Aufgabenbeschreibung
Softwareentwicklungspraktikum

Wintersemester 2008/2009
Universität München
Martin Lange und Ulrich Schöpp

Teil I – Spielverlauf
Version 2

1 Einleitung

In diesem Praktikum wollen wir die Ameisenart *formica artificiosa* entwickeln.

Ameisen dieser Art leben auf einem zweidimensionalen Feld mit sechseckigen Zellen. Sie verhalten sich völlig autonom, wissen was ihre Aufgabe in der Ameisengesellschaft ist und gehen dieser fleißig nach. Wie üblich sind die Ameisen in verschiedenen Ameisenstämmen organisiert, so dass sie bei der Futtersuche nicht auf sich allein gestellt sind.

Damit sich der Zusammenschluss zu einem Ameisenstamm für die Ameisen auch lohnt, muss dieser natürlich vernünftig organisiert sein. Ein schlecht organisierter Ameisenstamm wird nicht viel Futter abbekommen, da ihm dieses von den besser organisierten Stämmen längst weggeschnappt wurde. Bei dieser Aufgabe wollen wir den Ameisen in diesem Praktikum helfen, indem wir einzelnen Ameise sagen, wie sie sich verhalten sollen, damit ihr Ameisenstamm dann erfolgreich ist.

Die erste Aufgabe im Praktikum ist dann also, die Welt der formica artificiosa zu implementieren. Mit so einer Implementierung können wir dann leicht herausfinden, wie gut sich ein Ameisenstamm gegen einen anderen behaupten kann. Wir setzen dazu einfach mehrere Ameisenstämme auf ein Feld und überlassen es dann den Ameisen, das Futter zu finden und nach Hause zu bringen. Der Ameisenstamm, der nach einer bestimmten Zeit das meiste Futter gesammelt hat, wird dann zum Sieger erklärt.

Die zweite Aufgabe besteht dann darin, sich gut organisierte Ameisenstämme auszudenken. Einen Ameisenstamm anzugeben bedeutet einfach, dass man die Ameisen auf ihren Ameisenhügel setzt und das Gehirn jeder Ameise entsprechend „programmiert“. Die Ameisen können somit verschiedene Aufgaben übernehmen, wie zum Beispiel die Futtersuche oder die Bauverteidigung. Die Ameisengehirne können sich nicht ändern und müssen daher von Anfang an so gewählt werden, dass das Verhalten der Ameise dem Ziel des Ameisenstamms zuträglich ist.

Um erfolgreiche Ameisenstämme herzustellen, wird es nötig sein, sich Werkzeuge zum Entwurf von Ameisen auszudenken. Zum Beispiel wird man die Ameisen sicher in verschiedenen Situationen ausprobieren wollen, bevor man sie dann ins Rennen schickt.

2 Aufgabe

Die Praktikumsaufgabe ist, eine Anwendung zu schreiben, mit der Ameisenstämme gegeneinander antreten können und die es erlaubt, Ameisenstämme zu entwickeln. Dieses Anwendung soll durch netzwerkfähige Client/Server-Programme realisiert werden.

Ein Server hat die Aufgabe, Ameisenstämme von mehreren Mitspielern anzunehmen und diese gegeneinander antreten zu lassen. Die Mitspieler sollen dem Server ihre Ameisenstämme übermitteln. Diese werden vom Server gespeichert. Die Mitspieler können dann vom Server erfahren, wie sich ihr Ameisenstamm gegen die anderen eingereichten Stämme schlägt. Dazu berechnet der Server, was die Ameisen machen, wenn man verschiedene Ameisenstämme auf ein Feld setzt, und welcher Ameisenstamm nach einer bestimmten Anzahl von Schritten die meisten Futterstücke zurück zu seinem Hügel gebracht hat. Die Mitspieler können sich diesen Ablauf vom Server geben lassen und so das Verhalten der Ameisen nachverfolgen.

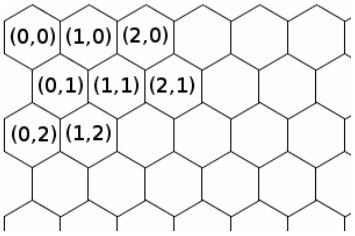
Alle Serverprogramme sollen ein gemeinsames Protokoll unterstützen (siehe Abschnitt 5), so dass alle Gruppen auf einem einzigen Server ihre Ameisenstämme gegeneinander antreten lassen können.

Neben dem Serverprogramm muss jede Gruppe natürlich auch noch eine Client-Anwendung schreiben. Mit dieser soll man Ameisenstämme entwerfen und dann beim Server einreichen können. Sie soll außerdem die Spielverläufe anzeigen können, die vom Server ausgerechnet werden und von ihm bereitgestellt werden.

Wie die Ameisenstämme entworfen werden, ist den Gruppen überlassen. Es bietet sich an, dazu ein Programm zu schreiben (evtl. als Teil des Clients), mit dem man Ameisen entwickeln und in verschiedenen Situationen ausprobieren kann.

3 Spielfeld

Das Spielfeld besteht aus einem Gitter hexagonaler Zellen in folgender Anordnung:



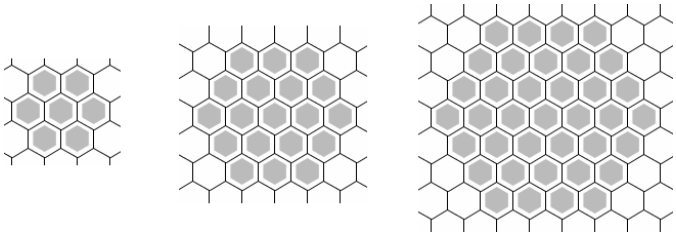
Es gibt verschiedene Spielfeldgrößen. Eine typische Größe ist 100 mal 100 Zellen.

Jede Zelle in dem hexagonalen Gitter kann folgende Eigenschaften haben:

- Sie kann ein Hindernis enthalten (vielleicht einen Stein oder Wasser, aber das ist nicht näher spezifiziert).
- Enthält sie kein Hindernis, so können alle folgenden Fälle eintreten.
  - Sie kann *eine* Ameise enthalten.
  - Sie kann eine beliebige Anzahl von Futterstücken enthalten.
  - Sie kann ein Teil des Ameisenhügels eines bestimmten Ameisenstammes sein.
  - Sie kann Duftspuren enthalten. Jeder Ameisenstamm hat sechs verschiedene Duftmarker, von dem jeder entweder in der Zelle vorhanden ist oder nicht. Für jeden Ameisenstamm sind die sechs verschiedenen Typen von Duftmarken von 0 bis 5 durchnummeriert.

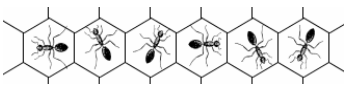
Jede Zelle am Rand des Spielfeldes enthält ein Hindernis. Eine Zelle liegt am Spielfeldrand, wenn sie weniger als sechs Nachbarn auf dem Spielfeld hat.

Man kann annehmen, dass Ameisenhügel „rund“ sind. Das heißt, dass die Ameisenhügel aus allen Zellen bestehen, die sich von der mittleren Zelle in eine bestimmten Höchstanzahl von Schritten erreichen lassen. Unten abgebildet sind Ameisenhügel vom Radius 1, 2 und 3. Ein typischer Ameisenhügel besteht aus 7 bis 50 Zellen.



Eine Ameise in einer Zelle hat folgende Eigenschaften:

- Jede Ameise gehört genau einem Ameisenstamm an.
- Jede Ameise ist in die Richtung einer der sechs Nachbarzellen ausgerichtet. Die Ausrichtung ist durch eine Zahl von 0 bis 5 gegeben. Das folgende Bild zeigt von links nach rechts die Bedeutung der Ausrichtungen 0, 1, 2, 3, 4 und 5.



- Eine Ameise kann ein Futterstück tragen.
- Jede Ameise hat ein Gehirn, das sich in einem bestimmten Zustand befindet. Wie die Gehirne genau funktionieren, wird in einem anderen Teil der Aufgabenbeschreibung spezifiziert. Dort wird dann die Programmiersprache AntBrain definiert, mit der man die Gehirne der Ameisen angeben kann.

