GOETHE UNIVERSITÄT FRANKFURT
Fachbereich Informatik
Künstliche Intelligenz und Softwaretechnologie
Prof. Dr. Schmidt-Schauß

Diplomarbeit

Implementierung eines Algorithmus in einer funktionalen Programmiersprache zum String-Matching von komprimierten Patterns mit Zeichen-Variablen in komprimierten Strings

vorgelegt von: Frank Abromeit im Dezember 2013

Student:	Frank Abromeit
Studiengang:	Informatik
Matrikelnummer:	1262151
E-Mail:	abromeit@informatik.uni-frankfurt.de

Erklärung gemäß DPO §11 Abs. 11

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt, den 27.12. 2013

_____

(Frank Abromeit)

Anlagen

1 x DVD :
- Quellcodes der Haskellimplementierung
- Haddock Dokumentation
- Virtualbox-Image CentOS 6.4 mit Implementierung

Inhalt

Verzeichnis der Abbildungen

Verzeichnis der Abkürzungen

Alg	Algorithmus
CCSM	Compressed Character Submatch
CFG	Context Free Grammar
LCCSM	Linear Compressed Character Submatch
Lochproduktion	Produktion der Form $T \rightarrow t$, definiert eine wildcard Position im Suchmuster
Nichtterminalproduktion	Produktion der Form $T \rightarrow T_1 T_2$
SCFG	Singleton Context Free Grammar
SLP	Straight Line Program
SLP-Grammatik	SCFG
Terminalproduktion	Produktion der Form $T \rightarrow t$

1 Einleitung

Der Algorithmus von Schmidt-Schauss[3] beschreibt einen polynomialzeit Algorithmus für das LCCSM Problem [1]. Dieses Problem ist eine Variante des allgemeinen komprimierten Pattern-Matching Problems für das effiziente Algorithmen (Lifshits[6], Jez[7]) mit polynomieller Laufzeit existieren. Dabei sind im komprimierten Suchstring zusätzlich Zeichenvariablen erlaubt. Zeichenvariablen sind wildcards, also Positionen im Suchstring an denen ein beliebiges Zeichen stehen darf. Um die polynomielle Laufzeit einzuhalten dürfen maximal $|G|$ solcher Variablen im Suchstring enthalten sein, wobei G eine SLP-Grammatik ist mit der die Strings komprimiert worden sind . Der Algorithmus benutzt eine Verallgemeinerung des Theorems von Fine & Wilf [1, Anhang] für periodische Strings auf Strings mit Zeichenvariablen.

1.1 SLP und SCFG

SLP (*Straigth line Programs*) und SCFG sind spezielle kontextfreie Grammatiken, die sich für die Stringkomprimierung eignen. SCFG (*Singleton Context Free Grammar*) sind nichtrekursive kontextfreie Sprachen in Chomsky-Normalform. Ein Nichterminal einer solchen Sprache erzeugt genau ein Wort (*singleton*) . Eine SCFG lässt sich durch das Tupel (Σ, V, P, S) beschreiben, wobei Σ das Alphabet der Terminalzeichen, V die Menge der Nichterminale, P die Produktionenmenge und S das Startsymbol der Grammatik beschreibt. Produktionen haben die Form

$$A \rightarrow B\ C \quad \text{oder} \quad A \rightarrow t, \quad \text{wobei } A, B, C \in V \text{ und } t \in \Sigma \text{ ist.}$$

Beispiel :

Für die SCFG $G = (\Sigma = \{a, b\}, V = \{S, A, B, C\}, P = \{S \rightarrow A\,B, B \rightarrow C\,C, A \rightarrow a, C \rightarrow c\}, S)$ ist der Ableitungsbaum T gegeben. T ist ein binärer Baum, für den gilt, dass im Teilbaum mit Wurzel X X selbst nicht mehr vorkommt (G ist nichtrekursiv). Das von G erzeugte Wort val(G) ist *acc*.

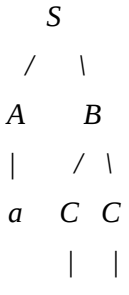


Abb 1: (SCFG Ableitungsbaum)

[1] Linear compressed chararcter submatch problem

Der Formalismus der SCFG ist äquivalent zu SLP insoweit, dass SLP nichtrekursive kontextfreie Grammatiken (CFG) sind, die aber nicht in Chomsky Normalform (CNF) sein müssen([4]. S.3). Das Umwandeln eines SLP in die CNF (SCFG) gelingt in linearer Zeit ([4]. S.3). Mit dem Plandowski-Algorithmus[5, Kap.4] ist es möglich in polynomieller Zeit zu testen ob zwei SLP das gleiche Wort erzeugen. Auch die Transformation von LZ77[8] komprimierten Strings in SLP ist effizient möglich [4, Theorem 6]. Mit SCFG und SLP lassen sich exponentielle Kompressionsraten erzielen.

1.2 Das CCSM und das LCCSM Problem [3, Kap. 2.4]

Das hier betrachtete LCCSM Problem ist eine vereinfachte Variante des CCSM Problems. Dabei sind der Suchstring *S* und der Zielstring *T* (in dem gesucht wird) durch die komprimierten Grammatiken G_S und G_T gegeben, d.h. S = val(G_S), T = val(G_T) . $G_S = (\sum \cup Z , V_1 , P_1 , S_1)$ und $G_T = (\sum , V_2 , P_2 , S_2)$, wobei das Z in der Definition von G_S eine Menge von Zeichenvariablen beschreibt. Das CCSM Problem besteht dann darin festzustellen, ob es eine Ersetzung $\sigma : Z \rightarrow \sum$ für die Zeichenvariablen, die in P_1 vorkommen gibt, so dass *S* ein Substring von *T* ist. Im LCCSM Problem gilt ausserdem die Einschränkung, dass eine solche Variable v höchstens einmal benutzt werden darf[2]. D.h. v kommt in der Ableitung von S_1 genau einmal vor. Das vereinfacht das Problem wesentlich, weil bei Mehrfachvorkommen darauf geachtet werden muss, dass alle Vorkommen die gleiche Ersetzung haben.

Beispiel: Sei $P_1 = \{S \rightarrow A\ B, B \rightarrow C\ D, A \rightarrow v, C \rightarrow c, D \rightarrow d\}$ für das Suchmuster S, und eine Zeichenvariable *v*. Die unkomprimierter Darstellung s = val(S) ist dann vcd. Das Suchmuster würde dann im String *ccd* oder *dcd* gefunden werden. Da hier aber nach einem Submatch gefragt wird kommt vcd auch im einem Zielstring *abccda* vor. Anwendungen für das LCCSM Problem sind allgemein die Suche in komprimierten Texten und in der Biologie. Hier sind Fragestellungen der DNA - Analyse wo Strings mit besonderen Eigenschaften wie z.B. Single Nucleotide Polymorphism [3] untersucht werden interresant.

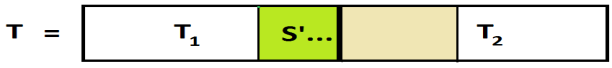
[2] Worte des LCCSM Problems werden in der Literatur [8] auch *partial words* bezeichnet

1.3 Match-Algorithmus für LCCSM mit dynamischer Programmierung

Der Match-Algorithmus nutzt die Komposition von Eigenschaften einzelner Nichtterminale aus, um Aussagen über die Vorkommen des Suchmusters im dem von G_T erzeugten Wort zu machen.
Die Nichterminale von SCFGs sind über eine Präfix/Suffix Relation miteinander verbunden, denn jedes Nichterminal ist Präfix und / oder Suffix mind. eines anderen. Das wird durch die Struktur einer SCFG bestimmt. Sei die Produktion $T \rightarrow T_1\ T_2$ im Bild unten gegeben, wobei der grüne Bereich einen Suffix von T_1 markiert und der beige Bereich einen Präfix von T_2 bezeichnet. Die Relationen *istPräfixVon* ist transitiv bezgl. T und T_1, da ein Präfix von T_1 immer auch ein Präfix von T ist. Die Relation *istSuffixVon* ist *transitiv* bezgl. T und T_2, da ein Suffix von T_2 immer auch ein Suffix von T ist. D.h. bei der Konkatenation von T_1 und T_2 lassen sich die Suffix und die Präfix Eigenschaften auf T übertragen.

(Abb. 2 Präfix / Suffix Komposition)

Sei nun S das Nichterminal, dass den Suchstring repräsentiert. Für die Vorkommen von S in T gibt es zwei Möglichkeiten. Entweder ist S vollständig in T_1 oder T_2 enthalten oder S beginnt in T_1 und endet in T_2. In ersterem Fall kann man diese Eigenschaft auch auf T vererben, denn wenn S vollständig in T_1 (T_2) vorkommt, dann kommt S auch in T vor. Wie lässt sich nun ein Vorkommen von S, dass in T_1 beginnt und in T_2 endet aus Inforrmation von T_1 und T_2 zusammensetzen. Hier kommt nun die Präfix / Suffix Information zum Einsatz. In einer Präfixtabelle werden für ein Nichtterminal T alle Präfixe von S, die Suffixe von T sind gespeichert. Wie man im Bild unten sieht lässt sich diese Information ausnutzen.



(Abb. 3 Ein Präfix S' von S ist ein Suffix von T_1)

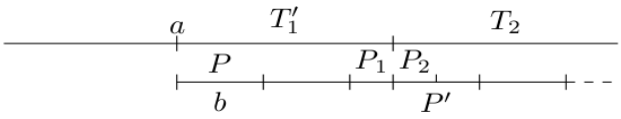
Für die Fortführung des Präfix S' von S in T_2 hinein ergeben sich drei Fälle (siehe Bild oben)

1. Die Konkatenation von S' und T_2 ist wieder ein Suffix von T
2. Die Konkatenation von S' mit einem Präfix von T_2 ergibt S (vollständiges Vorkommen)
3. Die Konkatenation von S' T_2 ist ein Präfix von T

Im Fall 1) kann S'T_2 in der Präfixtabelle von T gespeichert werden und im zweiten Fall muss das Vorkommen von S über die Verlängerung von S' in T_2 hinein bestimmt werden. Der 3. Fall bringt keine nutzbare Information. Die Präfixtabelle und die Ergebnistabelle für S=val(G_S) und T=val(G_T) können dann erstellt werden, wenn die Nichterminale der Zielgrammatik G_T in der *a*ufsteigenden Reihenfolge[3] der topologischen Sortierung abgearbeitet werden.

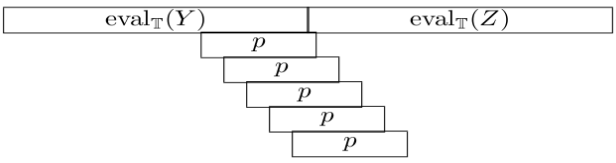
1.3.1 Komplexe Vorkommen von S

Der Fall 2 (oben) soll nun genauer betrachtet werden.



(Abb. 4 aus [3], S.283 Komplexe Vorkommen von S)

Für Vorkommen von S in T, die in T_1 beginnen und in T_2 aufhören gibt es einen Suffix T_1' von T_1 und einen Präfix P' von T_2 und S = $T_1'T_2$. Die Struktur von *S* kann man ausnutzen, wenn S *periodisch*[4] ist. Nur soviel, die Periodizität von S wird später dazu benutzt, um die Präfixtabelle 'klein zu halten'.



(Abb. 4a aus [4] S. 10)

In der Abbildung 4a) ist das Suchmuster mit p bezeichnet. P ist offensichtlich periodisch, denn es lässt sich mit sich selbst überlappen. Der Algorithmus von Schmidt-Schauss nutzt diese Eigenschaft aus. D.h. es wird versucht Perioden des Suchmuster zu finden, um diese dann als Information in der Präfixtabelle und Ergebnistabelle mit abzuspeichern.

---

[3] D.h. unabhängige vor Abhängigen Produktionen. Z.b. ist für *X → a, Y → X X, Y abhängig von X, weil X in der rechten Seite von Y vorkommt.*

[4] Ein String s hat die Periode p, wenn s[i] = s[i+p]. So hat s= aabaab die Periode 3. Das gilt auch für s=aabaabaa.

