1 NATÜRLICHSPRACHIGE MORPHOLOGIE MIT (F)LEX

1.0 Allgemeines zu (f)lex

1.0.1 Standardanwendung von (f)lex

Compiler für Programmiersprachen enthalten üblicherweise ein Modul für lexikalische Analyse ("Scanner"), das überflüssige Leerzeichen entfernt, die Grundeinheiten der Sprache ("Token") erkennt, und entsprechende Informationen an ein Syntax-Modul weitergibt. Z. B. wird ein Scanner für C-Programme aus dem Quelltext in (1) die Information extrahieren, daß das Schlüsselwort "int", gefolgt vom Bezeichner "count" und den Symbolen "=" und ";" vorliegt, und dies dem eigentlichen Compiler mitteilen, der dann feststellen kann, ob dieser Ausdruck der C-Syntax entspricht.

(1) int count = 1;

(f)lex ist ein Programm-Generator, der es erlaubt, Scanner in einer übersichtlichen und leicht handhabbaren Repräsentationssprache zu spezifizieren und **(f)lex-Programme** in C- Code für lauffähige und effiziente Scanner auf der Basis von *Finite-State-Automaten* übersetzt.

1.0.2 Geschichte und Versionen

lex wurde in den 70er Jahren zur Unterstützung von **yacc**, einem Werkzeug zur automatischen Generierung von Parsern entwickelt. Seitdem wurden mehrere neuere Versionen herausgebracht, auf deren Unterschiede ich hier nicht eingehen werde[1]. **lex** wird bei Unix(-Derivaten) und Linux serienmäßig mitgeliefert und ist auch für MS-DOS verfügbar. **flex** ("fast **lex**") ist eine frei verfügbare **lex**-Version mit vielen Vorteilen gegenüber Standard-**lex**: Mit **flex** generierte Scanner sind im allgemeinen schneller, zuverlässiger[2] und tolerieren (im Gegensatz zu **lex**) Kommentare an (fast) beliebiger Stelle im Quelltext. Die Implementation, die in **§2/5** vorgestellt wird, ist aus diesen Gründen in **flex**. Auch von **flex** gibt es verschiedene Versionen mit geringfügig voneinander abweichenden Eigenschaften, aber die Grundsyntax, die hier vorgestellt wird, ist für alle **lex**- und **flex**-Versionen identisch. Um anzudeuten, daß ich mich auf **lex** *und* **flex** beziehe, schreibe ich "**(f)lex**".

1.0.3 Literatur

Levine et al. (1992) und Herold (1995) eignen sich gleichermaßen als Einstieg für den praktischen Umgang mit **f(lex)**. Herold ist etwas ausführlicher in den Erklärungen, Levine et al. in einigen Punkten informativer. Aho et al. (1988, Kap. 3) besprechen die hinter **(f)lex** stehenden Algorithmen. Aho et al. (1972) gehen tiefer auf deren mathematische Grundlagen ein. Holub (1990:Kap.2) beschreibt im Detail (einschließlich Quelltext) seine C-Implementation einer **lex**-Version ("**LEX**").

[1] zu verschiedenen **(f)lex** -Versionen siehe die Appendices von Levine et al. (1992:252) Die hier verwendete Version (2.5.2) ist dort aber nicht beschrieben. Ich verweise auf die entsprechenden Manual-Seiten.
[2] Z.B. hat **lex** Schwierigkeiten, *reguläre Definitionen* zu erkennen, die nicht am Zeilen-Anfang stehen.

1.1 Syntax und Funktionsweise von (f)lex

1.1.1 Syntax

(f)lex-Programme bestehen aus drei Teilen, die jeweils durch eine Zeile, die ausschließlich "%%" enthält[3], getrennt werden:

(2)
```
Definitions-Teil
%%
(f)lex-Regeln
%%
C-Code
```

Die zweite "%%" - Zeile kann weggelassen werden, wenn der dritte Programm-Teil leer ist. Obligatorisch ist lediglich der erste Begrenzer. Der Scanner zum **(f)lex**-Programm "%%" liefert für jede Eingabe die dazu identische Ausgabe. **(f)lex**-Regeln bestehen aus einem *Regulären Ausdruck* (*RA*)[4] am Zeilenanfang und - durch Leerzeichen davon getrennt - einer Folge von C-Anweisungen, die ausgeführt werden, falls ein String in der Eingabe den *RA* matcht. Der Scanner für das folgende **(f)lex**-Programm liefert für einige albanische Wörter deren syntaktische Kategorie ("printf" ist der C-Druck-Befehl):

(3)
```
%%
(hap|pi)          printf("Verb");
(madh|vogël)      printf("Adjektiv");
(gisht|derë)      printf("Nomen");[5]
```

Reguläre Ausdrücke lassen sich auf bestimmte Kontexte beschränken. Rechte Kontexte können dabei direkt durch den Kontextoperator "/" und *RA*s ausgedrückt werden, linke Kontexte mit Hilfe von Anfangszuständen. Anfangszustände müssen im Definitionsteil des **(f)lex**-Programms nach "%start" aufgelistet und in Regeln am Zeilenanfang in spitze Klammern ("<, >")gesetzt werden:

