Universität Trier - Fachbereich IV - Informatik

Visualisierung Abstrakter Maschinen

Diplomarbeit
bei Prof. Dr. Helmut Seidl

eingereicht von Peter Ziewer

Trier, 5. März 2001

Erklärung zur Diplomarbeit

Hiermit erkläre ich, dass ich die Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Diplomarbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher auch nicht veröffentlicht.

Trier, 5. März 2001 Peter Ziewer

Inhaltsverzeichnis

1 Einleitung

Wie der Titel schon zeigt, beschäftigt sich diese Diplomarbeit mit dem Thema Visualisierung. ,,Die Leistung moderner Computersysteme hat die Entwicklung rechnergestützter Darstellungen komplizierter Sachverhalte in den letzten Jahren stark beschleunigt. So sind viele Bereiche der Industrie und Wirtschaft in hohem Maße von rechnergestützten Visualisierungen komplexer Sachverhalte abhängig, nicht zuletzt deshalb, weil Computersysteme Informationen als abstrakte Symbole repräsentieren, wobei ihre Repräsentationsformen in der Regel darauf ausgelegt sind, Berechnungen und Analyseoperationen zu machen und den Informationzugriff zu erleichtern.

So ist beispielsweise für einen Aktionär der Kursverlauf eines Wertpapieres von großem Interesse, um ihn bei einer Kaufs- bzw. Verkaufsentscheidung zu unterstützen. Die hier am häufigsten verwendete Informationsquelle ist der Chart, der die historische Entwicklung eines Wertpapieres in Form von Kurven veranschaulicht. Ein erfahrener Chartanalyst erstellt aus diesen Informationen und der Anwendung geeigneter Indikatoren sowie anderen branchenspezifischen Informationen, eine Entwicklungsprognose für das einzelne Wertpapier, die dann je nach Erfahrung des Analysten und mit ein wenig Glück des Anlegers auch eintreten."[Zlot2000]

Diese Arbeit widmet sich einem speziellen Bereich der Software-Visualisierung, der Darstellung Abstrakter Maschinen. Diese Maschinen finden vor allem beim Übersetzerbau Verwendung und ihre Visualisierung dient als Hilfsmittel im Bereich der Lehre, sowie der Verbesserung vorhandener und der Entwicklung neuer abstrakter Maschinen. Die Visualisierung findet vor allem als dynamische Darstellung in Form von Animation statt und ihre Generierung kann durch geeignete Hilfsmittel deutlich vereinfacht werden.

In Kapitel 2 wird zunächst die Zielsetzung dieser Diplomarbeit aufgezeigt.

Das Kapitel 3 dient als einführende Beschreibung des Konzepts der Abstrakten Maschinen. Der Aufbau einer solchen Maschine wird erklärt und Anwendungsbereiche des Konzepts werden genannt. Anschließend wird diskutiert, was bei einer Visualisierung zu beachten ist und wie sich abstrakte Maschinen visualisieren lassen (Kapitel 4).

In Kapitel 5 wird das an der Universität des Saarlandes in Rahmen des **Ganimal**-Projektes [GANIMAL] entworfene System zur Visualisierung abstrakter Maschinen, **GANIMAM**, beschrieben, sowie dessen Vor- und Nachteile diskutiert, um daraus Aufschlüsse zur Entwicklung eines eigenen Visualisierungs-Pakets zu erlangen.

Der Kern der Diplomarbeit ist das von mir entwickelte und implementierte System zur **V**isualisierung **A**bstrakter **M**aschinen, das **VAM**-Paket. Seine Konzepte und Komponenten, sowie seine Verwendung werden in Kapital 6 erklärt. In Kapitel 7 erfolgt die Beschreibung der Implementierung der beiden abstrakten Maschinen **CMa** und **MaMa**. Diese Beispiel-Implementierungen sollen die Möglichkeiten des **VAM**-Systems zeigen.

Das letzte Kapitel widmet sich möglichen Erweiterungen und Verbesserungen des **VAM**-Pakets.

Der Anhang enthält eine komplette Auflistung der Klassen des **VAM**-Pakets und ihrer Methoden mit einer knappen Beschreibung der jeweiligen Funktion und der Parameter.

2

Zielsetzung

Das Ziel dieser Diplomarbeit ist, die Arbeitsweise abstrakter Maschinen zu veranschaulichen und zu verdeutlichen. Dazu soll ein System zur Visualisierung abstrakter Maschinen entworfen, implementiert und vorgestellt werden. Anhand von Beispiel-Maschinen aus der Vorlesung ,,Abstrakte Maschinen" [Seidl2000] soll gezeigt werden, dass das von mir generierte System auch seine Aufgaben erfüllt.

Es müssen die von den abstrakten Maschinen verwendeten Datenstrukturen graphisch dargestellt werden und die auf diese Strukturen zugreifenden Befehle mittels Animation visualisiert werden.

Dieses Visualisierungs-System soll Hilfsmittel und Datenstrukturen zur Verfügung stellen, mit denen eine möglichst einfache Implementierung abstrakter Maschinen beziehungsweise die Erweiterung des Befehlssatzes einer vorhandenen Maschine ermöglicht wird. Die Erzeugung der zur Veranschaulichung dienenden optischen Darstellung und der Animationen sollte weitestgehend von meinem System automatisiert werden. Die Benutzung einer Datenstruktur, die eine Visualisierung hervorruft, soll sich möglichst nicht unterscheiden von der normalen Verwendung dieser Struktur. So sollen die, durch einen Befehl einer implementierten abstrakten Maschine erzeugten, Speicherzugriffe neben der erwarteten Funktionalität, die darin besteht Speicherinhalte zu lesen, schreiben oder zu kopieren, auch die entsprechenden Vorgänge visualisieren, d.h. die betroffenen Speicherzellen hervorheben, animiert bewegen oder ihre Inhalte ändern.

Es sei hier angemerkt, dass diese Automatisierung in einigen Punkten nicht ohne Kontextinformationen auskommen kann. So können Zeiger (*Pointer*) oft nicht als solche erkannt werden, da die hierfür erforderlichen Typinformationen nur dem

Übersetzer (*Compiler*) bekannt sind, aber in dem übersetzten Programmcode, welcher der abstrakten Maschine als Eingabe dient, nicht mehr zur Verfügung stehen. Außerdem lassen sich oft Abläufe durch gezielte Visualisierungsmassnahmen, wie der Verwendung unterschiedlicher Farben und Formen, verdeutlichen. Das Einbringen dieser Zusatzinformationen ist Aufgabe des Animationsentwicklers der eine abstrakte Maschine für das Visualisierungssystem umsetzt.

Um diese Ziele zu erreichen, habe ich das **VAM**-System (**V**isualisierung **A**bstrakter **M**aschinen) entwickelt und implementiert. Es bietet Datenstrukturen, deren Schnittstellen der Funktionalität der entsprechenden Struktur innerhalb der Definition einer abstrakten Maschine möglichst entspricht, aber zusätzlich die Visualisierung ermöglicht und weitestgehend automatisiert. Desweiteren stelle ich dem Benutzer eine Umgebung zur Verfügung, die die graphische Darstellung ermöglicht. In dieser soll die zu visualisierende Maschine sowie das darzustellende Programm für diese Maschine ausgewählt werden. Außerdem bietet die Umgebung Einstellmöglichkeiten zum Ablauf der Animation, wie etwa die Regelung der Geschwindigkeit oder eine Start- und Pausefunktion.

3

Abstrakte Maschinen

In diesem Kapitel wird erklärt was eine abstrakte Maschine ist, aus welchen Komponenten sie besteht und wie sie arbeitet. Auf die Möglichkeiten der Visualisierung dieser Maschinen wird im nächsten Kapitel eingegangen.

3.1 Was ist eine Abstrakte Maschine ?

Die Leistungsfähigkeit der Hardware hat sich in den letzten Jahren deutlich erhöht. Heutige Rechner bieten ein Vielfaches an Rechenleistung und Speicherkapazität, sowie eine größere Anzahl an Geräten die verwendet werden können. Die Bussysteme und Protokolle zwischen den einzelnen Komponenten sind zwar leistungsfähiger, aber oft auch komplizierter geworden. Es wurden Prozessoren mit leistungsteigernden Spezialbefehlen entwickelt oder es finden gar mehrere Prozessoren innerhalb eines Rechners Verwendung. Die Prozessoren bieten eine Unmenge an Befehlen und Addressierungsmöglichkeiten. Dies erschwert die direkte Programmierung und macht es in der Praxis unmöglich, große Projekte direkt auf der Hardware zu realisieren.

Eine abstrakte Maschine ist eine idealisierte Hardware, für die sich einerseits leicht Code erzeugen lässt, die andererseits aber auch leicht auf realer Hardware implementierbar ist. Sie verwendet weniger Befehle mit vereinfachter Adressierung und ist entkoppelt von der Ausnutzung spezieller Hardware-Features. Häufig werden abstrakte Maschinen als plattformunabhängige Zwischenarchitekturen bei der Übersetzung höherer Programmiersprachen verwendet. Die Anweisungen dieser Maschine sind

für die Übersetzung einer bestimmten Quellsprache oder für Quellsprachen desselben Sprachparadigmas (imperativ, funktional, logisch, objektorientiert) maßgeschneidert. Diese Aufteilung in zwei (oder mehr) Ebenen findet in der Informatik häufig Verwendung, da sie es ermöglicht, die Ebenen unabhängig voneinander zu realisieren und auszutauschen.

Bei der Übersetzung von Programmen wird aus dem in einer Hochsprache wie C oder Java gegebenen Quellcode Code für eine abstrakte Maschine erzeugt. Dieser kann dann entweder interpretiert ausgeführt werden oder durch einen Übersetzer in konkreten Maschinencode umgesetzt werden. Die Portierung auf neue Zielarchitekturen vereinfacht sich, da nur die Maschine, die Programm-Code ausführt, portiert werden muss, aber nicht der auszuführende Code. Der Übersetzer (*Compiler*) wird flexibler, da mehrere Sprachen durch unterschiedliche Frontends auf eine abstrakte Maschine abgebildet werden können oder sich einzelne Teile des Übersetzers neu implementieren und verbessern lassen, ohne das Gesamtpaket zu verändern.

Das bekannteste Beispiel einer abstrakten Maschine ist wohl die **JVM** (**J***ava* **V***irtual* **M***achine*) der Sprache Java. Ein Java-Programm wird für die Ausführung auf der **JVM** generiert und kann auf jeder Architektur ausgeführt werden, für die eine Implementierung der abstrakten Maschine existiert. Dies ermöglicht die Erstellung plattformunabhängigen Codes, was sicherlich mit zu dem Erfolg der Programmiersprache Java beigetragen hat.

Beispiele abstrakter Maschinen:

- die JVM (**J***ava* **V***irtual* **M***achine*) für Java
- die P–Maschine von Pascal
- die bei Prolog verwendete WAM (**W***arren* **A***bstract* **M***achine*)
- die STGM bei den funktionalen Sprachen SML und Haskell
- oder auch der für Smalltalk verwendete Bytecode

Einfache abstrakte Maschinen werden in [Seidl2000] beschrieben :

- C $\longrightarrow$ CMa (imperativ)
- PuF $\longrightarrow$ MaMa (funktional)
- PuP $\longrightarrow$ WiM (logikbasiert)

3.2 Der Aufbau einer abstrakten Maschine

Die Architektur einer abstrakten Maschine stellt einen Satz von Instruktionen zur Verfügung, welcher auf der abstrakten Hardware ausgeführt wird. Diese Hardware wird als Menge von Datenstrukturen aufgefasst.

Die kleinste Struktur ist eine einzelne Speicherzelle, die Werte wie zum Beispiel Zahlen, Buchstaben oder Adressen von anderen Zellen speichern kann. Desweiteren gibt es Speicherzellen mit speziellen Bedeutungen — die Register. Diese werden häufig verwendet, um andere Datenstrukturen zu verwalten (z.B. *Stack Pointer*, *New Pointer*), oder dienen als Verweise auf wichtige Komponenten (z.B. *Programm Counter*, *Global Pointer*). Weitere Datenstrukturen werden aus Zellen zusammengesetzt, wie etwa der Keller (*Stack*) und die Halde (*Heap*) oder einzelne Objekte innerhalb der Halde (z.B. Vektoren).

Die Instruktionen greifen auf Datenstrukturen zu, führen Operationen auf ihnen aus und verändern sie. Eine Menge von Instruktionen ergibt ein Programm, welches im Code-Speicher liegt und in der Reihenfolge, die der Kontrollfluss vorgibt, abgearbeitet wird.

3.3 Ausführung von Programmen

Der Ablauf zur Ausführung von Programmen ist bei allen abstrakten Maschinen der gleiche. Im *Code Memory* `C[]` wird das auszuführende Programm gespeichert. Das Register *Programm Counter* `PC` gibt die Position des nächsten Befehls innerhalb des Programmspeichers an. Die Maschine lädt die auszuführende Instruktion aus `C[PC]` in ein *Instruction Register* `IR` und führt sie dann aus. Der Maschinen-Zyklus sieht also wie folgt aus:

```
while(true)
 {
   IR = C[PC];
   PC++;
   execute(IR);
 }
```

Hierbei muss der `PC` vor der Ausführung der Instruktion erhöht werden, da Sprung-Befehle das Register eventuell ändern. Der Maschinen-Zyklus wird durch die Instruktion `HALT` beendet. Innerhalb meiner Implementierung wird der Zyklus durch einen nicht gültigen `PC` verlassen. Eine Instruktion, die das Programm terminieren soll, setzt hierzu den `PC` auf `-1`.

4

Visualisierung

Visualisierung ist die Transformation von Information, um sie dem Anwender in einer geeigneten Sicht in einer möglichst nützlichen Form zu präsentieren. Es sollen also (in der Regel nicht sichtbare) Informationen in optisch darstellbare Komponenten umgesetzt werden.

Abbildung 4.1: Visualisierung (Uli Stein)

4.1 Warum Visualisierung ?

Ausgehend von der Erkenntnis, dass der Mensch bildliche Eindrücke besonders effizient verarbeitet und konkrete Objekte besser erfassen kann als abstrakte Beschreibungen, wurde nach Möglichkeiten gesucht, die unsichtbaren Vorgänge beim Ablauf von Software sichtbar zu machen. Dabei kann eine Visualisierung in unterschiedlichen Gebieten und für unterschiedliche Zielsetzungen erfolgen:

- (statische) graphische Repräsentation von Datenstrukturen
  (z.B. Bäume, Listen)
- Algorithmenanimation
- Algorithmenerklärung
- Visualisierung der Daten und Abläufe in Anwendungen
- Visuelles Debuggen
- Visualisierung des Ablaufs nebenläufiger Programme
- Lernsoftware im Bereich Informatik
- Graphlayoutalgorithmen in der Software Visualisierung
- Verwendung von Diagrammen in der Softwarevisualisierung
  (z.B. UML, *Concept Maps*)
- Softwarevisualisierung im Internet

Zur Generierung einer Visualisierung ist es wichtig zu wissen, was visualisiert werden soll und für wen dies geschieht. Der Betrachter sollte nicht von einer Unmenge an Visualiserungskomponenten überflutet werden und dadurch die wichtigen Elemente übersehen. Die Abbildung des gesamten Hauptspeichers eines Rechners ist zur Algorithmen-Visualisierung oder zur Darstellung einer Datenstruktur sicherlich wenig geeignet. Soll hingegen die Auslastung der Systemresourcen visualisiert werden, so macht die Darstellung des gesamten Speichers eher Sinn. Von Interesse ist dann, ob eine Zelle belegt ist, aber nicht der Inhalt der einzelnen Speicherzelle selbst.

Es sollte also nur das in der Visualisierung erscheinen, was von Interesse für den Betrachter ist und bei einer dynamischen Visualisierung sollten die ,,wichtigen" Elemente die Aufmerksamkeit des Betrachters erlangen. Dies kann zum Beispiel durch farbliche Hervorhebung oder durch Bewegung erfolgen.

4.2 Visualisierung abstrakter Maschinen

Zur Visualisierung abstrakter Maschinen stellt sich die Frage, was zu visualisieren ist. Dies sind die Datenstrukturen und die durch die Ausführung eines Programms gegebenen Operationen auf ihnen. Ein Paket zur Visualisierung sollte also die Möglichkeit bieten, die Objekte der Maschine graphisch darzustellen und dem Programmfluß folgend die Operationen zu visualisieren. Folgende Komponenten sollten visualisiert werden:

- Auszuführender Programm-Code
- Register(-inhalte)
- Speicherinhalte
- Veränderung der Inhalte
- Datenstrukturen innerhalb des Speichers
  (z.B. Keller, Vektoren)
- Zusammenhänge zwischen Datenstrukturen
  (z.B. Zeiger)

Zur Generierung der Visualisierung kann ein vorhandenes Laufzeitsystem abstrakter Maschinen an einigen Stellen um Visualisierungselemente erweitert werden oder aber es wird eine neue Umgebung geschaffen, die in der Lage ist, den abstrakten Maschinen-Code auszuführen und zu visualisieren. Das Laufzeitsystem muss neben den Datenstrukturen der Maschine auch diejenigen der graphischen Repräsentation verwalten. Instruktionen lösen durch ihre Zugriffe auf die Strukturen der Maschine neben der gewünschten Manipulation bzw. Abfrage von Daten automatisch die Generierung bzw. Entfernung von Visualisierungskomponenten aus und erzeugen Animationen.

5

GANIMAM

GANIMAM steht für ,,**G**enerierung interaktiver **ANIM**ationen von **A**bstrakten **M**aschinen" und wurde an der Universität des Saarlandes entwickelt. **GANIMAM** soll als webbasierten Generator für interaktive Animationen abstrakter Maschinen dienen.

Im Folgenden wird beschreiben, wie **GANIMAM** verwendet werden kann und was das System generiert. Anschließend möchte ich Vor- und Nachteile dieses Systems aufzeigen, um daraus Schlussfolgerungen zur Entwicklung meines Visualisierungs-Paketes zu ziehen.

Dieses Kapitel enthält (übersetzte) Teile der **GANIMAM**-Dokumentation [GANIMAL].

5.1 Technischer Überblick

GANIMAM kann über die Webseite des GANIMAL-Projekts [GANIMAL] aufgerufen werden. Ein Java-Applet erlaubt dem Benutzer eine vordefinierte Spezifikation einer abstrakten Maschine auszuwählen, bei Bedarf zu ändern oder eine neue Maschine zu definieren. Diese Spezifikation einer abstrakten Maschine wird dann zum Server geschickt. Ein CGI-Skript auf dem Server erzeugt Java-Quellcode, der durch einen Kompiler in Klassendateien übersetzt wird, die dann dynamisch in das laufende Applet geladen werden. In Kombination mit den Klassen des Basispakets von **GANIMAM** ergeben die Klassendateien ein interaktives Java-Applet.

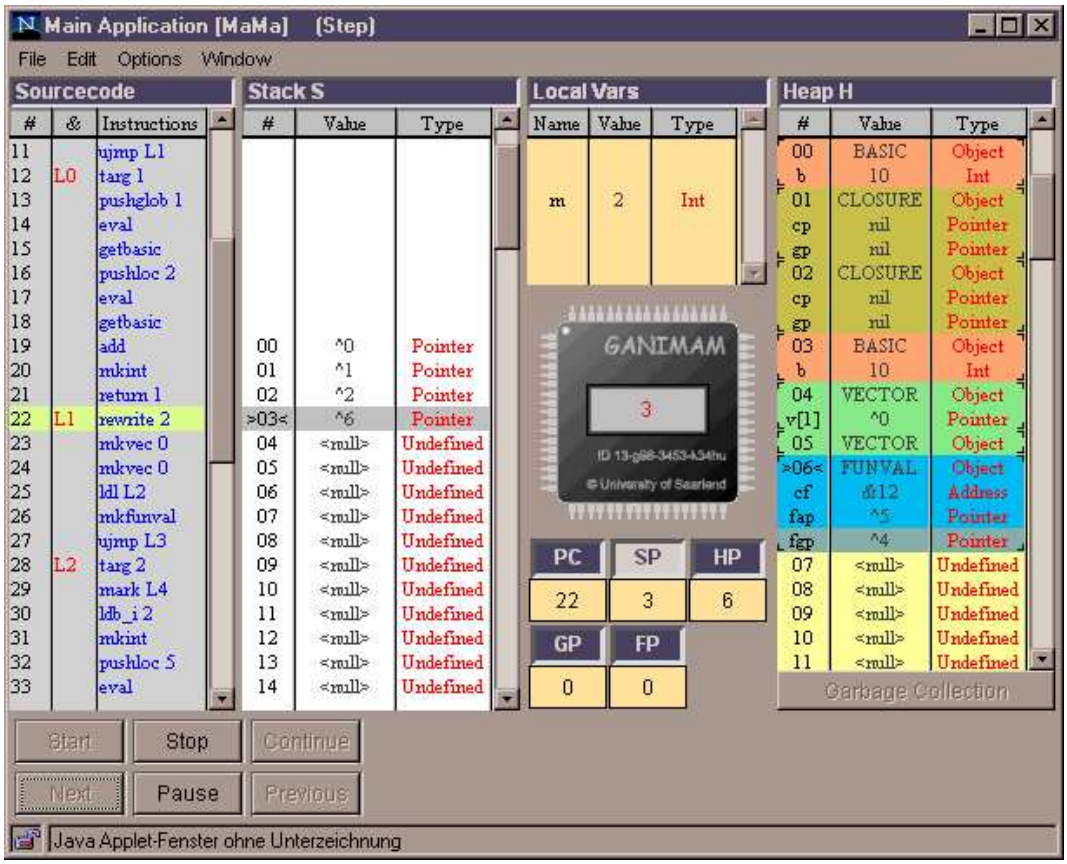


Abbildung 5.1: Ganimam Screenshot

Der Benutzer kann dann Maschinenprogramme eingeben und die animierte Ausführung seines Maschinenprogramms steuern. Abstrakte Maschinen können eine unterschiedliche Anzahl von Registern, Kellern und Halden haben, daher wird für jede generierte Maschine ein automatisches Layout generiert. Das automatische Layout gruppiert die verschiedenen Speicherarten um den Akkumulator, eine konzeptionelle Recheneinheit, herum. Programmcode und Keller werden links, die Halde rechts, lokale Variablen über und Register unter dem Akkumulator plaziert. Mit dem Akkumulator ist ein Akkumulatorfenster verbunden, das den Ausdruck anzeigt der gerade im Akkumulator berechnet wird, sowie die Definition der Instruktion bzw. Funktion, die gerade ausgeführt wird. Das Layout der verschiedenen Teile der visualisierten abstrakten Maschine kann durch den Benutzer geändert und individuell angepasst werden

Durch einen zweifachen Mausklick mit der rechten Maustaste auf eine Instruktion im Programmcode wird die Definition der Instruktion in das Akkumulatorfenster geladen. Ein zweifacher Mausklick mit der linken Maustaste auf eine Instruktion im Programmcode setzt den Wert des Programmzählers auf die Adresse der Instruktion,

d.h. die Ausführung des Programms wird an dieser Stelle fortgesetzt. Ein Klick auf eine Zelle des Kellers, der Halde oder auf ein Register öffnet ein Fenster. In diesem Fenster kann der Benutzer den Wert und den Typ, bei Registern nur den Wert ändern.

GANIMAM bietet noch weitere Arten der Interaktion. Der Benutzer kann Spezifikationen einer abstrakten Maschine eingeben oder ändern. Nach der Erzeugung der Implementierung der abstrakten Maschine kann der Benutzer Programme in der Sprache der abstrakten Maschine eingeben, diese Schritt für Schritt ausführen und den Inhalt jedes Registers und jeder Speicherzelle inspizieren. Während der Ausführung einer Instruktion zeigt eine Animation den Fluss der Information von Registern oder Speicherzellen in den Akkumulator und vom Akkumulator zurück zu Registern oder Speicherzellen. Die Berechnung, die im Akkumulator durchgeführt wird, wird in einem separaten Fenster angezeigt.

5.2 Die Spezifikations-Sprache

GANIMAM bietet eine Spezifikations-Sprache für abstrakte Maschinen, die sich anlehnt an die Notation, die im Buch zum Übersetzerbau von Wilhelm und Maurer [Wilh1997] verwendet wird, um imperative, funktionale und logische Maschinen zu beschreiben. Diese Spezifikations-Sprache ermöglicht die Steuerung des Kontrollflusses durch Zuweisungen, Ausdrücke, Bedingungen und Schleifen.

Eine abstrakte Maschine kann deklariert werden durch Angabe von Keller- und Halden-Speicher, Registern, sowie der Definition der benötigten Befehle. Zusätzlich zu den vordefinierten Datentypen können eigene, komplexere Strukturen definiert werden, die aber auch in einer Speicherzelle abgelegt werden. Außerdem können neben den Befehlen der abstrakten Maschine noch Hilfsfunktionen definiert werden.

Ein Beispiel:

```
// declarations
STACK     S[100] with SP=-1;
HEAP      H[100] with HP=-1;
REGISTER  PC,MP,EP;

// specification of auxiliary functions
fun int base (int p, int dv) =  // compute static predecessor
  if (p=0) then                 // of the current procedure
    return dv;
  else
    return base(p-1,S[dv+1]);
  fi;
nuf
```

```
// specification of an instruction
def mst (int p) =      // create a procedure frame
  S[SP+2]:=base(p,MP); // pointer to frame of static predecessor
  S[SP+3]:=MP;         // pointer to frame of dynamic predecessor
  S[SP+4]:=EP;         // max. depth to evaluate expressions
  SP:=SP+5;            // procedure frame has at least 5 cells
fed
```

5.3 Verwendbarkeit von GANIMAM

Durch **GANIMAM** lassen sich abstrakte Sprachen definieren und visualisieren, als Applet implementiert bietet das System einen web-basierten Zugriff und eine Reihe von Interaktionsmöglichkeiten. Das Layout wird den Gegebenheiten der jeweiligen Maschinen-Definition angepasst. Der Programmcode wird mit den verwendeten symbolischen Sprungzielen (*Labels*) angezeigt und die Zeile markiert, die gerade ausgeführt wird. Die Visualisierung der Register ist übersichtlich, die tabellarischen Darstellungen von Keller und Halde enthalten allerdings auch Einträge ungenutzter Speicherzellen. Die Typen der in den Speicherzellen enthaltenen Werte werden genannt und unterschiedliche Halden-Objekte farblich voneinander getrennt.

