Sommersemester 2020

Programmierpraktikum - Block 6

AG Programmiersprachen und -werkzeuge

Stefan Schulz

Hinweis

Sie finden ein IntelliJ Projekt mit Vorgaben für die Aufgaben 1 und 2 im ILIAS.

Aufgabe 1 - UPN Rechner (20 Punkte)

Programme in Hochsprachen (vor allem Skriptsprachen wie Lua oder JavaScript) werden typischerweise mithilfe eines sogenannten Interpreters ausgeführt, statt diese zu Maschinen- oder Zwischencode (wie z.B. C) zu kompilieren. Dies hat den Vorteil, dass die Programme plattformunabhängig entwickelt und ausgeführt werden können. Dabei ist der Interpreter selbst ein Programm, das die Befehle des auszuführenden Programms einliest und die gewünschte Funktionalität mit eigenen Routinen simuliert.

Ziel dieser Aufgabe ist es, einen Interpreter für arithmetische Ausdrücke in umgekehrter polnischer Notation (UPN) mit verschiedenen Entwicklungsstrategien zu implementieren. Bei der UPN werden zuerst die Operanden aufgeführt und der Operator zuletzt (z.B. `1 2 +`, `3 4 * 2 +`). Durch diese Notation werden Operatorpräzedenzen und Klammern überflüssig, da es immer nur genau eine Möglichkeit gibt (nämlich zeichenweise von links nach rechts), einen Ausdruck auszuwerten. Wegen ihrer eingeschränkten Hardware verwendeten frühe elektronische Taschenrechner häufig diese Notation.

Sie dürfen bei der Umsetzung dieser Aufgabe davon ausgehen, dass die einzelnen Operanden und Operatoren der übergebenen arithmetischen Ausdrücke immer durch genau ein Leerzeichen voneinander getrennt sind. Im ILIAS finden Sie die Datei `RPNexpressions.txt`, welche Beispieleingaben enthält.

a) Naive Implementierung (7 Punkte)

Zunächst sollen Sie eine einfache Implementierung des Rechners umsetzen. Verwenden Sie hierbei das Programmskelett aus dem Paket `de.uni.marburg.plt.calculator.basic`:

- Erstellen Sie die Klasse `BasicRPNCalculatorImpl`, die das Interface `BasicRPNCalculator` realisiert. Implementieren Sie zunächst die Methode `evaluate(String expression, Map<String, String> variableAssignments)`. Diese soll den übergebenen String elementweise einlesen und dabei den arithmetischen Ausdruck auswerten. Dabei kann der Ausdruck Variablen enthalten. Aus diesem Grund werden mit der `Map variableAssignments` die Belegungen für die verschiedenen Variablen in einem Ausdruck übergeben. Am Ende soll das Ergebnis in Form eines `double` Wertes zurückgegeben werden, falls bei der Auswertung ein Fehler auftritt (z.B. Variable nicht belegt oder illegale Operation) soll stattdessen eine entsprechende Exception geworfen werden. Die folgenden binären Operationen sollen unterstützt werden:

  - `+` (Addition)
  - `-` (Subtraktion)
  - `*` (Multiplikation)
  - `/` (Division)
  - `%` (Modulo)
  - `^` (Potenzierung)

  `evaluate(String expression)` soll die überladene Methode mit einer leeren Liste aufrufen.

  Hinweis: Mit `Double.parseDouble(string)` kann ein als `String` vorliegender `double`-Wert eingelesen werden. Verwenden Sie zur Zwischenspeicherung der Operanden bzw. Zwischenergebnisse einen Stack. Achten Sie bei der Implementierung der Logik für die verschiedenen Operationen darauf, dass die Operanden in der richtigen Reihenfolge verwendet werden.

- Erweitern Sie `BasicRPNCalculatorImpl` nun zusätzlich um das Interface `BasicRPNConverter`.

  Dafür müssen Sie die Methode `convertInfixToRPN(String expression)` implementieren, die einen gegebenen UPN-Ausdruck (z.B. `1 2 +`) in einen entsprechenden Infix-Ausdruck (`(1 + 2)`) konvertiert. Sie dürfen dabei jeden Teilausdruck klammern.

