3 (F)LEX UND MO_LEX

3.0 Überblick

In diesem Kapitel zeige ich, daß sich **mo_lex**-Theorien - mit gewissen Einschränkungen - in **(f)lex** implementieren lassen. Es wäre sicher wünschenswert, einen **mo_lex**-Compiler zu entwerfen, der für **mo_lex**-Theorien ohne den Umweg über **(f)lex** Übergangstabellen generiert und entsprechende Treiberprogramme für Generierung und Parsing vorsieht, was jedoch den Rahmen dieser Arbeit sprengen würde. Mein Ziel hier ist lediglich, die Möglichkeit zu demonstrieren, **mo_lex** (als Generator) unter Rückgriff auf bekannte Programmier-Techniken zu realisieren. Um von Details der C-Syntax wegzuabstrahieren, gehe ich für **(f)lex** von der Syntax für **reguläres (f)lex** (s. **§1/1**) aus. Für die Darstellung von **mo_lex** in **regulärem (f)lex** stellt sich neben einigen kleineren Fragen, die ich in **3.1** behandle, als Haupt-Problem die Umsetzung von linken Kontexten (**3.2**).

3.1 Allgemeines

3.1.1 Einschränkungen

Für die folgenden Ausführungen erweist es sich als vorteilhaft, Kerne in **mo_lex**-Theorien auf die Disjunktion von Symbolen zu beschränken. Praktisch alle Constraints in **§2/4** folgen ohnehin dieser Beschränkung oder verletzen sie in trivialer Weise durch Einbeziehung einzelner Klammern. Constraints mit komplexen Kernen lassen sich außerdem normalerweise durch mehrere Constraints mit atomaren Kernen ersetzen, z.B. (1a) durch (1b):

(1)

a.			b.		
C\AB/D	-->	X	C\A/BD	-->	X
			CA\B/D	-->	;

3.1.2 Die Auflösung von Constraint-Konflikten

mo_lex unterscheidet sich in der Gewichtung der Konfliktlösungs-Strategien von **(f)lex**. Unter der Annahme, daß Constraint-Kerne disjungierte Symbole sind, erübrigen sich aber sowohl die Bevorzugung längerer Strings (alle Kerne haben jetzt die Länge 1) als auch die Links-Rechts-Strategie: Für die Übersetzung jedes Buchstaben B sind jetzt nur Constraints der Form X\B/Y --> Z relevant, die sich, was die Anwendungs-Präferenzen angeht, ausschließlich durch ihre lineare Anordnung in der Theorie unterscheiden.

3.2 Linke Kontexte

Für die Diskussion gehe ich von einigen vereinfachenden Annahmen aus, die ich nach und nach auflöse. In **3.2.1** skizziere ich die Behandlung von linken Kontexten, die aus einzelnen Symbolen bestehen, in **3.2.2** verallgemeinere ich den Algorithmus für Symbol-Strings ohne Selbstüberschneidung. In **3.2.3** berücksichtige ich zusätzlich linke Kontexte, die sich mit sich selbst überschneiden (z.B. *aa*) und in **3.2.4** linke Kontexte in disjunktiver Normalform ($(String_1 | String_2 | .... | String_n)$).

3.2.1 Linke Kontexte der Länge 1

Linke Kontexte der Länge 1, d.h. solche, die aus **mo_lex**-Symbolen bestehen, bilden ihrerseits den Kern von **mo_lex**-Constraints. Ein einfaches Beispiel findet sich in (2):

(2)

C1	aor	-->	;
C2	aor[])\3sg	-->	u
C3	S(pi)	-->	pi
z.B.	(((S(pi))aor)3sg)	=>	pi-u

Wir ersetzen nun jeden linken Kontext LK durch einen charakteristischen Start-Zustand S(LK) und versehen jedes Constraint, dessen Kern gleich LK ist, mit der Start-Anweisung "BEGIN S(LK);". (3) enthält den expliziten Algorithmus, (4) die Anwendung auf (2):

(3) **Ersetzungs-Algorithmus (1.Version)**

Ersetze alle Constraints der Form $LK_1\backslash K_1/RK_1$
durch $<S(LK_1)>K_1/RK_1$.

