Forschungsdatenzentrum
der Bundesagentur für Arbeit
im Institut für Arbeitsmarkt-
und Berufsforschung

FDZ-Methodenreport

Methodische Aspekte zu Arbeitsmarktdaten

07/2014
DE

Das Splitten von Episodendaten mit Stata

Prozeduren zum Splitten sehr umfangreicher und/ oder tagesgenauer Episodendaten

Klaudia Erhardt,
Ralf Künster

Bundesagentur für Arbeit

Das Splitten von Episodendaten mit Stata

Prozeduren zum Splitten sehr umfangreicher und/oder tagesgenauer Episodendaten

Klaudia Erhardt, Deutsches Institut für Wirtschaftsforschung (DIW)
Ralf Künster, Wissenschaftszentrum Berlin (WZB)

Juli 2014

Die FDZ-Methodenreporte befassen sich mit den methodischen Aspekten der Daten des FDZ und helfen somit Nutzerinnen und Nutzern bei der Analyse der Daten. Nutzerinnen und Nutzer können hierzu in dieser Reihe zitationsfähig publizieren und stellen sich der öffentlichen Diskussion.

FDZ-Methodenreporte (FDZ method reports) deal with the methodical aspects of FDZ data and thus help users in the analysis of data. In addition, through this series users can publicise their results in a manner which is citable thus presenting them for public discussion.

Inhaltsverzeichnis

Zusammenfassung 4

Abstract 4

1 Einleitung 5

2 Warum Episodensplitting? 6

3 Andere Verfahren und Utilities zum Splitten von Episoden 8

3.1 Episodensplitting durch expandieren und aggregieren der Episoden 8

3.2 Episodensplitting mittels "stsplit" 9

3.3 Episodensplitting mittels "spellsplit" 9

3.4 Das Programm "newspell" 10

4 Episodensplitting mit dem Programm splitspells.do 10

4.1 Technische und datenstrukturelle Voraussetzungen 10

4.2 Struktur und Bestandteile des Programms splitspells 12

4.3 Exkurs: Reproduktion des FDZ BA/IAB-internen Episodensplittings 21

5 Das Programm combispells.do: Variablen zur Anzeige von Parallelitäten 22

5.1 Das Grundprinzip numerischer Kombinationsvariablen 23

5.2 Eine String-Kombinationsvariable erzeugen und nutzen 24

5.3 Umgang mit der String-Kombinationsvariablen 26

5.4 Eine numerische Kombinationsvariable konstruieren und belabeln. 27

5.5 Erläuterung der zusätzlichen Programmbestandteile in combispells.do 30

6 Resumé und Ausblick 32

7 Anhang 34

7.1 splitspells 34

7.2 combispells 44

Literatur 51

Zusammenfassung

In diesem Report wird ein von den Autoren entwickeltes Stata-Programm zum Splitten von Episodendaten vorgestellt und detailliert beschrieben, das sich auch für sehr umfangreiche und/oder tagesgenau gemessene Episodendaten - wie etwa die SIAB-Daten - eignet. Zusätzlich wird die Generierung von Variablen behandelt, die parallele Episodentypen im gesplitteten File sowohl in einer numerischen als auch in einer für Menschen unmittelbar lesbaren Stringvariante anzeigen. Diese sind vor allem hilfreich für die Aufbereitung und Edition von gesplitteten Episodenfiles. Die verwendete Syntax wird ausführlich erläutert, so dass dieser Methodenreport auch als Anleitung zur Entwicklung von Stata-Syntax für ähnlich gelagerte Datenaufbereitungsaufgaben dienen kann.

Abstract

This report describes a Stata procedure on spell splitting that has been developed by the authors. It is particularly suitable for the treatment of very large and/or daily spelldata where the usual way of producing time-unit records is not feasible. The report presents also the generation of an indicator showing the combined spell types of parallel splits, and that in two versions: a numeric and a human readable string version. These variables are specifically helpful with the preparation and edition of splitted spell data. We explain the involved do-files in detail and with examples, so that this report can be used as a kind of textbook for similar data management tasks.

Keywords:

Episodensplitting, Episodendaten, überlappende Episoden, Verlaufsdaten, Datenaufbereitung, Stata

Spellsplitting, spell data, overlapping spells, longitudinal data, data preparation, stata

Anmerkung:

Für die Richtigkeit und Lauffähigkeit der Programme sind allein die Autoren verantwortlich. Das Ergebnis weicht von dem in den FDZ-Daten bereits implementierten Episodensplitting nur in Bezug auf die Reihenfolge von zeitgleichen Episoden des gleichen Typs ab. Anzahl sowie Anfangs- und Endzeitpunkte der Splits stimmen mit dem Ergebnis des FDZ-internen Episodensplittings überein. Für Möglichkeiten zur Reproduktion der Original-Reihenfolge der SIAB-Daten siehe Abschnitt 4.3.

1 Einleitung

Die Analyse von Verlaufsdaten nimmt inzwischen einen breiten Raum in der empirischen Sozialforschung ein. Zu Beginn der 80er Jahre wurde mit der Deutschen Lebensverlaufsstudie[1] eine der ersten sozialwissenschaftlichen Studien im deutschsprachigen Raum ins Leben gerufen, die mit einem quantitativ ausgerichteten Erhebungsdesign systematisch vollständige Lebensverläufe erhob. Seither folgten zahlreiche andere Surveys, die retrospektiv oder durch regelmäßige Wiederholungsbefragungen Verlaufsdaten erheben. Auch werden im Rahmen von Forschungsdatenzentren zunehmend prozessproduzierte Verlaufs- und Längsschnittdaten für die wissenschaftliche Forschung zugänglich gemacht[2].

Es gibt zahlreiche Lehrbücher, die sich mit der Analyse von Längsschnittdaten befassen. Hingegen finden sich nur wenige aktuelle Anleitungen für ihre Aufbereitung. Ausnahmen sind die FDZ-Methodenreports 6/2007 (Drews/Groll/Jacobebbinghaus) und 4/2013 (Eberle/ Schmucker/Seth), in denen Stata-Prozeduren für die Aufbereitung der Personendaten des Forschungsdatenzentrums der Bundesagentur für Arbeit im Institut für Arbeitsmarkt- und Berufsforschung (FDZ BA/IAB) beschrieben werden.

Die Lücke soll durch den vorliegenden Methodenreport weiter geschlossen werden. Er befasst sich mit einem speziellen Schritt in der Aufbereitung von Verlaufsdaten, nämlich dem Episodensplitting.

Episodendaten haben eine Struktur, in der zeitliche Verläufe durch einzelne Datenzeilen repräsentiert sind, die jeweils durch Fall-ID, Spell-ID, Beginn- und Enddatum sowie dem Episodentyp definiert werden. Im Gegensatz zu Querschnittsdaten sind hier einem Fall in der Regel mehrere Datenzeilen zugeordnet, die für einzelne Episoden stehen, welche den Verlauf des jeweiligen Falles abbilden.

Die Episoden eines Verlaufs können sich zeitlich überlappen oder zueinander parallel liegen. Episodensplitting bereitet die Daten derart auf, dass alle gesplitteten Episoden entweder vollkommen parallel oder vollkommen überschneidungsfrei zu den anderen gesplitteten Episoden des gleichen Falles sind (s. Beschreibung des Episodensplitting in den Datendokumentationen FDZ BA/IAB, z.B. in: vom Berge/König/Seth: FDZ-Datenreport 01/2013: 24).

Das von uns vorgestellte Verfahren zum Episodensplitting erzeugt im Gegensatz zu dem üblicherweise verwendeten Verfahren des Expandierens der Episodendaten auf Basis ihrer Dauern (siehe Abschnitt 3.1) die geringstmögliche Anzahl an zusätzlichen Datenzeilen. Es minimiert damit die Auslastung des verfügbaren Arbeitsspeichers und ist daher besonders geeignet für sehr große Datenbestände. Das Resultat entspricht der Datenstruktur, in der die SIAB-Daten des FDZ BA/IAB zur Verfügung gestellt werden[3].

---

1 Die Deutsche Lebensverlaufsstudie wurde unter Leitung von Karl Ulrich Mayer am Max-Planck-Institut für Bildungsforschung durchgeführt (siehe https://www.mpib-berlin.mpg.de/de/forschung/beendete-bereiche/bildung-arbeit-und-gesellschaftliche-entwicklung/deutsche-lebensverlaufsstudie, Abruf: 09.04.2014).

2 s. http://www.ratswd.de/forschungsdaten/fdz (Zugriff: 09.04.2014)

3 s. http://fdz.iab.de/de/FDZ_Individual_Data/integrated_labour_market_biographies.aspx

Unser Verfahren ist auch für die NutzerInnen von bereits gesplittet zu beziehenden Daten interessant. Daten werden in der Regel vor der Auswertung nach den jeweiligen Anforderungen aufbereitet. Manche Aufbereitungsschritte, wie z.B. die Eliminierung von Episoden, welche eine gewisse Mindestdauer unterschreiten, oder die Generierung neuer Episodentypen, können ein erneutes Episodensplitting notwendig machen. Es ist daher wünschenswert, seine Verlaufsdaten in eigener Regie jederzeit neu splitten zu können.

Während gesplittete Daten sehr einfach mit einer einzigen Befehlszeile in die Ursprungsform zurückverwandelt werden können (und zwar mit dem Stata Befehl `keep if begepi == begorig`, siehe Abschnitt 4.1), ist der Vorgang, die Episoden zu splitten, nicht so leicht. Es gibt unseres Wissens bisher keine veröffentlichte detaillierte Anleitung, wie diese Aufgabe in Stata umgesetzt werden kann. Dies soll der vorliegende Methodenbericht leisten.

Im folgenden Abschnitt beschreiben wir, was Episodensplitting ist und wozu es gebraucht wird. Im 3. Abschnitt gehen wir kurz auf bereits in der Praxis eingesetzte Verfahren des Episodensplittings ein. Das von uns entwickelte Verfahren des Episodensplittings mit Stata wird im 4. Abschnitt Schritt für Schritt erläutert. In diesem Zusammenhang stellen wir zudem eine Routine vor, die den Gesamtdatensatz in kleinere Portionen zerlegt und nach der Bearbeitung wieder zusammenfügt. Dieser Vorgang ist eine zusätzliche Hilfe für das Splitten großer Datensätze, da dies ziemlich speicherintensiv ist. Im 5. Abschnitt behandeln wir schließlich die Konstruktion einer Variablen in einer String- und einer numerischen Version, welche die vorliegenden Episodentyp-Parallelitäten anzeigt.

Die Programme für das Episodensplitting und die Konstruktion der Parallelitätsvariablen wurden auf dem German Stata Users Meeting 2014 in Hamburg vorgestellt (Erhardt/Künster 2014). Auf der Website des FDZ BA/IAB stehen sie als Stata-do-files mit deutscher oder englischer Kommentierung unter http://doku.iab.de/fdz/reporte/2014/MR_07-14_Programme.zip zur Verfügung.

2 Warum Episodensplitting?

Die meisten Verfahren der Verlaufsdatenanalyse untersuchen, wann und warum Übergänge von einem Zustand in einen anderen Zustand eintreten. Die empirisch erhobenen Verläufe von Personen bestehen jedoch in der Regel nicht aus einer klaren Abfolge eindeutiger Zustände, sondern es finden sich häufig zeitliche Überschneidungen von Episoden. Die meisten erhobenen Zustandsarten (Episodentyp, Status) sind nicht wechselseitig ausschließend, sondern können in unterschiedlichen Konstellationen gleichzeitig auftreten.

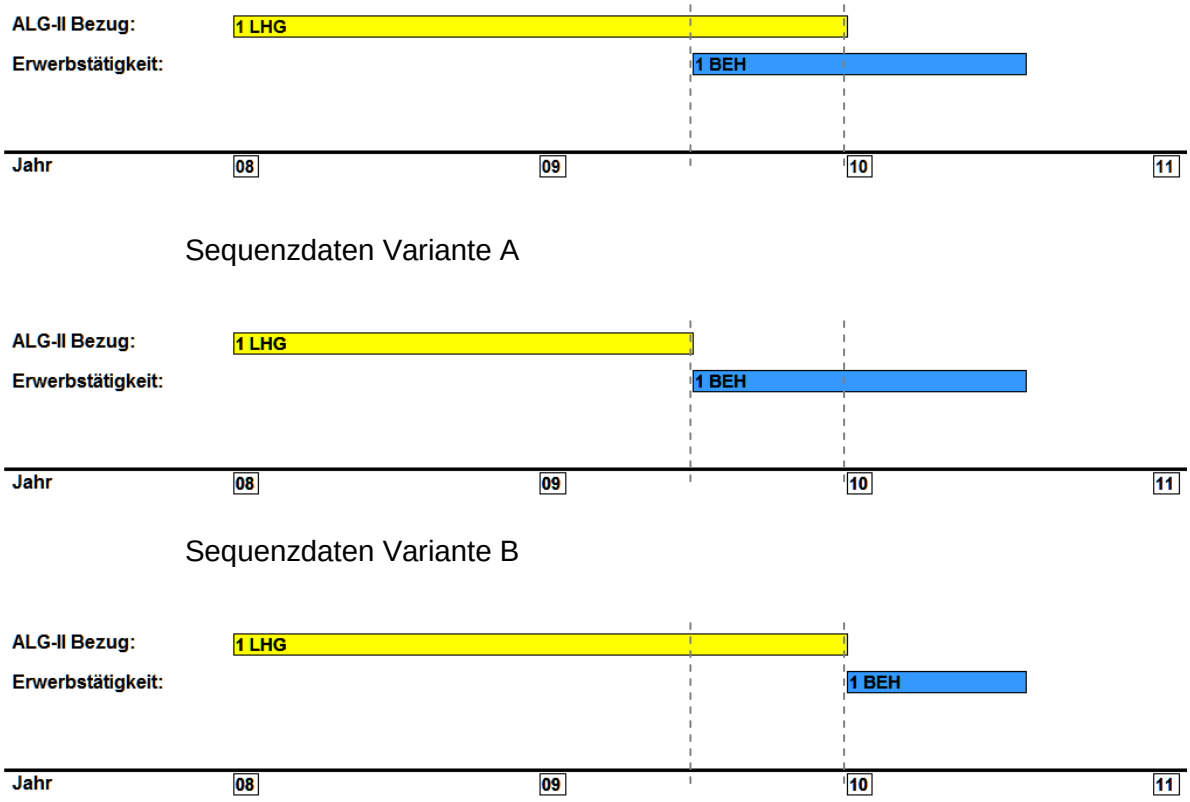
Die Längsschnittdatenanalyse setzt dagegen in der Regel eine "eindimensionale" Datenstruktur voraus, bei der in jedem Zeitintervall ein eindeutig bestimmbarer Zustand herrscht. Solcherart strukturierte Daten werden als "Sequenzdaten" bezeichnet. Sequenzdaten sind zudem passend für Sequenzanalysen[4] oder die Erstellung von Statusverteilungsgrafiken.

Episodensplitting ist eine Zwischenstufe auf dem Weg zu Sequenzdaten. Nach dem Splitting ist jeder Zeitabschnitt eines Verlaufs eindeutig einem oder mehreren Episodentypen zuge-

ordnet. Durch Anwendung von theoretisch festgelegten oder empirisch abgeleiteten Prioritätenregeln lassen sich gesplittete Daten leicht in Sequenzdaten überführen.

Parallelitäten sind ambivalent: Wann wurde in dem in Abbildung 1 gezeigten Beispiel der Übergang vom Leistungsbezug zur Erwerbstätigkeit vollzogen? Es gibt zwei mögliche Antworten: Zum einen fand der Übergang in die Erwerbstätigkeit statt zum Zeitpunkt, als die Erwerbstätigkeit begann (Variante A), zum anderen fand er statt, als der Leistungsbezug endete (Variante B). Die Entscheidung, welcher dieser beiden Übergänge als der relevante gelten soll, wird im Rahmen von (theorie- und/oder empiriegeleiteten) Priorisierungsregeln festgelegt.

Abbildung 1: Datenmodifikation bei der Überführung von Verlaufsdaten in Sequenzdaten[5]



Quelle: eigene Darstellung (fiktiver Verlauf), Grafik generiert mit LDEX 2003 Ralf Künster [6]

Datentechnisch bedeutet die Entscheidung für eine der beiden Varianten, dass die Überschneidungszeit einer der beteiligten Episoden zugerechnet und die jeweils andere um diesen Zeitraum gekürzt wird. Denkbar wäre auch, die Phase der parallelen Aktivität als eigenständigen neuen Status zu definieren.

---

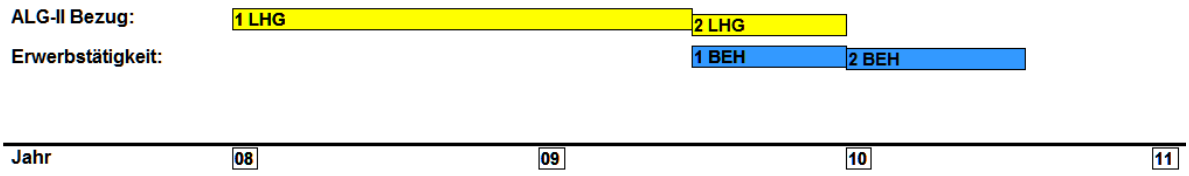
4 Brzinsky-Fay, C./Kohler, U./Luniak, M. (2006)

5 "LHG" und "BEH" sind Kurzbezeichnungen der SIAB für die Datenquellen "Leistungshistorik Grundsicherung" und "Beschäftigtenhistorik".

6 Dokumentation zum Programm LDEX s. Hillmert/Künster/Spengemann/Mayer (2004), Teil VIII.

Für alle Varianten ist jedoch die Information "Beginn und Ende der Parallelität" unabdingbar, und hier kommt das Episodensplitting ins Spiel. Es zerlegt jede Episode in Zeitabschnitte, welche innerhalb des gleichen Falles vollkommen überschneidungsfrei sind.[7] Nach dem Episodensplitting kann also jedes Zeitstück eindeutig einem oder mehreren Episodentypen zugeordnet werden (Abbildung 2). Dieses Format ist die Ausgangsbasis für die Herstellung von Sequenzdaten.

