Das Internationale Phonetische Alphabet mit Haskell

Handreichung

Robert Max Polter
phi12grm@studserv.uni-leipzig.de
M.Sc Bioinformatik
Universität Leipzig

Inhaltsverzeichnis

1. Das internationale phonetische Alphabet und Haskell 1
2. Modularisierung 1
3. Der Datentyp IPA 1
    1. Varianten 1
    2. Typen 5
    3. Komplexität 5
4. Grenzen 6
5. Luftstrom 8
    1. Varianten 8
    2. rekursive Datenstruktur 8
6. Diakritika 9
    1. Implementation 9
        1. Varianten 9
        2. Symbolverzeichnisse mit Duplikaten 10
    2. Eigenschaften mit Standardwert 10
        1. Die Funktion assign 10
        2. Eigenschaftsabfragen mit fixem Standardwert 11
        3. Modifikation zugrundeliegender Eigenschaften 14
7. Laute 15
    1. Implementation 15
        1. Stimmhaftigkeit 16
        2. konsonantische Eigenschaften 16
        3. vokalische Eigenschaften 16
    2. Speicherstruktur und komplexe Symbole 17
8. Parser 17
    1. Vorwärtsparser 17
    2. Rückwärtsparser 20
9. Überblick: Benennungen 20
10. Ausblick 22
    1. fehlende Symbole 22
    2. Erweiterung 22
    3. metrischer Parser 22
    4. Leipziger Glossierungsregeln 22

1. Das internationale phonetische Alphabet und Haskell

Das Ziel dieses Moduls ist es das Internationale Phonetische Alphabet mit Haskell benutzbar zu machen, um damit computerlinguistische Anwendungen mit realsprachlichen Daten möglichst nah an der gängigen Wissenschaftspraxis umsetzen zu können. Als Maßstab dient hierbei die offizielle Tabelle der International Phonetic Association von 2018 (Version 2015: Kiel)[1].
Haskell ist eine funktionale Programmiersprache, die in der Computerlinguistik weitreichende Verwendung findet. Als Grundlage für das Verständnis seitens der Studentinnen, für die dieses Modul entwickelt wurde, diente Graham Huttons "Programming in Haskell"[2]. Diese Handreichung ist mit dem Ziel geschrieben, dass auch rudimentäres Wissen über Haskell ausreicht, um mit diesem Modul arbeiten zu können.

Dieses Modul ist im Rahmen des Bachelormoduls "Computerlinguistik" des Studiengangs Linguistik an der Universität Leipzig entstanden. Ziel der Veranstaltungen war unter anderem phonologische Akzeptoren und Transduktoren mit Haskell zu programmieren. Ich habe das vorliegende Modul konzipiert, um im Rahmen der Übungen mit realistischen Daten arbeiten zu können. Die implementierten Funktionen sind daher auf die Abfrage verschiedener Informationen ausgelegt. Der Mehrwert dieser Handreichung sollte daher sein, die Nutzerinnen zu befähigen, Modifikations- oder komplexe Abfragefunktionen selbst ergänzen zu können.[3]

2. Modularisierung

Dieses Modul besteht aus der Konstruktion des Kerndatentyps **IPA**, seiner Varianten, Spezifikationen und Bestandteile **IPA_data.hs**, dem Parserverzeichnis **IPA.hs**, einer Sammlung für dieses Modul erstellter nicht modulspezifischer, polymorpher Hilfsfunktionen **IPA_util.hs** und einer Sammlung von IPA-Beispieltexten **IPA_test.hs**. Zur Einbettung aller relevanten Funktionen in andere Module empfiehlt sich der Import des Moduls **IPA**, da sowohl alle Datenkonstruktoren und Parser, als auch polymorphe Funktionen von diesem Modul exportiert werden.
Die jeweiligen Funktionsweisen der einzelnen Datentypen und Funktionen sind im Folgenden erklärt.

3. Der Datentyp IPA

1. Varianten

Der IPA-Datentyp hat vier Konstruktoren, von denen in einem fehlerfreien Parse jedoch nur die ersten zwei auftreten.

```
data IPA = IPA [((Sound,Airstream), [Modif])] | IPAb Boundary | IPAs
((Sound,Airstream),[Modif]) | IPAm Modif | Unreadable Char
  deriving (Eq, Read, Ord)
```

Der Hauptkonstruktor IPA nimmt sich als Argument eine Liste Tupel bestehend aus einem Sound-Lufstrom-Paar (Abschnitt 7: Laute, Abschnitt 5: Luftstrom) und einer Liste Modifikatoren (Abschnitt 6). Die Implementation als Liste ermöglicht ein Verarbeiten von komplexen Lauteinheiten (z.B. Affrikate, Diphthonge) ohne direkte Unterscheidung auf der Ebene der Datenkonstruktoren. Dies ermöglicht das Erstellen uniform operierender Funktionen, die sowohl auf komplexe, als auch simple Laute zugreifen können.[4]

1 siehe https://upload.wikimedia.org/wikipedia/commons/8/8e/IPA_chart_2018.pdf
2 Hutton, G. (2016) "Programming in Haskell". Cambridge: Cambridge University Press.
3 Erweiterungen und neuere Versionen werde ich auf GitHub unter https://github.com/DannekAnnohen/IPA zugänglich machen, bei Fragen oder Anregungen bin ich unter phi12grm 'at' studserv.uni-leipzig.de zu erreichen

Komplexe Laute und Präartikulationseffekte werden mit einem Bogen zwischen beiden Symbolen gekennzeichnet. Triphtonge oder komplexere Konsonanten (z.B. Ejektiv-Kontour-Clicks) werden mit mehreren Bögen verbunden.

x	parseIPA x
ǂk͡xʼ / ǂǁ͡xʼ	[Complex Sound: (Voiceless PalatoAlveolar Click, Voiceless Velar Plosive),Voiceless Velar Fricative Ejective]
ǂk͡xʼ / ǂk͡ǁ͡xʼ	[Complex Sound: (Voiceless PalatoAlveolar Click, Voiceless Velar Plosive, Voiceless Velar Fricative Ejective)]

Darüber hinaus können über Bögen Präartikulationseffekte ausgezeichnet werden, die für den Standardparser sonst nicht von Postartikulationseffekten des Vorgängerlauts zu unterschieden wären. Diese Ambiguität ließe sich nur über das explizite Angeben von Silbengrenzen auflösen, was jedoch im Standardparser zugunsten der Benutzbarkeit vermieden wurde. (siehe hierzu Abschnitt 10.4: metrischer Parser).

Präartikulation im Kornischen, Wikipedia[5]

pɛᵈn -> pɛd͡n	mæᵇm -> mæb͡m	ˈhɛᵇmɐ -> hɛb͡mɐ
KOPF	MUTTER	DIESES

Postartikulation im Kantonesischen[6]

mᵇa	ŋᵍy
MUTTER	FISCH

Die meisten **IPA**s sind als Tupel von **Sound**-Information (Artikulationsart, -ort, Stimmhaftigkeit/ respektive Vokalqualität) und Luftstrom gespeichert[7] und werden während des Parsings mit ihren Modifikatoren zusammengesetzt. Um jedoch auch einzelne Symbole parsen zu können, werden die Konstruktoren **IPAs** (für Laute) und **IPAm** (Modifikatoren) verwendet. Der Konstruktor **IPAm** parst außerdem alleinstehende Modifikatoren, **IPAs** findet außschließlich in der Funktion *parseSym* Verwendung (siehe Abschnitt 8.1: Vorwärtsparser).

Silbengrenzen, Leerstellen, metrische Maßeinheiten und andere Trennungselemente werden mit dem Konstruktor **IPAb** gebildet, als Argumente werden hierbei Konstruktoren des Datentyps Boundary verwendet (Abschnitt 4: Grenzen).