2 Implementierung des Match-Algorithmus

2.1 Der Match Algorithmus

Die Arbeit beschreibt eine Implentierung des Algorithmus (Alg. 4.1, S.283 [3]) in der Programmiersprache *Haskell* [7]. Die Eingabe für den Algorithmus sind die beiden Grammatiken für den Suchstring S und den Zielstring T, wobei diese jeweils als SCFG[5] vorliegen. Zeichenvariablen werden in G_S entweder mit sog. *Lochproduktionen* oder mittels absoluter Positionen im Suchstring definiert. (vergl. Kap. 3.6). Die Nichtterminale von G_T werden in aufsteigender topologischer Reihenfolge durchlaufen und die Präfix- und die Ergebnistabelle entsprechend der Fallunterscheidung ([3] Alg. 4.1 S.282, 283) erneuert. Nach jedem Updateschritt wird die Präfixtabelle komprimiert (Kap. 2.2.1), um die polynomielle Laufzeit des Alg. einzuhalten.

Eingabe : SCFG G_S, G_T, Lochproduktionen oder Lochpositionen für S

Ausgabe : *1. Präfixtabelle mit allen Präfixvorkommen des Suchstring in T*
2. Ergebnistabelle mit allen vollständigen Vorkommen des Suchstrings in T

Match - Algorithmus : (Die genannten Fälle beziehen sich auf [3], S. 283, 284)
1. Transformiere die Eingabe so, dass die match - Prozedur aufgerufen werden kann in Kap 2.1.1 .
2. Für alle T in G_T in aufsteigender topologischer Reihenfolge :
Falls (Fall 1 oder 2a) :
Gehe zu Kap. 2.1.2
sonst :
Falls (Fall 2b) :
Gehe zu Kap. 2.1.3
sonst :
Berechne Parameter für die komplexen Fälle 1 - 5 in Kap. 2.1.3
gehe dann führe die Fallunterscheidung 1 - 5 (s. Kap. 1.3.1)
in Kap. 2.1.6 aus.

3. Komprimiere Präfixtabelle [T]
4. Merge die Ergebnistabelle in Kap. 2.2.2

[5] Die verwendeten Haskell-Datentypen sind in der Haddock Dokumentation genau erklärt.

2.1.1 Prepare_match (Vorbereiten der Parameter für match)

Vor dem Start werden einige Vorarbeiten ausgeführt. Dazu gehören

- Berechnung der absoluten Positionen von Löchern im Suchmuster (siehe 2.3.1)
- Umwandlung der Bezeichner der Nichtterminale in Integerzahlen - Das erleichtert später das Hinzufügen neuer Produktionen, sowie die Iteration über die Nichtterminale von G_T, da nur eine Zahl hochgezählt werden muss.
- Topologische Sortierung von G_T - Für die dynamische Programmierung müssen die Nichterminale von G_T in der Reihenfolge unabhängiger zu abhängigen Produktionen durchlaufen werden.
- Vereinigung der Grammatiken G_S und G_T zu einer Grammatik G - Für Gleichheitstests auf Nichtterminalen (z.B. NT_1 ist Präfix von NT_2 wird der *Plandowski* Algorithmus[5] verwendet , der nur eine Grammatik als Eingabe hat.
- Berechnen der unkomprimierten Längen der Nichtterminale - Damit der Alg. funktioniert wird die Länge eines Nichtterminals in jeder Produktion von G gespeichert.
- Initialisieren der Präfix und Resultat Tabellen - Die Datenstrukturen für die Tabellen müssen angelegt werden.

prepare_match (G_S, G_T, S, hole_positions)

(Umwandeln der Grammatiktypen in SCFG a Int)

G_S, s_map = gr_with_integer (G_S)

G_T, t_map = gr_with_integer (G_T)

(Topologisches Sortieren der Zielgrammatik)

topsorted_productions_G_T = topSort G_T

G_T_top = listToSCFG (wordlength (listToSCFG (topsorted_productions_G_T)))

(Hole Name von Startsymbol der Zielgrammatik)

G_T_start = topsorted_productions_G_T[|topsorted_productions_G_T| - 1)]

(Vereinige G_S und G_T)

G = scfg_append (G_T_top, G_S)

(Berechne Wortlängen)

G = listToSCFG (wordLength G)

(Initialisiere die Präfixtabelle)
PT = init_prefix_map (Nichterminalbezeichner aus G_T)

(Initialisiere die Ergebnistabelle)
RT = init_result_map (Nichterminalbezeichner aus G_T)

(Hole neuen Bezeichner für S)
S = $|G_T|$ + (s_map!S[6])

(Schleifen Start / End Indizes für Iteration über Produktionen aus G_T definieren)
from = 1; to = $|G_T|$

(Aufruf der match-Funktion)
RT, PT = match (G, S, |S|, from, to, RT, PT, hole_positions)

(Berechne die fertige Ergebnistabelle)
RT_acc, _ = merge_results (G_T_top, G_T_start, RT, [])

(Gebe Ergebnis (an GUI) zurück - t_map wird verwendet, um das Ergebnis auf die ursprünglichen Nichterminalnamen zu mappen. RT und G_T_top wurden fürs Debuggen verwendet)
return (((t_map, RT_acc), (RT, PT)), G_T_top)

2.1.2 Match (Einfache Fälle 1 und 2a , Alg.4.1 [3])

Die match-Funktion startet den eigentlichen Algorithmus (Schmidt-Schauss[3], Alg. 4.1 S.282). Die Grammatik *G* im Aufruf ist die Vereinigungsgrammatik von G_S und G_T, wobei in der Liste der Produktionen, zuerst die von G_T, (topologisch aufgesteigend sortiert[7]) und danach die von G_S stehen. *S* ist das Nichterminal des Suchmusters, *next_ix* zeigt auf die nächste zu bearbeitende Produktion und *end_ix* ist ein Stopper für die letzte Produktion von G_T.

[6] In G sind haben alle Nichtterminale Integerbezeichner. S_map ist eine Funktion *alter_Name → neuer _Integername. Der ! Operator wird benutzt, da s_map eine 'Hashtable' ist.*
[7] *Topologisch aufsteigend = unabhängige Produktionen (Terminalproduktionen) zuerst.*

match (G, S, |S|, next_ix, end_ix, RT, PT, hole_positions)

(Fertig ? - Alle Nichterminale von G_T abgearbeitet)
Falls nextix > endix :
return (RT, PT) *(Fertig)*

T, rhs = G[next_ix] *(Hole nächste Produktion T → rhs)*

*(*** Alg. 4.1 Simple cases 1 ***)*
Falls |T| == 1:
Falls isPrev (G, T, S, hole_positions) : *(Teste ob T Präfix von S ist)*

PT' = add_table (PT, T, [Pf (1,T)]) *(Neuer Eintrag in PT [8])*
RT' = add_table (RT, T, [Rs 1]) *(Neuer Eintrag in RT)*

Falls |S| > 1 : *(1a)* *(Update PT)*
match (G, S, |S|, next_ix+1, end_ix, RT, PT', hole_positions)

sonst : (|S| == 1) *(1b)* *(Update RT, PT[9])*
match (G, S, |S|, next_ix+1, end_ix, RT', PT' , hole_positions)

sonst : *(Kein Update)*
match (G, S, |S|, next_ix+1, end_ix, RT, PT, hole_positions)

*(*** Alg. 4.1 Simple cases 2 ***)*
sonst : *(T → T_1 T_2)*
T_1 , T_2 = rhs
(Kopiere die Einträge der Präfixtabelle von T_2 mit offset $|T_1|$ in die Präfixtabelle von T)
PT' = copy_pt_n2_to_n (T_2, $|T_1|$, T, PT) *(2a)*

(Schleife über alle Einträge der Präfixtabelle von T_1 findet in match_complex statt)
G', RT'', PT'' = match_complex (G, S, T, T_1, T_2, PT[T_1]) RT, PT', hole_positions)

[8] *Das 1. Zeichen im Wort hat immer die Position 1 ! (und nicht den Index 0)*

[9] Der Fall für |S| = 1 und |T| > 1 fehlt, da z.B. für S = 1, T = 2 1 nicht erkannt wird, dass S Suffix von T ist. Weder Fall 2 noch die komplexen Fälle 1-5 können angewendet werden, da die Präfixtabellen von 1 und 2 leer sind. D.h. die Eigenschaft '*1 ist Suffix von 2 1*' kann durch die Komposition der Präfixtabellen von 1 und 2 nicht festgestellt werden. Man kann das Problem beheben, indem bevor der Alg. startet die Identität T ist Präfix von T in die Präfixtabelle geschrieben wird. Alternativ wird hier im Fall |S| = |T| = 1 neben dem Eintrag in die RT auch der Eintrag in PT geschrieben.

(Rekursiver Aufruf für nächstes T)
match (G', S, |S|, next_ix+1, end_ix, RT'', PT'', hole_positions)

2.1.3 Match_complex (Einfacher Fall 2b und Vorbereitung komplexe Fälle 1-5)

Schleife über alle Einträge der Präfixtabelle von T_1. PT_T_1 ist die Liste der Einträge der Präfixtabelle von T_1 ([3] Alg.4.1 2b).

match_complex (G, S, T,T_1, T_2, PT_T_1 RT, PT, hole_positions)

Falls PT_T_1 == [] : *(Alle Einträge von PT_T_1 sind abgearbeitet !)*
return (G , RT, PT)

x = PT_T_1[0] *(Hole nächsten Eintrag)*
xs = PT_T_1[1:] *(Alle restlichen Einträge ohne x)*

*(*** Alg. 4.1 Simple cases 2b ***)*
Falls x == Pf (a, A) : *(Der Suffix T_1 [a..] ist Präfix von S, A das Nichtterminal für den Suffix)*

G', A' = scfg_add_word ([A, T_2], G) *(Erzeuge die Produktion A' → A T_2)*
PT' = add_table (PT, T, [Pf (a,A')]) *(Neuer Eintrag in PT)*
RT' = add_table (RT, T, [Rs a]) *(Neuer Eintrag in RT)*
A'_Präfix_von_S = isPrev (G', A', S, hole_positions) *(A' Präfix von S ?)*
S_Präfix_von_A' = isPrev (G', S, A', hole_positions) *(S Präfix von A' ?)*

Falls S_Präfix_von_A' und (|S| != |A'|[10]) : *(Update RT)*
match_complex (G', S, T, T_1, T_2, xs, RT' ,PT, hole_positions)
sonst :
Falls A'_Präfix_von_S : *(Update PT)*
match_complex (G', S, T, T_1, T_2, xs, RT, PT' ,hole_positions)

[10] Die Bedingung |*S*| != |*A'*| impliziert val(S) != val(A') und ist erforderlich, da sonst triviale Einträge iin der Resulttable landen. Z.B. S=1, 3 → 1 2 würde den Eintrag Rs 1 in RT[3] schreiben.

sonst : *(Kein Update)*

match_complex (G, S, T, T_1, T_2, xs, RT, PT ,hole_positions)

(Bereite die Fallunterscheidung 1-5 vor, 'complex case ' Schmidt-Schauss[3] S.283 unten)

Falls x == Par (a, b, T_1') :

(1. Erzeuge ein Nichterminal P für die Periode T_1'[a..a+b-1])

G , P = getPre (G, T_1', b)

(2. Teile P in P_1 und P_2 auf, so daß T_1' = (P^{k-1}) P_1)

r = |T_1'| mod b

G[11], P_1 = getPre (G, T_1' , r)

G, P_2 = getSuf (G, T_1' , |T_1'| - r)

(3. Erzeuge P' → P_2 P_1)

Falls P_1 leer ist :

G, P' = G, P_2

sonst :

G, P' = scfg_add_word ([P_2, P_1], G)

(4. Intervall Halbierung für T_2 -

Finde die maximale Potenz von P', die ein Präfix von T_2 ist)

m = ceiling (|T_2| / b) *(Berechne m : |P^m| >= |T_2| und |P^{m-1}| < |T_2|)*

G, n2_ibs_pm =match_complex_intervall_start (G, P', T_2, m, hole_positions)

(5. Intervallhalbierung für S -

Finde das max. k, so daß S Präfix von P^k ist)

u = ceiling (|S| / b) *(Berechne u : |P^u| >= |S| und |p^{u-1}| < |S|)*

G , s_ibs_pm = match_complex_intervall_start (G, P, S, u, hole_positions)

(Berechne Parameter für den Aufruf der Fallunterscheidung für die komplexen Fälle 1-5)

(Berechne Nichterminal für T_1'T_2)

G, T_1'T_2 = scfg_add_word ([T_1', T_2], G)

v = floor (|T_1'T_2| / b)

lipl_T_1'T_2 = |T_1'T_2| mod b *(lipl[12])*

[11] Der Wert von *G* wird mit dem Ergebnis des Funktionsaufrufs überschrieben.