Die Granularität der Visualisierung ist sehr klein. Die Visualisierung findet nicht auf Ebene der Befehle der abstrakten Machine statt, sondern vielmehr auf der der Spezifikations-Sprache. Das Inkrementieren des Kellerpegels bewirkt das animierte Laden des aktuellen Werts aus dem Register `SP` in den Akkumulator, einer Operation im Akkumulatorfenster und anschließend das animierte Setzen des neuen Registerinhalts. Es stellt sich die Frage, ob der Betrachter nicht eher an der Arbeitsweise eines Befehls interessiert ist, als an Operationen aus denen der Befehl zusammengesetzt ist. Die Abarbeitung eines nur kurzen Maschinen-Programms ist bei dieser Granularität der Visualisierung sehr langwierig und die Aussagekraft schmälert sich durch ständige Wiederholungen, da zum Beispiel ,,`SP++;`" oder ,,`SP--;`" in nahezu jedem Befehl vorkommen.

Die Visualisierung setzt meiner Meinung nach zu wenig auf das Konzept der Animation. Obwohl der Begriff **ANIM**ation Teil der Bezeichnung G**ANIM**AM ist, beschränkt sich das System darauf, die Bewegung von Werten zwischen den einzelnen Speicherbereichen und dem Akkumulator zu animieren. Die Darstellung der Bewegung erfolgt dabei nicht zwischen der Position des bewegten Wertes im Speicher und der des Akkumulators, sondern zwischen der Mitte des Speicherbereichs zu dem der Wert gehört und dem Akkumulator. Die Animation ist also für alle Elemente eines zusammengehörigen Speicherbereichs identisch, natürlich bis auf die Darstellung des Werts selbst. Wird ein Wert vom Akkumulator im Keller abgelegt, so lässt sich an der Animation nur erkennen, dass er im Keller gespeichert wird,

aber nicht in welcher Zelle des Kellers. Dies zeigt sich nur daran, dass die Belegung einer Keller-Zelle sich ändert. Einzig bei den Registern erfolgt die Animation direkt zwischen der Position des Registers und dem Akkumulator.

Ist ein Wert erst einmal in den Akkumulator geladen worden, so findet die weitere Visualisierung der Ausführung des Befehls in einem eigenen Fenster statt und nicht an dem Punkt, an dem die Animation endete. Der Blick des Betrachters muss zwischen den Fenstern hin und her springen.

Desweiteren erfolgt zur Visualisierung von Referenzen lediglich die Darstellung eines Wertes, der der Adresse des Zielobjekts entspricht. Zwischen den beiden Komponenten besteht keine optische Verbindung, wie dies ein Pfeil sein könnte.

Vorteilhaft ist die automatische Generierung einer abstrakten Maschine durch Angabe ihrer Definition in einer Spezifikations-Sprache und die für einige Sprachen gelieferten Übersetzer.

6

Das VAM-System

Das im Rahmen dieser Diplomarbeit von mir entwickelte und implementierte System zur **V**isualisierung **A**bstrakter **M**aschinen, **VAM** , ermöglicht die einfache Implementierung abstrakter Maschinen mit dem Ziel, die Visualisierung des Ablaufs von Programmen für diese Maschinen zu automatisieren.

Zur Laufzeit wird eine implementierte abstrakte Maschine und ein Programm in der Sprache dieser Maschine angegeben. Das Programm wird dann von der Maschine abgearbeitet und löst durch Zugriffe auf die durch das System zur Verfügung gestellten Datenstrukturen die Visualisierung aus.

Das **VAM**-Paket stellt einfache Komponenten, wie Zellen oder Zeiger, mit einer entsprechenden graphischen Repräsentation zur Verfügung. Diese Objekte werden von komplexeren Datenstrukturen, wie der Speicherverwaltung, verwendet, um die Visualisierung zu generieren.

Es folgt nun zunächst die Beschreibung der Visualisierungs-Komponenten und der Datenstrukturen des Systems. Anschließend wird erklärt, wie abstrakte Maschinen implementiert werden müssen, um eine Visualisierung zu ermöglichen. Am Ende dieses Kapitels wird die Bedienung der Visualisierungsoberfläche erklärt.

6.1 Komponenten der Visualisierung

Hier folgt eine Beschreibung der zur Verfügung stehenden Komponenten zur graphischen Repräsentation. Anhand einiger Codefragmente wird beispielhaft deren Verwendung gezeigt. Die von mir bereitgestellten Visualisierungsmöglichkeiten beruhen weitestgehend auf den Java-eigenen Objekten zur Erzeugung graphischer Oberflächen (AWT und Swing). Meine Objekte sind Unterklassen von `JLabel` bzw.

`JPanel`. Neben diesen können beliebige JComponent-Objekte in die Visualisierung eingefügt werden.

Komponenten werden durch die Methoden `addVisualization(`*component*`)` und `removeVisualization(`*component*`)` der Klasse `ViewMachine` in die Visualisierung eingefügt bzw. aus ihr entfernt. Für Zellen, Register und Zeiger müssen diese Funktionen nicht explizit aufgerufen werden, da diese Objekte dies selbst übernehmen.

Die Komponenten dienen zur Erzeugung einer Visualisierung. In der Regel erfolgt jedoch kein direkter Zugriff auf Einzelkomponenten, sondern ein indirekter über die Speicherverwaltung des **VAM**-Systems, da diese die Generierung der Visualisierung weitestgehend automatisiert.

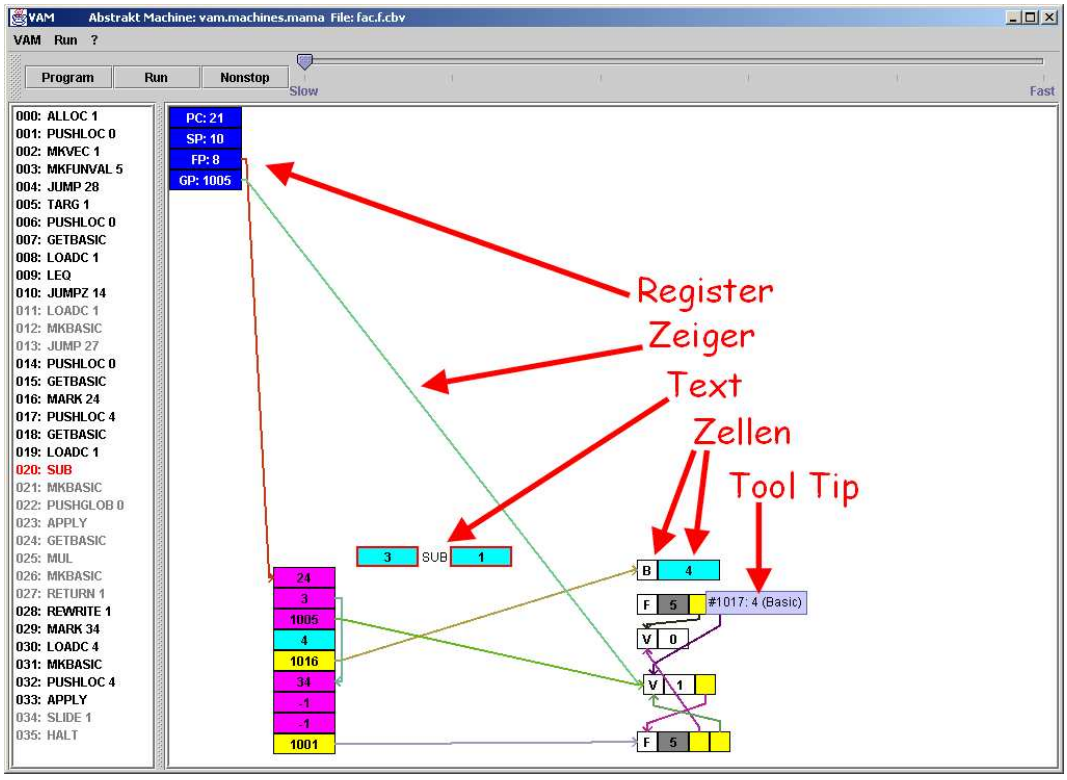


Abbildung 6.1: Komponenten der Visualisierung

6.1.1 Zellen (Cell)

Zellen symbolisieren Speicherzellen, deren jeweiliger Wert gesetzt und gelesen werden kann. Sie werden als farbige Rechtecke dargestellt und können mit einer Animation über den Darstellungsbereich verschoben sowie ein- und ausgeblendet werden. Innerhalb des Rechtecks wird der Wert jeder Zelle als Text angezeigt. Zu den Zellen kann eine Typangabe erfolgen. Diese Angabe dient zur Festlegung der graphischen

Repräsentation. Zellen eines Typs erhalten die gleiche Farbe und Größe. Im Allgemeinen sollte die Bezeichnung für den Typ der Bedeutung des Wertes für die jeweilige abstrakte Maschine entsprechen, z.B. „*int*" oder „*Pointer*". Die Klasse übernimmt jedoch keine Typ-Überprüfung. Dies ist die Aufgabe der Implementierung der jeweiligen Maschine. Intern werden Werte immer als `String` gespeichert.

Desweiteren können Zellen als Referenzen markiert werden. Diese spezielle Markierung von Zellen bewirkt die Erzeugung einer graphischen Repräsentation eines Zeigers (Kap. 6.1.3) auf die referenzierte Speicheradresse oder vielmehr auf die Zelle, die wiederum diese Speicheradresse repäsentiert.

Animation

Zur Erzeugung einer Animation können Zellen bewegt werden, unterschiedliche farbliche Hervorhebungen erhalten, sowie ein- und ausgeblendet werden. Durch

```
String value = "42";
String type = "int";
Point location = new Point(111,99);
boolean visibility = false;
Cell cell = new Cell( value, type, location, visibility );
cell.highlight( true );
cell.fadeIn();
cell.sync();
cell.moveTo( 47, 11 );
cell.sync();
cell.highlight( false );
```

wird eine neue zunächst nicht sichtbare Zelle mit dem Inhalt `42` an den Koordinaten `(111,99)` angelegt. Ihr Typ ist „*int*".

Um die Aufmerksamkeit des Betrachter zu erlangen, wird nun zuerst mit `highlight(true)` die Zelle durch einen roten Rahmen von den übrigen Zellen hervorgehoben.

Dann erfolgt durch `fadeIn()` die Visualisierung der Zelle. Sie wird eingeblendet indem eine Erhöhung der Sättigung der Farbe vom zuvor durchsichtigen Zustand in mehreren Stufen bis hin zur vollen Sättigung erfolgt.

`moveTo(47,11)` generiert eine Animation, in der das Rechteck der Zelle zu den Koordinaten (47,11) wandert. Alternativ kann mit `moveBy(10,20)` auch ohne Kenntnis der absoluten Koordinaten eine Verschiebung um die angegebene Distanz erfolgen.

Die letzte Zeile löscht die anfangs gesetzte Hervorhebung der Zelle wieder.

Die Aufrufe von `sync()` blockieren die Ausführung bis zur Vollendung aller diese Zelle betreffenden Animationen. Sollen mehrere Zellen gleichzeitig animiert werden,

so werden erst alle Komponenten der Animation gestartet und erst dann erfolgt eine Synchronisation:

```
cell1.fadeIn();
cell2.moveBy( 100, 0 );
cell3.moveBy( 100, 0 );
cell1.sync();
cell2.sync();
cell3.sync();
cell1.moveTo( 50, 100 );
cell2.moveTo( 50, 150 );
cell3.fadeOut();
Cell.syncAll();
```

Die statische Methode `syncAll()` blockiert bis zur Synchronisation aller Zellen, also der vollständigen Abarbeitung aller Animationen.

Eine weitere Möglichkeit zur Animation mehrerer Zellen bietet die Klasse `CellGroup`, die Zellen zu einer Gruppierung zusammenfasst und dann als ganzes visualisiert (siehe Gruppierung Kap. 6.1.5).

Werte

Der Wert einer Zelle ist ein `String`, der den Inhalt einer Speicherzelle repräsentiert. Mit `setValue(`*`newValue`*`)` kann er neu gesetzt werden und `getValue()` liefert den aktuellen Wert. Die Interpretation des Wert-Strings als Datentyp einer bestimmten abstrakten Maschine obliegt dem jeweiligen Implementierer dieser speziellen Maschine, da Datentypen nicht für alle denkbaren Maschine einheitlich sind.

Meist werden zwar einfache Datentypen wie ,,*int*" oder ,,*char*" verwendet, aber es lassen sich auch kompliziertere Objekte wie Vektoren in einer entsprechenden `String`-Repräsentation in einer einzelnen Zelle speichern. Natürlich ist es auch sinnvoll einen Vektor über mehrere Zellen zu verteilen, wie ich das bei der Beispielimplementierung der *MaMa* (Kap. 7.2) getan habe.

Für den häufig auftretenden Fall der Verwendung einer Zahl als Wert bietet die Klasse `Cell` das lesen und schreiben durch die Methoden `setIntValue(`*`int`*`)` und `getIntValue()` an. Diese rufen dann `setValue()` bzw. `getValue()` auf.

Typ-Definition

Der Typ einer Zelle definiert seine graphische Darstellung, d.h. seine Größe und Farbe sowie die Positionen, an denen ein- oder ausgehende Zeiger angesetzt werden dürfen. Außerdem kann die Darstellung eines Zeigers für alle Zellen eines Typs verlangt werden.

Obwohl die Bezeichnung der Typen oft den, der Zelle zugewiesenen, Werten angepasst ist, beeinflusst sie diese nicht. Eine Zelle, der der vordefinierte Typ ,,*int*" zugewiesen wurde, kann nach wie vor jeden beliebigen Wert (`String`) beinhalten.

Es stehen eine Reihe vordefinierter Typen wie ,,*int*", ,,*String*", ,,*Register*" oder ,,*Pointer*" zur Verfügung. Zusätzlich lassen sich neue Typen mit der statischen Methode `defineType()` definieren. Durch die Zeile

```
defineType( "myType", Color.green, Color.red, 50, 20, false );
```

wird ein neuer Datentyp mit der Bezeichnung ,,*myType*" angelegt. Die Rechteckdarstellung der Zellen, denen dieser Typ zugewiesen wird, erhält als Hintergrundfarbe Grün und ihr Inhalt wird in rotem Text dargestellt. Die Breite beträgt 50 und die Höhe 20 Pixel. Zur Größenangabe können die vordefinierten Werte `stdWidth` bzw. `stdHeight` genutzt werden. Der letzte Parameter legt fest, ob die Darstellung eines von dieser Zelle ausgehenden Zeigers erfolgen soll (siehe Zeiger Kap. 6.1.3).

Die Methode `defineType` kann noch um zwei Parameter zur Angabe der Zeigermodi (Kap. 6.1.1) ergänzt werden oder auch mit weniger Parametern aufgerufen werden. Die nicht angegebenen Parameter werden dann auf Standardwerte gesetzt.

Erfolgt bei der Generierung einer neuen Zelle keine Angabe über ihren Typ, so wird der Typ ,,*undefined*" verwendet. Die Bestimmung des Typs und seine Festlegung oder Änderung kann auch zu einem späteren Zeitpunkt mit `getType()` bzw. `setType(`*`Type`*`)` erfolgen.

Zeiger

Eine Zelle kann als Referenz auf eine andere Zelle dienen. Hierzu wird der Zelle ein Typ zugewiesen, bei welchem der Parameter für Zeigerdarstellung gesetzt ist. Meist ist dies der vordefinierte Typ ,,*Pointer*", es kann aber auch ein selbst definierter Typ sein.

Der Inhalt der Zelle wird als Adresse interpretiert und die Visualisierung eines Zeigers, der von dieser Zelle ausgeht und auf das der Adresse entsprechende Ziel verweist, wird generiert. Ändert sich der Inhalt der Zelle, so wird die Zeigerdarstellung automatisch angepasst.

Zeigermodi

Von Zellen können Zeiger ausgehen und es können Zeiger auf sie verweisen. Da die Darstellung eines Zeigers als Linie, die vom Zellenrand ausgeht bzw. als Pfeilspitze die dort endet, erfolgt und die Rechteckdarstellung vier Ränder besitzt, bieten sich somit vier Möglichkeiten den Zeiger an die Zelle anzusetzen. Von diesen

Möglichkeiten sollte möglichst eine ,,gute" gewählt werden. ,,Gut" bedeutet hier das Ansetzen an einer freien Seite, d.h. es soll keine Seite ausgewählt werden an der eine weitere Zelle angrenzt, wie dies beim Keller oberhalb und unterhalb oder bei Halden-Objekten rechts oder links der Fall sein kann. Freie Seiten werden durch `setPointerMode(`*`outMode`*`,`*`inMode`*`)` angegeben, wobei `outMode` die möglichen Seiten für ausgehende Zeiger festlegt und `inMode` diejenigen für auf die Zelle verweisende Zeiger.

Als Argumente von `setPointerMode(..)` dienen die in der Klasse `Pointer` definierten Konstanten `NORTH`, `SOUTH`, `EAST` und `WEST`. Diese dürfen durch `|` miteinander verknüpft werden.

Der Aufruf

```
cell.setPointerMode( Pointer.EAST, Pointer.SOUTH | Pointer.NORTH )
```

erlaubt für diese Zelle einen ausgehenden Zeiger nur an der rechten Seite des Rechtecks. Eintreffende Zeiger dürfen hingegen oben und unten angesetzt werden.

Tool Tips

Zu jeder Zelle wird automatisch ein Tool-Tip generiert, der bei längerem Verweilen mit dem Mauszeiger über einer Zelle erscheint. Hier werden weitere Informationen zu der jeweiligen Zelle angezeigt. Dies sind im Einzelnen der Wert und der Typ der Zelle, sowie seine Adresse in der Speicherumgebung.

Der Wert erscheint auch im Tool-Tip, da seine Anzeige als Text innerhalb der Zelle durch die Größe des zur Visualisierung verwendeten Rechtecks beschränkt ist und er daher eventuell nicht vollständig angezeigt werden kann. Der Tool-Tip ermöglicht die Darstellung in voller Länge.

Eine Adresse wird nur dann angezeigt, wenn diese zuvor mit `setAddress(`*`address`*`)` gesetzt wurde, was automatisch durch die Speicherverwaltung des **VAM**-Pakets erfolgt. Keine Adresse besitzen Zellen die nicht über die Speicherverwaltung generiert wurden oder die durch eine Operation als Teil einer Animation entstanden sind und erst nach deren Abarbeitung eine Adresszuweisung erhalten.

Der Tool-Tip wird automatisch an geänderte Daten angepasst und er verschwindet, sobald die Maus aus dem Darstellungsbereich der Zelle bewegt wird.

Interaktion mit dem Benutzer

Der Betrachter der Visualisierung kann Zellen (bzw. Gruppierungen von Zellen) mit der Maus verschieben und erhält dadurch die Möglichkeit, eventuelle Überlagerungen von Zeigern und Zellen innerhalb der Darstellung zu entwirren. Hierzu klickt er eine Zelle an und zieht sie bei gedrückter Maustaste zum gewünschten Zielpunkt.

Desweiteren besteht die Möglichkeit zur Laufzeit Zellenwerte zu ändern. Durch einen Doppelklick auf die entsprechende Zelle erscheint ein Dialogfenster und fordert den Betrachter zur Eingabe des von ihm gewünschten Werts auf, der dann als neuer Inhalt der Zelle gesetzt wird. Diese Möglichkeit der Interaktion ist vor allem zu Testzwecken während der Implementierungsphase einer Maschine gedacht oder dient zur Erzeugung bestimmter Situationen, die in der Lehre veranschaulicht werden sollen.

6.1.2 Register

Register sind spezielle Zellen mit einem Registernamen (bzw. der Abkürzung eines solchen) wie zum Beispiel PC (*Programm Counter*) oder SP (*Stack Pointer*). Sie unterscheiden sich ansonsten nicht von Zellen.

6.1.3 Zeiger (Pointer)

Zeiger werden als Pfeile von einer Zelle und zu einer anderen dargestellt. Sie besitzen eine Ursprungzelle, die bereits im Konstruktor angegeben wird, und deren ganzahliger, positiver Inhalt über die Speicherverwaltung (Abschnitt 6.2) als Zieladresse mit zugehöriger Zelle interpretiert wird. Bei einem negativen Wert oder einer nicht belegten Adresse erfolgt keine Darstellung

Falls nun der Wert der Ursprungszelle und somit die Zieladresse geändert wird, so ruft die Zelle die `update()`-Methode ihres Zeigers auf und bewirkt damit seine Aktualisierung. Diese wird auch ausgelöst, wenn die Position der Ursprungszelle sich ändert. Dabei ist es egal ob diese Positionsänderung die Folge einer durch eine Instruktion ausgelösten Animation ist oder ob sie zur Laufzeit interaktiv vom Betrachter verursacht wird.

Die (Neu-)Berechnung eines Zeigers erfolgt in `calculatePointer()`. Die statische Methode `updateAll()` berechnet alle Zeiger neu. Es wird aus den durch den *Pointer Mode* der jeweiligen Zelle vorgegebenen Möglichkeiten zur Anbringung eines Zeiger eine passende ausgewählt und der Verlauf des Zeigers sowie seine Spitze generiert. Zeiger verlassen eine Zelle orthogonal zum Zellenrand und knicken dann in Richtung der Zielzelle ab. Zwischen Zellenrand und dem Knick wird ein zufälliger Abstand gewählt, um weniger Überlagerungen von Zeigern zu erhalten. Die Auswahl der Zellenseiten zur Anbringung des Zeigers berücksichtigt die Positionen der beiden Zellen, um keinen spitzen Knick zu erhalten (siehe Abbildung 6.1).

Zeiger können übrigens auch von Registern ausgehen. Was im Falle des in funktionalen Maschinen verwendeten *Global Pointers* durchaus zweckmäßig ist. Hingegen erscheint es mir weniger sinnvoll die graphische Repräsentation eines Zeigers für den *Stack Pointer* zu verwenden, da dieser durch die Höhe des Stacks ohnehin augenscheinlich ist. Aber diese Entscheidung bleibt dem Implemetierer der jeweiligen Maschinen überlassen.

6.1.4 Text

Dieser Objekttyp erlaubt die Darstellung eines Textes innerhalb der Visualisierung. Durch ihn werden bei Operationen die Operatoren — etwa ADD, MUL, NEG, +, −, ... — zwischen den Operanden dargestellt.

Der Implementierer einer zu visualisiernden abstrakten Maschine erhält hiermit auch die Möglichkeit beliebige Texte innerhalb des Visualisierungsfensters anzuzeigen und zu plazieren.

```
Text text = new Text( "Wichtiger Text", 100, 100, Color.red );
viewMachine.addVisualization( text );
```

erzeugt die Visualisierung des Textes `Wichtiger Text` in roter Schrift an den Koordinaten (100,100) im Darstellungsbereich der **VAM**-Umgebung `viewMachine`. Die Schriftfarbe kann auch mit `setColor(`*`Farbe`*`)` geändert werden. Falls keine Farbangabe erfolgt, wird der Text in schwarz angezeigt.

6.1.5 Gruppierungen von Zellen (CellGroup)

Zellen lassen sich gruppieren, um dann als Einheit behandelt zu werden. So können zum Beispiel alle Register als eine Gruppe verwaltet werden. Ebenso lassen sich der Keller oder Halden-Objekte wie Vektoren, die aus mehreren Speicherzellen bestehen, zu Einheiten zusammenfassen.

Eine solche Gruppierung ist eigentlich keine eigenständige Visualisierungs-Komponente, sondern fasst solche zusammen und vereinfacht dadurch deren Benutzung. Gruppierungen bieten die Möglichkeit als Ganzes animiert zu werden. Die zur Verfügung stehenden Animations-Methoden sind identisch mit denen der Klasse `Cell`. Dies ist vor allem eine Erleichterung in Bezug auf die Interaktion mit dem Betrachter der Visualisierung, dem hierdurch die Möglichkeit gegeben wird, Zellen zu verschieben ohne dabei unbeabsichtigt zusammengehörige Komponenten zu trennen. Eine solche Trennung würde meistens die Aussagekraft der Visualisierung mindern. Ein Vektor besteht nun einmal aus einer Folge von Zellen. Werden diese Einzelzellen über den gesamten Bildbereich verstreut dargestellt, so verliert die Visualisierung ihre Aussagekraft. Bei einer Darstellung in einer anderen, als der der internen Repräsentation entsprechenden Reihenfolge, ergibt sich durch das gezeigte Bild sogar eine falsche Aussage.

Die Zuweisung von Zellen zu Gruppen erfolgt für Register und Keller automatisch. Auch aus mehreren Zellen bestehende Objekte, die durch die Speicherverwaltung mittels `alloc()` angelegt werden, bilden jeweils eine Gruppe. Wünscht der Animations- und Maschinenentwickler zusätzliche oder andere Gruppierungen, so kann er eine Instanz der Klasse `CellGroup` erzeugen ihr mittels `add(`*`cell`*`)` Zellen hinzufügen.

6.2 Speicherverwaltung (Memory)

Die Klasse `Memory` bietet eine Speicherverwaltung, deren Schnittstelle die erwarteten Lese-, Schreib- und Kopieroperationen zur Verfügung stellt. Aber zusätzlich zur Grundfunktionalität dieser Operationen, also dem Setzen und Ermitteln von Werten, erfolgt die automatische Generierung von statischen sowie dynamischen Visualisationskomponenten. So erzeugt ein Schreibzugriff auf eine bisher unbenutzte Adresse die graphische Repräsentation dieser Zelle, und das Kopieren des Inhalts von Speicherzellen bzw. Registern in andere Zellen erzeugt eine Animation, die den Kopiervorgang dem Betrachter verdeutlichen soll.

