Eberhard Karls
Universität
Tübingen

Lösungsskizze 8a: Strukturierte Analyse

Inhaltsverzeichnis

1 Bemerkungen zur verwendeten Notation

Die in dieser Lösungsskizze verwendete Notation ist in 3 Aspekten gegenüber der im Skript zur Vorlesung (Klaeren, 2002) erläuterten erweitert worden.

Die erste Erweiterung besteht darin, Datenflüsse, die im übergeordneten Diagramm nicht verwertet werden, im untergeordneten Diagramm in ein kleines, schraffiertes Rechteck zu führen (Block), so, wie den Datenfluss $Image_{DS}$ in DFD 1.2. Man kann sich natürlich fragen, wozu solche Datenflüssen überhaupt sichtbar gemacht werden sollten (wo sie doch gar nicht verwendet werden), und ob man sie nicht besser vollständig unterdrücken sollte. Nach meinem Dafürhalten gibt es zwei Situationen, in denen ein Einzeichnen solcher Datenflüsse spürbaren Gewinn bringt. Zum Einen dann, wenn die nichtbenötigten Daten im Verlauf eines Verarbeitungsschrittes (*Splitte Images*) praktisch sowieso anfallen – der zusätzliche Datenfluss hilft dann, zu illustrieren, was der Verarbeitungsprozess tut. Zum zweiten hilft der zusätzliche Datenfluss dabei, aus dem jeweiligen DFD für für einen späteren detaillierten Entwurf abstrakte Datentypen herauszulesen. So zeigt zum Beispiel DFD 1.2 (*Isoliere Codesegment*) einen Prozess, der später fast 1 : 1 in die Initialisierungsprozedur des abstrakten Datentyps *program* umgesetzt wird (siehe TeachSWT@Tü, 2002b), eine Tatsache, die sicher noch obskurer bleiben würde, wenn aus dem DFD wichtige Attribute der abstrakten Sicht eines Programms, nämlich $Image_{DS}$, abwesend wären.

Die zweite Erweiterung ist im Skript und in der einschlägigen Literatur schon angedeutet. Ich zitiere das Skript (Klaeren, 2002):

> Die Konvention besagt, daß Datenflüsse stets durch Substantive, gegebenenfalls in Kombination mit Adjektiven, zu bezeichnen sind. […] Ändern sich die bekannten Eigenschaften von Daten durch einen Prozeß, so ist es sinnvoll, diese geänderten Eigenschaften auch in der Benennung des ausgehenden Datenflusses klarzumachen […].

Dabei ist zu betonen, dass die Datenwörterbücher nur die Substantive aufschlüsseln, die Struktur der Daten sich also auch dem Substantiv richtet. Das Adjektive liefert lediglich weitere Informationen über die Bedeutung der Daten (*Kontonr.* vs. *richtige Kontonr.*), die vom Kontext abhängt, in dem Daten im Diagramm vorkommen.

In der dieser Lösungsskizze werden nicht nur Adjektive so gehandhabt, sondern auch Substantivteile, die in Klammern vor dem eigentlichen Datenelement stehen (etwa *(Lade-) Adresse*), Ergänzungen, die in Klammern hinter diesem stehen (etwa *Länge (CS)*), oder Text, der tiefgestellt hinter das Substantiv gesetzt wird (etwa $Image_{CS}$).

Die dritte und letzte Erweiterung betrifft die Bedeutung der Iterationsnotation „[…]“ in den Datenwörterbüchern. Ich habe den Eindruck gewonnen, dass – modern ausgedrückt – nicht immer klar ist, ob eine Iteration als gewöhnliche Menge (von Werten), oder aber als Multimenge (Elemente können mehrfach vorkommen, auch *Bag* genannt) zu interpretieren ist. Sicher ist auf jeden Fall, dass keine Reihenfolge von Elementen impliziert werden soll. Wird dennoch eine Reihenfolge benötigt, kann dies durch Hinzufügen eines entsprechenden Schlüssels ausgedrückt werden, wie im folgenden Beispiel:

$$Ausgabetext \quad = \quad \{\underline{Zeilen\text{-}Nr.} + Zeichenkette\}$$

McMenamin und Palmer (1984) führen für Teile eines Elementes der Iteration, die als Zugriffsschlüssel verwendet werden können (also als eine Angabe, die ein Element in der Iteration eindeutig identifiziert) die Notation ein, diesen Teil zu unterstreichen. Diese Notation (die im Skript nicht vorkommt), habe ich übernommen, wo es mir jeweils nützlich erschien.

