Dr. David Sabel
Institut für Informatik

Fachbereich Informatik und Mathematik
Johann Wolfgang Goethe-Universität Frankfurt am Main

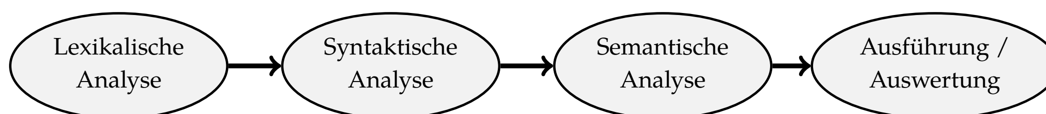
Praktikum BKSPP

Sommersemester 2012

Aufgabenblatt Nr. 4

Abgabe: Mittwoch, 11. Juli 2012

Auf diesem Aufgabenblatt wird ein Interpreter für Ausdrücke des Lambda-Kalküls implementiert. Ein *Compiler* gliedert sich in die Phasen *Lexikalische Analyse*, *Syntaktische Analyse*, *Semantische Analyse*, *Zwischencodeerzeugung*, *Codeoptimierung* und *Codegenerierung* auf (siehe z.B. [SS12]). *Interpreter* erzeugen im Allgemeinen keinen Zwischen- und Maschinencode, sondern führen das Quellprogramm nach Abschluss der drei Analysenphasen aus. D.h. die *Phasen eines Interpreters* lassen sich durch das folgende Schaubild beschreiben:



Die lexikalische Analyse (der *Lexer*) erhält als Eingabe den Quelltext (als String) und zerlegt diesen in syntaktische Einheiten (so genannte *Token*) und entfernt dabei Zeilenkommentare, Leerzeichen, Zeilenumbrüche etc.

Die syntaktische Analyse (der *Parser*) erhält die Ausgabe des Lexers (den Tokenstrom) und prüft, ob diese ein syntaktisch korrektes Programm darstellt (i.A. ob das Programm zu einer durch eine kontextfreie Grammatik festgelegten Sprache gehört). Es gibt verschiedene Parse-Methoden.

Die semantische Analyse dient dazu semantische Fehler im Programm zu erkennen. Dazu gehört z.B. in imperativen Programmiersprachen der Test, ob alle Variablen vor ihrer Benutzung deklariert werden. In statisch getypten Programmiersprachen findet die Typüberprüfung im Rahmen der semantischen Analyse statt. Wir werden im Rahmen dieses Aufgabenblattes die semantische Analyse zunächst überspringen, am Ende jedoch einen Typcheck für den einfach getypten Lambda-Kalkül implementieren.

Für die letzte Phase – der Ausführung – werden wir eine Auswertung der Ausdrücke des Lambda-Kalküls implementieren

1 Syntax des Lambda-Kalküls, Lexen und Parsen

In diesem Abschnitt führen wir zunächst die Syntax des Lambda-Kalküls formal ein und widmen uns anschließend der lexikalischen und syntaktischen Analyse.

Der Lambda-Kalkül wurde von Alonzo Church in den 1930er Jahren eingeführt [Chu41][1]. Sei V eine unendliche Menge von Variablennamen und sei $x \in V$. Die Syntax für Ausdrücke des Lambda-Kalküls ist durch folgende Grammatik gegeben:

$$e, e_i \in E := x \mid \lambda x.e \mid e_1 \mathbin{@} e_2$$

D.h. Variablen sind Ausdrücke, *Abstraktionen* $\lambda x.e$ sind Ausdrücke, wobei durch den Lambda-Binder λx die Variable x innerhalb des Unterausdrucks (dem *Rumpf*) e gebunden wird, d.h. umgekehrt: Der Gültigkeitsbereich von x ist e.

Weitere Ausdrücke sind *Anwendungen* $(e_1 \mathbin{@} e_2)$ wobei e_1 und e_2 beliebige Ausdrücke sind. Im Gegensatz zu Haskell (und auch der ursprünglichen Definition des Lambda-Kalküls) stellen wir dabei Anwendungen explizit durch den binären Infix-Operator $@$ dar.

Lambda-Ausdrücke sind anonyme Funktionen, d.h. Funktionen, die keinen Namen haben, z.B. kann man die Identitätsfunktion $id(x) = x$ durch die Abstraktion $\lambda x.x$ im Lambda-Kalkül darstellen. Argumente von Funktionen können selbst Funktionen sein, z.B. kann die Anwendung der Identitätsfunktion auf sich selbst (d.h. $id(id)$) im Lambda-Kalkül als $(\lambda x.x) \mathbin{@} (\lambda x.x)$ geschrieben werden.

