INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

Generierung didaktischer Fehlermeldungen für Minisprachen

Galina Keil

Bachelorarbeit

Aufgabensteller	Prof. Dr. François Bry
Betreuer	Prof. Dr. François Bry, Niels Heller
Abgabe am	21.11.2019

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 21.11.2019 Galina Keil

Abstract

The web-based Knoala editor, a component of the Backstage 2 learning platform, offers students the opportunity to write and test their programs in various programming languages.

The main objective of this work is to extend the functionality of the Knoala editor by generating error messages for the mini languages LOOP, WHILE and GOTO while a user is writing a program. In this report the conception and the implementation of these functionalities and their integration into the existing software are represented.

First, the concept of mini languages in general is introduced and the syntax of LOOP, WHILE and GOTO programs is precisely explained. Furthermore, the theoretical basics of the implemented parser, the PEG grammars, are presented.

Parts of this work are also the description of the parser generator PEG.js, which was used for the implementation of the syntax check and error messages, a detailed introduction of the grammars for LOOP, WHILE and GOTO written in PEG.js notation, as well as the representation of the syntax highlighting in Ace Editor, which was created for the mini language GOTO.

Finally, a user study is described which evaluated whether the new functionalities facilitate working with the editor and help students to solve programming tasks and better understand the syntax and the programs' structures.

Zusammenfassung

Der webbasierte Knoala-Editor, ein Bestandteil der Lernplattform Backstage 2, bietet Studenten die Möglichkeit, ihre Programme in verschiedenen Programmiersprachen zu schreiben und zu testen.

Das Hauptziel dieser Arbeit besteht darin, die Funktionalität des Knoala-Editors zu erweitern, indem für die Minisprachen LOOP, WHILE und GOTO Fehlermeldungen generiert werden, während ein Nutzer ein Programm schreibt. In diesem Bericht werden die Konzeption und die Implementierung dieser Funktionalitäten und ihre Einbindung in die bestehende Software dargestellt.

Zunächst wird der Begriff der Minisprachen im Allgemeinen eingeführt und auf die Syntax von LOOP-, WHILE- und GOTO-Programmen genauer eingegangen. Außerdem werden die theoretischen Grundlagen zum implementierten Parser, die PEG-Grammatiken, dargelegt.

Teil dieser Arbeit sind auch die Beschreibung des Parser-Generators PEG.js, der für die Implementierung der Syntaxüberprüfung und der Fehlermeldungen verwendet wurde, eine genaue Vorstellung der Grammatiken für LOOP, WHILE und GOTO geschrieben in PEG.js-Notation, sowie die Darstellung der Syntaxhervorhebung in Ace Editor, die für die Minisprache GOTO erstellt wurde.

Schließlich wird eine Nutzerstudie beschrieben, in der evaluiert wurde, ob die neuen Funktionalitäten das Arbeiten mit dem Editor erleichtern sowie den Studenten helfen, Programmieraufgaben zu lösen und besser die Syntax und den Aufbau des Programms zu verstehen.

Danksagung

Ich möchte mich bei Prof. Dr. François Bry für die Möglichkeit bedanken, diese Bachelorarbeit an der Lehr- und Forschungseinheit für Programmier- und Modellierungssprachen zu schreiben. Zudem danke ich ihm für das Feedback während des Entstehens dieser Arbeit. Ich möchte mich ganz besonders bei meinem Betreuer Niels Heller bedanken, dass er mir dieses Thema vorgeschlagen hat und mich während der Entwicklung stets unterstützt hat. Ich bin ihm auch für sein Feedback beim Schreiben dieser Arbeit dankbar. Außerdem danke ich Elisabeth Lempa für das Durchführen des Tutoriums für meine Nutzerstudie.

Inhaltsverzeichnis

KAPITEL 1

Einführung

Im Informatikstudium muss jeder Student Grundlagenwissen aus den Bereichen der theoretischen Informatik und der Berechenbarkeitstheorie erwerben. Um den Begriff der Berechenbarkeit formal zu untersuchen, werden verschiedene Formalismen eingeführt, z.B. Turing-Maschinen, LOOP-Programme, WHILE-Programme und GOTO-Programme. Man versteht die Syntax und die Semantik einer Programmiersprache am besten, wenn man selbst in dieser Sprache Programme schreibt und Aufgaben löst [4]. Dabei kann ein guter Editor mit Syntaxhervorhebung, Syntaxüberprüfung und einem eingebauten Compiler, der die Möglichkeit eröffnet, das geschriebene Programm sofort zu testen, sehr nützlich sein und den Lernprozess unterstützen.

Am Lehrstuhl für Programmier- und Modellierungssprachen der Ludwig-Maximilians-Universität München wird die Lernplattform Backstage 2 entwickelt. Diese Plattform wird in der Vorlesung genutzt und unterstützt die Interaktion zwischen dem Dozenten und den Studenten. Backstage 2 vereint die Konzepte eines Backchannels mit einem Audience Response System sowie projektbasiertem Lernen. Es besteht aus zwei Hauptkomponenten: Backstage-Kursen und Backstage-Projekten.

Backstage-Kurse beinhalten Folien und andere Materialien zur Vorlesung. Studenten können über die Plattform Fragen stellen und erhalten Feedback. Zusätzlich zu den Folien sind Quizze integriert, die Studenten beantworten Fragen zum Vorlesungsstoff, der vor kurzem erklärt wurde. Anhand ihrer Antworten kann der Dozent Schlussfolgerungen bezüglich der Frage ziehen, wie gut der Stoff verstanden wurde und das Tempo der Vorlesung anpassen.

Eine andere Komponente von Backstage 2 sind Projekte, in denen Unterrichtsmaterialien organisiert werden. In einem Projekt ist es möglich, dass eine Gruppe länger

zusammenarbeitet sowie Dokumente und Quellcodes für gemeinsame Softwareentwicklungsprojekte verwaltet. [2]

Ein Bestandteil dieser Plattform ist auch der Editor Knoala, der unter anderem auch die Minisprachen LOOP, WHILE und GOTO unterstützt. Mit diesem Editor können Studenten ihre Programme schreiben und auch sofort ausführen. Der Compiler liefert das berechnete Ergebnis. Alternativ erscheint, wenn das Programm fehlerhaft ist, eine Fehlermeldung. Allerdings sind die Fehlermeldungen der Compiler für Minisprachen LOOP, WHILE und GOTO oft schwer zu verstehen. Es wird keine Zeile mit dem Fehler angegeben, und oft ist nur mühsam nachzuvollziehen, worin der Fehler besteht. [5] In der Abbildung 1.1 ist ein LOOP-Programm mit einem syntaktischen Fehler dargestellt.

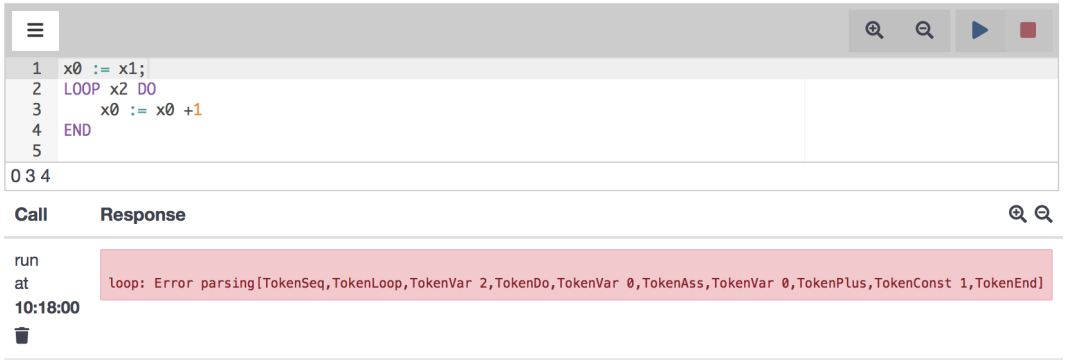

Abbildung 1.1: Programm mit einem Syntaxfehler und einer Fehlermeldung des Compilers

In der ersten Zeile steht die Wertzuweisung "x0 := x1;", die in der Minisprache LOOP nicht erlaubt ist. Wenn man versucht, das Programm auszuführen, erscheint die Fehlermeldung des Compilers, wie in der Abbildung 1.1 im roten Viereck zu sehen ist.

Ziel dieser Arbeit ist es, in den Editor folgende Funktionen zu integrieren:

- die Meldungen von Syntaxfehlern verbessern
- Fehlermeldungen generieren, noch während der Nutzer das Programm schreibt (wie in vielen Entwicklungsumgebungen üblich)

Zu diesem Zweck wurde ein Parser für jede der Minisprachen LOOP, WHILE und GOTO geschrieben und in den Editor implementiert. Nach jeder Benutzereingabe wird der Parser aufgerufen, die Syntax vom Programm analysiert und die Zeile mit einem syntaktischen Fehler markiert sowie die Fehlermeldungen mit erwarteter Eingabe angezeigt. Anschließend wurde eine Nutzerstudie mit Studenten durchgeführt, um zu evaluieren, ob die Fehlermeldungen ihnen helfen, Programme zu schreiben, Fehler im Quellcode zu finden sowie die Syntax der Programmiersprache und den Aufbau von Programmen besser zu verstehen.

Diese Arbeit ist wie folgt gegliedert: Zuerst werden die theoretischen Grundlagen erörtert, welche Voraussetzung für die Implementierung des Parsers sind. In

diesem Zusammenhang werden die Syntax und die Semantik der Minisprachen LOOP, WHILE und GOTO beschrieben. Es werden PEG-Grammatiken vorgestellt, welche als Grundlage der Grammatik im Parser dienen.

Das dritte Kapitel beschäftigt sich mit der Implementierung des Parsers für die Minisprachen LOOP, WHILE und GOTO sowie mit dem Integrieren des Parsers in den bestehenden Editor. Außerdem wird vorgestellt, in welcher Weise die Syntaxhervorhebung für die Minisprache GOTO in den Editor eingebaut wurde.

Im letzten Teil der Arbeit wird die Nutzerstudie beschrieben und ihre Ergebnisse werden dargestellt.

KAPITEL 2

Einordnung in den Forschungsstand

Dieses Kapitel dient der Einleitung in die theoretischen Grundlagen von Minisprachen im Allgemeinen und der Programmiersprachen LOOP, WHILE, GOTO im Speziellen, für die in den Editor eine Syntaxüberprüfung implementiert wird. Es wird der Formalismus von PEG-Grammatiken vorgestellt, auf dem die Grammatik für die Syntaxüberprüfung im Editor basiert. Im ersten Unterkapitel wird auf die Begriffe domänenspezifische Sprachen und Minisprachen eingegangen. Zum Thema Minisprachen werden außerdem einige Beispiele vorgestellt. Im zweiten Unterkapitel werden die Semantik und die Syntax der Minisprachen LOOP, WHILE und GOTO ausführlich beschrieben. Am Ende dieses Kapitels werden PEG-Grammatiken definiert und der Unterschied zu kontextfreien Grammatiken hervorgehoben.

