Seminararbeit

BASIC V2

Hannes Dorfmann (0716197)
hannes.dorfmann@uibk.ac.at

18. Februar 2011

Betreuer: Dr. René Thiemann

Zusammenfassung (Englisch)

BASIC was developed in the 1960's by John George Kemeny and Thomas Eugene Kurtz at the Dartmouth College. The acronym BASIC stands for "**B**eginner's **A**llpurpose **S**ymbolic **I**nstruction **C**ode" and it quickly became a widespread programming language in the United States. It was designed as an imperative universal interpreter programming language, which had programming beginners and non computer scientists as target audience. Additionally it should offer the power to solve all kind of problems as quickly and easily as possible. Since BASIC was released under a royalty-free license and is not enhanced and standardized by a central institution, hundreds of BASIC dialects were developed, like Commodore BASIC V2 which is the subject of this present paper. The bloom of BASIC was in the 1970's - 1980's when BASIC-Interpreters were available for nearly every computer architectures of those days, like the legendary Commodore C64. But over the years BASIC lost the supremacy and fell away into insignificance. Nowadays BASIC dialects are used as scripting languages within different software applications. With this paper we will try to figure out the strengths and weaknesses of BASIC as well as the advantages and disadvantages by comparing BASIC with well known programming languages

like C [1], Java [2] or C++ [3]. We will report on the history of BASIC, will check out whether the goals of the language have been reached and will take a closer look at the features of BASIC.

Inhaltsverzeichnis

[1] C99: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf
[2] http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html
[3] ANSI ISO IEC 14882 2003 (2003-10-15)
http://openassist.googlecode.com/files/C%2B%2B%20Standard%20-%20ANSI%20ISO%20IEC%2014882%202003.pdf

1 Die BASIC Welt

Basic ist eine imperative Programmiersprache, d.h. es beschreibt eine Berechnung bzw. einen Programmablauf durch eine Folge von Anweisungen. Basic wurde 1964 am Dartmouth College von John George Kemeny und Thomas Eugene Kurtz entwickelt und wurde als Interpreter-Sprache umgesetzt [7]. Das Akronym BASIC steht für "**B**eginner's **A**llpurpose **S**ymbolic **I**nstruction **C**ode" und ist als Allzweckprogrammiersprache für Programmieranfänger entworfen worden. Dank seiner einfachen Syntax hat sich BASIC schnell weltweit verbreitet.

1.1 Geschichte

Das erste BASIC Programm lief am 1. Mai 1964 auf einen GE-225-Computer von General Electric im Keller des Dartmouth College [7]. Am selben Institut, unter Aufsicht der "Väter" John George Kemeny und Thomas Eugene Kurtz, wurde BASIC von den Informatikstudenten weiterentwickelt. BASIC wurde, im Gegensatz zu anderen Programmiersprachen der damaligen Zeit, für Schulen kostenlos zur Verfügung gestellt, welche daraufhin BASIC dankend in ihr Unterrichtsproramm aufnahmen. Große Computerhersteller boten wegen der lizenzgebührfreien Verwendung und der leichten Erlernbarkeit bald BASIC Interpreter an. So kamen viele mittelständische Unternehmen mit BASIC in Kontakt. Den Höhepunkt erreichte BASIC Ende der 1970er und Anfang der 1980er, als die ersten Heimcomputer, die nahezu alle als Benutzeroberfläche und Programmierumgebung einen BASIC Interpreter besaßen, ihren Einzug in die heimischen Wohnzimmer feierten. An dieser Stelle muss man den legendären Commodore 64, der meistverkaufste Heimcomputer aller Zeiten, erwähnen, welcher direkt mit dem Einschalten einen BASIC Interpreter startete und durch diesen Interpreter das System steuerte. Die Verbreitung von BASIC war so groß, dass praktisch jedem damals verkauftem Computersystem ein BASIC Interpreter zur Verfügung stand. Auch Microsoft lieferte anfangs in seinem MS-DOS einen BASIC Interpreter aus. Doch nach und nach wurden andere Hochsprachen wie C für Heimcomputer verfügbar und liefen BASIC durch deutlich bessere Performance den Rang ab. Zudem mussten die großen Heim- und Bürocomputer Hersteller Atari und Commodore mit der Zeit ihre Vormachtstellung abtreten. So geriet BASIC etwas in Vergessenheit, ehe 1991 Microsoft, deren aller erstes entwickeltes Produkt ein Altair BASIC 2.0 Interpreter war, Visual Basic für schnelle Entwicklung von Applikationen auf Windows Systemen auf den Markt brachte. Vor allem dank der visuellen Entwicklungsumgebung feierte Visual Basic weltweit Erfolge und brachte somit BASIC wieder auf die Erfolgsspur. Mit Visual Basic 2, erschienen 1992, wurden Objekte eingeführt um konkurrenzfähiger zum objektorientierten C++ zu werden. Jedoch blieb der Performanceunterschied zu anderen Hochsprachen noch bestehend, da Visual Basic, so wie das ursprüngliche BASIC, zur Laufzeit durch einen Interpreter interpretiert wurde. Erst ab Visual Basic 5 (1997) wurden Visual Basic Programme durch einen eigenen Compiler direkt zu Maschinencode kompiliert. Doch auch andere Hochsprachen konnten in Laufe der Zeit passende Entwicklungsumgebungen und Bibliotheken für schnelle und einfache GUI-Entwicklung zur Verfügung stellen und so ver-

