

JumpVM

Java Unified Multi Paradigm Virtual Machine

Tim Wiederhake, B.Sc.

29. Mai 2015

JumpVM ist die „Java Unified Multi Paradigm Virtual Machine“: Grafische Oberfläche, Parser, Compiler und Virtuelle Maschine für einige verschiedene Programmiersprachen und Programmierparadigmen. Das Hauptziel dieser Software ist es, die Vorlesungen „Einführung in den Übersetzerbau“ und „Übersetzung fortgeschrittener Sprachkonzepte“ an der Universität Ulm zu unterstützen. Die Namen und Fähigkeiten der verschiedenen unterstützten Programmiersprachen lehnen sich an das Buch „Übersetzerbau: Theorie, Konstruktion, Generierung“ [8] von R. Wilhelm und D. Maurer an, das in den genannten Vorlesungen verwendet wird.



Inhaltsverzeichnis

1	Einleitung	3
1.1	Motivation	3
1.2	Aufbau	3
2	Grundlagen	3
3	Bestandsaufnahme	5
3.1	pmach	5
3.2	mama	7
3.3	wim	8
4	Designziele	9
5	Entwicklung	10
5.1	PaMa	14
5.2	MaMa	16
5.3	WiMa	16
5.4	BfMa	17
5.5	Organisation des JumpVM Quellcodes	18
6	Bedienung	19
7	Fazit	22
	Literatur	23

© 2015 Tim Wiederhake

Die Weitergabe und Bearbeitung dieses Dokumentes ist unter den Bedingungen der „Creative Commons Namensnennung 3.0 Deutschland“-Lizenz gestattet:
<http://creativecommons.org/licenses/by/3.0/de/>.

1 Einleitung

Neben den geläufigen imperativen Programmiersprachen gibt es einige weitere Programmiersprachen, die auf anderen Programmierparadigmen beruhen, etwa funktionale oder logische Programmiersprachen. Allen gemein ist, dass Programme, die in diesen Programmiersprachen geschrieben sind, in aller Regel nicht direkt auf einem Prozessor ausgeführt werden können, sondern erst in einen zum Prozessor kompatiblen Instruktionssatz übersetzt werden müssen.

1.1 Motivation

Der Compilerbau ist diejenige Disziplin der Informatik, die sich mit dem Entwurf, der Entwicklung und dem mathematischen Unterbau dieser Übersetzer beschäftigt.

Die hier vorgestellte, im Rahmen einer Projektarbeit an der Universität Ulm entstandene Software JumpVM soll die Lehre unterstützen und die Übersetzung unterschiedlicher Programmierparadigmen praktisch erfahrbar machen.

1.2 Aufbau

Dieses Dokument ist in sieben Kapitel unterteilt. Nach der Einleitung in Kapitel 1 wird in Kapitel 2 auf Grundbegriffe und Grundlagen des Compilerbaus eingegangen. In Kapitel 3 wird eine Bestandsaufnahme der bisherigen Implementierung der vorlesungsunterstützenden Programme gemacht und die Probleme mit dieser Implementierung aufgezeigt. Darauf aufbauend werden in Kapitel 4 Designschwerpunkte und Anforderungen an die zu entwickelnde Software formuliert. Kapitel 5 widmet sich dem Entwurf und der Entwicklung der Software, deren Bedienung in Kapitel 6 erläutert wird. Abschließend wird in Kapitel 7 ein Fazit gezogen.

2 Grundlagen

In diesem Kapitel werden einige Grundbegriffe und Grundlagen der Arbeitsweise eines Compilers erläutert.

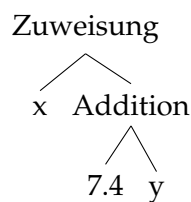
Lexer, Token Aufgabe des Lexers ist es, Quellcode einer bestimmten Programmiersprache zeichenweise einzulesen und in Symbolen, „Tokens“, zu gruppieren. Beispiels-

2 Grundlagen

weise könnte ein Lexer die Zeichenfolge „`x = 7.4 + y;`“ in die folgende Tokenliste übersetzen:

Bezeichner (`x`), Zuweisungszeichen, Zahl (`7.4`), Additionszeichen, Bezeichner (`y`), Befehlstrenner.

Parser, Syntax Tree Der Parser verwandelt die lineare Darstellung des Programms als Tokenliste in einen Syntax Tree, eine Baumstruktur, die der semantischen Struktur des Programms entspricht. Dieser Baum ist die Grundlage für Optimierungen und Überprüfungen und könnte für obiges Beispiel folgendermaßen aussehen:



Compiler, Assemblerbefehl Der Syntax Tree wird vom Compiler in eine Liste von Assemblerbefehlen übersetzt. Diese Befehle sind textuelle Darstellungen von Instruktionen eines bestimmten Prozessors und entsprechen in ihrer Bedeutung dem Quellprogramm. Eine mögliche Folge von Assemblerbefehlen, die dem oben genannten Beispielprogramm entsprechen könnten, wäre:

```
lda 5, ldc 7.4, lda 6, ind, add, sto.
```

Assembler, Linker Die textuelle Darstellung von Assemblerbefehlen wird vom Assembler in Maschinencode übersetzt und vom Linker mit eventuellen anderen Teilen Maschinencode zusammengefügt. Hierbei entsteht ein lauffähiges, binäres Programm.

Virtual Machine Existiert der Prozessor zum Ausführen eines Programms nicht real, sondern selber nur als Software, so spricht man von einer Virtuellen Maschine. Da reale Prozessoren zunehmend komplexer werden, eignen sich Virtuelle Maschinen besonders, um in der Lehre die Funktionsweise eines Compilers bzw. Assemblers zu verdeutlichen. Die Instruktionen einer Virtuellen Maschine sind selber definierbar und an die gewählte Quellsprache anpassbar.

3 Bestandsaufnahme

In den Linux-Rechnerpools der SGI sind für die Abteilung Compilerbau im Verzeichnis `/opt/Abteilungen/pm/` mehrere Programme installiert. Diese sind mit Tk-Gofer 2.0 programmiert und stammen aus den Jahren 1995–1996 (mama) und 1998 (pmach); für wim konnte keine Datierung gefunden werden.

Gofer ist ein Haskell-Dialekt, der seit 1999 nicht weiterentwickelt wird [5] und auf den Rechnern, auf denen das Betriebssystem Ubuntu Linux 13.10, Kernel 3.11.0-18 läuft, nicht aus den Repositories installierbar. Daher liegt im angegebenen Verzeichnis eine selbsterstellte Gofer/Tk-Gofer-Installation.

Tk ist ein Toolkit zur Entwicklung von grafischen Benutzeroberflächen, entwickelt für die Scriptsprache Tcl. Die Tk-Gofer Installation empfiehlt die Version 4.2 oder höher für Tk und Version 7.6 oder höher für Tcl. Die niedrigste Version der Softwarepakete Tcl und Tk in den Ubuntu-Repositories ist Version 8.5.

Die Programmiersprache Haskell liegt im Tiobe-Index der populärsten Programmiersprachen (Stand: Oktober 2014) auf Platz 40 mit einem Rating von 0,341% [7]. Die Programmiersprache Gofer ist im Ranking nicht vertreten. Davon ausgehend kann angenommen werden, dass die Zahl der Programmierer, die Goferprogramme warten können, äußerst gering ist. Die Verwendung einer selbstgeschriebenen Bibliothek zur Anbindung von Gofer an Tk/Tcl erschwert die Wartbarkeit der vorliegenden Programme weiter.

Installieren lassen sich diese Programme etwa unter Windows oder MacOS gar nicht, unter Linux nur mit einigem Aufwand. Beispielsweise sind die in Debian 8 (Jessie) vorzufindenden Bibliotheken libtk8.5 und libtcl8.5 mit den Programmen inkompatibel.