- Testen Sie Ihre Implementierung mithilfe von JUnit Tests. Verwenden Sie dazu die Beispieleingaben aus `RPNexpressions.txt`. Überprüfen Sie dabei sowohl die Evaluierung der Ausdrücke, als auch die Konvertierung in Infix-Ausdrücke.

b) Implementierung mit Design-Pattern (13 Punkte)

Die Erweiterbarkeit der naiven Implementierung ist stark eingeschränkt. So müssen die Methoden `evaluate` und `convertInfixToRPN` für jede neu hinzugefügte Operation modifiziert werden. Abhilfe schafft hier das sogenannte Interpreter Pattern. Die Idee hierbei ist, die erlaubten Elemente (Operanden und Operatoren) über ein gemeinsames Interface als Klassenhierarchie zu modellieren. Dies hat einerseits den Vorteil, dass sie die Elemente selbst entscheiden, wie sie ausgewertet werden sollen und dadurch die Erweiterbarkeit des Programms verbessert wird. Andererseits können sich die Elemente untereinander referenzieren und somit Informationen austauschen. Verwenden Sie für diese Teilaufgabe das Programmskelett aus dem Paket `de.uni.marburg.plt.calculator.pattern`.

- In den Projektvorgaben finden Sie das Interface `Expression`, das die Methoden `evaluate(Map<String, Double> variableMapping)` und `toInfixExpression()` vorschreibt. Implementieren Sie die folgenden Klassen, die dieses Interface realisieren:
    - `Number`, repräsentiert eine Zahl im arithmetischen Ausdruck. Die Klasse enthält dafür ein `double` Feld, das mittels Konstruktor (`Number(double value)`) gesetzt wird. Außerdem verfügt sie über einen weiteren Konstruktor, der einen `String` als Argument entgegennimmt und dieses, sofern möglich, in einen `double`-Wert konvertiert und speichert.
    - `Variable`, repräsentiert eine Variable im arithmetischen Ausdruck. Die Klasse verfügt über das Feld `name`, das den Namen der Varaible speichert und mittels Konstruktor gesetzt werden kann. Die `evaluate`-Methode dieser Klasse soll eine `ArithmeticException` werfen, falls die Variable in der übergebenen Map nicht gefunden werden konnte.
    - Die **abstrakte** Klasse `Operator`, welche die Basis für die verschiedenen binären Operationen bildet. Diese verfügt über die Felder `left` und `right` vom Typ `Expression`, welche den linken bzw. rechten Operanden der Operation darstellen. Die beiden Felder sollen mittels Konstruktor gesetzt werden. Weiterhin soll die Klasse die **abstrakte** Methode `getOperatorSymbol()` enthalten, welche einen `String` zurückgibt. Diese Methode soll in den Konkreten Implementierungen von `Operator` das Symbol des Operators zurückgeben.
    - Erstellen Sie pro Operation (+, -, *, /, %, ^) eine konkrete Implementierung von `Oprator`.
- Erstellen Sie nun die Klasse `PatternRPNCalculatorImpl`, die das Interface `PatternRPNCalculator` realisiert.

  Implementieren Sie zunächst die Methode `convertStringToExpression(String expression)`, die analog zu (a) einen UPN-Ausdruck entgegennimmt und einliest. Statt das Ergebnis des Ausdrucks zurückzugeben soll diese Methode aber ein `Expression`-Element zurückgeben, das transitiv den gesamten Ausdruck enthält. Bei einer Eingabe von `1 2 +` würde diese Methode beispielsweise ein `Addition`-Objekt zurückgeben, das als `left` ein `Number`-Objekt mit dem Wert `1` und als `right` ein `Number`-Objekt mit dem Wert `2` enthält.

  Implementieren Sie anschließend die Methode, `evaluate(String expression, Map<String, Double> variableMappings)`, die den übergebenen UPN-Ausdruck zunächst in eine `Expression` konvertiert und diese anschließend gemäß der Variablenbelegung `variableMappings` auswertet.
- Testen Sie Ihre Implementierung mithilfe von JUnit Tests. Verwenden Sie dazu die Beispieleingaben aus `RPNexpressions.txt`. Überprüfen Sie dabei sowohl die Evaluierung der Ausdrücke, als auch die Konvertierung in Infix-Ausdrücke.

Aufgabe 2 - Morsecode-Übersetzer (20 Punkte)

Im Bereich der Programmiersprachen spielen sogenannte Compiler eine wichtige Rolle. Sie sind dafür zuständig, ein Programm in einer Quellsprache (z.B. Java) in ein äquivalentes Programm in Maschinensprache übersetzen. Die kompilierte Version des Programms kann anschließend auf dem Rechner ausgeführt werden.

Diese Aufgabe beschäftigt sich mit drei essentiellen Bausteinen eines Compilers: Lexikalische Analyse (Tokenisierung), Zwischencodeerzeugung und Codegenerierung. Hierzu sollen Sie ein Programm implementieren, das Textnachrichten zunächst in Morsecode und anschließend in Binärsignale übersetzt. Verwenden Sie bei Ihrer Implementierung das Programmskelett aus dem Paket `de.uni.marburg.plt.morsecode`.

a) Text-to-Morse (12 Punkte)

In dieser Teilaufgabe sollen Sie zunächst ein Programm entwickeln, das Textnachrichten zu Morsecode bzw. Morsecode zu Textnachrichten konvertieren kann.