Um das Verhalten der Ameisen zu beschreiben, ist es aber gar nicht nötig, die genaue Definition der Sprache AntBrain zu kennen. Die Ameisen verhalten sich folgendermaßen. In jedem Schritt kann die Ameise zwei Zellen sehen: die Zelle, auf der sie gerade steht, und die Nachbarzelle, in deren Richtung die Ameise ausgerichtet ist. In Abhängigkeit vom Aussehen dieser beider Zellen und ihrem Gehirnzustand macht die Ameise dann eine Aktion und nimmt einen neuen Gehirnzustand an. Dieses Verhalten werden wir dann durch eine Funktion

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

genau angeben. Dabei steht *Brain* für die Menge der möglichen Gehirnzustände der Ameise, *Cell* für die Menge aller Zellen und *Action* für die Menge aller möglichen Aktionen der Ameise.

Die Funktion *step* nimmt den momentanen Gehirnzustand und die zwei Zellen, die die Ameise sehen kann, als Eingabe und gibt eine Aktion sowie den neuen Gehirnzustand nach der Aktion aus.

Konkret ist die Menge der möglichen Aktionen jeder Ameise dabei gegeben durch:

$$\begin{aligned}\mathit{Action} = {} & \{\mathbf{skip}, \mathbf{walk}, \mathbf{pickup}, \mathbf{drop}\} \\ & \cup (\{\mathbf{turn}(i), \mathbf{leavescent}(i), \mathbf{clearscent}(i) \mid 0 \leq i \leq 5\})\end{aligned}$$

Was diese Aktionen zu bedeuten haben, wird im nächsten Abschnitt über den Spielablauf beschrieben.

4 Spielverlauf

4.1 Initialisierung

Die Teilnehmer reichen Ameisenstämme beim Server ein. So ein Ameisenstamm besteht einfach aus einer Anzahl von Ameisen, die dann später auf einem Ameisenhügel platziert werden. Für jede Ameise auf dem Ameisenhügel wird dabei ihre Ausrichtung gegeben, sowie ein Programm in der Sprache AntBrain, welches das Gehirn der Ameise definiert.

Das Spiel beginnt nun, indem die Ameisenstämme auf die Ameisenhügel des Spielfeldes gesetzt werden und ihre Gehirne mit den gegebenen AntBrain-Programmen in den Anfangszustand versetzt werden. Danach werden bis zu 1.000.000 Spielrunden ausgeführt (die genaue Zahl hängt vom Spielfeld ab) und der Ameisenstamm, auf dessen Hügel danach die meisten Futterstücke liegen, hat gewonnen.

4.2 Eine Runde

Eine Spielrunde besteht nun daraus, dass jede Ameise genau einen Schritt macht. Diese Schritte erfolgen nacheinander, d.h. zuerst macht eine Ameise einen Schritt, dann eine

andere und so weiter, bis alle genau einmal dran waren. Die Reihenfolge, in der die Ameisen in einer Spielrunde zum Zug kommen, ist dabei nicht vorgegeben. Sie darf von der Serverimplementierung abhängen.

4.3 Bewegung einer Ameise

Ein Schritt einer Ameise ist dabei folgendermaßen definiert. Sei **here** die Zelle, auf der die Ameise steht, und sei **ahead** die Nachbarzelle, in deren Richtung die Ameise schaut. Sei b der momentane Gehirnzustand der Ameise. Wir wenden dann die Funktion *step* auf die Argumente b, **here** und **ahead** an und erhalten ein Paar $(a, b') = \mathit{step}(b, \textbf{here}, \textbf{ahead})$ bestehend aus einem neuen Gehirnzustand b' und einer Aktion a. Die Ameise nimmt dann den neuen Gehirnzustand an und verhält sich folgendermaßen:

- $a = \textbf{skip}$. Die Ameise macht nichts.
- $a = \textbf{walk}$. Die Ameise geht auf die Zelle **ahead**, wenn diese nicht ein Hindernis ist oder dort bereits eine andere Ameise steht. Ansonsten bleibt sie stehen.
- $a = \textbf{pickup}$. Liegt in **here** ein Futterstück und trägt die Ameise nicht bereits eines, so hebt sie ein Futterstück auf. Ansonsten macht sie, wie bei **skip**, nichts.
- $a = \textbf{drop}$. Trägt die Ameise ein Futterstück, so lässt sie es in der Zelle **here** fallen. Ansonsten macht sie nichts.
- $a = \textbf{turn}(i)$. Die Ameise dreht sich um i Schritte nach rechts. Wenn also ihre Ausrichtung vorher d war, so ist sie nach diesem Schritt $(d + i) \bmod 6$.
- $a = \textbf{leavescent}(i)$. Die Ameise hinterlässt die Duftmarke vom Typ i und bewegt sich nicht.
- $a = \textbf{clearscent}(i)$. Die Ameise entfernt die Duftmarke vom Typ i und bewegt sich nicht. Ist die Duftmarke nicht vorhanden, so macht die Ameise nichts.

Ist nach einem solchen Schritt irgendeine Ameise von mindestens fünf fremden Ameisen umgeben, so stirbt sie und verschwindet.

5 Interfaces

5.1 Das Spielbrett in der Übertragung vom Server zum Client

Spielbretter müssen offensichtlich zwischen einem Spielserver und einem Client übertragen werden können. Dafür geben wir Interfaces vor, die auf der Homepage des Praktikums zu finden sind. Zu Details bzgl. der einzelnen Methoden in diesen Interfaces und ihrer intendierten Bedeutung verweisen wir auf die jeweiligen JavaDoc-Dokumentationen.

Bei den Interfaces, die Übertragungen von Objekten zwischen Server und Clients standardisieren, muss eine Implementierung natürlich evtl. auch noch `Remote` oder `Serializable` implementieren. Es bleibt jeweils den Gruppen überlassen, abzuwägen, was dabei sinnvoll ist.

5.1.1 Aufstellung des Ameisenstamms

Um einen Ameisenstamm auf einem bestimmten Spielbrett aufzustellen, lässt sich der Client vom Server ein Objekt vom Typ `Arena` geben. Dieses Objekt stellt Funktionen bereit, mit denen mehrere Mitspieler ihre Ameisenstämme auf einem bestimmten Spielfeld aufstellen können und diese gegeneinander antreten lassen können.

Mit dem Interface `Arena` kann ein Mitspieler nun zunächst erfragen, wie groß die Ameisenhaufen auf dem Spielfeld sind. Mit diesem Wissen kann der dann die Ameisen präparieren, die er einreichen will.

Das Aufstellen der Ameisen erfolgt dann durch Aufruf der Methode `submitTribe` auf einem Objekt vom Typ `Arena`. Diese Methode erwartet u.a. ein Array von Objekten vom Interface-Typ `AntClientToServer`. Solche Objekte erlauben es dem Server, Informationen über eine spezifische Ameise zu erfragen, insbesondere einen String, der den Anfangszustand des Gehirns der Ameise darstellt, und ihre anfängliche Ausrichtung, aber auch einen Namen, der möglichst eindeutig sein sollte.

Die Reihenfolge beim Aufstellen ist wie folgt. Die erste Ameise (also die an Index 0 in dem vom Client übergebenen `AntClientToServer[]`-Objekt steht in der Mitte des Ameisenhügels. Danach werden die Ameisen an den Indizes 1,...,6 auf den innersten Ring um den Mittelpunkt gestellt. Ameise 1 steht dabei links neben dem Mittelpunkt, die restlichen werden in ihrer natürlichen Reihenfolge im Uhrzeigersinn aufgestellt. Danach wird mit den Ameisen 7,...,18 in derselben Weise der nächstinnere Ring gefüllt, etc.

Hinweis: Die Gehirne von Ameisen werden durch Programme der Sprache AntBrain beschrieben. Diese Sprache wird in einem anderen Teil der Aufgabenbeschreibung spezifiziert. Aus diesem Grund kann die Methode `getBrain` momentan noch so implementiert werden, dass sie eine beliebige Zeichenkette zurückliefert, welche dann vom Server, der AntBrain-Programme ausführt, ignoriert wird.