[12] Lipl = last incomplete period length

G, $T_1'T_2$_ibs_pm =match_complex_intervall_start (G, P, $T_1'T_2$, v, hole_positions)
st_pm = (S, |S| , T, |T|)
pt_pm = (a,b, RT, PT)
n_pm = (T_1, T_1',|T_1'|, T_2, |T_2|, $T_1'T_2$)
(Last incomplete period length (simply the rest modulo p))
lipl_pm = (u, lipl_$T_1'T_2$)
(Fallunterscheidung , siehe Schmidt-Schauss[3], S. 283)
match_complex_cases_1to5
(G, st_pm, pt_pm, n_pm, n2_ibs_pm, s_ibs_pm,
n1'n2_ibs_pm, lipl_pm, hole_positions)

2.1.4 Match_complex_intervall_start (Teste auf periodischen Präfix von T2)

Bestimme die maximale Ausdehnung eines periodischen Suffixes von T_1 in T_2 hinein. Teste ob T_2 periodisch mit P ist. Wenn nicht, dann bestimme mit der Funktion *match_complex_intervall* einen maximalen Präfix von T_2, d.h., die maximale Potenz von *P*, die ein Präfix von T_2 ist.

match_complex_intervall_start (G, P, T_2 , m, hole_positions)

(Teste ob T_2 periodisch mit P ist)

(a. Erzeuge eine Potenz von P mit Länge |P| >= |T_2|)
G', pow_p = scfg_add_power (G, P, m)

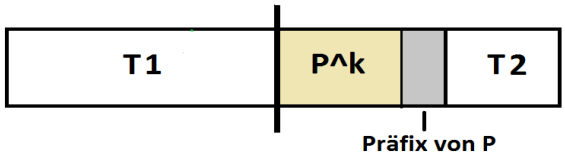
(b. Berechne Löcher für Potenz)
pow_holes = period_holes (hole_positions, |P|, m)

(c. Mache beide Nichterminale gleich lang - Hole Präfix der Potenz)
G'', pre_pow_p = getPre (G', pow_p, |T_2|)

(d. Teste Gleichheit)
T_2_ist_periodisch_mit_P = isPrev (G'', pre_pow_p, T_2, pow_holes)

Falls T_2_ist_periodisch_mit_P :
return (G', (m, pow_p, T_2)) *(T_2 ist periodisch mit P)*
sonst :

(Gibt es einen Präfix von T_2, der periodisch mit P ist ?)
(Bestimme die maximale Potenz von P^k, die ein Präfix von T_2 ist)
G3',(max_p_nt, max_p)) = match_complex_intervall (G, P, T_2, 1, m-1, 0, hole_positions)



(Abb. 5 maximaler Präfix)

Falls max_p > 0 : *(Kein Präfix von T_2 ist periodisch mit P)*
return (G, (0,0,0))

(Um den maximalen Präfix von T_2 zu bestimmen, der periodisch mit P ist
muss auch eine nicht vollständige Periode (oben grau) überprüft werden)

*Länge_pow_p = max_p * |P|*
Länge_Rest = |T_2| - Länge_pow_p

(Hole Nichtterminal für Rest von T_2)
G, Rest = getSuf (G3', T_2, Länge_Rest)

(Mappe Löcher auf Rest)
hole_positions' = shifted_holes (hole_positions, |P|)

(Bestimme den maximalen Präfix des Rests, der Präfix von P ist (oben grau))
G, (max_pre_nt, max_pre_len)) = max_prefix (G, Rest, P, hole_positions')

Falls max_pre_len > 0 :
(Erzeuge Nichterminal für maximalen Präfix von T_2 ,der periodisch mit P ist)
G, max_pre_n2 = scfg_add_word ([max_p_nt, max_pre_nt], G)

(Erzeuge ein Nichterminal für $P^{(max\_p + 1)}$)
G, max_p_nt' = scfg_add_word ([max_p_nt, P], G)

(T_2 hat einen Präfix der aus P^{max_p} und einem Präfix von P besteht)
return (G, (max_p + 1, max_p_nt', max_pre_n2))

sonst: (T_2 hat den Präfix P^{max_p})
return (G3' , (max_p, max_p_nt, max_p_nt))

2.1.5 Match_complex_intervall (Teste auf periodischen Präfix von T2)

Bestimme die maximale Ausdehnung eines periodischen Suffixes von T_1 in T_2 hinein. Bestimme dazu mit Intervallhalbierung (vergl. 2.6.3) die maximale Potenz von *P*, die ein Präfix von T_2 ist.

match_complex_intervall (G, P, T_2, a, b, maxpower, hole_positions)

(Teile Suchintervall in der Mitte)
c = a+ceiling ((b - a) / 2)

(Erzeuge Nichterminal P^c)
G' , pow_P = scfg_add_power (G, P, c)

(Berechne Löcher für Potenz)
pow_holes = period_holes (hole_positions, |P|, c)

(Teste ob P^c ein Präfix von T_2 ist)
P^c_Präfix_von_T_2 = isPrev (G', pow_p, T_2, pow_holes)

(Neuer bester Wert)
Falls P^c_Präfix_von_T_2 :
new_maxpower = c
sonst :
new_maxpower = maxpower

Falls a == b : *(Intervallgrenze links oder rechts erreicht ?)*
(Erzeuge Nichterminal für Potenz)
G, NT_für_maxpower = scfg_add_power (G, P, new_maxpower)
return (G , (NT_für_maxpower, new_maxpower))

sonst : *(Suche in neuem Suchintervall)*

Falls P^c_Präfix_von_T_2 :

(Suche Potenz im Intervall [c+1, b])

match_complex_intervall_b (G, P, T_2, c+1, b, new_maxpower, hole_positions)

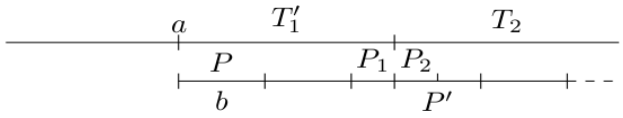
sonst :

(Suche Potenz im Intervall [a, c-1])

match_complex_intervall_b (G, P, T_2, a, c-1, new_maxpower hole_positions)

2.1.6 Match_complex_cases_1to5 (für einen Eintrag (a,b) der Präfixtabelle)

Nachdem die Parameter der Zerlegung für die 'komplexen Fälle' (siehe 2.4.3) berechnet worden sind wird das Update für die Präfix- und Ergebnistabelle damit berechnet. Es werden 5 Fälle unterschieden, die in Schmidt-Schauss[3] (S.283) dargestellt sind.



(Abb. 6 Zerlegung der Produktion $T \rightarrow T_1\ T_2$ für den Eintrag (a,b) in der Präfixtabelle von T_1[13])

Die Parameter der Funktion sind der Übersicht halber weggelassen. Bezeichner ergeben sich aus dem Zusammenhang bzw. aus der Abbildung oben. (siehe dazu auch die Haddock Dokumentation)

match_complex_cases_1to5 (G, S, T, T_2, hole_positions, PT, RT ...)

*Falls b * maxp_n2 >= $|T_2|$:* *(Komplexe Fälle 1-3 : T_2 ist periodisch with P')*

(Erzeuge Nichterminal für T_1' T_2)

G' , F = scfg_add_word ([T_1', T_2], G)

(Fall 1 : F ist Präfix von S)

Falls isPrev (G' , F, S, hole_positions) :

return (G' , RT, add_table (PT, T, [Par (a, b, F)])

(Fall 2)

*Falls b * maxp_s >= |S| :*

[13](Aus Schmidt-Schauss [3] (S.283 unten)

k = maxp_n1suf_n2 - maxp_s

RT' = add_table (RT, T, [Rar (a,b,k)])

*PT' = add_table (PT, T, [Par (a,b*k, S)])*

(lipl = last incomplete period length)

Falls lipl_s >= lipl_n1sufn2 :

return (G,RT', PT') *(Update RT, PT)*

sonst :

return (G, RT', PT)

(Fall 3)

sonst :

*Falls maxp_s > 0 und b * maxp_s < |S| :*

k = maxp_n1suf_n2 - maxp_s

*PT' = add_table (PT, T, [Par (a + (b * k) , b , maxp_s_nt)])*

return (G, RT, PT') *(Update PT)*

sonst :

(Alle anderen Fälle bei denen T_2 periodisch mit F= $T_1'T_2$ ist aber keiner der Fälle 1-3 passt)

return (G, RT, PT) *(Kein Update)*

(Fälle 4 und 5 - T_2 nicht periodisch mit P' aber ein Prefix von T_2 ist periodisch mit P')

sonst :

(Fall 4 - ähnlich wie Fall 2)

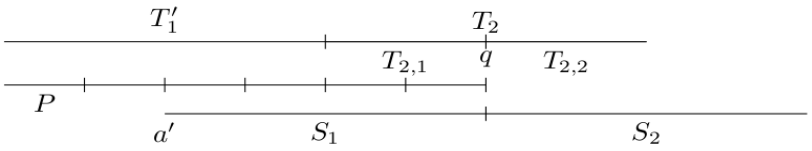
*Falls maxp_n2 > 0 und (b * maxp_n2 < $|T_2|$) :*

k = maxp_n1suf_n2 - maxp_s

RT' = add_table (RT, T, [Rar (a,b,k)])

return (G, RT', PT) *(Update RT)*

(Fall 5 - S kein Präfix von P^k)



sonst : *(Abb. 6a Zerlegung von S - Fall 5[14])*

*Falls maxp_s > 0 und b * maxp_s < |S| :*

[14] Aus [3,S.284]

(Zerlege S : S → S_1 S_2 (S_1 ist max. Präfix von S, der Präfix von P^k ist)
(S_1 liegt schon als maxp_s_pre vor)
S_1 = maxp_s_pre (Funktionsparameter)
(Berechne S_2)
G' , S_2 = getSuf (G, S, |S| - |S_1|)
(Zerlege T_2 : T_2 → T_{21} T_{22} (T_{21} ist max. Präfix von T_2, der Präfix von P'^k ist)
T_{21} = maxp_n2_pre (Funktionsparameter)
(Berechne T_{22})
G'',T_{22} = getSuf (G', T_2 , |T_2| - |T_{21}|)
q = a + |T_1'| + |T_{21}| -- test index
(S_2 Präfix von T_{22} ?)
S_2_ist_Präfix_von_T_{22} = isPrev (G'', S_2 , T_{22} , hole_positions)
(T_{22} Präfix von S_2 ?)
T_{22}_ist_Präfix_von_S_2 = isPrev (G'', T_{22} , S_2 , hole_positions)
(S_1 Suffix von F= T_1'T_2)
S_1_Suffix_von_F = isSufv (G'', S_1, F, hole_positions)

*Falls **nicht ((**S_2_ist_Präfix_von_T_{22} || T_{22}_ist_Präfix_von_S_2) und*
S_1_Suffix_von_F) :
(Keiner der Fälle 1-5 passt)
return (G, RT, PT) (Kein Update)

a' = q - |S_1|
(Ist S vollständig an der Position a' in T enthalten ?)
S_enthalten_in_T = isInfv (G'', a' , S, T, hole_positions)

(Berechne Suffix von T, der an der Position a' beginnt)
G''', suf_t_a' = getSuf (G'', T, |T| - a')

(Gibt es einen Präfix von S, der ein Präfix von suf_t_a' ist ?)
G'''', pfx , len_pfx = max_prefix (G''', S, suf_t_a', hole_positions)

Falls S_enthalten_in_T : (Update RT)
return (G, add_table (RT, T, [Rs a'], PT)
sonst :

Falls len_pfx > 0 : *(Update PT)*

return (G'''', RT, add_table (PT, T, [Pf (a', pfx)])

sonst :

(Weder S noch ein Präfix von S beginnen in T an der Position a')

return (G, RT, PT) *(Kein Update)*

2.2 Ausgabe der Ergebnisse

2.2.1 Komprimieren der Präfixtabelle (nicht implementiert)

Mit diesem Schritt wird die Präfixtabelle nach jedem Durchlauf der *match_complex* Funktion wieder auf polynomielle Grösse komprimiert. Die Grösse der Präfixtabelle wird in Schmidt-Schauss[3] mit $O(|G|^4)$ angegeben.

