Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

ClojureBlocks: Ein visueller Editor für Clojure

Bachelorarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von
Jakob Walter Handke

Beginn der Arbeit:	03. Mai 2023
Abgabe der Arbeit:	26. Juli 2023
Erstgutachter:	Prof. Dr. Michael Leuschel
Zweitgutachter:	Dr. Jens Bendisposto

Selbstständigkeitserklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Duisburg, den 26. Juli 2023

Jakob Walter Handke

Zusammenfassung

Diese Bachelorarbeit befasst sich mit der Erarbeitung und Implementierung einer visuellen Programmierumgebung für die funktionale Programmiersprache Clojure. Die zu entwickelnde Programmierumgebung soll zu Beginn der Lehre der funktionalen Programmierung und der Programmiersprache Clojure eingesetzt werden.

Die Programmierumgebung soll es ermöglichen, mit einem blockbasierten Nutzerinterface Programme zu erstellen und für selbige evaluierbaren Clojure-Code zu generieren. Außerdem soll sie den generierten Code evaluieren und die Evaluationsresultate darstellen. Zur Unterstützung insbesondere zu Beginn der Lehre soll sie außerdem Aufrufe von Higher-order-Funktionen expandiert darstellen können, sodass diese genauer inspiziert werden können.

Verwandte Arbeiten werden vorgestellt, Architekturentscheidung diskutiert und Features der Programmierumgebung besprochen, die für den Einsatz in der Lehre hilfreich sind. Zuletzt wird die Funktionalität und Verwendbarkeit der Programmierumgebung anhand von zwei Programmieraufgaben getestet und bewertet.

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Bachelorarbeit befasst sich mit der Erarbeitung und Implementierung einer visuellen Programmierumgebung für die funktionale Programmiersprache Clojure. Die Programmierumgebung soll besonders für den Einsatz zu Beginn der Lehre der funktionalen Programmierung beziehungsweise der Programmiersprache Clojure geeignet sein. Für die professionelle Softwareentwicklung ist sie ausdrücklich nicht vorgesehen.

1.1 Motivation

Weltweit lernen die meisten Informatikstudierenden zu Beginn ihres Studiums eine vorwiegend imperative Programmiersprache wie Java oder Python, die wenigsten lernen eine funktionale Sprache wie Haskell oder Scheme [SHBLS21]. Auch der Autor dieser Arbeit hat als erste Programmiersprache im Studium Java gelernt. Im Modul „Einführung in die Funktionale Programmierung“ hat der Autor die Grundlagen der funktionalen Programmierung sowie die Programmiersprache Clojure gelernt. Dabei sind ihm einige Schwierigkeiten und Besonderheiten bei der Umstellung von der imperativen Programmierung zur deklarativen oder funktionalen Programmierung aufgefallen. Dazu zählen Funktionen als First-Class-Objekte, persistente Datenstrukturen, Funktionen höherer Ordnung sowie pure Funktionen. Einige davon existieren auch in imperativen Programmiersprachen, wie First-Class-Funktionen in Swift oder persistente Datenstrukturen in Java. Allerdings treten diese, anders als in funktionalen Sprachen, nur vereinzelt auf.

Die genannten Schwierigkeiten bei der Umstellung sind nicht exklusiv beim Autor aufgetreten, sondern treten, nach Besprechung mit dem Dozenten, bei vielen Studierenden dieses Moduls auf.

Zur Unterstützung der Lehre im genannten Modul oder in anderen Modulen soll ein visueller Editor für Clojure entwickelt werden. Dieser soll die Verständnishürden am Anfang der Lehre bei der Lisp-typischen Syntax mindern und den Fokus von syntaktisch korrektem Code hin zu den Konzepten der funktionalen Programmierung und der Programmiersprache Clojure lenken.

1.2 Grundlagen

1.2.1 Clojure

Clojure [Hic20] ist ein dynamischer und funktionaler Lisp-Dialekt. Die erste Version wurde von Rich Hickey im Jahr 2007 veröffentlicht. Die Sprache wird kompiliert und auf der JVM ausgeführt und ist dadurch für fast alle Plattformen verfügbar. Durch die beidseitige Interoperabilität mit Java können Komponenten in Clojure in bestehende Java-Anwendungen eingebunden werden.

Clojure beinhaltet unveränderbare, persistente Datenstrukturen und Unterstützung für nebenläufige Programmierung mittels transaktionalem Speicher und asynchronen Agenten. ClojureScript ist ein Compiler für Clojure nach JavaScript. Mit ClojureScript können auch Webanwendungen funktional erstellt werden und die meisten Funktionen und Vorteile von Clojure genutzt werden [McG11].

1.2.2 Visuelle Programmierung

„Visual programming is programming in which more than one dimension is used to convey semantics“ schreibt Burnett [Bur99]. Der Zeilenumbruch im klassischen, textbasierten Programmieren, also die zweite Dimension neben dem Text in einer Zeile, wird dabei als nicht bedeutungstragend betrachtet. Konkret lässt sich visuelle Programmierung dadurch beschreiben, dass Programme nicht durch Texteingabe, sondern durch eine visuelle Beschreibung wie Zeichnen von Datenzusammenhängen oder durch Zusammensetzen von Bausteinen oder Ähnlichem erstellt werden.

Die Ziele der visuellen Programmierung sind nach Burnett, die Programmierung einem größeren beziehungsweise bestimmten Kreis von Menschen zugänglich zu machen sowie die Korrektheit und die Geschwindigkeit der Programmierarbeit zu erhöhen. Dazu definiert sie die folgenden vier Strategien für visuelle Programmiersprachen:

Konkretheit ist das Gegenteil von Abstraktheit und bedeutet für visuelle Programmiersprachen, dass ein Aspekt eines Programms immer durch eine bestimmte Instanz ausgedrückt wird.

Eine visuelle Programmiersprache ist nicht konkret, wenn beispielsweise eine Struktur durch eine andere Struktur ausgedrückt oder konstruiert werden kann.

Direktheit bedeutet für visuelle Programmiersprachen, dass ein spezifisches Objekt direkt verändert werden kann, ohne dass die Semantik textbasiert beschrieben werden muss.

Beispielsweise muss kein Text geschrieben werden, um ein Objekt zu verändern, sondern das Objekt wird durch Schalter oder Auswahlmenüs modifiziert.

Explizitheit bedeutet für visuelle Programmiersprachen, dass Semantik nicht hergeleitet werden muss, sondern direkt in der Visualisierung sichtbar ist.

Eine visuelle Programmiersprache ist explizit, wenn es zum Beispiel nicht notwendig ist, die zu erzeugende Programmiersprache zu kennen. So soll die Visualisierung der Programmiersprache alle Informationen über das Programm geben.

unmittelbares visuelles Feedback bietet eine visuelle Programmiersprache, wenn sie während der Bearbeitung des Programms Änderungen unmittelbar in der Visualisierung ausgibt. Tanimoto definiert dazu verschiedene Stufen von „Liveness“ [Tan90], die hier nicht näher beschrieben werden.

Kaučič und Asič sowie Poole zeigen, dass visuelle Programmierumgebungen zu Beginn des Lernprozesses die Probleme der Programmiersprache beziehungsweise des Programmierens ausblenden und den Fokus von syntaktisch korrektem Code hin zu den grundlegenden Schwierigkeiten der Programmierung lenken können [KA11, Poo15].

1.3 Verwandte Arbeiten

Bereits seit den 1960er Jahren werden visuelle Programmiersprachen und -umgebungen erforscht und implementiert [BD04]. Einige davon sollen hier vorgestellt werden.

1.3.1 Scratch

Das wohl prominenteste Beispiel für visuelle Programmierumgebungen ist Scratch [MRR+10]. Scratch wurde ab 2003 entwickelt und das erste mal 2007 veröffentlicht. Das ursprüngliche Ziel von Scratch war jungen Menschen den Einstieg in die Programmierung zu erleichtern. Mittlerweile wird Scratch auch in der Schule und der Erwachsenenbildung eingesetzt.

In Scratch werden Programme geschrieben, indem verschiedene Arten von Blöcken aneinandergereiht und verschachtelt werden. Die Blocktypen sind *command* Blöcke, die Anweisungen in der klassischen Programmierung entsprechen, *control structure* Blöcke, die zum Beispiel durch `if...else`-Anweisungen die Ausführung von *command* Blöcken steuern, *function* Blöcke, die, wenn sie aufgerufen werden, einen Wert zurückgeben und *trigger* Blöcke, die durch einen bestimmten Auslöser wie dem Drücken der Leertaste ausgeführt werden. Die Blöcke für Datenliterale in Scratch sind ähnlich überschaubar: Es existieren Blöcke für Zahlen, für boolsche Werte und für Zeichenketten, die in anderen Blöcken verwendet werden können.

Blöcke können in Scratch nur in sinnvollen Weisen kombiniert oder verschachtelt werden, was syntaktische Fehler eliminiert. Außerdem gibt es keine Unterbrechung durch Kompilation oder Interpretation, was Scratch immer live erscheinen lässt.

In Abbildung 1 ist ein Scratch-Projekt abgebildet, in dem ein Spiel implementiert ist. Ziel des Spiels ist es, einen Kreis durch das blaue Labyrinth zum grünen Viereck rechts am unteren Rand zu bewegen. Dazu werden Tastatureingaben in Bewegungen des Kreises übersetzt und bei jeder Bewegung geprüft, ob eine blaue Wand berührt wird. Falls das der Fall ist, bewegt sich der Kreis nicht weiter in diese Richtung.

1.3.2 MIT App Inventor

Der MIT App Inventor [PDV13] ist eine visuelle Programmierumgebung, die es erlaubt, mobile Anwendungen für iOS- und Android-Geräte zu erstellen. Als Browseranwendung gibt es dabei keine Hürde bei der Installation.