Die Rückgabe der Ergebnisse vom Server zum Client wird so gehandhabt, dass der Client beim Server einen `ResultNotifier` hinterlässt. Ein Objekt eines solchen Typs wird dann vom Server verwendet, um den Client darüber zu informieren, ob seine Anfrage – z.B. nach einer berechneten Partie – erfolgreich war oder nicht. Dieses Interface ist generisch. Dadurch kann es vom Server benutzt werden, um auf eine Anfrage nach einer Partie gleich ein Objekt vom Typ `Match` zu übergeben, bzw. den Client über den Erfolg oder Misserfolg beim Aufstellen eines Stammes zu berichtigen, wo kein Objekt vom Typ `Match` benötigt wird bzw. vorhanden ist.

5.1.2 Übertragung einer ausgerechneten Partie

Die Übertragung einer vollständig berechneten Partie vom Server zum Client erfolgt mithilfe des Interfaces `BoardServerToClient`. Dies ist mit einer Klasse zu implementieren, welche es dem Client erlaubt, Abfragen zum Spielbrett, insbesondere zur Position von Hindernissen, Ameisen, Futter, etc. zu machen, um das Spielbrett dann grafisch anzuzeigen.

Die Implementierung von `BoardServerToClient` muss es natürlich auch dem Server gestatten, die Positionen der Hindernisse, Ameisen, etc. entsprechend anzugeben. Man beachte, dass diese Implementierung nicht identisch sein muss mit der internen Darstellung eines Spielbretts im Server, welches dieser zur Berechnung der Spiele verwendet. Insbesondere kann es sinnvoll sein, bei der Berechnung Datenstrukturen zu verwenden, die einen schnellen Zugriff erlauben, während – je nach dem – es bei der Übertragung des Spielbretts an einen Client sinnvoll sein kann, möglichst wenig Speicherplatz mit solchen Objekten zu verbrauchen.

5.2 Partien

Eine Partie wird über das Interface `Match` modelliert. Ein Objekt eines solchen Typs enthält die Startaufstellung als `BoardServerToClient` sowie Informationen wie die Anzahl der Runden in dieser Partie. Es erlaubt dem Client auch, anzufragen, welche Aktionen in welchen Runden durchgeführt wurden.

Die verschiedenen Typen der am Ende von Abschnitt 4 beschriebenen Aktionen werden hier im Aufzählungstyp `ActionKind` modelliert. Die Klasse `Action` verbindet dann solch eine Aktion mit einer Position auf dem Spielbrett.

Aufgabenbeschreibung
Softwareentwicklungspraktikum

Wintersemester 2008/2009
Universität München
Martin Lange und Ulrich Schöpp

Teil II – Architektur
Version 1

1 Allgemeine Architektur

Im ersten Teil der Aufgabenstellung wurde der Ablauf der Ameisenspiele beschrieben. In diesem Teil wollen wir nun die allgemeine Architektur des zu entwickelnden Programms festlegen.

Die Aufgabe beinhaltet die Entwicklung zweier unabhängiger Komponenten, einem Server und einem Client, die über eine RMI-Netzwerkverbindung miteinander kommunizieren können. Das Kommunikationsprotokoll ist dabei durch die Interfaces in dem Package `de.lmu.SEP0809.Communication` festgelegt, das von der Praktikumshomepage bezogen werden kann.

Dieses Package definiert Interfaces, die in diesem zweiten Teil der Aufgabenstellung näher beschrieben werden. Es werden weiterhin Anforderungen formuliert, welchen eine Implementierung dieser Interfaces durch Server und Client genügen muss.

1.1 Server

Der Server stellt über RMI einen Dienst zur Verfügung, mit dem sich mehrere Mitspieler mit ihren Client-Programmen verbinden können. Er erlaubt den Mitspielern, auf einer Anzahl von Spielbrettern ihre Ameisenstämme einzureichen und diese gegeneinander antreten zu lassen. Wie die Mitspieler das machen können ist durch die Schnittstellen in `de.lmu.SEP0809.Communication` geregelt, die der Server implementieren muss und die im Anschluss erklärt sind.

Der Server muss lediglich diese Schnittstellen implementieren, entsprechend Anforderungen in Abschnitt 4. Er muss keine graphische Benutzeroberfläche haben und kann lokal wie auch entfernt administriert werden.

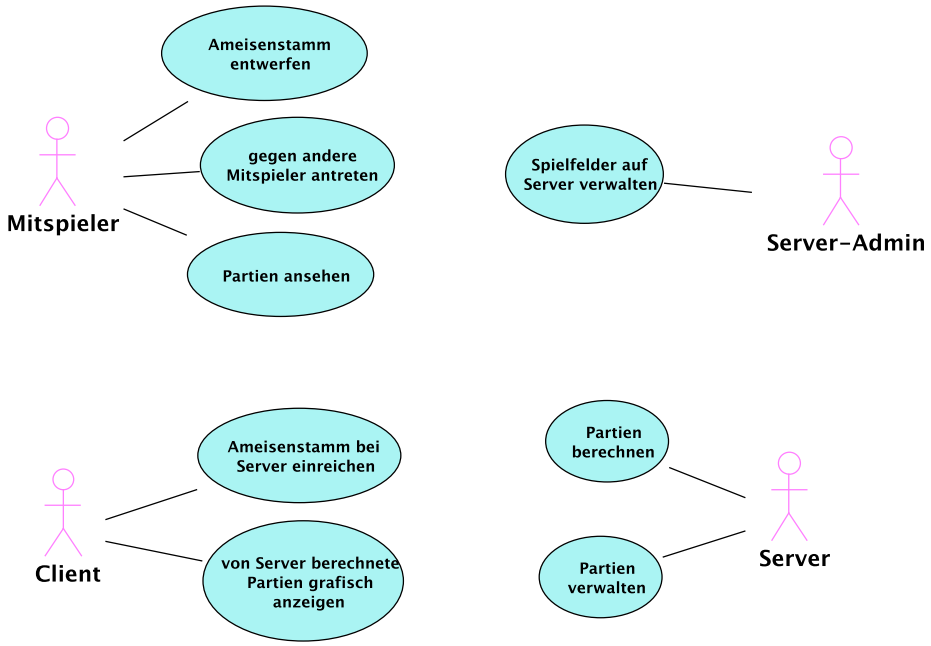


Abbildung 1: Use-Cases

1.2 Client

Die Clientprogramme müssen dazu in der Lage sein, sich mit jedem die Schnittstellen in `de.lmu.SEP0809.Communication` implementierenden Server zu verbinden. Sie müssen dort Ameisenstämme einreichen können und sie müssen in der Lage sein, den Ablauf von Spielpartien anzuzeigen.

Dazu müssen die Clientprogramme eine graphische Benutzeroberfläche haben. Sie sollten eine Benutzerschnittstelle bereitstellen, mit der es leicht möglich ist, Ameisenstämme einzureichen (und vielleicht auch zu editieren und ausprobieren) und sich Spielpartien vom Server geben zu lassen und anzuzeigen.

2 Kommunikationsschnittstellen

Die Kommunikation zwischen Client und Server ist durch die Interfaces in dem Package `de.lmu.SEP0809.Communication` geregelt. Das Klassendiagramm in Abbildung 2 gibt einen Überblick über dieses Package.

Das Hauptinterface des Servers ist `ServerInterface`. Jeder Server muss eine Implementierung dieser Klasse unter dem Namen „ServerInterface“ bei seiner RMI-Registry anmelden. Unter Benutzung dieser Referenz können sich dann eine Reihe von Clients mit dem Server verbinden.