Abbildung 2: Episode nach dem Splitting



Quelle: eigene Darstellung (fiktiver Verlauf), Grafik generiert mit LDEX 2003 Ralf Künster

3 Andere Verfahren und Utilities zum Splitten von Episoden

3.1 Episodensplitting durch expandieren und aggregieren der Episoden

Ein kaum dokumentiertes, aber in der Praxis[8] häufig genutztes Verfahren zum Splitten von Episoden besteht darin, für jede Zeiteinheit einer Episode eine eigene Datenzeile anzulegen. Wurden beispielsweise die Datumsangaben der Episoden monatsgenau gemessen und dauert eine Episode 12 Monate, dann werden für diese Episode 12 Datenzeilen angelegt, wobei jeder Zeile einen der 12 Monate repräsentiert. Dazu wird die Dauer der Episoden berechnet, die anschließend per expand-Befehl zum Anlegen der monatsbasierten Datenzeilen genutzt wird. In der folgenden Beispielsyntax wird anhand der Stata-Variablen _n und dem Startdatum der Episode eine Variable erzeugt, die jeder Datenzeile einen Monat der Episodenzeitspanne zuweist. Hier beispielhaft die entsprechende Befehlssequenz:

```
gen dauer = endorig – begorig + 1
expand dauer
bysort persnr spell: gen x = _n – 1
gen monat = begorig + x
```

Parallelitäten in den Episoden lassen sich nun einfach durch die Sortierung der Datenzeilen nach Fallnummer und Monatsvariable entdecken: Haben aufeinander folgende Datenzeilen den gleichen Monatswert, so liegen parallele Zustände vor. Der nächste Schritt besteht darin, eine Variable zu generieren, die den Episodenarten eine hierarchische Abfolge zuweist (Priorisierung). Bei parallelen Zuständen wird die Datenzeile mit dem höchsten Hierarchie-

[7] Vgl. die Beschreibungen des Episodensplittings in den Dokumentationen des FDZ BA/IAB, z.B. in: vom Berge/König/Seth (2013), S. 24.

[8] Z.B. im Rahmen der Deutschen Lebensverlaufsstudie von Karl-Ulrich Mayer am Max-Planck-Institut für Bildungsforschung Berlin oder im Kontext von Nutzerschulungen zu den Daten des Nationalen Bildungspanels des Leibniz-Instituts für Bildungsverläufe e.V. Bamberg.

wert im Datensatz belassen, die anderen werden gelöscht. Danach können die Monatsdatenzeilen mit dem collapse-Befehl wieder in Episoden umgewandelt werden.

Der Vorteil dieses Verfahrens ist, dass nur sehr wenige und einfach nachzuvollziehende Befehle benötigt werden. Außerdem entspricht das Zwischenprodukt der in die kleinsten Zeiteinheiten zerlegten Episoden (im obigen Beispiel der Monatsdatenzeilen-File) nach der Eliminierung der Parallelitäten bereits den datenstrukturellen Anforderungen für Sequenzanalysen oder die Berechnung von zeitabhängigen Statusverteilungen. Ein entscheidender Nachteil ist jedoch, dass durch das Expandieren der Datenzeilen auf Basis der kleinsten gemessenen Zeiteinheit der Datenfile leicht eine Größe erreicht, die die Grenzen des Arbeitsspeichers übersteigt. Dieses Problem kann auftreten, wenn große Fallzahlen vorhanden sind, wenn pro Fall lange Zeiträume erfasst wurden, und wenn die Genauigkeit der Zeitmessung hoch ist. Alle drei Kriterien treffen beispielsweise auf die oben erwähnten SIAB-Daten zu. Der SIAB-SUF von 1975-2008 (siab_r_7508) enthält ungesplittet rund 29,7 Millionen tagesgenau gemessene Episoden. Auf Tagesbasis expandiert würden daraus 7,1 Milliarden Datenzeilen. Selbst nach der Umwandlung der tagesgenauen in monatsgenaue Datumsangaben würden immer noch ca. 240,5 Millionen Monatsepisoden entstehen - eine Größenordnung, die die heute üblichen Arbeitsplatzrechner überfordert. Unser Verfahren umgeht dagegen das Zerlegen der Episoden in die kleinsten Zeiteinheiten. Die Zeitangaben der Episoden müssen vor dem Splitten nicht in gröbere Einheiten umgewandelt werden. Zusätzlich kann die Geschwindigkeit des Splittings durch die automatisierte Aufteilung der Daten in einzelne Portionen erheblich beschleunigt werden. Lässt sich der Quelldatenfile in den Arbeitsspeicher laden, dann ist nahezu gewährleistet, dass auch für den Splittingprozess genügend Arbeitsspeicherressourcen vorhanden sind.

3.2 Episodensplitting mittels "stsplit"[9]

Der stata-Befehl stsplit erlaubt es, Episoden an vorgegeben Zeitpunkten oder in bestimmten Intervallen zu splitten. Das ist sehr hilfreich, wenn bereits bekannt ist, zu welchen Zeitpunkten das Splitting erfolgen soll oder wenn alle Episoden einer bestimmten Zeitspanne zu einem festen Zeitpunkt gesplittet werden sollen. Unser Ausgangspunkt ist jedoch, dass reale Prozesse oder Lebensverläufe erfasst wurden, bei denen die Splittingzeitpunkte in Abhängigkeit von den vorliegenden Episodenüberschneidungen erst ermittelt werden müssen. Diese Ermittlung von unvorhersehbaren und multiplen Splittingzeitpunkten stellt stsplit nicht zur Verfügung. Daher ist stsplit für die Anforderungen, die Splittingzeitpunkte aus den Parallelitäten der Daten abzuleiten, nicht geeignet.

3.3 Episodensplitting mittels "spellsplit"[10]

Das Programm spellsplit wurde von Edwin Leuven 2003 in einem utility package für Episodendaten (spellutil) veröffentlicht. Das Programm splittet Episoden, die sich überschneiden, und aggregiert gleichzeitig die jeweils parallelen Episodensplits zu einer einzigen Episode. Als Parameter zur Durchführung des Splits werden die Namen von Start- und Enddatumsvariablen und die Fallkennungsvariable (CaseID) benötigt. Die in einer Variablenliste beim

[9] s. http://www.stata.com/manuals13/ststsplit.pdf. Beispiele zum Einsatz von stsplit sind in Blossfeld/Golsch/Rohwer (2007): S. 118ff. und S. 147ff. zu finden.

[10] s. http://fmwww.bc.edu/repec/bocode/s/spellsplit.html.

Befehlsaufruf benannten Variablen werden bei der Aggregation entweder aufsummiert oder es wird der Mittelwert der Variablen gebildet. Die restlichen, eventuell im Datensatz vorhandenen Variablen werden aus dem Datensatz entfernt.

Im Gegensatz zu diesem Programm belässt das von uns vorgestellte Verfahren alle Variablen des Datensatzes in ihrem Originalzustand. Die parallelen Episoden werden nicht aggregiert. Die ursprünglichen Episoden werden stattdessen ausschließlich in überschneidungsfreie Stücke zerlegt. Das Programm spellsplit von E. Leuven hat daher ein anderes Ziel als unser Programm.

3.4 Das Programm "newspell"[11]

Das Programm newspell ist ein ado-File, der von Hannes Kröger auf dem German Stata Users Group Meeting 2013 vorgestellt wurde, und das die Weiterentwicklung eines früher mit den SOEP-Daten mitgelieferten exe-Programms gleichen Namens ist. newspell bietet eine Reihe von interessanten Funktionen, wie beispielsweise das Schließen von Lücken mit Lückenepisoden, das Zusammenfassen unterschiedlicher Episodentypen zu einem neuen Episodentyp oder die Generierung von Sequenzdaten auf Basis einer vom Nutzer einzugebenden Prioritätenliste (Ranking) der Episodentypen und einiges mehr.

Als Quelldaten werden jedoch immer Episodendaten vorausgesetzt, die - wie im SOEP - mittels "Kalendarien" erfragt wurden. Jeder Episodentyp kann pro Zeiteinheit nur einmal auftreten, Parallelitäten des gleichen Typs (z.B. mehrere gleichzeitige Erwerbstätigkeiten) können daher nicht abgebildet werden. Nutzt man newspell im Zusammenhang mit Episodendaten, bei denen auch zeitliche Parallelitäten des gleichen Episodentyps vorhanden sind, schlägt der Versuch fehl, daraus mittels Ranking Sequenzdaten zu generieren, da Episoden des gleichen Typs nach dem Ranking-Befehl auch weiterhin die ursprünglichen Parallelitäten aufweisen.

Darüber hinaus ist es mit dem Programm newspell nicht möglich, ein reines Episodensplitting durchzuführen. Das Splitten mit newspell ist stets mit einem Ranking und dem Löschen der niedriger priorisierten parallelen Episodensplits verbunden.

Laut Auskunft des Autors befindet sich der ado-File im Review-Prozess und ist im Anschluss daran als qualitätsgeprüfter Stata-ado-File herunterladbar.

4 Episodensplitting mit dem Programm splitspells.do

4.1 Technische und datenstrukturelle Voraussetzungen

Das nachfolgend beschriebene Programm splitspells wurde mit Stata/SE Version 12 entwickelt und in Version 12 und 13 auf Lauffähigkeit getestet. Das Episodensplitting kann mit Small Stata wegen der Limitierung auf 99 Variablen nur ausgeführt werden, wenn kein Fall mehr als ca. 35 Episoden aufweist und außer den Kernvariablen (Fallnummer, Episodennummer, Start- und Enddatum, Episodentyp) keine weiteren Variablen im Datensatz vorhan-

[11] s. http://www.stata.com/meeting/germany13/abstracts/materials/de13_kroeger.pdf (Zugriff: 06.05.2014)

den sind. Stata/IC kann verwendet werden, wenn das Limit von 2.047 Variablen eingehalten wird, wenn also ein Maximum von ca. 1.000 Episoden pro Fall nicht überschritten wird und der Datensatz außer den Kernvariablen nur wenige weitere Variablen enthält.

Wenn der Datensatz sehr viele Fälle umfasst, sollte eine portionsweise Verarbeitung gewählt werden (s. Abschnitt 4.2.2), weil Stata mit zunehmender Arbeitsspeicherbelegung überproportional langsamer wird. Es dauert dann länger, den ungeteilten als den in Einzelportionen aufgeteilten Datensatz zu bearbeiten.

Die Syntax ist für die Anwendung auf **ungesplittete Datensätze** gedacht**.** Gesplittete Datensätze müssen daher zuvor "entsplittet" werden (`keep if begorig==begepi` bzw. `keep if endorig==endepi`, s. vom Berge/König/Seth: FDZ-Datenreport 01/2013: 24). Eventuell vorhandene, auf gesplittete Files bezogene technische Merkmale sollten entfernt werden, da sie im ungesplitteten File keinen Sinn machen[12]. Beim erneuten Splitting werden die Episodenzähl-Variablen[13] neu erzeugt.

Folgende Kernvariablen müssen zwingend vorhanden sein (in Klammern sind die SIAB-Variablenbezeichnungen aufgeführt; im Definitionsteil des Programms splitspells können die jeweiligen Namen der Kernvariablen angegeben werden):

- Fall-ID (persnr),
- Spell-ID (spell),
- Episodentyp-Variable (quelle in der SIAB bzw. quelle_gr im SIAB-SUF),
- Beginn- und Enddatum jeder Episode (begorig, endorig),

Fall-ID und **Spell-ID** müssen zusammen einen eindeutigen Identifikator für jede Episode ergeben.

Die **Episodentyp-Variable** darf keine System Missings haben, negative Zahlen als Codes sind jedoch zulässig.

In den **Datumsangaben** dürfen keine missings vorkommen und das Beginndatum darf nicht größer als das Enddatum sein (vor Anwendung von splitspells prüfen!). Die Datumsangaben müssen (intern) als positive Integer-Werte vorliegen: eine Erhöhung um 1 bedeutet 1 Zeiteinheit später. Bei Datumsangaben in der Stata Internal Form (SIF)[14] ist diese Voraussetzung erfüllt, denn Stata rechnet intern mit ganzen Zahlen, welche die seit dem 1.1.1960 vergangenen Zeiteinheiten repräsentieren. Welche Zeiteinheit gemeint ist, kann mit dem Format-Befehl angegeben werden, dies beeinflusst jedoch nur die Darstellung im Datenblatt und im Output, nicht die dahinter stehenden Werte und nicht die Ausführung des Programms splitspells. Datumsangaben, die nicht intern als Integerzahl repräsentiert sind (z.B. Angaben, in denen Monate und Jahre in getrennten Variablen stehen, oder Datumsangaben, in denen

12 Beispielsweise die in den gesplitteten SIAB-Daten (Version 7508) enthaltenen Episodenzählungsvariablen level1 und level2, vgl. S. 16f.

13 Neben der Episodenzählung über den gesamten Fall beispielsweise die Zählung und Gesamtzahl zeitgleicher Episoden, s. ebda.

14 s. http://www.stata.com/manuals13/ddatetime.pdf

für ungewisse Zeiten Nullen vergeben wurden (z.B. "00.00.2003"), müssen vor Anwendung des Programms in das interne Stata-Format umgewandelt werden.[15]

Das Programm splitspells übernimmt die Format-Eigenschaft der Beginn-Datumsvariablen der Quelldaten und überträgt sie auf die neu erzeugten Variablen für Beginn und Ende der gesplitteten Episoden. Dadurch kann splitspells für Daten aller Zeiteinheiten angewendet werden, vorausgesetzt, sie sind intern als Integerzahlen repräsentiert.

Der splitspells.do-File darf nicht "blind" aufgerufen werden, da vor dem Aufruf spezifische Parameter wie Datei- und Variablennamen im Definitionsabschnitt der Syntax eingetragen werden müssen. Mit Ausnahme dieser Spezifikationen im Definitionsteil sind keine weiteren Anpassungen an die jeweiligen Gegebenheiten erforderlich.

Wir haben durchgehend lokale Makros bzw. mit tempname benannte Scalare verwendet, so dass das Programm keine Konflikte mit eventuellen globalen Definitionen in der Nutzerumgebung erzeugt. Zum Schutz der Quelldaten arbeiten wir außerdem mit tempfile und tempvar, so dass ein Überschreiben von vorhandenen Datenfiles oder Variablen ausgeschlossen ist. Wenn man Syntaxbestandteile anderweitig einsetzen will, sollte man mit der Verwendung von tempname, tempfile und tempvar sowie Makros und Scalaren in Stata vertraut sein, damit die Übertragung in einen anderen Kontext gelingt.

Viele NutzerInnen bevorzugen globale Makros, die ihre Gültigkeit nach Programmende behalten, weil so auch Ausschnitte einer Syntax ausgeführt werden können. Eine partielle Ausführung wird für splitspells nur selten sinnvoll sein. Falls es dennoch erforderlich sein sollte, kann das Problem, dass lokale Makros mit Programmende erlöschen, umgangen werden, indem der gesamte Definitionsteil vor die betreffende Syntax kopiert und mit ihr zusammen ausgeführt wird.

Nach Programmende steht nach wie vor die unveränderte Quelldatei im Arbeitsspeicher. Ursache dafür ist der Befehl 'preserve', einer weiteren Schutzmaßnahme, welche dafür sorgt, dass bei einem unvorhergesehenen Programmabbruch nicht unbeabsichtigt die Quelldatei beschädigt wird. Die Ergebnisdatei muss aufgrund dieses Befehls eigens geladen werden und steht nicht wie üblich direkt zur Verfügung.

4.2 Struktur und Bestandteile des Programms splitspells

In der folgenden Beschreibung des Progamms wird auf die Wiedergabe von Syntax weitgehend verzichtet. Die komplette, ausführlich kommentierte Stata-Syntax findet sich im Anhang.

Zuerst ein Überblick über den Aufbau des Programms:

- In einem einleitenden Definitionsteil werden datensatzspezifische Parameter (z.B. Variablennamen und Dateipfadangaben) sowie die im Programmablauf benötigten tempfiles, tempvars und tempnames definiert,
- darauf folgt das Modul zur Zerlegung des Gesamtdatensatzes in einzelne Portionen,

[15] Unschädlich ist hingegen, wenn die Datumsangaben auf ein anderes Referenzdatum als der 1.1.1960 bezogen sind, ansonsten aber dem SIF entsprechen.

- als nächstes kommt das eigentliche Episodensplitting-Programm,
- dann werden die einzelnen Portionen wieder zusammengefügt,
- und als letztes werden diverse Episodenzählvariablen generiert.

4.2.1 Definitionsteil

Der Definitionsteil ist in drei Abschnitte gegliedert: a) zwingend erforderliche manuelle Vergabe von Parametern, b) optionale manuelle Vergabe von Parametern und c) automatische Vergabe von Parametern.

Im Abschnitt a) legen die NutzerInnen folgende Parameter fest:

- Zahl der Portionen, in die der File zerlegt werden soll.

  Die Zahl der Portionen muss größer Null und kleiner als die Zahl der Fälle (nicht Observationen!) sein. Die optimale Portionsgröße hängt von den zur Verfügung stehenden Ressourcen und den Daten ab - unter Stata/SE und mit den SIAB-Daten könnte man mit etwa 100 Portionen starten und testen, wie schnell das Programm durchläuft. Die vollzogene Bearbeitung jeder Portion wird unter Angabe der Uhrzeit gemeldet, so dass gegebenenfalls das Programm abgebrochen und mit erhöhter Portionszahl neu gestartet werden kann. [16]

  Wenn dieser Parameter auf 1 gesetzt wird, findet keine Zerlegung in Portionen statt.
- Übergabe von Pfaden und Dateinamen an lokale Makros. U.a. werden Quell- und Zieldatei und die Pfade für Daten- und Outputfiles spezifiziert.
- Spezifikation der in den Ausgangsdaten vorhandenen Variablenamen für Beginn- und Enddatum, Episodentyp, Fall-ID und Spell-ID. Die Variablen müssen unter den angegebenen Namen in der Quelldatei vorhanden sein.
- Optional kann ein weiteres Sortierkriterium angegeben werden, das die Reihenfolge von zeitgleichen Splits des gleichen Episodentyps bestimmt.