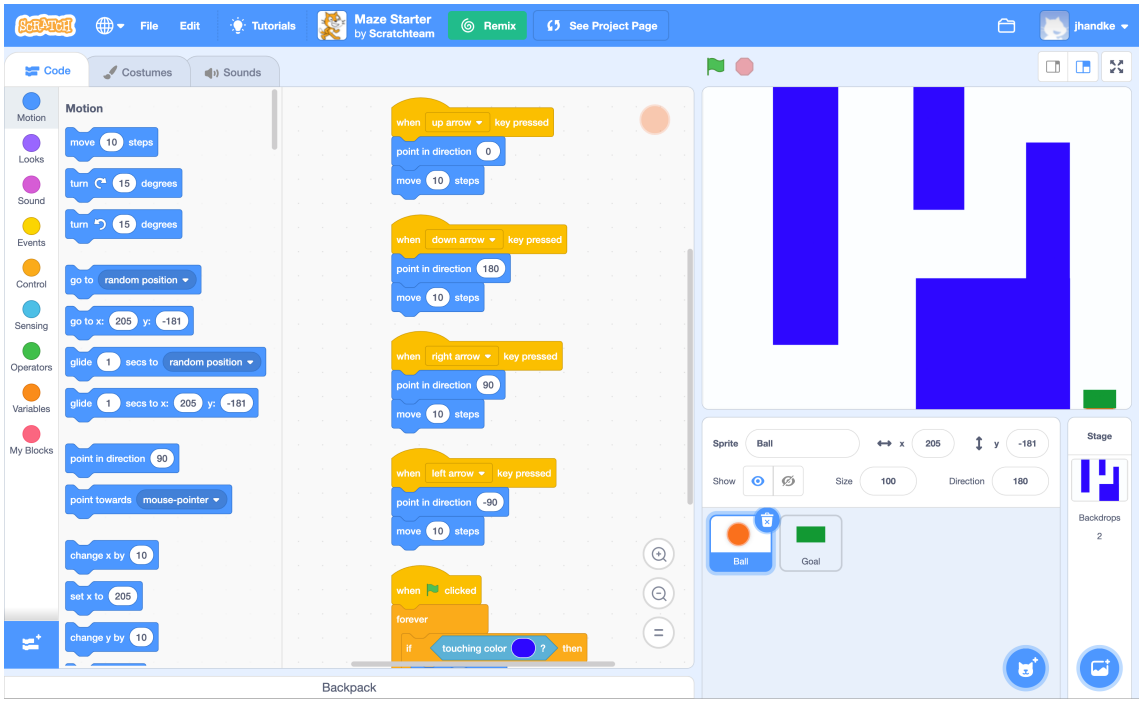


Abbildung 1: Labyrinth in Scratch

Die Programmierumgebung ist zweigeteilt, um Aussehen und Logik der zu entwickelnden Anwendung zu trennen. Einerseits gibt es einen visuellen Interface-Builder, in dem sichtbare Objekte wie Textfelder, Schaltflächen und Listen auf einem virtuellen Smartphone-Bildschirm positioniert und angepasst werden können. Andererseits gibt es einen blockbasierten Codeeditor, in dem die Anwendungslogik erstellt werden kann. Im Logikeditor gibt es die Möglichkeit, lokale und globale Variablen zu erstellen, Events vom Nutzerinterface abzufangen und das Nutzerinterface zu manipulieren. In Abbildung 2 ist ein Beispielprogramm gezeigt, welches bei jedem Klick auf einen Button die Zahl in einem Textfeld auf dem Bildschirm erhöht.

1.3.3 Alice

Cooper et al. haben im Jahr 1998 die visuelle Programmierumgebung Alice [CDP00] veröffentlicht. Mit Hilfe von Alice können dreidimensionale Animationen und Spiele erstellt werden. Die Konfiguration der virtuellen Welt und die Platzierung der visuellen Objekte im Raum geschieht über einen Szeneneditor, in dem die Objekte mittels drag & drop ohne Eingabe von Code bewegt werden können. Das Verhalten dieser Objekte und die Abfrage von Nutzereingaben wird mit einer blockbasierten Programmiersprache entwickelt, die auf Python aufbaut.

Dabei sind alle Kontrollstrukturen und Funktionen aus Python vorhanden. Außerdem existieren Methoden auf den visuellen Objekten, welche sie im Raum bewegen und welche sie initialisieren, zerstören, verstecken und zeigen.

In Abbildung 3a ist der Szeneneditor und in Abbildung 3b ist der Codeeditor von Alice abgebildet.

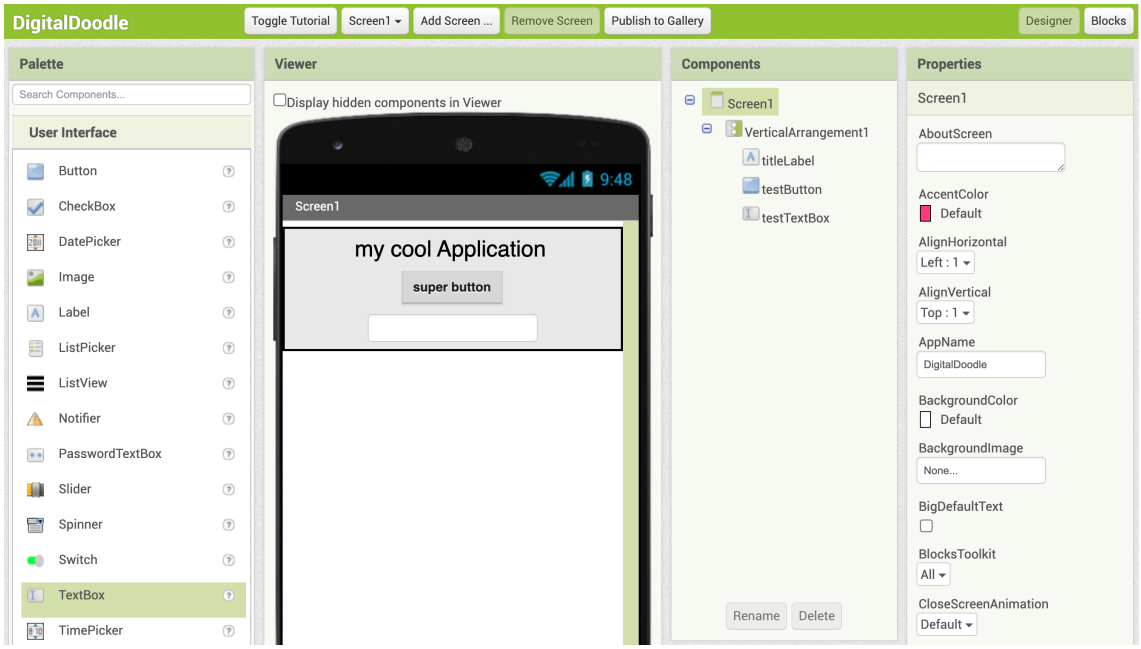
1.4 Zielsetzung

Das Ziel dieser Arbeit ist eine blockbasierte visuelle Programmierumgebung für die Programmiersprache Clojure.

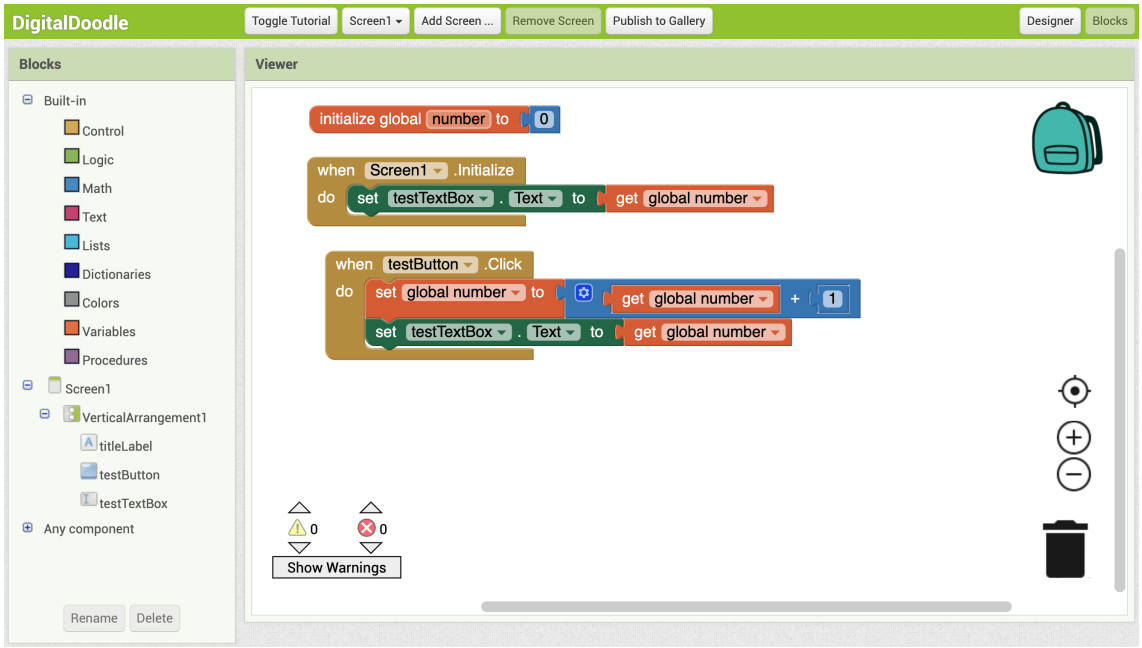
Für Datenstrukturen, Funktionsaufrufe, Literale, Macros und special forms sollen Blöcke erstellt werden, die aneinandergereiht beziehungsweise ineinander verschachtelt werden können. Diese Blöcke sollen mit der Maus aus einer Ablage auf einen Arbeitsbereich gezogen werden können. Aus dem Arbeitsbereich, der aus beliebig vielen Blöcken bestehen kann, soll ohne Benutzerinteraktion syntaktisch korrekter Clojure-Code generiert werden, der an geeigneter Stelle angezeigt wird. Im Idealfall wird der generierte Code direkt im Block angezeigt, aus dem er generiert wird.

ClojureBlocks soll weiter eine Funktion zur Evaluation des generierten Codes enthalten. Dabei ist es sinnvoll, dass verschachtelte Blöcke einzeln evaluiert werden können, um die Zusammenhänge zwischen den Ausdrücken besser nachvollziehen zu können. Das Ergebnis jeder Evaluation soll wie in einer Clojure-REPL hinter dem evaluierten Ausdruck stehen, zum Beispiel `(+ 3 4) => 7`. Fehler bei der Evaluation sollen ebenfalls im genannten Format angezeigt werden, zum Beispiel `(square 5) => Unable to resolve symbol: square in this context`.

Um Verständnishürden bei Higher-Order-Funktionen zu nehmen, soll in ClojureBlocks die Inspektion dieser möglich sein. Einzelne Funktionsaufrufe auf Elemente einer Sequenz sollen anhand von einem oder mehreren Elementen dargestellt werden, zum Beispiel wird `(map inc [1 2 3])` zu `(inc 1) => 2, (inc 2) => 3, (inc 3) => 4`. Dies soll auch für Higher-Order-Funktionen möglich sein, die die Länge einer Sequenz ändern (zum Beispiel `filter`) und die nur ein Element zurückgeben (zum Beispiel `reduce`).