(4)
```
%start N V
%%
madh                {printf("Adj ");}
hap                 {printf("Verb "); BEGIN V;}      /* Anfangszustand <= V */
gisht               {printf("Nom "); BEGIN N;}       /* Anfangszustand <= N */
<N>i                {printf("Art "); BEGIN 0;}       /* falls Anfangszustand = N */
<V>i                {printf("aor 3sg "); BEGIN 0;}   /* falls Anfangszustand = V */
i/" "(madh|vogël)   {printf("Gelenkartikel ");}  /* falls madh oder vogël folgt */
```

i kommt vor Adjektiven als eine Form des sogenannten Gelenkartikels vor (***i** madh*, "groß")[6], in Verben als Endung für den **aor 3sg** (*hap-**i***, "er öffne-**te**") und in Nomen als Artikel-Endung (*gisht-**i***, "**Der** Finger"). Nehmen wir an, der Scanner analysiert die Sequenz "*hapi gishti i madh*". Beim Einlesen von *hap* (Regel **2**) geht er durch die **BEGIN**-Anweisung in Zustand **V** und ermöglicht dadurch die Anwendung von Regel **5** auf das darauffolgende *i*, nicht aber von Regel **4**. Regel **2** matcht *gisht*, da sie weder Startbedingungen noch rechte Kontexte besitzt und startet Zustand **N**. Jetzt ist für das nächste *i* nur die Anwendung von Regel **3** möglich. Regel **3** und **4** gehen jeweils in den Defaultzustand **0** über, um das Einlesen zuvieler *i*s durch eine Regel zu verhindern (z.B. würde Regel **3** ohne diese Maßnahme auch auf das zweite *i* in *gishti i* passen). Das nächste *i* steht unmittelbar vor *madh* und wird durch Regel **5** gelesen, *madh* durch Regel **1**. Wir erhalten also die Ausgabe: "Verb aor 3sg Nom Art Gelenkartikel Adj"

---

[3] "%%" muß am Zeilenanfang stehen.

[4] Die Syntax von **(f)lex**-*RA*s ist in Anhang **C** beschrieben.

[5] ASCII (und damit C und damit **(f)lex**) enthält kein "ë". Ich übergehe dies und ähnliche Details.

[6] Für eine ausführlichere Behandlung des Gelenkartikels s. Trommer (in Vorb.).

Das folgende Programm soll kurz die Verwendung des C-Routinen-Teils zeigen: Durch "int i = 0;" wird eine globale Integer-Variable deklariert, die bei jedem Einlesen von *dhe* (albanisch: "und") um 1 hochgesetzt wird. Der entsprechende Scanner zählt die in einem Text vorkommenden "*dhe*s", deren Anzahl beim Einlesen von "ENDE" ausgegeben wird. Dies erledigt die C-Funktion "schluss()", die im dritten Programm-Teil definiert ist, "exit(0);" bricht das Programm ab.

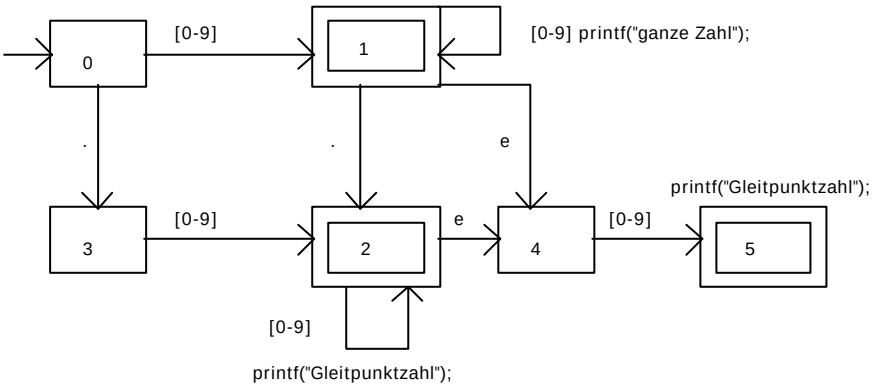
(5)
```
        int     i = 0;
%%
dhe     ++i;
ENDE    schluss();
%%

schluss()
{
printf("Der Text enthält %d mal dhe \n", i);
exit(0);
}
```

1.1.2 *Reguläre Ausdrücke, FSA*s und *DSA*s

Die Konstruktion von **(f)lex**-Scannern beruht auf Algorithmen zur Übersetzung von *RA*s in äquivalente *Deterministische Finite-State-Automaten* (*DFA*s), die optimiert und durch komprimierte Transitions-Tabellen dargestellt werden (Aho et al. 1988, Holub 1990). Das Ergebnis demonstriere ich kurz an einem Beispiel aus Holub (1990:60 ff). Das **(f)lex**-Programm in (6) beschreibt ganze Zahlen und Gleitpunktzahlen, z.B. *1.2*, *1. 1.2e3*, ***2e3***, *1*(*1* ist eine ganze Zahl, die anderen sind Gleitpunktzahlen), (7) zeigt einen *DFA*, wie ihn der daraus gebildete Scanner benützt:

(6)
```
%%
[0-9]+                                  printf("ganze Zahl ");
([0-9]+|[0-9]*\.[0-9]+|[0-9]*)(e[0-9]*)?    printf("Gleitpunkt-Zahl ");
```

(7)



Wie zu sehen ist, erzeugt **(f)lex** aus (möglicherweise) vielen einzelnen Regeln genau einen Automaten[7]. Dies ist möglich, da die Vereinigung von *Regulären Mengen* wieder eine *Reguläre Menge* ergibt. Nur die Verknüpfung von C-Aktionen mit Endzuständen spiegelt noch die Regel-Struktur im ursprünglichen **(f)lex**-Programm wider. Man beachte, daß Zustand **1** nur mit "printf("ganze Zahl ");" annotiert ist, nicht mit "printf("Gleitpunktzahl ");", obwohl jede Zahl die Regel **1** matcht, auch Regel **2** matcht. Dies ist eine der Strategien, die **(f)lex** benützt, um für jede Eingabe deterministisch *eine* Ausgabe zu liefern. **f(lex)** folgt dabei den Maximen, daß im Zweifelsfall längere Strings anstatt kürzerer eingelesen werden und (bei gleich

[7] Im Gegensatz zu den meisten Implementationen von Two-Level-Morphology.

langen Mustern) die Anweisung *der* Regel ausgeführt wird, die weiter vorn im Programm steht, (im eben beschriebenen Fall Regel **1**).