sank auch Visual Basic mit der Zeit in der Bedeutungslosigkeit, ein Schicksal, welches bereits etliche andere BASIC Dialekten zuvor erreicht hatte. Heute begegnet uns BASIC im Computeralltag meist in Form von Skriptsprachen, wie zum Beispiel Visual Basic for Applications (VBA), welches unter anderem in Microsoft Office Paket und AutoCAD als Makrosprache Verwendung findet.

1.2 Ziele

Zur damaligen Zeit entstanden die ersten Hochsprachen, wie FORTRAN [1] (1954), LISP [2] (1959) , COBOL [3] (1959) oder Algol 60 [4] (1963), die jedoch alle eher auf spezifische Anwenderschichten wie naturwissenschaftliche Ingenieure oder den betriebswirtschaftlichen Bereich zugeschnitten waren und nicht für Anfänger leicht zu erlernen waren [8]. Genau da setzt BASIC an. Es soll möglichst einfach zu erlernen und universell für alle Anwendungsbereiche einsetzbar sein. So wurde auch bei der Strukturierung der Syntax darauf geachtet, Kommandos möglichst nach der menschlichen Sprache zu richten. So wurde die englische Sprache bzw. Abkürzungen davon verwendet um die BASIC Befehle möglichst einfach, verständlich und übersichtlich zu halten. Man hat sich dazu entschieden BASIC als Interpreter-Sprache zu entwerfen, um so Hardware- und Betriebssystemunabhänigkeit zu erreichen, da es zur damaligen Zeit noch keine "Standardsystemarchitektur" à la x-86 gab, sondern eine Vielzahl an verschiedenen Systemen. Zudem konnte man mit BASIC als Interpreter-Sprache erreichen, dass ein BASIC Interpreter klare Fehlermeldungen von sich gibt, um den Programmieranfänger nicht durch kryptische Fehlermeldungen abzuschrecken.

[1] http://gcc.gnu.org/wiki/GFortranStandards
[2] http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+INCITS+226-1994+(R2004)
[3] http://www.cobolstandards.com
[4] http://en.wikipedia.org/wiki/ALGOL_60

2 Commodore BASIC V2

Commodore BASIC V2 ist ein BASIC Dialekt, basierend auf Altair BASIC von Microsoft aus dem Jahre 1975 [9]. Es kam primär auf den Commodore C64 und VC20 zum Einsatz. Auf den C64 war der BASIC-Interpreter direkt im ROM enthalten und diente gleichzeitig als Benutzerschnittstelle. BASIC V2 arbeitet, wie das Ur-BASIC, zeilenorientiert, d.h. jeder Befehlszeile wird eine eindeutige Zeilennummer vorangestellt. Es gibt aber keinen Befehl, der eine automatische Nummerierung bzw. Renummerierung durchführt, deshalb muss sich der Programmier um die Nummerierung kümmern. Die Zeilennummern können später für Sprunganweisungen wie z.B. GOTO verwendet werden, welche eine zentrale Rolle und den Programmfluss eines BASIC V2 Programmes regelt, da die Sprache selbst nur IF-Anweisung und die FOR-Schleife als Komponenten der heutzutage üblichen strukturierten Programmierung vorsieht. Auch leichte prozedurale Programmierung ist vorgesehen, welche mit GOSUB zeilennummernorientiert realisiert wurde. Intern im BASIC-Interpreter läuft ein BASIC Programm immer von der Zeile mit der niedrigsten Zeilennummer bis zur Zeile mit der höchsten Zeilennummer. Dabei kann eine Zeile maximal 80 Zeichen lang sein. Ist eine Zeile länger als 80 Zeichen so wird der Rest einfach stillschweigend ignoriert. Zeilennummern können im Bereich zwischen 0 und 63999 vergeben werden [5]. In einer Zeile kann mehr als ein Befehl stehen, jedoch müssen die Befehle mit einem Doppelpunkt ":" voneinander getrennt werden. Anweisungen und Variablen werden in Großbuchstaben geschrieben. Jede Anweisung wird im Interpreter zu einem Token von 1 Byte Länge, welche im Arbeitsspeicher (RAM) abgespeichert werden. Maximal können bei einem C64 38911 Zeichen (Bytes) eingegeben werden, da der Arbeitsspeicher auf diese Zahl begrenzt ist [5].

