- «`%token` *MATCHLIST*» Hier werden den Token, die vom Lexer erzeugt wurden, die *Terminals* zugewiesen, die in der BNF verwendet werden.[15]

Es schließt sich der Grammatikteil an (vom zweiten Teil durch ein `%%` getrennt), in dem also in einer BNF ähnlichen Notation die Syntax, wie man sie sich zuvor überlegt hat, aufgeschrieben wird. Auf die kleinen Unterschiede zur BNF möchte ich hier nicht eingehen, wichtiger ist, daß man hinter jede Regel eine sogenannte Aktion schreiben kann. Als Beispiel diene:

```
> start    : {- empty -}   { [] }
>          | start ATOM    { show $2 : $1 }
```

Zunächst bezeichnet `{- empty -}` das leere Wort — dies ist aus Lesbarkeitsgründen Konvention. Hinter den Regeln steht in geschweiften Klammern jeweils ein Stück Haskell-Code. Dies sind „Aktionen“, die immer dann ausgeführt werden, wenn diese Regel abgeleitet wird. Mittels `$1` wird auf den Wert von *start* und mittels `$2` auf den Wert von ATOM zugegriffen. Der Wert eines Terminals ist dabei normalerweise das Terminal selbst. Durch die Aktionen hat der Parser also eine Ausgabe (und ist nicht nur ein reiner Syntax-Überprüfer). In unserem Beispiel ist die Ausgabe eine Liste von ”ATOM“-Strings.

Der vierte Teil eines `Lucky`-Skripts ist wieder ein in geschweifte Klammern gesetzter Block mit Haskell-Code, welcher unverändert ans Ende der erzeugten Datei gesetzt wird. Hier muß zumindest die Funktion `luckyError` stehen, welche im Fall eines Syntax-Fehlers von der Parser-Funktion automatisch angesprungen wird (damit dies funktioniert, darf für diese Funktion kein anderer Name verwendet werden.) Oft ist hier auch der Lexer implementiert, wenn der Programmierer zu faul war, ihn in ein eingenes Modul zu stecken.

Ich möchte noch einmal auf das Beispiel in [Klo97] verweisen, welches sehr gut für das Verständnis des eben Besprochenen geeignet ist.[16]

7 Aufgabe 1: Aussagenlogischer Beweiser in Haskell

In der ersten Aufgabe sollen aussagenlogische Beweiser in Haskell geschrieben werden. Die Zielfunktionalität ist ein Programm, das eine textuelle Darstellung einer aussagenlogischen Formel einliest und überprüft, ob es sich dabei um eine Tautologie handelt. Als Ausgabe soll eine textuelle Darstellung des Beweises generiert werden. Es gliedert sich die Aufgabe dabei in folgende Teilaufgaben:

- ein Parser
- ein Beweiser über Wahrheitstafeln

[15] Warum man hier noch einmal zusätzlich zwischen Token und Terminals unterscheidet, verstehe ich nicht ganz. Der Grund könnte sein, daß auf diese Weise evtl. komplexe Token, die Zusatzinformationen wie zB. Positionsangaben beinhalten (s. Beispiel in [Klo97]), auf diese Weise abgekürzt werden können.

[16] Hier ein eigenes Beispiel zu bringen, würde den Rahmen dieses Skriptes sprengen.

Wir müssen nun eine Einigung über die verschiedenen Datentypen treffen. Diese dienen dann als Darstellung von aussagenlogischen Formeln als Haskelldaten für die Schnittstelle zwischen Parser und Beweiser über Wahrheitstafeln.

7.1 Der Parser

Der Parser soll eine Funktion sein, die einen String als Eingabe hat und als Ausgabe die in diesem String kodierte Formel als Haskell-Datentyp. Hierzu müssen wir zunächst definieren, was eine aussagenlogische Formel ist, und wie sie als String dargestellt wird.
Die atomare Einheit der Formeln sind aussagenlogische Variablen. Die Menge der Atome **S** sei die Menge aller Wörter über den Groß- und Kleinbuchstaben.
Die Menge aller aussagenlogischen Formeln For_S, sei die kleinste Menge von Wörtern über $\mathbf{S} \cup \{(,),\neg,\rightarrow,\vee,\wedge,\leftrightarrow\}$ mit folgenden Eigenschaften:

i) $\mathbf{S} \subseteq \mathrm{For}_S$

ii) Wenn $A \in \mathrm{For}_S$ und $B \in \mathrm{For}_S$, dann sind auch $\neg A$ und $(A \otimes B) \in \mathrm{For}_S$ mit $\otimes \in \{\rightarrow,\vee,\wedge,\leftrightarrow\}$.