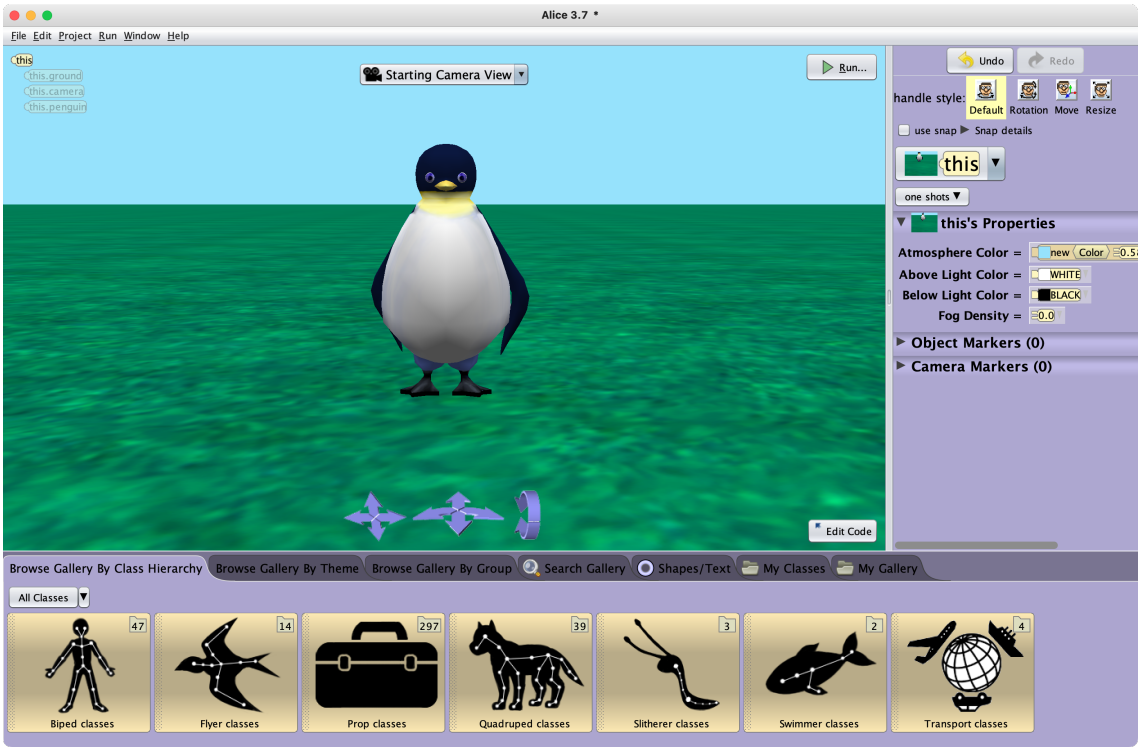


(a) Interface-Builder

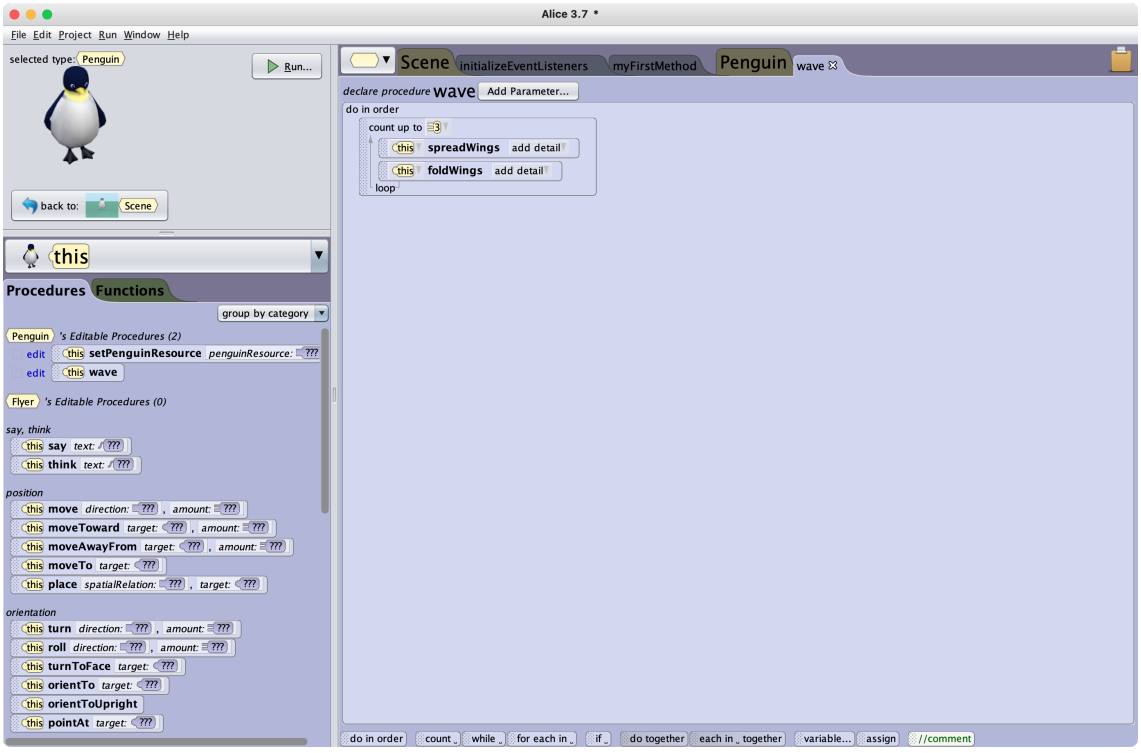


(b) Logik

Abbildung 2: MIT App Inventor



(a) Szene bearbeiten



(b) Prozedur bearbeiten

Abbildung 3: Programmierumgebung Alice

2 Architektur

Dieser Abschnitt soll Aufschluss über die verwendete Programmiersprache und die Softwarearchitektur von ClojureBlocks geben.

Die grundlegende Architekturentscheidung war die der Zielplattform für ClojureBlocks. In Frage kamen die Plattformen Desktop, Webanwendung und mobile App. Die Plattform Desktop wurde schnell verworfen, da der Code dann für alle drei führenden Betriebssysteme Linux, macOS und Windows funktionieren müsste. Außerdem müsste dazu ein Updatemanagement implementiert werden, welches neue Veröffentlichungen der Anwendung verteilt und installiert. Denselben Nachteil hätte eine plattformübergreifende Desktopanwendung, die mit Hilfe von Java beziehungsweise der Java Virtual Machine oder mit einem Framework wie Electron[1] und JavaScript entwickelt worden wäre.

ClojureBlocks als native mobile App zu entwickeln wäre ebenso schwierig geworden, da auch hier Code für die beiden dominanten mobilen Betriebssysteme Android und iOS geschrieben werden müsste. Mit Hilfe von App-Frameworks wie React Native[2] hätte eine plattformübergreifende native App für beide mobilen Betriebssysteme entwickelt werden können. Allerdings widersprach das Prinzip der installierbaren Anwendung dem Ziel, ClojureBlocks auf einfachste Weise verfügbar zu machen, da ClojureBlocks sowohl als Desktopanwendung, als auch als mobile App installiert und regelmäßig aktualisiert werden müsste.

Um den Implementierungsaufwand zu mindern sowie um die Installationshürde bei der Verwendung zu nehmen, wurde sehr früh vor der Entwicklung des ersten Prototypen die Entscheidung getroffen, ClojureBlocks als Browseranwendung zu entwickeln. Damit ist der Aufwand für Nutzende sowie Entwickelnde für eine plattformübergreifende Lösung am geringsten. Dass einige nützliche JavaScript- und ClojureScript-Bibliotheken existieren, sprach auch für die Plattform Browser.

2.1 Blockly

Zur Darstellung der Blöcke und der Anzeige von generiertem Code ist ein Nutzerinterface nötig. Dazu gäbe es die Möglichkeit, eine JavaFX- oder ClojureFX-Anwendung zu implementieren, die den Arbeitsbereich, Blöcke mit Verbindungs- und Eingabemöglichkeiten sowie einen Stapel zum Generieren von Blöcken im Arbeitsbereich bereitstellt. Dies hätte den Vorteil, dass das Nutzerinterface genau an die Anforderungen angepasst werden könnte. Andererseits existieren bereits Webanwendungen, welche mittels blockbasiertem Nutzerinterface Rätsel und Spiele bereitstellen. Beispiele davon sind clj-tiles[3] oder der Python-Editor BlockPy[4].

[1] https://www.electronjs.org/
[2] https://reactnative.dev/
[3] https://github.com/kloimhardt/clj-tiles
[4] https://think.cs.vt.edu/blockpy

Beide genannten Projekte sowie die in Abschnitt 1.3 vorgestellten verwandten Arbeiten Scratch und der MIT App Inventor verwenden die Bibliothek Blockly, um ein Nutzerinterface zu erstellen.

Da der Entwicklungsaufwand einer eigenen grafischen Oberfläche für drei Monate Bearbeitungszeit einer Bachelorarbeit zu hoch ist und Blockly bereits einige der geforderten Funktionen des Nutzerinterfaces bietet, wurde Blockly zur Entwicklung von ClojureBlocks verwendet.

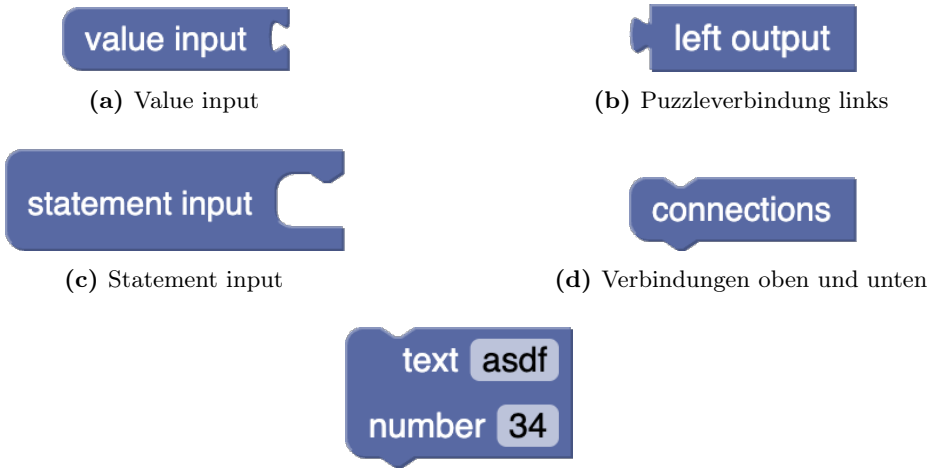
Blockly ist eine JavaScript-Bibliothek von Google, die einen visuellen Editor für Codeblöcke in Browseranwendungen einbindet. Sie umfasst vorgefertigte Blöcke für imperative Sprachen und kann Code für Dart, JavaScript, Lua, PHP und Python generieren. Um Blockly mit weiteren Sprachen oder Konzepten zu verwenden, können Blöcke mit Eingabefeldern oder Verbindungsmöglichkeiten zu anderen Blöcken definiert werden. Außerdem können Codegeneratoren eingebunden werden, die anhand eines Blocks, den eingegebenen Werten und den Verbindungen zu anderen Blöcken Code generieren.

Blöcke können verschiedene Eingabemöglichkeiten haben. Eine Möglichkeit ist der sogenannte *value input*, der in Abbildung 4a gezeigt ist. Mit dem *value input* kann mittels einer Puzzleverbindung ein anderer Block mit einer Puzzleverbindung, wie in Abbildung 4b gezeigt, verknüpft werden. Die andere Möglichkeit ist der *statement input*, der in Abbildung 4c gezeigt ist. Mit dem *statement input* können beliebig viele weitere Blöcke mit Verbindungsmöglichkeiten oben und unten, wie in Abbildung 4d gezeigt, verbunden werden, wobei die Reihenfolge der verknüpften Blöcke beachtet wird.