1.1 Lexikalische Analyse

Für die Darstellung im Programm verwenden wir anstelle des Symbols λ einen Backslash, d.h. statt $\lambda x.e$ schreiben wir $\backslash x.e$. Außerdem erlauben wir beliebige Klammerung (d.h. z.B. ist $(\lambda x.x)$ oder (x) erlaubt). Als zusätzliches Feature erlauben wir Zeilenkommentare, die mit `--` eingeleitet werden, sowie beliebige Zeilenumbrüche und Leerzeichen. Sämtliche Leerzeichen und Zeilenumbrüche trennen stets syntaktische Einheiten. Variablennamen dürfen nur aus Buchstaben und Zahlen bestehen und beginnen stets mit einem Buchstaben. Zur Erläuterung: Der Text `x y` wird als zwei Token (eines für die Variable `x` und eines für die Variable `y`) erkannt, der Text `xy` wird als ein Token (die Variable `xy`) erkannt und der Text `1xs` führt zu einem Fehler.

Während der lexikalischen Analyse wird der Quelltext eingelesen und die relevanten syntaktischen Einheiten werden als Liste von Token zurückgegeben. D.h. der Lexer entfernt Kommentare und Leerzeichen. Für Token soll der folgende Datentyp verwendet werden:

```
data Token = TokBackSl TokPos        -- '\'
           | TokDot    TokPos        -- '.'
           | TokBopen  TokPos        -- '('
           | TokBclose TokPos        -- ')'
           | TokApp    TokPos        -- '@'
           | TokVar    TokPos String -- Variablen
deriving(Eq,Show)
```

[1] Es gibt reichlich Literatur zum Lambda-Kalkül. Ein Hauptnachschlagewerk ist [Bar84], eine gute Einführung ist z.B. in [Han04] zu finden.

D.h. es gibt Token für den Backslash, den Punkt, öffnende und schließende Klammern, für den Anwendungsoperator, sowie für Variablen (diese werden durch Strings dargestellt). Die Konstruktoren des Token-Typs erhalten als zusätzliches Argument einen Wert vom Typ `TokPos`, der definiert ist als

```
type TokPos = (Int,Int) -- (Zeile,Spalte)
```

Dieses Paar von Zahlen gibt an, wo im Quelltext das Token gefunden wurde. Es dient zur komfortablen Behandlung von Fehlern, denn der Benutzer des Interpreters möchte nicht nur wissen, dass sein eingegebenes Programm syntaktisch falsch ist, sondern möchte auch wissen, wo der syntaktische Fehler sitzt.

Aufgabe 1 *Implementieren Sie im Modul* `Lexer`[2] *eine Funktion* `lexInput :: String -> [Token]`, *die einen Quelltext erhält und diesen lexikalisch analysiert und eine Tokenliste als Ausgabe zurück gibt. Sollten unbekannte Symbole auftreten, so soll mit der* `error`*-Funktion ein Fehler generiert werden, der die Position des Fehlers enthält.*

Hinweis: *Da sie die Zeilen- und Spaltennummer der einzelnen Token stets mitberechnen müssen, empfiehlt es sich, eine Hilfsfunktion der Art*

```
lexInputMitZuS ausdruck zeile spalte = ....
```

zu definieren. Die Funktion `lexInput` *ruft dann diese Hilfsfunktion mit den Werten 1 und 1 für die anfängliche Zeile und die anfängliche Spalte auf.*

1.2 Syntaktische Analyse

Für die interne Darstellung von Ausdrücken des Lambda-Kalküls steht der Datentyp `Expr` im Modul `LambdaLang` zur Verfügung:

```
data Expr  = Lambda VarName Expr   -- \x.e
           | App Expr Expr         -- e1 @ e2
           | Var VarName
deriving(Eq)

type VarName = String
```

Z.B. kann die Eingabe `\x1.(x1 x2)` mit diesem Datentyp dargestellt werden als `Lambda "x1" (App (Var "x1") (Var "x2"))`.

Die Aufgabe des Parsers ist es nun, den von der lexikalischen Analyse erhaltenen Tokenstrom in den Datentyp `Expr` zu konvertieren, bzw. eine Fehlermeldung zu generieren, falls dies nicht möglich ist.

[2] Das Modul ist bereits vorgegeben, und enthält die Datentypdefinition für Token

Das Schreiben eines Parsers per Hand ist i.A. ziemlich aufwändig. Deshalb gibt es so genannte *Parsergeneratoren*, die anhand der kontextfreien Grammatik und weiteren Festlegungen (z.B. Assoziativitäten und Prioritäten von Operatoren) automatisch einen Parser erzeugen. Die Ausgabe des Parses ist i.A. ein Parse- oder auch ein Syntaxbaum. Der wohl bekannteste Parsergenerator ist Yacc (yet another compiler compiler), der einen Parser in der Programmiersprache C erzeugt. Für Haskell steht der Parsergenerator Happy (haskell.org/happy) zur Verfügung. Happy erwartet als Eingabe die kontextfreie Grammatik und einige weitere Angaben und erzeugt daraus einen Shift-Reduce-Parser. Im Anhang A finden Sie ein Beispiel mit Erläuterungen zur Benutzung von Happy.