1.1.3 Eigenschaften des (f)lex-Treibers

Die Bevorzugung längerer Pattern spiegelt sich nicht unmittelbar in **(f)lex**-*DFA*s (d.h. den Übergangs-Tafeln) wider. Z.B. könnte der Scanner in (7) bei der Eingabe von *12* "ganze Zahl ganze Zahl" oder einfach nur "ganze Zahl" zurückgeben, da sowohl *1* und *2* als auch *12* zur Sprache des Automaten gehören. Faktisch enthält **(f)lex** jedoch einen Automaten-Treiber, der - prozedural - auch für diesen Fall Determinismus in der Ausgabe garantiert und den längsten matchenden String sucht, bevor er eine Aktion startet. Im Fall von *12* merkt sich der Treiber nach dem Lesen von *1* den erreic hten Endzustand (Zustand **1**), versucht aber gleichzeitig, den eingelesenen String weiter zu "verlängern", geht mit *2* wieder nach Zustand **1** über, und führt, da das Eingabeende erreicht ist, die Aktion aus, die zum aktuellen Endzustand gehört. Dieser "greedy algorithm"(Holub 1990:60-62) hat einige Nachteile: Er kostet Zeit, da der Scanner nach weiteren Endzuständen sucht, die u.U. nicht existieren, und er erschwert eine deklarative Interpretation von **f(lex)**, da das Verhalten von Scannern sich nicht unmittelbar aus den Übergangstabellen ergibt. Dennoch ist es für viele Anwendungen einschließlich des Beispiels in (6) sinnvoll, so zu verfahren. Als Analyse für *12* ist normalerweise "zwölf" das gewünschte Resultat, nicht "eins zwei". Ein Treiber, der *alle* Analysen für *12* in (6) liefert, müßte im übrigen "eins zwei" *und* "zwölf " liefern und würde sicher mehr Backtracking-Zeit kosten als der tatsächliche **(f)lex**-Treiber.

1.1.4 Ein weiterer Desambiguierungsmechanismus

wird in der gängigen Literatur zu **(f)lex** wohl deshalb übergangen, weil er sich unmittelbar aus der Konstruktion von **(f)lex**-Scannern ergibt. Im **(f)lex**-Programm in (8) steht Regel **1** vor Regel **2** und die linken Seiten beider Regeln sind gleichlang. Trotzdem führt die Eingabe *123* zur Ausgabe "Zwölf Rest" und nicht zu "Rest Dreiundzwanzig", da **(f)lex**-Scanner Regeln konsequent von links nach rechts "anwenden". Dieses Vorgehen bezeichne ich im Folgenden als "Links-Rechts-Strategie".

(8)
```
%%
23      printf("Dreiundzwanzig");
12      printf("Zwölf");
[0-9]   printf("Rest");
```

Wie das Beispiel ebenfalls zeigt, hat die Links-Rechts-Strategie Präzedenz vor der Bevorzugung weiter vorne stehender Regeln. Sie überschreibt auch die **(f)lex**-Taktik, nach Möglichkeit längere Strings einzulesen. Ersetzen wir etwa Regel **1** durch "1 printf("Eins");", erhalten wir für *123* "Eins Rest Rest". Insgesamt ergibt sich für die Auflösung bei Konflikten innerhalb und zwischen Regeln die folgende Hierarchie: (1) die Links-Rechts-Strategie, (2) die Bevorzugung längerer Strings, (3) die Regel-Präzedenz-Strategie. **(f)lex** verfolgt bei der Lösung solcher Konflikte jeweils die erste dieser Strategien, die zu einem eindeutigen Ergebnis führt. Diese Hierarchie ist logisch keineswegs notwendig, und ich schlage in **§1/2** im Rahmen von **mo_lex** Alternativen dazu vor.

1.1.5 Anfangszustände

Die **(f)lex**-Literatur geht nur am Rand auf die Implementation von Anfangszuständen und Lookaheads in **(f)lex** ein (Aho et al. 1990:161-162, Aho et al. 1972:259-260)[8]. Die Realisierung von Anfangszuständen in einem Transducer ist relativ einfach, indem man den Transducer, der der Regel mit Anfangszustand entspricht, an den Transducer "hängt", in dem der Anfangszustand eingeführt wird[9]: (10) zeigt einen leicht modifizierten Teil von (4) (s. (9)) als Transducer :

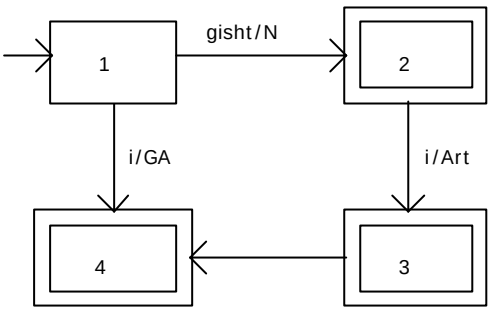
---

[8] In Holubs (1990) **LEX** gibt es keinen Lookahead-Operator.