2.1 Minisprachen

In der Informatik gibt es neben universellen Programmiersprachen, die für viele Anwendungsfälle einsetzbar sind, auch domänenspezifische Sprachen.

Eine domänenspezifische Sprache ist eine Programmiersprache, die sich auf eine bestimmte Problemdomäne konzentriert und in der Regel auf diese beschränkt ist. Sie stellt eine auf eine Anwendungsdomäne zugeschnittene Notation bereit und basiert auf den relevanten Konzepten und Merkmalen dieser Domäne. Das hat den Vorteil, dass Programme auf einer Abstraktionsebene der Problemdomänen geschrieben werden können, so können Fachleute für diese Domäne Programme verstehen, validieren, modifizieren und oft auch entwickeln. Zu den potenziellen

Vorteilen von domänenspezifischen Sprachen gehören auch geringere Wartungskosten sowie eine höhere Portabilität und Testbarkeit [13, 12].

Domänenspezifische Sprachen sind zum Beispiel SQL für relationale Datenbanken, HTML für die Strukturierung elektronischer Dokumente, UML für die Modellierung, Spezifikation, Konstruktion und Dokumentation von Software und Gherkin für die Beschreibung von Software Testfällen.
In der Literatur werden domänenspezifische Sprachen auch Minisprachen oder kleine Sprachen genannt.

Minisprachen werden speziell auch die Programmiersprachen genannt, die dazu dienen, den Programmieranfängern einen vereinfachten Einstieg in die Programmierung zu ermöglichen. Sie vermitteln algorithmisches Denken, heben grundlegende Konzepte der Programmierung hervor und fördern deren Erlernen. Universelle Programmiersprachen haben einen großen Sprachumfang und erfordern viel Lernzeit. Schon die Grundlagen der Sprache mit den Hauptprinzipien der Programmierung bilden eine große Menge an Material. Eine universelle Sprache ruft viele nebensächliche Begriffe hervor, die die Feinheiten und die Umsetzung der jeweiligen Sprache widerspiegeln. Im Gegensatz dazu sind die pädagogischen Minisprachen in dem Sinne klein, dass sie über wenige Syntaxelemente verfügen, eine einfache Semantik haben sowie Grundprinzipien und grundlegende Strukturen der Programmierung betonen. [6]

Einige bekannte Minisprachen sind:

- Logo. Die Entwicklung der Minisprache Logo wurde stark von der Turtle-Grafik von Logo beeinflusst. Eine oder mehrere Schildkröten lassen sich bewegen, indem der Benutzer die Befehle angibt. Die Programmiersprache, die dafür verwendet wird, heißt Logo. [6]

- Karel the Robot. Karel enthält alle wichtigen Pascal-ähnlichen Kontrollstrukturen und lehrt die Grundbegriffe der sequentiellen Ausführung, prozeduralen Abstraktion, bedingten Ausführung und Wiederholung. Karel enthält keine Variablen, Typen oder Ausdrücke. Der Roboter Karel führt Befehle aus und bewegt sich im Spielfeld, das aus horizontalen und vertikalen Linien besteht und Wänden hat, die den direkten Weg blockieren. [6]

- Kara. Kara ist eine Mini-Umgebung basierend auf der Idee der endlichen Automaten. Kara, der Marienkäfer lebt in einer einfachen grafischen Welt auf dem Bildschirm. Er kann programmiert werden, in seiner Welt verschiedene Aufgaben zu erledigen. Die Programme werden graphisch mit der Maus als endliche Automaten erstellt. [10]

Im nächsten Teil werden die Minisprachen LOOP, WHILE und GOTO vorgestellt. Diese Sprachen haben einen didaktischen Zweck, aber im Gegensatz zu den oben genannten Minisprachen sind sie nicht dafür konzipiert, Anfängern das Programmieren auf spielerische Art und Weise beizubringen. Sie ermöglichen es, den Begriff der Berechenbarkeit in der theoretischen Informatik näher zu erläutern und

zu untersuchen. Das Programmieren mit den Minisprachen LOOP, WHILE und GOTO soll helfen, eine Intuition für den Berechenbarkeitsbegriff zu schaffen, z.B.: Welche Probleme kann ich lösen, wenn ich eine LOOP-Schleife, WHILE-Schleife oder einen Bedingten Sprung zur Verfügung habe?

2.2 LOOP, WHILE, GOTO

LOOP, WHILE und GOTO sind Minisprachen, die im Zusammenhang mit der Berechenbarkeitstheorie eine Rolle in der theoretischen Informatik spielen. Unter anderem wird mithilfe dieser Minisprachen der Berechenbarkeitsbegriff definiert. Als Grundlage dieser Arbeit wird die Definition der Syntax von Prof. Dr. Uwe Schöning verwendet [11].

2.2.1 LOOP-Programme

LOOP ist eine Minisprache mit einer sehr eingeschränkten Syntax.

LOOP-Programme bestehen aus folgenden syntaktischen Komponenten:

- Variablen: x0, x1, x2 ...
- Konstanten: 0 1 2 ...
- Trennsymbole: ; :=
- Operationszeichen: + -
- Schlüsselwörter: LOOP DO END

Sie berechnen Funktionen über natürliche Zahlen. Bei Beginn der Programmausführung werden die Eingabewerte n1, ... , nk an Variablen x1, ... , xk gebunden. Alle anderen vorkommenden Variablen haben den Anfangswert 0. Nach Initialisierung wird das Programm abgearbeitet.

Jede Wertzuweisung der Form xi := xj + c bzw. xi := xj - c ist ein LOOP-Programm. Die Wertzuweisung xi := xj + c bedeutet, dass der Variable xi als neuer Wert die Summe des Wertes xj und der Zahl c zugewiesen wird. Im Falle von xi := xj - c wird die modifizierte Subtraktion verwendet, also falls c > xj, so wird das Resultat auf 0 gesetzt.

Sind P1 und P2 bereits LOOP-Programme, so ist auch

P1; P2

ein LOOP-Programm. Es wird interpretiert, indem zuerst P1 und dann P2 interpretiert werden.

Falls P ein LOOP-Programm ist und xi eine Variable, dann ist auch

LOOP xi DO P END

ein LOOP-Programm. Das Programm dieser Form wird so interpretiert, dass das Programm P so oft ausgeführt wird, wie der Wert der Variable xi zu Beginn angibt. Zuweisungen an xi im Innern von P haben keinen Einfluss auf die Anzahl der Schleifendurchläufe. Ausgabewert der Berechnung ist der Wert von x0 nach Beendigung des Programms. [11]

2.2.2 WHILE-Programme

Das WHILE-Programm ist eine Erweiterung der LOOP-Programme. Die Syntax von WHILE-Programmen enthält alle Konzepte, wie sie auch bei LOOP-Programmen vorkommen. Jedes LOOP-Programm ist zugleich ein WHILE-Programm. Zusätzlich beinhalten WHILE-Programme eine WHILE-Schleife.

Falls P ein WHILE-Programm ist und xi eine Variable, dann ist auch

WHILE xi DO P END

ein WHILE-Programm.

Die Semantik dieses neuen Konstrukts ist so definiert, dass das Programm P solange wiederholt auszuführen ist, wie der aktuelle Wert von xi ungleich Null ist. Der Wert von xi kann durch P geändert werden. [11]

2.2.3 GOTO-Programme

GOTO-Programme sind aus folgenden syntaktischen Komponenten aufgebaut:

- Variablen: x0, x1, x2 . . .
- Konstanten: 0 1 2 ...
- Trennsymbole: ; : :=
- Operatoren: + - =
- Schlüsselwörter: IF THEN GOTO HALT

GOTO-Programme bestehen aus Sequenzen von Anweisungen Ai, getrennt durch jeweils ein Semikolon. Jede Anweisung beginnt mit einer eindeutigen Marke Mi, gefolgt von einem Doppelpunkt:

M1: A1; M2: A2; . . . . ; Mk: Ak

GOTO-Programme haben eine endliche Anzahl von Variablen xi und Konstanten c. Als mögliche Anweisungen Ai sind zugelassen:

- Wertzuweisungen: xi := xj + c bzw. xi := xj - c

- unbedingter Sprung: GOTO Mi
- bedingter Sprung: IF xi = c THEN GOTO Mj
- Stopanweisung: HALT

Wertzuweisungen verfügen über dieselbe Semantik wie in LOOP- und WHILE-Programmen. GOTO legt die nächste auszuführende Programmanweisung fest. Eine Stopanweisung bedeutet, dass die Abarbeitung des Programms abgebrochen wird. Der Ausgabewert ist der Wert von x0 nach der Ausführung des Programms. [11]

2.3 PEG-Grammatiken

In diesem Abschnitt werden PEG-Grammatiken (Engl. "parsing expression grammars") vorgestellt. Mit diesen Grammatiken wird die Syntax der Minisprachen LOOP, WHILE und GOTO beschrieben.

Parsing-Ausdrucksgrammatiken (PEGs) sind formale analytische Grammatiken zur Beschreibung der Syntax von Programmiersprachen. Sie beschreiben eine Sprache mit Regeln zum Erkennen von Zeichenfolgen innerhalb der Sprache. Die Syntax von PEG-Grammatiken ist der Syntax von kontextfreien Grammatiken ähnlich. Der Unterschied ist, dass in PEG-Grammatiken der Auswahloperator die erste Übereinstimmung auswählt, während in kontextfreien Grammatiken die Auswahl nicht eindeutig ist. Eine PEG kann als formale Beschreibung eines Top-Down-Parsers angesehen werden. Syntaktisch sind PEG-Grammatiken aussagekräftiger als die Sprachklasse LL(k), mit welcher gewöhnlich Top-Down-Parsers assoziiert werden, und kann alle deterministische LR(k)-Sprachen ausdrücken. [3]

LL(k) sind Grammatiken, in denen jeder Ableitungsschritt eindeutig durch die nächsten k Symbole der Eingabe bestimmt ist.