Aufgabe 2 *Erstellen Sie einen Parser mithilfe des Generators Happy, der einen Tokenstrom parst und als Ausgabe den Syntaxbaum als Objekt vom Typ* `Expr` *liefert. Dabei muss gelten:*

- *Der Anwendungsoperator @ ist linksassoziativ, d.h. die Eingabe* `e1 @ e2 @ e3` *steht vollgeklammert für* `((e1 @ e2) @ e3)` *und* **nicht** *für* `(e1 @ (e2 @ e3))`.
- *Der Rumpf einer Abstraktion reicht so weit wie möglich. Z.B. wir daher* `\x.e1 @ e2` *vollgeklammert als* `(\x.(e1 @ e2))` *erkannt und* **nicht** *als* `((\x.e1) @ e2)`.

Um diese Konventionen einzuhalten empfiehlt es sich, die Assoziativitäten und die Prioritäten der „Operatoren“ . und @ in der Parserdefinition festzulegen.

Implementieren Sie in der Parserdefinition die Funktion `happyError` *in geeigneter Weise, so dass beim Auftreten eines Parsefehlers eine Fehlermeldung* mit Angabe der Position des fehlerhaften Tokens.

1.3 Alpha-Umbenennung

Wie bereits erwähnt ist x im Rumpf e einer Abstraktion $\lambda x.e$ gebunden. Um die Gültigkeitsbereiche von Variablen formal festzuhalten, definieren wir die Funktionen FV und GV. Für einen Ausdruck e ist $GV(e)$ die Menge seiner *gebundenen Variablen* und $FV(e)$ die Menge seiner *freien Variablen*, wobei diese (induktiv) durch die folgenden Regeln definiert sind:

$$
\begin{array}{lcllcl}
FV(x) & = & x & GV(x) & = & \emptyset \\
FV(\lambda x.e) & = & FV(e) \setminus \{x\} & GV(\lambda x.e) & = & GV(e) \cup \{x\} \\
FV(e_1 \mathbin{@} e_2) & = & FV(e_1) \cup FV(e_2) & GV(e_1 \mathbin{@} e_2) & = & GV(e_1) \cup GV(e_2)
\end{array}
$$

Gilt $FV(e) = \emptyset$ für einen Ausdruck e, so sagen wir e ist *geschlossen* oder auch e ist ein *Programm*. Ist ein Ausdruck e nicht geschlossen, so nennen wir e *offen*.

Um die operationale Semantik des Lambda-Kalküls zu definieren, benötigen wir den Begriff der *Substitution*: Wir schreiben $e_1[e_2/x]$ für den Ausdruck, der entsteht, indem alle freien Vorkommen der Variable x in e_1 durch den Ausdruck e_2 ersetzt werden. Um Namenskonflikte bei dieser Ersetzung zu vermeiden, nehmen wir an, dass $x \notin GV(e_1)$ gilt.

Unter dieser Annahme kann man die Substitution formal durch die folgenden Gleichungen definieren:

$$\begin{array}{lcl} x[e/x] & = & e \\ y[e/x] & = & y, \text{ falls } x \neq y \\ (\lambda y.e_1)[e_2/x] & = & \lambda y.(e_1[e_2/x]) \\ (e_1 \mathbin{@} e_2)[e_3/x] & = & (e_1[e_3/x] \mathbin{@} e_2[e_3/x]) \end{array}$$

Z.B. ergibt $(\lambda x.z \mathbin{@} x)[(\lambda y.y)/z]$ den Ausdruck $(\lambda x.((\lambda y.y) \mathbin{@} x))$.

Kontexte C sind Ausdrücke, wobei genau ein Unterausdruck durch ein Loch (dargestellt mit $[\cdot]$) ersetzt ist. Man kann Kontexte auch mit der folgenden kontextfreien Grammatik definieren:

$$C \in \mathbf{C} = [\cdot] \mid \lambda x.C \mid (C \mathbin{@} e) \mid (e \mathbin{@} C),$$

wobei e ein Ausdruck und x eine Variable ist. In Kontexte kann man Ausdrücke *einsetzen*, um so einen neuen Ausdruck zu erhalten. Sei C ein Kontext und e ein Ausdruck. Dann ist $C[e]$ der Ausdruck, der entsteht, indem man in C anstelle des Loches den Ausdruck e einsetzt. Diese Einsetzung kann Variablen einfangen. Betrachte als Beispiel den Kontext $C = \lambda x.[\cdot]$. Dann ist $C[\lambda y.x]$ der Ausdruck $\lambda x.(\lambda y.x)$. Die freie Variable x in $\lambda y.x$ wird beim Einsetzen eingefangen.