Der Server stellt eine gewisse Anzahl von Spielarenen bereit (als Objekt vom Typ `Arena`). Eine Spielarena verwaltet die Einreichung von Ameisenstämmen auf einem einzigen

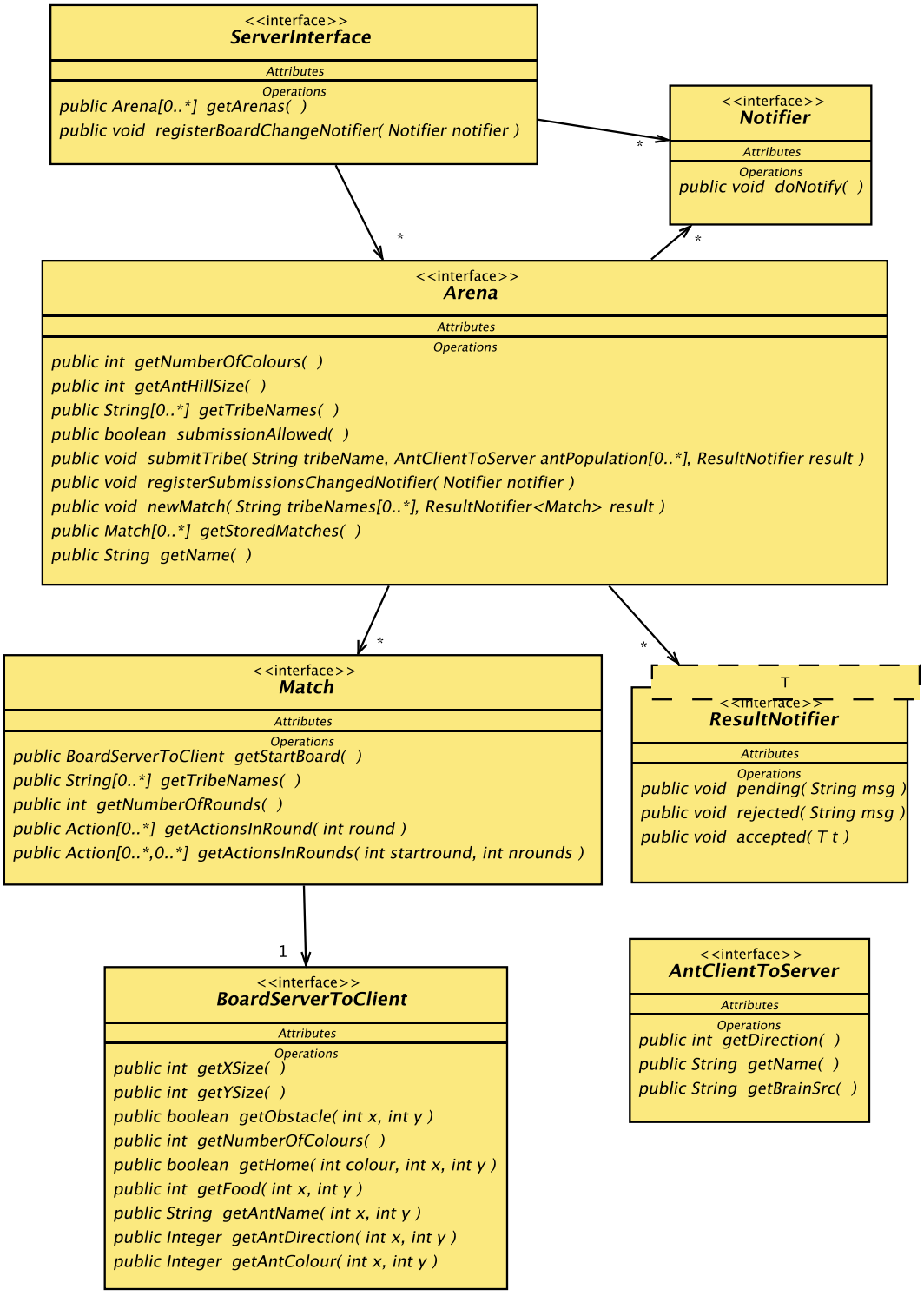


Abbildung 2: Klassendiagramm für das Package `de.lmu.SEP0809.Communication`

Spielfeld und wickelt die Spiele zwischen den verschiedenen Stämmen auf diesem Spielfeld ab. Die Client-Programme haben durch die Funktion `getArenas` vom Hauptinterface des Servers aus Zugriff auf die vom Server gespeicherten Arenen.

Hat ein Client erst einmal ein Objekt von Typ `Arena` ausgewählt, so kann er mittels dessen Funktionen Ameisenstämme einreichen sowie Spiele zwischen Ameisenstämmen ansehen (Typ `Match`). Im nächsten Abschnitt wird beschreiben, wie die Kommunikation zwischen Client und Server mit den gegebenen Interfaces ablaufen soll.

3 Benutzung des Server-Interfaces durch den Client

3.1 Einreichen eines Ameisenstamms

In Abbildung 3 ist der typische Verlauf der Einreichung eines Ameisenstammes dargestellt. Zunächst verbindet sich der Client mit dem Server, indem er sich eine Referenz auf dessen `ServerInterface` von der RMI Registry holt. Dann ruft er `getArenas` auf, um die Liste aller vom Server bereitgestellten Spielarenen zu erhalten. Daraus wählt er dann ein Objekt `arena:Arena` aus. Von diesem Objekt kann er dann Informationen über die Arena erfragen. In der Abbildung erfragt der Client den Namen der Arena und er stellt fest, dass neue Ameisenstämme darauf eingereicht werden können. Danach ruft der Client `submitTribe` auf, um seinen eigenen Ameisenstamm einzureichen. Wie Ameisenstämme kodiert sind, d.h. was das Argument `antPopulation` zu bedeuten hat, ist in der Javadoc-Dokumentation der Methode `submitTribe` erklärt.

Die Ergebnisrückgabe von der Funktion `submitTribe` wird über ein Objekt der Klasse `ResultNotifier` gehandhabt. Der Client erzeugt kurz vor dem Aufruf von `submitTribe` ein Objekt `result` vom Typ `ResultNotifier`. Dieses Objekt hat drei Funktionen `pending`, `rejected` und `accepted`. Der Client möchte, dass der Server `accepted` aufruft, wenn die Einreichung erfolgreich war. Der Server hat aber auch die Möglichkeit, `rejected` aufzurufen und die Einreichung so abzulehnen. Braucht der Server noch Bedenkzeit, so kann er dies dem Client mit einer Funktion `pending` mitteilen.

In dem Beispiel in Abbildung 3 sagt der Server zunächst, dass er noch etwas Bedenkzeit braucht, um die Einreichung dann zu akzeptieren. Nachdem die Einreichung angenommen wurde, hat das Objekt `result` seine Aufgabe erfüllt und wird vom Client zerstört.

In Abbildung 3 leben die beiden Objekte vom Typ `ServerInterface` und `Arena` in der JVM des Servers. Die anderen beiden Objekte befinden sich dagegen auf der JVM des Clients. Somit sind alle Methodenaufrufe in diesem Beispiel entfernte Methodenaufrufe, die durch RMI übertragen werden.

3.2 Ameisenstämme gegeneinander antreten lassen

Wurden in einer Arena bereits mehrere Ameisenstämme eingereicht, so kann ein Client diese gegeneinander antreten lassen und sich ansehen, wie sich die Ameisen dabei verhalten. Der