6.2.1 Adressen und Speicherinhalte

Adressen sind positive ganze Zahlen (oder `-1` für „*null*"). Einer Adresse entspricht ein Zellen-Objekt und der Inhalt dieses Objekts ist der Speicherinhalt dieser Adresse. Das Setzen und Lesen des Inhalts, sowie die Zuweisung eines Zellentyps erfolgt analog zu den jeweiligen Methoden der Klasse `Cell` jeweils erweitert um eine Adressangabe. Also weist `setValue( 42,"XY")` der Speicheradresse `42` den Wert `XY` zu. Ebenso funktionieren `setIntValue()`, `getValue()`, `getIntValue()`, `setType()`, `getType()`.

Wird der Wert einer bisher nicht genutzten Adresse gesetzt, so generiert die Speicherumgebung automatisch die zugehörige Zelle und blendet diese animiert ein. Der gegenteilige Fall, also die Freigabe einer Adresse und somit auch die Entfernung aus der Visualisierung durch Ausblenden erfolgt durch Aufruf von `remove(`*`address`*`)`. Mit `exists(`*`address`*`)` kann erfragt werden, ob die angegebene Adresse zur Zeit verwendet wird.

6.2.2 Operationen

Die Speicherverwaltung unterstützt die Visualisierung von unären und binären Operationen. Hierzu müssen die Adressen der Operandenzellen, der Operator, das Ergebnis der Operation und die Adresse, der das Ergebnis zugewiesen werden soll, angegeben werden.

Zuerst erfolgt eine Hervorhebung der betroffenen Zellen. Im binären Fall werden dann die Operanden animiert verschoben, so dass sie nebeneinander zu liegen kommen. Anschließend erfolgt die eigentliche Operation visualisiert durch textuelle Darstellung des Operators zwischen den Operanden und Zuweisung des Ergebnisses in eine Zelle, die dann zur Ergebnisadresse verschoben wird.

Bei einer unären Operation wird der Operator vor dem Operanden visualisiert. Ansonsten erfolgt die Behandlung einstelliger Operationen analog zum zweistelligen Fall.

Die Zeile

```
binaryOperation("ADD",42,43, true, 24,"99" );
```

erzeugt die Visualisierung einer zweistelligen Operation namens *ADD* zwischen den Adressen *42* und *43*. Das Ergebnis ist *99* und wird an Adresse *24* abgelegt. Der boolesche Parameter gibt an, ob die Operanden konsumiert werden sollen. Bei diesem Beispiel ist dies der Fall und somit verschwinden die Darstellungen der Zellen an Adresse *42* und *43*. Wird der Parameter auf `false` gesetzt, so bleiben die Operandenzellen erhalten.

Die Adressen der Operanden und der Ergebniszelle müssen nicht zwangsläufig verschieden sein. Unäre Operationen legen das Ergebnis meist in der gleichen Speicherzelle ab, auf deren Inhalt die Operation ausgeführt wurde. Und auch im binären Fall wird häufig einer der Operanden überschrieben. Die verschiedenen Variationen der Adressüberdeckung werden bei der Visualisierung berücksichtigt. Es erfolgt je nach Bedarf eine Animation der Zelle, die der Adresse entspricht, oder einer Kopie dieser Zelle.

6.2.3 Hervorhebung von Zellen

Zellen können optisch hervorgehoben werden, um die Aufmerksamkeit des Betrachters zu erlangen. Die Klasse `Memory` bietet zu diesem Zweck zwei unterschiedliche Methoden an. Die eine erzeugt einen dickeren roten Rahmen um die entsprechenden Zellen und die andere weist den Zellen eine neue Hintergrundfarbe zu.

Der erste Fall ist zur Hervorhebung von den Zellen gedacht, die anschließend verändert oder animiert werden sollen. Er dient also eher der zeitlich kurz begrenzten Markierung bestimmter Speicherinhalte.

Die farbliche Hervorhebung durch die zweite Möglichkeit soll hingegen als längerfristige Markierung verwendet werden, um die besondere Bedeutung bestimmter Zellen zu zeigen. Dies bietet sich zum Beispiel für die Markierung der organisatorischen Zellen bei Verwendung von Kellerrahmen an.

Die unterschiedlichen Aufrufe lauten:

```
highlight( 42, true, 5 );
highlight( 99, Color.green, 5 );
```

Der erste Parameter nennt die Anfangsadresse der Hervorhebung. Das zweite Argument bestimmt die Art der Markierung durch Angabe einer Farbe für Markierungen über einen längeren Zeitraum oder durch Angabe von `true` zur Generierung eines roten Rahmens bzw. durch `false` zum Entfernen desselben. Der letzte Parameter gibt die Anzahl der hervorzuhebenden Zellen an. Entfällt dieser, so wird die Hervorhebung auf eine Zelle angewendet.

6.2.4 Speichern und Kopieren

Ein Großteil der Tätigkeit abstrakter Maschinen besteht im Verändern und Kopieren von Speicherinhalten. Werte werden von einer Zelle in eine andere kopiert oder verschoben. Zur Visualisierung dieser Vorgänge dienen die `store()`-Methoden. Durch Versionen mit unterschiedlicher Argumentanzahl wird deren Verwendung vereinfacht.

Der Kopf der mächtigsten dieser Methoden lautet:

```
public void store( int sourceAddress, int destinationAddress,
                   int number, boolean copy, boolean ascending )
```

Der Parameter `sourceAddress` gibt die Adresse der Speicherzelle an, deren Inhalt an der Adresse `destinationAddress` abgelegt werden soll. `number` nennt die Anzahl der Speicherzellen, die zu berücksichtigen sind. `copy` bestimmt, ob die Zellen kopiert werden sollen (`true`) oder ob eine Verschiebung erfolgt (`false`). Im zweiten Fall verschwindet die Visualisierung der Ursprungszellen und die Inhalte der betroffenen Adressen sind wieder unbestimmt. Falls für `ascending` die Angabe von `true` erfolgt, so werden die Werte ab Adresse `destinationAddress` aufsteigend abgelegt, ansonsten absteigend. Für die booleschen Parameter gibt es auch die vordefinierten Konstanten `COPY`, `NOCOPY`, `ASCENDING` und `DESCENDING`.

Das Kopieren des Speicherbereichs von Adresse 42 bis 50 in den Bereich 20 bis 28 erfolgt durch

```
store( 42,20,9, Memory.COPY );
```

Soll der Speicherbereich stattdessen verschoben werden, so ist dies durch

```
store( 42,20,9, Memory.NOCOPY );
```

möglich.

Werden Parameter nicht bestimmt, so erfolgt die Nutzung der Standardwerte. Diese sind:

Parameter	Voreinstellung
number	1
copy	COPY
ascending	ASCENDING

6.2.5 Zeiger

Die Visualisierung eines Zeigers kann erzeugt werden, indem einer Adresse und damit der Zelle, die dieser Adresse entspricht, durch `setType(`*`address`*`,`*`type`*`)` ein Zellentyp zugewiesen wird, der eine Zeigerdarstellung bewirkt, wie dies zum Beispiel bei dem vordefinierte Typ ,,*Pointer*" der Fall ist.

Auch die Methode `pointer(`*`address`*`)` hat zur Folge, das der Inhalt der Adresse „*address*" als Zeiger interpretiert und dieser dann dargestellt wird. Der Zellentyp bleibt jedoch unverändert und somit erfolgt natürlich auch keine typspezifische farbliche Hervorhebung. Diese Möglichkeit bietet sich vor allem für ungetypte abstrakte Maschinen an.

Die Entfernung der Zeigervisualisierung sollte im ersten Fall durch Setzen eines entsprechenden Zellentyps ohne Zeigerfunktionalität erfolgen. Für den zweiten Fall gibt es die Methode `removePointer(`*`address`*`)`.

6.2.6 Anlegen komplexer Objekte

Die Werte der meisten einfachen Datentypen, wie Zahlen (`int`), können jeweils in einer einzelnen Zelle abgelegt werden. Zusätzlich verwenden viele abstrakte Maschinen jedoch auch komplexere Datentypen, die aus einer Menge von Werten bestehen. Eine Möglichkeit zur Nutzung solcher Datentypen besteht darin, die Werte in eine `String`-Repräsentation zu transformieren. Diese lässt sich dann auch in einer Zelle abspeichern. Dies bewirkt jedoch den Verlust einer Reihe von Visualisierungsmöglichkeiten, wie der Zeigerdarstellung oder dem Setzen unterschiedlicher farblicher Markierungen für Werte unterschiedlicher Bedeutung.

Um die Visualisierungmöglichkeiten des **VAM**-Systems uneingeschränkt nutzen zu können, ist es besser die Einzelwerte komplexer Datentypen auch einzeln in Zellen abzulegen. Diese können dann bei Bedarf einzeln behandelt werden.

Die Speicherverwaltung bietet hierzu die Möglichkeit zusammenhängende Speicherbereiche zu reservieren. Dabei kann dies zum einen durch Angabe der gewünschten Anzahl an Speicherzellen geschehen und zum anderen durch Angabe der gewünschten Zellentypen. Im ersten Fall wird einfach die Startadresse des reservierten Speicherbereichs zurückgeliefert. Beginnend mit dieser Adresse wird ein zusammenhängender Block mit der angegebenen Zellenanzahl visualisiert. Diesen Zellen wird der Typ „*undefined*" zugewiesen, sie erhalten also alle die gleiche graphische Repräsentation.

In der zweiten Möglichkeit zur Erzeugung eines zusammenhängenden Speicherbereichs können Zellen mit unterschiedlichen Typen generiert werden. Hierzu wird der Methode `alloc()` ein Feld von Typen übergeben. Die Größe des Feldes bestimmt die Größe des zusammenhängenden Speicherbereichs und die Feldelemente bestimmen die Typen der Zellen.

Die abstrakte Maschine kann dann, unter Verwendung der normalen Methoden zum Speicherzugriff, ab der zurückgelieferten Adresse auf die gewünschte Anzahl an Zellen zugreifen.

Beispiel:

```
int firstAddress = alloc(5);
String[] types =  "Tag", "int", "int", "Pointer", "Pointer" ;
int secondAddress = alloc(types);
```

Beide Aufrufe erzeugen die Visualisierung eines Objekts bestehend aus fünf Zellen. Im zweiten Fall besitzen die Zellen jedoch unterschiedliche Typen und somit auch unterschiedliche Darstellungen. So werden die letzten beiden Zellen des zweiten Objekts als Zeiger interpretiert.

6.2.7 Direkter Zellenzugriff

Im Allgemeinen sollten die durch Verwendung der Schnittstelle der Speicherverwaltung automatisch generierten Visualisierungen die Vorgänge innerhalb der Maschine ausreichend verdeutlichen. Reichen dem Maschinen-Implementierer und Animationsentwickler die beschriebenen Visualisierungsmöglichkeiten jedoch nicht aus, so kann er durch `getCell(`*`address`*`)` direkt auf die Instanz der Zellenklasse zugreifen, die der angegebenen Adresse entspricht und dann die Methoden dieser Klasse zur Visualisierung nutzen.

Diese Möglichkeit ist weniger zur Verwendung durch einzelne Befehle einer abstrakten Maschine gedacht, sondern vielmehr zur Entwicklung größerer Hilfspakete, die außerhalb der Basisklassen implementiert werden. Die Klasse `Stack` dient als Beispiel für diesen Fall.

6.3 Kellerspeicher (Stack)

Die Klasse `Stack` bietet einen Kellerspeicher mit Visualisierung. Werte bzw. Zellen können nach dem FIFO-Prinzip (*First In First Out*) gespeichert und wieder gelesen werden. Die Visualisierung besteht aus gestapelten Zellen und einem Register mit der Bezeichnung ,,*SP*" für ,,*Stack Pointer*". Dieses Register gibt die Adresse der obersten Zelle des Stapels an, wobei der Registerinhalt `-1` einem leeren Keller entspricht. Im Konstruktor der Klasse muss die zugehörige Speicherverwaltung angegeben werden. Der Keller beginnt an Adresse 1 und wächst nach Bedarf.

Die Implementierung dieser Klasse ist relativ einfach, da sie die Speicherverwaltung des **VAM**-Systems und die damit verbundene automatisierte Visualisierung nutzen kann. Die Klasse `Stack` ist nicht Teil des Basispakets, denn hier soll gezeigt werden, wie sich das Visualisierungssystem erweitern lässt, ohne die Basisklassen zu änderen oder neu übersetzen zu müssen. Die meisten Methoden dieser Klasse lassen sich direkt abbilden auf Methoden der Klasse `Memory`. Lediglich das Register `SP` muss bei Bedarf noch verändert werden.

6.3.1 Zugriffe auf den Keller

Werte werden mit `push(`*`wert`*`)` gekellert, wobei als Argument der Wert als `String` oder `int` angegeben wird. `push()` ohne Argument legt eine leere Zelle auf den Kellerstapel. Die Methoden `pop()` und `popInt()` entfernen das oberste Element und liefern dessen Wert als `String` bzw. `int` zurück.

Unter Verwendung der Methoden `pushFromAddress(`*`address`*`)` und `popToAdress(`*`address`*`)` kann statt eines Werts die Adresse einer Zelle angegeben werden, die den Wert enthält bzw. der der Wert zugewiesen werden soll. `store(`*`address`*`)` ist `popToAdress(`*`address`*`)` sehr ähnlich. Der Wert der obersten Zelle wird bei beiden an der angegebenen Adresse abgelegt. `store(`*`address`*`)` speichert jedoch eine Kopie des obersten Kellerelements, d.h. die Kellerhöhe verringert sich nicht um ein Element.

Die Höhe des Kellers kann auch direkt geändert werden. Durch `increase(`*`number`*`)` und `decrease(`*`number`*`)` wird sie um die angegebene Zellenanzahl erhöht bzw. verringert oder durch `setSize(`*`size`*`)` auf die gewünschte Höhe festgesetzt. Bei Erhöhung werden neue leere Zellen generiert und der Speicherverwaltung und Visualisierung hinzugefügt. Dekrementierung bewirkt die Entfernung überschüssiger Zellen.

Durch `isEmpty()` lässt sich ermitteln, ob der Keller zur Zeit Elemente enthält und `top()` liefert die aktuelle Höhe des Stapels.

6.3.2 Operationen

Die von mir implementierten abstrakten Maschinen verwenden Operationen auf der oder den obersten Kellerzellen. Einstellige Operationen greifen dabei auf die oberste Zelle zu und verändern diese. Die zweistelligen Operationen nehmen die obersten beiden Zellen als Operanden und konsumieren diese. Der Keller wird also um diese zwei Zellen verringert. Das Ergebnis wird anschließend wieder auf den Kellerstapel gelegt und erhöht diesen somit wieder um ein Element. Da viele abstrakte Maschinen Operationen in dieser Art verwenden, habe ich Methoden zur Visualisierung solcher Operationen in der Klasse `Stack` integriert. Dies sind `unaryOperation()` und `binaryOperation()`, die die Methoden zur Visualisierung von Operationen der Klasse `Memory` verwenden. Als Parameter wird der darzustellende Operator als `String` angegeben und das Ergebnis der Operation als `String` oder `int`.

Die Visualisierung von Operationen anderer Art können über die mächtigeren Methoden der Klasse `Memory` erfolgen.

6.3.3 Weitere Visualisierungshilfen

Der Keller bietet neben der Visualisierung von Operationen noch die Möglichkeit die obersten Kellerzellen hervorzuheben. Es reicht die Angabe der Anzahl der Zellen und der gewünschten Hervorhebungsart. Nähere Angaben zur Verwendung der

Hervorhebungsarten befinden sich im Kapitel zur Speicherverwaltung (Abschnitt 6.2.3).

6.4 Implementierung abstrakter Maschinen

Eine abstrakte Maschine ist gegeben durch die Definition ihrer Datenstrukturen und Befehle. Die Implementierung einer abstrakten Maschine für mein Visualisierungssystem VAM besteht aus einer Menge von Klassen, welche in einem Paket (`package`) zusammengefasst werden. Diese Klassen implementieren die einzelnen Befehle. Der Klassenname ist identisch mit dem Namen des jeweiligen Befehls. Zur Laufzeit wird für jede Befehlszeile des auszuführenden (und zu visualisierenden) Programms eine Instanz des entsprechenden Befehls erzeugt. Dem Programmfluss folgend wird dann der jeweils aktuelle Befehl ins Befehlsregister IR (*Instruction Register*) geladen und ausgeführt. Die Ausführung des Befehls besteht im Aufruf seiner `execute()`-Methode. Diese Methode implementiert den Befehl. Sie führt also Operationen auf den Datenstrukturen der abstrakten Maschine aus. Registerinhalte werden gelesen und gesetzt, Lese-, Schreib- und Kopieroperation auf Speicherzellen ausgeführt und gegebenenfalls neue Datenstrukturen erzeugt.

Zusätzlich zu den in der Spezifikation der abstrakten Maschine gegebenen Operationen auf den Datenstrukturen der Maschine, kann bei Bedarf neben der eigentlichen Implementierung der Funktionalität eines Befehls auch noch die Visualisierung beeinflusst werden. So lassen sich Elemente der Visualisierung erzeugen, entfernen oder verändern. Das Setzen einer farblichen Hervorhebung ist eine solche explizite Beeinflussung der Visualisierung. In den meisten Fällen wird die graphische Darstellung jedoch indirekt durch die benutzten Datenstrukturen (`Memory` und `Stack`) des **VAM**-Systems gesteuert.

Die Klasse `Command` dient als Gerüst für alle weiteren Befehle. Von ihr werden alle Befehle abgeleitet. Wird eine neue Instanz eines Befehls erzeugt so erhält der Konstruktor die Argumente des Befehls als Parameter. Zulässig sind beliebig viele Argumente vom Typ `String` und `int`.

Datenstrukturen, die erst zur Laufzeit entstehen, wie etwa Halden-Objekte, können von den einzelnen Befehlen direkt erzeugt werden. Statische Datenstrukturen, wie Register oder der Keller, die für die gesamte Lebensdauer der abstrakten Maschine bestehen, werden in einer speziellen, ebenfalls von `Command` abgeleiteten Klasse generiert und initialisiert. Diese Klasse trägt den Namen `MainCommand`. Es bietet sich an, die Klassen aller Befehle einer Maschine als Unterklassen der `MainCommand`-Klasse dieser Maschine zu implementieren und ihnen somit einen einfachen Zugriff auf die darin enthaltenen Datenstrukturen zu geben. Auch können hier bei Bedarf eigene Methoden (z.B. Hilfsfunktionen), die für alle Befehle dieses Pakets zugänglich sein sollen, eingefügt werden. Zur Initialisierung der statischen Strukturen wird eine

Instanz der Klasse `MainCommand` erzeugt und deren `initialize`-Methode aufgerufen.

6.5 Dynamisch generierte Befehlsinstanzen

Als Eingabe für die Visualisierung dient eine Textdatei, die das darzustellende Programm enthält. Sie wird zur Laufzeit durch den Betrachter angegeben und eingelesen. Aus dieser Datei werden die einzelnen Befehlszeilen extrahiert und gleichzeitig die Sprungziele (*Labels*) tabelliert. Kommentare werden ignoriert. Es werden die beiden gängigen Formen von Kommentaren erkannt:

```
kein Kommentar // Kommentar bis zum Zeilenende
kein Kommentar /* Kommentar */ kein Kommentar
```

Beide Formen müssen sich auf eine Zeile beschränken. Kommentare werden vollständig ignoriert und nicht als Leerzeichen interpretiert. Somit ist „`Bef/* Kommentar */ehl`" identisch zu „`Befehl`".

Für jede Befehlszeile wird nun ein Befehl erzeugt, indem eine Instanz der dem Befehl entsprechenden Klasse generiert wird. Die passende Klasse und ein passender Konstruktor werden zur Laufzeit bestimmt. Der Befehl oder genauer sein Name ist hierbei identisch mit dem Klassennamen. Ein passender Konstruktor wird über die Argumente der Befehlszeile ermittelt.

Da Instanzen der Befehle zur Laufzeit dynamisch generiert werden, dürfen die Konstruktoren keine beliebige Parameterliste haben. Mein System lässt beliebig viele Argumente vom Typ `int` und `String` zu. Falls eine Befehlsklasse mehrere Konstruktoren anbietet, sollten sich diese in der Anzahl ihrer Argumente unterscheiden, um eine deterministische Entscheidung über den Aufruf eines Konstruktors zur Laufzeit zu ermöglichen. Der gewünschte Determinismus kann nicht durch unterschiedliche Argumenttypen erreicht werden, da das Programm als Text(-datei) vorliegt und daher die einzelnen Befehlszeilen und somit auch ihre Argumente immer erst als `String` eingelesen werden. Die Umwandlung des Datentyps eines Arguments erfolgt nur bei Bedarf, also dann, wenn kein passender Konstruktor vorliegt. Enthält die jeweilige Klasse einen Konstruktor mit der passenden Anzahl an Argumenten und diese sind alle vom Typ `String` so ist ein passender Konstruktor gefunden. Falls eines (oder mehrere) der Argumente dieses Konstruktors vom Typ `int` ist, so wird versucht aus der `String`-Repräsentation des Arguments ein `int` zu erzeugen um diesem Konstruktor zu entsprechen. Ist die Umwandlung nicht möglich, da das Argument keine Zahl ist, so kann auch diesem Konstruktor nicht entsprochen und somit keine Instanz eines Befehls für diese Befehlszeile erzeugt werden.

Für `LOADC 5` wird der Konstruktor `LOADC(String arg)` verwendet falls er existiert. Ansonsten erfolgt die Umwandlung der `5` in `int` und `LOADC(int arg)` wird aufgerufen. Existiert keiner der beiden Konstruktoren so ist die Befehlszeile ungültig

und ein ,,leerer" Befehl wird erzeugt. ,,Leer" bedeutet hier, dass die `execute()`-Methode leer ist und somit keine Berechnung stattfindet. Hierzu wird eine Instanz der Klasse `Command` erzeugt, die als Namen die komplette Befehlszeile plus den Zusatz `(undefined)` erhält.

Mögliche Konstruktoren für die Zeile `COMMAND xy 42` sind `COMMAND(String arg1,String arg2)` und `COMMAND(String arg1,int arg2)`. Die beiden anderen denkbaren Konstruktoren mit zwei Argumenten bei denen das erste vom Typ `int` ist passen nicht auf diese Befehlszeile.

Die dynamische Auswahl von Klasse und Konstruktor ermöglicht es, den Befehlssatz einer vorhandenen abstrakten Maschine beliebig zu erweitern oder den Befehlssatz für eine neue Maschine zu generieren, ohne das VAM-System neu zu kompilieren. Auch wenn das System bereits gestartet ist können noch neue Befehle hinzugefügt werden, da ein Zugriff auf die entsprechende Befehlsklasse erst bei der Generierung einer Instanz dieses Befehls, also beim ersten Vorkommen im Programm einer abstrakten Maschine, erfolgt. Wird jedoch ein bereits verwendeter Befehl verändert und neu kompiliert, so muss das VAM-System beendet und erneut gestartet werden, da ansonsten immer die veraltete Implementierung dieser Klasse verwendet wird.

6.6 Animation

Eine Animation ist eine Sequenz von Einzelbildern (*Frames*), die beim Betrachter die Illusion flüssiger Bewegung erzeugt. Um eine Animation ablaufen zu lassen, wird, wie im *Sun Java Tutorial* [Sun2.2000] beschrieben, eine Animations-Schleife verwendet. Diese besteht aus einem Zeitgeber (`javax.swing.Timer`), der in einem festen Interval Ereignisse (`ActionEvent`) auslöst. Jedes Ereignis führt zur Berechnung des nächsten Einzelbildes.

Die durch das **VAM**-System generierte Animation besteht aus einer Reihe von Teilanimationen. So trägt bei einer gleichzeitigen Verschiebung mehrere Zellen jede ihren Teil zur Berechnung des Gesamtbildes bei. Aus diesem Grund stellt mein Paket die Klasse `VisualizationAction` zur Verfügung, von der für jeden Animationsteil eine Instanz generiert wird. Die Liste `visualizationList` beinhaltet alle Instanzen die zur Animation beitragen sollen. Bei jedem Timer-Event wird nun die `nextFrame()`-Methode aller Elemente der Liste aufgerufen, um ihren Teil der Daten für das nächste Einzelbildes zu ermitteln. Durch `addVisualization(`*`action`*`)` und `removeVisualization(`*`action`*`)` werden Aktionen in die Visualisierungsliste eingetragen bzw. aus ihr entfernt. Und `sync` synchronisiert die Animation, d.h. es wird mit der weiteren Abarbeitung der abstrakten Maschine gewartet, bis die aktuelle Animation vollendet ist, also bis keine Elemente mehr in der Visualisierungsliste enthalten sind. Hierzu ist es wichtig, dass alle Aktionen aus der Liste entfernt werden, die keinen Beitrag mehr zur Animation liefern.

6.6.1 Aktionen (VisualizationAction)

Die abstrakte Klasse `VisualizationAction` dient als Basis aller Visualisierungs-Aktionen. Sie beinhaltet die Zelle zu der die Aktion gehört und die Methode `nextFrame()` zur Berechnung der Daten für das nächste Einzelbild. Um eine Animation zu erzeugen wird eine solche Aktion generiert. Diese berechnet bei Bedarf den nächsten Animationsschritt und erzeugt somit jeweils die Daten für ein Einzelbild der Animation.

Das System stellt zwei Aktionen zur Verfügung. Weitere können als Unterklasse von `VisualizationAction` implementiert werden.

Bewegung (MoveAction)

Eine neue Bewegungs-Aktion wird durch `new MoveAction(`*`cell, destination`*`)` generiert. Der Konstruktor benötigt die zu bewegende Zelle und den Zielpunkt. In jedem Animationsschritt wird die Zelle weiter bewegt und zwar um eine bestimmte Anzahl an Bildpunkten. Dieser Wert ergibt sich aus der Ablaufgeschwindigkeit der Animation. In `nextFrame()` werden nur die Koordinaten der Zelle geändert und die Aktualisierung der Ausgabe erfolgt dann durch die Java-Laufzeitumgebung.

Stimmen die Koordinaten der Zelle mit dem Zielpunkt überein, so entfernt die Aktion sich selbst aus der Visualisierungsliste und ist damit beendet.