Nun können wir α-Umbenennung definieren: Ein *α-Umbenennungsschritt* hat die Form:

$$C[\lambda x.e] \xrightarrow{\alpha} C[\lambda y.e[y/x]] \text{ falls } y \notin GV(\lambda x.e) \cup FV(\lambda x.e)$$

Die reflexiv-transitive Hülle solcher α-Umbenennungen heißt *α-Äquivalenz*. Wir unterscheiden α-äquivalente Ausdrücke nicht. Z.B. sind $\lambda x.x$ und $\lambda y.y$ α-äquivalent, während $\lambda x.y$ und $\lambda y.x$ nicht α-äquivalent sind.

Im Folgenden nehmen wir an, dass in einem Ausdruck alle gebundenen Variablen unterschiedliche Namen haben und dass Namen gebundener Variablen stets verschieden sind von Namen freier Variablen. Mithilfe von α-Umbenennungen kann diese Konvention (kurz bezeichnet mit DVC[3]) stets eingehalten werden.

Aufgabe 3 *Implementieren Sie ein Modul* `Util`*, welches eine Funktion*

```
renameExpr :: [VarName] -> Expr -> (Expr,[VarName])
```

enthält, die eine Liste von (neuen) Variablennamen und einen Ausdruck erhält und den Ausdruck in einen α-äquivalenten Ausdruck umformt, der die DVC einhält. Die Rückgabe der Funktion ist ein Paar bestehend aus dem umbenannten Ausdruck und den nicht verwendeten frischen Variablennamen.

Hinweis: *Für die Implementierung genügt es sämtliche gebundenen Variablen umzubenennen, indem man rekursiv den Ausdruck von oben nach unten abarbeitet. Hierbei muss man sich merken, wie diese Umbenennung aussieht. Z.B. betrachte den Ausdruck $\lambda x.\lambda y.(x \mathbin{@} y)$: Will man*

[3] DVC = Distinct Variable Convention

$(x \ @ \ y)$ umbenennen, so muss man wissen, wie man zuvor x und y an den jeweiligen Bindern umbenannt hat. Es bietet sich daher an eine Hilfsfunktion zu definieren, die neben den Argumenten von `renameExpr` *als zusätzliches Argument die aktuelle Umbenennung speichert. Diese kann durch eine Liste vom Typ* `[(VarName,VarName)]` *repräsentiert werden.*

Aufgabe 4 *Fügen Sie dem Modul* `Util` *eine Funktion*

```
substitute :: [VarName] -> VarName -> Expr -> Expr -> (Expr,[VarName])
```

hinzu, welche die Substitution durchführt, d.h. der Aufruf `substitute` $vars\ x\ e_1\ e_2$ *ersetzt im Ausdruck e_2 alle Vorkommen von x durch* **umbenannte Kopien** *von e_1. Dies entspricht der Substitution $e_2[e_1/x]$ wobei gleichzeitig die DVC eingehalten wird, d.h.: Erfüllt e_2 die DVC, so erfüllt auch das Resultat von* `substitute` *die DVC.*

Das erste Argument von `substitute` *ist eine Liste frischer Variablennamen, die für die Umbenennungen verwendet wird. Die Rückgabe von* `substitute` *ist ein Paar bestehend aus dem Ausdruck und den nicht verwendeten Variablennamen.*

Hinweis: *Es bietet sich an, für die Umbenennungsschritte die zuvor implementierte Funktion* `renameExpr` *zu verwenden. Als Beispiel betrachte $\lambda w.(z \ @ \ z)[\lambda x.\lambda y.x/z]$ und die Liste* `["a","b","c","d","e","f"]` *als Liste neuer Variablennamen. Die mit* `substitute` *durchgeführte Substitution sollte ergeben:*

$$\lambda w.((\lambda a.(\lambda b.a)) \ @ \ (\lambda c.(\lambda d.c)))$$

wobei die Liste `["e","f"]` *die Liste der nicht verbrauchten Namen darstellt.*

Um sicherzustellen, dass die Liste der neuen Variablennamen nur frische Namen enthält, bietet sich z.B. die Liste `["_x" ++ show i | i <- [1..]]` *an.*

