1.3 Programmstrukturen/Variablen und Datentypen

1.3.1 Wiederverwendung von Zwischenergebnissen – geht das?

Eine typische Aufgabe in der Elektrotechnik kann so aussehen: Ein Widerstand mit $R = 529\,\Omega$ liegt an einer Spannung von $U = 230\,\mathrm{V}$. Gesucht sind die Stromstärke I und die im Widerstand umgesetzte Leistung P. Die Berechnung läuft in zwei Schritten:

a) Berechnung der Stromstärke: $I = \frac{U}{R}$

b) Berechnung der Leistung: $P = U \cdot I$

Man erhält im ersten Schritt die Stromstärke, und man könnte sie im zweiten Schritt wiederverwenden — vorausgesetzt, es gäbe eine Möglichkeit, sie zwischenzuspeichern.

Dazu bietet sich das RAM an: Dort können ziemlich viele Daten kurzzeitig gespeichert werden. Von der Software-Seite bieten die meisten Programmiersprachen Hilfe an, Daten im RAM zu speichern. In SPS-Sprachen wird so etwas als *Merker* bezeichnet, in C spricht man von *Variablen*.

1.3.2 Wiederverwendung mit Hilfe einer Variablen

Das folgende Beispiel zeigt, wie man mit Variablen umgeht:

src/uripui1.c

```
1  #include <stdio.h>
2  int main(void)
3  {
4                          /* Vereinbarungsteil */
5      double i;
6                          /* Anweisungsteil */
7      i = 230.0/529.0;
8
9      printf("I=%g A\n", i);
10     printf("P=%g W\n", 230.0*i);
11     return 0;
12 }
```

Zeile 5 Hier wird eine *Variable* vereinbart. Eine Variable ist ein Speicherplatz (meist im RAM), an den man eine Information schreiben und von dem man dieselbe Information später wieder lesen kann.

Zur leichteren Benutzung bekommt dieser Speicherplatz einen *Namen*. Man muss sich also nicht merken, wo (=an welcher Adresse im RAM) man etwas gespeichert hat, sondern nur, wie der Speicherplatz hieß.

In diesem Fall hat die Variable (=der Speicherplatz) den Namen `i`. Der Name ist frei wählbar, er muss nur zusammenhängend sein und darf aus einem oder mehreren Buchstaben, Ziffern und Unterstrichen bestehen. Er darf nur nicht mit einer Ziffer beginnen, damit man ihn nicht für eine Zahl hält.

Der Speicherplatz hat also erstens den Namen `i` und zweitens eine Eigenschaft, die *Datentyp* heißt. In diesem Fall ist der Datentyp `double`. Der Datentyp gibt an, was die Variable aufnehmen soll, welchen Wertebereich sie aufnehmen kann und welche Operationen mit unserer Variablen `i` erlaubt sind. Und er bestimmt auch, wie viele Bytes der Speicherplatz der Variablen umfasst.

In diesem Fall ist der Datentyp also `double`: Die Variable soll Gleitkommazahlen aufnehmen, der Wertebereich umfasst viele Zahlen von etwa $-1,8 \cdot 10^{308}$ bis $1,8 \cdot 10^{308}$, und eine Variable dieses Typs benötigt genau 64 Bit an Speicherplatz.

Nun, da die Variable vereinbart wurde, darf sie auch benutzt werden.

Zeile 7 Hier finden wir die erste Anweisung, den ersten Befehl.

Es handelt sich um eine *Zuweisung*. Die Variable `i` bekommt den Wert des Ausdrucks `230.0/529.0` zugewiesen.

Das Gleichheitszeichen bedeutet hier – anders als in der Mathematik – keine Feststellung, sondern eine Aktion: Die linke Seite erhält den Wert, der rechts vom Gleichheitszeichen (Zuweisungsoperator) ausgerechnet wurde.

Man kann eine Variable also einfach benutzen, indem man ihren Namen schreibt.

Zeile 9 Erst jetzt, nachdem in Zeile 7 der Wert für `i` berechnet wurde, lohnt es sich, mit der Ausgabe zu beginnen.

In dieser Zeile wird `i` ausgegeben. `printf` erlaubt es also auch, Variablen auszugeben. In der Regel kann man an allen Stellen, an denen Konstanten erlaubt sind, auch Variablen verwenden (es gibt Ausnahmen).

Zeile 10 In dieser Zeile wird die Leistung berechnet und ausgegeben.

