# Aufgabenbeschreibung
# Softwareentwicklungspraktikum

Wintersemester 2008/2009
Universität München
Martin Lange und Ulrich Schöpp

Teil III – Programmiersprache AntBrain
Version 4

## 1 Einleitung

AntBrain ist eine einfache Sprache zur Programmierung von Ameisengehirnen. Folgendes ist zum Beispiel ein AntBrain-Programm.

```
lookingForFood = 1;
while true do {
  if sense(here, food) then {pickup; lookingForFood = 0} else skip;
  if sense(here, home) then {drop; lookingForFood = 1} else skip;
  t = rand%2 + rand%2 - 1;  turn(t);
  walk
}
```

Darin wird der Kontrollfluss mit `"while"` und `"if"` gesteuert, es werden Variablen verwendet (`"lookingForFood"` und `"t"`) und es werden einige fest in AntBrain eingebaute Befehle zur Ameisensteuerung benutzt (z.B. `"turn"` und `"walk"`).

Die Variablen in AntBrain sind die einzige Möglichkeit der Ameisen sich etwas zu merken. Allerdings sind Ameisen nicht gerade für ihr großes Gedächtnis bekannt. Und tatsächlich können sich die Ameisen nur Werte von 0 bis 5 merken, d.h. der Wert einer Variable ist ein Integer aus dem Bereich $\{0, 1, 2, 3, 4, 5\}$. Außerdem können die Ameisen nicht mehr als 12 Variablen handhaben, d.h. in einem AntBrain Programm können höchstens 12 verschiedene Variablen vorkommen.

Immerhin können die Ameisen mit diesen Zahlen rechnen (wahrscheinlich, indem sie die Werte an ihren sechs Beinen abzählen). Sie beherrschen

arithmetische Operationen wie Addition, Multiplikation und Division. Da sie sich keine großen Zahlen merken können, rechnen sie dabei immer modulo 6, z.B. $5 + 2 = 1$ oder $3 * 7 = 3$. Die Ameisen können mit dem Schlüsselwort `"rand"` auch zufällig Zahlen erzeugen.

Variablen müssen in AntBrain nicht vorher deklariert werden. Ihnen kann einfach ein Wert zugewiesen werden. Die Variablennamen können beliebige Bezeichner sein. Im obigen Beispiel hätten wir auch `"s"` anstelle von `"lookingForFood"` schreiben können. Im Gegensatz zu Java z.B. haben Variablen keine vordefinierten Werte. Einer Variablen muss zuerst ein Wert zugewiesen werden, bevor sie verwendet wird. Wird eine Variable benutzt, der vorher kein Wert zugewiesen wurde, dann terminiert das Programm.

Um AntBrain einfach zu halten, gibt es trotz der `if`-Verzweigung und der `while`-Schleife keine separaten booleschen Ausdrücke. Die Rolle von booleschen Ausdrücken wird auch von den arithmetischen Ausdrücken übernommen. Die Zahl 0 steht dabei für die falsche Aussage und jede Zahl größer 0 zählt als wahrer Ausdruck. Die Schlüsselwörter `"false"` und `"true"` sind einfach Abkürzungen für 0 und 1.

Der Befehl `"skip"` macht gar nichts und ist im obigen Programm nur vorhanden, weil man in den `"else"`-Teil nach einem `"if"` in AntBrain nicht einfach weglassen darf. Diese Bedingung macht es einfacher, gültige AntBrain-Programme zu erkennen.

# 2 Syntax

Eine Grammatik, mit der man die zulässigen AntBrain-Programme erkennen kann, ist in Abbildung 1 gegeben.

In dieser Grammatik werden Terminalsymbole wie `"skip"` oder *Identifier* benutzt. Dabei handelt es sich um Tokens, in die der Eingabestring zunächst zerlegt wird.

In AntBrain ist ein Token entweder ein Schlüsselwort, ein Symbol, ein Bezeichner, eine Zahl, oder die Markierung *Eof* für das Ende der Eingabe. Diese Tokens sind folgendermaßen definiert:

**Schlüsselwörter** Folgende Strings sind Schlüsselwörter: `"skip"`, `"turn"`, `"walk"`, `"pickup"`, `"drop"`, `"leavescent"`, `"clearscent"`, `"while"`, `"do"`, `"if"`, `"then"`, `"else"`, `"rand"`, `"sense"`, `"here"`, `"ahead"`, `"food"`, `"obstacle"`, `"home"`, `"foehome"`, `"ownscent"`, `"foescent"`, `"friend"`, `"foe"`, `"carriesfood"`, `"true"`, `"false"`. In der Grammatik haben wir Schlüsselwörter einfach durch den entsprechenden String dargestellt. Wenn wir besonders betonen wollen, dass es sich bei einem String um ein Schlüsselwort handelt, schreiben wir z.B. auch

Programme:

$$
\begin{aligned}
\mathit{Prg} &::= \mathit{PrgSeq}\ \mathit{Eof} \\
\mathit{PrgSeq} &::= \mathit{PrgAtom}\ (\texttt{";"}\ \mathit{PrgAtom})^* \\
\mathit{PrgAtom} &::= \texttt{"skip"} \mid \texttt{"walk"} \mid \texttt{"turn"}\ \texttt{"("}\ \mathit{IExp}\ \texttt{")"} \mid \texttt{"pickup"} \mid \texttt{"drop"} \\
&\quad \mid \texttt{"leavescent"}\ \texttt{"("}\ \mathit{IExp}\ \texttt{")"} \mid \texttt{"clearscent"}\ \texttt{"("}\ \mathit{IExp}\ \texttt{")"} \\
&\quad \mid \mathit{Identifier}\ \texttt{"="}\ \mathit{IExp} \\
&\quad \mid \texttt{"\{"}\ \mathit{PrgSeq}\ \texttt{"\}"} \\
&\quad \mid \texttt{"if"}\ \mathit{IExp}\ \texttt{"then"}\ \mathit{PrgAtom}\ \texttt{"else"}\ \mathit{PrgAtom} \\
&\quad \mid \texttt{"while"}\ \mathit{IExp}\ \texttt{"do"}\ \mathit{PrgAtom}
\end{aligned}
$$