Ein- und Ausblenden (FadeAction)

Die zweite bereitgestellte Aktion dient zum Ein- und Ausblenden der Zellen. Sie wird verwendet wenn neue Zellen zur Visualisierung hinzugefügt werden und wenn nicht mehr benötigte Zellen entfernt werden.

Der Konstruktor benötigt wieder die Zelle die Gegenstand der Aktion ist, sowie einen booleschen Parameter der angibt, ob eine Ein- oder eine Ausblendung erfolgen soll.

Die Visualisierung wird durch setzen der Farbe der Zelle beeinflusst. Zunächst wird die Zellenfarbe ermittelt und dann deren Alpha-Wert, der die Transparenz angibt, verändert. Zum Einblenden wird die Transparenz vom durchsichtigen Zustand bis hin zur vollen Deckungskraft schrittweise erhöht und zum Ausblenden umgekehrt. Die Schrittweite wird wiederum durch die vom Betrachter gewählte Animationsgeschwindigkeit festgelegt.

Entspricht die Transparenz der Farbe dem gewünschten Endzustand, so wird die Aktion aus der Visualisierungsliste entfernt.

6.7 Verwendung des Systems

Das System verwendet folgende Verzeichnisstruktur:

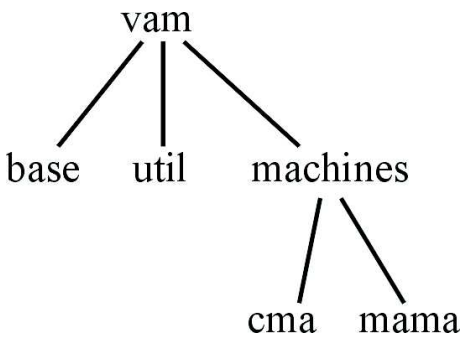


Abbildung 6.2: Verzeichnisstruktur

- **base** beinhaltet die Basisklassen des **VAM**-Systems
- in **util** liegen Zusatzlassen (zur Zeit nur `Stack`)
- **machines** enthält die implementierten abstrakten Maschinen

Das Starten des **VAM**-System erfolgt durch ,,`java vam.base.ViewMachine`". Der Klassenpfad muss hierzu das Verzeichnis enthalten, welches wiederum ,,`vam`" enthält. Ist er nicht korrekt gesetzt, so kann dies durch die Option `-classpath` *`path`* geschehen.

Es erscheint ein Fenster mit Schaltflächen und Menüs, in dem die Visualisierung stattfindet (siehe Abbildung 6.1 auf Seite 20). Der größte Teil dieses Fensters dient der Darstellung der abstrakten Maschine. In ihm werden neue Datenstrukturen visualisiert und Animationen gezeigt. Auf der linken Seite des Fensters befindet sich der Bereich zur Darstellung des auszuführenden Programms. Die aktuell bearbeitete Zeile wird in rot dargestellt. Falls der Modus zur schrittweisen Abarbeitung des Programms gesetzt ist, so erfolgt eine Färbung der nächsten abzuarbeitenden Programmzeile in blau. Bei einem Programm, dass aufgrund seiner Größe nicht vollständig dargestellt werden kann, erscheint die aktuelle Zeile automatisch im sichtbaren Bereich.

Zur Visualisierung wird zunächst die abstrakte Maschine bestimmt. Die erfolgt über den Menüeintrag ,,**Select Machine**". Es erscheint ein Fenster, in dem das Paket der Maschine anzugeben ist. Für die im nächsten Kapitel beschriebene Beispielimplementierung der abstrakten Machine **MaMa** ist hier also ,,`vam.machines.mama`" anzugeben. Die Änderung der Maschine erfolgt dann für das nächste Programm, welches visualisiert werden soll.

Durch Anwahl des Menüeintrags „**Open Program ...**" wird das zu visualisierende Programm für die abstrakte Maschine ausgewählt. Alternativ kann auch die Schaltfläche „**Program**" angeklickt werden, um das Programm zu bestimmen.

Die Visualisierung kann nun beginnen. Dies erfolgt über den Menüeintrag **Run**, durch Anwahl der Schaltfläche **Run** oder einfach durch Anklicken des Hintergrundes des Darstellungsbereichs. Ebenso lässt sich die Visualsierung anhalten und wieder fortsetzen. Mit dem Schieberregler am oberen Fensterrand kann der Betrachter die Ablaufgeschwindigkeit der Animation bestimmen, um interessante Vorgänge bei langsamer Geschwindigkeit genauer betrachten zu können oder weniger wichtige Animationen zu beschleunigen.

Desweiteren lässt sich noch der Abarbeitungs-Modus über den Menüeintrag „**Step-By-Step**" oder die entsprechende Schaltfläche wählen. Ist der Modus „**Step-By-Step**" eingestellt, so erfolgt die Abarbeitung des Programms schrittweise, d.h. nach jeder Programmzeile wird die Visualisierung automatisch gestoppt und erst durch eine entsprechende Benutzerinteraktion wieder fortgesetzt. Dies ermöglicht die genauere Betrachtung der Funktionsweise einzelner Befehle. Ist der Modus zur schrittweisen Abarbeitung nicht ausgewählt, so läuft die Visualisierung bis zur vollständigen Abarbeitung des Programms oder bis zu einer Unterbrechnung durch den Betrachter. Durch Laden eines neuen Programms oder durch Anwahl der Schaltfläche „**Restart**" beginnt die Visualisierung erneut.

Die Beschriftung der Schaltflächen ändert sich entsprechend ihrer Funktion. Zum Beispiel besitzt die zweite Schaltfläche die Funktionen „**Run**", „**Pause**" und „**Restart**". Die Beschriftung entspricht immer der Funktion, die durch Anklicken ausgelöst wird.

Während der Visualisierung kann der Betrachter Visualisierung-Komponenten (Zellen oder Gruppierungen) mit der Maus verschieben und erhält dadurch die Möglichkeit, eventuelle Überlagerungen von Zeigern und Zellen innerhalb der Darstellung zu entwirren. Hierzu klickt er eine Zelle an und zieht sie bei gedrückter Maustaste zum gewünschten Zielpunkt. Außerdem besteht die Möglichkeit zur Laufzeit Zellenwerte zu ändern. Durch einen Doppelklick auf die entsprechende Zelle erscheint ein Dialogfenster und fordert den Betrachter zur Eingabe des von ihm gewünschten Werts auf, der dann als neuer Inhalt der Zelle gesetzt wird. Diese Möglichkeit der Interaktion ist vor allem zu Testzwecken während der Implementierungsphase einer Maschine gedacht oder dient zur Erzeugung bestimmter Situationen, die in der Lehre veranschaulicht werden sollen.

Das **VAM**-System wird durch anwählen des Menüeintrags „**Exit**" oder durch Schließen des Fensters verlassen.

7

Beispiel-Implementierungen abstrakter Maschinen

Im Rahmen meiner Diplomarbeit habe ich zwei abstrakte Maschinen mit Hilfe des VAM-Pakets implementiert, um dessen Anwendbarkeit zu zeigen. Die **CMa** als Beispiel einer Maschine für eine imperative Sprache und die **MaMa** als funktionale Variante. In diesem Kapitel erfolgt eine Beschreibung dieser Implementierungen, die natürlich die Definition der jeweiligen Maschine erfüllen müssen. An einigen Stellen wird die reine Funktionalität jedoch um zusätzliche Visualisierungeffekte ergänzt, um die Anschauung zu verbessern. Solche Ergänzungen können zum Beispiel die Darstellung eines Zeigers oder die farbliche Hervorhebung von Zellen bestimmter Bedeutung sein.

Dieses Kapitel widmet sich der Implementierung. Eine ausführliche Beschreibung und Erklärung der Maschinen erfolgt in [Seidl2000].

7.1 CMa

Die C-Maschine (**CMa**) ist eine abstrakte Maschine für die imperative Programmiersprache **C** (bzw. einer Untermenge dieser Sprache). Die Implementierung besteht aus einer Reihe von Befehlsklassen die in einem eigenen Paket Namens „`vam.maschines.cma`" zusammengefasst sind. Zur Laufzeit wird für jede Zeile eines zu visualisierenden Programms eine Instanz der entsprechenden Klasse erzeugt. Als Superklasse aller Befehle dient die von `Command` abgeleitete Klasse `MainCommand`, durch die auch die Initialisierung der Maschine erfolgt.

7.1.1 Datenstrukturen und Register

Der Programmspeicher und das dazugehörige Register `PC` (*Programm Counter*) sowie der Datenspeicher werden für jede abstrakte Maschine durch das VAM-System bereitgestellt.

Der Datenspeicher der **CMa** ist in zwei Bereiche aufgeteilt, den Keller (*Stack*) und die Halde (*Heap*), welche von den entgegengesetzten Enden des Speichers aufeinander zu wachsen (Abbildung 7.1). Für den Keller dient eine Instanz der Klasse `Stack`, deren Schnittstelle die von der Maschine benötigte Funktionalität bietet. Zur Implementierung der Halde wir die Speicherverwaltung des VAM-Systems verwendet.

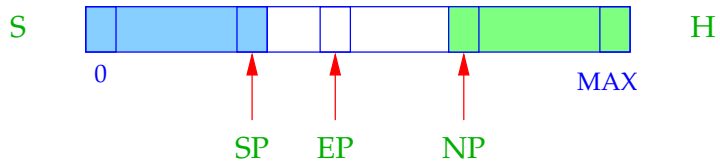


Abbildung 7.1: Keller und Halde der **CMa**

Die verwendeten Register sind:

- *Programm Counter* `PC` — enthält die Adresse des nächsten abzuarbeitenden Befehls; wird vom VAM-System erzeugt
- *Stack Pointer* `SP` — enthält die Adresse der obersten Zelle des Kellers
- *Frame Pointer* `FP` — zeigt auf die letzte organisatorische Zelle und dient zur Adressierung der formalen Parameter sowie der lokalen Variablen
- *Extreme Pointer* `EP` — zeigt auf die Zelle, bis zu der der Keller innerhalb der aktuellen Funktionsabarbeitung wachsen kann
- *New Pointer* `NP` — gibt die Adresse der untersten belegten Haldenzelle an

7.1.2 Initialisierung

Die Generierung und Initialisierung der verwendeten Datenstrukturen und Register erfolgt durch `MainCommand`:

```
package vam.machines.cma;
import vam.base.*;                // Basispaket zur Visualisierung
import vam.util.Stack;            // Kellerspeicher

public class MainCommand extends Command
{
  static ViewMachine viewMachine; // ermöglicht Zugriff
                                  // auf die Visualisierung
  static Memory memory;           // Speicherverwaltung
  static Stack stack;             // Kellerspeicher
  static Register FP;             // Frame Pointer
  static Register EP;             // Extreme Pointer
  static Register NP;             // New Pointer
  static CellGroup heapGroup;     // Gruppierung der Halde

  public MainCommand( String command )
  {
    super( command );
  }
  public MainCommand()
  {
    super( "MainCommand CMA" );
  }

  ...
}
```

- Die Register und der Kellerspeicher sind Datenstrukturen der **CMa**.
- `viewMaschine` ermöglicht den Zugriff auf den `PC` und die Fehlerausgabe des VAM-Systems.
- `memory` wird als Abkürzung für `viewMaschine.memory` verwendet und dient zum Zugriff auf die Speicherverwaltung. Dies ist zwar unnötig dient aber aufgrund der häufigen Verwendung in der Implementierung der Übersichtlichkeit, was in der Lehre wünschenswert ist.
- Die Gruppierung erleichtert die interaktive Verschiebung der Halde durch den Betrachter.

```
public void initialize( ViewMachine vm )
{
  viewMachine = vm;

  // Datenstrukturen
  memory = viewMachine.memory;
  stack = new Stack( memory );

  // Register setzen
  FP = new Register( "FP" );
  FP.set( -1 );
  FP.isPointer( true );

  EP = new Register( "EP" );
  EP.isPointer( true );

  NP = new Register( "NP" );
  NP.set(1000);
  NP.isPointer( true );

  // Gruppe zur Gruppierung der Halde
  heapGroup = new CellGroup("Heap");
}
```

- Die Register werden gemäß der **CMa**-Definition gesetzt. Die Inhalte der Register `FP` und `NP` sind Adressen und sollen daher durch Zeiger visualisiert werden. `EP` gibt ebenfalls eine Adresse an. Da diese in der Regel jedoch auf eine nicht reservierte und daher auch nicht visualisierte Zelle verweist, erfolgt hier keine Zeigerdarstellung.
- Die Initialisierung des `NP` erfolgt mit `1000` da das Layout für die Visualisierung der Halde in diesem Speicherbereich gedacht ist. Um zu Lehrzwecken einen *Stack Overflow*-Fehler zu erzeugen kann auch eine kleinere Adresse angegeben werden.

Nun wird der Registerzugriff durch einige Methoden etwas vereinfacht und somit auch übersichtlicher. Alle Register können dann mit `XY()` gelesen und durch `XY(`*`value`*`)` gesetzt werden.

```
int SP()
{
  return stack.top();
}
```

```
void SP( int newSP )
{
  stack.setSize( newSP );
}
```

Für die anderen Register existieren analoge Vereinfachungen des Zugriffs.

Außerdem gibt es noch die Methode `cmaError` für die Weiterleitung von Fehlermeldungen, wie etwa *Stack Overflow*, die zur Laufzeit durch die abstrakte Maschine erzeugt werden.

```
void cmaError( String errorMessage )
{
  viewMachine.error( "CMA: "+errorMessage );
}
```

7.1.3 Konstruktoren und Argumente

Die Befehle der **CMa** sind parameterlos oder sie besitzen ein Argument vom Typ `String` oder `int`. Der Konstruktor jeder Befehlsklasse setzt den Namen des Befehls und speichert das übergebene Argument in einer Variablen der erzeugten Instanz dieser Klasse. Es ergeben sich also drei mögliche Konstruktoren:

1. Ohne Argument:

```
public COMMAND()
{
  super("COMMAND");
}
```

2. Ein Argument vom Typ `int`:

```
int argument;
public COMMAND( int argument )
{
  super("COMMAND "+argument);
  this.argument = argument;
}
```

3. Ein Argument vom Typ `String`:

```
String argument;
public COMMAND( String argument )
{
  super("COMMAND "+argument);
  this.argument = argument;
}
```

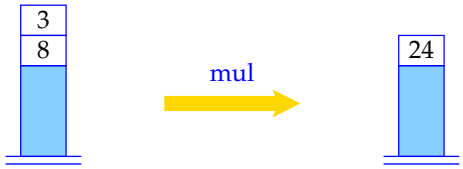
Die Konstruktoren aller Befehlsklassen der **CMa** sind bis auf die Bezeichnungen identisch zu einem dieser drei Fälle und es bedarf daher in den folgenden Ausführungen keiner gesonderten Erklärung zu den jeweiligen Konstruktoren. Welche Variablen Objektdaten sind sollte aus dem jeweiligen Kontext klar werden.

7.1.4 Unäre und binäre Operationen

Die **CMa** besitzt eine Reihe von Befehlen für zweistelligen Operatoren. Alle greifen auf die obersten beiden Kellerelemente zu, konsumieren diese und berechnen daraus das Ergebnis der Operation und legen dieses dann wieder auf den Kellerstapel.

CMa-Definition des Befehls MUL:

```
SP--;
S[SP] = S[SP] * S[SP+1];
```



Implementierung für das VAM-System:

```
package vam.machines.cma;
public class MUL extends MainCommand
{
  public MUL()
  {
    super("MUL");
  }
  public void execute()
  {
    int op2 = memory.getIntValue( SP() );
    int op1 = memory.getIntValue( SP()-1 );
    stack.binaryOperation( "MUL", op1 * op2 );
  }
}
```

Bemerkungen:

- Da die Operation keine Argumente besitzt benötigt der Konstruktor auch keine. Er gibt nur den Namen des Befehls an den Konstruktor der Superklasse weiter.
- In den ersten beiden Zeilen der `execute`-Methode werden die Werte der Operandenzellen gelesen, um aus ihnen das Ergebnis berechnen zu können. Hier erfolgt keine Visualisierung.
- In Zeile drei erfolgt schließlich die Visualisierung der Operation durch Aufruf der entsprechenden Methode der Klasse `Stack`. Der `String MUL` dient hier als Text, der zwischen den Operandenzellen dargestellt wird und der zweite Parameter gibt das Ergebnis an, das nach Ausführung der Operation oben auf dem Keller liegen soll.
- Die Korrektur des Registers `SP` (*Stack Pointer*) erfolgt automatisch.

Die Implementierung der anderen binären Operationen der **CMa** erfolgt analog. Zu ändern ist die Berechnung des Ergebnisses und alle Vorkommen des Namens `MUL` müssen natürlich durch den neuen Befehlsnamen ersetzt werden.

Implementiert sind die binären arithmetischen und logischen Operationen `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `AND`, `OR` und `XOR`, sowie die Vergleiche `EQ`, `NEQ`, `LE`, `LEQ`, `GE` und `GEQ`.

Die Befehle `NEG` und `NOT` sind einstellige Operationen. Die Implementierung ist dem Beispiel oben sehr ähnlich.

CMa-Befehl	**CMa-Definition**	**execute()**
`NEG`	`S[SP] = -S[SP];`	`int op1 = memory.getIntValue(SP());` `stack.unaryOperation( "NEG",-op1 );`

7.1.5 Sprung-Befehle

In der Implementierung eines Sprungbefehls ist das Register `PC` (*Programm Counter*), das die Adresse des nächsten auszuführenden Befehls enthält, neu zu setzen. Die Sprungbefehle der **CMa** haben alle ein Argument und zwar das Sprungziel. Damit sieht der Konstruktor wie folgt aus:

```
int address;
public JUMP(int address)
{
  super("JUMP "+address);
  this.address = address;
}
```

Die Implementierungen der `execute()`-Methoden der Sprungbefehle sind :

CMa-Befehl	**CMa-Definition**	**execute()**
`JUMP adr`	`PC = adr;`	`PC( adr );`
`JUMPI adr`	`PC = adr + S[SP];` `SP--;`	`PC( adr + parseInt(stack.pop()) );`

`JUMPZ adr`	`if( S[SP]==0 )`
	`  PC = adr;`
	`SP--;`
		`if( parseInt(stack.pop())==0 )`
		`  PC( adr );`

`HALT`	Maschine anhalten	`PC( -1 );`

Die Verwendung von `stack.pop()` passt den *Stack Pointer* `SP` automatisch an und macht somit dessen explizite Inkrementierung überflüssig.

Der Befehl `HALT` bewirkt durch Setzen des `PC`s auf einen nicht gültigen Wert das Anhalten der Maschine und wird daher ebenfalls hier aufgeführt.

7.1.6 Laden von Werten

Das Laden von Werten bedeutet oben auf den Keller legen. Dies kann immer mit `stack.push(`*`value`*`)` erreicht werden. Die `LOAD`-Befehle der **CMa** unterscheiden sich in Bezug auf den Ursprung der zu ladenden Werte.

Laden von Konstanten:

CMa-Befehl	**CMa-Definition**	**execute()**
`LOADC value`	`SP++;` `S[SP] = value;`	`stack.push( value );`
`LOADRC value`	`SP++;` `S[SP] = FP + value;`	`stack.push( FP() + value );`

Laden von Werten aus Speicherzellen:

CMa-Befehl	**CMa-Definition**	**execute()**

`LOAD`	`S[SP]=S[S[SP]];`	`int address = memory.getIntValue( SP() );`
		`memory.pointer( SP() );`
		`memory.store( address, SP() );`
		`memory.removePointer( SP() );`

`LOADA address`	`LOADC address`
	`LOAD`
		`stack.push( address );`
		`memory.pointer( SP() );`
		`memory.store( address, SP() );`
		`memory.removePointer( SP() );`

`LOADR address`	`LOADRC address`
	`LOAD`
		`int absoluteAddress = FP()+address;`
		`stack.push( absoluteAddress );`
		`memory.pointer( SP() );`
		`memory.store(absoluteAddress,SP());`
		`memory.removePointer( SP() );`

Diese Befehle unterscheiden sich nur in der Adresse von der sie laden. `LOAD` besitzt keinen Parameter, da die zu verwendende Adresse oben auf dem Keller erwartet wird. `memory.getIntValue(SP())` liefert diese Adresse und `memory.store(address,SP())` speichert ihren Inhalt im obersten Kellerelement ab. Die Funktionalität des Befehls für die **CMa** ist damit bereits erreicht.

Die Markierung als Zeiger und die Entfernung dieser Markierung bewirkt die zusätzliche Visualisierung der Adresse als Zeiger. Sie soll dem Betrachter zeigen, woher der Wert geladen wird. Die `execute`-Methode könnte auch nur aus der Anweisung `memory.store(memory.getIntValue(SP()),SP())` bestehen.

Die anderen beiden Befehle kellern erst eine Adresse und verfahren dann wie `LOAD`.

7.1.7 Speichern von Werten

Die `STORE`-Befehle speichern auf dem Keller liegende Werte an einer angegeben Adresse im Speicher ab. Es ist dabei egal ob die Zieladresse eine Keller- oder aber eine Haldenzelle bezeichnet. Es gibt wie bei den `LOAD`-Kommandos eine Variante, die eine Adresse als oberstes Kellerelement erwartet und zwei weitere, die erst eine Adresse kellern und dann wie Variante eins verfahren. Gespeichert wird immer der Wert, der sich auf dem Keller unmittelbar unter der Zieladresse befindet.

CMa-Befehl	**CMa-Definition**	**execute()**

STORE	
	
	S[S[SP]]=S[SP-1];
	SP--;
		int address
		= memory.getIntValue(SP());
		memory.highlight(SP(), true);
		memory.pointer(SP());
		memory.store(SP()-1, address);
		stack.pop();

STOREA address	LOADC address
	
	STORE
		stack.push(address);
		memory.highlight(SP(), true);
		memory.pointer(SP());
		memory.store(SP()-1, address);
		stack.pop();

STORER address	
	
	LOADRC address
	
	STORE
		int absoluteAddress
		= FP() + address;
		stack.push(absoluteAddress);
		memory.highlight(SP(), true);
		memory.pointer(SP());
		memory.store(SP()-1, address);
		stack.pop();

Die erste bzw. die ersten beiden Zeile dienen zum lesen bzw. zum kellern der Zieladresse.

Die letzten vier Zeilen der Befehle sind identisch. Das oberste Kellerelement, das ja die Zieladresse enthält, wird hervorgehoben und als Zeiger visualisiert. Dann erfolgt das eigentliche speichern des zweitobersten Wertes an der angegebenen Adresse, die anschließend durch `stack.pop()` vom Keller entfernt wird. Die Adresse wird also konsumiert, der Wert selbst bleibt jedoch auf dem Keller erhalten.

Die eigentliche Funktionalität nach der **CMa**-Definition ist bereits durch die Anwendung von `memory.store(..)` erreicht. Die Hervorhebung der Zieladresse und deren Darstellung als Zeiger ist eine Erweiterung der automatisch generierten Visualisierung.

7.1.8 Befehle zur Unterstützung von Funktionen

Zum Aufruf einer Funktion müssen Register gerettet und dann neu gesetzt werden, und die formalen Parameter sowie die lokalen Variablen werden angelegt. Die **CMa** verwendet hierzu den Keller. Ein Speicherbereich der benötigten Größe, der sogenannte Keller-Rahmen (Abbildung 7.2), wird auf dem Keller angelegt und initialisiert. Die notwendigen Aktionen verteilen sich auf mehrere **CMa**-Befehle, die dann nacheinander abgearbeitet werden.

Aktionen beim Betreten einer Funktion	Befehl	Programmstelle
1. Retten der Register FP und EP	MARK	stehen in der aufrufenden Funktion
2. Bestimmung der aktuellen Parameter inklusive der Anfangsadresse der Funktion	durch den Compiler	
3. Setzen des neuen FP	CALL	
4. Retten von PC und Sprung an den Anfang des Rumpfes der Funktion		
5. Setzen des neuen EP	ENTER	in aufzurufenden Funktion
6. Allokieren der lokalen Variablen	ALLOC	

Aktionen beim Verlassen einer Funktion	Befehl	Programmstelle
1. Zurücksetzen der Register FP, EP und SP	RETURN	in aufgerufener Funktion
2. Rücksprung, d.h. Restauration des PC		

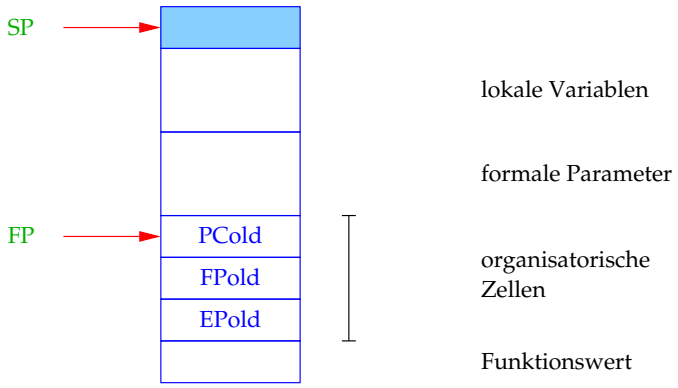


Abbildung 7.2: Kellerrahmen der **CMa**

Der Befehl MARK (Abb. 7.3) reserviert auf dem Keller Platz für den Rückgabewert und die organisatorischen Zellen, in denen die aktuellen Inhalte der Register FP und EP gerettet werden.

CMa-Befehl	CMa-Definition	execute()

MARK	S[SP+2]=EP
	S[SP+3]=FP
	SP=SP+4
		stack.push();
		stack.push(EP());
		stack.push(FP());
		stack.push();
		stack.highlight(4,Color.magenta);

Es werden vier Zellen auf dem Stack reserviert, jedoch nur die mittleren beiden initialisiert. Die anderen sind für die Rücksprungadresse und den Rückgabewert. Außerdem werden die vier organisatorischen Zellen dauerhaft farblich hervorgehoben und sind somit in ihrer besonderen Bedeutung für den Betrachter zu erkennen.