4 Siehe hierzu Abschnitt 3.2: Komplexität für Funktionen zur Unterscheidung. Die Rückgabewerte aller Funktionen für komplexe Laute sind rekursive Varianten der jeweiligen Ausgabedatentypen (exemplarisch Abschnitt 5.2). Das Funktionsdesign erfolgt über die Funktion *assign*, die komplexitätssensitiv Operationen auf IPA-Instanzen vornehmen kann (Abschnitt 6.2.1)

5 https://en.wikipedia.org/wiki/Pre-stopped_consonant

6 Durvasula, K. "Understanding Nasality", PhD-Dissertation, University of Delaware

7 für komplexe Symbole wie 'ɚ' oder 'ɝ', welche sowohl Lautinformationen als auch Modifikatoren enthalten siehe Abschnitt 7.2: komplexe Symbole

```
readable :: IPA -> Bool
readable (Unreadable _) = False
readable _ = True

properIPA :: IPA -> Bool
properIPA (IPA _) = True
properIPA (IPAb _) = True
properIPA _ = False

unreadableChar :: IPA -> Maybe Char
unreadableChar (Unreadable c) = Just c
unreadableChar _ = Nothing
```

Für unlesbare Symbole, die nicht in den Verzeichnissen der IPA-Symbole zu finden sind wird der Konstruktor **Unreadable** verwendet, der zusätzlich das nicht interpretierte Symbol als Argument nimmt und somit eine Rekonstruktion über die Funktion *unreadableChar* erlaubt. Eine Liste aller interpretierbaren Symbole findet sich in *ipaSym*. Darüber hinaus können alle Instanzen von **IPA** mit der Funktion *readable* auf ihre Lesbarkeit hin überprüft werden. Eine allgemeinere Version von *readable* bietet die Funktion *properIPA*, welche nur vollwertige IPA-Sounds und Grenzen akzeptiert.

```
satisfy :: Foldable t => (a -> Bool) -> t a -> Bool
satisfy f ipal = not $ foldl (flip ((==) . f)) False ipal

elemIndexSatisfy :: (a -> Bool) -> [a] -> [Int]
elemIndexSatisfy f ipal = jumperate 0 $ map f ipal
  where jumperate _ [] = []
        jumperate n b = case elemIndex False b of
          Just i  -> [n+i] ++ jumperate (i+n+1) (drop (i+1) b)
          Nothing -> []
```

Allgemeine Tests über Listen von **IPA** lassen sich mit den Funktionen *satisfy* und *elemIndexSatisfy* aus dem Modul IPA_util.hs durchführen. Dabei verrechnet *satisfy* jedes Element der Liste nach einer Transformation mit der Argumentfunktion f rekursiv mit dem nächsten, die Liste wird somit auf sich selbst abgebildet/ gefaltet. Die Funktion *elemIndexSatisfy* gibt die Indices derjenigen Listenelemente zurück, die die Argumentfunktion f erfüllen. Für beide Funktionen müssen die Argumentfunktionen den Rückgabedatentyp **Bool** haben. So lässt sich beispielsweise schnell der Parsingerfolg eines Tests ermitteln und gegebenenfalls Problemregionen identifizieren.

input, x = parseIPA x	satisfy readable $ x	elemIndexSatisfy readable x	satisfy properIPA x	elemIndexSatisfy properIPA x
"aɪns ˈʃtʁɪtn̩ zɪç ˈnɔe̯tvɪnt ʊntˈz̥ɔnə | veːe̯ fɔn ˈiːnː ˈbaɪdn̩ voːl deːe̯ ˈʃteɐ̯kəʁə ˈvɛːʁə"	True	[]	True	[]
"aɪns ˈʃtʁɪtn̩ zɪç ˈnɔe̯tvɪnt ʊntˈz̥ɔnə | veːe̯ fɔn ˈiːnː ˈbaɪdn̩ **v#:l** deːe̯ ˈʃteɐ̯kəʁə ˈvɛːʁə"	False	[58]	False	[58, 59]
		input!![58] = '#'		input!![58] = '#' input!![59] = ':'

Aufgrund der Unübersichtlichkeit der inneren Struktur von IPA-Elementen in diesem Modul ist die Darstellung in zwei Darstellungsklassen vereinfacht worden: **Show** und die moduleigene Klasse **ShowIPA**, deren Methode analog zu *show* ist, jedoch über weitere Optimierungen verfügt. *showIPA* stellt dabei die linguistischen Bezeichnungen der Fachbegriffe natürlichsprachlich dar und findet modulseitig nur in der Darstellungsfunktion *present* Verwendung (Abschnitt 8: Parser). Die Funktion *show* behält die äußere Struktur der Lauteigenschafts-, Luftstrom- und Diakritikakonstruktoren bei um eine einheitliche Nutzungserfahrung zu gewährleisten.

```
class ShowIPA a where
  showIPA :: a -> [Char]

instance ShowIPA IPA where
  showIPA ipa@(IPA (((sound,airstream),modifs):xs)) | xs == [] = (showL showIPA
"" ", " " " " "" modifs')++pres sound
                                    | otherwise = showL show "Complex Sound: ("
", " ")" "" $ splitIPA ipa
    where
      modifs' = (\\) modifs $ (lateralM ++ voicingM ++ toneM)
      pres (C p m v) = (if modified $ voicing ipa then showIPA $ voicing ipa
else show v)++" "++showIPA p++" "++(if Lateral `elem` modifs then "Lateral "
else "")++ (if airstream == GlottalicEgressive then "Ejective " ++ showIPA m
else showIPA m ++ showIPA airstream)
      pres (V o f r) = (if modified $ voicing ipa then (showIPA $ voicing ipa) +
+ " " else "")++showIPA o++" "++showIPA f++" "++showIPA r++" Vowel"++(if tone
ipa /= NoTone then ", " ++ (showIPA $ tone ipa) else "")
  showIPA (IPAb boundary) = showIPA boundary
  showIPA (IPAm modif) = showIPA modif
  showIPA (IPAs ((sound, airstream),modif)) = "IPAs (("++showIPA sound ++", "++
show airstream++"), "++show modif++")"
  showIPA (Unreadable char) = "Unreadable " ++ show char


instance Show IPA where
  show ipa@(IPA (((sound,airstream),modifs):xs)) | xs == [] = (showL show "" ",
" " " "" modifs)++pres sound ++(if airstream == GlottalicEgressive then "
Ejective" else "")
                                                 | otherwise = showL show
"Complex Sound: (" ", " ")" "" $ splitIPA ipa
    where
      pres (C p m v) = show v++" "++show p++" "++ show m
      pres (V o f r) = show o++" "++show f++" "++show r++" Vowel"
  show (IPAb boundary) = "IPAb "++show boundary
  show (IPAm modif) = "IPAm "++show modif
  show (IPAs ((sound, airstream),modif)) = "IPAs (("++show sound ++", "++ show
airstream++"), "++show modif++")"
  show (Unreadable char) = "Unreadable " ++ show char
```

Komplexe Laute werden in beiden Klassenmethoden mit "*Complex Sound:* " dargestellt, gefolgt von den Lautbestandteilen als durch Kommata getrennte Auflistung in runden Klammern, eine **[a] -> [Char]** Transformation, die durch die IPA_util-Funktion *showL* ausgeführt wird.

Modifikatoren sind ebenfalls durch Kommata getrennt und stehen in *show* vor den Lauteigenschaften; *showIPA* positioniert davon abweichend etwaige Lateralität und Ejektivität vor der Artikulationsart (*((if Lateral `elem` modifs then "Lateral " else "")++ (if airstream == GlottalicEgressive then "Ejective " ++ showIPA m else showIPA m ++ showIPA airstream)*), instantiierten Ton hinter vokalischen Eigenschaften (*if tone ipa /= **NoTone** then ", " ++ (showIPA $ tone ipa) else ""*), Stimmhaftigkeit verrechnet mit konsonantischer Stimmhaftigkeit (*if modified $ voicing ipa then showIPA $ voicing ipa else show v*) und im Falle des Verlusts vokalischer Stimmhaftigkeit (*if modified $ voicing ipa then (showIPA $ voicing ipa*) unmittelbar vor der Lautinformation. Die Darstellung von unmodifizierten Lauteigenschaften erfolgt in der Reihenfolge Artikulationsort - Artikulationsart bzw. Öffnungsgrad - Frontalität - Rundung. Der Luftstrom wird explizit bei Ejektiven einerseits und Clicks und Implosiven andererseits (in *show* als Artikulationsart, bei *showIPA* als direkte Angabe des Luftstroms) angezeigt. Pulmonische Egressive werden als cross-linguistisch frequentester Fall nicht gesondert aufgeführt.

input	show $ parse1 input	showIPA $ parse1 input
ǁ	Lateral Voiceless Alveolar Click	Voiceless Alveolar Lateral Click
t̪ʼ	Dentalized Voiceless Alveolar Plosive Ejective	Dentalized Voiceless Alveolar Ejective Plosive
ʊ̞̃	LowRisingTone, Devoiced CloseCloseMid BackCentral Rounded Vowel	Devoiced Close-Close-Mid Back-Central Rounded Vowel, Low-Rising-Tone
t̬	Envoiced Voiceless Alveolar Plosive	Envoiced Alveolar Plosive
ŋᵍ	VelarVoicedPoststopped Voiceless Velar Fricative	Velar-Voiced-Poststopped Voiceless Velar Fricative
.	IPAb SyllableBoundary	SyllableBoundary

2. Typen

Alle IPA-Elemente können über die Funktion *typing* einer Unterkategorie zugeordnet werden, wodurch sie für verschiedene Arbeitsschritte leichter vorsortiert werden können. Affrikate und Diphthonge werden als **ComplexC** bzw **ComplexV** klassifiziert, wobei nicht spezifiziert sind, ob es sich nicht um Konsonantenkonstruktionen aus mehr als zwei Elementen oder Tri- (oder mehr) phthonge handelt. Gemischten Komplexen, insofern sie auftreten sollten, wird der Typenwert **MixedComplex** zugeschrieben. Beim Verwenden der Typfunktionen ist zu beachten, Präartikulationseffekte (insofern in den Daten enthalten) und komplexe Konsonanten nicht zu verwechseln, da sie gleichwertig interpretiert werden, wenn sie mit Bögen verbunden sind.