Arithmetische Ausdrücke:

$$
\begin{aligned}
\mathit{IExp} &::= \mathit{IExp}_1 \\
\mathit{IExp}_i &::= \mathit{IExp}_{i+1}\ (\mathit{op}_i\ \mathit{IExp}_{i+1})^* \qquad \text{für } i = 1, \ldots, 5 \\
\mathit{IExp}_6 &::= \texttt{"-"}\ \mathit{IExp}_6 \mid \texttt{"!"}\ \mathit{IExp}_6 \mid \texttt{"("}\ \mathit{IExp}_1\ \texttt{")"} \mid \mathit{IExpAtom} \\
\mathit{IExpAtom} &::= \mathit{Identifier} \mid \mathit{Number} \mid \texttt{"true"} \mid \texttt{"false"} \mid \texttt{"rand"} \\
&\quad \mid \texttt{"sense"}\ \texttt{"("}\ \mathit{Where}\ \texttt{","}\ \texttt{"food"}\ \texttt{")"} \\
&\quad \mid \texttt{"sense"}\ \texttt{"("}\ \mathit{Where}\ \texttt{","}\ \texttt{"obstacle"}\ \texttt{")"} \\
&\quad \mid \texttt{"sense"}\ \texttt{"("}\ \mathit{Where}\ \texttt{","}\ \texttt{"home"}\ \texttt{")"} \\
&\quad \mid \texttt{"sense"}\ \texttt{"("}\ \mathit{Where}\ \texttt{","}\ \texttt{"friend"}\ \texttt{")"} \\
&\quad \mid \texttt{"sense"}\ \texttt{"("}\ \mathit{Where}\ \texttt{","}\ \texttt{"foe"}\ \texttt{")"} \\
&\quad \mid \texttt{"sense"}\ \texttt{"("}\ \mathit{Where}\ \texttt{","}\ \texttt{"foehome"}\ \texttt{")"} \\
&\quad \mid \texttt{"sense"}\ \texttt{"("}\ \mathit{Where}\ \texttt{","}\ \texttt{"carriesfood"}\ \texttt{")"} \\
&\quad \mid \texttt{"sense"}\ \texttt{"("}\ \mathit{Where}\ \texttt{","}\ \texttt{"foescent"}\ \texttt{")"} \\
&\quad \mid \texttt{"sense"}\ \texttt{"("}\ \mathit{Where}\ \texttt{","}\ \texttt{"ownscent"}\ \texttt{","}\ \mathit{IExp}\ \texttt{")"} \\
\mathit{Where} &::= \texttt{"here"} \mid \texttt{"ahead"}
\end{aligned}
$$

Die Binäroperatoren sind entsprechend ihrer Präzedenz gegeben:

$$
\begin{aligned}
\mathit{op}_1 &::= \texttt{"||"} \\
\mathit{op}_2 &::= \texttt{"\&\&"} \\
\mathit{op}_3 &::= \texttt{"=="} \mid \texttt{"<="} \mid \texttt{"<"} \mid \texttt{">="} \mid \texttt{">"} \\
\mathit{op}_4 &::= \texttt{"+"} \mid \texttt{"-"} \\
\mathit{op}_5 &::= \texttt{"*"} \mid \texttt{"/"} \mid \texttt{"\%"}
\end{aligned}
$$

Abbildung 1: Grammatik

*Keyword*(`"skip"`), ... (in der Grammatik hätte das zu viel Platz weggenommen).

**Symbole** In AntBrain gibt es folgende Symbole: `","`, `";"`, `"("`, `")"`, `"{"`, `"}"`, `"!"`, `"||"`, `"&&"`, `"+"`, `"-"`, `"*"`, `"/"`, `"%"`, `"="`, `"=="`, `"<="`, `"<"`, `">="`, `">"`. Wie bei Schlüsselwörtern notieren wir Symboltokens auch durch *Symbol*(`";"`), ....

**Bezeichner** Ein Bezeichner (engl. Identifier) ist ein Wort aus Buchstaben und Zahlen, das mit einem Buchstaben beginnt. Beispiele: `"x1"`, `"i23"`, `"lookingForFood"`. Schlüsselwörter zählen nicht zu den Bezeichnern, d.h. `"rand"` ist kein Bezeichner, aber `"rand1"` ist einer.

Bezeichner sind also alle Wörter, die keine Schlüsselwörter sind und die von dem regulären Ausdruck *Alpha* (*Alpha* | *Digit*)* beschrieben werden, wobei

$$\begin{aligned} \mathit{Alpha} &::= \texttt{"a"} \mid \cdots \mid \texttt{"z"} \mid \texttt{"A"} \mid \cdots \mid \texttt{"Z"} \ , \\ \mathit{Digit} &::= \texttt{"0"} \mid \cdots \mid \texttt{"9"} \ . \end{aligned}$$

In der Grammatik schreiben wir *Identifier* für einen beliebigen Bezeichner.

**Zahlen** Eine Zahl besteht aus einer Folge von Ziffern (d.h. Vorzeichen werden nicht zum Token für Zahlen gezählt). Regulärer Ausdruck: $\mathit{Digit}^+$.

In der Grammatik schreiben wir *Number* für einen beliebigen Zahlausdruck.

Die Grammatik in Abb. 1 leitet also nicht Programme wie das Beispiel in der Einleitung her, sondern Listen von Tokens, die Programme wie das in der Einleitung repräsentieren. Dabei werden für die Programmstruktur also z.B. Leerzeichen und Zeilenumbrüche nicht mehr berücksichtigt. Zum Beispiel werden die Strings `"1==x"` und `"1 == x "` beide durch die Tokenliste

$$\mathit{Number}(1), \mathit{Symbol}(\texttt{"=="}), \mathit{Identifier}(\texttt{"x"})$$

repräsentiert. Das Beispielprogramm am Anfang der Einführung wird zu der Tokenliste

$$\mathit{Identifier}(\texttt{"lookingForFood"}), \mathit{Symbol}(\texttt{"="}), \mathit{Number}(\texttt{"1"}), \mathit{Symbol}(\texttt{";"}),$$
$$\mathit{Keyword}(\texttt{"while"}), \mathit{Keyword}(\texttt{"true"}), \mathit{Keyword}(\texttt{"do"}), \mathit{Symbol}(\texttt{"\{"}), \ldots$$

Es ist Aufgabe eines Lexers für AntBrain, Programme wie das in der Einleitung in entsprechende Tokenliste umzuwandeln, aus denen dann der Parser die Programmstruktur aufbauen kann.

# 3 Erkennen von gültigen Programmen

Als nächstes wird ein Parser für AntBrain benötigt. Dieser erhält als Eingabe ein Tokenliste und konstruiert dazu einen Ableitungsbaum bezüglich der Grammatik in Abb. 1 – oder er bricht den Versuch mit einer entsprechenden Fehlermedlung ab, falls die vorgelegte Tokenliste kein Programm darstellt, welches bezüglich dieser Grammatik gültig ist.

Das Startsymbol der Grammatik ist dabei *Prg*, d.h. die Tokenliste ist nur dann erfolgreich geparst, wenn aus ihr ein Ableitungsbaum mit *Prg* an der Wurzel hergestellt wurde.

Die Grammatik in Abb. 1 gehört zu einer Klasse von Grammatiken, den sogenannten LL($k$)-Grammatiken, aus denen sich besonders einfach Parser bauen lassen. Hier ist $k = 5$, Erklärung folgt. Ein LL($k$)-Parser läßt sich folgendermaßen konstruieren.

