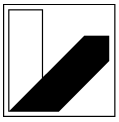

UNIVERSITÄT
BAYREUTH

Fakultät für Mathematik, Physik und Informatik
Institut für Informatik
Lehrstuhl für Angewandte Informatik I
Software Engineering

MuLE - eine multiparadigmatische Sprache für die Lehre

Version 1.0

Nikita Dümmel

3. Dezember 2019

Zusammenfassung

MuLE ist konzipiert als eine multiparadigmatische Programmiersprache für die Lehre, die sich an Programmieranfänger richtet. In der aktuellen Version ist der prozedurale Teil der Sprache fertiggestellt. Damit lässt sich die Sprache zum Einen in Einführungsveranstaltungen einsetzen und zum Anderen bietet sie eine Grundlage für die Umsetzung weiterer Paradigmen an. In diesem Bericht werden die Entwurfsentscheidungen der Sprache erörtert, sowie einzelne Sprachkonstrukte erklärt.

Inhaltsverzeichnis

1. Einleitung

Programmieren lernen ist eine große Herausforderung. Bei diesem Prozess müssen die Programmieranfänger sich mit zwei Problemen gleichzeitig auseinandersetzen: Erlernen einer Programmiersprache und Lösen von abstrakten Problemen. Studien [6][5] und eigene Beobachtungen belegen, dass Problemlösen sich meistens als das schwierigere Problem für Programmieranfänger entpuppt, welches aber durch den Einsatz einer schwer zu erlernenden Programmiersprache zusätzlich verkompliziert wird. Es muss also eine leicht erlernbare Sprache verwendet werden, die jedoch in der Lage ist, die wichtigsten Konzepte der Programmierung demonstrieren zu können.

MuLE (*Multi-Paradigm Language* for *Education*)[1] ist eine multiparadigmatische Sprache für die Lehre, die sich an Programmieranfänger richtet. Die Sprache soll demnach drei Programmierparadigmen unterstützen: prozedural, objektorientiert und funktional. Weiterhin soll die Sprache eine leicht erlernbare Syntax besitzen und ausdrucksmächtig sein bei einem minimalen Sprachumfang. Es soll möglich sein, die Sprache in Erstsemesterveranstaltungen sowie im Schulunterricht einsetzen zu können. Die Sprachkonstrukte sollen leicht verständlich sein und es muss möglich sein, einfache Programme mit minimalem Einsatz an Sprachkonstrukten zu schreiben, wobei die Konstrukte nacheinander eingeführt werden können. Überladen von Operationen und Operatoren wird nicht unterstützt, um Mehrdeutigkeiten zu vermeiden und die Sprache für Anfänger nicht komplexer zu machen. Weiterhin soll das Abstraktionsniveau der Sprache nicht zu hoch aber auch nicht zu niedrig sein [4]. Es muss möglich sein, in der Informatik geläufige Probleme, die z.B. durch Hardwarelimitierungen verursacht werden, demonstrieren zu können. Gleichzeitig sollen wenig gebrauchte Konzepte, die die Syntax der Sprache unnötig größer machen, ausgelassen werden.

Derzeit ist der prozedurale Teil der Sprache implementiert. In dieser Ausarbeitung werden die Konzepte des implementierten prozeduralen Teils zusammengefasst und die zugrundeliegenden Entwurfsentscheidungen begründet. Die einzelnen Kapiteln sind nicht aufeinander aufbauend geschrieben. Im Anhang befindet sich eine Übersicht über getroffene Entwurfsentscheidungen, Übersicht über alle Schlüsselwörter, Operatoren und Symbole sowie die gesamte Grammatik der Sprache.

2. Sichtbarkeitsbereiche und Namensräume

Sichtbarkeitsregeln definieren Bereiche im Quelltext, in denen benannte Sprachelemente referenziert werden können. Zu den benannten Elementen zählen:

- Übersetzungseinheiten, d.h. MuLE Programme oder Bibliotheken

[1] http://www.ai1.uni-bayreuth.de/de/projects/MuLE/index.html

- Importanweisungen
- Typdeklarationen
- Literale von Aufzählungstypen
- Attribute von Kompositionen
- Operationen
- Parameter von Operationen
- Variablen

Innerhalb einer Übersetzungseinheit werden die Sichtbarkeitsbereiche hierarchisch aufgebaut. Die unterste Ebene stellt der Bereich dar, der die Referenz auf das benannte Element beinhaltet. Der Behälter dieses Bereichs bestimmt den überliegenden Sichtbarkeitsbereich, dessen Elemente in unteren Bereichen sichtbar sind, sofern es sich nicht um Vorwärtsreferenzen handelt.

```
1  program scope1
2
3  main
4      variable x : integer
5      loop
6          variable y : integer
7          x := 2
8          variable z : integer
9      endloop
10 endmain
```

Quelltext 1: Beispiel zur Veranschaulichung von Sichtbarkeitsbereichen.

Ein einfaches Beispiel ist in Quelltext 1 zu sehen. Darin befindet sich eine einzelne Referenz auf ein benanntes Element in der Wertzuweisung auf die Variable `x`. Der Sichtbarkeitsbereich für diese Referenz sieht folgendermaßen aus:

`[y] -> [x] -> [] -> NULLSCOPE`

Die Zuweisung, und dementsprechend die Referenz, befindet sich innerhalb einer Schleife, die den unmittelbaren Sichtbarkeitsbereich darstellt. Darin werden zwei Variablen deklariert: `y` und `z`, jedoch nur die Variable `y` wird im Kontext der Zuweisung im Sichtbarkeitsbereich der Schleife aufgelistet, da es sich im Fall von `z` um eine Vorwärtsreferenz handeln würde. Die Schleife befindet sich in der `main`-Prozedur, für die der umfassende Sichtbarkeitsbereich, der die referenzierte Variable `x` beinhaltet, berechnet wird. Das Programm, in dem sich die Hauptprozedur befindet, besitzt keine weiteren Deklarationen, weshalb der entsprechende Bereich leer ist. Der `NULLSCOPE` stellt aus technischen Gründen immer die oberste Ebene in der Hierarchie dar und entspricht einem leeren Sichtbarkeitsbereich.

2.1. Lokaler Sichtbarkeitsbereich

In einem lokalen Sichtbarkeitsbereich ist der Bezeichner nur innerhalb von einem Block (sowie seinen Unterblöcken, falls vorhanden) sichtbar und demnach referenzierbar. Blöcke sind zusammenhängende Sammlungen von Anweisungen und gehören zu einem Sprachkonstrukt (Kapitel 6). Vorwärtsreferenzen auf Variablen sind nicht erlaubt. Im Fall von Operationen und `foreach`-Anweisungen sind die entsprechenden Parameter bzw. die Iteratorvariable Teil des lokalen Sichtbarkeitsbereichs der Operation bzw. der `foreach`-Anweisung.
Bei geschachtelten Blöcken, wie im Beispiel in Quelltext 1, sind die Bezeichner des äußeren Blocks im inneren Sichtbar. Es entsteht ein hierarchisch aufgebauter Sichtbarkeitsbereich, in dem keine zwei gleiche Bezeichner vorkommen dürfen. Dementsprechend ist in MuLE keine Variablenüberdeckung erlaubt. Generell ist der lokale Sichtbarkeitsbereich stets Teil von einem anderen lokalen, oder von dem globalen Sichtbarkeitsbereich.

2.2. Globaler Sichtbarkeitsbereich

Im globalen Sichtbarkeitsbereich ist der Bezeichner innerhalb der gesamten Übersetzungseinheit sichtbar. Das betrifft Namen der Importe, Typdeklarationen, Literale von Aufzählungstypen und Operationen. Vorwärtsreferenzen sind im globalen Sichtbarkeitsbereich erlaubt, so kann z.B. im Rumpf der Operation `foo` eine andere Operation `bar` aufgerufen werden, auch wenn `foo` im Quelltext vor `bar` deklariert wurde. Überladen von Operationen ist nicht erlaubt, es dürfen keine zwei Operationen mit dem gleichen Namen vorkommen. Ebenso muss der Name von jeder Typdeklaration einzigartig sein. Der globale Sichtbarkeitsbereich ist stets Teil des gesamten Sichtbarkeitsbereichs in einer Übersetzungseinheit.

2.3. Sichbarkeitsbereich für Übersetzungseinheiten

MuLE bietet eine Reihe von Standardbibliotheken (Kapitel 8) an, die beispielsweise Prozeduren zur Ausgabe von Werten oder mathematische Funktionen anbieten. Diese Bibliotheken sind im Kontext einer Importanweisung immer sichtbar. Um benutzerdefinierte Bibliotheken importieren zu können, muss eine der folgenden Bedingungen erfüllt werden: die Bibliothek befindet sich im gleichen Projekt wie das importierende Programm, oder in einem Projekt, welches in der Liste der Abhängigkeiten im Projekt des importierenden Programms aufgelistet wird.

Es gibt keine Sichtbarkeitsmodifikatoren für Deklarationen von Typen, Attributen und Operationen in MuLE. In einer Bibliothek deklarierte Typen und Operationen sind nach außen sichtbar.

2.4. Qualifizierter Sichtbarkeitsbereich

Auf Elemente von importierten Bibliotheken bzw. Attribute von Kompositionen wird mit Hilfe von Qualifizierung zugegriffen. Zu diesem Zweck wird der . (Punkt) Operator (Abschnitt 4.4) verwendet, der auf einen Wert oder Import angewendet werden kann. Im ersten Fall muss es sich um Werte (Variablen, Parameter, Attribute oder Rückgabewerte einer Operation) von einem Kompositionstyp handeln. Bei einer Importanweisung wird der Name der importierten Bibliothek angegeben und der Import selbst bekommt einen Alias, der für den qualifizierten Zugriff auf importierte Elemente verwendet wird.

2.5. Namensräume

Sichtbarkeitsbereiche definieren eindeutige Namensräume, in denen kein Bezeichner (Abschnitt 4.2) doppelt vorkommen darf, damit einzelne Deklarationen eindeutig identifiziert werden können. So ist es möglich, in einem Programm zwei Variablen mit dem gleichen Namen zu deklarieren, sofern sie sich in unterschiedlichen Namensräumen befinden (siehe Quelltext 2). Sollten sich aber zwei benannte Elemente im gleichen Sichtbarkeitsbereich befinden, wird eine Fehlermeldung angezeigt. In Quelltext 3 ist ein Beispiel für gleiche Beizeichner im selben Namensraum zu sehen: die Operation `a` befindet sich im globalen Sichtbarkeitsbereich während die Variable `a` sich im lokalen Bereich der Hauptprozedur befindet. Da der lokale Bereich dem globalen untergliedert ist und sie gemeinsam den gleichen Namensraum für diese beiden Deklarationen bilden, ist dieses Programm fehlerhaft und wird daher nicht übersetzt.

```
1 program scope2
2 main
3     if true then
4         variable x : integer
5     else
6         variable x : integer
7     endif
8 endmain
```

Quelltext 2: Getrennte lokale Sichtbarkeitsbereiche.

```
1 program scope3
2
3 operation a()
4 endoperation
5
6 main
7     variable a : integer
8 endmain
```

Quelltext 3: Zwei gleiche Bezeichner im selben Namensraum.

```
1 program scope4
2 import IO as io
3
4 operation writeString(parameter str : string)
5     io.writeString("myWriteString: " & str)
6 endoperation
7
8 main
```

```
9     io.writeString("Hello\n")
10    writeString("Hello, world!")
11 endmain
```

Quelltext 4: Unterschied von einfachen und qualifizierten Namen.

In Quelltext 4 ist ein legitimes Beispiel für zwei Operationen mit gleichen Bezeichnern in getrennten Namensräumen zu sehen. Die Standardbibliothek `IO` definiert die Operation `writeString`, genauso wie das Programm `scope4`, welches diese Standardbibliothek importiert. Hiermit kann im selben Programm auf zwei `writeString` Operationen zugegriffen werden, wobei der qualifizierte Name verwendet werden muss, um die importierte Operation aufzurufen.

3. Grammatiknotation

Die Grammatik der Sprache besteht aus einer Reihe von Produktionsregeln. Jede Regel wird aus einem Nichtterminalsymbol auf der linken, und einer Menge von Terminal- und Nichtterminalsymbolen auf der rechten Seite zusammengesetzt. Grundlegend gilt:

- Terminal- und Nichtterminalsymbole werden mit `Maschinenschrift` geschrieben.
- Terminalsymbole werden zusätzlich 'blau und mit einfachen Anführungszeiten' markiert.
- `:` trennt die linke von der rechten Seite in einer Produktionsregel.
- `;` schließt eine Produktionsregel ab.
- `()` besitzen die übliche Funktion der Klammerung, z.B. um einen Operator auf eine Gruppe von Elementen anzuwenden.
- `|` wird zum Aufzählen von alternativen Elementen verwendet.
- `?` stellt ein optionales Vorkommen (0 .. 1) eines Elements dar.
- `*` stellt ein mehrfaches optionales Vorkommen (0 .. n) eines Elements dar.

```
CompilationUnit : ('program'|'library') ID
  Import* ProgramElement* MainProgram?;
```

- `+` stellt ein mehrfaches Vorkommen (1 .. n) eines Elements dar.
- `x..y` stellen einen Bereich zwischen x und y dar.

```
INT : ('0'..'9')+;
```

- `[Element]` verdeutlicht, dass an dieser Stelle ein existierendes Element referenziert wird.

```
Import : 'import' [CompilationUnit] 'as' ID;
```

Abschnitt C beinhaltet die gesamte Grammatik. Es ist anzumerken, dass die Sprache nicht nur durch die Grammatik, sondern zusätzlich durch Validierungmechanismen definiert wird. Diese Mechanismen werden an den nötigen Stellen ebenfalls angesprochen.

4. Lexikalische Einheiten

Logisch zusammenhängende Zeichengruppen bilden im Quelltext lexikalische Einheiten, die sogenannten Tokens. Dabei wird jeweils die längste erlaubte Zeichenkette als eine lexikalische Einheit ausgewertet. Tokens stellen im Programmcode die kleinste unabhängige Einheit mit einer Bedeutung, die durch Sprachregeln festgelegt wird, dar. Folgende Arten von Tokens gibt es:

- Bezeichner
- Schlüsselwörter
- Operatoren
- Trennsymbole
- Literale

Nebeneinander stehende Bezeichner und Schlüsselwörter müssen mit Leerräumen voneinander getrennt werden. Zu den Leerräumen zählen Leerzeichen, Tabulatoren, Zeilenumbrüche und Kommentare.

4.1. Kommentare

Kommentare werden beim Übersetzungsvorgang ignoriert und haben dementsprechend keine Auswirkung auf die Funktionalität eines Programms. Sie dienen zum Einen zum Erläutern von Quelltext zur besseren Wartbarkeit und zum Anderen zum *Auskommentieren* von Quelltextabschnitten. Es gibt zwei Arten von Kommentaren:

- `//` (doppelter Schrägstrich) symbolisiert einen Zeilenkommentar. Der gesamte Inhalt einer Zeile hinter diesem Symbol wird als Kommentar ausgewertet.
- `/*` (Schrägstrich mit Stern) initiieren einen Blockkommentar, der mit `*/` Abgeschlossen wird. Der gesamte Inhalt zwischen diesen Symbolen, der mehrere Zeilen umfassen kann, wird als Kommentar ausgewertet. Es ist nicht möglich einen Blockkommentar in einen Anderen zu platzieren.

Kommentare, die sich in einem Zeichenkettenliteral befinden, werden als Teil dieses Literals ausgewertet und nicht als Kommentar. In Quelltext 5 ist ein Programm mit Beispielen für die Kommentarfunktion zu sehen. Die Ausgabe des Programms lautet:

```
/* KEIN KOMMENTAR */, // KEIN KOMMENTAR
```

```
1  program kommentare
2
3  import IO as io
4
5  /*
6   * Blockkommentar
7   */
8  main
9      variable var : string // Zeilenkommentar
10     //  var := "Hello, world!"
11     io.writeString(var & "/* KEIN KOMMENTAR */, // KEIN KOMMENTAR")
12 endmain
```

Quelltext 5: Beispiele eines Programms mit Kommentaren.

4.2. Bezeichner

Die Bezeichner erfüllen die Rolle der eindeutigen Identifikation von Deklarationen in einem Namensraum (Abschnitt 2.5). Sie können beliebig lang sein und müssen mit einem kleinen bzw. großen Buchstaben des englischen Alphabets oder mit einem Unterstrich anfangen. Als fortgehende Zeichen können noch Ziffern, zusätzlich zu den bereits erwähnten Symbolen, verwendet werden.

```
1 ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Quelltext 6: Die Grammatikregel für die Bezeichner.

Es muss auf Groß- und Kleinschreibung geachtet werden: `Foo`, `foo` und `FOO` stellen als Beispiel drei unterschiedliche Bezeichner dar. In Quelltext 5 kommen folgende Bezeichner vor:

- `kommentare` ist der Name des Programms. Dieser muss mit dem Namen der Datei übereinstimmen.
- `IO` ist der Name der importierten Bibliothek.
- `io` ist der Alias, unter dem die importierte Bibliothek im Programm referenziert wird.
- `writeString` ist der Name der Operation zum Ausgeben einer Zeichenkette aus der importierten Bibliothek.

- `var` ist der Name der deklarierten Variable.

Außer der Einschränkung, dass ein Bezeichner in einem Namensraum einzigartig sein muss, gibt es eine Reihe von reservierten Wörtern, welche nicht als Bezeichner verwendet werden dürfen. Zum Einen gehören dazu alle MuLE Schlüsselwörter (Abschnitt 4.3), zum Anderen dürfen aus Gründen der Implementierung reservierte Wörter der Programmiersprache Java [3] als Bezeichner nicht verwendet werden.

Folgende Wörter können beispielsweise als Bezeichner verwendet werden:

- `_variable`
- `var1`
- `MAX_VALUE`

Beispiele für nicht legitime Bezeichner:

- `2variable` – beginnt mit einer Ziffer.
- `Tür` – verwendet Sonderzeichen.
- `integer` – reserviertes Wort in MuLE.
- `int` – reserviertes Wort in Java.

4.3. Schlüsselwörter

Schlüsselwörter sind in der Sprache reservierte Wörter und können nicht als Bezeichner verwendet werden. Sie erfüllen bestimmte Funktionen in der Semantik der Sprache.

- **program** – Legt fest, dass es sich bei der Übersetzungseinheit um ein Programm handelt. Nach diesem Schlüsselwort muss der Name des Programms kommen, der mit dem Namen der Datei übereinstimmen muss. Im Quelltext wird eine Hauptprozedur erwartet.
- **library** – Es handelt sich um eine Bibliotheksdatei. Nach diesem Schlüsselwort muss der Name des Bibliothek kommen, der mit dem Namen der Datei übereinstimmen muss. Dieser Name wird beim Importieren der Bibliothek angegeben. Die Hauptprozedur darf im Quelltext nicht vorkommen.
- **import** – Eine Bibliothek wird importiert.
- **as** – Einem Import wird ein Alias vergeben.
- **main** – Start einer Hauptprozedur.

- **endmain** – Ende einer Hauptprozedur.
- **integer** – Primitiver Datentyp für ganze Zahlen.
- **rational** – Primitiver Datentyp für Gleitkommazahlen.
- **string** – Primitiver Datentyp für Zeichenketten.
- **boolean** – Primitiver Datentyp für Wahrheitswerte.
- **reference** – Im Kontext einer Deklaration deutet es auf einen Referenztyp. In einem Ausdruck wird es als Operator (Abschnitt 4.4) verwendet und bedeutet die Erschaffung einer neuen Referenz auf einen Wert.
- **list** – Es handelt sich um einen Listentyp.
- **type** – Leitet eine Typdeklaration ein.
- **endtype** – Schließt eine Typdeklaration ab.
- **composition** – Bei der Typdeklaration handelt es sich um einen Verbund.
- **enumeration** – Bei der Typdeklaration handelt es sich um einen Aufzählungstyp.
- **operation** – Leitet eine Operation ein.
- **endoperation** – Schließt eine Operation ab.
- **attribute** – Deklaration eines Attributs in einem Verbund.
- **parameter** – Deklaration eines Parameters im Kopf einer Operation.
- **variable** – Deklaration einer Variable.
- **return** – Wertrückgabe.
- **exit** – Verlassen der unmittelbar umfassenden Schleife.
- **loop** – Beginn einer `loop`-Anweisung.
- **endloop** – Ende einer `loop`-Anweisung.
- **foreach** – Beginn einer `foreach`-Anweisung. Wird von einer Variablendeklaration gefolgt.
- **in** – Folgt der Variablendeklaration im Kopf der `foreach`-Anweisung und wird von eine Listenangabe gefolgt. Die deklarierte Variable iteriert über die angegebene Liste.
- **do** – Leitet den Rumpf der `foreach`-Anweisung ein.

- **endforeach** – Ende einer `foreach`-Anweisung.
- **if** – Beginn einer `if`-Anweisung.
- **then** – Beginn eines Codeblocks nach einer Bedingung entweder in einem `if`, oder in einem `elseif` Zweig.
- **elseif** – Beginn eines `elseif` Zweigs einer `if`-Anweisung.
- **else** – Beginn eines `else` Zweigs einer `if`-Anweisung.
- **endif** – Ende einer `if`-Anweisung.

4.4. Operatoren und Ausdrücke

Ausdrücke sind zusammenhängende Kombinationen aus einem oder mehreren Werten, Operatoren und Operationsaufrufen, die in den meisten Fällen zur Laufzeit zu einem Wert ausgewertet werden. Zum Beispiel wird der Ausdruck 4 + 3 * 2 als 10 ausgewertet. Dieser Wert kann nun beispielsweise in einem Datenbehälter mit Hilfe einer Zuweisung (Abschnitt 6.2) gespeichert werden. Dieser Beispiel zeigt, dass in MuLE ähnlich wie in der Mathematik und anderen Programmiersprachen Operatoren eine bestimmte Rangfolge haben, mehr dazu in Abschnitt 4.6.

Operatoren sind reservierte Tokens, die eine Operation auf einen oder zwei Operanden anwenden. Operatoren kommen in Ausdrücken und, im Fall des Zuweisungsoperators, in Zuweisungen (Abschnitt 6.2) vor. Bei Operatoren mit zwei Operanden wird die Infixnotation verwendet da diese, aus dem Mathematikunterricht bekannte Notation leichter zu verstehen ist.

- **:=** – Zuweisungsoperator ist der einzige Operator, der als Teil einer Anweisung, nämlich der Zuweisung, in der Sprache vorkommt. Dieser Operator erwartet einen Datenbehälter auf der linken, und einen Ausdruck auf der rechten Seite. Die Typen müssen auf beiden Seiten übereinstimmen.