3. Komplexität

Mithilfe der Funktionen complexity und isComplexity lassen sich IPA-Elemente zwischen komplexen und simplen Lauten unterscheiden.

```
data Complexity = Simplex | Complex
  deriving (Eq, Show, Read)

complexity :: IPA -> Complexity
```

```
complexity (IPA ipa) = case length ipa of
  1 -> Simplex
  0 -> error "empty sound"
  _ -> Complex

isComplexity :: IPA -> Complexity -> Bool
isComplexity ipa c = (==) c $ complexity ipa
```

Besteht ein IPA-Element aus einem einzelnen Laut wird ihm der Wert **Simplex** zugeordnet, ansonsten **Complex**. Die Funktion *isComplexity* ermöglicht es darüber hinaus direkte Abfragen auf Komplexitätswerte lesbar zu implementieren.

input x = parseIPA input	map complexity x	x!!0 `isComplexity` Complex
"pfe͡ːɐd̥"	[Complex, Simplex, Simplex, Simplex]	True

Es ist auch möglich komplexe Laute in ihre Bestandteile zu zerlegen um diese einzeln zu betrachten oder modifizieren.

```
splitIPA :: IPA -> [IPA]
splitIPA (IPA []) = []
splitIPA (IPA (i:is)) = [IPA [i]] ++ splitIPA (IPA is)
  where jumperate _ [] = []
        jumperate n b = case elemIndex False b of
          Just i  -> [n+i] ++ jumperate (i+n+1) (drop (i+1) b)
          Nothing -> []
  where jumperate _ [] = []
        jumperate n b = case elemIndex False b of
          Just i  -> [n+i] ++ jumperate (i+n+1) (drop (i+1) b)
          Nothing -> []
```

Diese Funktion ist insbesondere deswegen notwendig, weil im gesamten Modul nie ein expliziter Unterschied zwischen komplexen und simplen Lauten gemacht wird. Jede Funktion ist ebenso auf simple wie komplexe IPA-Elemente anzuwenden (für Rückgabetypen komplexer Laute siehe Abschnitt 5.2: rekursive Datenstruktur)

4. Grenzen

Grenzen zwischen Wörtern, Silben, metrischen Füßen oder anderen phonologischen Einheiten werden als IPA-Elemente aus einer Kombination des **IPA**-Konstruktors **IPAb** und einem Element des Datentyps **Boundary** gebildet. Zusammen korrespondieren sie zu jeweils einem **Char**, eine Relation, die in den Verzeichnissen (**Map**) *bInventory* und *bInventory_inv* festgehalten ist.

```
bInventory = Map.fromList $ tuplify bSym bSpec
bInventory_inv = Map.fromList $ (flip tuplify) bSym bSpec

bSym = [' ', '.', '\8255', '\712', '\716', '|', '\8214', '\8599', '\8600']
bSpec = [IPAb Whitespace,
 IPAb SyllableBoundary,
```

```
 IPAb Undertie,
 IPAb PrimaryStress,
 IPAb SecondaryStress,
 IPAb MinorGroup,
 IPAb MajorGroup,
 IPAb GlobalRise,
 IPAb GlobalFall]
```

Bei einer **Map** handelt es sich um einen Datentyp, in dem je ein Schlüssel (key) mit einem Wert (value) assoziiert ist. Die Speicherung findet in einem geordneten, binären, balancierten Baum statt.
'Geordnet' bedeutet, dass jedes Element der Schlüsselmenge (im Fall von *bInventory* handelt es sich um **Char**s) im Baum so platziert ist, dass ihre linken Töchter allesamt einen geringeren Wert haben und ihre rechten einen höheren. Dies ist möglich, da alle Elemente des Datentyps **Char** zu einem Unicode-Wert korrespondieren.
'Balanciert' bezieht sich dabei auf die Struktureigenschaft des Speicherbaumes, dass alle Blätter gleich weit (maximaler Unterschied von 1) vom Wurzelknoten entfernt sind.

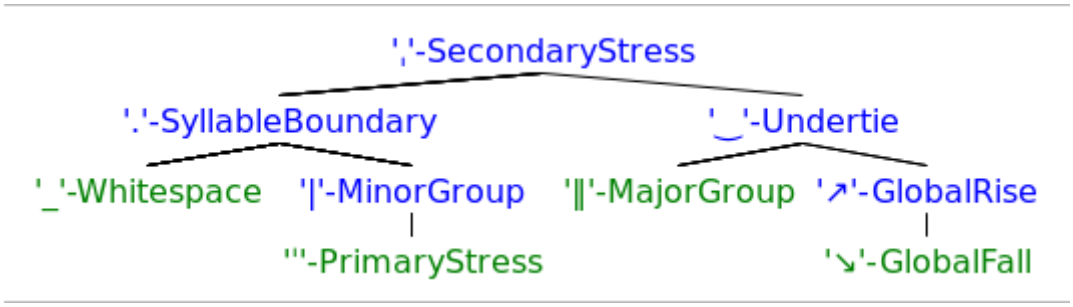

Abb. 1: Speicherstruktur **Boundary**

Beim Suchen nach einem **Char** in *bInventory* wird der Unicodewert des gesuchten Schlüssels mit dem des aktuellen Kontens (Beginn beim Wurzelknoten) verglichen. Aufgrund der Ordnung des Baumes kann der gesuchte Schlüssel, insofern im Verzeichnis enthalten, in sublinearer Zeit gefunden werden, da im Gegensatz zum Speichern in einer Liste nicht jedes Element überprüft werden muss (O(log n)).

Die **Map** *bInventory_inv* ist ein inverses Verzeichnis zum transformieren von **IPA** zu **Char**. Damit eine Speicherung als **Map** mit **IPAb Boundary**s als Schlüssel dennoch möglich ist, sind **IPA** und **Boundary** Instanzen der Klasse **Ord**, wodurch eine Größenordnung möglich ist. Dabei haben die zuerst genannten Konstruktoren einen jeweils geringeren Wert als die höheren; die Ordnung korrespondiert jedoch nicht zu der der jeweiligen **Char**s sondern orientiert sich grob an phonologischen Kriterien.

```
data Boundary = SyllableBoundary | Whitespace | Undertie | PrimaryStress |
  SecondaryStress | MinorGroup | MajorGroup | GlobalRise | GlobalFall
  deriving (Eq, Show, Read, Ord)
```

GHCI:

```
*IPA_data> SyllableBoundary < MajorGroup
True
```

```
*IPA_data> IPAb SyllableBoundary < IPAb MajorGroup
True
```

Um welche Art von Trennung von **IPA**s es sich handelt lässt sich mit *boundary* und *isBoundary* überprüfen. Es ist dabei jedoch zu beachten, dass diese Funktionen nur für **Boundary** definiert sind. Ob es sich bei einem **IPA** um eine phonologische Grenze handelt, kann mit *typing* oder *isTyping* (Abschnitt 3.2: Typen) abgefragt werden, um keine irreführenden Funktionsrückgabewerte zu produzieren.