Zuerst einmal fassen wir die Tokenliste der Eingabe als Liste von Ableitungsbäumen auf. Jeder Ableitungsbaum darin besteht zu Anfang lediglich aus einem einzigen Knoten, nämlich dem entsprechenden Token. Beachte, dass es die Aufgabe ist, diese Bäume auf geeignete Weise zu einem einzigen Baum zusammenzufügen, der ein gültiger Ableitungsbaum bezüglich der Grammatik in Abb. 1 ist.

Dazu ordnet man jedem Nichtterminalsymbol der Grammatik, also z.B. *PrgAtom*, eine Funktion zu, die als Eingabe eine Liste von Ableitungsbäumen – also z.B. auch die ursprüngliche Tokenliste – nimmt, und die ersten darin zu einem neuen Ableitungsbaum zusammenfasst, an dessen Wurzel das entsprechende Nichtterminalsymbol steht. So könnte die Funktion für *PrgAtom*, angesetzt auf die Tokenliste am Ende des vorherigen Abschnitts – jetzt aufgefasst als Liste von Ableitungsbäumen der Höhe 1 – daraus z.B. die Liste

$$t, \mathit{Symbol}(\texttt{";"}), \mathit{Keyword}(\texttt{"while"}), \mathit{Keyword}(\texttt{"true"}), \mathit{Keyword}(\texttt{"do"}), \ldots$$

machen, wobei $t$ der in Abb. 2 dargestellte Ableitungsbaum ist.

Es sollte klar sein, dass solch ein Zusammenfügen von Ableitungsbäumen, die dann alle Teilbäume direkt unterhalb einer neuen Wurzel werden, von einer Regel in der Grammatik beschrieben werden muss. Ansonsten ist nicht klar, wieso der am Ende resultierende Baum ein Ableitungsbaum bezüglich dieser Grammatik sein soll. Beachte, dass sich die Frage, welches Nichtterminalsymbol neue Wurzel werden soll, nicht stellt: *Jedem* Nichtterminalsymbol wird ja eine Funktion zugeordnet, welche Ableitungsbäume mit ihr an der Wurzel erzeugt. Es stellt sich aber bei jeder dieser Funktionen berechtigterweise die Frage, welche *Regel* dazu verwendet werden soll, die nächsten Ableitungsbäume in der noch vorliegenden Liste zu einem zusammenzubauen.


Abbildung 2: Ableitungsbaum $t$.

An dieser Stelle kommt die Tatsache ins Spiel, dass es sich bei der Grammatik in Abb. 1 um eine LL(5)-Grammatik handelt. Dies bedeutet, dass jede dieser Funktionen lediglich die Wurzel der *ersten fünf* Ableitungsbäume in der Liste, die ihr vorliegt, inspizieren muss, um die eindeutige Regel auszuwählen, mit der der Rest geparst wird. Beachte, dass in den allermeisten Fällen bereits die erste Wurzel eindeutig festlegt, welche Regel dies ist. Nur bei *IExpAtom* muss man eventuell die ersten 5 anschauen.

Im vorliegenden Beispiel war es so, dass der erste Ableitungsbaum in der vorliegenden Liste als Wurzel *Identifier* hatte. Somit konnte nur die 8. Regel für *PrgAtom* zum Erfolg führen. Diese besagt, dass das nächste Token ein *Identifier* sein muss, gefolgt von einem Token, welches ein *Symbol* mit Wert `"="` ist, gefolgt von einer Liste von Token, die zu einem *IExp* geparst werden können. Die Funktion zum Parsen eines *PrgAtom* kann also in solch einem Fall die beiden ersten Token entsprechend überprüfen und die Funktion zum Parsen eines *IExp* auf der restlichen Liste aufrufen.

Insgesamt entsteht so eine Menge von rekursiven Funktionen, von denen diejenige für *Prg* zum Parsen der Gesamtliste verwendet wird. Das vorliegende Programm ist dann erfolgreich geparst, wenn diese Funktion einen Ableitungsbaum aus der gesamten Liste konstruiert, d.h. alle Tokens darin konsumiert und in den Baum einbaut. Eine beispielhafte Implementation eines LL(1)-Parsers mit entsprechendem Lexer für eine einfache Sprache arithmetischer Ausdrücke ist auf der WWW-Seite des Praktikums erhältlich.

Natürlich möchten wir nicht nur erkennen, ob ein Eingabestring ein gültiges Programm ist, sondern wir wollen das Programm auch ausführen. Da die Ausführung eines Programms von seiner Struktur abhängt – arithmetische Ausdrücke werden zum Beispiel zu einem Integer-Wert ausgerechnet, während Konstrukte wie ein `if` den Kontrollfluss steuern – ist es notwendig, die einzelnen Programmteile bezüglich ihres Typs zu unterscheiden. Dazu eignet sich im Prinzip der vom Parser hergestellte Ableitungsbaum. Schaut man sich diese jedoch genauer an, so stellt man fest, dass sie viel redundante Information enthalten. Dies wird bereits in Abb. 2 deutlich. Als zweites Beispiel ist der Ableitungsbaum für den arithmetischen Ausdruck (*IExp*) `"3 * (2 + 4)"` in Abb. 3 angegeben. Dieser enthält unnötige Information, die eigentlich nur zum Erkennen des Eingabestrings nötig war, z.B. dass Punktrechnung vor Strichrechnung geht. Dies muss zwar in der Auswertung des arithmetischen Ausdrucks ebenfalls berücksichtigt werden, braucht jedoch nicht die Unterscheidung durch verschiedene Nichtterminalsymbole, sondern kann einfach in der Struktur des Baums kodiert werden (Operator `"+"` unterhalb des Operators `"*"` in dieser Fall).

Wir beschreiben daher als nächstes eine einfachere Möglichkeit, Programme zu repräsentieren. Dies ist auch unter dem Begriff *abstrakte Syntax* be-