[9] Dies läßt sich natürlich für Anfangszustände, die von mehreren Regeln eingeführt werden, oder mehreren Regeln zugrundeliegen, generalisieren.

(9)
```
%%
gisht          {printf("Nom "); BEGIN N;}        /*Anfangszustand <= N*/
<N>i           {printf("Art "); BEGIN 0;}        /*falls Anfangszustand = N */
i               printf("GA ");                   /*"Gelenkartikel" */
```

(10)



Anders als in (7) muß das **(f)lex**-Treiber-Programm hier beim Einlesen eines Strings die Aktionen für mehrere Endzustände ausführen. Z.B. sollte es für *gishti* beim Erreichen von Zustand **1** die dort spezifizierte Aktion starten, obwohl es unmittelbar darauf einen weiteren Endzustand erreicht (Zustand **2**).

1.1.6 Lookaheads

Endliche rechte Kontexte sind als Automaten mit Linksbewegungen[10] interpretierbar, für Lookaheads mit "*" oder "+" ist jedoch fraglich, ob dies möglich ist. Praktisch können wir davon ausgehen (Aho et al. 1990:161), daß **f(lex)** für ein Pattern der Form X/RK einen Automaten generiert, der XRK (die Verkettung von X und RK) erkennt und sich an der Stelle zwischen X und RK die Stelle im Eingabepuffer merkt (etwa als Zeiger). Nach dem Erreichen des Endzustands für XRK schiebt der **(f)lex**-Scanner dann RK zurück in den Eingabe-Puffer und beginnt unmittelbar nach dem letzten X wieder mit dem Lesen.

1.2 Natürlichsprachige Morphologie mit (f)lex

1.2.1 Morphologie, Transducer und Direktionalität

Obwohl **(f)lex** zur Analyse von Strings konzipiert ist, können wir es ebensogut zur Generierung verwenden, wenn wir Input in geeignetem String-Format liefern. Auf diese Weise lassen sich viele für Flexionsmorphologie typische Phänomene elegant beschreiben.
Wenn wir mit **(f)lex** ausschließlich *Reguläre Sprachen* beschreiben[11] (d.h. *Finite-State-Maschinen* mit einem Band) und die Aktionen, die mit Regeln assoziiert sind, auf Schreibbefehle beschränken, ergibt sich die Repräsentation einer *Finite-State-Maschine* mit zwei Bändern, also eines *Finite-State-Transducers* (*FST*, Aho et al. 1972:223)[12]. Da die Wertemengen von *FST*s wiederum *Reguläre Sprachen* sind, reduziert sich das Problem, zu einem **(f)lex**-Generator einen Parser zu konstruieren, seinerseits auf einen Erkenner

[10] Solche Automaten sind äquivalent zu *FSA*s ohne Linksbewegungen.
[11] s. **1.3.4**.
[12] Möglich ist z.B. folgende Konstruktion: Ersetze jeden Übergang des Automaten mit Label X durch X/ε. Ersetze jeden Endzustand E, der mit printf("B_1... B_n"); assoziiert ist, durch n Zustände (E | B_1) ... (E | B_{n-1}). Füge für jedes Paar (E | B_i) (E |B_j) für i, j ∈ {1, ..., n} und j = i+1 den Übergang ε/B_i von (E | B_i) nach (E |B_j) hinzu. Ersetze alle Übergänge von beliebigen Zuständen Z nach E durch Übergänge von Z nach E_1 und alle Übergänge von E nach beliebigen Zuständen Z durch entsprechende Übergänge von E_n nach Z.

für *Reguläre Sprachen* mit Ausgabe. D.h. obwohl **(f)lex** als Generator für Wortformen per se keinen Wort-Parser liefert, ist damit garantiert, daß sich ein Parser finden läßt, der formal genauso restriktiv ist.[13]

1.2.2 Vollständige Synkretismen

sig-prs-Formen im Albanischen tragen einheitlich das Null-Suffix. Statt der ausführlichen Darstellung in (11) erlaubt es **(f)lex**, die drei Regeln **2**-**4** wie in (11´) zusammenzufassen[14]:

```
(11)    %%
        V1          printf("hap");
        1sg         ;           /* ((V1)1sg)    -->    hap */
        2sg         ;           /* ((V1)2sg)    -->    hap */
        3sg         ;           /* ((V1)3sg)    -->    hap */

(11´)   [1-3]sg     ;
```

1.2.3 Teilweise Synkretismen:

Alle albanischen **1pl**-Endungen enthalten *-m*, die konkrete Gestalt des **1pl**-Morphems ist jedoch in vielen Fällen leicht voneinander verschieden (12). Dies können wir z.T. durch Regeln über die Ausbuchstabierung von Tense-Morphemen wie **aor** und **opt** darstellen (z.B. **opt** in (13)). Im Programm in (13) taucht dabei eine weitere Abkürzungskonvention auf: Regeln, die mit "|" statt mit C-Aktionen enden, übernehmen die Aktion der folgenden Regel (Zeilen **3**, **4**). Es scheint sinnvoll, auch *-i-* vor *-m* durch eine eigene Regel einzuführen, ebenso wie *-sh-*, um die jeweilige Verteilung dieser Formative (*i* bei **prs-akt**, **imf-akt/nak**, **opt** , *sh* bei **imf-nak**, **opt**) sichtbar zu machen. Dafür fehlen jedoch motivierbare Morpheme, z.B. kann im **opt** entweder *-sh-* oder *-i-* entsprechen, aber nicht beiden, während **S1** und **1pl** offensichtlich zu *hap-* und *-m* gehören. Insertion ist in **(f)lex** nicht darstellbar. Ich löse das Problem, indem ich eine Klammer im Kontext **1pl** nach *-i-* übersetze. Dieser Schritt erscheint auf den ersten Blick etwas ad hoc, spiegelt aber die Tatsache wider, daß sowohl die Klammer als auch *-i-* einen eher peripheren Status auf ihren jeweiligen Repräsentationsebenen haben. Dadurch daß die Klammerung ihrerseits syntaktisch (bzw. morphotaktisch) motiviert ist, ist eine Proliferation von willkürlich eingeführten "Dummy"-Klammern ausgeschlossen.