5. Luftstrom

1. Varianten

Zur Spezifikation der Luftstsröme für **IPA** sind 4 Optionen als Konstruktoren von **Airstream** angelegt: pulmonisch-egressiv (**PulmonicEgressive**), glottalisch-egressiv/Ejektive (**GlottalicEgressive**), glottalisch-ingressiv/Implosive (**GlottalicIngressive**) und lingual-ingressiv/Clicks (**LingualIngressive**), von denen jedoch nur Ejektive explizit dargestellt werden. Dies liegt daran, dass 'Click' und 'Implosiv' bereits als Artikulationsart in der **Sound**-information spezifiziert sind und der pulmonisch-egressive Luftstrom als Standard nicht dargestellt wird. Die Darstellung innerhalb von **IPA** erfolgt über Instanzen der Klassen **Show** und **ShowIPA**, wobei beide aequivalente Transformationen bezüglich der Luftströme ausführen.

Airstream leitet darüber hinaus auch die Klasse **Ord** ab, um die in Abschnitt 4 beschriebene Speicherung von **Map**s zu ermöglichen. Dies ist notwendig, da **IPA** nur dann selbst **Ord** ableiten kann, wenn alle Bestandteile von **IPA** ebenfalls **Ord** instanziieren.

2. rekursive Datenstruktur

Simple **IPA**, welche mit dem Konstruktor **IPA** erstellt wurden, lassen sich mit den Funktionen *airstream* und *isAirstream* auf ihren Luftstrom hin untersuchen und liefern als Rückgabewert jeweils eine der vier in 5.1. beschriebenen Varianten zurück. Anders verhält es sich mit komplexen Lauten, für den die Funktionen als Rückgabewert den fünften Konstruktor **Air** gefolgt von einer Liste von simplen **Airstream**s haben. Innerhalb dieser Liste korrespondiert jeder Luftstrom zu einem Laut.

```
airstream :: IPA -> Airstream
airstream (IPA s) = case length $ air of
  0 -> error "call to function 'airstream' for empty sound"
  1 -> head air
  _ -> Air air
  where getAirstreams [] = []
        getAirstreams (((_,a),_):xs) = [a] ++ getAirstreams xs
        air = getAirstreams s
airstream _ = error "call to function 'airstream' for non-sound ipa"
```

So wird im ersten Schritt die Anzahl der **Airstream**s im Input-**IPA** überprüft (air = getAirstreams s) und je nach Größe ausgewertet. Bei einer Anzahl von 0 wird eine Fehlermeldung ausgegeben, da es sich hierbei um ein ungültiges **IPA** handelt, für 1 wird der jeweilige **Airstream** ausgegeben und für jede andere Länge der rekursive Datenkonstruktor **Air** mit der Liste von Luftströmen.

input x = parseIPA input	map airstream x	x!!0 `isComplexity` Air [PulmonicEgressive, PulmonicEgressive]
"p͡fɐːɐd̥"	[Air [PulmonicEgressive, PulmonicEgressive], PulmonicEgressive, PulmonicEgressive, PulmonicEgressive]	True

```
isAirstream :: IPA -> Airstream -> Bool
isAirstream ipa air = (==) air $ airstream ipa

complexAirstream :: Airstream -> [Airstream]
complexAirstream (Air xs) = xs
```

Die Funktion *isAirstream* funktioniert daher für komplexe Laute nur mit einer Übergabe von ebensolchen rekursiven **Airstream**s. Eine Extraktion der eingebetteten Liste ist über die Funktion *complexAirstream* möglich (für die weitere synchrone Verarbeitung solcher und nicht-rekursiver Objekte siehe Abschnitt 6.2.1: Die Funktion *assign*).

6. Diakritika

1. Implementation

1. Varianten

Alle Diakritika sind in dem Datentyp **Modif** implementiert, zu denen jeweils ein oder mehrere Symbole korrespondieren. Dabei wird keine Unterscheidung gemacht, welche Lauteigenschaft damit definiert wird. **Modif** leitet ebenfalls die Klasse **Ord** ab um inverse Verzeichnisse als **Map** erstellen zu können(siehe Abschnitt 6.1.2).
Analog zu **Airstream** gibt es auch einen rekursiven Konstruktor **Modif [Modif]**, welcher jedoch im Unterschied zu **Air [Airstream]** in der Verarbeitungsfunktion *assign* als Element einer Liste ausgegeben wird, dessen Position der Position des Lautbestandteiles entspricht. Diese Rückgabewertstruktur wurde so gewählt um die Parallelität zwischen simplen und komplexen Lauten zu gewährleisten (für die Funktionen *isModifier*, s.u.). An der Oberfläche ist der Rückgabetyp von **Modif**-Abfrageoperationen **Modif**.

input	x = parse1 input	modifiers x
t̪ʰ	Dentalized, Aspirated Voiceless Alveolar Plosive	[Dentalized, Aspirated]
p͡f̪	Complex Sound: (Voiceless Bilabial Plosive, Dentalized Voiceless Labiodental Fricative)	[Modif [], Modif [Dentalized]]
a͡ʊ̈ⁿ	Complex Sound: (CreakyVoiced Open Front Unrounded Vowel, BreathyVoiced, NasalReleased Close Back Rounded Vowel)	[Modif [CreakyVoiced], Modif [BreathyVoiced, NasalReleased]]

Dies ist bei der Verwendung der Funktionen *modifiers* und *isModifier* zu beachten, da diese immer auf **[Modif]** operieren. Eine Alternative bietet die *containsModifier*, mit welcher überprüft werden kann, ob ein Diakritikon im Eingabe-**IPA** enthalten ist (Die Funktion *assign*, welche allgemein zum

Operieren auf sowohl komplexen als auch simplen **IPA** verwendet wird, wird näher in Abschnitt 6.2.1 erklärt).

```
containsModifier :: IPA -> Modif -> Bool
containsModifier ipa modif = assign (foldl (||) False) (elem modif) id
complexModifiers modifiers ipa
```

input x = parse1 input	x `isModifier` [BreathyVoiced]	x `isModifier` [BreathyVoiced, NasalReleased]	x `containsModifier` BreathyVoiced
a͡ʊ̤ⁿ	False	False	True
ʊ̤ⁿ	False	True	True
ʊ̤	True	False	True

```
*IPA> x = parse1 "a͡ʊ̤ⁿ"
*IPA> x `isModifier` [Modif [CreakyVoiced], Modif [BreathyVoiced,NasalReleased]]
True
```

Passend hierzu gibt die Funktion *complexModifiers* eine Liste von Listen von Diakritika zurück. Mit der Funktion *complexModif* lassen sich auch die Listen von **Modif**s einzelner rekursiver **Modif**s extrahieren.

2. Symbolverzeichnisse mit Duplikaten

Im Gegensatz zu **Boundary** sind Diakritika in **Map**s als **Modif**s selbst gespeichert, da sie in regulären Parses nie einzeln auftreten und nach Parsingfails mit dem **IPA**-Konstruktur **IPAm** von *properIPA* abgelehnt können werden sollen.

```
mInventory = Map.fromList $ tuplify mSym mSpec
mInventory_inv = Map.fromListWith (++) $ (flip tuplify) (map (replicate 1) mSym)
mSpec
```

Da beispielsweise Bögen zum markieren komplexer Laute sowohl als '_͡ ', als auch als ' ͜_ ' auftreten können, wurden das inverse Verzeichnis über die Funktion *fromListWith* des Moduls **Data.Map** erstellt, welche es erlaubt bei redundanten Schlüsselwerten die verschiedenen Werte (hier: **Char**) über eine Funktion miteinander zu verrechnen. Für *mInventory_inv* wurden zunächst alle **Char**s zu **[Char]**s (*map (replicate 1)* mSym) gecastet und je nach Bedarf bei Dopplung des entsprechenden Schlüssels über die Funktion (++) miteinander verkettet. Die Funktion *flip* vor *tuplify* sorgt dafür, dass das entstehende Verzeichnis *mSpec* als Schlüsselmenge verwendet, dabei aber die Reihenfolge von *mSym* und *mSpec* im Code zur deutlicheren Lesbarkeit beibehält.
Bei einer Abfrage eines **Modif**s über *mInventory_inv* wird dabei immer der erste Wert ausgegeben (siehe Abschnitt 8.2: Rückwärtsparser).

