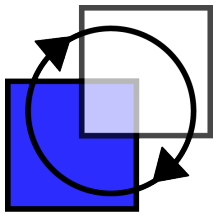
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Prof. Bernd Finkbeiner, Ph.D.
Jana Hofmann, M.Sc.
Reactive Systems Group



Programmierung 1 (WS 2020/21)
Zusatzerklärung zum Thema
Grammatiken & Parsen

Bei diesem Rezept handelt es sich um eine Vorgehensweise, die auf viele Standardaufgaben zum Parsen angewendet werden kann. Häufig ist es möglich, diese Schritte nach Schema abzuarbeiten und so eine korrekte Grammatik aufzustellen und den dazugehörigen Parser zu schreiben. Dies gilt insbesondere für Grammatiken, die nur Terme aus links- und rechtsklammernden binären Operatoren, unären Operatoren, Klammern und primitiven Ausdrücken (beispielsweise Konstanten) beschreiben.

Die Verwendung dieses Rezepts ist nicht bei allen Aufgaben möglich und sollte daher mit Vorsicht genossen werden.

1 Ein Grundrezept zum Entwickeln von Grammatiken und Parsern

1.1 Grammatik aufstellen

(a) Es wird eine Kategorie pro Operator mit unterschiedlichen Prioritäten benötigt und eine zusätzliche für primitive Ausdrücke.

(b) Man beginnt mit dem Operator, der am schwächsten klammert. Diese Kategorie kommt nach ganz oben, die anderen Kategorien werden in aufsteigender Reihenfolge nach ihrer Priorität darunter angeordnet.

(c) Man muss von jeder Kategorie auf die primitiven Ausdrücke “runterfallen” können. In jeder Kategorie (außer der untersten) muss also die Möglichkeit gegeben sein, in die darunterstehende Katogorie zu kommen, *ohne* weitere Symbole abzuleiten.

(d) Mit Klammerung muss ein “Rundlauf” möglich sein. Zwischen den Klammern, welche bei den primitiven Ausdrücken anzusiedeln sind, muss also die oberste, am schwächsten klammernde, Kategorie stehen.

(e) Zu den primitiven Ausdrücken müssen die “Basisfälle” – also Konstanten, Identifier, etc. – hinzugefügt werden.

(f) Bei den Operator-Kategorien müssen die Operatoren in Optionalklammern hinzugefügt werden. In die Optionalklammern gehört der Operator und bei binären Operatoren zusätzlich die gleiche Kategorie wie die, in der wir uns gerade befinden.

  (i) Für linksklammernde Operatoren gilt, dass die Optionalklammern auf der linke Seite der bereits dort stehenden Kategorie stehen müssen.

  (ii) Bei rechtsklammernden Operatoren gehören die Optionalklammern auf die rechte Seite der bereits dort stehenden Kategorie.

1.2 Grammatik RA-tauglich machen

(a) Problematische Kategorien sind solche, die eine Optionalklammer mit Kategorie auf der linken Seite stehen haben. Sie sind daher nicht für den rekursiven Abstieg geeignet und müssen identifiziert und markiert werden. Anschließend wird die Klammerung in diesen Kategorien umgedreht, also werden die Optionalklammern von der linken Seite auf die rechte verschoben, als ob es sich um rechtsklammernde Operatoren handeln würde.

(b) Als nächstes müssen Hilfskategorien eingeführt werden – pro problematischer Kategorie wird eine Hilfskategorie benötigt. Die Optionalklammern in der problematischen Kategorie werden durch die Hilfskategorie ersetzt, die Hilfskategorie besteht lediglich aus ebendieser Optionalklammer.

(c) Die Hilfskategorien müssen nun noch angepasst werden. In der Optionalklammer steht noch die Kategorie, von der die Hilfskategorie stammt. Diese muss durch die Definition der Kategorie, von der die betrachtete Hilfskategorie abstammt, ersetzt werden.

Achtung: Die nun erzeugte Grammatik entspricht nicht mehr der ursprünglichen Grammatik und ist damit falsch – sie klammert rechts. Wir müssen die neu eingeführten Hilfskategorien beim Erstellen des Parsers gesondert behandeln, um diesen Fehler zu korrigieren.