Als direkte Eingabemöglichkeit für Daten gibt es in Blockly Felder, die in einem Block platziert werden können, wie in Abbildung 4e gezeigt wird. Felder existieren für die Eingabe von Text, numerischen Werten, Winkel, boolschen Werte, Farben, Bildern und für Drop-Down-Menüs, wobei Felder auch selbst erstellt werden können, etwa für eine Schaltfläche. Diese können direkt mit Werten gefüllt werden, ohne dass ein weiterer Block verknüpft werden muss.

Benutzerdefinierte Blöcke können in Blockly mit Hilfe des Datenformats JSON oder mit der Blockly-API registriert werden. Dabei werden jeweils JavaScript-Objekte definiert, welche die Eingabemöglichkeiten, die Ausgabe sowie das Aussehen des Blocks beschreiben.

Da es viele gewünschten Funktionen beinhaltet und aktiv entwickelt wird, erwies Blockly sich als geeignet, darauf aufbauend ClojureBlocks zu entwickeln. Ein Nachteil an Blockly ist, dass die Ausgabe von Code und die Evaluation nicht pro Block erfolgen kann, da Blockly nicht vorsieht, Blöcke als Datendarstellung zu verwenden. Daher müssen der generierte Code und die Evaluationsergebnisse in einem Bereich außerhalb des Arbeitsbereichs angezeigt werden.



(a) Value input

(b) Puzzleverbindung links

(c) Statement input

(d) Verbindungen oben und unten

(e) Felder im Block

Abbildung 4: Verbindungsmöglichkeiten von Blöcken in Blockly

2.2 Sprache

ClojureBlocks soll eine Browseranwendung werden. Daher kamen als Programmiersprachen nur JavaScript beziehungsweise JavaScript-Derivate in Frage. Die Idee, ClojureScript zu verwenden, wurde vorerst verworfen, da die Blockly-API für viele Funktionen aufgerufen werden muss, welche in JavaScript geschrieben ist. Dadurch hätten kaum Vorteile und Funktionen von ClojureScript genutzt werden könnten und ein großer Teil des Codes wäre JavaScript-Interoperation gewesen. Außerdem liegt die Dokumentation von Blockly nur für JavaScript vor. Von daher lag es nahe, ClojureBlocks ebenfalls in JavaScript zu implementieren.

Als Folge der Entscheidung, zur Evaluation des Clojure-Codes SCI zu verwenden (s. Abschnitt 2.3), ist doch ClojureScript als Programmiersprache gewählt worden, da SCI sich nahtlos in ClojureScript-Programme eingliedert und kein Wrapper für JavaScript geschrieben werden muss. Die Nachteile daran sind der zusätzliche Kompilationsschritt bei der Entwicklung und Bereitstellung sowie viele JavaScript-Interoperationen, die die Lesbarkeit des Codes verschlechtern.

Als Build-System wird shadow-cljs [Hel19] verwendet. Shadow-cljs ist ein vollumfängliches Build-System für ClojureScript und JavaScript. Es kompiliert alle ClojureScript-Namensräume nach JavaScript und bündelt dann alle entstandenen Namensräume und separate JavaScript-Dateien in eine Datei, die einem Webserver bereitgestellt werden kann. Außerdem stellt shadow-cljs einige Hilfestellungen bei der Entwicklung bereit. So wird beim Kompilieren für die Entwicklung mit Hilfe der Anweisung `watch` ein Teil von shadow-cljs in die Webanwendung integriert, sodass Code live nachgeladen werden kann. Weiter stellt shadow-cljs mit `watch` einen einfachen Webserver bereit.

2.3 Evaluator

Um den von Blockly erzeugten Clojure-Code zu evaluieren, ist ein Evaluator nötig. Dieser soll eine Zeichenkette mit einem oder mehreren Clojure-Ausdrücken bekommen und das Evaluationsresultat zurückgeben. Die Rückgabe des Resultats oder eines Fehlers soll in einer geeigneten Datenstruktur erfolgen, die die Nutzerausgabe und ihre Formatierung vereinfacht. Dabei ist darauf zu achten, dass nicht der im Programm angezeigte Code evaluiert wird, da er im Browser vom Nutzer beliebig verändert werden kann. Außerdem können bereits Resultate und Fehlermeldungen von einer früheren Evaluation im Ausgabefenster stehen, die nicht evaluiert werden sollen.

Um die Gefahren und Probleme von Nutzercode im Programmcode von ClojureBlocks zu vermeiden, soll der von ClojureBlocks generierte Code ausschließlich in einer Sandbox evaluiert werden. Dazu wurde zunächst die Anbindung einer Leiningen-REPL in Betracht gezogen. Diese Idee wurde mit Hilfe der JavaScript-Bibliothek node-nrepl-client[5] umgesetzt. Sie brachte allerdings einige Nachteile in der Nutzbarkeit mit: Nutzende müssten Clojure und Leiningen lokal installieren, um ClojureBlocks verwenden zu können. Außerdem müssten sie vor der Verwendung eine headless REPL auf einem bestimmten Port und anschließend den Webserver für ClojureBlocks starten.

Zur niederschwelligen Nutzbarkeit wurde eine Möglichkeit gesucht, einen Clojure- oder ClojureScript-Evaluator in der Browserumgebung selbst zu starten. Es existieren bereits einige Webseiten mit eingebundenem Clojure-Evaluator, wie beispielsweise die 4ever-clojure Webseite[6]. Viele dieser Seiten verwenden den „simple clojure interpreter“ SCI[7] zur Evaluation des Codes.

SCI ist ein Interpreter für Clojure, der die meisten Funktionen von Clojure und Anwendungen eines Grundlagenkurses abdeckt. Er wird von Michiel Borkent entwickelt, der auch Babashka entwickelt, und ist unter der Eclipse Public License 1.0 veröffentlicht. SCI stellt eine Funktion `eval-string` bereit, die eine Zeichenkette als Eingabe entgegennimmt und versucht, die in der Zeichenkette enthaltenen Ausdrücke zu lesen und zu evaluieren. Die Funktion gibt dabei das Ergebnis der Evaluation als Wert oder im Falle eines Evaluationsfehlers eine Map mit der Fehlerbeschreibung zurück.

Die Evaluation von SCI kann konfiguriert werden. Beispielsweise können bestimmte Namensräume oder Symbole erlaubt oder verboten werden. So kann spezifisch kontrolliert werden, welche Ausdrücke evaluiert werden können und welche, statt evaluiert zu werden, einen Fehler erzeugen. Außerdem können Funktionen in SCI mit Funktionen im aufrufenden Programm überschrieben werden, was bei den verschiedenen `print`-Funktionen hilfreich ist. Mit diesen Vorteilen gegenüber einer manuellen Anbindung einer REPL stellte SCI sich als geeignet dar, die Evaluation in ClojureBlocks zu übernehmen.

[5] https://github.com/rksm/node-nrepl-client
[6] https://4clojure.oxal.org/
[7] https://github.com/babashka/sci

3 ClojureBlocks

In diesem Abschnitt werden die Kernfunktionen und die wichtigsten Namespaces von ClojureBlocks besprochen.

Der Code für ClojureBlocks ist in folgenden Git-Repositories zu finden:

https://codeberg.org/jhandke/ClojureBlocks

https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/handke-bachelor

Außerdem ist eine Live-Version zu finden unter:

https://jhandke.codeberg.page/ClojureBlocks

3.1 Core

Der Core-Namespace ist dafür zuständig, ClojureBlocks zu initialisieren. Dabei wird Blockly instanziiert und alle Blocktypen in Blockly registriert. Außerdem werden für alle Schaltflächen, die nicht zu Blockly gehören, wie zum Beispiel die Schaltfläche zur Evaluation des Codes, Eventhandler registriert. Weiter sind im Core-Namespace die Funktionen implementiert, die den von Blockly generierten Code anzeigen, formatieren und evaluieren.

Der Dialog zum Setzen von Optionen für die Ausgabe und die Evaluation ist ebenfalls in der Core-Komponente implementiert. Beim Start des Programms werden die gespeicherten Einstellungen aus dem Speicher des Browsers geladen (s. Abschnitt 3.9.2) und angewendet. Nach dem Ändern von Optionen werden diese ebenfalls angewendet und abgespeichert.

3.2 Blockly-Wrapper

Um die Funktionen von Blockly nur an wenigen Stellen aufzurufen, wurde ein Wrapper implementiert, der die häufigsten Aufrufe der Blockly-API kapselt. Blockly verwendet als Zustandsobjekt den sogenannten Workspace, der die Blöcke auf der Arbeitsfläche sowie deren Positionen und Verbindungen beinhaltet. Dieser Workspace wird im blockly-wrapper in einem Atom gespeichert.

Außerdem ist im blockly-wrapper eine Funktion implementiert, die den Bereich auf der Webseite, der vom Blockly-Arbeitsbereich belegt wird, bei einer Änderung der Fenstergröße entsprechend anpasst. Diese Funktion wurde sinngemäß aus der Entwicklerdokumentation zu Blockly von Google übernommen und in ClojureScript übersetzt.

3.3 Blöcke

Für alle Datenstrukturen, Literale, einige special forms, Macros und Funktionen der Standardbibliothek von Clojure wurden Blöcke in Blockly erstellt. Um besonders die zu Beginn der Lehre wichtigen Aspekte von Clojure abzudecken, wurden viele Funktionen aus den ersten REPL-Sitzungen[8] des Moduls „Einführung in die funktionale Programmierung“ hinzugefügt.

Da in ClojureScript weder Refs noch transaktionaler Speicher existieren, musste darauf verzichtet werden, diese in ClojureBlocks einzubauen. Atome hingegen sind sowohl in Clojure, als auch in ClojureScript verfügbar und funktionieren auf die gleiche Art und Weise. Agenten sind zum Zeitpunkt der Bearbeitung noch nicht in ClojureScript implementiert und fehlen daher auch in ClojureBlocks.

Die Blöcke sind in Kategorien unterteilt und werden innerhalb ihrer Kategorie in der Toolbox am linken Rand des Arbeitsbereichs angezeigt.

3.4 Generator

Der Generator ist dafür zuständig, die Blöcke in syntaktisch korrekten Clojure-Code zu wandeln. Dazu wurde für jeden Blocktypen eine Funktion implementiert, welche die Eingaben und die verschachtelten Blöcke ausliest und in die korrekte Clojure-Form übersetzt.

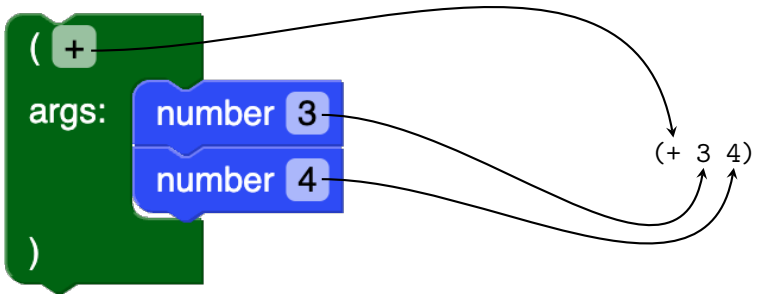
In Abbildung 5 ist beispielhaft der Zusammenhang zwischen Blöcken und dem zugehörigen Code gezeigt.