Hänge an alle Constraints mit dem Kern $LK_2\backslash K_2/RK_2$, so daß K_2 in einem Constraint der Theorie ein linker Kontext ist, {START $S(K_2)$;} an.

Hänge an alle anderen Constraints {START 0;} an.

(4)

C1	aor	-->	;	{START Z;}
C2	<Z>3sg	-->	u	{START 0;}
C3	S(pi)	-->	;	{START 0;}

(f)lex wird sich jetzt genau dann im Zustand Z befinden, wenn *aor* eingelesen wurde.

3.2.2 Strings ohne Überschneidung als linke Kontexte

Zu Zwecken der Exposition bedienen wir uns vorläufig des vollen Ausdrucksreichtums von **(f)lex**, einschließlich globaler C-Variablen. Wir führen für jeden linken Kontext der Länge n $LK = B_1...B_n$ in der **mo_lex**-Theorie in das **(f)lex**-Programm als FLAG eine globale Integervariable F(LK) = 0; ein. Wir suchen Constraints mit Kern B_j $(1 \leq j \leq n\text{-}1)$ und versehen sie mit der C-Anweisung in (5):

(5)

```
IF(F(LK) == j-1)    F(LK) = j;
ELSE F(LK1) = 0;
```

Constraints mit dem Kern B_n fügen wir die Anweisung in (6) hinzu:

(6)

```
IF(F(LK) == n-1)   F(LK) = 0;
BEGIN Z(LK);
```

Constraints, deren Kern in keinem linken Kontext enthalten ist, erhalten die Anweisung "F(LK) = 0;" Constraints mit dem linken Kontext LK ersetzen wir durch eine **(f)lex**-Regel mit dem Anfangszustand "<Z(LK)>". Alle Regeln ohne BEGIN-Anweisungen schließen wir mit "BEGIN NOR;" (als Default-Zustand) und alle Regeln mit BEGIN-Anweisungen mit "ELSE BEGIN NOR;" ab.

Sobald der **(f)lex**-Scanner den ersten Buchstaben von LK einliest, setzt er F(LK) auf 1 und erhöht es, falls LK vorliegt, solange, bis es n-1 erreicht. An diesem Punkt geht **(f)lex** in den Zustand Z(LK); falls eine Regel mit Startzustand <Z(LK)> den nächsten Buchstaben matcht, wird sie angewendet. Umgekehrt wird F(LK) auf 0 gesetzt, sobald **(f)lex** auf einen Buchstaben stößt, der nicht die Stelle i+1 (für i der aktuelle Wert von F(LK)) von LK einnimmt, oder der Kontext-String beendet ist. D.h. jede Unterbrechung (und jedes Ende) eines linken Kontextes führt zum Abbruch des Erkennungsvorgangs. Entsprechend wird jede Regel, die im Zustand Z(LK) angewendet wird, in NOR oder (aufgrund eines anderen FLAGs) in einen anderen Zustand übergehen, d.h. Regeln der Form <Z(LK)>... können nur unmittelbar rechts von einem korrekt eingelesenen LK angewendet werden. (7´) zeigt die Anwendung des Algorithmus auf (7) (die rechte Seite von Constraints bzw. Regeln ist weggelassen, da sie für das Beispiel irrelevant ist):

(7)		(7´)		
C1	abc\d	**R1**	$<Z_1>$d	{F_1 = 0; BEGIN NOR;}
C2	a	**R2**	a	{IF(F_1 == 0) F_1 = 1; ELSE F_1 = 0; BEGIN NOR;)}
C3	b	**R3**	b	{IF (F1 == 1) F1 = 2; ELSE F1 = 0; BEGIN NOR;}
C4	c	**R4**	c	{IF(F_1 == 2) BEGIN Z_1; ELSE {F_1 = 0; BEGIN NOR;}}