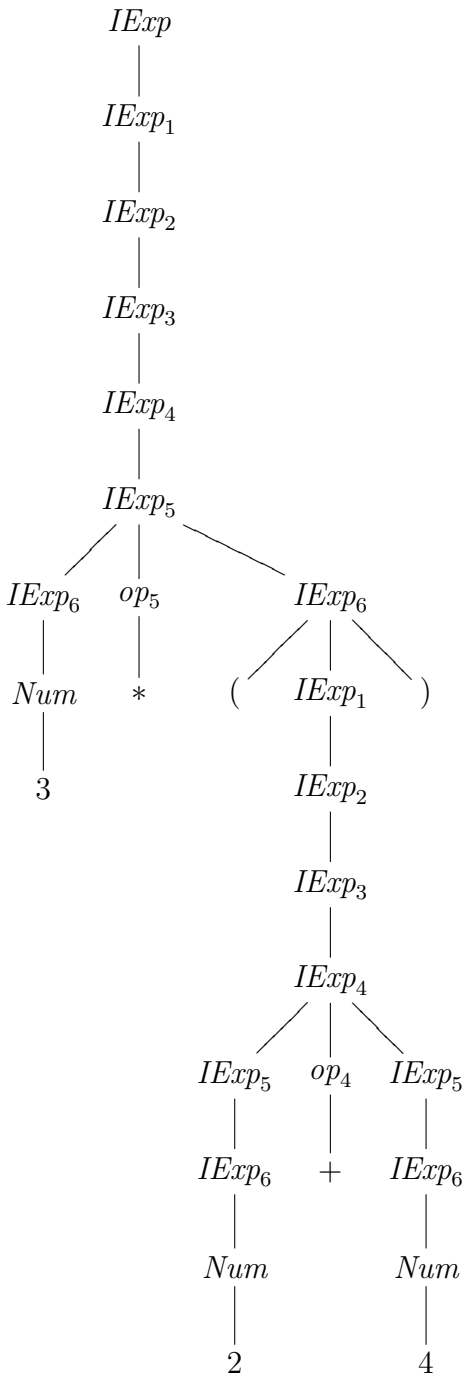


Abbildung 3: Ableitungsbaum für 3 * (2 + 4) in der konkreten Syntax.

kannt, im Gegensatz zu der *konkreten Syntax* aus Abb. 1. Zur abstrakten Syntax kann man prinzipiell auf zweierlei Wegen kommen. Entweder man programmiert den Parser so, dass er zuerst einen Ableitungsbaum bezüglich der konkreten Syntax baut und vereinfacht diesen dann anschließend, sodass er der abstrakten Syntax entspricht. Oder man konstuiert den Parser so, dass er gleich Ableitungsbäume bezüglich der abstrakten Syntax herstellt. Theoretisch gibt es auch noch die Möglichkeit, einfach bei der Auswertung der Programme mit der konkreten Syntax zu arbeiten. Dies führt aber zu einem unnötigen Mehraufwand, sowohl beim Programmieren des Interpreters (z.B. werden arithmetische Ausdrücke formal unterschieden in $IExp$,$IExp_1$,..., $IExp_6$, ausgewertet werden diese jedoch alle auf dieselbe Art und Weise) als auch beim Speicherverbrauch im laufenden Betrieb (Ableitungsbäume in der abstrakten Syntax sind kleiner).

# 4 Repräsentierung von Programmen

Eine Klassenhierarchie zur Repräsentierung von AntBrain-Programmen in Java ist in den Klassendiagrammen in Abbildungen 5 und 4 angegeben. Mit diese Klassen werden Programme so repräsentiert, dass man einfach mit ihnen arbeiten kann, ohne sich mit syntaktischen Details wie Operatorpräzedenzregeln beschäftigen zu müssen.

Der Ausdruck `"3*(2+4)"` zum Beispiel ist durch das Objektdiagramm in Abb. 6 repräsentiert. Beachte, dass in diesem Objektdiagramm keine Klammern oder Präzedenzregeln mehr nötig sind, im Gegensatz zum Ableitungsbaum in Abb. 3.

Die abstrakte Klasse Prg ist der Typ von Programmen. Durch ihre Unterklassen sind die verschiedenen Möglichkeiten für Programme gegeben.

**PrgAction** Die Instruktionen für Ameisenaktionen `"skip"`, `"walk"`, `"pickup"`, `"drop"`, `"leavescent(i)"`, `"clearscent(i)"` und `"turn(i)"` werden als Objekte der Klasse PrgAction repräsentiert. Der Typ der Instruktion ist im Feld action gespeichert. Manche Aktionen haben einen Parameter `"i"`, der ein arithmetischer Ausdruck ist. Dieser ist dann im Feld actionParameter gespeichert.

**PrgAssign** Zuweisungen `"x = t"` werden durch Objekte vom Typ PrgAssign gespeichert.

**PrgComp** Die Komposition `"P1; P2"` von zwei Programmen ist durch PrgComp gegeben. Dabei speichert das Feld p1 das erste Programm und das Feld p2 das zweite Programm.