Die Vorgaben des Abschnitts a) dienen nur als Beispiele und müssen durch die jeweiligen Werte ersetzt werden.

In Abschnitt b) des Definitionsteils werden die Namen der im Programmablauf erzeugten Variablen definiert. Die Vorgabe orientiert sich an den SIAB-Konventionen und kann übernommen werden. Wenn stattdessen (frei wählbare) andere Variablennamen vergeben werden, sollten diese nicht bereits in der Quelldatei vorhanden sein. Zwar wird die Quelldatei nicht überschrieben, es könnte aber Verwirrung stiften, wenn in der Quell- wie in der Zieldatei gleich benannte Variablen vorkommen, die unterschiedliche Inhalte haben.

[16] Als Anhaltspunkt für institutionelle Umgebungen mit zeitgemäßer EDV-Ausstattung: das Splitten des SIAB-SUF r7508 mit ca. 1,5 Mio Fällen und ungesplittet knapp ca. 30 Mio Episoden dauerte am DIW Berlin bei einer Zerlegung in 1.000 Portionen ca. 2 1/4 Stunden. Am Wissenschaftszentrum Berlin dauerte das Splitten eines annähernd gleich großen künstlichen Testfiles nur wenig länger. - Die Bearbeitungsdauer kann durch ein Zerlegen in eine größere Zahl von Portionen nur verkürzt werden, soweit die Ressourcen des Rechners Ursache für lange Bearbeitungszeiten sind. Falls in einer EDV-Umgebung dagegen Engpässe in den *Datenleitungen zum Server* auftreten, ist eine weitere Verkleinerung der Portionen eher kontraproduktiv.

Die in Abschnitt c) des Definitionsteils automatisch vergebenen Parameter dürfen nicht verändert werden.

4.2.2 Modul zur Zerlegung des Gesamtdatensatzes in einzelne Portionen

Die Zerlegungsprozedur umklammert das eigentliche Episodensplitting-Programm. Sofern der Wert für die Zahl der Portionen größer ist als 1, wird der Datensatz entsprechend der vom Nutzer eingegebenen Portionszahl zerlegt. Anschließend wird das Episodensplitting in einer Schleife mit jeder einzelnen Portion ausgeführt, danach werden die Portionen wieder zusammengesetzt.

Bei der Zerlegung dürfen Fälle nicht auf verschiedene Portionen aufgeteilt werden. Daher muss die Zahl der Fälle - nicht der Observationen - durch die Zahl der Portionen geteilt werden. Zu diesem Zweck wird eine neue Fall-ID generiert, die die Fälle kontinuierlich aufwärts zählt und damit auch die Gesamtzahl der Fälle anzeigt. Die Division der Gesamtzahl der Fälle durch die Zahl der Portionen ergibt die Zahl der Fälle, die in einer Portion enthalten sind. Da die Fälle aber kontinuierlich aufsteigend durchnummeriert sind, kann diese Zahl auch als Spanne zwischen der niedrigsten und der höchsten Fall-ID in den Portionen interpretiert werden. Dies wird für die Selektion der Fälle in jedem einzelnen Schleifendurchgang genutzt. Die Bedingung für die Selektion würde normalerweise lauten: Alle Fall-IDs ab der letzten bereits einer Portion zugewiesenen Fall-ID bis zu der Fall-ID, die im i-ten Schleifendurchgang den Wert hat: i mal Spanne. Aus Gründen, die gleich noch erläutert werden, geschieht die Aufteilung in Portionen jedoch von der höchsten Fall-ID abwärts, der letzte Portionsfile hat die niedrigsten Fall-IDs. Der Algorithmus für die obere Fallnummer einer Portion lautet nun: Gesamtfallzahl minus Spanne mal (i minus 1). Die niedrigste Fallnummer errechnet sich aus der oberen Fallnummer der Portion minus der Spanne plus 1.

Ein Beispiel zur Verdeutlichung: Der Quell-Datenfile hat 1.028 Fälle, die Portionszahl beträgt 10, also beträgt die Spanne der Fall-iDs 102. Im ersten Durchgang werden die Fälle 927 bis einschließlich 1.028 selektiert, im nächsten 825 bis einschließlich 926 und so fort. Die obere Fallnummer einer Portion ergibt sich aus der Gesamtfallzahl minus Spanne mal (i = Zahl der bereits erfolgten Schleifendurchgänge, minus 1), für die erste Portion also 1.028 minus 102 mal 0 = 1.028. Die untere Fallnummer errechnet sich aus der oberen Fallnummer der Portion minus der Spanne plus 1, also für die erste Portion: 1.028 minus 102 plus 1 = 927.

Es geht aber nur im Sonderfall die Division der Gesamtzahl der Fälle durch die Zahl der Portionen glatt ohne Rest auf, und nur dann wäre die Spanne der Fall-IDs für alle Portionen gleich. Deshalb lautet die Bedingung zur Selektion der letzten Portion: Alle Fall-IDs von 1 bis zur oberen Fallnummer dieser Portion.

Nun zum Grund für diese etwas umständlich erscheinende Portionierung von der höchsten Fallnummer an abwärts: Die einzelnen Fälle von Verlaufsdaten weisen häufig eine sehr unterschiedliche Zahl von Episoden auf. Da die Bearbeitungszeit von der maximal vorkommenden Episodenanzahl abhängt, werden die Fälle bei der Generierung der neuen fortlaufenden Fall-ID nach ihrer Episodenzahl sortiert. Durch diese Maßnahme verkürzte sich die Laufzeit

des Programms deutlich[17]. Es galt jedoch zu vermeiden, dass die letzte Portion sowohl die Fälle mit den meisten Episoden als auch den bei der Division verbleibenden Rest an Fällen zugewiesen bekommt. Die egen(group)-Funktion, mit der die neue Fall-ID unter einer Sortierung nach Episodenzahl erzeugt wird, lässt keine absteigende Sortierung der Gruppierungsvariablen zu, daher blieb nur der Weg, die Portionierung "von oben an" durchzuführen. Dadurch wird der letzten Portion zwar den bei der Division verbleibenden Rest an Fällen, zugeteilt, aber es handelt sich um die Fälle mit der niedrigsten Zahl an Episoden.

Im Anschluss an die Zerlegung in Portionen folgt innerhalb des Programms splitspells die Schleife, in der das Episodensplitting auf die einzelnen Portionen angewendet wird. Dieses wird im folgenden Abschnitt beschrieben. An dieser Stelle wird aber noch kurz auf den zweiten Teil des Zerlegungsprogramms, das Wieder-Zusammensetzen, eingegangen: Die Portionen werden in einer Schleife mit so vielen Durchgängen, wie es Portionen gibt, per "append" zusammengefügt.

Die Routinen zum Zerlegen der Files und zum Episodensplitting sind voneinander unabhängig programmiert, daher kann das Zerlegungsmodul auch für andere speicherintensive Prozeduren genutzt werden, indem die von dem Zerlegungsmodul eingerahmte Syntax ausgetauscht wird.

4.2.3 Episodensplitting

Das Episodensplitting findet im Programm splitspells innerhalb einer Schleife statt, welche die in einzelne Portionen aufgeteilten Datenfiles abarbeitet, jedoch für die eigentliche Splitting-Prozedur keine Rolle spielt.

Das Splitting funktioniert folgendermaßen: Für jede Episode eines Falles wird ermittelt, ob Anfangs- und/oder Endzeitpunkte der anderen Episoden dieses Falles in ihre zeitlichen Grenzen fallen. Denn die Zeitpunkte, an denen eine Episode gesplittet werden muss, sind diejenigen, an denen andere Episoden des gleichen Falles beginnen oder enden[18]. Dazu werden alle (um Doppelungen bereinigte) Datumsangaben der Episoden gesammelt und als Variablen fallweise an jede Episode des Ursprungsfiles angehängt. Danach erfolgt die Prüfung, welche der angehängten Datumsangaben in die zeitlichen Grenzen der jeweiligen Episode fallen. Datumsangaben, die nicht in diese Zeitspanne fallen, werden gelöscht. Die verbleibenden Datumsangaben zeigen nun Anzahl sowie Beginn- und Enddatum der gesplitteten Episoden an. Diese Informationen werden schließlich zum Anlegen der Episodensplits benutzt.

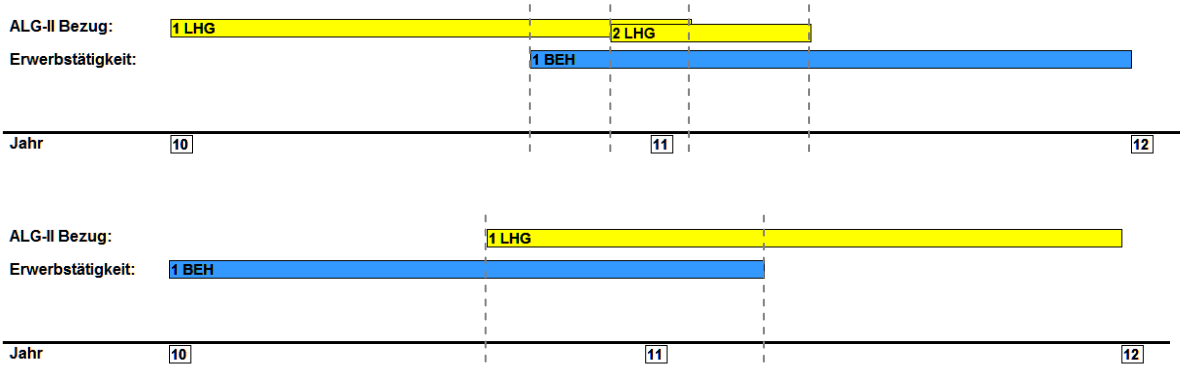
Um die Funktionsweise der im Folgenden detailliert beschriebenen Episodensplitting-Syntax leichter nachvollziehbar zu machen, demonstrieren wir die Auswirkungen der einzelnen Teilschritte an einem fiktiven Beispieldatensatz mit 2 Fällen und 3 bzw. 2 ungesplitteten Episoden (s. Abbildung 3 bis Abbildung 13).

---

17 In der EDV-Umgebung des DIW verkürzte sich die Zeit zur Bearbeitung des SIAB-r-7508-Files von knapp viereinhalb Stunden auf zwei Stunden und 12 Minuten.

18 Exakt ausgedrückt ergeben sich die Splittingspunkte einer Episode aus den Beginn- und den *um eine Zeiteinheit erhöhten* Endzeitpunkten der anderen zum gleichen Fall gehörenden Episoden.

Abbildung 3: Zwei Beispielfälle



Quelle: eigene Darstellung, Grafik generiert mit LDEX 2003 Ralf Künster

Die Kernvariablen des Episodendatensatzes sind die Fallnummer (persnr), das Beginn- und Enddatum der Episoden (begorig und endorig), die Episodenart (sptyp) und die Spell-ID (spell).

Abbildung 4: Datensatz zu den beiden Beispielen aus Abbildung 1[19]

persnr	begorig	endorig	spell	sptyp
11001	01.01.2010	31.01.2011	1	LHG
11001	01.10.2010	31.12.2011	2	BEH
11001	01.12.2010	30.04.2011	3	LHG
11002	01.01.2010	31.03.2011	1	BEH
11002	01.09.2010	31.12.2011	2	LHG

Die Programmschritte im Einzelnen:

- Der Quelldatensatz wird nach Fall-ID, Beginndatum, Enddatum und Episodenart sortiert. Dann wird ein aufsteigender Zähler (spell2) für die neue Sortierung erzeugt.[20] Das Zwischenergebnis wird als temporärer Datenfile gespeichert.
- Alle Beginn-Datumsangaben der Episoden werden zusammen mit der Fall-ID und dem Episodenzähler im Zwischenfile "datum_1" gespeichert.

Abbildung 5: Datensatz "datum_1", generiert aus begorig

persnr	datum
11001	01.01.2010
11001	01.10.2010
11001	01.12.2010
11002	01.01.2010
11002	01.09.2010

- Alle End-Datumsangaben der Episoden werden um eine Zeiteinheit erhöht und mit der Fall-ID und dem Episodenzähler im Zwischenfile "datum_2" gespeichert. Die Erhöhung

19 Die Variable sptyp entspricht der Variablen quelle bzw. quelle_gr in den SIAB-Daten.

um 1 erzeugt aus den End-Datumsangaben potentielle Beginn-Datumsangaben der gesplitteten Teilstücke, die ja um eine Zeiteinheit später beginnen als der davorliegende Split endet (Abbildung 6).

Abbildung 6: Datensatz "datum_2", generiert aus endorig

persnr	datum
11001	01.02.2011
11001	01.01.2012
11001	01.05.2011
11002	01.04.2011
11002	01.01.2012

- Die beiden Dateien datum_1 und datum_2 werden zusammengefügt und nach Fall-ID und Datum sortiert. Pro Fall mehrfach vorkommende identische Datumsangaben werden gelöscht. Zudem werden zwei neue Indikatoren erzeugt: Der eine (y) zählt die Datumsangaben pro Fall aufwärts, der andere (max_t) enthält die höchste Zahl an Datumsangaben, die im gesamten Datenfile für einen Fall auftritt. max_t wird hier nur der Anschaulichkeit halber wie eine Variable behandelt. Innerhalb des Programms splitspells wird max_t nicht als Variable, sondern als Skalar (Konstante) angelegt. Der Zweck der beiden Indikatoren y und max_t wird in den beiden nächsten Absätzen erläutert.

Abbildung 7: Zwischenstand nach den beschriebenen Operationen

persnr	datum	y	max_t
11001	01.01.2010	1	6
11001	01.10.2010	2	6
11001	01.12.2010	3	6
11001	01.02.2011	4	6
11001	01.05.2011	5	6
11001	01.01.2012	6	6
11002	01.01.2010	1	6
11002	01.09.2010	2	6
11002	01.04.2011	3	6
11002	01.01.2012	4	6

- Der Zähler y wird als "j"-Parameter für den folgenden reshape-Befehl benötigt, mit dem die Datumsangaben in das "wide"-Format gedreht werden, so dass sie anschließend fallweise den Episoden der Ursprungsdaten zugespielt werden können:

```
reshape wide datum, i(persnr) j(y)
```

[20] Zufällig stimmt die Ursprungssortierung mit der neuen Sortierung überein, daher sind spell und spell2 in den Beispieldaten identisch, vgl. Abbildung 8.

Abbildung 8: Beispieldatensatz nach dem Hinzufügen der ins "wide"-Format transformierten Datumsangaben

persnr	begorig	endorig	spell	sptyp	spell2	datum1	datum2	datum3	datum4	datum5	datum6	max_t
11001	01.01.2010	31.01.2011	1	LHG	1	01.01.2010	01.10.2010	01.12.2010	01.02.2011	01.05.2011	01.01.2012	6
11001	01.10.2010	31.12.2011	2	BEH	2	01.01.2010	01.10.2010	01.12.2010	01.02.2011	01.05.2011	01.01.2012	6
11001	01.12.2010	30.04.2011	3	LHG	3	01.01.2010	01.10.2010	01.12.2010	01.02.2011	01.05.2011	01.01.2012	6
11002	01.01.2010	31.03.2011	1	BEH	1	01.01.2010	01.09.2010	01.04.2011	01.01.2012			6
11002	01.09.2010	31.12.2011	2	LHG	2	01.01.2010	01.09.2010	01.04.2011	01.01.2012			6

- Der Indikator max_t zeigt an, wie viele Datumsvariablen beim reshape-Befehl insgesamt angelegt wurden und dient im nächsten Schritt als Schleifenzähler für den Abgleich der Datumsvariablen mit den zeitlichen Grenzen einer jeden Episode.
- Nun kann der Abgleich zwischen den Datumsangaben eines Falles und den zeitlichen Grenzen jeder Episode dieses Falles erfolgen. Es wird gezählt, wie viele Datumsangaben in die zeitlichen Grenzen der Episode fallen (n_epidat). Dies ergibt den Parameter für den anschließenden expand-Befehl, n_epidat gibt also an, wie oft eine Episode gesplittet wird. Alle Datumsangaben, die nicht in die zeitlichen Grenzen der jeweiligen Episoden fallen, werden auf missing gesetzt.
- Danach wird in einer Schleife der Index der jeweils ersten Datumsvariablen ermittelt, die einen non-missing-Wert enthält, und in der Variablen first gespeichert. Wenn z.B. die erste gefüllte Datumsvariable datum3 ist, wird der Wert 3 in der Variablen first gespeichert.

Abbildung 9: Beispieldatensatz nach den bisherigen Operationen

persnr	begorig	endorig	spell	sptyp	spell2	datum1	datum2	datum3	datum4	datum5	datum6	max_t	n_epidat	first
11001	01.01.2010	31.01.2011	1	LHG	1	01.01.2010	01.10.2010	01.12.2010				6	3	1
11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		6	4	2
11001	01.12.2010	30.04.2011	3	LHG	3			01.12.2010	01.02.2011			6	2	3
11002	01.01.2010	31.03.2011	1	BEH	1	01.01.2010	01.09.2010					6	2	1
11002	01.09.2010	31.12.2011	2	LHG	2		01.09.2010	01.04.2011				6	2	2

- Nach dem nun folgenden `expand n_epidat`-Befehl sind im Grunde schon alle Episoden des gesplitteten Files angelegt (Abbildung 10).

