Vorkus
Programmierung 2 (SS 2021)
Universität des Saarlandes
Fakultät MI

# Blatt - Befehlsreferenz (Tag 2/3)

Daniel Weber
Jessica Schmidt,
Luise Puhl,
Pascal Lauer

Der Vorkurs ist ein Angebot der (teilweise ehemaligen) Programmierung 2 Tutoren. Der Kurs ist keine offizielle Lehrveranstaltung. Es gibt keine CP, die Teilnahme ist optional, und eine HISPOS-Anmeldung ist weder möglich noch erforderlich. Bei Fragen zum Vorkurs und Programmierung 2 könnt ihr euch gerne an die Dozenten und Tutoren wenden. Wir wünschen euch ein erfolgreiches Semester und freuen uns, euch in Programmierung 2 wieder zu sehen.

## Grammatik

```
program:
(
'move'
| 'turn' LeftRight
| 'end'
| 'var' TypeDef Name
| 'set' Name Value
| 'get_position' Name Direction?
| 'get_direction' Name RelativeDirection?

| 'add' Name (Int | Position | Name) (Int | Position | Name)
| 'sub' Name (Int | Position | Name) (Int | Position | Name)
| 'and' Name (Int | Name) (Int | Name)
| 'or' Name (Int | Name) (Int | Name)
| 'xor' Name (Int | Name) (Int | Name)

| LabelDefinition
| 'test' Comparator Name Value Value
| 'explore' Name (Position | Name)
| 'call' Name
| 'jump' Name
| 'branch' Name Name
| 'set_arg' (Int | Name) Value
| 'get_arg' Name (Int | Name)

| 'mark' Direction?
| 'unmark' Direction?
| 'get_mark' Name
)* ;

TypeDef: 'int' | 'position' | 'direction' | 'object';
Name: [A-Za-z][A-Za-z0-9_]*;
LabelDefinition: Name ':';
Int: [1-9][0-9]*;
```

```
Position: '(' Int ',' Int ')' | '(' Name ',' Int ')'
        | '(' Name ',' Name ')' | '(' Int ',' Name ')';

LeftRight: 'left' | 'right';
RelativeDirection: LeftRight | 'front' | 'back' | 'here';
CompassDirection: 'north' | 'east' | 'south' | 'west';
Direction: CompassDirection | RelativeDirection;

Object: 'wall' | 'path';

Value: Int | Name | Direction | Object | Position;

Comparator: 'equal' | 'lequal' | 'less' | 'gequal' | 'greater' | 'nequal';
```

# Grundlegende Anweisungen

## move

`move`

Lässt die Eule einen Schritt gehen.

## turn

`turn LeftRight`

Dreht die Eule in die angegebene Richtung.

- `turn left` dreht die Eule einmal nach links.
- `turn right` dreht die Eule einmal nach rechts.

## var

`var TypeDef Name`

Deklariert eine neue Variable vom Typ `Typedef` mit dem Namen `Name`. Der Name (Bezeichner) darf lediglich einmal in einem Bereich deklariert werden. Gültige Bezeichner beginnen mit einem Buchstaben und beinhalten nur Buchstaben, Zahlen und Unterstriche. Wörter die fester Bestandteil von Ausrücken (`move`, `turn`, ...) sind, sind keine gültigen Bezeichner.

Ein Typ ist einer der folgenden:

- `int` um Zahlen zu speichern.
- `position` um ein Tupel von Zahlen zu speichern. Typischerweise wird dieser Typ benutzt um Positionen (Koordinaten) im Spielfeld abzuspeichern.
- `direction` um Richtungen zu speichern. Wir unterscheiden zwischen zwei Arten von Richtungen: `CompassDirection` und `RelativeDirection`. Eine `CompassDirection` ist unabhängig von der Eule. Also: `north, east, south, west`. Eine `RelativeDirection` hängt von der Eule ab. Also: `left, right, front, back, here`.