Durch Hinzufügen entsprechender Schlüssel lässt sich auch jedes Element der Iteration eindeutig machen. Ich habe dies für die Lösungsskizze immer getan, um der Unterscheidung *Menge* versus *Multimenge* aus dem Weg zu gehen und klarer strukturierte Daten zu erhalten. Es sei bemerkt, dass Zugriffsschlüssel dieser Art nicht immer als Datenelement in der Implementation präsent sind. Zwar ist

$$\mathit{Ausgabetext} \quad = \quad \{\underline{\mathit{Zeilen\text{-}Nr.}} + \mathit{Zeichenkette}\}$$

aber die *Zeilen-Nr.* wird (implizit) in der Reihenfolge der Prozeduraufrufe implementiert. Die obige Datenbeschreibung soll nur klar machen, dass die ausgegebenen Zeilen in einer bestimmten Reihenfolge vorkommen und gleichzeitig ein Vokabular zur Verfügung stellen, damit man über die erste, zweite usw. Zeile sprechen kann.

2 Bemerkungen zum Entwurf

Lösungsskizze 5 (TeachSWT@Tü, 2002b) ist der *Entwurf* für ein Programm, das die hier *analysierten* Verarbeitungsprozesse durchführt. Die einschlägige Literatur führt verschiedene Verfahren auf, wie man Verarbeitungsprozesse auf Prozeduren abbildet und wie man aus den Grapheneigenschaften der Datenflussdiagramme Hauptprozeduren und untergeordnete Prozeduren bestimmt.

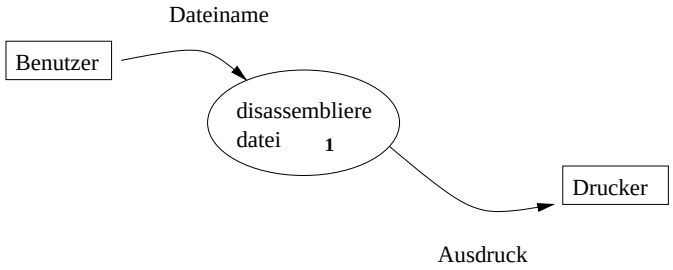
Ich habe für kleine Anwendungen gute Erfahrungen mit dem Ansatz erzielt, Programme aus abstrakten Datentypen zusammenzusetzen und datenverarbeitende Leistungen entweder in Attributfunktionen oder Initialisierern dieser abstrakten Datentypen unterzubringen. Ich kann jedoch (noch) kein definites Verfahren angeben, wie man von den Diagrammen der strukturierten Analyse zu den ADTs gelangt, dafür ist noch „geschicktes Hinsehen“ nötig. Was sicher auf diesem Weg hilft, ist, von den Daten auszugehen, d. h. die wichtigsten problemspezifischen Datentypen zu identifizieren, diese als ADTs zu schreiben und sich von da aus in die Programmperipherie vorzutasten. Ich bin versucht, dies (in Alternative zu *Bottom-Up* und *Top-down*) als *Inside-Outwards* oder *Essence-to-Environment* zu bezeichnen.

3 Datenflussdiagramme

3.1 Einleitung

Ich werde nun im Folgenden die Datenflussdiagramme, Minispezifikationen und Datenwörterbücher für die Disassemblierung des Mikrocontroller-Programms vorführen. Da die Minispezifikationen in natürlicher Sprache gehalten sind, erübrigen sich meist weitere Kommentare. Es sei noch darauf hingewiesen, dass ich, um das vorliegende Dokument lesbar zu halten, Daten immer dort aufschlüssele, wo sie zum ersten Mal benötigt werden. Für gewöhnlich – und das ist, wenn solche Dokumente als Referenz verwendet werden, auch besser so – sammelt man alle Datendefinitionen in einem eigenen Abschnitt, dem *Datenwörterbuch* oder *Data Dictionary*.

