;
   obj = new Derived();
   print(obj->getClassName(),“\n“);
   obj = delete obj;
}
```

Hier werden ein- und derselben Variablen (obj) nacheinander Instanzen der Klassen Base bzw. Derived zugewiesen und jeweils deren getClassName()-Methoden aufgerufen. Dabei wird beim

ersten Aufruf die ursprüngliche Version aus Base und beim zweiten die *überschriebene* aus Derived aktiviert.

Häufig kommt es vor, dass eine abgeleitete Klasse eine Methode ihrer Basisklasse mit dem Ziel überschreibt, deren Funktionalität zu erweitern aber nicht vollständig zu ersetzen. Dann ist es wünschenswert, den Inhalt der geerbten Methode einfach aufzurufen, anstatt ihn vollständig neu zu implementieren. In BOB ist eine solche Möglichkeit nicht vorgesehen. BOB+ erlaubt dagegen den Aufruf der überschriebenen Methode, benutzt hierfür aber eine besondere syntaktische Konstruktion:

Um innerhalb der Implementierung einer überschreibenden Methode die geerbte Vorfahrmethode aufzurufen, wird dem Funktionsnamen das Präfix ***BC_*** *(für BaseClass) vorangestellt.*

Beispiel:

```
class MyBaseClass
{
   MyBaseClass();
   writeInfo();
}

MyBaseClass::MyBaseClass() {}    // ctor does nothing
MyBaseClass::writeInfo() { print("MyBaseClass::writeInfo() called\n"); }

MyDerivedClass : MyBaseClass
{
    MyDerivedClass();
    writeInfo();
}

MyDerivedClass::MyDerivedClass() {}    // ctor does nothing
MyDerivedClass::writeInfo()
{
   print("MyDerivedClass::writeInfo() called\n");
   BC_writeInfo();    // call inherited writeInfo method
}

main(;obj)
{
   obj = new MyDerivedClass();
   obj->writeInfo();
   delete obj;
   return 0;
}
```

10.5 Partielle Klassen

BOB/BOB+ unterstützt partielle Klassen in ähnlicher Form wie C# 2.0 oder Smalltalk. Dabei meint ***Partielle Klasse*** eine Klasse, deren Definition auf mehrere Blöcke (die ggf. auch in unterschiedlichen Quelldateien stehen können) verteilt ist. Dies eröffnet mehrere Möglichkeiten:

- Verteilen umfangreicher Klassen auf mehrere Dateien
- Behandlung inhaltlich verschiedener Aspekte einer Klasse in getrennten Abschnitten
- Klasseninhalte lassen sich variabel gestalten, d.h. in Abhängigkeit vom jeweiligen Kontext kann eine konkrete Klasse aus unterschiedlichen Teilen zusammengesetzt werden.
- Nutzung zur Vorwärtsdeklaration von zur Kompilierzeit eines Moduls unbekannten Klassen
- Bestehende Klassen können „nachträglich" erweitert werden.

Für das folgende Beispiel nehmen wir an, dass ein Programm eine Klasse `Class1` verwendet, von der es eine Instanz bildet und darauf eine Methode `doAction` aktiviert. Die Klasse Class1 selbst wird in einem anderen Modul abschließend definiert und implementiert, welches beim Aufruf mit dem Hauptprogramm kombiniert wird. Zunächst das Hauptprogramm:

```
/* doAction example – main module action.bp */

class Class1      // class prototype
{
   doAction();
}

main(;obj)
{
   obj = new Class1();
   obj->doAction();
   delete obj;
}
```

Class1 wird hier nur als leerer Prototyp definiert[31] – den fordert der Compiler. Eine erste Version der vollständigen Definition und Implementierung erfolgt in einem Modul actcl1_1:

```
/* first version of Class1 module – actcl1_1.bp */

class Class1
{
    Class1();                       // ctor
    ~Class1();                      // dtor
    doAction();
    _callCnt;                       // counter for calls of doAction
}
// implementation

Class1::Class1() { _callCnt = 0; }
Class1::~Class1() {}                // dtor does nothing
Class1::doAction()
{
   print(++_callCnt,“. call of doAction in module actcl1_1\n“);
}
```

Nun können Hauptprogramm und Modul jeweils in Bytecode übersetzt und anschließend das Hauptprogramm unter Nutzung des Bibliotheksmoduls Bibliothek gestartet werden:

```
bp –c action –o action.bpm
bp –c actcl1_1 –o actcl1_1.bpm

bp –r action actcl1_1
```

[31] Die Deklaration von doAction hätte man auch weglassen können.

Ohne Änderung am Hauptprogramm können wir jetzt die Klasse Class1 in einem anderen Modul erneut implementieren und alternativ zur obigen Variante verwenden:

```
/* second version of Class1 module – actcl1_2.bp */

class Class1
{
    Class1();                       // ctor
    ~Class1();                      // dtor
    doAction();
}
// implementation

Class1::Class1() {}                 // ctor does nothing
Class1::~Class1() {}                // dtor does nothing
Class1::doAction()
{
   print("doAction in module actcl1_2 activated\n");
}
```

Übersetzung und Aufruf erfolgen analog:

```
bp –c actcl1_2 –o actcl1_2.bpm
bp –r action actcl1_2
```

Hinweis:
Um das Beipiel mit BOB benutzen zu können, müssen die Konstruktor-Implementierungen um die Anweisung `return this;` ergänzt und die `delete`-Anweisung aus dem Hauptprogramm entfernt werden. Ein Vorkompilieren ist hier nicht möglich.

10.6 Ersetzen von Member-Funktionen (Redefinition)

So, wie man bestehende „normale" Funktionen durch einfache Neudefinition ersetzen kann (siehe 9.5), ist dies auch für Funktionen möglich, die Bestandteil einer Klasse sind. Die neu definierte Variante ersetzt die alte vollständig. Diese Eigenschaft lässt sich zur nachträglichen Änderung des Verhaltens einer Klasse nutzen. Betrachten wir dazu wieder das Beispiel aus dem vorangegangenen Abschnitt. Jetzt wird jedoch die Klasse Class1 sofort implementiert.