```
a := "Hello, world!"
```

- **xor** – Logisches exklusives ODER. Erwartet boolesche Werte als Operanden. Das Ergebnis ist `false`, wenn beide Operanden gleichwertig sind und `true` in anderen Fällen.

```
a xor b
```

- **or** – Logisches ODER. Erwartet boolesche Werte als Operanden. Das Ergebnis ist `false`, wenn beide Operanden `false` sind und `true` in anderen Fällen.

```
a or b
```

- **and** – Logisches UND. Erwartet boolesche Werte als Operanden. Das Ergebnis ist `true`, wenn beide Operanden `true` sind und `false` in anderen Fällen.

```
a and b
```

- **not** – Negation eines booleschen Werts. Bei diesem Operator wird die Präfixnotation verwendet.

```
not a
```

- =, / = – Prüfung auf Gleichheit bzw. Ungleichheit für alle Typen. Die Typen müssen auf beiden Seiten übereinstimmen. Liefern Wahrheitswerte als Ergebnis.

```
a = b, a /= b
```

- <, <=, >, >= – Vergleichsoperatoren für numerische Typen. Liefern Wahrheitswerte als Ergebnis.

```
a < b, a <= b, a >= b, a > b
```

- +, - – Addition, Subtraktion. Beide Operanden müssen numerische Typen haben. Das Ergebnis hat den Typ `integer`, wenn beide Operanden den Typ `integer` haben, sonst ist der Typ `rational`.
  Vorzeichen bei numerischen Literalen, wobei es in diesem Fall nur einen Operanden haben kann. Mehrere Vorzeichen dürfen hintereinander geschrieben werden.

```
a + b, a - b, -a, +-b
```

- & – Zeichenkettenkonkatenation. Beide Operanden müssen Zeichenketten sein. Das Ergebnis ist eine Zeichenkette die aus den Werten der beiden Operanden zusammengesetzt ist.

```
a & b
```

- *, / – Multiplikation, Division. Beide Operanden müssen numerische Typen haben. Bei einer Multiplikation hat der resultierende Wert den Typ `integer`, wenn beide Operanden auch den Typ `integer` haben. Das Ergebnis einer Division hat immer den Typ `rational`.

```
a * b, a / b
```

- **mod** – Modulo bzw. Bestimmung des Rests bei einer Division. Beide Operanden müssen numerische Typen haben. Das Ergebnis hat den Typ `integer`, wenn beide Operanden den Typ `integer` haben, sonst ist der Typ `rational`.

```
a mod b
```

- **exp** – Potenzrechnung. Beide Operanden müssen numerische Typen haben. Der resultierende Wert hat den Typ `rational`.

```
a exp b
```

- @ – Dereferenzierung. Erwartet einen Referenztyp und liefert als Ergebnis den referenzierten Wert zurück. Die Postfixnotation wird bei diesem Operator verwendet.

```
a@
```

- **reference** – Im Kontext einer Deklaration deutet es auf einen Referenztyp (Abschnitt 4.3). In einem Ausdruck wird es als ein Operator mit der Präfixnotation verwendet und bedeutet die Erschaffung einer neuen Referenz auf einen Wert.

```
a := reference b
```

- ** – Wiederholungsoperator bei einer Listeninitialisierung. Der linke Operand ist ein ganzzahliger Ausdruck, der die Anzahl der Elemente darstellt. Der rechte Operand ist der zu wiederholende Element. Als Ergebnis wird eine Liste erzeugt. Der Ausdruck muss im Kontext einer Listeninitialisierung stehen.

```
[a ** b]
```

- **..** – Ganzzahliger Bereich bei einer Listeninitialisierung. Die beiden Operanden müssen ganzzahlige Werte sein, der Wert des rechten Operand muss größer oder gleich dem Wert des linken Operand sein. Als Ergebnis wird eine Liste erzeugt. Der Ausdruck muss im Kontext einer Listeninitialisierung stehen.

```
[a .. b]
```

- **.** – Qualifizierter Zugriff auf Elemente von Kompositionen oder Importe. Bei diesem Operator wird die Postfixnotation verwendet. Nach dem Operator muss ein Bezeichner folgen.

```
a.b
```

4.5. Trennsymbole

- <> – Spitze Klammern schließen den Typparameter bei Referenz- und Listentypen bzw. den Bezeichner bei generischen Typen um.

```
reference<string>, list<integer>, generic<T>
```

- [ ] – Eckige Klammern werden bei der Initialisierung von Listen oder Zugriff auf Listenelemente verwendet.

```
[3 ** 42], [1 .. 4], [1, 2, 3]
    list1[1], list2[0][0]
```

- ( ) – Klammerung der Ausdrücke oder Angabe der Parameter einer Operation (sowohl bei der Deklaration, als auch beim Aufruf).

```
(2 * (3 + 1)), (true and (false or true))
   foo(parameter x : integer), foo(42)
```

- , – Aufzählung von Parametern (sowohl bei der Deklaration, als auch beim Aufruf), Literalen eines Aufzählungstyps oder Elementen einer Liste.
  `foo(parameter x : integer, parameter y : integer) foo(42, 5)`
  `type RGB : enumeration RED, GREEN, BLUE endtype`
  `[1, 2, 3]`

4.6. Rang- und Auswertungsfolge der Operatoren und Trennsymbole

Um Mehrdeutigkeiten bei Auswertungen von Ausdrücken zu vermeiden, haben Operatoren eine feste Rangfolge. Gleichrangige Operatoren haben eine Auswertungsfolge. Tabelle 1 zeichnet die Rangfolge und die Assoziativität der Operatoren, wobei Rang 1 die höchste Priorität darstellt.

Manche Operatoren haben keine Auswertungsfolge. Dies liegt daran, dass maximal ein Operator von diesem Typ in einem bestimmten Kontext vorkommen darf. So sind keine mehrere Zuweisungen in einer Anweisung erlaubt, d.h. nur ein Zuweisungsoperator kann verwendet werden. Vergleichsoperatoren erwarten numerische Werte als Operanden und werden zu booleschen Werten ausgewertet, weshalb es nicht möglich ist, einen Ausdruck in der Art `0 < x < 10` zu schreiben. Dieser Ausdruck müsste `0 < x and x < 10` lauten. Bei Listeninitialisierungsoperatoren legt die Grammatik fest, dass eckige Klammern zu jedem einzelnen Listeninitialisierungsaudruck fest dazu gehören, weshalb es nicht möglich ist eine zweidimensionale Liste, also eine Liste mit Listen, als `[2 ** 3 ** 1]` zu initialisieren. Dieser Ausdruck müsste `[2 ** [3 ** 1]]` lauten.

Die Auswertungsfolge bei arithmetischen Operatoren mit zwei Operanden ist *links nach rechts*, d.h. bei einem Ausdruck `9 / 3 / 3` wird zuerst `9 / 3` berechnet und dann das Ergebnis nochmal durch 3 geteilt, was den Erwartungen aus den mathematischen Vorkenntnissen entspricht. Dies bedeutet allerdings auch, dass der Ausdruck `3 exp 2 exp 3` als `(3 exp 2) exp 3` ausgewertet wird, und nicht als `3 exp (2 exp 3)`.

Operatoren mit nur einem Operanden werden in der Reihenfolge ausgewertet, wie nahe sie zum Operanden stehen, d.h. zuerst wird der am Operanden direkt stehende Operator ausgewertet, dann der nächste, usw. Ein Beispiel ist in Quelltext 7 gegeben. In diesem Beispiel wird eine Referenz auf eine weitere Referenz deklariert und initialisiert. Bei der Initialisierung wird zuerst die Referenz auf den Wert zwei erschaffen, und dann die Referenz auf die Referenz, d.h. der Ausdruck in der Zeile 2 wird als `reference (reference 2)` ausgewertet. Bei der Dereferenzierung wird der Ausdruck `a@@` als `(a@)@` ausgewertet.

```
1 variable a : reference<reference<integer>>
2 a := reference reference 2
3 io.writeInteger(a@@)
```

Quelltext 7: Beispiel für unäre Operatoren in einer Anweisung.

Rang	Operator	Beschreibung	Auswertungsfolge
1	@ [] . ()	Dereferenzierung Listenzugriff Memberzugriff Klammerung	links nach rechts
2	not + -	Negation von logischen Ausdrücken Vorzeichen bei arithmetischen Ausdrücken	rechts nach links
3	reference	Erstellen einer Referenz	rechts nach links
4	exp	Potenzierung	links nach rechts
5	* / mod	Multiplikative Operatoren	links nach rechts
6	+ - &	Additive Operatoren	links nach rechts
7	< <= > >=	Vergleichsoperatoren	keine
8	= /=	Gleichheitsoperatoren	links nach rechts
9	and	logisches UND	links nach rechts
10	or	logisches ODER	links nach rechts
11	xor	logisches exklusives ODER	links nach rechts
12	** .. ,	Operatoren zur Initialisierung von Listen	keine
13	:=	Zuweisungsoperator	keine

Tabelle 1: Rang- und Auswertungsfolge der Operatoren und Trennsymbole

Einen besonderen Fall stellen Operatoren dar, die sowohl auf einen als auch auf zwei Operanden angewendet werden können. So wird zum Beispiel der Ausdruck `5--3` als `5-(-3)` ausgewertet.

4.7. Literale

Ein Literal ist die textuelle Repräsentation von Werten mit elementaren Datentypen (Kapitel 5).

4.7.1. `integer` Literale

Ganzzahlige Literale werden als dezimale Zahlen dargestellt. `integer` Literale mit mehreren Ziffern dürfen nicht mit einer Null anfangen, ansonsten ist jede Ziffer im Bereich [0 .. 9] erlaubt. Die Literale sind zusätzlich durch den Wertebereich ([-2147483648 .. +2147483647]) für den Datentyp `integer` begrenzt. Sollte der Wert des Literals sich außerhalb dieses Bereichs befinden, erscheint eine Fehlermeldung und das Programm wird nicht übersetzt.

Beispiele für gültige `integer` Literale:

- 0
- 2147483647
- -2147483648

Beispiele für ungültige `integer` Literale:

- 007
- 2147483648
- -2147483649

4.7.2. `rational` Literale

Literale für Fließkommazahlen benutzen ebenfalls das Dezimalsystem. Die Vorkommastelle darf nicht mit einer Null anfangen, sofern sie aus mehr als einem Ziffer besteht. Für die Komma wird wie üblich bei Programmiersprachen der Punkt verwendet. Die Literale unterstützen die Exponentialschreibweise für besonders große oder kleine Zahlen. So entspricht das Literal `2.5E10` der Zahl 2,5 * 10^{10}. Der Wertebereich für den Datentyp `rational` liegt bei 4.9E-324 für die kleinste positive oder negative Zahl und 1.7976931348623157E308 für die Größte. Für Literale außerhalb dieses Bereichs wird eine Übersetzungsfehlermeldung angezeigt.
Beispiele für gültige `rational` Literale:

- 0.0
- 3.14
- -0.25
- 1.5E23
- 0.5E-2

Beispiele für ungültige `rational` Literale:

- 00.2
- .23
- 1.
- 1.0E
- 1E100

4.7.3. `boolean` Literale

Literale für Wahrheitswerte sind `true` und `false`.

4.7.4. `string` Literale

Literale für Zeichenketten werden mit doppelten Anführungszeichen (") eingegrenzt, zwischen denen jedes Symbol aus der Symbolmenge CP1252[2] verwendet werden darf. Reservierte Wörter, Operatoren und Trennsymbole der Sprache werden in einem `string` Literal nicht als solche erkannt, sondern als Teil der Zeichenkette.

```
1 program stringLiterale
2 import IO as io
3
4 main
5     io.writeString("Hello, world!")            io.writeLine()
6     io.writeString("\"Hello\", \\world\\ \n")
7     io.writeString("\u00C6")                   io.writeLine()
8     io.writeString("\u01A9")
9 endmain
```

Quelltext 8: Beispiele für `string` Literale und escape Sequenzen.

Um doppelte Anführungszeichen (und manche andere Sonderzeichen) innerhalb eines `string` Literals abbilden zu können, muss eine *escape Sequenz* verwendet werden. In Quelltext 8 sind Beispiele für Zeichenkettenliterale mit und ohne escape Sequenzen zu sehen. Die Ausgabe des Programms lautet:

```
Hello, world!
"Hello", \world\
Æ
?
```

[2] http://www.cp1252.com/ - Unicode Kodierungen für Sonderzeichen sind ebenfalls aufgelistet.

Das erste Literal ist eine einfache Zeichenkette, im zweiten Literal werden jedoch doppelte Anführungzeichen verwendet, was durch den Einsatz einer escape Sequenz mit Hilfe des *backslash* Symbols ermöglicht wird. Da dieser Symbol für escape Sequenzen verwendet wird, muss es ebenfalls im Kontext einer escape Sequenz geschrieben werden, sollte man es in einer Zeichenkette abbilden wollen, wie in der Zeile 6 des Beispielprogramms. In der gleichen Zeile wird noch eine escape Sequenz verwendet, die den Zeilenumbruch verursacht. Weiterhin können escape Sequenzen zur Darstellung von Unicode Symbolen eingesetzt werden. Sollte der entsprechende Symbol allerdings nicht von der CP1252 Menge abgedeckt sein, wird es als ein Fragezeichen dargestellt, wie es bei dem letzten Unicode Zeichen (U+01A9 $\rightarrow \Sigma$) der Fall ist.

4.7.5. Literale in Aufzählungstypen

Aufzählungstypen sind die einzigen benutzerdefinierten elementaren Typen in MuLE. Die Literale sind dabei Bezeichner und befolgen dementsprechend die gleichen Regeln, die für Bezeichner gelten (Abschnitt 4.2).

Sie müssen mit einem großen oder kleinen Buchstaben des englischen Alphabets oder mit einem Unterstrich beginnen und können zusätzlich Ziffern im Bereich `0..9` als weitere Zeichen haben. Alle Aufzählungsliterale von allen Aufzählungstypen in einer Übersetzungseinheit befinden sich im globalen Sichtbarkeitsbereich und werden ohne Qualifizierung referenziert. Dies bedeutet, dass alle Literale in einem Programm unterschiedlich benannt sein müssen, selbst wenn sie in unterschiedlichen Aufzählungstypen aufgelistet werden.

5. Datentypen und Werte

Um mögliche Typfehler zur Laufzeit zu vermeiden und Programmieranfänger bereits auf der lexikalischen Ebene mit dem Typsystem zu konfrontieren, wurde auf schwache Typisierung verzichtet. MuLE ist daher streng, statisch und explizit typisiert. Dies bedeutet, dass Datenbehältern bereits bei der Deklaration explizit ein Typ zugewiesen wird und die Prüfung auf Typkorrektheit eines Programms zur Übersetzungszeit geschieht. Implizite Typumwandlungen sind nur von Untertypen in Obertypen erlaubt, z.B. ganze Zahlen in Gleitkommazahlen. In anderen Fällen müssen Typumwandlungen explizit angegeben werden.

MuLE verfügt über vier vordefinierte primitive Datentypen, welche für das Erlernen von Programmieren ausreichen sollten. Es wurde bewusst darauf verzichtet, nur einen Datentyp für numerische Daten zu verwenden, genauso wie auf die Unterstützung eines breiteren Wertebereichs für ganze und Fließkommazahlen. Es muss möglich sein, mit der Sprache die Aspekte der Hardwarelimitierungen demonstrieren zu können.

Die lexikalische Repräsentation der Werte für jeweilige Datentypen ist in Abschnitt 4.7 gegeben.

5.1. Ganze Zahlen

Mit dem Datentyp `integer` lassen sich ganzzahlige Werte im Bereich zwischen -2^{31} und $2^{31}-1$ darstellen, was der *32bit Zweierkomplement* Repräsentation für ganze Zahlen entspricht. Wird ein Literal verwendet, der nicht in den Wertebereich für diesen Typ passt, wird eine Fehlermeldung angezeigt. Bei Rechenoperationen kann es jedoch zu einem Überlauf kommen.

Folgende Operatoren können mit Werten dieses Typs verwendet werden:

- Arithmetische Operatoren `+`, `-`, `*`, `/`, `exp`, `mod`.
- Vergleichsoperatoren `<`, `<=`, `>`, `>=`.
- Gleichheitstestoperatoren `=`, `/=`.

Weiterhin bietet die Standardbibliothek `Mathematics` (Abschnitt 8) unter Anderem eine Reihe von Typspezifischen Funktionen an:

- `getMaxIntegerValue() : integer` – gibt die größte darstellbare ganze Zahl zurück ($2^{31}-1$).
- `getMinIntegerValue() : integer` – gibt die kleinste darstellbare ganze Zahl zurück (-2^{31}).

5.2. Fließkommazahlen

Die Menge der rationalen Zahlen ist bereits aus der Schulzeit bekannt, weshalb für das Schlüsselwort `rational` entschieden wurde, anstatt z.B. `float` (Gleitkommazahl) zu verwenden. Die Werte für diesen Datentyp entsprechen dem 64bit IEEE 754 Format [1]. Die kleinste darstellbare Zahl ist 4.9E-324, die Größte liegt bei 1.7976931348623157E308. Sollte eine Zahl durch 0 geteilt werden, wird der Wert als `Infinity` angezeigt, Gleiches Passiert bei einem Überlauf. In manchen Fällen kann der Wert auch als `NaN` (Not a Number) ausgewertet werden, z.B. bei einer Teilung von 0 durch 0, oder einer Multiplikation von `Infinity` mit 0. Bei einem Unterlauf beträgt der Wert 0.0.

Die Zahlen können sowohl positiv, als auch negativ sein. Es ist anzumerken, dass nicht jede mathematisch berechenbare rationale Zahl durch diesen Typ darstellbar ist. Jeder Wert des `integer` Typs kann verlustfrei in einen `rational` Wert umgewandelt werden.

Folgende Operatoren können mit Werten dieses Typs verwendet werden:

- Vorzeichen und Arithmetische Operatoren `+`, `-`, `*`, `/`, `exp`, `mod`.

- Vergleichsoperatoren `<`, `<=`, `>`, `>=`.
- Gleichheitstestoperatoren `=`, `/=`.

Weiterhin bietet die Standardbibliothek `Mathematics` (Abschnitt 8) unter Anderem eine Reihe von Typspezifischen Funktionen an:

- `getMaxRationalValue() : rational` – gibt die größte darstellbare Fließkommazahl zurück (1.7976931348623157E308).
- `getMinIntegerValue() : rational` – gibt die kleinste darstellbare Fließkommazahl zurück (4.9E-324).

5.3. Zeichenketten

Mit Zeichenketten (`string`) kommt man sofort beim ersten `"Hello, world!"` Programm in Berührung. Streng gesehen ist eine Zeichenkette kein primitiver Typ, da Zeichenketten aus mehreren Zeichen kombiniert werden. Es wurde allerdings auf einen Datentyp `character` verzichtet, um den Sprachumfang zu reduzieren.

Folgende Operatoren können mit Werten dieses Typs verwendet werden:

- Gleichheitstestoperatoren `=`, `/=`.
- Stringkonkatenation `&`.

Die Standardbibliothek `Strings` (Abschnitt 8) erlaubt unter Anderem Werte von anderen Typen in Zeichenketten umzuwandeln (Abschnitt 5.10).

Quelltext 9 demonstriert den Einsatz von Stringkonkatenation, Zeichenkettenvergleich und Konvertierung von Wahrheitswerten in Zeichenketten. Es ist anzumerken, dass die Zahlen im Beispiel als Zeichenketten miteinander verglichen werden, es ist also möglich zu prüfen, dass 42 ungleich 5 (als Zeichenketten) ist, beim Vergleich, ob eines der Werte größer oder kleiner ist, würde eine Übersetzungsfehlermeldung kommen. Die Ausgabe des Programms lautet:

```
42 = 42 and 42 /= 5 is true
```

```
1 program stringsAndBooleans
2
3 import IO as io
4 import Strings as str
5
6 main
7     io.writeString("42 = 42 and 42 /= 5 is "
8             & str.booleanToString("42" = "42" and "42" /= "5"))
9 endmain
```

Quelltext 9: Beispiele für Operationen mit Zeichenketten und Wahrheitswerten.

5.4. Wahrheitswerte

Der Datentyp `boolean` repräsentiert logische Wahrheitswerte. Variablen mit diesem Datentyp können den Wert `true` oder `false` annehmen. Wahrheitswerte werden hauptsächlich zur Bestimmung des Kontrollflusses eines Programms verwendet.

Folgende Operatoren können mit Werten dieses Typs verwendet werden:

- Gleichheitstestoperatoren `=`, `/=`.
- Logische Operatoren `and`, `or`, `xor`, `not`.

5.5. Verbunde

Verbunde stellen eine strukturierte Sammlung von Daten mit potenziell unterschiedlichen Datentypen. Ein Verbund muss im Programm als ein neuer Typ explizit deklariert werden. Hierzu wird mit dem Schlüsselwort `type` eine Typdeklaration angekündigt, gefolgt vom Namen und zusätzlichen Schlüsselwort `composition`, welches sicherstellt, dass es sich um einen Verbund handelt. Im Rumpf der Verbunddeklaration werden die Attribute aufgezählt. Jedes Attribut wird durch das Schlüsselwort `attribute` eingeleitet, besitzt einen Namen sowie einen Typ (Quelltext 10).