3.2 Übersicht, DFD 0



***Minispec* Disassembliere Datei, DFD 1:** Der *Benutzer* gibt den *Dateiname[n]* in das System ein. Die Datei mit dem gegebenen *Dateiname[n]* wird geöffnet, disassembliert und als *Ausdruck* auf den *Drucker* ausgegeben.

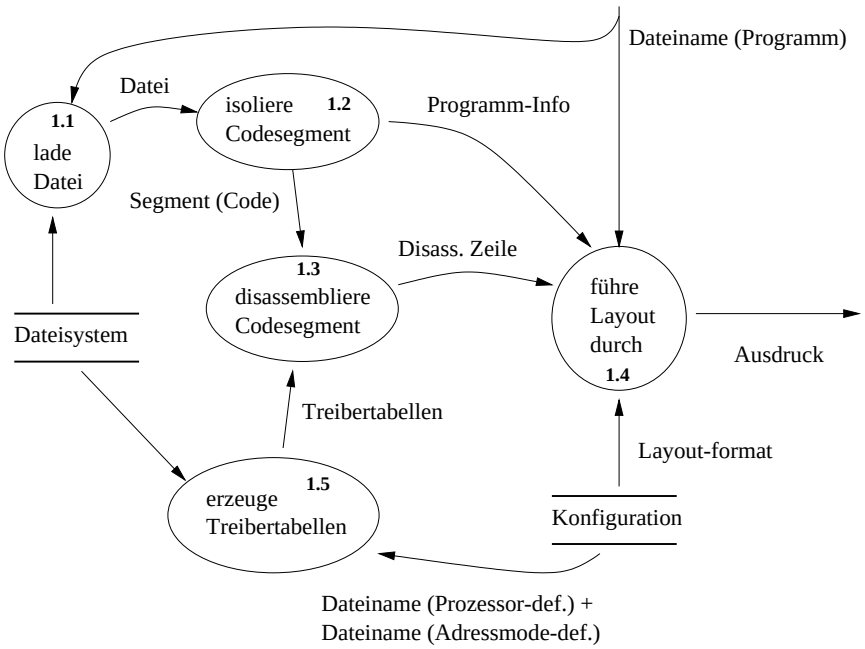
Data Dictionary

Dateiname : Gültige (wohlgeformte) Pfadangabe für Unix-Dateisystem.

Ausdruck : Strom von Kommandos der den Drucker steuert.

Bemerkung: Der Übersichts-DFD charakterisiert die Leistung des Systems als Ganzes, d. h. Hardware, Betriebssystem und Anwendung *zusammen*. Nur so ist es möglich, dass der Benutzer (als Terminator) und seine Eingaben (als Datenflüsse) explizit beschrieben werden.

3.3 Disassembliere Datei, DFD 1



Data Dictionary

Byte	:	8 Bit, ganze Zahl aus $[0, 256[$.
Byte-Sequenz	$=$	$\{\underline{\textit{Byte-No.}} + \textit{Byte}\}$
	$\star$	Folge von Bytes des Mikro-Controllers Max. Länge s. *Länge*.
Datei	$=$	*Byte-sequenz*
Dateisystem	$=$	$\{\underline{\textit{Dateiname}} + \textit{Datei}\}$

***Minispec* Lade Datei, DFD 1.1** Die Datei mit dem gegebenen *Dateiname[n]* wird geöffnet, und der in ihr enthaltene Bytestrom (hier als *Datei* bezeichnet) eingelesen.

Bemerkung: In der Praxis bedeutet das natürlich nicht, dass die Datei tatsächlich erst im Ganzen in ein Array gelesen wird, und dann an eine Prozedur übergeben wird, die den Prozess *Isoliere Codesegment* implementiert. Vielmehr besteht die Aufgabe des Prozesses *Lade Datei* darin, den Dateinamen in eine programminterne Darstellung (oder ein Handle) auf ein Array von Bytes zu übersetzen. Dies kann durchaus implizit geschehen, etwa dadurch, das der Inhalt der Datei an ein Prozedur indirekt mittels eines geöffneten Dateideskriptors übergeben wird (Vielleicht sollte der Prozess in diesem Fall besser als *Öffne Datei* bezeichnet werden).