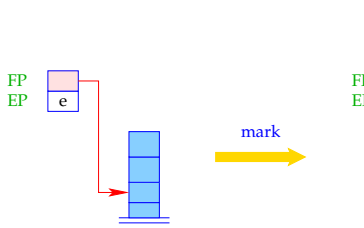


Abbildung 7.3: MARK

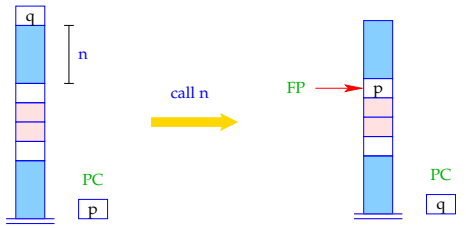


Abbildung 7.4: CALL n

Der Befehl `CALL` (Abb. 7.4) rettet die aktuelle Fortsetzungsadresse, zu der nach Abarbeitung der Funktion zurückgesprungen werden muss, und setzt außerdem den *Frame Pointer* auf den aktuellen Keller-Rahmen. Dann erfolgt der Sprung in die aufzurufende Funktion durch Setzen des `PC`.

CMa-Befehl	**CMa-Definition**	**execute()**

`CALL n`	`FP=SP-n-1;`
	`S[FP]=PC;`
	`PC=S[SP];`
	`SP--;`
		`FP( SP()-n-1 );`
		`memory.setIntValue( FP(), PC() );`
		`PC( stack.popInt() );`

Der Befehl `ENTER` (Abb. 7.5) setzt den *Extreme Pointer* auf die maximale Kellerhöhe die während der Abarbeitung der aufgerufenen Funktion erreicht werden kann. Steht nicht genügend Platz zur Verfügung wird die Programmausführung abgebrochen.

Der Befehl `ALLOC` reserviert auf dem Keller Platz für die lokalen Variablen. Der *Stack Pointer* wird hierzu einfach um die benötigte Anzahl an Zellen erhöht.

Befehl	**CMa-Definition**	**execute()**

`ENTER n`	`EP = SP+n;`
	`if( EP ≥ NP )`
	`  Error("Stack Overflow");`
		`EP( SP() + n );`
		`if( EP() >= NP() )`
		`  cmaError("Stack Overflow");`

`ALLOC n`	`SP = SP+n`	`stack.increase(n);`

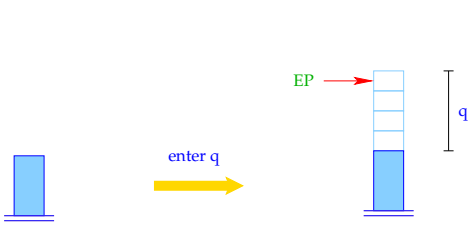


Abbildung 7.5: ENTER n

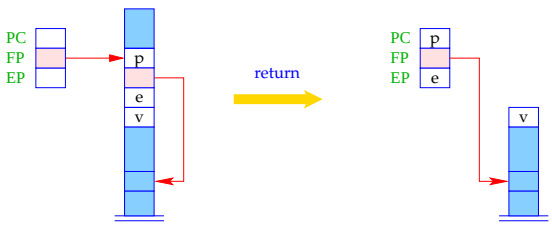


Abbildung 7.6: RETURN

Nach Abarbeitung einer Funktion gibt der Befehl `RETURN` (Abb. 7.6) den aktuellen Keller-Rahmen auf, indem er die Register `PC`, `EP` und `FP` restauriert. Der berechnete Rückgabewert wird oben auf dem Keller hinterlassen.

Befehl	**CMa-Definition**	**execute()**

`RETURN`	`PC = S[FP];`
	`EP = S[FP-2];`
	`if( EP ≥ NP )`
	`  Error("Stack Overflow");`
	
	`SP = FP-3;`
	`FP = S[SP+2]`
		`PC(memory.getIntValue( FP() ));`
		`EP(memory.getIntValue(FP()-2));`
		`if( EP() >= NP() )`
		`  cmaError("Stack Overflow");`
		`int newFP =`
		`  memory.getIntValue(FP()-1);`
		`SP( FP()-3 );`
		`FP( newFP );`

Achtung: Die letzte Zeile kann **nicht** durch `FP(memory.getIntValue(SP()+2));` ersetzt werden, da die Dekrementierung des *Stack Pointers* in der Zeile zuvor alle Zellen oberhalb des neuen `SP` entfernt hat und die Adresse `SP+2` somit nicht mehr existiert. Aus diesem Grund wird der neue Wert von `FP` vorher gesichert. In der Definitionssprache ist ein Zugriff auf Zellen oberhalb des `SP` möglich.

7.1.9 Weitere Befehle

Weitere Befehle der **CMa** sind `POP` zum Entfernen und `DUP` zum Duplizieren des obersten Keller-Elements.

CMa-Befehl	**CMa-Definition**	**execute()**
`POP`	`SP--;`	`stack.pop();`
`DUP`	`S[SP+1] = S[SP];` `SP++;`	`stack.pushFromAddress( SP() );`

Der Befehl `NEW` (Abb. 7.7) reserviert Speicher der Halde und liefert eine Referenz darauf zurück.

CMa-Befehl: NEW	
CMa-Definition	**execute()**

`if( NP-S[SP] ≤ EP )`
`  S[SP] = NULL;`
`else`
`{`
`  NP = NP-S[SP];`
 	
`  S[SP] = NP;`
 	
`}`
	`memory.highlight( SP(),true );`
	`if( (NP()-memory.getIntValue(SP())) <= EP() )`
	`  memory.setIntValue( SP(), -1 );`
	`else`
	`{`
	`NP( NP() - memory.getIntValue( SP() ) );`
	`int number = memory.getIntValue( SP() );`
	`for( int i=0; i<number; i++ )`
	`{`
	`memory.setValue( NP()+i, );`
	`heapGroup.add( memory.getCell( NP()+i ) );`
	`}`
	`memory.setIntValue( SP(), NP() );`
	`memory.pointer( SP() );`
	`}`
	`memory.highlight( SP(), false );`

In der `for`-Schleife werden die reservierten Zellen der Halde initialisiert, wodurch implizit deren Visualisierung erfolgt. Die Zeile `heapGroup.add(memory.getCell(NP()+i));` ordnet alle Haldenzellen einer Gruppe zu. Dies ist nicht zwingend notwendig aber empfehlenswert, da ansonsten der Betrachter die Halde nicht als Ganzes sondern Zellenweise durch die Maus verschieben kann.

Da die oberste Keller-Zelle nach Abarbeitung des Befehls eine Referenz (auf den reservierten Speicher) enthält, wird sie als Zeiger markiert.

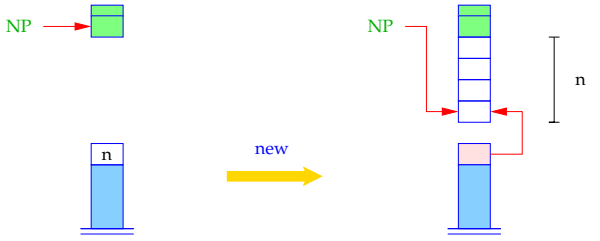


Abbildung 7.7: NEW

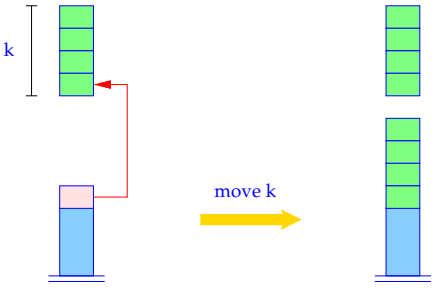


Abbildung 7.8: MOVE k

Der letzte zu beschreibende Befehl meiner **CMa**-Implementierung ist `MOVE k` (Abb. 7.8), der zum Kopieren von Strukturen verwendet wird. Er erwartet eine Adresse auf dem Keller und kopiert ab der referenzierten Zelle die als Befehlsparameter angegebene Anzahl an Zellen auf den Keller.

CMa-Befehl: `MOVE k`	
CMa-Definition	**execute()**

`for(i=k-1; i≥0; i--)`
`  S[SP+i] = S[S[SP]+i];`
 	
`SP=SP+k-1;`
	`stack.highlight( 1,true );`
	`memory.pointer( SP() );`
	`if( number>0 )`
	`{`
	`  int address = memory.getIntValue(SP());`
	`  memory.store( address, SP(), number );`
	`  if( !memory.isPointer( address ) )`
	`  memory.removePointer( SP() );`
	`  SP( SP()+number-1 );`
	`}`
	`else stack.pop();`

Auch hier ist eine Referenz Inhalt der obersten Kellerzelle, die deshalb als Zeiger markiert wird. Da diese Zelle durch einen neuen Wert überschrieben wird, muss die Zeigermarkierung wieder entfernt werden, falls es sich bei dem neuen Wert nicht ebenfalls um eine Referenz handelt.

Da mittels `memory.store(..)` mehrere Zellen gleichzeitig kopiert werden können ist in meiner Implementierung keine `for`-Schleife notwendig. Die Zellen könnten natürlich auch in einer Schleife einzeln kopiert werden, jedoch ist die Laufzeit der dabei generierten Animation wesentlich länger.

Ist die angegebene Anzahl nicht größer oder gleich eins, so erfolgt kein Kopiervorgang. Das oberste Keller-Element mit der Referenz wird in diesem Fall jedoch nicht überschrieben und muss deshalb explizit durch `stack.pop()` entfernt werden.

7.1.10 Ein Programm-Beispiel

Hier ein **CMa**-Programm zur Berechnung der Fakultät der Zahl 4 aus [Seidl2000] ergänzt um `main`:

```
        ENTER 5              STORER -3       _a:     LOADR 1
        ALLOC 0              RETURN                  MARK
        MARK          _fac:  ENTER 6                 LOADR 1
        LOADC _main          ALLOC 0                 LOADC 1
        CALL 0               LOADR 1                 SUB
        HALT                 LOADC 0                 LOADC _fac
_main:  ENTER 5              LEQ                     CALL 1
        ALLOC 0              JUMPZ _a                MUL
        MARK                 LOADC 1                 STORER -3
        LOADC 4              STORER -3               RETURN
        LOADC _fac           RETURN          _b:     RETURN
        CALL 1               JUMP _b
```

Dies entspricht:

```
int fac(int x)
{
    if( x≤0 ) return 1;
    else return x * fac(x-1);
}
main()
{
    fac(4);
}
```

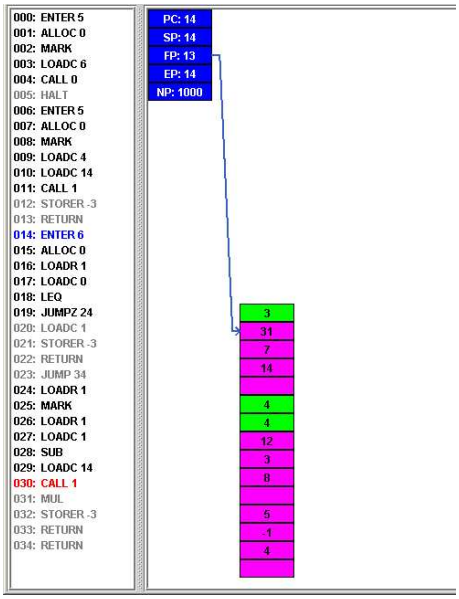


Abbildung 7.9:

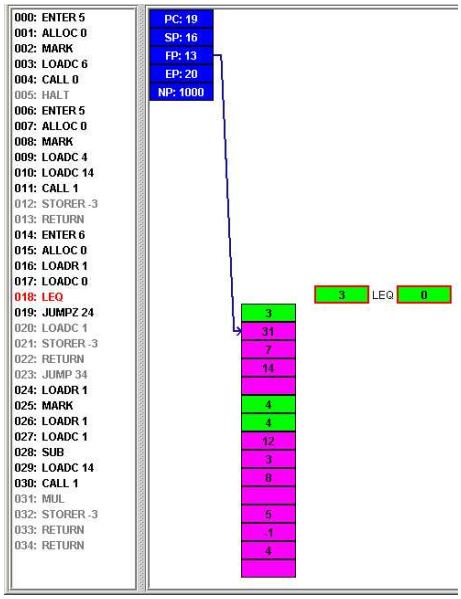


Abbildung 7.10:

Abbildung 7.9 zeigt die Situation nach Aufruf der Funktion `fac(3)` aber vor deren Abarbeitung. Die rote Zeile im Programm-Fenster links im Bild zeigt den aktuellen Befehl (`CALL`) und die blaue die nächste Befehlszeile. Die violetten Zellen sind die organisatorischen Zellen der Kellerrahmen (Abb. 7.2 Seite 49). Hier sind bereits drei Kellerrahmen angelegt – für des Programm, für `main` und für die erste Rekursion von `fac`, also `fac(4)`.

Durch `CALL` wird die Funktion `fac` angesprungen. Dort erfolgt zunächst die Überprüfung der Bedingung `if(x≤0)`, was in Abbildung 7.10 gezeigt ist. Da die Bedingung nicht erfüllt ist, wird durch `MARK` ein neuer Kellerrahmen für die nächste Rekursion angelegt (Abb. 7.11).

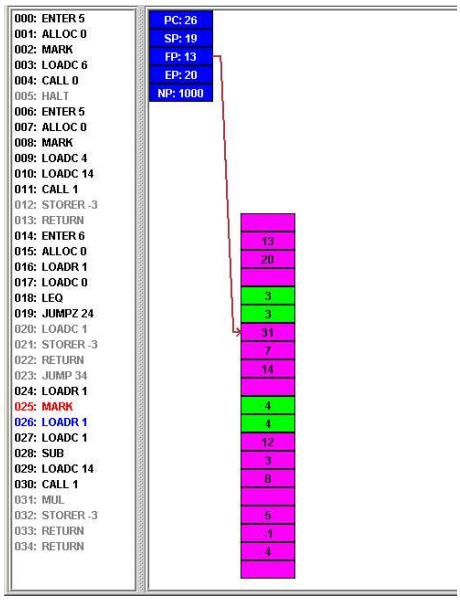


Abbildung 7.11:

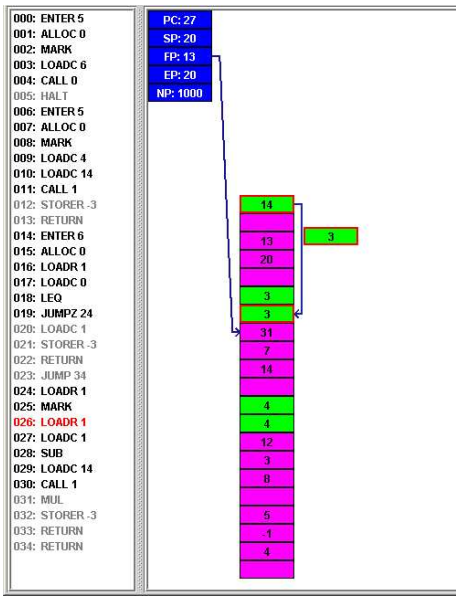


Abbildung 7.12:

Nun wird der Parameter für den Aufruf von `fac` bestimmt. Hierzu wird das aktuelle `x` geladen (Abb. 7.12) und um eins dekrementiert (Abb. 7.13).

Abbildung 7.14 zeigt schließlich die Situation vor Aufruf der Funktion `fac(2)`. Die oberste Kellerzelle enthält die Adresse der Funktion und die Zelle darunter den Parameter.

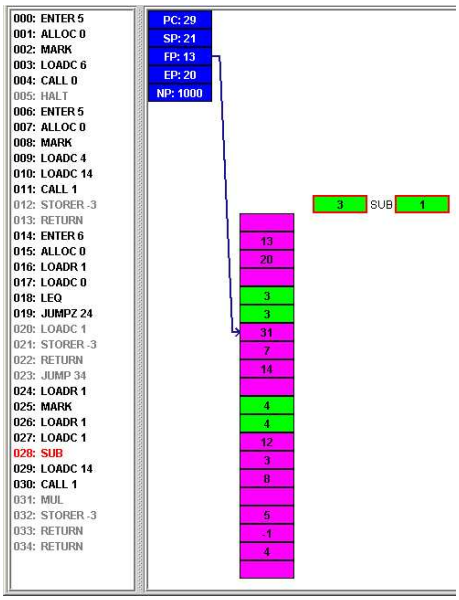


Abbildung 7.13:

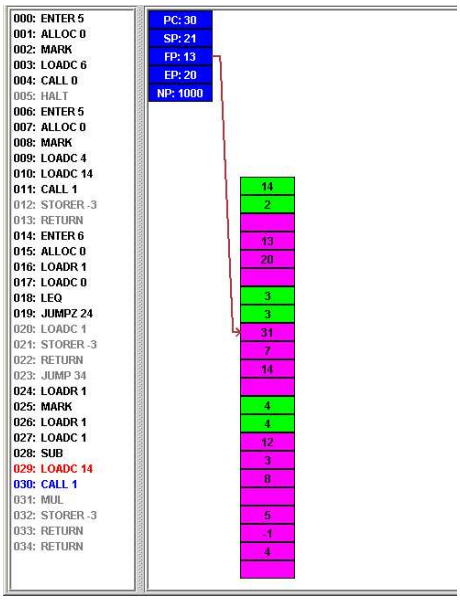


Abbildung 7.14:

Zum besseren Verständnis sei dem Leser die Betrachtung der vollständigen und animierten Visualisierung mit Hilfe des **VAM**-Systems nahe gelegt.

7.2 MaMa

Die **MaMa** ist eine abstrakte Maschine für die funktionale Mini-Sprache **PuF** (*Pure Functions*). Im Gegensatz zur **CMa** ist diese Maschine getypt, z.B. werden Adressen von Zahlen unterschieden. Aus diesem Grund kann in diesem Beispiel das Konzept der Zellen-Typen (Kap. 6.1.1) zur unterschiedlichen Visualisierung von Werten verwendet werden. Die Implementierung ist im Paket „`vam.maschines.mama`" zusammengefasst. Auch hier dient die Klasse `MainCommand` zur Initialisierung und als Superklasse für die einzelnen Befehle.

7.2.1 Datenstrukturen und Register

Die **MaMa** verwendet eine Reihe der Datenstrukturen und Register der **CMa** und zwar

- den Programmspeicher
- mit dem Register `PC` (*Programm Counter*)
- und den Kellerspeicher (*Stack*)
- mit dem Register `SP` (*Stack Pointer*)
- sowie das Register `FP` (*Frame Pointer*)

Zusätzlich wird auch hier eine Halde benötigt. In der **CMa** können in diesem Speicher Bereiche reserviert und dann beliebig genutzt werden. Die **MaMa** definiert ihre Halde als abstrakten Datentyp, in dem Daten-Objekte (Abb. 7.16) abgelegt werden können. Über Details der Adresszuordnung der Halde und der Daten-Objekte gibt die Definition keine Auskunft.

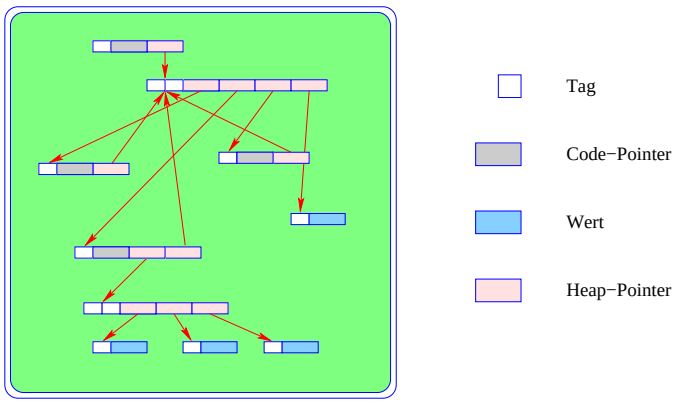


Abbildung 7.15: Halde der **MaMa**

7.2.2 Initialisierung

Die Konstruktoren werden wie in der **CMa** verwendet. Der Befehl `MOVE` erwartet lediglich zwei Argumente.

Die Register werden analog zur **CMa** generiert und initialisiert. Und auch die Methoden zum vereinfachten Registerzugriff stehen den **MaMa**-Befehlen zur Verfügung.

Neu ist allerdings die Verwendung von Zellen-Typen. Die Halden-Objekte (Abb. 7.16) der **MaMa** bestehen aus Einträgen unterschiedlicher Bedeutung. Um dies dem Betrachter zu vermitteln, werden Zellen-Typen mit unterscheidbarer Visualisierung verwendet, die zunächst definiert werden müssen:

```
Cell.defineType( "Tag"     , Color.white,Color.black
       , 20, Cell.stdHeight, Cell.NOPOINTER
       , Pointer.WEST|Pointer.NORTH|Pointer.SOUTH
       , Pointer.WEST|Pointer.NORTH|Pointer.SOUTH );
Cell.defineType( "List Tag", Color.white, 35, Cell.stdHeight );
Cell.defineType( "Counter" , Color.white, 30, Cell.stdHeight );
Cell.defineType( "Basic"   , Color.cyan );
Cell.defineType( "Code Pointer", Color.gray, 30, Cell.stdHeight );
Cell.defineType( "Heap Pointer", Color.yellow, Color.yellow
       , 20, Cell.stdHeight, Cell.POINTER
       , Pointer.NORTH|Pointer.SOUTH, Pointer.NORTH|Pointer.SOUTH );
Cell.defineType( "Dummy"   , Color.white,0,0 );
```

Zur Verwendung:

- `TAG` — Kennzeichnung der Halden-Objekte (`B`,`C`,`F`,`V`,`L`)
- `List Tag` — Kennzeichnung für Listen (`Nil`,`Cons`)
- `Counter` — Zähler für Vektoren
- `Basic` — Wert eines *Basic*-Objekts
- `int` (vordefiniert) — Werte im Kellerspeicher
- `Code Pointer` — Programm-Adresse
- `Heap Pointer` — Referenz auf Objekte
- `Pointer` (vordefiniert) — Referenz auf Objekte
- `Dummy` — Zelle ohne Visualisierung

Die Typen ,,*Pointer*" und ,,*Heap Pointer*" dienen beide zur Visualisierung von Referenzen. ,,*Heap Pointer*" erzeugt jedoch einen kleinerer Kasten und sein Inhalt ist nicht zu sehen, da Hinter- und Vordergrundfarbe identisch sind. Dieser Typ ist so definiert, um die graphische Repräsentation der Halden-Objekte klein zu halten. Ein Vektor besteht zum Beispiel aus einer Reihe von Referenzen und ein Vektor mit nur vier Einträgen hätte bei Verwendung des Typs ,,*Pointer*" eine Größe von fast 300 Bildpunkten. Die Zellen der Referenzen, die im Keller liegen, erhalten als Typ ,,*Pointer*" damit alle Zellen des Kellerspeichers dieselbe Größe besitzen. Sie lassen sich jedoch in der Farbe voneinander unterscheiden.

Zellen des Typs ,,*Dummy*" erhalten eine Höhe und eine Breite von 0 Bildpunkten und sind demzufolge nicht für den Betrachter zu erblicken. Solche Zellen werden bei manchen Halden-Objekten verwendet, um die interne Repräsentation zu verbergen. (siehe z.B. Vektoren Kap. 7.2.5 und Befehl `REWRITE` Kap. 7.2.8)

In der Implementierung der **MaMa**-Befehle wird im Gegensatz zur **CMa** von der Möglichkeit der Speicherverwaltung, Typinformation bei Wertzuweisung und Kopieroperation zu verwenden, Gebrauch gemacht. Hierzu wird einfach der gewünschte Typ der Ziel-Zelle als zusätzliches Argument beim Methoden-Aufruf übergeben.

Zum Beispiel:

```
memory.store( sourceAddress, destAddress, "Pointer" );
```

statt:

```
memory.store( sourceAddress, destAddress );
```

7.2.3 Operationen

Implementiert sind auch hier die binären arithmetischen und logischen Operationen `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `AND`, `OR` und `XOR`, sowie die Vergleiche `EQ`, `NEQ`, `LE`, `LEQ`, `GE` und `GEQ` sowie die einstelligen Operationen `NEG` und `NOT`. Diese Operator-Befehle und deren Implementierungen sind identisch zu denen der **CMa**.

7.2.4 Sprung-Befehle

Die Sprung-Befehle `JUMP` und `JUMPZ`, sowie `HALT` zum Beenden der Programmausführung, können ebenfalls aus der **CMa** übernommen werden. Die Version `JUMPI` für indizierte Sprünge gibt es bei der `MaMa` nicht.

7.2.5 Implementierung der Halden-Objekte

Die **MaMa** besitzt eine Reihe von Daten-Objekten:

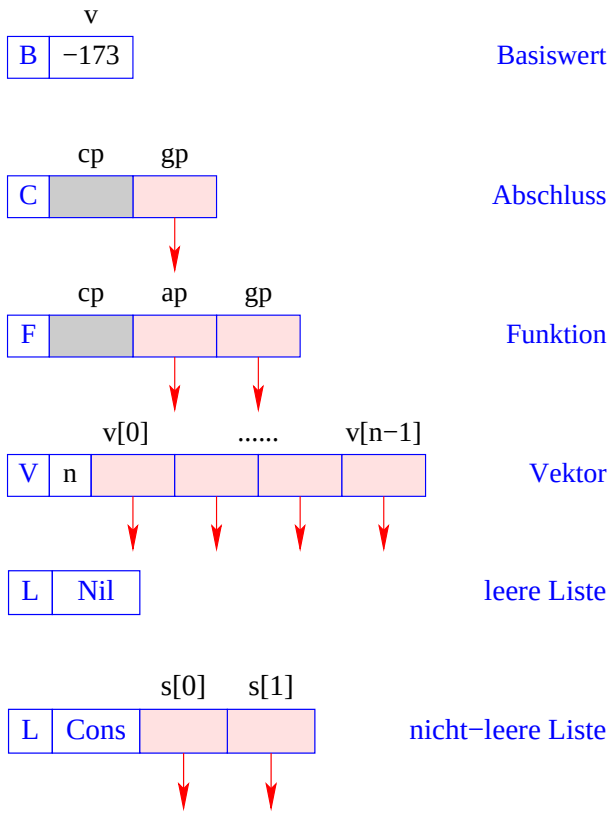


Abbildung 7.16: Daten-Objekte der **MaMa**

Die **MaMa**-Definition beschreibt zwar die verwendeten Objekte. Sie sagt aber nichts darüber aus, wie diese Objekte im Speicher abgelegt werden. Zur Nutzung der automatisierten Visualisierungsmöglichkeiten des **VAM**-Systems ist jedoch eine Zuordnung zu Speicheradressen notwendig. In dieser Beispiel-Implementierung erhalten alle Halden-Objekte eine Zelle für das Kennzeichen (*Tags*: `B`,`C`,`F`,`V`,`L`) und dann je eine Zelle für jeden weiteren Eintrag nach der **MaMa**-Definition.