```
1 Composition:
2     'type' ID ':' 'composition'
3     (Attribute)*
4     'endtype'
5 ;
6
7 Attribute: 'attribute' ID ':' DataType ;
```

Quelltext 10: Grammatikalische Regeln für Verbunde und Attribute.

In Programmier- und Modellierungssprachen werden Begriffe wie Attribut (*attribute*), Feld (*field*) und Eigenschaft (*property*) des Öfteren als Synonyme, bzw. Begriffe mit geringen Unterschieden, benutzt. In Objektorientierten Sprachen werden *Membervariablen* als Felder bezeichnet und ihre Zugriffsmethoden als *properties*. In UML ist Eigenschaft ein Oberbegriff, dem unter anderem Attribute einer Klasse untergeordnet werden. Da Attribut und Eigenschaft wesentlich verständlicher für einen Programmieranfänger sind als Feld und Eigenschaft eine abstraktere Bedeutung hat wurde in MuLE letztendlich für `attribute` entschieden.

In Quelltext 11 werden Schachfiguren und -felder als Verbunde definiert. Hierbei werden deklarierte Aufzählungstypen für manche Attribute verwendet. Werte von Verbundtypen können miteinander mit Gleichheitsoperatoren verglichen werden, hierbei werden die Werte der jeweiligen Attribute miteinander verglichen.

Die Typen der beiden Werte müssen dabei übereinstimmen. Die Werte sind gleich, wenn alle entsprechenden Werte ihrer Attribute gleich sind. Weiterhin kann ein solcher Wert mit der Operation `genericToString(wert : generic<T>)` der Standardbibliothek `Strings` in eine Zeichenkette konvertiert werden. Beide Operationen kommen im Programm in Quelltext 11 zum Einsatz. Die Ausgabe des Programms lautet:

```
Figure{color = WHITE, figureType = KING}
true
false
```

```
1  program test
2
3  import IO as io
4  import Strings as str
5
6  type Color : enumeration
7      BLACK, WHITE
8  endtype
9
10 type FigureType : enumeration
11     PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING
12 endtype
13
14 type Figure : composition
15     attribute color : Color
16     attribute figureType : FigureType
17 endtype
18
19 type Field : composition
20     attribute color : Color
21     attribute coordX : string
22     attribute coordY : integer
23     attribute figure : Figure
24 endtype
25
26 main
27     variable whiteKing : Figure
28     whiteKing.color := WHITE
29     whiteKing.figureType := KING
30
31     io.writeString(str.genericToString(whiteKing))
32
33     variable whiteKing2 : Figure
34     whiteKing2 := whiteKing
35     variable whiteQueen : Figure
36     whiteQueen.color := WHITE
```

```
37      whiteQueen.figureType := QUEEN
38
39      io.writeBoolean(whiteKing = whiteKing2) io.writeLine()
40      io.writeBoolean(whiteKing = whiteQueen)
41  endmain
```

Quelltext 11: Beispiele für Verbunde und Aufzählungstypen.

5.6. Aufzählungstypen

Aufzählungstypen legen eine begrenzte Menge an Literalen fest, welche als Werte in einem Datenbehälter dieses Typs abgelegt werden können. Da es sich um einen benutzerdefinierten Typ handelt, muss dieser vor der Verwendung deklariert werden. Hierzu wird das Schlüsselwort `type` verwendet, gefolgt vom Namen, wobei das Schlüsselwort `enumeration` für den Aufzählungstyp steht. Im Rumpf der Deklaration werden die Literale, von denen mindestens eins vorhanden sein muss, getrennt mit Kommas aufgelistet. Beispiele sind in Quelltext 11 zu sehen.

```
1  Enumeration:
2      'type' ID ':' 'enumeration'
3      EnumerationValue (',' EnumerationValue)*
4      'endtype' ;
5
6  EnumerationValue: ID;
```

Quelltext 12: Grammatikalische Regeln für Aufzählungstypen.

Die Werte von Aufzählungstypen können mit den `=` und `/=` auf Gleichheit getestet werden. Außerdem kann ein solches Wert in eine Zeichenkette umgewandelt werden.

5.7. Listen

Bei dem Entwurf von MuLE wurde dagegen entscheiden, separat Arrays und Listen zu implementieren. Stattdessen wurden die Listen so entworfen, dass sie analog wie Arrays in Java oder C verwendet werden können und gleichzeitig über die Flexibilität einer Listen-ähnlichen Datenstruktur verfügen. Alle Elemente einer Liste müssen den gleichen Datentyp haben.

In Quelltext 13 sind die unterschiedlichen Arten eine Liste zu initialisieren zu sehen. Somit gibt es drei Wege eine Liste zu erzeugen:

- Liste mit Elementen – bei der die Elemente mit Kommas getrennt nacheinander aufgelistet werden. Einen Sonderfall stellt dabei eine leere Liste dar.

- Integer Bereich – zählt ganze Zahlen von Unter- bis Obergrenze, die Grenzen sind dabei eingeschlossen. Die Syntax ist folgende `[Untergrenze .. Obergrenze]`.
- Wiederholung – hat folgende Syntax: `[Anzahl ** Element]`. Das ermöglicht eine einfache Erstellung von besonders großen Listen, die an jeder Stelle den gleichen Inhalt haben.

Mehrdimensionale Listen sind möglich, die Anzahl der Dimensionen ist in der Sprache unbegrenzt. Hierfür wird eine Mehrdimensionale Liste als eine eindimensionale Liste mit einer weiteren Liste als Typparameter deklariert. Bei der Initialisierung von mehrdimensionalen Listen können unterschiedliche Schreibweisen vermischt, sowie bereits vorhandene Listen wiederverwendet werden.

Für den Zugriff auf Listenelemente wird nach dem Bezeichner eines Datenbehälters der Index des Elements in der Liste in eckigen Klammern geschrieben. Hierbei wird eine Kopie des Eintrags an der angegeben Position zurückgegeben. Beispiele sind in Quelltext 13 in Zeilen 11 und 25 gegeben. Es ist anzumerken, dass es nicht erlaubt ist, den Eintrag von einer Listeninitialisierung abzufragen, d.h. `[1 .. 9][0]` würde nicht die 1 zurückgeben, stattdessen käme ein Syntaxfehler.

Es ist möglich Listen miteinander auf Gleichheit zu prüfen und in eine Zeichenkette zu konvertieren. Beispiele dafür sind ebenfalls in Quelltext 13 gegeben. Die Ausgabe des Programms lautet:

```
true
true
[[0, 1, 2], [3, 3], [4, 5, 6]]
```

Listen sind gleich, wenn alle Elemente gleichen Typ und gleiche Werte haben und in gleicher Reihenfolge angeordnet sind.

```
1  program lists
2
3  import IO as io
4  import Strings as str
5
6  main
7      variable empty : list<string>
8      empty := []
9      variable elements : list<string>
10     elements := ["a", "b", "c"]  // ["a", "b", "c"]
11     elements[1] := "x"           // ["a", "x", "c"]
12     variable repetition : list<rational>
13     repetition := [3 ** 3.14]    // [3.14, 3.14, 3.14]
14
15     variable list1 : list<integer>
```

```
16      list1 := [0 .. 2]
17      variable list2 : list<integer>
18      list2 := [3, 4, 5]
19      variable lists : list<list<integer>>
20      lists := [list1, list2]      // [[0, 1, 2], [3, 4, 5]]
21      lists := [[0, 1, 2], [2 ** 3], [4 .. 6]]
22      // [[0, 1, 2], [3, 3], [4, 5, 6]]
23
24      io.writeBoolean(list1 /= list2)  io.writeLine()
25      io.writeBoolean(list1 = lists[0])  io.writeLine()
26      io.writeString(str.genericToString(lists))
27  endmain
```

Quelltext 13: Beispiele für Initialisierungen von MuLE-Listen.

5.8. Referenztypen

Um rekursive Datentypen oder Graphenähnliche Strukturen zu implementieren, werden Referenzen benötigt, weshalb es notwendig war, Referenztypen zu implementieren. Nach der Deklaration haben die Variablen mit Referenztypen standardmäßig den Wert `null` und müssen vor ihrer Verwendung manuell initialisiert werden.

Die Referenzen sind typisiert, z.B. akzeptiert eine `reference<integer>` nur Referenzen auf ganze Zahlen als referenzierte Werte. Neue Referenzen werden explizit mit dem Schlüsselwort `reference`, gefolgt von einem Wert, angelegt. Um den Wert einer Referenz zu bekommen, muss die Referenz mit dem `@` Operator dereferenziert werden. Beim Dereferenzieren ist auf `null`-Referenzen zu achten. Manuelle Speicherfreigabe ist nicht möglich, diese Aufgabe wird vom Garbage Collector übernommen, wodurch unter anderem das Problem der hängenden Referenzen entfällt.

Ein Beispiel für die Verwendung von Referenztypen ist in Quelltext 14 zu sehen. Es werden zwei Variablen as Referenztypen deklariert und initialisiert. Ihre Werte werden darauffolgend in einer Prozedur vertauscht. Nach dem Ausführen der Prozedur sind die referenzierten Werte der beiden Variablen auch im Hauptprogramm vertauscht.

```
1   program referenzen
2
3   import IO as io
4
5   operation swap(parameter a : reference<integer>,
6                  parameter b : reference<integer>)
7       variable c : integer
8       c := a@
9       a@ := b@
10      b@ := c
```

```
11 endoperation
12
13 main
14     variable a : reference<integer>
15     variable b : reference<integer>
16     a := reference 42
17     b := reference 5
18     swap(a, b)
19     io.writeString(a@ as string & " " & b@ as string & "\n")
20 endmain
```

Quelltext 14: Beispiel für Prozeduren mit Einsatz von Referenztypen.

Quelltext 15 demonstriert das Verhalten bei Tests auf Gleichheit bei Referenzen. Die Ausgabe des Programms ist:

```
false
true
true
```

Bei diesen Tests werden die Adressen der Referenzen verglichen, weshalb beim ersten Test false angezeigt. Die Referenzen haben zwar die gleichen Werte (weshalb `ref1@ = ref2@` als `true` ausgewertet wird), es sich jedoch zwei Referenzen mit unterschiedlichen Adressen. Nachdem `ref2` der Wert von `ref1` zugewiesen wird, wird der Ausdruck `ref1 = ref2` als `true` ausgewertet.

```
1  program referenzen2
2
3  import IO as io
4
5  main
6      variable ref1 : reference<integer>
7      variable ref2 : reference<integer>
8
9      ref1 := reference 42
10     ref2 := reference 42
11     io.writeBoolean(ref1 = ref2)  io.writeLine()
12     io.writeBoolean(ref1@ = ref2@)  io.writeLine()
13
14     ref2 := ref1
15     io.writeBoolean(ref1 = ref2)  io.writeLine()
16 endmain
```

Quelltext 15: Gleichheit bei Referenzen.

```
1  program referenzen3
2
3  import IO as io
4
```

```
5  main
6      variable ref : reference<integer>
7      variable num : integer
8
9      num := 42
10     ref := reference num
11     num := 21
12
13     io.writeInteger(ref@)
14 endmain
```

Quelltext 16: Kopiersemantik bei Referenzen.

Quelltext 16 demonstriert die Kopiersemantik (Kapitel 10) beim Erstellen von Referenzen. Der `integer`-Variable `num` wird ein Wert zugewiesen, woraufhin der `reference<integer>`-Variable `ref` eine Referenz auf die Variable `num` zugewiesen wird. Im Hintergrund wird eine Kopie des Werts von `num` erstellt, und diese Kopie wird danach referenziert. Deshalb ändert sich der Wert der Referenz nicht wenn der Variable `num` ein neuer Wert zugewiesen wird. Die Ausgabe des Programms ist 42.

5.9. Generische Typen

Generische Typen erlauben es Datenstrukturen und Operationen, die mit unterschiedlichen Typen arbeiten können, zu definieren. Der generische Typ hat einen beliebigen Bezeichner, womit zwischen mehreren generischen Typen in einem Kontext unterschieden werden kann. Zur Laufzeit wird der generische Typ durch einen aktuellen Typ ersetzt.

Als Beispiel kann der vordefinierte MuLE Listentyp, welcher einen beliebigen Typ als Typparameter akzeptiert, betrachtet werden. Die Listenbibliothek beinhaltet eine Reihe von Operationen, deren eine Liste als einer der Parameter übergeben werden. Der Typ dieser Liste ist jeweils `list<generic<T>>`, d.h. die Liste akzeptiert einen generischen Typ mit dem Bezeichner `T`. Auf diese Weise kann der Operation jede mögliche Liste übergeben werden.

Der Quelltext 17 demonstriert ein Beispiel, bei dem die Listenoperation `append` aus der Listenbibliothek verwendet wird. Diese Operation hat folgende Signatur: `append(l : list<generic<T>>, e : generic<T>) : list<generic<T>>`. Ihr werden zwei Parameter übergeben: eine Liste und ein Element, welches der Liste hinzugefügt werden soll. Hierbei hat der Element denselben generischen Typ, der auch der Liste als Typparameter übergeben wird. Der Rückgabetyp der Operation hat denselben Typ, wie die übergebene Liste. Im Beispiel im Quelltext 17 werden zwei Listen mit unterschiedlichen Typparametern angelegt und es wird jeweils die gleiche `append` Operation verwendet. Der generische Typ wird einmal durch `string` und einmal durch `integer` ersetzt.

```
1  program genericList
```

```
2  import Lists as lst
3
4  main
5      variable listOfStrings : list<string>
6      listOfStrings := ["a", "b", "c"]
7      listOfStrings := lst.append(listOfStrings, "d")
8
9      variable listOfIntegers : list<integer>
10     listOfIntegers := [1, 2, 3]
11     listOfIntegers := lst.append(listOfIntegers, 4)
12 endmain
```

Quelltext 17: Beispiel für verwenden von Listenoperationen.

5.10. Typkonvertierungen

Implizite Konvertierungen sind nur von Untertypen in Obertypen möglich. Beispielsweise wird `integer` als ein Untertyp von `rational` verstanden, d.h. wird eine ganze Zahl in einer Additionsanweisung mit einer Gleitkommazahl implizit in eine Gleitkommazahl umgewandelt. Bei dem prozeduralen Teil der Sprache ist dies auch der einzige erlaubte Fall für implizite Konvertierungen.

Standardbibliotheken (Abschnitt 8) bieten Operationen an, bestimmte Typen in Andere zu konvertieren. Folgende Möglichkeiten gibt es:

- `Strings` – `integerToString(wert : integer) : string` wandelt eine ganze Zahl in eine Zeichenkette um.
- `Strings` – `rationalToString(wert : integer) : string` wandelt eine Fließkommazahl in eine Zeichenkette um.
- `Strings` – `booleanToString(wert : boolean) : string` wandelt einen Wahrheitswert in eine Zeichenkette um.
- `Strings` – `genericToString(wert : generic<T>) : string` wandelt jedes beliebige Wert, z.B. eine Liste oder Komposition, in eine Zeichenkette um (Quelltext 11). Diese Operation macht die Anderen scheinbar überflüssig. Allerdings bieten sich die anderen beiden Operationen vor allem in den Anfängen einer Programmierveranstaltung an, um die Anzahl der Konzepte, die den Studierenden auf einmal erklärt werden, gering zu halten.
- `Mathematics` – `floor(wert : rational) : integer` – konvertiert eine Fließkommazahl in eine ganze Zahl, indem es die Nachkommastelle verwirft.
- `Mathematics` – `round(wert : rational) : integer` – konvertiert eine Fließkommazahl in eine ganze Zahl, indem es die Fließkommazahl rundet.

5.11. Variablen, Parameter, Attribute

Werte und Referenzen können in Datenbehältern gespeichert oder von einer Funktion zurückgegeben werden. Bei den Datenbehältern handelt es sich um Variablen, Parameter und Attribute. Parameterwerte werden bei dem Aufruf einer Operation gesetzt, Parameter müssen dementsprechend im Kopf einer Operation vorher deklariert worden sein. Variablen und Attributen werden Werte zugewiesen, ansonsten erhalten sie nach ihrer Deklaration einen typspezifischen Standardwert. Variablen werden im Kontext einer Operation oder der Hauptprozedur deklariert, Attribute sind Bestandteile von Kompositionen (Abschnitt 5.5).

MuLE verwendet Kopiersemantik (Abschnitt 10) bei Zuweisungen, Parameterübergabe und Wertrückgabe. Hierbei wird stets eine tiefe Kopie von einem Wert erzeugt, die dann im entsprechenden Kontext verwendet wird. Dadurch werden für Anfänger schwer nachvollziehbare Seiteneffekte vermieden. Der explizite Referenztyp ermöglicht es dem Benutzer, Prozeduren mit Seiteneffekten zu implementieren.

6. Anweisungen

Anweisungen sind ein zentrales Sprachkonstrukt in prozeduralen Programmiersprachen, welche einzelne, potentiell zustandsverändernde, Schritte eines Programms beschreiben. In MuLE müssen die Anweisungen immer Teil eines Blocks sein, z.B. im Rumpf des Hauptprogramms. Ein Block kann mehrere Anweisungen beinhalten.

Ausdrücke werden ausgewertet und liefern einen Wert zurück. Somit verändern die Ausdrücke an sich den Zustand nicht und müssen im Kontext einer Anweisung eingesetzt werden. Den einfachsten und besten Beispiel stellt die Zuweisung dar, eine Anweisung die den Wert eines Datenbehälters (Abschnitt 5.11) setzt, wobei zuerst ein Ausdruck ausgewertet werden muss, um den entsprechenden Wert zu bekommen.