***Minispec* Isoliere Codesegment, DFD 1.2** Das Codesegment ($Segment_{Code}$) wird aus der Datei ausgelesen.

***Minispec* Disassembliere Codesegment, DFD 1.3** Das Codesegment ($Segment_{Code}$) wird in eine Folge von Text-Darstellungen der einzelnen Maschineninstruktionen (*Disass. Zeile*) übersetzt. Dabei werden Informationen über den Befehlssatz des Prozessors (*Opcode[s]*) und die Adressierungarten benötigt. Diese stehen in entsprechenden *Treibertabellen* zur Verfügung.

***Minispec* Führe Layout durch, DFD 1.4** Die Folge von Textdarstellungen der Maschineninstruktionen (*Disass. Zeile*) werden so in Steuerkommandos an den Drucker übersetzt (hier als *Ausdruck* bezeichnet), dass die einzelnen Instruktionen wie gewünscht auf einer Folge von Seiten ausgegeben werden. Auf der ersten Seite werden allgemeine Informationen über das Programm (*Programm-Info*) und der Dateiname ausgegeben. Der Dateiname erscheint weiterhin in der Kopfzeile jeder einzelnen Seite. Angaben über das *Layout-Format* (Ränder, Spaltenbreiten, usw.) werden aus der *Konfiguration* des Programmes entnommen.

***Minispec* Erzeuge Treibertabellen, DFD 1.5** Die *Dateiname[n]* der Datei, welche die Definition der *Opcode[s]* enthält ($Dateiname_{Prozessor\text{-}def.}$), und der Datei, welche die Definition der Adressierungsarten (*Adressmode*) enthält, ($Dateiname_{Adressmode\text{-}def.}$), werden aus der Konfiguration des Programms entnommen. Aus den in beiden Dateien enthaltenen Tabellen werden die *Treibertabellen* für die Disassemblierung erzeugt.

Data Dictionary

$$
\begin{array}{lcl}
\textit{Programm} & = & \textit{Segment}_{Code} + \textit{Segment}_{Daten} \\
\textit{Segment} & = & \textit{Adresse}_{Ladeposition} + \textit{Byte-Sequenz} \\
\textit{Länge} & : & \text{Ganze Zahl im Intervall } [0, 16^4[\\
\textit{Adresse} & : & \text{Ganze Zahl im Intervall } [0, 16^4[\\
\textit{Programm-Info} & = & \textit{(Lade-) Adresse}_{DS} + \textit{Länge}_{DS} \\
 & + & \textit{(Lade-) Adresse}_{CS} + \textit{Länge}_{CS}
\end{array}
$$

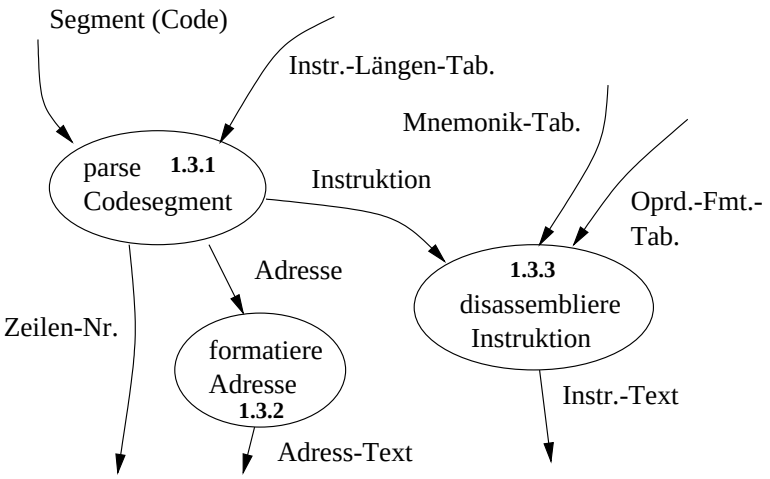
Die Definition der einzelnen *Treibertabellen* ist im nächsten Abschnitt zu finden.