3.2.3 Strings mit Überschneidungen als linke Kontexte

Dieses Vorgehen berücksichtigt allerdings nicht die Möglichkeit, daß linke Kontexte sich mit sich selbst überschneiden. Z.B. würde in *ababa* für den linken Kontext *aba* der FLAG beim Einlesen des dritten *a* auf 2 stehen und das zweite *aba* überlesen werden. Ein ähnliches Problem ergibt sich generell, wenn Buchstaben an mehreren Stellen eines Kontext-Strings vorkommen, etwa mit *aab* und dem linken Kontext *ab*. Nachdem *a* eingelesen wurde, ist der FLAG auf 1 und wird beim zweiten *a* - inkorrekterweise - nicht mit 1 initialisiert. Außerdem ist es beim Einlesen eines Buchstabens möglich, daß mehrere Kontexte gleichzeitig erfüllt sind, z.B. für die linken Kontexte *abc* und *bc*. **(f)lex** kann aber immer nur in einen Anfangszustand übergehen.
Befassen wir uns zuerst mit dem letzten Problem. Es tritt offensichtlich nur ein, wenn linke Kontexte Suffixe anderer linker Kontexte sind ("konkurrierende linke Kontexte", vgl. (8b)), was wir leicht überprüfen können. Wir konstruieren Zustände, die den möglichen Kombinationen mehrerer Startzustände entsprechen. Wenn B der letzte Buchstabe mehrerer konkurrierender linker Kontexte ist, gehen wir für alle Regeln, deren Kern B ist, in den Zustand über, der mit allen linken Kontexten korrespondiert, die jeweils mit B abschließen. Linke Kontexte ersetzen wir jetzt durch eine Liste aller Zustände, die mit LK korrespondieren.
Die beiden anderen Punkte betreffen den Anfang, d.h. die potentiellen ersten Buchstaben von linken Kontexten. Die Lösung, die ich skizziere, beruht ebenfalls auf einer Menge von FLAGs für jeden linken Kontext. Wenn der erste Buchstabe (B_1) eines linken Kontexts LK mit korrespondierendem FLAG F eingelesen wird, trifft außerdem genau eine der folgenden Feststellungen zu:

(1) F = 0 (der Fall, der oben berücksichtigt ist).
(2) F = i > 0, und B_1 ist der i+1-te Buchstabe in LK.
(3) F1 = i > 0, und B_1 ist *nicht* der i+1-te Buchstabe in LK.

Im dritten Fall scheitert die Überprüfung des Kontexts. Dennoch ist es möglich, daß das neu eingelesene B_1 der Anfang eines korrekten LK ist. Wir setzen F also auch in diesem Fall auf 1. Der zweite Fall ist etwas komplexer, weil wir gleichzeitig mehrere identische, gegeneinander verschobene Kontexte berücksichtigen müssen. Da linke Kontexte endliche Ketten sind, enthalten sie endlich viele (j) Instanzen

des ersten Buchstaben B_1, und es gibt maximal j Möglichkeiten, B1 einzulesen, solange F(LK) ungleich 0 ist. Wir schaffen j FLAGs für LK. Sobald B_1 gelesen wird, suchen wir einen freien FLAG, d.h. einen, der auf 0 steht oder mit B_1 an dieser Stelle unvereinbar ist, und setzen ihn auf 1. Alle anderen FLAGs setzen wir um 1 weiter, wenn ihre aktuelle Position ungleich 0 und mit B_1 vereinbar ist, andernfalls auf 0. DaDa Da jeder FLAG maximal j-mal B_1 einlesen kann, ohne daß die Identifikation von LK scheitert, können wir, solange alle FLAGs frei sind, für jedes neu eingelesene B_1 einen FLAG initialisieren. Da LK nur j-mal B_1 enthält, ist spätestens nach j+1-mal B_1 der LK, der vom ersten FLAG geprüft wird, vollständig eingelesen oder gescheitert, d.h es wird ein FLAG frei. Dementsprechend wird für jede weitere Instanz von B_1 jeweils ein FLAG frei, den wir initialisieren können.