```
/* doAction example – main module action.bp */

class Class1
{
   Class1();
   doAction();
}

Class1::Class1() {}
Class1::doAction()
{
   print("Class1::doAction called\n");
}

main(;obj)
{
   obj = new Class1();
   obj->doAction();
   delete obj;
}
```

Das Beispiel sollte in BOB+ direkt lauffähig sein.

Nun wollen wir ein Patch-Modul schreiben, das das Verhalten der doAction-Methode aus Class1 ändert:

```
/* module patch.bp replaces doAction method of Class1 */
Class1::doAction()
{
   print(“patched version of CtClass::doAction called\n“);
}
```

Bitte beachten Sie, dass unser Patch-Modul nicht separat übersetzbar ist, da es keine Deklaration von Class1 enthält, folgender Aufruf ist aber möglich:

```
bp action patch
```

Hier wird der Patch nach dem ursprünglichen Programm übersetzt, so dass Class1 bereits vorhanden und der Compiler zufrieden ist.

10.7 Defineren von Operatoren

In BOB+ (nicht in BOB) ist es möglich, einige Operatoren für Objekte (oder genauer: deren Klassen) umzudefinieren bzw. überhaupt zu definieren. Die Möglichkeiten sind in dieser Hinsicht nicht so umfangreich wie bei C++, sollten in vielen Fällen aber dennoch ausreichend sein.

Generell wird ein Operator indirekt definiert, indem eine Klasse eine Memberfunktion definiert, deren Namen einem der reservierten Bezeichner der Form OP_XXX (siehe Abschnitt 7) entspricht.

10.7.1 Definieren des Funktionsoperators

Indem eine Klasse den Funktionsoperator () umdefiniert, werden ihre Instanzen zu Funktionsobjekten (mitunter auch als *Funktoren* bezeichnet). Solche Objekte können innerhalb des Programms wie Funktionen gebraucht werden.

Der Operator wird mit Hilfe einer Member-Funktion OP_CALL definiert.

Beispiel:

```
/* operator () example */
class StringWriter
{
   StringWriter(file);
   ~StringWriter();
   OP_CALL();
   _fp;
}

StringWriter::StringWriter(file)
{
   _fp = fopen(file,“wt“);  // open textfile for write
}

StringWriter::~StringWriter()
{
   fclose(_fp);
}

StringWriter::OP_CALL(;n,i)
{
   n = argcnt();
   for (i=0;i<n; i++)
     fputs(arg(i),_fp);
   return n;
}
```

```
main()
{
   writer = new StringWriter(“txtfile.txt“);
   writer(“Hello“,“World“);
   delete writer;
}
```

10.7.2 Definieren des Zugriffsoperators

Der Zugriffsoperator [] dient normalerweise dem indizierten Zugriff auf einzelne Elemente einer Zeichenkette oder eines Vektors. Wird der Operator für Objekte definiert, so sind hierfür zwei Member-Funktionen – eine für den Lesezugriff (OP_VREF) und eine für den Schreibzugriff (OP_VSET) – zu definieren[32]. Im Gegensatz zur Standardvariante für Strings und Vektoren muss das Index-Argument nicht zwingend eine Ganzzahl sein.

Als Beispiel soll ein Fragment einer Klasse Point dienen, die einen n-dimensionalen Punkt repräsentiert. Die Koordinaten sollen mit Hilfe des Zugriffsoperators gelesen und geschrieben werden können.

```
class Point()
{
   Point(dim);
   ~Point();
   OP_VSET(index,value);
   OP_VREF(index);
   _coords;                // vector of coordinates
}

Point::Point(dim;i)
{
   _coords = newvector(dim);
   for (i=0;i<dim;i++) _coords[i] = 0;
}
Point::~Point() { free(_coords); }

Point::OP_VSET(index value) { _coords[index] = value; }

Point::OP_VREF(index) { return _coords[index]; }

main(;point)
{
   point = new Point(2);
   point[0] = 12;         // set x-coordinate
   point[1] = 4.7;        // set y-coordinate;
   print(“x=“,point[0],“ y=“,point[1],“\n“);
   delete point;
}
```

[32] Wird nur eine der Funktionen implementiert, funktioniert der Operator nur in einer Richtung.

10.7.3 Weitere Operatoren

Neben dem Funktions- und dem Zugriffsoperator können folgende weitere Operatoren umdefiniert werden:

Operator	Funktionsname
Arithmetische Operatoren	
+	OP_ADD
-	OP_SUB
*	OP_MUL
/	OP_DIV
%	OP_REM
Bitweise Operatoren	
|	OP_BOR
&	OP_BAND
^	OP_XOR
Verschiebeoperatoren	
<<	OP_SHL
>>	OP_SHR

Tabelle 11 Weitere umdefinierbare Operatoren in BOB+

Das Prinzip ist für alle diese Operatoren gleich und soll hier am Beispiel des Additionsoperators gezeigt werden. Dazu erweitern wir die Klasse Point aus 10.7.2 (siehe auch 10.5) und definieren den Operator + so, dass mit seiner Hilfe ein neuer Punkt erzeugt wird, dessen Koordinaten als Summe der Koordinaten der Operanden gebildet werden.

```
class Point
{
   Point(dim);           // replaced ctor
   OP_ADD(otherPoint);   // operator function
   getDim();             // get dimension
   _dim;                 // saved dim value
}

Point::Point()
{
   _dim = dim;
   _coords = newvector(dim);
   for (i=0;i<dim;i++) _coords[i] = 0;
}

Point::getDim() { return _dim; }

Point::OP_ADD(otherPoint;n,i)
{
   n = otherPoint->getDim();
   if (n > _dim) n = _dim;
   result = new Point(n);
   for (i=0;i<n;i++)
     result[i] = _coords[i] + otherPoint[i];
   return result;
}
```

```
main(;p1,p2,pt)
{
   p1 = new Point(2);
   p1[0] = 12;         // set x-coordinate
   p1[1] = 4.7;        // set y-coordinate;
   p2 = new Point(2);
   p2[0] = 24;         // set x-coordinate
   p2[1] = 14.7;       // set y-coordinate;
   pt = p1 + p2;
   print("x=",pt[0]," y=",pt[1],"\n");
   delete p1;
   delete p2;
   delete pt;
}
```

Hinweis:
Bitte beachten Sie, dass der Additionsoperator in diesem Beispiel ein neues Objekt erzeugt, das explizit wieder freigegeben werden sollte.

11 Vordefinierte Funktionen

Nachfolgend werden die verfügbaren vordefinierten Funktionen im Sinne einer Referenz beschrieben. Die Typen der Parameter und Rückgabewerte werden in den jeweiligen Signaturen mit angegeben.

11.1 Speicherverwaltungs- und Testfunktionen

Die mit (*) gekennzeichneten Funktionen dieses Abschnitts stehen nur in BOB+ zur Verfügung.

Funktion free (*):

`null free(variable)`

Die Funktion gibt den Speicherplatz einer Variablen des Typs String, Vektor oder Objekt frei.

Parameter:

- variable: freizugebende Variable

Rückgabewert:

Die Funktion gibt stets den Wert **null** zurück.

Hinweise:

Die Funktion dient der Freigabe des Speicherplatzes von nicht mehr benötigten Variablen der Typen String, Vektor und Objekt. Wird sie auf andere („einfache“) Datentypen angewendet, bleibt sie wirkungslos.

Achtung: Die versehentliche mehrmalige Anwendung auf ein- und dieselbe Variable kann zum Programmabsturz führen. Deshalb sollte einer freigegebenen Variablen stets explizit der Wert **null** zugewiesen werden (als Funktionsergebnis von ***free*** oder ***null***).

Beispiel:

```
/* example for using free-function */
main(;text)
{
   text = “This Text “;
   text += “should be disposed after using.\n“;
   print(text);
   /* free memory and set variable to null */
   text = free(text);
   print(text); /* writes “null“ to stdout */
}
```

Funktion memsize (*):

`int memsize()`

Die Funktion prüft dir Größe des noch verfügbaren freien Speichers.

Rückgabewert:

Größe des verfügbaren freien Speichers in Bytes

Funktion newstring:

`string newstring(int size)`

Die Funktion erzeugt einen neuen String aus *size* Zeichen. Alle Elemente werden mit 0x00 initialisiert

Parameter:

- size: Anzahl der Elemente

Rückgabewert:

neu erzeugter String

Hinweise:

Genau genommen erzeigt die Funktion einen Zeichenpuffer (Zeichenarray) der angegebenen Größe. Die Größe des Puffers ist fest, d.h. es gibt keine Möglichkeit einer dynamischen Größenänderung. Der Zugriff auf die einzelnen Elemente erfolgt über den Zugriffsoperator [].

Ein mit dieser Funktion angelegter Puffer sollte, sobald er nicht mehr benötigt wird, mit ***free*** wieder freigegeben werden.

Funktion newvector:

`vector newvector(int size)`

Die Funktion erzeugt einen neuen Vektor aus *size* Elementen. Alle Elemente werden mit **null** initialisiert

Parameter:

- size: Anzahl der Elemente

Rückgabewert:

neu erzeugter Vektor

Hinweise:

Ein mit dieser Funktion angelegter Vektor sollte, sobald er nicht mehr benötigt wird, mit ***free*** wieder freigegeben werden.

Die Größe des so angelegten Vektors ist fest, d.h. es gibt keine Möglichkeit einer dynamischen Größenänderung.

Die Elemente eines Vektors können beliebigen Typs sein. Insbesondere können sie auch selbst Vektoren sein, was den Aufbau mehrdimensionaler Strukturen erlaubt. Der Zugriff auf die einzelnen Elemente erfolgt über den Zugriffsoperator [].

Funktion T oder Vec (*):

`vector T(...)` oder

`vector Vec(...)`

Beide Funktionen sind synonym verwendbar. Sie konstruieren einen neuen Vektor, dessen Elemente die übergebenen Argumente in der angegebenen Reihenfolge sind.

Parameter:

- kommaseparierte Liste beliebiger Werte

Rückgabewert:

neu erzeugter Vektor

Hinweis:

Ein mit dieser Funktion angelegter Vektor sollte, sobald er nicht mehr benötigt wird, mit ***free*** wieder freigegeben werden.
Achtung: Argumente, die als Literale oder Variablen übergeben werden, werden von der Funktion nicht kopiert. Deshalb ist darauf zu achten, dass sie nicht mehrfach freigegeben werden dürfen.

Beispiel:

```
/* T() example */
main(;vec,i)
{
   /* make a vector using strsplit-function */
   vec = strsplit("This is a text."," ");
   for(i=0;i<vecsize(vec);i++)
      print(vec[i],"\n");
   /* make new vector using T-Function
      Note that element at index 3 is the vector constructed above. */
   vec = T(1,2,vecsize(vec),vec,"TEST");
   for(i=0;i<vecsize(vec);i++)
      print(vec[i],"\n");
   /* free inner vector */
   free(vec[3]);
   /* free outer vector */
   free(vec);
}
```

Funktion vecsize (*):

```
int vecsize(vector v)
```

Die Funktion liefert die Anzahl der Elemente von *v*

Parameter:

- v: Vektor

Rückgabewert:

Anzahl der Elemente von *v*

11.2 Typkonvertierungen und RTTI

BOB/BOB+ kennt keine impliziten Typumwandlungen und keinen cast-Operator. Deshalb sind Werte verschiedener Typen innerhalb eines Ausdrucks im allgemeinen inkompatibel[33]. Zur Lösung dieses Problems wurden Funktionen für die explizite Umwandlung einfacher Datentypen (int, float, string) eingeführt.

Alle Funktionen dieses Abschnitts stehen nur in BOB+ zur Verfügung.

Funktion dynamic_cast:

```
object dynamic_cast(class dest, expr)
```

Die Funktion verhält sich ähnlich dem dynamic_cast-Operator in C++. Zunächst wird der Ausdruck *expr* ausgewertet. Ist das Ergebnis ein Objekt der Klasse *dest* oder einer ihrer Ableitungen, so wird dieses Ergebnis, in allen anderen Fällen **null** zurückgegeben.

Parameter:

- dest: Klasse, zu der die Zugehörigkeit von *expr* geprüft werden soll
- expr: Ausdruck beliebigen Typs

Rückgabewert:

Ergebnis von *expr* bei erfolgreichem Test, sonst **null**

Hinweis:

Im Gegensatz zu C++ wird kein Laufzeitfehler erzeugt, wenn *expr* kein Objekt ist.

Beispiel:

```
/* dynamic_cast example */
class Base
{
   Base();
}
Base::Base() { }

class Derived : Base
{
   Derived();
}
Derived::Derived() { }
```

[33] BOB+ erlaubt das Mischen von int und float bei Operanden von arithmetischen und Vergleichsoperatoren. Dabei wird implizit eine Konvertierung nach float durchgeführt.

```
main()
{
   obj1 = new Base();
   obj2 = new Derived();
   o = dynamic_cast(Base, obj1);    // returns obj1
   o = dynamic_cast(Base, obj2);    // returns obj2
   o = dynamic_cast(Derived, obj1); // returns null
   o = dynamic_cast(Derived, obj2); // returns obj2
   i = dynamic_cast(Base, 17);      // returns null
}
```

Funktion float:

`float float(`*`expr`*`)`

Die Funktion konvertiert einen Ausdruck der Typen int, float oder string in einen float-Wert.

Parameter:

- expr: Ausdruck des Typs int, float oder string

Rückgabewert:

float-Wert von *expr*

Hinweis:

Bei der Umwandlung aus einer Ganzzahl können bis zu vier Stellen Genauigkeit verloren gehen. Bei der Umwandlung aus einem String wird das Ergebnis des entsprechenden Aufrufs der C-Funktion sscanf zurückgegeben.

Funktion getclassname:

`string getclassname(`*`expr`*`)`

Die Funktion wertet den Ausdruck expr aus und gibt den Klassennamen des Resultats zurück, falls *expr* ein Objekt oder eine Klasse ergibt.

Parameter:

- expr: Ausdruck des Typs class oder object

Rückgabewert:

Klassenname von *expr* bei Erfolg, sonst **null**

Hinweis:

Das Ergebnis wird dynamisch erzeugt und sollte mittels ***free*** explizit wieder freigegeben werden.

Funktion gettype:

`int gettype (`*`expr`*`)`

Die Funktion wertet den Ausdruck expr aus und gibt dessen Typ zurück.

Parameter:

- expr: beliebiger Ausdruck

Rückgabewert:

Datentyp-ID von *expr*

Hinweise:

Das Ergebnis kann mit *gettypename* in eine lesbare Darstellung umgewandelt werden.

Die ermittelte ID ist zum Typvergleich mit anderen Ausdrücken verwendbar.

Funktion gettypename:

`string gettypename(int typeid)`

Die Funktion ermittelt den Namen des durch *typeid* spezifizierten Typs.

Parameter:

- typeid: ID des Datentyps, dessen Name ermittelt werden soll.

Rückgabewert:

Typname von *typeid*

Hinweise:

Das Ergebnis wird dynamisch erzeugt und sollte mittels ***free*** explizit wieder freigegeben werden.

Die ID eines Datentyps kann mit *gettype* ermittelt werden.

Funktion int:

`int int(`*`expr`*`)`

Die Funktion konvertiert einen Ausdruck der Typen int, float oder string in einen int-Wert.

Parameter:

- expr: Ausdruck des Typs int, float oder string

Rückgabewert:

int-Wert von *expr*

Hinweis:

Bei der Umwandlung aus float kann ein Überleuf des Zahlbereichs eintreten. Bei der Umwandlung aus einem String wird das Ergebnis des entsprechenden Aufrufs der C-Funktion sscanf zurückgegeben.

Funktion string:

`string string(`*`expr`*`)`

Die Funktion konvertiert einen Ausdruck der Typen int, float oder string in einen string-Wert.

Parameter:

- expr: Ausdruck des Typs int, float oder string

Rückgabewert:

string-Wert von *expr*

Hinweis:

> Die gegenwärtige Implementierung begrenzt die Länge des zurückgegebenen Strings auf maximal 199 Zeichen. Das Ergebnis wird dynamisch erzeugt und sollte mittels ***free*** explizit wieder freigegeben werden.

11.3 Ein-/Ausgabe

Die mit (*) gekennzeichneten Funktionen dieses Abschnitts stehen nur in BOB+ zur Verfügung.

Funktion fclose:

```
int fclose(FILE fp)
```

Die Funktion schließt eine vorher mit ***fopen*** geöffnete Datei.

Parameter:

- fp: Handle der zu schließenden Datei

Rückgabewert:

> Bei Erfolg 0, sonst Wert ungleich 0

Hinweis:

> Die Funktion darf nicht mit den Standard-Streams stdin, stdout und stderr verwendet werden.

Funktion feof (*):

```
int feof(FILE fp)
```

Die Funktion testet, ob das Ende der angegebenen Datei erreicht ist..

Parameter:

- fp: Handle der zu überprüfenden Datei

Rückgabewert:

> Wert ungleich 0, wenn Dateiende erreicht, sonst 0

Funktion fgets (*):

```
string fgets(FILE fp)
```

Die Funktion liest eine Zeichenkette bzw. Zeile aus der in *fp* angegebenen geöffneten Textdatei. Das Lesen endet, wenn ein Zeilenumbruch oder das Dateiende erreicht bzw. die interne Puffergröße von 200 Zeichen ausgeschöpft ist.

Parameter:

- fp: Handle der Datei

Rückgabewert:

> Bei Erfolg wird die gelesene Zeile als Zeichenkette zurückgegeben. Bei Fehler oder erreichtem Dateiende ist das Ergebnis eine leere Zeichenkette.

Hinweise:

Das Erreichen des Dateiendes sollte mit der Funktion ***feof*** geprüft werden.
Ein ggf. aus der Datei gelesener Zeilenumbruch (‘\n‘) bleibt im Ergebnis erhalten.
Da für das Funktionsergebnis dynamisch Speicher belegt wird, sollte dieser mit Hilfe von ***free*** wieder freigegeben werden.

Funktion fopen:

```
FILE fopen(string filename, string mode)
```

Die Funktion versucht, eine Datei mit dem in *filename* angegebenen Namen im durch *mode* spezifizierten Modus zu öffnen. Sie entspricht in ihrem Verhalten der gleichnamigen C-Funktion.

Parameter:

- filename: Name der zu öffnenden Datei. Hier ist die Angabe eines einfachen Dateinamens oder eines Dateipfades (absolut oder relativ) im DOS-Stil möglich.
- mode: Der Parameter beschreibt den Öffnungsmodus der Datei. Dabei sind folgenden Modi möglich:
  - r: Öffnen einer existierenden Datei zum Lesen.
  - w: Anlegen einer Datei und Öffnen zum Schreiben. Existiert die angegebene Datei bereits, so wird sie überschrieben.
  - a: Öffnen einer Datei zum Schreiben ab dem aktuellen Dateiende (anhängen von Daten). Ist die Datei noch nicht vorhanden, so wird sie neu angelegt.
  - r+: Öffnen einer vorhandene Datei zum Lesen und Schreiben.
  - w+: Anlegen einer Datei und Öffnen zum Lesen und Schreiben. Existiert die angegebene Datei bereits, so wird sie überschrieben.
  - t: Öffnen der Datei im Textmodus (kombinierbar mit a, r, r+, w und w+).
  - b: Öffnen der Datei im binären Modus (kombinierbar mit a, r, r+, w und w+).

Rückgabewert:

FILE-Handle bei Erfolg, sonst NULL

Hinweis:

Mit dieser Funktion geöffnete Dateien sollten stets mit ***fclose*** geschlossen werden.

Funktion fputs (*):

```
int fputs(string text, FILE fp)
```

Die Funktion schreibt die Zeichenkette in *text* in die durch *fp* angegebenee geöffnete Textdatei..

Parameter:

- text: zu schreibende Zeichenkette
- fp: Handle der Datei

Rückgabewert:

Bei Erfolg der Zeichencode des zuletzt geschriebenen Zeichens, andernfalls der Code von EOF zurückgegeben.

Funktion freadval (*):

`var freadval(FILE fp)`

Die Funktion liest einen (beliebigen) Wert aus einer Binärdatei und gibt ihn als Ergebnis zurück. Das Format der Datei muss dem von der Funktion *fwriteval* erzeugten entsprechen.

Parameter:

- fp: Handle einer geöffneten Binär-Datei

Rückgabewert:

Bei Erfolg wird der gelesene Wert, andernfalls ***null*** zurückgegeben..

Hinweis:

Der Rückgabewert sollte i.a. einer Variablen zugewiesen und mit free wieder freigegeben werden, sobald er nicht mehr benötigt wird.

Funktion fwriteval (*):

`int fwriteval(value, FILE fp)`

Die Funktion schreibt einen (beliebigen) Wert in eine geöffnete Binärdatei .

Parameter:

- value: beliebiger Wert
- fp: Handle einer geöffneten Binär-Datei

Rückgabewert:

Bei Erfolg 1, sonst 0

Hinweis:

Mit Hilfe der Funktionen fwriteval und freadval lassen sich komplexe Datenstrukturen – insbesondere auch Objekthierarchien – serialisieren und deserialisieren. Dabei werden bei der Deserialisierung Querverweise zwischen Objekten automatisch wieder hergestellt.

Beispiel:

```
/* Serialization example using fwriteval and freadval */
/* first we define a class */
 class ListEntry
 {
   static _maxID;
   _a;
   _b;
   _ID;
   _next;
 }
```

```
 ListEntry::ListEntry(a,b)
 {
   if (!_maxID)
      _maxID = 0;
   _ID = ++_maxID;
   _a=a;
   _b=b;
   _next = null;
}

 ListEntry::getA() { return _a; }

 ListEntry::pr()
 {
    print("ID: ",_ID," ",getA(), " ",_b,"\n");
 }

 ListEntry::getNext() { return _next; }

 ListEntry::setNext(next) { _next = next; }

 ListEntry::append(var)
 {
    if (_next)
       _next->append(var);
    else
    {
      _next = var;
    }
 }

