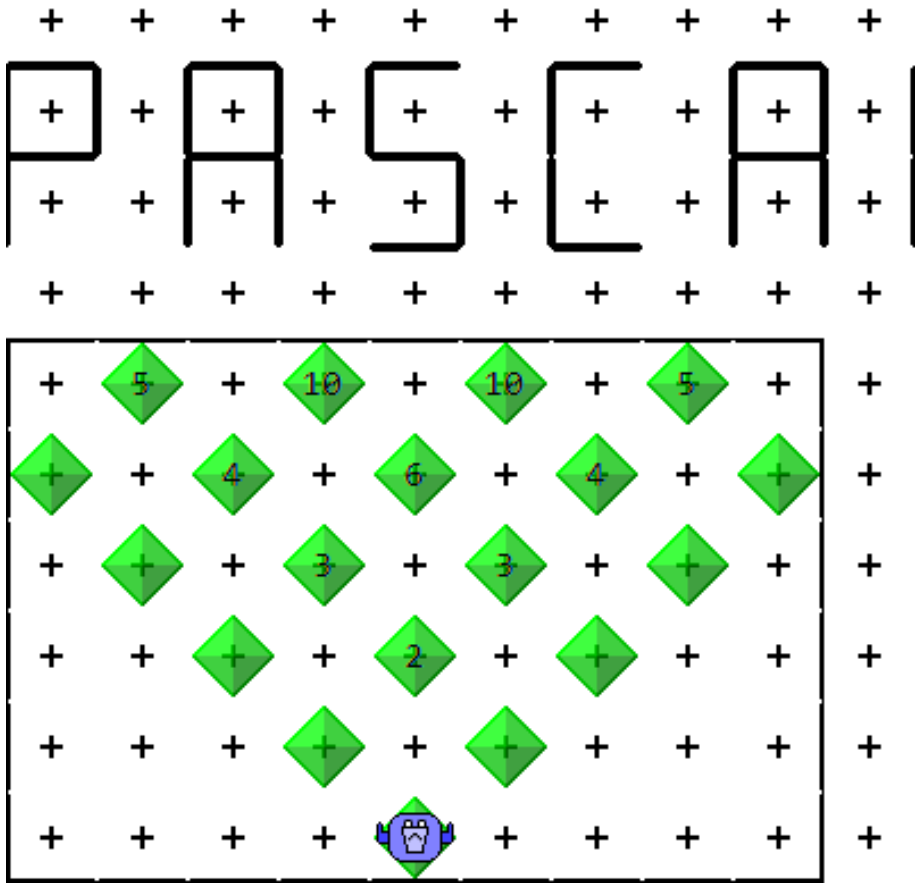
# Karel the Robot – Lektion III

- Rekursion



# Wiederholung: Kontrollfluss

- **Sequenz**: Karel führt die Befehle nacheinander aus.
- **Fallunterscheidung**: Abhängig von einer Bedingung führt Karel einen von zwei Blöcken aus (wobei der zweite Block optional ist).
- **Wiederholung**
  - **Zählschleife**: Karel führt einen Block n Mal aus.
  - **Bedingte Schleife**: Abhängig von einer Bedingung führt Karel einen Block immer wieder aus.
- **Eigene Befehle**: Karel merkt sich, wo der Aufruf steht und springt in den Befehl hinein. Am Ende des Befehls springt er hinter den Aufruf zurück.

# Grenzen der bedingten Schleife

Angenommen, wir wollen zur nächsten Wand laufen und dann wieder zurück:

```
void moveToWallAndBack()
{
    while (frontIsClear())
    {
        moveForward();
    }
    turnAround();
    ???
}
```

Leider kann Karel sich an der blau markierten Stelle nicht mehr daran erinnern, wie oft er `moveForward` auf dem Hinweg ausgeführt hat. In anderen Programmiersprachen würde man dafür eine Zählvariable verwenden, aber Karel kennt keine Variablen.