2.2.2 Zusammenfassen der Ergebnisstabellen

Der match-Algorithmus liefert als Ausgabe die Ergebnistabelle. Die Tabelle speichert für jedes Nichterminal der Zielgrammatik die Vorkommen des Suchstrings. Für Produktionen $A \rightarrow B\ C$ sind für A aber nur die Vorkommen des Suchstrings gespeichert, die in B beginnen und in C enden. Die anderen Vorkommen sind in den Ergebnistabellen für A und B gespeichert. Um zu einer vollständigen Darstellung der Ergebnisse zu kommen werden mit dem folgenden Algorithmus die Tabellen aus B und C mit der aus A vereinigt.

Algorithmus : (Merge Ergebnistabellen)

Die Knoten des Ableitungsbaums für den Zielstring sind die Nichtterminale in G_T. Die Blätter sind Nichtterminale, die ein Terminalsymbol ableiten, und das Startsymbol ist die Wurzel.

1. Durchlaufe den Ableitungsbaum von G_T in *preorder* Reihenfolge.
2. Für einen inneren Knoten X mit den Kindern A, B:

   X
   / \
   A B

   Vereinige die Ergebnistabelle von X mit denen von A und B (wobei die Einträge in B um die Länge von A korrigiert werden).
3. Für einen Terminalknoten : A → t, t ein Zeichen des Terminalalphabets von G. Tue nichts.

2.2.2.1 Implementierung (Merge Ergebnistabellen)

Sei G_T die SCFG des Zielstrings, *Next* ein Nichterminal aus G_T, RT die Ergebnistabelle aus dem match-Algorithmus und *visited* die Liste der bereits bearbeiteten Nichtterminale. Die Ergebnistabelle RT speichert alle Vorkommen des Suchtrings für jedes Nichterminal aus G ab. Unter RT[X] ist die Liste aller Vorkommen des Suchstrings im Nichterminal X gespeichert.

```
merge_results  (G, Next, RT, visited) :
    Falls Next in visited :
        return (RT, visited)                          (Knoten schon besucht)
    sonst :
        visited := visited ⋃ {Next}                   (Knoten als 'besucht' markieren)

        Falls Next die linke Seite einer Terminalproduktion ist     (Produktion Next → t)
            return (RT, visited)
        sonst:                                                      (Produktion Next → A B)

            c := Länge (A)
            (RT_left, visited) = merge_results (G, A, RT, visited)          (Linkes Kind)
            RT_left' = add_table (RT_left, Next, RT_left[A])                (Update f. l. Kind)
            (RT_right, visited) = merge_results (G, B, RT_left', visited)   (Rechtes Kind)
            add_table (RT_right, Next,
                    update_rt_entries (c, RT_right[B]), visited)            (Update f. r. Kind)
```

Die update_rt_entries Funktion korrigiert die Position des Vorkommens um die Länge des linken Kindes A nachdem es in der Produktion Next → A B auftritt.

```
update_rt_entries (Korrektur_Wert, [Vorkommen in Next]) :
    Für alle Vorkommen in Next:
        Vorkommen := Vorkommen + Korrektur_Wert
```

Die add_table überschreibt alte Einträge mit den neuen Einträgen der Kinder in der Ergebnistabelle.

```
add_table (RT, Next, Liste) :
    RT[Next] := Liste
```

2.2.2.2 Laufzeit von Merge

Der Ableitungsbaum von *G* wird in *preorder* Reihenfolge durchlaufen. Die Anzahl der besuchten Knoten ist gleich der Anzahl unterschiedlicher Knoten im Ableitungsbaum (|G||). Pro besuchtem inneren Knoten wird die *merge_results* Prozedur einmal aufgerufen. Die Laufzeit eines Aufrufs ergibt sich aus zwei

Aufrufen von add_*table* und einem Aufruf von *update_entries*, wobei die Laufzeit eines Aufrufs immer *O (poly(|G|)* ist, da nach der Komprimierung der Resulttatstabellen *(siehe Schritt Komprimieren der Ergebnistabellen)* die Grösse eines Eintrags in der Ergebnistabelle *(RT[i])* durch *O (poly |G|)* beschränkt ist. Dazu kommt noch die Abfrage ob der Knoten schon besucht wurde, die ebenfals *O(|G|)* Zeit kostet. Als Gesamtlaufeit ergibt sich für die *merge_results* Prozedur deshalb im schlechtesten Fall $O(|G|^6)$, da die Grösse der Ergebnisstabelle mit $O(|G^4|)$ (Schmidt-Schauss [3]) angegeben wird.

2.3 Wichtige Funktionen und deren Implementierung

Es sollen hier die wichtigsten Funktionen, die für die Implemtierung des Match-Algorithmus benutzt wurden vorgestellt werden. Die komplette Funktionsübersicht ist der Haddock-Dokumentation zu entnehmen. Dort wo Namen von Funktionen nicht mit dem Implementierungsnamen übereinstimmen wurde der Implementierungsname als Fussnote hinzugefügt.

2.3.1 Umwandlung von Lochproduktionen in Lochpositionen

Wenn in der Eingabe im GUI der Variablenmodus *Lochproduktionen* ausgewählt ist, dann werden daraus zuerst die Positionen der Löcher im entpackten Suchmuster berechnet. Diese sind die Eingabe für den *match*-Algorithmus.

Eingabe des Algorithmus :

1. Liste der aufsteigend topologisch sortierten Produktionen einer SLP-Grammatik G.
2. Liste der Lochproduktionen
3. Ein Nichtterminal *Z* aus *G* für das die Lochpositionen bestimmt werden sollen.

Ausgabe des Algorithmus :

Liste der Positionen der Löcher im Wort, das durch Z erzeugt wird.

Sei *A* der Ableitungsbaum der SLP-Grammatik *G* und *L* eine Lochproduktion, *Z* eine Nicht terminalproduktion aus *G* und *w* das von Z erzeugte Wort.

Algorithmus : (Bottom-up Parse)

1. Für alle Lochproduktionen *L*:
2. Berechne die Position von *L* in *w* durch Traversierung von A vom Blatt *L* bis zu Z. Die Position von *L* in *w* entspricht der Summe der Längen aller besuchten Nichtterminale, die rechte Kinder sind.

Beispiel : (Bottom-up Parse)

Ableitungsbaum	*Suchpfad in topologische Sortierung*
Z / \	5. Z → N3 K (Zielproduktion erreicht)
N3 K	4. N3 → Z N2 (N3 ist linkes Kind)
/ \	
Z N2	3. N2 → N1 Y (N2 ist rechtes Kind)
/ \	
N1 Y	2. N1 → L X (N1 ist linkes Kind)
/ \	
L X	1. L ist ein Loch

(Abb. 7 Berechnung Lochpositionen)

Berechnung der Pfadlänge

Ein Pfad von *L* nach *Z* entsteht durch max. |*G*| Ableitungsschritte von *L* . Der Algorithmus arbeitet bottom-up. Unabhängige Produktionen stehen in der topologischen Sortierung vor abhängigen Produktionen. D.h. bespielsweise wenn *X* in der Ableitung von *Y* vorkommt, dann steht *X* in der topologischen Sortierung vor *Y*. Auf diese Weise wird zunächst eine Produktion gesucht in der *L* auf der rechten Seite vorkommt. Wird *L* in der rechten Seite der Produktion *N1* gefunden und *N1* ist nicht *Z*, dann wird die Suche auf gleiche Weise für *N1* fortgesetzt. Die Position von L wird in jeder Produktion mittels der Länge des Nichtterminals aufsummiert, die in jedem Knoten gespeichert ist. Dabei wird 0 addiert, wenn das Vorkommen ein linkes Kind ist und die Länge des rechten Kindes addiert, wenn das Vorkommen ein rechtes Kind ist. Daraus folgt, dass die Position von L in Z := Summe (Längen der besuchten rechten Kinder) + 1 ist. (Die Additon am Ende um 1 definiert die 1. Position als 1)

Aufwand für den Algorithmus

1) *Topologische Sortierung und Berechnung der Wortlängen (SCFG)*

Mit der Funktion *wordLength im Modul Topsort* ist der Aufwand $O(|G| \log |G|)$

2) *Bottom-up Parse*

Jeder Parse besucht max. $\log_2 |G|$ Produktionen. Dabei werden immer max. 2 Symbole auf der rechten Seite einer Produktion angeschaut und jeweils eine, zwei oder keine Addtion ausgeführt. Daraus ergibt sich eine Laufzeit von höchstens $4 * (|G| * |G| * |Löcher|) = |Löcher| * O(|G|^2)$.

2.3.1.1 Implementierung des Alg. in 2.3.1 (liefert Lochpositionen)

Für alle Lochproduktionen p und das Nichtterminal Z der SCFG G:

getHolePositions (p, aufsteigend top. sortierte Produktionen von G , Z, 0)

getHolePositions (p, P= Liste der top. sortierten Produktionen von G, Z, PositionsSumme) :

Falls P = leer:

return 0 *(Das Ergebnis 0 heisst nicht gefunden)*

Falls p eine Terminalproduktion ist:

Falls p != P[0]:

return 0

sonst:

return 1

Falls P[0] eine Terminalproduktion ist: *(Verwerfe P[0])*

getHolePositions (p, P[1:], Z, Summe)

sonst: *(P[0] hat zwei Nichterminale p1,p2 auf der rechten Seite)*

Falls p == p1 & p = p2 & P[0] == Z : (Fertig - Position gefunden)

PositionsSumme + 1, PositionsSumme + Länge(p2) + 1

Falls p == p1 & P[0] == Z : *(Fertig - Position gefunden)*

Summe + 1

Falls p == p2 & P[0] == Z : *(Fertig - Position gefunden)*

Summe + Länge(p2) + 1

Falls p == p1 & p == p2:

getHolePositions z xs target sum (++[15])

getHolePositions z xs target (PositionsSumme + Länge(p2))) (++)

getHolePositions (p, P[1:], Z, PositionsSumme)

Falls p == p1:

getHolePositions (p, P[1:], Z, PositionsSumme) (++)

getHolePositions (p, P[1:], Z, PositionsSumme)

Falls p == p2:

getHolePositions (p, P[1:], Z, (PositionsSumme + Länge(p2)) (++)

getHolePositions (p, P[1:], Z, PositionsSumme)

Sonst: *(Verwerfe P[0])*

getHolePositions (p, P[1:], Z, PositionsSumme)

[15] *Der (++) Operator verknüpft die Ergebnisse wie folgt: a (++) b := [a,b]. D.h. a und b werden in einer Liste hintereinander geschrieben. Ausserdem gilt [a,b] (++) [c,d] := [a,b,c,d])*

2.3.2 Infixberechnung auf Nichtterminalen

Im Match-Algorithmus werden Funktionen für Präfix, Suffixe und Infixe auf Nichtterminalen benötigt. Der Algorithmus *InfixT* ist die Grundlage für alle Funktionen *getPre, getInf, getSuf, isInf, isPre, isPrev, isSufv, isInfv (siehe Haddock Dokumentation)*. Der Algorithmus erzeugt ein Nichterminal für einen beliebigen Infix eines gegebenen Nichtterminals *S* einer SCFG *G*. Der Algorithmus arbeitet in zwei Phasen. In der ersten Phase wird eine Prefixsuche für dem 'weggelassenen Teil' vor dem Beginn des Infix ausgeführt. Das liefert den Suchpfad für den Prefix im Ableitungsbaum. Im Ableitungsbaum von *S* sind Knoten als Tupel *(X, n)* gespeichert, wobei *X* ein Nichterminal und *n* die Länge des von ihm erzeugten Worts bezeichnet.