Da der ASCII-Zeichensatz die für unsere Formeln benötigten Sonderzeichen nicht bereit stellt, müssen wir eine Darstellung dieser Zeichen als ASCII-String definieren. Hierbei bedienen wir uns Stringsequenzen ähnlich denen aus dem Textverarbeitungssystem LaTeX. Die einzelnen Sonderzeichen werden dabei wie folgt dargestellt:

Sonderzeichen	textuelle Darstellung
$\neg$	\nicht
$\rightarrow$	\folgt
$\vee$	\oder
$\wedge$	\und
$\leftrightarrow$	\gdw

Die Benutzung der LaTeX-Befehle erlaubt es uns dann auch, die Formeln in Textdokumente zu übernehmen und sie in lesbarer Form auszudrucken.
Der Parser soll jetzt also einen String, der eine aussagenlogische Formel darstellt einlesen und in ein internes Syntaxformat der Formel übersetzen. Dabei soll es erlaubt sein Leerzeichen, Zeilenendemarken und ähnliches zu überlesen. Der erzeugte Syntaxbaum wird dann an den Beweiser übergeben. Eine Eingabe kann z.B. so aussehen:

```
A \und \nicht A \folgt B
```

Wie wir gesehen haben, sind aussagenlogische Formeln Terme, die ein übergeordnetes Funktionszeichen (einen Junktor) mit aussagenlogischen Formeln als Argumente haben. Wir können diese Terme in einem rekursiven Typ als Bäume darstellen. Hierfür definieren wir uns einen Datentypen in naheliegender Weise:

```
> data For a
>   = Atom a
>   | Negat (For a)
>   | Impl  (For a) (For a)
>   | Equi  (For a) (For a)
>   | Und   (For a) (For a)
>   | Oder  (For a) (For a)
```

Man beachte wie direkt sich dieser Datentyp aus der Spezifikation der Formeln ergibt. Wir haben hier einen abstrakten Datentypen definiert. Dieser Typ besteht aus sechs Konstruktoren. Der Typ ist aber noch allgemeiner gehalten, als wir ihn in der Aufgabenstellung brauchen. Wir haben als Typ für atomare Formeln einen variablen Typ `a` angenommen. Diesen können wir theoretisch mit beliebigen Typen instanziieren. Wenn wir unser Programm hierbei möglichst allgemein schreiben, können wir es einfach portieren oder abändern. Für unsere spezielle Aufgabenstellung wird `a` mit `S` instanziiert:

```
> type ForS = For String
```

Wir sollten aber versuchen die Funktionen soweit wie möglich auf dem Typ `For a` zu definieren und wenn nötig lediglich zu verlangen, daß der Typ `a` in der Klasse `Eq` enthalten ist. Ein Beispiel für eine Formel ist:

```
> f1 = Negat (Impl (Und (Atom "A") (Negat (Atom "A"))) (Atom "B"))
```

