

2 Eine einfache Programmiersprache

Eine Programmiersprache soll

- Datenstrukturen anbieten;
- Operationen auf Daten erlauben;
- **Kontrollstrukturen** zur Ablaufsteuerung bereit stellen.

Als Beispiel betrachten wir **MiniJava**.

2.1 Variablen

Um Daten zu speichern und auf gespeicherte Daten zugreifen zu können, stellt **MiniJava Variablen** zur Verfügung. Variablen müssen erst einmal eingeführt, d.h. **deklariert** werden.

Beispiel:

```
int x, result;
```

Diese Deklaration führt die beiden Variablen mit den **Namen** x und result ein.

Erklärung:

- Das Schlüsselwort `int` besagt, dass diese Variablen ganze Zahlen (“Integers”) speichern sollen.
`int` heißt auch **Typ** der Variablen `x` und `result`.
- Variablen können dann benutzt werden, um anzugeben, auf welche Daten Operationen angewendet werden sollen.
- Die Variablen in der Aufzählung sind durch Kommas “,” getrennt.
- Am Ende steht ein Semikolon “;”.

2.2 Operationen

Die Operationen sollen es gestatten, die Werte von Variablen zu modifizieren. Die wichtigste Operation ist die [Zuweisung](#).

Beispiele:

- `x = 7;`
Die Variable `x` erhält den Wert 7.
- `result = x;`
Der Wert der Variablen `x` wird ermittelt und der Variablen `result` zugewiesen.
- `result = x + 19;`
Der Wert der Variablen `x` wird ermittelt, 19 dazu gezählt und dann das Ergebnis der Variablen `result` zugewiesen.

- `result = x - 5;`

Der Wert der Variablen `x` wird ermittelt, 5 abgezogen und dann das Ergebnis der Variablen `result` zugewiesen.

Achtung:

- **Java** bezeichnet die Zuweisung mit "=" anstelle von "==" (Erbschaft von **C** ... :-)
- Jede Zuweisung wird mit einem Semikolon ";" beendet.
- In der Zuweisung `x = x + 1;` greift das `x` auf der rechten Seite auf den Wert **vor** der Zuweisung zu.

Weiterhin benötigen wir Operationen, um Daten (Zahlen) einlesen bzw. ausgeben zu können.

- `x = read();`
Diese Operation liest eine Folge von Zeichen vom Terminal ein und interpretiert sie als eine ganze Zahl, deren Wert sie der Variablen `x` als Wert zu weist.
- `write(42);`
Diese Operation schreibt 42 auf die Ausgabe.
- `write(result);`
Diese Operation bestimmt den Wert der Variablen `result` und schreibt dann diesen auf die Ausgabe.
- `write(x-14);`
Diese Operation bestimmt den Wert der Variablen `x`, subtrahiert 14 und schreibt das Ergebnis auf die Ausgabe.

Achtung:

- Das Argument der `write`-Operation in den Beispielen ist ein `int`.
- Um es ausgeben zu können, muss es in eine **Folge von Zeichen** umgewandelt werden, d.h. einen `String`.

Damit wir auch freundliche Worte ausgeben können, gestatten wir auch **direkt** `Strings` als Argumente:

- `write("Hello World!");`
... schreibt `Hello World!` auf die Ausgabe.

2.3 Kontrollstrukturen

Sequenz:

```
int x, y, result;  
x = read();  
y = read();  
result = x + y;  
write(result);
```


- Zu jedem Zeitpunkt wird nur eine Operation ausgeführt.
- Jede Operation wird genau einmal ausgeführt. Keine wird wiederholt, keine ausgelassen.
- Die Reihenfolge, in der die Operationen ausgeführt werden, ist die gleiche, in der sie im Programm stehen (d.h. nacheinander).
- Mit Beendigung der letzten Operation endet die Programm-Ausführung.

⇒ Sequenz alleine erlaubt nur sehr einfache Programme.