1.3 Prozeduren schreiben

(a) Zunächst müssen Datentypen definiert werden – einer für die Tokens und einer für die zusammengebauten Objekte (`exp`).

(b) Für jede Kategorie der Grammatik muss eine Prozedur erstellt werden.

(c) Diese Prozeduren müssen verschränkt rekursiv geschrieben werden – also mit dem Schlüsselwort `and` – da jeder Prozedur die Prozedur, die zur in der Grammatik darunter stehenden Kategorie gehört, bekannt sein muss.

(d) Um "normale" Kategorien ohne Hilfskategorien zu realisieren, müssen die Kategorien von links nach rechts abgearbeitet werden.

(i) Zunächst wird ein Objekt (1) der ersten in der Grammatik vorkommenden Kategorie weggeparst. Dazu wird die entsprechende zu dieser Kategorie zugehörige Prozedur auf die `token list` angewendet.

(ii) Um die Optionalklammern zu realisieren, muss anhand des ersten Wortes der Restliste, die der obige Prozeduraufruf zurückgibt, entschieden werden, ob man die Optionalklammern braucht, also ob ein Ausdruck gemäß dem, was in den Optionalklammern steht, folgt.

- Ist dies der Fall, so muss ein Objekt (2) der nächsten in der Grammatik vorkommenden Kategorie weggeparst werden und anschließend werden die Objekte (1) und (2) zu einem neuen Objekt zusammengefügt.
- Ist dies nicht der Fall, ist das Objekt (1) bereits das fertige Objekt der Kategorie.

(iii) Das gebaute Objekt muss nun als Tupel gemeinsam mit der zuletzt erhaltenen Restliste zurückgegeben werden.

(e) Kategorien, die Hilfskategorien aufrufen, können wie folgt realisiert werden:

(i) Zunächst wird ein Objekt der ersten in der Grammatik vorkommenden Kategorie weggeparst. Dazu wird wieder die entsprechende zu dieser Kategorie zugehörige Prozedur auf die `token list` angewendet.

(ii) Auf das Ergebnis des obigen Prozeduraufrufs wird die zur Hilfskategorie zugehörige Prozedur erneut angewendet.

Achtung: Die zur Hilfskategorie zugehörige Prozedur nimmt einen anderen Argumenttypen als die Prozeduren für "normale" Kategorien! Statt einer `token list` nimmt sie ein Tupel bestehend aus einem (fertigen) Objekt (3) und einer `token list`.

(f) Zur Realisierung von Hilfskategorien, geht man ähnlich wie bei der Umsetzung der Optionalklammern bei einer Kategorie ohne Hilfskategorie vor. Anhand des ersten Wortes der (übergebenen) Restliste muss entschieden werden, ob die Optionalklammern genutzt werden.

- Ist dies der Fall, so muss ein Objekt (4) der nächsten in der Grammatik stehenden Kategorie weggeparst werden und anschließend werden die Objekte (3) und (4) zu einem neuen Objekt zusammengefügt. Um die Rekursion aus der Grammatik zu realisieren, muss die Hilfskategorie erneut rekursiv auf das Tupel bestehend aus dem gerade neu zusammengefügten Objekt und der Restliste angewendet werden.
- Ist dies nicht der Fall, so kann das aus einem fertigen Objekt und der Restliste bestehende Argument einfach wieder zurückgegeben werden, da nichts mehr zum Abarbeiten der Kategorie getan werden muss, und das Objekt (3) bereits das fertige Objekt der Kategorie ist.

(g) Zu guter Letzt müssen noch die primitiven Ausdrücke realisiert werden. Dabei unterscheidet man im Vorgehen danach, was das erste Element der Restliste ist.

(i) Handelt es sich um eine Konstante, so muss das passende konstante Objekt gemeinsam mit der Restliste zurückgegeben werden.