7.2 Beweiser über Wahrheitstafeln

Das Verfahren Wahrheitstafeln für aussagenlogische Formeln aufzustellen, sollte aus der Praktischen Informatik 1 bekannt sein. Eine Wahrheitstafel hat insgesamt so viele Spalten wie es unterschiedliche Variablen gibt plus der Anzahl aller auftretenden Junktoren. Eine Wahrheitstafel hat bekanntlich 2^n Zeilen, wobei n die Anzahl der unterschiedlichen Variablen ist. Zusätzlich gibt es noch eine Kopfzeile, die die Spalten bezeichnet. Die Aufgabe besteht also darin die Eingabe zu parsen und für jede Belegung der Variablen, die in der Eingabe auftreten den Wert aller Unterausdrücke zu berechnen. Der Beweiser soll dabei einen String, der als Eingabe für LaTeX benützt werden kann ausgeben. Das entsprechende Format, um eine Tabelle in LaTeX auszugeben, ist die *tabular*-Umgebung. Es wird ein Beispiel gegeben, wie eine solche Tabelle aussehen könnte:

```
\begin{tabular}{|c|c|c||c|c|c|}\hline
A&B&C&A$\rightarrow$C&$\neg($A$\rightarrow$C)&
$(\neg$(A$\rightarrow$C))$\vee$B\\\hline\hline
0&0&0&1&0&0\\\hline
0&0&1&1&0&1\\\hline
0&1&0&1&0&0\\\hline
0&1&1&1&0&1\\\hline
1&0&0&0&1&1\\\hline
1&0&1&0&1&1\\\hline
1&1&0&1&0&0\\\hline
1&1&1&1&0&1\\\hline
\end{tabular}\\[1ex]
Die Formel ist keine Tautologie.\\
```


Wir haben diesmal den Text zur besseren Lesbarkeit nicht mit den Escape-Sequenzen von Haskell ausgeschrieben. Dies muß bei der Programmierung natürlich wieder berücksichtigt werden.
Obige Tabellendefinition ergibt im Druckbild dann folgende Tabelle:

A	B	C	A→C	¬(A→C)	(¬(A→C))∨B
0	0	0	1	0	0
0	0	1	1	0	1
0	1	0	1	0	0
0	1	1	1	0	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	1	0	0
1	1	1	1	0	1

Die Formel ist keine Tautologie.

Eine Funktion, die ein Feld in einen solchen LaTeX-String umwandelt wird zur Verfügung gestellt.

Erzeugen Sie geeignete Eingaben, um Ihr Programm zu testen und sorgen Sie dafür, daß es mit den auf den WWW-Seiten des Praktikums gegebenen Tests geprüft werden kann.

7.3 Exkurs: I/O in lazy funktionalen Sprachen - eine Übersicht

Problem

Von einem Programm, das über I/O Operationen mit dem Betriebssystem kommuniziert haben wir in der Regel die folgende Vorstellung:

```
(print stdout "Geben sie eine Zahl ein");
(read stdin number);
(print stdout "das Quadrat von ");
(print stdout (int2string number));
(print stdout "ist: ");
(print stdout (int2string (square number)))
```

Von derartigen Programmen interessiert nicht mehr der Wert einer Funktion sondern ihr Seiteneffekt, in Form eines Bildschirmausdrucks oder einer geänderten Datei. Reines funktionales Programmieren ist aber auf das Programmieren ohne Seiteneffekte ausgelegt.
Unsere Vorstellung von Input/Output Operationen widerspricht so dem funktionalen Paradigma im Wesentlichen an zwei Punkten.

- **referentielle Transparenz**
  I/O kann als ein Kommunikationsprozess zwischen einem Programm und dem Betriebssystem betrachtet werden. Das Programm schickt so z.B. mit einem *readFile* Befehl eine Botschaft an das Betriebssystem, die je nach Status des Dateisystems mit einem bestimmten String oder aber einer Fehlermeldung vom System beantwortet wird. Betrachten wir den *readFile* Befehl als eine Funktion, die als Wert die Antwort des Betriebssystems hat, so verletzt diese Funktion die Forderung nach referentieller Transparenz. Je nach dem Status des Dateisystems kann derselbe Funktionsaufruf `(readFile datei)` verschiedene Werte bekommen, abhängig davon, ob die Datei existiert, gelesen werden darf, gelesen werden kann und welcher Text unter *datei* abgespeichert ist. Dies sind alles dem Programm unbekannte Zustände.