Es tritt das folgende Problem auf. Der `REWRITE`-Befehl überschreibt einen Abschluss, also ein *Closure*-Objekt, durch ein anderes Objekt, ohne die Referenzen auf die Objekte ändern zu dürfen, d.h. das neue Objekt muss an der Adresse, an der zuvor der Abschluss gespeichert war, abgelegt werden. Es muss allerdings auch der benötigte zusammenhängende Speicherplatz ab dieser Adresse zur Verfügung stehen, um die verschiedenen Objekte, die von unterschiedlicher Größe sein können, ablegen zu können. Da alle Halden-Objekte mit Ausnahme der Vektoren in vier Zellen gespeichert werden können, werden beim Anlegen der Abschlüsse immer vier

Zellen reserviert. Um nun auch ein „*rewrite*" auf Vektoren zu ermöglichen, werden die *Vector*-Objekte der **MaMa** aufgespalten in einen Kopf, der in vier Zellen zu speichern ist , und einen Rumpf. Der Kopf enthält das Kennzeichen „`V`", die Anzahl der Einträge und einen Verweis auf den Rumpf. Dieser enthält dann die Einträge selbst. Der Kopf benötigt also drei Zellen und stellt somit für den `REWRITE`-Befehl kein Problem dar. Der Rumpf darf an einer beliebigen Adresse abgelegt werden und ist nicht in seiner Größe beschränkt. Ein Vektor kann also beliebig viele Einträge besitzen.

Die interne Darstellung eines Vektors muss bei allen Befehlen, die Vektoren erzeugen oder auf diese zugreifen, berücksichtigt werden. Die Visualisierung wird dahingehend angepasst, dass dem Betrachter die Zweiteilung verborgen bleibt. Hierzu wird die Zelle mit dem Verweis auf den Rumpf nicht visualisiert und die graphische Repräsentation des Rumpfes direkt hinter der des Kopfes dargestellt (siehe Befehl `MKVEC` auf Seite 64).

In der **MaMa**-Definition wird zur Generierung der Halden-Objekte die Notation `new(`*`Tag`*`,`*`value1`*`,`*`value2`*`,...)` verwendet. *Tag* gibt den Typ des Objekts an und durch *valueX* werden die weiteren Werte dieses Objekts gesetzt. In meiner Implementierung wird Speicherplatz durch `alloc(`*`types`*`)` reserviert. *types* ist dabei ein `String`-Feld mit den gewünschten Typen für die zu reservierenden Speicherzellen. Der Rückgabewert ist die Anfangsadresse des reservierten Speicherbereichs, dem dann innerhalb der `execute()`-Methode Werte zugewiesen werden können.

Der Befehl `MKBASIC` legt ein *Basic*-Objekt mit dem Wert des obersten Keller-Elements an. Die oberste Keller-Zelle wird dann durch eine Referenz auf das angelegte Objekt ersetzt.

Definition von `MKBASIC`:

```
S[SP] = new(B,S[SP]);
```

execute():

```
String[] types = {"Tag","Basic"};  // Zellen-Typen
int address = memory.alloc(types); // Platz reservieren
memory.setValue( address,"B" );    // Tag setzen
stack.popToAddress( address+1 );   // Basic-Wert setzen
stack.push( address );             // Referenz auf Objekt kellern
memory.setType( SP(),"Pointer" );  // Referenz markieren
```

Der Befehl `MKCLOS` wird zur Generierung von Abschlüssen verwendet. Er legt ein *Closure*-Objekt mit der als Parameter angegebenen Programm-Adresse und der auf dem Keller liegenden Referenz an. Das oberste Keller-Element wird auch hier durch eine Referenz auf das angelegte Objekt ersetzt.

Definition von `MKCLOS A`:

```
S[SP] = new(C,A,S[SP]);
```

execute():

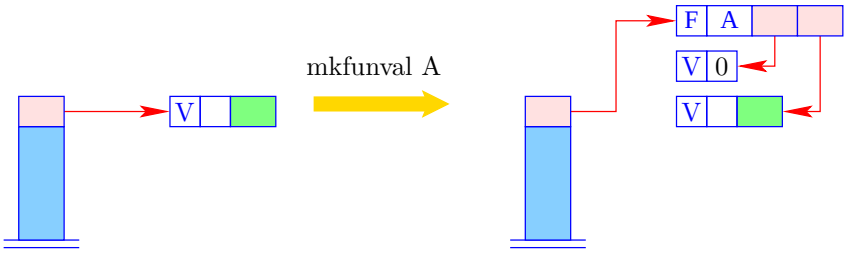
```
String[] types = {"Tag","Code Pointer","Heap Pointer","Dummy"};
int closureAddress = memory.alloc(types); // Platz reservieren
memory.setValue( closureAddress,"C" );    // Tag setzen
memory.setIntValue( closureAddress+1,A ); // Wert setzen
stack.popToAddress( closureAddress+2 );   // Wert setzen
stack.push( closureAddress );             // Referenz kellern
memory.setType( SP(),"Pointer" );         // Referenz markieren
```

Ein *Closure*-Objekt besteht eigentlich nur aus drei Zelle. Aufgrund der Implementierung des `REWRITE`-Befehls muss jedoch eine vierte Zelle angelegt werden. Diese wird durch Verwendung des Zellentyps „*Dummy*" jedoch nicht visualisiert und somit ergibt sich für den Betrachter kein Unterschied.

Durch `ALLOC` wird die angegebene Anzahl an *Closure*-Objekten generiert. Dies geschieht in einer Schleife wobei die Generierung der Einzelobjekte analog zu `MKCLOS` abläuft.

Der Befehl `MKFUNVAL` erzeugt zuerst einen Vektor ohne Elemente und generiert dann ein *Funval*-Objekt bestehend aus der übergebenen Programm-Adresse, einer Referenz auf den leeren Vektor sowie der auf dem Keller liegenden Referenz. Anschließend wird wie bei allen `MKxxx`-Befehlen die oberste Keller-Zelle durch eine Referenz auf das erzeugte Objekt ersetzt.

Definition von `MKFUNVAL A`:



a = new (V,0);
S[SP] = new (F, A, a, S[SP]);

execute():

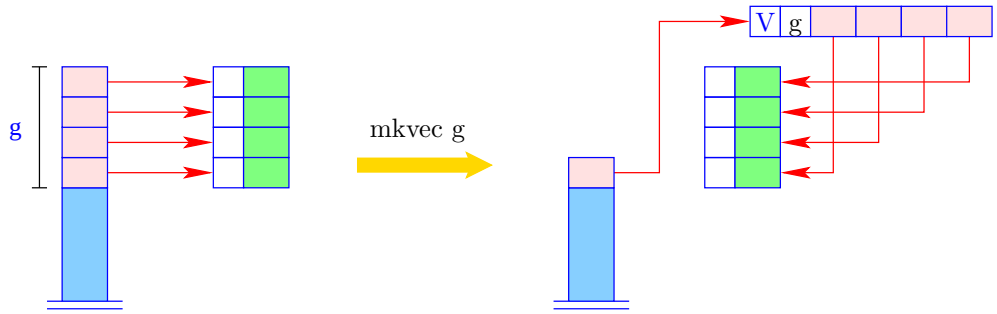
```
// einen leeren Vector erzeugen
String[] typesVector = {"Tag","counter"}; // Zellen-Typen
int a = memory.alloc( typesVector );      // Platz reservieren
memory.setValue( a,"V" );                 // Tag setzen
memory.setIntValue( a+1,0 );              // leerer Vektor

// Funval-Objekt erzeugen
String[] types={"Tag","Code Pointer","Heap Pointer","Heap Pointer"};
int adr = memory.alloc( types );  // Platz reservieren
memory.setValue( adr,"F" );       // Tag setzen
memory.setIntValue( adr+1,A );    // Wert setzen
memory.setIntValue( adr+2,a );    // Referenz auf leeren Vektor
stack.popToAddress( adr+3 );      // Wert setzen
stack.push( adr );                // Referenz auf Objekt kellern
memory.setType( SP(),"Pointer" ); // Referenz markieren
```

Der Befehl `WRAP` erzeugt analog zu `MKFUNVAL` ein *Funval*-Objekt und belegt es dann mit Werten.

Die Implementierung des Befehls `MKVEC` ist komplexer als die der vorangegangen. Er generiert nicht einfach Vektoren der angegebenen Länge, da der `REWRITE`-Befehl nicht mit beliebig großen Objekten umgehen kann. Vektoren werden aufgespalten in einen Kopf mit Kennzeichen (*Tag* „`V`"), der Anzahl der Einträge und einem Verweis auf den Rumpf. Der Rumpf enthält dann die Feldeinträge des Vektors. Es erfolgt eine Anpassung der Darstellung der Elemente, so dass es für den Betrachter den Anschein hat, der Vektor sei ein Objekt.

Definition von `MKVEC n`:



h = new (V, n);
SP = SP - g + 1;
for (i=0; i<g; i++)
  h→v[i] = S[SP + i];
S[SP] = h;

execute():

```
String[] types;
int address = -1;       // Initialisierung sonst Fehlermeldung
if ( number < 0 )
  mamaError("Trying to allocate Vector [V,"+number+",_].");
else
  { // Kopf des Vektors
    types = new String[3];                // Zellen-Typen
    types[0] = "Tag";
    types[1] = "Counter";
    types[2] = "Dummy";
    address = memory.alloc( types );      // Platz für Kopf
    memory.setValue( address,"V" );       // Tag setzen
    memory.setValue( address+1, number ); // Vektorgröße setzen
    if( number == 0 )   // kein Rumpf
        memory.setValue( address+2, -1 );
    else
    {
```

```
        // Rumpf des Vektors
        types = new String[number];         // Zellen-Typen
        for( int i=0; i<number; i++ ) types[i] = "Heap Pointer";
        // Platz für Rumpf reservieren; Zellen unsichtbar
        int address2 = memory.alloc( types, Memory.INVISIBLE );
        // Referenz auf Rumpf im Kopf eintragen
        memory.setIntValue( address+2,address2 );
        // Visualisierung anpassen
        CellGroup group = memory.getCell( address2 ).getCellGroup();
        // Rumpf hinter Kopf schieben und sichtbar machen
        group.setLocation(memory.getLocation(address+2));
        group.fadeIn();
        // eine Gruppe für kompletten Vektor
        group.add( memory.getCell( address ) );
        group.add( memory.getCell( address+1 ) );
        group.add( memory.getCell( address+2 ) );

        // PointerModes anpassen - sonst ist WEST möglich - unschön
        memory.getCell( address2 ).setPointerMode
          (Pointer.SOUTH|Pointer.NORTH,Pointer.NORTH|Pointer.SOUTH);
        // Vektor füllen
        memory.store( SP()-number+1,address2,number,Memory.NOCOPY );
      }
  }
  stack.decrease( number );         // Werte vom Stack entfernen
  stack.push( address, "Pointer" ); // Referenz auf Objekt kellern
```

Der Zelle, die den Verweis auf den Rumpf des Vektors enthält, wird wieder der Zellen-Typ „*Dummy*", der keine Visualisierung erzeugt, zugewiesen. Die Zelle ist für den Betrachter also nicht zu sehen.

Der Rumpf wird zunächst als eigenes Objekt mit `alloc(types,Memory.INVISIBLE)` generiert und eine Referenz darauf in der entsprechenden Kopfzelle eingetragen. Es wird ein unsichtbares Objekt erzeugt, da die Position noch geändert werden soll. Hierzu wird erst die Gruppe, zu der die Rumpf-Elemente gehören, durch `getCellGroup()` ermittelt, um diese dann mittels `setLocation(..)` an die Position des „*Dummy*"-Elements zu versetzen. Die Gruppe kann nun für den Betrachter visualisiert werden. Dies geschieht bei neu erzeugten Objekten üblicherweise durch `fadeIn()`. Auch hier wird die Methode auf die gesamte Gruppe angewendet.

Jetzt folgen noch einige Korrekturen. Zum einen erscheint der Vektor zwar jetzt in der Darstellung als ein Objekt. Nutzt der Betrachter jedoch die Möglichkeit, dieses Objekt interaktiv zu verschieben, so wird er den Kopf und den Rumpf voneinander getrennt verschieben können. Dies wird verhindert, indem den Kopf-Zellen dieselbe

Gruppe zugewiesen wird wie den Rumpf-Zellen. Alle Zellen des Vektor sind nunmehr in einer Gruppe. Die Gruppenzugehörigkeit der „*Dummy*"-Zelle ist eigentlich unwichtig, da diese ohnehin nie zu sehen ist. Sie wird aber der Vollständigkeit halber auch gesetzt.

Die zweite Korrektur betrifft die Positionierung des Zeigers der ersten Rumpfzelle. Da der Rumpf zunächst als eigenständiges Objekt angelegt wurde, hat das System den Ansatz eines Zeiger an der linken Seite der Zelle erlaubt. Dort befindet sich nun aber der Kopf des Vektors. Durch `setPointerMode(..)` werden für diese Zelle die möglichen Ansatzpunkte für Zeiger auf oben und unten (`NORTH | SOUTH`) beschränkt.

Nun ist der Vektor in der gewünschten Form generiert und die Wertzuweisung kann mit der zugehörigen automatisch generierten Visualisierung statt finden.

Am Ende von `execute()` wird die Kellerhöhe um die Feldelemente des Vektors nach unten korrigiert und wieder eine Referenz auf das neue Objekt gekellert.

Es bleiben noch die beiden zur Verwendung von Listen notwendigen Objekttypen. Zum einen die leere Liste *Nil* und zum anderen *Cons* mit zwei Verweisen auf den linken und den rechten Teil der Liste.

Der Befehl `NIL` legt einen Verweis auf eine leere Liste auf den Keller.

Definition von `NIL`:

```
S[SP] = new(L,Nil);
SP++;
```

execute():

```
String[] types={ "Tag","List Tag" }; // Zellen-Typen
int address = memory.alloc( types ); // Platz reservieren
memory.setValue( address, "L" );     // Tag setzen
memory.setValue( address+1, "Nil" ); // Listen-Tag setzen
stack.push( address );               // Referenz auf Objekt kellern
memory.setType( SP(),"Pointer" );    // Referenz markieren
```

Der Befehl `CONS` konsumiert die Verweise auf den linken und den rechten Teil der Liste vom Keller und kellert eine Referenz auf die erzeugte Liste.

Definition von `CONS`:

```
S[SP] = new(L,Cons,S[SP-1],S[SP]);
SP--;
```

execute():

```
String[] types = { "Tag","List Tag","Heap Pointer","Heap Pointer" };
int address = memory.alloc( types );  // Platz reservieren
memory.setValue( address, "L" );      // Tag setzen
memory.setValue( address+1, "Cons" ); // Listen-Tag setzen
memory.store( SP()-1,address+2,2,Memory.NOCOPY ); // Werte setzen
stack.decrease( 2 );                  // Stack korrigieren
stack.push( address );                // Referenz auf Objekt kellern
memory.setType( SP(),"Pointer" );     // Referenz markieren
```

7.2.6 Laden von Werten und Objekten

Laden bedeutet auch hier immer das Kellern eines Wertes bzw. mehrerer Werte. Bei Objekten wird ein Verweis auf ein Objekt in der obersten Keller-Zelle erwartet und dieser durch den Inhalt des referenzierten Objekts überschrieben. Die Befehle und die zu kellernden Werte sind:

MaMa-Befehl	**zu ladende(r) Wert(e)**
`LOADC c`	Konstante c
`GETBASIC`	Inhalt eines *Basic*-Objekts
`GETVEC k`	alle k Einträge eines *Vector*-Objekts
`GET i`	i-ter Eintrag eines *Vector*-Objekts
`PUSHLOC i`	Wert einer lokalen Variablen
`PUSHGLOB i`	Wert einer globalen Variablen
`TLIST A`	Einträge einer Liste (setzt zusätzlich PC neu)

Der Ablauf ist bei allen Befehlen sehr ähnlich, wobei einige Befehle nicht alle Schritte benötigen:

1. Überprüfung des referenzierten Objekts.
2. Bestimmung der Adresse.
3. Kellern des Wertes bzw. der Werte durch `STORE` oder `PUSH`.

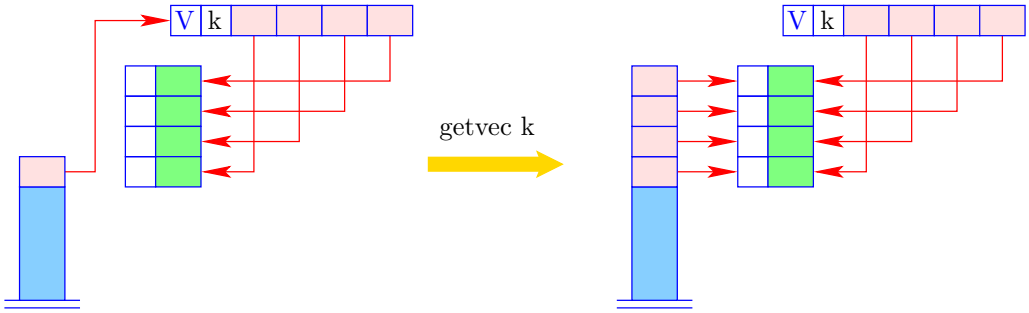
Stellvertretend sollen nun der einfachste und der aufwendigste dieser Befehle erklärt werden.

Das Laden von Konstanten auf den Keller erfolgt analog zur **CMa** durch den Befehl `LOADC`. Dieser wird lediglich um eine Typzuweisung ergänzt. Hier fällt selbstverständlich Schritt eins und zwei der obigen Auflistung weg.

MaMa-Befehl	MaMa-Definition	execute()
`LOADC c`	`SP++;` `S[SP] = c;`	`stack.push( c, "int" );`

`GETVEC` zum kellern der Einträge eines Vektors benötigt jedoch alle Schritte.

Definition von `GETVEC`:



if (S[SP] == (V,k,v)) {
  SP--;
  for(i=0; i<k; i++) {
    SP++; S[SP] = v[i];
    }
} else Error “Vector expected!”;

execute():

```
int addressHead = memory.getIntValue( SP() );
String tag = memory.getValue( addressHead );
if( ! tag.equals("V") )
    mamaError("Vector expected ! "+tag+" found" );
else
{
  // Anzahl der Einträge des Vektors ermitteln
  int k = memory.getIntValue( addressHead+1 );
  if ( number == k )
  { // angegebene und tatsächliche Größe stimmen überein
    if ( 0 < number ) ;
```

```
      {
        int addressBody = memory.getIntValue( addressHead+2 );
        memory.store( addressBody, SP(), number, "Pointer" );
      }
      stack.setSize( SP()+number-1 ); // Stackhöhe korrigieren
    }
    else
     mamaError("Vector [V,"+number+",_] expected [V,"+k+",_] found");
  }
```

Zuerst erfolgt die Überprüfung des Kennzeichens des referenzierten Objekts. Ist dies nicht „`V`" für einen Vektor wird eine Fehlermeldung generiert. Anschließend wird getestet, ob die Anzahl der Einträge des Vektors mit der erwarteten Anzahl übereinstimmt. Ist auch dies der Fall, so kann die Adresse der Einträge, also die Adresse des Vektor-Rumpfes (`addressBody`), ermittelt und der Kopiervorgang durch `STORE` gestartet werden. Eine Schleife wie in der Definition wird nicht benötigt, weil durch Angabe der Anzahl „`number`" alle Einträge durch eine Aufruf von `STORE` gespeichert werden. Da die zu ladenden Einträge alle Referenzen sind, wird beim Kellern den entsprechenden Zellen der Typ „*Pointer*" zugewiesen. Anschließend erfolgt noch die Korrektur der Kellerhöhe.

7.2.7 Befehle zur Unterstützung von Funktionen

Der Keller-Rahmen zur Ausführung einer Funktion wird wie bei der **CMa** durch den Befehl `MARK` angelegt. Die drei organisatorischen Zellen werden auch hier farblich hervorgehoben.

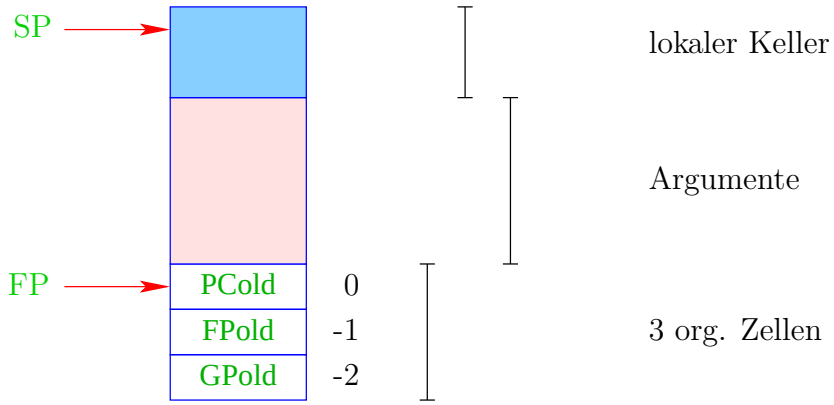


Abbildung 7.17: Kellerrahmen der **MaMa**

Das Gegenstück ist der Befehl `POPENV`. Er baut den Keller-Rahmen ab, korrigiert die Register-Inhalte und springt in die aufrufende Funktion zurück. In der Implementierung ist der zu setzende Wert für den *Frame Pointer* in der Hilfsvariablen `newFP` zu speichern, da nach dem Setzen des *Stack Pointers* die Zellen oberhalb des `SP` nicht mehr zugänglich sind aber der benötigte Wert in diesem Bereich liegt.

MaMa-Befehl	**MaMa-Definition**	**execute()**

`MARK A`	`S[SP+1] = GP;`
	`S[SP+2] = FP;`
	`S[SP+3] = A;`
	
	`FP = SP = SP+3;`
		`stack.push( GP(), "Pointer" );`
		`stack.push( FP(), "Pointer" );`
		`stack.push( A , "int" );`
		`stack.highlight(3,Color.magenta);`
		`FP( SP() );`

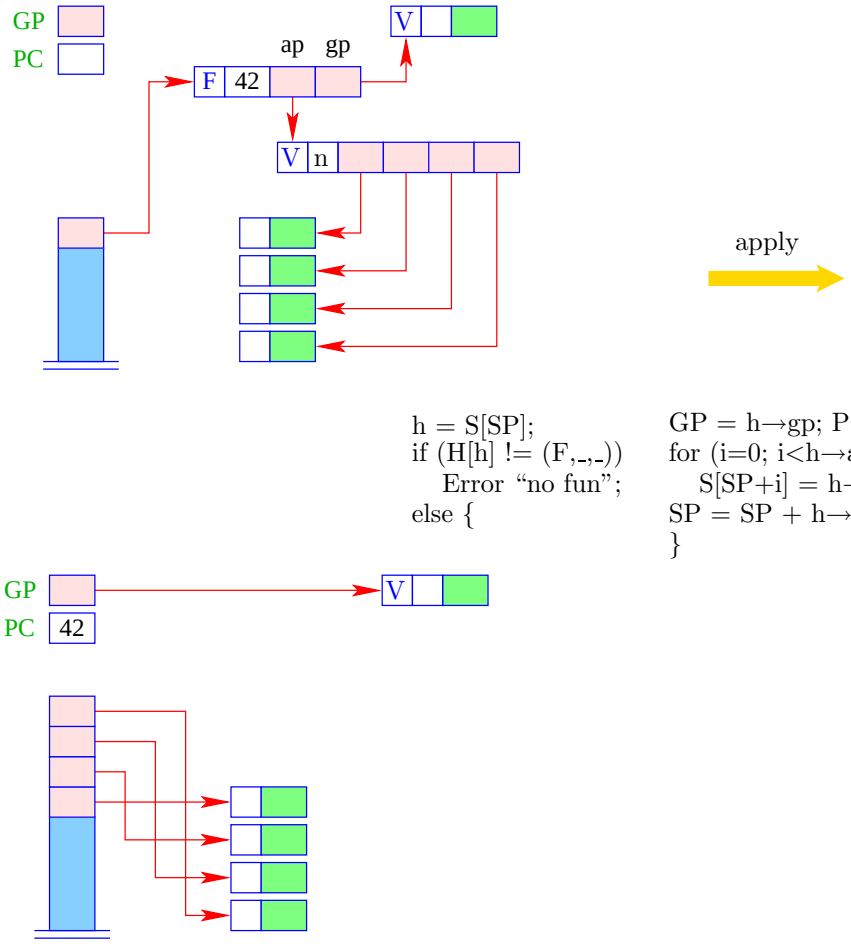
`POPENV`	`GP = S[FP-2];`
	`S[FP-2] = S[SP];`
	
	`PC = S[FP];`
	
	`SP = FP-2;`
	`FP = S[FP-1];`
		`GP( memory.getIntValue(FP()-2) );`
		`memory.store`
		`( SP(),FP()-2,Memory.NOCOPY );`
		`PC( memory.getIntValue( FP() ) );`
		`int newFP =`
		`memory.getIntValue( FP()-1 );`
		`SP( FP()-2 );`
		`FP( newFP );`

`APPLY0`	`h = S[SP];`
	`SP--;`
	`GP = h→gp;`
	`PC = h→cp;`
		`int h = stack.popInt();`
		
		`GP( memory.getIntValue( h+2 ) );`
		`PC( memory.getIntValue( h+1 ) );`

Die Abarbeitung der Funktion wird durch `APPLY` gestartet. Dieser Befehl muss das oben auf dem Keller liegende „*Funval*"-Objekt auspacken, d.h. die Argumente des *Argument Pointers* kellern und den *Global Pointer* setzen, und dann an der angegebenen Programm-Adresse fortfahren.