Des Weiteren ist die Lizenzierung der ursprünglichen Programme unbekannt, daher muss vom restriktivsten Fall ausgegangen werden, der keine Installation auf einem eigenen Rechner erlaubt. Dies ist ungünstig, wenn die Programme für den Übungsbetrieb einer Vorlesung eingesetzt werden sollen und nur an einer sehr eingeschränkten Zahl Computern benutzbar sind.

3.1 pmach

Abbildung 1 zeigt das Programm pmach. In der oberen Hälfte des Fensters werden die Programminstruktionen des kompilierten Beispielprogramms, der Stack und der Heap des aktuell laufenden Programms angezeigt. Die Programminstruktionen sind teilweise mit Abkürzungen annotiert. Die Instruktion `lda 0 0 "FP: fak"` bezieht sich etwa

3 Bestandsaufnahme

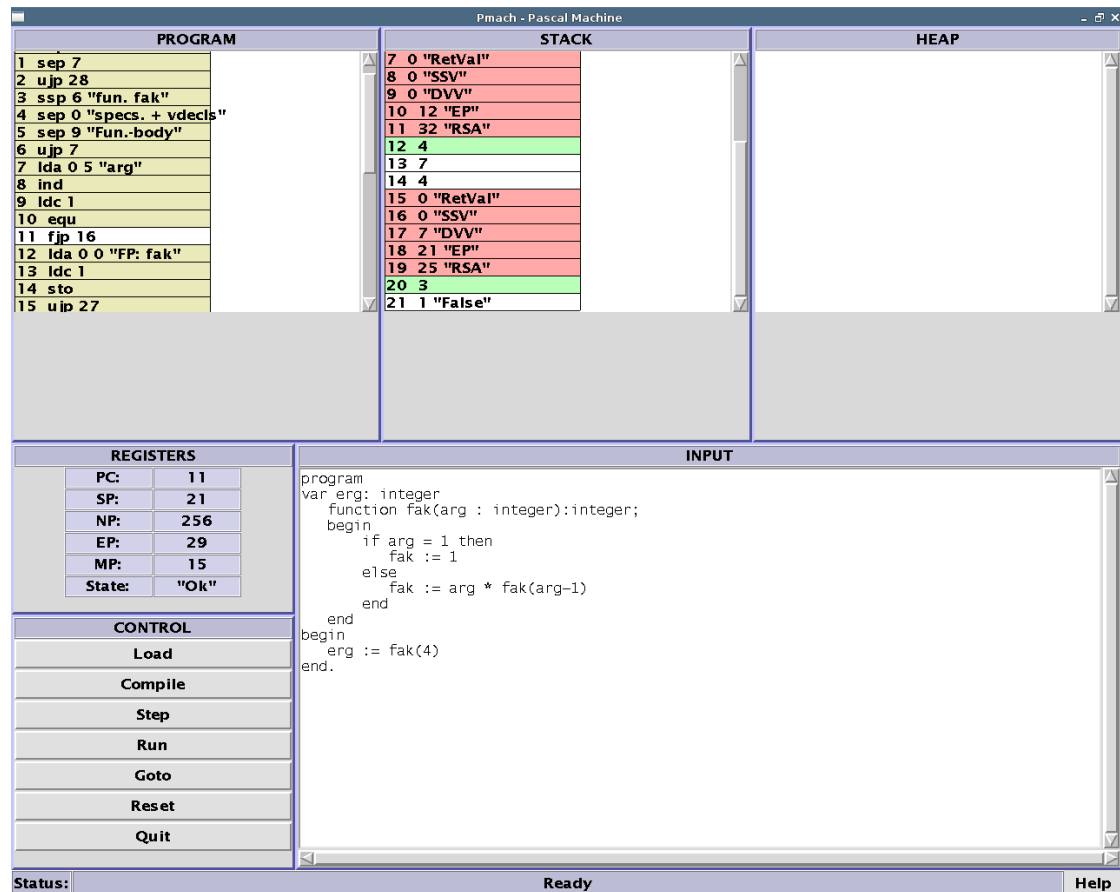


Abbildung 1: Die pmach beim Ausführen eines Beispielprogramms.

auf die Framepointer der Funktion „fak“. Ähnlich sind die organisatorischen Zellen der einzelnen Frames auf dem Stack beschriftet, zusätzlich werden diese und die Zellen, die lokale Variablen enthalten, jeweils farblich hervorgehoben.

Die untere Hälfte des Fensters beinhaltet eine Übersicht über die Register der Virtuellen Maschine und ihren jeweiligen Wert. Darunter befinden sich einige Schaltflächen, die der Steuerung des Programms dienen, etwa dem Laden, Speichern, Compilieren und Ausführen von Programmen. Rechts davon ist ein Eingabefeld zu sehen, in dem der Quellcode des aktuellen Programms angezeigt wird und geändert werden kann.

Eine Statuszeile zeigt den momentanen Zustand der VM an (etwa „Ready“ oder „Running program ...“), ein Klick auf die Schaltfläche „Help“ zeigt ein Fenster mit der verwendeten Grammatik für die Übersetzung der Programme in EBNF-Syntax an und beschreibt kurz die verfügbaren Schaltflächen und verwendeten Farben.

3 Bestandsaufnahme

Ein Fehler in der pmach sorgt dafür, dass boolesche Werte falsch angezeigt werden. Die Größe der einzelnen Bereiche „Program“, „Stack“ etc. passen sich teilweise nicht der tatsächlichen Fenstergröße an, wodurch immer maximal 15 Einträge des jeweiligen Speichers sichtbar sind. Die Ausführung größerer Programme verlangsamt sich mit jedem Schritt; ab einer bestimmten Länge der Ausführung oder beim Übersetzen größerer Programme beendet sich das Programm aufgrund von Speichermangel mit folgender Fehlermeldung im Terminal:

```
ERROR: Garbage collection fails to reclaim sufficient space
```