```
1  Statement:
2          VariableDeclaration
3      |   Assignment
4      |   IfStatement
5      |   LoopStatement
6      |   ForEachStatement
7      |   'return' Expression
8      |   'exit';
9
10 Assignment:
11     FeatureCall (':=' Expression)?;
```

Quelltext 18: Grammatikregeln für Anweisungen.

Mögliche Anweisungen sind in Quelltext 18 aufgelistet. Die Regel für Zuweisungen (`Assignment`) ist so definiert, um Mehrdeutigkeiten in der Grammatik der Sprache zu vermeiden. Demnach können Operationen als `FeatureCall` (dazu gehören Verweise auf Datenbehälter und Operationen) auf der Anweisungsebene aufgerufen werden. Sobald dem `FeatureCall` ein Zuweisungsoperator folgt, handelt es sich bei der Anweisung um eine Zuweisung. Validierungsregeln müssen dafür sorgen, dass keine Variablenreferenzen lose (nicht im Kontext einer Zuweisung oder eines Ausdrucks) im Quelltext vorkommen. Weiterhin unterstützt MuLE Variablendeklarationen, bedingte Anweisungen, zwei Arten von Schleifen, Wertrückgabeanweisungen und Abbruchsanweisungen.

6.1. Variablendeklaration

Die Variablen müssen zunächst deklariert werden, bevor sie mit einer Zuweisung initialisiert werden können. Deklaration mit gleichzeitiger Wertzuweisung durch Benutzer wurde nicht implementiert, um diese Konzepte voneinander zu trennen. Eine Mehrfachdeklaration von Variablen ist nicht möglich. Nach der Deklaration erhalten die Variablen einen Standardwert gemäß ihres Datentyps (das Gleiche gilt für Attribute):

- `integer` – `0`
- `rational` – `0.0`
- `string` – leere Zeichenkette
- `boolean` – `false`
- `enumeration` – bekommt das erste Literal des Aufzählungstyps zugewiesen.
- `composition` – ein Verbund wird erzeugt, dessen Attribute wiederum mit Standardwerten belegt werden.
- `list<DataType>` – leere Liste
- `reference<DataType>` – `null`-Referenz

```
1 VariableDeclaration: 'variable' ID ':' DataType;
```

Quelltext 19: Grammatikregel für Variablendeklarationen.

6.2. Zuweisung

Zuweisungen sind zustandsverändernde Anweisungen, die dazu dienen, Werte von Datenbehältern zu verändern. Bei einer Zuweisung steht wie üblich links der Bezeichner eines Datenbehälters und rechts der Ausdruck, dessen Wert ausgewertet und im Datenbehälter gespeichert wird. Eine Mehrfachzuweisung ist zur besseren Übersichtlichkeit nicht erlaubt. Bei Zuweisungen wird stets die Kopiersemantik verwendet, d.h. es wird eine Kopie des Werts einem Datenbehälter zugewiesen.

Als Zuweisungssymbol wurde `:=` gewählt, welches im mathematischen Kontext als *wird definiert als* Operator verstanden wird. Es wurden weitere Alternativen in Betracht gezogen und aus diversen Gründen jeweils verworfen:

- `=` - das Gleichheitssymbol, welches in vielen C-ähnlichen Programmiersprachen als Zuweisungsoperator verwendet wird, kann bei Anfängern für Verwirrung sorgen. In der Mathematik bedeutet die Gleichheit, dass beide Seiten einer Gleichung gleich sind. Dabei ist `a = a + 1` eine legitime Operation in einem Programm und bedeutet den Inkrement des Werts der Variable `a` um 1. Außerdem ist bei der Verwendung von '=' für einige am Anfang unklar, in welche Richtung zugewiesen wird. Weiterhin kann '=' nicht mehr für die Überprüfung einer Gleichheit eingesetzt werden, für die in solchen Sprachen der Operator '==' verwendet wird. Die Anfänger neigen dazu, für die Gleichheit weiterhin den '=' zu nutzen, was bei solchen Sprachen zu schwer erkennbaren semantischen Fehlern führen kann.

- `<--` - sieht dem Zuweisungspfeil aus algorithmischen Notationen ähnlich aus, was allerdings in der Praxis dazu führte, dass die Studierenden den Operator zunächst mit dem Vergleich mit einer negativen Zahl verwechselten.

- `<<` - wäre eine kürzere Alternative zu `<--`, allerdings wurde es verworfen, um Verwechslungen mit dem *left shift* Operator aus anderen Sprachen bzw. mit dem *kleiner als* Operator zu vermeiden.

- `is` - oder eine ähnliche textuelle Variante für den Zuweisungsoperator wurde ebenfalls als eine Alternative betrachtet. Diese Variante wurde letztendlich verworfen, da die Syntax dadurch zu stark an natürliche Sprachen angelehnt wäre.

```
1 main
2     variable var1 : integer
3     variable var2 : rational
4     variable var3 : string
5     variable var4 : boolean
6
7     var1 := 42
```

```
8       var2 := 3.14
9       var3 := "Hello World!"
10      var4 := true
11
12      variable ref1 : reference<integer>
13      variable ref2 : reference<integer>
14      ref1 := reference var1
15      ref2 := ref1
16  endmain
```

Quelltext 20: Beispiele für Variablendeklarationen und Zuweisungen.

Quelltext 20 demonstriert Beispiele für Variablendeklarationen und Zuweisungen für vordefinierte elementare Typen und Referenztypen.

6.3. `if`-Anweisung

Eine der Kontrollstrukturen in MuLE ist die `if`-Anweisung, bei welcher abhängig von einer Bedingung bestimmte Codeblöcke ausgeführt oder übersprungen werden. Dabei werden folgende Schlüsselwörter verwendet:

- `if` - nach diesem Schlüsselwort wird die Bedingung erwartet.
- `then` - gefolgt von einem Codeblock, welcher ausgeführt wird, sollte die vorherige Bedingung WAHR sein.
- `elseif` - stellt eine zusätzliche Bedingung für den Fall, dass vorherige nicht zutreffend waren. Nach der Bedingung folgt wieder ein `then` Bereich. Beliebig viele `elseif` Konstrukte sind innerhalb einer `if`-Anweisung erlaubt.
- `else` - ist optional. Der entsprechende Codeblock wird ausgeführt, falls alle vorherigen Bedingungen FALSCH waren.

```
1  IfStatement:
2      'if' Expression 'then' Statement*
3      ('elseif' Expression 'then' Statement*)*
4      ('else' Statement*)?
5      'endif';
```

Quelltext 21: Grammatikalische Regel für `if`-Anweisungen.

```
1  if a then
2      // if block
3  elseif b < 5 then
4      // elseif block
5  elseif b = 5 and not a then
6      // elseif block
```

```
else
    // else block
endif
```

Quelltext 22: Beispiel für eine if-Anweisung.

Durch das `endif` Schlüsselwort wird eine bedingte Anweisung explizit abgeschlossen. Damit wird auch unter anderem das *dangling else* Problem eliminiert, da der jeweilige `else` Block problemlos zugeordnet werden kann. Obwohl ein `elseif` Sprachkonstrukt durch Verschachtlung von mehreren bedingten Anweisungen dargestellt werden kann, wurde nicht darauf verzichtet, um Bedingungskaskaden eleganter zu machen. Dadurch kann eine Mehrfachverzweigung auf eine übersichtliche Weise simuliert werden, was zu der Entscheidung führte, auf einen `switch ... case` Sprachkonstrukt zu verzichten.

6.4. `loop`-Anweisung

MuLE bietet eine loop-Anweisung an, welche in ihrer Grundform eine Endlosschleife darstellt. Die Schleife kann mit einer `exit`-Anweisung an jeder Stelle verlassen werden, mehrere `exit`-Anweisungen innerhalb einer Schleife sind möglich. Die Abbruchbedingung muss mit einer `if`-Anweisung geprüft werden (Quelltext 24). Es lassen sich sowohl durchlaufende, als auch abweisende Schleifen simulieren.

```
LoopStatement:
    'loop'
    Statement*
    'endloop';
```

Quelltext 23: Grammatikalische Regel für `loop`-Anweisungen.

```
loop
    io.writeString("Geben Sie ein N ein,
    um die Schleife zu verlassen: ")
    if (io.readString() = "N") then
        exit
    endif
endloop
```

Quelltext 24: Beispiel für eine `loop`-Anweisung.

6.5. `foreach`-Anweisung

Zum Iterieren durch Listen wird in MuLE eine `foreach`-Anweisung angeboten. Im Kopf dieser Anweisung wird eine Laufvariable definiert, die bei jedem Durchlauf der Schleife die Werte einer gegebenen Liste annimmt. Es wäre möglich,

diese Schleife durch eine `loop`-Anweisung zu simulieren, was allerdings wesentlich umständlicher wäre. Daher wurde entschieden, auf diese Anweisung nicht zu verzichten. Somit bietet die Sprache explizit eine Möglichkeit für finite Iteration zusätzlich der infiniten Iteration mittels einer `loop`-Schleife.

```
ForEachStatement:
    'foreach' VariableDeclaration 'in' Expression 'do'
    Statement*
    'endforeach';
```

Quelltext 25: Grammatikalische Regel für `foreach`-Anweisungen.

Der Definition in Quelltext 25 nach, folgt nach dem Schlüsselwort `in` ein Ausdruck. Zusätzliche Validierungsregeln prüfen, dass es bei diesem Ausdruck sich entweder um eine Listendefinition oder um ein Verweis auf eine bereits deklarierte Liste handelt. In Quelltext 26 ist ein Beispiel für eine `foreach`-Schleife demonstriert. Bei diesem Beispiel wird 0123 ausgegeben.

```
foreach variable i : integer in [0 .. 3] do
    io.writeInteger(i)
endforeach
```

Quelltext 26: Beispiel für eine `foreach`-Anweisung.

Der `foreach`-Schleife kann eine bereits existierende Liste übergeben werden. Wird diese Liste innerhalb der Schleife verändert, ändert dies nichts am Verhalten der Schleife. Bei der Ausführung wird weiterhin der Initialwert der übergebenen Liste verwendet. Dies kann am Beispiel in Quelltext 27 demonstriert werden. Die Liste im Beispiel wird mit dem Wert `[1, 2, 3, 4]` initialisiert und an eine `foreach`-Schleife übergeben. Im Rumpf der Schleife wird die Liste bei jedem Verlauf mit dem Wert `[0]` überschrieben und der Wert der Variable `i` wird ausgegeben. Die Ausgabe lautet:

```
Value of _list: [0], Value of i: 1
Value of _list: [0], Value of i: 2
Value of _list: [0], Value of i: 3
Value of _list: [0], Value of i: 4
```

```
variable _list : list<integer>
_list := [1, 2, 3, 4]
foreach variable i : integer in _list do
    _list := [0]
    io.writeString("Value of _list: " & str.genericToString(_list) & ",
        Value of i: ")
    io.writeInteger(i) io.writeLine()
endforeach
```

Quelltext 27: Änderung der Liste innerhalb einer `foreach`-Anweisung.

Daran erkennt man, dass der Wert der Liste zwar sofort verändert wurde, die iterierende Variable jedoch bei jedem Durchlauf die Werte aus dem ursprünglichen Inhalt der Liste annimmt. Dies mag etwas kontraintuitiv erscheinen, vermeidet dafür mögliche Fehler und potentielle Fälle von unerwartetem Verhalten. Dadurch kann z.B. ein einfaches Sortieralgorithmus (Quelltext 28) implementiert werden, bei dem jeweils ein Minimum aus einer Liste in eine andere Liste eingefügt wird und aus der Originalliste entfernt wird, wobei man sich mit zwei geschachtelten `foreach`-Schleifen über die Originalliste iteriert. Zum Schluss ist die Originalliste leer und man erhält eine neue sortierte Liste. Hierbei wird der Minimum jeweils in der äußeren Schleife entfernt, wodurch die Anzahl der Durchläufe der inneren Schleife immer kleiner wird, das Verhalten der äußeren Schleife bleibt aber unverändert. Die Ausgabe der Programms lautet:

```
Original list before sorting: [99, 98, 31, 70, 29, 12, 6, 88, 8, 13]
Original list: [99, 98, 31, 70, 29, 12, 88, 8, 13]
Sorted list: [6]
Original list: [99, 98, 31, 70, 29, 12, 88, 13]
Sorted list: [6, 8]
Original list: [99, 98, 31, 70, 29, 88, 13]
Sorted list: [6, 8, 12]
Original list: [99, 98, 31, 70, 29, 88]
Sorted list: [6, 8, 12, 13]
Original list: [99, 98, 31, 70, 88]
Sorted list: [6, 8, 12, 13, 29]
Original list: [99, 98, 70, 88]
Sorted list: [6, 8, 12, 13, 29, 31]
Original list: [99, 98, 88]
Sorted list: [6, 8, 12, 13, 29, 31, 70]
Original list: [99, 98]
Sorted list: [6, 8, 12, 13, 29, 31, 70, 88]
Original list: [99]
Sorted list: [6, 8, 12, 13, 29, 31, 70, 88, 98]
Original list: []
Sorted list: [6, 8, 12, 13, 29, 31, 70, 88, 98, 99]
```

```
1 variable l : list<integer>
2 variable l2 : list<integer>
3 foreach variable i : integer in [1 .. 10] do
4     l := lst.append(l, math.randomInteger(1, 100))
5 endforeach
6 io.writeString("Original list before sorting: " &
      str.genericToString(l) & "\n")
7 foreach variable j : integer in l do
8     variable min : integer
```

```
9     min := 101
10    foreach variable i : integer in l do
11        if i < min then
12            min := i
13        endif
14    endforeach
15    l2 := lst.append(l2, min)
16    l := lst.removeElement(l, min)
17    io.writeString("Original list: " & str.genericToString(l) & "\n")
18    io.writeString("Sorted list: " & str.genericToString(l2) & "\n")
19 endforeach
```

Quelltext 28: Sortieralgorithmus mit geschachtelten `foreach`-Schleifen.

7. Aufbau eines MuLE Programms

In MuLE wird generell zwischen Programm- und Bibliotheksdateien unterschieden. Im Unterschied zu einer Bibliothek besitzt eine MuLE Programmdatei ein Hauptprogramm, welches als Einstiegspunkt in ein MuLE Programm dient. Somit lässt sich zum Einen ein größeres Programm in kleinere Dateien trennen, zum Anderen ist es möglich, wiederverwendbare Bibliotheken anzulegen. Die Bibliotheken sind nicht alleine lauffähig und müssen importiert werden.

In Quelltext 29 ist ein Ausschnitt aus der MuLE Grammatik zu sehen. In diesem Ausschnitt werden die Regeln, die den Aufbau des eines MuLE Programms festlegen, sowie die unterstützten Datentypen definieren, aufgelistet.

```
1  CompilationUnit:
2      ('program' | 'library') ID
3      Import*
4      ProgramElement*
5      MainProcedure?;
6
7  Import:
8      'import' [CompilationUnit] 'as' ID;
9
10 MainProcedure:
11     'main' Block 'endmain';
12
13 ProgramElement:
14     TypeDeclaration | Operation;
15
16 TypeDeclaration:
17     Composition | Enumeration;
```

Quelltext 29: Aufbau einer MuLE Datei und Regeln für Datentypen.

Demnach wird zuerst festgelegt, ob es sich um eine Programm- oder eine Bibliotheksdatei handelt gefolgt von einem Namen, welcher mit dem Dateinamen übereinstimmen muss. Nachfolgend werden Importe deklariert, gefolgt von beliebig vielen Programmelementen, bei denen es sich um Operationen (Abschnitt 7.3) und Typdeklarationen handelt. Benutzerdefinierte Typen sind Verbunde (Abschnitt 5.5) und Aufzählungstypen (Abschnitt 5.6). Anschließend kann eine Hauptprozedur kommen, was davon abhängt, ob es sich um eine Bibliothek oder ein Programm handelt. Ein `"Hello, World!"` Beispielprogramm ist in Quelltext 30 zu sehen.

```
1 program HelloWorld
2
3 import IO as io
4
5 main
6     io.writeString("Hello, world!")
7 endmain
```

Quelltext 30: Beispiel für ein "$Hello, World$!" Programm.

7.1. Importe

Um Inhalte aus Bibliotheken verwenden zu können, müssen diese Bibliotheken explizit importiert werden. Bei einem Import wird der Name einer Bibliothek angegeben sowie ein Alias, unter dem diese Bibliothek im Quelltext referenziert wird. In Quelltext 30 sieht man den Import der Standardbibliothek `IO`, welche Operationen für Ein- und Ausgabe beinhaltet (mehr in Abschnitt 8). Dieser Import bekommt den Alias `io`, welcher dann zum aufrufen der Operation `writeString` verwendet wird.

```
1 Import:
2     'import' [Program] 'as' ID;
```

Quelltext 31: Grammatikalische Regel für Importe.

Es ist möglich, existierende Java Klassen als Bibliotheken für MuLE verwenden zu können. Bereitgestellte Standardbibliotheken (Kapitel 8) sind in Wirklichkeit mit Java implementiert und verfügen über eine MuLE Schnittstelle, welche in einem MuLE Programm importiert wird.

7.2. Blöcke und die Hauptprozedur

Codeblöcke dienen dazu, Anweisungen zu gruppieren und den Quelltext somit strukturiert zu halten. Typischerweise ist ein Block der Rumpf eines Elements, z.B. einer Operation oder einer Schleife. Eine weitere Funktion von Blöcken ist die Bestimmung von Scope. In den existierenden Sprachen treten unterschiedliche lexikalische Variationen auf, um Blöcke einzugrenzen:

- **Markierung durch Klammerung** – kommt vor allem bei C-ähnlichen Programmiersprachen vor. Hierbei wird ein Block durch geschweifte Klammern ($'\{'$ und $'\}'$) gekennzeichnet. Der Vorteil dieser Lösung ist, dass Blöcke eindeutig markiert sind. Der Schreibaufwand bleibt dabei sehr gering. Nicht geschlossene Blöcke werden zur Übersetzungszeit erkannt. Allerdings führt dies relativ oft zu Fehlern bei den Programmieranfängern, die dazu neigen zu wenig schließende Klammern zu schreiben. Kombiniert mit Nichteinhaltung von Quellcodekonventionen und des Öfteren wenig aussagekräftigen Fehlermeldungen sind Programmieranfänger erfahrungsgemäß öfters nicht in der Lage, solche Fehler eigenhändig auszubessern.

- **Markierung durch Einschübe** – kommt z.B. bei Python vor. Dabei wird ein zusammenhängender Block durch die gleiche Anzahl von Einschüben vor jeder dazugehörigen Anweisung gekennzeichnet. Dies hat den Vorteil, dass keine Fehler zur Übersetzungszeit durch fehlerhafte Klammerung entstehen können. Der Schreibaufwand ist am geringsten und es wird eine einheitliche Quelltextformatierung gezwungen, was zum besser lesbaren Code führt. Allerdings kann dies bei falscher Einrückung zur fehlerhaften Ergebnissen zur Laufzeit führen. Solche Fehler sind für Programmieranfänger noch schwerer auszubessern, da sie manchmal gar nicht erkennen, dass ein Fehler vorliegt.

- **Markierung durch Schlüsselwörter** – kommt vor allem bei ALGOL-ähnlichen Sprachen vor. Hierbei kann ein Block durch Schlüsselwörter `begin` und `end` oder alternativ durch das umgedrehte Startschlüsselwort eines Sprachkonstrukts, z.B. `if ... then ... fi`, gekennzeichnet werden. Vor allem die letztere Variante hat den Vorteil, dass es für Anfänger leichter zu erkennen ist, an welche Stelle der Block eines bestimmten Sprachkonstrukts endet [4]. Des Weiteren werden nicht geschlossene Blöcke analog zur ersten Alternative bereits zur Übersetzungszeit erkannt, wobei eine verständlichere Fehlermeldung bedingt durch den Aufbau der Grammatik ausgegeben werden kann. Als Nachteil bringt diese Variante einen erhöhten Schreibaufwand mit sich.

Da MuLE nicht darauf spezialisiert ist, mit wenig Schreibaufwand zu programmieren, sondern für Anfänger verständlich zu sein, wurde eine an die letzte Variante angelehnte Lösung entwickelt. In Quelltext 32 ist die Syntax des Hauptprogramms definiert, ein konkretes Beispiel ist in Quelltext 30 zu sehen. Darin ist zu sehen, dass das Hauptprogramm mit dem Schlüsselwort `main` eingeleitet und `endmain` beendet wird, mit einer beliebigen Anzahl an Anweisungen dazwischen. Diese Schlüsselwörter grenzen den Rumpf des Hauptprogramms und somit den entsprechenden Block ein. Gleiches gilt für weitere Sprachkonstrukte mit Blöcken in MuLE, beispielsweise wird eine Operation mit `operation` eingeleitet und beendet.

```
MainProgram
    :   'main'
        Statement*
        'endmain';
```

Quelltext 32: Grammatikalische Regel für das Hauptprogramm.

7.3. Operationen

Die Operationen in MuLE werden mit dem Schlüsselwort `operation` eingeleitet, gefolgt vom Namen und beliebig vielen Parametern. Weiterhin ist die Angabe eines Rückgabetyps optional, Operationen mit einem Rückgabetyp müssen eine `return` Anweisung in ihrem Rumpf aufweisen. In Quelltext 33 sieht man weiterhin, dass der Operationsrumpf ebenfalls optional ist. Dies beruht auf der Tatsache, dass die Sprache in Zukunft Objektorientierung unterstützen muss mit Methodendeklarationen ohne Rümpfe in Schnittstellen. Sofern eine Operation sich direkt im Programm befindet, wird durch die Validierung sichergestellt, dass sie einen Rumpf tatsächlich besitzt.

```
Parameter: 'parameter' ID ':' DataType;

Operation:
    'operation' name=ID '(' (params+=Parameter (',' params+=Parameter)*)? ')'
    (':' type=DataType)? (block=Block 'endoperation')?
;
```

Quelltext 33: Grammatikalische Regeln für Operationen und Parameter.

Parameter werden mit dem Schlüsselwort `parameter` gekennzeichnet, gefolgt vom Namen und dem Datentyp. Das einleitende Schlüsselwort erscheint an dieser Stelle überflüssig, da allerdings regelmäßig festgestellt wird, dass die Programmieranfänger Schwierigkeiten haben, zwischen Parametern, lokalen Variablen und Feldern zu unterscheiden, wurde dafür entschieden diese Konstrukte mit jeweils eigenen Schlüsselwörtern auf lexikalischer Ebene zu kennzeichnen. MuLE befolgt einer *Pass-by-Value* Semantik, es werden also immer Kopien von Parameterwerten übergeben. Wird ein Referenztyp übergeben, so wird eine Kopie dieser Referenz erzeugt, welche weiterhin den gleichen Speicherbereich im Heap referenziert.