Die Reihenfolge der im Arbeitsbereich vorhandenen Blöcke spielt beim Generieren von Code eine Rolle. Blöcke werden von links nach rechts und von oben nach unten sortiert übersetzt.

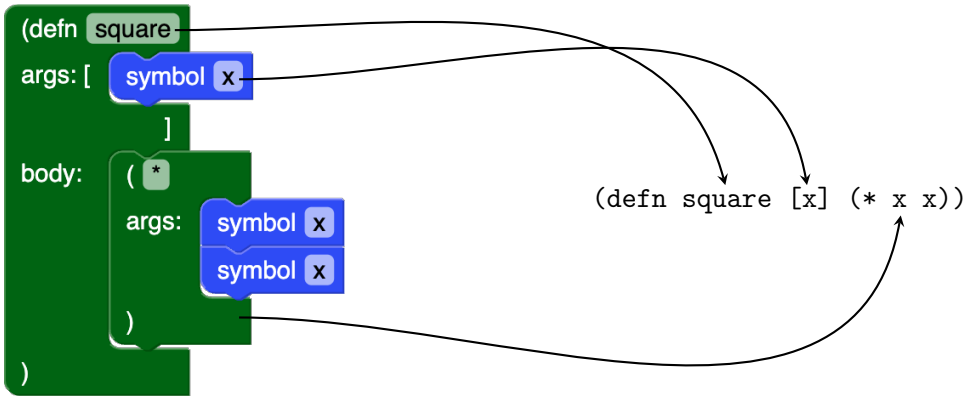
Beim Generieren von Code für einen Block ist zu beachten, dass weitere Blöcke innerhalb eines *statement inputs* verknüpft sein können. Das bedeutet, dass der Generator rekursiv auf den Blöcken in einem *statement input* aufgerufen werden muss. Dazu wurde eine Funktion `generate-statement-code` in `clojureblocks.generator.clojure` implementiert, die rekursiv Code für die Blöcke in einem *statement input* generiert und den Code für alle Blöcke zurückgibt.

Um nicht in jeder Generatorfunktion Zeichenketten manipulieren zu müssen, wurde eine Funktion `expression` implementiert, welche eine Zeichenkette mit den übergebenen Argumenten in einer Clojure-Liste formatiert zurückgibt. So muss an nur einer Stelle Code geschrieben werden, der Leerzeichen zwischen optionalen und potentiell nicht vorhandenen Argumenten entfernt.

[8] https://github.com/pkoerner/functional-programming-course



(a) Funktionsaufruf



(b) Funktionsdefinition

Abbildung 5: Zusammenhang Block und Code

3.5 Codeformatierung

Da der Codegenerator von Blockly nur den Text generiert und nicht formatiert, werden lange Ausdrücke in der Codevorschau abgeschnitten. Außerdem leidet die Lesbarkeit des Codes, wenn jeder Top-Level-Ausdruck in nur einer Zeile steht. Um dem entgegenzuwirken, wurde eine Lösung gesucht, Clojure-Ausdrücke zu formatieren. Dafür standen mehrere Methoden zur Verfügung:

Datenstrukturen und Code können mit `clojure.pprint/pprint` beziehungsweise in ClojureScript mit `cljs.pprint/pprint` formatiert werden, indem der jeweilige Funktionsaufruf im Ausdruck `(with-out-str body)` gekapselt wird. Die `pprint`-Funktionen nehmen allerdings keine Zeichenketten mit Clojure-Code entgegen, sondern nur Datenstrukturen

beziehungsweise Code. Daher müsste der generierte Code zunächst von einem Clojure-Reader gelesen werden und dann mit der `pprint`-Funktion formatiert werden. Außerdem ist der Formatierungsstil mit dieser Methode nicht einstellbar.

Die beiden Bibliotheken cljfmt[9] und zprint[10] hingegen können Zeichenketten mit Clojure-Code formatieren. Sie wandeln die Zeichenketten mit einem Reader in Datenstrukturen und formatieren sie anschließend. Die Bibliothek cljfmt ist dabei eher dazu geeignet, eine falsche Formatierung zu erkennen und Fehler zu beheben. Sie fügt keine Zeilenumbrüche hinzu und versucht hauptsächlich, Ausdrücke in verschiedenen Zeilen, die im Syntaxbaum auf der gleichen Ebene stehen, richtig einzurücken. zprint hingegen kann auch Ausdrücke, die in nur einer Zeile stehen, nach einem konfigurierbaren Schema formatieren, Zeilen zwischen Ausdrücken hinzufügen oder entfernen und Ausdrücke in verschiedenen Zeilen auf der gleichen Ebene im Syntaxbaum richtig einrücken.

Zur Codeformatierung in ClojureBlocks wurde zprint gewählt, da damit die umfangreichere Konfiguration möglich ist. Bevor Code in der Core-Komponente angezeigt wird, wird er mit Hilfe der Funktion `zprint/zprint-file-str` formatiert, sodass zwischen allen Top-Level-Ausdrücken eine leere Zeile steht und Zeilen nicht länger als 80 Zeichen lang werden.

3.6 Evaluation

Um die Evaluation zu kapseln und dadurch die verwendete Bibliothek SCI austauschbar zu machen, wurde ein Wrapper implementiert. Dieser Wrapper stellt die Funktion zum Evaluieren des von Blockly generierten Codes bereit. Diese `evaluate`-Funktion kapselt den Aufruf an die `eval-str*`-Funktion von SCI, welche einen String mit einem Clojure-Ausdruck sowie einen Kontext erhält, in dem der aktuelle Namespace und alle vorherigen Evaluationen gespeichert sind. `evaluate` gibt in einer Map den zu evaluierenden Ausdruck sowie das Resultat oder einen Fehler zurück. Um die Verwendung einer REPL zu simulieren und zu vermeiden, dass die Evaluation von beispielsweise `(range)` das Programm zum Absturz bringt, wird einerseits die Variable `*print-length*` zunächst auf den Wert 12 gesetzt. Andererseits wird vor der Evaluation von generiertem Code jeder Ausdruck in `(pr-str ...)` gekapselt. Dadurch werden lazy Sequenzen nicht vollständig evaluiert und ausgegeben, sondern nach 12 Elementen angehalten und das Resultat zurückgegeben. Die `*print-length*` ist in den Programmeinstellungen unten rechts veränderbar.

Weiter stellt der Wrapper eine Funktion zum internen Evaluieren von Clojure-Ausdrücken bereit, welche bei der Inspektion von Higher-Order-Funktionen verwendet wird. Diese Funktion kapselt den Ausdruck nicht in `(pr-str ...)`, um eine vollständige Ausgabe des Evaluationsresultats zu erhalten.

Außerdem stellt er eine Funktion zum Zurücksetzen des verwendeten Namespaces und von allen bisherigen Evaluationen bereit.

[9] https://github.com/weavejester/cljfmt
[10] https://github.com/kkinnear/zprint

Leider ist es mit Hilfe von SCI derzeit nicht möglich, eine maximale Evaluationszeit einzustellen. Die Möglichkeit, die maximale Rekursionstiefe von SCI vorab festzulegen, wurde vom Entwickler aufgrund von Fehlverhalten wieder entfernt. Aus diesen Gründen ist es in ClojureBlocks nicht möglich, Endlosschleifen vollständig zu unterbinden oder frühzeitig abzubrechen.

3.7 Higher-order-Funktionen

Um Higher-order-Funktionen zu erläutern, wurde eine Inspektion der Blöcke, die diese darstellen, implementiert. Sie ist erreichbar über das Kontextmenü des jeweiligen Blocks. Bei der Inspektion eines Higher-order-Funktion-Blocks öffnet sich ein Dialog, in dem die Higher-order-Funktion im Kontext genauer erläutert wird.

Unendliche Sequenzen und Funktionen darauf werden in der Inspektion nach zunächst 12 Aufrufen oder Schritten abgebrochen und weitere Aufrufe durch drei Punkte abgekürzt. Diese `preview-length` genannte Zahl ist in den Programmeinstellungen unten rechts veränderbar.

3.7.1 Map

Der Higher-order-Funktion `map` werden eine Funktion und mindestens eine Sequenz übergeben. Sie ruft die Funktion auf jedem Element der Sequenz auf und gibt die Ergebnisse als Sequenz in gleicher Reihenfolge zurück. Die Inspektion eines `map`-Blocks zeigt den Funktionsaufruf und dessen Ergebnis für jedes Element aus der Sequenz. Quellcode 1 zeigt die Inspektion eines `map`-Blocks.

Quellcode 1: Inspektion vom Block `(map inc (range 4))`

```
(map inc (range 4))
; => (1 2 3 4)

(inc 0) ; => 1
(inc 1) ; => 2
(inc 2) ; => 3
(inc 3) ; => 4
```

3.7.2 Filter

Die Higher-order-Funktion `filter` bekommt eine Funktion übergeben, die ein Argument bekommt und `true` oder `false` beziehungsweise einen truthy Wert oder `nil` zurückgibt. Außerdem bekommt `filter` eine Sequenz übergeben. Die übergebene Funktion wird mit jedem Element aus der Sequenz aufgerufen und anhand des Ergebnisses das Element aus der Sequenz gefiltert. Ist der Rückgabewert `true`, so bleibt das Element in der Sequenz

enthalten, ist er `false`, so wird es entfernt. Die Inspektion eines `filter`-Blocks zeigt, ähnlich wie die des `map`-Blocks, für jedes Element aus der Sequenz den Funktionsaufruf und das Ergebnis der Filterfunktion sowie einen Hinweis, ob das Element ausgefiltert wird. Quellcode 2 zeigt die Inspektion eines `filter`-Blocks.

Quellcode 2: Inspektion vom Block `(filter odd? (range 4))`

```
(filter odd? (range 4))
; => (1 3)

(odd? 0) ; => false (removed)
(odd? 1) ; => true
(odd? 2) ; => false (removed)
(odd? 3) ; => true
```

3.7.3 Remove

Die Higher-order-Funktion `remove` bekommt ebenfalls eine einstellige Funktion, die einen boolschen Wert beziehungsweise einen truthy Wert oder `nil` zurückgibt, sowie eine Sequenz übergeben. Sie entfernt alle Elemente aus der Sequenz, bei denen der Aufruf der übergebenen Funktion mit dem jeweiligen Element als Argument einen truthy Wert zurückgibt. `remove` entfernt also genau die Elemente, die von der `filter`-Funktion behalten worden wären und umgekehrt. Die Inspektion eines `remove`-Blocks zeigt, wie die Inspektion eines `filter`-Blocks, den Funktionsaufruf und das Ergebnis für jedes Element aus der Sequenz und einen Hinweis, ob das Element entfernt wird. Quellcode 3 zeigt die Inspektion eines `remove`-Blocks.