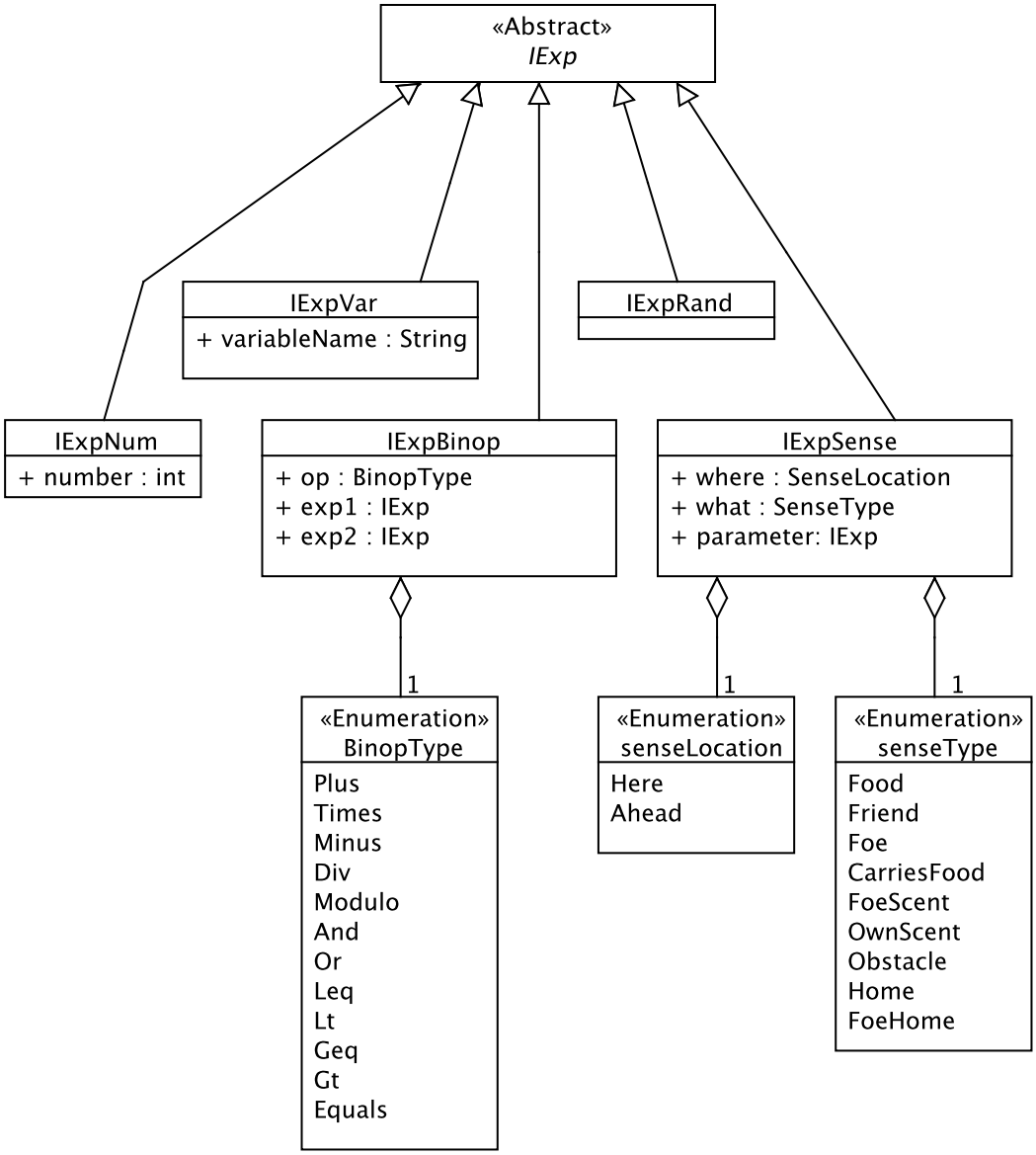


Abbildung 4: Klassen für die Repräsentierung von arithmetischen Ausdrücken

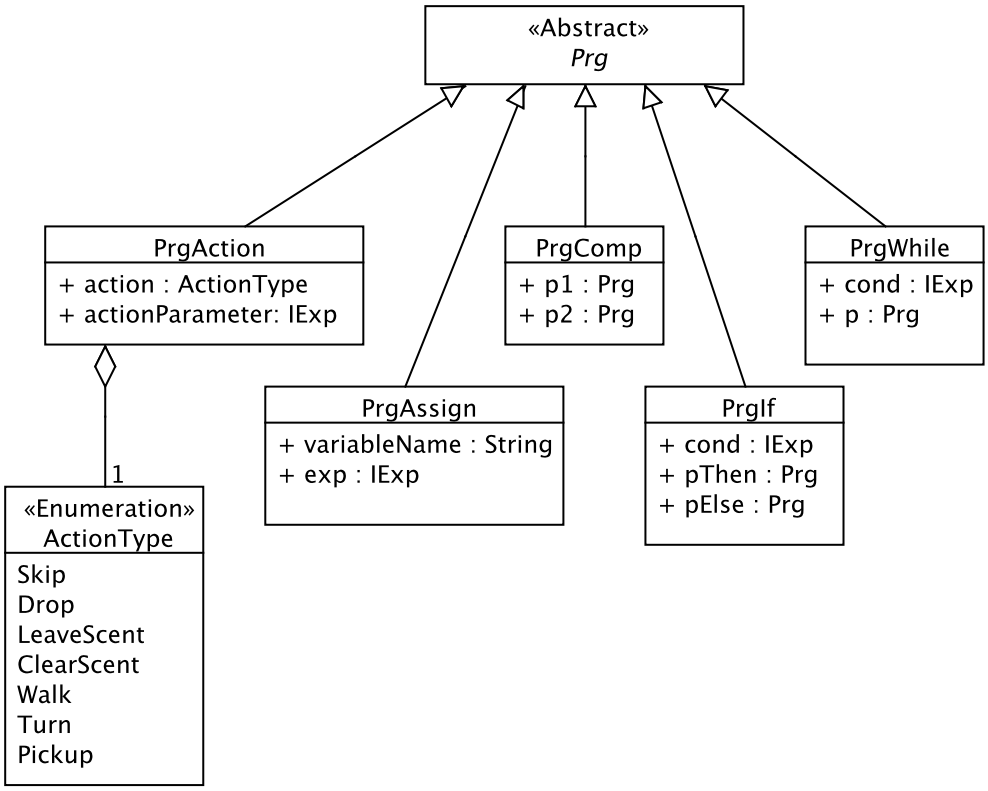


Abbildung 5: Klassen für die Repräsentierung von Programmen

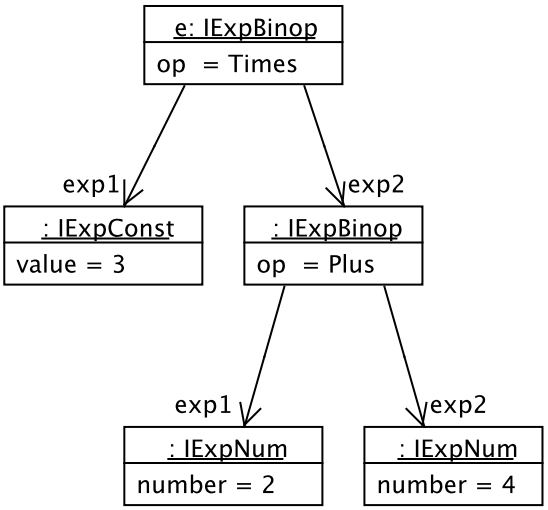


Abbildung 6: Repräsentierung des Ausdrucks 3 * (2+4)

**PrgIf, PrgWhile** If-Tests und while-Schleifen sind analog zur Komposition repräsentiert.

Die arithmetischen Ausdrücke, die in den Programmen vorkommen, sind analog zu den Programmen als Objekte vom Typ IExp repräsentiert. Bei den arithmetischen Ausdrücken gibt es zwei Punkte zu beachten:

1. Es gibt in der Klassenhierarchie keine Entsprechung für die Ausdrücke `"true"`, `"false"`, `"-t"` und `"!t"`, die in der Grammatik vorkommen. Der Ausdruck `"true"` ist eine Abkürzung für die Zahl 1 und `"false"` ist eine Abkürzung für die Zahl 0. Der Ausdruck `"-t"` kürzt `"0-t"` ab und `"!t"` schlussendlich ist eine Abkürzung für `"t==0"`.

2. Verknüpfungen von mehreren Binäroperatoren gleicher Präzedenz (Operatoren haben die gleiche Präzedenz, falls sie in der gleichen $op_i$-Regel vorkommen) sollen als nach links geklammert betrachtet werden. Das bedeutet, dass zum Beispiel $1 + 2 + 3$ wie $(1 + 2) + 3$ gespeichert wird und $2 * 2/3$ wie $(2 * 2)/3$ gespeichert wird. Beachte, dass die richtige Klammerung wichtig ist: $(2 * 2)/3$ wertet zu 1 aus, während $2 * (2/3)$ zu 0 auswertet.

   In der Grammatik ist ein Nichtterminal $IExp_i$ durch eine Folge von $IExp_{i+1}\, op_i \ldots op_i IExp_{i+1}$ gegeben. Beim Einlesen einer solchen Folge sollte man die Operationen also nach links klammern.[1] Wie man das machen kann, ist im Parserbeispiel auf der Praktikumshomepage angegeben (Funktionen parseExp1 und parseExp2).

Im nächsten Abschnitt werden wir beschreiben, wie man die durch Objekte des Typs Prg repräsentierten Programme ausführen kann. Da wir dort nicht immer ein Objektdiagramm zeichnen wollen, wenn wir ein Objekt vom Typ Prg angeben wollen, werden wir eine informelle Notation für diese Objekte benutzen.