(12)

prs akt	**prs nak**	**imf akt**	**imf nak**	**aor**	**opt**
((S1)1pl)	(((S1)nak)1pl)	(((S1)imf)1pl)	((((S1)nak)imf)1pl)	(((S1)aor)1pl)	(((S1)opt)1pl)
*hap-i**m***	*hap-e-**m**i*	*hap-n-i**m***	*hap-e-sh-i**m***	*hap-ë**m***	*hap-sh-i**m***

[13] Vertauscht man in einem **(f)lex**-Programm, das nur Schreibbefehle enthält, die rechten und linken Seiten der Regeln, erhält man nicht automatisch den inversen Transducer. Ein Grund dafür ist, daß die Erkennungsmuster von **(f)lex**-Regeln *Reguläre Ausdrücke* sind, Schreibbefehle hingegen Strings. Bei einer Umkehrung von Regeln ergibt sich u.U. kein sinnvoller Schreibbefehl: Z.B. würde die Umkehrung einer typischen Regel aus Anhang **B** ((1a)) (1b) ergeben, keinesfalls das intendierte Ergebnis, da *Reguläre Ausdrücke*, als Strings interpretiert, opak sind:

(1) a. K2[0-46-8] printf("t"); b. t printf("K2[0-46-8]");

[14] ";" ist der "leere" Befehl, d.h. er bewirkt *nichts*, und damit in diesem Fall, daß dem eingelesenen Strings keine Ausgabe entspricht.

(13)
```
%start IM MI ËM
%%
S1              {printf("hap");   BEGIN IM;}
nak             {printf("e");     BEGIN MI;}
opt             |
<MI>imf         {printf("sh");    BEGIN IM;}
imf             printf("n");
aor             {printf("ë");     BEGIN ËM;}

<IM>[)]/1pl     printf("i");
1pl             printf("m");
[() ]           ;
```

Die Behandlung von *-mi* ist in diesem Kontext etwas problematisch. Ich gehe darauf in **1.3.1** näher ein.

1.2.4 Default-Werte

Albanische **kon**-Formen werden durch Voranstellung des Partikels *të* vor die entsprechende Indikativ-Form gebildet. Weitere Allomorphie taucht nur im **prs** in der **2/3sig** auf, deren Endungen sich in **ind** und **kon** unterscheiden:

(14)

prs 1sg	**prs 2sg**	**prs 3sg**	**prs 1pl**	**prs 2pl**	**prs 3pl**
*hap-**0***	*hap-**0***	*hap-**0***	*hap-im*	*hap-ni*	*hap-in*
të** hap-**0	***të** hap-**ësh***	*të hap-**ë***	***të** hap-im*	***të** hap-ni*	***të** hap-in*

Wir können die **ind**-Endungen als Default-Werte auffassen, die nur dann durch andere Endungen ersetzt werden, wenn spezifischere Regeln dies erfordern. Wieder stellt **(f)lex** einen geeigneten Repräsentationsmechanismus zur Verfügung:

(15)
```
%start KON
%%
S1              printf("hap");
kon             {printf("të"); BEGIN KON;}

<KON>2sg        printf("ësh");
<KON>3sg        printf("ë");

[1-3]sg         ;
1pl             printf("im");
2pl             printf("ni");
3pl             printf("in");
```

Dadurch, daß **(f)lex** die erste passende Regel anwendet, wird die Endung in "kon V1 3sg" zu *ë*, da der Scanner sich beim Lesen von *3sg* im zustand **KON** befindet und Regel **4** die erste ist, die auf *3sg* paßt. Bei "kon V1 1pl" hingegen ist das Muster in Regel **6** das erste passende und führt zur Endung *im* wie bei V1 1pl.

1.2.5 Innere Flexion

können wir mit Hilfe einer reicheren Ausgangsstruktur beschreiben. Statt atomarer Stämme kodieren wir veränderliche Stamm-Vokale und -Konsonanten innerhalb des Strings, der den Stamm repräsentiert, z.B. den Stamm von *marr* als (S1+V1+K1), den von *dal* als (S2+V2+K2):

(16)

prs 1sg	prs 2sg	prs 3sg	prs 1pl	prs 2pl	prs 3pl
*d-**a**-l-0*	*d-**e**-l-0*	*d-**e**-l-0*	*d-**a**-l-im*	*d-**i**-l-ni*	*d-**a**-l-in*
*m-**a**-rr-0*	*m-**e**-rr-0*	*m-**e**-rr-0*	*m-**a**-rr-im*	*m-**e**-rr-ni*	*m-**a**-rr-in*

(17)
```
%%
S1                              printf("m");
S2                              printf("d");
K1                              printf("rr");
K2                              printf("l");

V2/"+"K1"+"2pl                  printf("i");
V[12]/"+"K[12][)]([23]sg|2pl)   printf("e");
V[12]                           printf("a");
```

Wieder garantieren *Reguläre Ausdrücke* und **(f)lex**-Defaults eine kompakte Darstellung.