Abbildung 10: Beispieldatensatz nach dem „expand“-Befehl

persnr	begorig	endorig	spell	sptyp	spell2	datum1	datum2	datum3	datum4	datum5	datum6	max_t	n_epidat	first
11001	01.01.2010	31.01.2011	1	LHG	1	01.01.2010	01.10.2010	01.12.2010				6	3	1
11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		6	4	2
11001	01.12.2010	30.04.2011	3	LHG	3			01.12.2010	01.02.2011			6	2	3
11002	01.01.2010	31.03.2011	1	BEH	1	01.01.2010	01.09.2010					6	2	1
11002	01.09.2010	31.12.2011	2	LHG	2		01.09.2010	01.04.2011				6	2	2
11001	01.01.2010	31.01.2011	1	LHG	1	01.01.2010	01.10.2010	01.12.2010				6	3	1
11001	01.01.2010	31.01.2011	1	LHG	1	01.01.2010	01.10.2010	01.12.2010				6	3	1
11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		6	4	2
11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		6	4	2
11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		6	4	2
11001	01.12.2010	30.04.2011	3	LHG	3			01.12.2010	01.02.2011			6	2	3
11002	01.01.2010	31.03.2011	1	BEH	1	01.01.2010	01.09.2010					6	2	1
11002	01.09.2010	31.12.2011	2	LHG	2		01.09.2010	01.04.2011				6	2	2

Wir rekapitulieren: Der aktive File enthält nun neben allen im Ursprungsfile enthaltenen Merkmalen X Datumsvariablen, wobei X der maximalen Zahl der Datumsangaben entspricht, die in einem Fall des Ursprungsfiles auftrat. Die Datumsvariablen sind für jede einzelne Epi-

sode mit den Datumswerten derjenigen Episoden gefüllt, die zu ihr parallel oder überlappend sind. Die Datumswerte, die nicht in ihre Grenzen fallen, wurden gelöscht. Die Variable first enthält den Index derjenigen Datumsvariablen, in der das Beginndatum des ersten Splittingteilstücks der Ursprungsepisoden gespeichert ist. Außerdem enthält der File nun bereits alle gesplitteten Episoden, jedoch noch ohne die Start- und Enddaten der Splits. Die Episodenzählung ist nicht mehr eindeutig, sondern alle Splits einer Episode enthalten die gleiche Episodennummer wie ihre "Mutterepisode" (im Beispieldatensatz: spell2).

Im Folgenden werden Beginn- und Enddatum der gesplitteten Teilstücke ermittelt.

- Dafür wird zunächst ein Zähler der einzelnen Splits jeder Episode angelegt (spid).
- Der neu generierten Variablen first2 wird der Index derjenigen Datumsvariablen zugewiesen, die dem Beginndatum des jeweiligen Splits entspricht. Dafür nutzen wir die Beziehung zwischen "first" und "spid" auf der einen Seite und dem Index der Datumsvariablen auf der anderen Seite: first gibt den Index der Datumsvariablen an, in der das Beginndatum des ersten gesplitteten Teilstücks zu finden ist, spid zeigt an, um das wievielte Teilstücke der gesplitteten Episode es sich handelt (vgl. Abbildung 11). Daher ergibt die Formel first2 = first + spid -1 den Index der Datumsvariablen, die das Beginndatum des jeweiligen Splits enthält.

Abbildung 11: Beispieldatensatz nach Erzeugung der Variablen first2

	persnr	begorig	endorig	spell	sptyp	spell2	datum1	datum2	datum3	datum4	datum5	datum6	max_t	first	spid	first2
	11001	01.01.2010	31.01.2011	1	LHG	1	01.01.2010	01.10.2010	01.12.2010				6	1	1	1
	11001	01.01.2010	31.01.2011	1	LHG	1	01.01.2010	01.10.2010	01.12.2010				6	1	2	2
	11001	01.01.2010	31.01.2011	1	LHG	1	01.01.2010	01.10.2010	01.12.2010				6	1	3	3
	11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		6	2	1	2
	11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		6	2	2	3
	11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		6	2	3	4
	11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		6	2	4	5
	11001	01.12.2010	30.04.2011	3	LHG	3			01.12.2010	01.02.2011			6	3	1	3
	11001	01.12.2010	30.04.2011	3	LHG	3			01.12.2010	01.02.2011			6	3	2	4
	11002	01.01.2010	31.03.2011	1	BEH	1	01.01.2010	01.09.2010					6	1	1	1
	11002	01.01.2010	31.03.2011	1	BEH	1	01.01.2010	01.09.2010					6	1	2	2
	11002	01.09.2010	31.12.2011	2	LHG	2		01.09.2010	01.04.2011				6	2	1	2
	11002	01.09.2010	31.12.2011	2	LHG	2		01.09.2010	01.04.2011				6	2	2	3

Zur Verdeutlichung des Vorgangs betrachten wir Episode 2 des Falls 11001. Für sie wurde in den vorhergehenden Schritten anhand der Analyse der parallelen Datumsangaben die Zahl von 4 Splits ermittelt. Also wurde Episode 2 vervierfacht (es wurden 3 zusätzliche Kopien angelegt). Diese Splits haben die Spell-ID (spell2) 2 und Split-IDs (spid) von 1 bis 4. Von allen Datumsangaben des Falls 11001 fällt datum2 als erstes in die zeitlichen Grenzen der Episode 2, also bekommt die Variable first für alle 4 Episoden den Wert 2. first2 bekommt für die erste Episode den Wert 2 + 1 - 1 = 2, für die zweite Episode 2 + 2 - 1 = 3 usw. bis zur vierten Episode: 2 + 4 - 1 = 5. Damit ist das Beginndatum des ersten Splits in datum2 zu finden, des zweiten Splits in datum3 usw. bis zum Beginndatum des vierten Splits in datum5.

- Der Variablen für das Beginndatum der Splits (begepi) wird also im nächsten Schritt der Wert derjenigen Datumsvariablen zugewiesen, deren Index mit dem in first2 gespeicherten Wert übereinstimmt.

Abbildung 12: Beispieldatensatz nach Erzeugung des Beginndatums für den Splittingteil auf Basis von "first2"

persnr	begorig	endorig	spell	sptyp	spell2	datum1	datum2	datum3	datum4	datum5	datum6	spid	first2	begepi
11001	01.01.2010	31.01.2011	1	LHG	1	01.01.2010	01.10.2010	01.12.2010				1	1	01.01.2010
11001	01.01.2010	31.01.2011	1	LHG	1	01.01.2010	01.10.2010	01.12.2010				2	2	01.10.2010
11001	01.01.2010	31.01.2011	1	LHG	1	01.01.2010	01.10.2010	01.12.2010				3	3	01.12.2010
11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		1	2	01.10.2010
11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		2	3	01.12.2010
11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		3	4	01.02.2011
11001	01.10.2010	31.12.2011	2	BEH	2		01.10.2010	01.12.2010	01.02.2011	01.05.2011		4	5	01.05.2011
11001	01.12.2010	30.04.2011	3	LHG	3			01.12.2010	01.02.2011			1	3	01.12.2010
11001	01.12.2010	30.04.2011	3	LHG	3			01.12.2010	01.02.2011			2	4	01.02.2011
11002	01.01.2010	31.03.2011	1	BEH	1	01.01.2010	01.09.2010					1	1	01.01.2010
11002	01.01.2010	31.03.2011	1	BEH	1	01.01.2010	01.09.2010					2	2	01.09.2010
11002	01.09.2010	31.12.2011	2	LHG	2		01.09.2010	01.04.2011				1	2	01.09.2010
11002	01.09.2010	31.12.2011	2	LHG	2		01.09.2010	01.04.2011				2	3	01.04.2011

- Nun fehlen noch die Enddatumsangaben der Splits. Diese werden - da gesplittete Episoden per Definition keine Überlappungen mehr aufweisen - einfach als Beginndatum der nachfolgenden gesplitteten Episode minus 1 berechnet. Das Enddatum des jeweils letzten Splits jeder Episode ergibt sich aus dem Enddatum der Ursprungsepisode.

Wie der Beispieldatensatz nach dem Splitting aussieht, zeigt Abbildung 13.

Abbildung 13: Beispieldatensatz nach vollständigem Splitting

persnr	begorig	endorig	spell	sptyp	spid	begepi	endepi
11001	01.01.2010	31.01.2011	1	LHG	1	01.01.2010	30.09.2010
11001	01.01.2010	31.01.2011	1	LHG	2	01.10.2010	30.11.2010
11001	01.01.2010	31.01.2011	1	LHG	3	01.12.2010	31.01.2011
11001	01.10.2010	31.12.2011	2	BEH	1	01.10.2010	30.11.2010
11001	01.10.2010	31.12.2011	2	BEH	2	01.12.2010	31.01.2011
11001	01.10.2010	31.12.2011	2	BEH	3	01.02.2011	30.04.2011
11001	01.10.2010	31.12.2011	2	BEH	4	01.05.2011	31.12.2011
11001	01.12.2010	30.04.2011	3	LHG	1	01.12.2010	31.01.2011
11001	01.12.2010	30.04.2011	3	LHG	2	01.02.2011	30.04.2011
11002	01.01.2010	31.03.2011	1	BEH	1	01.01.2010	31.08.2010
11002	01.01.2010	31.03.2011	1	BEH	2	01.09.2010	31.03.2011
11002	01.09.2010	31.12.2011	2	LHG	1	01.09.2010	31.03.2011
11002	01.09.2010	31.12.2011	2	LHG	2	01.04.2011	31.12.2011

Der Weg vom Episodensplitting- zum Sequenzdatensatz ist nun nicht mehr schwer: Über eine Variable, die - beispielsweise auf Basis von Episodentyp und Spell-ID - eine eindeutige Prioritätenrangfolge festlegt, kann bei Parallelität die Episode mit der höchsten Priorität selektiert und so der für die jeweilige Forschungsfrage geeignete Sequenzdatensatz generiert werden. Der gesplittete File lässt dem Datennutzer hierfür alle Freiheiten und bildet daher eine sehr flexible Grundlage für weitere Datenmodifikations- und Analyseschritte.

Zum Abschluss des Programms wird eine neue Episodenzählung generiert, da die gesplitteten Episoden bis dahin nur über die Kombination der Variablen spell und spid eindeutig identifizierbar sind. Außerdem werden 4 Zählvariablen für gesplittete Episodendaten erzeugt, die ursprünglich teilweise im Lieferumfang der gesplitteten Personendaten des FDZ BA/IAB enthalten waren. Da sie jedoch leicht zu generieren sind, wurde ab der SIAB 7510 darauf verzichtet (FDZ-Datenreport 01/2013: 12). Ohnehin müssen sie nach jedem Episodensplitting neu erzeugt werden.

Wir halten uns hier in Bezug auf die Variablennamen an die Konvention, die das FDZ BA/IAB eingeführt hat. Im Programm splitspells sind sie als default-Werte vorgesehen, können jedoch im Definitionsteil frei geändert werden:

- level1 Zähler für zeitlich parallele Episoden pro Episodentyp und Fall
- level2 Zähler für zeitlich parallele Episoden pro Fall
- nlevel1 Anzahl der zeitlich parallel liegenden Episoden pro Episodentyp und Fall
- nlevel2 Anzahl der zeitlich parallel liegenden Episoden pro Fall

Es folgt die Syntax zur Generierung dieser Zählvariablen. Statt der im Programm verwendeten lokalen Makros benutzen wir hier im Text die Variablennamen der SIAB-Personendaten.

```
by persnr begepi quelle_gr: gen byte level1 = _n-1
by persnr begepi: gen int level2 = _n-1
by persnr begepi quelle_gr: gen byte nlevel1 = _N
by persnr begepi: gen int nlevel2 = _N
```

Damit der Vorteil von gesplitteten Daten, einen Überblick über parallel vorkommende Zustände zu bieten, voll ausgeschöpft werden kann, braucht man Indikatoren, welche die gleichzeitig vorkommenden Zustände bzw. Episodentypen anzeigen. Ihre Konstruktion geschieht mit dem Programm combispells.do, das im nächsten Abschnitt vorgestellt wird.

4.3 Exkurs: Reproduktion des FDZ BA/IAB-internen Episodensplittings

In den gesplitteten SIAB-Daten, wie sie vom FDZ BA/IAB ausgeliefert werden, findet sich folgende Besonderheit: Die Reihenfolge zeitlich paralleler Erwerbstätigkeitsepisoden enthält eine Information über Haupt- und Nebentätigkeit bzw. über die relative Bedeutung hinsichtlich Umfang und Verdienst bei mehreren gleichzeitigen Beschäftigungen. Beschrieben ist diese Eigenschaft in vom Berge/Burghardt/Trenkle (2013), S. 45:

> "Die Sortierung erfolgt grundsätzlich zunächst nach dem Beginndatum der Episode und dann nach der Quelle in der Reihenfolge BeH, LeH, (X)LHG, (X)ASU. Innerhalb der Quelle BeH steht die sozialversiche-rungspflichtige Beschäftigung mit dem höchsten Entgelt vorn, geringfügige Beschäftigung wird nach hinten sortiert. Die Quelle LeH ist nach der Leistungsart sortiert."

Diese spezifische Reihenfolge wird mit dem Programm splitspells nicht wiederhergestellt. Durch eine Vorbehandlung der ungesplitteten Daten mit der am Ende dieses Abschnitts abgedruckten Syntax, mit der Hilfsvariablen für die Sortierung erzeugt werden, kann jedoch eine Reihenfolge erzeugt werden, die nahezu identisch zu der des Original-Datenfiles ist. Wir haben dies mit dem SIAB-SUF siab_r_7508_v2.dta getestet, indem wir ihn entsplittet, mittels der Syntax zusätzliche Sortierkriterien erzeugt und anschließend mit splitspells neu gesplittet haben. Das Ergebnis:

Nur 36 von 25.825.849 BeH-Episoden waren nicht in der richtigen Reihenfolge. Dies war ausschließlich darauf zurückzuführen, dass in diesen Fällen im Originalfile die Reihenfolge "absteigend nach Tagesentgelt" nicht eingehalten war, möglicherweise aufgrund von nachträglichen Datenkorrekturen oder von Rundungsprozessen des vergröberten Merkmals tagentg_gr.

Die Reihenfolge der LEH-, ASU- und LHG-Episoden[21] war in beiden Files identisch.

Falls also gewünscht ist, die Information zu Haupt- und Nebentätigkeiten, die in der Original-Reihenfolge der gesplitteten Original-SIAB-Daten steckt, bei weiteren Aufbereitungs- und Splittingprozessen beizubehalten, müssen die erforderlichen Sortierkriterien vor der Anwendung von splitspells anhand der nachfolgenden Syntax erzeugt werden. Diese werden dann im Definitionsteil von splitspells, im Abschnitt a) unter "optionales zusätzliches Sortierkriterium" wie folgt angegeben:

```
local srt2 "lrt gf -tentgelt_2"
```

Die Beschreibung der Sortierreihenfolge in: vom Berge/Burghardt/Trenkle (2013), S. 45 gibt nicht an, wie mit den missing values verfahren wird. Offenbar werden bei der Sortierung der Original-SIAB-Daten die missing values in den Sortierkriterien so behandelt, als seien sie kleiner als die gültigen Werte. Stata behandelt die missings als größer als alle anderen Werte, daher werden Hilfsvariablen für die Sortierung benötigt, wenn man nicht die Original-Variablen verändern will.

Die Syntax ist für die SUF-Variante der SIAB-Daten entwickelt worden. Für die schwach anonymisierte Version der SIAB, die uns nicht zur Verfügung stand, müssen entsprechende Anpassungen vorgenommen werden.

```
use "<Pfad und Name des ungesplitteten Files>", clear
gen gf = cond(erwstat_gr==3, 1, 0) // Marker für geringfügige Beschäftigung
gen lrt = cond(quelle_gr==2, erwstat_gr, 0) // Hilfsvariable für die Leistungsart
replace lrt = -1 if lrt== .n // missings sollen kleiner als die anderen Werte sein
replace lrt = -2 if lrt== .z
gen tentgelt_2 = tentgelt_gr // Hilfsvariable für das Tagesentgelt
replace tentgelt_2 = -1 if tentgelt_gr== .n
replace tentgelt_2 = -2 if tentgelt_gr== .z
save "<Pfad und Name des vorbehandelten ungesplitteten Files>", replace
```

5 Das Programm combispells.do: Variablen zur Anzeige von Parallelitäten

Die Möglichkeit, sich gleichzeitig vorliegende Zustände anzeigen zu lassen, ist für Datenbereinigung, explorative Analyse von Verlaufsdaten und Hypothesengenerierung sehr hilfreich. So sind z.B. problematische Fälle mit sehr vielen Parallelitäten oder mit unvereinbaren bzw. für unvereinbar gehaltenen Parallelitäten (z.B. Vollerwerbstätigkeit bei gleichzeitiger Arbeitslosigkeit) auf einen Blick identifizierbar, und es lässt sich die quantitative Verteilung der verschiedenen Kombinationen für unterschiedliche Gruppen vergleichen.

Die Abbildung der Episodentyp-Kombinationen kann grundsätzlich in einer numerischen oder einer String-Variablen geschehen. Beides hat Vor- und Nachteile, deshalb erzeugt das Programm combispells beide Varianten. Um die Nutzerfreundlichkeit der numerischen Variante zu erhöhen, werden ihre Ausprägungen zudem programmgesteuert gelabelt.

[21] "LeH", "ASU" und "LHG" sind Kurzbezeichnungen der SIAB für die Datenquellen "Leistungsempfängerhistorik", "Arbeitsuchendenhistorik" und "Leistungshistorik Grundsicherung".

Das Programm soll nur auf gesplittete Daten angewendet werden, also auf Episodendaten, in denen alle Episoden entweder vollkommen parallel oder vollkommen überschneidungsfrei zu den anderen gesplitteten Episoden des gleichen Falles sind.[22]

Wie im Programm splitspells werden in der Syntax temporäre Variablen und Makros verwendet. Im Gegensatz zur Beschreibung von splitspells folgt hier keine Schritt-für-Schritt-Dokumentation des Programms, sondern es werden in den Abschnitten 5.1 bis 5.4 die Konstruktionsprinzipien der Variablen erläutert und der Umgang mit der Stringvariante der Kombinationsvariablen beschrieben. Die in diesen Abschnitten zur Veranschaulichung vorgestellte Syntax ist so formuliert, dass sie außerhalb des Programms combispells verwendet werden kann. Diese Teile sollen nicht zuletzt auch als Anregung für die Entwicklung von Lösungen für ähnliche Problemstellungen dienen. In Abschnitt 5.5 folgt dann die Dokumentation der bis dahin noch nicht behandelten Bestandteile des Programms.