Wir benutzen eine Syntax, die der von AntBrain entspricht. Wir schreiben zum Beispiel **while** 1 **do** $(x = \textbf{rand}; \textbf{turn}(x); \textbf{walk})$ für ein Objekt p, dass das entsprechende AntBrain-Programm repräsentiert. Das heißt, p ist vom Typ PrgWhile und p.cond ist ein Objekt vom Typ IExpNum mit der Eigenschaft p.cond.number=1 und so weiter, d.h. p.p ist ein Objekt, das das Programm $x = \textbf{rand}; \textbf{turn}(x); \textbf{walk}$ auf analoge Weise repräsentiert.

[1] Beachte, dass die Klammerung in der Repräsentation der Regel nichts damit zu tun hat.

# 5 Funktionsweise der Ameisengehirne

Nun da wir AntBrain Programme in Java repräsentieren und auch einlesen können, müssen wir nur noch erklären, wie die Ameisen durch solche Programme gesteuert werden.

Zur Erinnerung: In jedem Schritt kann die Ameise zwei Zellen sehen: die Zelle, auf der sie gerade steht, und die Nachbarzelle, in deren Richtung sie ausgerichtet ist. In Abhängigkeit vom Aussehen dieser beider Zellen und ihrem Gehirnzustand macht die Ameise dann eine Aktion und nimmt einen neuen Gehirnzustand an.

Einfach gesagt ist ein Schritt der Ameise nichts Anderes als das Ausführen einer Instruktion in ihrem AntBrain-Programm. Sind die ersten Instruktionen im Programm zum Beispiel **walk**; $x = y + 1$; ..., so wird die Ameise im ersten Schritt die Instruktion **walk** ausführen. Das Ausführen dieser Instruktion führt zur Aktion **walk**, d.h. die Ameise macht einen Schritt nach vorn. Im zweiten Schritt führt die Ameise die Instruktion $x = y + 1$ aus. Dadurch wird die Variable $x$ im Gehirn der Ameise auf den Wert von $y$ plus 1 (modulo 6) gesetzt. Die Instruktion $x = y + 1$ führt zur Aktion **skip**, d.h. die Ameise braucht diesen Schritt ganz für ihren Denkaufwand und bewegt sich nicht.

Aus dieser Beschreibung sollte klar werden, dass der Gehirnzustand der Ameisen zum einen die als nächstes auszuführende Instruktion und zum anderen die Werte von Programmvariablen umfassen muss. Als nächstes definieren wir, auf welche Art sich die Ameisen diese Informationen merken können.

Der Gehirnzustand einer Ameise besteht aus einem Gedächtnis, in dem sich die Ameise die Werte von Variablen merkt, und einem Programmstack, in dem sich die Ameise merkt, welche Programmteile in welcher Reihenfolge noch ausgeführt werden müssen.

**Gedächtnis** Das Gedächtnis einer Ameise ist eine endliche partielle Funktion von Variablen in die Menge $\{0, 1, 2, 3, 4, 5\}$. Wir bezeichnen die Menge aller solcher Funktionen mit *Memory* und schreiben $\perp$ für die Funktion, die auf allen Argumenten undefiniert ist. Ist $x$ eine Variable und $n \in \{0, 1, 2, 3, 4, 5\}$, so schreiben wir $\rho[n/x]$ für die partielle Funktion, die für alle Variablen $y$ folgender Gleichung genügt:

$$\rho[n/x](y) = \begin{cases} n & \text{wenn } x = y \\ \rho(y) & \text{sonst} \end{cases}$$

*Hinweis:* Wer mit den endlichen partiellen Funktionen $\rho$ Schwierigkeiten

hat, kann sie sich auch als Werte `Map<String, Integer> memory` vorstellen. Der Wert `memory` repräsentiert eine endliche Abbildung von Variablennamen (`String`) auf ihre Zahlwerte (`Integer`). Man kann den Wert der Variablen `"x"` erfragen, indem man `memory.get("x")` ausführt. Das entspricht der Funktionsanwendung $\rho(x)$. Wenn die partielle Funktion $\rho$ der Abbildung `memory` entspricht, dann entspricht die Funktion $\rho[n/x]$ der Abbildung, die man mittels `memory.update("x", n)` erhält.

**Programmstack** Neben einem Gedächtnis hat jede Ameise einen Stack von Programmen (= Werte von Typ *Prg*), der ihre nächsten Denkschritte bestimmt. Das Programm ganz oben auf dem Stack soll als nächstes ausgeführt werden, dann kommt das Programm darunter, usw. Wir schreiben *Empty* für den leeren Stack. Ist $P$ ein Programm und $E$ ein Stack von Programmen, dann schreiben wir $P{:}E$ für den Stack, der entsteht, wenn man $P$ oben auf den Stack $E$ legt.

**Gehirnzustand** Der Gehirnzustand jeder Ameise besteht aus einem Gedächtnis $\rho$ und einem Programmstack $E$. Wir schreiben *Brain* für die Menge aller solcher Paare.

Eine neue Ameise mit Programm $P$ hat den Zustand $(\bot, P\colon \mathit{Empty})$. Das heißt, ihr Gedächtnis ist noch leer und als nächstes möchte sie gerne das Programm $P$ ausführen. Danach ist sie fertig, braucht also keine weiteren Programme nach $P$ auszuführen.

## 5.1 Schrittfunktion

Wir definieren nun die Schrittfunktion

$$\mathit{step}\colon \mathit{Brain} \times (\mathit{Cell} \times \mathit{Cell}) \longrightarrow \mathit{Action} \times \mathit{Brain},$$

die das Verhalten der Ameisen in einem Schritt angibt. Intuitiv sind die Argumente an diese Funktion der momentane Gehirnzustand einer Ameise, also noch auszuführende Programme und Werte für die Variablen, sowie die beiden Zellen unter und vor ihr. Die Funktion berechnet daraus eine Aktion, z.B. die Ameise sich drehen zu lassen, sowie einen neuen Gehirnzustand. Dieser entsteht z.B. dadurch, dass Variablen neue Werte erhalten.

Zunächst erklären wir, wie die Ameise die arithmetischen Ausdrücke in ihrem Programm auswertet. Will sie z.B. die Instruktion **if** $t$ **then** $P$ **else** $Q$ ausführen, so muss sie den arithmetischen Ausdruck $t$ auswerten, um zu sehen, ob die nächste auszuführende Instruktion in $P$ oder in $Q$ ist.

### 5.1.1 Auswertung arithmetischer Ausdrücke

Der Wert eines arithmetischen Ausdrucks kann sowohl vom Gedächtnis der Ameise, als auch von den beiden Zellen, welche die Ameise sehen kann, abhängen. Um zum Beispiel $x * \textbf{sense}(\textbf{here}, \textbf{food})$ auszuwerten, muss die Ameise den Wert von $x$ in ihrem Gedächtnis nachsehen und sie muss schauen, wie viele Futterstücke auf der Zelle liegen, auf der sie gerade steht (der Term $\textbf{sense}(\textbf{here}, \textbf{food})$ steht für die Anzahl der Futterstücke auf dieser Zelle).