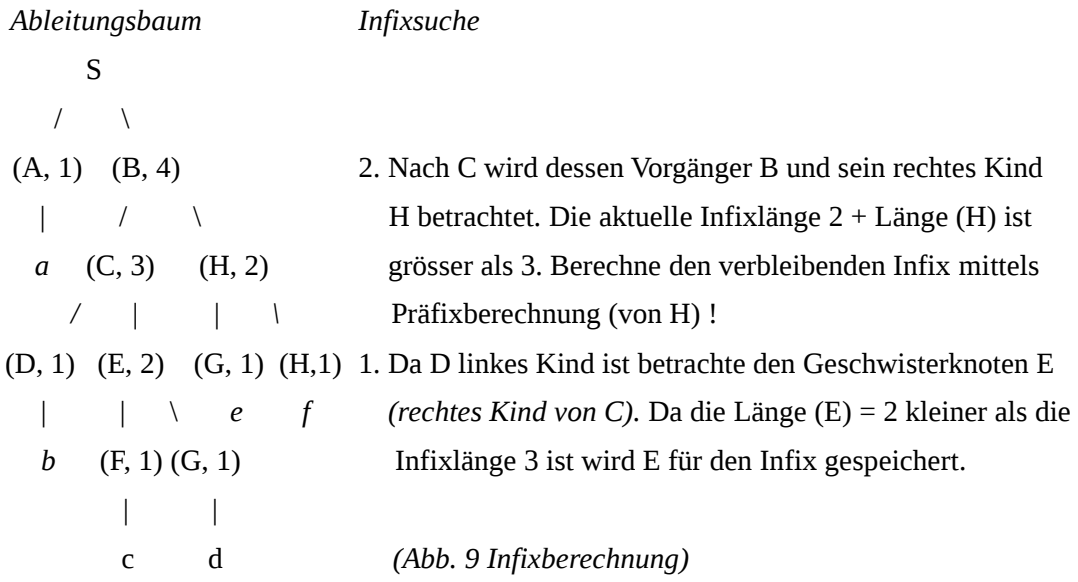
Beispiel für Infixberechnung S[3,5] = 'cde'
1. Berechne Suchpfad für den Prefix von S der Länge 2 [16] *(Suche Anfang von Infix im Ableitungsbaum)*

Ableitungsbaum	*Präfixsuchpfad*
S / \ (A, 1) (B, 4) | / \ *a* (C, 3) (H, 2) / | | \ (D, 1) (E, 2) (G, 1) (H,1) | | \ *e* *f* *b* (F, 1) (G, 1) | | *c* *d*	1. Die Suche beginnt an der Wurzel 2. Da die Länge von A (1) kleiner ist als 2 wird die Suche im im rechten Kind fortgesetzt und <u>A wird gespeichert.</u> 3. Da die Länge von A (1) + die Länge von B (4) grösser ist als 2 wird die Suche im linken Kind C fortgesetzt 4. Da die Länge von A (1) + die Länge von C (2) grösser als 2 ist wird die Suche im linken Kind D fortgesetzt und terminiert. (<u>D wird gespeichert)</u> *(Abb. 8 Präfixsuchpfad)*

Aus A und D lässt sich eine Produktion für den Präfix der Länge 2 *'ab'* erzeugen. Wichtiger ist jetzt aber der Suchpfad der besuchten Knoten S, *B*, *C* und *D* , der für die Infixsuche gleich verwendet wird. In der zweiten Phase wird der eigentliche Infix berechnet. Die Infixsuche beginnt an dem letzten besuchten Nichterminal der Präfixsuche (D) . Von dort aus steigt der Algorithmus nach oben im Prefixsuchpfad und betrachtet jeweils *rechte* Kinder. Die Suche beginnt in D, dem letzten besuchten Knoten der Präfixsuche.

[16] *Nebenbei werden die Nichterminale gemerkt, die man für die Erzeugung einer Produktion für den Präfix braucht.*

2. Berechne Infix S[3,5]



(Abb. 9 Infixberechnung)

Die Infixsuche beginnt letzten besuchten Knoten der Prefixsuche. Der Algorithmus läuft den Ableitungsbaum hinauf und sammelt rechte Kinder ein deren Länge in die verlangte Infixlänge passen. Wenn ein rechtes Kind gefunden wird, das nicht mehr in den Infix hineinpasst, weil es zu lang ist muss die Suche den Ableitungsbaum wieder hinunterlaufen. Der übrige Teil des Infix wird dann mit einer einfachen Prefixberechnung wie oben gezeigt erzeugt.

2.3.2.1 Implementierung Infixberechnung

Für den Infix S[a,b] wird zunächst der Prefix für den weggelassenen Teil S[0, a-1] berechnet (s. 2.6.2). Danach werden die Nichterminale gesammelt, die für den Infix notwendig sind, damit daraus später eine Produktion erzeugt werden kann. Schliesslich wird dann aus dieser Nichtterminalliste eine Grammatik erzeugt, die eine Produktion enthält, die den Infix erzeugt.

1. Berechne Suchpfad für den Prefix

Gegeben ist eine SCFG *G*, ein Nichtterminal *S* aus *G* und die Präfixlänge *n. Prefix_path* gibt die besuchten Knoten des Ableitungsbaums von S bei der Präfixsuche zurück.

[Besuchte Knoten der Präfixsuche] = prefix (G, S, a-1,[])

prefix_path[17] *(G, S, length_prefix , length_S, visited)*

Falls length_prefix == length_S oder length_prefix == 0 :
return visited
Falls length_prefix > length_S :
return error
sonst : *(Hole Produktion für S : S → A B)*
(A, length_a), (B, length_b) := G[S]
Falls (length_S - length_b) >= length_prefix :
(Rekursion linkes Kind : negatives Vorzeichen in -length_a bez. linkes Kind)
prefix_path (G, A, length_prefix, (length_S - length_b) (- length_a:[18] *visited)*
sonst :
(Rekursion rechtes Kind : positives Vorzeichen in length_b bez. rechtes Kind)
prefix_path (G, B, length_prefix, length_S (length_b:visited)

Der Algorithmus berechnet den Suchpfad von S (Wurzel) bis zum Nichterminal für das letzte Zeichen des Präfix von S im Ableitungsbaum.

2. Berechne Infix eines Nichterminals

Gegeben ist eine SCFG *G*, ein Nichtterminal *S* aus *G* und die Intervallgrenzen a,b für den Infix S[a,b]. InfixT gibt eine neue Grammatik *G'* und den Namen des Nichtterminals aus *G'* zurück, dass den Infix erzeugt.

Grammatik G' mit Nichterminal für Infix, InfixNichterminalname = infix (G, S, |S|, a, b)

infix[19] *(G, S, length_S, a, b)*
Falls a > b oder (b - a + 1) > length_S
oder a > length_S oder b > length_S oder length_S == 0 : *(Eingabefehler)*
return (G, error)

sonst:
(Suche Anfang von Infix im Ableitungsbaum)
visited = prefix_path (G, S, (a-1) length_S, [S])

[17] Implementiert als *prefixT2*
[18] Der : Operator ist defiinert als 1:[2] := [1,2]
[19] *Implementiert als infixT*

(Sammele bottom-up rechte Kinder ein, die in die Infixlänge passen - siehe 2.6.2)
[Gesammelte Nichterminale für Infix (unvollständig)], S' , length_infix_rest
= infix_get_nt_list (G, S, (b - a + 1), 0, visited, 0 , [])

Falls length_infix_rest > 0: (Berechne restlichen Infix als Präfix von S')[20]
G', InfixNichterminal = prefix
(G, S', length_S', length_infix_rest, reverse[21]*[Ges. Nichterminale f. Infix (unvoll.)])*

sonst: (Erzeuge gleich die Grammatik G' mit Produktion der InfixNichterminal)
G', InfixNichterminal =
scfg_add_word (G, [Gesammelte Nichterminale für Infix])

Sammele bottom-up rechte Kinder auf dem Präfixpfad auf, die in die Infixlänge passen (wie im Beispiel oben beschrieben). Die Funktion gibt die Liste der gesammelten Nichterminale für den Infix zurück, die jedoch unvollständig sein kann, falls der komplette Infix nicht gefunden wurde. In diesem Fall wird der zweite Rückgabeparameter, als Startknoten für die folgende Präfixsuche benutzt und der 3. Rückgabeparamter gibt die noch fehlende Länge des Infixes an.

infix_get_nt_list[22] *(G, S, length_infix, length_infix_sofar, visited, visited_next, infix_nonterminals)*
Falls length |visited| == 1 :
return (infix_nonterminals, (visited[0], length_infix))
sonst :
Falls if visited [visited_next] > 0 : (rechtes Kind)
infix_get_nt_list
(G,S, length_infix, length_infix_sofar, visited, (visited_next+1) , infix_nonterminals)

sonst : (Hole rechtes Kind von Elternknoten)
parent := abs (visited[visited_next + 1])
(Hole Produktion von parent : parent → A B)
_ , B = G[parent]
new_length = length_infix_sofar + length(B)

[20] *Der Aufruf von scfg_add_word ist in prefix enthalten.*
[21] *Die reverse Funktion ist definiert als revesese [1,2,3] := [3,2,1]*
[22] Implemeniert als *prefixT3*

```
Falls  new_length == length_infix :                    (Infix fertig !)
        return  (reverse (B:infix_nonterminals) ,(-1,0))

(Infix immer noch zu kurz nach Hinzufügen von rechtem Kind)
Falls  new_length < length_infix :
 infix_get_nt_list
  (G, S, length_infix, new_length, visited, (visited_next+1) (B:infix_nonterminals)

(Infix wird zu lang, wenn rechtes Kind hizugefügt wird)
sonst :
   return (reverse infix_nonterminals , (B , length_infix - length_infix_sofar))
        (Wende die prefix Funktion auf B an nachdem return, um die restlichen
        Nichterminale für den infix zu finden.)
```

Sammele Nichtterminale für Präfix, und gebe Grammatik mit Nichterminal für den Präfix zurück.[23] Der Algorithmus arbeitet genauso wie der *prefix_path* Algorithmus, gibt aber die Nichtterminale für den Präfix anstatt des Suchpfads zurück.

```
prefix[24] (G, S, length_S, length_prefix, coll_nonterminals)
    Falls length_prefix == length_S && (length coll_nonterminals) == 0 :
        return (G, S)
   Falls length_prefix == length_S && (length coll_nonterminals) > 0 :
        return  scfg_add_word (G, reverse(S:coll_nonterminals))

   Sonst :                                          (Hole Produktion für S : S → A B)
        (A, length_a), (B, length_b) := G[S]
        Falls  (length_S - length_b) >= length_prefix :
            (Rekursion linkes Kind : Verwerfe rechtes Kind und reduziere Länge)
            prefix (G, A, length_prefix, (length_S - length_b),  coll_nonterminals)
        sonst :
            (Rekursion rechtes Kind : Sammele linkes Kind A)
            prefix (G, B, length_prefix, length_S,  reverse(A:coll_nonterminals))
```

[23] *Scfg_add_word* wird direkt aufgerufen.
[24] Implementiert als *prefixT*

2.3.3 Hinzufügen von Produktionen zu einer SCFG

Der Match-Algorithmus braucht Funktionen mit denen sich neue Worte zur Grammatik hinzufügen lassen. Worte sind Präfixe, Suffixe, Infixe schon existierender Nichtterminale in G oder auch Potenzen[25] von solchen. Ein Wort entspricht in der Grammatik einem Nichterminal (Produktion), das das Wort erzeugt. Da die Produktionen der SCFG nur von der Form $S \rightarrow A\ B$ oder $S \rightarrow t$ sein dürfen ist es im Regelfall notwending für ein Wort mehrere Produktionen hinzufügen. Das neue Wort wird durch eine Liste von Nichtterminalen beschrieben, die nacheinander entpackt das neue Wort ergeben sollen.

Beispielgrammatik (1 erzeugt das Wort abbbb)

1 → 4 2	Es soll ein Nichtterminal für das Wort *abbb* hinzugefügt werden,
2 → 3 3	dass durch die Nichtterminale 4,3 und 5 beschrieben ist. Es werden
3 → 5 5	solange zwei Nichterminale zu einem neuen zusammengefasst bis nur
4 → a	noch ein einziges übrig ist. D.h. erzeuge zuerst die neue Produktion
5 → b	6 → *4 3* und danach 7 → *6 5*.

In einem anderen Ansatz könnte man Produktionen in einem binären Baum anordnen und dann von den Blättern bis zur Wurzel immer paarweise zusammenfassen. Das braucht zwar diesellbe Anzahl Produktionen ist aber wesentlich schneller, wenn parallel gerechnet wird.