(ii) Handelt es sich um eine öffnende Klammer, so muss überprüft werden, ob danach ein korrekter Ausdruck gemäß der Grammatik und anschließend eine schließende Klammer folgt. Dazu geht man wie folgt vor:

i. Zunächst wird ein Objekt (5) der nächsten in der Grammatik stehenden Kategorie – in diesem Fall die oberste, da ein "Rundlauf" durch Klammern möglich sein muss und daher zwischen den Klammern die oberste Kategorie steht – weggeparst. Dazu wird erneut die entsprechende zu dieser Kategorie zugehörige Prozedur auf die übergebene Liste, aus der die öffnende Klammer an erster Stelle entfernt wurde, angewendet.

ii. Anhand des ersten Wortes der Restliste des Ergebnisses der obigen Prozeduranwendung muss nun entschieden werden, ob eine schließende Klammer folgt.

- Ist dies der Fall, so handelt es sich um einen korrekt geklammerten Ausdruck und das Zusammenbauen des Objektes (5) – dies geschieht per Rekursion durch wegparsen dieses Objekts – war legitim. Da die Klammern nicht explizit in das zusammengebaute Objekt aufgenommen werden – dies geschieht nur durch die Verschachtelung der einzelnen Teilobjekte – können diese entfernt werden, sodass das Tupel bestehend aus dem Objekt (5) und der Restliste, aus der die schließende Klammer an erster Stelle entfernt wurde, zurückgegeben werden kann.
- Ist dies nicht der Fall, so handelt es sich nicht um einen korrekt geklammerten Ausdruck, da auf den Ausdruck nach der öffnenden Klammer keine schließende Klammer mehr folgt. Somit kann der Ausdruck auch nicht zu einem Objekt zusammengebaut werden und das Werfen einer Ausnahme ist notwendig.

(iii) Handelt es sich nicht um etwas nach der Grammatik Erlaubtes, so liegt ein Fehler im Ausdruck vor und eine Ausnahme muss geworfen werden.

2 Beispiel

2.1 Aufgabenstellung

Gegeben sind folgende Operatoren: $\wedge, \rightarrow, \neg$, sowie die Konstanten *true* und *false*. Dabei sind $\wedge$ und $\rightarrow$ binäre Operatoren, und $\neg$ ein unärer Operator. $\wedge$ klammert links und $\rightarrow$ klammert rechts; $\wedge$ klammert stärker als $\rightarrow$, und $\neg$ klammert am stärksten. Es dürfen explizite Klammern gesetzt werden.

Geben Sie eine konkrete Grammatik an, definieren Sie einen passenden Datentypen `exp` und schreiben Sie einen Parser.

2.2 Grammatik aufstellen

(a) Für jeden Operator (mit unterschiedlich starker Klammerung) wird eine Kategorie erstellt. Diese Kategorien werden so geordnet, dass die Kategorie des am schwächsten klammernden Operators ganz oben und die des am stärksten klammernden Operators ganz unten steht. Zusätzlich wird eine Kategorie für primitive Ausdrücke hinzugefügt.

$$\begin{aligned} \mathit{exp} &::= \\ \mathit{uexp} &::= \\ \mathit{mexp} &::= \\ \mathit{aexp} &::= \end{aligned}$$

Hier bezeichnen *exp* die $\rightarrow$-Ausdrücke, *uexp* die $\wedge$-Ausdrücke, *mexp* die $\neg$-Ausdrücke und *aexp* die primitiven Ausdrücke.

(b) Nun werden die Kategorien mit Leben gefüllt. Wichtig dabei ist, dass aus jeder Kategorie ein Ausdruck der darunter liegenden Kategorie abgeleitet werden kann. Jeder *exp*-Ausdruck muss also ein *uexp*-Ausdruck sein können, jeder *uexp*-Ausdruck ein *mexp*-Ausdruck und so weiter.

$$\begin{aligned} \mathit{exp} &::= \mathit{uexp} \mid \ldots \\ \mathit{uexp} &::= \mathit{mexp} \mid \ldots \\ \mathit{mexp} &::= \mathit{aexp} \mid \ldots \\ \mathit{aexp} &::= \end{aligned}$$

Nur die unterste und damit stärkste Kategorie braucht keine passende Regel.