Die Vereinbarung einer Variablen hat also immer die Form `datentyp name;`. Man kann übrigens mehrere Variablen desselben Datentyps in einer Vereinbarung unterbringen:

```
1     double strom, leistung, widerstand;
2     int anzahl_leds, anzahl_taster;
```

In diesem Beispiel wurden drei Variablen vom Datentyp `double` vereinbart und danach zwei Variablen vom Typ `int`.

1.3.3 Noch mehr Variablen und das EVA-Prinzip

Das folgende Programm löst dasselbe Problem wie oben, benutzt aber für jede physikalische Größe eine eigene Variable:

src/uripui2.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      /* Vereinbarungsteil, Reihenfolge egal */
6      double u;
7      double i;
8      double p;
9      double r;
10
11     /*************************************/
12     /* Anweisungsteil, Reihenfolge wichtig */
13
14     u = 230.0;              /* Eingabe */
15     r = 529.0;
16     i = u/r;                /* Verarbeitung */
17     p = u*i;
18     printf("I=%g␣A\n", i); /* Ausgabe */
19     printf("P=%g␣W\n", p);
20     return 0;
21 }
```

Zeile 5 Zu Beginn eines jeden Blockes (direkt nach der geschweiften Klammer auf) dürfen (auch mehrere) Variablenvereinbarungen stehen, das ist der so genannte *Vereinbarungsteil*. In welcher Reihenfolge die Variablen vereinbart werden, ist egal. Viele bevorzugen eine alphabetische Reihenfolge.

Zeilen 6–9 Die Variablen `u`, `r`, `i` und `p` werden vereinbart.

Zeile 12 Nach der letzten Vereinbarung folgt mit der ersten Anweisung der *Anweisungsteil*[1].

Im Anweisungsteil ist die Reihenfolge der Anweisungen wichtig. Hier hat sich die Reihenfolge Eingabe – Verarbeitung – Ausgabe (abgekürzt EVA) bewährt.

Zeilen 14–15 `u` und `r` bekommen je einen Wert zugewiesen (Eingabe). Ab jetzt kann dieser Wert benutzt werden.

Zeile 16 `i` bekommt den Wert zugewiesen, der aus dem Ausdruck `u/r` berechnet wird (Verarbeitung 1).

Zeile 17 `p` bekommt den Wert zugewiesen, der aus dem Ausdruck `u*r` berechnet wird (Verarbeitung 2). Hier ist es wichtig, dass vorher `i` berechnet wurde!

Zeilen 18–19 Nun können die berechneten Größen ausgegeben werden (Ausgabe).

1.3.4 Datentypen, Konstanten und Platzhalter bei printf

In der Tabelle 1 sind in der linken Spalte die elementaren Datentypen in C aufgeführt. In den weiteren Spalten sieht man Beispiele für Konstanten und Platzhalter bei printf, dazu später. Es

Datentyp	Bit	Konstante (Bsp.)	Platzhalter bei printf
Zweierkomplement			
`signed char`	8	`123`	`%hhi, %hhd`
`short`	A	`5200`	`%hi, %hd`
`int`	B	`5200`	`%i, %d`
`long`	C	`5200L`	`%li, %ld`
`long long`	D	`5200LL`	`%lli, %lld`
Vorzeichenlos			
`unsigned char`	8	`123U`	`%hhu, %hho, %hhx`
`unsigned short`	A	`5200U`	`%hu, %ho, %hx`
`unsigned int`	B	`5200U`	`%u, %o, %x`
`unsigned long`	C	`5200UL`	`%lu, %lo, %lx`
`unsigned long long`	D	`5200ULL`	`%llu, %llo, %llx`
Gleitkommazahlen			
`float`	32	`5200.0F, 1E-12F`	`%f`
`double`	64	`5200.0, 1E-12`	`%lf (oder %f)`
`long double`	80 bis 128	`5200.0L, 1E-12L`	`%Lf`
Zweierkomplement oder vorzeichenlos			
`char`	8	`123`	`%c`

Tabelle 1: Elementare Datentypen in C und ihre Platzhalter bei printf

gibt zwei Arten elementarer Datentypen: Ganzzahl-Datentypen und Gleitkomma-Datentypen.

[1] Seit C99 dürfen Vereinbarungen auch irgendwo zwischen Anweisungen stehen. Diese Praxis ist Anfängern jedoch nicht zu empfehlen.

1.3.4.1 Ganzzahl-Datentypen Bei den Ganzzahl-Datentypen gibt es einen Vorzugstyp, den Datentyp `int`. Er belegt in der Regel so viele Bits, wie der Datenbus des Systems breit ist. Momentan sind das bei PCs 32 oder 64 Bit. Auf einem Mikrocontroller-System kann eine `int`-Variable aber auch 16 oder 8 Bit breit sein. Eine `int`-Variable kann positive, negative Wert und null aufnehmen. Meistens wird dafür das Zweierkomplement benutzt.