# Lösungen für Wege der maximalen Länge 1

```
void journey1()
{
    if (!frontIsClear())
    {
        turnAround();
    }
    else
    {
        moveForward();
        turnAround();
        moveForward();
    }
}
```

# Lösungen für Wege der maximalen Länge 2 bzw. 3

```
void journey2()
{
    if (!frontIsClear())
    {
        turnAround();
    }
    else
    {
        moveForward();
        journey1();
        moveForward();
    }
}
```

Wir lösen ein Problem der Größe 2, indem wir es auf ein bereits gelöstes Problem der Größe 1 zurückführen.

```
void journey3()
{
    if (!frontIsClear())
    {
        turnAround();
    }
    else
    {
        moveForward();
        journey2();
        moveForward();
    }
}
```

Wir lösen ein Problem der Größe 3, indem wir es auf ein bereits gelöstes Problem der Größe 2 zurückführen.

# Lösung für beliebig lange Wege

Wir erkennen folgendes Muster:

```
void journey_n()
{
    if (!frontIsClear())
    {
        turnAround();
    }
    else
    {
        moveForward();
        journey_n_minus_one();
        moveForward();
    }
}
```

Anstatt unendlich viele `journey_n` Befehle zu schreiben, reicht auch ein einziger Befehl, der sich selbst aufruft!

```
void journey()
{
    if (!frontIsClear())
    {
        turnAround();
    }
    else
    {
        moveForward();
        journey();
        moveForward();
    }
}
```

# Wie kann das funktionieren? Vollständige Induktion

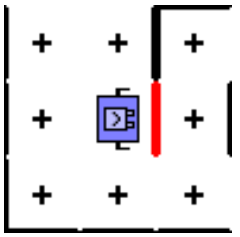
Mit vollständiger Induktion können wir uns davon überzeugen, dass diese Lösung für Reisen beliebiger Länge funktioniert:

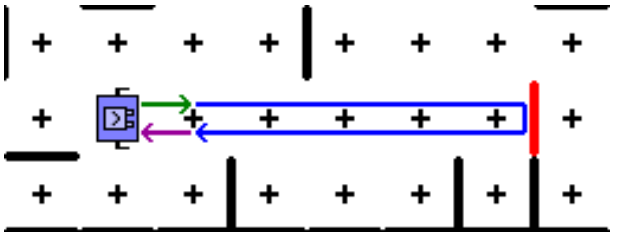
1. **Induktionsanfang:** Funktioniert die Lösung für Reisen der Länge 0? Ja, in dem Fall dreht Karel sich einfach um.

2. **Induktionsannahme:** Die Lösung funktioniert für Reisen der Länge n-1.

   **Induktionsschluss:** Funktioniert die Lösung dann auch für Reisen der Länge n? Ja; eine Reise der Länge n besteht aus einem zusätzlichen Schritt auf dem Hinweg, einer Reise der Länge n-1 und einem zusätzlichen Schritt auf dem Rückweg.

```
void journey()
{
    // Abbruchbedingung
    if (!frontIsClear())
    {
        // Basisfall
        turnAround();
    }
    else
    {
        // Rekursiver Fall
        moveForward();
        journey();
        moveForward();
    }
}
```





# Ein zweites Beispiel zur vollständigen Induktion

Karel soll die Anzahl der Diamanten, die sich unter ihm befinden, verdoppeln.

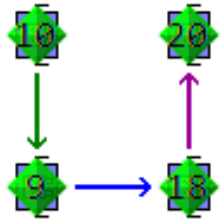
Mit vollständiger Induktion können wir uns davon überzeugen, dass diese Lösung für beliebig viele Diamanten funktioniert:

1. **Induktionsanfang:** Funktioniert die Lösung für 0 Diamanten? Ja, in dem Fall ist nichts zu tun, denn 2*0 = 0.

2. **Induktionsannahme:** Die Lösung funktioniert für n-1 Diamanten.

   **Induktionsschluss:** Funktioniert die Lösung dann auch für n Diamanten? Ja, denn 2*n = 2*(n-1)+1+1.

```
void doubleBeepers()
{
    // Abbruchbedingung
    if (!onBeeper())
    {
        // Basisfall: nix zu tun
    }
    else
    {
        // Rekursiver Fall
        pickBeeper();
        doubleBeepers();
        dropBeeper();
        dropBeeper();
    }
}
```



# Bedingte Schleifen durch Rekursion simulieren

Theoretisch benötigt Karel keine bedingte Schleife, da alle bedingten Schleifen mit Rekursion simuliert werden können.