5.1 Das Grundprinzip numerischer Kombinationsvariablen

Eine numerische Kombinationsvariable muss Werte enthalten, welche die eindeutige Identifikation der jeweiligen Zustandskombination der gesplitteten Episoden ermöglichen. Wenn die Episodentyp-Variable z.B. von 1 bis 8 codiert ist, kann die Kombinationsvariable nicht eine einfache Addition der Episodentypcodes sein, denn ein bestimmter Wert der Kombinationsvariablen - z.B. 12 - könnte dann durch unterschiedliche Kombinationen von Episodentypen zustande gekommen sein: z.B. 1, 4, 7 oder 2, 4, 6.

Ein Weg, um den eindeutigen Rückschluss von der Kombinationsvariablen auf die durch sie repräsentierten Episodentypen zu garantieren, ist die Codierung der Episodentyp-Ausprägungen nicht mit fortlaufenden Ziffern, sondern mit den Werten einer Exponentialfunktion a^x. Die Addition der Werte der parallelen Episodentypen ergibt dann Werte, die jeweils nur durch eine einzige Kombination von Episodentypen zustande kommen können.

Leicht verständlich ist das anhand der Exponentialfunktion 10^x - mit x = 0, 1, 2, …n ergibt sie die Folge 1, 10, 100 …10^n. Wenn die Episodentypen also nicht wie üblich mit 1, 2, 3 … n codiert sind, sondern mit 1, 10, 100… dann ergibt die Addition der Werte der Episodentypen jeweils eine Folge von 1-0-Kombinationen, die eindeutig auf die Ursprungs-Episodentypen verweisen. Z.B. ist 1001 die Kombination der Episodentypen 1 und 4. Eine Methode zur Erzeugung einer solchen, als "Bitmustervariable" bezeichneten Kombinationsvariablen mit Stata wird in Abschnitt 3.3 des bereits erwähnten FDZ-Methodenreports 06/2007 beschrieben. Weiter unten im vorliegenden Methodenreport (S. 32) folgt die Lösung, die wir entwickelt haben.

Bei einer kleinen Menge unterschiedlicher Episodentypen (etwa 4-5) sind Bitmustervariablen auf einen Blick interpretierbar. Die Episodentyp-Variable des Kalendariums der SOEP-Daten[23] über alle bisherigen Panelwellen hinweg hat jedoch 16 Ausprägungen, und es könn-

[22] Das gilt, wenn das Programm als Ganzes benutzt wird. Mit Modifikationen sind die Prozeduren auch für ungesplittete Daten anwendbar, wenn z.B. nicht die Episodentypen zeitgleicher Splits, sondern die in einem Verlauf insgesamt vorkommenden Episodentypen ermittelt werden sollen.

[23] Sozioökonomisches Panel, File "artkalen$$"

ten in zukünftigen Wellen weitere hinzukommen. Je mehr Episodentypen es gibt, desto mehr Stellen hätte die Bitmustervariable und sie könnte nicht mehr auf Anhieb interpretiert werden.

Da also parallele Episodentypen bei vielen Ausprägungen mittels Bitmuster-Variablen ohnehin nicht mehr visuell erfasst werden können, bietet es sich an, hier Exponentialfunktionen zur Basis 2 zu verwenden, also 2^x, was wesentlich kleinere Zahlen erzeugt: 6 Ausprägungen können immerhin noch in einer nur maximal 2-stelligen Zahl (63) abgebildet werden, statt in einer 6-stelligen (111111). Natürlich wäre man ohne Labels hilflos, was die Deutung der Zahlen anbelangt. Dass sich z.B. hinter "35072" die Episodentypen a, b und c verbergen, codiert mit 256, 2048 und 32768, kann ein menschliches Gehirn in der Regel nicht mehr in überschaubarer Zeit entschlüsseln.

Hier helfen Labels. Jedoch ist die Idee, die entsprechenden Labeldefinitionen 'per Hand' zu erstellen, wenig attraktiv, da bereits für 4 Episodentypen 15 verschiedene Kombinationsmöglichkeiten existieren - für 16 Episodentypen sind es 65.535 mögliche Kombinationen[24]. Auch wenn in einem gegebenen File bei weitem nicht alle theoretisch möglichen Kombinationen auch tatsächlich auftreten, ist klar, dass man sowohl eine automatisierte Berechnung der in einem File auftretenden Episodentyp-Parallelitäten als auch eine automatische Label-Erstellungs-Routine braucht.

Für beides bietet das Programm combispells Lösungen in Stata. Es generiert außerdem sowohl Kurzlabels wie die 2 hoch x-Codierung der Episodentypen, so dass der Nutzer sich nicht damit aufhalten muss, diese selbst zu erstellen.

Wer nicht allzu viele Ausprägungen in seiner Episodentyp-Variablen hat, oder wem die String-Variante der Kombinationsvariablen ausreicht, der kann sich anhand der Syntax der folgenden Abschnitte sein eigenes kleines Programm zusammenstellen, etwa auch zur Erzeugung einer numerischen Bitmuster-Variablen zur Anzeige der Parallelitäten.

Da die Belabelung der numerischen Kombi-Variablen auf der String-Kombi-Variablen aufbaut, wird nun zuerst die Konstruktion der String-Variablen beschrieben.

5.2 Eine String-Kombinationsvariable erzeugen und nutzen

Im Gegensatz zu der numerischen Kombinationsvariablen beruht die Konstruktion der String-Kombinationsvariablen nicht darauf, dass die Codes der Episodentyp-Variablen Werte einer Exponentialfunktion sind. Vielmehr werden die Labels einer numerischen Episodentyp-Variablen benutzt, um die String-Kombinationsvariable zu erzeugen. Die Ausprägungen der String-Kombinations-Variablen bestehen aus den aneinandergereihten Labels der jeweils vorliegenden Episodentyp-Parallelitäten.

Somit versteht sich, dass die Labels kurz sein müssen und möglichst selbsterklärend sein sollten. Außerdem - dies ist wichtig - darf ein Label nicht in der Zeichenfolge eines anderen Labels enthalten sein. Das Label "ASU" würde diese Anforderung nicht erfüllen, wenn

[24] Berechnet als 2 hoch 16, minus 1 (weil die Kombination "gar kein Episodentyp" ausgeschlossen werden kann).

gleichzeitig das Label "XASU" vorkommt. Wenn Labels in der Zeichenfolge anderer Labels enthalten sind, können sie später nicht mit if-Bedingungen zielsicher selektiert werden.

Der Weg zu der Kombi-String-Variablen führt über einen Zwischendatensatz, der nur die Variablen persnr (Fall-ID), begepi (Beginn der gesplitteten Episode) und die mit Kurzlabeln versehene Variable spelltyp enthält, und zwar ohne Doppelungen, d.h. Datenzeilen mit gleichen Werten in diesen drei Variablen werden eliminiert. Damit wird erreicht, dass parallele Episoden des gleichen Typs nicht mehrmals ihr Label in den String der Kombinationsvariablen eintragen.[25]

```
use <…>, clear
keep persnr begepi spelltyp
duplicates drop persnr begepi spelltyp, force
```

Der decode-Befehl erzeugt aus der gelabelten numerischen Variablen spelltyp die Stringvariable sptstr und weist ihr als Ausprägungen die Labels von spelltyp zu:

```
decode spelltyp, gen(sptstr)
```

Der Zwischendatensatz wird so sortiert, dass pro Fall alle zeitgleichen Episoden in der Reihenfolge der Episodentypen aufeinanderfolgen, dann wird die Variable combistr als Kopie von sptstr angelegt. Sie dient dazu, die Kurzlabels der jeweils zeitgleichen Episodentypen "einzusammeln".

```
sort persnr begepi spelltyp
gen str combistr = sptstr
```

Es folgt das schrittweise Einsammeln der Kurzlabels der zeitgleichen Episoden:

```
by persnr begepi: replace combistr = combistr[_n-1]+"+"+combistr if _n > 1
```

Als Resultat enthält die Variable combistr in jeder Zeile jeder Gruppe zeitgleicher Episoden pro Fall ein Kurzlabel mehr: die Kurzlabels der vorhergehenden Episoden plus das der aktuellen Episode. Infolgedessen enthält combistr nur in der letzten Episode jeder Gruppe das vollständige Set an Kurzlabeln aller beteiligten Episodentypen. Dieses muss nun im nächsten Schritt auf die Variable combistr in den anderen Zeilen der Gruppe übertragen werden. Dazu wird der File so umsortiert, dass jede zeitgleiche Episodengruppe mit dem vollständigen String beginnt. Der Wert von combistr der ersten Episode wird dann für die nachfolgenden Episoden übernommen:

```
by persnr begepi: gen temp = _n
gsort persnr begepi -temp
```

[25] Falls mehrfaches Vorkommen jedoch ausdrücklich angezeigt werden soll, kann der duplicates drop-Befehl in der aufgelisteten Befehlssequenz bzw. im Programm combispells.do auskommentiert werden. Man sollte vorher aber kontrollieren, wieviele parallele Episoden des gleichen Typs maximal vorkommen. Bis Stata 12 waren String-Variablen auf 244 Zeichen beschränkt, in Version 13 sind es, abgesehen von dem neuen StrL-Variablentyp, 2.045 Zeichen. Gemessen an dem Anwendungszweck sollte die Länge des erzeugten Strings übersichtlich bleiben. Die Information, wieviele parallele Episoden des gleichen Typs vorliegen, kann man auch der Variablen level1 entnehmen (s. Seite 14 dieses Reports). Darüber hinaus ist zu beachten, dass die *numerische* Kombinationsvariable dann zwar erzeugt wird, aber nicht genutzt werden kann, da die eindeutige Zuordnung zwischen Zahlenwert und einer bestimmten Kombination von Episodentypen nicht mehr gegeben ist, wenn gleiche Episodentypen mehrmals in einer Kombination vertreten sind (z. B.: 1 + 2 ergibt 3, aber 3 * 1 ergibt ebenfalls 3).

```
by persnr begepi: replace combistr = combistr[_n-1] if _n>1
drop temp
```

Die Variable combistr enthält nun die jeweils beteiligten Kurzlabel, getrennt durch ein "+"-Zeichen, damit sie besser lesbar sind. Über die Schlüsselvariablen persnr, begepi und spelltyp wird die neue Variable an den Ursprungsdatensatz 1:m (one-to-many) angefügt. Man kann aber auch zuerst die numerische Kombinationsvariable und ihre Labels wie in Abschnitt 5.4 beschrieben erzeugen, und die neuen Variablen erst im Anschluss daran an den ursprünglichen gesplitteten File anhängen.

Die obige Syntax kann leicht an den Zweck angepasst werden, die Episodentypen zu ermitteln, die in einem gesamten Verlauf (statt in parallelen Episodensplits) vorkommen. Man muss in diesem Fall die Variable begepi aus der Gruppierungsdefinition ("by"-Bestimmung) entfernen, so dass die Gruppierung nur anhand der Variablen persnr vorgenommen wird.[26]

Diese Syntax eignet sich auch, um andere Kennwerte eines Verlaufes als den Episodentyp "aufzusammeln".

Im folgenden Abschnitt werden Hinweise zum Umgang mit den resultierenden, u.U. recht langen Ausprägungsketten der Variablen gegeben.

5.3 Umgang mit der String-Kombinationsvariablen

Zwei Punkte sind wichtig, um die neu erzeugte Variable nutzen zu können:

- Da der Gebrauchswert der Variablen vor allem in ihrer Lesbarkeit für Menschen liegt, müssen die Ausprägungen in einer Häufigkeitstabelle angezeigt werden können, ohne rechts abgeschnitten zu werden.
- Man muss wissen, wie man bestimmte Ausprägungen der Variablen innerhalb von if-Bedingungen anspricht.

5.3.1 Häufigkeitstabellen mit fre.ado

Die Stata-Befehle tab und table ermöglichen nicht, die linke Spalte einer Häufigkeitstabelle so breit einzustellen, wie es für die Anzeige der Variablen combistr nötig wäre. Bei 16 Ausprägungen und Kurzlabels von 4-5 Zeichen sowie dem 'plus'-Zeichen zwischen jedem Label kommt man auf z.B. ca. 90 Zeichen, die maximal angezeigt werden müssen.

Der ado-File "fre" leistet das jedoch. Er kann mit folgendem Stata-Befehl heruntergeladen, werden, falls er noch nicht installiert ist:

```
. ssc install fre, replace
```

In EDV-Umgebungen, in denen eine direkte Installation von Stata aus nicht gestattet ist, lädt man den Befehl unter http://fmwww.bc.edu/repec/bocode/f/ herunter und speichert ihn in das Default-Ado-Verzeichnis (dieses wird mit dem Stata-Befehl sysdir angezeigt).

[26] Um diese Änderung für das Programm combispells.do zu übernehmen, müssen alle Vorkommen von "`by `id' `bege'`" gesucht und durch "`by `id' `" ersetzt werden.

Wenn fre zur Verfügung steht, kann bei Bedarf die Output-Zeilenbreite vergrößert und die Verteilung von combistr mit fre statt mit tab ausgegeben werden. Im folgenden Befehl wurde die Output-Zeilenbreite auf 240 Zeichen verbreitert. Wichtig sind die Optionen nowrap (damit die Ausprägungen in eine einzige Zeile geschrieben werden) und all (bei vielen Ausprägungen unterdrückt fre sonst die Ausgabe eines Teils der Zeilen per default).

```
set linesize 240
fre combistr, nowrap all
```

5.3.2 Ausprägungen von combistr selektieren

Wenn man "==" verwendet, um eine bestimmte Ausprägung von combistr zu selektieren, muss die Bedingung Zeichen für Zeichen mit der Ausprägung übereinstimmen. Es ist meist besser, die Stringfunktion strpos() zu verwenden, wenn es z.B. darum geht, alle Episoden zu selektieren, die eine bestimmte Kombination von zeitgleichen Episodentypen aufweisen.[27]

Die Funktion strpos() ermittelt die Position, an der eine bestimmte Zeichenfolge innerhalb des gesamten Strings vorkommt.

```
gen byte h1=strpos(combistr, "ASU")
```

Die Position ist der Zeichenfolge ist nur dann größer Null, wenn das betreffende Label in der Kombination vertreten ist. Damit können if-Bedingungen formuliert werden:

```
… if strpos(combistr, "ASU") > 0
```

Die Bedingung

```
… if strpos(combistr, "ASU+LHG") > 0
```

ermittelt jedoch nur dann alle Episoden, die sowohl zu einem ASU- als auch zu einer LHG-Episode parallel sind, wenn ASU- und LHG-Episodentyp in der Sortierreihenfolge direkt aufeinander folgen. Wenn das nicht der Fall ist, könnte sich ein anderer Episodentyp mit seinem Kurzlabel dazwischengeschoben haben. Besser ist es daher, die Bedingung wie folgt zu formulieren:

```
… if strpos(combistr, "ASU") > 0 & strpos(combistr, "LHG") > 0
```

Es kommt allerdings vor - z.B. in der Datenprüfung -, dass gerade Fälle bzw. Zeiträume mit sehr vielen Parallelitäten genauer unter die Lupe genommen werden sollen. Dies ist ein Anwendungsfall, wo eine numerische Kombinationsvariable deutlich praktischer ist. Statt das gesamte, vielleicht 70 Zeichen umfassende Label anzugeben, kann der Zahlencode der Kombination für die if-Bedingung verwendet werden.

5.4 Eine numerische Kombinationsvariable konstruieren und belabeln.

Unter der Voraussetzung, dass die Episodentypvariable des gesplitteten Datenfiles bereits wie in Abschnitt 5.1 beschrieben mit den Werten einer Exponentialfunktion codiert ist, gestaltet sich die Erzeugung der numerischen Kombinationsvariablen einfach als gruppenbezogene Addition:

```
by persnr begepi: egen combi=total(spelltyp)
```

[27] Weitere String-Funktionen werden von Stata aus mit "`help string functions`" angezeigt.

```
format combi %20.0f
```

Die Belabelung dieser Variablen fällt dagegen deutlich komplexer aus als ihre Erzeugung.

Im Prinzip könnte man wegen der eindeutigen Beziehung zwischen den beteiligten Episodentypen und dem Wert der numerischen Kombinationsvariablen (combi) die Labels rechnerisch aus combi ableiten. Dafür muss aber bekannt sein, welche Exponentialfunktion den Werten des Episodentyps zugrunde liegt. Im Programm combispells wäre diese Voraussetzung erfüllt, denn dort braucht der Nutzer nur die Original-Werte seiner Episodentyp-Variablen und die Kurzlabels anzugeben. Die Umcodierung mit Werten der Funktion 2^x erledigt das Programm. Wir haben jedoch einen anderen Weg gewählt, die Labels zu ermitteln, der den Vorteil hat, für alle Exponentialcodierungen anwendbar zu sein, wenn die Syntax außerhalb des Programms combispells verwendet wird, etwa für Bitmustervariablen.

Der Lösungsweg besteht darin, die vorher erzeugte String-Variable combistr in die Ermittlung der Labels einzubeziehen. Das funktioniert wie folgt: Der Datensatz wird nach den Werten von combi aufsteigend sortiert, dann wird in einer Schleife der gesamte Datenfile Zeile für Zeile durchlaufen. Jedesmal, wenn ein neuer Wert von combi auftritt, werden die Werte von combi und combistr paarweise entsprechend der Syntax-Anforderungen für label-do-files in eine externe Text-Datei geschrieben. Dieser Label-do-File kann dann aufgerufen werden, um die Variable combi zu belabeln.