2 Operationale Semantik, Auswertung von Ausdrücken

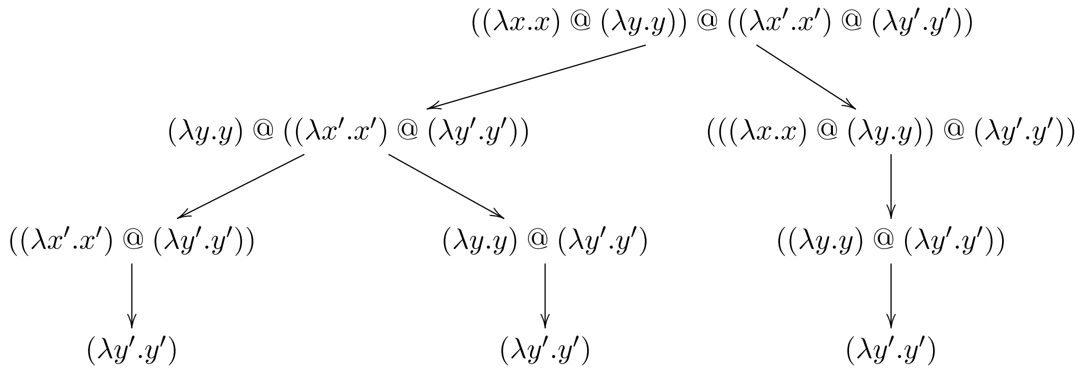
In diesem Abschnitt soll ein kleiner Interpreter für den Lambda-Kalkül implementiert werden, der die call-by-name-Auswertung durchführt. Die call-by-name Auswertung zeichnet sich dadurch aus, dass die Funktionsanwendung ohne vorherige Argumentauswertung durchgeführt wird.

Der call-by-name Lambda-Kalkül kennt nur eine so genannte *Reduktionsregel*: Die (call-by-name)-β-Reduktion. Sie wertet die Anwendung von Funktionen (d.h. Abstraktionen) auf Argumente aus und ist definiert durch die folgende Regel:

$$(\beta) \qquad (\lambda x.e_1) \ @ \ e_2 \to e_1[e_2/x]$$

Wenn $e_1 \xrightarrow{\beta} e_2$, so sagt man auch e_1 *reduziert unmittelbar* zu e_2. Z.B. kann man reduzieren $(\lambda x.x \ @ \ x) \ @ \ (\lambda y.\lambda z.z) \xrightarrow{\beta} (\lambda y.\lambda z.z) \ @ \ (\lambda y.\lambda z.z) \to \lambda z.z$ Es reicht i.A. jedoch nicht, die (β)-Reduktion ausschließlich auf oberster Ebene eines Ausdrucks durchzuführen, da z.B.

$e = ((\lambda x.x) \mathbin{@} (\lambda y.y)) \mathbin{@} ((\lambda x'.x') \mathbin{@} (\lambda y'.y'))$ auf oberster Ebene nicht reduzierbar ist (die oberste Anwendung hat keine Abstraktion links, sondern wiederum eine Anwendung), aber noch β-Reduktionen möglich sind. Erlaubt man die (β)-Reduktion an beliebiger Position im Ausdruck, so kann man e auf verschiedene Arten reduzieren, wie der folgende Baum aller Auswertungsmöglichkeiten zeigt:



Die *call-by-name Auswertung* sucht immer den am weitesten oben und am weitesten links stehenden Redex[4]. Formal kann man die call-by-name Auswertung (auch Normalordnungsreduktion genannt) mithilfe von Reduktionskontexten definieren.

Call-by-name Reduktionskontexte R werden durch die folgende Grammatik definiert:

$$R ::= [\cdot] \mid (R \mathbin{@} \mathbf{Expr})$$

Wenn $e_1 \xrightarrow{\beta} e_2$, dann ist $R[e_1] \xrightarrow{name} R[e_2]$ ein *call-by-name-Reduktionsschritt* (oft auch als Normalordnungsreduktion bezeichnet).

Zum Implementieren bietet sich ein alternatives Verfahren an, dass mehr oder weniger gerade der Idee entspricht, den Reduktionskontext R während der Redexsuche auf einem Stack abzulegen, und diesen im Anschluss nach und nach abzuarbeiten.

Die Auswertung operiert daher auf einem Paar bestehend aus Ausdruck und einem Stack (wobei die Einträge auf dem Stack wiederum Ausdrücke sind). Wir benutzen Listennotation für den Stack. Für einen Ausdruck e startet die Auswertung mit $(e, [])$ und wendet anschließend die folgenden beiden Regeln solange iteriert an, bis keine Regel mehr anwendbar ist:

$$\begin{array}{ll} \text{(push)} & ((e_1 \mathbin{@} e_2), S) \to (e_1, e_2 : S) \\ \text{(take)} & (\lambda x.e_1), e_2 : S) \to (e_1[e_2/x], S) \end{array}$$

Die Regel (push) legt das Argument einer Anwendung auf den Stack. Die Regel (take) führt gerade die (β)-Reduktion durch: Wenn eine Abstraktion gefunden wurde, wird das oberste Element vom Stack entnommen und als Argument für die Abstraktion verwendet.

Es ist keine Regel anwendbar, wenn einer der folgenden Fälle auftritt:

- Der Zustand ist von der Form $(\lambda x.e, [])$. In diesem Fall wurde der Ausdruck erfolgreich zu einer Abstraktion reduziert.

[4]Redex = reducible expression, der reduzierte Unterausdruck

- Der Zustand ist von der Form (x, S), d.h. es wurde eine freie Variable in Reduktionsposition gefunden.