Selektion (bedingte Auswahl):

```
int x, y, result;  
x = read();  
y = read();  
if (x > y)  
    result = x - y;  
else  
    result = y - x;  
write(result);
```

- Zuerst wird die Bedingung ausgewertet.
- Ist sie erfüllt, wird die nächste Operation ausgeführt.
- Ist sie nicht erfüllt, wird die Operation nach dem `else` ausgeführt.

Beachte:

- Statt aus einzelnen Operationen können die Alternativen auch aus Statements bestehen:

```
int x;  
x = read();  
if (x == 0)  
    write(0);  
else if (x < 0)  
    write(-1);  
else  
    write(+1);
```

- ... oder aus (geklammerten) Folgen von Operationen und Statements:

```
int x, y;  
x = read();  
if (x != 0) {  
    y = read();  
    if (x > y)  
        write(x);  
    else  
        write(y);  
} else  
    write(0);
```

- ... eventuell fehlt auch der else-Teil:

```
int x, y;  
x = read();  
if (x != 0) {  
    y = read();  
    if (x > y)  
        write(x);  
    else  
        write(y);  
}
```

Auch mit Sequenz und Selektion kann noch nicht viel berechnet werden ... :-)

Iteration (wiederholte Ausführung):

```
int x, y;  
x = read();  
y = read();  
while (x != y)  
    if (x < y)  
        y = y - x;  
    else  
        x = x - y;  
write(x);
```

- Zuerst wird die Bedingung ausgewertet.
- Ist sie erfüllt, wird der **Rumpf** des `while`-Statements ausgeführt.
- Nach Ausführung des Rumpfs wird das gesamte `while`-Statement erneut ausgeführt.
- Ist die Bedingung nicht erfüllt, fährt die Programm-Ausführung hinter dem `while`-Statement fort.

Jede (partielle) Funktion auf ganzen Zahlen, die überhaupt berechenbar ist, läßt sich mit Selektion, Sequenz, Iteration, d.h. mithilfe eines MiniJava-Programms berechnen :-)

Beweis: ↑ Berechenbarkeitstheorie.

Idee:

Eine Turing-Maschine kann alles berechnen...

Versuche, eine Turing-Maschine zu simulieren!

MiniJava-Programme sind ausführbares Java.

Man muss sie nur geeignet dekorieren :-)

MiniJava-Programme sind ausführbares Java.

Man muss sie nur geeignet dekorieren :-)

Beispiel: Das GGT-Programm

```
int x, y;
x = read();
y = read();
while (x != y)
    if (x < y)
        y = y - x;
    else
        x = x - y;
write(x);
```

Daraus wird das Java-Programm:

```
public class GGT extends MiniJava {
    public static void main (String[] args) {

        int x, y;
        x = read();
        y = read();
        while (x != y)
            if (x < y)
                y = y - x;
            else
                x = x - y;
        write(x);

    } // Ende der Definition von main();
} // Ende der Definition der Klasse GGT;
```

Erläuterungen:

- Jedes Programm hat einen **Namen** (hier: GGT).
- Der Name steht hinter dem Schlüsselwort `class` (was eine Klasse, was `public` ist, lernen wir später ... :-)
- Der Datei-Name muss zum Namen des Programms “passen”, d.h. in diesem Fall `GGT.java` heißen.
- Das **MiniJava**-Programm ist der Rumpf des **Hauptprogramms**, d.h. der Funktion `main()`.
- Die Programm-Ausführung eines **Java**-Programms startet stets mit einem Aufruf von dieser Funktion `main()`.
- Die Operationen `write()` und `read()` werden in der Klasse `MiniJava` definiert.
- Durch `GGT extends MiniJava` machen wir diese Operationen innerhalb des GGT-Programms verfügbar.