(c) Nun leiten wir für alle Kategorien, die Operatoren beschreiben, die Regeln her. Dazu müssen wir je nach Art des Operators unterschiedlich vorgehen:

- Binäre Operatoren:

  Generell wird ein Operator-Ausdruck durch einen Ausdruck derselben und der nächst niedrigeren Kategorie gebildet, die durch den Operator verbunden werden. Man muss jedoch zwischen rechts- und linksklammernden Operatoren unterscheiden:

  – rechtsklammernde Operatoren:

    Die Rechtsklammerung von Operatoren wird durch Rechtssrekursion erreicht. Das bedeutet, dass zunächst der Ausdruck der niedrigeren Kategorie kommt, dann der Operator und dann der Ausdruck der eigenen Kategorie.

    Damit folgt für *exp*:

    $$\mathit{exp} ::= \mathit{uexp} \mid \mathit{uexp}\ \text{“}\rightarrow\text{”}\ \mathit{exp}$$

  – linksklammernde Operatoren:

    Die Linksklammerung wird analog dazu durch Linksrekursion erreicht. Das heißt, dass zunächst der Ausdruck der eigenen Kategorie geschrieben wird, gefolgt von dem Operator und dem Ausdruck der nächstniedrigeren Kategorie.

    Damit folgt für *uexp*:

    $$\mathit{uexp} ::= \mathit{mexp} \mid \mathit{uexp}\ \text{“}\wedge\text{”}\ \mathit{mexp}$$

- Unäre Operatoren:

  Unäre Operatoren-Ausdrücke werden (im Allgemeinen) durch den Operator selbst und einen Ausdruck der nächstniedrigeren Kategorie gebildet.

  Damit folgt für *mexp*:

$$mexp ::= aexp \mid \text{“}\neg\text{”}\ aexp$$

(d) Nun entwickeln wir die Regeln für primitive Ausdrücke.

Primitive Ausdrücke sind entweder Konstanten oder geklammerte (beliebige) Ausdrücke (diese können aus *exp* abgeleitet werden).

Damit folgt für *aexp*:

$$aexp ::= \text{“}false\text{”} \mid \text{“}true\text{”} \mid \text{“(”}\ exp\ \text{“)”}$$

(e) Nun haben wir alle Regeln entwickelt. Gemeinsam bilden diese die vollständige Grammatik:

$$\begin{aligned} exp &::= uexp \mid uexp\ \text{“}\rightarrow\text{”}\ exp \\ uexp &::= mexp \mid uexp\ \text{“}\wedge\text{”}\ mexp \\ mexp &::= aexp \mid \text{“}\neg\text{”}\ aexp \\ aexp &::= \text{“}false\text{”} \mid \text{“}true\text{”} \mid \text{“(”}\ exp\ \text{“)”} \end{aligned}$$

(f) Um die Grammatik noch einfacher lesbar zu machen und ein Stück näher zur RA-Tauglichkeit zu kommen, schreiben wir die Grammatik noch um, indem wir Optionalklammern verwenden. Aus

$$uexp \mid uexp\ \text{“}\rightarrow\text{”}\ exp$$

wird also

$$uexp\ [\text{“}\rightarrow\text{”}\ exp]$$

Wir erhalten die folgende Grammatik:

$$\begin{aligned} exp &::= uexp\ [\text{“}\rightarrow\text{”}\ exp] \\ uexp &::= [uexp\ \text{“}\wedge\text{”}]\ mexp \\ mexp &::= [\text{“}\neg\text{”}]\ aexp \\ aexp &::= \text{“}false\text{”} \mid \text{“}true\text{”} \mid \text{“(”}\ exp\ \text{“)”} \end{aligned}$$

2.3 Grammatik RA-tauglich machen

Leider ist diese Grammatik noch nicht komplett RA-tauglich, sodass wir diese nicht direkt in einen Parser umsetzen können.