**Beispiel:** `var int x`: Deklariert eine Variable namens x von Typ `int`. In C würde das z.B. `(int x;)` entsprechen.

### set

`set Name Value`

Legt den Wert einer zuvor deklarierten Variable fest.

**Beispiel:** `set x 5` Setzt den Wert der Variable x auf 5. In C würde das z.B. `x = 5;` entsprechen.

### get_position

`get_position Name Direction?`

Beschreibt die angegebene Variable `Name` mit der aktuellen Position der Eule. Ist `Direction` angegeben, wird stattdessen die Position relativ zur Eule in die Variable geschrieben.

**Beispiele:**

- `get_position p` schreibt die Position der Eule in `p`.
- `get_position p left` schreibt die Position links der Eule in `p`.

### get_direction

`get_direction Name RelativeDirection?` Beschreibt die Variable `Name` mit der aktuellen `CompassDirection` die der Blickrichtung der Eule entspricht. Ist `(RelativeDirection)` angegeben, wird stattdessen die Blickrichtung relativ zur Eule in die Variable geschrieben.

**Beispiele:**

- `get_direction d` (entspricht: `get_direction d front`) schreibt die momentane Blickrichtung (als `CompassDirection`) in `d`.
- `get_direction d left` schreibt die `CompassDirection` in `d`, die der Blickrichtung der Eule entspräche wenn die Eule nach links schauen würde. Schaut die Eule z.B. nach Norden (`north`) würde, `d` auf Westen (`west`) gesetzt.

### explore

`explore Name Direction?`

Teste, ob das Feld in gegebener Richtung `Direction` begehbar ist. Speichere 0 für ein nicht begehbares Feld, 1 für ein begehbares Feld in gegebener Variable `Name`. Ist keine explizite Richtung `Direction?` gegeben, wird getestet, ob das in Blickrichtung liegende Feld begehbar ist. Die Zielvariable `Name` muss dementsprechend den Typen `int` haben.

## Arithmetische und logische Anweisungen

Kann einer der Parameter sowohl eine Konstante eines bestimmten Typs `type` sein, als auch eine Variable `name`, dann darf `name` in der Anweisung benutzt werden wenn die Variable mit dem Typ `type` deklariert wurde. In diesem Fall benutzt der Interpreter den Wert der Variable zur Auswertung. Ist z.B. `v` eine Variable vom Typ `int` mit dem Wert 5, ist `sub v v v` oder `add v v 5` eine valide Anweisung. Das ist in diesem Fall dann gleichbedeutend mit `sub v 5 5` bzw. `add v 5 5`. Deswegen beziehen wir uns im folgenden für diese Parameter nur auf ihre Typen.

## add/sub

`add Name (Int | Position | Name) (Int | Position | Name)`
als `add var1 val1 val2`
---
`sub Name (Int | Position | Name) (Int | Position | Name)`
als `add var1 val1 val2`

Hierbei steht `add` für eine Addition von `val1` und `val2`. Und `sub` für eine Subtraktion von `val1` und `val2`. Das Ergebnis wird jeweils in `var1` geschrieben. In einer gültigen Anweisung müssen `var1`, `val1` und `val2` den gleichen Typ haben. Gültige Typen für diese Operationen sind `int` und `position`. Die Addition und Subtraktion von Positionen agiert paarweise.

**Beispiele:**

- `add v 1 1` schreibt den Wert 2 in die Variable `v`.
- `sub v (1,2) (3,4)` schreibt den Wert (-2, -2) in die Variable `v`.
- Sei `a` eine Variable vom Typ `int` mit dem Wert 1. Sei `b` eine Variable vom Typ `position` mit dem Wert `(3,4)`. Dann ist die Anweisung `add v (a,2) b` gleichbedeutend mit `add v (1,2) (3, 4)` womit die Anweisung (4,6) in die Variable `v` schreibt.