```
program Operationen

import IO as io

operation funktion(parameter a : integer, parameter b : integer) : integer
    return a + b
endoperation

operation prozedur(parameter a : integer, parameter b : integer)
```

```
10      variable c : integer
11      c := a
12      a := b
13      b := c
14  endoperation
15
16  main
17      io.writeInteger(funktion(2, 3)) io.writeLine()
18      variable x : integer
19      variable y : integer
20      x := 2
21      y := 3
22      prozedur(x, y)
23  endmain;
```

Quelltext 34: Beispiele für Operationen mit und ohne Rückgabetyp.

Beispiele für Operationen werden in Quelltext 34 aufgeführt. Die Operation mit dem Namen `funktion` akzeptiert zwei Parameter, dessen Werte summiert zurückgegeben werden. Der Operation `prozedur` werden ebenfalls zwei Parameter übergeben, deren Werte vertauscht werden. An dieser Operation wird verdeutlicht, dass MuLE einer *Pass-by-Value* Semantik folgt, denn die Werte der Variablen, die als Parameter an die Operation übergeben werden, werden nicht außerhalb der Operation vertauscht. Möchte man also eine Prozedur mit Seiteneffekten implementieren, muss man Referenztypen (Abschnitt 5.8) einsetzen.

8. Standardbibliotheken

MuLE bietet eine Reihe von Standardbibliotheken, welche die Sprache um hilfreiche Funktionalitäten erweitern. Eine Übersicht über die Bibliotheken und ihre Operationen wird in folgenden Abschnitten gegeben.

8.1. IO

Die Standardbibliothek für Ein- und Ausgabeoperationen. Es wurde entschieden, unterschiedliche Operationen für die unterstützten primitiven Typen anzubieten, was den Umfang an Operationen größer macht. Als Alternative gäbe es die Möglichkeit, Ein- und Ausgabeoperationen nur für Zeichenketten anzubieten. Dies wäre eine wesentlich schlankere Lösung, die allerdings den Einsatz von Typkonvertierungen bei anderen Typen voraussetzen würde. Dem didaktischen Ansatz zu folge, die Programmierkonzepte möglichst nacheinander zu vermitteln, wurde die erste Alternative als die besser geeignete umgesetzt. Folgende Operationen werden angeboten:

- `writeLine()` – erzeugt einen Zeilenumbruch.

- `writeString(p1 : string)` – Ausgabe einer Zeichenkette. Erzeugt kein Zeilenumbruch (dies gilt für alle folgenden Ausgabeoperationen).
- `writeBoolean(p1 : boolean)` – Ausgabe eines Wahrheitswerts.
- `writeInteger(p1 : integer)` – Ausgabe einer ganzen Zahl.
- `writeRational(p1 : rational)` – Ausgabe einer Gleitkommazahl.
- `readString() : string` – Liest eine Eingabe auf der Konsole ein und gibt diese als eine Zeichenkette zurück.
- `readInteger() : integer` – liest eine Eingabe auf der Konsole ein und gibt diese als eine ganze Zahl zurück, sofern die Eingabe als solche ausgewertet werden kann. Bei falscher Eingabe wird die `java.util.InputMismatchException` geworfen.
- `readRational() : rational` – liest eine Eingabe auf der Konsole ein und gibt diese als eine Gleitkommazahl zurück, sofern die Eingabe als solche ausgewertet werden kann. Bei falscher Eingabe wird die `java.util.InputMismatchException` geworfen.

8.2. Mathematics

Diese Bibliothek beinhaltet eine Reihe von mathematischen Operationen, die nicht bereits durch Operatoren abgedeckt werden, was das Konzipieren von mathematisch orientierten Übungsaufgaben für Lehrpersonen erleichtern sollte. Hierbei sollte beachtet werden, dass mathematische Funktionen sich durch Hardwarelimitierungen nicht zu hundert Prozent auf einem Rechner abbilden lassen, bei den berechneten Werten handelt es sich um Annäherungen. Folgende Mathematische Operationen werden in MuLE zur Verfügung bereitgestellt:

- `randomInteger(min : integer, max : integer) : integer` – gibt einen zufälligen ganzzahligen Wert zwischen den Grenzen `min` und `max` zurück ($min \leq Zahl \leq max$).
- `randomRational() : rational` – gibt eine zufällige positive Gleitkommazahl zurück ($0.0 \leq Zahl < 1.0$).
- `pi() : rational` – gibt eine Annäherung der Kreiszahl π zurück, der zurückgegebene Wert beträgt 3.141592653589793.
- `e() : rational` – gibt eine Annäherung der Eulerschen Zahl e zurück, der zurückgegebene Wert beträgt 2.718281828459045.
- `sin(a : rational) : rational` – akzeptiert einen Winkel in Bogenmaß als Parameter und berechnet die Sinusfunktion.

- `cos(a : rational) : rational` – akzeptiert einen Winkel in Bogenmaß als Parameter und berechnet die Kosinusfunktion.
- `tan(a : rational) : rational` – akzeptiert einen Winkel in Bogenmaß als Parameter und berechnet die Tangensfunktion.
- `asin(a : rational) : rational` – berechnet die Umkehrfunktion der Sinusfunktion.
- `acos(a : rational) : rational` – berechnet die Umkehrfunktion der Kosinusfunktion.
- `atan(a : rational) : rational` – berechnet die Umkehrfunktion der Tangensfunktion.
- `log(a : rational) : rational` – berechnet den Logarithmus zur Basis e.
- `log10(a : rational) : rational` – berechnet den Logarithmus zur Basis 10.
- `round(a : rational) : integer` – rundet die übergebene Gleitkommazahl.
- `floor(a : rational) : integer` – schneidet die Nachkommastelle der übergebenen Gleitkommazahl ab.
- `absoluteInteger(a : integer) : integer` – gibt den Betrag der übergebenen ganzen Zahl zurück.
- `absoluteRational(a : rational) : rational` – gibt den Betrag der übergebenen Gleitkommazahl zurück.
- `toDegrees(a : rational) : rational` – Konvertierung eines in Bogenmaß gemessenen Winkels in Gradmaß.
- `toRadians(a : rational) : rational` – Konvertierung eines in Gradmaß gemessenen Winkels in Bogenmaß.
- `getMaxIntegerValue() : integer` – gibt die größte unterstützte ganze Zahl zurück.
- `getMinIntegerValue() : integer` – gibt die kleinste unterstützte ganze Zahl zurück.
- `getMaxRationalValue() : rational` – gibt die größte unterstützte Gleitkommazahl zurück.
- `getMinRationalValue() : rational` – gibt die kleinste unterstützte Gleitkommazahl zurück.

8.3. Strings

Diese Bibliothek bietet Funktionen zum Interagieren mit Zeichenketten an. Analog zu den Listen hat das erste Zeichen in einer Zeichenkette den Index 0. Folgende Operationen werden angeboten:

- `subString(str : string, startPos : integer, endPos : integer) : string` – gibt einen Teil der Zeichenkette `str` beginnend mit `startPos` und endend mit `endPost` zurück. Die Grenzen sind inklusiv.
- `lengthOf(str : string) : integer` – gibt die Anzahl der Zeichen in der übergebenen Zeichenkette zurück.
- `indexOfSubString(str : string, subStr : string) : integer` – gibt die Position einer Teilkette in einer Zeichenkette zurück. Bei mehreren Vorkommen einer Teilkette wird die Position der ersten zurückgegeben. Die Position einer Teilkette ist der Index des ersten Zeichens in der gesamten Zeichenkette. Sollte die Teilkette nicht vorkommen, wird `-1` zurückgegeben.
- `replace(str : string, regex : string, repl : string) : string` – ersetzt alle Vorkommnisse eines regulären Ausdrucks in der übergebenen Zeichenkette `str` durch die Zeichenkette `repl` und gibt das Ergebnis zurück.
- `toUpperCase(str : string) : string` – ersetzt alle kleingeschriebene Buchstaben einer Zeichenkette durch großgeschriebene.
- `toLowerCase(str : string) : string` – ersetzt alle großgeschriebene Buchstaben einer Zeichenkette durch kleingeschriebene.
- `integerToString(num : integer) : string` – konvertiert eine ganze Zahl in eine Zeichenkette.
- `rationalToString(num : rational) : string` – konvertiert eine Fließkommazahl in eine Zeichenkette.
- `booleanToString(bool : boolean) : string` – konvertiert ein Wahrheitswert in eine Zeichenkette.
- `genericToString(obj : generic<T>) : string` – konvertiert jedes beliebige Wert in eine Zeichenkette.

8.4. Lists

Zum Arbeiten mit MuLE Listen werden folgende, seiteneffektfreie, Funktionen angeboten:

- `isEmpty(l : list<generic<T>>) : boolean` – prüft, ob eine Liste leer ist.
- `lengthOf(l : list<generic<T>>) : integer` – gibt die Anzahl der Elemente in einer Liste zurück.
- `indexOf(l : list<generic<T>>, element : Type) : integer` – gibt die Position eines Elements in der Liste zurück.
- `append(l : list<generic<T>>, element : generic<T>) : list<generic<T>>` – hängt ein Element hinten an eine Liste an. Die übergebene Liste wird nicht verändert, es wird eine neue Liste erstellt und zurückgegeben.
- `head(l : list<generic<T>>) : generic<T>` – gibt den Wert an der ersten Position einer Liste zurück.
- `tail(l : list<generic<T>>) : list<generic<T>>` – gibt die Liste ohne des ersten Werts zurück.
- `last(l : list<generic<T>>) : generic<T>` – gibt den Wert an der letzten Position einer Liste zurück.
- `subList(l : list<generic<T>>, min : integer, max : integer) : list<generic<T>>` – gibt eine Teilliste ausgehend aus den übergebenen Grenzen zurück. Die Grenzen sind inklusiv.
- `insert(l : list<generic<T>>, element : generic<T>, pos : integer) : list<generic<T>>` – fügt ein Element an der übergebenen Position ein.
- `removeElement(l : list<generic<T>>, element : generic<T>) : list<generic<T>>` – Durchsucht die Liste und entfernt das erste Vorkommen des übergebenen Werts.
- `removePosition(l : list<generic<T>>, pos : integer) : list<generic<T>>` – Entfernt den Eintrag an der übergebenen Position.

8.5. Turtle

MuLE bietet eine eigene Implementierung der in der Lehre breit eingesetzten Turtle Graphics Bibliothek [2]. Die Implementierung ist so konzipiert, dass sie unabhängig von objektorientierten Konzepte verwendet werden kann. Ein Aufruf einer Turtle Operation im Programmcode führt dazu, dass das Turtle Fenster beim ausführen angezeigt wird, es werden also keine spezifischen Operationen zum Öffnen des Fensters benötigt.

Folgende Aufzählungstypen werden in dieser Bibliothek definiert:

- `Speed` – legt eine Reihe von Optionen für die Animationsgeschwindigkeit fest. Die Werte sind `SLOW, MEDIUM, FAST und INSTANT`.
- `Colors` – legt eine Menge von vordefinierten Farben fest. Die Werte sind `WHITE, BLACK, RED, GREEN, BLUE, YELLOW, MAGENTA, CYAN, PINK, ORANGE, LIGHT_GRAY und DARK_GRAY`.
- `Orientation` – legt eine Menge von vordefinierten Richtungen für den Stift fest. Die Werte sind `NORTH, SOUTH, EAST und WEST`.

Weiterhin beinhaltet diese Bibliothek folgende Operationen:

- `forward(length : integer)` – bewegt den Stift um die angegebene Anzahl der Pixel vorwärts. Zeichnet eine Linie sofern vorher nicht `penUp()` aufgerufen wurde.
- `backward(length : integer)` – bewegt den Stift um die angegebene Anzahl der Pixel rückwärts. Zeichnet eine Linie sofern vorher nicht `penUp()` aufgerufen wurde.
- `right(degree : rational)` – dreht den Stift um den angegebenen Winkel nach Rechts.
- `left(degree : rational)` – dreht den Stift um den angegebenen Winkel nach Links.
- `penUp()` – beim Bewegen des Stifts wird nicht mehr gezeichnet.
- `penDown()` – aktiviert das Zeichnen wieder, sollte vorher `penUp()` aufgerufen worden sein.
- `setColor(color : Colors)` – setzt die Farbe des Stifts auf eine der vordefinierten Farben.
- `setColorRGB(r : integer, g : integer, b : integer)` – setzt die Farbe des Stifts auf einen RGB Wert. Die Werte der Parameter müssen im Bereich [0 .. 255] liegen.
- `setThickness(thickness : integer)` – setzt die Breite des Stifts in Pixel. Der Standardwert liegt bei 1 Pixel.
- `moveTo(x : integer, y : integer)` – bewegt den Stift zu der übergebenen Koordinate und zeichnet dabei eine Linie, sofern kein `penUp()` vorher aufgerufen wurde. Am Ende der Bewegung behält der Stift die Bewegungsrichtung.

- `setPosition(x : integer, y : integer)` – bewegt den Stift zu der übergebenen Koordinate ohne zeichnet dabei eine Linie zu zeichnen. Am Ende der Bewegung behält der Stift seine ursprüngliche Richtung. Standardmäßig befindet sich der Stift bei den Koordinaten x = 300 und y = 200 Pixel.
- `setDirection(arc : rational)` – dreht den Stift in die übergebene Richtung als Winkel. Standardmäßig ist der Stift in Richtung Norden ausgerichtet, was dem Wert 0 Grad entspricht.
- `setOrientation(o : Orientation)` – dreht den Stift in die übergebene vordefinierte Himmelsrichtung. Standardmäßig ist der Stift in Richtung Osten gedreht.
- `setFrameSize(x : integer, y : integer)` – legt die Größe des Fensters in Pixel fest.
- `getX() : integer` – gibt den Wert der x-Koordinate des Stifts zurück.
- `getY() : integer` – gibt den Wert der y-Koordinate des Stifts zurück.
- `getArc() : rational` – gibt die Ausrichtung des Stifts als ein Winkel zurück. 0 Grad entspricht der Himmelsrichtung Norden.
- `startFilledPolygon(color : Colors)` – initiiert das Zeichnen eines Polygons mit der übergebenen vordefinierten Farbe.
- `startFilledPolygonRGB(r : integer, g : integer, b : integer)` – initiiert das Zeichnen eines Polygons mit durch RGB Werte (im Bereich [0 .. 255]) definierter Farbe.
- `endFilledPolygon()` – beendet das Zeichnen eines Polygons.
- `circle(radius : integer)` – zeichnet einen Kreis.
- `filledCircle(radius : integer, color : Colors)` – zeichnet einen gefüllten Kreis mit einer vordefinierter Farbe.
- `filledCircleRGB(radius : integer, r : integer, g : integer, b : integer)` – zeichnet einen gefüllten Kreis mit einer durch RGB Werte (im Bereich [0 .. 255]) definierter Farbe.
- `setAnimationSpeed(speed : Speed)` – bestimmt die Geschwindigkeit der Zeichenanimation.
- `showCoordinateSystem(bool : boolean)` – bestimmt, ob das Koordinatensystem visuell eingeblendet wird. Der Ursprung befindet sich im linken oberen Eck des Fensters.

- `showCursor(bool : boolean)` – bestimmt, ob der Zeiger des Stifts visuell dargestellt wird.
- `activateDrawMode(speed : Speed)` – startet den mit Pfeiltasten gesteuerten Zeichenmodus. Dies kann vom Nutzer als Einstieg verwendet werden, um einen Gefühl vom Werkzeug zu bekommen.

```
1  program nEcke
2  import Turtle as turtle
3
4  operation draw(parameter n : integer,
5                 parameter l : integer,
6                 parameter color : Colors)
7      turtle.startFilledPolygon(color)
8      foreach variable a : integer in [1 .. n] do
9          turtle.forward(l)
10         turtle.left(360.0/n)
11     endforeach
12     turtle.endFilledPolygon()
13 endoperation
14
15 operation move(parameter a : integer)
16     turtle.penUp()
17     turtle.forward(a)
18     turtle.penDown()
19 endoperation
20
21 main
22     turtle.showCoordinateSystem(true)
23     turtle.setFrameSize(300,200)
24     turtle.setPosition(50, 150)
25     turtle.setThickness(2)
26     draw(4, 60, RED)
27     move(65)
28     draw(3, 45, GREEN)
29     move(50)
30     draw(6, 45, BLUE)
31     turtle.showCursor(false)
32 endmain
```

Quelltext 35: Beispiel für die Verwendung der Turtle Bibliothek. Das Ergebnis ist in Abbildung 1 zu sehen.

9. Ausführungssemantik

Beim Speichern einer MuLE Datei, die keine Übersetzungsfehler hat, wird diese in ein Java-Programm übersetzt. Die übersetzten Java-Quelldateien befinden

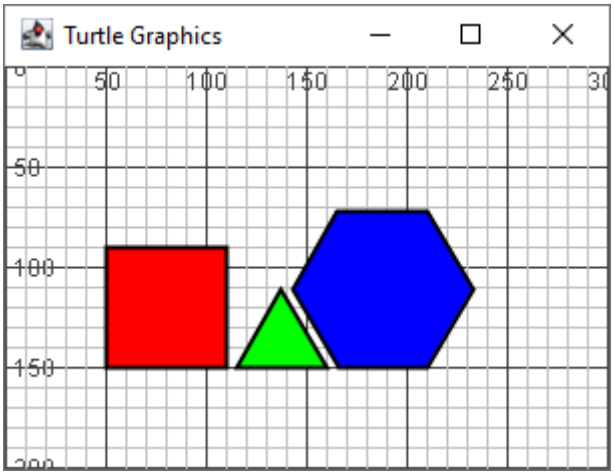

Abbildung 1: Das Resultat der Ausführung des Programms in Quelltext 35.

sich im gleichen Projekt im `src-gen` Ordner. Der Benutzer kann nun die MuLE Datei, sofern es sich um ein Programm und nicht um eine Bibliotheksdatei handelt, ausführen. Im Hintergrund wird die generierte Java-Quelldatei auf der JVM ausgeführt. Es wird also eine JRE Installation auf dem Rechner benötigt.

10. Speichermodell und die Kopiersemantik bei Wertübergabe

Jede Operation und die Hauptprozedur besitzen ihren lokalen Speicher, den Stack, auf dem die deklarierten Datenbehälter (Abschnitt 5.11) in der Reihenfolge ihrer Deklaration angeordnet sind. Alle Werte befinden sich im Heap Speicher des Programms, d.h. rein technisch gesehen sind alle Datenbehälter Referenzen auf Werte. Der Unterschied zwischen den Werttypen und Referenztypen wird durch die Kopiersemantik bei Werttypen umgesetzt.

Bei Zuweisungen an Variablen und Attribute, Parameterübergabe und Wertrückgabe wird stets eine tiefe Kopie des Werts erstellt und weitergereicht. Im Beispiel in Quelltext 36 wird der `integer` Variable `num` der Wert 5 zugewiesen. Hierbei wird eine Kopie des Werts 5 erstellt und diese Kopie wird in der Variable abgelegt. Bei elementaren Werten, wie es bei diesem Beispiel der Fall ist, erscheint dies wenig Sinn zu ergeben, was aber bei komplexen Typen, wie Listen und Kompositionen, wiederum anders ist.

Die Variable ist in MuLE semantisch als ein Behälter für Werte und nicht als der Zeiger auf einen Bereich im Speicher zu verstehen, da diese Semantik für Programmieranfänger leichter zu verstehen ist. Etwas anders verhalten sich die Referenztypen. Wird einer Referenz eine Andere zugewiesen, z.B. `ref1 := ref2`, so zeigt die Referenz `ref1` auf den gleichen Wert im Speicher wie die Referenz

`ref2` und hat somit die gleiche Adresse. Ist `ref2` eine `null`-Referenz, so wird `ref1` ebenfalls zu einer `null`-Referenz.

```
1  program memoryExample
2
3  import IO as io
4
5  operation foo(parameter a : reference<integer>)
6      a@ := a@ + a@
7  endoperation
8
9  operation bar(parameter a : integer) : integer
10     variable b : integer
11     b := 2 * a
12     return b
13 endoperation
14
15 main
16     variable num : integer
17     variable ref : reference<integer>
18     num := 5
19     ref := reference num
20
21     foo(ref)
22     num := bar(num)
23     io.writeInteger(ref@)
24 endmain
```

Quelltext 36: Beispiel zur Erklärung des Speichermodells.

Abbildungen 2 bis 7 demonstrieren vereinfacht den Zustand im Speicher bei der Ausführung des Programms in Quelltext 36 zu unterschiedlichen Zeitpunkten. Das Programm wurde in der jeweiligen Zeile angehalten, d.h. die entsprechende Zeile wurde zu diesem Zeitpunkt noch nicht ausgeführt. Der Kommentar unter jeder Abbildung beschreibt jeweils kurz den Stand im Programm und die Vorgänge im Speicher.