Wir betrachten als Beispiel die Auswertung des Ausdrucks $(((\lambda x.\lambda y.x) @ ((\lambda w.w) @ (\lambda z.z))) @ (\lambda u.u))$:

	Ausdruck	Stack
	$(((\lambda x.\lambda y.x) @ ((\lambda w.w) @ (\lambda z.z))) @ (\lambda u.u))$	$[]$
$\xrightarrow{(push)}$	$((\lambda x.\lambda y.x) @ ((\lambda w.w) @ (\lambda z.z)))$	$[\lambda u.u]$
$\xrightarrow{(push)}$	$\lambda x.\lambda y.x$	$[((\lambda w.w) @ (\lambda z.z)), \lambda u.u]$
$\xrightarrow{(take)}$	$\lambda y.((\lambda w.w) @ (\lambda z.z))$	$[\lambda u.u]$
$\xrightarrow{(take)}$	$((\lambda w.w) @ (\lambda z.z))$	$[]$
$\xrightarrow{(push)}$	$\lambda w.w$	$[\lambda z.z]$
$\xrightarrow{(take)}$	$\lambda z.z$	$[]$

Aufgabe 5 *Implementieren Sie ein Modul* `Evaluator` *und dort eine Funktion*

```
evaluate :: [VarName] -> Expr -> Expr
```

die eine Liste (frischer) Variablennamen und einen Ausdruck erhält und versucht diesen Ausdruck bis zu einer Abstraktion auszuwerten. Im Erfolgsfall wird diese Abstraktion als Ergebnis geliefert, anderenfalls (eine freie Variable wird gefunden) soll eine Fehlermeldung ausgegeben werden. Beachten Sie, dass es durchaus Ausdrücke gibt, für welche der Interpreter nicht anhält (der (ungetypte) Lambda-Kalkül ist Turing-mächtig).

Verwenden Sie für die Implementierung das vorgestellte Verfahren mit einem Zustand bestehend aus Ausdruck und Stack. Den Stack können Sie dabei durch eine Liste vom Typ `[Expr]` *darstellen.*

3 Semantische Analyse, Typcheck

Im Rahmen der Semantischen Analyse werden wir für den Lambda-Kalkül einen Typcheck bezüglich des einfach-getypten Lambda-Kalküls[5] durchführen.

3.1 Einfaches Typsystem

Einfache Typen für den (einfach getypten) Lambda-Kalkül werden durch die folgende Grammatik gebildet:

$$\tau, \tau_i \in \mathit{Typ} ::= 0 \mid \tau_1 \rightarrow \tau_2$$

Hierbei stellt 0 eine Typkonstante dar (wir hätten auch irgendein anderes Symbol wählen können).

[5] im Englischen: simply-typed lambda calculus

Sei Γ eine Typumgebung, d.h. eine Abbildung von Variablen auf Typen. Die Notation $\Gamma \vdash e : \tau$ bedeutet, dass es unter der Typumgebung Γ möglich ist, für den Ausdruck e den Typ τ herzuleiten. Die folgenden Typherleitungsregeln verwenden die so genannte „Bruchstrichnotation“, d.h. die Regeln sind von der Form $\frac{\text{Voraussetzung}}{\text{Konsequenz}}$. Will man die Konsequenz schließen, so muss man stets zeigen, dass die Voraussetzung erfüllt ist. Eine leere Voraussetzung ist immer erfüllt, d.h. Regeln der Form $\frac{}{\text{Konsequenz}}$ sind Axiome. Die Typherleitungsregeln sind:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \text{ und } \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1 @ e_2) : \tau_2} \qquad \frac{\Gamma \cup \{x \mapsto \tau_1\} \vdash e : \tau_2}{\Gamma \vdash (\lambda x.e) : \tau_1 \to \tau_2} \qquad \frac{}{\Gamma \cup \{x \mapsto \tau\} \vdash x : \tau}$$

Wir betrachten als Beispiel eine Typisierung des Ausdrucks $(\lambda x.x) @ (\lambda y.y)$ und starten mit leeren Typumgebung Γ:

$$\frac{\dfrac{\dfrac{}{\{x \mapsto (0 \to 0)\} \vdash x : (0 \to 0)}}{\emptyset \vdash \lambda x.x : (0 \to 0) \to (0 \to 0)} \text{ und } \dfrac{\dfrac{}{\{y \mapsto 0\} \vdash y : 0}}{\emptyset \vdash \lambda y.y : 0 \to 0}}{\emptyset \vdash (\lambda x.x) @ (\lambda y.y) : 0 \to 0}$$

Diese Herleitung sollte man von unten nach oben lesen. Wir haben damit gezeigt, dass $(\lambda x.x) @ (\lambda y.y)$ mit dem Typ $0 \to 0$ typisierbar ist. Allerdings ist das nicht der einzige Typ, der möglich ist. Dies liegt an der wahllos getroffenen Auswahl $y \mapsto 0$ bei der Verwendung der Regel für die Abstraktion. Tatsächlich sind die so angegeben Typregeln nicht richtig algorithmisch, da man in der Regel für Abstraktionen den Typ τ_1 für x raten bzw. durchprobieren müsste.