Bemerkung: In den Datenflussdiagrammen kommt das Datum *Programm* nicht (mehr) vor. Ich habe es hier trotzdem in das Wörterbuch aufgenommen, da es nützlich zur Konzeptbildung und für den späteren Übergang auf einen Entwurf mit abstrakten Datentypen ist.

Data Dictionary

$$
\begin{array}{lcl}
\textit{Konfiguration} & = & \textit{Layout-Format} \\
 & + & \textit{Dateiname}_{Prozessor\text{-}def.} + \textit{Dateiname}_{Adressmode\text{-}def.}
\end{array}
$$

3.4 Disassembliere Codesegement, DFD 1.3



***Minispec* Parse Codesegement, DFD 1.3.1** Mit Hilfe der Informationen aus der Tabelle *Instr.-Längen-Tab.*, die angeben, aus wievielen Bytes der Operand besteht, der auf einen gegebenen *Opcode* folgt, wird $Segment_{Code}$ in eine Folge von *Instruktion[en]* unterteilt. Die *Adresse* an die die Instruktion später geladen wird, wird ermittelt. Die Zeilennummer *Zeilen-Nr.* wird für jede Instruktion jeweils um eins erhöht.

Bemerkung: Wie schon einleitend erläutert, wird die *Zeilen-Nr.* in der Implementation wahrscheinlich nicht explizit auftreten, sondern implizit der Reihenfolge der Prozeduraufrufe zu entnehmen sein (Jeweils im N-ten Aufruf wird die N-te Zeile übergeben bzw. als Ergebnis zurückgegeben).

***Minispec* Formatiere Adresse, DFD 1.3.2** Die *Adresse* wird in eine Textdarstellung umgewandelt (*Adress-Text*).

***Minispec* Disassembliere Instruktion, DFD 1.3.3** Die *Instruktion* wird in eine Textdarstellung umgewandelt (*Instr.-Text*). Dazu werden die Informationen in den Tabellen *Mnemonik-Tab.* und *Oprd.-Fmt.-Tab.* herangezogen, die angeben, als welcher *Mnemonik* ein gegebener Opcode dargestellt werden soll und mit welcher Notation die Adressierungsart dargestellt werden soll.

Data Dictionary

$$
\begin{array}{lcl}
\textit{Instruktion} & = & \textit{Opcode} + (\textit{Operand}) \\
\textit{Operand} & = & [\textit{1-Byte-Operand} \mid \textit{2-Byte-Operand}] \\
\textit{1-Byte-Operand} & : & \text{Ganze Zahl im Intervall } [0, 256[\\
\textit{2-Byte-Operand} & : & \text{Ganze Zahl im Intervall } [0, 16^4[
\end{array}
$$

Data Dictionary

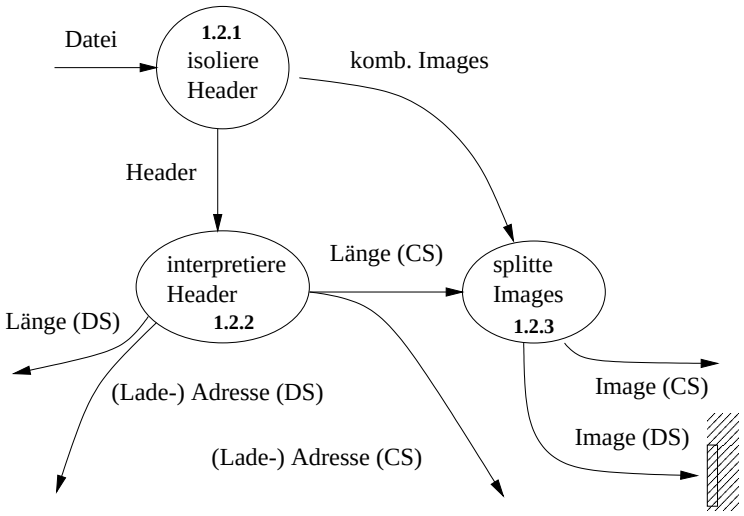
Treibertabellen	=	*Instr.-Längen-Tab.*
	+	*Mnemonik-Tabelle.* + *Oprd.-Fmt.Tab.*
Instr.-Längen-Tab.	=	$\{\underline{Opcode} + Instr.\text{-}Länge\}$
Mnemonik-Tabelle.	=	$\{\underline{Opcode} + Mnemonik\}$
Oprd.-Fmt.Tab.	=	$\{\underline{Opcode} + Oprd.\text{-}Fmt.\}$