2.1 Allgemeine Syntax und Syntax-Schablone

Für die anschließende Syntaxerklärung führen wir folgende Syntax-Schablone ein, die einer vereinfachten EBNF[5] ähnelt:

- | : Dieses Zeichen ist wie in EBNF äquivalent zu einem “oder”.
- [...] : Eckige Klammern bedeuten, dass der darin enthaltene Text (z.B. Parameter) optional sind.
- { ... } : Geschwungene Klammern bedeuten, dass der enthaltene Text beliebig oft wiederholt werden kann, inklusive kein Mal.
- (...) : Normale Klammern werden verwendet um die Präzendenz eindeutig auszudrücken und soll lediglich den Vorrang bzw. Priorität aufzeigen. Will man explizit zeigen dass eine Klammer als Zeichen erforderlich ist, so wird die Klammer unter Anführungszeichen gestellt z.B. ")". Gleiches gilt auch für das Istgleich-Zeichen "=".

[5] `http://de.wikipedia.org/wiki/Erweiterte_Backus-Naur-Form`

- A-Z bedeutet alle Buchstaben zwischen A und Z, analog bedeutet 0-9 alle Ziffern zwischen 0 und 9, sowie 0 – 63999 alle natürlichen Zahlen zwischen 0 und 63999.
- Alle anderen Zeichen wie zum Beispiel Semikolon ; oder Doppelpunkt : oder Komma oder Dollar $ oder Prozent % bedeuten, dass genau dieses Zeichen gemeint ist und werden nicht unter Anführungszeichen geschrieben.

Commodore BASIC V2 Programme haben folgende Form:

EBNF 1:
PROGRAMM = {LINE}
LINE = LINENUMBER (**REM** | INSTRUCTION | **END**) \n
LINENUMBER = 0 - 63999
INSTRUCTION = (COMMAND | ASSIGNMENT
| FLOW_CONTROL) { : INSTRUCTION }
ASSIGNMENT = VARIABLE [$ | %] "="(VARIABLE
| NUMERICAL_EXPRESSION
| STRING_EXPRESSION)
VARIABLE = A-Z{ A-Z | 0-9 }
FLOW_CONTROL = **IF** | **FOR** | **GOTO**
| **GOSUB** LINENUMBER {LINE} **RETURN**

Zum leichteren Verständnis wird hier nur eine vereinfachte und nicht komplette EBNF behandelt. Man soll lediglich einen Überblick über die Syntax in BASIC bekommen. Die Verwendung von Variablen und die wichtigsten Befehle (in EBNF 1 als COMMAND bezeichnet) werden nachfolgend in diesem Kapitel behandelt und erklärt.

Wie man anhand der EBNF 1 sieht, ist ein BASIC Programm in Zeilen aufgeteilt. Jede Zeile beginnt mit einer eindeutige Zeilennummer. Der BASIC Interpreter führt dann das Programm von der kleinsten Zeilennummer bis zur größten Zeilennummer aus. Eine END - Anweisung in einer Zeile lässt den Interpreter das Programm in dieser Zeile beenden. Ein END ist nicht zwingend erforderlich um ein BASIC Programm zu beenden, jedoch empfiehlt es sich als Programmierer immer ein END in seinem Programm-Code zu haben, um ein korrektes Terminieren zu gewährleisten (siehe Beispiel 6).

2.2 Variablen

BASIC bietet wie andere Programmiersprachen auch Variablen an. Sie können zu jeden Zeitpunkt im Programmablauf definiert und dessen Wert verändert werden. Alle Variablen sind global. Eine Variable wird in folgender Form definiert:

EBNF 2 (Variablendeklaration):
[**LET**] VARIABLENAME[$|%] "=" VARIABLE
| NUMMERISCHER_AUSDRUCK
| STRING_AUSDRUCK

Code-Beispiel 1:
10 **LET** A = 5 : B = 6

Das Schlüsselwort LET ist nicht zwingend nötig. Wie man in Beispiel 1 sieht, kann man die Variable A mit LET deklarieren oder wie beispielsweise die Variable B durch erstmalige Wertzuweisung. Bei der Wahl des Namens für eine Variable muss darauf geachtet werden, dass der BASIC Interpreter nicht auf Leerzeichen als Trennzeichen angewiesen ist.

Code-Beispiel 2:
10LETAB=2:IF2=ABTHENPRINT"HELLO" :**PRINT**"WORLD"
20 **LET** AB = 2 : **IF** 2 = AB **THEN PRINT** "HELLO" : **PRINT** "WORLD"

Für den BASIC Interpreter sind die Zeilen 10 und 20 aus Beispiel 2 gleich. Dies ist möglich, da kein Anweisungsschlüsselwort im Variabelenname vorkommen darf. Zum Beispiel kann eine Variable nicht den Namen "HAND" annehmen, da dort das logische Verknüpfung-Schlüsselwort AND vorkommt.
Besonders zu beachten ist, dass der BASIC Interpreter nur die ersten zwei Zeichen des Variablennamens verwendet. Das bedeutet, die Variablenamen können beliebig lang sein (bis zum BASIC Maximum von 80 Zeichen pro Zeile), jedoch sind nur die ersten zwei Zeichen für eine Variable ausschlaggebend [4]. Zum Beispiel ist die Variable GEWINN und GEHALT für den Interpreter ein und dieselbe Variable, da nur die ersten zwei Zeichen, nämlich "GE", von Bedeutung sind.