 ListEntry::~ListEntry()
 {
    if (_next)
       _next = delete _next;
    print("delete ",_ID,"\n");
}

main()
{
  lst = new ListEntry(-1,0);
  for(i=0;i<9;i++)               // append more objects to lst
  {
     lst->append(new ListEntry(i,i+1));
  }
  for(p = l;p;p=p->getNext())    // print contents
     p->pr();
  fp=fopen("obj.dat","wb");      // open file for write
  fwriteval(lst,fp);             // write entire list to file
  fclose(fp);
  lst = delete lst;              // free list
  print(lst,“\n“);               // should print „null“
  fp=fopen("obj.dat","rb");      // open file for read
  lst=freadval(fp);              // read contents and assign to lst
  fclose(fp);
  for(p = lst;p;p=p->getNext())  // print contents
     p->pr();
  lst = delete lst;              // free list
}
```

Funktion getc:

```
int getc(FILE* fp)
```

Die Funktion liest ein einzelnes Zeichen aus der durch *fp* spezifizierten Datei.

Parameter:

- fp: Handle der Datei

Rückgabewert:

> Zeichencode des gelesenen Zeichens. . Bei Fehler oder Dateiende wird EOF zurückgegeben

Funktion gets (*):

```
string gets()
```

Die Funktion liest eine Zeichenkette von maximal 199 Zeichen von der Standardeingabe. Das Lesen endet bei Eingabe eines Zeilenwechsels.

Rückgabewert:

> Bei Erfolg wird die gelesene Zeile als Zeichenkette andernfalls eine leere Zeichenkette.

Hinweise:

Der abschließende Zeilenumbruch (‘\n‘) ist nicht im Ergebnis enthalten.
Da für das Funktionsergebnis dynamisch Speicher belegt wird, sollte dieser mit Hilfe von ***free*** wieder freigegeben werden.

Funktion print:

```
void print(...)
```

Die Funktion schreibt die Werte der übergebenen Argumente auf die Standardausgabe.

Parameter:

> Kommaseparierte Liste von Argumenten beliebigen Typs. Die Argumente werden in der angegebenen Reihenfolge in die Standardausgabe geschrieben. Dabei werden für primitive Typen die jeweiligen Werte implizit in Zeichenketten konvertiert. Für alle anderen Typen (File, Vektor, Klasse, Objekt, Funktion) wird der jeweilige Typ mit der zugehörigen Adresse ausgegeben.

Hinweis:

> Im Gegensatz zu *printf* aus der C-Standardbibliothek kennt *print* keine Format-Spezifikation.

Funktion putc:

```
int putc(int c, FILE* fp)
```

Die Funktion schreibt ein einzelnes Zeichen in die durch *fp* spezifizierte Datei.

Parameter:

- c: Zeichencode des zu schreibenden Zeichens
- fp: Handle der Datei

Rückgabewert:

Zeichencode des geschriebenen Zeichens. Bei Fehler wird EOF zurückgegeben

11.4 Zeichenkettenfunktionen

Alle Funktionen dieses Abschnitts stehen nur in BOB+ zur Verfügung.

Funktion strcmp:

```
int strcmp(string s1, string s2)
```

Die Funktion vergleicht die Zeichenketten *s1* und *s2* zeichenweise, wobei zwischen Groß- und Kleinschreibung unterschieden wird. Sie verhält sich wie die gleichnamige Standard-C-Funktion.

Parameter:

- s1: erster Operand
- s2: zweiter Operand

Rückgabewert:

< 0, falls s1 < s2
0, falls s1 == s2
> 0, falls s1 > s2

Funktion stricmp:

```
int stricmp(string s1, string s2)
```

Die Funktion vergleicht die Zeichenketten s1 und s2 zeichenweise, wobei zwischen Groß- und Kleinschreibung *nicht* unterschieden wird. Sie verhält sich wie die gleichnamige Standard-C-Funktion.

Parameter:

- s1: erster Operand
- s2: zweiter Operand

Rückgabewert:

< 0, falls s1 < s2
0, falls s1 == s2
> 0, falls s1 > s2

Hinweis:

Die Funktion arbeitet nur fehlerfrei, wenn in s1 und s2 keine Zeichen mit einem ASCII-Code > 128 (z.B. Umlaute) vorkommen.

Funktion strlen:

```
int strlen(string s)
```

Die Funktion liefert die Länge der Zeichenkette *s*. Sie verhält sich wie die gleichnamige Standard-C-Funktion.

Parameter:

- s: Zeichenkette

Rückgabewert:

Anzahl der Zeichen in *s*

Funktion strsplit:

`vector strsplit(string source, string delimiter)`

Die Funktion zerlegt die in *source* übergebene Zeichenkette in Teilketten entsprechend der in *delimiter* angegebenen Trennzeichen. Sie verwendet intern die Standard-C-Funktion *strtok*.

Parameter:

- source: zu zerlegende Zeichenkette
- delimiter: Zeichenkette mit einem oder mehreren Trennzeichen

Rückgabewert:

Die Funktion gibt einen Vektor zurück, dessen Elemente vom Typ string sind und die einzelnen Teilketten enthalten.

Hinweis:

Sowohl der zurückgegebene Vektor als auch dessen Elemente werden dynamisch erzeugt. Sie sollten deshalb mit ***free*** wieder freigegeben werden.

Beispiel:

```
/* strsplit example */
main(;text,result,cnt,i)
{
   text = “This text will be splitted.“;
   result = strsplit(text,“ .“);
   cnt = vecsize(result);
   /* print result */
   for (i=0;i<cnt;i++)
      print(result[i],“\n“);
   /* free allocated memory */
   for (i=0; i<cnt; i++)
      free(result[i]);
   result = free(result);
   text = free(text);
}
```

11.5 Datum und Uhrzeit

Alle Funktionen dieses Abschnitts stehen nur in BOB+ zur Verfügung

Funktion ctime:

```
string ctime(int t)
```

Die Funktion erzeugt aus der numerischen Datums- und Zeitangabe in *t* eine druckbare Zeichenkette. Sie verhält sich wie die gleichnamige C-Funktion.

Parameter:

- t: numerischer Datums- und Zeitwert (wie time_t in C)

Rückgabewert:

Zeichenkettendarstellung von *t* in der Form
Wed Aug 10 21:52:54 2005\n

Hinweis:

Die zurückgelieferte Zeichenkette sollte mit ***free*** explizit freigegeben werden.

Funktion localtime:

```
vector localtime(int t)
```

Die Funktion erzeugt aus der numerischen Datums- und Zeitangabe in *t* einen Vektor mit den Bestandteilen der lokalisierten Zeitangabe.

Parameter:

- t: numerischer Datums- und Zeitwert (wie time_t in C)

Rückgabewert:

Vektor aus neun ganzzahligen Elementen entsprechend der tm-Struktur aus C mit folgender Bedeutung:

Index	0	1	2	3	4	5	6	7	8
Wert	sec	min	hour	mday	mon	year	wday	yday	isdst

Dabei erfolgt die Jahresangabe (*year*) relativ zum Jahr 1900. Das Feld *isdst* hat den Wert 1, wenn sich die Zeitangabe auf Sommerzeit bezieht, sonst 0.

Hinweis:

Der zurückgelieferte Vektor sollte mit ***free*** explizit freigegeben werden.

Funktion time:

```
int time()
```

Die Funktion liefert die aktuelle Zeit in Sekunden seit dem 1. Januar 1970 0:00 Uhr GMT. Sie verhält sich wie die gleichnamige C-Funktion.

Rückgabewert:

Vergangene Sekunden seit dem 1. Januar 1970 0:00 Uhr GMT

Funktion timer:

```
int timer()
```

Die Funktion liefert die verstrichene Zeit seit dem Programmstart in Millisekunden.

Rückgabewert:

Vergangene Zeit seit dem Programmstart in ms.

Hinweis:

Die Funktion verwendet clock()-Funktion aus C. Deshalb ist die tatsächliche Auflösung geringer als 1 ms. Unter MS-DOS liegt sie bei etwa 55 ms.

11.6 Mathematische Funktionen

Alle Funktionen dieses Abschnitts stehen nur in BOB+ zur Verfügung. Wenn nicht gesondert ausgewiesen, akzeptieren die Funktionen anstelle eines float-Arguments auch ein Argument von Typ int.

Funktion abs:

```
int abs(int x)
```
oder
```
float abs(float x)
```

Die Funktion berechnet den Absolutbetrag des als Argument übergebenen Wertes.

Parameter:

- x: numerischer Wert vom Typ int oder float

Rückgabewert:

Betrag von *x*. Der Typ des Rückgabewertes entspricht dem des Arguments.

Funktion atan:

```
float atan(float x)
```

Die Funktion berechnet den Arcus-Tangens des als Argument übergebenen Wertes.

Parameter:

- x: Argument

Rückgabewert:

Winkel im Bogenmaß im Bereich (-pi/2, +pi/2)

Funktion cos:

```
float cos(float rad)
```

Die Funktion berechnet den Kosinus des als Argument übergebenen Winkels.

Parameter:

- rad: Winkelangabe im Bogenmaß

Rückgabewert:

Kosinus von *rad*

Funktion exp:

```
float exp(float x)
```

Die Funktion berechnet Wert der Expotentialfunktion e^x.

Parameter:

- x: Exponent

Rückgabewert:

Wert von e^x

Funktion log:

```
float log(float x)
```

Die Funktion berechnet natürlichen Logarithmus ln(x).

Parameter:

- x: Argument

Rückgabewert:

Wert von ln(x)

Funktion pow:

```
float pow(float x, float y)
```

Die Funktion berechnet Wert der Expotentialfunktion x^y.

Parameter:

- x: Basis
- y: Exponent

Rückgabewert:

Wert von x^y

Funktion sin:

```
float sin(float rad)
```

Die Funktion berechnet den Sinus des als Argument übergebenen Winkels.

Parameter:

- rad: Winkelangabe im Bogenmaß

Rückgabewert:

Sinus von *rad*

Funktion sqrt:

```
float sqrt(float x)
```

Die Funktion berechnet die Quadratwurzel von *x*.

Parameter:

- x: Argument

Rückgabewert:

Quadratwurzel von *x*

Funktion tan:

```
float tan(float rad)
```

Die Funktion berechnet den Tangens des als Argument übergebenen Winkels.

Parameter:

- rad: Winkelangabe im Bogenmaß

Rückgabewert:

Tangens von *rad*

11.7 Systemfunktionen

Die Funktionen dieses Abschnitts ermöglichen es, Systemaufrufe durchzuführen. Sie sind nur in BOB+ verfügbar.

Funktion addr:

```
int addr(var)
```

Die Funktion liefert die Adresse der Variablen *var*. Sie wird u.a. für Interrupt-Aufrufe benötigt, die Adressen als Eingabeparameter benötigen.

Parameter:

- var: Variablenbezeichner

Rückgabewert:

Offset-Adresse der angegebenen Variablen.

Hinweise:

Die Funktion liefert nicht die vollständige Adresse sondern nur deren Offset-Anteil. Da BOB im Small-Modell übersetzt ist, sind alle internen Adressen Near-Zeiger. Die Segment-Adresse entspricht immer dem Wert des Registers DS.

Der zurückgelieferte Wert ist für Variablen vom Typ String der Offset des Anfangs der tatsächlichen Zeichenkette, für alle anderen Typen der des jeweiligen Datenobjekts selbst.

Funktion getreg:

```
int getreg(string regname)
```

Die Funktion liest den aktuellen Wert der angegebenen Register-Variablen.

Parameter:

- regname: Name eines Prozessorregisters, möglich sind:
  ax, bx, cx, dx, di, si, al, ah, bl, bh, cl, ch, dl, dh, cflag, flags, ds, es, ss, cs.
  Die Groß- und Kleinschreibung wird bei der Angabe der Registernamen ignoriert.

Rückgabewert:

aktueller Wert der angegebenen Register-Variablen

Hinweis:

Die Funktion liest nicht die tatsächlichen aktuellen Werte der Prozessorregister aus sondern greift auf eine interne Datenstruktur zu, die vorher innerhalb des Programms explizit (mit setreg bzw. segread) oder implizit (in Folge eines int86- oder int86x-Aufrufs) initialisiert wurde.

Funktion inport:

```
int inport(int portid)
```

Die Funktion liest ein Wort (16 Bit) ab Portadresse *portid*. Dabei wird das Low-Byte direkt von *portid* und das High-Byte von *portid*+1 gelesen.

Parameter:

- portid: Port-Nummer

Rückgabewert

gelesenes Wort

Hinweis:

Die Funktion benutzt intern die gleichnamige Turbo-C-Funktion und verhält sich entsprechend..

Funktion inportb:

```
int inportb(int portid)
```

Die Funktion liest ein einzelnes Byte von Portadresse *portid*.

Parameter:

- portid: Port-Nummer

Rückgabewert:

gelesenes Byte

Hinweis:

Die Funktion benutzt intern die gleichnamige Turbo-C-Funktion und verhält sich entsprechend..

Funktion int86:

`int int86(int intnr)`

Die Funktion führt löst einen Aufruf des durch intnr spezifizierten x86-Interrupts aus. Vor dem Aufruf werden die Prozessorregister AX, BX, CX, DX, DI, SI, FLAGS mit den Inhalten der entsprechenden internen Registervariablen geladen. Nach dem Aufruf werden die Registerwerte in die internen Variablen übernommen und die ursprünglichen Registerinhalte wieder hergestellt.

Parameter:

- intnr: Nummer des aufzurufenden Interrupts

Rückgabewert:

Wert des Registers AX nach dem Interrupt-Aufruf

Funktion int86x:

`int int86x(int intnr)`

Die Funktion führt löst einen Aufruf des durch intnr spezifizierten x86-Interrupts aus. Vor dem Aufruf werden die Prozessorregister AX, BX, CX, DX, DI, SI, FLAGS, DS und ES mit den Inhalten der entsprechenden internen Registervariablen geladen. Nach dem Aufruf werden die Registerwerte in die internen Variablen übernommen und die ursprünglichen Registerinhalte wieder hergestellt.

Parameter:

- intnr: Nummer des aufzurufenden Interrupts

Rückgabewert:

Wert des Registers AX nach dem Interrupt-Aufruf

Hinweis:

Vor Benutzung der Funktion sollten die internen Variablen der Segmentregister mit Hilfe der Funktion ***segread*** auf die tatsächlichen Werte gesetzt werden.

Beispiel:

```
/* interrupt call example – prints current directory path */
main()
{
   buf = newstring(200);   /* buffer for result */
   segread();              /* init variables for seg-registers */
   setreg("ah",0x47);      /* dosfunc nr. */
   setreg("dl",0);         /* get path for current drive */
   setreg("si",addr(buf)); /* load buffer adress to si */
   int86x(0x21);           /* call DOS-Interrupt */
   print(buf,"\n");
   free(buf);
}
```

Funktion outport:

```
void outportb(int portid, int value)
```

Die Funktion schreibt ein Wort (16 Bit) nach Portadresse *portid*. Dabei wird das Low-Byte direkt nach *portid* und das High-Byte nach *portid*+1 ausgegeben.

Parameter:

- portid: Port-Nummer
- value: auszugebender Wert

Hinweis:

Die Funktion benutzt intern die gleichnamige Turbo-C-Funktion und verhält sich entsprechend. Es wird das niederwertige Wort von *value* berücksichtigt.

Funktion outportb:

```
void outportb(int portid, int value)
```

Die Funktion schreibt ein einzelnes Byte nach Portadresse *portid*.

Parameter:

- portid: Port-Nummer
- value: auszugebender Wert

Rückgabewert:

gelesenes Byte

Hinweis:

Die Funktion benutzt intern die gleichnamige Turbo-C-Funktion und verhält sich entsprechend. Es wird das niederwertigste Byte von *value* berücksichtigt.

Funktion segread:

```
void segread()
```

Die Funktion lädt die aktuellen Werte der Segmentregister des Prozessors (DS, ES, SS, CS) in die entsprechenden internen Variablen.

Funktion setreg:

```
void setreg(string regname, int value)
```

Die Funktion setzt den Wert der angegebenen internen Register-Variablen.

Parameter:

- regname: Name eines Prozessorregisters, möglich sind:
  ax, bx, cx, dx, di, si, al, ah, bl, bh, cl, ch, dl, dh, cflag, flags, ds, es, ss, cs.
  Die Groß- und Kleinschreibung wird bei der Angabe der Registernamen ignoriert.
- value: neuer Wert

Hinweis:

> Die Werte der internen Register-Variablen werden beim Aufruf der Funktionen int86 bzw. int86x in die entsprechenden Prozessorregister übernommen.

11.8 Grafikfunktionen

Die Grafikfunktionen ermöglichen elementare Ausgaben im Grafikmodus. Intern werden BIOS-Aufrufe verwendet.

Alle Funktionen dieses Abschnitts stehen nur in BOB+ zur Verfügung.

Funktion setscrmode:

```
void setscrmode(int mode)
```

Die Funktion löscht den Bildschirm und schaltet die Anzeige in den angegebenen Modus um.

Parameter:

- mode: neuer Bildschirmmodus

Hinweis:

> Die konkrete Bedeutung von *mode* hängt vom verwendeten Grafik-Adapter bzw. dessen BIOS ab. Allgemein steht der Wert 7 für den MDA-Modus (monochromer Textmodus)[34].

Funktion setpixel:

```
void setpixel(int x, int y, int color)
```

Die Funktion setzt einen einzelnen Bildschirmpunkt auf den mit *color* spezifizierten Farbwert..

Parameter:

- x: x-Koordinate
- y: y-Koordinate
- color: Farbwert

Hinweise:

> Die Funktion ist nur bei eingeschaltetem Grafikmodus sinnvoll verwendbar.
>
> Die Werte für die Koordinaten sollten stets nichtnegativ sein und werden nach oben durch die jeweilige Bildschirmauflösung begrenzt. Der Punkt (0,0) bezeichnet die linke obere Bildschirmecke.
>
> Beim in *color* angegebenen Wert sind nur die niederwertigsten 8 Bit signifikant. Die konkrete Bedeutung hängt vom Bildschirmmodus ab[35].

[34] Auf dem Atari-Portfolio hängt die Zeilen- und Spaltenzahl im Textmodus von der eingestellten Bildschirm-Betriebsart ab. Der Grafikmodus hat dort immer eine Auflösung von 240 x 64 Pixeln und lässt sich mit den Werten 4 oder 10 aufrufen.

[35] Beim Atari-Portfolio steht der Wert 0 für einen nicht gesetzten, alle anderen Werte für einen gesetzten Punkt.

Funktion line:

```
void line(int x1, int y1, x2, y2, int color)
```

Die Funktion zeichnet eine Linie in der mit *color* spezifizierten Farbe vom Punkt *(x1,y1)* nach Punkt *(x2,y2)*.

Parameter:

- x1: x-Koordinate des Anfangspunktes
- y1: y-Koordinate des Anfangspunktes
- x2: x-Koordinate des Endpunktes
- y2: y-Koordinate des Endpunktes
- color: Farbwert

Hinweise:

> Die Funktion ist nur bei eingeschaltetem Grafikmodus sinnvoll verwendbar.
>
> Die Werte für die Koordinaten sollten stets nichtnegativ sein und werden nach oben durch die jeweilige Bildschirmauflösung begrenzt. Der Punkt (0,0) bezeichnet die linke obere Bildschirmecke.
>
> Beim in *color* angegebenen Wert sind nur die niederwertigsten 8 Bit signifikant. Die konkrete Bedeutung hängt vom Bildschirmmodus ab.

Beispiel:

```
/* Portfolio-example for using graphic functions */
main()
{
   /* resolution */
   px=240;
   py=64;
   /* switch to graphic mode */
   setscrmode(4);
   /* get current time */
   ms = timer();
   n=0;
   /* draw some lines */
   for (x=0;x<px;x+=2)
   {
      n++;
      x2= px -x;
      line(x,0,x2,py-1,1);
   }
   for (y=0;y<py;y+=2)
   {
      n++;
      y2= py-1 -y;
      line(0,y,px-1,y2,2);
   }
   /* get elapsed time */
   ms = timer() - ms;
   r = n*1000 / ms;
   gets();
   /* switch to text mode */
   setscrmode(7);
   print(r, " lines per second\n");
}
```

11.9 Sonstige Funktionen

In diesem Abschnitt werden einige Funktionen beschrieben, die sich keiner der obigen Kategorien zuordnen lassen. Alle diese Funktionen sind nur in BOB+ verfügbar.

Funktion argcnt:

```
int argcnt()
```

Die Funktion liefert einen die Anzahl der Argumente, die der aufrufenden Funktion übergeben wurden.

Rückgabewert:

Anzahl der Argumente der aufrufenden Funktion

Funktion arg:

```
var arg(int idx)
```

Die Funktion liefert das Argument mit dem Index idx der aufrufenden Funktion. Sie kann in Verbindung mit der Funktion argcnt zur Konstruktion von Funktionen mit variablen Argumentlisten verwendet werden..

Parameter:

- idx: Index eines Arguments der aufrufenden Funktion

Rückgabewert:

Argument mit dem Index *idx*

Hinweis:

Die Funktion prüft die Gültigkeit des in *idx* übergebenen Wertes.

Beispiel:

```
/* using functions argcnt() and arg(idx) */


f(x;i,n)            // only first argument has a name
{
   n = argcnt();
   print("\n",n," arguments:\n");
   for (i=0;i<n;i++)
     print("\t",arg(i));
   print("\n---\n");
   return x;
}

main()
{
   print(f(1,3,"test”));
 /* should print
   3 arguments :
      1     3     test
   ---
   1
 */
}
```

Funktion getargs:

`vector getargs()`

Die Funktion liefert einen Vektor mit allen beim Aufruf von BOB+ übergebenen Kommandozeilenargumenten zurück. Dabei ist das Element mit dem Index 0 der Name des ausgeführten Programms (i.a. also BP.EXE oder BPR.EXE).

Rückgabewert:

Vektor mit allen Kommandozeilenargumenten

Hinweis:

Der zurückgelieferte Vektor sollte mit ***free*** explizit freigegeben werden.

Funktion getusrargs:

`vector getusrargs()`

Die Funktion liefert einen Vektor mit allen beim Aufruf übergebenen Benutzer-Argumenten zurück. Dabei ist das Element mit dem Index 0 das erste Kommandozeilenargument, das hinter dem Kennzeichen # angegeben wurde.

Rückgabewert:

Vektor mit allen Benutzer-Argumenten

Hinweis:

Der zurückgelieferte Vektor sollte mit ***free*** explizit freigegeben werden. Er kann leer sein, wenn keine benutzerdefinierten Argumente angegeben wurden.

Beispiel:

```
/* using functions getargs() and getusrargs() */
main(;vec1,vec2,n,i)
{
   vec1 = getargs();
   vec2 = getusrargs();
   print("args:\n");
   n = vecsize(vec1);
   for (i=0;i<n;i++)
      print(vec1[i],"\n");
   print("usrargs:\n");
   n = vecsize(vec2);
   for (i=0;i<n;i++)
      print(vec2[i],"\n");
   free(vec1);
   free(vec2);
}
```

Funktion rand:

`int rand()`

Die Funktion berechnet eine Pseudo-Zufallszahl. Sie verhält sich wie die gleichnamige C-Funktion.

Rückgabewert:

Zufallszahl zwischen 0 und 0x7FFF

Hinweis:

> Bei mehreren Aufrufen der Funktion ergibt sich eine annähernd gleichverteilte Zufallsfolge, die vom Initialwert des Zufallsgenerators abhängt. Ohne explizite Initialisierung des Zufallsgenerators (vgl. Funktion srand) wird nach dem Programmstart immer die gleiche Folge generiert.

Funktion srand:

```
void srand(int seed)
```

Die initialisiert den internen Zufallsgenerator mit dem Wert *seed*. Sie verhält sich wie die gleichnamige C-Funktion.

Parameter:

- seed: Initialwert für den Zufallsgenerator

Hiweis:

> Beim Programmstart wird der Zufallsgenerator implizit mit dem Wert 1 initialisiert. Um tatsächlich „zufällige Zufallsfolgen“ zu erhalten, ist eine Initialisierung mit einem zeitabhängigen Wert üblich.

Beispiel:

```
/* random number generator example */
main()
{
   for(l=0;l<5;l++)
   {
      seed = timer();
      srand(seed);
      print(“initial value: “,seed,“\n“);
      for (i=0;i<10;i++)
      {
         for (j=0;j<10;j++)
            print(rand(),“ “);
         print(“\n“);
      }
   }
}
```

Funktion compile:

```
int compile(string filename)
```

Die Funktion bindet zur Laufzeit Quellcode in das aktuelle Programm ein. Die durch *filename* spezifizertee Datei wird gelesen und in Bytecode übersetzt.

Rückgabewert:

> 1 bei Erfolg, sonst 0
> Dabei bedeutet „Erfolg“ das erfolgreiche Übersetzen der Quelle in Bytecode.

Hinweise:

> Die Funktion unterscheidet sich grundsätzlich von include-Anweisungen anderer Programmiersprachen. Der in *filename* enthaltene Code wird nicht zur Übersetzungs- sondern zur Laufzeit eingebunden.
>
> Der Aufruf von ***compile*** kann an jeder beliebigen Stelle erfolgen, an der ein Funktionsaufruf möglich ist. Die Einbindung der übersetzten Quellcodes erfolgt immer in den globalen

Kontext des Programms. Bereits vorhandene gleichnamige Symbole (Bezeichner) werden dabei durch die neuen ersetzt. Diese Eigenschaft lässt sich verwenden, um z.B. abhängig von Bedingungen bestimmte Funktionen zur Laufzeit auszutauschen, ohne den sie verwendenden Clientcode ändern zu müssen.

Beispiel:

```
/* using compile() example – main module */
testfunc(n)
{
   print(“testfunc(“,n,“) from main module\n“);
}

main()
{
   testfunc(“Hello World“);
   if (compile(„tstfunc.bp“))
      testfunc(“Hello World“);
}
```

```
/* using compile() example – module tstfunc.bp */
testfunc(n)
{
   print(“testfunc(“,n,“) from tstfunc.bp\n“);
}
```

Funktion loadmodule:

`int loadmodule(string filename)`

Die Funktion bindet zur Laufzeit ein bereits vorkompiliertes Bytecode-Modul in das aktuelle Programm ein.

Rückgabewert:

1 bei Erfolg, sonst 0
Dabei bedeutet „Erfolg“ das erfolgreiche Lesen des Bytecodes.

Hinweise:

Die Funktion unterscheidet sich von der Verarbeitungsanweisung #use insofern, als der in *filename* enthaltene Bytecode nicht zur Übersetzungs- sondern zur Laufzeit eingebunden wird.

Der Aufruf von ***loadmodule*** kann an jeder beliebigen Stelle erfolgen, an der ein Funktionsaufruf möglich ist. Die Einbindung des geladenen Bytecodes erfolgt immer in den globalen Kontext des Programms. Bereits vorhandene gleichnamige Symbole (Bezeichner) werden dabei durch die neuen ersetzt. Diese Eigenschaft lässt sich verwenden, um z.B. abhängig von Bedingungen bestimmte Funktionen zur Laufzeit auszutauschen, ohne den sie verwendenden Clientcode ändern zu müssen.

12 Quellen

[1] Bob-Source http://www.atari-portfolio.co.uk/library/language/bob.zip
[2] Betz, D. M.: Bob: A Tiny Object-Oriented Language. In: Dr.Dobbs Journal, Sep. 1991, S.26ff. http://www.ddj.com/documents/s=1009/ddj9415e/9415e.htm
[3] Neue Bob-Sourcen von D. M. Betz: XLISP HomePage, http://www.mv.com/ipusers/xlisper/bob.zip