1.2.6 Morpho-Phonologie

Zwischen vokalisch auslautenden Stämmen und bestimmten Endungen oder Klitika treten Hiatus-Tilger:

(18)

aor 1sg	prs nak 2sg	imv 2sg+a3s	imv 2sg+d3p	imv 2sg+klnak
hap-a	*hap-e-sh*	*hap-e*	*hap-u*	*hap-u*
*pi-**v**-a*	*pi-**h**-e-sh*	*pi-**j**-e*	*pi-**j**-u*	*pi-**h**-u*

Die Insertion dieser Phoneme ist eindeutig phonologisch motiviert. Trotzdem kann sie nicht durch produktive phonologische Prozesse zustandekommen. Den putativen Formen * *pia, *piem, *pie, *piu* entsprechen korrekte Formen anderer Wörter *mia,* "meine" (**fem plu**)*, biem,* "wir fielen" (**aor**)*, bie,* "er fällt"*, miu,* "die Maus"). Den Status dieser Phoneme zwischen Morphologie und Phonologie bringe ich durch eine weitere ebenfalls in **(f)lex** implementierte Transducer-Ebene zum Ausdruck, die "sprechende" Diakritika des Allomorphie-Moduls phonologisch interpretiert:

(19) **Allomorphie**

```
%start AOR NAK
%%
V1              {printf("hap");}
V2              {printf("pi");}
aor             {printf("(v)");    BEGIN AOR;}
nak             {printf("(h)e");   BEGIN NAK;}
<AOR>2sg        {printf("e");}
<NAK>2sg        {printf("sh");}
```

(20) **Morphophonologie**

```
Vokal    [aeiouyë]
%%
{Vokal}"([vhj])"        printf("%c%c", yytext[0], yytext,[2]);
"([vhj])"               printf("%c",yytext[1]);
```
[15]

[15] SPE-Phonologie (Chomsky & Halle 1968) ohne Features läßt sich in **(f)lex** weitgehend implementieren indem man nach Bedarf Transducer-Ebenen hintereinanderschaltet.

Die erste Zeile des Morphophonologie-Programms zeigt ein weiteres **(f)lex**-Konstrukt, eine *reguläre Definition*. Überall im Programm wird "{Vokal}" durch den *Regulären Ausdruck* "[aeiouyë]" ersetzt. Die Variable yytext[n] beinhaltet jeweils den n-ten Buchstaben des Input-Strings. Insgesamt faßt Regel **1** also 24 Regeln zusammen, z.B. die folgenden:

```
(21)    a"([v])"        printf("av");
        e"([v])"        printf("ev");
        ë"([v])"        printf("ëv");
        a"([h])"        printf("ah");
        a"([j])"        printf("aj");
```

Der Transducer übersetzt *pi(v)e, pi(h)esh, pi(j)e, pi(h)u* und *pi(j)u* in die Formen in (19). Die zweite Regel-Zeile in (21) löscht alle *(v), (h), (j),* die nicht durch Regel **1** abgedeckt werden, also in *hap(v)e, hap(h)esh , hap(j)e, hap(h)u* und *hap(j)u*.

1.3 Probleme mit (f)lex

1.3.1 Linke Kontexte

Es gibt mehrere Gründe, der Verwendung von Anfangszuständen in **(f)lex** kritisch gegenüberzustehen:

- Die Darstellung von Kontextspezifikationen durch Kontexte (rechts) *und* Anfangszustände (links) ist uneinheitlich
- Anfangszustände unterminieren den deklarativen Charakter von **(f)lex**. Sie beziehen sich auf die Realisierung des Scanners als Automat und sind nur im Kontext mehrerer Regeln interpretierbar, also inhärent prozedural.
- Beim Schreiben von größeren Morphologie-Komponenten führt die Verwendung von Anfangszuständen schnell zu Unübersichtllichkeit.

Den letzten Punkt veranschauliche ich kurz an der Endung *-mi* aus (12). *i nach m* taucht immer in einem bestimmten Kontext auf, nämlich rechts von "nak)1pl". Mit Hilfe von Klammerersetzung und eines linken Kontexts läßt sich das wie folgt ausdrücken:

```
(22)    nak[)]1pl\[)]     printf("i");
```

Natürlich gibt es keine derartige **(f)lex**-Regel, da die **(f)lex**-Syntax keine linken Kontexte enthält. Durch Anfangszustände ist dies nur mit etwas Aufwand zu rekonstruieren, etwa indem wir dem **(f)lex**-Programm in (13) die Regeln in (23) hinzufügen:

```
(23)
        <MI>1pl                 {printf("m"); BEGIN MI2;}
        1pl                     {printf("m"); BEGIN 0;}
        <MI2>[)]1pl\[)]         {printf("i");}
```

Dies scheint nur ein minimales Problem zu sein, ähnliche Phänomene summieren sich jedoch bei größeren Fragmenten schnell (vgl. die **(f)lex**-Implementation in Anhang **B**).

1.3.2 Determinismus

(f)lex liefert für jeden Input genau einen Output. Das führt zu Problemen mit alternativen Formen, die gerade in einer erst seit relativ kurzer Zeit standardisierten Sprache wie dem Albanischen reichlich auftauchen. Z.B. finden sich für den einfachen **imv sig** von *vë*, "legen" 4 mögliche Formen: *vurë, vur, ver, vër.*[16]

[16]Dies sind Formen, die schriftsprachlich vorkommen. Sie spiegeln bei weitem nicht die dialektale Vielfalt des gesprochenen Albanisch wider.