Der Wert eines arithmetischen Ausdrucks $t$ ist bezüglich einer Gedächtnisfunktion $\rho \in \mathit{Memory}$ und zwei Zellen **here** und **ahead** definiert. Die Zelle **here** wird dabei für die Zelle stehen, auf der sich die Ameise gerade befindet, und die Zelle **ahead** steht für deren Nachbarzelle in der Richtung, in die die Ameise schaut. Wir schreiben kurz $\sigma$ für das Paar von Zellen $(\textbf{here}, \textbf{ahead})$. Der Wert eines arithmetischen Ausdrucks $t$, in Abhängigkeit von dem Zellenpaar $\sigma$ und der Gedächtnisfunktion $\rho$, wird als $\|t\|^\sigma_\rho$ notiert. Dies ist dann entweder eine Zahl in $\{0, 1, 2, 3, 4, 5\}$ oder undefiniert (das kann passieren, wenn z.B. in $t$ eine Variable benutzt wird, die in $\rho$ nicht definiert ist). Der Wert von $\|t\|^\sigma_\rho$ berechnet sich wie folgt:

**Variablen** Ist der Term $t$ eine Variable $x$, so erhält man seinen Wert, indem man den Wert der Variable im Gedächtnis nachschlägt:

$$\|x\|^\sigma_\rho = \rho(x)$$

Beachte, dass $\rho(x)$ (und damit auch $\|x\|^\sigma_\rho$) undefiniert sein kann.

**Zahlen** Ist der Term $t$ eine Zahl $n$, so ist seine Interpretation gegeben durch:

$$\|n\|^\sigma_\rho = n \bmod 6$$

**Binäre Operationen** Als nächstes definieren wir die Bedeutung von Ausdrücken, die durch eine binäre Operation gegeben sind.

Zunächst fordern wir, dass $\|t_1 \; op \; t_2\|^\sigma_\rho$ undefiniert ist, wenn $\|t_1\|^\sigma_\rho$ oder $\|t_2\|^\sigma_\rho$ undefiniert ist. Dabei ist $op$ eine beliebige binäre Operation.

Nehmen wir also an, dass $\|t_1\|^\sigma_\rho$ und $\|t_2\|^\sigma_\rho$ beide definiert sind.

Die Bedeutung der Vergleichsoperatoren ist dann folgendermaßen defi-

niert:

$$\|t_1 \;==\; t_2\|_\rho^\sigma = \begin{cases} 1 & \text{wenn } \|t_1\|_\rho^\sigma = \|t_2\|_\rho^\sigma \\ 0 & \text{sonst} \end{cases}$$

$$\|t_1 \;<\; t_2\|_\rho^\sigma = \begin{cases} 1 & \text{wenn } \|t_1\|_\rho^\sigma < \|t_2\|_\rho^\sigma \\ 0 & \text{sonst} \end{cases}$$

$$\|t_1 \;\leq\; t_2\|_\rho^\sigma = \begin{cases} 1 & \text{wenn } \|t_1\|_\rho^\sigma \leq \|t_2\|_\rho^\sigma \\ 0 & \text{sonst} \end{cases}$$

$$\|t_1 \;>\; t_2\|_\rho^\sigma = \begin{cases} 1 & \text{wenn } \|t_1\|_\rho^\sigma > \|t_2\|_\rho^\sigma \\ 0 & \text{sonst} \end{cases}$$

$$\|t_1 \;\geq\; t_2\|_\rho^\sigma = \begin{cases} 1 & \text{wenn } \|t_1\|_\rho^\sigma \geq \|t_2\|_\rho^\sigma \\ 0 & \text{sonst} \end{cases}$$

Die logischen Operatoren || und && sind definiert durch:

$$\|t_1 \;||\; t_2\|_\rho^\sigma = \begin{cases} 1 & \text{wenn } \|t_1\|_\rho^\sigma > 0 \text{ oder } \|t_2\|_\rho^\sigma > 0 \\ 0 & \text{sonst} \end{cases}$$

$$\|t_1 \;\&\&\; t_2\|_\rho^\sigma = \begin{cases} 1 & \text{wenn } \|t_1\|_\rho^\sigma > 0 \text{ und } \|t_2\|_\rho^\sigma > 0 \\ 0 & \text{sonst} \end{cases}$$

Es bleibt nur noch, die arithmetischen Operationen zu interpretieren:

$$\begin{aligned} \|t_1 + t_2\|_\rho^\sigma &= (\|t_1\|_\rho^\sigma + \|t_2\|_\rho^\sigma) \bmod 6 \\ \|t_1 - t_2\|_\rho^\sigma &= (\|t_1\|_\rho^\sigma - \|t_2\|_\rho^\sigma) \bmod 6 \\ \|t_1 * t_2\|_\rho^\sigma &= (\|t_1\|_\rho^\sigma \cdot \|t_2\|_\rho^\sigma) \bmod 6 \\ \|t_1/t_2\|_\rho^\sigma &= \lfloor \|t_1\|_\rho^\sigma / \|t_2\|_\rho^\sigma \rfloor \bmod 6 \\ \|t_1 \;\%\; t_2\|_\rho^\sigma &= \|t_1\|_\rho^\sigma \bmod \|t_2\|_\rho^\sigma \end{aligned}$$

*Hinweis:* Die mathematische Definition von $x \bmod y$ ist $x - \lfloor x/y \rfloor \cdot y$. Das bedeutet insbesondere $-1 \bmod 6 = 5$. Das ist nicht das Gleiche wie der Rest-Operator `%` in Java, welcher durch $x = (x/y) + (x\%y)$ definiert ist! Zum Beispiel gilt $(-1)\%6 = -1$. Die mathematische Version ($x \bmod y$) kann in Java zum Beispiel durch `((x % y) + y) % y` implementiert werden.