2.3.3.1 Implementierung SCFG Produktion hinzufügen

Die Funktion *scfg_add_word* hat als Eingabe eine Grammatik *G* und eine Liste von Nichtterminalen *Q*, die zusammengefasst werden sollen. Die Funktion liefert eine neue Grammatik *G'* und den Namen des Nichterminals *Z* zurück, dass die Konkatenation der Symbole aus *Q* erzeugt.

```
scfg_add_word (G, Nichterminal_liste)
    Falls die Nichterminal_liste == [Y,Y,..] :        (Die Nichtterminale sind alle gleich)
        Neue_Produktionen = powersk (head prods) n (length ntlist))
    sonst :
        (Füge rekursiv jeweils zwei Nichtterminale aus der Liste zusammen bis eins übrig ist)
        Neue_Produktionen = scfg_add_word3 (prods, [], [], n)

    Name_von_Z := |G| + | Neue_Produktionen |
    G' = add_productions (reverse Neue_Produktionen, G,  Name_von_Z)
    return (G',  Name_von_Z)
```

[25] *Für ein Nichtterminal S aus G ist die zweite Potenz S*S = SS und S*S*S = SSS, usw.*

Für das Hinzufügen eines Worts, dass aus n verschiedenen Nichterminalen zusammengesetzt werden soll werden n-1 Produktionen benötigt. Für das Hinzufügen einer beliebigen Potenz eines Nichterminals ist eine logarithmische Anzahl Produktionen ausreichend.

Die Funktion s*cfg_add_word3* erzeugt rekursiv aus zwei Produktionen eine neue Produktion bis nur noch eine Produktionen übrig ist. Das Ergebnis ist die Liste der neu erzeugten Produktionen.

scfg_add_word3 (Q, new_productions , result , counter)

Falls | Q | == 0 :

Falls Länge (new_productions) < 2 :

return = result

sonst : (Fasse die neu erzeugten Produktionen wieder zusammen !)

scfg_add_word3 (reverse new_productions, [], result, counter)

Falls | Q | == 1 : *(Nur noch eine Produktion X übrig)*

scfg_add_word3 ([], (X:new_productions), result, counter)

Falls | Q | > 1 : *(Hole die nächsten beiden Produktionen A, B, in*

C → A B *der Liste und erzeuge die Produktion C → A B)*

(Rekursiver Aufruf : speichere C im Ergebnis und in new_productions)

Q' := Q \ {A,B}

scfg_add_word3 Q' (C:new_productions) (C:result) (counter+1)

In der *result* Variable wird das Ergebnis gespeichert. Wenn die Liste von *Q* abgearbeitet ist müssen alle bisher neu erzeugten Produktionen (*C*) in *new_productions* wieder zusammengefasst werden. D.h. der Algorithmus beginnt von vorne mit *new_productions* als Q.

Hinzufügen einer Potenz S ^{m}

Wenn alle Nichterminale der Liste Q gleich sind werden $log_2 |Q|$ neue Produktionen gebraucht. Z.B. sei *G* die Grammatik, die aus nur einer Produktion $S \rightarrow a$ besteht und es soll die Produktion S^6 , die das Wort aaaaaa erzeugt zu *G* hinzugefügt werden. Der Algorithmus arbeitet in zwei Schritten. Im ersten Schritt werden 2er Potenzen für S erzeugt, die in k vorkommen. Z.B. ist 6 keine Zweierpotenz. Berechne S^2 und S^4 und konkateniere diese Potenzen im zweiten Schritt zu S^6.

Die Funktion *powersk* hat als Eingabe die Produktion *S*, die potenziert werden soll, eine Zählvariable *counter* für neue Produktionsnamen und den Exponent m. Das Ergebnis sind die Produktionen, die für das Erzeugen von S^m gebraucht werden.

```
powersk (S, counter, m)
    Falls m < 2 :
        return []

    k := floor (log2 m)
    (Berechne Produktionen für alle 2er Potenzen  S^2i  (2i <= m) )
    2er_Potenzen = powers2 (Länge(S), S, k, counter, [S])

    Falls m eine 2er Potenz ist :
        return   2er_Potenzen                    (Produktion für  S^m schon erzeugt)
    sonst :
        (Filtere genau die 2er Potenzen aus allen 2er Potenzen, deren Summe  2^m ist)
        pointer : = | 2er_Potenzen | - 1
        2er_Potenzen_für_m = filter_powers (m, pointer, 2er_Potenzen, [])

        (Erzeuge aus den Produktionen der 2er-Potenzen  S^m)
        2er_Potenz_Kombinationen := scfg_add_word3
                                        (2er_Potenzen_für_m, [], [], counter+pointer)
        return   (2er_Potenzen ++ 2er_Potenz_Kombinationen)
```

Im Ergebnis werden im Fall, dass m keine 2er-Potenz ist die Liste der 2er Potenzen S^{2i} *(2i <= m)* und der kombinierten 2er Potenzen, die beim Zusammenfassen von S^m entstanden sind zurückgegeben.

Die Funktion erhält ein Nichterminal *S* und die Länge von S. Das Ergebnis sind die Produktionen S^{2i} (*2i <= k)*, um daraus S^k zu erzeugen. Die Länge von *S* wird für die Generierung der neuen Produktion gebraucht, weil diese in jeder Produktion gespeichert ist.

```
powers2 (length_S, S , doubling_steps_left, counter, result)
    Falls doubling_steps_left == 0 :                              (Fertig)
        return reverse (result)
    sonst :
```

counter : = counter + 1 *(Freier Bezeichner für neues Nichtterminal)*
(Erzeuge neue Produktion, die S verdoppelt)
counter → S S *(Der Bezeichner für die linke Seite der Produktion ist eine Zahl)*
(Rekursiver Aufruf)
*powers2 (2*length_S, counter , doubling_steps_left -1, counter , counter:result)*

Bestimme 2er Potenzen, die zusammen die Produktion S^m ergeben. Z.B. lässt sich S^{14} aus S^2, S^4 und S^8 zusammensetzen. Die Funktion hat als Eingabe m und die Liste aller 2er Potenzen S^{2i} *(1 <= 2i < m)*. Die Funktion gibt die Liste der Produktionen mit der geforderten Eigenschaft zurück.

```
filter_powers (m, k, 2er_Potenz_Produktionen, result)
    Falls  m == 0 :
        return  result
    sonst :
        Falls  2^k > m :                          (Auslassen - Potenz zu gross)
            filter_powers m (ix-1) powers result
        sonst :                                   (Nehmen - Potenz passt in m)
            filter_powers (m- 2^k, k-1,
                2er_Potenz_Produktionen, 2er_Potenz_Produktionen[k]:result)
```

2.3.4 Bestimmung eines maximalen Präfix (mit Intervallhalbierung)

Die Funktion *max_prefix* bestimmt für zwei Nichterminale *A, B* einer SCFG *G* den maximalen Präfix von *A*, der auch ein Präfix von *B* ist, wobei *B* Löcher haben kann. Die Funktion gibt *G'* und *Z*, den Namen des Nichtterminals für den maximalen Präfix in G'.

max_prefix (G, A, B, 1, min (|A|,|B|), hole_positions_in_B, 0)

max_prefix[26] *(G, A, B, a, b, hole_positions, actual_length)*

(Teile Suchintervall in der Mitte)
c = a+ceiling ((b - a) / 2)

[26] Implementiert in *Match.max_prefix'*

```
(Hole Präfix von A mit Länge c)
G' , A' = getPre (G, A, c)

(Teste ob A' ein Präfix von B ist)
A'_Präfix_von_B = isPrev (G', A' B, hole_positions)

(Neue maximale Präfixlänge)
Falls  A'_Präfix_von_B :
        newlength = c
sonst :
        newlength = actual_length

Falls a == b :                          (Intervallgrenze links oder rechts erreicht ?)
        G, A' = getPre (G, A, newlength)
        return (G, (A' , newlength))
sonst :                                 (Suche in neuem Suchintervall)
        Falls A'_Präfix_von_B :
                (Suche weiter im Intervall [c+1, b])
                max_prefix (G, A, B, c+1, b, hole_positions, newlength)
        sonst :  (Suche weiter im Intervall [a, c-1])
                max_prefix (G, A, B, a, c-1, hole_positions,  newlength)
```

2.3.5 Prefix, Suffix und Infix Test auf Nichterminalen mit Löchern

Die Funktion *isPrevSufvInfv* wird für den *Präfix/Infix/Suffix* Test mit zwei Nichterminalen *A, B* der SCFG *G* benutzt. Dabei ist *A* Infix von *B* und *B* kann Löcher haben. Der Funktionstyp *(Präfix/Suffix/Infix)* wird als Parameter übergeben. Das Ergebnis ist , ob die jeweilige Eigenschaft für *A* erfüllt ist.

```
        isPrevSufvInfv (G, A, B, hole_positions, prefsufinf_function, 0)

isPrevSufvInfv (G, A, B, hole_positions, prefsufinf_function, offset)

Falls |A| > |B| :
        return False                                    (Fehler : immer falsch)

(Hole prefix/infix/suffix mit der Länge von A aus B)
```

G' , B' = prefsufinf_function (G, B, |A|)

(Umrechnung von Lochpositionen auf Präfix/Infix/Suffix Position,
(getPre, offset 0; getSuf, offset |B| - |A|, getInf, offset Startpos. - 1)

hole_positions' : = [x' | x' = x - offset, x' > 0, x in hole_positions]

(Berechne Lochfreie Intervalle (a,b))
subintervalls = getntsubs (hole_positions', |A|)

Falls subintervalls == [] : *(Prefix/Infix/Suffix*
return True *besteht nur aus Löchern)*

(Erzeuge Nichtterminale für (a,b) in A und B',
subnt_pairs = [(N1_A,N1_B),..] für alle lochfreien Intervalle in A und B'. Diese werden im nächsten Schritt auf Gleichheit getestet.)

G'' , subnt_pairs = makentsubs (G', A, B', subintervalls, [])

(Rufe den Plandowski-Algorithmus auf allen Paaren auf) *(Plandowski testet Gleicheit*
return plandowski (G'', subntpairs) *aller lochfreien Subintervalle)*

Der Plandowski - Algorithmus [5] wird benutzt um *lochfreie* Subintervalle der beiden Nichtterminale auf Gleichheit zu testen.

2.4 Laufzeit der Implementierung

Die Gesamtlaufzeit berechnet sich aus dem Aufwand für die aufgerufenen Funktionen für die Vorbereitung, Nachbereitung sowie dem Aufwand in der Hauptschleife über alle Nichtterminale von G_T.

Vorbereitung in 2.1.1 : *(Sei $G = G_S \cup G_T$)*

- Berechnung der absoluten Positionen von Löchern im Suchmuster : *in 2.3.1*
  O (|Löcher| * $|G_S|^2$)
- Umwandlung der Bezeichner der Nichtterminale in Integerzahlen : *SCFGUtils.gr_with_integer*
  $O(\log_2 |G|)$[27] * (|G|) *(SCFG.applySubst)*
- Topologische Sortierung von G_T : *Topsort.topSort*
  O ($|G_T| * \log_2 |G_T|$)
- Vereinigung der Grammatiken G_S und G_T : *SCFGUtils.scfg_append*
  O (|G|) * $O(\log_2 |G|)$ *(Lookup für alle Nichterminale)*
- Berechnen der unkomprimierten Längen der Nichtterminale : *Topsort.wordLength*
  O (($_|$G|) * $\log_2(|G|)|$) + |G| * ($_|$G|) * $\log_2(|G|)|$ *(Topsort + O (|G| * Insert/Lookup) für Zählen der NT)*

[27] *Die Lookup-Operation auf Data.Map.Map kostet $O(\log_2 |Map|)$*

- Initialisieren der Präfix und Resultat Tabellen : *Data.Map.Map.empty in* O (1)

Die Vorbereitung wird deshalb durch die Laufzeit O (|Löcher| * $|G_S|^2$) bestimmt.