LR(k) sind Grammatiken, in denen jeder Reduktionsschritt eindeutig durch die nächsten k Symbole der Eingabe bestimmt ist. LR(k) Grammatiken bilden die Grundlage für Bottom-Up-Parsers. [7]

Alle PEGs können in linearer Zeit analysiert werden.
Formal werden PEG-Grammatiken folgendermaßen definiert:

Eine Parsing-Ausdrucksgrammatik (PEG) ist ein 4-Tupel

$$G = (V_N; V_T; R; e_S).$$

- V_N ist eine endliche Menge Nichtterminalsymbolen,
- V_T ist eine endliche Menge von Terminalsymbolen,
- R ist eine endliche Menge von Regeln,

- e_S ist ein Start-Ausdruck.

Und $V_N \cap V_T = \emptyset$. Jede Regel $r \in R$ ist ein Paar (A; e), und hat die Form $A \leftarrow e$ [1] , wobei $A \in V_N$ und e ein Parsing-Ausdruck ist. Für jedes nichtterminale A gibt es genau ein e: $A \leftarrow e \in R$. R ist also eine Funktion von Nichtterminalen zu Ausdrücken.

Parsing-Ausdrücke sind wie folgt definiert: Wenn e, e_1 und e_2 Parsing-Ausdrücke sind, dann auch

- ε, die leere Zeichenfolge
- a, beliebiges Terminalsymbol, $a \in V_T$.
- A, beliebiges Nichtterminalsymbol, $A \in V_N$.
- $e1e2$ eine Sequenz.
- $e1/e2$, priorisierte Wahl.
- e^*, null oder mehr Wiederholungen.
- $e+$, eine oder mehr Wiederholungen.
- $e?$, null oder eine Wiederholung.
- $\&e$, und-Prädikat
- $!e$, kein Prädikat.

Der Sequenzausdruck 'e1 e2' sucht nach einer Übereinstimmung von e1, unmittelbar gefolgt von einer Übereinstimmung von e2, und kehrt zum Startpunkt zurück, wenn eines der Muster fehlschlägt. Der Ausdruck für die priorisierte Wahl 'e1 / e2' versucht zuerst Muster e1, und nur, wenn es fehlschlägt, wird vom Ausgangspunkt aus das Muster e2 erprobt.

Die Operatoren e^* , $e+$, $e?$ konsumieren 0 oder mehr, 1 oder mehr oder 0 oder 1 aufeinanderfolgende Wiederholungen ihres Unterausdrucks e. Im Gegensatz zu kontextfreien Grammatiken und regulären Ausdrücken verhalten sich diese Operatoren jedoch immer "gierig" , konsumieren so viel Eingaben wie möglich und führen niemals ein Backtracking durch.

Die Operatoren & und ! bezeichnen syntaktische Prädikate. Der Ausdruck $\&e$ ruft den Unterausdruck e auf und kehrt dann bedingungslos zum Ausgangspunkt zurück. Es bleibt nur die Information erhalten, ob die Übereinstimmung erfolgreich hergestellt wurde oder der Versuch fehlgeschlagen ist. Umgekehrt schlägt der Ausdruck $!e$ fehl, wenn e erfolgreich ist, ist aber erfolgreich, wenn e fehlschlägt. [3]

Der Unterschied zwischen kontextfreien Grammatiken und PEGs liegt darin, dass PEGs Mehrdeutigkeiten bei der Definition der Grammatiksprache vermeiden. Die

[1] Schreibweise nach Bryan Ford "Parsing Expression Grammars: A recognition-based syntactic foundation"

Varianten werden der Reihe nach ausprobiert. Wenn die erste Variante Eingabezeichen erkennt, wird keine andere Alternative ausprobiert, andernfalls geht der Parser zurück und prüft die nächste Variante auf die Übereinstimmung.[9]

So sind die Regeln '$A \rightarrow ab|a'$ und '$A \rightarrow a|ab'$ in einer kontextfreien Grammatik äquivalent, in einer PEG-Grammatik sind aber die Regeln '$A \leftarrow ab/a'$ und '$A \leftarrow a/ab'$ unterschiedlich. Die zweite Alternative der PEG-Regel wird niemals ausgewählt, da die erste Wahl immer dann getroffen wird, wenn die zu erkennende Eingabezeichenfolge mit 'a' beginnt.

PEG bietet eine leistungsfähige, formal strenge, und effizient implementierbare Grundlage für die Beschreibung der Syntax von maschinenorientierten Sprachen, die so gestaltet sind, dass sie eindeutig sind. [3]

KAPITEL 3

Implementierung

Dieses Kapitel beschreibt, wie die Funktion der Syntaxüberprüfung implementiert und in den Knoala Editor integriert wurde. Zuerst werden der Parser-Generator PEG.js und seine Grammatik vorgestellt. Im Anschluss werden die in PEG.js- Notation geschriebenen Grammatiken für die Minisprachen LOOP, WHILE und GOTO erläutert. Im Abschnitt 3.2 wird der Ace Editor, auf dem Knoala basiert, kurz vorgestellt und erklärt, wie die Syntaxhervorhebung für die Minisprache GOTO mit dem Ace Editor erstellt wurde. Im letzten Abschnitt des Kapitels geht es um das Integrieren des erstellten Parsers in den Knoala Editor.

3.1 Parsen von LOOP, WHILE und GOTO mit PEG.js

3.1.1 Der Parser-Generator PEG.js

PEG.js [8] ist ein freier Parser-Generator für JavaScript, entwickelt von David Majda, einem tschechischen Software Engineer. PEG.js basiert auf den PEG- Grammatiken, die in Kapitel zwei vorgestellt wurden. Ein Parser kann per Kommandozeile oder mithilfe einer JavaScript API generiert werden. PEG.js generiert einen Parser aus einer Grammatik, die die erwartete Eingabe beschreibt, und kann spezifizieren, was der Parser zurückgibt. Der generierte Parser selbst ist ein JavaScript-Objekt mit einer einfachen API.

Die Grammatik von PEG.js ähnelt syntaktisch JavaScript, sie ist nicht zeilenorientiert und ignoriert Leerzeichen zwischen Tokens. Es ist möglich, Kommentare im JavaScript-Stil zu verwenden. Auf der obersten Ebene besteht die Grammatik aus Regeln (*rules*). Jede Regel hat einen Namen, der zu ihrer Identifizierung dient, und

einen Parsing-Ausdruck (*parsing expression*), der das Muster definiert, nach dem die Übereinstimmung mit dem Eingabetext geprüft wird. Der Parsing-Ausdruck kann einen JavaScript-Code enthalten, der festlegt, was passiert, wenn das Muster erfolgreich geparst wird, was man benutzen kann, um z.B. arithmetische Ausdrücke auszuwerten. Eine Regel kann auch einen von Menschen lesbaren Namen enthalten, der in Fehlermeldungen verwendet wird. Das Parsen beginnt mit der ersten Regel, die auch als Startregel bezeichnet wird.

Ein Regelname muss ein JavaScript-Bezeichner sein. Nach dem Regelnamen folgen ein Gleichheitszeichen ("= ") und ein Parsing-Ausdruck. Wenn die Regel einen von Menschen lesbaren Namen aufweist, wird er als ein String zwischen dem Regelnamen und dem Gleichheitszeichen geschrieben. Regeln müssen nur durch Leerzeichen getrennt werden, aber ein Semikolon (";") nach dem Parsing-Ausdruck ist auch zulässig.

Vor der ersten Regel kann ein Initialisierer (*initializer*) eingefügt werden. Dieser beinhaltet einen JavaScript-Code in geschweiften Klammern, der vor dem Parsen ausgeführt wird. Auf alle im Initialisierer definierten Variablen und Funktionen kann in Regeln und Parsing-Ausdrücken zugegriffen werden. So können die Hilfsfunktionen und Variablen definiert werden, die den Parsing-Prozess verbessern sollen.

Die Parsing-Ausdrücke der Regeln werden verwendet, um den eingegebenen Text mit der Grammatik zu vergleichen und Übereinstimmungen zu finden. Es gibt verschiedene Arten von Ausdrücken. Sie stimmen mit den Arten von Parsing-Ausdrücken, die im Abschnitt 2.3, in dem PEG-Grammatikem im Allgemeinen vorgestellt wurden, überein. Ausdrücke können auch Verweise auf andere Regeln enthalten.

Wenn ein Ausdruck beim Ausführen des generierten Parsers mit einem Teil des Texts übereinstimmt, wird ein Ergebnis generiert, das ein JavaScript-Wert ist, zum Beispiel:

- Ein Ausdruck, der mit einer Zeichenfolge übereinstimmt, erzeugt einen JavaScript-String, der den übereinstimmenden Text enthält.
- Ein Ausdruck, der dem wiederholten Auftreten eines Teilausdrucks entspricht, erzeugt ein JavaScript-Array mit allen Übereinstimmungen.

Wenn die Regeln aus den Ausdrücken bestehen, die Verweise auf andere Regeln enthalten, werden die geparsten Ergebnisse bis zur Startregel durch alle enthaltene Unterregeln weitergegeben. Ist das Parsen erfolgreich verlaufen, gibt der Parser das Ergebnis der Startregel zurück.

Eine besondere Art des Parser-Ausdrucks ist eine Parser-Aktion (*parser action*): ein JavaScript-Code in geschweiften Klammern, der Ergebnisse der Ausdrücke, die schon vorher geparst wurden, verwendet und einen neuen JavaScript-Wert zurückgibt. [8]

In weiteren Unterkapiteln werden die Grammatiken für Minisprachen LOOP, WHI-

LE und GOTO vorgestellt, die mit PEG.js geschrieben wurden.

3.1.2 Grammatik für LOOP

Der Aussschnitt aus der Grammatik für die Minisprache LOOP ist in Listing 3.1 dargestellt. Er beinhaltet die wesentlichen Teile der LOOP-Grammatik. Die komplette Grammatik berücksichtigt noch mehr, z.B. ist es möglich, einzeilige Kommentare in jedem Teil von einem Programm zu schreiben und beliebige Anzahlen von Leerzeichen zwischen den einzelnen Komponenten zu verwenden. Weiter unten wird der Aufbau der einzelnen Regeln betrachtet.