```
1  program memoryExample
2
3  import IO as io
4
5  operation foo(parameter a : reference<integer>)
6      a@ := a@ + a@
7  endoperation
8
9  operation bar(parameter a : integer) : integer
10     variable b : integer
11     b := 2 * a
12     return b
13 endoperation
14
15 main
16     variable num : integer
17     variable ref : reference<integer>
18     num := 5
19     ref := reference num
20
21     foo(ref)
22     num := bar(num)
23     io.writeInteger(ref@)
24 endmain
```

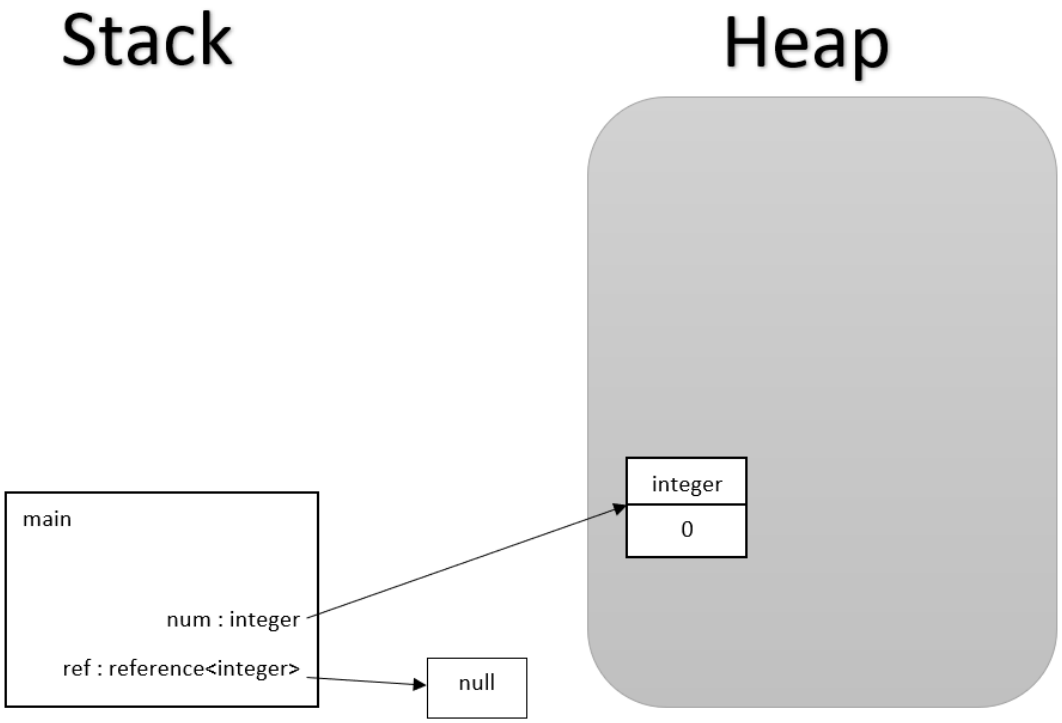

Abbildung 2: Zustand im Speicher in der Zeile 18. Variablen `num` und `ref` wurden in der Hauptprozedur deklariert und haben ihre Standardwerte.

```
1  program memoryExample
2
3  import IO as io
4
5  operation foo(parameter a : reference<integer>)
6      a@ := a@ + a@
7  endoperation
8
9  operation bar(parameter a : integer) : integer
10     variable b : integer
11     b := 2 * a
12     return b
13 endoperation
14
15 main
16     variable num : integer
17     variable ref : reference<integer>
18     num := 5
19     ref := reference num
20
21     foo(ref)
22     num := bar(num)
23     io.writeInteger(ref@)
24 endmain
```

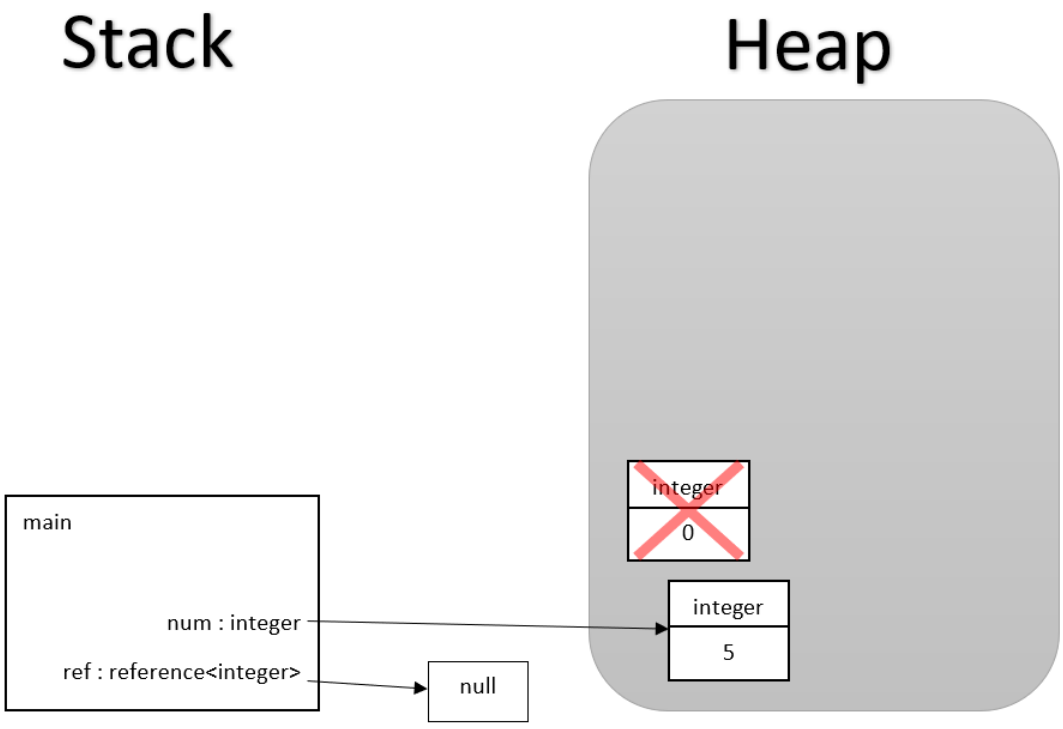

Abbildung 3: Zustand im Speicher in der Zeile 19. Der Variable `num` wurde der Wert 5 zugewiesen. Im Speicher wurde ein neuer `integer`-Element mit diesem Wert erzeugt und die Variable `num` zeigt nun auf diesen Speicherelement. Der alte Element wird nicht mehr benötigt und wird vom Garbage Collector automatisch freigegeben.

```
1  program memoryExample
2
3  import IO as io
4
5  operation foo(parameter a : reference<integer>)
6      a@ := a@ + a@
7  endoperation
8
9  operation bar(parameter a : integer) : integer
10     variable b : integer
11     b := 2 * a
12     return b
13 endoperation
14
15 main
16     variable num : integer
17     variable ref : reference<integer>
18     num := 5
19     ref := reference num
20
21     foo(ref)
22     num := bar(num)
23     io.writeInteger(ref@)
24 endmain
```

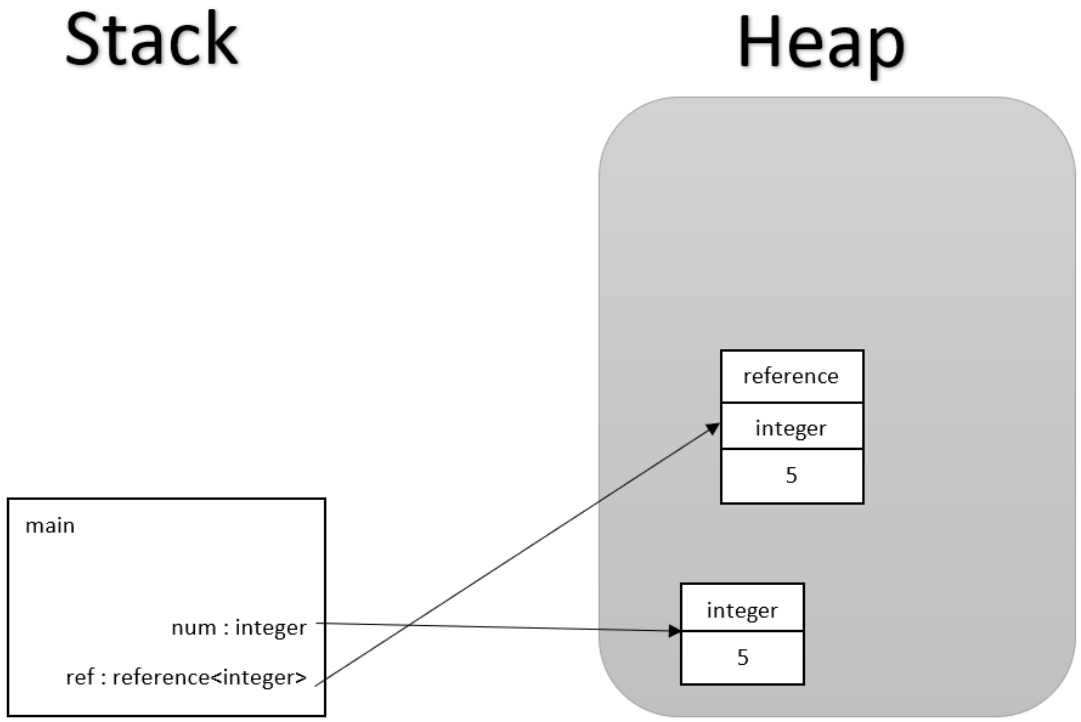

Abbildung 4: Zustand im Speicher in der Zeile 21. Der Variable `ref` wurde der Wert `reference num` zugewiesen. Zu diesem Zweck wurde eine Kopie des Werts von `num` erzeugt und als ein Referenztyp verpackt.

```
1  program memoryExample
2
3  import IO as io
4
5  operation foo(parameter a : reference<integer>)
6      a@ := a@ + a@
7  endoperation
8
9  operation bar(parameter a : integer) : integer
10     variable b : integer
11     b := 2 * a
12     return b
13 endoperation
14
15 main
16     variable num : integer
17     variable ref : reference<integer>
18     num := 5
19     ref := reference num
20
21     foo(ref)
22     num := bar(num)
23     io.writeInteger(ref@)
24 endmain
```

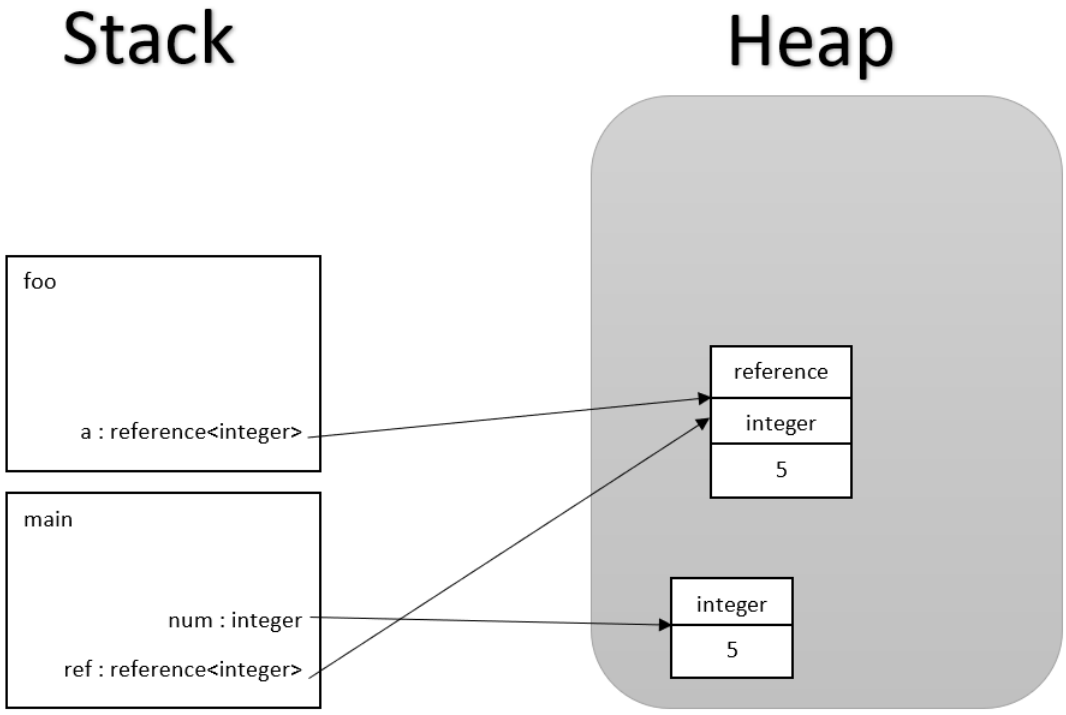

Abbildung 5: Zustand im Speicher in der Zeile 6. In der Hauptprozedur wurde die Operation `foo` mit der Variable `ref` als Parameter aufgerufen. Auf dem Stack wurde Speicher für die Operation `foo` und ihren Parameter `a` angelegt. Es ist anzumerken, dass sowohl die Variable `ref` als auch der Parameter `a` den gleichen Speicherbereich im Heap referenzieren, d.h. nach dem Abarbeiten der Operation wird der Wert der Variable in der Hauptprozedur verändert.

```
1  program memoryExample
2
3  import IO as io
4
5  operation foo(parameter a : reference<integer>)
6      a@ := a@ + a@
7  endoperation
8
9  operation bar(parameter a : integer) : integer
10     variable b : integer
11     b := 2 * a
12     return b
13 endoperation
14
15 main
16     variable num : integer
17     variable ref : reference<integer>
18     num := 5
19     ref := reference num
20
21     foo(ref)
22     num := bar(num)
23     io.writeInteger(ref@)
24 endmain
```

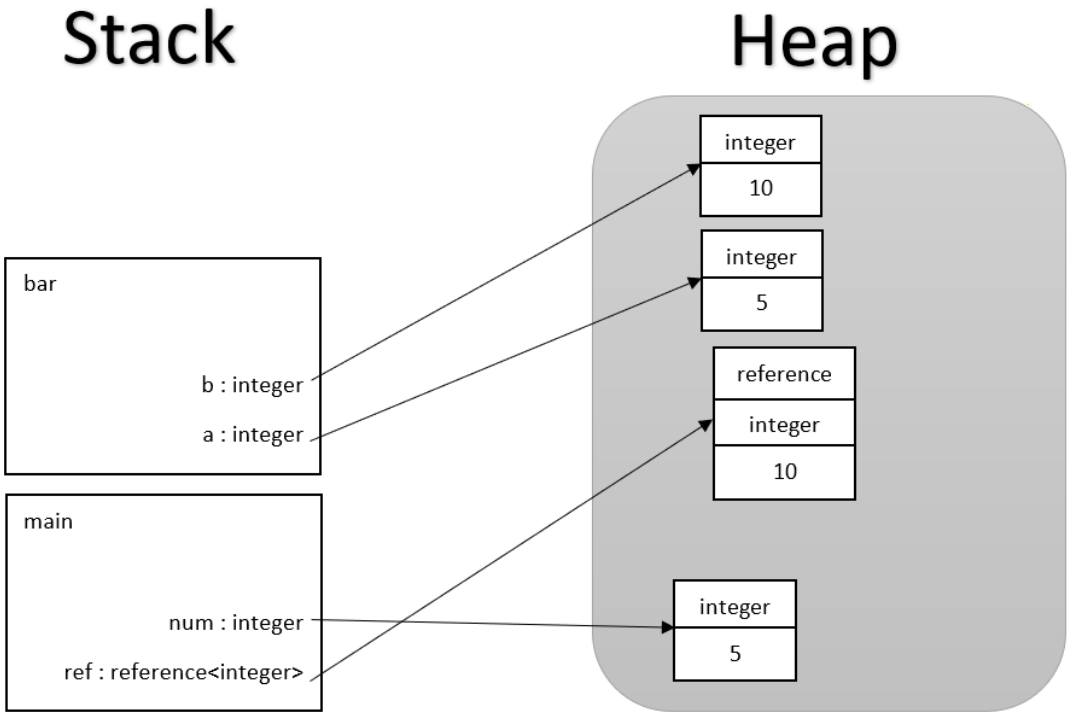

Abbildung 6: Zustand im Speicher in der Zeile 12. Die Operation `foo` wurde abgearbeitet und ihr Speicher im Stack freigegeben. Die Operation `bar` wurde aufgerufen mit einem Parameter und einer lokalen Variable. Obwohl der Parameter mit dem Wert der Variable `num` arbeitet, wurde eine Kopie des Werts bei der Parameterübergabe angelegt.