Nachbereitung :

- Merge der Ergebnistabelle in 2.2.2 : O($|G|^6$)

Hauptschleife über alle Nichtterminale von G_T

Die Schleife wird $|G_T|$ mal durchlaufen. In jedem Schleifendurchlauf wird u.U. die Präfixtabelle des aktuellen T (PT[T]) upgedated. Dabei werden max. poly(|G|) neue Einträge erstellt, denn die Produktion $T \rightarrow T_1\ T_2$ durchläuft PT[T_1], die poly(|G|) viele Einträge haben kann. Die teuerste Operation in der Fallunterscheidung sind die Berechnung der komplexen Parameter in 2.1.3 und die Zerlegung von S in Fall 5 (2.1.6). In 2.1.3 werden folgende Funktionen aufgerufen:

- scfg_add_word n : *(n ist die Anzahl Nichterminale der Eingabe)*
  Im Besten Fall O (|G|) * *(log_2 |G|) * n (Potenz erzeugen)*
  Im schlechtesten Fall O (|G|) * *|G| * n (beliebiges Wort erzeugen)*
- add_table : *insert in Data.Map.Map in O (log_2 |G|)*
- isPrev :

*Die Funktionen isPre, isSuf, isInf, isPre, isSuf, isInf werden dominiert durch den Plandowski-Algorithmus. Die Laufzeit der Implementierung in Plandowski.plandowski kenne ich nicht, ist aber sicherlich O(poly(|G|). Für n[28] Löcher werden die n lochfreien Subintervalle getestet. Deswegen ist die Laufzeit O (|Löcher| * poly (|G|)).*

- getPre : *Alle getPre, getInf, getSuf Funktionen nutzen die Funktion für die Infixberechnung auf Nichterminalen (2.3.2). Dabei wird der Ableitungsbaum von G jeweils einmal topdown und einmal bottomup traversiert. Da die Tiefe eines Ableitungsbaums log_2 |G| im besten Fall und |G| im schlechtesten Fall ist wird der Aufwand dafür O (|G|) sein, da in einem Knoten des Baums nur eine lookup und eine Additionsoperationen ausgeführt wird.*

*Danach wird scfg_add_word auf mit der Eingabelänge O (|G|) ausgeführt[29]. Damit ist die Gesamtlaufzeit O(log_2 |G|) * (*O (|G|) * *(log_2 |G|)) = O (|G|) * *(log_2 |G|) im besten und O($|G|^2$) im schlechtesten Fall.*

- match_complex_intervall :

*Das Suchinverall [a,b] kann exponentielle Grösse haben in |G|. Es gibt logarithmisch viele Aufrufe, in denen eine Potenz p eines Worts erzeugt wird. Beim Erzeugen der Potenz werden log_2 p Produktionen erstellt. Der Aufwand dafür wird durch Map.insert der Produktionen dominiert und ist deswegen log_2 p * (log |G| * O(|G|). Der Gesantaufwand ist damit poly (|G|) * (log_2 p * (log_2 |G| * O(|G|)) = poly (|G|).*

[28] Die Anzahl der Zeichenvariablen (Löcher) wird in der Eingabe nicht beschränkt.
[29] Anzahl der Nichterminale aus denen das Wort gebildet wird.

- match_complex_intervall_start :

*Die Funktion testet ob T_2 periodisch mit P ist . Die Laufzeit wird durch den Aufruf von isPrev (Plandowski) dominiert und ist deswegen O (|Löcher| * poly (|G|)).*

- match_complex

*Die Laufzeit wird durch den Aufruf von isPrev (Plandowski) und ist deswegen O (|Löcher| * poly (|G|)).*

*Die dominierende Laufzeit für 2.1.6 (Match_complex_cases_1to5) wird ebenfalls durch den Aufruf von isPrev (Plandowski) und ist deswegen O (|Löcher| * poly (|G|))*

Daraus ergibt sich als Gesamtlaufzeit für die Hauptschleife :

Schleifendurchläufe * ((maximale Länge von Präfixtabelle[i]) * dominierende Laufzeit in 2.1.3) =

$|G_T| * (O(|G|^4)$ [30] * *$(|Löcher| * poly (|G|^{31}))) = |Löcher| * O (|G|^k)$, k mind. 6.*

Die Gesamtlaufzeit der Implementierung ist :

Vorbereitung + Hauptschleife + Nachbereitung =

$O (|Löcher| * |G_S|^2) + (|Löcher| * O (|G|^k)) + O(|G|^6)$, *(k >= 6)*

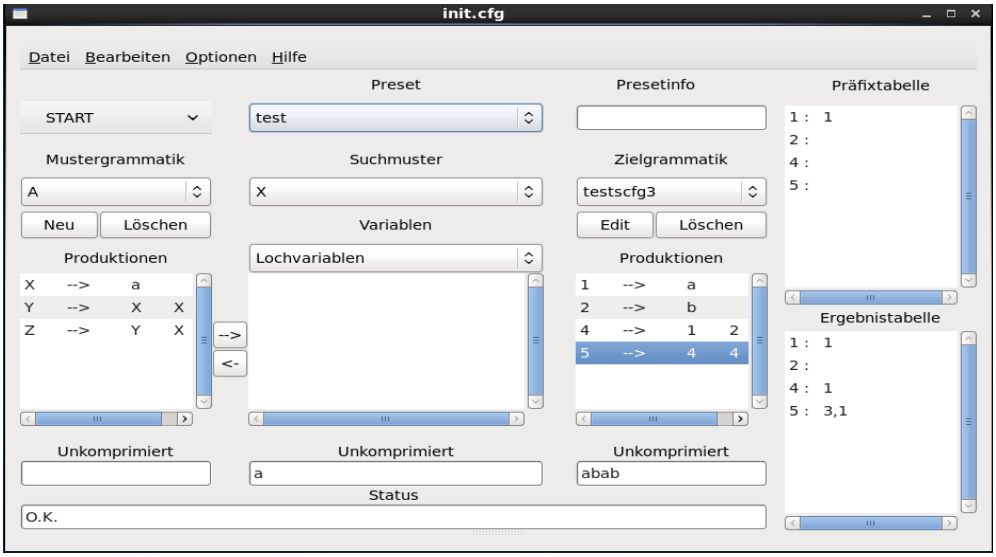
Wenn die Anzahl Zeichenvariablen = |Löcher| = O(|G|) ist dann wird eine polynomielle Laufzeit in der Grössenordnung $|G|^6$ erwartet.

3 Benutzeroberfläche (GUI)

Für die Benutzung des Programms steht eine graphische Benutzeroberfläche zur Verfügung mit der der *LCCSM*-Test gestartet werden kann. Das GUI verfügt ausserdem über Funktionen zum Erstellen von neuen und dem Importieren und Verwalten von vorhandenen SCFGs. Einzelne Tests lassen sich als Preset abspeichern und können inklusive der Grammatiken in einer Datei abspeichert werden.

[30] [3] S. 284, Proposition 4.4

[31] Die genaue Laufzeit der verwendeten Implementierung des Plandowski Alg. ist nicht bekannt



(Abb. 10 Screenshot Editor)

3.1 Importieren von SCFGs

Über das Menü 'Datei → Importiere Grammatik' lassen sich SCFGs importieren, die in einer externen Haskelldatei gespeichert sind . Es lassen sich Grammatiken mit dem Typ *SCFG Char b*, (*b = String, Char, Integer* oder *Int*) importieren . Nach dem Auswählen der Quelldatei, des SCFG-Typs und der Eingabe des Funktionaufrufs der den ausgewählten Ergebnistyp hat, wird die Grammatik unter dem angegeben Namen importiert und steht danach als Muster- oder Zielgrammatik zur Verfügung.

3.2 Erstellen von SCFGs

Im Menü *'Bearbeiten → Neue Grammatik'* lassen sich neue leere SCFGs definieren. Eine Grammatik kann später sowohl als Muster- als auch als Zielgrammatik verwendet werden.

3.3 Neue Produktion hinzufügen

Mit der Schaltfläche *'Neu'* lassen sich neue Produktionen zu einer Grammatik hinzufügen. Als Bezeichner für Nichterminale sind dabei nur Worte zulässig, die aus Ziffern und Buchstaben bestehen. Gleiches gilt für Terminale, die zusätzlich nur ein Zeichen lang sein dürfen. Nach der Def. von SCFGs sind alle Produktionen nichtrekursiv und sind entweder von der Form T → T_1 T_2 oder T → *t*, wobei T ein Nichterminal und t ein Terminal ist.

3.4 Produktionen löschen

Das Löschen von Produktionen ist über das Markieren von Produktionen und die Schaltfläche *'Löschen*' möglich. Es ist auch möglich mehrer Produktionen auf einmal zu löschen. Wenn durch das Löschen einer Produktion eine unvollständige Grammatik entstehen würde wird der Löschbefehl nicht ausgeführt.

3.5 Definition des Suchmusters

Das Suchmuster kann jede Produktion aus der Mustergrammatik sein. Dieses lässt sich über den Auswahlknopf in der Mitte auswählen, in dem alle Nichterminale der Mustergrammatik aufgelistet sind.

3.6 Definition von wildcards im Suchmuster

Im *LCCSM* Problem kann das Suchmuster Zeichenvariablen enthalten. Eine Zeichenvariable (vergl. Kap. 1.2) definiert eine Position im unkomprimierten Suchmuster, d.h. dem Wort, das durch es erzeugt wird, an der ein beliebiges Zeichen aus dem Terminalalphabet der Mustergrammatik stehen darf. Es ist also vergleichbar mit einer *wildcard* für ein Zeichen oder auch dem Punkt '.' in einem regulären Ausdruck. Es wird hier als kurz als *Loch* bezeichnet. Für die Definition von *Löchern* stehen zwei Modi zur Verfügung.

3.6.1 Definition von Löchern über Terminalproduktionen der Mustergrammatik

Im Modus '*Lochproduktionen*' können Terminalproduktionen der Mustergrammatik über die Schaltfläche '→' ausgewählt werden. Jede dieser Produktionen kann dann jedes Zeichen aus dem Terminalalphabet der Mustergrammatik produzieren. Es wird in der unkomprimierten Darstellung des Suchmusters als '_' dargestellt.

3.6.2 Definition von Löchern über Positionen in dem vom Muster erzeugten Wort

Im Modus 'Lochpostitionen' werden die wildcard Positionen direkt über die Stelle an der sie im unkomprimierten Wort vorkommen sollen definiert. Ein 'Rechts-Click' im Fenster für Lochpositionen öffnet den Dialog in dem die Positionen als Integerwerte > 0 durch Leerzeichen getrennt eingegeben werden können. Zu grosse Werte für Positionen, die nicht vorhanden sind werden automatisch aus der Liste entfernt (bzw. nicht angewendet).

3.7 Starten des Algorithmus

Vor dem Start mittels *'Start-Knopfs'* müssen Mustergrammatik, Zielgrammatik und das Suchmuster ausgewählt sein. Der Algorithmus startet ausserdem automatisch nach dem Laden eines neuen Presets oder nach dem Ändern des Suchmusters.

3.8 Ausgabe der Ergebnisse eines Durchlaufs

Nach der Terminierung des Algorithmus werden die Ergebnisse in Form der *Präfix- und Ergebnistabelle* im den jeweiligen Fenstern dargestellt. Die *Präfixtabelle* zeigt für jedes Nichtterminal der Zielgrammatik, welche Präfixe des Musters Suffix des jeweiligen Nichterminals sind. Die *Ergebnistabelle* zeigt für jedes Nichterminal der Zielgrammatik an welchen Positionen das Suchmuster vollständig vorkommt.

3.9 Ausgabe des von einer Produktion erzeugen Wortes

In den Einträgen '*Unkomprimiert*' wird jeweils das von einer markierten Produktion erzeugte Wort angezeigt, wenn die Option '*Zeige unkomprimiert*' aktiv ist. Im Suchmuster werden dann alle wildcard Positionen als *'_'* dargestellt. Da die Laufzeit für das 'Entpacken' eines Nichtermianls exponentiell mit der Anzahl der Produktionen einer Grammatik wächst, sollte man die Funktion abschalten, wenn die Ausgabe der Oberfläche lange dauert . (Z.B. erzeugt die Grammatik: P0 → 'a', Pi → Pi-1 Pi-1, (1 <= i <= n) das Wort, dass nur aus 'a' besteht und die Länge 2^n hat. [32]

3.10 Statusleiste

In der Statusleiste am unteren Rand des GUI wird der Zustand der ausgewählten Grammatiken und Fehlermeldungen die während des Laufs auftreten angezeigt.

3.10.1 Fehler : Mehrere Startsymbole

Um den Fehler zu beheben sollte man versuchen Produktionen zu löschen oder durch Hinzufügen von Produktionen eine Grammatik zu erzeugen, die genau ein Startsymbol hat.

3.10.2 Fehler : Unbenutzes Terminalsymbol

Unbenutzte Terminalsymbole sind in einer SCFG nicht erlaubt. Solche Produktionen müssen gelöscht werden bevor der Test gestartet werden kann.

[32] Die Funktion zum Berechenen von val(T) stand bereits im Modul *SCFG* zur Verfügung.