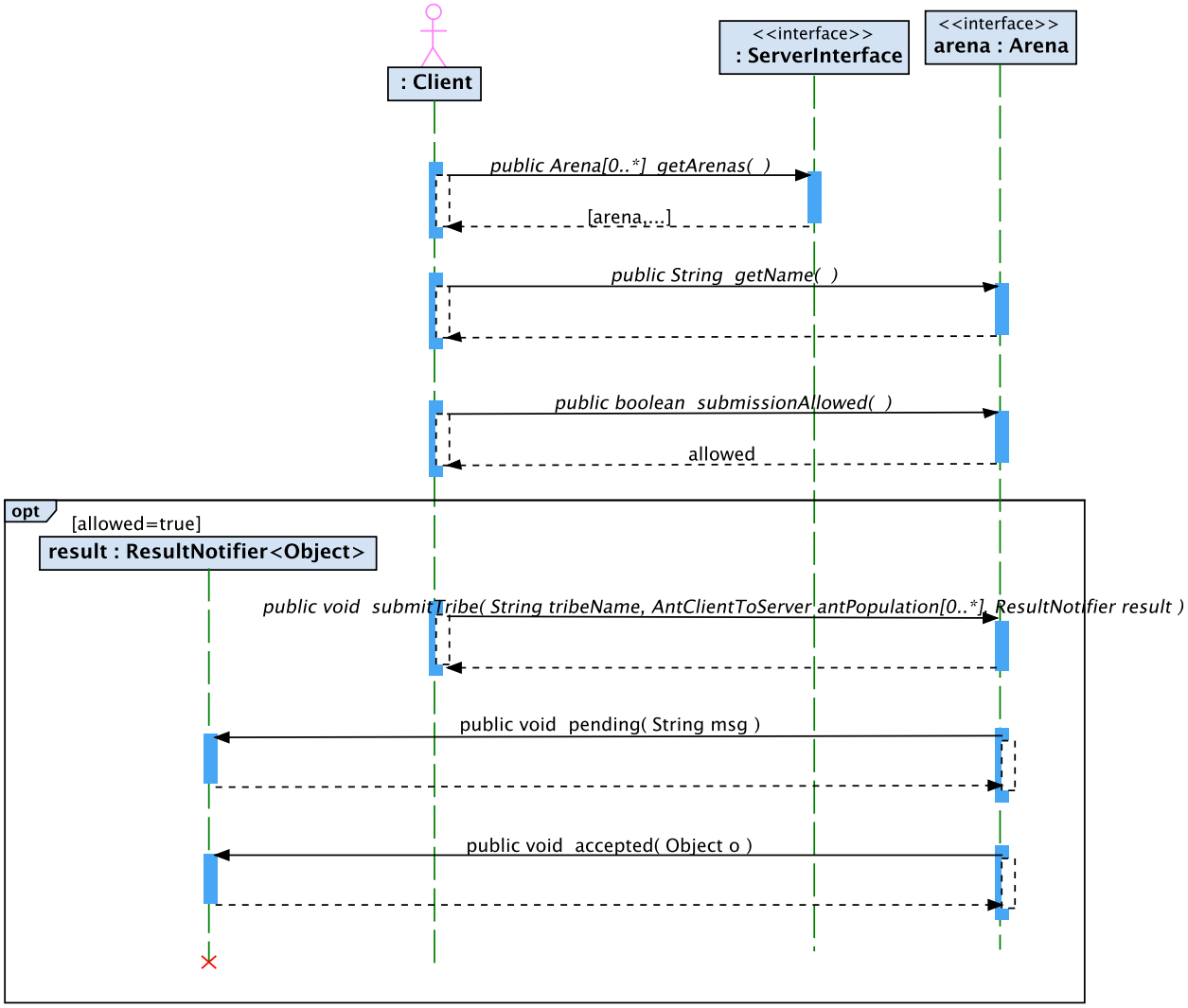


Abbildung 3: Beispielablauf für die Einreichung eines Ameisenstamms

Ablauf, wie ein Client eine solche Partie auf dem Server erzeugt und sie sich ansehen kann, ist in Abbildung 4 dargestellt.

In dieser Abbildung wird angenommen, dass der Client bereits eine Arena `arena` ausgewählt hat. Er erfragt dann zunächst, welche Ameisenstämme überhaupt schon in der Arena eingereicht wurden. Dazu ruft er die Funktion `getTribeNames` auf. Als nächstes erfragt er, wie viele Ameisenhaufen es denn überhaupt auf dem Spielfeld der Arena gibt, d.h. wie viele Ameisenstämme er gegeneinander antreten lassen kann. Dazu benutzt er die Funktion `getNumberOfColours`.

Mit diesen Informationen kann der Client eine Spielpartie von der Arena erfragen. Er schreibt sich die Liste der Namen der Ameisenstämme, die er in dieser Reihenfolge auf die Ameisenhaufen setzen will, auf und fordert mit der Funktion `newMatch` eine Spielpartie zwischen diesen Ameisenstämmen an.

Die Funktion `newMatch` liefert ihren Rückgabewert wieder mit einem `ResultNotifier` zurück. Diesen hat sich der Client vor dem Aufruf von `newMatch` erzeugt. In der Abbildung 4 akzeptiert der Server die Anfrage nach einer Spielpartie sofort. Er erzeugt ein Objekt `m:Match` und gibt es mit der `accept`-Funktion an den Client zurück.

Dieser kann nun das Objekt `m` nutzen, um sich die Spielpartie anzusehen. Zunächst erfragt er mit der Funktion `getStartBoard`, wie das Spielfeld am Anfang der Partie aussieht. Dieses Spielfeld wurde vom Server konstruiert, indem er die vom Client gewünschten Ameisenstämme auf die entsprechenden Ameisenhaufen gesetzt hat. Beachte, dass der Client das Spielfeld nur sehen kann, indem er sich ein `Match` vom Server holt. Da der Server Anfragen nach neuen Matches ablehnen oder mittels `pending` verzögern kann, kann er somit zum Beispiel erzwingen, dass sich niemand erst das Spielfeld ansieht und dann mit diesem Wissen einen Ameisenstamm einreicht.

Nachdem der Client das Startspielbrett erfragt hat, kann er nun erfragen, aus wie vielen Runden die Partie `m` besteht. Dann kann er die Aktionen der Ameisen in den einzelnen Spielrunden nacheinander erfragen und so den Ablauf des Spiels rekonstruieren. Die mögliche Aktionen der Ameisen sind in der Klasse `Match.Action` formuliert. Diese entsprechen den Aktionen im ersten Teil der Aufgabenbeschreibung.

Wie im vorangegangenen Abschnitt befindey sich das Objekt `arena` auf dem Server und die Objekte vom Typ `Client` und `ResultNotifier<Match>` befinden sich beim Client. Für die Position des Objekts `m:Match` gibt es mehrere sinnvolle Möglichkeiten. Dieses Objekt wird auf dem Server erzeugt, also wäre es eine Möglichkeit, dass es auch dort verbleibt. Möglich wäre jedoch auch, dass es vom Server serialisiert wird und dann zum Client übertragen wird.

3.3 Werterückgabe

Die Abläufe in Abbildungen 3 und 4 zeigen, wie Objekte vom Typ `ResultNotifier` benutzt werden, um Ergebniswerte vom Server an Clients zurückzugeben. Diese Objekte sollen von jedem Client so implementiert sein, dass sie beliebig viele `pending`-Meldungen annehmen können, bis letztendlich entweder die Methode `rejected` oder `accepted` aufgerufen wird. Wurde einmal eine dieser beiden Methoden aufgerufen, haben diese Objekte ihre Aufgabe