2.2.1 Typisierung

In BASIC gibt es eine simple Typisierung für Variablen von Typ Ganzzahlen, Gleitkommazahlen und Strings. Dies geschieht, wie man anhand der EBNF 2 sieht, über das Zusatzzeichen Dollar "$" für Strings bzw. Prozent "%" für Integer, welches am Ende des Variablennames hinzugefügt wird. Wird kein Typisierungszusatzzeichen ($ oder %) am Ende des Variablennames angefügt so handelt es sich automatisch um eine Gleitkommavariable. Zahlenvariablen werden mit 0 und Strings mit dem leerer String initialisiert. BASIC unterscheidet Variablen auch nach deren Typ. So können gleichzeitig mehrere Variablen mit denselben Namen existieren, falls diese einen unterschiedlichen Typ aufweisen. Daher sind die Variablen A, A% und A$ drei verschiedene Variablen.

Der BASIC Interpreter des C64 rechnet prinzipiell mit Gleitkommazahlen, sodass Ganzzahlen beim Rechnen intern vom Interpreter zu Gleitkommazahlen umgewandelt werden. Dadurch sind Berechnungen mit Ganzzahlen langsamer als Berechnungen mit Gleitkommazahlen.
BASIC unterstützt die wissenschaftliche Notation. So kann eine Zahl 5123.4 auch als $5.1234 * 10^3$ oder wie in BASIC 5.1234E3 , wobei das "E" für "Zehner-Exponent" steht dargestellt werden. Ganzzahlwerte brauchen weniger Speichplatz als Gleitkommazahlen. Dies sollte man als Programmierer beachten, da der Arbeitsspeicher begrenzt ist. Ganzzahlen sind auf den C64 16 Bit Integers und haben dadurch einen Wertebereich von -32767 bis 32767 [3]. Der BASIC Interpreter bietet eine eingebaute Sicherung um einen Überlauf aus den gennaten

Bereichen zu vermeiden: Die Fehlermeldung *ILLEGAL QUANTITY* unterbricht die Ausführung und vermeidet so einen Überlauf.

2.3 Ein- und Ausgabe

2.3.1 PRINT

Um etwas auf der Bildschirmkonsole auszugeben bietet BASIC einen simplen PRINT Befehl an:

EBNF 3 (PRINT):
PRINT PRINTABLE { (; | ,) PRINTABLE } [; | ,]

PRINTABLE = "TEXT" | NUMERISCHER_AUSDRUCK | VARIABLE

Wie man sieht kann man mit PRINT einen String, einen numerischen Ausdruck oder den Wert einer Variable auf den Bildschirm präsentieren lassen, wobei ein numerischer Ausdruck mathematisch ausgewertet wird und somit das Ergebnis ausgegeben wird. Mehrere Werte können durch eine PRINT Anweisung ausgegeben werden, indem die auszugebenden Werte durch Semikolon ; oder Komma , voneinander getrennt werden. Dabei bedeutet ein Semikolon das die Ausgabe ohne Unterbrechung bzw. Leerraum in der aktuellen Zeile weitergeht, ansonsten macht ein PRINT automatisch einen Zeilenumbruch und der Cursor steht für den nächsten PRINT Befehl in der neuen Zeile. Aus diesem Grund kann am Ende einer PRINT Anweisung ein Semikolon alleine stehen. Dasselbe gilt für das Komma, wobei ein Komma die auszugebene Werte durch einen Abstand voneinander trennt. Dabei entspricht der Abstand, der durch ein Komma realisiert wird, einem Viertel der Bildschirmbreite [3]. Für eine formatierte Ausgabe stehen folgende Befehle zur Verfügungung:

- TAB “(” n “)”, welcher n Tabulatoren einfügt, und
- SPC “(” n “)”, welcher n Leerzeichen einfügt.

Beide Formatierungsanweisungen sind nur in Kombination mit PRINT anwendbar:

Code-Beispiel 3:
```
PRINT "Drei";SPC(3);"+";SPC(3);"Zwei";TAB(2);"=";3+2
```

2.3.2 INPUT

Mit INPUT kann man einen Wert über eine Konsoleneingabe in eine Variable einlesen.

EBNF 4 (INPUT):
INPUT ["AUSZUGEBENDER TEXT" ;] VARIABLE[$|%]

Die Variable muss vorher nicht explizit deklariert worden sein. Existiert die Variable noch nicht, so wird sie durch das Einlesen via INPUT erstellt. Der Typ der Variable bestimmt ob ein String, eine Ganzzahl oder eine Gleitkommazahl eingelesen wird.