Manchmal braucht man Variablen, die weniger Platz belegen. Dafür gibt es den Datentyp `short int` (abgekürzt `short`). Er kann z. B. auf einem PC 16 Bit belegen.

Häufiger ist es so, dass man Variablen braucht, die einen größeren Wertebereich umfassen. Die belegen dann natürlich mehr Speicherplatz. Dafür gibt es den Datentypen `long int` (abgekürzt `long`), der auf PCs 64 Bit umfasst. Zusätzlich gibt es seit C99 den Datentypen `long long int` (abgekürzt `long long`).

Falls man in der Systemprogrammierung direkt mit einzelnen Bits arbeitet, ist das Zweierkomplement der bisher vorgestellten Datentypen umständlich. Für diesen Zweck bietet C alle diese Datentypen in einer vorzeichenlosen Version an. Sie heißen dann:

- `unsigned short int` (abgekürzt `unsigned short`)
- `unsigned int` (abgekürzt `unsigned`)
- `unsigned long int` (abgekürzt `unsigned long`)
- `unsigned long long int` (abgekürzt `unsigned long long`), verfügbar seit dem Standard C99

Die vorzeichenlosen Versionen benutzen den natürlichen Binärcode und erleichtern dadurch das Hantieren mit Bits.

Wie schon angedeutet, ist im C-Standard der Speicherbedarf einer Variablen eines bestimmten Datentyps nicht festgelegt. Damit wächst der C-Standard mit der Hardware-Entwicklung mit. Festgelegt wurde nur:

a) `int` ist mindestens so lang wie `short int` ($B \geq A$ in Spalte 2 von Tabelle 1)

b) `long int` ist mindestens so lang wie `int` ($C \geq B$)

c) `long long int` ist mindestens so lang wie `long int` ($D \geq C$)

d) `long int` ist länger als `short int` ($C > A$)

e) Jeder `unsigned`-Datentyp ist genauso lang wie der entsprechende `int`-Datentyp.

1.3.4.2 Ganzzahl-Datentypen mit 8 Bit Speicherbedarf Ein Sonderfall ist der Datentyp `char`. Er hat 8 Bit Breite und wird daher gerne für den Umgang mit alphanumerischen Daten (Buchstaben, Ziffern und Satzzeichen) verwendet.

Das Besondere bei `char`: Je nach Realisierung (*Implementation*) ist er vorzeichenlos oder nicht. Beide Realsierungen sind nach Standard erlaubt.

Falls man für den direkten Umgang mit Bytes einen 8-Bit-Datentyp braucht, der mit Sicherheit kein Vorzeichen hat, verwendet man den Datentyp `unsigned char`. Auf der anderen Seite gibt es noch den Datentyp `signed char`, der positive und negative Zahlen umfasst.

1.3.4.3 Gleitkomma-Datentypen Bei den Gleitkomma-Datentypen ist der Vorzugstyp `double`. Er entspricht bei PCs in der Regel dem IEEE-754-Datentyp *double precision* und hat einen Speicherbedarf von 64 Bit.

Für Sonderfälle gibt es außerdem noch den Datentyp `float` (bei IEEE-754 *single precision* genannt) mit 32 Bit und den Datentyp `long double` (bei bei IEEE-754 *extended precision* genannt) mit 80 Bit oder 96 Bit (je nach Systemarchitektur).

1.3.4.4 Zahlen-Konstanten Den Typ einer literalen (=wörtlichen) Zahlen-Konstante **in einem Quelltext** ermittelt der Compiler je nach Schreibweise der Zahl. So ist 27 eine Ganzzahl und 3.5 eine Gleitkommazahl. Falls der Wert der Zahlenkonstante nicht in den Datentyp hineipasst, wird (nach bestimmten Regeln) der nächstgrößere Datentyp genommen. So ist 27 vom Typ `int` und 12345678901 möglicherweise vom Typ `long`.

Am wichtigsten ist dabei die Unterscheidung zwischen Gleitkomma- und Ganzzahlkonstanten:

a) Eine Zahlenkonstante, die mit der Zeichenkette `0x` beginnt, wird als Ganzzahl erkannt und hexadezimal interpretiert.

b) Eine Zahlenkonstante, die mit dem Zeichen `0` beginnt und danach nur Ziffern von 0 bis 7 hat, wird als Ganzzahl erkannt und oktal interpretiert.