```
1  program memoryExample
2
3  import IO as io
4
5  operation foo(parameter a : reference<integer>)
6      a@ := a@ + a@
7  endoperation
8
9  operation bar(parameter a : integer) : integer
10     variable b : integer
11     b := 2 * a
12     return b
13 endoperation
14
15 main
16     variable num : integer
17     variable ref : reference<integer>
18     num := 5
19     ref := reference num
20
21     foo(ref)
22     num := bar(num)
23     io.writeInteger(ref@)
24 endmain
```

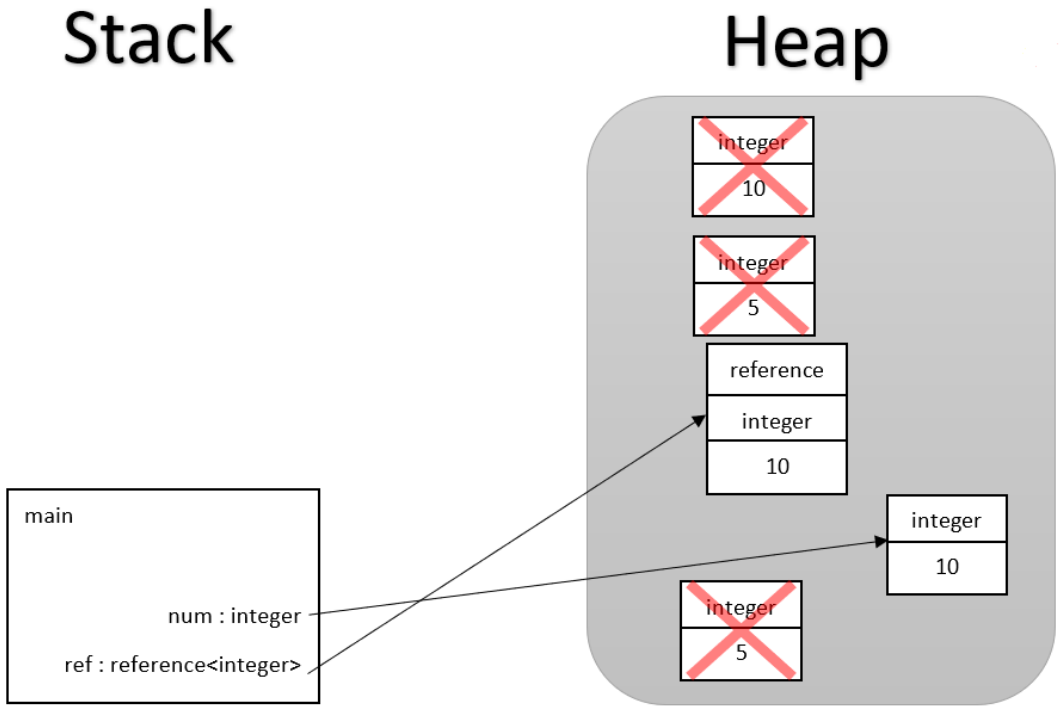

Abbildung 7: Zustand im Speicher in der Zeile 23. Die Operation `bar` wurde abgearbeitet und ihr Rückgabewert wurde der Variable `num` zugewiesen. Es wurde ein neuer Element im Speicher mit dem berechneten Rückgabewert erzeugt, auf dem nun die Variable `num` zeigt. Der alte Wert der Variable `num` sowie die lokalen Datenbehälter der Operation `bar` sind nun nicht mehr benötigt und werden aufgeräumt.

11. Werkzeugunterstützung

Derzeit ist Eclipse die einzige IDE, die MuLE unterstützt. Weiterhin ist MuLE momentan nicht alleinstehend lauffähig und muss in Kombination mit Eclipse verwendet werden. Dafür ist es beim Programmieren mit MuLE möglich, auf die mächtigen Werkzeuge, die von Eclipse angeboten werden, zurückzugreifen:

- *Integrierter Texteditor* mit Quelltexthervorhebung. Schlüsselwörter und Datentypen werden jeweils mit eigenen Farben im Quelltext markiert. Übersetzung erfolgt beim Speichern der Datei. Der Benutzer kann darauffolgend das MuLE Programm ausführen, wobei im Hintergrund die generierte Java Klasse ausgeführt wird.
- *Debugger*, mit dessen Hilfe das Programm zeilenweise ausgeführt werden kann. Dies ist nicht nur zur Beseitigung von Fehlern geeignet, sondern auch zum Demonstrieren von bestimmten Programmierkonzepten wie z.B. Schleifen.
- *Fehlermeldungen* sollen aussagekräftig, kurz und nicht kryptisch sein, sodass ein Programmieranfänger lernt, auf Fehlermeldungen zu achten und als hilfreich wahrzunehmen, statt sie zu ignorieren [4]. Sie sollen genau auf den Fehler hinweisen und technische Details auslassen.
- *Outline tree provider*, welcher eine schnelle Übersicht über größere Programme ermöglicht.
- *Refactoring*, z.B. zum einfacheren Umbenennen eines Elements mit Aktualisierung von allen Vorkömmnissen dieses Elements im Quelltext.
- *Wizards* zum einfachen Erstellen von Projekten und Programmdateien.
- *Automatischer Formatierer* von Quelltext ist besonders hilfreich, um den Code von Programmieranfängern, die dazu neigen, keine Quelltextkonventionen zu befolgen, lesbar zu machen.
- *Content assist*, derzeit in Entwicklung.
- *Quickfix provider*, derzeit in Entwicklung.

12. Einsatz in der Praxis

Diese Sprache wurde bereits zwei Mal (2018 und 2019) im Rahmen eines Programmiervorkurses in der Praxis eingesetzt. Die daraus gewonnen Erfahrungen helfen die Sprache zu verbessern. Zum Einen schreiben Programmieranfänger ihre Programme auf unkonventionelle Weise und helfen dabei immens, Fehler in der

Implementierung der Sprache zu finden indem sie bestimmte Sprachkonstrukte auf unvorhergesehene Art nutzen. Zum Anderen geben sie gelegentlich konstruktive Verbesserungsvorschläge. Weiterhin nutzen die Studierenden unterschiedliche Betriebssysteme, was eine gute Gelegenheit bietet, die Implementierung auf diesen Systemen zu testen.

12.1. Programmiervorkurs

Der Programmiervorkurs, in dem diese Sprache eingesetzt wird, findet am Anfang des Semesters vor der Vorlesung *„Konzepte der Programmierung“* (CS1) statt. Ziel dieses Vorkurses ist es, Studierenden ohne Programmiervorkenntnisse erste Einblicke in diese Disziplin zu gewähren und ihnen den Einstieg in das Informatikstudium zu erleichtern. Zu diesem Zweck werden wichtige Konzepte des prozeduralen Programmierparadigmas vermittelt, ohne jedoch zu tief ins Detail zu gehen. So werden z.B. Variablen als Datenbehälter erklärt ohne genau auf die Wertsemantik und auf das Speichermodell einzugehen. Auf die Speicherlimitierungen wird lediglich kurz beim Erläutern der Datentypen eingegangen.

Der Kurs ist in folgende thematische Einheiten aufgeteilt:

1. **Einstieg** – kurze Erklärung des Kurses, der Sprache und Einrichten der Programmierumgebung.

2. **Algorithmen, Daten, Anweisungen** – Zum Anfang werden Algorithmen besprochen, wobei abstrakte natürlichsprachige Lösungen schrittweise in formale Algorithmen überführt werden. Algorithmen werden in ihre Bestandteile, Daten und Anweisungen, zerlegt. Es wird der Konzept der Variable als Datenbehälter eingeführt gefolgt vom Konzept der Datentypen. Ausgabe wird als eine einfache Anweisung besprochen und die Studierenden implementieren ihr erstes `Hello, world!` Programm. Daraufhin werden numerische Typen und arithmetische Operatoren eingeführt und die Studierenden müssen eine komplexere Aufgabe mit mehreren Variablen lösen. Weitere Konzepte sind Bibliotheken, Eingabe, Zeichenkettenkonkatenation, Typumwandlungen und Fehlerarten.

3. **Kontrollstrukturen** – Verzweigungen werden anhand von einem Beispielalgorithmus besprochen woraufhin die if-Anweisungen eingeführt werden. Als nächstes werden Wahrheitswerte und der Typ `boolean` sowie die entsprechenden Operatoren besprochen. Die Turtle Bibliothek wird eingeführt und einfache Lösungen zum Zeichnen eines Quadrats werden implementiert, gefolgt von der Einführung des Schleifenkonzepts und Verallgemeinerung des Turtle-Quadrats zu einem n-Eck. Zum Abschluss wird ein kurzer Ausblick auf `foreach`-Schleifen verschaffen.

4. **Benutzerdefinierte Typen** – Endliche Mengen an möglichen Werten werden besprochen gefolgt von der Einführung von Aufzählungstypen. Verbundtypen werden als nächstes eingeführt, Attribute und qualifizierter Zugriff müssen ebenfalls erklärt werden.

5. **Listen** – Dieses Thema befasst sich komplett mit den in MuLE vordefinierten Listen und `foreach`-Anweisungen.

6. **Operationen** – Die Studierenden lernen eigene Operationen zu definieren. Konzepte der Übergabeparameter und Rückgabewerte werden eingeführt und es wird auf die Unterschiede zwischen Prozeduren und Funktionen eingegangen.

Wie man sieht, werden manche Konzepte der Sprache im Kurs gar nicht behandelt, z.B. die Referenztypen. Dies liegt vor allem an mangelnder Zeit, es sind zwei Wochen für den Vorkurs vorgesehen mit 90 minütigen Sitzungen am Tag. Das Ziel ist allerdings auch nicht, alle möglichen Konzepte zu behandeln, sondern die wichtigsten für den Anfang auf eine verständliche Weise zu vermitteln. Für die schnelleren Studierenden werden zusätzlich die Kapiteln Rekursion und Referenztypen zum Selbststudium vorbereitet, sowie eine Reihe von Zusatzaufgaben angeboten.

12.2. Eingesetzte Lehrstrategien Und Werkzeuge

Vor der Einführung eines neues Konzepts werden Alltagsbeispiele oder aus der Schule bekannte Konzepte herangeführt und besprochen. Die Studierenden werden dazu motiviert, sich kurz darüber Gedanken zu machen, wie diese Probleme mit bekannten Programmierwerkzeugen lösbar sein könnten, wenn überhaupt. Daraufhin wird der neue Konzept eingeführt mit einem Beispielprogramm, welches sich auf das vorherige abstrakte Beispiel bezieht. Meistens wird dabei eine Kombination aus Live-Coding und gemeinsamen Diskussionen verwendet gefolgt von eigenständiger Erweiterung des Beispielprogramms. Daraufhin müssen die Studierenden eigenständig eine Aufgabe bearbeiten, bei der der Neue sowie die bereits bekannten Konzepte verwendet werden müssen. Der Dozent unterstützt dabei die Studierenden bei Fragen und nach eigenem Ermessen. Im Anschluss kommt eine gemeinsame Besprechung der Lösung, meistens ebenfalls im Live-Coding Format. Bei schwierigeren Aufgaben werden Zwischenschritte gezeigt.

Programmieranfänger werden beim Lösen von Aufgaben für gewöhnlich gleich zwei Problemen ausgesetzt. Zum Einen müssen Sie neue Werkzeuge verwenden, deren Sinn sie noch nicht ganz verstanden haben. Zum Anderen müssen Sie einen Algorithmus für ein abstraktes Problem entwerfen, womit sie ebenfalls noch nicht viel Erfahrung haben. Daher haben diese Studierende teilweise große Schwierigkeiten etwas komplexere Aufgaben zu lösen. Um dies zu vermeiden, wird die

algorithmische Lösung bei solchen Aufgaben vor der Bearbeitung gemeinsam ausgearbeitet und auf die benötigten Sprachkonstrukte wird hingedeutet anhand von wiederkehrenden Mustern aus den vorherigen Aufgaben und Beispielen. Die Studierenden müssen nur noch diese Werkzeuge einsetzen um den Algorithmus in ein Programm zu übersetzen. Bei der gemeinsamen Diskussion sollen die Studierenden sich mit der Vorgehensweise, solche Probleme zu lösen, bekannt machen um diese in Zukunft bei ähnlichen Problemen anwenden zu können.

Als Beispiel kann eine Aufgabe zum Sortieren einer Liste betrachtet werden. Bei der Aufgabe müssen die Studierenden eine `integer`-Liste mit zufallsgenerierten Werten füllen und Aufsteigend sortieren. Zum Anfang wird eine Liste mit zufälligen Werten an die Tafel geschrieben und abstrakte Lösungswege werden herangezogen. Danach wird ein Lösungsweg gewählt, z.B. eine einfache Variante des Bubblesort Algorithmus, und an der Tafel schrittweise vorgeführt. Dabei wird bei jedem Schritt diskutiert, welches Sprachkonstrukt dabei am besten passen würde. Daraufhin wird diese Aufgabe bearbeitet und nach einiger Zeit werden Zwischenschritte eingeblendet. In Vergangenheit hat diese Aufgabe sich als eine der schwierigsten erwiesen, weshalb diesmal dieses schrittweises Vorgehen beim Lösen eingesetzt wurde. Als Ergebnis hatten die meisten Studierenden eine einigermaßen akzeptable Lösung gegen Ende der Bearbeitungszeit mit geringer persönlicher Hilfe seitens des Dozenten während der Bearbeitung und es wurden keine Fälle einer Denkblockade beobachtet.

Des Weiteren wurden die Studierenden dazu motiviert, in Paaren oder in Gruppen zu arbeiten. Hiermit ist kein *Pair Programming* gemeint, sondern gemeinsame Ausarbeitung der Lösung. Es war den Studierenden überlassen, ob sie zu zweit an einem Rechner arbeiten, oder jeder an seinem eigenen, solange sie gemeinsam mit ihren Nachbarn eine Lösung erarbeiten. Weiterhin wurde darauf hingewiesen, dass alle Gruppenmitglieder sich an der Erarbeitung der Lösung beteiligen sollten. Damit sollte zum Einen eine schnellere Bearbeitung der Aufgaben erreicht werden, zum Anderen sollten so die langsameren Studierenden mehr Hilfe durch ihren Nachbarn bekommen. Selbst wenn beide Studierenden große Schwierigkeiten bei der Bearbeitung haben, sollte der Frustfaktor durch den sozialen Kontakt geringer gehalten werden.

Neben der Sprache selbst werden im Programmierkurs weitere Werkzeuge verwendet. Da die Sprache derzeit nur in Verbindung mit der Entwicklungsumgebung Eclipse eingesetzt werden kann, fällt dementsprechend die Wahl auf diese IDE aus. Zwar ist Eclipse wesentlich komplexer als speziell für Anfänger entwickelte Programmierumgebungen, z.B. BlueJ, Greenfoot oder Scratch, die Kursteilnehmer hatten keine Probleme die für den Kurs benötigten Funktionen der Eclipse IDE zu benutzen. Zum Zweck der Auflockerung der Inhalte werden spielerische Werkzeuge, wie Turtle Graphics und *Code.org* Beispiele zum Erklären bestimmter Konzepte verwendet. Die Turtle Bibliothek wird mit der Sprache ausgeliefert (Abschnitt 8.5), als Alternative zu den *Code.org* Beispielaufgaben ist für die Zukunft eine eigene Microworlds Bibliothek geplant. Als Plattform zum Hochladen

von Kursfolien und Musterlösungen wird das Elearning System der Universität Bayreuth eingesetzt. Die Studierenden arbeiten entweder an den Rechnern des Computerraum oder an eigenen Laptops.

12.3. Beispiele für Aufgaben

Eine der anfänglichen Aufgaben ist in Abbildung 8 zu sehen. Die Konzepte, die bei dieser Aufgabe verwendet werden müssen sind *Importe*, *Wahl von passenden Datentypen*, *Deklarieren von Variablen*, *Wertzuweisungen*, *arithmetische Operationen* und *Ausgabe*. Nach dem Vorlesen der Aufgabe wird die Notwendigkeit für diese Konzepte bei einer gemeinsamen Diskussion ausgearbeitet, woraufhin die Studierenden sich mit der Bearbeitung der Aufgabe beschäftigen. Im Anschluss wird die Lösung (Quelltext 37) besprochen. Zusätzlich können die Konzepte *Kommentare*, *Zeichenkettenkonkatenation* und *Typumwandlung* eingesetzt werden. Die Ausgabe für das Programm in Quelltext Quelltext 37 lautet:

```
Der Fahrer war mit 60.19486356824808km/h unterwegs.
Der Fahrer war mit 60km/h unterwegs.
```

```
1  program Aufgabe05
2
3  import IO as io
4  import Mathematics as math
5  import Strings as str
6
7  main
8      // Gegeben
9      variable s : rational
10     variable mu : rational
11     variable g : rational
12     s  := 19.0 // m
13     mu := 0.75
14     g  := 9.81 // m/s^2
15
16     //Gesucht
17     variable v : rational
18
19     //Loesung
20     v := (2 * s * g * mu) exp 0.5 // m/s
21     v := v * 3.6 // km/h
22
23     // Einfache Ausgabe
24     io.writeString("Der Fahrer war mit ")
25     io.writeRational(v)
26     io.writeString("km/h unterwegs.")
27     io.writeLine()
```

```
28
29      // Ausgabe mit Stringkonkatenation und Typkonvertierung
30      io.writeString("Der Fahrer war mit " & str.integerToString(math.round(v))
            & "km/h unterwegs.\n")
31  endmain
```

Quelltext 37: Die Lösung der *Temposünder* Aufgabe.

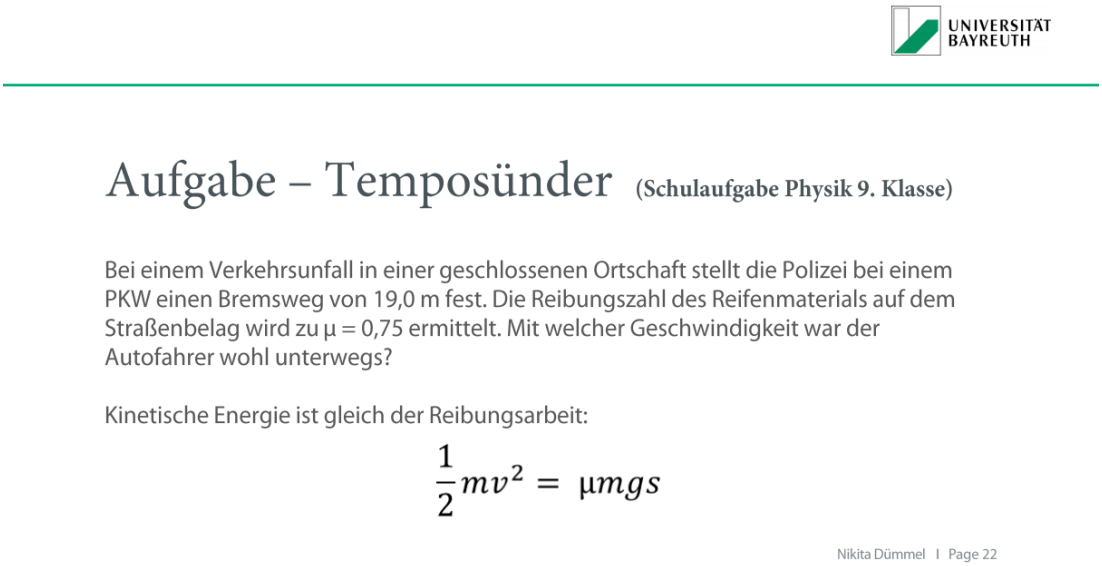

Abbildung 8: Folie mit der Aufgabenstellung zu der *Temposünder* Aufgabe.

Eine der Aufgaben wird in mehreren Schritten gelöst, d.h. Die Lösung wird erweitert sobald ein neuer Konzept eingeführt wurde. Bei dieser Aufgabe müssen die Teilnehmer die Grundlage für ein Schachspiel implementieren. Nach dem die Konzepte der *Aufzählungstypen* und *Verbunden* eingeführt wurden, wird den Teilnehmen ein Bild mit mehreren Schachfiguren gezeigt und ihre Eigenschaften werden bei einer Diskussion festgehalten. Daraufhin wird diskutiert, wie diese Eigenschaften nun mit Programmcode dargestellt werden können. Die Studierenden müssen die Aufzählungstypen `Farbe` (Schwarz und Weiss) und `Figurtyp` (Bauer, Springer, Läufer, Turm, Dame und König) definieren, sowie die Verbunde `Figur` (hat eine `Farbe` und einen `Figurtyp`) und `Feld` (hat x und y Koordinaten, eine Farbe und eine Figur). Da der Konzept der Referenztypen aus zeitlichen Gründen nicht erklärt wird, besitzt jeder Feld eine Figur. Beim Besprechen von zweidimensionalen Listen wird diese Aufgabe erweitert. Die Teilnehmer müssen das Schachbrett als eine zweidimensionale Liste mit Feldern mit korrekten Koordinaten und Farben befüllen und das Ergebnis aus der Konsole ausgeben.

Beim Thema Operationen wird diese Aufgabe ein weiteres Mal erweitert. Die Teilnehmer müssen die bisherige Implementierung in semantisch zusammenhängende

Operationen `initialisiereBrett()` und `gebeBrettAus()` aufteilen, sowie eine neue Operation `zeichneBrett()` implementieren, die das Brett mit Hilfe der Turtle Bibliothek zeichnet. Die Musterlösung ist in Quelltext 38 zu sehen, ein Teil der Ausgabe sowie das gezeichnete Bild ist in Abbildung 9 ersichtlich. Durch den Einsatz von Referenztypen könnte man mit `null`-Referenzen das Fehlen einer Figur auf einem Feld symbolisieren. Das dieser Konzept den meisten Studierenden nicht bekannt ist, muss diese Semantik mit anderen Mitteln erreicht werden.

```
1  program Aufgabe06_schachbrett_forts
2  import Turtle as turtle
3  import IO as io
4  import Strings as str
5
6  type Figurtyp : enumeration
7      FAKE, BAUER, TURM, SPRINGER, LAEUFER, DAMER, KOENIG
8  endtype
9
10 type Farbe : enumeration
11     NONE, SCHWARZ, WEISS
12 endtype
13
14 type Figur : composition
15     attribute typ : Figurtyp
16     attribute farbe : Farbe
17 endtype
18
19 type Feld : composition
20     attribute farbe : Farbe
21     attribute zahl : integer
22     attribute buchstabe : string
23     attribute figur : Figur
24 endtype
25
26 operation initialisiereBrett(parameter schachbrett : list<list<Feld>>) :
       list<list<Feld>>
27     schachbrett := [8 ** [8 ** default]]
28
29     variable abc : list<string>
30     abc := ["a", "b", "c", "d", "e", "f", "g", "h"]
31
32     foreach variable i : integer in [0 .. 7] do
33         foreach variable j : integer in [0 ..7] do
34             variable feld : Feld
35             feld.zahl := 8-i
36             feld.buchstabe := abc[j]
37             if (i + j) mod 2 = 1 then
38                 feld.farbe := SCHWARZ
```

```
            else
                feld.farbe := WEISS
            endif
            schachbrett[i][j] := feld
        endforeach
    endforeach
    return schachbrett
endoperation

operation gebeBrettAus(parameter schachbrett : list<list<Feld>>)
    foreach variable zeile : list<Feld> in schachbrett do
        io.writeString(str.genericToString(zeile) & "\n")
    endforeach
endoperation

operation zeichneBrett(parameter schachbrett : list<list<Feld>>)
    turtle.setAnimationSpeed(FAST)
    variable size : integer
    size := 8
    foreach variable i : integer in [0 .. size-1] do
        foreach variable j : integer in [0 .. size-1] do
            turtle.setPosition(i*20+100, j*20+100)
            if schachbrett[i][j].farbe = SCHWARZ then
                turtle.startFilledPolygon(BLACK)
                foreach variable temp : integer in [0 .. 3] do
                    turtle.forward(20)
                    turtle.right(90)
                endforeach
                turtle.endFilledPolygon()
            endif
        endforeach
    endforeach
    turtle.setPosition(100, 100)
    foreach variable temp : integer in [0 .. 3] do
        turtle.forward(size*20)
        turtle.right(90)
    endforeach
    turtle.showCursor(false)
endoperation