Quellcode 3: Inspektion vom Block `(remove odd? (range 4))`

```
(remove odd? (range 4))
; => (0 2)

(odd? 0) ; => false
(odd? 1) ; => true (removed)
(odd? 2) ; => false
(odd? 3) ; => true (removed)
```

3.7.4 Reduce

Der Higher-order-Funktion `reduce` werden eine zweistellige Funktion, optional ein Startwert und eine Sequenz übergeben. Falls der Startwert vorhanden ist, ruft `reduce` im ersten Schritt die übergebene Funktion mit dem Startwert und dem ersten Element der Sequenz auf, andernfalls mit den ersten beiden Elementen der Sequenz. Anschließend ruft sie die übergebene Funktion mit dem Resultat des ersten Funktionsaufrufs und dem nächsten Element der Sequenz auf und fährt so fort, bis das letzte Element verarbeitet wurde. Die

Inspektion eines `reduce`-Blocks zeigt die Funktionsaufrufe mit ihrem Ergebnis einzeln. Quellcode 4 zeigt die Inspektion eines `reduce`-Blocks und Quellcode 5 zeigt die Inspektion eines `reduce`-Blocks mit einem Startwert.

Quellcode 4: Inspektion vom Block `(reduce + [1 4 7 11])`

```
(reduce + [1 4 7 11])
; => 23

(+ 1 4) ; => 5
(+ 5 7) ; => 12
(+ 12 11) ; => 23
```

Quellcode 5: Inspektion vom Block `(reduce + 5 [1 4 7 11])`

```
(reduce + 5 [1 4 7 11])
; => 28

(+ 5 1) ; => 6
(+ 6 4) ; => 10
(+ 10 7) ; => 17
(+ 17 11) ; => 28
```

3.7.5 Apply

Die Higher-order-Funktion `apply` löst das Problem, eine Funktion mit einer Sequenz von Argumenten aufzurufen. Sie bekommt eine Funktion, beliebig viele Argumente sowie eine Sequenz mit weiteren Argumenten übergeben. Dann wird die übergebene Funktion mit den übergebenen Argumenten sowie den Argumenten in der Sequenz aufgerufen. Die Inspektion eines `apply`-Blocks zeigt den so zusammengesetzten Funktionsaufruf. Quellcode 6 zeigt die Inspektion eines `apply`-Blocks und Quellcode 7 zeigt die Inspektion eines `apply`-Blocks mit einem Argument vor der Sequenz.

Quellcode 6: Inspektion vom Block `(apply str ["m" "e" "t" "e" "r"])`

```
(apply str ["m" "e" "t" "e" "r"])
; => "meter"

(str "m" "e" "t" "e" "r")
```

Quellcode 7: Inspektion vom Block `(apply str "kilo" ["m" "e" "t" "e" "r"])`

```
(apply str "kilo" ["m" "e" "t" "e" "r"])
; => "kilometer"

(str "kilo" "m" "e" "t" "e" "r")
```

3.7.6 Partial

Die Higher-order-Funktion `partial` nimmt eine Funktion sowie beliebig viele Argumente entgegen. Sie gibt eine partielle Funktion zurück, die auf weitere Argumente angewandt werden kann. Die partielle Funktion ruft `apply` mit den an `partial` übergebenen Argumenten und den Argumenten, welche an die partielle Funktion übergeben wurden, auf. In der Inspektion eines `partial`-Blocks wird diese erzeugte Funktion dargestellt. Quellcode 8 zeigt die Inspektion eines `partial`-Blocks.

Quellcode 8: Inspektion vom Block `(partial + 10)`

```
(partial + 10)

(fn [& args] (apply + 10 args))
```

3.7.7 Juxt

Der Higher-order-Funktion `juxt` wird mindestens eine Funktion übergeben. `juxt` gibt eine Funktion zurück, die die Juxtaposition der übergebenen Funktionen in Form eines Vektors erzeugt. Die Inspektion eines `juxt`-Blocks zeigt diese erzeugte Funktion mit beliebig vielen Argumenten an. Quellcode 9 zeigt die Inspektion eines `juxt`-Blocks.

Quellcode 9: Inspektion vom Block `(juxt identity type)`

```
(juxt identity type)

(fn [& args] [(apply identity args)
              (apply type args)])
```

3.8 Print

Clojure und ClojureScript beinhalten die Möglichkeit, Daten und Objekte aus dem Programm heraus auszugeben. Ziel dessen ist üblicherweise die Standardausgabe, also die Konsole, aus der das Programm gestartet wurde. Es gibt in Clojure die Möglichkeit, in einen Java-StringWriter zu schreiben, um die Ausgabe in Form einer Zeichenkette an eine andere Stelle weiterzuleiten. Um eine Konsole zu emulieren, in der `print-` und `println`-Ausdrücke angezeigt werden, gibt es verschiedene Möglichkeiten.

Die ursprüngliche Idee, nach der Evaluation aller Ausdrücke ein Fenster zu öffnen, in dem die vollständige Standardausgabe angezeigt wird, wurde verworfen. Der Aufwand, einen StringBuffer beziehungsweise einen StringWriter in ClojureScript zu implementieren, wurde als zu groß eingeschätzt. Stattdessen werden `println-` und `print`-Ausdrücke nun in die JavaScript-Konsole im Browser gedruckt, indem die beiden Funktionen mit den äquivalenten Funktionen von JavaScript überschrieben wurden. Diese Lösung hat den Nachteil, dass sich

sowohl `println`, als auch `print` wie JavaScripts `console.log` verhalten, also jeder Aufruf in eine neue Zeile gedruckt wird.

3.9 Export und Import

Um Projekte in ClojureBlocks speichern und öffnen zu können, wurde eine Möglichkeit zum Import und Export implementiert. Dabei wird zunächst bei jeder Änderung des Arbeitsbereichs dieser serialisiert und im local storage des Browsers gespeichert. Außerdem kann mit Hilfe der Buttons unten rechts in der Anwendung der serialisierte Arbeitsbereich als JSON-Datei heruntergeladen werden oder eine JSON-Datei mit einem serialisierten Arbeitsbereich hochgeladen werden.

3.9.1 Serialisation

Zur Serialisation des Arbeitsbereichs wird auf die entsprechende Funktion der Blockly-API zugegriffen. Diese erhält den Arbeitsbereich und gibt ein JavaScript-Objekt zurück. Dieses Objekt wird dann mit der JavaScript-Funktion `JSON.stringify` zu einer Zeichenkette gewandelt und kann dann abgespeichert werden.

3.9.2 Localstorage

In der HTML-Spezifikation ist der local storage spezifiziert, welcher es Browseranwendungen ermöglicht, im lokalen Speicher des Browsers mit Hilfe von Schlüssel-Wert-Paaren Daten abzulegen. Damit ClojureBlocks nicht von der API für diesen Speicher abhängig ist und der Speicher austauschbar ist, um ClojureBlocks auf anderen Plattformen als im Browser zu verwenden, wurde ein Wrapper für die local storage API implementiert. Dieser Wrapper beinhaltet die Funktionen `(set-item! key value)`, welche einen Wert mit einem Schlüssel abspeichert und gegebenenfalls überschreibt und `(get-item key)`, welche einen Wert zu einem Schlüssel liest und zurückgibt.

3.9.3 Upload und Download

Der gespeicherte Arbeitsbereich soll zur Sicherung und Wiederherstellung von Nutzenden heruntergeladen und wieder hochgeladen werden können, um einen vorherigen Stand weiterzuverwenden. Da ClojureBlocks nur im Browser ausgeführt wird und kein Backend für die Anwendung existiert, ist das Hoch- und Herunterladen von Dateien nicht trivial.

Die Funktion, die das Hochladen von Dateien initiiert, bekommt nur eine Callback-Funktion übergeben. Sie erzeugt ein `<input type="file">`-Element und löst das `click`-Event darauf aus. Dadurch öffnet sich im Browser ein Dialog zur Dateiauswahl. Nachdem eine Datei

ausgewählt wurde, wird versucht, die Datei zu lesen. Falls der Versuch erfolgreich ist, wird die übergebene Callback-Funktion mit dem gelesenen Dateiinhalt aufgerufen, welche den Dateiinhalt mit `JSON.parse` in ein JavaScript-Objekt wandelt und den Workspace von Blockly mit diesem Objekt überschreibt.

Die Funktion zum Herunterladen von Dateien bekommt den Dateiinhalt, den Dateinamen und den Mimetype der Daten übergeben. Sie erzeugt dann einen Datei-Blob und fügt der Seite ein unsichtbares `<a>`-Element mit dem erzeugten Blob als Ziel hinzu. Anschließend wird auch hier das `click`-Event auf dem Element ausgelöst und die Datei so heruntergeladen. Zuletzt wird das `<a>`-Element wieder von der Seite entfernt.

3.10 Design

ClojureBlocks verwendet den von Blockly voreingestellten Renderer für die Blöcke. Es gibt die Möglichkeit, einen eigenen Renderer zu erstellen und einzubinden, um Blöcke anders darzustellen. Allerdings ist dies für den Bearbeitungszeitraum dieser Bachelorarbeit zu zeitintensiv.

In Blockly ist die Hintergrundfarbe von Blöcken als Farbton konfigurierbar. Die Helligkeit und Farbsättigung sind dabei von Blockly vorgegeben. Google empfiehlt, diese Variante der Farbeinstellung zu verwenden, um die Barrierefreiheit für Farbenblinde nicht einzuschränken. Außerdem existiert die Möglichkeit, Hintergrundfarben in Hexadezimalschreibweise anzugeben. Da mit der Auswahl des Farbtons nicht für alle zehn Blockkategorien hinreichend unterscheidbare Farben eingestellt werden können, wurden mit Hilfe des Werkzeugs „I want visually distinct colors!“[11] zehn Farben generiert, die sich stark unterscheiden. Diese Farben werden für die Blockkategorien in der Toolbox sowie als Hintergrundfarbe für die jeweiligen Blöcke verwendet.

Um die Sichtbarkeit von Blöcken und Texten in ClojureBlocks in hellen oder dunklen Umgebungen oder am Beamer zu verbessern, wurde ein Hell- und Dunkelmodus implementiert, welcher die Hintergrundfarbe des Arbeitsbereichs ändert. Die Modi können über einen Schalter unten rechts in der Anwendung umgeschaltet werden.

Blockly bietet die Möglichkeit, das Aussehen über Themen dynamisch zu verändern, sodass kein CSS mit verschiedenen Klassen für verschiedenes Aussehen geschrieben werden muss. Von dieser Möglichkeit wurde Gebrauch gemacht und das Blockly theme-dark[12] für den Dunkelmodus verwendet.

Um die beiden Dialoge für die Einstellungen sowie für die Blockinspektion an den jeweiligen Modus anzupassen, mussten dennoch CSS-Klassen geschrieben werden, welche mit ClojureScript für die jeweiligen Elemente aktiviert beziehungsweise deaktiviert werden.