3.2 mama

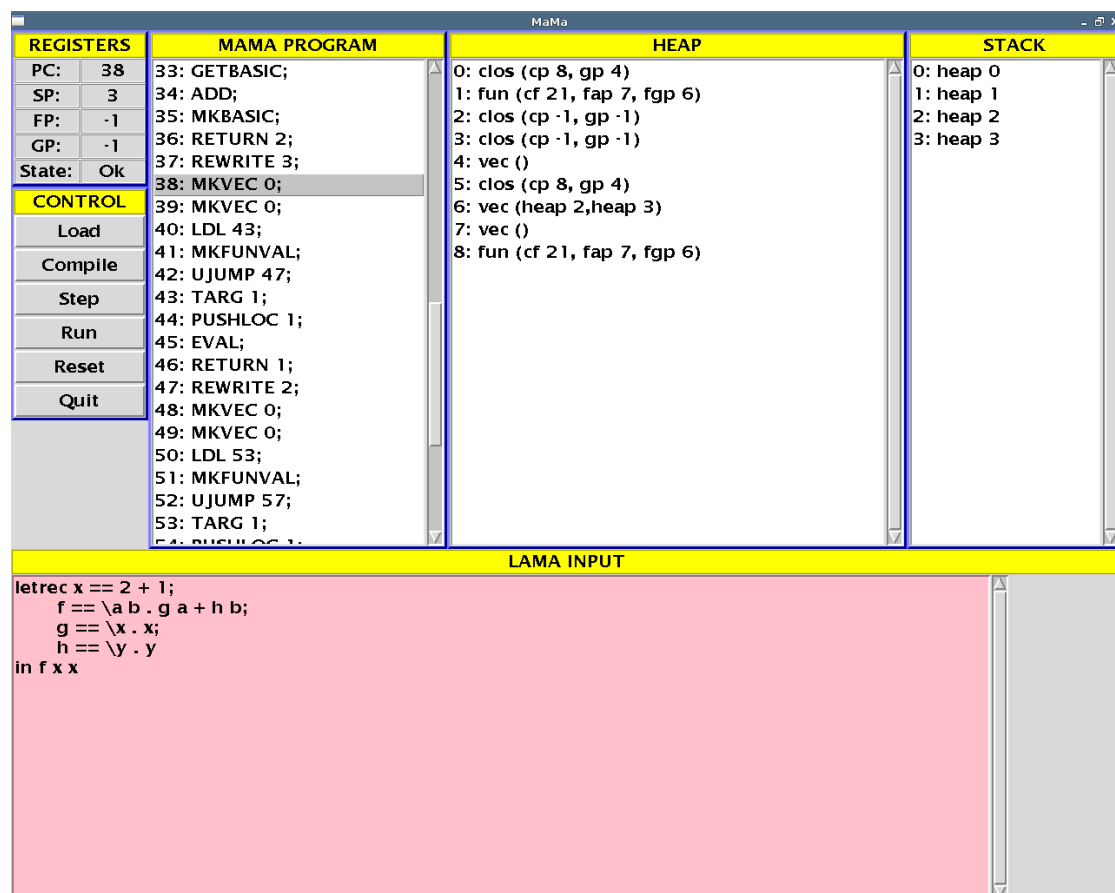


Abbildung 2: Die mama beim Ausführen eines Beispielprogramms.

Abbildung 2 zeigt das Programm mama. In der oberen Hälfte des Programms wird, wie bei der pmach, der Inhalt des Speichers angezeigt, allerdings mit vertauschter Reihenfolge von Stack und Heap.

3 Bestandsaufnahme

Die Registeranzeige und die Kontrollschaltflächen finden sich hier in der oberen Hälfte des Fensters, im Gegensatz zur pmach. Die Schaltfläche „Goto“ fehlt. Die Elemente im Speicher werden nicht eingefärbt und bei der Benutzung des Programms fällt auf, dass der Versuch syntaktisch inkorrekte Programme zu übersetzen zum Beenden des Programms führt. Eine Fehlermeldung wird nur im Terminal ausgegeben, beispielsweise:

```
Program error: unknown identifier z
```

3.3 wim

Wim - Prolog Machine			
PROGRAM	HEAP	STACK	TRAIL
74 pusharg 1	0 Ref (0)	14 Hp Addr (1)	0 2
75 uatom martin	1 Atom ("jan")	15 Hp Addr (0)	1 3
76 popenv	2 Ref (4)	16 Hp Addr (1)	
77 pushenv 7	3 Ref (5)	17 Hp Addr (2)	
78 setbtp 86	4 Atom ("hugo")	18 Hp Addr (3)	
79 pusharg 1	5 Atom ("bertha")	19 Hp Addr (-1)	
80 uatom jan		20 Prq Addr (28)	
81 pusharg 2		21 FP (8)	
82 uatom hugo		22 BTP (8)	
83 pusharg 3		23 Tr Addr (-1)	
84 uatom bertha		24 HP (4)	
85 restore		25 Prq Addr (86)	
86 pushenv 7		26 Hp Addr (1)	
87 nextalt 95		27 Hp Addr (2)	
88 pusharg 1		28 Hp Addr (3)	
REGISTERS	SYSTEM		
PC: 85	Compilation successful		
SP: 28			
FP: 21			
BTP: 21			
HP: 6			
TP: 1	Goal: grandfather(X,jan)		
Mode Read			
State: Ok			
CONTROL			
Load			
Compile			
Step			
Run			
Reset			
Quit			

Abbildung 3: Die wim beim Ausführen eines Beispielprogramms.

Abbildung 3 zeigt das dritte Programm wim. Wieder finden sich in der oberen Hälfte des Fensters die Inhalte der verschiedenen Speicherbereiche, ohne Kommentare, dafür eingefärbt. Die Farbgebung wird nicht erläutert. In der unteren Hälfte des Fensters befinden sich die Schaltflächen zur Steuerung des Programms, die Registerübersicht und ein Textfeld zur Eingabe des Programms. Zusätzlich ist ein Feld vorhanden, in dem

4 Designziele

Ausgaben angezeigt werden, etwa Meldungen zur erfolgreichen Übersetzung eines Programms oder Ausgaben durch die Ausführung des Programms.

Das Eingabefeld für Programmtext ist zweigeteilt. Das Ziel („Goal“) eines wim-Programms kann nicht zusammen mit seinem restlichen Quelltext gespeichert oder geladen werden. Leerzeichen im Quelltext werden als syntaktische Fehler betrachtet.

Außerdem scheint die Ausführung von wim-Programmen fehlerhaft zu sein, beispielsweise erzeugt die Ausführung des Programms mit der Faktenbasis „equal(X,X).“ und dem Ziel „equal(a,b)“ die Ausgabe „Yes“.

4 Designziele

Für die Entwicklung eines Nachfolgers ergeben sich aus der Untersuchung der Programme, ihrer Stärken und Schwächen, einige Schwerpunkte und Designziele:

Plattformunabhängigkeit und Portabilität Als Werkzeug der Übung und zur Unterstützung der Vorlesung soll das Programm auf allen verwendeten Plattformen des Dozenten, der Hilfskräfte und der Studenten lauffähig sein. Das Programm soll ohne Installation lauffähig sein.

Freizügige Lizenzierung Die Lizenzierung der Software muss es den Studenten erlauben, die Software auf einem eigenen Rechner mindestens für die Dauer des Besuches der Vorlesung zu verwenden.

Benutzerfreundlichkeit Die Dokumentation der Software muss es den Benutzern erlauben, die Handhabung und die Funktionsweise des Programms zu verstehen. Verwendete Symbolik, wie etwa farbliche Hervorhebungen, muss selbsterklärend sein oder explizit erläutert werden. Fehleingaben müssen abgefangen werden, Fehlermeldungen auf die Art des Fehlers oder die erwartete Eingabe hinweisen.

Wartbarkeit Die Software soll in einer Programmiersprache geschrieben sein, die aufgrund hoher Verbreitung oder einfacher Erlernbarkeit Änderungen und Korrekturen durch Dritte in der Zukunft erlauben. Die Anwendung von Best Practices aus dem Bereich des Software Engineering soll die Qualität der Software maximieren.

5 Entwicklung

Ausgehend von den in Kapitel 4 genannten Anforderungen wurde im Wintersemester 2013 und Sommersemester 2014 die Software JumpVM entwickelt.

Als Programmiersprache wurde Java gewählt, da Java-Programme auf allen zu erwartenden Plattformen (Windows, MacOS, Linux) lauffähig sind und ohne zusätzliche Softwarebibliotheken grafische Oberflächen auf allen diesen Plattformen erzeugen können. Java ist im Vergleich zu etwa in C oder C++ weniger performant. Da die Anforderungen an die Benutzerhardware als gering eingeschätzt wurden, ist dies vernachlässigbar.

Eine Recherche zu verwendbaren, bereits existierenden Bibliotheken zum Parsen, Compilieren und Ausführen von Programmen und der Entwicklung von Programmiersprachen brachte zwei interessante Ergebnisse:

LLVM („Low Level Virtual Machine“) ist eine Sammlung von modularen und wiederverwendbaren Compiler- und Toolchain-Technologien [6]. Herzstück ist die LLVM-IR („LLVM Intermediate Representation“), eine architekturunabhängige Pseudo-Assemblersprache. LLVM enthält eine Vielzahl an Modulen, die Code in LLVM-IR optimieren und in Assemblerprogramme für unterschiedlichste Plattformen übersetzen kann. Die Verwendung von LLVM wurde allerdings auf Grund des unüberschaubaren Aufwandes, mehrere Backends für LLVM zu entwickeln, verworfen.