Die Klasse MiniJava ist in der Datei MiniJava.java definiert:

```
import javax.swing.JOptionPane;
import javax.swing.JFrame;
public class MiniJava {
    public static int read () {
        JFrame f = new JFrame ();
        String s = JOptionPane.showInputDialog (f, "Eingabe:");
        int x = 0; f.dispose ();
        if (s == null) System.exit (0);
        try { x = Integer.parseInt (s.trim ());
        } catch (NumberFormatException e) { x = read (); }
        return x;
    }
    public static void write (String x) {
        JFrame f = new JFrame ();
        JOptionPane.showMessageDialog (f, x, "Ausgabe",
            JOptionPane.PLAIN_MESSAGE);
        f.dispose ();
    }
    public static void write (int x) { write (""+x); }
}
```

... weitere Erläuterungen:

- Die Klasse MiniJava werden wir im Lauf der Vorlesung im Detail verstehen lernen :-)
- Jedes Programm sollte **Kommentare** enthalten, damit man sich selbst später noch darin zurecht findet!
- Ein Kommentar in **Java** hat etwa die Form:

```
// Das ist ein Kommentar!!!
```

- Wenn er sich über mehrere Zeilen erstrecken soll, kann er auch so aussehen:

```
/* Dieser Kommentar geht  
"über mehrere Zeilen! */
```

Das Programm GGT kann nun übersetzt und dann ausgeführt werden:

```
seidl> javac GGT.java  
seidl> java GGT
```

- Der Compiler `javac` liest das Programm aus den Dateien `GGT.java` und `MiniJava.java` ein und erzeugt für sie JVM-Code, den er in den Dateien `GGT.class` und `MiniJava.class` ablegt.
- Das Laufzeitsystem `java` liest die Dateien `GGT.class` und `MiniJava.class` ein und führt sie aus.

Achtung:

- **MiniJava** ist sehr primitiv.
- Die Programmiersprache **Java** bietet noch eine Fülle von Hilfsmitteln an, die das Programmieren erleichtern sollen. Insbesondere gibt es
- viele weitere Datenstrukturen (nicht nur int) und
- viele weitere Kontrollstrukturen.

... kommt später in der Vorlesung :-)

3 Syntax von Programmiersprachen

Syntax (“Lehre vom Satzbau”):

- formale Beschreibung des Aufbaus der “Worte” und “Sätze”, die zu einer Sprache gehören;
- im Falle einer **Programmier**-Sprache Festlegung, wie Programme aussehen müssen.

Hilfsmittel bei natürlicher Sprache:

- Wörterbücher;
- Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- Ausnahme-Listen;
- Sprach-“Gefühl”.

Hilfsmittel bei Programmiersprachen:

- Listen von **Schlüsselwörtern** wie `if`, `int`, `else`, `while` ...
- Regeln, wie einzelne Worte (**Tokens**) z.B. **Namen** gebildet werden.

Frage:

Ist `x10` ein zulässiger Name für eine Variable?
oder `_ab$` oder `A#B` oder `0A?B` ...

- Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.

Frage:

Ist ein `while`-Statement im `else`-Teil erlaubt?

- Kontextbedingungen.

Beispiel:

Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

⇒ formalisierter als natürliche Sprache

⇒ besser für maschinelle Verarbeitung geeignet

Semantik (“Lehre von der Bedeutung”):

- Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine **Bedeutung**, d.h. teilt einem Hörer/Leser einen Sachverhalt mit (↑**Information**)
- Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine **Bedeutung** :-)

Die Bedeutung eines Programms ist

- alle möglichen **Ausführungen** der beschriebenen Berechnung (↑ **operationelle Semantik**); oder
- die definierte **Abbildung** der Eingaben auf die Ausgaben (↑ **denotationelle Semantik**).

Die Bedeutung eines Programms ist

- alle möglichen **Ausführungen** der beschriebenen Berechnung (↑ **operationelle Semantik**); oder
- die definierte **Abbildung** der Eingaben auf die Ausgaben (↑ **denotationelle Semantik**).

Achtung!

Ist ein Programm **syntaktisch korrekt**, heißt das noch lange nicht, dass es auch das “richtige” tut, d.h. **semantisch korrekt** ist !!!