[11] https://mokole.com/palette.html
[12] https://www.npmjs.com/package/@blockly/theme-dark

3.11 ClojureDocs

Die ClojureDocs-Webseite[13] stellt eine von der Clojure-Gemeinde betriebene Dokumentation der Standardbibliothek von Clojure bereit. Weiter zeigt sie für viele Funktionen Beispielaufrufe. In den gängigen Editoren und Tools für Clojure können Funktionsdefinitionen und -dokumentationen schnell erreicht werden. Um eine ähnlich leicht zu erreichende Hilfe bereitzustellen, wurde für jeden Funktionsaufrufblock das Kontextmenü „Help“ implementiert. Dazu bietet Blockly das Feld `helpUrl` in einer Blockdefinition an, welches mit der der Funktion entsprechenden URL ausgefüllt wurde. So wird bei einem Klick auf das Kontextmenü „Help“ der ClojureDocs-Eintrag zur Funktion in einem neuen Browsertab geöffnet.

[13] https://clojuredocs.org/

4 Evaluation

4.1 Große Projekte in ClojureBlocks

Um zu testen, wie sich ClojureBlocks mit größeren Projekten verhält, werden zwei Anwendungen in ClojureBlocks implementiert. Das erste Projekt ist das Marsrover-Kata[14] der Coding-Dojo-Webseite. Dabei wurde eine bereits existierende Implementierung in ClojureBlocks übertragen, um zu testen, wie sich die Größe der Blöcke verhält und wie übersichtlich und praktisch Verwendbar ClojureBlocks mit einem großen Projekt ist. Der Quellcode der existierenden Implementierung ist unter `https://codeberg.org/jhandke/marsrover.clj` zu finden.

Als zweites Projekt wird eine 4clojure-Aufgabe der Kategorie *medium* oder *hard* mittels ClojureBlocks gelöst. Dabei soll festgestellt werden, ob die Hilfsmittel, die ClojureBlocks anbietet, beim Lösen der Aufgabe nützlich sind. Hierbei fiel die Wahl auf die Aufgabe Word Chains[15] der Kategorie *hard*.

4.1.1 Marsrover

Beim Marsrover-Kata soll ein Programm entwickelt werden, welches einen Marsrover bewegt, der auf einer zweidimensionalen Oberfläche positioniert ist. Die Oberfläche besteht dabei aus freien Flächen und Hindernissen, welche auf der Oberfläche platziert sind. Das Programm bekommt die Oberfläche, die Position des Marsrovers auf dieser Oberfläche, die Himmelsrichtung, in die der Marsrover ausgerichtet ist, und eine Liste von Anweisungen übergeben. Die möglichen Anweisungen, um den Marsrover zu bewegen, sind „links drehen“, „rechts drehen“ und „vorwärts fahren“. Falls eine Anweisung den Marsrover auf ein Hindernis oder über den Rand der Oberfläche hinaus bewegen würde, soll diese Anweisung übersprungen werden.

Die Implementierung des Marsrover-Programms enthält eine Multimethode, die für jede der zwölf Kombinationen von Orientierung des Marsrovers und auszuführender Anweisung einen neuen Zustand berechnet. Davon müssen vier Funktionen eine neue Position des Marsrovers berechnen und die neue Position überprüfen. Der Code, um herauszufinden, ob sich eine Position innerhalb der Oberfläche befindet und ob sich auf der Position ein Hindernis befindet, wurde in zwei weitere Funktionen ausgelagert. Außerdem existieren Funktionen zum Generieren einer zufälligen Oberfläche sowie zum Ausgeben eines Zustands.

Der gesicherte Arbeitsbereich der Implementierung ist in den Git-Repositories (s. Abschnitt 3) im Ordner `/example` zu finden. Außerdem zeigt Abbildung 6 die Implementierung in ClojureBlocks.

[14] `https://codingdojo.org/kata/mars-rover/`
[15] `https://4clojure.oxal.org/#/problem/82`

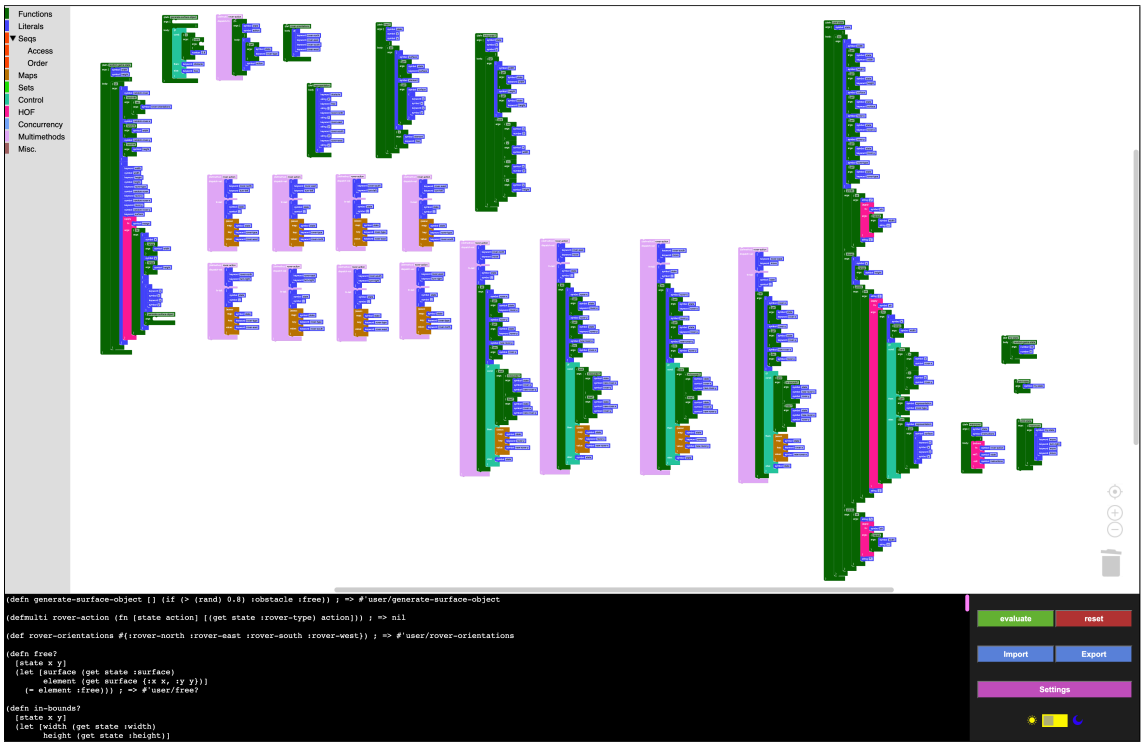


Abbildung 6: Marsrover Implementierung in ClojureBlocks

4.1.2 4clojure-Aufgabe Word Chains

Zum Lösen der Aufgabe Word Chains muss eine Funktion implementiert werden, welche eine Sequenz mit Wörtern übergeben bekommt und prüft, ob aus diesen Wörtern eine Wortkette gebildet werden kann. Zwei Wörter bilden eine Kette, wenn sie sich nur in einem Zeichen unterscheiden. Falls alle Wörter in der übergebenen Sequenz auf diese Art verkettet werden können, soll die Funktion `true` und andernfalls `false` zurückgeben.

Um herauszufinden, welche Wörter sich in nur einem Zeichen unterscheiden, muss zunächst die Levenshtein-Distanz dieser Wörter berechnet werden. Anschließend können Ketten gesucht werden, wobei jedes Wort ein Startwort einer Kette sein kann.

Der gesicherte Arbeitsbereich dieser Implementierung ist ebenfalls in den Git-Repositiories (s. Abschnitt 3) im Ordner `/example` zu finden. Abbildung 7 zeigt die Implementierung in ClojureBlocks.

4.1.3 Evaluation

Die Implementierung des Marsrover-Katas in Clojure ist 148 Zeilen lang. Das Übertragen des Codes in ClojureBlocks-Blöcke hat etwa dreißig Minuten gedauert. Dabei ist zu beachten,

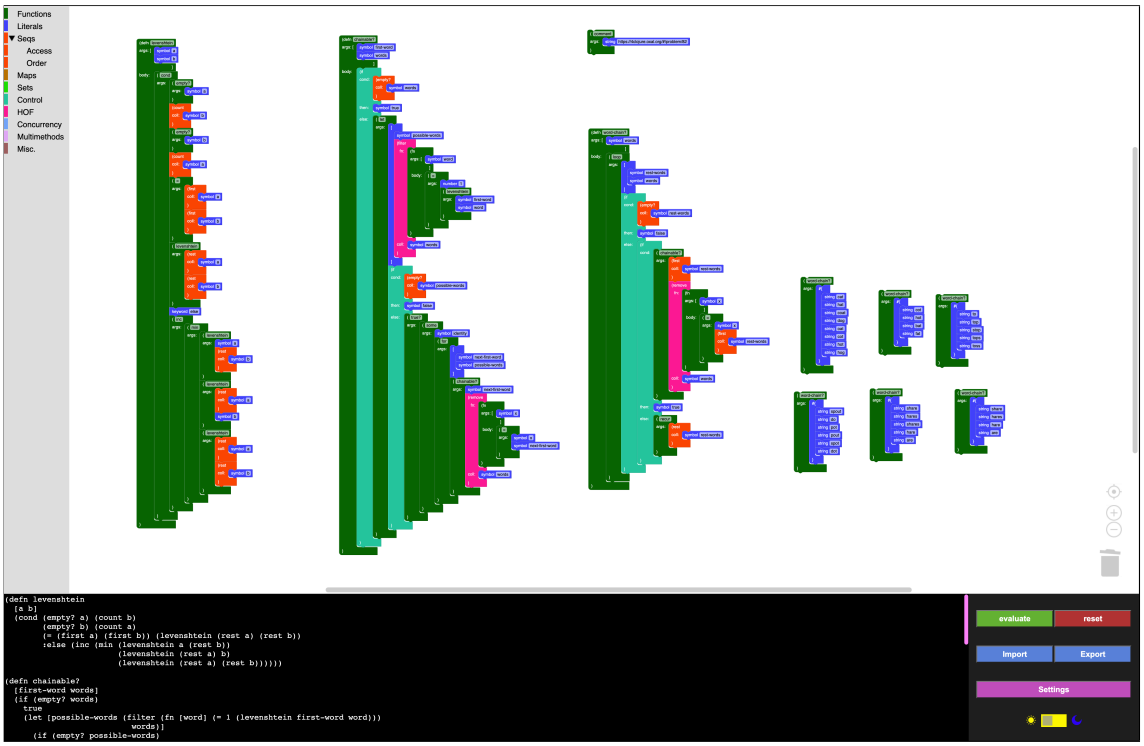


Abbildung 7: Word Chains Implementierung in ClojureBlocks

dass die Implementierung bereits vorlag und nichts neues entwickelt oder getestet werden musste. Hilfreich war dabei die Möglichkeit, Blöcke in ClojureBlocks zu duplizieren, da viele Funktionen der genannten Multimethode syntaxgleich sind. In der voreingestellten Vergrößerungsstufe von ClojureBlocks passten allerdings viele Blöcke nicht in den sichtbaren Bereich des Arbeitsbereichs. Dadurch konnte schnell die Übersicht verloren werden, was Fehler bei der korrekten Verschachtelungen der Blöcke provozierte. Außerdem haben viele Blöcke, welche oft verschachtelt werden, wie zum Beispiel Funktionsaufrufe, dieselbe Farbe, was dieses Problem ebenfalls verschärfte. Nur mit Hilfe des generierten Codes oder durch Verschieben oder Ändern der Größe des Arbeitsbereichs konnten die aufgetretenen Fehler festgestellt und behoben werden.