c) Eine Zahlenkonstante, die nicht mit `0` beginnt und nur Ziffern enthält, wird als Ganzzahl erkannt und dezimal interpretiert.

d) Eine Zahlenkonstante, die außer Dezimalziffern einen Punkt oder ein `e` (oder `E`) enthält (aber nicht mit `0x` beginnt), wird als Gleitkommazahl erkannt.

Manchmal ist es wichtig, einer Zahlenkonstante ausdrücklich einen bestimmten Datentyp zuzuschreiben. Dazu gelten folgende Regeln:

a) Für ganze Zahlen (egal, ob dezimal, oktal, hexadezimal) gilt:

   1) Standardmäßig ist der Datentyp `int`.
   2) Mit einem angehängten `U` ist der Datentyp `unsigned int`.
   3) Mit einem angehängten `L` ist der Datentyp `long int`.
   4) Mit dem Anhang `UL` ist der Datentyp `unsigned long int`.
   5) Mit dem Anhang `LL` ist der Datentyp `long long int`.
   6) Mit dem Anhang `LL` ist der Datentyp `unsigned long long int`.
   7) Für Datentypen, die kürzer als `int` sind, gibt es keine besondere Konstanten-Markierung

b) Für Gleitkommazahlen gilt:

   1) Standardmäßig ist der Datentyp `double`.
   2) Mit einem angehängten `F` ist der Datentyp `float` .
   3) Mit einem angehängten `L` ist der Datentyp `long double`.

Spalte 3 von Tabelle 1 gibt zu jedem Datentyp mindestens ein Beispiel einer passenden Konstante.

1.3.4.5 Platzhalter bei printf Es ist sinnvoll, wenn man den Inhalt einer Variablen mit `printf` ausgeben kann, egal welchen Datentyp sie hat. Deshalb sollte es zu jedem Datentyp einen Platzhalter bei `printf` geben. In Spalte 4 der Tabelle 1 sind diese Platzhalter für jeden Datentyp zu sehen.

Bei Gleitkomma-Datentypen gibt es zwei Besonderheiten:

- Vor C99 war `%f` der Platzhalter für `double`. Seit C99 darf man für `double` auch `%lf` verwenden. Das ist einfacher, weil die Eingabefunktion `scanf` das auch tut. Und logischer, weil `double` gewissermaßen so etwas wie `long float` ist.
- Der Platzhalter für `long double` ist nicht `%llf`, sonder `%Lf` (warum auch immer).

Der Platzhalter `%c` für den Datentyp `char` gibt den Inhalt der Variablen nicht als Zahl, sondern als Zeichen aus.

1.3.5 Arithmetik und Grenzwerte

Damit es bei einer Rechenoperation nicht zu einem Überlauf kommt, muss der Programmierer vorher wissen, wie groß der Zahlenbereich des jeweiligen Datentyps ist. Diese Information ist in den Konstanten gespeichert, die nach Einbindung von `<limits.h>` verfügbar sind (`#include <limits.h>`). Tabelle 2 gibt einen Überblick. Ebenso gibt es Grenzwerte auch für

Datentyp	Minimum	Maximum
`char`	`CHAR_MIN`	`CHAR_MAX`
`signed char`	`SCHAR_MIN`	`SCHAR_MAX`
`short`	`SHRT_MIN`	`SHRT_MAX`
`int`	`INT_MIN`	`INT_MAX`
`long`	`LONG_MIN`	`LONG_MAX`
`long long`	`LLONG_MIN`	`LLONG_MAX`
`unsigned char`	0	`UCHAR_MAX`
`unsigned short`	0	`USHRT_MAX`
`unsigned int`	0	`UINT_MAX`
`unsigned long`	0	`ULONG_MAX`
`unsigned long long`	0	`ULLONG_MAX`

Tabelle 2: Grenzwerte der einzelnen Ganzzahl-Datentypen

Gleitkommazahlen. Diese Grenzwerte kann man nutzen durch Einbindung von `<float.h>`. Tabelle 3 zeigt einen kleinen Auszug.

Datentyp	kleinster normalis. Betrag	größter Betrag
`float`	`FLT_MIN`	`FLT_MAX`
`double`	`DBL_MIN`	`DBL_MAX`
`long double`	`LDBL_MIN`	`LDBL_MAX`

Tabelle 3: Grenzwerte der einzelnen Gleitkomma-Datentypen

1.3.6 Initialisierung einer Variablen mit einer Konstanten

In C ist es so, dass eine Variable, nachdem sie angelegt wurde, einen undefinierten Inhalt hat[2]: Wenn das Programm läuft und der Speicherplatz für die Variable reserviert wird, wird er nicht mit irgendeinem Standardwert überschrieben. Was vorher an der Stelle im RAM stand, bleibt und ist ab jetzt der Inhalt der Variablen:

src/init0.c