1.3.3 Anfangszustände, Kontexte und Morphotaktik

(f)lex bietet für sich genommen keine Morphotaktik, wobei es vom theoretischen Standpunkt abhängt, ob man dies als Mangel, oder - im Sinn einer Modularisierung von Morphologie in Morphotaktik und Allomorphie - als Vorzug auffaßt. Konkret bedeutet es u.a., daß **(f)lex**-Programme bei inkorrekter Eingabe inkorrekte Ausgabe produzieren, etwa für "imf 1sg (S(*hap*)" (statt "S(*hap*)" imf 1sg") **jahap* statt *hapja*:

(24)
```
%%
S(hap)          printf("hap");
imf             printf("j");
1sg             printf("a");
```

Dies läßt sich grundsätzlich durch Gebrauch von Anfangszuständen (und/oder rechten Kontexten) vermeiden. Das Programm in (24`) würde Input in der inkorrekten Reihenfolge ablehnen:

(24´)
```
%start STAMM INFL
%%

S(hap)              {printf("hap"); BEGIN STAMM;}
<STAMM>imf          {printf("j");  BEGIN INFL;}
<INFL>1sg           {printf("a"); BEGIN 0;}
.                   return(ERROR);}
```

Ich halte es allerdings grundsätzlich für transparenter, **(f)lex**-Programme von vorneherein mit (durch ein unabhängiges Modul) korrekt linearisiertem Input zu "versorgen", und werde diese Möglichkeit hier nicht weiter verfolgen.

1.3.4 Generative Kapazität

Mit **(f)lex** lassen sich problemlos nicht-*reguläre* Transducer beschreiben, z.B. läßt sich die *String-Relation* $a^n b^n : c^n d^n$ durch ein **(f)lex**-Programm simulieren (25). Dies entspricht einem Transducer, der beliebige $a^n b^n$ auf $c^n d^n$ abbildet. Wie Kaplan & Kay (1994:342) zeigen, impliziert die *Regularität* einer *String-Relation* die *Regularität* ihrer Domäne. Aus De Morgans Gesetz folgt, daß eine nicht-*reguläre* Domäne eine nicht-*reguläre String-Relation* impliziert. Da sowohl $a^n b^n$ wie $c^n d^n$ nicht-*regulär* sind, ist auch $a^n b^n : c^n d^n$ nicht *regulär*.[17]

[17] Eine ähnliche Konstruktion benutzt Herold (1995:330), um zu zeigen, daß bestimmte **yacc**-Parser kontextsensitive Sprachen parsen.

(25)

```
int state, count, i = 0;

%state B
%%

a* /b       {if(state != 0)
                    return(ERROR);
             else
                    {count = yylength;
                    BEGIN B;
                    state = 1;}}
<B>b*       {if(yylength != count)
                    return(ERROR);
             else
                    for(i,1,count)
                           printf("c");
                    for(i,1,count)
                           printf("d");}
.            return(ERROR);
```

Das **(f)lex**-Programm in (25) erkennt nur Ketten der Form $a^n b^n$. Akzeptierte Strings müssen mit $a^n b^n$ beginnen, da b explizit nur im Zustand B eingelesen wird, und alle anderen Zeichen (".") zur Ausgabe von ERROR führen. Auf a^n muß b^n folgen, da aufgrund der Kontextbeschränkung von a* mindestens ein b folgt, b* aber nur akzeptiert wird, wenn es die Länge von a* hat. Nach b* ist weder a möglich, da state auf 1 steht, noch b, da **(f)lex** maximal lange Ketten einliest. Alle anderen Zeichen führen wiederum zu einer ERROR-Meldung.

1.3.5 Prozeduralität und *Regularität*

Im Verlauf der bisherigen Diskussion sind wir auf mehrere Einzelpunkte gestoßen, in denen **(f)lex** prozedural vorgeht, insbesondere die Desambiguierungsmechanismen des **(f)lex**-Treibers (s. **1.1.3**) und die Interpretation von rechten Kontexten. Dies bedeutet, daß möglicherweise der **(f)lex**-Treiber selbst Eigenschaften hat, die die mit **(f)lex** modellierten Transducer nicht-*regulär* machen. Mit Ausnahme der Regelpräzedenz scheinen die genannten Desambiguierungsmechanismen bei den Anwendungen von **(f)lex** auf natürlichsprachige Morphologie allerdings praktisch irrelevant zu sein (vgl. **§1/2**). Rechte Kontexte und Determinismus in der Ausgabe ergeben sich in **mo_lex** direkt aus der (*regulären*) Semantik der Repräsentationssprache.
In **1.4** beschreibe ich die Syntax einer Meta-Sprache für **(f)lex**, die nur die wichtigsten Elemente von **(f)lex** beinhaltet (**reguläres (f)lex**). **mo_lex** läßt sich seinerseits weitgehend in **regulärem (f)lex** und via Transitivität in **(f)lex** implementieren (s. **§1/3**). Da **mo_lex** selbst *regulär* ist, ist die Frage nach der formalen Mächtigkeit von **(f)lex** damit nicht mehr unmittelbar relevant.

1.4 reguläres (f)lex

1.4.1 Unterschiede zu (f)lex

In diesem Abschnitt stelle ich die Meta-Sprache vor, die ich bei der Implementation in Anhang**B** verwende[18]. Der Name "**reguläres (f)lex**"soll andeuten, daß die Sprache die Möglichkeiten zur Erkennung nicht-*Regulärer Sprachen* in **1.3.4** ausschließt. Abgesehen davon enthält sie einige elementare Abkürzungskonventionen, um mit größeren Mengen von Anfangszuständen und längeren rechten Kontexten umzugehen.