Der Befehl `APPLY0` ist ein Spezialfall von `APPLY`. Hier werden keine Argumente auf den Keller gelegt.

Definition von `APPLY`:



execute():

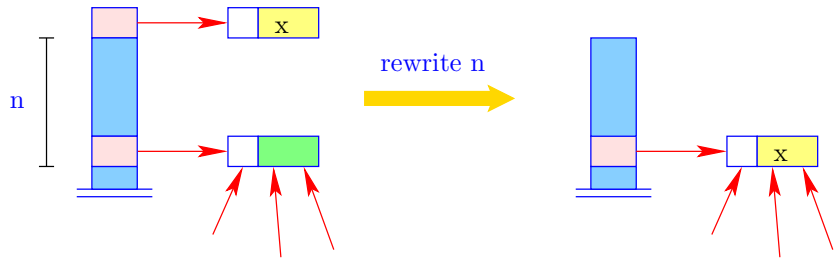
```
int h = memory.getIntValue( SP() );
String tag = memory.getValue(h);
if ( !tag.equals("F") )
    mamaError( "no fun ! ["+tag+"]" );
else
{
  GP( memory.getIntValue( h+3 ) );          // h->gp
  PC( memory.getIntValue( h+1 ) );          // h->cp
  int ap = memory.getIntValue( h+2 );       // h->ap
  int number = memory.getIntValue( ap+1 ); // h->ap->n
  if ( number > 0 )
```

```
    { // Vektor-Rumpf kopieren
      int bodyAddress = memory.getIntValue( ap+2 );
      memory.store( bodyAddress, SP(), number, "Pointer" );
      stack.increase( number-1 );
    }
    else stack.pop();
  }
```

7.2.8 Der Befehl REWRITE

Diesem Befehl ist ein eigener Abschnitt gewidmet, da er trotz seiner kurzen Definition die Implementierung der abstrakten Maschine **MaMa** stark beeinflusst.

Definition von `REWRITE n`:



H[S[SP-n]] = H[S[SP]];
SP = SP - 1;

Hinter der nur zwei Zeilen umfassenden Definition steckt ein mächtiger Befehl. Eine Kopie des Objekts auf das die oberste Keller-Zelle verweist, wird an die Adresse eines anderen Objekts kopiert. Die zweite Adresse stammt ebenfalls aus dem Keller und zwar aus der n-ten Zelle unter dem obersten Element. Zuerst erfolgt die Bestimmung des Typs des zu kopierenden Objekts und dann wird der entsprechende Kopiervorgang eingeleitet.

execute():

```
int sourceAddress = memory.getIntValue( SP() );
int destinationAddress = memory.getIntValue( SP()-n );
String tag = memory.getValue( destinationAddress );
if ( !tag.equals("C") )
  mamaError( "REWRITE Error: no Closure but ["+tag+"] found." );
else
{
  tag = memory.getValue( sourceAddress );
  if ( tag.equals("V") )
```

```
{
  int vecNumber = memory.getIntValue( sourceAddress+1 );
  // zweigeteilter Vektor
  // Kopf des Vektors
  memory.store( sourceAddress,destinationAddress,2 );
  memory.setType( destinationAddress+1, "Counter" );
  memory.setType( destinationAddress+2, "Dummy" );
  if( n == 0 )   // kein Rumpf
    memory.setValue( destinationAddress+2, -1 );
  else
  {
    // Rumpf des Vektors
    String[] types = new String[vecNumber];   // Zellen-Typen
    for( int i=0; i<vecNumber; i++ ) types[i] = "Heap Pointer";
    // Platz für Rumpf reservieren; Zellen unsichtbar
    int address2 = memory.alloc( types, Memory.INVISIBLE );
    // Referenz auf Rumpf im Kopf eintragen
    memory.setIntValue( destinationAddress+2,address2 );

    // Rumpf hinter Kopf schieben
    CellGroup group = memory.getCell( address2 ).getCellGroup();
    group.setLocation(memory.getLocation(destinationAddress+2));
    // eine Gruppe für kompletten Vektor
    group.add( memory.getCell( destinationAddress ) );
    group.add( memory.getCell( destinationAddress+1 ) );
    group.add( memory.getCell( destinationAddress+2 ) );
    // PointerModes anpassen - sonst ist WEST möglich - unschön
    memory.getCell( address2 ).setPointerMode
        (Pointer.SOUTH|Pointer.NORTH,Pointer.NORTH|Pointer.SOUTH);
    // Vektor mit Werten füllen
    memory.store(memory.getIntValue(sourceAddress+2),address2,vecNumber);
    // Rumpf sichtbar machen
    group.setVisible( true );
  }
}
else if ( tag.equals("B") )
{
  memory.store( sourceAddress,destinationAddress,2 );
  memory.setType( destinationAddress+1, "Basic" );
  memory.setType( destinationAddress+2, "Dummy" );
}
else if ( tag.equals("C") )
    memory.store( sourceAddress,destinationAddress,3 );
else if ( tag.equals("F") )
```

```
    {
      memory.setType( destinationAddress+3, "Heap Pointer" );
      memory.store( sourceAddress,destinationAddress,4 );
    }
  else mamaError( "REWRITE Error: Can't rewrite ["+tag+"]." );
  stack.pop();
```

Visualisiert werden nur die Zellen, die auch für den Betrachter bestimmt sind. Werden nicht alle vier zu „*rewrite*"-Zwecken reservierten Zellen benötigt, so erscheinen die nicht verwendeten Zellen durch Zuweisung des Zellen-Typs „*Dummy*" auch nicht in der Visualisierung.

Soll durch `REWRITE` ein Vektor kopiert werden, so wird das *Closure*-Objekt durch den Vektor-Kopf überschrieben und eine Kopie des Rumpfes rechts daneben visualisiert. Dies ist der Implementierung des Befehls `MKVEC` sehr ähnlich.

7.2.9 Weitere Befehle

MaMa-Befehl	**MaMa-Definition**	**execute()**
`COPYGLOB`	`SP++;` `S[SP] = GP;`	 `stack.push( GP(), "Pointer" );`

`SLIDE n`	`S[SP-n] = S[SP];`
	
	`SP = SP-n;`
		`String type`
		`  = memory.getType(SP());`
		`memory.store( SP(), SP()-n`
		`  , Memory.NOCOPY, type );`
		`SP( SP()-n );`

`MOVE r k`	`SP = SP-k-r;`
	`for( i=1; i≤k; i++ )`
	`  S[SP+i]=S[SP+i+r];`
	`SP = SP+k`
		
		`memory.store(SP()-k+1`
		`  ,SP()-r-k+1,k,Memory.NOCOPY);`
		`SP( SP()-r );`

Es gibt noch einige Befehle, deren Definitionen andere Befehle verwenden, also aus diesen zusammengesetzt sind. Die Implementierung lässt sich durch Einfügen des jeweiligen Programmtextes der anderen Befehle erzeugen (*Code Inlining*). Solche Befehle sind `TARG`, `EVAL`, `UPDATE` und `RETURN`.

Alternativ kann durch

```
viewMachine.createCommand("vam.machines.mama","BEFEHL").execute();
viewMachine.sync();
```

zur Laufzeit die Instanz eines Befehls erzeugt und dann aufgerufen werden. Die beiden Parameter sind das Paket, welches den Befehl enthält, und der Name des Befehls. Der Aufruf von `sync()` dient zur Synchronisierung der Visualisierung, d.h. es wird gewartet bis die Animation des Befehls abgearbeitet ist.

7.2.10 Ein Programm-Beispiel

Hier ein **PuF**-Programm zur Berechnung der Fakultät der Zahl 4.

```
letrec
  f = fn x => if x <= 1 then 1
              else (x * (f (x-1)))
in f 4
```

Die Übersetzung von **PuF** durch einen Compiler von Prof. Dr. H. Seidl [PSI-Trier] liefert folgendes Programm für die abstrakte Maschine **MaMa**:

```
    /* 0 */    alloc 1
    /* 1 */    pushloc 0  /* f */
    /* 2 */    mkvec 1
    /* 2 */    mkfunval _0
    /* 2 */    jump _1
_0:
    /* 0 */    targ 1
    /* 0 */    pushloc 0  /* x */
    /* 1 */    getbasic
    /* 1 */    loadc 1
    /* 2 */    leq
    /* 1 */    jumpz _2
    /* 0 */    loadc 1
    /* 1 */    mkbasic
    /* 1 */    jump _3
_2:
    /* 0 */    pushloc 0  /* x */
    /* 1 */    getbasic
    /* 1 */    mark _4
    /* 4 */    pushloc 4  /* x */
    /* 5 */    getbasic
    /* 5 */    loadc 1
    /* 6 */    sub
    /* 5 */    mkbasic
    /* 5 */    pushglob 0 /* f */
    /* 6 */    apply
_4:
    /* 2 */    getbasic
    /* 2 */    mul
    /* 1 */    mkbasic
_3:
    /* 1 */    return 1
_1:
    /* 1 */    rewrite 1
    /* 1 */    mark _5
    /* 4 */    loadc 4
    /* 5 */    mkbasic
    /* 5 */    pushloc 4  /* f */
    /* 6 */    apply
_5:
    /* 2 */    slide 1
    /* 1 */    halt
```

Die vorderen Kommentare geben den Kellerpegel an und die hinteren die Variablennamen.

Eine ausführliche, mit *Screenshots* dokumentierte Beschreibung der visualisierten Abarbeitung dieses Programms wäre zu aufwendig. Daher werden hier lediglich drei Momentaufnahmen gezeigt.

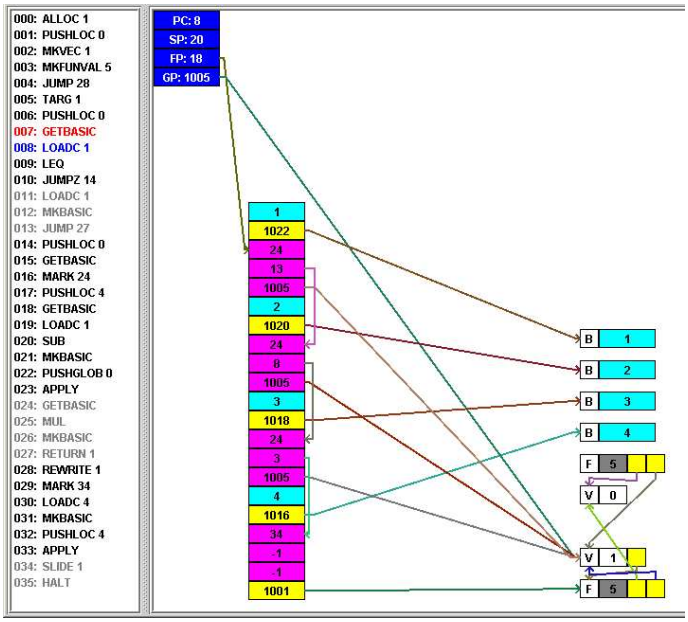


Abbildung 7.18:

In Abbildung 7.18 befindet sich die Maschine in der vierten Rekursion. Gut zu erkennen sind die vier Kellerrahmen in der Bildmitte und die vier zugehörigen, durch Verweise gekennzeichneten, *Basic*-Objekte rechts in der Halde. Das *Vector*-Objekt, auf welches das Register `GP` (*Global Pointer*) verweist, beinhaltet nur einen Eintrag, und zwar das *Funval*-Objekt, das der Funktion `fac` entspricht. Dies ist das einzige globale Objekt.

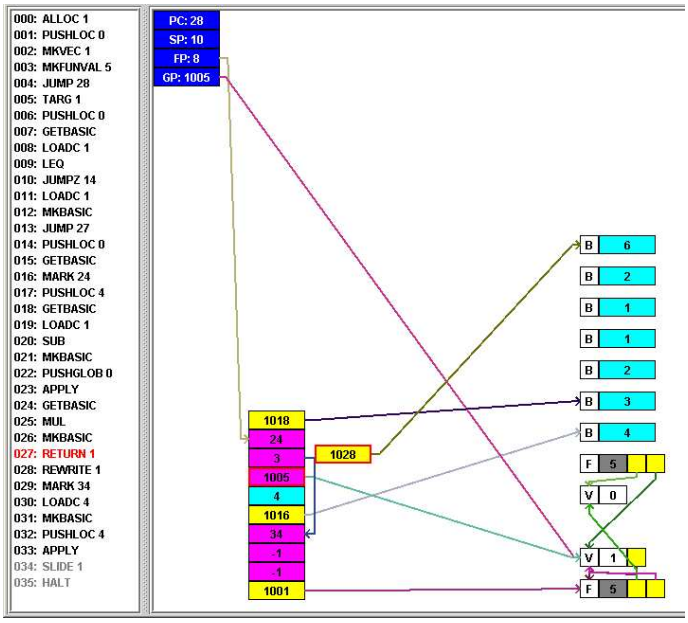


Abbildung 7.19:

In Abbildung 7.19 zeigt eine Szene während des Abbaus der Rekursionstiefe und damit auch des Kellers (Befehl `RETURN 1`). In der Halde sind die *Basic*-Objekte mit den Werten der bereits berechneten Fakultäten von 1, 2 und 3 zu sehen.

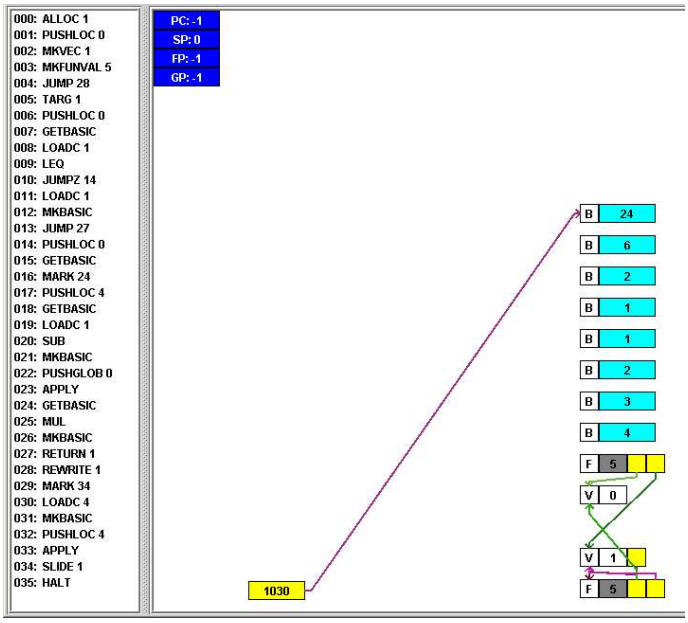


Abbildung 7.20:

Die Abbildung 7.20 zeigt das vollständig abgearbeitete Programm. Die einzige Kellerzelle ist ein Verweis auf die berechnete Fakultät der Zahl 4. Da die **MaMa** die angelegten Objekte nicht entfernt und sie auch keine *Garbage Collection* verwendet, sind eine Reihe von Halden-Objekten zu erkennen, auf die keine Verweise mehr existieren.

8

Zusammenfassung und Ausblick

Die Beispiele des vorangegangenen Kapitels zeigen, dass sich abstrakte Maschinen oder vielmehr die Vorgänge bei der Abarbeitung von Programmen dieser Maschinen, durch das **VAM**-System visualisieren lassen. Hierzu ist die Maschine für das **VAM**-System zu implementieren. Durch die Verwendung der Datenstrukturen des **VAM**-Pakets wird die Generierung der Visualisierung weitestgehend automatisiert, kann jedoch bei Bedarf auch durch den Animationsentwickler erweitert werden, um Kontextinformationen einzubringen (siehe Kap. ziel).

Der Betrachter kann interaktiv in das Geschehen eingreifen und hat somit die Möglichkeit, die Visualisierung seinen Bedürfnissen anzupassen, etwa durch Bestimmung der Ablaufgeschwindigkeit oder durch Abänderung der Zelleninhalte.

In den folgenden Abschnitten werden nun noch Vorschläge zur Erweiterung des Systems gemacht. Der erste betrifft die Anordnung der dargestellten Komponenten und dient zur Verbesserung der Visualisierung. Der zweite Vorschlag zielt auf eine Erleichterung der Implementierung abstrakter Maschinen und der letzte betrifft die Maschinen-Programme, die als Eingabe der Visualisierung dienen.

8.1 Layout

Ein gutes Layout trägt zur Verbesserung der Visualisierung bei. Zusammengehörige Zellen sollten als solche zu erkennen sein, sich nicht überdecken und auch die Zeiger

sollten sich möglichst selten kreuzen. Für den Kellerspeicher ist die Darstellung recht einfach, da dessen Elemente nach dem FIFO-Prinzip hinzugefügt bzw. entfernt werden. Eine gute Anordnung der Halden-Objekte der **MaMa** zu finden ist deutlich komplizierter, vor allem da diese untereinander durch Zeiger verknüpft sind.

Mein **VAM**-System positioniert die einzelnen Halden-Objekte einfach in der Reihenfolge ihrer Generierung im linken Bereich der Darstellung (Abb. 8.2). Bei der Visualisierung der Zeiger wird zwar versucht, aus den möglichen Ansatzpunkten einen auszuwählen, der eine gute Darstellung ermöglicht (Kap. 6.1.3 S. 25), es erfolgt jedoch keine Berücksichtigung der bereits vorhandenen Zeiger oder Zellen, um Überschneidungen zu verhindern.

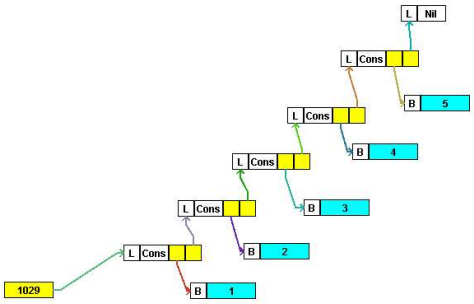


Abbildung 8.1: Verbessertes Layout

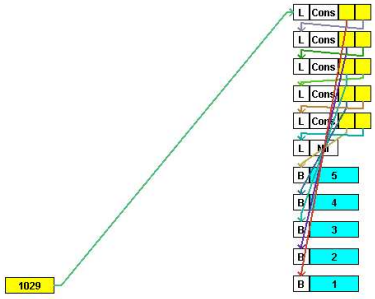


Abbildung 8.2: normales Layout

Der Betrachter besitzt zwar durch interaktives Verschieben die Möglichkeit, die Zellen individuell zu positionieren, ein Layout-Manager, der alle Positionierungen von Zellen und Zeigern verwaltet, könnte allerdings automatisch eine bessere Darstellung generieren. Abbildung 8.1 zeigt, wie eine verbesserte Darstellung aussehen könnte. Ausserdem könnte die Notwendigkeit zur Verwendung der Zeiger-Modis eingeschränkt werden bzw. ganz entfallen.

Die zusätzliche Berücksichtigung des abzuarbeitenden Programms könnte das Layout weiterhin verbessern. Auch besteht die Möglichkeit Visualisierungskomponenten nicht nur bei ihrer Generierung zu positionieren, sondern sie nachträglich zu verschieben, wenn dadurch eine Verbesserung der Visualisierung erlangt werden kann. Hier ist allerdings zu beachten, dass eine häufige Neupositionierung die Visualisierung auch negativ beeinflussen kann, da der Betrachter zwischen Layout-Korrekturen und eigentlicher Animation unterscheiden muss.

Um das Layout des **VAM**-Systems zu verbessern ist in folgenden Methoden anzusetzen:

Klasse	**Methode**	**Funktion**
`Memory`	`calculateLocation`	Zellenpositionierung
`Pointer`	`calculatePointer`	Zeigerberechnung

8.2 Abstraktionsgrad der Maschinen

Abstrakte Maschinen besitzen einen unterschiedlich hohen Abstraktionsgrad. Bei der **CMa** werden Keller und Halde in einem großen Speicherbereich untergebracht und es ist bereits in der Definition dieser Maschine beschrieben, wie diese Datenstrukturen zu verwalten sind. Es gibt zu diesem Zweck sogar eigene Register (*Extreme Pointer*).

In Der **MaMa**-Definition hingegen wird über die Halde nur gesagt, dass es eine Datenstruktur ist, die Objekte bestimmter Typen aufnehmen kann. Über eine Abbildung auf Speicheradressen wird keine Aussage gemacht. Um die Vorzüge der automatisierten Generierung der Visualisierung durch die Speicherschnittstelle des **VAM**-Systems nutzen zu können, bedarf es jedoch einer Adresszuordnung. Wenn eine Objekt als Ganzes einer Zelle zugeordnet wird und die Adresse als Objekt-Referenz betrachtet wird, so lässt sich der hohe Abstraktionsgrad der Maschine beibehalten. In der bisherigen Implementierung des Systems ist dies durch Verwendung einer String-Repräsentation für Objekte denkbar, aber es besteht dann keine Möglichkeit mehr, die einzelnen Komponenten eines Objekts innerhalb der Zelle optisch durch Farbe oder Größe voneinander zu trennen. Es wird nur der dem Objekt entsprechende Text innerhalb der Zelle angezeigt. Und die Möglichkeit zur Visualisierung von Zeigern für einzelne Objekt-Komponenten geht ebenfalls verloren, da einer Zelle immer nur genau ein Zeiger zugeordnet werden kann.

Um diese Problematik zu lösen, ist es denkbar das Visualisierungs-System dahingehend zu ändern, dass eine Verschachtelung der Zellen möglich ist. Ein Objekt kann dann aus einer beliebigen Anzahl an Zellen unterschiedlicher graphischer Repräsentation und mit der vollen Zeigerfunktionalität bestehen. Und diese Zellen werden dann in ihrer Gesamtheit einer weiteren Zelle zugeteilt, die dann wiederum einer Speicheradresse zugeordnet wird. Die Einzelkomponenten der Objekte können dann wie bisher visualisiert werden, aber der Abstraktionsgrad erhöht sich.

8.3 Code-Generatoren

Die Eingabe zur Visualisierung besteht aus einem Programm in der Sprache der entsprechenden abstrakten Maschine. Die Erstellung eines solchen Programms von Hand ist umständlich. Einfacher verhält sich dies bei Hochsprachen, die für den Menschen leichter verständlich sind. Für den Benutzer des Visualisierungssystems ist es daher vorteilhaft, wenn er ein Programm aus einer Hochsprache automatisiert in ein Programm für eine implementierte abstrakte Machine umsetzen kann und dieses dann visualisieren lässt. Hierzu werden Übersetzer (*Compiler*) benötigt, deren Implementierung nicht Teil dieser Diplomarbeit war.

Die Entwicklung von Code-Generatoren erfolgt jedoch zur Zeit an den Universitäten Saarbrücken [GANIMAL] und Trier [PSI-Trier].

Ein einfacher von Prof. Dr. H. Seidl [PSI-Trier] geschriebener Übersetzer von **PuF** nach **MaMa** ist auf der CD zu dieser Diplomarbeit enthalten.

Literaturverzeichnis

[Seidl2000] Prof. Dr. Helmut Seidl.
Skript zur Vorlesung Abstrakte Maschinen.
Universität Trier, Sommersemester 2000.

[Wilh1997] Reinhard Wilhelm und Dieter Maurer.
Übersetzerbau: Theorie, Konstruktion, Generierung.
Springer, 1997.

[GANIMAL] Das Ganimal Projekt.
www.cs.uni-sb.de/RW/users/ganimal.
Universität des Saarlands, Saarbrücken, 2000.

[Zlot2000] Oliver Zlotowski.
Diplomarbeit: Design und Implementierung eines Systems zur Animation von Algorithmen und Datenstrukturen.
Halle, 2000.

[PSI-Trier] Lehrstuhl für Programmiersprachen und Compiler.
www.informatik.uni-trier.de/PSI
Universität Trier.

[Sun1.2000] JavaTM 2 SDK, Standard Edition.
Documentation, Version 1.3.
java.sun.com, 2000.

[Sun2.2000] The JavaTM Tutorial.
java.sun.com, 2000.