2. Eigenschaften mit Standardwert

1. Die Funktion *assign*

Für die Extraktion von Eigenschaften zentral ist die Funktion *assign*, die bereits in *containsModifier* eingesetzt wurde.

```
assign :: ([b1] -> b1) -> (b2 -> b1) -> (t -> b2) -> (t -> [t]) -> (IPA -> t) ->
IPA -> b1
assign constr assignF filterF merger extract ipa = case complexity ipa of
  Simplex -> assignF $ filterF $ extract ipa
  Complex -> constr $ map assignF $ map filterF $ merger $ extract ipa
```

Diese Funktion nimmt sich eine Reihe von Funktionen, zentral hierbei *assignF* und *filterF*, und ein IPA als Argument und ermöglicht so die flexible Anwendung von Operationen auf komplexe und simple Laute zugleich. Dabei wird zunächst der Komplexitätswert des **IPA**-Arguments abgefragt (*case complexity ipa of*) und anschließend die relevante Teilinformation über das Funktionsargument *extract* entzogen. Für Diakritika handelt es sich dabei um die Funktion *modifiers*. Die Rückgabewerte von komplexen und simplen Lauten sind dabei beide jeweils **[Modif]**, für komplexe Laute sind alle Elemente der Liste jedoch mit dem Konstruktor **Modif [Modif]** erstellt, weswegen diese in der letzten Zeile der Funktion nach dem Anwenden von *extract* durch die Funktion *merger* zu einer Liste von **[Modif]** (also **[[Modif]]**) zusammengesetzt werden. Im Anschluss werden mit *filterF* und *assignF* die gewünschten Operationen durchgeführt. Dies ist möglich, da im Funktionsanwendungstrack für komplexe Laute beide Funktionen erst miteinander komponiert und anschließend durch *map* modifiziert, also auf jedes Element einer Liste angewendet werden. So können Funktionen mit **[Modif]** als Eingabewert gleichermaßen für *assignF* und *filterF* eingesetzt werden. Die Trennung zwischen den beiden letzteren Funktionen ist streng genommen nicht notwendig, ermöglicht aber eine bessere Lesbarkeit des Codes, wenn *assignF* beispielsweise nur auf einen Teil der Modifikatoren angewendet werden soll (siehe dazu Abschnitte 6.2.2 und 6.2.3).
Im letzten Schritt werden die Ausgabediakritikalisten der komplexen Laute wieder über einen Konstruktor zusammengesetzt, damit der Ausgabetyp der gesamten Funktion wieder uniform ist.
Besonders zu beachten ist bei Funktionen, die unter Zuhilfenahme von *assign* geschrieben wurden, dass die Rückgabewerte für komplexe Laute aus dem Konstruktor **Modif** und einer Liste von **Modif**s bestehen, anders als bei den Funktion *modifiers*, welche eine Liste von **Modif**s dieser Art als Eingabewert nimmt. Diese Unterscheidung ergibt sich daraus, dass die Eigenschaftsabfragefunktionen, für die *assign* verwendet wird jeweils nur ein Objekt zurückgeben, nie eine ganze Liste. Die Implementation aller Rückgabetypen als Listen würde den Benutzercode unnötig verkomplizieren, da komplexe Laute im cross-linguistischen Vergleich mit simplen Lauten eher selten sind. Ein direkter Zugriff auf die Liste der **Modif**s ist über die *complexModif* möglich. Ein explizites Beispiel für die Anwendung von *assign* findet sich im nächsten Abschnitt.

2. Eigenschaftsabfragen mit fixem Standardwert

Zur Abfrage von Lauteigenschaften, deren Wert nicht in **Sound** oder **Airstream** festgelegt ist, aber von **Modif** definiert werden gibt es die Funktionen *duration*, *tongueroot*, *tone*, *phonation*, *rhotacized*, *lateral* und das Funktionspaar *syllabic* und *uSyllabic*, wobei letztere eine Sonderrolle einnehmen, da bei Ihnen der Standardwert abhängig vom Lautstatus ist (**Syllabic** für Vokale, **NonSyllabic** für Konsonanten: *if ipa `isTyping` Vowel then **Syllabic** else **NonSyllabic*** in *assignF*). Die Funktion *uSyllabic* gibt dabei immer den Standardwert aus.

```
syllabicM = [Syllabic, NonSyllabic]
syllabic :: IPA -> Modif
syllabic ipa = assign Modif assignF (intersect syllabicM) complexModifiers
modifiers ipa
  where
    assignF xs = case length xs of
      0 -> if ipa `isTyping` Vowel then Syllabic else NonSyllabic
```

```
      1 -> head xs

uSyllabic :: IPA -> Modif
uSyllabic ipa = assign Modif syll id complexSound sound ipa
  where
    syll (C _ _ _) = NonSyllabic
    syll (V _ _ _) = Syllabic
```

Alle Funktionen geben als Rückgabewert **Modif** zurück und haben ihnen zugehörig einen Standardwert (z.B. *durationD* = ***Short***, mit Ausnahme von *syllabic*) und eine Liste relevanter **Modif**s (z.B *durationM* = *[**Long**, **HalfLong**, **ExtraShort**]*), in denen der Standardwert nur enthalten ist, wenn er einen korrespondierenden **Char** hat (z.B. **RetractedTongueRoot**).

```
durationD = Short
durationM = [Long, HalfLong, ExtraShort]
duration :: IPA -> Modif
duration ipa = assign Modif (defaultize durationD) (intersect durationM)
complexModifiers modifiers ipa

tongue rootD = RetractedTongueRoot
tongue rootM = [AdvancedTongueRoot, RetractedTongueRoot]
tongue root :: IPA -> Modif
tongue root ipa = assign Modif (defaultize tongue rootD) (intersect tongue rootM)
complexModifiers modifiers ipa
```

Beispiel für *duration*:

simpler Laut:

```
1  duration (parse1 "tːʲ")
```

```
2  duration (IPA [(( (C Alveolar Plosive Voiceless), PulmonicEgressive), [Long,
   Palatalized])])
```

```
3  duration Long, Palatalized Voiceless Alveolar Plosive
```

```
4  assign Modif (defaultize durationD) (intersect durationM) complexModifiers
   modifiers Long, Palatalized Voiceless Alveolar Plosive
```

```
5  case complexity Long, Palatalized Voiceless Alveolar Plosive of
     Simplex -> (defaultize durationD) $ (intersect durationM) $ modifiers
   Long, Palatalized Voiceless Alveolar Plosive
     Complex -> Modif $ map ((defaultize durationD) . (intersect durationM)) $
   complexModifiers $ modifiers  Long, Palatalized Voiceless Alveolar Plosive
```

```
6  case Simplex of
     Simplex -> (defaultize durationD) $ (intersect durationM) $ modifiers
   Long, Palatalized Voiceless Alveolar Plosive
```

```
    Complex -> Modif $ map ((defaultize durationD) . (intersect durationM)) $
  complexModifiers $ modifiers  Long, Palatalized Voiceless Alveolar Plosive
```

```
7 (defaultize durationD) $ (intersect durationM) $ modifiers Long, Palatalized
  Voiceless Alveolar Plosive
```

```
8 (defaultize durationD) $ (intersect durationM) [Long, Palatalized]
```

```
9 (defaultize durationD) [Long]
```

```
10 Long
```

komplexer Laut:

```
1 duration (parse1 "oːi̯")
```

```
2 duration (IPA [(( (V CloseMid Back Rounded), PulmonicEgressive), [Long,
  Palatalized, BreathyVoiced]), ( ( (V Close Front Unrounded),
  PulmonicEgressive), [BreathyVoiced])])
```

```
3 duration Complex Sound: (BreathyVoiced, Long CloseMid Back Rounded Vowel,
  BreathyVoiced Close Front Unrounded Vowel)
```

```
4 assign Modif (defaultize durationD) (intersect durationM) complexModifiers
  modifiers Complex Sound: (BreathyVoiced, Long CloseMid Back Rounded Vowel,
  BreathyVoiced Close Front Unrounded Vowel)
```

```
5 case complexity Complex Sound: (BreathyVoiced, Long CloseMid Back Rounded
  Vowel, BreathyVoiced Close Front Unrounded Vowel) of
    Simplex -> (defaultize durationD) $ (intersect durationM) $ modifiers
  Complex Sound: (BreathyVoiced, Long CloseMid Back Rounded Vowel,
  BreathyVoiced Close Front Unrounded Vowel)
    Complex -> Modif $ map ((defaultize durationD) . (intersect durationM)) $
  complexModifiers $ modifiers  Complex Sound: (BreathyVoiced, Long CloseMid
  Back Rounded Vowel, BreathyVoiced Close Front Unrounded Vowel)
```

```
6 case Complex of
    Simplex -> (defaultize durationD) $ (intersect durationM) $ modifiers
  Complex Sound: (BreathyVoiced, Long CloseMid Back Rounded Vowel,
  BreathyVoiced Close Front Unrounded Vowel)
    Complex -> Modif $ map ((defaultize durationD) . (intersect durationM)) $
  complexModifiers $ modifiers  Complex Sound: (BreathyVoiced, Long CloseMid
  Back Rounded Vowel, BreathyVoiced Close Front Unrounded Vowel)
```

```
7 Modif $ map ((defaultize durationD) . (intersect durationM)) $
  complexModifiers $ modifiers  Complex Sound: (BreathyVoiced, Long CloseMid
  Back Rounded Vowel, BreathyVoiced Close Front Unrounded Vowel)
```

```
8  Modif $ map ((defaultize durationD) . (intersect durationM)) $
   complexModifiers
   [Modif [BreathyVoiced, Long],Modif [BreathyVoiced]]
```

```
9  Modif $ map ((defaultize durationD) . (intersect durationM))
   [[BreathyVoiced, Long],[BreathyVoiced]]
```

```
   Modif $ map (defaultize durationD) [[Long],[]]
```

```
10 Modif [Long, Short]
```

```
11 Modif [Long, Short]
```