[18] Anhang**B** enthält den Quellcode eines **(f)lex** -Programms zur Übersetzung von **regulärem (f)lex** in **(f)lex**.

1.4.2 Syntax

Programme beginnen mit dem Begrenzer "%%", dem möglicherweise - wie in **(f)lex** - eine Deklaration von Anfangszuständen vorangeht (%start Z_1 Z_2...Z_n). Regeln haben die Form:[19]

(26) *Regulärer Ausdruck* (/Rechter Kontext) "-->" String (**{**Zustandsanweisungen**}**)

Abgesehen von den Zustandsanweisungen ist dies eine Abkürzung für:

(26´) *Regulärer Ausdruck* (/Rechter Kontext) printf("String");

Falls "String" am linken oder rechten Rand Leerzeichen enthält, muß es gequotet werden ". xyz " ents pricht dann der Anweisung printf(" xyz ");". ";" steht für printf("");

1.4.3 Zustandsanweisungen

Zustandsanweisungen dienen der Zusammenfassung mehrerer Regeln, die sich nur durch ihre Anfangszustände (bzw. BEGIN-Befehle) unterscheiden.

(27) 1pl --> {MERGE MI2 MI; ELSE MI2;}

faßt Regeln **1** und **2** aus (24) zusammen.

MERGE kann beliebig viele Argumente nehmen:

(28) X --> Y {MERGE Z Z_1 Z_2...Z_n;}

entspricht

(28´) <Z_1>X {printf("Y"); BEGIN Z;}
<Z_2>X {printf("Y"); BEGIN Z;}
.
.
<Z_nn>X {printf("Y"); BEGIN Z;}

"ELSE Z"; oder "START Z"; steht für Regeln ohne Anfangszustände, z.B. wird die folgende Regel in Regel**2** in (16) übersetzt:

(29) kon --> të {START KON;}

"STAY Z_1...Z_n"; führt zu Regeln, die ihren Anfangszustand beibehalten.

(30) X --> Y {STAY Z;} entspricht:

(30´) <Z>X {printf("Y"); BEGIN Z;}

Sinnvoll ist STAY in Verbindung mit ELSE, um die Anwendung einer BEGIN-Anweisung in einem bestimmten Zustand abzufangen:

(31) aor {STAY AO3; ELSE AOR;}

---

[19] Token der Repräsentationssprache sind gequotet, alle anderen Ausdrücke wie "*Regulärer Ausdruck*" sind metasprachlich. Ausdrücke in runden Klammern ("(,)") sind optional.

Alle Zustandsanweisungen werden durch Semikolon (";")abgeschlossen. Innerhalb der geschweiften Klammern können beliebig viele Zustandsanweisungen aufeinanderfolgen.

1.4.4 Implizite rechte Kontexte

Längere rechte Kontexte machen **(f)lex**-Programme schnell unübersichtlich und enthalten unnötige prädiktable Information. Nehmen wir z.B. Stämme, die Allomorphie in Abhängigkeit von ihren Suffixen aufweisen. Die Grundstruktur von Stämmen ist, wie oben erläutert (S*xy*+V*uv*+K*st*). (32) zeigt ein typisches Beispiel, (33) die nötigen Regeln:

(32)

shoh aor 3sg	***p**-a-0*	(((S93+V2o+K3b)aor)3sg)
shoh prs 3sg	***sh**-eh-0*	((S93+V2o+K3b)3sg)

(33)
```
%%
S93/"+V2o+K3b[)]aor     -->     p
S93                     -->     sh
```

Da die Ausbuchstabierung von (absoluten) Stämmen generell nicht vom Rest des Stammes abhängt, wäre die Formulierung von Regel **1** in (33) als (33´) sehr viel durchsichtiger.

(33´)
```
S93/"+V2o+K3b[)]aor     -->     p
```

In **regulärem (f)lex** bewirkt die Zeile:

(34)
```
&KONTEXT:     "+V2o+K3b[)]"
```

daß der Kontext -Operator ("/") in allen folgenden Zeilen (bis zur nächsten Kontext-Deklaration) bei der Übersetzung nach **(f)lex** zu "/V2o+K3b[)]" expandiert wird. (33´) ist also nach (34) äquivalent zu Regel **1** in (33) (ohne (34)). Diese Strategie lohnt sich nur, wenn eine Kontextdeklaration viele Möglichkeiten abdeckt. "V2o+K3b" wird sicher nicht auf alle Stämme passen. Eine Lösung bieten wieder *Reguläre Ausdrücke*: "+V[123][0-9a-z]+K[1-3][0-9a-z[)]" matcht alle bisher besprochenen Stammerweiterungen. (35) zeigt einen etwas längeren Ausschnitt aus der Implementation in Anhang **B** im Vergleich zur Alternative ohne Kontextdeklaration (b):

(35)

a.
```
&KONTEXT: "+V[123][0-9a-z]+K[1-3][0-9a-z[)]"
S93/aor     -->     pa
S93         -->     sh

S95/aor     -->     dh
S95         -->     j

S97/aor     -->     p
S97         -->     k
```

b.
```
S93/"+V2o+K3b)aor"      -->     pa
S93                     -->     sh

S95/"+V2p+K3c)aor"      -->     dh
S95                     -->     j

S97/"+V3d+K3e)aor"      -->     p
S97                     -->     k
```