Listing 3.1: Ausschnitt aus der Grammatik für die Minisprache LOOP in der PEG.js-Notation

```
1  Program =
2       (((valueAssignment / LoopProgram) semicolon)+
3         (valueAssignment / LoopProgram))
4       / valueAssignment
5       / LoopProgram
6
7  LoopProgram =  LOOP v:variable DO p:Program END {
8               return {
9                   type : "Loop Program",
10                loopKeyword: "LOOP",
11                variable: v,
12                doKeyword: "DO",
13                innerProgram : p,
14                endKeyword: "END"
15              };
16  }
17 valueAssignment = v1:variable equal v2:variable o:operator c:constant {
18              return{
19                  type: "value assignment",
20                  variable_1 : v1,
21                  delimiter: ":=",
22                  variable_2: v2,
23                  operator: o,
24                  constant: c
25              }
26  }
27 variable "variable like xi" =  _ "x" i: Integer{
28              return "x"+ i;
29  }
30
31 Integer "integer" = digits:[0-9]+ {
32              return parseInt(digits.join(""), 10);
33   }
34 operator "+ or -" = _ sign:("+"/"-"){
35    if(sign[0]=="+"){
36            return "+"}
37    else return "-";
38 }
39 comment "comment" = (_ "#" (!LineTerminator .)*)*_{
40              return "comment"
```

```
41 }
```

Die Startregel hat den Namen `Program`, nach dem Gleichheitszeichen stehen alle mögliche Ausdrücke, mit denen ein gültiges LOOP-Programm gebildet werden kann. In Klammern sind die Ausdrücke aufgeführt, die nicht getrennt ausgewertet werden dürfen. Mögliche Alternativen sind durch einen Schrägstrich geteilt. Sie werden in der Reihenfolge der Deklaration ausprobiert. Wenn der Parser Übereinstimmung gefunden hat, werden keine weitere Alternativen ausprobiert. In der LOOP-Grammatik ist der erste Ausdruck:
`(((valueAssignment / LoopProgram) semicolon )+`
`(valueAssignment / LoopProgram)).`
Dieser stellt eine Sequenz aus zwei Unterausdrücken vor, beide vom Typ Regel:

- `((valueAssignment / LoopProgram) semicolon )+`
  Operator + bedeutet, dass der Ausdruck mindestens einmal vorkommen soll. Es kann eine Wertzuweisung `(valueAssignment)` gefolgt von einem Semikolon sein, oder eine LOOP-Schleife `(LoopProgram)` gefolgt von einem Semikolon.

- `(valueAssignment / LoopProgram)`
  Dieser Ausdruck soll unmittelbar nach dem oben geschrieben Ausdruck vorkommen, sonst schlägt das komplette Muster fehl. Es kann eine Wertzuweisung oder eine LOOP-Schleife sein.

Diese Regel beschreibt also eine Konstruktion "P1; P2", die im Abschnitt 2.2.1 vorgestellt wurde. Außerdem kann das Programm nur aus einer Wertzuweisung (`valueAssignment`) oder einer LOOP-Schleife (`LoopProgramm`) bestehen. Es sind auch nur Kommentare oder Leerzeichen erlaubt, damit keine Fehlermeldung erscheint, wenn der Nutzer noch keine Eingaben gemacht hat oder das Programm mit einem Kommentar beginnt.

Die nächste Regel `LoopProgramm` beschreibt die Konstruktion

LOOP xi DO P END

Es wird innerhalb `LoopProgram` rekursiv die Startregel `Program` aufgerufen.

Die Regel `LoopProgramm` enthält markierte Ausdrücke und ein JavaScript-Code in geschweiften Klammern. Die Markierungen sind mit einem Doppelpunkt von dem Ausdruck getrennt. In einer Markierung wird das geparste Ergebnis des Ausdrucks, das der Markierung folgt, gespeichert. Auf so gespeicherte Werte kann in JavaScript-Code zugegriffen werden. Dieser Code wird ausgeführt, wenn der komplette Ausdruck erfolgreich geparst wurde. Es wird ein Syntaxbaum des geparsten Programms ausgegeben.

Die Regel `valueAssignment` beschreibt die Wertzuweisungen der Form xi := xj + c bzw. xi := xj - c.

Eine Variable wird in der Regel `variable` beschrieben. Sie beginnt immer mit dem Zeichen "x", danach folgt eine natürliche Zahl, für die es eine eigene Regel `Integer`

gibt. In den Anführungszeichen zwischen dem Regelnamen und dem Gleichheitszeichen steht ein zusätzlicher von Menschen lesbarer Name, dieser wird in der Fehlermeldung verwendet.

Die Regel `comment` für Kommentare hat folgende Aufbau:

- _ Es können Leerzeichen vor und nach dem Kommentar vorkommen.
- # Mit diesem Zeichen beginnt ein Kommentar.
- `! LineTerminator` Es soll kein Zeilenumbruch vorkommen, es sind also nur einzeilige Kommentare erlaubt.
- `.` Nach dem Zeichen # kann ein beliebiges Zeichen vorkommen.
- `*` Diese Zeichen sind beliebig oft (auch 0-mal) wiederholbar.

Die restlichen Regeln, die in der Listing nicht vertreten sind, beschreiben die elementaren syntaktischen Komponenten, wie sie im Abschnitt 2.2.1 erläutert wurden: Konstanten, Trennsymbole (; :=), Schlüsselwörter (LOOP DO END) und Leerzeichen.

Wenn das Programm erfolgreich geparst wurde, wird ein Syntaxbaum erstellt. Ein Beispiel für ein einfaches LOOP-Programm und den Syntaxbaum dazu ist in Abbildung 3.1 zu sehen.

```
LOOP x1 DO
    x0 := x0 + 1
END
```

```
{
  type: "Loop Program",
  loopKeyword: "LOOP",
  variable: "x1",
  doKeyword: "DO",
  innerProgram: {
      type: "value assignment",
      variable_1: "x0",
      delimiter: ":=",
      variable_2: "x0",
      operator: "+",
      constant: 1
  },
  endKeyword: "END"
}
```

Abbildung 3.1: LOOP-Programm mit seinem Syntaxbaum

3.1.3 Grammatik für WHILE

Die Grammatik für die Minisprache WHILE ähnelt stark der LOOP-Grammatik. Sie weist alle Elemente der LOOP-Grammtik auf und eine zusätzliche Regel für die WHILE-Schleife `(WhileProgram)` wie in Listing 3.2 dargestellt ist.

Listing 3.2: Ausschnitt aus der Grammatik für die Minisprache WHILE in der PEG.js-Notation

```
Program =
  (((valueAssignment / LoopProgram / WhileProgram) semicolon)+
      (valueAssignment / LoopProgram / WhileProgram))
    / valueAssignment
    / LoopProgram
    / WhileProgram

 WhileProgram = WHILE v:variable DO p:Program END {
             return {
                 type : "WHILE Program",
                 whileKeyword: "WHILE",
                 variable: v,
                 doKeyword: "DO",
                 innerProgram : p,
                 endKeyword: "END"
      }
    }
```

3.1.4 Grammatik für GOTO

In Listing 3.3 ist ein Ausschnitt aus der Grammatik für die Minisprache GOTO dargestellt. Die oben bezüglich der Grammatik LOOP erläuterten Regeln wurden weggelassen. Weiter unten werden spezifische Regel aus der Grammatik GOTO erklärt.

Listing 3.3: Ausschnitt aus der Grammatik für die Minisprache GOTO in der PEG.js-Notation

```
Program = ((mark colon GotoProgram semicolon)+
          mark colon GotoProgram)
          / mark colon GotoProgram

GotoProgram = valueAssignment
            /condition
            / jumpCommand
            / termination

condition = IF v:variable  equal constant THEN GOTO m:mark {
           return {
               type: "condition",
               ifKeyword: "IF",
               variable: v,
               equalityOperator: "=",
               thenKeyword: "THEN",
               gotoKeyword: "GOTO",
               mark: m
      }
  }
jumpCommand = GOTO comment m:mark {
           return{
               type: "jump command",
```

```
24                  gotoKeyword: "GOTO",
25                  mark: m   }
26    }
27  termination   = HALT {
28              return {
29                  stopKeyword: "HALT"}
30    }
31  mark   = "mark like Mi" = _ "M" i: Integer {
32                return "M" +i;
33    }
```

Die erste Regel, die Startregel, besagt, dass ein Programm aus einer Sequenz von GOTO-Programmen, getrennt durch ein Semikolon oder einem einzelnen GOTO-Programm bestehen kann. Jedes GOTO-Programm soll mit einer eindeutigen Marke gefolgt von einem Doppelpunkt beginnen.

Ein GOTO-Programm selbst, wie anhand der zweiten Regel ersichtlich ist, kann eine Wertzuweisung (`valueAssignment`), ein bedingter Sprung (`condition`), ein unbedingter Sprung (`jumpCommand`) oder eine Stopanweisung (`termination`) sein.

Die Regel `valueAssignment` hat die gleiche Bedeutung wie in LOOP-und WHILE-Programmen. Sie beschreibt die Wertzuweisungen der Form xi := xj + c bzw. xi := xj - c.

Die Regel `condition` impliziert einen bedingten Sprung der Form IF xi = c THEN GOTO Mj, in dessen Kontext xi eine Variable, c eine Konstante und Mj eine eindeutige Marke ist.

Die nächste Regel `jumpCommand` beschreibt den unbedingten Sprung der Form GOTO Mj. Die Marke ist in der Regel `mark` beschrieben, hier ist allerdings etwas abgekürzte Form der Regel dargestellt. Als eine Marke ist nur ein großgeschriebenes M mit einer natürlichen Zahl erlaubt.

Die Stopanweisung ist in der Regel `termination` beschrieben.

Analog zu den Grammatiken für LOOP und WHILE verfügt die Grammatik GOTO über Regeln zur Beschreibung aller Schlüsselwörter, Trennsymbole, Operatoren, Kommentare und Leerzeichen.

3.2 Syntaxhervorhebung für GOTO

Für die Minisprache GOTO wurde eine Syntaxhervorhebung implementiert. Im nächsten Abschnitt wird es um den Ace Editor gehen, der die Grundlage des Knoala Editors ist. Im zweiten Abschnitt wird erklärt, wie die Syntaxhervorhebung für GOTO im Ace Editor erstellt wurde.

3.2.1 Ace Editor

Ace ist ein in JavaScript programmierter Texteditor, der auf das Bearbeiten von Programm-Code spezialisiert ist. Der Ace Editor kann in jede Webseite oder jede JavaScript-Anwendung eingebettet werden und verfügt über eine umfangreiche API. Für verschiedene Programmiersprachen stehen sogenannte Modes zur Verfügung. Ein Modus definiert die Syntaxhevorhebungen für viele Programmiersprachen. Des Weiteren bietet der Editor verschieden Möglichkeiten der farblichen Anpassung mit vordefinierten Themes. [1] Im nächsten Abschnitt wird der Modus für die Minisprache GOTO erstellt, der die Syntaxhervorhebung für die Minisprache GOTO implementiert.

3.2.2 Syntaxhervorhebung in Ace Editor

Jede Sprache im Ace Editor benötigt einen eigenen Modus. Um die Syntaxhervorhebung für GOTO zu definieren, wird ein `GotoMode` erstellt. Um den Modus für die GOTO-Sprache zu erstellen, wurde ein schon im Ace Editor existierender Modus `TextMode` mit Regeln für die Syntaxhervorhebung `TextHighlightRules` verwendet und erweitert. Die neuen Regeln für die Syntaxhervorhebung `GotoHighlightRules` erben sämtliche ihrer Eigenschaften von `TextHighlightRules`. Und `GotoMode` erbt alles von `TextMode`.