2.4 Kommentare

Kommentare zum Quellcode sind für einen Programmierer lebensnotwendig. In BASIC werden Kommentare durch das Schlüsselwort REM (remark) gesetzt. REM Anweisungen sind daher keine echten Anweisungen, sondern dienen einzig dem Programmierer zur Dokumentation und werden vom BASIC Interpreter schlicht ignoriert. Alles was nach dem Schlüsselwort REM in derselben Zeile steht ist für den Interpreter der Kommentartext.

2.5 Arrays

Arrays gibt es für die bereits erwähnten drei Datentypen und müssen durch das Schlüsselwort DIM definiert werden:

EBNF 5 (Arraydeklaration):
DIM VARIABLENAME[%|$] "(" GANZZAHL { , GANZZAHL} ")"

Bei der Deklaration eines Arrays durch das Schlüsselwort DIM wird in den Klammern ( ) die Dimensionsgröße angegeben. Dabei kann theoretisch ein Array bis zu 256 Dimensionen haben, jedoch wäre in der Praxis der Arbeitsspeicher des C64 nicht ausreichend groß dafür. Über den Variablennamen und den sich in Klammern befindlichen Index kann auf ein Arrayelement zugegriffen werden. Dabei geht der Index von 0 bis Dimensionsgröße - 1. Indexes werden intern mit 16-Bit Integer (ohne Vorzeichen) verwaltet. Dadurch kann eine Dimension maximal 65535 groß sein [5].

Code-Beispiel 4:
```
10 DIM A(2) : A(0) = 0 : A(1) = 1
20 DIM S$(2,2) : S$(0,0)="NULL NULL" : S$(1,0)="EINS NULL"
30 PRINT A(0); A(1), S$(0,0) , S$(1,0) , S$(0,1)
```

Ausgabe (Beispiel 4):
```
0  1          NULL NULL            EINS NULL
```

Dem aufmerksamen Leser wird aufgefallen sein, dass in Beispiel 4 in Zeile 30 versucht wird, der Wert des Arrays S$(0,1), welcher zuvor nie gesetzt wurde, auszugeben. Wie wir aber schon in Kapitel 2.2.1 gelernt haben, werden in BASIC Variablen mit 0 für Zahlen bzw. dem leeren String für Strings initialisiert. Dasselbe gilt auch für Arrays und somit entspricht der Wert von S$(0,1) dem leeren String, welcher auch am Ende der Zeile 30 ausgegeben wird.

2.6 FOR - Schleife

Die FOR-Schleife ist die einzige Schleife die BASIC in der Syntax vorsieht.

EBNF 6 (FOR - Schleife):
FOR ZAHLER = STARTWERT TO ENDWERT [STEP SCHRITTWEITE]
{**LINE**}
NEXT ZAHLER

Bei einer FOR-Schleife werden die Anweisungen so oft wiederholt bis die Zählvariable vom Startwert den Endwert erreicht hat. Das Schlüsselwort NEXT zeigt das Ende der FOR Schleife an und erhöht die angegebene Zählvariable. So lassen sich auch Schleifen innerhalb von Schleifen realisieren. Mit dem optionalen Schlüsselwort STEP kann festgelegt werden, um wie viel die Zählvariable bei jedem Schleifendurchlauf erhöht wird.

Wird STEP nicht angegeben, so wird automatisch bei jedem Schleifendurchlauf um 1 inkrementiert. Wird für Schrittweite (im STEP) eine negative Zahl angegeben kann eine FOR-Schleife realisiert werden, die rückwärts zählt.

Code-Beispiel 5 (Schleife innerhalb einer Schleife):

```
10 FOR Z = 0 TO 4 STEP 2
20 FOR X = 1 TO 2
30 PRINT Z;"-";X,
40 NEXT X
50 NEXT Z
```

Ausgabe (Beispiel 5):

```
0-1     0-2     2-1     2-2     4-1     4-2
```

2.7 Sprunganweisung

Vielleicht hat sich der aufmerksame Leser schon gefragt, warum wir eigentlich die ganze Zeit unsere Programmcode-Zeilen mit Zeilennummern versehen? Das liegt daran, dass der BASIC Interpreter zeilenorientiert mit Zeilennummern arbeitet. Dadurch kann man mit GOTO innerhalb des Programmes zu einer anderen beliebigen nummerierten Programmzeile springen.

EBNF 7 (GOTO):

GOTO Zeilennummer

GOTO wird benutzt um Schleifen, als Alternative zur FOR-Schleife, oder Programmverzweigungen (in Verbindung mit IF) zu realisieren. Existiert die Zeilennummer nicht in welche gesprungen werden soll, so wird die Fehlermeldung *UNDEF'D STATEMENT ERROR* ausgegeben und das Programm terminiert.