ANTLR („ANother Tool for Language Recognition“) ist, wie etwa auch YACC („Yet Another Compiler Compiler“), ein Parsergenerator [1]: Aus einer formal definierten Grammatik werden Lexer und Parser für diese Grammatik erzeugt. Zusätzlich stehen einige Werkzeuge und eine grafische Oberfläche namens ANTLRWorks für die Erstellung der Grammatik zur Verfügung.

Da für die zu erstellenden Parser keine Grammatiken zur Verfügung standen bzw. die Grammatik der pmach inkonsistent war, wurden die Grammatiken mit ANTLRWorks erstellt. Insbesondere die Möglichkeit des automatischen Testens der Grammatiken und die grafische Darstellung der einzelnen Produktionen einer Grammatik erwiesen sich als äußerst nützlich. Abbildung 4 zeigt einige Beispiele für die grafische Darstellung von Produktionen aus den entstandenen Grammatiken.

Letztendlich wurde der Einsatz des ANTLR Parsergenerators in der zu erstellenden Software verworfen, da die generierten Lexer und Parser zu komplex waren und sich nicht an die gewünschte Funktionalität anpassen ließen. Auf Basis der mit ANTLRWorks erstellten Grammatiken wurden daher händisch einfache Recursive Descent Parser entwickelt.

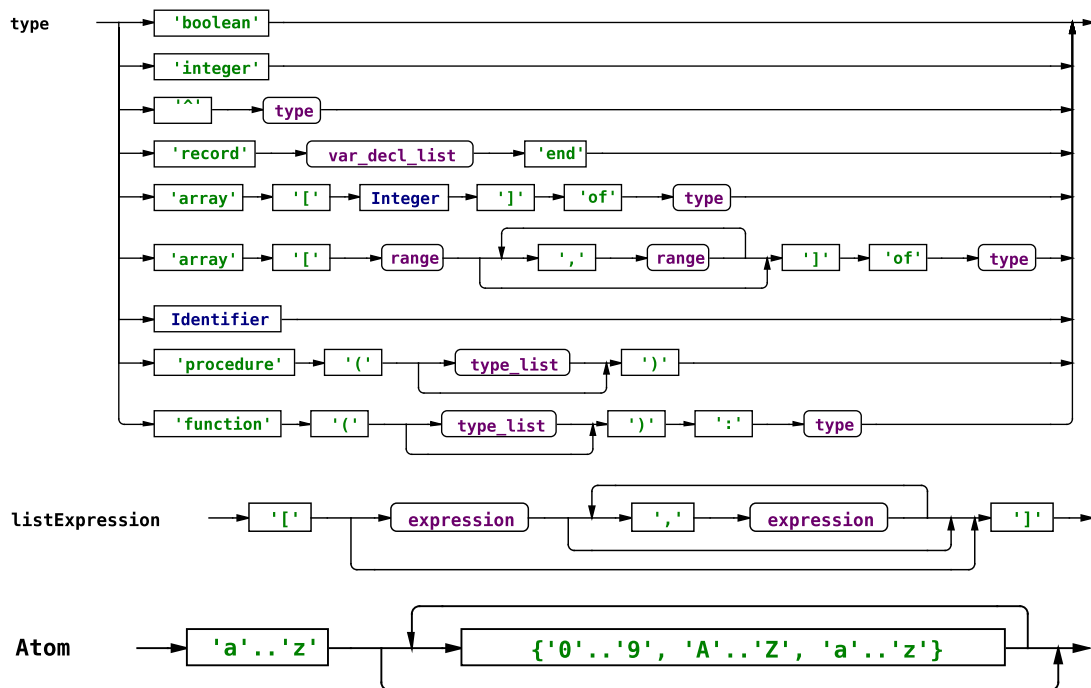


Abbildung 4: Grafische Darstellung einiger Produktionen in ANTLRWorks. Die zugehörigen Grammatiken liegen der Software bei.

Die Beispielprogramme der bestehenden Implementation wurden entnommen, ergänzt und als Testfälle für Unit-Tests der einzelnen Komponenten (Lexer, Parser, Compiler und VM) verwendet. So konnte sichergestellt werden, dass die Software die gleichen Daten wie die bisherigen Implementationen produziert, also die gleichen Maschinenbefehle aus dem gleichen Quelltext erzeugt. Abbildung 5 zeigt das Ergebnis eines solchen Testlaufs: Für jede Beispieldatei stimmen die Ergebnisse von Lexer, Parser, Compiler und Ausführung des Programms mit den erwarteten Ergebnissen überein.

Checkstyle [2] ist ein Werkzeug für statische Codeanalyse von Java-Programmen, mit dem die Einhaltung von Programmierstandards überprüft werden kann. Häufige Fehler können so erkannt und behoben werden. Die Spanne der erkannten Fehler reicht vom direkten Vergleich von Zeichenketten mit dem „==“-Operator (statt „equals“) bis zu Verletzungen der Prinzipien objektorientierten Designs (wie etwa mangelnde Datenkapselung) und fehlender Dokumentation. Um die Qualität des Quellcodes zu erhöhen, wurden alle von Checkstyle erkannten Mängel behoben.

Die Dokumentation des Quellcodes erfolgt mit dem Java-eigenen Werkzeug JavaDoc. In Kombination mit Checkstyle kann so sichergestellt werden, dass jede Klasse, Funktion und Klassenfeld für Änderungen und Korrekturen durch Dritte dokumentiert und in

5 Entwicklung

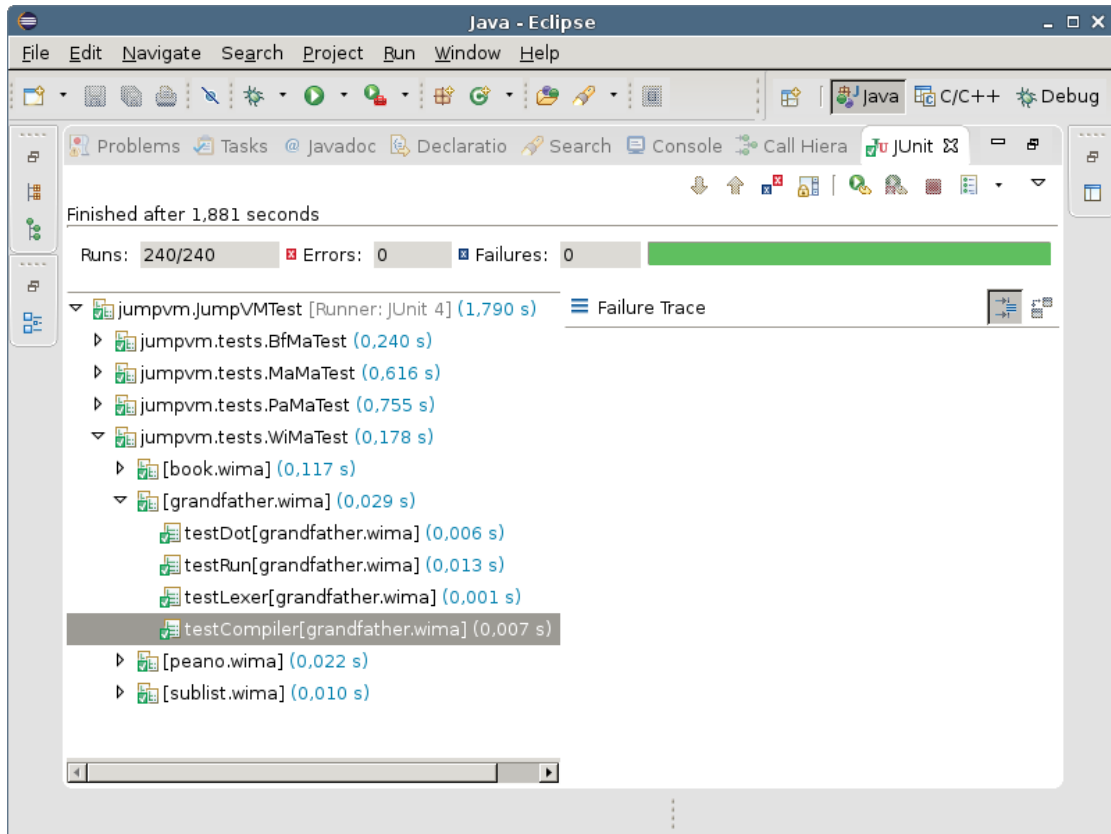


Abbildung 5: Testen der Software mit JUnit.