Morsecode findet vor allem in der Telegraphie Anwendung, um Nachrichten über lange Strecken über per Licht, Ton oder über elektrische Signale zu übertragen. Morsecode verwendet zwei verschiedene Grundsymbole: `.` (`dit`, kurz) und `-` (`dah`, lang)

Kombinationen dieser beiden Symbole werden benutzt, um das Alphabet, Zahlen, sowie die gängigsten Satzzeichen zu kodieren (siehe https://de.wikipedia.org/wiki/Morsezeichen). Das heißt also, dass Morsenachrichten äquivalent zu Textnachrichten sind, die Länge einer Morsenachricht also der Zeichenlänge einer Textnachricht entspricht. Im Interface `MorseCodeTranslator` finden Sie die Arrays `characters` (enthält Buchstaben, Zahlen und Satzzeichen) und `morse` (enthält die entsprechenden Morsezeichen zu den Einträgen in `characters`). Die Arrays sind so angeordnet, dass das Morsezeichen auf Index `i` in `morse` dem Textzeichen auf Index `i` in `characters` entspricht.

Weiterhin werden im Morsecode Pausen verwendet, die einer Länge von 7 `dit`s entsprechen, um Worte voneinander zu trennen.

- Erstellen Sie die Klasse `MorseToken`, die `AbstractMorseToken` erweitert. Die Methode `toBinaryString()` müssen Sie zunächst nicht implementieren.
- Erstellen Sie die Klasse `MorseCodeTranslatorImpl`, die das Interface `MorseCodeTranslator` realisiert. Implementieren Sie hierzu die folgenden Methoden:
    - `convertCharacterToMorse` und `convertMorseToCharacter` die ein Textzeichen in den entsprechenden Morsecode bzw. ein Morsezeichen in das entsprechende Textzeichen konvertiert.
    - `tokenizeMessage(String message)`, die eine gegebene Nachricht tokenisiert und in eine Liste aus `TextToken`s konvertiert. Dabei soll zu jedem Wort und jedem Satzzeichen ein einzelnes `TextToken` erstellt und der Ergebnisliste hinzugefügt werden. Achten Sie insbesondere darauf, dass Leerzeichen nicht tokenisiert werden sollen. *Diese Methode ist äquivalent zur Analysephase eines Compilers.*
    - `encodeMessage(List<TextToken> clearTextMessage)`, die eine Liste von `TextToken`s als Argument erhält und diese in eine Liste aus `AbstractMorseToken`s übersetzt. Dabei soll zu jedem `TextToken` ein `MorseToken` erzeugt werden, das für jedes Zeichen des `TextToken`s das entsprechende Morsezeichen in einer Liste speichert. Beachten Sie, dass im Morsealphabet nicht

zwischen Groß- und Kleinschreibung unterschieden wird. *Die Funktionalität dieser Methode ist äquivalent zur Zwischencodegenerierung, die ein Compiler durchführt.*
    - `decodeMessage`, die eine übergebene Liste von `AbstractMorseTokens` erhält und diese tokenweise in eine Liste von `TextTokens` konvertiert.
    - `textTokenListToString(List<TextToken> textTokens)`, die einen übergebene Liste von `TextTokens` in einen String konvertiert. Achten Sie hierbei auf die korrekte Setzung von Leerzeichen.
- Testen Sie Ihre Implementierung mit JUnit-Tests. Im ILIAS finden Sie die Datei `TextToMorse.txt`. Diese enthält Beispieleingaben zum Testen. Jede Eingabe besteht aus zwei Zeilen. Dabei ist die erste Zeile die Nachricht im Klartext, die zweite der Morsecode zu dieser Nachricht, wobei einzelne Morsecodes durch ein Leerzeichen voneinander getrennt sind und Wörter durch sieben Leerzeichen. Das Ende der Testdaten ist durch eine Leerzeile markiert. Gehen Sie beim Testen wie folgt vor:
    - Tokenisieren Sie die Eingabenachricht mit `tokenizeMessage`. Konvertieren Sie das Ergebnis der Methode mit `textTokenListToString` in einen String und überprüfen Sie anschließend, ob dieser **exakt** der Originalnachricht entspricht.
    - Übergeben Sie die tokenisierte Nachricht an `encodeMessage`, um daraus eine Liste von `AbstractMorseTokens` zu erzeugen. Verwenden Sie diese anschließend als Eingabe für `decodeMessage`, um diese zurück in eine Liste von `TextTokens` zu konvertieren. Überprüfen Sie anschließend, ob die beiden `TextToken`-Listen übereinstimmen, indem Sie tokenweise überprüfen, ob die Tokens den gleichen Klartext enthalten. Beachten Sie, dass `encodeMessage` die konvertierte Nachricht implizit in Kleinbuchstaben umwandelt. Dies bedeutet, dass Sie beim Vergleich einer dekodierten Nachricht mit der Originalnachricht die Groß- und Kleinschreibung ignorieren müssen.