**Spezielle Operationen** Es bleibt nur noch, die Bedeutung der speziellen Schlüsselwörter **rand** und **sense** zu erklären. Diese sind folgendermaßen gegeben.

$$\|\mathbf{rand}\|_\rho^\sigma = \text{eine Zufallszahl aus der Gleichverteilung über } \{0,\ldots,5\}$$

$$\|\mathbf{sense}(\mathbf{here},\mathbf{food})\|_\rho^\sigma = \min(5, \text{Anzahl der Futterstücke in Zelle } \mathbf{here})$$

$$\|\mathbf{sense}(\mathbf{here},\mathbf{obstacle})\|_\rho^\sigma = \begin{cases} 1 & \text{wenn Zelle } \mathbf{here} \text{ in } \sigma \text{ ein Hindernis enthält} \\ 0 & \text{sonst} \end{cases}$$

$$\|\mathbf{sense}(\mathbf{here},\mathbf{friend})\|_\rho^\sigma = \begin{cases} 1 & \text{wenn Zelle } \mathbf{here} \text{ in } \sigma \text{ eine Ameise desselben Stammes enthält} \\ 0 & \text{sonst} \end{cases}$$

$$\|\mathbf{sense}(\mathbf{here},\mathbf{foe})\|_\rho^\sigma = \begin{cases} 1 & \text{wenn Zelle } \mathbf{here} \text{ in } \sigma \text{ eine Ameise von einem anderen Stamm enthält} \\ 0 & \text{sonst} \end{cases}$$

$$\|\mathbf{sense}(\mathbf{here},\mathbf{home})\|_\rho^\sigma = \begin{cases} 1 & \text{wenn Zelle } \mathbf{here} \text{ in } \sigma \text{ zum Ameisenhügel der Ameise gehört} \\ 0 & \text{sonst} \end{cases}$$

$$\|\mathbf{sense}(\mathbf{here},\mathbf{foehome})\|_\rho^\sigma = \begin{cases} 1 & \text{wenn Zelle } \mathbf{here} \text{ in } \sigma \text{ zum Ameisenhügel eines fremden Ameisenstammes gehört} \\ 0 & \text{sonst} \end{cases}$$

$$\|\mathbf{sense}(\mathbf{here},\mathbf{carriesfood})\|_\rho^\sigma = \begin{cases} 1 & \text{wenn Zelle } \mathbf{here} \text{ in } \sigma \text{ eine Ameise enthält, die Futter trägt} \\ 0 & \text{sonst} \end{cases}$$

$$\|\mathbf{sense}(\mathbf{here},\mathbf{foescent})\|_\rho^\sigma = \begin{cases} 1 & \text{wenn Zelle } \mathbf{here} \text{ in } \sigma \text{ eine beliebige Duftmarke eines beliebigen anderen Ameisenstammes enthält} \\ 0 & \text{sonst} \end{cases}$$

$$\|\mathbf{sense}(\mathbf{here}, \mathbf{ownscent}, t)\|_\rho^\sigma = \begin{cases} \text{undefiniert} & \text{wenn } \|t\|_\rho^\sigma \text{ undefiniert} \\ 1 & \text{wenn Zelle } \mathbf{here} \text{ in } \sigma \text{ die} \\ & \text{Duftmarke Nummer } \|t\|_\rho^\sigma \\ & \text{des eigenen} \\ & \text{Ameisenstammes enthält} \\ 0 & \text{sonst} \end{cases}$$

Die Fälle für $\mathbf{sense}(\mathbf{ahead}, \dots)$ erhält man, indem man in den obigen Fällen überall **here** durch **ahead** ersetzt.

Beachte, dass manche Argumentkombinationen für **sense** nicht sehr sinnvoll sind. Zum Beispiel wird $\mathbf{sense}(\mathbf{here}, \mathbf{friend})$ immer gleich 1 sein. Mit $\mathbf{sense}(\mathbf{ahead}, \mathbf{friend})$ erhält man jedoch sinnvolle Informationen.

### 5.1.2 Schrittfunktion

Wir haben nun alles zusammen, um die Schrittfunktion *step* zu definieren. Wir definieren den Wert von $step((\rho, E), \sigma)$ durch Fallunterscheidung über den Stack $E$:

- Fall $E \equiv Empty$. Dann $step((\rho, E), \sigma) = (\mathbf{skip}, (\rho, E))$.
- Fall $E \equiv P{:}F$.
  - Unterfall $P \equiv \mathbf{skip}$. Dann $step((\rho, E), \sigma) = (\mathbf{skip}, (\rho, F))$.
  - Unterfall $P \equiv (x = t)$. Dann

$$step((\rho, E), \sigma) = \begin{cases} (\mathbf{skip}, (\rho[\|t\|_\rho^\sigma / x], F)) & \text{wenn } \|t\|_\rho^\sigma \text{ definiert} \\ (\mathbf{skip}, (\bot, Empty)) & \text{wenn } \|t\|_\rho^\sigma \text{ undefiniert} \end{cases}$$

  - Unterfall $P \equiv \mathbf{if}\ t\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2$. Dann

$$step((\rho, E), \sigma) = \begin{cases} step((\rho, P_1{:}F), \sigma) & \text{wenn } \|t\|_\rho^\sigma > 0 \\ step((\rho, P_2{:}F), \sigma) & \text{wenn } \|t\|_\rho^\sigma = 0 \\ (\mathbf{skip}, (\bot, Empty)) & \text{wenn } \|t\|_\rho^\sigma \text{ undefiniert} \end{cases}$$

  - Unterfall $P \equiv \mathbf{while}\ t\ \mathbf{do}\ P'$. Dann

$$step((\rho, E), \sigma) = \begin{cases} step((\rho,\ P'{:}P{:}F), \sigma) & \text{wenn } \|t\|_\rho^\sigma > 0 \\ (\mathbf{skip}, (\rho, F)) & \text{wenn } \|t\|_\rho^\sigma = 0 \\ (\mathbf{skip}, (\bot, Empty)) & \text{wenn } \|t\|_\rho^\sigma \text{ undefiniert} \end{cases}$$

– Unterfall $P \equiv P_1; P_2$. Dann $\mathit{step}((\rho, E), \sigma) = \mathit{step}((\rho, P_1{:}P_2{:}F), \sigma)$.

– Unterfall $P \equiv \mathbf{walk}$. Dann $\mathit{step}((\rho, E), \sigma) = (\mathbf{walk}, (\rho, F))$.

– Unterfall $P \equiv \mathbf{turn}(t)$. Dann

$$\mathit{step}((\rho, E), \sigma) = \begin{cases} ((\mathbf{turn}, \|t\|_\rho^\sigma), (\rho, F)) & \text{wenn } \|t\|_\rho^\sigma \text{ definiert} \\ (\mathbf{skip}, (\bot, \mathit{Empty})) & \text{wenn } \|t\|_\rho^\sigma \text{ undefiniert} \end{cases}$$

– Die noch verbleibenden Fälle, in denen $P$ entweder **pickup**, **drop**, **leavescent**$(t)$ oder **clearscent**$(t)$ ist, sind analog zu den letzten beiden Fällen definiert.

Um nun eine Ameise auf einem Spielbrett laufen zu lassen, muss deren Programm – durch Lexer und Parser in die Form der abstrakten Syntax laut Abschnitt 4 gebracht – lediglich durch sukzessives Anwenden der Schrittfunktion ausgeführt werden. Dazu braucht man z.B. eine Funktion, die arithmetische Ausdrücke – gegeben als Objekte vom Typ `IExp` – auswerten, was wiederum eine Datenstruktur verlangt, die Variablenbelegungen speichert. Wie oben bereits genannt, eignet sich dafür z.B. ein Objekt vom Typ `Map<String, Integer>`. Mit solchen Hilfsmitteln kann dann die Funktion *step*, so wie sie hier präsentiert ist, in Java realisiert werden.