3. Modifikation zugrundeliegender Eigenschaften

Zur Abfrage von Lauteigenschaften, die in **Sound** definiert sind, aber von Diakritika überschrieben werden können, gibt es die Funktionen *place*, *manner*, *voicing*, *openness*, *frontness* und *roundedness*, wobei *voicing* eine Sonderrolle einnimmt, da sie bei Konsonanten den zugrundeliegenden Wert aus **Sound** extrahiert, bei Vokalen hingegen den Standardwert **Voiced** zurückgibt, insofern in beiden Fällen kein dem entgegenstehendes Diakritikon vorhanden ist.

```
voicingM = [Devoiced, Envoiced]
voicing :: IPA -> SurfaceFeat
voicing ipa = assign Com adjust (intersect voicingM) complexModifiers modifiers
ipa
  where
    adjust xs = case typing ipa of
      Consonant -> defaultize (Vo $ uVoicing ipa) $ map Mo xs
      Vowel -> defaultize (Vo Voiced) $ map Mo xs
```

Der Rückgabedatentyp ist in jedem Fall **SurfaceFeat**, wobei dieser jeweils einen Konstruktor für die verschiedenen vokalischen und konsonantischen Eigenschaften, einen für potentiell auszugebende Modifikatoren und einen rekursiven Konstruktor für komplexe Laute bereitstellt. Darüber hinaus sind die Funktionen analog zu denen im vorausgehenden Abschnitt gestaltet.

```
data SurfaceFeat = Pl Place | Ma Manner | Vo Voicing | Op Openness | Fr
Frontness | Ro Roundedness | Mo Modif | Com [SurfaceFeat]
  deriving (Eq, Show, Read)
```

Ist ein relevantes Diakritikon vorhanden, so wird dieses als **SurfaceFeat** mit dem Konstruktor **Mo** ausgegeben. Ob dies der Fall ist, lässt sich mit der Funktion *modified* überprüfen. Rekursiv implementierte **SurfaceFeat**s lassen sich nach dem von **Airstream** bekannten Schema mit der Funktion *complexSurfaceFeat* extrahieren. Die Reduktion der nicht-rekursiven Konstruktoren von **SurfaceFeat** erfolgt über die Funktionen *showModif*, *showPlace*, *showManner*, *showVoicing*, *showOpenness*, *showFrontness* und *showRoundedness* respektive. Bei fehlerhafter Anwendung der Funktionen erfolgen spezifische Fehlermeldungen.

```
showModif :: SurfaceFeat -> Modif
showModif (Mo m) = m
```

```
showModif _ = error "call to function 'showMod' for non-modified surface
feature"

showPlace :: SurfaceFeat -> Place
showPlace (Pl p) = p
showPlace (Mo _) = error "call to function 'showPlace' for modified surface
feature"
showPlace _ = error "call to function 'showPlace' for non-place surface feature"
```

7. Laute

1. Implementation

Die zugrundeliegende Lautinformation, als zentraler Teil eines **IPA** ist im Datentyp **Sound** gespeichert, welcher über zwei grundständige und einen rekursiven Konstruktor verfügt. Die grundständigen Konstruktoren unterscheiden sich in ihrer Spezifikation für Konsonanten und Vokale, der rekurisve Konstruktor ermöglicht die Extraktion lautlicher Eigenschaften komplexer **IPA**. Analog zu den anderen Bestandteilen von **IPA** gibt es die Funktionen *sound* und *isSound* für ganzheitliche Abfragen und die Funktion *complexSound* zum extrahieren von Informationen aus dem rekursiven Konstruktor **Sound**.

```
data Sound = C Place Manner Voicing | V Openness Frontness Roundedness | Sound
[Sound]
  deriving (Eq, Show, Read, Ord)

instance ShowIPA Sound where
  showIPA (C p m v) = show v++" "++show p++" "++show m
  showIPA (V o f r) = show o++" "++show f++" "++show r++" Vowel"

sound :: IPA -> Sound
sound (IPA s) = case length $ i of
  1 -> head i
  0 -> error "call to function 'sound' for empty"
  _ -> Sound i
  where
    i = getSounds s
    getSounds [] = []
    getSounds (((s,_),_):xs) = [s] ++ getSounds xs
sound _ = error "call to function 'sound' for non-sound ipa"

isSound :: Sound -> IPA -> Bool
isSound snd ipa = (==) snd $ sound ipa

complexSound :: Sound -> [Sound]
complexSound (Sound ss) = ss
```

Darüber hinaus instanziiert **Sound** die Klassen **Ord**, um inverse Verzeichnisse wie für **Modif** und **Boundary** erstellen zu können, und **ShowIPA**, um **Sound** innerhalb von **IPA** lesbarer darzustellen. Dabei werden Konsonantischen in der kanonischen Reihenfolge **Voicing** - **Place** - **Manner**

dargestellt, Vokale werden als String von **Openness**(Öffnungsgrad) - **Frontness**(Frontalität) und **Roundedness** dargestellt und um "*Vowel*" entsprechend gängiger Formulierungskonventionen.

Zur Implementierung jeder Eigenschaft gibt es einen entsprechenden Datentyp mit grundständigen und rekursiven Konstruktoren und entsprechende Abfragefunktionen.

1. Stimmhaftigkeit

Stimmhaftigkeit ist in **Sound** als primär konsonantische Eigenschaft implementiert, da die entsprechenden Symbole im IPA Auskunft über diese lautliche Dimension geben. Vokale sind kanonisch stimmhaft, daher ist diese Information nicht explizit gespeichert, lässt sich jedoch ebenso über die Funktionen *uVoicing* und *voicing* (Abschnitt 6.2.3) abfragen.

```
data Voicing = Voiced | Voiceless | Voicing [Voicing]
  deriving (Eq, Show, Read, Ord)

-- for conosnants and vowels
uVoicing :: IPA -> Voicing
uVoicing ipa = assign Voicing voicing id complexSound sound ipa
  where
    voicing (C _ _ voicing) = voicing
    voicing (V _ _ _) = Voiced

complexVoicing :: Voicing -> [Voicing]
complexVoicing (Voicing xs) = xs
```

Bei der Funktion *uVoicing* ist zu beachten, dass sie lediglich die zugrundeliegenden Eigenschaften des u.U. anderweitig modifizierten **Char**s zurückgeben. Für exakte Eigenschaftsabfragen ist daher die Funktion *voicing* zu bevorzugen.

2. konsonantische Eigenschaften

Die simultanen Doppelartikulationen Labial-Velar, Labial-Palatal und Alveolar-Palatal sind in der CamelCase-Schreibweise implementiert, ebenso wie das schwedische 'ɧ' (**PostalveoloVelar**). Außerdem ist der Artikulationsort **Epiglottal** ebenfalls trotz seiner Abwesenheit in der offiziellen IPA-Tabelle implementiert.
Clicks und **Implosive**s sind als Artikulationsart angegeben, decken sich jedoch in den Symbolverzeichnissen stets mit ihren korrespondierenden Luftstrommechaniken.

Die Funktionen *uPlace* und *uManner* funktionieren analog zu *uVoicing*, Rückgabewerte komplexer Laute lassen sich mit *complexPlace* und *complexManner* extrahieren.