```c
#include <stdio.h>

int main(void)
{
    double a, b, c;
    printf("Zufaelliger Inhalt von a, b, c: %f, %f, %f\n", a, b, c);
    return 0;
}
```

[2]Es gibt allerdings Ausnahmen, z. B. globale Variablen

Terminal
```
schueler@debian964:~$ gcc init0.c
schueler@debian964:~$ a.out
Zufaelliger Inhalt von a, b, c: 0.000000, 0.000000, 0.000000
```

In diesem einfachen Beispiel auf diesem System ist **zufällig** der Inhalt aller drei Variablen null. **Das muss aber nicht so sein!** Gerade bei umfangreichen Programmen mit mehreren Funktionsbausteinen ist der Inhalt höchstwahrscheinlich noch von vorherigen Geschehnissen beschrieben.

In C gibt es daher die Möglichkeit, eine Variable direkt bei ihrer Vereinbarung mit einem Startwert zu versehen. Man nennt das *Initialisierung*. Das sieht dann so aus:

src/init1.c
```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double a=1.0, b=2.0, c=3.0;
6     printf("Festgelegter Inhalt von a, b, c: %f, %f, %f\n", a, b, c);
7     return 0;
8 }
```

Terminal
```
schueler@debian964:~$ gcc init1.c
schueler@debian964:~$ a.out
Festgelegter Inhalt von a, b, c: 1.000000, 2.000000, 3.000000
```

Zeile 5 Die Variablen `a`, `b` und `c` werden angelegt mit dem Datentyp `double`. `a` bekommt den Startwert 1, `b` den Startwert 2 und `c` den Startwert 3.

Der Startwert ist hier jeweils eine Konstante.

Für die Berechnung von I und P sähe das dann so aus:

src/uripui3.c
```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      /* Vereinbarungsteil, Reihenfolge egal */
6      double u=230.0;
7      double r=529.0;
8      double i;
9      double p;
10
11     /*************************************/
12     /* Anweisungsteil, Reihenfolge wichtig */
13
14     i = u/r;                /* Verarbeitung */
15     p = u*i;
16     printf("U=%g V\n", u); /* Ausgabe */
17     printf("R=%g Ohm\n", r);
18     printf("I=%g A\n", i); /* Ausgabe */
19     printf("P=%g W\n", p);
20     return 0;
21 }
```

Ob man jeder Variablen durch Initialisierung einen Startwert gibt oder nicht, ist Geschmackssache. Der Vorteil liegt darin, dass man auf diese Art keinen zufälligen Inhalt erhält, falls man eine Zuweisung vergisst. Der Nachteil liegt darin, dass die Initialisierung und die Benutzung der Variablen im Quelltext weit auseinander liegen können.

1.3.7 Initialisierung einer Variablen allgemein

Der Startwert darf aber auch der Inhalt einer anderen Variablen oder das Ergebnis eines mathematischen Ausdrucks sein. Das kann dann so aussehen:

src/uripui4.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      /* Vereinbarungsteil, Reihenfolge jetzt wichtig */
6      double u=230.0;
7      double r=529.0;
8      double i=u/r;
9      double p=u*r;
10
11     /*************************************/
12     /* Anweisungsteil                    */
13
14     printf("I=%g A\n", i); /* Ausgabe */
15     printf("P=%g W\n", p);
16     return 0;
17 }
```

Zeile 8 Die Variable `i` wird angelegt mit dem Datentyp `double`. Sie bekommt als Startwert das Ergebnis des Ausdrucks `u/r`.

Zeile 9 Die Variable `p` wird angelegt mit dem Datentyp `double`. Sie bekommt als Startwert das Ergebnis des Ausdrucks `u*i`.