3.11 Implementierung des GUI

Für die Implementierung der Oberfläche wurden die Haskell-Bindings Gtk2Hs[11] des GTK Toolkit benutzt. Mit dem Glade Tool[12] lassen sich GUIs mit einem Editor erstellen, die als XML - Datei in verschiedenen Sprachen wie z.B. C, Python, Java und auch Haskell verwendet werden können. Die Layoutdefinitionen, die als XML gespeichert sind, können dann mit Hilfe der glade-Bibliothek in ein Haskellmodul importiert werden und stehen dem Programmierer als Referenzen zur Verfügung. Der Vorteil dieser Vorgehensweise ist natürlich der Zeitvorteil bei der Erstellung. Andererseits ist es möglich ein erstelltes GUI unabhängig vom Programmcode später weiterzuentwickeln. Die Portierbarkeit der Oberfläche zu anderen Sprachen ist ein weiterer Vorteil. Dennoch bringt die Entwicklung von Gui's in Haskell ein paar Hindernisse mit sich, die es nicht so populär erscheinen lassen. Da es in Haskell keine globalen Variablen gibt muss man die Referenzen der Objekte quasi jeder Funktion mitübergeben. Das Speichern von Zuständen ist ein weiteres Problem, da es in Haskell keine veränderliche '*Variablen*' gibt, denn es werden nur Ausdrücke / Funktionen ausgewertet. Da alle gtk2hs Gui-Funktionen den Rückgabewert *IO()* haben wurde die Speicherung der Zustände (Presets) mit einer *IORef* Referenz verwaltet. Die ConfigParser Bibliothek, die an die gleichnamige Python-Implementierung angelehnt ist bietet eine einfache Alternative zur Speicherung der GUI-Daten (Grammatikinformationen) im XML-Format, da Funktionen zum Schreiben bzw. Lesen schon vorhanden sind und keine extra XML-Parser Routinen erstellt werden müssen.

4 Zusammenfassung

Diese Arbeit stellt eine Implementierung für den Algorithmus von Schmidt-Schauss[3, Alg.4.1] für das LCCSM Problem vor, die polynomielle Zeit in |G| benötigt , wenn die Anzahl der Zeichenvariablen durch O (|G|) beschränkt ist. Es wird eine graphische Benuzteroberfläche zur Verfügung gestellt mit der SCFG Grammatiken erstellt und der Test aufgerufen werden kann. Die Haskell Module SCFGUtils und MatchUtlis für Operationen auf SCFGs sind auch unabhängig von der Implenentierung nutzbar.

4.1 Ausblick

Die Implementierung lässt sich noch optimieren. Das betrifft vor allem die rekursive Implementierung von Funktionen, die wegen der begrenzten Rekursionstiefe zu Abstürzen führen kann, wenn der Stack überläuft. Das hängt damit zusammen, dass Einträge der Präfixtabelle polynomielle Länge in |G| haben können. Die Implementierung der komplexen Fälle (Kap. 2.1.6) ist ausserdem unübersichtlich und müsste vereinfacht werden. Optimiert werden könnte auch das Erzeugen von Potenzen für Präfixtests etc. indem man erzeugte Potenzen in einer eigenen Tabelle abspeichert und so eine Neuberechnung vermeidet.

5 Anhang

5.1 Fine und Wilf's Theorem [1]

Ein Wort mit Perioden p und q, und p + q - ggt(p,q) <= Länge(Wort) hat auch die Periode ggt(p,q).

Beweis:
Das Wort *w* habe die Perioden p und q und m = ggt(p,q).
Wegen x ≡ 0 mod (m) und y ≡ 0 mod (m) ⇨ x ≡ y mod (m) .
Weiter sei *index* die Funktion, die für die Eingabe 0 <= i <= Länge(w) - 1, das Zeichen an der Stelle i von *w* zurückgibt.

Da w die Perioden p und q besitzt gilt: (Voraussetzung)

x ≡ y mod (p) ⇨ index(x) = index(y) und
x ≡ y mod (q) ⇨ index(x) = index(y)

Zu Zeigen:
x ≡ y (m) ⇨ index(x) = index(y)

O.b.d.A sei p = l * m, l ∈ ℕ , wegen m | p,

x ≡ y mod (p) ⇔ x = k (l*m) + y, für ein k ∈ ℕ
⇔ x = kl (m) + y
⇔ x ≡ y mod (m).

Auf die gleiche Weise geht man für q vor. Daraus folgt:
x ≡ y mod (m) ⇨ index(x) = index(y) , d.h. w hat auch die Periode m.

Die Zusatzvoraussetzung p + q - ggt(p,q) <= Länge(w) muss gelten wie ein Gegenbeispiel zeigt. Das Wort *w* der Länge 8 habe die Perioden 5 und 3. Aus dem Satz folgt, dass *w* auch die Periode 1 hat.

Aufbau von w

`i)01234567`	Die Buchstaben an den Positionen 0,3,5 und 6
`aaaa`	müssen alle gleich sein.

`ii)01234567`	1. Wegen 5 ≡ 2 mod (3) ist index(2) = a
`aaaaaa`	2. Wegen 7 ≡ 2 mod (5) ist index(7) = a
`iii)01234567`	1. Wegen 7 ≡ 1 mod (3) ist index(1) = a
`aaaaaaaa`	2. Wegen 4 ≡ 1 mod (3) ist index(4) = a

Wenn man jetzt die Länge von w auf 7 verkürzt lässt sich die Implikation in ii) 2. nicht mehr anwenden. Daraus folgt:

`ii)01234567`	1. Wegen 5 ≡ 2 mod (3) ist index(2) = a
`aaaaa`	
`iii)01234567`	Es bleibt nur die Bedingung index(1) = index(4)
`abaabaa`	wegen der Periode 3 übrig.

D.h. wenn die Längenbedingung verletzt wird gilt das Theorem nicht mehr.

5.2 Erweiterung von Fine und Wilf's Theorem für Worte mit genau einem Loch

Ein Wort mit lokalen Perioden p und q hat die Periode ggt(p,q), wenn p + q <= Länge(Wort) ist.

Lemma 3.1 [3] sagt etwas darüber aus, wie ein p-periodisches Wort mit einem Loch entsteht. Dabei gibt es zwei Fälle. (w_1, w_2 jeweils p-periodisch)

a) Sei w_1=ababab w_2=cbcbcb
w= ababab_bcbcb ⇨ w ist 2-lokal periodisch

b) Sei w_1=ababab w_2=ababab
w= ababab_babab ⇨ w ist 2-periodisch
Nur dieser Fall erzeugt ein p-periodisches Wort.

5.3 Module der Implementierung

Die Implementierung umfasst die Haskellmodule *Main*, *Match*, *MatchUtils*, *SCFGUtils* und *Test*.

Main	Gui Implementierung
Match	Implementierung des Algorithmus 4.1[3] und komplexe Funktionen zur Bestimmung *maximaler Präfixe/In/Suffixe* von SCFG komprimierten Worten.
MatchUtils	Präfix/Infix/Suffix Operationen auf SCFG komprimierten Worten (mit und ohne Löcher)
SCFGUtils	Funktionen auf SCFG's wie z.B. Vereinigung und Hinzufügen von Produktionen
Test	Testfunktionen

Die Module *SCFG*, *Topsort*, *Plandowski* standen vor der Implenentierung bereits zur Verfügung und wurden unverändert übernommen[33].

SCFG	Datentypen für Muster- und Zielgrammatik und val - Funktion .
Topsort	Algorithmus zum Bestimmen einer topologischen Sortierung
Plandowski	Implementierung des Plandowski-Algorithmus [4], der für zwei Nichterminale *A, B* einer *SCFG G* in polynomieller Zeit abh. von |G| testet, ob die von A und B erzeugten Worte val(A) und val(B) gleich sind.

Für Details sei auf die Haddock-Dokumentation verwiesen.

[33] (Im Modul SCFG wurde der Datentyp *Prod* zusätzlich von *Typeable* abgeleitet. Das war für den Grammatik-import mit Language.Haskell.Interpreter notwendig. Das Modul lässt sich mit der Option -XDeriveDataTypeable des GHC Compilers compilieren.)

5.4 Entwicklungsdokumentation

Für die Entwicklung der Haskell Module wurde folgende Software verwendet :

- GHC 7.2 [10] (Compiler)
- Glade 3.6.7 [12] (Oberflächenentwicklung)
- SciTE [13] (Texteditor, Entwicklungstools)
- CentOS 6.4 (Betriebssystem)

Zusätzliche Haskell Bibliotheken:
glade-0.12.1, gtk-0.12.4, ConfigFile-1.1.1

Die Ausarbeitung wurde mit LibreOffice 3.4.5 und der Dokumentvorlage von Gerald Leppert (LGPL) http://templates.services.openoffice.org/de/template/vorlage-fur-eine-wissenschaftliche-0 erstellt.

5.5 Formatbeschreibung einer Editor - Konfigurationsdatei

Syntax der Konfigurationsdatei des Editor. In einer Konfigurationsdatei werden Presets und Grammatiken gespeichert.

[Default]
version : *Versionsname*

Programmoptionen
[OPTIONS]
preset: *Name des zuletzt geöffneten Presets*
show_uncompressed: *Option für Anzeige des unkomprimierten Wortes (True/False)*

Presetdefinition
[Pr_Presetname]
hole_positions: *Liste der Lochpositionen im Suchmuster (z.B. 1,2,4)*
hole_productions: *Namen der Nichtterminale die Löcher produzieren (z.B. A,C,D)*
name_gr_a: *Name der ausgewählten Mustergrammatik*
name_gr_b: *Name der ausgewählten Zielgrammatik*

pattern: Suchmuster - *Name des Nichtterminals (aus der Mustergrammatik)*
preset_info: *Beschreibung des Presets*
target: *Zuletzt markierte Produktion in der Zielgrammatik*
variable_mode: *Lochpositionen oder Lochvariablen*

Grammatikdefinition
[Gr_Beispiel]
x: a
y: b
4: X,Y
5: 4,4
Einträge für Produktionen haben die Form x:a oder x: Y Z
wobei *x,Y,Z* Nichtterminale sind und *a* ein Nichterminal bezeichnet.
(Das Nichtterminal der linken Seite einer Produktion wird immer in Kleinschrift dargestellt.)
Die Beispielgrammatik definiert die Produktionen:
$X \rightarrow a$, $Y \rightarrow b$, $4 \rightarrow X\ Y$, $5 \rightarrow 4\ 4$ und *5* erzeugt das Wor*t abab.*

6 Literaturverzeichnis

1 N.J.Fine und H.S. Wilf 'Uniqueness Theorem For Periodic Functions', Proc.Am.Math.Soc 16: 109-114 1965

2 Jean Berstel & Luc Boasson 'Partial words and a theorem of Fine and Wilf' Theo.Comp.Sci, 218(1):135-141 1999

3 Schmidt-Schauss, M. , 'Matching of Compressed Patterns with Character-Variables', 23rd International Conference on Rewriting Techniques and Applications (RTA'12) 272-287, Leibniz International Proceedings in Informatics (LIPIcs), Online http://drops.dagstuhl.de/opus/volltexte/2012/3498

4 Lohrey, Markus, 'Algorithmics on SLP-Compressed Strings: A Survey ', Groups Complexity Cryptology 4(2):241-299 (2012)

5 Plandowski, W. 'Testing equivalence of morphisms on context-free languages' ,In Proceedings of the 2nd Annual European Symposium on Algorithms, ESA 1994, Nummer 855 in Lecture Notes in Computer Science, S. 460–470. Springer, 1994.

6 Yury Lifshits , 'Processing Compressed Texts: A Tractability Border ' in CPM, Volume 4580 von Lecture Notes in Computer Science, Seite 228-240. Springer, (2007)

7 A. Jez. 'Faster fully compressed pattern matching by recompression'. CoRR, abs/1111.3244, 2011

8 J. Ziv und A. Lempel 'A universal algorithm for sequential data compression', IEEE transactions on Computers, pp:17:8 - 19 (1984)

9 F. Blanchet-Sadri. 'Algorithmic combinatorics on partial words.' Chapman & Hall/CRC, 2008

10 Haskell Programmiersprache - www.haskell.org

11 Gtk2Hs - Haskell Gui Bibliothek, http://projects.haskell.org/gtk2hs/

12 Glade GUI - Builder. http://glade.gnome.org

12 SciTE Texteditor u. Entwicklungstool - http://www.scintilla.org/SciTE.html