ihrer Funktion beschrieben ist. Abbildung 6 zeigt exemplarisch die Kommentierung einer Funktion der Klasse „jumpvm.compiler.Parser“. Durch den formellen Aufbau der Kommentare ist es möglich, die gesamte Code-Dokumentation beispielsweise im HTML oder PDF-Format zu extrahieren und konvertieren.

Zur möglichst einfachen Verwendung wurde die gesamte Software unter die „GNU General Public License“ in der Version 3 oder höher gestellt. Diese Lizenzierung ermöglicht es jedem Anwender, die Software nach Belieben zu nutzen, kopieren oder weiterzuentwickeln. Dies kommt besonders Nutzern im akademischen Bereich zu Gute, da es das Untersuchen der Software, um sie verstehen zu lernen, explizit erlaubt und die Weiterentwicklung vereinfacht.

Die Software ist modular angelegt, mit generischen Lexer-, Parser- und Compiler-Funktionen, die von den Modulen der einzelnen Übersetzern und Virtuellen Maschinen verwendet und angepasst werden können. Die Organisation des JumpVM-Quellcodes ist in Kapitel 5.5 genauer erläutert. Die Verwendung der generischen Klassen erlaubt es, die Oberfläche des Programms einheitlich zu gestalten.

```

1  /**
2  * Discard {@code expectedToken} and read next token.
3  *
4  * @param expectedToken expected token
5  * @throws ParseException if current token != expected token
6  */
7  protected final void eat(final T expectedToken) throws
    ParseException {
8      expect(expectedToken);
9      token = lexer.nextToken();
10 }

```

Abbildung 6: Kommentierung mit JavaDoc.

Durch diese Modularisierung und Vereinheitlichung der Schnittstellen war es möglich, einige neue Funktionen, die in der bisherigen Implementierung nicht existierten, in die Software einzubauen. Unabhängig vom verwendeten Modul kann die Ausführung eines Programms schrittweise rückgängig gemacht werden, um den Effekt einer Instruktion genauer zu untersuchen. Die Werte der Register und Speicherzellen lassen sich auch während der Ausführung ändern, um das Austesten von „Was wäre wenn...“-Szenarien zu erlauben. Das Konzept der teilweise annotierten Speicherzellen der pmach wurde übernommen und erweitert: Jede Speicherzelle ist zum besseren Verständnis mit ihrer jeweiligen Bedeutung und ihrem Datentyp annotiert. Beispielsweise könnte eine Speicherzelle anzeigen, dass es sich bei ihrem Wert um einen Zeiger in den Programmspeicher (Datentyp) auf die Funktion „f“ (Bedeutung des Wertes) handelt.

Eine weitere Verbesserung gegenüber der vorherigen Implementierung ist die interaktive Zuordnung von Quellcode eines Programms, seinem Syntaxbaum und den produzierten Instruktionen des Maschinencodes. Sie erlaubt es dem Benutzer unmittelbar nachzuvollziehen, welcher Teil seines Quellcodes zu welchen Instruktionen geführt hat. Abbildung 13 zeigt, wie diese Zuordnung im fertigen Programm umgesetzt wird: Zusammengehörende Instruktionen, Knoten im Syntax-Baum und Abschnitte im Quellprogramm sind jeweils identisch eingefärbt.

Die Syntax der Programmiersprachen der bisherigen Implementierung wurde um Befehle zur Ein- und Ausgabe von Daten erweitert, um die möglichen Programme für die Benutzer interessanter machen zu können.

Um die Software auch als Werkzeug zur Vorlesungsvorbereitung attraktiv zu machen und den Datenaustausch mit anderen Programmen zu ermöglichen, wurden Funktionen zum Export des Syntax-Baumes als Graphviz-DOT-Dateien [3] implementiert, wie

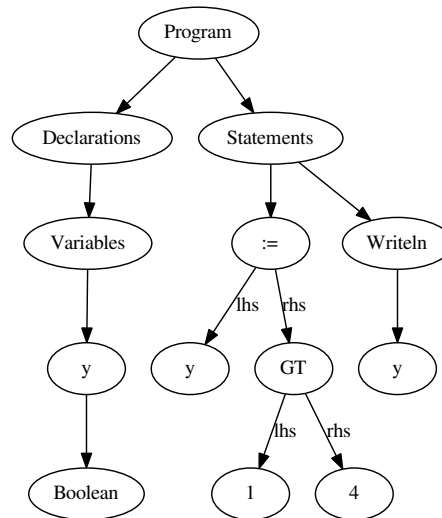


Abbildung 7: Darstellung eines Programms als Syntaxbaum.

etwa in Abbildung 7 zu sehen. Zusätzlich ist der Export des momentanen Zustandes der laufenden Virtuellen Maschine im XML-Format möglich.

5.1 PaMa

Das Modul PaMa ist in der Lage, Programme in einer Pascal-ähnlichen, imperativen Sprache zu übersetzen und auszuführen. Abbildung 8 zeigt das Beispielprogramm „fak.pama“, das die Fakultätsfunktion berechnet.

PaMa wurde gegenüber pmach um Funktionen zur Ein- und Ausgabe von Daten (`readln` und `writeln`) erweitert. Die fehlerhafte Darstellung von Booleschen Werten wurde korrigiert und die Setzung von „;“ (Semikolon) im Deklarationsteil von Funktionen und Prozeduren vereinheitlicht.

Während Pascal noch einige weitere Datentypen kennt, sind die einzigen primitiven Datentypen dieser Programmiersprache die Ganzzahl (`integer`) und der Wahrheitswert (`boolean`). Darüber hinaus sind zusammengesetzte Datentypen (`record`), Felder (`array`) und Zeiger (`pointer`) möglich.

Vor der Übersetzung der Programme werden diese auf Typkorrektheit geprüft. So werden Fehler gefunden, die die syntaktische Analyse durch den Parser nicht aufspüren

```
1 program
2     function fak(arg : integer): integer
3     begin
4         if arg = 1 then
5             fak := 1
6         else
7             fak := arg * fak(arg - 1)
8         end
9     end
10 begin
11     writeln(fak(4))
12 end.
```

Abbildung 8: Rekursive Berechnung der Fakultätsfunktion.

kann, etwa die Zuweisung eines ganzzahligen Wertes an eine Variable vom Type Wahrheitswert.

Der Compiler eröffnet für jede Prozedur / Funktion des Programms einschließlich der Hauptprozedur eine Übersetzungsumgebung, die Informationen, etwa zur Positionierung von Variablen im Speicher beinhaltet. Nun werden die einzelnen Anweisungen der Prozeduren übersetzt.