Deshalb erweitern wir das Verfahren und benötigen hierfür Typen, die auch Typvariablen und Typgleichungen der Form $\tau_1 =^? \tau_2$ zulassen. Die Syntax für erweiterte Typen ist

$$\tau, \tau_i ::= 0 \mid \tau_1 \to \tau_2 \mid \alpha$$

wobei α eine Typvariable aus einer unendlichen Menge von Typvariablen ist. Sei in den folgenden Regeln α **je eine neue Typvariable** und E, E_i Mengen von Typgleichungen. Die Notation $\Gamma \vdash e : \tau, E$ bedeutet bei diesen Regeln: Der Ausdruck e ist unter der Typannahme Γ mit dem Typ $\gamma(\tau)$ typisierbar, wenn γ sich aus der Lösung der Gleichungen in E ergibt. Die Regeln sind:

$$\frac{\Gamma \vdash e_1 : \tau_1, E_1 \text{ und } \Gamma \vdash e_2 : \tau_2, E_2}{\Gamma \vdash (e_1 @ e_2) : \alpha, \{\tau_1 =^? \tau_2 \to \alpha\} \cup E_1 \cup E_2} \qquad \frac{\Gamma \cup \{x : \alpha\} \vdash e : \tau, E}{\Gamma \vdash (\lambda x.e) : \alpha \to \tau, E} \qquad \frac{}{\Gamma \cup \{x \mapsto \tau\} \vdash x : \tau, \emptyset}$$

Die eigentlich Typherleitung geschieht mit der folgenden Regel. Hierbei muss die Lösung γ eines Typgleichungssystems mithilfe von Unifikation berechnet werden (siehe nächster Abschnitt).

$$\frac{\emptyset \vdash e : \tau, E \text{ und } \gamma \text{ ist Lösung von } E}{\emptyset \vdash e : \rho(\gamma(\tau))} \text{ für jede Grundsubstitution } \rho$$

Dabei ist eine *Grundsubstitution* eine Substitution, die jede Typvariable durch einen Grundtypen (Typ ohne Typvariablen) ersetzt. In der Implementierung müssen wir die Unifikation durchführen, jedoch keine (der i.A. beliebig vielen) Grundsubstitutionen anwenden, da uns nur interessiert, ob der Ausdruck typisierbar ist, d.h. wir geben $\gamma(\tau)$ als „Typ“ zurück, ohne ρ anzuwenden.

3.2 Unifikationsalgorithmus

Wir stellen einen Unifikationsalgorithmus vor. Dieser verwaltet zwei Mengen von Typgleichungen. Zum Einen die bereits gelösten Gleichungen G und zum Anderen die noch zu lösenden Gleichungen E. Der Algorithmus startet mit leerer Menge G und den Gleichungen E und wendet anschließend die folgenden Regeln solange an, bis entweder ein Fehler auftritt, oder die Menge E leer ist: Die Notation $F[\tau/\alpha]$ meint hierbei: Ersetze in allen Gleichungen in F die Typvariable α durch den Typ τ.

Löschregel	$(G, E \cup \{\tau =^? \tau\}) \to (G, E)$, wobei τ eine Typvariable oder $\tau = 0$
Dekomposition	$(G, E \cup \{\tau_1 \to \tau_2 =^? \tau_3 \to \tau_4\}) \to (G, E \cup \{\tau_1 =^? \tau_3, \tau_2 =^? \tau_4)$
Gelöst	$(G, E \cup \{\alpha =^? \tau\}) \to (G[\tau/\alpha] \cup \{\alpha = \tau\}, E[\tau/\alpha])$ wobei α Typvariable und α kommt nicht in τ vor
Occurs Check	$(G, E \cup \{\alpha =^? \tau\}) \to \textit{Fail}$, falls α in τ vorkommt
Fail	$(G, E \cup \{\tau_1 \to \tau_2 =^? 0\}) \to \textit{Fail}$
Fail	$(G, E \cup \{0 =^? \tau_1 \to \tau_2\}) \to \textit{Fail}$
Orientierung	$(G, E \cup \{\tau =^? \alpha\}) \to (G, E \cup \{\alpha =^? \tau\})$ wobei α Typvariable, τ keine Typvariable

Liefert der Unifikationsalgorithmus *Fail*, so existiert keine Lösung und der ursprüngliche Ausdruck ist nicht typisierbar. Im Erfolgsfall (E ist leer) enthält G die Substitution γ.