2.8 Verzweigungen mit der IF - ANWEISUNG

Eine Verzweigung oder Fallunterscheidung kann mit einer IF-Anweisung erreicht werden:

EBNF 8 (IF):

```
IF BEDINGUNG   ( THEN Zeilennummer )
             | ( GOTO Zeilennummer )
             | ( THEN ANWEISUNG { : ANWEISUNG} )
```

Bedingungen können mit kleiner <, größer >, gleich =, ungleich <>, kleiner-gleich <=, größer-gleich >= und NOT definiert und können mit den Schlüsselworten AND bzw. OR mit weiteren Bedingungen verknüpf werden.

Ein ELSE ist in BASIC nicht vorgesehen, kann jedoch leicht durch eine zweite IF Anweisung (umgekehrte Bedingung) realisiert werden.

Code-Beispiel 6:

```
10 INPUT "EINE ZAHL ZWISCHEN 0 UND 10 "; A%
20 IF A%<0 OR A%>10 GOTO 10
30 IF A%<5 THEN PRINT A%; :PRINT" < 5"
40 IF A%>=5 THEN 100
50 END
100 PRINT A%;" >= 5"
```

In Zeile 10 von Beispiel 6 soll eine Ganzzahl eingegben werden (wird in die Variable A% gespeichert) die zwischen 0 und 10 liegt. Dies wird in der Zeile 20 mit einer IF Abfrage überprüft und falls die Zahl nicht im gewünschten Berreich liegt, wird wieder zurück in Zeile 10 gesprungen und erneut eine Zahleneingabe im Bereich zwischen 0 und 10 verlangt. Dies wird so lange wiederholt, bis eine Eingabe im gewünschten Bereich erfolgt. In Zeile 30 wird überprüft, ob die eingegebene Zahl kleiner als 5 ist. Ist das der Fall, so wird dies gleich mit einem PRINT ausgegeben (immer noch Zeile 30). In Zeile 40 wird überprüft ob die eingegebene Zahl größer als 5 ist und gegebenfalls in die Zeile 100 gesprungen. In Zeile 100 erfolgt dann die Ausgabe "A% >= 5" mit PRINT.

Zeile 60 weist ein END auf und stellt somit sicher, dass beim erreichen dieser Zeile terminiert wird. Wäre die Zeile 60 nicht vorhanden, würde das Programm immer von Zeile 10 bis Zeile 110 abblaufen. Dies würde aber bei einer Eingabe von beispielsweise A% = 3 zu einem nicht erwünschten Programmablauf führen, da zwar nicht in Zeile 40 nach Zeile 100 gesprungen würde, dennoch Zeile 100 durch den normalen Programmablauf erreicht werden würde und somit die Ausgabe "A% >= 5" erfolgen würde.

2.9 Unterprogramme

BASIC besitzt Ansätze einer prozeduralen Programmiersprache, die sogenannten Unterprogramme.

EBNF 9 (GOSUB):

GOSUB Zeilennummer
{**LINE**}
Zeilennummer ANWEISUNG { : ANWEISUNG }
{**LINE**}
RETURN

Ein Unterprogramm wird über die Anweisung GOSUB Zeilennummer aufgerufen. Dabei ist die Zeilennummer der Anhaltspunkt für den BASIC Interpreter um zu zeigen, dass von dieser Zeilennummer aus ein Unterprogramm startet. Das bedeutet, dass in BASIC, anders als beispielsweise in C, eine Prozedur oder Unterprogramm keinen Funktionsheader, wie Name, Parameter und Rückgabewert besitzt. Daher ist im Prinzip jede nummerierte Zeile ein Beginn eines Unterprogrammes und man kann mit GOSUB dorthin springen. Parameter, so

wie man es aus C kennt, kann man nicht an eine Unterprogramm übergeben. Da aber in BASIC alle Variablen global sind kann auf jede Variable in jedem Unterprogramm zugegriffen werden. GOSUB scheint auf den ersten Blick ähnlich zum GOTO Befehl (Kapitel 2.7) zu sein, denn auch mit GOTO kann man Sprünge in eine beliebige nummerierte Programmezeile realisieren. Der große Unterschied ist, dass ein Unterprogramm am Ende ein RETURN aufweisen muss. Das RETURN zeigt das Ende des Unterprogrammes auf und bei erreichen des RETURN Statements wird ein Rücksprung zur Programmzeile des GOSUB Aufrufes bewirkt und der Programmablauf geht in der folgenden Zeile weiter. Realisiert wird dies durch Speichern der Rücksprungadresse auf den Stack. So ist auch Rekursionen möglich, jedoch beträgt der Stack-Speicher nur eine begrenzte Größe. Daher sind Rekursionen auf dem C64 auf 26 Rekursionsaufrufe limitiert [1].