b) Morse-to-Binary (8 Punkte)

Damit eine Morsenachricht beispielsweise elektrisch übertragen werden kann, muss diese in eine binäre Signalfolge konvertiert werden. Dabei gelten die folgenden Regeln:

- Kurze Signale (`.`) werden als eine `1` übertragen.
- Lange Signale (`-`) werden als `111` übertragen.
- Zwischen zwei Symbolen eines Buchstaben wird immer genau eine `0` eingefügt
- Zwischen zwei Buchstaben eines Wortes werden immer genau drei Nullen (`000`) eingefügt. (z.B. `MR = --  .-. = 1110111 000 1011101`)
- Zwischen zwei Worten wird immer eine Pause, bestehend aus sieben Nullen (`0000000`) eingefügt.

Erweitern Sie nun Ihre Implementierung aus Aufgabenteil (a) so, dass `MorseCodeTranslatorImpl` die Listen aus Morsezeichen in Binärsignale umwandeln kann:

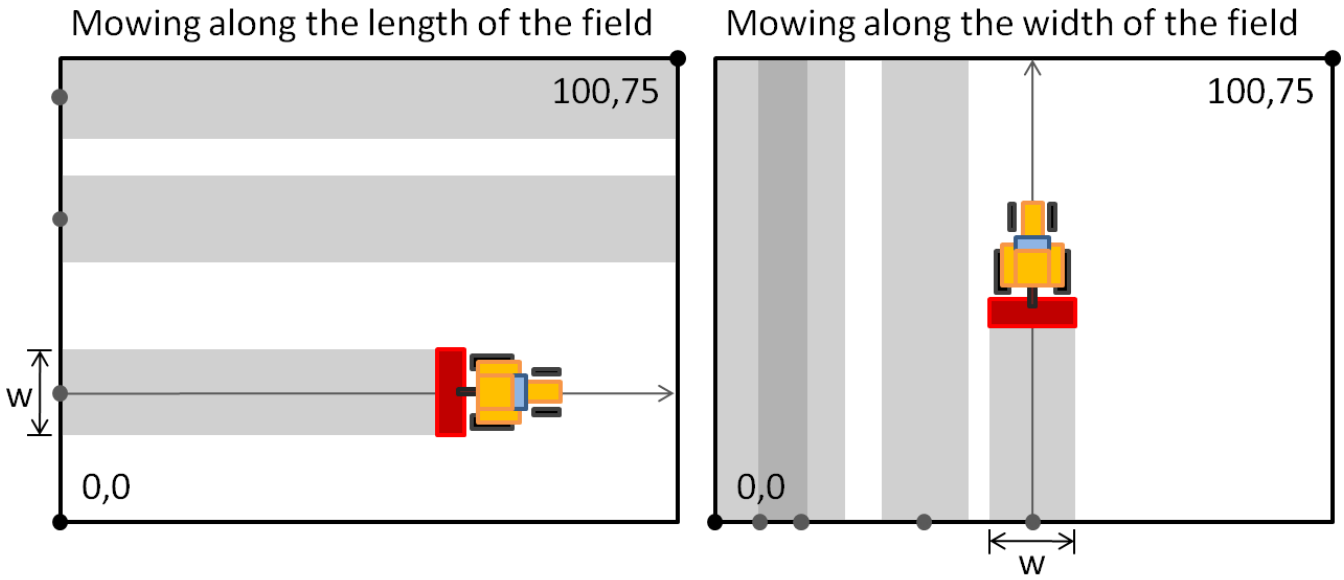
- Implementieren Sie zunächst die Methode `toBinaryString()` in `MorseToken` so, dass diese die im Token enthaltene Liste von Morsezeichen in einen `String` umwandelt, der die Binärdarstellung des Tokens enthält.
- Erweitern Sie `MorseCodeTranslatorImpl` um das Interface `MorseCodeTransmitter`. Dazu müssen Sie die folgenden Methodne implementieren:
  - `convertMorseCodeToBinary(List<AbstractMorseToken> morseCode)`, die eine Liste von `AbstractMorseToken`s enthält und diese zu einem Binärstring zusammenfügt. Achten Sie bei der Implementierung dieser Methode darauf, dass zwischen den einzelnen Worten (also Tokens) eine Pause eingefügt werden muss.
  - `convertBinaryToMorseCodes(String binary)` die einen Binärstring in eine Liste von `AbstractMorseToken`s konvertiert.
- Testen Sie Ihre Implementierung, indem Sie aus einer Nachricht eine Liste von `AbstractMorseToken`s erzeugen und diese in einen Binärstring überführen. Verwenden Sie anschließend den Binärstring, um daraus wieder eine Textnachricht zu generieren, die Sie abschließend mit der Originalnachricht vergleichen. Beachten Sie, dass Sie hierfür `encodeMessage` verwenden müssen und somit beim Vergleich der beiden Textnachrichten die Groß- und Kleinschreibung ignorieren müssen. Im ILIAS finden Sie die Datei `TextToBinary.txt`. Diese enthält Beispieleingaben zum Testen. Jede Eingabe besteht aus zwei Zeilen. Dabei ist die erste Zeile die Nachricht im Klartext, die zweite der aus dem dazugehörigen Morsecode generierte Binärstring. Das Ende der Testdaten ist durch eine Leerzeile markiert.