3. vokalische Eigenschaften

Die Zwischenzustände der Frontalität von Vokalen sind zur exakteren Beschreibung mit **FrontCentral** und **BackCentral** angegeben, für den Öffnungsgrad gibt es eine siebenfache Unterscheidung - zusätzlich zu den jeweiligen rekursiven Konstruktoren.

```
data Openness = Close | CloseCloseMid | CloseMid | Mid | OpenMid | OpenOpenMid |
Open | Openness [Openness]
  deriving (Eq, Show, Read, Ord)
```

Die Funktionen *uOpenness*, *complexOpenness*, *uFrontness*, *complexFrontness*, *uRoundedness*, *complexRoundedness* implementieren das selbe Schema wie in den vorangegangenen Abschnitten.

2. Speicherstruktur und komplexe Symbole

Neben dem Symbolverzeichnis *sInventory* und seiner Inverse *sInventory_inv*, in denen reflexiv **Char**s mit **Sound-Airstream**-Tupeln miteinander assoziiert sind, verfügt das Modul über die Verzeiche *xInventory* und *xInventory_inv*, in denen zusätzlich zu den Laut- und Luftstromeigenschaften auch Eigenschaften gespeichert sind, die sonst von Diakritika definiert werden, wie z.B. 'ɚ' - **Rhotacized** oder 'ɭ' - **Lateral**.

Diese Verzeichnisse ermöglichen das Implementieren beliebig komplexer Sondersymbole, wie feststehende Laut-Ton Symbole. Diese wurden im Rahmen dieses Moduls jedoch außen vor gelassen, da es sich dabei oft um nicht-standardisierte Lautschriften handelt, die nicht notwendigerweise im Sinne des IPA von den hier angebotenen Parsern interpretiert werden könnten.

8. Parser

1. Vorwärtsparser

Zur Übersetzung von IPA-lesbaren Symbolen in den oben vorgestellten Datentyp gibt es primär die Funktion *parseIPA*, welche Listen von **Char**s in Listen von **IPA**s umwandelt.

```
parseIPA :: [Char] -> [IPA]
parseIPA str = spellback str [] [] []

spellback :: [Char] -> [(Sound,Airstream)] -> [Modif] -> [IPA] -> [IPA]
spellback [] zw dia out = (out ++ mergeIPA zw dia)
spellback (x:xs) zw dia out | x `elem` sSym = if checklast Tiebar dia
  then spellback xs (zw ++ [sTemp]) (dia ++ [TieEnd]) out
  else spellback xs [sTemp] [] (out ++ mergeIPA zw dia)

                            | x `elem` xSym = if checklast Tiebar dia
  then spellback xs (zw ++ [fst xTemp]) (dia ++ (snd xTemp)) out --doesn't need
TieEnd
  else spellback xs [fst xTemp] (snd xTemp) (mergeIPA zw dia ++ out)

                            | x `elem` mSym = spellback xs zw (dia ++ [mTemp])
out

                            | x `elem` bSym = spellback xs [] [] (out ++
mergeIPA zw dia ++ [bTemp])

                            | x == ejm = spellback xs (ejectify zw) dia out

                            | otherwise = spellback xs [] [] (out ++ mergeIPA zw
dia ++ [Unreadable x])
  where
    sTemp = fromJust $ Map.lookup x sInventory
    mTemp = fromJust $ Map.lookup x mInventory
    bTemp = fromJust $ Map.lookup x bInventory
    xTemp = fromJust $ Map.lookup x xInventory
```

```
    ejectify = (uncurry (++)) . (\ (xs,[(s,_)]) -> (xs,
[(s,GlottalicEgressive)])) . splitAt (length zw-1)
    checklast i l = if (&&) ((&&) (i `elem` l) $ (==) i $ last l) $ zw /= []
then True else False

mergeIPA :: [(Sound, Airstream)] -> [Modif] -> [IPA]
mergeIPA [] dia = map IPAm $ dia
mergeIPA sa dia = flip (:) [] $ IPA $ tuplify sa $ cut $ filter (/= TieEnd) dia
  where
    cut [] = [[]]
    cut x = case elemIndex Tiebar x of
      Just int -> [take int x] ++ cut (drop (int+1) x)
      Nothing  -> [x]
```

Die nur innerhalb *parseIPA* verwendete Funktion *spellback* regelt dabei alle wichtigen Operationen. Als Argumente erhält sie den Input **String**, eine Liste von **Sound** - **Airstream**-Tupeln bereits erkannter Symbole *zw*, eine Liste entsprechender **Modif**s *dia* und eine Liste fertig-konstruierter **IPA** *out*, die nach Abarbeiten des Inputs ausgegeben wird.
Über Guarded Expressions wird das aktuelle Element zunächst auf Zugehörigkeit zu den Symbollisten hin überprüft

Symboltyp	Symbolliste	**Map**	Beispiele
Char -> (**Sound**, **Airstream**)	*sSym*	*sInventory*	`('a', (V Open Front Unrounded, PulmonicEgressive)) , ('b', (C Bilabial Plosive Voiced, PulmonicEgressive)) , ('c', (C Palatal Plosive Voiceless, PulmonicEgressive))`
Char -> ((**Sound**, **Airstream**), **[Modif]**)	*xSym*	*xInventory*	`('l', ((C Alveolar Approximant Voiced, PulmonicEgressive), [Lateral])), ('ǁ', ((C Alveolar Click Voiceless, LingualIngressive), [Lateral])), ('ɚ', ((V Mid Central Unrounded, PulmonicEgressive), [Rhotacized]))`
Char -> (**IPAb Boundary**)	*bSym*	*bInventory*	`('‖', IPAb MajorGroup),('↗', IPAb GlobalRise), (''↘', IPAb GlobalFall)`

Bei erfolgreicher Zuordnung zu *sSym* oder *xSym* wird zwischen den Fällen unterschieden, ob das oberste Element von *dia* Tiebar ist und somit gerade ein komplexer Laut über Bögen aufgebaut wird oder nicht. Ist dem so, wird der neue Laut mit *zw* verkettet, ansonsten werden die bisherigen

Lautinformationen und Diakritika über die Funktion *mergeIPA* zusammengesetzt und an *out* angefügt.
Im besonderen Fall von Symbolen aus *sSym* wird zusätzlich der Spezialmodifikator **TieEnd** angefügt, um für *mergeIPA* zu signalisieren, dass alle relevanten Laute der Verkettungsoperation in *zw* abgelegt sind. Würde **TieEnd** nicht eingefügt werden, würden weitere folgende Laute ebenfalls in *zw* verschoben werden, insofern sie nicht durch einen Modifkator unterbrochen werden. Die Funktion *mergeIPA* wirft darüber hinaus eine Fehlermeldung, da sie die Modifikatorenmenge anhand der **Tiebar**s auf die Laute distribuiert und dazu die Funktion *tuplify* aus **IPA_util.hs** verwendet, die analog zu *zip* funktioniert, jedoch nur gleichlange Listen akzeptiert. Für Symbole aus *xSym* ist dies nicht notwendig, da sie eine **Modif**-Spezifikation bereits aus *xInventory* mitbringen.

Bei einer Zuordnung zu *mSym* wird das erkannte Element im dazugehörigen Verzeichnis *mInventory* nachgeschlagen und zu *dia* hinzugefügt. Im Regelfall werden alle Diakritika einem **Sound-Airstream**-Tupel zugeordnet. Sollte dies fehlschlagen, findet ein Typecast zu **IPA** über den Konstruktor **IPAm Modif** in *mergeIPA* statt. Dies kann passieren, wenn das bindende vorangegangene Symbol unlesbar war oder ein Formatierungsfehler vorliegt. In diesem Fall ist *zw* leer und *dia* wird als "*map **IPAm** dia*" ausgegeben.

Sollte das aktuelle Element jedoch keiner Symbolliste zugeordnet werden, wird überprüft, ob es sich um das Diakritikon für Ejektive *ejm* handelt. Ist dies der Fall, wird der letzte in *zw* abgelegte Luftstrom über die gebundene Funktion *ejectify* mit **GlottalicEgressive** überschrieben.
Ansonsten wird das Symbol als nicht lesbar mit dem **IPA**-Konstruktor **Unreadable Char** in den Output aufgenommen.