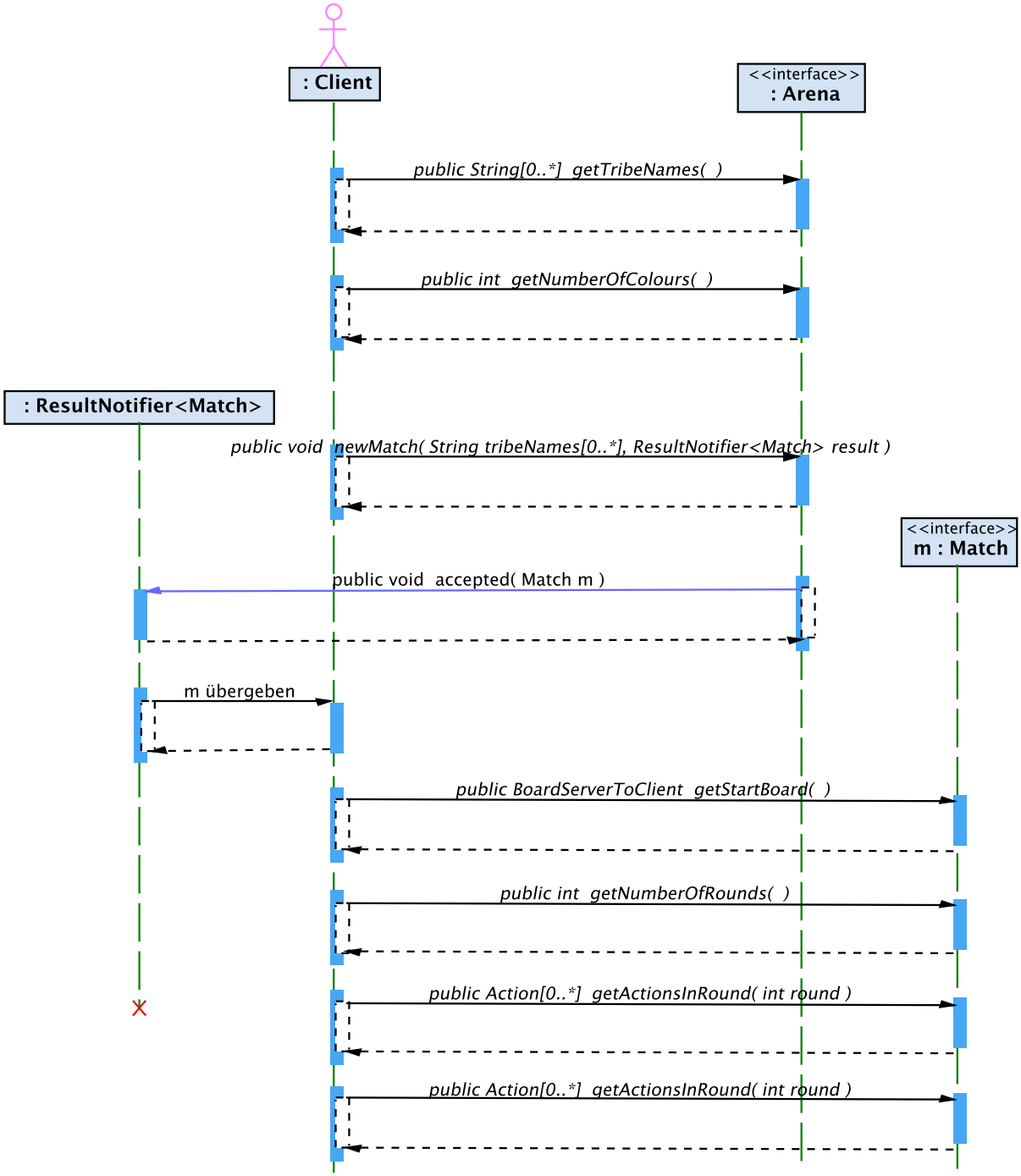


Abbildung 4: Beispielablauf für das Ansehen eines Spiels zwischen Ameisenstämmen

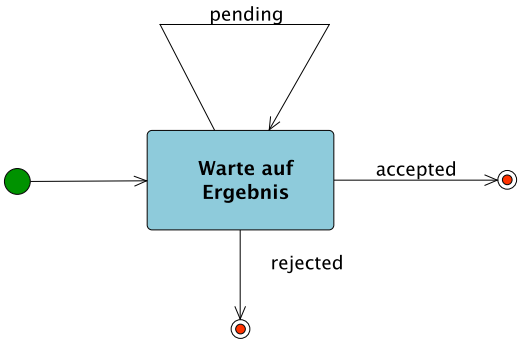


Abbildung 5: Zustandsdiagramm für `ResultNotifier`

erfüllt und können gelöscht werden. Sie müssen dann keine weiteren `pending`-Nachrichten annehmen.

Diese Anforderung ist im Zustandsdiagramm in Abbildung 5 dargestellt.

Die Klasse `ResultNotifier` ist parametrisiert. So kann zum Beispiel bei der Klasse `ResultNotifier<Match>` nach Annahme einer Anfrage ein Wert vom Typ `Match` mitgeliefert werden, siehe Abbildung 4. Dies ist dann gewissermaßen als Rückgabewert zu verstehen, also als Antwort des Servers auf eine Anfrage des Clients, bei der er einen `ResultNotifier` übergeben hat.

Wird `ResultNotifier` ohne Parameter benutzt, so darf ein beliebiger Wert (z.B. `null`) als Rückgabewert übergegeben werden.

3.4 Benachrichtigung über Zustandsänderungen im Server

Um über Änderungen auf dem Server informiert zu werden, können die Client-Programme Notifikationsobjekte beim Server registrieren. Dafür implementieren sie das Interface `Notifier` und übergeben eine Referenz auf ein Objekt von diesem Typ an den Server (mit einer der Funktionen `registerBoardChangeNotifier` und `registerSubmissionsChangedNotifier`). Ändert sich dann der Zustand im Server, so ruft er die Funktion `doNotify` im `Notifier`-Objekt des Clients auf.

4 Anforderungen an den Server

Insgesamt kann man den Server als Programm verstehen, das eine Liste von Arenen speichert über das RMI-Interface `ServerInterface` exportiert.

Der Zustand des Serverprogramms enthält neben der Liste der Arenen auch noch eine Liste von Referenzen auf `Notifier`-Objekte. Die `doNotify`-Methode in all diesen `Notifier`-Objekten soll vom Server immer dann aufgerufen werden, wenn sich die Liste der verfügbaren Arenen ändert.

Wie die Arenen auf den Server kommen und wie Arenen erzeugt und gelöscht werden,

ist nicht spezifiziert. Möglich wäre zum Beispiel, den Server mit einem Kommandozeileninterface oder einer GUI auszustatten. Es wäre aber auch möglich, diese Funktionalität in einen erweiterten Client zu integrieren.

4.1 Eine Arena

Eine Arena besteht aus einem festen Spielfeld und erfasst ein Protokoll zur Einreichung von Ameisenstämmen für dieses Spielfeld und zum Austragen von Spielpartien zwischen diesen.

Am Anfang ihrer Existenz ist eine jede Arena noch leer, d.h. es gibt darin noch keine Ameisenstämme und es gibt dann natürlich auch noch keine Spielpartien. Ameisenstämme werden erst im Laufe der Zeit von Mitspielern mit der Funktion `submitTribe` eingereicht und Spielpartien werden von ihnen mit der Funktion `newMatch` erzeugt.

Da der Wettkampf zwischen mehreren Mitspielern in einer Arena nur dann fair ist, wenn keiner der Mitspieler das Spielfeld gesehen hat, bevor er seinen Ameisenstamm einreicht, kann sich eine Arena in zwei Zuständen befinden. Im Anfangszustand ist es erlaubt, dass neue Ameisenstämme eingereicht werden. Die Funktion `submissionsAllowed` gibt dann `true` zurück. Um sicherzustellen, dass kein Mitspieler, der das Spielfeld schon gesehen hat, einen Ameisenstamm einreicht, kann die Arena in einen Zustand übergehen, in dem keine neuen Einreichungen mehr erlaubt sind. Dann gibt `submissionsAllowed` den Wert `false` zurück. Es ist dann nicht möglich, wieder in einen Zustand übergehen, in dem Einreichungen erlaubt sind.

Unter welchen Bedingungen der Übergang vom ersten zum zweiten Zustand stattfindet, ist nicht genau vorgegeben. Ein sinnvoller Zeitpunkt, um neue Einreichungen zu verbieten, wäre zum Beispiel wenn ein Mitspieler das Spielfeld gesehen hat, z.B. indem er einen Wert vom Typ `Match` für diese Arena erhalten hat. Beachte aber, dass der Server mit dem `ResultNotifier` in `newMatch` selbst festlegen kann, wann das erste Match auf der Arena an einen Mitspieler ausgegeben wird. So kann er zum Beispiel erzwingen, dass es erst eine gewisse Mindestanzahl von Einreichungen geben muss, bevor jemand eine Spielpartie sehen kann, indem er Anfragen nach einer neuen Partie solange mit `pending` beantwortet, bis die erforderliche Anzahl von Mitspielern sich registriert hat.

In einer Arena sind weiterhin noch folgende Daten gespeichert:

1. Eine Liste von erfolgreich mit `submitTribe` eingereichten Ameisenstämmen, wobei ein Ameisenstamm aus einem Paar von einem Namen und einer Liste von Werten des Typs `AntClientToServer` ist. Jeder Name darf nur einmal vorkommen und kann durch eine Neueinreichung nicht überschrieben werden.

2. Eine Menge von eingereichten Ameisenstämmen, über deren Akzeptanz noch nicht entschieden ist. Diese Menge besteht aus den Ameisenstämmen, die mit einem Funktionsaufruf `submitTribe(name,ants,result)` eingereicht wurden, für die aber noch nicht `result.rejected()` oder `result.accepted(...)` aufgerufen wurde.

   Für diese neu eingereichten Ameisenstämme gibt es folgende Forderungen:

- Wenn sich die Arena in dem Zustand befindet, in dem keinen Neueinreichungen mehr erlaubt sind, dann müssen diese neuen Ameisenstämme abgelehnt werden und `result.rejected` mit einer entsprechenden Nachricht aufgerufen werden.
- Wenn sich die Akzeptanz eines eingereichten Ameisenstammes voraussichtlich verzögern wird (z.B. weil erst noch ein Moderator darüber entscheiden muss), dann soll `result.pending` mit einer entsprechenden Nachricht aufgerufen werden.