Während des Übersetzens wird zwischen dem sogenannten Linkswert und dem Rechtswert einer Variable bzw. eines formellen Ausdrucks unterschieden. Der Rechtswert bezeichnet den tatsächlichen Wert eines Ausdrucks und könnte etwa „5“ für den Ausdruck „x“ sein, oder „7“ für den Ausdruck „3 + 4“. Der Linkswert bezeichnet eine Adresse im Speicher und wird benötigt, um dem bezeichneten Datum einen neuen Wert zuzuweisen. Für den Ausdruck „x“ könnte der Linkswert, also die Adresse der Variable, „17“ sein. Für den Ausdruck „3 + 4“ existiert kein Linkswert: Diesem Ausdruck kann kein neuer Wert zugewiesen werden und er hat keinen benannten Platz im Speicher.

Die Virtuelle Maschine, die die übersetzten PaMa-Programme ausführt, ist eine Stackmaschine. Werte werden auf dem Stack manipuliert, statt wie etwa bei x86-Prozessoren in Registern gehalten. Dies ermöglicht eine sehr einfache Übersetzung und Virtuelle Maschine geringer Komplexität. Da bei Stackmaschinen sehr viele Speicheroperationen durch die Manipulation des Stacks anfallen, sind diese allerdings weniger Effizient als Registermaschinen, die einige Werte in Registern zwischenspeichern können.

5.2 MaMa

Das Modul MaMa übersetzt und führt Programme in einer funktionalen, Haskell-ähnlichen Sprache aus. Abbildung 9 zeigt das Beispielprogramm „example01.mama“, in dem eine Lambda-Funktion definiert und aufgerufen wird.

```

1 letrec
2   a == 1;
3   f == \x y . if x = 0 then y else f (x-1) y
4 in f 3 a

```

Abbildung 9: Definition einer Lambda-Funktion.

MaMa-Programme stellen jeweils einen zu berechnenden Ausdruck dar. Im obigen Beispiel etwa soll der Ausdruck „f 3 a“ berechnet werden, wobei Definitionen für „f“ und „a“ gemacht werden, die wiederum rekursiv Ausdrücke enthalten können.

Die MaMa-VM ist wie die PaMa-VM eine Stackmaschine. Allerdings werden hier nicht die Werte direkt, sondern Zeiger auf Speicherzellen im Heap auf den Stack gelegt. Dieser Zeiger kann nun auf einen Wert oder eine Funktion, die den Wert des Ausdrucks berechnet („Closure“), zeigen. Nach dem Auswerten der Closure wird der berechnete Wert an der passenden Speicherzelle für zukünftige Zugriffe gespeichert, damit der Ausdruck nicht mehrfach berechnet werden muss.

5.3 WiMa

Das Modul WiMa kann Programme in einer logischen, Prolog-ähnlichen Sprache übersetzen und ausführen. Abbildung 10 zeigt das Beispielprogramm „peano.wima“, in dem die Peano-Arithmetik mit logischen Klauseln nachgebildet wird und die Anfrage „Welche Zahl erfüllt die Anforderung, dass sie die zweite Potenz der Zahl Drei ist?“ beantwortet.

Das Ausführen eines solchen Programms bewirkt nun, dass geprüft wird, ob die Anfrage aus der bestehenden Datenbasis, genannt Fakten, geschlussfolgert werden kann, also die Anfrage gemäß der Datenbasis wahr ist und für welche Variablenbelegungen dies der Fall ist.

Dafür wird bei der Übersetzung Code generiert, der versucht, die gegebenen Fakten mit der jeweiligen Teilanfrage in Übereinstimmung zu bringen („unifizieren“). Die dafür getroffenen Annahmen über die Variablenbelegung werden in einem separaten Speicher

5 Entwicklung

```
1 % 0 is an integer.
2 int(0) .
3 % the successor of an integer is an integer.
4 int(s(M)) :- int(M) .
5
6 % 0 + M = M.
7 sum(0, M, M) .
8 % (N + 1) + M = K + 1.
9 sum(s(N), M, s(K)) :- sum(N, M, K) .
10
11 % 0 * M = 0.
12 prod(0, M, 0) .
13 % (N + 1) * M = N * M + M
14 prod(s(N), M, P) :- prod(N, M, K), sum(K, M, P) .
15
16 % 0^0 = 0.
17 pow(0, 0, 0) .
18 % X^0 = 1.
19 pow(0, s(X), s(0)) .
20 % M^(N+1) = M * M^N.
21 pow(s(N), M, P) :- pow(N, M, K), prod(K, M, P) .
22
23 % 3^2?
24 ?- pow(s(s(0)), s(s(s(0))), X) .
```

Abbildung 10: Peano-Arithmetik.

vorgehalten, um sie, im Falle eines Widerspruchs, also der Unmöglichkeit Anfrage und Fakten zu unifizieren, einfach rückgängig machen zu können.

Der Befehlssatz der WiMa ist von den bisher vorgestellten Befehlssätzen der wahrscheinlich hardware-fernste. Er enthält die komplexesten Instruktionen die deutlich sichtbar an die Quellsprache angepasst sind. Ebenfalls fällt auf, dass die WiMa-VM über ein ungewöhnliches Register („Modus“) verfügt, das keine numerischen Werte annimmt sondern nur die Zustände „read“ und „write“ kennt.

5.4 BfMa

Um die Modularisierung der Software zu testen und die Anwendungsmöglichkeiten der Software zu demonstrieren, wurde ein weiteres Modul für eine ungewöhnliche Programmiersprache entwickelt.

„Brainfuck“ ist eine sogenannte „esoterische“ Programmiersprache, die nicht ernsthaft für die Entwicklung größerer Programme gedacht ist. Das Ziel dieser Sprache ist es, durch möglichst kleinen Befehlsumfang den kleinstmöglichen Übersetzer zu ermöglichen. Das Programmieren in Brainfuck ähnelt dem direkten Programmieren einer Turing-Maschine und ist sehr unübersichtlich, was zum Namen der Sprache führte.

Da diese Sprache über nur acht Befehle verfügt eignet sie sich optimal, um die Verwendung der generischen JumpVM-Funktionen zum Parser- bzw. Compilerbau zu demonstrieren, ohne durch ihre Komplexität abzulenken.

5.5 Organisation des JumpVM Quellcodes

Der JumpVM-Quellcode [4] ist in sieben Verzeichnisse, in Java „Packages“ genannt, aufgeteilt. Diese Packages enthalten:

- *ast*: Datenstrukturen, aus denen die Syntaxbäume der zu übersetzenden Programme zusammengesetzt werden.
- *code*: Die Implementation der Maschinenbefehle der Virtuellen Maschinen.
- *compiler*: Lexer, Parser und Hilfsklassen, die Quelltext in Syntaxbäume und Maschinenbefehle umwandeln.
- *exception*: Hilfsklassen für die Fehlerbehandlung und Präsentation.
- *gui*: Die Grafische Oberfläche des Programms.
- *memory*: Klassen für Speicher und Speicherzellen der Virtuellen Maschinen.
- *vm*: Die Virtuellen Maschinen der einzelnen Module.

Die Packages *ast*, *code* und *compiler* enthalten neben den generischen JumpVM-Klassen jeweils Unterverzeichnisse für die einzelnen Module. Die Klassen in diesen Modul-Verzeichnissen erben von den generischen Klassen und können so Funktionalität auslagern und verallgemeinern.

Im Gegensatz dazu sind beispielsweise die Exception- und Speicherklassen generisch, und werden in unterschiedlichem Ausmaß von allen Modulen genutzt.

6 Bedienung

Abbildung 11 zeigt das Programmfenster nach dem Start der Software. Da diese Software die Funktionalität aller in Kapitel 3 genannten Module vereinigt, ist eine Auswahlmöglichkeit zu sehen, welches Modul gewünscht ist.