## and/or/xor

`and Name (Int | Name) (Int | Name)`
als `and var1 val1 val2`
---
`or Name (Int | Name) (Int | Name)`
als `or var1 val1 val2`
---
`xor Name (Int | Name) (Int | Name)` als `xor var1 val1 val2`

Hierbei steht `and` für die logische Operation und, also: `val1` & `val2`. `or` steht für die logische Operation entweder oder, also: `val1` | `val2`. Und `xor` steht für die logische Operation entweder oder, also: `val1` ^ `val2` auch bekannt als `val1` $\oplus$ `val2`. Alle Operationen erfolgen bit-weise. Das Ergebnis wird jeweils in `var1` geschrieben. In einer gültigen Anweisung müssen `var1`, `val1` und `val2` den Typ `int` haben.

**Beispiele:**

- `and v 1 0` schreibt den Wert 0 in die Variable `v`.
- `or v 1 0` schreibt den Wert 1 in die Variable `v`.
- `xor v 1 0` schreibt den Wert 1 in die Variable `v`.

# Kontrollfluss-Anweisungen

## LabelDefinition

`Name:`

Markiert den Beginn eines Unterprogramms dem Namen `Name`. Bzw. setzt eine Sprungmarke mit dem Namen `Name` an die gegebene Stelle. Ein Label muss einzigartig sein. D.h. der `Name` eines Labels darf genau einmal innerhalb des Programms vergeben werden. Wörter die fester Bestandteil von Ausrücken (`move`, `turn`, ...) sind, sind keine gültigen Label.

## jump

`jump Name`

`Name` muss ein im Programm definiertes Label sein. Springt zur Sprungmarke `Name`.

## call

`call Name`

`Name` muss ein im Programm definiertes Label sein. Ruft das Unterprogramm `Name` auf. Der Unterschied zwischen `call` und `jump` ist dass bei `call` der Punkt des Aufrufs gespeichert wird um dahin zurück zu kehren. (Siehe `end`.)

## set_arg

`set_arg (Int | Name) Value`
als `set_arg num val`

Setzt das `num`-te Argument vor dem Aufruf (siehe `call`) auf `val`.

## get_arg

`get_arg Name (Int | Name)`
als `get_arg var num`

Schreibt das `num`-te Argument im Unterprogramm in die Variable `var`.

## branch

`branch Name Name`
als `branch cond label`

`label` muss ein im Programm definiertes Label sein. `cond` muss eine Variable vom Typ `int` sein. Wenn der Wer von `cond` ungleich 0 ist, springt das Programm zu Sprungmarke `label`.

### test

`test Comparator Name Value Value`
als `test cmp var1 val1 val2`

Auswertung von `val1` und `val2` erfolgt wie für arithmetische und logische Anweisungen. `var1`, `val1` und `val2` müssen den gleichen Typ haben. `cmp` muss einem der folgenden Werte entsprechen:

- `equal` für ein Vergleich mit Gleich (=).
- `lequal` für ein Vergleich mit Kleiner-Gleich (≤).
- `less` für ein Vergleich mit Kleiner (<).
- `gequal` für ein Vergleich mit Größer-Gleich (≥).
- `greater` für ein Vergleich mit Größer (>).
- `nequal` für ein Vergleich mit Ungleich (≠).

`equal` und `nequal` können auf allen Typen ausgewertet werden. Die anderen Vergleichsoperatoren nur auf `int`. Das Ergebnis (1 für richtig, 0 für falsch) des durch `cmp` definierten Vergleichs wird in `var1` geschrieben. Diese Vergleiche sind meistens dazu gedacht in Kombination mit `branch` verwandt zu werden.

**Beispiele:**

- `test equal v 0 1` schreibt 0 in `v`.
- `test lequal v 0 1` schreibt 1 in `v`.

### end

`end`

Beendet die momentan laufende Unterfunktion. (Kehrt zur Zeile des Aufrufs, sprich der zuletzt ausgeführten `call` Anweisung, zurück.)