Dank der Optionalklammern ist gut ersichtlich, welche der Kategorien RA-tauglich sind und welche nicht. RA-tauglich sind diese, bei denen man anhand des ersten Zeichens in der Optionalklammer entscheiden kann, ob noch der Ausdruck in der Optionalklammer folgt, oder ob die durch die Optionalklammer dargestellte Alternative nicht gewählt wird.

Vereinfacht gesagt, sind Kategorien, bei denen zu Beginn der Optionalklammern ein konkretes Zeichen steht, meistens RA-tauglich und solche, bei denen eine andere Kategorie steht (in den meisten Fällen) nicht.

In unserem Fall wäre folglich nur die Kategorie *uexp* problematisch. Um dieses Problem zu umgehen, dreht man die Kategorien einfach um.

Aus

$$uexp ::= [uexp\ \text{“}\wedge\text{”}]\ mexp$$

wird also

$$uexp ::= mexp\ [\text{“}\wedge\text{”}\ uexp]$$

Dadurch wird nun zunächst die Eigenschaft der Linksklammerung nicht mehr erfüllt, diese wird jedoch später beim Schreiben des Parsers wieder hergestellt. Um das zu erleichtern, führt man für die geänderte Kategorie eine Hilfskategorie ein, die den Ausdruck in Optionalklammern ersetzt:

$$\begin{aligned} uexp &::= mexp\ uexp' \\ uexp' &::= [\text{“}\wedge\text{”}\ uexp] \end{aligned}$$

Schließlich ersetzen wir die Kategorie in den Optionalklammern noch einmal durch ihre Definition.

$$\begin{aligned} uexp &::= mexp\ uexp' \\ uexp' &::= [\text{“}\wedge\text{”}\ mexp\ uexp'] \end{aligned}$$

Diese beiden formalen Umschreibungen ändern zunächst erst einmal nichts an der Grammatik – sie klammert also weiterhin falsch. Sie erlauben jedoch im Folgenden eine kleinschrittigere Bearbeitung während des Parsens. Dadurch werden wir die Möglichkeit haben, die Linksklammerung während des Parsing-Vorgangs wieder herzustellen.

Nun haben wir eine Grammatik erstellt, auf der aufbauend im Folgenden der Parser geschrieben werden kann. Diese sieht wie folgt aus:

$$\begin{aligned} exp &::= uexp\ [\text{“}\rightarrow\text{”}\ exp] \\ uexp &::= mexp\ uexp' \\ uexp' &::= [\text{“}\wedge\text{”}\ mexp\ uexp'] \\ mexp &::= [\text{“}\neg\text{”}]\ aexp \\ aexp &::= \text{“}false\text{”}\ \mid\ \text{“}true\text{”}\ \mid\ \text{“(”}\ exp\ \text{“)”} \end{aligned}$$

2.4 Prozeduren schreiben

(a) Bevor die Parser-Prozeduren geschrieben werden können, müssen zunächst ein paar Datentypen gesichtet oder erstellt werden.

Geparst wird eine Liste von Tokens, also von Konstanten. Diese Tokens müssen alle verwendeten Operatoren, Konstanten und Klammern umfassen. In unserem Beispiel verwenden wir den folgenden Datentyp:

```
datatype token = LPAR | RPAR | NEG | AND | IMP | TRUE | FALSE
```

Außerdem wird ein Datentyp benötigt, der die geparsten Ausdrücke darstellt, also eine baumartige Struktur hat. Für unser Beispiel bietet sich der folgende Datentyp an:

```
datatype exp = Neg of exp
             | And of exp * exp | Imp of exp * exp
             | True | False
```

(b) Für jede Kategorie der Grammatik erstellen wir eine Prozedur. Dazu verwenden wir verschränkte Rekursion, da sich die Prozeduren gegenseitig aufrufen werden. Statt des Schlüsselwortes **`fun`** verwenden wir also (abgesehen von der ersten Prozedur) **`and`**.

Die Prozeduren nehmen als Argumente (bis auf eine Ausnahme, auf die wir später eingehen) eine Liste von Tokens und liefern einen geparsten Ausdruck (vom Typ `exp`) und eine Restliste von Tokens, auf der weiter geparst werden kann.