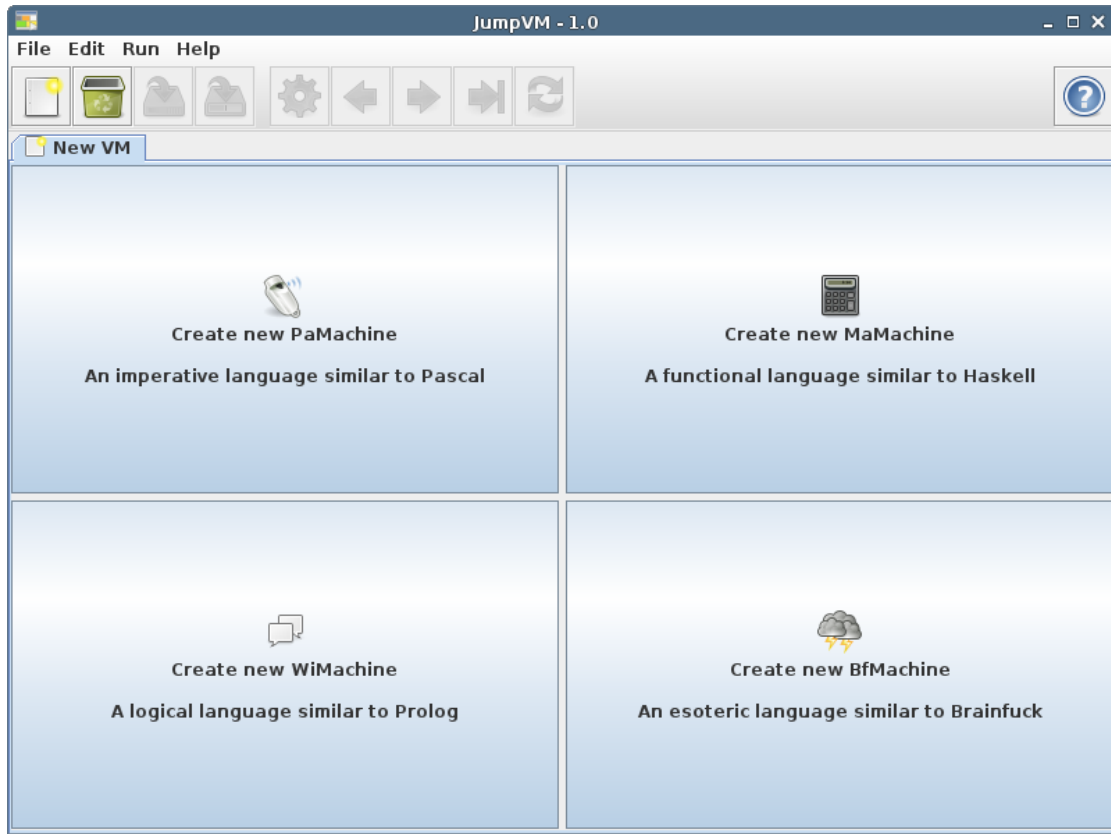



Abbildung 11: JumpVM nach dem Start der Software.

Der Hilfe-Button auf der rechten Seite öffnet das in Abbildung 12 zu sehende Fenster mit der Bedienungsanleitung. In dieser Anleitung findet sich eine generelle Erklärung zum Zweck der Software, eine Schnellstart-Gebrauchsansweisung, detaillierte Erklärungen zur Funktionsweise der Menüs und Elemente der Werkzeugleiste, dem visuellen Aufbau der einzelnen Virtuellen Maschinen und Anmerkungen zu deren jeweiligen Leistungsgrenzen.

Öffnet man eine neue Virtuelle Maschine und gibt Quellcode ein, oder verwendet ein Beispielprogramm aus dem Menü „File → Example“, so kann man es mit einem Klick auf das Zahnradsymbol  oder den Menüpunkt „Compile“ im Menü „Run“ in Instruktionen der aktuellen Virtuellen Maschine übersetzen lassen. Durch einen Klick auf das

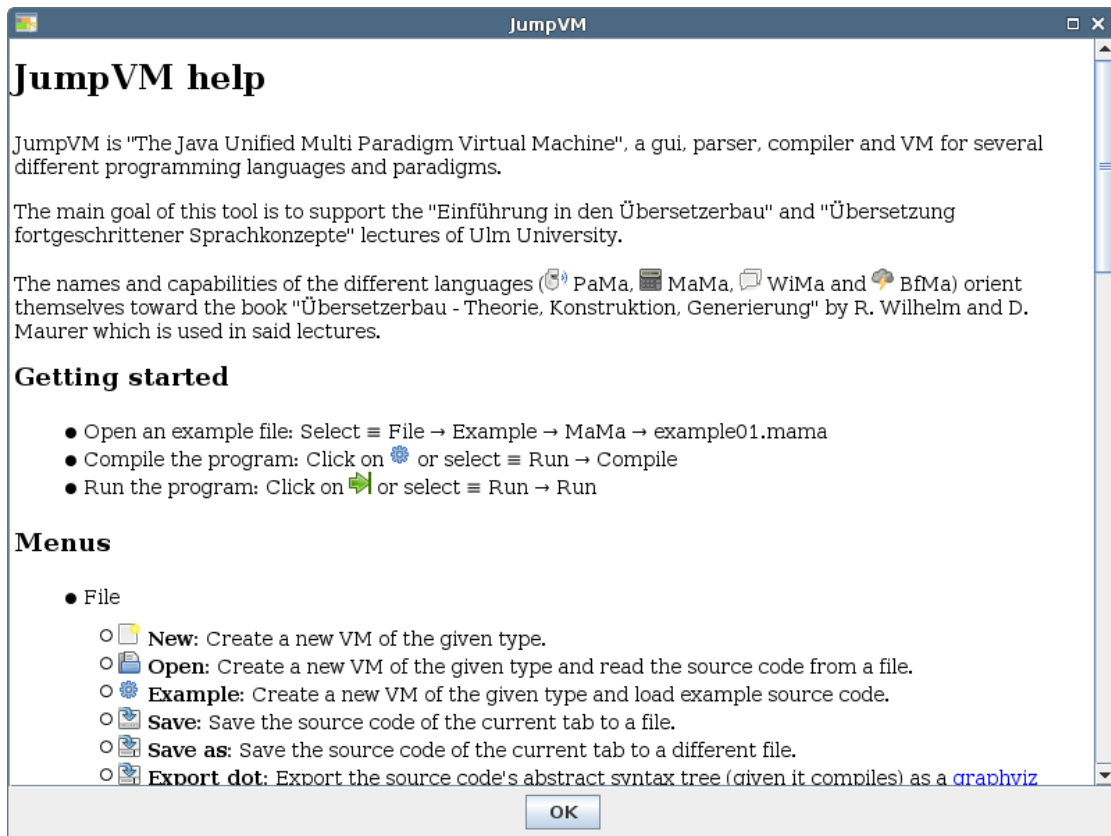


Abbildung 12: Auszug aus der Bedienungsanleitung.

Symbol „Pfeil rechts“ kann das Programm nun schrittweise, durch einen Klick auf das Symbol „Pfeil rechts bis Ende“ kontinuierlich ausgeführt werden. Das Symbol „Pfeil links“ macht einen Ausführungsschritt rückgängig, das Symbol „Pfeile im Kreis“ setzt die Ausführung zum Startzustand zurück. Die vier linken Symbole , , und dienen zum Öffnen einer neuen Virtuellen Maschine, dem Schließen der aktuellen Virtuellen Maschine, dem Speichern des Quellcodes und dem Speichern des Quellcodes unter einem anderen Namen.