Der Prozess der Syntaxhervorhebung kann im Ace Editor als eine Zustandsmaschine vorgestellt werden. Reguläre Ausdrücke definieren die Token für den aktuellen Zustand sowie die Übergänge in einen anderen Zustand.

Die Token-Zustandsmaschine arbeitet mit den in `$rules` definierten Elementen. Sie beginnt stets im Startzustand `"start"` und überprüft die Liste nach einem passenden regulären Ausdruck. Wird ein Ausdruck gefunden, so wird der resultierende Text in ein `<span class =" ace_ <token>">` -Tag eingeschlossen, wobei `<token>` als Token-Eigenschaft definiert ist. Reguläre Ausdrücke können eine RegExp- oder eine String-Definition sein. In den Regeln für die GOTO - Syntaxhervorhebung sind alle Zustände Startzustände.

Für die Hervorhebung der Schlüsselwörter von GOTO wird Token `keywordMapper` genutzt. In dieser Variable werden die Schlüsselwörter der Minisprache GOTO gespeichert. Anschließend werden die neu definierte Regeln `GotoHighlightRules` als neue Regeln für die Syntaxhevorhebung in den Mode gesetzt. [1]

Es ist möglich, in einem Ace-Modus auch eine eigene Syntaxüberprüfung zu definieren und Syntaxfehler zu erkennen. Dafür muss zusätzlich ein `worker` erstellt werden, der die Grammatik der Programmiersprache beschreibt. Diese Funktionalitäten sind aber sehr schlecht dokumentiert. Deshalb wurde entschieden, für das Parsen der Programmen das PEG.js zu verwenden und den Parser direkt in den Knoala Editor zu integrieren.

3.3 Integrieren des Parsers in den Editor

Für jede der Minisprache LOOP, WHILE und GOTO wurde eine JavaScript-Datei mit dem Parser erstellt: loopParser.js, whileParser.js und gotoParser.js. Die Grammatik für jede Sprache wird als ein String der Methode PEG.js `peg.generate(grammar)` übergeben und ein Parser für die jeweilige Sprache erstellt. Der generierte Parser wird in der Variable `customParser` gespeichert. Der Knoala Editor hat eine Funktion, die ausgeführt wird, wenn der Benutzer die Sprache ändert. Diese Funktion wurde erweitert und prüft nun, ob es für die neue Sprache einen `customParser` gibt. Ist ein solcher vorhanden, wird noch geprüft, ob die Fehlermeldungen für die betreffende Sprache aktiviert sind. Es ist möglich, die Fehlermeldungen ein- bzw. abzuschalten, wenn die Instanz von dem Knoala Editor erstellt wird.

Wenn die Fehlermeldungen aktiv sind, wird der Parser bei jeder Nutzereingabe ausgeführt. Wenn das von Nutzer eingegebene Programm erfolgreich geparst werden konnte, wird der Syntaxbaum des Programms erstellt, wie im Abschnitt 3.1.2 am Beispiel von LOOP-Programm gezeigt wurde. Der Syntaxbaum wird aber für die Nutzer nicht ausgegeben.

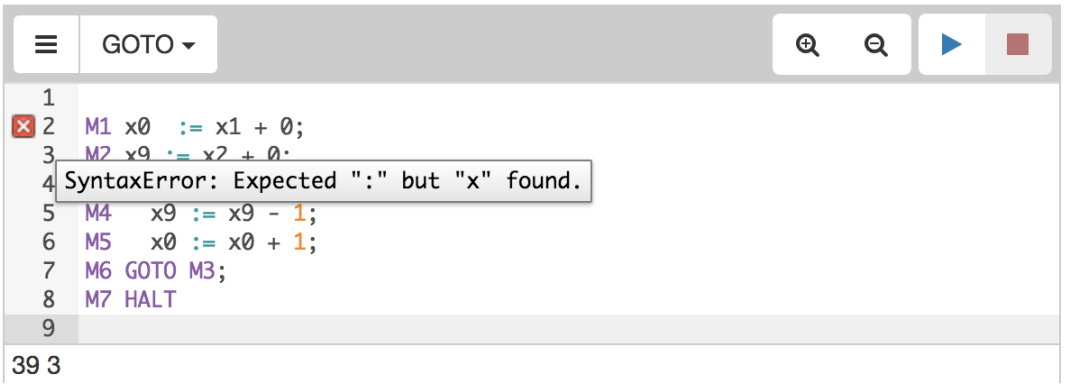

Abbildung 3.2: Programm mit einem Syntaxfehler und einer Fehlermeldung

Wenn die Eingabe ungültig ist, wird eine Ausnahme (exception) geworfen. Eine Exception enthält die Stelle (location), an der der Parser abgebrochen hat, die erwartete Eingabe (excepted), die gefundene Eingabe (found) und die Fehlermeldung (message) mit der Information zum Fehler. Eine Exception wird abgefangen und an die Methode des Ace Editors `setAnnotations` übergeben. Diese beinhaltet die Zeile, die Spalte, der Text und der Typ der Annotation. Aus der in Exception enthaltenen Information zur Position werden die Zeile und die Spalte abgeleitet. Der Typ der Fehlermeldung wird auf `"error"` gesetzt, dadurch wird an der linken Seite im Editor in der entsprechender Zeile ein rotes Kreuz angezeigt. Wenn man mit dem Cursor über das rote Kreuz fährt, erscheint der Text der Fehlermeldung (Abbildung 3.2).

Der Parser wird auch nach der Auswahl der Sprache initial einmal mit dem aktuellen Inhalt des Editors aufgerufen. Das ist nötig, damit der schon enthaltene Text

geparst und eventuell die Fehlermeldung angezeigt wird und nicht erst, wenn der Nutzer die erste eigene Eingabe tätigt.

KAPITEL 4

Nutzerstudie

Es wurde eine Nutzerstudie durchgeführt. Sie soll eine Antwort auf die Frage geben, ob die generierten Fehlermeldungen helfen, Fehler im Programm schneller zu finden, Programmieraufgaben zu lösen und die Syntax sowie den Aufbau des Programms zu verstehen. Im ersten Unterkapitel werden die Methoden der Nutzerstudie beschrieben. Im zweiten Unterkapitel werden die Ergebnisse vorgestellt und evaluiert.

4.1 Methoden

Zum Zweck der Bewertung der neuen Funktionalitäten des Editors wurde ein A/B-Test durchgeführt. Die Studenten haben zufällig zwei unterschiedliche Versionen derselben Software erhalten. Die eine Gruppe bekam einen Editor ohne integrierte Fehlermeldungen, die andere Gruppe einen mit Fehlermeldungen, die von dem Parser, der in Kapitel 3 beschrieben wurde, für die Minispachen LOOP, WHILE und GOTO generiert werden.

Verlauf.

Die Nutzerstudie wurde an einem Tag durchgeführt und nahm eine Stunde in Anspruch. Sie hatte die Form eines Tutoriums. Zuerst wurden anhand der Folien, die im Anhang A.1 zu finden sind, die Syntax und der Aufbau von LOOP-Programmen vorgestellt. Dann wurden zwei Beispiele ausführlich erklärt. Danach hatten Teilnehmer 20 Minuten Zeit, drei Aufgaben mit dem Knoala Editor zu den LOOP-Programmen zu bearbeiten.

Es gab eine Korrekturaufgabe und zwei Programmieraufgaben. Bei der Korrekturaufgabe war ein Programm schon vorgegeben, es enthielt aber mehrere syntaktische Fehler. Bei den Programmieraufgaben sollten Teilnehmer selbständig ein Programm schreiben oder ein vorgegebenes Programm ändern. Weiter unten wird auf jede Aufgabe noch ausführlicher eingegangen. Nach Ablauf der zwanzigminütigen Bearbeitungsphase sollten die Teilnehmer nicht mehr an den Aufgaben arbeiten, auch in dem Falle, dass sie diese noch nicht beendet haben.

In einem weiteren Schritt wurden die Syntax und der Aufbau von WHILE - Programmen erklärt und ein Beispiel-Programm vorgestellt. Danach mussten die Teilnehmer noch zwei Aufgaben zu den WHILE-Programmen machen. Dafür hatten sie 15 Minuten Zeit. Während der Bearbeitung aller Aufgaben durften die Teilnehmer die Folien benutzen.

Anschließend füllten sie einen Fragebogen aus. Erfasst wurden das Geschlecht, der Studiengang, das Semester sowie Vorkenntnisse zu den LOOP-und WHILE-Programmen und Erfahrungen zum Benutzen des Editors. Außerdem waren zwei Aufgaben zu den LOOP-Programmen eingebaut. Beim Bearbeiten von diesen Aufgsben durften sie keine Hilfsmittel verwenden.

Beim Bearbeiten der Aufgaben nutzten zwei Gruppen von Teilnehmern unterschiedliche Versionen des Knoala Editors. Eine Gruppe hatte den Editor mit der integrierten Syntaxüberprüfung und mit den Fehlermeldungen, die am linken Rand des Textfeldes vom Editor angezeigt wurden. Diese Gruppe wird weiter als "Gruppe mit Fehlermeldungen" bezeichnet. Die andere Gruppe benutzte die Version des Editors, in welche noch die Fehlermeldungen nicht implementiert waren. Diese Gruppe wird im Folgenden "Gruppe ohne Fehlermeldungen" genannt.

Jeder Teilnehmer meldete sich für das Benutzen des Editors mit einer Kennung an. Seine Aktivitäten im Editor, wie Öffnen, Schließen, Editieren und Kompilieren, wurden in eine Log-Datei geschrieben. Die Log-Datei und der Fragebogen wurden ausgewertet.

Aufgaben.

Es waren drei Aufgaben zu den LOOP-Programmen und zwei Aufgaben zu den WHILE-Programmen enthalten.

Die erste Aufgabe sowohl zu den LOOP-Programmen als auch zu den WHILE-Programmen war eine Korrekturaufgabe und bestand darin, in einem vorgegeben fehlerhaften Programm alle Syntaxfehler zu korrigieren. Nach der Korrektur sollte das Programm lauffähig sein, also von dem Compiler ohne Fehlermeldungen ausgeführt werden. Der berechnete Ergebnis spielte dabei keine Rolle.

In der zweiten Aufgaben sollten die Teilnehmer selbständig ein LOOP- bzw. WHILE-Programm schreiben, das x0 := x1 + x2 berechnet. Es sollte syntaktisch richtig sein und berechnen, was gefordert wurde.