A

Anhang

A.1 vam.base.Cell

Klasse zur Darstellung von Zellen.

```
package vam.base;
public class Cell extends JLabel implements MouseListener

public static final boolean POINTER     = true;
public static final boolean NOPOINTER   = false;
public static final boolean VISIBLE     = true;
public static final boolean INVISIBLE   = false;
```

Standardwerte:

```
public static int stdWidth = 60;
public static int stdHeight= 20;
static Color stdBackgroundColor = Color.cyan;
static Color stdForegroundColor = Color.black;
static Border stdBorder = stdBorder();
static Border stdHighlightedBorder = stdHighlightedBorder();
```

Konstruktoren:

```
public Cell( String value, String type, int x,int y, boolean visible )
public Cell( String value, int x, int y, boolean visible )
```

```
public Cell( String value, int x, int y )
public Cell( String value, String type,Point location,boolean visible)
public Cell( String value, Point location, boolean visible )
public Cell( String value, Point location )
public Cell( String value, boolean visible )
public Cell( String value )
```

Erzeugt eine neue Zelle.
`value` : Wert der Zelle
`type`: Visualisierungs-Typ der Zelle
`x`: x-Koordinate der Darstellung (default: 0)
`y`: y-Koordinate der Darstellung (default: 0)
oder `location`: Koordinaten der Zelle. (default: (0,0))
`visible`: `true` wenn die Zelle direkt sichtbar sein soll (ohne Einblenden) (default: `true`)

```
public static void init( ViewMachine viewMachine )
```

Initialisierung. Muß vor der Erzeugung der ersten Zelle aufgerufen werden.
`viewMachine`: gibt die zugehörige Laufzeitumgebung an.

```
public CellGroup getCellGroup()
```

Liefert Gruppierung, zu der die Zelle zugeordnet ist.

```
void setAddress( int address )
```

Festsetzen der Adresse die im Tool-Tip angezeigt wird soll. Diese sollte mit der der Speicherverwaltung übereinstimmen
`address`: Adresse

```
void updateToolTipText()
```

Neuberechnung des Tool-Tips.

```
public void setValue( String value )
public void setValue( int value )
public void setIntValue( int value )
```

Setzt den Wert der Zelle.
`value`: neuer Wert

```
public String getValue()
public int getIntValue()
```

Liefert den Wert der Zelle.

```
public Cell copy()
```

Liefert eine Kopie der Zelle und (falls vorhanden) ein Kopie des zugehörigem Zeigers.

```
public void remove()
```

Entfernt Visualisierung der Zelle (mit ihres Zeigers).

```
void removePointer()
```

Entfernt Visualisierung des Zeigers der Zelle.

```
void setPointer( Pointer pointer )
```

Setzt den Zeiger der Zelle.
`pointer`: Neuer Zeiger

```
public void setPointerMode( int outMode,int inMode )
```

Setzt den Modus für ein- und ausgehende Zeiger.
`outMode`: Modus für ausgehende Zeiger.
`inMode`: Modus für eingehende Zeiger.

```
void updatePointer()
```

Aktualisiert den Zeiger der Zelle.

```
public boolean isPointer()
```

Liefert `true` falls die Zelle im Zeigermodus ist.

```
public void isPointer( boolean isPointer )
```

Setzt den Zeigermodus der Zelle.
`isPointer`: `true` setzt den Zeigermodus

```
void pointTo( Cell destination )
```

Erzeugt einen Zeiger für diese Zelle.
`destination`: Gibt die Zelle an, auf die gezeigt werden soll

```
public void highlight( boolean highlight )
```

Erzeugt und entfernt optische Hervorhebung der Zelle (roter Rahmen).
`highlight`: `true` setzt optische Hervorhebung

```
public void fadeIn()
```

Die Zelle wird eingeblendet.

```
public void fadeOut()
```

Die Zelle wird ausgeblendet.

```
public void moveTo( Point destination )
public void moveTo( int x, int y )
public void moveTo( Cell destination )
```

Visualisierung einer Bewegung der Zelle.
`destination`: Zielkoordinaten der Bewegung
oder
`x`: x-Zielkoordinate der Bewegung
`y`: y-Zielkoordinate der Bewegung
oder
`destination`: Zielzelle, zu der die Bewegung erfolgen soll

```
public void moveBy( int x, int y )
public void moveBy( Point distance )
```

Visualisierung einer Bewegung der Zelle.
`x`: x-Abstand zu den Zielkoordinaten der Zelle
`y`: y-Abstand zu den Zielkoordinaten der Zelle
oder
`distance`: Abstand zu den Zielkoordinaten der Zelle

```
public void setColor( Color color )
```

Setzt die (Hintergrund-)Farbe derZelle.
`color`: neue Farbe

```
public Color getColor()
```

Liefert die (Hintergrund-)Farbe derZelle.

```
public static void defineType( String type, Color color,
    Color textColor, int width, int height, boolean isPointer,
    int pointerOutMode, int pointerInMode )
public static void defineType( String type, Color color,
    Color textColor, int width, int height, boolean isPointer )
public static void defineType( String type, Color color,
    int width, int height, boolean isPointer )
public static void defineType( String type, Color color,
    int width, int height )
public static void defineType( String type, Color color )
public static void defineType( String type, Color color,
    boolean isPointer )
```

Definiert einen neuen Typ für die Darstellung von Zellen
`type`: Name für diesen Typ
`color`: Zellenfarbe
`textColor`: Schriftfarbe
`width`: Breite
`height`: Höhe
`isPointer`: tt true falls die Zelle ein Zeiger sein soll `pointerOutMode`: Zeiger-Modus für ausgehende Zeiger
`pointerInMode`: Zeiger-Modus für eingehende Zeiger

```
public void setType( String type )
```

Setzt den Typ der Zelle und passt Die Zelle diesem Typ an.
`type` neuer Typ der Zelle

```
public String getType()
```

Liefert den Typ der Zelle.

```
public void sync()
```

Blockiert bis alle (Visualisierungs-)Aktionen dieser Zelle beendet sind.

```
public static void syncAll()
```

Blockiert bis alle Aktionen aller Objekte beendet sind.

```
static void sleep( int msec )
```

Verzögert die Ausführung der Visualisierung.
`msec` Zeit in Millisekunden, die die Visualisierung verzögert werden soll

```
static Border stdBorder()
```

Liefert einen Standard-Rahmen.

```
static Border stdHighlightedBorder()
```

Liefert einen Standard-Rahmen für die optische Hervorhebung einer Zelle.

Benutzer-Interaktion

```
public void mousePressed(MouseEvent e)
```

Speichert Startpunkt für eventuelle Verschiebung.

```
public void mouseReleased(MouseEvent e)
```

Führt interaktive Verschiebung aus.

```
public void mouseClicked(MouseEvent e)
```

Bewirkt interaktive Änderung des Zellenwerts.

```
public void mouseEntered(MouseEvent e)
public void mouseExited(MouseEvent e)
```

Ohne Bedeutung.

TypeData

Diese Klasse dient zur Speicherung der Typinformation für Zellen.

```
class TypeData
```

```
TypeData( Color backgroundColor, Color foregroundColor,
    int width, int height,
    boolean isPointer, int pointerOutMode, int pointerInMode )
```

`backgroundColor`: Zellenfarbe
`foregroundColor`: Schriftfarbe
`width`: Breite
`height`: Höhe
`isPointer`: tt true falls die Zelle ein Zeiger sein soll `pointerOutMode`: Zeiger-Modus für ausgehende Zeiger
`pointerInMode`: Zeiger-Modus für eingehende Zeiger

A.2 vam.base.CellGroup

Gruppierung von Zellen.

```
package vam.base;
public class CellGroup extends ArrayList
```

Konstruktoren:

```
public CellGroup( String name )
public CellGroup()
```

Erzeugt neue Gruppierung.
`name`: Name dieser Gruppierung

```
public void add( Cell cell )
```

Fügt der Gruppierung eine Zelle hinzu.
`cell`: hinzuzufügende Zelle

```
public void moveTo( Point destination )
public void moveTo( int x, int y )
public void moveTo( Cell destination )
```

Visualisierung einer Bewegung der Zellengruppe.
`destination`: Zielkoordinaten der Bewegung
oder
`x`: x-Zielkoordinate der Bewegung
`y`: y-Zielkoordinate der Bewegung
oder
`destination`: Zielzelle zu der die Bewegung erfolgen soll

```
public void moveBy( int x, int y )
public void moveBy( Point distance )
```

Visualisierung einer Bewegung der Zellengruppe.
`x`: x-Abstand zu den Zielkoordinaten der Bewegung
`y`: y-Abstand zu den Zielkoordinaten der Bewegung
oder
`distance`: Abstand zu den Zielkoordinaten der Bewegung

```
public void highlight( boolean highlight )
```

Erzeugt und entfernt optische Hervorhebung der Gruppierung
`highlight`: `true` setzt optische Hervorhebung

```
public void fadeIn()
public void fadeOut()
```

Die Gruppierung wird ein- bzw. ausgeblendet.

```
public void setVisible( boolean visible )
```

Bestimmt, ob die Gruppierung sichtbar ist oder nicht
`visible`: durch `true` ist die Gruppierung sichtbar

```
public void translate( int x, int y )
public void translate( Point distance )
```

Setzt eine neue Position für diese Gruppierung relativ zur alten fest (ohne Animation).
`x`: x-Abstand zu den Zielkoordinaten der Verschiebung
`y`: y-Abstand zu den Zielkoordinaten der Verschiebung
oder
`distance`: Abstand zu den Zielkoordinaten der Verschiebung

```
public void setLocation( int x, int y )
public void setLocation( Point location )
```

Setzt eine neue Position für diese Gruppierung relativ zur alten fest (ohne Animation).
`x`: x-Koordinate der neuen Position
`y`: y-Koordinate der neuen Position
oder
`location`: Koordinaten der neuen Position

A.3 vam.base.Register

Klasse zu Darstellung von Registern.

```
package vam.base;
public class Register extends Cell
```

Konstruktor:

```
public Register( String name )
```

Erzeugt ein neues Register.
`name` Name des Registers

```
static public void reset()
```

Entfernt alle Register.

```
public void set( int value )
public void setValue( int value )
public void setIntValue( int value )
```

Setzt den Wert des Registers
`value`: Neuer Wert

```
public String getValue()
public int getIntValue()
```

Liefert den Wert des Registers

```
public void mouseClicked(java.awt.event.MouseEvent e)
```

Bewirkt interaktive Änderung des Registerwerts.

A.4 vam.base.Text

Klasse zu Darstellung von Text.

```
package vam.base;
class Text extends JPanel
```

Konstruktoren:

```
Text( String text, Point location, Color color )
Text( String text, int x, int y, Color color )
Text( String text, int x, int y )
Text( String text, Point location )
```

`x`: x-Koordinate zur Positionierung des Textes
`y`: y-Koordinate zur Positionierung des Textes
oder `location`: Koordinaten zur Positionierung des Textes
`color`: Farbe des Textes

```
void setColor( Color color )
```

Setzt die Textfarbe.
`color`: Schriftfarbe

```
public void paintComponent( Graphics gfx )
```

Zeichnet den Text.

A.5 vam.base.Pointer

Klasse zu Darstellung von Zeigern.

```
package vam.base;
public class Pointer extends JPanel

public static final int NORTH = 1;
public static final int SOUTH = 2;
public static final int WEST  = 4;
public static final int EAST  = 8;
```

Konstruktor:

```
public Pointer( Cell source )
```

Erzeugt neuen Zeiger.
`source`: Zelle, von der der Zeiger ausgeht

```
static void reset()
```

Reset der Klasse.

```
void remove()
```

Entfernt den Zeiger.

```
void calculatePointer( Cell sourceCell, int sourceMode,
                  Cell destinationCell, int destinationMode )
```

Berechnet die Darstellung des Zeigers.
`sourceCell`: Zelle, von der der Zeiger ausgeht
`sourceMode`: Zeiger-Modus der Ursprungszelle
`destinationCell`: Zelle, auf die der Zeiger verweist
`destinationMode`: Zeiger-Modus der Zielzelle

```
Shape arrow( Point destination, int mode )
```

Liefert Pfeilspitze.
`destination`: Zielkoordinaten des Zeigers
`mode`: Zeigermodus der Zeilzelle

```
void update()
void updateAll()
```

Neuberechnung des Zeigers bzw. aller Zeiger.

```
public void paintComponent( Graphics gfx )
```

Zeichnen des Zeigers.

```
static int setMode( int mode, int a,int b,int c,int d )
```

Hilfsmethode zur Modus-Auswahl in der Priorität: a,b,c,d
`mode`: Zeiger-Modus
`a`,`b`,`c`,`d`: Modi der vier Seiten in Reihenfolge ihrer Priorität

A.6 vam.base.Command

Diese Klasse dient als Basis aller Befehle der zu visualisierenden abstrakten Maschine.

```
package vam.base;
public class Command
```

Konstruktor:

```
public Command( String command )
```

Erzeugt einen neuen (leeren) Befehl.

```
public void execute()
```

Wird aufgerufen, um den Befehl auszuführen.

```
public void initialize( ViewMachine viewMachine )
```

Wird aufgerufen, wenn die Abstrakte Maschine gestartet werden soll. Hier sollten sämtliche statischen Strukturen erzeugt werden.
`viewMachine` Gibt die Laufzeitumgebung an.

```
public String toString()
```

Liefert ein `String`-Repräsentation des Befehls.

A.7 vam.base.VisualizationAction

Diese Klasse dient als Basis aller VisualisierungsAktionen (Animation).

```
package vam.base;
public abstract class VisualizationAction
```

```
abstract void nextFrame()
```

Wird aufgerufen, um den nächste Schritt der Animation zu generieren.

A.8 vam.base.MoveAction

Diese Klasse erzeugt die animierte Bewegung einer Zelle.

```
package vam.base;
class MoveAction extends VisualizationAction
```

Konstruktor:

```
MoveAction( Cell cell, Point destination )
```

Erzeugt Bewegungs-Animation.
`cell`: Zu bewegende Zelle
`destination`: Koordinaten, zu denen die Zelle bewegt werden soll

```
void nextFrame()
```

Wird aufgerufen, um den nächste Schritt der Animation zu generieren.

A.9 vam.base.FadeAction

Diese Klasse erzeugt Animation zum Ein- und Ausblenden einer Zelle.

```
package vam.base;
class FadeAction extends VisualizationAction
```

Konstruktor:

```
FadeAction( Cell cell, boolean fadeIn )
```

Erzeugt Bewegungs-Animation.
`cell`: Zu animierende Zelle
`fadeIn`: `true` zum Einblenden, `false` zum Ausblenden

```
void nextFrame()
```

Wird aufgerufen, um den nächste Schritt der Animation zu generieren.

A.10 vam.base.Memory

Die Speicherverwaltung.

```
package vam.base;
public class Memory

public static final boolean COPY       = true;
public static final boolean NOCOPY     = false;
public static final boolean ASCENDING  = true;
public static final boolean DESCENDING = false;
public static final boolean VISIBLE    = true;
public static final boolean INVISIBLE  = false;
```

Konstruktor:

```
public Memory( ViewMachine vm )
```

Erzeugt die Speicherverwaltung.
`vm`: Die Laufzeitumgebung

Zellentyp und -inhalt:

```
public void setType( int address, String type )
```

Setzt den Typ der Adresse.
`address`: Adresse
`type`: neuer Zellen-Typ

```
public void getType( int address )
```

Liefert den Typ der Adresse.
`address`: Adresse

```
public void setValue( int address, String value )
public void setValue( int address, String value, String type )
public void setValue( int address, int value )
public void setValue( int address, int value, String type )
public void setIntValue( int address, int value )
public void setIntValue( int address, int value, String type )
```

Setzt den Wert der Adresse.
`address`: Adresse
`value`: neuer Wert der Adresse
`type`: neuer Zellen-Typ (default: `"undefined"` für bisher unbenutzte Adresse, ansonsten alter Typ der Adresse)

```
public String getValue( int address )
public int getIntValue( int address )
```

Liefert den Wert der Adresse.
`address`: Adresse

Speichern:

```
public void store( int sourceAddress, int destinationAddress )
public void store( int sourceAddress, int destinationAddress,
                   String type )
public void store( int sourceAddress, int destinationAddress,
                   int number )
public void store( int sourceAddress, int destinationAddress,
                   int number, String type )
public void store( int sourceAddress, int destinationAddress,
                   boolean copy )
public void store( int sourceAddress, int destinationAddress,
                   boolean copy, String type )
public void store( int sourceAddress, int destinationAddress,
                   int number, boolean copy )
public void store( int sourceAddress, int destinationAddress,
                   int number, boolean copy, boolean ascending )
public void store( int sourceAddress, int destinationAddress,
                   int number, boolean copy, boolean ascending,
                   String type )
```

Speichert oder kopiert Inhalte von Adressen.
`sourceAddress`: Ursprungsadresse
`destinationAddress`: Zieladresse
`number`: Anzahl der zu kopierenden Werte
`copy`: `true` zum Kopieren, `false` zum Verschieben der Zellen
`ascending`: `true` aufsteigend, `false` absteigend speichern
`type`: neuer Zellen-Typ (default: `"undefined"` für bisher unbenutzte Adresse, ansonsten alter Typ der Adresse)

Hervorhebung:

```
public void highlight( int address, boolean highlight )
public void highlight( int address, boolean highlight, int number )
public void highlight( int address, Color color )
public void highlight( int address, Color color, int number )
```

Hebt Adressen/Zellen hervor.
`address`: Adresse
`number`: Anzahl der hervorzuhebenden Zellen
`highlight`: `true` falls die Zelle hervorgeheoben werden soll
oder
`color`: Die Farbe in der die Zelle hervorgehoben werden soll

Zeiger:

```
public void pointer( int address )
```

Markiert eine Speicherzelle als Zeiger und stellt diesen graphisch dar.
`address`: Adresse

```
public void removePointer( int address )
```

Entfernt die Markierung als Zeiger.
`address`: Adresse

```
public boolean isPointer( int address )
```

Liefert `true`, falls die Adresse als Zeiger markiert ist.
`address`: Adresse

Operationen:

```
public void unaryOperation( String operator, int addressOperand,
                            String newValue )
public void unaryOperation( String operator, int addressOperand,
                            boolean consumeOperand,
                            int addressResult, String newValue )
```

Generiert Visualisierung einer einstelligen Operation.
`operator`: Textdarstellung des Operators
`addressOperand`: Speicherzelle des Operanden
`consumeOperand`: `true`, falls der Operand konsumiert (entfernt) wird
`addressResult`: Speicherzelle, der das Ergebnis zugewiesen wird
`newValue`: der Wert, den die Speicherzelle für das Ergebnis nach der Operation enthalten soll

```
public void binaryOperation( String operator, int addressOperand1,
                     int addressOperand2, boolean consumeOperands,
                     int addressResult, String newValue )
```

Generiert Visualisierung einer zweistelligen Operation.
`operator`: Textdarstellung des Operators
`addressOperand1`: Speicherzelle des ersten Operanden
`addressOperand2`: Speicherzelle des zweiten Operanden
`consumeOperand`: `true`, falls der Operand konsumiert (entfernt) wird
`addressResult`: Speicherzelle, der das Ergebnis zugewiesen wird
`newValue`: der Wert, den die Speicherzelle für das Ergebnis nach der Operation enthalten soll

Layout:

```
Point calculateLocation( int address )
```

Berechnet und liefert Koordinaten der Darstellung einer Adresse.
`address`: Adresse

```
public Point getLocation( int address )
```

Liefert Koordinaten der Darstellung einer verwendeten Adresse oder berechnet diese für unverwendete Adresse.
`address`: Adresse

Verwaltung

```
public boolean exists( int address )
```

Liefert `true`, falls die Adresse verwendet wird.
`address`:

```
void free( int address )
```

Entfernt die Visualisierung dieser Adresse ohne Ausblenden.
`address`: Adresse

```
public String remove( int address )
```

Entfernt die Visualisierung dieser Adresse mit Ausblenden.
`address`: Adresse

```
public void setCell( int address, Cell cell )
public void setCell( int address, Cell cell, String type )
public void setCell( int address, Cell cell, boolean visible )
public void setCell( int address, Cell cell, boolean visible,
                                                  String type )
```

Ordnet einer Adresse eine Zelle zu.
`address`: Adresse
`cell`: Zelle
`type`: Typ zur Visualisierung der Zelle
`visible`: `true`, falls die Zelle sichtbar sein soll

```
public Cell getCell( int address )
```

Liefert Zelle, die die Adresse repräsentiert.
`address`: Adresse

```
public CellGroup getCellGroup( int address )
```

Liefert Gruppierung der Zelle, die die Adresse repräsentiert.
`address`: Adresse

Zusammenhängende Speicherbereiche:

```
public int alloc( String[] types )
public int alloc( String[] types, boolean visible )
public int alloc( int number )
public int alloc( int number, boolean visible )
```

Reserviert einen zusammenhängenden Speicherbereich und liefert dessen Anfangsadresse zurück.
`types`: Feld mit Zellen-Typen, die die Visualisierung der reservierten Zellen bestimmen; Feldgröße bestimmt Größe des zu reservierenden Bereichs
oder
`number`: Größe des zu reservierenden Bereichs
`visible`: `true`, falls die Zelle sichtbar sein soll

A.11 vam.util.Stack

Diese Klasse bietet den Kellerspeicher.

```
package vam.util;
public class Stack
```

Konstruktor:

```
public Stack( Memory memory )
```

Erzeugt einen neuen Keller und das Register SP für den Stackpointer.
`memory`: Gibt denn Speicher an, in dem der Keller liegen soll

```
public void push( String value, String type )
public void push( int value, String type )
public void push( String value )
public void push( int value )
public void push()
```

Legt eine neue oberste Stackzelle an.
`value` Inhalt der neuen obersten Stackzelle
`type` Typ zur Visualisierung der Zelle

```
public void pushFromAddress( int sourceAddress )
public void pushFromAddress( int sourceAddress, String type )
```

Legt eine Kopie einer Zelle auf den Keller.
`sourceAddress`: Adresse der zu kopierenden Zelle
`type` Typ zur Visualisierung der Zelle

```
public String pop()
public int popInt()
```

Entfernt oberste Zelle vom Keller und liefert deren Inhalt zurück.

```
public void popToAddress( int destinationAddress )
public void popToAddress( int destinationAddress, String type )
```

Entfernt oberste Zelle vom Keller und speichert deren Inhalt an einer anderen Adresse.
`destinationAddress`: Adresse an der der Inhalt der obersten Zelle gespeichert werden soll
`type` Typ zur Visualisierung der Zelle

```
public void store( int destinationAddress )
```

Speichert Kopie der obersten Stackzelle an einer anderen Adresse.
`destinationAddress`: Adresse an der die Kopie der obersten Zelle gespeichert werden soll

```
public int top()
```

Liefert die Adresse der obersten Stackzelle.

```
public boolean isEmpty()
```

Liefert `true` falls der Stack leer ist.

```
public void increase( int number )
```

Legt leere Zellen oben auf den Stack.
`number` Anzahl der Zellen die hinzugefügt werden sollen

```
public void decrease( int number )
```

Entfernt Zellen vom Stack (falls vorhanden).
`number` Anzahl der Zellen die entfernt werden sollen

```
public void setSize( int size )
```

Setzt die Größe des Stacks fest.
`size`: neue Größe des Stacks. Bei Bedarf werden Zellen entfernt bzw. neu hinzugefügt.

```
public void unaryOperation( String operator, int newValue )
public void unaryOperation( String operator, String newValue )
```

Stellt eine unäre Operation der obersten Stackzelle graphisch dar.
`operator`: gibt den Operator an
`newValue`: Wert, den die oberste Stackzelle nach der Operation enthalten soll

```
public void binaryOperation( String operator, int newValue )
public void binaryOperation( String operator, String newValue )
```

Stellt eine binäre Operation zwischen den obersten beiden Stackzellen graphisch dar.
Konsumiert die Operanden.
`operator`: gibt den Operator an
`newValue`: Wert, den die oberste Stackzelle nach der Operation enthalten soll

```
public void highlight( int number, boolean highlight )
public void highlight( int number, Color color )
```

Hebt die obersten Zellen des Stacks optisch hervor bzw. entfernt die Hervorhebung.
`number`: Anzahl der Zellen
`highlight`: `true` erzeugt `false` entfernt Hervorhebung
oder
`color` Farbe der Hervorhebung

A.12 vam.base.ViewMachine

Die Laufzeitumgebung und Kern des Pakets.

```
package vam.base;
public class ViewMachine implements ActionListener

public Register PC;
public Memory memory;
```

Konstruktor:

```
public ViewMachine()
```

Erzeugt Laufzeitumgebung mit Fenster zur Darstellung.

```
void addVisualization( JComponent component, int layer )
void addVisualization( JComponent component )
```

Fügt (statische) graphische Darstellung ein.
`component`: (statische) Komponente die eingefügt werden soll
`layer`: Ebene, in der Komponente eingefügt werden soll (default: unterschiedliche Default-Ebenen für unterschiedliche Objekte)

```
void addVisualization( VisualizationAction action )
```

Fügt (dynamische) graphische Darstellung ein.
`action` Aktion die eingefügt werden soll

```
void removeVisualization( JComponent component )
void removeVisualization( VisualizationAction action )
```

Entfernt graphische Darstellung.
`component` Komponente die entfernt werden soll
oder
`action` Aktion die entfernt werden soll

```
public void sync()
```

Blockiert bis alle Aktionen beendet.

```
static void sleep( int msec )
```

Unterbricht Abarbeitung.
`msec`: Zeit der Unterbrechung in Millisekunden

```
public static int parseInt( String arg )
```

Liefert int-Wert des Strings. Fehler (Exception) werden abgefangen.
`arg`: String der int-Wert enthält

Animation:

```
void initTimer( int fps )
```

Initialisiert Timer für Animation.
`fps`: Anzahl der Bilder, die pro Sekunde berechnet werden sollen.

```
public synchronized void runAnimation()
```

Lässt die Animation laufen.

```
public synchronized void stopAnimation()
```

Stoppt die Animation.

```
public void actionPerformed(ActionEvent e)
```

Bewirkt Berechnung des nächsten Bildes der Animation. Wird vom Animationtimer aufgerufen.

Oberfläche:

```
void createWindow()
JToolBar createToolBar()
JMenuBar createMenus()
JSlider createSlider()
static void createOutput()
```

Generierung des Ausgabe-Fensters und seiner Elemente.

```
void displayCode( ArrayList code )
```

Zeigt den Programm-Code an.
`code` Programm-Code

```
void highlightCodeLine( int PC )
void highlightCodeLine( int PC, Color color )
```

Hebt die Zeile hervor.
`PC`: hervorzuhebende Programmzeile
`color`: Farbe der Hervorhebung

Progammcode bearbeiten:

```
public ArrayList loadCode()
```

Lädt das auszuführende Programm aus einer Datei und liefert Programm-Code.

```
public ArrayList parseCode( ArrayList code, String baseDir )
```

Erzeugt eine Liste von Befehls-Objekten aus einem Programm.
`code`: Programm-Code als Liste von Strings
`baseDir`: Name der Pakets in dem die Command-Objekte definiert sind

```
public Command createCommand( String baseDir, String commandLine )
```

Erzeugt einen Befehl aus einer Befehlszeile.
`baseDir`: gibt das Verzeichnis an in dem die Befehlsklasse gesucht wird
`commandLine`: enthält die Befehlszeile mit Parametern

```
void showConstructors(Class c)
```

Zeigt alle Konstruktoren einer Klasse an.
`c`: Klasse deren Konstruktoren angezeigt werden sollen

```
Command createCommand( Class commandClass, Object[] arguments )
```

Erzeugt einen Befehl.
`commandClass`: Klasse des zu erzeugenden Befehls
`arguments`: Feld der Argumenttypen

```
Object[] parseArgumentToFitConstructor( Class commandClass,
Object[] arguments )
```

Wandelt Argumente passend zu einem vorhandenem Konstruktor um.
`commandClass`: Klasse des zu erzeugenden Befehls
`arguments`: Feld der Argumenttypen

Ausgabe:

```
public void error( String errorMessage )
```

Erzeugt eine Fehlermeldung.
`errorMessage`: Auszugebende Fehlermeldung

```
public static void output( String text )
```

Erzeugt eine Ausgabe.
`text`: Text der Ausgabe

System starten:

```
public void run()
public static void main(String[] args)
```

Startet das System.