main
    variable schachbrett : list<list<Feld>>
    schachbrett := initialisiereBrett(schachbrett)
    gebeBrettAus(schachbrett)
    zeichneBrett(schachbrett)
endmain
```

Quelltext 38: Die Lösung der *Schachbrett* Aufgabe.

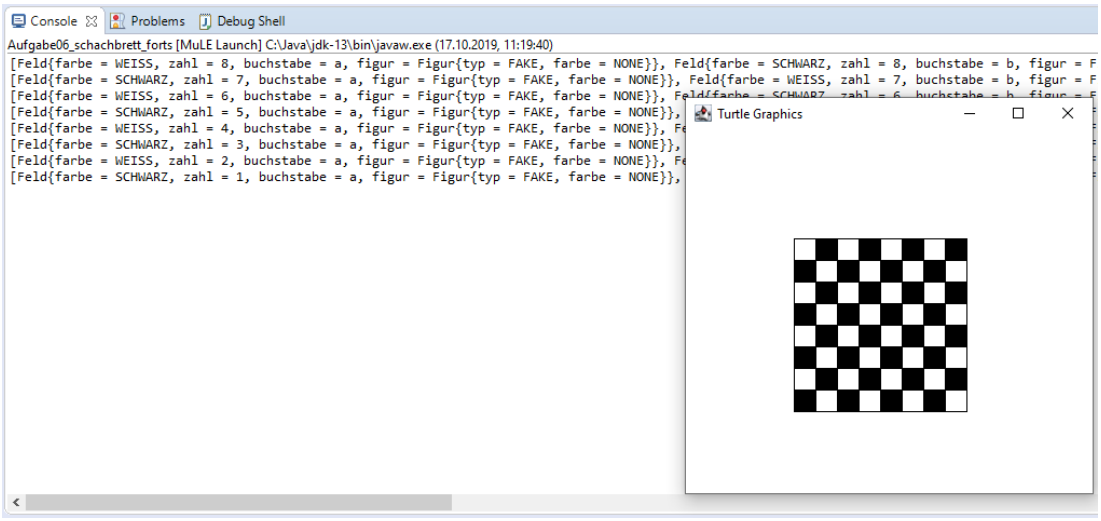

Abbildung 9: Ergebnis der *Schachbrett* Aufgabe.

12.4. Umfragen

Zur Evaluation des Programmierkurses werden anonyme Umfragen am Anfang und am Ende des Kurses durchgeführt. Um die beiden Umfragen trotzdem miteinander verbinden zu können, muss jeder Teilnehmer einen Pseudonym angeben, der nach einem bestimmten Muster zusammengesetzt wird.

Die erste Umfrage beinhaltet vor allem Fragen bezüglich des Hintergrundwissens (Programmierkenntnisse, Berufliche Erfahrung, Umfang des Informatikunterrichts und der Schule, etc.) der Teilnehmer sowie ihrer Erwartungen an den Kurs. Zusätzlich wird diese Umfrage genutzt, um die Meinung von Menschen ohne Programmiererfahrung zu bestimmten Sprachkonzepten abzufragen. So wurden z.B. in dem Jahr 2018 kurze Codeteile aus den Sprachen MuLE, Java und Quorum angeboten und die Studierenden sollten angeben, was der jeweilige abschnitt macht und welche Sprache für sie am Verständlichsten ist. Im Jahr 2019 wurden stattdessen Paare vom gleichen Code mit leichten syntaktischen Unterschieden vorgegeben und die Studierenden sollten jeweils ankreuzen, welches Beispiel für sie leichter zu verstehen ist. Die getesteten Konzepte waren: Blockgrenzen gekennzeichnet durch Schlüsselwörter oder Geschweifte Klammern, Einsatz von Semikolon am Ende einer Anweisung und Operatoren für Wertzuweisungen und Tests auf Gleichheit.

Die zweite Umfrage dient zum Sammeln von Feedback für den Kurs. Die Teilnehmer können Verbesserungsvorschläge in Form von Freitext geben sowie bestimmte Aspekte des Kurses (Verständlichkeit, Bearbeitungszeit für Aufgaben, Erklärung der Lösungen, Schwierigkeit etc.) an einer Skala bewerten. Bei dieser Umfrage wurde zusätzlich gefragt, ob die Teilnehmer alleine oder zu zweit gearbeitet haben, ob sie sich dabei von ihrem Partner abgehängt fühlten und ob sie

in Zukunft gerne weiter so arbeiten würden. Letztlich um den Wissensstand der Teilnehmer nach dem Kurs einschätzen zu können, müssen die Kursteilnehmer 10 kleine Wissensfragen, die sich auf Kursinhalte beziehen, beantworten.

DIE UMFRAGEBÖGEN UND DIE AUSWERTUNG FOLGEN.

A. Übersicht über getroffene Entscheidungen

1. MuLE soll eine integrierte Sprache sein, die mehrere Paradigmen erlaubt.
2. Beim Entwurf von MuLE sollen zunächst die Sprachkonstrukte für die prozedurale Programmierung entworfen werden.
3. Die Syntax von MuLE soll so gestaltet werden, dass Programme möglichst einfach lesbar sind.
4. MuLE soll streng, statisch und explizit typisiert sein.
5. Implizite Typkonvertierungen sollen nur von einem Untertyp in einen Obertyp durchgeführt werden.
6. Fast alle Deklarationen und Anweisungen werden durch ein Schlüsselwort eingeleitet.
7. Feld-, Variablen- und Parameterdeklarationen werden explizit durch Schlüsselworte unterschieden.
8. Es wird kein Semikolon oder Ähnliches verwendet, um Sprachkonstrukte voneinander zu trennen.
9. Blöcke werden stets durch Schlüsselwörter des jeweiligen Sprachkonstrukts gekennzeichnet.
10. Es ist möglich, ein Programm in mehrere Dateien zu trennen. Dies benötigt einen *import* Mechanismus.
11. Ein MuLE-Programm hat ein explizites Hauptprogramm. Eine MuLE-Bibliothek hat kein Hauptprogramm. Elemente einer Bibliothek können mit dem Sichtbarkeitsmodifikator `private` markiert werden, in diesem Fall werden sie nach außen nicht freigegeben.
12. Für die Ein- und Ausgabe werden vordefinierte Unterprogramme in einer Standardbibliothek zur Verfügung gestellt.
13. Die Standardbibliotheken werden explizit importiert.
14. Für jeden vordefinierten elementaren Datentyp wird jeweils eine Funktion für die Eingabe und eine Prozedur für die Ausgabe zur Verfügung gestellt.

15. Alle Datenobjekte werden implizit mit einem Standardwert initialisiert.

16. Das Schlüsselwort `default` bezeichnet den Standardwert des jeweiligen Datentyps.

17. Für Zuweisungen und Vergleiche wird das Wertmodell für Variablen zugrunde gelegt.

18. Für das Rechnen mit Zahlen wird jeweils ein Datentyp für ganze und rationale Zahlen zur Verfügung gestellt, die auf von der Hardware unterstützte Datentypen abgebildet werden.

19. Zeichenketten werden als elementare Daten mit Hilfe des Typs `string` repräsentiert.

20. Für Wahrheitswerte wird der Datentyp `boolean` mit entsprechenden logischen Operatoren bereitgestellt.

21. Als einziger benutzerdefinierbarer elementarer Datentyp werden Aufzählungstypen angeboten.

22. Für typheterogene Strukturen werden Verbundtypen zur Verfügung gestellt.

23. Bei einer Zuweisung an eine Variable von einem Verbundtyp wird das bisher in diesem Behälter gespeicherte Verbundobjekt ersetzt.

24. Für homogene Strukturen werden Listen anstelle von Arrays zur Verfügung gestellt.

25. Als Komponententyp werden beliebige Typen zugelassen, als Indizes nur nichtnegative ganze Zahlen, wobei die untere Schranke des Indexbereichs stets 0 ist.

26. Beim Vergleich zweier Listen werden deren Inhalte verglichen (sie sind gleich, wenn sie gleich lang sind und paarweise gleiche Elemente enthalten). Bei einer Zuweisung an eine Listenvariable wird eine Kopie der Liste angelegt. Analoge Regel gilt für Verbunde.

27. Für Listen gibt es eine Standardnotation. Für Verbunde gibt es keine Standardnotation für Wertzuweisungen, lediglich eine Zeichenkettenrepräsentation für Standardausgabe.

28. Referenzen werden so unterstützt, dass keine hängenden Referenzen entstehen können.

29. Eine Referenz ist typisiert. Dabei sind beliebige Datentypen als referenzierte Typen zulässig.

30. Variablen von Referenztypen werden automatisch mit `null` initialisiert.

31. Wie Referenztypen, so werden auch Referenzwerte durch das Schlüsselwort `reference` gekennzeichnet.

32. Es werden keine expliziten Operationen zur Speicherverwaltung angeboten. Die Belegung von Speicher auf der Halde erfolgt implizit bei der Auswertung eines Referenzausdrucks der Form `reference` value.

33. Eine Referenz wird durch den Postfixoperator `@` derefenziert. Dabei muss stets explizit dereferenziert werden. Eine Derefenzierung ist nur möglich, wenn die Variable einen Wert verschieden von null hat.

34. Es seien `r1` und `r2` Referenzvariablen vom selben referenzierten Typ. Dann liefert `r1 = r2` genau dann den Wert `true`, wenn `r1` und `r2` dasselbe Datenobjekt referenzieren. `r1@ = r2@` vergleicht dagegen die referenzierten Werte. Analog hat die Zuweisung `r1 := r2` Referenzsemantik3, d.h. `r1` und `r2` referenzieren danach dasselbe Objekt. Weiterhin hat `r1@ := r2@` Wertsemantik, d.h. der bisherige Wert von `r1@` wird durch den Wert `r2@` ersetzt.

35. Zwei Typen `t1` und `t2` sind gleich (`t1 = t2`), wenn einer der folgenden Fälle zutrifft:
    a) `t1` und `t2` sind jeweils von einem vordefinierten elementaren Datentyp, und $t1 \equiv t2$.
    b) `t1` und `t2` sind jeweils Aufzählungstypen, und $t1 \equiv t2$.
    c) `t1` und `t2` sind jeweils Verbundtypen, und $t1 \equiv t2$.
    d) `t1` und `t2` sind Listentypen mit Komponententypen `ct1` und `ct2`, und `ct1 = ct2`.
    e) `t1` und `t2` sind jeweils Referenztypen mit referenzierten Typen `rt1` und `rt2`, und `rt1 = rt2`.

    Dabei gilt $t1 \equiv t2$, wenn `t1` und `t2` identisch sind.

36. In der Zuweisung `v := e` müssen `v` und `e` vom gleichen Typ sein, beim Vergleich `e1 = e2` müssen `e1` und `e2` vom gleichen Typ sein.

37. Das Typkonzept ist bezüglich der Typkonstruktoren orthogonal, d.h. als Komponententypen können beliebige Typen verwendet werden.

38. Sequenzen von Anweisungen lassen sich zu einem Block zusammenfassen.

39. Für Verzweigungen wird eine bedingte Anweisung mit beliebig vielen optionalen `elseif`-Teilen und einem optionalem `else`-Teil angeboten. Die Zweige einer bedingten Anweisung sind Blöcke.

40. Auf eine Mehrfachverzweigung (`case` oder `switch`) wird verzichtet, da sie sich durch bedingte Anweisungen simulieren lässt.

41. MuLE bietet eine `loop`-Anweisung an, die über eine `exit`-Anweisung verlassen werden kann.

42. MuLE bietet eine `foreach`-Anweisung zur finiten Iteration über alle Elemente einer Liste an.

43. In einer `foreach`-Schleife muss eine Laufvariable deklariert werden, die nur in der Schleife selbst verwendet werden kann.

44. Alle Schleifen können über eine `exit`-Anweisung verlassen werden. Die `exit`-Anweisung beendet die unmittelbar umfassende Schleife.

45. Funktionen und Prozeduren werden nicht explizit unterschieden.

46. Operationen können nur auf der obersten Ebene deklariert werden.

47. Die Parameter einer Operation werden explizit durch das Schlüsselwort `parameter` gekennzeichnet.

48. Als einziger Mechanismus zur Parameterübergabe wird *Call by Value* unterstützt.

49. Kompositionen werden mit Operationen erweitert, um Objekte darstellen zun können.

50. Abstrakte Kompositionen können deklariert werden.

51. Abstrakte Kompositionen können abstrakte Operationen enthalten.

52. Es gibt kein explizites Sprachkontrukt für Schnittstellen, diese Semantik kann mit abstrakten Typen und Vererbung simulert werden

53. Es wird nur Einfachvererbung unterstützt.

54. Es wird auf explizite Konstruktoren verzichtet.

55. Bei einer Variablendeklaration kann der abstrakte Typ zusätzlich zum konkreten angegeben werden. Der konkrete Typ wird benötigt um ein Standardwert zu erzeugen und dient als der dynamische Typ. Der abstrakte Typ dient als der statische Typ.

56. Es gibt optionale Sichbarkeitsmodifikatoren `private` und `protected`, welche die Sichtbarkeit von Elementen eines Typs nach außen einschränken können. Elemente ohne eines Sichbarkeitsmodifikators sind nach außen sichtbar.

57. Operationen können redefiniert werden, dabei müssen sie mit einem Schlüsselwort `override` gekennzeichnet werden.

58. Schlüsselwörter `this` und `super` können verwendet werden, um den Scope innerhalb einer Vererbungshierarchie flexibler aufzulösen.

59. `this` ist keine Selbstreferenz, ein Objekt ist nicht in der Lage, eine Selbstereferenz weiterzugeben. Die Kommunikation zwischen zwei existierenden Objekten muss von außen organisiert werden.

B. Übersicht über Schlüsselwörter, Operatoren und Symbole

- **program** – Legt fest, dass es sich um ein Programm handelt. Hauptprogramm wird im Quelltext erwartet.
- **library** – Es handelt sich um eine Bibliotheksdatei. Hauptprogramm darf nicht vorkommen.
- **import** – Eine Bibliothek wird importiert.
- **as** – Im Kontext eines Imports wird dem Import ein Alias vergeben. In einem Ausdruck handelt es sich um eine Typumwandlung.
- **Java** – Markiert einen Java-Import.
- **main** – Start eines Hauptprogramms.
- **endmain** – Ende eines Hauptprogramms.
- **integer** – Primitiver Datentyp für ganze Zahlen.
- **rational** – Primitiver Datentyp für Gleitkommazahlen.
- **string** – Primitiver Datentyp für Zeichenketten.
- **boolean** – Primitiver Datentyp für Wahrheitswerte.
- **reference** – Gefolgt von einem Datentyp in spitzen Klammern deutet es auf einen Referenztyp. Vor einem Ausdruck bedeutet dies die Erschaffung einer neuen Referenz auf den ausgewerteten Wert.
- **list** – Es handelt sich um einen Listentyp.
- **type** – Leitet eine Typdeklaration ein.
- **endtype** – Schließt eine Typdeklaration ab.

- **composition** – Bei der Typdeklaration handelt es sich um einen Verbund.
- **enumeration** – Bei der Typdeklaration handelt es sich um einen Aufzählungstyp.
- **operation** – Leitet eine Operation ein.
- **endoperation** – Schließt eine Operation ab.
- **attribute** – Deklaration eines Attributs.
- **parameter** – Deklaration eines Parameters.
- **variable** – Deklaration einer Variable.
- **return** – Wertrückgabe.
- **exit** – Verlassen einer Schleife.
- **loop** – Beginn einer `loop`-Anweisung.
- **endloop** – Ende einer `loop`-Anweisung.
- **foreach** – Beginn einer `foreach`-Anweisung. Wird von einer Variablendeklaration gefolgt.
- **in** – Folgt der Variablendeklaration im Kopf der `foreach`-Anweisung und wird von einer Liste gefolgt. Die deklarierte Variable iteriert über die angegebene Liste.
- **do** – Leitet den Rumpf der `foreach`-Anweisung ein.
- **endforeach** – Ende einer `foreach`-Anweisung.
- **if** – Beginn einer `if`-Anweisung.
- **then** – Beginn eines Codeblocks nach einer Bedingung entweder in einem `if`, oder in einem `elseif` Zweig.
- **elseif** – Beginn eines `elseif` Zweigs einer `if`-Anweisung.
- **else** – Beginn eines `else` Zweigs einer `if`-Anweisung.
- **endif** – Ende einer `if`-Anweisung.
- **:=** – Zuweisungsoperator.
- **xor** – Exklusives ODER.
- **or** – ODER.

- **and** – UND.
- **not** – Negation eines Ausdrucks.
- **true, false** – Wahrheitswerte.
- **=, /=** – Prüfung auf Gleichheit bzw. Ungleichheit für alle Typen.
- **<, <=, >, >=** – Vergleichsoperatoren für numerische Typen.
- **+, -** – Addition, Subtraktion, Vorzeichen bei einer numerischen Zahl.
- **&** – Stringkonkatenation.
- ***, /** – Multiplikation, Division.
- **mod** – Modulo.
- **exp** – Potenzrechnung.
- **is** – Typprüfung.
- @ – Dereferenzierung.
- **( )** – Klammerung der Ausdrücke, Parameter einer Operation.
- **<>** – Typparameter.
- **[ ]** – Initialisierung von Listen, Zugriff auf Listenelemente.
- ** – Wiederholungsoperator bei einer Listeninitialisierung.
- **..** – Ganzzahliger Bereich bei einer Listeninitialisierung.
- **,** – Trennung von Parametern, Literalen eines Aufzählungstyps, Elementen einer Liste.
- **.** – Qualifizierter Zugriff.
- **;** – Signalisiert das Ende eines Sprachkonstrukts. Sprachkonstrukte mit Blöcken besitzen zwar jeweils ein eigenes Abschlusswort, müssen jedoch zum Zweck der Einheitlichkeit ebenfalls mit einem Semikolon beendet werden.

C. Grammatik

```
1  CompilationUnit:
2      ('program' | 'library') ID
3      Import*
4      ProgramElement*
5      MainProcedure?;
6
7  Import:
8      'import' [CompilationUnit] 'as' ID;
9
10 MainProcedure:
11     'main' Block 'endmain';
12
13 ProgramElement:
14     TypeDeclaration | Operation;
15
16 NamedElement:
17     EnumerationValue | TypeDeclaration | Feature | Import;
18
19 DataType:
20     BasicType | ComplexType | ReferenceType | ListType | GenericType;
21
22 ComplexType:
23     [TypeDeclaration];
24
25 BasicType:
26     'integer' | 'rational' | 'string' | 'boolean';
27
28 ReferenceType:
29     'reference' '<' DataType '>';
30
31 ListType:
32     'list' '<' DataType '>';
33
34 GenericType:
35     'generic' '<' ID '>';
36
37 TypeDeclaration:
38     Composition | Enumeration;
39
40 Enumeration:
41     'type' ID ':' 'enumeration' EnumerationValue (','
          EnumerationValue)* 'endtype';
42
```

```
43 EnumerationValue:
44     ID;
45
46 Composition:
47     'type' ID ':' 'composition'
48     Attribute*
49     'endtype';
50
51 Feature:
52     Attribute | VariableDeclaration | Parameter | Operation ;
53
54 Attribute:
55     'attribute' ID ':' DataType;
56
57 Parameter:
58     'parameter' ID ':' DataType;
59
60 Operation:
61     'operation' ID '(' (Parameter (',' Parameter)*)? ')'
62     (':' DataType)? Block 'endoperation' ;
63
64 Block:
65     Statement*;
66
67 Statement:
68         VariableDeclaration
69     |   Assignment
70     |   IfStatement
71     |   LoopStatement
72     |   ForEachStatement
73     |   'return' Expression
74     |   'exit';
75
76 VariableDeclaration:
77     'variable' ID ':' DataType ;
78
79 Assignment:
80     FeatureCall (':=' Expression)?;
81
82 LoopStatement:
83     'loop'
84     Block
85     'endloop';
86
87 ForEachStatement:
```

```
88      'foreach' VariableDeclaration 'in' Expression 'do'
89      Block
90      'endforeach';
91
92  IfStatement:
93      'if' Expression 'then' Block
94      ElseIf*
95      ('else' Block)?
96      'endif';
97
98  ElseIf:
99      'elseif' Expression 'then' Block;
100
101 Expression:
102     XorExpression;
103
104 XorExpression:
105     OrExpression ('xor' OrExpression)*;
106
107 OrExpression:
108     AndExpression ('or' AndExpression)*;
109
110 AndExpression:
111     EqualityExpression ('and' EqualityExpression)*;
112
113 EqualityExpression:
114     ComparisonExpression (('=' | '/=') ComparisonExpression)*;
115
116 ComparisonExpression:
117     AdditiveExpression (('<' | '<=' | '>' | '>=') AdditiveExpression)*;
118
119 AdditiveExpression:
120     MultiplicativeExpression
121         (('+' | '-' | '&') MultiplicativeExpression)*;
122
123 MultiplicativeExpression:
124     ExponentExpression (('*' | '/' | 'mod') ExponentExpression)*;
125
126 ExponentExpression:
127     TerminalExpression (('exp') TerminalExpression)*;
128
129 TerminalExpression:
130         FeatureCall
131     |   STRING
132     |   INTEGER
```

```
133     |   RATIONAL
134     |   ('true' | 'false')
135     |   'null'
136     |   ('+'|'-'|'not') TerminalExpression
137     |   'reference' TerminalExpression
138     |   'default'
139     |   '(' Expression ')'
140     |   ListInit;
141
142
143 FeatureCall:
144     [NamedElement] ('(' (Expression (',' Expression)*)? ')')?
145     FeatureCallAccessModifier? ('.' FeatureCall)?
146 ;
147
148 FeatureCallAccessModifier:
149     '[' Expression ']' FeatureCallAccessModifier? |
150     '@' FeatureCallAccessModifier?
151 ;
152
153
154 ListInit:
155     "[" (Expression (ListInitFunction | ListInitElements))? "]";
156
157 ListInitFunction:
158     ("**" | "..") Expression;
159
160 ListInitElements:
161     "," Expression)*;
162
163 ID: ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
164
165 INTEGER: INT;
166
167 RATIONAL: INT '.' INT ('E' ('+' | '-')? INT)?;
168
169 INT: ('0'..'9')+;
170
171 STRING:
172     '"' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ |
            !('\\'|'"') )* '"' |
173     "'" ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ |
            !('\\'|"'") )* "'" ;
174
175
```

```
176 ML_COMMENT: '/*' -> '*/';
177 SL_COMMENT: '//' !('\n'|'\r')* ('\r'? '\n')?;
178
179 WS: (' '|'\t'|'\r'|'\n')+;
180
181 ANY_OTHER: .;
```

Quelltext 39: MuLE Grammatik.

Abbildungsverzeichnis

Quelltextverzeichnis

Literatur

[1] IEEE Standard for Floating-Point Arithmetic. In: *IEEE Std 754-2008* (2008), Aug, S. 1–70

[2] Caspersen, Michael E. ; Christensen, Henrik B.: Here, There and Everywhere - on the Recurring Use of Turtle Graphics in CS1. In: *Proceedings of the Australasian Conference on Computing Education*. New York, NY, USA : ACM, 2000 (ACSE '00), S. 34–40. – URL http://doi.acm.org/10.1145/359369.359375. – ISBN 1-58113-271-9

[3] Gosling, James ; Joy, Bill ; Steele, Guy L. ; Bracha, Gilad ; Buckley, Alex ; Smith, Daniel: *The Java Language Specification, Java SE 12 Edition*. https://docs.oracle.com/javase/specs/jls/se12/jls12.pdf. 2019. – Accessed: 2019-08-20

[4] McIver, Linda ; Conway, Damian: Seven deadly sins of introductory programming language design. In: *Proceedings 1996 International Conference Software Engineering: Education and Practice* (1996), S. 309–316

[5] Pane, John ; Ratanamahatana, Chotirat ; Myers, Brad: Studying the language and structure in non-programmers' solutions to programming problems. In: *International Journal of Human Computer Studies* 54 (2000), 10, S. 237–264

[6] Qian, Yizhou ; Lehman, James: Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. In: *ACM Trans. Comput. Educ.* 18 (2017), Oktober, Nr. 1, S. 1:1–1:24. – URL http://doi.acm.org/10.1145/3077618. – ISSN 1946-6226