3.2.4 Ersetzungs-Algorithmus (endgültige Version)

(9) enthält den expliziten Algorithmus, (8) einige nützliche Definitionen. Fettgedruckte, geschweifte Klammern ("**{**, **}**") markieren informell C-Text, der am Ende der jeweiligen Regeln einzufügen ist.

(8)

a. **Definition *Subsumption***
Ein linker Kontext LK_1 subsumiert einen anderen linken Kontext LK_2, gdw. LK_2 ein Suffix von LK_1 ist.

b. **Definition *Konkurrenz***
LK_1 und LK_2 *konkurrieren*, gdw. LK_1 LK_2 *subsumiert*, oder LK_2 LK_1 *subsumiert*.

c. **Definition (*Maximale*) *Fraktion***
Eine Menge F von linken Kontexten aus einer **mo_lex**-Theorie M, so daß jedes Element von F mit allen anderen linken Kontexten in F *konkurriert*, nenne ich eine *Fraktion*. Eine *Fraktion*, die für jedes Element E alle linken Kontexte in M enthält, die von E *subsumiert* werden, nenne ich *maximal.*

(9) **Ersetzungs-Algorithmus**

a. Konstruiere für jede maximale Fraktion F in der **mo_lex**-Theorie den Zustand Z(F).

b. Ersetze jeden linken Regelkontext LK durch $<Z_1, ..., Z_{1m}>$, mit $Z_1, ..., Z_m$ eine Liste aller Z(F), so daß F LK enthält.

c. Führe für jeden linken Kontext LK, der j-mal den ersten Buchstaben First(LK) enthält, j Integer-Variablen V_{1-j} ein.

/* Beende Einlesen eines linken Kontexts, gehe in den entsprechenden Zustand */

d. Für jede Regel, deren Kern K == Last(LK_i) ($1 \le i \le n$)

{Sei int n = die maximale Größe von *maximalen Fraktionen* aus LK_i;

Solange (n > 0)

{**Für** jede maximale Fraktion F(LK_1...LK_m) aus der Menge der LK_i, so daß Länge(LK_i) = n

Wenn für jeden linken Kontext LK in F mit Länge(LK) > 1 mindestens eine Variable V_j(LK) == Länge(LK)-1

{BEGIN Z(F); BREAK;}

–n;

}}

/* Starte Einlesen eines linken Kontexts */

e. **Für** jeden linken Kontext LK, der Länge L > 1 mit j Variablen
Für jede Regel, deren Kern K == First(LK)
{ int z = 1;
Solange (z ≤ j)
Wenn (V_z == 0 oder LK[V_{z+1}] != First(LK))
{
V_z = -1;
BREAK;
}
++z; **}**

f. **/* Einlese-Prozedur */**

Für jeden linken Kontext LK mit j Variablen
Für jeden Buchstaben K in LK, der in LK an i-ter Stelle steht (i != Länge(LK))
Für jede Regel mit Kern K
{int z = 1;
Solange (z ≤ j)
Wenn (V_z == (Länge(LK)))
V_z = 0;
Sonst Wenn (K == LK[V_{z-1}])
++V_z;
Sonst Wenn (V_z == -1)
V_z = 1;
Sonst
V_z = 0;
++z; }**}**

3.2.5 Linke Kontexte in regulärem (f)lex

Wir haben jetzt endlich viele Variablen mit endlich vielen Werten (für jede Variable entsprechend der Länge des jeweiligen linken Kontexts). Bei jeder **(f)lex**-Regel werden die Variablen in Abhängigkeit von den aktuellen Va riablen-Werten und Konstanten der Theorie (Länge von linken Kontexten etc.) verändert. Da es nur endlich viele Kombinationen von Variablen-Werten gibt, können wir jeder Wert-Kombination aller Variablen eine natürliche Zahl zuordnen. Wir erhalten endlich viele Zahlen und können diese als Werte einer einzigen Variablen betrachten. Die Veränderungen von einzelnen Variablen-Werten stellen wir als Übergänge eines (komplexen) Variablen-Werts zu einem anderen dar.