Aufgabe 3 - Rasenmäher (20 Punkte)

Quelle: https://open.kattis.com/problems/lawnmower

Die *International Collegiate Soccer Competition (ICSC)* verwendet genormte Fußballfelder mit einer Länge von 100 Metern und einer Breite von 75 Metern. Jede Woche wird der Rasen dieser Felder auf die gleiche Weise gemäht:

- Zuerst wird das Spielfeld `nX` mal der Länge nach gemäht (linke Grafik).
- Anschließend wird das Spielfeld `nY` mal der Breite nach gemäht (rechte Grafik).



Die ICSC hat einen neuen Gärtner namens Guido eingestellt. Guido ist sehr zerstreut und anstatt strukturiert vorzugehen, fängt er lieber an zufällig gewählten Stellen der Seitenlinie an zu mähen.

Damit die Spielfelder der ICSC dennoch perfekt gemäht sind, hat er Sie damit beauftragt, ein Programm zu entwickeln, das überprüft, ob er jede Stelle auf dem Feld mindestens einmal gemäht hat.

Eingabe

Die Eingabe erfolgt in Form einer Textdatei. Diese besteht aus maximal 50 Testfällen, wobei **jeder** dieser Testfälle aus drei Zeilen besteht.

Die erste Zeile besteht aus zwei ganzen Zahlen, `nX` und `nY` mit `0 < nX < 1000` und `0 < nY < 1000`, sowie einer Gleitkommazahl `w` mit `0 < w <= 50`, die die Schneidbreite des Rasenmähers darstellt.

Die zweite Zeile enthält `nX` Gleitkommazahlen, `xI` mit `0 <= xI <= 75`, die die Startpositionen der Mähfahrten entlang der Länge des Spielfelds darstellen.

Die zweite Zeile enthält `nY` Gleitkommazahlen, `yI` mit `0 <= yI <= 75`, die die Startpositionen der Mähfahrten entlang der Breite des Spielfelds darstellen.

Das Ende der Datei wird mit der Zeile `0 0 0.0` angezeigt. Hierfür soll keine Ausgabe erfolgen.

Ausgabe

Ihr Programm soll für jeden Mähversuch `true` ausgeben, wenn Guido alle Stellen des Rasens mindestens einmal gemäht hat und `false`, falls er es nicht geschafft hat.

Tests

Im ILIAS finden Sie die Dateien `rasenmaeher_in.txt` (Input) und `rasenmaeher_out.txt` (Output), mit der Sie Ihre Implementierung auf Korrektheit testen können.

Beispiel

Eingabe:

```
8 11 10.0
0.0 10.0 20.0 30.0 40.0 50.0 60.0 70.0
0.0 10.0 20.0 30.0 40.0 50.0 60.0 70.0 80.0 90.0 100.0
8 10 10.0
0.0 10.0 20.0 30.0 40.0 50.0 60.0 70.0
0.0 10.0 30.0 40.0 50.0 60.0 70.0 80.0 90.0 100.0
4 5 20.0
70.0 10.0 30.0 50.0
30.0 10.0 90.0 50.0 70.0
4 5 20.0
60.0 10.0 30.0 50.0
30.0 10.0 90.0 50.0 70.0
0 0 0.0
```

Ausgabe:

```
true
false
true
false
```