## Markierungs Anweisungen

### mark

`mark Direction?`

Setze auf dem jetzigen Feld des Spielers eine Markierung für eine bestimmte Richtung `Direction`. Ist keine Richtung `Direction` angegeben, wird eine unbestimmte Markierung, also alle Markierungen für alle Richtungen, gesetzt.

### unmark

`unmark Direction?`

Entferne auf dem jetzigen Feld des Spielers eine Markierung für eine bestimmte Richtung `Direction`. Ist keine Richtung `Direction` angegeben, werden alle Markierungen entfernt.

### get_mark

`get_mark Name`

Speichere alle Markierungen des jetzigen Feld des Spielers in einer Variablen `Name`. `Name` muss den Typ int haben. Alle Markierungen werden in einer Variablen gespeichert. Hierfür wird eine sogenannte Binär-Codierung verwendet:

- 0b0000 = 0: keine Richtung ist markiert
- 0b1111 = 15: alle Richtungen sind markiert
- 0b0001 = 1: Westen ist markiert
- 0b0010 = 2: Süden ist markiert
- 0b0100 = 4: Osten ist markiert
- 0b1000 = 8: Norden ist markiert

Die einzelnen Richtungen können dann noch entsprechend kombiniert werden. Da diese Kombinationen eindeutig sind, kann durch die verschiedenen Ergebnisse berechnet werden, welche Markierungen gesetzt sind.

**Beispiel:** Westen (0b0001) + Norden (0b1000) entspricht 0b0001 + 0b1000 = 0b1001 = 9

## Array und Matrix

Pro Task existiert eine globales Array und eine globale Matrix, die verwendet werden können um Daten zu speichern. Das Array erlaubt den Zugriff auf eine hohe Anzahl von Speicherbehältern über einen beliebigen `int`. Das Array erlaubt den Zugriff auf eine hohe Anzahl von Speicherbehältern über eine beliebige `position`. Die Matrix kann beispielsweise dazu verwendet werden sich Informationen über Positionen im Spielfeld zu speichern. Das Array kann z.B. dazu verwendet werden um sich eine Reihenfolge zu merken.

Das Array und Matrix sind nicht getypt. Das heißt, dass beispielsweise an Position 1 ein Wert des Typen `int` liegen kann, auch wenn an Position 2 ein Wert des Typen `direction` gesetzt ist. Denke daran, dass Variablen allerdings immer getypt sein müssen. Willst du also einfach nur über die Liste iterieren und jeden einzelnen Wert auslesen, muss immer eine Variable mit dem richtigen Typ verwendet werden. Befinden sich verschiedene Typen in der Liste, kann dies kompliziert werden, da du dir merken musst, welcher Typ an welcher Position verwendet wird, um dann die richtige Variable auszuwählen. Ein Zugriff auf einen Wert in der Liste oder Matrix, der vorher nicht gesetzt wurde, führt zu einem Fehler.

### arr_set

`arr_set (Int | Name)  Value`
als `arr_set i val`

Setze den Wert an einem bestimmten Index `i` auf den gegebenen Wert `val`. Dabei muss `i` vom Typ `int` sein und der Typ von `val` ist beliebig.

### arr_get

`arr_get Name (Int | Name)`
als `arr_get  var1 i`

Schreibt den Wert des Arrays an Index `i` in die Variable `var1`. Dabei muss `var1` den gleichen Typ haben wie der Inhalt der im Array an Index `i` hinterlegt wurde.

### matr_set

`matr_set (Position | Name) Value` als `arr_set pos val`

Setze den Wert an einer bestimmten Position `pos` auf den gegebenen Wert `val`. Dabei muss `pos` vom Typ `position` sein und der Typ von `val` ist beliebig.

### matr_get

`matr_get Name (Position | Name)`
als `matr_get var1 pos`

Schreibt den Wert der Matrix an Position `pos` in die Variable `var1`. Dabei muss `var1` den gleichen Typ haben wie der Inhalt der in der Matrix an Position `pos` hinterlegt wurde.