Die Struktur der Regeln, die wir so gewinnen, ist in (10) dargestellt (Runde Klammern "(,)" stehen dabei für Optionalität):

(10) (<Z_{A1}, ..., Z_{1AN}>)A --> C {IF FLAG == X_1 {FLAG = X_1´; (BEGIN Z_1;)}}
{IF FLAG == X_2 {FLAG = X_2´; (BEGIN Z_2;)}}
.
.
.
{IF FLAG == X_{2n} {FLAG = X_n´; (BEGIN Z_n;)}}}

Dies können wir jedoch genausogut mit **(f)lex**-Zuständen darstellen:

(11) $<Z(X_1)>A$ --> C {BEGIN $Z(X_1´)$;}
$<Z(X_2)>A$ --> C {BEGIN $Z(X_2´)$;}
.
.
.
$<Z(X_n)>A$ --> C {BEGIN $Z(X_n´)$;}
A --> C {BEGIN NOR;}

Die einzigen Fälle, die sich damit nicht korrekt erfassen lassen, sind Regeln, die bereits nach dem Algorithmus in (9) mit Anfangszuständen beginnen oder Anweisungen für Zustandsübergänge enthalten. Letztere enthalten nun zwei sich widersprechende BEGIN-Anweisungen, nennen wir den alten Z_A, den neuen Z_N. Wir konstruieren für jeden solchen Fall einen neuen Zustand der der Kombination der beiden Zustände entspricht: $Z(Z_A, Z_N)$ und ersetzen die beiden BEGIN-Anweisungen durch "BEGIN $Z(Z_A, Z_N)$;". Analog verfahren wir mit Regeln, die mit Anfangszuständen beginnen. Jeder neugeschaffene Anfangszustand Z_N wird durch $Z(Z_A, Z_N)$ ersetzt. Nehmen wir als Beispiel für Zustandsanweisungen eine vereinfachte Version von (10):

(12) A --> C {IF FLAG == X_1 {FLAG = $X_1´$; BEGIN Z_1;}}

Dem FLAG-Wert X_1 ($X_1´$) entspricht wieder der Zustand $Z(X_1)$ ($Z(X_1´)$). Die Regel geht also in den Zustand $Z(Z_1, Z(X_1´))$ über. Der Anfangszustand der Regel ist wie in (10) $Z(X_1)$:

(12´) $<Z(X_1)>A$ --> C {BEGIN $Z(Z_1, Z(X_1´))$;}

In (13) sind die Anfangszustände von (10) realisiert:

(13) $<Z_{A1}, ..., Z_{AN}>A$ --> C {IF FLAG == X_1 FLAG = $X_1´$;}

Jeder Zustand Z_{Ai} wird jetzt durch $Z(Z_{Ai}, Z(X_1)$ ersetzt:

(13´) $<Z(Z_{A1}, Z(X_1)), ..., (Z_{AN}, Z(X_1))>A$ --> C {BEGIN $Z(X_1´)$;}

An diesem Punkt haben wir alle Anweisungen, die sich auf FLAGs beziehen, eliminiert. Wir können also auch die dazugehörigen Integer-Variablen streichen und haben eine Repräsentation, die wieder voll kompatibel mit **regulärem (f)lex** ist.

3.2.6 Disjunktionen von Strings als linke Kontexte

Disjunktionen von Strings in linken Kontexten erfordern nur eine leichte Modifikation des Algorithmus in **3.2.4** Wir ersetzen Klausel (9b) durch (9b´):

(9b) Ersetze jeden linken Regelkontext LK durch $<Z_1, ..., Z_m>$, mit $Z_1, ..., Z_m$ eine Liste aller Z(F), so daß F LK enthält.

(9b´) Ersetze jeden linken Regelkontext der Form ($LK_1 | LK_2 | ... | LK_{1n}$) durch $<Z_{11}, ..., Z_{1m1}, Z_{21}, ..., Z_{2m2}, ..., Z_{n1}, Z_{n2}, ..., Z_{n,n}>$, mit $Z_{1j}, ..., Z_{mj}$ eine Liste aller $Z(F_j)$, so daß F LK_j enthält