Vereinbarungs- und Anweisungsteil sind hier zwar noch immer voneinander getrennt, aber die Berechnung wird schon ausgeführt, bevor der Anweisungsteil beginnt. Und jetzt kommt es auch auf die Reihenfolge der Vereinbarungen und Initialisierungen an. Empfehlung: So etwas sollte man *sehr* sparsam verwenden.

1.3.8 Datentypen mit fester Speichergröße

Der Vorteil von Datentypen wie `int` ist es, dass sie beim Portieren von alten auf neue Systeme „mitwachsen“. In manchen Fällen braucht man jedoch Datentypen, die eine genau festgelegte Größe haben. Deshalb gibt es seit C99 eben solche Datentypen (Tabelle 4). Wenn man sie benutzen will, muss man `<stdint.h>` einbinden. Für die Ausgabe mit `printf` benötigt man darüberhinaus auch `<inttypes.h>`. Hier ein Beispiel:

src/datfest.c

```
1  #include <stdio.h>     /* fuer printf() */
2  #include <stdint.h>    /* fuer uint64_t */
3  #include <inttypes.h> /* fuer PRIu64   */
4  int main(void)
5  {
```

Datentyp	Breite	Platzhalter	Grenzwerte
Zweierkomplement			
`int8_t`	1	`PRIi8`	`INT8_MIN`, `INT8_MAX`
`int16_t`	2	`PRIi16`	`INT16_MIN`, `INT16_MAX`
`int32_t`	4	`PRIi32`	`INT32_MIN`, `INT32_MAX`
`int64_t`	8	`PRIi64`	`INT64_MIN`, `INT64_MAX`
Vorzeichenlos			
`uint8_t`	1	`PRIu8`	0, `UINT8_MAX`
`uint16_t`	2	`PRIu16`	0, `UINT16_MAX`
`uint32_t`	4	`PRIu32`	0, `UINT32_MAX`
`uint64_t`	8	`PRIu64`	0, `UINT64_MAX`

Tabelle 4: Datentypen mit fester Länge

```
6     uint64_t x;
7     x=123456789012345;
8     printf("x=%" PRIu64 "\n", x);
9     return 0;
10 }
```

1.3.9 Vereinbarungen und Zuweisungen im Struktogramm und im Flussdiagramm

Die Vereinbarung einer Variablen taucht im Struktogramm und im Flussdiagramm nicht auf[3].

Bei der Zuweisung ist sowohl beim Struktogramm als auch beim Flussdiagramm darauf zu achten, dass nicht das Gleichheitszeichen verwendet wird, sondern ein Pfeil. Das Gleichheitszeichen symbolisiert nämlich nur in C (und vielen verwandten Sprachen) die Zuweisung. In mancher anderen Sprache wird stattdessen das (modernere) Symbol `:=` verwendet (also `x:=7;` anstelle von `x=7;`). Struktogramm und Flussdiagramm sollen aber *unabhängig* von der benutzten Programmiersprache gelesen werden können. Abbildung 1 zeigt das Struktogramm zu `uripui2.c`. Abbildung 2 zeigt das entsprechende Flussdiagramm.

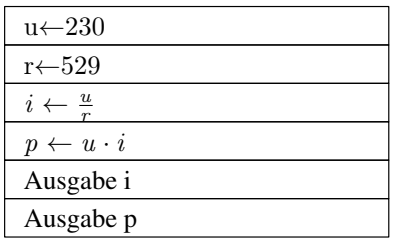

Abbildung 1: Beispiel für ein Struktogramm mit Zuweisung

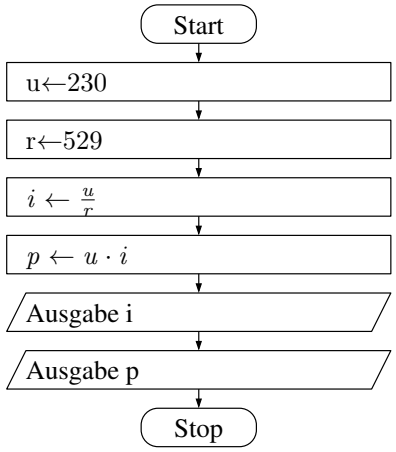

Abbildung 2: Beispiel für ein Flussdiagramm mit Zuweisung

[3]Das ist eine Lücke bei beiden Diagramm-Typen, dass sie sich so sehr auf den Programmablauf konzentrieren und die Daten ignorieren. Bei Flussdiagrammen gibt es allerdings ein eigenes Datenflussdiagramm, das aber vom Programmablauf getrennt ist.