Ansonsten kann es von der Serverimplementierung abhängen, welche Ameisenstämme angenommen werden. Es ist den Servern somit freigestellt, sinnvolle Richtlinien zu erzwingen, z.B. dass nur eine bestimmte Maximalanzahl von Ameisenstämmen eingereicht werden darf.

3. Eine Liste von gespeicherten Spielpartien. Diese Liste kann von dem Mitspielern mit der Funktion `getMatches` erfragt werden. Sie soll nur Spielpartien enthalten, die vorher durch einen Aufruf an `newMatch` erzeugt wurden. Es ist freigestellt, ob und welche Spielpartien gespeichert werden.

4. Eine Menge von Anfragen über neue Spielpartien, über deren Akzeptanz noch nicht entschieden ist, analog zum Fall für die Einreichung von Ameisenstämmen oben. Nach welchen Kriterien die Akzeptanz entschieden wird, ist nicht festgelegt und kann sich nach sinnvollen Kriterien, wie zum Beispiel dem Speicherverbrauch richten (aber der Server muss schon in der Lage sein, überhaupt Spielpartien zurückzugeben).

5. Schlussendlich speichert eine Arena noch eine Liste von `Notifiern`, deren `doNotify`-Funktionen aufgerufen werden, wenn sich die Liste der erfolgreich eingereichten Ameisenstämme ändert. Die Clients haben die Möglichkeit, Referenzen auf ihre `Notifier`-Objekte mit der Funktion `registerSubmissionsChangedNotifier` zu dieser Liste hinzuzufügen. Wenn der Aufruf zu einer `doNotify`-Funktion fehlschlägt, dann darf das entsprechende Objekt aus der Liste der `Notifier` gelöscht werden.

4.2 Ablauf von Partien auf dem Server

Die Hauptaufgabe des Server ist, den Ablauf von Spielpartien zwischen mehreren Ameisenstämmen zu berechnen. Ein Client kann die Erzeugung einer solchen Partie mit der Funktion `newMatch` erfragen.

Akzeptiert der Server die Anfrage eines Clients nach einer neuen Partie, so muss er diese Partie in Form eines Objekts `m` vom Typ `Match` bereitstellen. Das Spielfeld zu Anfang der Partie (nämlich `m.getStartBoard()`) entsteht dabei, indem die vom Client beim Aufruf von `newMatch` angegebenen Ameisenstämme auf die entsprechenden Ameisenhügel gesetzt werden. Auf welche Zellen in den Hügeln die Ameisen genau gesetzt werden, steht in der Javadoc-Dokumentation von `submitTribe`.

Das Objekt `m:Match` erlaubt nun, die Aktionen der Ameisen von dieser Startposition aus für eine gewisse Anzahl von Runden (nämlich `m.getNumberOfRounds()`) zu verfolgen.

Es ist dabei von der Implementierung des Interfaces `Match` abhängig, wann die Runden berechnet werden. Es wäre zum Beispiel möglich, gleich bei der Konstruktion des Objekts `m` die Aktionen in allen Runden zu berechnen und für die spätere Abfrage zu speichern. Andererseits wäre es aber auch möglich, außer dem Startspielbrett gar nichts zu speichern und erst bei Anfragen nach den Aktionen der Ameisen in einer bestimmten Runde diese dann zu berechnen. Natürlich sind zwischen diesen beiden Extremen auch Zwischenschritte möglich. Zum Beispiel könnte ein Objekt vom Typ `Match` die Spielfelder nach jeweils 1000 Spielrunden speichern; bei einer Anfrage nach den Aktionen in der 1012. Runde könnte es dann mit seiner Berechnung in der 1000. Runde anfangen und müsste nur noch zwölf Runden ausrechnen.

Welche Aktionen die Ameisen in jedem Schritt machen, ist durch die Programme in ihren Gehirnen gegeben. Die genaue Definition dieser Programmiersprache wird in einem weiteren Teil der Aufgabenstellung zu finden sein. Für den Moment kann man sich Ameisen noch beliebig verhalten lassen, zum Beispiel indem man die Ameisen einfach zufällige Aktionen machen lässt. Die Benutzung von zufälligen Aktionen ist gerechtfertigt, da die Programmiersprache ebenfalls auf zufällig erzeugte Zahlen zurückgreifen kann. Das Verhalten einer Ameise kann in jedem Schritt abhängen von ihrem (noch nicht beschriebenen) internen Gehirnzustand, von den zufällig erzeugten Zahlen, sowie dem Aussehen der Spielfeldzelle, auf der sie gerade steht, und dem Aussehen der Spielfeldzelle, in deren Richtung sie schaut. Ein Platzhalter für eine Ameise könnte folgendermaßen aussehen:

```
class Ant {
  private Random r;
  private final String name;

  public Ant(String name, String brainPrg) {
    this.name = name;
  }

  public Action step(Cell here, Cell ahead) {
    // benutze r um eine zufällige Aktion zurückzugeben
  }
  ...
}
```

Der nach außen hin sichtbare Zustand eines Objekts `m` vom Typ `Match` soll sich zu dessen Lebzeiten nicht ändern. Das heißt, die Funktion `getStartBoard` gibt immer ein gleich aussehendes Spielfeld zurück und mehrere Aufrufe zu den anderen Funktionen sollen immer das gleiche Ergebnis haben.

5 Erweiterungen

Es ist möglich, dass Server und Client erweiterte Interfaces implementieren, solange sie die offiziellen Interfaces ebenfalls unterstützen. Zum Beispiel könnte ein Server ein zweites Interface zur Serveradministration exportieren.

Die offiziellen Interfaces können, wie in jedem realistischen Softwareentwicklungsprojekt, ebenfalls noch Änderungen erfahren. Beispielsweise können bei Bedarf noch Erweiterungen zur Authentifizierung von Clients und zum Abfragen von Highscore-Listen aufgenommen werden. Auch können noch Funktionen zur Effizienzoptimierung hinzukommen.

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: 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-

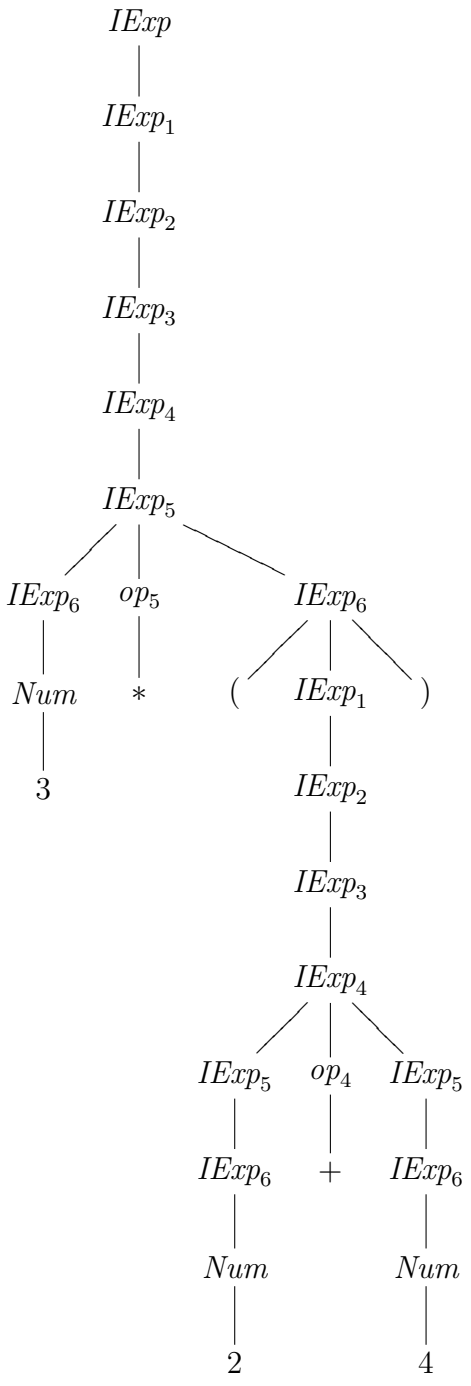


Abbildung 3: Ableitungsbaum für 3 * (2 + 4) in der konkreten Syntax.