Data Dictionary

Disass. Zeile	=	*Adress-Text* + *Instr.-Text*
Instr.-Text	=	*Mnemonik* + *Oprd.-Text*
Oprd.-Text	:	Textdarstellung eines Operanden, inklusive Adressmode-annotation, beispielsweise `$(EA31).`
Adress-Text	:	Textdarstellung einer Adresse.

Bemerkung: Die durchgehende Verwendung von *Opcode* als Schlüssel in den diversen Treibertabellen hat sich für mich als nützlicher Hinweis dafür erwiesen, dass *Opcode* möglicherweise im Entwurf in einen für das Programm zentralen abstrakten Datentyp umgesetzt werden könnte.

3.5 Isoliere Codesegment, DFD 1.2



***Minispec* Isoliere Header, DFD 1.2.1** Der *Header* wird aus der *Datei* gelesen. Was verbleibt ist eine Bytefolge, die die kombinierten Speicherabbilder (*komb. Images*) von Code- und Datensegment enthält.

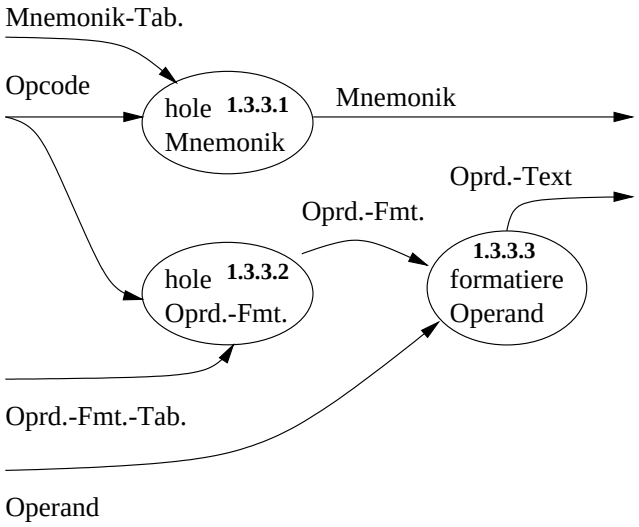
***Minispec* Interpretiere Header, DFD 1.2.2** Die 8 Byte des *Header* können als Angaben über die $Länge_{DS}$ und $(Lade\text{-})\ Adresse_{DS}$ des Datensegments, sowie $Länge_{CS}$ und $(Lade\text{-})\ Adresse_{CS}$ des Codesegments interpretiert werden.

***Minispec* Splitte Images, DFD 1.2.3** Mit der Information über die $Länge_{CS}$ des Codesegements können die kombinierten Speicherabbilder (*komb. Images*) voneinander getrennt werden: Die ersten $Länge_{CS}$ Bytes von *komb. Images* sind das Speicherabbild des Codesegments, der Rest ist das Speicherabbild des Datensegments.