Dadurch, dass `print-` und `println`-Ausdrücke immer eine ganze Zeile ausdrucken, musste die `print-state`-Funktion der Marsrover-Implementierung angepasst werden, sodass erst die auszudruckende Zeile konstruiert und anschließend ausgedruckt wird. Das verkompliziert die Arbeit mit Textausgabe in die JavaScript-Konsole erheblich.

Beim Lösen der 4clojure-Aufgabe 82 „Word Chains“ war es hilfreich, zunächst einzelne Komponenten, wie zum Beispiel das Finden möglicher Wörter mit Levenshtein-Distanz von 1 zu einem Startwort, anhand eines expliziten Beispiels zu implementieren. So konnte einfach getestet werden, an welcher Stelle ein Fehler vorliegt. Wenn eine Komponente funktioniert, war es leicht, diese als Funktion zu definieren. Dazu musste nur noch ein `defn`-Block erstellt

werden, die Literale des Beispiels durch passende Symbole ersetzt werden und dann der Block in den erstellten `defn`-Block verschoben werden.

Leider konnten die verwendeten Higher-order-Funktionen `filter` und `remove` nicht inspiziert werden, da die Sequenzen, auf denen sie aufgerufen werden, nicht als Funktionsaufruf oder als Literal im entsprechenden Block standen.

Allgemein ist aufgefallen, dass Blöcke, die nur wenigen Zeilen Clojure-Code entsprechen, in ClojureBlocks teilweise unverhältnismäßig groß werden. Durch die genannten Probleme eignet sich ClojureBlocks eher weniger für umfangreichere Projekte.

4.2 Erfüllte Strategien

Hier soll ein Überblick über die in Abschnitt 1.2.2 vorgestellten Strategien der visuellen Programmierung gegeben werden, die in ClojureBlocks erfüllt beziehungsweise nicht erfüllt werden.

Konkretheit: Dadurch, dass in Clojure Funktionsaufrufe durch Listen ausgedrückt werden, lassen sich Funktionsaufrufe in ClojureBlocks sowohl durch vorgefertigte Funktionsblöcke wie für `(first xs)`, als auch durch den allgemeinen Funktionsaufrufblock und den Block für Listen mit einem Symbolblock verschachtelt ausdrücken. ClojureBlocks ist daher keine konkrete visuelle Programmiersprache, wie es von Burnett definiert ist.

Direktheit und Explizitheit: Da ClojureBlocks eine visuelle Programmierumgebung für die Sprache Clojure und keine eigenständige Programmiersprache ist, sind die Strategien Direktheit und Explizitheit nicht in dem Sinne anzuwenden. Semantik wird in Clojure immer durch Text ausgedrückt. Weiter ist das Ziel von ClojureBlocks, beim Lernen der Programmiersprache Clojure zu unterstützen. Daher ist es gerade sinnvoll, dass die Blöcke etwa der Syntax von Clojure entsprechen und Grundzüge der Sprache bekannt sind oder bei der Verwendung von ClojureBlocks erlernt werden.

unmittelbares visuelles Feedback: Da bei jeder signifikanten Änderung im Arbeitsbereich der gesamte Code generiert und angezeigt wird, erfüllt ClojureBlocks die Strategie des unmittelbaren visuellen Feedbacks. Außerdem kann die automatische Evaluation aktiviert werden, sodass die Auswirkungen einer Änderung im Code sofort sichtbar werden.

4.3 Erreichte Ziele

Das Ziel, eine visuelle Programmierumgebung für die Programmiersprache Clojure zu entwickeln, wurde erreicht. Für Funktionsaufrufe, alle Literale und Datenstrukturen sowie ausgewähle Funktionen, Macros und special forms wurden Blöcke erstellt. Auch alle anderen Funktionen, Macros und special forms können mit ClojureBlocks verwendet werden, da sie durch die vorhandenen Datenstrukturen und Literale ausgedrückt werden können.

Für alle Blöcke wird ohne Nutzerinteraktion in Echtzeit Code generiert, welcher in ClojureBlocks am unteren Rand des Fensters angezeigt wird. Durch die Architekturentscheidung, Blockly zu verwenden, kann der generierte Code oder der berechnete Wert nicht im entsprechenden Block angezeigt werden.

Die Evaluation des generierten Codes ist ebenfalls möglich. Auf Nutzerwunsch oder optional automatisch werden alle Ausdrücke im generierten Code nacheinander evaluiert und das Ergebnis wie in einer REPL hinter dem zu evaluierenden Ausdruck angezeigt. Fehlermeldungen werden auf dieselbe Art angezeigt.

Außerdem wurden für die Higher-order-Funktionen `map`, `filter`, `remove`, `reduce`, `apply`, `partial` und `juxt` eine grundlegende Inspektion implementiert. Falls die Aufrufe dieser Funktionen ohne externe Variablen passieren, zeigt ihre Inspektion genauer, welche Funktionsaufrufe ausgeführt werden oder welche Funktion zurückgegeben wird.

4.4 Ausblick

Durch die Modularisierung von Blockdefinitionen und Codegeneratorfunktionen können mit wenig Aufwand neue Blöcke zu ClojureBlocks hinzugefügt werden. Auch der Inhalt und die Sortierung der Toolbox ist frei konfigurierbar. Außerdem gibt es die Möglichkeit, einen voreingestellten Arbeitsbereich zu laden. Damit kann ClojureBlocks auf den jeweiligen Anwendungszweck angepasst werden.

Da die Konfigurationsmöglichkeiten keine Funktionsdefinitionen oder -aufrufe beinhalten, sondern nur Datenstrukturen sind, können sie in Konfigurationsdateien ausgelagert werden. Damit ist es leicht möglich, Instanzen von ClojureBlocks zu erstellen, welche auf eine bestimmte Aufgabe zugeschnitten sind. Ein Beispiel dafür wäre „Implementieren sie ein Macro, welches die logische Implikation berechnet, ohne die Junktoren `and`, `or`, `not` zu verwenden“. In diesem Fall könnte ClojureBlocks nur die Blöcke bereitstellen, die verwendet werden dürfen und der Block für einen allgemeinen Funktionsaufruf entfernt werden.

Der verwendete Evaluator SCI ist an nur einer Stelle im Code eingebunden und kann durch einen anderen Evaluator ausgetauscht werden, solange er eine Zeichenkette mit Clojure-Code entgegennimmt und eine Zeichenkette mit dem Evaluationsresultat beziehungsweise dem Fehlertext zurückgibt. Dabei kann auch, wie zunächst in Abschnitt 2.3 beschrieben, eine lokale REPL angebunden werden, um den vollen Funktionsumfang und die Geschwindigkeit von Clojure zu erhalten.

Mit Hilfe der `getParent`-Methode der Blockly-API können die übergeordneten Blöcke von Higher-order-Funktion-Blöcken ausgewertet werden und die Inspektion dieser auch mit Eingaben möglich gemacht werden, welche nicht als Literal oder Funktionsaufruf existieren.

Ob und wie ClojureBlocks im Rahmen der Lehre der funktionalen Programmierung und der Programmiersprache Clojure zum Einsatz kommt und ob es dabei nützlich ist, geht über den Umfang dieser Bachelorarbeit hinaus.

Abbildungsverzeichnis

Quellcodeverzeichnis

Literatur

[BD04] Boshernitsan, Marat ; Downes, Michael S.: Visual Programming Languages: A Survey / Computer Science Division, EECS, University of California, Berkeley. Version: Dezember 2004. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2004/6201.html. 2004 (UCB/CSD-04-1368). – Forschungsbericht

[Bur99] Burnett, Margaret M.: Visual Programming. Version: 1999. http://dx.doi.org/10.1002/047134608X.W1707. In: Webster, J. (Hrsg.): *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, Inc, 1999. – DOI 10.1002/047134608X.W1707, S. 275–283

[CDP00] Cooper, Stephen ; Dann, Wanda ; Pausch, Randy: Alice: A 3-D Tool for Introductory Programming Concepts. In: *Journal of Computing Sciences in Colleges* 15 (2000), April, Nr. 5, 107—116. https://dl.acm.org/doi/10.5555/364133.364161

[Hel19] Heller, Thomas: What shadow-cljs is and isn't. (2019), März. https://code.thheller.com/blog/shadow-cljs/2019/03/01/what-shadow-cljs-is-and-isnt.html

[Hic20] Hickey, Rich: A history of Clojure. In: *Proceedings of the ACM on programming languages* 4 (2020), Nr. HOPL, S. 1–46. http://dx.doi.org/10.1145/3386321. – DOI 10.1145/3386321

[KA11] Kaučič, B. ; Asič, T.: Improving introductory programming with Scratch? (2011), 1095–1100. https://ieeexplore.ieee.org/document/5967218

[McG11] McGranaghan, Mark: ClojureScript: Functional Programming for JavaScript Platforms. In: *IEEE Internet Computing* 15 (2011), Nr. 6, S. 97–102. http://dx.doi.org/10.1109/MIC.2011.148. – DOI 10.1109/MIC.2011.148

[MRR+10] Maloney, John ; Resnick, Mitchel ; Rusk, Natalie ; Silverman, Brian ; Eastmond, Evelyn: The Scratch Programming Language and Environment. Version: November 2010. http://dx.doi.org/10.1145/1868358.1868363. In: *ACM Transactions on Computing Education* Bd. 10. 2010. – DOI 10.1145/1868358.1868363

[PDV13] Pokress, Shaileen ; Dominguez Veiga, José J.: MIT App Inventor: Enabling Personal Mobile Computing. (2013), Oktober. http://dx.doi.org/10.48550/arXiv.1310.2830. – DOI 10.48550/arXiv.1310.2830

[Poo15] Poole, Matthew: Design of a blocks-based environment for introductory programming in Python. (2015), S. 31–34. http://dx.doi.org/10.1109/BLOCKS.2015.7368996. – DOI 10.1109/BLOCKS.2015.7368996

[SHBLS21] Siegfried, Robert M. ; Herbert-Berger, Katherine G. ; Leune, Kees ; Siegfried, Jason P.: Trends Of Commonly Used Programming Languages in CS1 And CS2 Learning. Version: 2021. http://dx.doi.org/10.1109/ICCSE51940.2021.9569444. In: *2021 16th International Conference on Computer Science & Education (ICCSE)*. 2021. – DOI 10.1109/ICCSE51940.2021.9569444, S. 407–412

[Tan90] Tanimoto, Steven L.: VIVA: A visual language for image processing. In: *Journal of Visual Languages & Computing* 1 (1990), Nr. 2, 127–139. http://dx.doi.org/https://doi.org/10.1016/S1045-926X(05)80012-6. – DOI https://doi.org/10.1016/S1045–926X(05)80012–6