Es folgt die Syntax - im Programm combispells wurde mit lokalen Makros gearbeitet, hier wird die Syntax im Wesentlichen ohne Makros aufgelistet. Der aktive Datenfile ist entweder der vollständige, gesplittete Episodenfile oder der bei der Erzeugung von combistr benutzte Zwischenfile (der weniger Datenzeilen hat und deshalb schneller durchlaufen wird).

Wenn man aus Stata heraus programmgesteuert externe Dateien zum Lesen oder Schreiben heranziehen will, müssen sogenannte "file handler" definiert werden - in der folgenden Syntax ist das "d1".

- Zum Abfangen eines Programmabbruchs, falls d1 bereits offen ist:

```
capture file close d1
```

- Definieren und öffnen des Textfiles zum schreiben - er wird überschrieben, falls er bereits existiert, falls er noch nicht existiert, wird er erzeugt:

```
file open d1 using <pfadangabe>/labels_combi.do, write replace
```

- Definition lokaler Macros, um die Ausgabe von Sonderzeichen und Standard-Textzeilen zu vereinfachen:

```
local t = char(9)   // der Tabulator
local q = char(34)   // das doppelte Anführungszeichen
local header "lab def combi `t' /// "
local foot ", replace"
```

- Den Datenfile aufsteigend nach den Werten der numerischen Kombinationsvariablen sortieren, die Kopfzeile in die Label-Datei schreiben, an den Anfang der nächsten Zeile gehen ("_n"):

```
sort combi
file write d1 "`header'"_n
```

- Die Schleife so oft wiederholen, wie es Zeilen im Datensatz gibt:

```
local k= _N
forvalues i = 1(1)`k' {
```

- nur wenn ein neuer Wert auftritt, werden die Makros a und b mit den Werten von combi und combistr in der jeweils aktuellen Datenzeile gefüllt:

```
   if combi[`i'] !=  combi[`i'-1] {
         local a = combi[`i']
         local b = combistr[`i']
```

- Danach wird eine komplette Zeile der Label-Definition zusammengestellt und in den Textfile geschrieben. Wichtig sind die compound-Anführungszeichen (**`"** bzw. **"'**), mit denen die Textzeile eingeschlossen wird, weil innerhalb der Zeichenfolge doppelte Anführungszeichen vorkommen:

```
         file write d1 `" `t' `a' `t' `q'`b'`q' `t' ///"'_n
```

- Am Ende die schließenden Klammern der if-Bedingung und der forvalues-Schleife nicht vergessen:

```
   }
}
```

- Nachdem alle Datenzeilen abgearbeitet sind, wird die Fußzeile in die Label-Datei eingefügt:

```
file write d1 "`foot'"_n
```

- Sowie eine letzte Zeile, die unbedingt erforderlich ist, sonst wird der label-do-file ohne Fehlermeldung nicht ausgeführt:[28]

```
file write d1 _n
```

- Den File-Handler schließen:

```
capture file close d1
```

- Und die Zuweisung der Labels vornehmen:

```
do <Pfadangabe>/labels_combi.do
lab val combi combi
```

- Falls nicht schon geschehen, werden die neuen Variablen an den Ursprungsfile angefügt:

```
merge 1:m persnr begepi spelltyp using <Pfad/Dateiname>
```

Im Ergebnis gibt es nun eine gelabelte numerische Variable der auftretenden Episodentyp-Parallelitäten und eine String-Variable. Beide zeigen die gleiche Information, eignen sich aber für unterschiedliche Zwecke.

[28] Vermutlich fehlt ohne diese Zeile das von Stata erwartete Dateiendezeichen.

Zur Anzeige der gelabelten numerischen Variablen sollte wegen der Breite der Labels ebenfalls "fre" statt tab verwendet werden. Der erzeugte Label-Do-File ist auch als Konkordanztabelle zwischen numerischen Werten und Labels sehr nützlich.

5.5 Erläuterung der zusätzlichen Programmbestandteile in combispells.do

Wie das Programm splitspells arbeitet auch combispells durchgängig mit lokalen Makros, so dass nur im Definitionsteil bestimmte Parameter gesetzt werden müssen. Weitere Anpassungen der Syntax sind nicht erforderlich.

Zwischenfiles und vorübergehend genutzte Variablen werden unter Verwendung von tempfile und tempvar erzeugt. Der Ausgangsfile wird mit "preserve" geschützt, weshalb der Ergebnisfile nach Programmende nicht wie gewohnt im Arbeitsspeicher zur Verfügung steht, sondern eigens aufgerufen werden muss.

Da der Output des Programms über die Wiederholung der Syntaxzeilen hinaus nur wenig nützliche Information bietet, wurde er mit Ausnahme bestimmter Meldungen mit dem Befehl "set output error" unterdrückt. Wenn das nicht gewünscht ist, können alle "set output error"-Zeilen mit find-and-replace auskommentiert werden.

combispells bietet ein in den vorigen Abschnitten noch nicht beschriebenes programmgesteuertes Feature: die Zuweisung von Kurzlabels und von Werten einer Exponentialfunktion an eine Episodentyp-Variable. Dieser Programmbestandteil sowie der Definitionsteil werden im Folgenden erläutert. Die anderen Programmteile dürften anhand der bisherigen Darstellung trotz Makroverwendung ohne weitere Erklärungen verständlich sein.

5.5.1 Der Definitionsteil von combispells

Im Abschnitt "manuelle Vergabe von Parametern" des Definitionsteils müssen die eingetragenen Defaultwerte überprüft und an lokale Gegebenheiten angepasst werden. Folgende Defaults können unverändert übernommen werden, ohne dass ein Fehler erzeugt wird:

- Name der Logdatei
- Variablennamen der im Programmablauf neu erzeugten Variablen
- Name der Ergebnisdatei

Unbedingt anpassen muss man dagegen:

- Pfadangaben (Dateispeicherorte)
- Name der Quelldatei mit den Variablennamen für Fall-ID, Beginn-Datum der gesplitteten Episode und Episodentyp
- Die beiden Listen 'typcodes' und 'kurzlabels': In die Liste typcodes sind die codes der angegebenen Episodentyp-Variablen einzutragen[29]. In die Liste kurzlabels sind frei zu vergebende Kurzlabels für die Codes der angegebenen Episodentyp-Variablen einzutra-

gen. Die Zahl der Eintragungen und die Reihenfolge beider Listen müssen übereinstimmen. Die Einträge sind jeweils durch ein Leerzeichen zu trennen.

Wichtig: Die Einträge im Abschnitt "automatische Vergabe von Parametern" dürfen nicht verändert werden.

Nach dem Definitionsteil folgt die Überprüfung, ob die Zahl der Einträge in den beiden Listen typcodes und kurzlabels übereinstimmt - falls nicht, bricht das Programm mit einer entsprechenden Meldung ab. Ob die Semantik stimmt, ob also die inhaltliche Reihenfolge übereinstimmt, kann nicht geprüft werden. Dafür muss der Nutzer sorgen.

5.5.2 Automatische Zuweisung von Werten der Exponentialfunktion 2^x und Vergabe der Kurzlabels

Die Umcodierung und Belabelung geschieht an einer Kopie der vorhandenen Episodentypvariablen (im folgenden Syntaxauszug: "epixp"). Hier kommen die im Definitionsteil gefüllten Listen typcodes und kurzlabels zum Einsatz.

Beide Vorgänge geschehen innerhalb der gleichen Schleife, die so oft durchlaufen wird, wie es Einträge in typcodes bzw. kurzlabels gibt. Die Zahl der Einträge stimmt in beiden Listen überein und wurde im Teil "automatische Vergabe von Parametern" an das lokale Makro `n' übergeben. Die Syntax zum Öffnen und Beschreiben der Label-Textdatei außerhalb der Schleife wurde oben schon beschrieben und wird hier der Übersichtlichkeit halber unterdrückt. Die Schleife lautet:

```
forvalues i = 1(1)`n' {
      replace `epixp'=2^(`n'-`i') if `epixp'==real(word("`typcodes'",-`i'))
      local a = 2^(`n'-`i')
      local b = word("`kurzstrings'",-`i')
      file write d1 `" `t' `a' `t' `q'`b'`q' `t' ///"'_n
}
```

Hier folgt die Schritt-für-Schritt-Erläuterung:

Die Zuweisung der Werte der Exponentialfunktion 2^x an die Variable epixp geschieht mit dem Befehl:

```
      replace `epixp'=2^(`n'-`i') if `epixp'==real(word("`typcodes'",-`i'))
```

Der Befehl sagt: "ersetze epixp mit 2 hoch (der Zahl der Listeneinträge minus dem aktuellen Wert des Schleifenzählers i), falls epixp gleich <…> ist".

Angenommen, die Zahl der Einträge in den Listen typcodes bzw. kurzlabels beträgt 8, so lautet der Wert von epixp im ersten Durchgang 2 hoch 7, im zweiten Durchgang 2 hoch 6 und so fort, bis er im letzten Durchgang 2 hoch 0 beträgt. In umgekehrter Reihenfolge werden also die Werte 1, 2, 4, 8, 16, 32, 64 und 128 erzeugt. Die umgekehrte Reihenfolge ist zwingend notwendig, damit der replace-Befehl bildlich gesprochen die Werte am oberen Ende des Wertebereichs weiter nach rechts schiebt und damit Platz auf dem Zahlenstrahl für

[29] Dabei kommt es auf die Codes an, die in den Daten tatsächlich vorkommen. Es ist nicht erforderlich (schadet aber auch nicht), Codes anzugeben, die im Codeplan oder in der Label-Definition vorhanden sind, in den Daten aber nicht auftauchen.

die nachfolgenden Umcodierungen schafft. Wenn man nicht so vorginge, würden bereits umcodierte Werte erneut umcodiert: 1 würde zu 1, 2 zu 4, 3 zu 8, 4 zu 16 - jedoch besteht ein Teil der Ausprägungen "4" aus bereits umcodierten Ausprägungen 2!

Die beschriebene Ersetzung der Werte von epixp wird in dem replace-Befehl an die Bedingung geknüpft:

```
...if `epixp'==real(word("`typcodes'",-`i'))
```

Hier finden wir zwei ineinandergeschachtelte Funktionen: real(word()). Die Funktion `word("`typcodes'",-`i')` ermittelt den minus i-ten Eintrag in der Liste typcodes, das bedeutet: den i-ten Eintrag vom Ende her gezählt, also auch hier: die umgekehrte Reihenfolge. Da epixp eine numerische Variable ist, die Funktion word() jedoch Strings zurückgibt, muss die Zeichenfolge mit der Funktion real() in einen numerischen Datentyp umgewandelt werden.

Die Erzeugung einer Bitmuster-Variablen (vgl. Abschnitt 5.1) ist auf die gleiche Weise möglich. Statt `replace `epixp'=2^(`n'-`i') if <…>` muss es in diesem Fall `replace `epixp'=10^(`n'-`i') if <…>` heißen, der restliche Befehl bleibt gleich. Im Programm combispells müsste auch die Zuweisung `local a = 2^(`n'-`i')` in der folgenden Zeile entsprechend in `local a = 10^(`n'-`i')` geändert werden.

Die Liste typcodes enthält die ursprünglichen Codes der Episodentyp-Variablen. Die replace-Befehlszeile ordnet also die neuen, durch eine Exponentialfunktion bestimmten Codes den ursprünglichen Codes zu. Innerhalb der forvalues-Schleife wird jedoch noch mit dem "file write"-Befehl der Label-Do-File erzeugt, der die neuen Codes der Episodentypvariablen und die dazugehörigen Kurzlabels auflistet. Den Makros `t' und `q' enthalten das Tabulatorzeichen bzw. ein doppeltes Anführungszeichen. Wie die Syntax zur Erstellung des Label-Do-Files im Einzelnen funktioniert, haben wir bereits in Abschnitt 5.4 auf S. 29 beschrieben.

Nach der Umcodierung kann die numerische Kombinationsvariable, wie in Abschnitt 5.4 beschrieben, mit

```
by `id' `bege': egen `coq'=total(`epixp')
```

erzeugt werden.

6 Resumé und Ausblick

Das von uns entwickelte Programm splitspells ist gegenüber anderen Verfahren des Episodensplitting in der Lage, auch sehr umfangreiche prozessproduzierte Datensätze, die tagesgenau erfasst wurden - wie etwa die SIAB-Daten - in überschaubarer Zeit zu splitten und für die Generierung von Sequenzdaten vorzubereiten. Die Besonderheit des Programms liegt einerseits in der effektiven Gestaltung des Splitting-Algorithmus und andererseits in der portionsweise Verarbeitung der Daten, die nach der Verarbeitung wieder zu einem Gesamtdatensatz zusammengefügt werden.

Das in Abschnitt 0 beschriebene Programm combispells stellt eine nützliche Ergänzung zu splitspells dar und unterstützt die DatennutzerInnen bei der Aufbereitung und Edition der Daten.

Wir beabsichtigen, Kernbestandteile der Programme splitspells und combispells als ado-files auszuarbeiten und die Programme auch mit englischer Kommentierung zur Verfügung zu stellen. Zudem arbeiten wir an einer Übertragung der Programme in SPSS-Syntax. Für splitspells liegt bereits eine vorläufige SPSS-Fassung (bislang ohne die Aufteilung in Portionen) vor. Bei Interesse wenden Sie sich bitte an die Autoren.