Zu den LOOP-Programmen gab es noch eine dritte Aufgabe: Das vorgegebene und korrekte LOOP-Programm

LOOP x1 DO x0 := x0 + 1 END

sollte so modifiziert werden, dass es x0 := x1 * 2 berechnet.

Alle Aufgaben sind im Anhang A.1 in den Folien, die für das Tutorium verwendet wurden, vorgestellt.

4.2 Ergebnisse

An der Nutzerstudie nahmen 36 Bachelorstudenten teil. Sie studierten im 1. bis 10. Semester, wobei über die Hälfte aller Teilnehmer das 2. Semester absolvierten. Es waren 21 weibliche und 15 männliche Studierende. 20 von ihnen studierten Informatik, elf Bioinformatik, zwei Medieninformatik und die einzelnen Mathematik und Computerluinguistik mit dem Nebenfach Informatik. Alle Teilnehmer wurden zufällig in zwei Gruppen aufgeteilt. Die erste Gruppe nutzte den Editor mit den integrierten Fehlermeldungen, es waren 16 Personen. Die zweite Gruppe verwendete den Editor ohne Fehlermeldungen, es waren 20 Personen.

Auswertung der Aufgaben

Es wurde ausgewertet, ob die Aufgaben richtig gelöst wurden. Der Anteil der richtig gelösten Aufgaben war in der Gruppe mit Fehlermeldungen wesentlich höher als in der Gruppe ohne Fehlermeldungen. Besonders ausgeprägt ist der Unterschied in den Aufgaben, in denen ein vorgegebenes fehlerhaftes Programm korrigiert werden musste (Abbildung 4.1). Während in der Gruppe mit Fehlermeldungen die meisten Teilnehmer (13 von 16 Personen) die Aufgabe 1 zu den LOOP-Programmen richtig lösten, waren es in der Gruppe ohne Fehlermeldungen nur fünf Personen von 20 insgesamt. Die Ergebnisse der Aufgabe 1 zu den WHILE-Programmen sind ähnlich gelagert: in der Gruppe mit Fehlermeldungen waren es deutlich mehr als die Hälfte der Teilnehmer (11 von 16 Personen), die die Aufgabe richtig lösten, vier von 16 Personen, die in der vorgegeben Zeit die Aufgabe nicht richtig lösen konnten und eine Person, die diese Aufgabe nicht angefangen hat. In der Gruppe ohne Fehlermeldungen hat genau die Hälfte der Teilnehmer die Aufgabe 1 zu den WHILE-Programmen richtig gelöst. In den Diagrammen unten sind alle Ergebnisse zwischen 0 und 1 normiert, weil die Anzahl der Teilnehmer in zwei Gruppen nicht gleich war.

In der zweiten Aufgabe sowohl zu den LOOP-Programmen als auch zu den WHILE - Programmen mussten die Teilnehmer selbständig ein Programm schreiben, das x0 := x1 + x2 berechnet. Beide Gruppen erbrachten bessere Ergebnisse im LOOP-Programm als im WHILE-Programm (Abbildung 4.2). Von den Studenten, die den Editor mit Fehlermeldungen nutzten, haben die meisten (13 von 16) Aufgabe 2 zu den LOOP-Programmen richtig gelöst. In der Gruppe ohne Fehlermeldungen hat knapp über die Hälfte der Studenten (12 von 20) die Aufgabe korrekt gelöst. Dieselbe Aufgabe für die WHILE-Programme haben wesentlich weniger Studenten richtig gelöst, in der Gruppe mit Fehlermeldungen waren es aber trotzdem etwas über die Hälfte der Teilnehmer (9 von 16), wobei zwei Studenten überhaupt keine

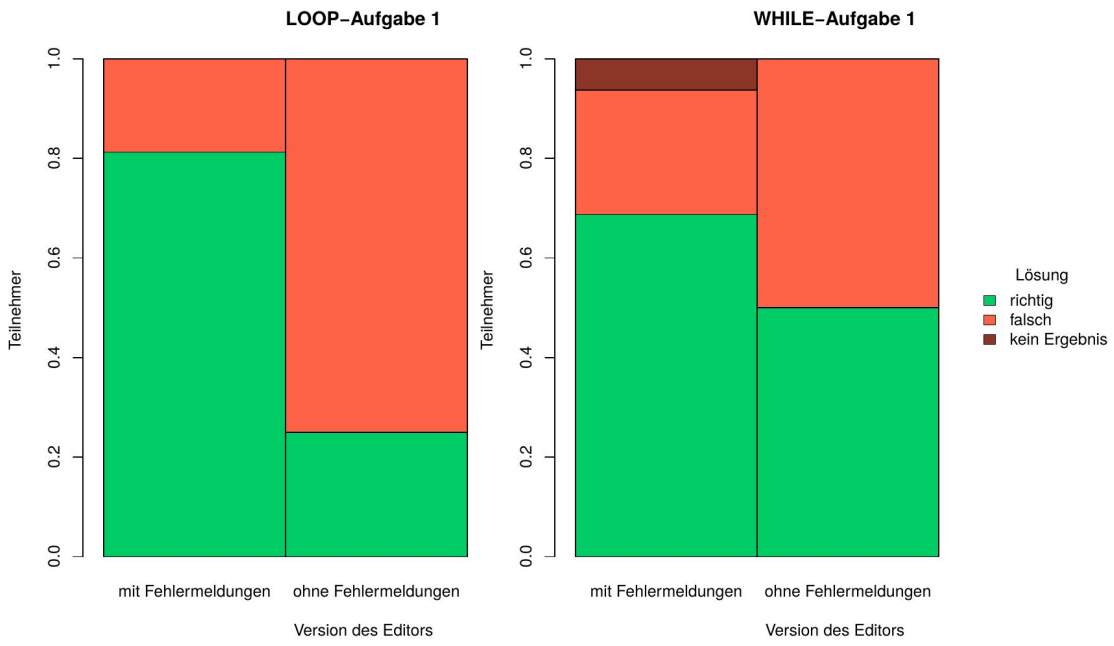

Abbildung 4.1: Anzahl von richtigen und falschen Lösungen der Aufgaben 1 zu LOOP-und-WHILE-Programmen in beiden Gruppen

Eingabe im Editor zu dieser Aufgabe vorgenommen haben. In der Gruppe ohne Fehlermeldungen haben alle Studenten Eingaben im Editor gemacht, aber nur 6 von 20 hatten am Ende ein funktionierendes und richtiges WHILE-Programm.

Um zu überprüfen, wie wahrscheinlich es ist, dass die oben beschriebenen Ergebnisse nicht zufällig entstanden sind, wurde die statistische Signifikanz mit Hilfe des Chi^2-Tests berechnet. Für die Berechnung wurde die Anzahl der richtigen Lösungen der Aufgaben in der Gruppe mit Fehlermeldungen im Vergleich zu der Gruppe ohne Fehlermeldungen verwendet. Das Signifikanzniveau wurde auf 5 % festgesetzt.

Die Ergebnisse der Aufgabe 1 zu LOOP-Programmen sind statistisch signifikant, die Wahrscheinlich, dass sie nicht zufällig sind, beträgt 99%. Die Ergebnisse der anderen Aufgaben sind statistisch nicht signifikant, da die festgelegte Grenze von 95 % nicht erreicht wurde. Jedoch beträgt die Wahrscheinlich, dass sie nicht zufällig sind, ungefähr 75 % für Aufgabe 1 zu WHILE-Programmen, 83 % für Aufgabe 2 zu LOOP-Programmen und 89 % für Aufgabe 2 zu WHILE-Programmen.

Diese Ergebnisse zeigen, dass die Fehlermeldungen den Editornutzern helfen, sowohl Fehler im Programm zu finden und zu korrigieren als auch syntaktisch richtig ein eigenes LOOP- oder WHILE-Programm zu schreiben.

Auswertung des Fragebogens

Am Ende der Studie füllten die Teilnehmer einen Fragebogen aus. Eine Kopie des

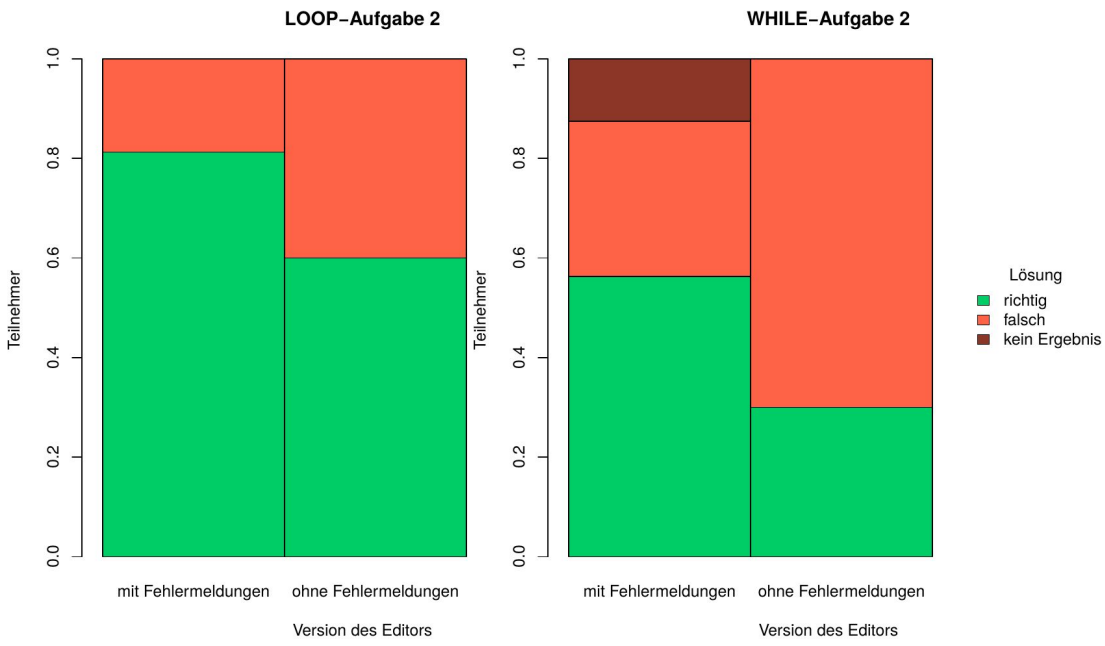

Abbildung 4.2: Anzahl von richtigen und falschen Lösungen der Aufgaben 2 zu LOOP-und-WHILE-Programmen in beiden Gruppen

Fragebogens befindet sich im Anhang A.2.