Die Funktion *parse1* ist ein Überbau zu *parseIPA*, die nur das erste vollständig geparste **IPA**-Element ausgibt: *parse1 :: **[Char]** -> **IPA***.
Außerdem verfügt das Modul über eine Funktion *parseSym*, die einzelene **Char** zu **IPA** castet. Diese Funktion ist ausschließlich zum Analysieren einzelner Symbole gedacht. Um Verwechslung mit regulären **IPA** zu vermeiden werden Laute mit dem Konstruktor **IPAs** erstellt.

```
parseSym :: Char -> IPA
parseSym x      | x `elem` sSym = case Map.lookup x sInventory of
      Just temp -> IPAs (temp, [])
      Nothing   -> Unreadable x

                | x `elem` mSym = case Map.lookup x mInventory of
      Just temp -> IPAm temp
      Nothing   -> Unreadable x

                | x `elem` bSym = case Map.lookup x bInventory of
      Just temp -> temp
      Nothing   -> Unreadable x

                | x `elem` xSym = case Map.lookup x xInventory of
      Just temp -> IPAs temp
      Nothing   -> Unreadable x

                | x == ejm = IPAm EjectivityMarker
                | otherwise = Unreadable x
```

2. Rückwärtsparser

Eine Umwandlung von **IPA** zu **Char** ist mit der Funktion *backparseIPA* möglich, wobei anders als bei den Vorwärtsparsern keine Unterscheidungen zwischen den Konstruktoren **IPA** und **IPAs** gemacht wird. Die Funktion *assign* strukturiert den zentralen (default-)Patternmatch für die Konstruktoren **IPA** und **IPAs** um komplexe und simple Laute synchron verarbeiten zu können.
Über die **IPA_util.hs**-Funktion *partial* (als *filterF*Argument für *assign*) für können partielle Matches aus den Spezifikationslisten extrahiert werden, was die Abfrage auf *xInventory_inv* ermöglicht, wenn andere Diakritika als die symbolzugeordneten vorliegen wie bspw. in 'l̥', welches sowohl **Lateral** als auch **Devoiced** ist, jedoch nur **Lateral** mit dem **Char** 'l' selbst assoziiert ist.
Die Funktion *backparseIPA rekonstruiert* auch alle unlesbaren **Char** im Output, irreguläre Ausdrücke werden mit "*#Unknown#*" zurückgegeben.

Aufgrund der Art der Implementation der inversen Verzeichnisse, können zwar duplikate Symbole via *parseIPA* interpretiert werden, *backparseIPA* wählt jedoch immer das selbe Symbol aus, so z.B. für Stimmlosigkeitsdiakritika, 'g'/'g' und Bögen.

9. Überblick: Benennungen

Eigenschaft	Dimensionen

Vokalqualität (**Frontness, Openness, Roundedness**)	`Front, FrontCentral, Central, BackCentral, Back`
	
	`Close, CloseCloseMid  CloseMid, Mid, OpenMid, OpenOpenMid, Open`
	
	`Rounded, Unrounded`

Modifs für Vokale	`MoreRounded, LessRounded, Centralized, MidCentralized, Raised, Lowered,`

Konsonantische Eigenschaften (**Manner**, **Place**)	`Plosive, Nasal, Trill, Tap, Flap, Fricative, Approximant, Click, Implosive`
	
	`Bilabial, Labiodental, Dental, Alveolar, Postalveolar, Retroflex, Palatal, Velar, Uvular, Pharyngeal, Glottal, LabialVelar, LabialPalatal, Epiglottal, AlveoloPalatal, PalatoAlveolar, PostalveolarVelar`

Modifs für Konsonanten	`Labialized, Palatalized, Velarized, Pharyngealized, LinguoLabial, VelarizedOrPharyngealized, Dentalized, Laminal, Apical, Advanced, Retracted, Lowered, Raised, NasalReleased, Rhotacized, Nasalized, Unreleased, Aspirated`
Voicing	`Voiced, Voiceless`
Modifs für Stimmhaftigkeit	`Devoiced, Envoiced`

SurfaceFeat	`Pl Place, Ma Manner, Vo Voicing, Op Openness, Fr Frontness, Ro Roundedness, Mo Modif`
Modifs für Syllabizität	`Syllabic, NonSyllabic`
Modifs für Lateralität	`Lateral, NonLateral`
Modifs für Rhotazität	`NonRhotacized, Rhotacized`
Modifs für Phonation	`BreathyVoiced, CreakyVoiced, MurmuredVoiced, NormalVoiced`
Modifs für Ton	`HighTone, MidTone, LowTone, ExtraHighTone, ExtraLowTone, FallingTone, RisingTone, HighRisingTone, LowRisingTone, RisingFallingTone, NoTone`
Modifs für RTR/ATR	`AdvancedTongueroot, RetractedTongueroot`
Modifs für Länge	`Long, HalfLong, ExtraShort, Short`
Sosnstige **Modif**s	`BilabialVoicedPrestopped, AlveolarVoicedPrestopped, VelarVoicedPrestopped, BilabialPrenasalized, AlveolarPrenasalized, VelarPrenasalized, RetroflexPrenasalized, PalatalPrenasalized, PalatalVoicedPrestopped, BilabialVoicedPoststopped, AlveolarVoicedPoststopped, VelarVoicedPoststopped, PalatalVoicedPoststopped, BilabialVoicelessPoststopped, Upstep, Downstep,`
Spezielle **Modif**s	`Tiebar, TieEnd, EjectivityMarker`
Airstream	`PulmonicEgressive, GlottalicEgressive, GlottalicIngressive, LingualIngressive`
Boundary	`SyllableBoundary, Whitespace, Undertie, PrimaryStress, SecondaryStress, MinorGroup, MajorGroup, GlobalRise, GlobalFall`
Complexity	`Simplex, Complex`
Typing	`ComplexV, ComplexC, Consonant, Vowel, Diacritic, Boundary, RawSound, None, MixedComplex`

10. Ausblick

1. fehlende Symbole

Aus der offiziellen IPA-Tabelle sind die alleinstehenden Konturtonsymbole nicht in diesem Modul implementiert, da es nicht möglich war, sie einem Unicodewert zuzuordnen.

2. Erweiterung

Im vorliegenden Modul wurden nur offizielle **Char**s implementiert, die von der International Phonetic Association offiziell anerkannt werden und 'ɚ', 'ɝ', welche besonders oft für die Beschreibung des Englischen verwendet werden. Weitere Symbole sind durchaus möglich hinzuzufügen, in dem jeweils korrespondierende Einträge an die jeweiligen Symbol- und Spezifikationslisten angefügt werden. Denkbar wäre dies bei bedarf für feststehende Vokal-Ton Kombinationen.

3. metrischer Parser

Eine nächste sinnvolle Erweiterung für dieses Modul ist die Implementation eines metrischen Parsers, der metrisch vollständig getaktete, mindestens jedoch aber um vollständig spezifizierende Sibengrenzen ergänzte Inputstrings voraussetzt. Über einen solchen wäre es möglich Präartikulationseffekte interpretieren zu können und Operationen auf bspw. Silben- oder Fuß-Ebene vorzunehmen. Die innere Struktur des Speicherformats müsste eine mehrstufige Speicherung entsprechend zulassen. Denkbar wären hier rekursive Datenkonstruktoren für **IPA**, dagegen spricht die dem entwachsende notorische Schwierigkeit die Ebenen konkret zu identifizieren, aber gleichzeitig dynamisch zu halten: Die Funktionalität des Parsers wäre erheblich eingeschränkt, müsste in jedem Input jede mögliche Strukturierung ausgezeichnet werden.
Alternativ könnte dieses Implementierungsproblem über ein Typensynonym :: [**IPA**] oder einen neuen Datentyp gelöst werden, was jedoch eine Dopplung des bestehenden Codes nicht komplett verhindern kann. Qualitätsmaßstab des metrischen Parsers muss nicht nur sein, welche neuen Funktionalitäten er einführt, sondern auch wie gut die alten Funktionen übertragbar bleiben.

Eine zusätzliche Option wäre die Unterspezifizierung des Parsers zur Ergänzung um Transduktoren oder Funktionen, die die Einteilung in phonologische Sequenzen selbst vornehmen, indem sie beispielsweise die exakte Sonoritätshierarchie der Sprache und dem entgegenstehende Konsonantencluster beschreiben.

4. Leipziger Glossierungsregeln

Das Modul ist mit der Beschreibung phonologischer Effekte im Sinn konzipiert worden, wäre jedoch bei einer Erweiterung um die Sonderzeichen, wie sie gemäß der Leipziger Glossierungsregeln verwendet werden, auch für andere Phänomene nutzbar. Insbesondere in Kombination mit einem metrischen Parser würde dies es erlauben morphophonologische oder linear-verstandene morphologische Effekte zu beschreiben.
Letztendlich wäre es wünchenswert Module für alle verschiedenen Ebenen der Repräsentation linguistischer Information zu gestalten und über Interfaces miteinander zu verknüpfen.