Data Dictionary

$$\begin{aligned} \textit{komb. Images} &= \textit{Byte-sequenz} \\ \textit{Header} &= \textit{Bytesequenz} \\ &\star \text{ Länge: 8 Byte.} \end{aligned}$$

3.6 Disassembliere Instruktion, DFD 1.3.3

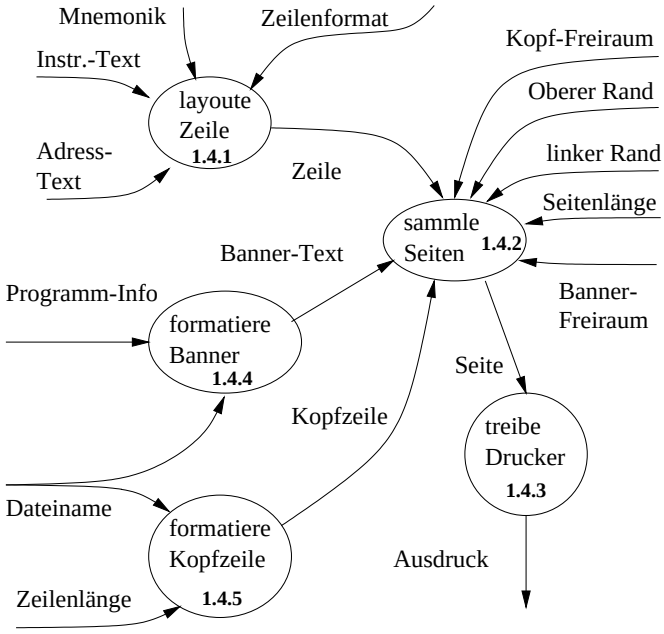


***Minispec* Hole Mnemonik, DFD 1.3.3.1** Der zum *Opcode* gehörige *Mnemonik* wird aus der *Mnemonik-Tab.* herausgesucht.

***Minispec* Hole Oprd.-Fmt., DFD 1.3.3.2** Das zum *Opcode* gehörige Operandenformat (*Oprd.-Fmt.*) wird aus der *Oprd.-Fmt.-Tab.* herausgesucht.

***Minispec* Formatiere Operand, DFD 1.3.3.3** Der *Operand* wird entsprechend den Angaben in *Oprd.-Fmt.* in eine Textdarstellung (*Oprd.-Text*) umgewandelt.

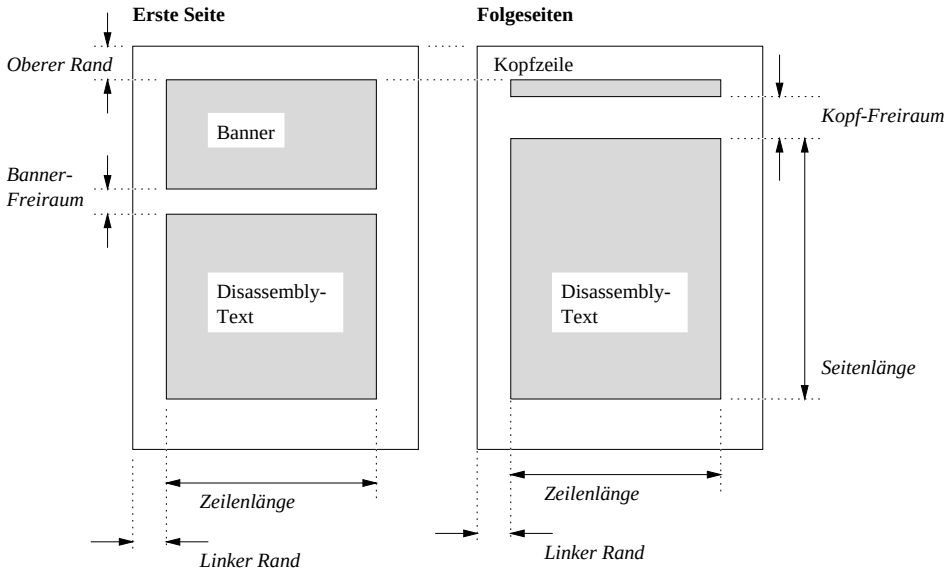
3.7 Führe Layout durch, DFD 1.4



Data Dictionary

Layout-Format	$=$	*Zeilenformat* $+$ *Zeilenlänge* $+$ *Seitenlänge* $+$ *Oberer Rand*
	$+$	*Kopf-Freiraum* $+$ *Banner-Freiraum* $+$ *Linker Rand*
	$\star$	Bedeutung: s. Minispec und Skizze.
Zeile	$=$	*Zeilen-Nr.* $+$ *Zeichenkette*
Zeichenkette	:	Zeichenkette aus druckbaren Zeichen aus dem ASCII (7 Bit!) Zeichensatz.
Seite	$=$	*Seiten-Nr.*$\{$*Zeile*$\}$
Banner-Text	$=$	$\{$*Zeile*$\}$
Kopfzeile	$=$	*Zeile*
Zeilenformat	:	Eine noch nicht näher spezifizierte Angabe, wie *Mnemonik*, *Instr.-Text* und *Adress-Text* innerhalb einer Zeile plaziert werden sollen.