Abbildung 13 zeigt ein Beispielprogramm während der Ausführung. Das Feld „Output“ zeigt eventuelle Ausgaben des Programms an. Diese lassen sich mit dem Knopf „clear“ löschen. Darüber befindet sich das Feld „Source“, in dem der Quellcode eingegeben werden kann. Rechts davon wird im Feld „Tree“ der zugehörige Syntax Tree des aktuellen kompilierten Programms angezeigt.

Im oberen Drittel des Fensters findet sich das Feld „Registers“, das die aktuellen Werte der Register der Virtuellen Maschine anzeigt. Daneben stellt das Feld „Program“ den Programmspeicher, also die erzeugten Instruktionen für das gegebene Programm dar.

6 Bedienung

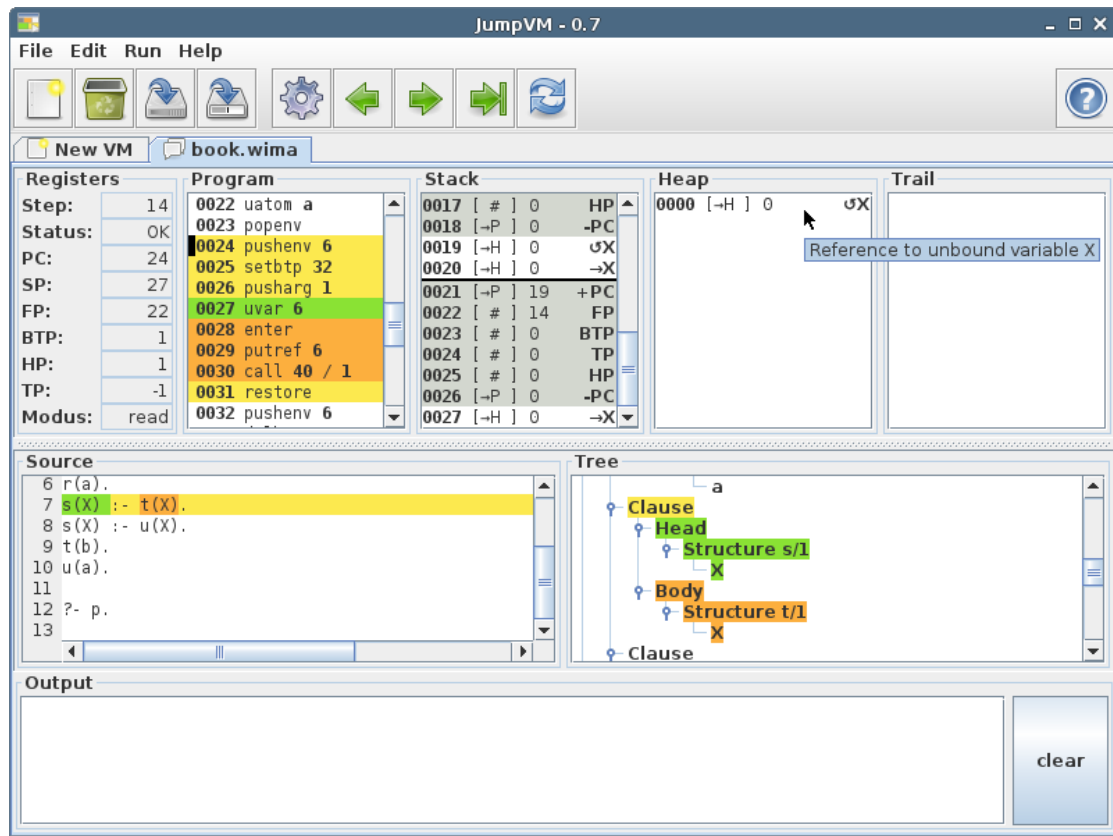


Abbildung 13: Die JumpVM führt ein Programm aus.

Die Felder „Stack“, „Heap“ und „Trail“ bieten Einblick in die verschiedenen Arbeitsspeicher der Virtuellen Maschine.

Die farbliche Markierung einiger Elemente in Gelb, Orange und Grün in den Feldern „Program“, „Source“ und „Tree“ stellen semantische Zusammenhänge zwischen dem Quellcode, der Baumdarstellung und den Maschineninstruktionen dar. Hier wurde beispielsweise auf die Instruktion „pushenv 6“ im Programmspeicher oder „Clause“ im Syntax Tree geklickt, wodurch verdeutlicht wird welche Teile Sourcecode welche Teile Syntax Tree erzeugt und letztendlich welche Instruktionen im Programmcode erzeugt haben.

Die graue Hinterlegung einiger Einträge im „Stack“-Speicher markiert organisatorische Zellen eines Kellerrahmens, dessen Beginn durch einen schwarzen Strich gekennzeichnet ist. Außerdem ist sichtbar, dass alle gespeicherten Werte in allen Speichern typisiert und annotiert sind: Etwa markiert „→ P“ in Zelle 18 im Stack einen Zeiger auf den Programmspeicher und „-PC“ für „Program Counter“ die negative Fortsetzungsadresse des Rahmens. Andere verwendete Symbole sind zum Beispiel ein durchbrochener

Pfeil für einen ungültigen Zeiger und eine Raute für numerische Werte. Führt man mit dem Mauscursor über eine Speicherzelle, so wird die Symbolik der Annotation textuell erklärt.

7 Fazit

Im Wintersemester 2014 wurden im Linux-Rechnerpool der SGI das Betriebssystem auf Fedora Linux 21, Kernel 3.17.4-302 umgestellt. Die in Kapitel 3 erwähnten Probleme in der Wartbarkeit kamen zum Tragen und durch die veränderte Ausführungsumgebung und fehlenden Bibliotheken waren die Programme pmach, mama und wim nicht mehr ausführbar.

Da die Entwicklung der Software JumpVM zu diesem Zeitpunkt nahezu fertiggestellt war, konnte der Übungsbetrieb allerdings ohne Unterbrechung fortgeführt werden. Durch die verwendete Programmiersprache Java und der gewählten Lizenzierung ist nicht zu erwarten, dass JumpVM in der Zukunft vergleichbare Probleme haben wird.

Wie in Kapitel 5 dargelegt, wurden die Schwerpunkte aus Kapitel 4 konsequent umgesetzt und die gesetzten Designziele erreicht und die bisherige Implementierung der Programme in ihrem Funktionsumfang, ihrer Benutzerfreundlichkeit und ihrer Fehler-toleranz übertroffen.

Einige Ideen konnten nicht umgesetzt werden, darunter die Entwicklung eines möglicherweise generischen Backends zur Erzeugung von x86-Assembler bzw. der direkten Erzeugung einer nativen, ausführbaren Datei. Ebenso wurde eine Visualisierung der Stack- und Heap-Speicher verworfen, da bei vielen Zeigern im Speicher die Darstellung sehr unübersichtlich würde und die Auswahl konkret interessanter Zeiger nicht trivial ist.

Eine in der Zukunft mögliche Nachfolgearbeit könnte sich mit der Erweiterung der JumpVM-Compiler um Optimierungsschritte wie Dead code elimination, Constant folding oder Loop unrolling beschäftigen.

Literatur

- [1] *ANTLR*. URL: <http://www.antlr.org/>.
- [2] *Checkstyle*. URL: <http://checkstyle.sourceforge.net/>.
- [3] *Graphviz - Graph Visualization Software*. URL: <http://www.graphviz.org/>.
- [4] *JumpVM*. URL: <https://github.com/twied/jumpvm>.
- [5] *Mark P Jones: Gofer Archive*. URL: <http://web.cecs.pdx.edu/~mpj/goferarc/index.html>.
- [6] *The LLVM Compiler Infrastructure*. URL: <http://llvm.org/>.
- [7] *Tiobe Index*. URL: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.
- [8] Reinhard Wilhelm und Dieter Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer-Verlag Berlin Heidelberg New York, 1997. ISBN: 3-540-61692-6.