Code-Beispiel 7:

```
10 PRINT "START",
20 GOSUB 100
30 GOSUB 200
40 PRINT "ENDE"
50 END
100 PRINT "UNTERPROGRAMM 1",
110 RETURN
200 PRINT "UNTERPROGRAMM 2",
210 RETURN
```

Ausgabe (Beispiel 7):

```
START           UNTERPROGRAMM 1         UNTERPROGRAMM 2         ENDE
```

Der Programmablauf zu Beispiel 7 sieht folgendermaßen aus: Zeile 10 bewirkt eine Ausgabe (PRINT), Zeile 20 bewirkt einen Sprung, folglich ist unser Programm dann in Zeile 100. Zeile 110 (RETURN) bewirkt einen Rücksprung nach Zeile 20 und das Programm geht in der folgenden Zeile, also Zeile 30, weiter. Zeile 30 bewirkt einen Sprung in das Unterprogramm in Zeile 200. Zeile 210 bewirkt erneut einen Rücksprung in Zeile 30, was zur Fortsetzung in Zeile 40 führt. Anschließend wird die Zeile 50 (END) erreicht und das Programm terminiert.

2.10 Sonstige Funktionen

Neben der bereits behandelten Funktionalität von BASIC gibt es auch noch andere hilfreiche Funktionen von denen die wichtigsten hier kurz erwähnt werden:

- **Mathematische Funktionen:** Es gibt bereits einige vordefinierte Funktionen, wie zum Beispiel die trigonometrischen Funktionen SIN(X), COS(X) und TAN(X), Zufallszahlengenerierung mit RND(X) oder Berechnung der Quadratwurzel mit SQR(X). Eigene einzeilige mathematische Funktionen können mit *DEF FN* definiert und später im Programmcode aufgerufen werden.
- **String Funktionen:** BASIC bringt die wichtigsten Stringverarbeitungsfunktionen mit, wie zum Beispiel das Ermitteln der Länge eines Strings

durch LEN(X$) oder das Ermitteln von Teilstrings mit LEFT(X$,SP%), RIGHT(X$,SP%) und MID(X$, SP%, EP%).

- **Datei Manipulation:** BASIC stellt mit OPEN, GET#, INTPUT#, PRINT# und CLOSE eine API zur Verfügung, um Dateien zu lesen bzw. zu schreiben.

- **Direkter Speicherzugriff:** Mit POKE kann man einen Wert im Wertebereich zwischen 0 und 255, entspricht also einem Byte, an eine bestimmte Speicheradresse des C64 schreiben und somit den C64 direkt auf Hardware-Ebene beeinflussen. Zum Beispiel kann man mit dem Befehl *POKE 53281, 7* die Speicherzelle 53281, welche für die Hintergrundfarbe der Konsole verantwortlich ist, auf 7 setzen, wobei 7 der Farbe Gelb entspricht. Das COMMODORE 64 Bedienungshandbuch [2] bietet eine übersichtliche Tabelle mit Erklärung des gesamten Adressraums. PEEK ist das Gegenstück zu POKE. Mit PEEK kann man den Wert einer bestimmten Speicherzelle auslesen.

3 BASIC im Vergleich

3.1 BASIC ist etwas für Anfänger

BASIC wurde mit der guten Absicht entwickelt, gerade Nichtprogrammieren den Einstieg in die Welt der Programmierer zu erleichtern und die Möglichkeit zu bieten Probleme anhand einer universellen Programmiersprache zu lösen. Dies kann man als geglückt bezeichnen. Es gibt wahrlich schwerer zu erlernende Programmiersprachen, wie Algol, C++ oder gar Assembler, wobei man Assembler nicht als klassische Programmiersprache bezeichnen kann. Die BASIC Syntax ist nah an die englische Sprache angelehnt. Dies soll eine leicht leserliche Syntax ermöglichen. Jedoch muss man sagen, dass Syntax reine Geschmacksache ist, ob man beispielsweise AND (BASIC) oder && (C, C++ oder Java) bevorzugt. Auch wurde darauf geachtet, den Programmieranfänger nicht zu überfordern. So wurde auf Zeiger und Speicherverwaltung durch die Programmiersprache, wie zum Beispiel in C oder C++ üblich, verzichtet. Klare Fehlermeldungen inklusive der konkreten Zeilennummer gehören zum Standardrepertoir eines BASIC Interpreters und helfen bei einer schnellen Fehlerquellenermittlung. Auch hierbei wurde auf den Programmieranfänger geachtet. Laufzeitfehler wie das legendäre "segfault" können in C bzw. C++ schon mal vorkommen, in BASIC nicht.

3.2 BASIC soll eine universelle Sprache sein

BASIC soll, wenn es nach den Erfindern geht, eine universelle für alle Anwendungsgebiete einsetzbare Programmiersprache sein. Ein schöner Vorsatz, aber nur schwer realisierbar. BASIC wurde 1964 erfunden und vielleicht galt damals BASIC als universell. Heute muss man sagen, dass BASIC natürlich grundlegende Konzepte einer modernen Programmiersprache Fehlen, wie etwa die Objektorientierung, nebenläufige Threads bzw. Prozesse, Sicherheitskonzepte (z.B. Java Policy), Exception Handling und Netzwerkbibliotheken. Ein großes Manko in BASIC ist die Möglichkeit, eigene Bibliotheken oder Bibliotheken von Drittanbietern einzubinden, um so den Funktionsumfang zu erweitern. BASIC wurde als imperative Sprache entwickelt und hat alleine deshalb schon einen anderen Anwendungsbereich wie eine funktionale Programmiersprache, wie Beispielsweise LISP, Haskell [6] oder Caml[7].