Bemerkung: Die Bedeutung der einzelnen Elemente des *Layout-Format[es]* ist in der folgenden Skizze dargestellt. Alle Angaben sind in Zeilen, der das Programm beschreibende Kopf des Ausdrucks wird als *Banner* bezeichnet.



***Minispec* Layoute Zeile, DFD 1.4.1** Die Texte *Mnemonik*, *Instr.-Text*, und *Adress-Text* werden in einer Zeile so plaziert, wie es das *Zeilenformat* angibt.

***Minispec* Sammle Seiten, DFD 1.4.2** *Bannertext* und danach die *Zeile[n]* in der Reihenfolge ihrer *Zeilen-Nr.* werden nacheinander auf *Seite[n]* plaziert, wobei, wo nötig, zwischen den Zeilen umgebrochen wird.

***Minispec* Treibe Drucker, DFD 1.4.3** Eine Folge von Seiten wird in einen Kommandostrom für den Drucker übersetzt.

Bemerkung: In der Regel bedeutet das, dass die Zeichen in den Zeilen der Seiten an den Drucker geschickt werden, wobei jedoch zwischen die Zeilen Zeilenvorschubkommandos und zwischen die Seiten Papiervorschubkommandos eingefügt werden, und dass vor dem Ausdruck ein initialisierender Prolog und danach ein Abspann, der den Druckjob abschließt, an den Drucker übermittelt wird.

***Minispec* Formatiere Banner, DFD 1.4.4** *Programm-Info* und *Dateiname* werden in eine Folge von Zeilen übersetzt (ein Beispiel ist in der Anleitung angegeben), d. h. der Banner-Text wird erzeugt.

***Minispec* Formatiere Kopfzeile** Die *Kopfzeile* enthält den Dateinamen rechtsbündig.

Bemerkung: Diese Minispezifikationen illustrieren, dass es zuweilen nützlich sein kann, sich auf nicht-formale Spezifikationsmittel zu stützen, z. B. den Verweis auf Beispiele oder ein anderes Dokument (Minispec 1.4.4), oder auf Skizzen (Minispec 1.4.2). Dies unterstreicht weiterhin den Wert auch von semiformalen Methoden für die Analyse, denn diese lassen sich ohne Aufwand mit Dokumenten beliebiger Art kombinieren.

Literatur

[Klaeren 2002] KLAEREN, Prof. Dr. H.: *Vorlesungsmanuskript Softwaretechnik*. Publiziert auf den Webseiten zur Vorlesung. 2002. – URL `http://www-pu.informatik.uni-tuebingen.de/users/klaeren/swt/`

[McMenamin und Palmer 1984] MCMENAMIN, Stephen M. ; PALMER, John F.: *Essential systems analysis*. Yourdon Press, 1984 (Yourdon computing series). – ISBN 0-13-287905-0

[TeachSWT@Tü 2002a] Wilhelm Schickart Institut (Veranst.): *Übungsmaterial, Fallbeispiele und Ergänzungen zur Softwaretechnikvorlesung 2002 in Tübingen*. Sand 13, D 72076 Tübingen, 2002. (Materialien zur Softwaretechnik). – URL `http://www-pu.informatik.uni-tuebingen.de/teachswt`, Release `teachswt-tue-2002-a.tar.gz`

[TeachSWT@Tü 2002b] LEYPOLD, M E.: Lösungsskizze 5: Entwurf eines Disassemblers. In: (TeachSWT@Tü, 2002a). – `handouts/loesung-05.vdm`