Wir haben also folgendes Grundgerüst:

```
fun exp ts = ...
and uexp ts =  ...
and uexp' ...
and mexp ts =
and aexp ts = ...
```

(c) Für die Prozedurrümpfe orientieren wir uns nun an den Kategorien der Grammatik. Wir werden im Folgenden dafür sorgen, dass die Prozedur zu jeder Kategorie aus einer Liste von Tokens einen entsprechenden Ausdruck dieser Kategorie (und eine Restliste) zurück liefert.

(i) Die Kategorie *exp* besagt, dass zunächst mindestens ein *uexp*-Ausdruck vorkommen muss. Dieser Ausdruck, sowie die Liste, die entsteht, wenn die Tokens, die diesen Ausdruck bilden, entfernt wurden, wird von der Prozedur `uexp` zurückgegeben (davon gehen wir hier einfach einmal aus, denn dies ist die Funktionalität, die am Ende gegeben sein soll).

Durch einen Aufruf von `uexp` können wir einen Ausdruck der Kategorie *uexp* parsen und mithilfe von Pattern Matching können wir auf das Ergebnis dieses Aufrufes zugreifen:

```
case (uexp ts) of (e, tr) ⇒ ...
```

Im nächsten Schritt muss entschieden werden, ob noch ein *exp*-Ausdruck folgt, also ob die Optionalklammer gewählt wurde. Dazu muss das erste Token der Ergebnisliste (`tr`) `IMP` sein. Falls dies nicht der Fall ist, ist man fertig und kann das Paar aus *uexp*-Ausdruck und Restliste zurück geben (das ist legitim – denn laut Grammatik ist ja jeder *exp*-Ausdruck ein *uexp*-Ausdruck). Es ergibt sich:

```
case (uexp ts) of (e, IMP :: tr) ⇒ ...
                | (e, ts') ⇒ (e, ts')
```

Wenn ein `IMP` folgt, muss ein weiterer *exp*-Ausdruck gefunden werden. Hier vertrauen wir auf das Wunder der Rekursion, nach dem ein solcher Ausdruck durch die Anwendung von `exp` geliefert wird.

```
val (e', tr') = exp tr
```

Die beiden so gewonnen Ausdrücke können nun mithilfe des Konstruktors `Imp` zusammengebaut werden:

```
fun exp ts = (case (uexp ts) of (e, IMP :: tr) ⇒ let val (e', tr') = exp tr
                                                 in (Imp (e, e'), tr')
                                                 end
                              | (e, ts') ⇒ (e, ts'))
```

Damit ist die Prozedur gemäß der Grammatik fertig gestellt.

(ii) Im Falle der Kategorie *uexp* müssen wir beachten, dass wir die Linksklammerung wieder herstellen wollen.

Zunächst müssen wir gemäß der entsprechenden Kategorie einen *mexp*-Ausdruck parsen. Mit diesem Ausdruck müssen wir nun jedoch anders verfahren.

Und hier kommt die Hilfskategorie ins Spiel: Die Prozedur der Hilfskategorie nimmt nicht nur eine Liste von Tokens, sondern ein Paar aus einem Ausdruck und einer solchen Liste als Argument.

Die Hilfsprozedur überprüft, ob (eingeleitet durch ein $\wedge$) weitere *uexp*-Ausdrücke folgen und parst diese dann schrittweise. Dies erlaubt es, dass in der Hilfskategorie-Prozedur zunächst zwei geparste Ausdrücke mittels des Konstruktors `And` zusammengefügt werden, bevor nach weiteren *uexp*-Ausdrücken gesucht wird.

Das Ergebnis von `(mexp ts)` übergeben wir also der Hilfsprozedur: `uexp' (mexp ts)`. Dort wird nun zuerst überprüft, ob nach weiteren *mexp*-Ausdrücken gesucht werden muss. Dies passiert, wie bei `exp`, indem das erste Element der Restliste auf das Vorkommen von `AND` überprüft wird.

Ist dies nicht der Fall, wird der bereits geparste *mexp*-Ausdruck zurück gegeben. Dies ist erlaubt, denn jeder *mexp*-Ausdruck ist ja laut Grammatik auch ein *uexp*-Ausdruck.