In den Fragebogen wurden noch zwei Aufgaben integriert: zwei kleine LOOP-Programme mit jeweils einem Syntaxfehler. Die Teilnehmer mussten die Zeile mit dem Fehler angeben und diesen kurz beschreiben. Im Gegensatz zu den oben dargelegten Aufgaben durften sie die Hilfsmittel und auch den Editor nicht benutzen.

Bezüglich der ersten Aufgabe waren in beiden Gruppen mehr richtige Antworten vorhanden als bei der zweiten Aufgabe. Der Anteil der richtigen Antworten war aber in der Gruppe, die den Editor mit den integrierten Fehlermeldungen benutzte, höher. Die Ergebnisse sind in der Abbildung 4.3 dargestellt.

Anhand dieser Ergebnisse lässt sich belegen, dass die Teilnehmer, die den Editor mit integrierten Fehlermeldungen benutzt haben, den Aufbau und die Syntax von LOOP-Programmen besser verstanden, sodass sie auch ohne Einsatz des Editors syntaktische Fehler in einem LOOP-Programm entdecken konnten.

Die Teilnehmer haben ihren persönlich Eindruck von Benutzen des Editors wiedergegeben, indem sie ausgewählt haben, ob sie den Aussagen aus dem Fragebogen zustimmen, eher zustimmen, eher nicht zustimmen oder nicht zustimmen. In der Abbildung 4.4 sind Ergebnisse der Abfrage zu den drei wesentlichen Aussagen vorgestellt. In beiden Gruppen ist eine Tendenz zur positiven Beurteilung des Editors zu verzeichnen. Aber in der Gruppe mit Fehlermeldungen gab es mehr Teilnehmer, die den Aussagen zugestimmt haben, als in der Gruppe ohne Fehlermeldungen. Ein grüner Farbton bedeutet, dass die Teilnehmer der Aussage zustimmen

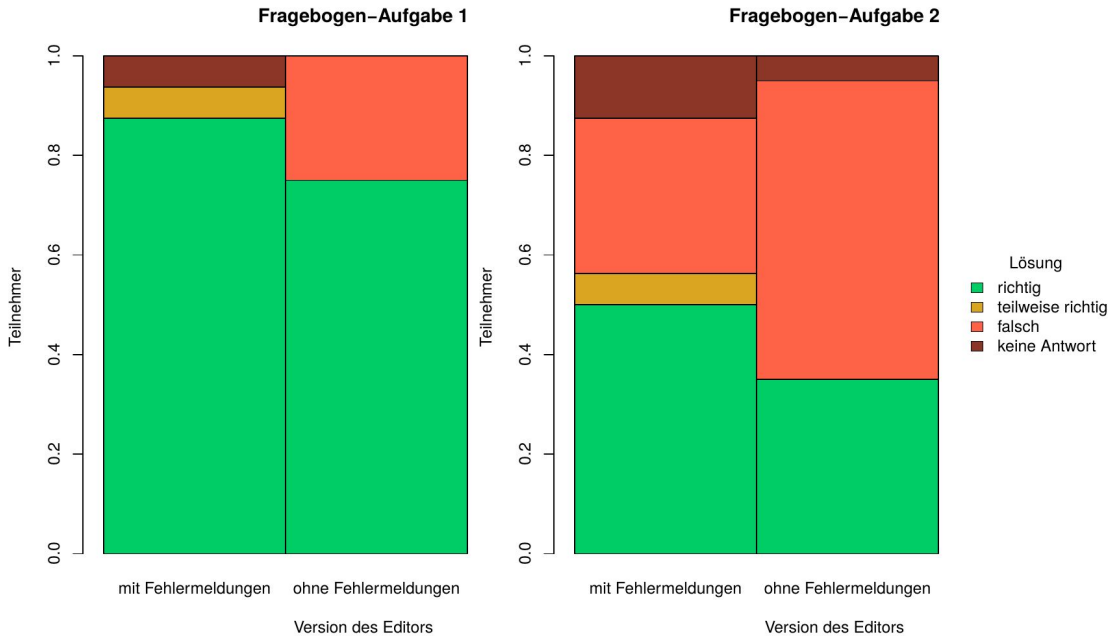

Abbildung 4.3: Anzahl von richtigen und falschen Lösungen der Aufgaben aus dem Fragebogen

und eher zustimmen, ein roter Farbton, dass sie der Aussage eher nicht zustimmen oder nicht zustimmen.

Die Studenten hatten die Möglichkeit, am Ende des Fragebogens einen Kommentar zum Benutzen des Editors zu schreiben und ihren Eindruck festzuhalten. Die meisten haben keinen Kommentar geschrieben. Einige Teilnehmer aus der Gruppe mit Fehlermeldungen haben vermerkt, dass die Fehlermeldungen hilfreich gewesen seien: "Das rote Kreuz am Rand hat geholfen Fehler zu finden.", "gute Fehlermeldungen".

Einige Studenten aus der zweiten Gruppe haben geschrieben, dass sie sich bessere Fehlermeldungen wünschen: "Der Editor ist wenig hilfreich, da er keine ausführliche Fehler wirft.", "Fehlermeldungen könnten ggf. bessere Aussagen über die Fehler selbst treffen", „Es wäre wesentlich einfacher gewesen, wenn angegeben worden wäre, in welcher Zeile die Fehler waren".

Diese Ergebnisse deuten darauf hin, dass sich die Einstellung der Nutzer zum Editor durch die integrierte Fehlermeldungen verbessert hat.

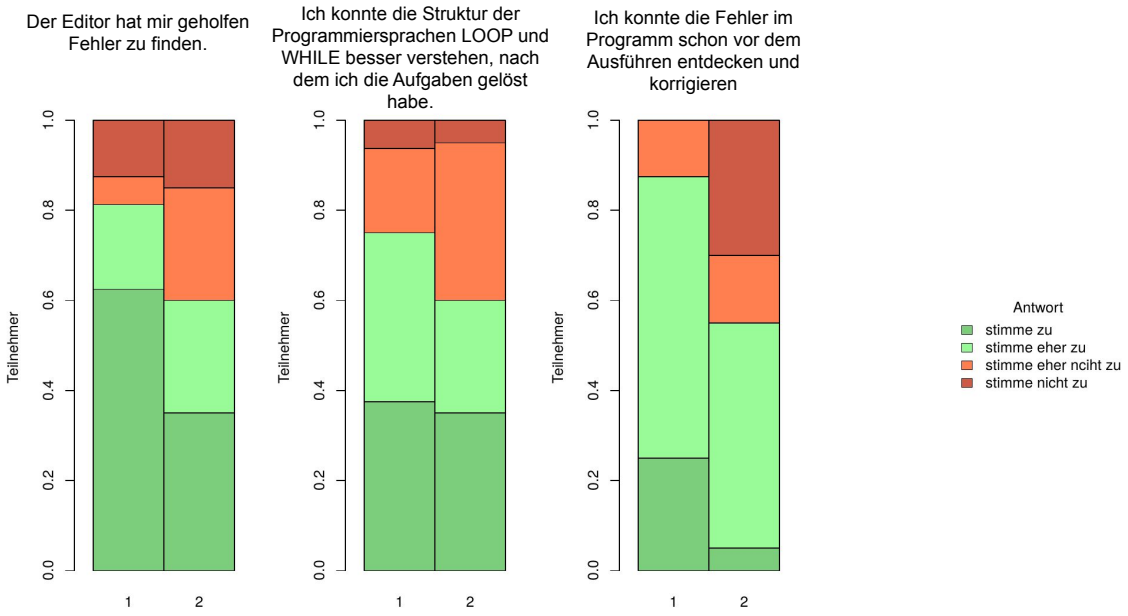

Abbildung 4.4: Feedback der Teilnehmer zum Benutzen des Editors: 1 - Editor mit Fehlermeldungen, 2 - Editor ohne Fehlermeldungen

KAPITEL 5

Fazit und Ausblick

Im Rahmen dieser Arbeit wurden Fehlermeldungen für die Minisprachen LOOP, WHILE und GOTO in den Knoala Editor eingebaut. Zuerst wurde der Begriff der Minisprachen erläutert sowie der Aufbau der Minisprachen LOOP, WHILE und GOTO analysiert und ausführlich beschrieben. Um Fehlermeldungen generieren zu können, wurde die Syntax der Minisprachen mit einer formalen Grammatik beschrieben. Als Grundlage für die Beschreibung der Sytax von Minisprachen wurden in dieser Arbeit PEG-Grammatiken ausgewählt.

Für die Implementierung der Syntaxüberprüfung und der Fehlermeldungen wurde ein Parser-Generator, PEG.js, der auf den PEG-Grammatiken basiert, verwendet. Mit PEG.js wurden die Grammatiken von LOOP, WHILE und GOTO beschrieben, sodass es für den Knoala-Nutzer hilfreiche Fehlermeldungen vom Parser produziert werden konnten. Der Parser wurde so in den Editor integriert, dass er nach jeder Nutzereingabe ausgeführt und die Syntax des Programms analysiert wird. Wenn das Programm syntaktisch nicht richtig ist, wird die Zeile mit dem Fehler markiert und eine Fehlermeldung für den Nutzer angezeigt.

Schließlich wurde die neue Funktionalität in einer Nutzerstudie evaluiert. In diesem Zusammenhang wurde ein A/B-Test durchgeführt und die Version des Editors ohne Fehlermeldungen mit der modifizierten Version mit eingebauten Fehlermeldungen verglichen.

Die Ergebnisse der Nutzerstudie zeigten, dass die integrierten Fehlermeldungen den Studenten halfen, die Fehler im Programm zu finden, zu korrigieren und ein syntaktisch richtiges Programm zu schreiben. Ferner legen die Ergebnisse der Studie offen, dass die Studenten, die den Editor mit Fehlermeldungen benutzt haben, den Aufbau und die Syntax von LOOP- und WHILE-Programmen besser verstanden haben.

Derzeit funktioniert der Parser so, dass er nach dem ersten Fehler im Programm abgebrochen und dem Nutzer der Fehler angezeigt wird. Weitere Abschnitte des Programms werden nicht mehr analysiert und eventuell vorhandene Fehler nicht erkannt, bevor der vorangegangene Fehler korrigiert wird. In Zukunft könnte man den Parser erweitern, dass er nach jedem Fehler versucht, das Programm weiter zu analysieren, alle vorhandenen syntaktischen Fehler zwischenspeichert und den Nutzern mehrere Fehlermeldungen gleichzeitig anzeigt.

Ferner könnte man untersuchen, wo Studenten trotz der generierten Fehlermeldungen Schwierigkeiten haben und oft gleiche syntaktische Fehler machen, analysieren, welche Art von Fehlern das sind, und dann anhand dieser Ergebnissen die Annotationen für Fehlermeldungen präziser zu machen.