- **Auswertungsreihenfolge**
  Als zweite Schwierigkeit stellt sich die Frage der Auswertungsreihenfolge: Verzögert auswertende funktionale Sprachen sind gerade so konzipiert, daß die Auswertungsreihenfolge der einzelnen Redexe im Programmablauf streng nach der Notwendigkeit gesteuert wird. Erst wenn der Wert eines Ausdrucks in Normalform benötigt wird, wird dieser auch ausgewertet. Bei I/O Funktionen im klassischen Sinn interessiert weniger der Wert der Funktion sondern der durch sie hervorgerufene Seiteneffekt. Es kann dem Programmierer mitunter entgehen, daß der Wert eines *write* Befehls zur Programmevaluierung nicht benötigt wird und so dieser auch nicht ausgeführt wird.
  Ein Codeoptimierer, der referentielle Transparenz voraussetzt, könnte eine zweimalige Abfrage an den Benutzer wegoptimieren, da sie über denselben Ausdruck, der aber verschiedene Werte erwartet realisiert wird.
  Im allgemeinen haben wir von I/O Prozessen die Vorstellung einer Sequenz einzelner I/O Befehle. Verzögert auswertende funktionale Sprachen wollen aber Sequenzialisierung möglichst nicht erzwingen. So wird auch kein expliziter Sequenzoperator wie in imperativen Sprachen (C, Pascal) aber auch in strikten funktionalen Sprachen (Scheme) zur Verfügung gestellt.

Wir können in verzögert auswertenden funktionalen Sprachen also nicht mit den uns vertrauten I/O Routinen rechnen.
Betrachten wir nun verschiedene Lösungen des I/O Problems in verzögert auswertenden funktionalen Sprachen.

I/O als Dialog in Haskell

In Haskell 1.2 wurden die I/O Funktionen als ein Dialog zwischen Programm und Betriebssystem modelliert. [O'D85] Dieser Dialog stellt sich dar als eine unendliche Liste

von Anfragen an das Betriebssystem `[Request]` auf Seite des Programms als Kommunikationspartner und eine ebensolche Liste von Antwortmeldungen auf diese Anfragen des Betriebssystems `[Response]`.
Wir haben also die zwei Listen:
$[req_1, req_2, req_3, ...]$
$[res_1, res_2, res_3, ...]$
als Teile unseres Kommunikationsprozesses, wobei jeweils res_i die Antwort des Betriebssystems auf die Meldung req_i des Programms sein soll. `Request` und `Response` sind abstrakte Datentypen. (zumindest für `Request` hätte man wohl eher Funktionen erwartet.)
Diese Listen erzwingen so schon die Sequenzialisierung der I/O Operationen.
Ein funktionaler Zusammenhang von den Anfragen zu den Antworten würde der referentiellen Transparenz widersprechen. Der Trick ist nun den Dialog gerade als Funktion von einer Liste von Antworten auf eine Liste von Anfragen zu definieren:
`Dialogue = [Response] -> [Request]`
Dieses ist zunächst einmal recht konterintuitiv und trägt sehr zur Verunsicherung von Programmierern bei. Es sei z.B. das i-te Element des Funktionswertes eines Dialogs die Meldung `ReadFile datei`; dann ist das i-te Element des Arguments die Antwortmeldung des Betriebssystems auf diesen Lesewunsch, also in der Regel der Art: `Str` *dateiInhalt*.
Eine Funktion `main` vom Typ `Dialogue` schreibt man nun in der Art, daß die Funktion für die zu erwartenden Antwortpattern spezifiziert wird und diese Pattern dann in weiteren Anfragen benutzt werden. Unser anfängliches Beispielprogramm sähe also wie folgt aus[17]:

```
>moin ~(~Success: ~(~(Str zahl) : ~(~Success:[]))) =
>          [AppendChan stdout "Geben Sie ein Zahl ein: ",
>           ReadChan stdin,
>           AppendChan stdout ("\n Das Quadrat von "
>                            ++ (head (lines zahl))
>                            ++ " ist: "
>                            ++ ausgabe
>                            ++ "!\n" )]
>            where
>             ausgabe = int2string ((string2int (head (lines zahl)))^2)
```

Damit diese Art der Programmierung auch funktioniert, muß man bei der Funktionsspezifikation sogenannte *irrefutable patterns* verwenden. Abweichend von der Auswertung bei *patterns* sonst wird auch die Evaluierung des Arguments zwecks *matchens* mit den *patterns* verzögert, was ja, da nur eine Funktionsalternative vorhanden ist, auch nicht weiter notwendig ist. Den *patterns* kommt also nicht wie gewohnt die Aufgabe der Steuerung von Alternativen zu, sondern die der Benennung von Argumentteilen, die im Grunde die eigentlich interessierenden Ergebnisse der I/O Operationen sind.