In der Praxis sind bedingte Schleifen aber meist besser verständlich als Rekursion.

Daher sollten wir uns Rekursion für diejenigen Fälle aufheben, die nicht (oder zumindest nicht elegant) mit bedingten Schleifen gelöst werden können.

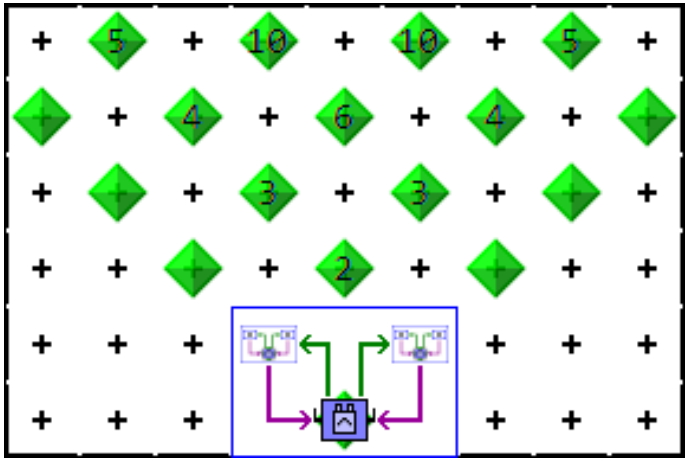
```
void moveToWallIterativ()
{
    while (frontIsClear())
    {
        moveForward();
    }
}

void moveToWallRekursiv()
{
    if (frontIsClear())
    {
        moveForward();
        moveToWallRekursiv();
    }
}
```

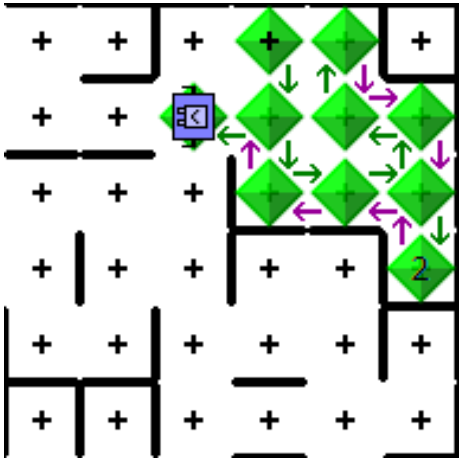
# Mehrfachrekursion (für Fortgeschrittene)

- Bisher haben wir Befehle gesehen, die nur einen Selbstaufruf enthielten. Es können aber beliebig viele Selbstaufrufe in einem Befehl stehen.

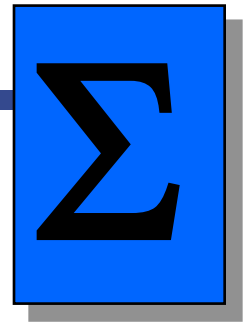
- Beim Zeichnen von binären Bäumen benötigt man zwei Selbstaufrufe.

- Bei der Wegfindung in komplexen Labyrinthen (mit Zyklen und offenen Flächen) benötigt man vier Selbstaufrufe (einen pro Himmelsrichtung).



# Zusammenfassung



- Ein **rekursiver Befehl** ist ein Befehl, der **sich selbst aufruft**.
- Rekursive Befehle sind so strukturiert, dass sie ein großes Problem auf kleinere Probleme zurückführen, welche zum Teil rekursiv gelöst werden.
- Irgendwann ist das Problem so trivial, dass kein rekursiver Aufruf mehr erforderlich ist (ansonsten würde Karel in einer **Endlosrekursion** landen).
- Den rekursiven Aufruf eines Befehls nennt man **rekursiven Abstieg**.
- Die Rückkehr aus einem rekursiven Aufruf heißt **rekursiver Aufstieg**.
- Die Fähigkeit, nach dem rekursiven Aufstieg noch etwas tun zu können, macht Rekursion in Karels Welt mächtiger als die bedingte Schleife.