Es fehlt schlicht die gut durchdachte Strukturierung bzw. Syntax, wie man sie Beispielsweise in Pascal vorfindet. Softwarearchitekur und Designpatterns sind nicht möglich. Verzweigungen und unübersichtliche Sprünge mit GOTO bzw. GOSUB machen ein BASIC Programm zu einer wahren "Buchhaltungsaufgabe" für jeden BASIC Programmier, insbesondere auch, da es keine lokalen Variablen gibt. Nichtsdestotrotz ist BASIC natürlich Turing vollständig.

Der niederländische Informatiker Edsger Dijkstra sagte über BASIC: "It is

[6] `http://www.haskell.org`
[7] `http://caml.inria.fr`

practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration" [6].

BASIC hat sich im Laufe der Jahre gegen die aufkommende Konkurrenz und Vielzahl an verschiedenen Programmiersprachen nicht durchsetzten können. Dies liegt wohl auch daran, dass BASIC nie einheitlich von einer Institution standardiesiert und weiter entwickelt worden ist, wie es Beispielsweise bei Java oder C++ der Fall ist. So enstanden einige BASIC Dialekte, die es zu mehr Ruhm als andere schafften, so z.B. Visual Basic, über welches der britische Informatiker Sir Charles Antony Richard Hoare folgende interessante und zugleich wahre Feststellung machte:

"Recently, I took a look at the very popular programming language called Visual Basic. Well, it's quite nice, but - forget about the BASIC. It is Pascal" [6].

In der Tat hat Visual Basic mit dem Ur-BASIC nicht mehr viel gemeinsam, im Gegensatz zu unseren Commodore Basic V2. BASIC bleibt primär eine Sprache für Anfänger und es lassen sich auch kleinere Probleme gut lösen. Mit Strings kann BASIC recht gut umgehen und bietet einige nützliche String-Manipulationsfunktionen, was für die damalige Zeit nicht selbstverständlich war. Auch eine simple API für Dateizugriffe ist vorgesehen.

3.3 Plattformunabhängigkeit mit BASIC

Die Entwickler von BASIC hatten die Vision von der Systemarchitektur- und Plattformunabhängigkeit. Dies sollte durch den Entwurf als Interpreter-Sprache realisiert werden. Auf jeden Computer sollte ein BASIC Interpreter zur Verfügung stehen und jedes BASIC Programm auf jeden BASIC Interpreter laufen. Jedoch blieb es nur bei einem Traum, denn die Entscheidung BASIC als Interpreter-Sprache zu entwerfen brachte auch Nachteile mit sich. So wurden BASIC-Programme von einem Interpreter zur Laufzeit interpretiert. Dadurch war die Programmausführung langsamer als ein anderes nicht BASIC-Programm, welches kompiliert und für eine spezielle Systemarchitektur optimiert wurde.

So war zum Beispiel effektive Spieleprogrammierung auf den C64 nur mit direkten Speicherzugriff möglich. Dies jedoch machte den BASIC Code nicht portierbar und konnte dementsprechend nicht mehr auf anderen Systemarchitekturen ausgeführt werden.

Literatur

[1] Prigmore Clive. *Top-Training BASIC*. Verlag Heinz Heise GmbH, 1985.

[2] Commodore. *COMMODORE 64 Bedienungshandbuch*. Commodore GmbH, Lyoner Straße 38, 6000 Frankfurt/M, Deutschland, 1982.

[3] Elsing J., Sterner H., and Wagner A. *BASIC auf den Commodore 64*. IWT, 1983.

[4] Wittig Siegmar. *BASIC-Brevier für den Commodore 64*. Klett, 1986.

[5] C64 Wiki. Website, 2010.
`http://www.c64-wiki.de/index.php/C64-Befehle`
`besucht am 25. Novemeber 2010`.

[6] Deutsche Software Entwickler Wiki. Website, 2010.
`http://www.wikiservice.at/dse/wiki.cgi?SpracheBasic`
`besucht am 5. Dezember 2010`.

[7] Wikipedia. `BASIC`. Website, 2010.
`http://de.wikipedia.org/wiki/BASIC`
`besucht am 25. Novemeber 2010`.

[8] Wikipedia. `Geschichte der Programmiersprachen`. Website, 2010.
`http://de.wikipedia.org/wiki/Geschichte_der_Programmiersprachen`
`besucht am 28. Novemeber 2010`.

[9] Wikipedia. `Microsoft Basic`. Website, 2010.
`http://de.wikipedia.org/wiki/Microsoft_BASIC`
`besucht am 28. Novemeber 2010`.