Anhang

A.1 Folien für die Nutzerstudie

LOOP- und WHILE-Programme

Nutzerstudie zur Bachelorarbeit
“Generierung didaktischer Fehlermeldungen für Minisprachen”

Ablauf

1. Semantik und Aufbau von LOOP-Programmen
2. zwei Beispiele von LOOP-Programmen
3. drei Aufgaben zu LOOP-Programmen
4. Semantik und Aufbau von WHILE-Programmen
5. ein Beispiel von WHILE-Programm
6. zwei Aufgaben zu WHILE-Programmen
7. Fragebogen zum Benutzen des Editors

LOOP-Programme sind wie folgt definiert:

Variablen: **x1 x2 x3 …**

Konstanten: **0, 1, 2, …**

Trennsymbole: **; :=**

Operationszeichen: **+ -**

Schlüsselwörter: **LOOP DO END**

Semantik von LOOP-Programmen

1. LOOP-Programme berechnen Funktionen über IN
2. Eingabewerte **n1, ..., nk** bei Beginn der Programmausführung an Variablen **x1, ..., xk** gebunden
3. Alle anderen Variablen haben Anfangswert **0**
4. Ausgabewert ist Wert von **x0** nach Beendigung des Programms

Aufbau der LOOP-Programmen:

Nach der Initialisierung wird das LOOP-Programm abgearbeitet:

- **xi:= xj + c**

  der Variable xi wird als neuer Wert die Summe des (alten) Wertes für xj und der Zahl c zugeweisen

- **xi:= xj - c**

  der Variable xi wird als neuer Wert die Differenz des (alten) Wertes für xj und der Zahl c zugeweisen, falls diese größer 0 ist; ansonsten wird xi der Wert 0 zugewiesen

- **P1; P2**

  Es wird zuerst P1 und dann P2 ausgeführt.

Aufbau der LOOP-Programmen (Fortsetzung):

- Ist P ein LOOP-Programm, so ist auch

LOOP xi DO P END

ein LOOP-Programm.
Interpretation: Führe P genau xi-mal aus.
Achtung: Zuweisungen an xi im Innern von P haben keinen Einfluss auf die Anzahl der Schleifendurchläufe!

- Das Ergebnis der Abarbeitung ist der Wert der Variable x0 nach dem Beenden der Berechnung

Beispiel 1

x0 := x1 * x2 als LOOP-Programm

```
LOOP x1 DO
    LOOP x2 DO
        x0 := x0 + 1
    END
END
```

Beispiel 1

```
#x1 :=2
#x2 :=2
LOOP x1 DO
    LOOP x2 DO
        x0 := x0 + 1
    END
END
```

1) x0 := 0 + 1
2) x0 := 1 + 1
=> x0 := 2

1) x0 := 2
2) 1. x0 := 2 + 1
2. x0 := 3 + 1

=> x0 := 4

Beispiel 2

```
#x0 := x1 - x2
#wenn x1 < x2, dann ist x0 := 0

x0 := x1 + 0;
LOOP x2 DO
    x0 := x0 - 1
END
```

Aufgaben

1. Finden und korrigieren Sie alle Syntaxfehler im Programm, führen Sie das Programm mit beliebigen Argumenten aus.

```
LOOP x 3 DO
    LOOP x4
        x0 := x1 -3
        LOOP x5 DO
            x0 = x3 - 5;
        END
END
```

Aufgaben

2. Geben Sie ein LOOP-Programm zur Berechnung der Funktion

   x0 := x1 + x2 an.

3. Modifizieren Sie das Programm so, dass x0 := x1 * 2 berechnet wird, führen Sie das Programm mit passenden Argumenten aus

```
LOOP x1 DO
    x0 := x0 + 1
END
```

Beispiel 1

x0 := x1 * x2

```
LOOP x1 DO
    LOOP x2 DO
        x0 := x0 + 1
    END
END
```

Beispiel 2

```
#x0 := x1 - x2
#wenn x1 < x2,
#dann ist x0 := 0

x0 := x1 + 0;
LOOP x2 DO
    x0 := x0 - 1
END
```

WHILE-Programme sind wie folgt definiert:

Variablen: **x1 x2 x3 …**

Konstanten: **0, 1, 2, …**

Trennsymbole: **; :=**

Operationszeichen: **+ -**

Schlüsselwörter: **WHILE LOOP DO END**

Aufbau von WHILE-Programmen

1. Jedes LOOP-Programm ist ein WHILE-Programm
2. Wenn P ein LOOP-Programm ist, dann ist

WHILE x1 DO P END

ein WHILE-Programm. WHILE-Schleife prüft stets nur auf Ungleichheit zur 0.

Interpretation: P wird ausgeführt, solange der aktuelle Wert von x1 ungleich 0 ist.

Achtung: dies hängt davon ab, wie P den Wert von x1 ändert

Ansonsten werden WHILE-Programme wie LOOP-Programme ausgewertet

Beispiel

x0 := x1 * x2 als WHILE-Programm

```
#x0 := x1 * x2
WHILE x1 DO
    LOOP x2 DO
        x0 := x0 +1
    END;
    x1 := x1 -1
END
```

x0 := x0 + x2

Aufgabe 1

1. Finden und korrigieren Sie alle Syntaxfehler im Programm, führen Sie dann das Programm mit beliebigen Argumenten aus.

```
WHILE x1 < 0 DO
    LOOP x2 DO
        x0 := x0 + 1
    END
    x1 := x1 -1;
End;
x2 := x4 + 4;
x0 := x2
```

Aufgabe 2

2. Geben Sie ein WHILE-Programm zur Berechnung der Funktion

   x0 := x1 + x2 (benutzen Sie die WHILE-Schleife)

A.2 Fragebogen zum Benutzen des Editors

Feedback zum Editor

1. **Ihr Benutzername**

2. **Ihr Studiengang:**

3. **In welchem Semester studieren Sie?**

4. **Ich bin**
   Markieren Sie nur ein Oval.
   - männlich
   - weiblich
   - keine Angabe

5. **Konnten Sie alle Aufgaben vollständig bearbeiten?**
   Markieren Sie nur ein Oval.
   - Ja
   - Nein

6. **War die Aufgabenstellung einfach zu verstehen?**
   Markieren Sie nur ein Oval.
   - Ja
   - Nein

7. **Haben Sie die Vorlesung "Formale Sprachen und Komplexität" oder "Theoretische Informatik für Medieninformatiker" besucht?**
   Markieren Sie nur ein Oval.
   - Ja
   - Nein

8.

Hatten Sie bereits die Vorkenntnisse in der Programmiersprache LOOP?

Markieren Sie nur ein Oval.

- Ja
- Nein

9.

Hatten Sie bereits die Vorkenntnisse in der Programmiersprache WHILE?

Markieren Sie nur ein Oval.

- Ja
- Nein

10.

Finden Sie den Syntaxfehler in diesem LOOP-Programm. Schreiben Sie die Nummer der Zeile mit Fehler und beschreiben Sie kurz den Fehler.

```
1 LOOP x1 DO
2     LOOP x2 DO
3         x0 := x0 + 10;
4     END
5 END
6
```

11.

Finden Sie den Syntaxfehler in diesem LOOP-Programm. Schreiben Sie die Nummer der Zeile mit Fehler und beschreiben Sie kurz den Fehler.

```
1  LOOP x1 DO
2      LOOP x2 DO
3          x0 := x0 + 10
4      END;
5      x3 := 5;
6      LOOP x3 DO
7          x0 := x0 -1
8      END
9  END
10
```

12.

Unterstützt der Editor das Lösen der Aufgaben?

Markieren Sie nur ein Oval pro Zeile.

	stimme zu	stimme eher zu	stimme eher nicht zu	stimme nicht zu
Der Editor hat mir geholfen Fehler zu finden				
Der Editor ist kompliziert zu bedienen				
Ich konnte die Struktur der Programmiersprachen LOOP und WHILE besser verstehen, nach dem ich die Aufgaben gelöst habe				
Das Benutzen des Editors war intuitiv				
Ich konnte die Fehler im Programm schon vor dem Ausführen entdecken und korrigieren				

13.

Haben Sie Kommentare zum Benutzen des Editors?

Literaturverzeichnis

[1] *Ace code editor*, `https://ace.c9.io/`, Aufgerufen: 10.09.2019.

[2] *Backstage 2 projects*, `https://backstage2projects.pms.ifi.lmu.de:8080/`, Aufgerufen: 02.10.2019.

[3] Bryan Ford, *Parsing expression grammars: A recognition-based syntactic foundation*, Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages **39** (2004), no. 1, 111–122.

[4] Scott Freeman, Sarah L. Eddy, Miles McDonough, Michelle K. Smith, Nnadozie Okoroafor, Hannah Jordt, and Mary Pat Wenderoth, *Active learning increases student performance in science, engineering, and mathematics*, Proceedings of the National Academy of Science of the United States of America **111** (2014), no. 23, 8410–8415.

[5] Niels Heller and François Bry, *Learning by fiddling: Patterns of behaviour in formal language learning*, International Conference in Methodologies and intelligent Systems for Techhnology Enhanced Learning, **Springer** (2019), 28–36.

[6] Jozef Hvorecky, Anatoly Kouchnirenko, Peter Brusilovsky, Eduardo Calabrese, and Philip Miller, *Mini-languages: a way to learn programming principles*, Education and Information Technologies **2** (1997), no. 1, 65–83.

[7] Donald E. Knuth, *Top-down syntax analysis*, Acta Informatica **1** (1971), no. 2, 79–110.

[8] David Majda, *Peg.js parser generator for javascript*, `https://pegjs.org/`, Aufgerufen: 20.09.2019.

[9] Sérgio Medeiros and Fabio Mascarenhas, *Syntax error recovery in parsing expression grammars*, Proceedings of the 33rd Annual ACM Symposium on Applied Computing (2018), 1195–1202.

[10] Raimond Reichert, Jürg Nievergelt, and Werner Hartmann, *Programmieren mit Kara*, Springer-Verlag Berlin Heidelberg (2005), 27–28.

[11] Uwe Schöning, *Theoretische informatik - kurz gefasst*, Springer Spektrum (2008), 92–100.

[12] Arie van Deursen and Paul Klint, *Domain-specific languages design requires feature descriptions*, Journal of Computing and Information Technology -CIT **10** (2002), no. 1, 1–17.

[13] Arie van Deursen, Paul Klint, and Joost Visser, *Domain-specific languages: An annotated bibliography*, ACM SIGPLAN Notices **35** (2000), no. 6, 26–36.