Ist dies der Fall, wird ein weiterer *mexp*-Ausdruck geparst. Dieser wird mit dem übergebenen Ausdruck zu einem neuen Ausdruck zusammen gebaut. Anschließend muss geprüft werden, ob weitere durch $\wedge$ getrennte *mexp*-Ausdrücke existieren. Diese Funktion übernimmt eben genau die Prozedur `uexp'`. Diese kann also rekursiv auf den eben erstellten Ausdruck und die Restliste angewendet werden. Folgt irgendwann kein `AND` mehr, wird der bisher erstellte Ausdruck samt Restliste einfach zurück gegeben.

In SML-Code übersetzt haben wir also:

```
and uexp ts = uexp' (mexp ts)
and uexp' (e, AND :: tr) = let val (e', tr') = mexp tr
                           in uexp' (And (e, e'), tr')
                           end
  | uexp' s = s
```

Diese Prozeduren folgen nicht genau der Grammatik, da sie den Baum falsch aufbauen: Der rekursive Aufruf von `uexp'` bestimmt keinen Unterbaum des bereits erkannten Baums – stattdessen wird ein schon erkannter Baum an `uexp'` übergeben, und dort, falls noch ein `AND` kommt, *über* den erkannten Baum gesetzt.

Durch diesen "Fehler" beim Bauen des Parser korrigieren wir den Fehler unserer Grammatik, die $\wedge$ fälschlicherweise rechts klammert. Unser Parser klammert also links, wie gefordert.

(iii) Die Prozedur zur Kategorie *mexp* wird prinzipiell analog zur ersten Prozedur gebildet. Hier kann direkt anhand des ersten Zeichens der übergebenen Liste entschieden werden, ob ein Ausdruck mit dem Konstruktor `Neg` gebildet werden muss. Falls das erste Zeichen ein `NEG` ist, so muss ein *aexp*-Ausdruck mittels `aexp` geparst werden, und der so gelieferte Ausdruck mit dem einstelligen `Neg`-Konstruktor zusammengebaut werden. Ansonsten genügt es, einen *aexp*-Ausdruck zu parsen und diesen direkt zurück zu geben.

```
and mexp (NEG :: tr) = let val (e, ts) = aexp tr
                       in (Neg e, ts)
                       end
  | mexp ts  = aexp ts
```

(iv) Um die primitiven Ausdrücke und damit die Kategorie `aexp` umzusetzen, erfolgt eine Unterscheidung darüber, was am Anfang der übergebenen `token list` steht. Hierzu wird Pattern Matching verwendet.

Zunächst betrachten wir den Fall, also dass die Konstante *true* vorliegt, dass also die übergebene `token list` mit dem token `TRUE` beginnt. Da es sich um eine Konstante und somit um eine Art Basisfall handelt, müssen keine weiteren Teilobjekte geparst werden, sondern das Bauen des Objekts `True` kann direkt vorgenommen werden.

```
and aexp (TRUE :: tr) = (True, tr)
```

Ebenso verfährt man mit der Konstante *false*:

```
  | aexp (FALSE :: tr) = (False, tr)
```

Als nächstes muss der Fall betrachtet werden, dass eine linke Klammer gelesen wird, die übergebene `token list` also mit `LPAR` beginnt. Wie in der Grammatik zu sehen ist, muss auf eine öffnende Klammer ein *exp*-Ausdruck folgen. Daher muss zunächst ein solcher Ausdruck geparst werden, indem die Prozedur `exp` auf die Restliste angewendet wird.

```
  | aexp (LPAR :: tr) = ... exp tr ...
```