[17] Wer dieses Programm oder ein ähnliches ausprobiert, wird sehen, in welcher Weise die lazy Auswertungsstrategie zu nicht ganz den erwarteten Ergebnis führt.

I/O mit unique types in Clean

Für die Programmiersprache **Clean** [NSvP91] wurde eine andere Methode zur Einbindung von I/O Operationen in das funktionale Paradigma gewählt. Der Grundgedanke hierbei ist: wird eine I/O Operation als Funktion betrachtet, so hängt ihr Wert neben den offensichtlichen explizit angegebenen Argumenten (wie z.B. den Dateinamen der zu lesenden Datei) auch noch von dem Status des gesammten I/O Systems ab. Dieser wird als Umgebung bezeichnet. Die Umgebung muß jetzt den I/O Funktionen als Parameter übergeben werden. Der Wert einer I/O Funktion ist dann eine neue Umgebung.
Ein **Clean**-Programm ist eine Funktion vom Typ `WORLD -> WORLD` wobei `WORLD` gerade eine Umgebung beschreibt. Der Typ `WORLD` ist dabei noch strukturiert in eine *Events*-Umgebung (die den Status von Tastatur, Maus etc angibt) und eine *Files*-Umgebung, welche wiederum in einzelne *files* strukturiert ist. Diese Umgebungen können mit vordefinierten Funktionen aus einer `WORLD` geöffnet werden, dann als Parameter zu I/O-Zwecken gebraucht werden und schließlich wieder im veränderten Status geschlossen werden.
Mit diesem Prinzip des *enviroment passings* ist das Problem der referentiellen Transparenz gelöst.
Es muß jetzt noch die Sequenzialisierung der einzelnen I/O Operationen sichergestellt werden, sonst kann es im Ausdruck
`F file -> (FWriteC 'a' file, FWriteC 'b' file);`
dazu führen, daß der erste oder der zweite Teil des Ergebnispaares zuerst evaluiert wird, was dann zu unterschiedlichen Ergebnissen führt, aber beides nicht zu dem wahrscheinlich erwarteten.
Das Problem ist hier, daß auch eine Datei nur einmal in der Umgebung vorhanden ist und, wenn sie denn mehrfach in einem Ausdruck gebraucht wird, *geshared* auf dieselbe Datei verwiesen wird. Ein Objekt, auf dem destruktive Updates gemacht werden, darf aber nicht gemeinsam genutzt werden. Um dies zu unterbinden wurde in **Clean** das Konzept der *unique types* [SBvEP93] eingeführt. Man kann beliebige Typen als *unique* erklären, womit man ausdrückt, daß ein Objekt dieses Typs nur einmal zur Zeit referenziert werden darf. Funktionen, die wie `FWriteC` destruktiv ein Objekt verändern, müssen von diesem Objekt einen als *unique* erklärten Typ verlangen.
Um nun den korrekten Gebrauch von *unique* attributierten Objekten zu gewährleisten, ist im Kompilierer noch ein weiterer Test eingebaut, der kontrolliert, daß in keiner Funktion ein *unique* attributiertes Objekt mehrfach referenziert wird.
Das Konzept der *unique types* ist verwandt mit den *linear types* auf Lambda-Ausdrücken. [Abr90]
Eine Kritik am Konzept in **Clean** ist, daß in Kompilierern bereits benutzte Codeoptimierungsverfahren eine weiterhin konsistente Bernutzung der *unique* attributierten Objekte nicht gewährleisten.

Monadisches I/O ab Haskell 1.3

Aus den Problemen, die viele Programmierer mit Dialogen hatten, wurde das monadische I/O-Konzept entwickelt. [PJW93] Es basiert auf einer Programmiermethode, die in geschickter Weise Daten implizit weiterreicht, den Zustands- insbesondere den I/O-

Monaden [Wad90, Wad92]. Hierbei wird ein I/O-Zustand verwendet, der von den I/O-Operationen verändert wird. Jede I/O-Operation erhält zunächst als zusätzliches Argument den I/O-Zustand und liefert als Bestandteil ihres Ergebnisses einen neuen moglicherweise veränderten I/O-Zustand. Will man nun zwei solche I/O-Operationen in einer vorgegebenen Reihenfolge ausführen, so kann man die sequentielle Ausführung sicherstellen, indem man die I/O-Zustands Komponente des Ergebnisses der ersten I/O-Opreation als Argument an die zweite I/O-Operation übergibt. Dadurch besteht zwischen den Anwendungen dieser beiden I/O-Operationen eine Daten-Abhängigkeit, die die Ausführung in der gewünschten Reihenfolge sicherstellt. Die I/O-Monaden erfüllen hier die Aufgabe, das Extrahieren und Weiterreichen des I/O-Zustands zu verbergen, damit das Programm besser lesbar bleibt.