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.

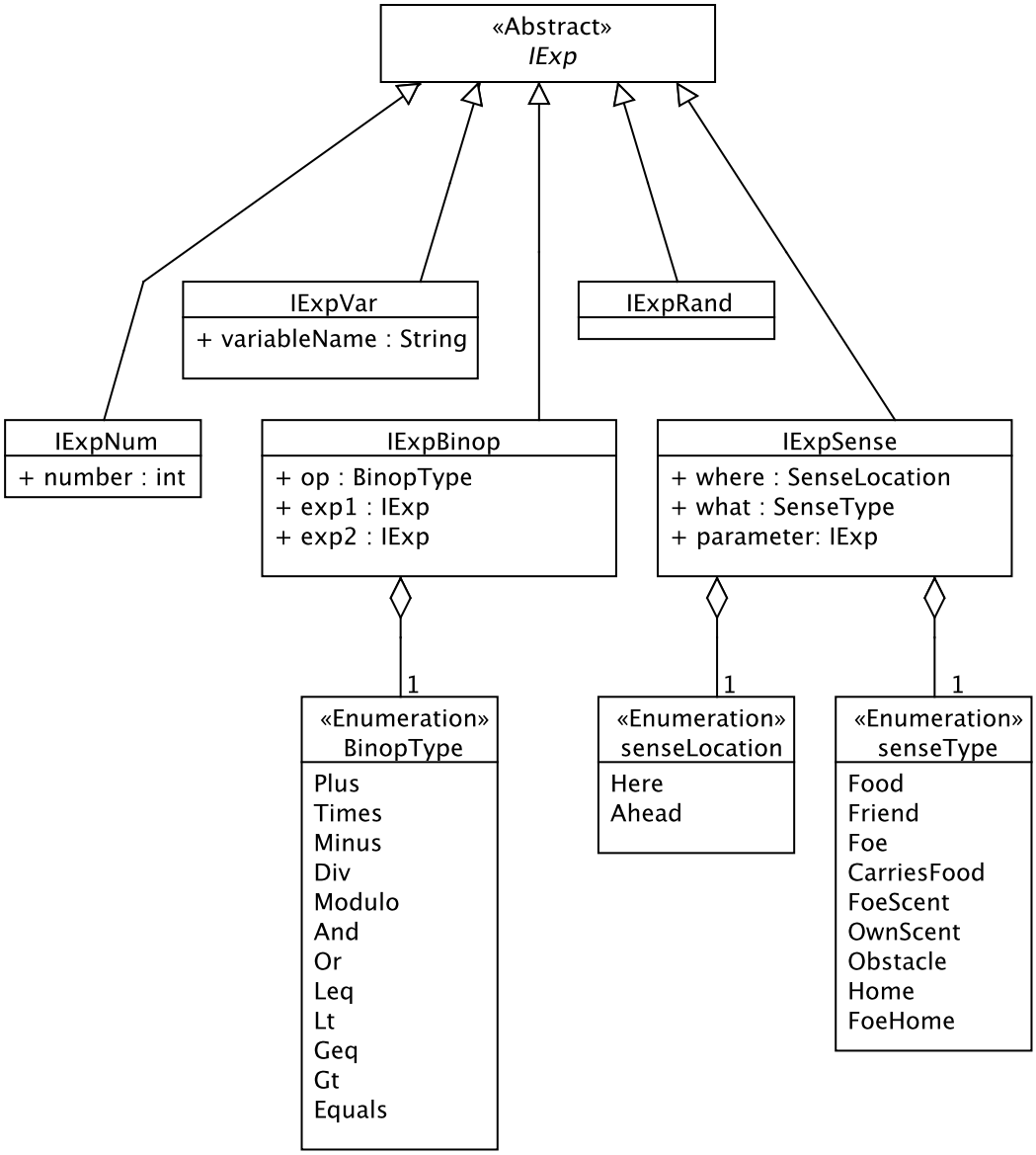


Abbildung 4: Klassen für die Repräsentierung von arithmetischen Ausdrücken

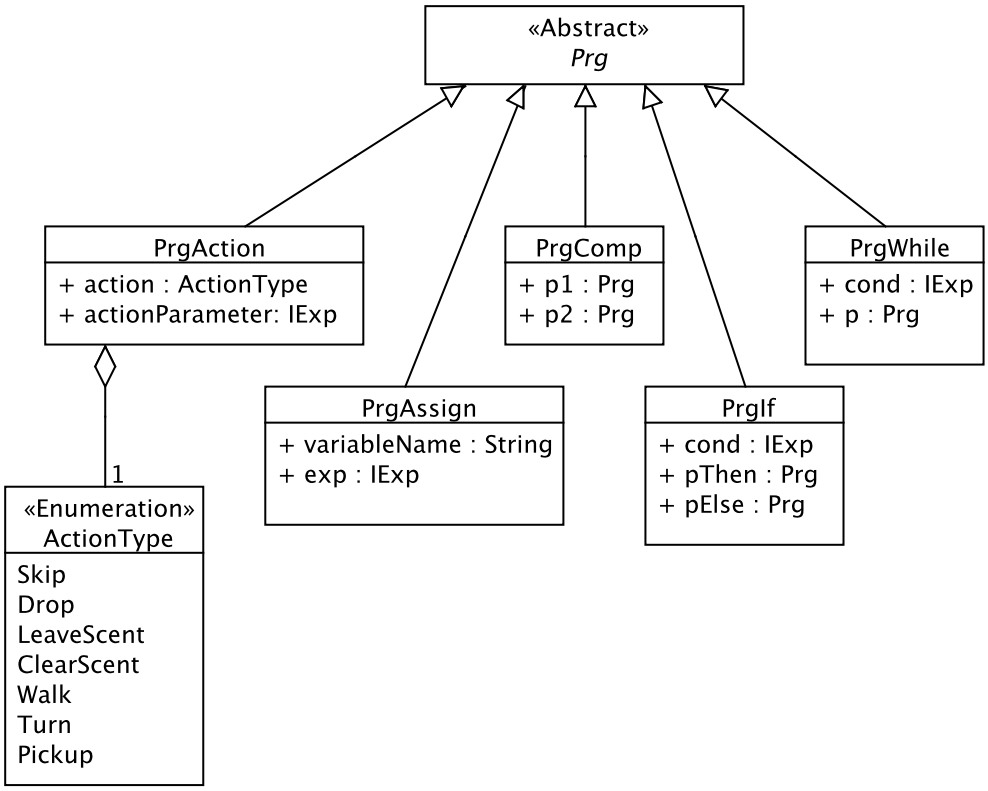


Abbildung 5: Klassen für die Repräsentierung von Programmen

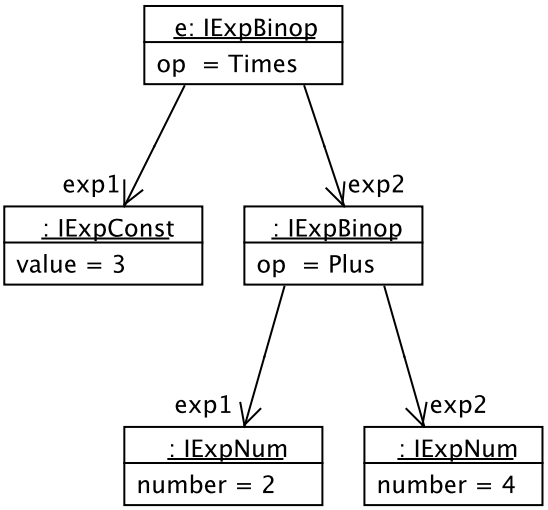


Abbildung 6: Repräsentierung des Ausdrucks 3 * (2+4)

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 ρ 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 ρ 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 ρ 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 σ für das Paar von Zellen $(\textbf{here}, \textbf{ahead})$. Der Wert eines arithmetischen Ausdrucks t, in Abhängigkeit von dem Zellenpaar σ und der Gedächtnisfunktion ρ, 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 ρ 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.

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

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

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

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

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

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

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

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

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

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

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 $\mathit{step}((\rho, E), \sigma)$ durch Fallunterscheidung über den Stack E:

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

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

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

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

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

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

– 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

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

– 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.