Durch das Wunder der Rekursion liefert dieser Prozeduraufruf das Tupel zurück, das aus dem fertig zusammengesetzten Objekt des *exp*-Ausdrucks nach der öffnenden Klammer sowie der danach übrig gebliebenen Restliste besteht. Dieses Ergebnis soll natürlich nicht verloren gehen. Ferner muss auch noch überprüft werden, ob nach diesem *exp*-Ausdruck eine schließende Klammer folgt, da es sich sonst nicht um einen korrekt geklammerten Ausdruck handelt. Dies kann mit einem `case`-Konstrukt realisiert werden, in dem überprüft wird, ob das erste Element der aus dem Aufruf der Prozedur `exp` resultierenden Restliste eine schließende Klammer ist.

```
  | aexp (LPAR :: tr) = case (exp tr) of (a, RPAR :: tr') ⇒ ...
```

In diesem Fall handelt es sich um einen korrekt geklammerten Ausdruck und das schon fertige Objekt a wurde bereits zu einem gültigen Baum zusammengebaut und kann gemeinsam mit der Restliste `tr'` zurückgegeben werden. Intuitiv werfen wir die Klammern also einfach weg, da diese im fertigen Baum nicht explizit vorkommen, sondern lediglich durch die Verschachtelung der einzelnen Teilobjekte realisiert werden.

```
  | aexp (LPAR :: rs) = case (exp tr) of (a, RPAR :: tr') ⇒ (a, tr')
```

Damit der `case`-Ausdruck erschöpfende Muster hat, muss auch noch der Fall betrachtet werden, dass die sich aus dem Prozeduraufruf ergebende Restliste nicht mit dem token `RPAR` beginnt. In diesem Fall handelt es sich nicht um einen korrekt geklammerten Ausdruck, da auf eine öffnende Klammer und einen *exp*-Ausdruck keine schließende Klammer folgt. In diesem Fall kann natürlich auch kein Objekt aus dem fehlerhaften Ausdruck zusammengebaut werden, weshalb in diesem Fall eine Ausnahme geworfen werden muss.

```
  | aexp (LPAR :: tr) = case (exp tr) of (a, RPAR :: tr') ⇒ (a, tr')
                                       | _ ⇒ raise Error "parse"
```

Anschließend muss noch der Fall betrachtet werden, dass es sich bei dem ersten Token der Liste weder um die Konstanten *true* oder *false*, noch um eine öffnende Klammer handelt. Dann kann es sich nicht um einen zu der gegebenen Grammatik korrekten Ausdruck handeln, da laut dieser nur *true*, *false* und geklammerte *exp*-Ausdrücke primitive Ausdrücke sind, und eine Ausnahme muss geworfen werden.

```
  | aexp _ = raise Error "parse"
```

Zu guter Letzt müssen diese vier Fälle noch zu einer einzigen Prozedur verschmolzen werden, die die Kategorie der primitiven Ausdrücke realisiert:

```
and aexp (TRUE :: tr) = (True, tr)
  | aexp (FALSE :: tr) = (False, tr)
  | aexp (LPAR :: tr) = (case (exp tr) of (a, RPAR :: tr') ⇒ (a, tr')
                                        | _ ⇒ raise Error "parse")
  | aexp _ = raise Error "parse"
```

2.5 Der fertige Parser

```
exception Error of string

datatype token = LPAR | RPAR | NEG | AND | IMP | TRUE | FALSE

datatype exp = Neg of exp
             | And of exp * exp | Imp of exp * exp
             | True | False

fun exp ts = (case (uexp ts) of (e, IMP :: tr) ⇒ let val (e', tr') = exp tr
                                                 in (Imp (e, e'), tr')
                                                 end
                              | (e, ts') ⇒ (e, ts'))
and uexp ts = uexp' (mexp ts)
and uexp' (e, AND :: tr) = let val (e', tr') = mexp tr
                           in uexp' (And (e, e'), tr')
                           end
  | uexp' s = s
and mexp (NEG :: tr) = let val (e, ts) = aexp tr
                       in (Neg e, ts)
                       end
  | mexp ts = aexp ts
and aexp (TRUE  :: tr) = (True, tr)
  | aexp (FALSE :: tr) = (False, tr)
  | aexp (LPAR  :: tr) = (case (exp tr) of (a, RPAR :: tr') ⇒ (a, tr')
                                         | _ ⇒ raise Error "parse")
  | aexp _ = raise Error "parse"
```