7 Anhang

7.1 splitspells

Hier abgedruckt: splitspells_de_v02.do (Version 02, deutsche Fassung)

```
version 13       /* verwendete Stata-Version */
set output error

/*#####################################################################################################
#######################################################################################################
######################                                                                ######################
######################    Stata-Episodensplitting-Programm zum Einsatz für ungesplittete    ######################
######################    Episodendaten                                               ######################
######################                                                                ######################
######################    entwickelt von Ralf Künster und Klaudia Erhardt             ######################
######################                                                                ######################
######################    Stand: 03.07.2014                                           ######################
######################                                                                ######################
####                                                                                               ####
####   Das Programm erzeugt aus Episodendaten (Spelldaten) mit zeitlich überschneidenden Episoden gesplittete   ####
####   Episoden, die entweder vollkommen parallel zu anderen Episoden oder Espisodensplits des gleichen Falls   ####
####   sind oder keine Parallelität aufweisen.                                                      ####
####   (vgl. z.B. SIAB-Datenreport 1-2013, http://doku.iab.de/fdz/reporte/2013/DR_1-13.pdf, Abschnitt 3.2)     ####
####                                                                                               ####
####   Das Splitting-Prinzip dieses Programms basiert auf dem fallweisen Abgleich aller Start- und Enddatums-   ####
####   angaben der Episoden mit jedem Episodenzeitraum. Dieser Abgleich ermittelt, wie oft und an welchen      ####
####   Zeitpunkten eine Episode gesplittet werden muss. Im Anschluss erfolgt die Umsetzung des Splittings.     ####
####                                                                                               ####
####   Zur Bearbeitung sehr großer Datensätze wird der Quelldatensatz vor dem Episodensplitting in eine vom    ####
####   Nutzer zu bestimmende Zahl von 'Portionen' aufgeteilt, die nach dem Splitting zu dem Zieldatensatz      ####
####   zusammen gefügt werden. Bei kleineren Datensätzen kann die Aufteilung umgangen werden, indem die  Zahl   ####
####   der Portionen auf 1 gesetzt wird.                                                            ####
####                                                                                               ####
####   Das Programm enthält zu Anfang einen Definitionsteil, in dem Pfadangaben sowie Datei- und Variablennamen ####
####   festgelegt werden.                                                                           ####
####                                                                                               ####
####   ACHTUNG, WICHTIGER HINWEIS:                                                                  ####
####   - Die Parameter im Definitionsteil a) müssen vom Nutzer zwingend festgelegt werden, da sie die nutzer-  ####
####     spezifischen Umgebungsvariablen und Daten beschreiben.                                     ####
####   - Die Parameter im Definitionsteil b) können vom Nutzer geändert werden, falls gewünscht ist, andere     ####
```

```
####     Variablen- bzw. Dateinamen zu verwenden.                                                               ####
####   - Die Parameter im Definitionsteil c) sollten keinesfalls vom Nutzer geändert werden, weil sie in dieser ####
####     Form vom Programm benötigt werden.                                                                     ####
####                                                                                                            ####
####                                                                                                            ####
####   Weitere Anpassungen des Programmcodes sind nutzerseitig nicht erforderlich.                              ####
####                                                                                                            ####
####################################################################################################################
##################################################################################################################*/

set more off
set varabbrev off

/*##################################################################################################################
####################################################################################################################
####################################                           #####################################################
####################################  Definitionsteil Anfang   #####################################################
####################################                           #####################################################
####################################################################################################################
##################################################################################################################*/

/*##################################################################################################################
############################                                           #############################################
############################   a)  MANUELLE VERGABE VON PARAMETERN     #############################################
############################       ZWINGEND ERFORDERLICH               #############################################
############################                                           ###########################################*/

/* Zahl der Portionen, in die der File zerlegt werden soll:  > 0  und < Fallanzahl (--> Personen, nicht Spells).
       Keine Zerlegung: n = 1 */
local n = 1000

/* Definition der nutzerspezifischen Pfade und Dateinamen - ANGABE OHNE "/" AM ENDE, da der Slash im Programm
       automatisch hinzugefügt wird */
local pfad "H:/Spellsplitting"                /* Arbeitsverzeichnis */
local pfad_d "H:/Spellsplitting/Datenfiles"   /* Verzeichnis für Quell- und Zieldatensatz */
local pfad_o "H:/Spellsplitting/Output"       /* Verzeichnis, in das die Logdatei geschrieben wird */
                                              /* HINWEIS: pfad, pfad_d und pfad_o können das gleiche Verzeichnis bezeichnen,
                                                 aber es müssen alle drei Angaben gemacht werden */
local quelle "siab_r_7508-uspV1.dta"          /* Name des Quelldatensatzes */
local ziel "siab_r_7508-gspV1.dta"            /* Name des gesplitteten Ergebnisdatensatzes */


/* Definition quelldatenspezifischer Variablennamen. DIESE MÜSSEN IM ZU BEARBEITENDEN DATENFILE VORHANDEN SEIN. */
local beg "begorig"                           /* Beginndatumvariable der Episoden im Quelldatensatz */
```

```
local end "endorig"                         /* Enddatumvariable der Episoden im Quelldatensatz */
local id "persnr"                           /* Fallidentifizierungsvariable im Quelldatensatz */
local epityp "quelle_gr"                    /* Episodentypvariable im Quelldatensatz */
local sp "spell"                            /* Episodenidentifikationsvariable im Quelldatensatz */

/* Optionales zusätzliches Sortierkriterium (einzelne Variable oder Variablenliste): regelt die Reihenfolge von
   parallelen Episodensplits des gleichen Typs.
   - Kein zusätzliches Sortierkriterium gewünscht: local srt2 ""
   - Absteigende Sortierreihenfolge wird durch ein vorangestelltes "-" erzeugt.
     Beispiel: local srt2 "lrt gf -tentgelt_2" */
local srt2 ""

/*##########################################################################################################
#######################                                                    ########################################
#######################  b)  MANUELLE ÄNDERUNGEN DER VORGABEN MÖGLICH,     ########################################
#######################      ABER NICHT ERFORDERLICH                       ########################################
#######################                                                    ######################################*/

/* Definition der Namen der im Programmablauf erzeugten Vars. Namen können frei vergeben werden, sollten aber im
      Quelldatensatz nicht schon vorhanden sein, um Verwirrung zu vermeiden */
local bege "begepi"                                /* Beginndatumsvariable der gesplitteten Episoden */
local ende "endepi"                                /* Enddatumsvariable der gesplitteten Episoden */
local nsp "nspell"                                 /* Anzahl der Spells pro Fall */
local spneu "spell_neu"                            /* Neue Episodenidentifikationsvariable nach dem Spellsplitting*/
local lev1 "level1"                                /* Numerierung der jeweils zeitgleichen Spells pro Episodentyp
                                                         und Fall */
local lev2 "level2"                                /* Numerierung der jeweils zeitgleichen Spells pro Fall */
local nlev1 "nlevel1"                              /* Summe der jeweils zeitgleichen Spells pro Episodentyp und Fall */
local nlev2 "nlevel2"                              /* Summe der jeweils zeitgleichen Spells pro Fall  */

local logdat "splitspells_${S_DATE}.smcl"          /* Name der Logdatei */

/*##########################################################################################################
############################                                           #############################################
############################  c) AUTOMATISCHE VERGABE VON PARAMETERN   #############################################
############################     HIER KEINE ÄNDERUNGEN VORNEHMEN !!!   #############################################
############################                                           ###########################################*/

tempfile  cid temp_usp datum_1 datum_2 datum_wide                      /* Definition temporärer Zwischenfiles */
tempvar nid temp datum x y z first first2 spell2 spid n_epidat gr2 nid1 /* Definition temporärer Variablen */
tempname cmax spanne b a max_t zeit                                     /* Scalare werden mit tempnames benannt */
scalar `cmax' = 0                                                /* diese und folgende Zeilen: Initialisierung der Scalare */
scalar `spanne' =  0
scalar `b' = 0
```

```
scalar `a' = 0
scalar `max_t' = 0
scalar `zeit' = 0

/*#########################################################################################
###########################################################################################
####################################                           ###########################
####################################   Definitionsteil Ende    ###########################
####################################                           ###########################
###########################################################################################
#########################################################################################*/

capture log close
log using "`pfad_o'/`logdat'", append
use "`pfad_d'/`quelle'", clear
local fmt : format `beg'                      /* Ermittlung des Datumsformats der `beg'-Datumsvariablen */
preserve
by `id' : gen `gr2' = _N         /* Zweite Gruppierungsvariable, um die Fälle nach Zahl ihrer Spells zu sortieren */
capture drop `bege'
capture drop `ende'
capture drop `nsp'
capture drop `lev1'
capture drop `lev2'
capture drop `nlev1'
capture drop `nlev2'

/*#########################################################################################
#######################        Zerlegen des Gesamtdatensatzes in Portionen          #########
#########################################################################################*/

local meldung "BEGIN Datensatz zerlegen mit dem Programm splitspells at $S_TIME on $S_DATE "
local meldung1 "Processing.... Please wait..... "
set output proc
display " `meldung' "
display " `meldung1' "
set output error
timer clear
timer on 1

/* Zerlegung des Quelldatenfiles in die vom Nutzer angegebene Anzahl von Portionen. */
if `n'==1 {
      save "`pfad_d'/tempf`n'", replace      /* Wurde für die Portionierung der Wert 1 angegeben, wird der Quell-
                                                datensatz unverändert unter dem Namen tempf1.dta zur Weiterverarbeitung
                                                gespeichert. */
```

```
        local meldung " `pfad_d'/tempf`n' saved at $S_TIME on $S_DATE "
        set output proc
        display " `meldung' "
        set output error
}
else {                                                  /* Wurde für die Portionierung ein Wert > 1 angegeben, dann... */
        egen `nid' = group(`gr2' `id')                  /* ...wird eine neue Fall-ID gebildet, die die Fälle beginnend mit 1
                                                           bis zur maximalen Fallzahl hochzählt, Fälle werden dabei nach
                                                           Spellzahl sortiert*/
        sum `nid'                                       /* Die Gesamtfallzahl wird ermittelt und in `cmax' gespeichert  */
        scalar `cmax' = r(max)
        scalar `spanne' =  int(`cmax'/`n')              /* Die Zahl der Fälle pro Portion wird errechnet */
        sort `nid' `sp'
        save `cid', replace
        local meldung "Die Files Nr. 1 - " `n'-1 " umfassen jeweils " `spanne' " Fälle"
        local meldung1 "File Nr. `n' umfasst " `cmax' - ((`n'-1) * `spanne') "  Fälle"
        set output proc
        display " `meldung' "
        display " `meldung1' "
        set output error

    /* Die folgende Zählschleife generiert die portionierten Datenfiles. Es werden jeweils nur die Fälle aus dem
          Quelldatensatz geladen und separat abgespeichert, deren Fallnummer in den Portionsgrenzen liegt.
          Da nun die höchsten Fallnummern die Fälle mit den meisten Spells sind, soll vermieden werden, dass sie in
          die letzte Portion geraten, die im Grenzfall fast doppelt so groß sein kann wie die anderen Portionen. Des-
          halb wird von "hinten nach vorn" aufgeteilt, so dass die letzte Portion die niedrigsten Fallnummern und damit
          die Fälle mit den wenigsten Spells enthält.
       */
       forvalues i = 1(1)`n' {
              scalar `b' = `cmax' - `spanne' * (`i'-1)      /* b enthält jew. die letzte zur Portion gehörende Fallnummer */
              scalar `a' = `b' - `spanne' + 1               /* a enthält jew. die erste zur Portion gehörende Fallnummer */
              if `i' < `n' {
                     use "`cid'" if `nid' >= `a' & `nid' <= `b', clear
              }
              else if `i' == `n' {                          /* die letzte Portion enthält die restl. Fälle bis zur höchsten Fallnr. */
                     use "`cid'" if `nid' >= 1 & `nid' <= `b', clear
              }
              drop `nid'                                    /* die neue Fallnummer wird nicht mehr gebraucht und wird gelöscht */
              save "`pfad_d'/tempf`i'", replace             /* der Portionsfile wird gespeichert */
              set output proc
              display " `pfad_d'/tempf`i' saved at $S_TIME on $S_DATE "
              set output error
       }
}
```

```
scalar drop `cmax' `spanne' `a' `b'

/*##############################################################################################################
##################     Episodensplitting in einer Schleife über jeden Portionen-File       ######################
##############################################################################################################*/

local meldung "Beginn des Spellsplitting at $S_TIME on $S_DATE "
local meldung1 "Processing.... Please wait..... "
set output proc
display " `meldung' "
display " `meldung1' "
set output error

/* Die Schleife lädt die portionierten Datenfiles und splittet die Episoden. */
forvalues k = 1(1)`n' {

      use "`pfad_d'/tempf`k'" ,clear

      /* Der geladene Datensatz wird nach Fallnummer, Datumsangaben und Spelltyp sortiert. Es wird eine zusätzliche
         Spellzählungsvariable erzeugt, die die Episoden pro Fall hochzählt. */
      sort `id' `beg' `end' `epityp'
      by `id': gen int `spell2' = _n
      save "`temp_usp'", replace

      /* Ziel der nachfolgenden Bearbeitungsschritte ist es, die Beginn- und Enddatumsangaben aller Episoden eines
         Falles so aufzubereiten, dass sie in einer Serie von Datumsvariablen an jede Episode angehängt werden
         können. Damit wird es möglich abzugleichen, ob eine oder mehrere dieser Datumsangaben innerhalb der jewei-
         ligen Episode liegen. Die Anzahl der Datumsangaben, die innerhalb einer Episode liegen, entspricht der
         Anzahl der Splits, in die diese Episode zerlegt werden muss. Die Datumsangaben selbst bilden das Beginndatum
         der gesplitteten Episoden. Ihr Enddatum wird aus dem Beginndatum des nachfolgenden Splits abgeleitet. */

      /* Das Beginndatum wird mit der Fall-ID und der neu generierten Spellzählungsvariablen in einen separaten
         Datensatz gespeichert. */
      keep `id' `beg'
      rename `beg' `datum'
      save "`datum_1'", replace

      /* Das Enddatum wird um den Wert 1 erhöht (damit es als Beginndatum eines potentiellen Splits dienen kann)
         und ebenfalls in einem separaten Datensatz gespeichert. */
      use "`temp_usp'", clear
      keep `id' `end'
      rename `end' `datum'
      replace `datum' = `datum' + 1
      save "`datum_2'", replace
```

```
/* Die beiden zuvor erzeugten Datensätze werden aneinander gefügt und sortiert. */
use "`datum_2'", clear
append using "`datum_1'"
sort `id' `datum'

/* Innerhalb eines Falles mehrfach vorkommende Datumsangaben werden gelöscht. */
duplicates drop `id' `datum', force

/* Ermittlung der Zahl der Datumsangaben, die maximal in einem Fall vorkommen. Diese in max_t gespeicherte
   Information gibt an, wieviele Datumsvariablen im nächsten Schritt angelegt werden. Die maximale Zahl
   der Datumsangaben dient später als Schleifenzähler. */
by `id': gen int `z' = _N
sum `z'
scalar `max_t' = r(max)
drop `z'

/* Die Datumsangaben werden in das "weite" Format übertragen und gespeichert. Dabei wird dem Namen der Datums-
   variablen automatisch ein Index hinzugefügt, der von 1 bis zu der zuvor ermittelten höchsten Anzahl von
   Datumsangaben läuft. */
by `id': gen int `y' = _n
reshape wide `datum', i(`id') j(`y')
save "`datum_wide'", replace

/* Die indizierten Datumsvariablen werden fallweise an die Episodendaten gekoppelt. */
use "`temp_usp'", clear
merge m:1 `id' using "`datum_wide'"
drop _merge

/* In der nachfolgenden Schleife werden alle Datumsangaben gelöscht, die außerhalb der zeitlichen Grenzen
   der jeweiligen Episode liegen. Außerdem wird der Zähler `n_epidat' mit der Anzahl derjenigen Datums-
   angaben gefüllt, die in die Grenzen der Originaldatumsangaben fallen. */
gen int `n_epidat' = 0
local md = `max_t'

forvalues i = 1(1)`md'{
      quietly replace `datum'`i' = . if `datum'`i' < `beg' | `datum'`i' > `end'
      quietly replace `n_epidat' = `n_epidat' + 1 if `datum'`i' != .
}

/* In einer Schleife wird ermittelt, welcher Index die Datumsvariable hat, in der der erste nonmissing Wert
   vorhanden ist. */
gen int `first' = 0
```

```
forvalues i = 1(1)`md'{
      quietly replace `first' = `i' if `datum'`i' != . & `first' == 0
}

/* Die Episoden werden auf Basis der Anzahl gefüllter indizierter Datumsvariablen vervielfacht, sortiert und
   mit einem Episodenzähler versehen. */
expand `n_epidat'
sort `id' `spell2'
by `id' `spell2': gen int `spid' = _n


/* Für jede gesplittete Episode wird ermittelt, welcher Index die Datumsvariable hat, in der ihr Startdatum
   steht. Er errechnet sich aus dem Index der Datumsvariable, in der das Startdatum der ungesplitteten Episode
   steht plus dem Wert des Episodenzählers für den jeweiligen Split. */
gen int `first2' = `first' + `spid' - 1

/* Eine neue Startdatumsvariable für die gesplitteten Episoden wird erzeugt. Über die Schleife wird die korrekte
   Zuordnung der indizierten Datumsangaben zur der Startdatumsvariablen des jeweiligen Splits gesteuert. */
gen int `bege' = .
format `bege' `fmt'
forvalues i = 1(1)`md'{
      quietly replace `bege' = `datum'`i' if `i' == `first2'
}
/* Eine neue Enddatumsvariable wird erzeugt. */
gen int `ende' = .
format `ende' `fmt'

/* Das Enddatum der gesplitteten Episodenteils wird über das Startdatum der zeitlich nachfolgenden Splits
   festgelegt. Ist kein nachfolgendes Split vorhanden, wird das Enddatum der Ursprungsepisode übernommen. */
by `id' `spell2': replace `ende' = `bege'[_n+1] - 1
quietly replace `ende' = `end' if `ende' == .

/* Die überflüssigen Variablen werden gelöscht */
drop `spell2' `n_epidat' `first' `first2'
drop `datum'*

/* Das Ergebnis wird gespeichert */
save "`pfad_d'/tempf`k'", replace

local meldung "gesplitteter Datenfile `pfad_d'/tempf`k' gespeichert at $S_TIME on $S_DATE "
set output proc
display " `meldung' "
set output error
```

```
}
scalar drop `max_t'

/*##########################################################################################################
###################   Die portionierten Files werden zu einem Gesamtdatensatz zusammen gefügt.   ###################
##########################################################################################################*/

if `n' > 1 {              /* nur wenn die Zerlegung in Portionen gewählt wurde */
       local h = 1
       use "`pfad_d'/tempf`h'", clear

       forvalues h = 2(1)`n' {
              append using "`pfad_d'/tempf`h'", nolabel
       }
       local meldung "zerlegte Files zusammengefügt at $S_TIME on $S_DATE.  "
       set output proc
       display " `meldung' "
       set output error
}
local meldung "Processing.... Please wait..... "
set output proc
display " `meldung' "
set output error

save "`pfad_d'/ttemp", replace

/*##########################################################################################################
############################   Konstruktion von Spellzählvariablen                ###############################
##########################################################################################################*/

gsort `id' `bege' `epityp' `srt2'
by `id': gen int `nsp' = _N            /* Anzahl der Spells pro Fall  */
by `id': gen int `spneu' = _n          /* Spellzählung pro Fall (ursprüngliche Spellzählung ist wegen des Splitting
                                          nicht mehr eindeutig) */
drop `spid'

by `id' `bege' `epityp': gen byte `lev1'=_n-1  //
by `id' `bege': gen int `lev2'=_n-1
by `id' `bege' `epityp': gen byte `nlev1'=_N
by `id' `bege': gen int `nlev2'=_N
lab var `spneu' "Spellzählung pro Fall"
lab var `lev1' "Spellzählung pro Fall, Episode u. Quelle"
lab var `lev2' "Spellzählung pro Fall u. Episode"
lab var `nsp' "Anzahl Spells pro Fall"
```

```
lab var `nlev1' "Anzahl Spells pro Fall, Episode u. Quelle"
lab var `nlev2' "Anzahl Spells pro Fall u. Episode"
lab var `nsp' "Anzahl der Spells pro Fall"

save "`pfad_d'/`ziel'", replace

/* Löschen der portionierten Datenfiles. */
forvalues h = 1(1)`n' {
      erase "`pfad_d'/tempf`h'.dta"
  }
capture erase "`pfad_d'/ttemp.dta"
/* Laufzeit des Programms in Stunden, Minuten und Sekunden ausgeben */
timer off 1
quietly timer list
scalar `zeit'=(r(t1))
local hrs = int(`zeit'/3600)
scalar `zeit' = (r(t1)) - (`hrs' * 3600)
local min = int(`zeit'/60)
local sec = int(`zeit' - (`min' * 60))
local meldung "Laufzeit des Programms: `hrs' Std `min' Min `sec' Sek  "
local meldung1 "END at $S_TIME on $S_DATE - der Ergebnisfile heißt `pfad_d'/`ziel' "
set output proc
display " `meldung' "
display " `meldung1' "
set output error
timer clear
capture log close
set output proc

/* HINWEIS: Der Ergebnisfile steht nach Beendigung dieses Programms nicht im Arbeitsspeicher,
   sondern muss eigens aufgerufen werden. */
```