Monaden stellen zwei Operatoren zur Verfügung: `bind` oder auch `>>=`, ein Sequenzoperator, der durch Komposition realisiert wird, und `unit` oder `return` einen Operator, der es erlaubt komplexe Datentypen in eine Monade zusammenzupacken, die dann implizit weitergereicht werden. Es kann also gut die für I/O benötigte Umgebung (die `World`) in einer Monade enthalten sein und weitergereicht werden.
Betrachten wir die beiden Aspekte des monadischen I/O getrennt:

- **Sequenzialisierung**
  Wir benötigen zunächst einmal einen Operator, der uns aus einem einfachen Datentyp ein für I/O-Operationen monadisch verpackten Datentyp generiert:
  `return :: a -> IO a`
  Als zweiten Operator benötigen wir eine Funktion, die es uns erlaubt Sequenzen von I/O-Operationen mit dem monadisch verpackten Datentyp zu definieren:
  `(>>=) :: IO a -> (a -> IO b) -> IO b`
  Das erste Argument wird hier das Ergebnis einer I/O-Operation oder aber der `return`-Funktion sein und das zweite Argument eine Funktion, die ein Argument erhält und in Abhängigkeit von diesem eine I/O-Operation durchführt. `>>=` geht dabei so vor, daß die erste I/O-Operation ausgeführt wird, das in `IO a` verpackte Datum ausgepackt wird und an die zweite Funktion übergeben wird, damit diese in Abhängigkeit des Ergebnisses der vorherigen Funktion operieren kann.
  Mit diesen beiden Funktionen und vordefinierten I/O Funktionen kann nun in natürlicher Weise eine Funktion definiert werden, die ein Wort aus der Tastatur einliest:

```
import IO

wort :: IO String
wort = wortaux []

wortaux :: String -> IO String
wortaux w = getChar >>= \a -> hugsIsEOF >>= \b ->
            if (b || (a == ' ')) then
               return (reverse w)
            else
```

```
                (return (a:w)) >>= wortaux

main = do
        x <- wort
        print x
```

- **referentielle Transparenz**
  Bisher haben wir noch nicht betrachtet, in welcher Weise der Datentyp `IO a` definiert ist, und wie die beiden Monadenfunktion auf ihm operieren.
  Der Typ `IO a` bleibt in Haskell abstrakt, d.h. seine Definition ist dem Programmierer nur über die auf ihm definierten Funktionen zugänglich, nicht aber um mittels Pattern-Matching Objekte dieses Typs zu dekomponieren.

8 Aufgabe 2: Verwalten, Erzeugen und Testen partieller Ordnungen in Haskell

8.1 Einige mathematische Begriffe

Diese und noch weitere Begriffe finden sich auch in [SS].

Relation

Eine zweistellige (binäre) Relation R auf einer Menge S ist eine Teilmenge von $S \times S$, d.h. geordnete Paare (s_1, s_2), s_1 und s_2 aus S, sowie $(s_1, s_2) \neq (s_2, s_1)$. Eine mögliche Darstellung ist: $s_1 R s_2$

Beispiel 3. Sei $S = 2, 3, 5, 6$ und sei die Relation R definiert durch "ist Teiler von". Dann stellt R die folgende Menge dar: $\{(2,2), (2,6), (3,3), (3,6), (5,5), (6,6)\}$

partielle Ordnung

Eine Relation R auf einer Menge S heißt (partielle) Ordnung, wenn für alle a, b, c aus S gilt:

1. R ist *reflexiv*:

$$aRa$$

2. R ist *antisymmetrisch*:

$$aRb \wedge bRa \iff a = b$$