# Operationale Semantik

# Semantik

legt Bedeutung von Programmen einer Sprache fest

verschiedene Arten von Semantik

- operationale Semantik: *wie* wird das Programm *ausgeführt*?
  - *small-step*: erkläre, wie man schrittweise zum Ergebnis kommt
  - *big-step*: setze Programme und Ergebnisse in Relation
- denotationelle Semantik: *welche Funktion* wird von dem Programm berechnet?
- axiomatische Semantik: *welche Eigenschaften* erfüllt das Programm?

hier: small-step operational

# Notwendigkeit einer formalen Semantik

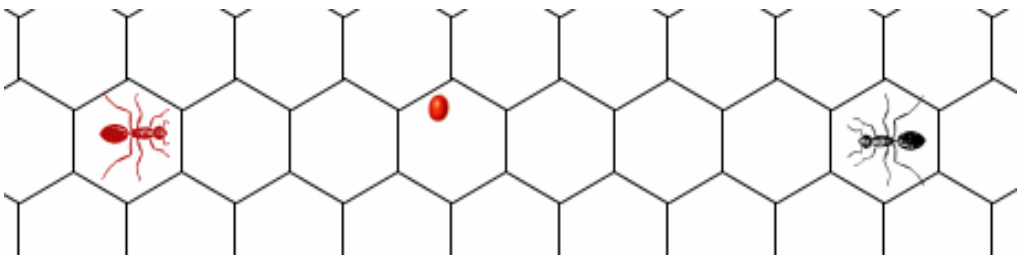
Programmierer und Compiler / Interpreter sollten Programm auf gleiche Art interpretieren

- unvermeidbar bei sicherheitsrelevanten Anwendungen
- evtl. störend sonst, z.B. browser-abhängige Darstellung von HTML-Dokumenten, Interpretation von JavaScript-Programmen

**Bsp.:**

```
while true {
  if sense(here,food)
  then pickup
  else skip;
  walk
}
```

```
while true {
  if sense(here,food)
  then pickup
  else walk
}
```



# Operationale Semantik

zur Erinnerung: small-step Semantik erklärt schrittweise Ausführung eines Programms

Schritte zeichnen sich aus durch

- Zustandsänderungen in der ausführenden Maschine
- Änderungen des auszuführenden Programms

**Bsp.:** führe `if x > 0 then { turn(1); walk } else skip` aus

1. werte `x > 0` aus (evtl. auch schrittweise)
2. je nach Ergebnis führe `turn(1); walk` oder `skip` aus

jetzt nur noch formal fassen ...

# Formale operationale Semantik

kann z.B. angegeben werden durch Funktion

*step*: *Programmzustände* $\times$ *Eingaben* $\rightarrow$ *Aktionen* $\times$ *Programmzustände*

Programmzustand = Variablenbelegung $\rho$ und noch auszuführende Programme $P_1 : \ldots : P_k$

Funktion *step* wird von Interpreter realisiert

Ausführen des Programms $P$:

- zu Beginn alle Variablen undefiniert: $\rho_0(x) = \perp$ für alle $x$
- Eingabe im $i$-ten Schritt ist $\sigma_{i-1}$
- iteriere:
  1. berechne $step((\rho_0, P), \sigma_0)$, Ergebnis ist $(a, (\rho_1, P_1 : \ldots : P_n))$
  2. führe $a$ aus (Bsp.: Ausgabe)
  3. berechne $step((\rho_1, P_1 : \ldots : P_n), \sigma_1)$ ...

# Variablenbelegungen

**Bsp.:** was ist der Wert des Ausdrucks `x > 0` ?

Antwort hängt von Wert von `x` ab

essentieller Teil des Programmzustands: Werte der im Programm auftretenden Variablen

Variablenbelegung ist endliche Funktion von Variablen auf Werte, kann auch partiell sein (in Java z.B. durch Wert `null` realisiert)

Verwendung z.B. in der Auswertung von Ausdrücken: Notation $\|\mathtt{x} > 0\|_\rho$ zu lesen als "Wert des Ausdrucks $\mathtt{x} > 0$ *unter* der Variablenbelegung $\rho$"

**Bsp.:** falls $\rho(\mathtt{x}) = 5$, dann $\|\mathtt{x} > 0\|_\rho = 1$

# Stacks

wieso eigentlich "noch auszuführende Programme $P_1 : \ldots : P_n$"?

**Bsp.1:** wie wird `while true do { x = x-1; walk }` ausgeführt?

- führe `x = x-1; walk` aus
- führe danach wieder `while true do { x = x-1; walk }` aus

und wie wird `x = x-1; walk` ausgeführt?

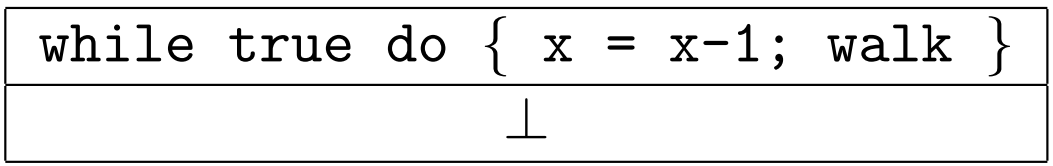
- führe erst `x = x-1` aus
- führe danach `walk` aus, aber noch *vor* nächstem Schleifendurchlauf!