7.2 combispells

Hier abgedruckt: combispells_de_v01.do (Version 01, deutsche Fassung)

```
version 13       /* verwendete Stata-Version */
set output error

/*################################################################################################################
##################################################################################################################
######################                                                                       #######################
######################    Stata-Programm zum Einsatz für gesplittete Episodendaten mit       #######################
######################    entweder vollkommen parallelen oder vollkommen überschneidungs-    #######################
######################    freien Splits.                                                     #######################
######################                                                                       #######################
######################    entwickelt von Klaudia Erhardt und Ralf Künster                    #######################
######################                                                                       #######################
######################    Stand: 30.06.2014                                                  #######################
######################                                                                       #######################
####                                                                                                         ####
####   Das Programm erzeugt aus aus einem gesplitteten Spelldatensatz 2 Variablen: eine String- und eine      ####
####   gelabelte numerische Variable, die anzeigen, welche Spelltypen in den jeweils zeitgleichen Spells eines  ####
####   Falles parallel auftreten. Zusätzlich wird eine Spelltypvariable erzeugt, deren Werte eine Exponential-  ####
####   funktion 2^x darstellen. Außerdem schreibt das Programm label-do-files für die numerische Kombinations-  ####
####   variable und die exponentialcodierte Spelltypvariable heraus.                                         ####
####                                                                                                         ####
####   Das Programm enthält zu Anfang einen Definitionsteil, in dem Pfadangaben sowie Datei- und Variablennamen ####
####   festgelegt werden.                                                                                    ####
####                                                                                                         ####
####   ACHTUNG, WICHTIGER HINWEIS:                                                                           ####
####   - Die Parameter im Definitionsteil a) müssen vom Nutzer zwingend festgelegt werden, da sie die nutzer-  ####
####     spezifischen Umgebungsvariablen und Daten beschreiben.                                              ####
####   - Die Parameter im Definitionsteil b) können vom Nutzer geändert werden, falls gewünscht ist, andere    ####
####     Variablen- bzw. Dateinamen zu verwenden.                                                            ####
####   - Die Parameter im Definitionsteil c) sollten keinesfalls vom Nutzer geändert werden, weil sie in dieser ####
####     Form vom Programm benötigt werden.                                                                  ####
####                                                                                                         ####
####   Weitere Anpassungen des Programmcodes sind nutzerseitig nicht erforderlich.                           ####
####                                                                                                         ####
##################################################################################################################
################################################################################################################*/

set more off
set varabbrev off    /*damit nicht versehentlich Variablen gelöscht werden, die nicht gemeint sind */
```

```
/*###################################################################################################################
#####################################################################################################################
####################################                           ######################################################
####################################  Definitionsteil Anfang   ######################################################
####################################                           ######################################################
#####################################################################################################################
###################################################################################################################*/

/*###################################################################################################################
############################                                           #############################################
############################   a)  MANUELLE VERGABE VON PARAMETERN     #############################################
############################                                           ###########################################*/

/* Definition der nutzerspezifischen Pfade und Dateinamen - ANGABE OHNE "/" AM ENDE, da der Slash im Programm
   automatisch hinzugefügt wird */
local pfad "H:/Zuwanderer/Datenverknüpfung_STATA/combi"
local pfad_d "`pfad'/Datenfiles"       /* Verzeichnis von Quell- und Zieldatei */
local pfad_s "`pfad'/MyFiles"                /* Verzeichnis, in welches die Label-Do-Files geschrieben werden sollen*/
local pfad_o "`pfad'/Output"                 /* Verzeichnis, in das die Logdatei geschrieben werden soll */
                                             /* HINWEIS: pfad, pfad_d, pfad_s und pfad_o können das gleiche Verzeichnis
                                                bezeichnen, aber es müssen alle vier Angaben gemacht werden */
local quelle "siab_r_2008_mod-gsp_V4.dta"    /* Name der Quelldatei */
local ziel "siab_r_2008_mod-gsp-combi_V4.dta"       /* Name der Ergebnisdatei */

/* Definition der Namen der während des Programmablaufs benötigten Variablen. DIESE MÜSSEN IM ZU BEARBEITENDEN
   DATENFILE VORHANDEN SEIN. */
local id "persnr"               /* Fall-ID in der Quell-Datei */
local bege "begepi"             /* Beginn-Datums-Angabe der gesplitteten Spells in der Quell-Datei*/
local epityp "quelle_gr"  /* numerische Variable, die die Spelltypen (Episodentypen) bezeichnet */

/* Ausprägungen der Episodentyp-Variablen und zu vergebende Kurzlabels. Reihenfolge und Anzahl der tokens in
   beiden Listen muss übereinstimmen*/
local typcodes "1 2 3 4"                     /* Zahlencodes der vorhandenen Spelltypvariablen */
local kurzstrings "BeH LeH ASU LHG"          /* Kurzlabels - entweder schon vorhanden oder hier neu festgelegt */

/*###################################################################################################################
#######################                                                    ########################################
#######################  b)  MANUELLE ÄNDERUNGEN DER VORGABEN MÖGLICH,      ########################################
#######################      ABER NICHT UNBEDINGT ERFORDERLICH             ########################################
#######################                                                    ######################################*/

/* Definition der Namen der im Programmablauf neu erzeugten Variablen. Namen können frei vergeben werden, sollten
   im Quelldatensatz aber nicht bereits vorhanden sein, um Verwirrung zu vermeiden. */
local coq "combi"         /* Summe der zeitgleichen Spelltypen (numerische Kombinationsvariable) */
```

```
local clab "combistr"     /* String-Kombinationsvariable, enthält die Kurzlabels der zeitgleichen Spelltypen */
local epixp "sptxp"       /* umcodierte Kopie von `epityp': Werte der Exponentialfunktion 2^x werden zugewiesen. */

local logdat "combisplits_${S_DATE}.smcl"     /*  Name der Logdatei */

/*##################################################################################################################
############################                                          #############################################
############################  C) AUTOMATISCHE VERGABE VON PARAMETERN   #############################################
############################     HIER KEINE ÄNDERUNGEN VORNEHMEN !!!   #############################################
############################                                          ###########################################*/

local n : word count `kurzstrings'
local n1 : word count `typcodes'
local t = char(9)         /* der Tabulator */
local q = char(34)        /* das doppelte Anführungszeichen */
local foot ", replace"    /* Fusszeile für die Label-Do-Files */
tempfile idfile
tempvar qlab temp
tempname zeit
scalar `zeit' = 0

/*##################################################################################################################
####################################################################################################################
####################################                           #####################################################
####################################   Definitionsteil Ende    #####################################################
####################################                           #####################################################
####################################################################################################################
##################################################################################################################*/

capture log close
log using "`pfad_o'/`logdat'", append

/* Prüfung ob die Zahl der Listeneinträge in kurzstrings und typcodes übereinstimmt */
local meldung "Das Programm bricht ab, weil die Zahl der Einträge in kurzstrings und typcodes nicht übereinstimmt."
if `n' != `n1' {
      set output proc
      display " `meldung' "
      capture log close
      exit
}

/*##################################################################################################################
#############################     Beginn des eigentlichen Progamms              ###################################
##################################################################################################################*/
```

```
local meldung "BEGIN Konstruktion der Kombinationsvariablen mit dem Programm combisplits at $S_TIME on $S_DATE "
set output proc
display " `meldung' "
set output error
timer clear
timer on 1          /* Zur Messung der Laufzeit des Programms */
use "`pfad_d'/`quelle'", clear
preserve            /* Der Ausgangsfile wird bewahrt */
keep `id' `bege' `epityp'
duplicates drop `id' `bege' `epityp', force  /* falls es mehrere zeitgleiche Spells des gleichen Typs pro Fall
      gibt, werden sie in diesem Schritt gedroppt. Die Kombinationsvariablen zeigen an, ob ein Spelltyp in einer
      Gruppe zeitgleicher Spells auftritt, jedoch nicht, wie oft.  */

local meldung "Processing.... Please wait..... "
set output proc
display " `meldung' "
set output error

/* Erzeugung einer Kopie der Spelltypvariablen und Zuweisung der Werte der Exponentialfunktion 2^x. Gleichzeitig
   wird ein label-do-File erzeugt, der die Werte der neuen Variablen mit den dazugehörigen Kurzlabels belegt */

/* a) Vorbereitung des Textfiles zur Erzeugung des label-do-Files  */
capture file close d1                          /* zum Abfangen eines Programmabbruchs, falls d1 bereits offen ist */
file open d1 using "`pfad_s'/labels_`epixp'.do", write replace    /* die Textdatei wird erzeugt bzw. geöffnet */
local header "lab def `epixp' `t' /// "

/* b) Erstellung der Kopie der Spelltypvariablen, Zuweisung der Werte an `epixp' und Erstellung der Label-Do-Datei
      - alles in der gleichen Schleife, die über die Zahl der Listeneinträge läuft. */
gen `epixp' = `epityp'
file write d1 "`header'"_n  /* die erste Zeile wird in die label-do-Datei geschrieben */
forvalues i = 1(1)`n' {
      quietly replace `epixp' = 2^(`n'-`i') if `epixp' == real(word("`typcodes'",-`i')) /* -`i' bezeichnet hier den
                                                          `i'-ten Eintrag in der Liste von hinten an gerechnet. */
      local a = 2^(`n'-`i')                               /* -`i' besagt hier, dass `i' von `n' subtrahiert werden soll */
      local b = word("`kurzstrings'",-`i')
      file write d1 `" `t' `a' `t' `q'`b'`q' `t' ///"'_n   /* der gesamte String, der geschrieben werden soll, muss
                        in compound Anführungszeichen (`" bzw. "') eingeschlossen werden, weil innerhalb der Zeichenfolge
                        doppelte Anführungszeichen vorkommen */
}
file write d1 "`foot'"_n
file write d1 _n    /* diese Zeile ist unbedingt erforderlich, sonst ergibt der label-do-file eine Fehlermeldung */
capture file close d1
```

```
/* c) Zuweisung der Labels */
quietly do "`pfad_s'/labels_`epixp'.do"
lab val `epixp' `epixp'

/* d) Erfolgsmeldung */
local meldung " Der Label-Do-File labels_`epixp'.do wurde in `pfad_s' gespeichert. "
set output proc
display " `meldung' "
set output error

/* Konstruktion der numerischen Variablen `coq', welche die Summe der Werte von `epixp' für die jeweils zeit-
      gleichen Episodentypen enthält. Sie wird weiter unten noch gelabelt. */

by `id' `bege': egen `coq'=total(`epixp')
format `coq' %20.0f

/* Konstruktion der String-Variablen `clab', welche die aneinandergereihten Kürzel der jeweils zeitgleichen
   Spelltypen enthält. */

decode `epixp', gen(`qlab')      /* die Stringvariable `qlab' erzeugen und ihr die Kurzlabels von `epixp' zuweisen */
sort `id' `bege' `epityp'        /* Alle zeitgleichen Spells in der Reihenfolge der Spelltypen sortieren */
gen str `clab' = `qlab'
by `id' `bege': replace `clab' = `clab'[_n-1]+"+"+`clab' if _n > 1      /* schrittweises Aufsammeln der Kurzlabels der
                                                                           Episodentypen */

/* im Folgenden werden die Strings aus dem jeweils letzten zeitgleichen Spell (nur diese enthalten alle Kürzel der
   jeweils beteiligten Episodentypen) auf die anderen Spells übertragen. Dazu wird umsortiert, so dass die voll-
   ständigen Strings immer als erstes kommen, um diese dann für die nachfolgenden Spells zu übernehmen */

by `id' `bege': gen `temp' = _n
gsort `id' `bege' -`temp' /* umsortieren */
by `id' `bege': replace `clab' = `clab'[_n-1] if _n>1      /* Übertragen */

local meldung "Processing.... Please wait..... "
set output proc
display " `meldung' "
set output error

/* Labels für die numerische Episodentyp-Kombinations-Variable `coq' erstellen. Die Werte von `coq' und `clab'
   werden paarweise entsprechend den Syntaxanforderungen für Label-Definitionen in eine externe Datei geschrieben.
   Im Unterschied zum vorigen label-do-File müssen die vorkommenden Werte record für record aus dem aktiven Daten-
   file ermittelt werden. */

capture file close d1    /* zum Abfangen eines Programmabbruchs, falls d1 bereits offen ist */
```

```
file open d1 using "`pfad_s'/labels_`coq'.do", write replace

/* die macros t, q und foot sind bereits definiert und bleiben unverändert. Das Macro header wird neu belegt */
local header "lab def `coq' `t' /// "
sort `coq'          /* Umsortierung: aufsteigend nach den Werten der numerischen Combi-Variablen. */
file write d1 "`header'"_n

local k=_N
forvalues i = 1(1)`k' {
      if `coq'[`i'] !=  `coq'[`i'-1] {       /* nur wenn ein neuer Wert auftritt, wird herausgeschrieben */
            local a = `coq'[`i']
            local b = `clab'[`i']
            file write d1 `" `t' `a' `t' `q'`b'`q' `t' ///"'_n   /* der gesamte String, der geschrieben werden soll,
                                                muss in compound-Anführungszeichen (`" bzw. "') eingeschlossen werden, weil
                                                innerhalb der Zeichenfolge doppelte Anführungszeichen vorkommen */
      }
}
file write d1 "`foot'"_n
file write d1 _n    /* diese Zeile ist unbedingt erforderlich, sonst ergibt der label-do-file eine Fehlermeldung */
capture file close d1

/* Zuweisung der Labels */
quietly do "`pfad_s'/labels_`coq'.do"
lab val `coq' `coq'
sort `id' `bege' `epityp'

/* Erfolgsmeldung */
local meldung "Die Variablen `epixp', `coq' und `clab' wurden erzeugt und die Label-Do-Files " ///
                     "labels_`coq'.do und labels_`epixp'.do wurden in `pfad_s' gespeichert."
set output proc
display " `meldung' "
set output error

/* Anfügen der neuen Variablen und Abspeichern des Ergebnisfiles */
save "`idfile'"
use "`pfad_d'/`quelle'", clear
capture drop `coq'  /* vorsorgliches Löschen von evtl. im File vorhandenen gleichnamigen Variablen */
capture drop `clab'
capture drop `epixp'

sort `id' `bege' `epityp'
merge m:1 `id' `bege' `epityp' using "`idfile'", keepusing (`epixp' `coq' `clab')
drop _merge
save "`pfad_d'/`ziel'", replace
```

```
/* Laufzeit des Programms in Stunden, Minuten und Sekunden ausgeben */
timer off 1
quietly timer list
scalar `zeit'=(r(t1))
local hrs = int(`zeit'/3600)
scalar `zeit' = (r(t1)) - (`hrs' * 3600)
local min = int(`zeit'/60)
local sec = int(`zeit' - (`min' * 60))
local meldung "Laufzeit des Programms: `hrs' Std `min' Min `sec' Sek  "
local meldung1 "END at $S_TIME on $S_DATE - der Ergebnisfile heißt `pfad_d'/`ziel' "
set output proc
display " `meldung' "
display " `meldung1' "
set output error
timer clear
capture log close
set output proc

/* HINWEIS: Der Ergebnisfile steht nach Beendigung dieses Programms nicht im Arbeitsspeicher,
   sondern muss eigens aufgerufen werden. */
```

Literatur

Blossfeld, Hans-Peter/Golsch, Katrin/Rohwer, Götz (Hg.) (2007): Event History Analysis with Stata, Mahwah NJ, London: Lawrence Earlbaum Associates

Brzinsky-Fay, C./Kohler, U./Luniak, M. (2006): Sequence analysis with Stata. The Stata Journal, 6(4), 435–460.

Drews, Nils/Groll, Dominik/Jacobebbinghaus, Peter (2007): Programmierbeispiele zur Aufbereitung von FDZ Personendaten in STATA. (FDZ-Methodenreport, 06/2007 (de)), http:// doku.iab.de/fdz/reporte/2007/MR_06-07.pdf

Eberle, Johanna/Schmucker, Alexandra/Seth, Stefan (2013): Programmierbeispiele zur Datenaufbereitung der Stichprobe der Integrierten Arbeitsmarktbiografien (SIAB) in Stata. Generierung von Querschnittdaten und biografischen Variablen (FDZ-Methodenreport 04/2013 (de)), http://doku.iab.de/fdz/reporte/2013/MR_04-13.pdf

Erhardt, Klaudia/Künster, Ralf (2014): Splitting Spells Using Very Large or Daily Data Sets. The stata syntax files splitspells.do and combispells.do. Präsentation auf dem 2013 German Stata Users Group meeting, Hamburg.
http://www.stata.com/meeting/germany14/abstracts/materials/de14_erhardt_kuenster.pdf (Zugriff: 16.07.14)

Hillmert, Steffen/Künster, Ralf/Spengemann, Petra/Mayer, Karl Ulrich (2004): Projekt "Ausbildungs- und Berufsverläufe der Geburtskohorten 1964 und 1971 in Westdeutschland": Dokumentation, Teil VIII Programmdokumentation. Materialien aus der Bildungsforschung Nr.78, Max-Planck-Institut für Bildungsforschung Berlin 2004 (https://www.mpib-berlin.mpg.de/sites/default/files/schriften/Materialien/Materialien_078/pdf/Materialien_Bildungsforschung_MPIB_078_IVff.pdf). (Zugriff: 05.05.2014)

Kröger, Hannes (2013): Newspell: Easy management of complex spell data. Präsentation auf dem 2013 German Stata Users Group meeting, Potsdam.
http://www.stata.com/meeting/germany13/abstracts/materials/de13_kroeger.pdf (Zugriff: 06.05.14)

vom Berge, Philipp/König, Marion /Seth, Stefan (2013): Stichprobe der Integrierten Arbeitsmarktbiografien (SIAB) 1975 - 2010, (FDZ-Datenreport 01/2013), http://doku.iab.de/fdz/reporte/2013/DR_01-13.pdf

vom Berge, Philipp/Burghardt, Anja/Trenkle, Simon (2013): Stichprobe der Integrierten Arbeitsmarktbiografien. Regionalfile 1975-2010 (SIAB-R 7510), (FDZ-Datenreport 09/2013), http://doku.iab.de/fdz/reporte/2013/DR_09-13.pdf

Impressum

FDZ-Methodenreport 7/2014

Herausgeber
Forschungsdatenzentrum (FDZ)
der Bundesagentur für Arbeit
im Institut für Arbeitsmarkt- und Berufsforschung
Regensburger Str. 104
90478 Nürnberg

Redaktion
Stefan Bender, Heiner Frank

Technische Herstellung
Heiner Frank

Rechte
Nachdruck - auch auszugsweise - nur mit
Genehmigung des FDZ gestattet

Bezugsmöglichkeit
http://doku.iab.de/fdz/reporte/2014/MR_07-14.pdf

Internet
http://fdz.iab.de/

Rückfragen zum Inhalt an:

Klaudia Erhardt, DIW
Mohrenstr. 58
10117 Berlin
mailto:kerhardt@diw.de

Ralf Künster, WZB
Reichpietschufer 50
10785 Berlin
mailto:ralf.kuenster@wzb.eu