Stack ist geeignete Struktur, um die Liste der noch auszuführenden Programme in ihrer Reihenfolge zu merken

# Stacks – Beispiel

Stackinhalte beim Ausführen des Programms `while true do { x = x-1; walk }`

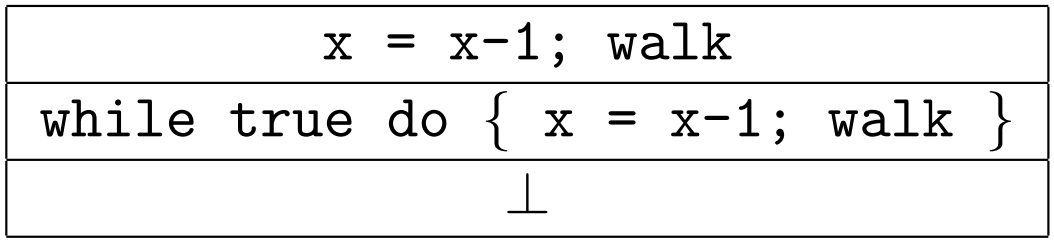
Ende des Stacks markiert mit speziellem Symbol $\perp$ (nichts mehr auszuführen)



# Stacks – Beispiel

Stackinhalte beim Ausführen des Programms `while true do { x = x-1; walk }`

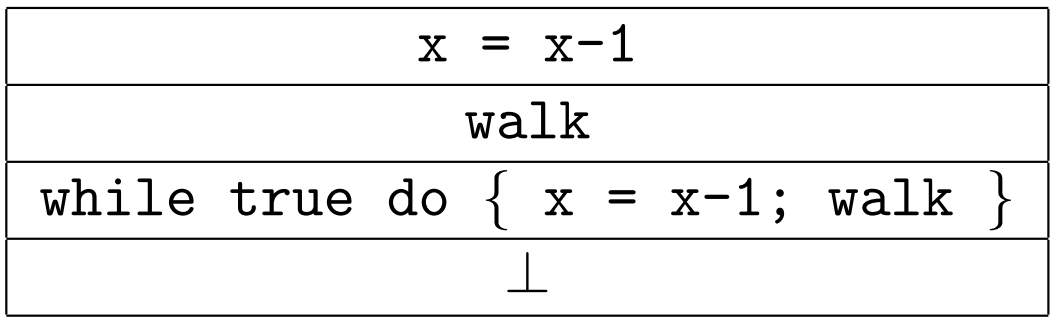
Ende des Stacks markiert mit speziellem Symbol $\perp$ (nichts mehr auszuführen)



# Stacks – Beispiel

Stackinhalte beim Ausführen des Programms `while true do { x = x-1; walk }`

Ende des Stacks markiert mit speziellem Symbol $\perp$ (nichts mehr auszuführen)



# Stacks – Beispiel

Stackinhalte beim Ausführen des Programms `while true do { x = x-1; walk }`

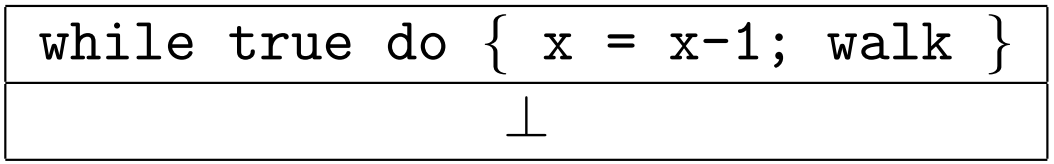
Ende des Stacks markiert mit speziellem Symbol $\perp$ (nichts mehr auszuführen)

| `walk` |
| --- |
| `while true do { x = x-1; walk }` |
| $\perp$ |

# Stacks – Beispiel

Stackinhalte beim Ausführen des Programms `while true do { x = x-1; walk }`

Ende des Stacks markiert mit speziellem Symbol $\perp$ (nichts mehr auszuführen)



# Interpreter für AntBrain

zur Erinnerung: muss inbesondere *step* realisieren

Modellierung (Vorschlag):

- Klasse `Ant` realisiert Ameise mit Interpreter für AntBrain mittels Funktion *step*
- Eingabe $\sigma$ als Parameter an *step* (hier: zwei Zellen)
- Programmzustand $(\rho, P_1 : \ldots : P_n)$ über Instanzvariablen realisiert, weil sie aktualisiert und in nächstem Schritt wiederverwendet werden müssen

```
class Ant {
  private Random random;
  private Map<String, Integer> rho;
  private Deque<Prg> stack;              // Deque ersetzt Stack
  ...
  public Action step(Cell here, Cell ahead) {
    ...
  }
}
```

# Zu beachten

- Interpreter kann wiederum mehrere Schritte (auf der JVM) brauchen, um einen Schritt in der operationalen Semantik von AntBrain auszurechnen
- es bietet sich an, Funktion `step` rekursiv zu implementieren
- Termination der `step`-Funktion i.A. ein Problem, hier aber gewährleistet (wegen Verkleinerung des obersten Programms auf dem Stack)