3.3 Beispiele

Wir betrachten die Typisierung von $(\lambda x.x)$ @ $(\lambda y.y)$ mit dem erweiterten Algorithmus

$$\dfrac{\dfrac{\dfrac{}{\{x : \alpha_1\} \vdash x : \alpha_1, \emptyset}}{\emptyset \vdash \lambda x.x : \alpha_1 \to \alpha_1, \emptyset} \quad \text{und} \quad \dfrac{\dfrac{}{\{y : \alpha_2\} \vdash y : \alpha_2, \emptyset}}{\emptyset \vdash \lambda y.y : \alpha_2 \to \alpha_2, \emptyset}}{\emptyset \vdash (\lambda x.x) \mathbin{@} (\lambda y.y) : \alpha_3, \{\alpha_1 \to \alpha_1 =^? (\alpha_2 \to \alpha_2) \to \alpha_3\}}$$

Nun müssen wir das Gleichungssystem $\{\alpha_1 \to \alpha_1 =^? (\alpha_2 \to \alpha_2) \to \alpha_3\}$ unifizieren, wir machen dies tabellarisch, wobei in der ersten Spalte die gelösten Gleichungen und in der zweiten Spalte die noch zu lösenden Gleichung stehen:

G	E	nächste verwendete Regel
	$\alpha_1 \to \alpha_1 =^? (\alpha_2 \to \alpha_2) \to \alpha_3$	(Dekomposition)
	$\alpha_1 =^? \alpha_2 \to \alpha_2$ $\alpha_1 =^? \alpha_3$	(Gelöst)
$\alpha_1 =^? \alpha_3$	$\alpha_3 =^? \alpha_2 \to \alpha_2$	(Gelöst)
$\alpha_1 =^? \alpha_2 \to \alpha_2$ $\alpha_3 =^? \alpha_2 \to \alpha_2$		

Dies ergibt als Unifikator $\gamma = \{\alpha_1 \mapsto \alpha_2 \to \alpha_2, \alpha_3 \mapsto \alpha_2 \to \alpha_2\}$ und $(\lambda x.x) \mathbin{@} (\lambda y.y)$ lässt sich typisieren mit $\gamma(\alpha_3) = \alpha_2 \to \alpha_2$[6].

Als weiteres Beispiel betrachten wir die Typisierung von $\lambda x.(x \mathbin{@} x)$:

$$\dfrac{\dfrac{\overline{\{x \mapsto \alpha_1\} \vdash x : \alpha_1, \emptyset} \quad \text{und} \quad \overline{\{x \mapsto \alpha_1\} \vdash x : \alpha_1, \emptyset}}{\{x \mapsto \alpha_1\} \vdash (x \mathbin{@} x) : \alpha_2, \{\alpha_1 \stackrel{?}{=} \alpha_2 \to \alpha_1\}}}{\emptyset \vdash \lambda x.(x \mathbin{@} x) : \alpha_1 \to \alpha_2, \{\alpha_1 \stackrel{?}{=} \alpha_2 \to \alpha_1\}}$$

Die Unifikation schlägt jedoch fehl:

G	E	nächste verwendete Regel
	$\alpha_1 \stackrel{?}{=} \alpha_2 \to \alpha_1$	(Occurs Check)
Fail		

D.h. der Ausdruck $\lambda x.(x \mathbin{@} x)$ ist nicht typisierbar.

3.4 Implementierung

Wir stellen (erweiterte) Typen in Haskell durch den folgenden Datentyp `Type` dar:

```
data Type = Unit             -- 0
          | Type  :-> Type   -- t1 -> t2
          | TVar TypVarName  -- Typvariable
  deriving (Eq)

type TypVarName = String
```

Hierbei repräsentiert der Konstruktor `Unit` die 0, `:->` ist ein Infix-Konstruktor, der den Typpfeil repräsentiert und Typvariablen werden durch Strings dargestellt. Für Umgebungen und Gleichungen verwenden wir Typsynonyme:

```
type Umgebung  = [(TypVarName, Type)]
type Gleichung = (Type,Type)
```

Aufgabe 6 *Implementieren Sie im Modul* `TypeCheck` *eine Funktion*

```
unify :: [Gleichung] -> [Gleichung] -> Maybe [Gleichung]
```

die den Unifikationsalgorithmus wie folgt implementiert: Die Funktion erwartet die Gleichungssysteme G und E und liefert bei Fehlschlagen das Ergebnis `Nothing` *und bei Erfolg das gelöste Gleichungssystem verpackt mit* `Just`.

Hinweis: *Eine Hilfsfunktion für die Ersetzung $F[\tau/\alpha]$ vom Typ* `ersetze :: TypVarName -> Type -> [Gleichung] -> [Gleichung]` *ist eventuell sinnvoll.*

[6] Wie bereits erwähnt ist dieser Typ noch kein einfacher Typ, er meint vielmehr: $(\lambda x.x)$ $(\lambda y.y)$ lässt sich mit allen Typen $\rho(\alpha_2 \to \alpha_2)$ typisieren, wobei ρ einen Grundtyp für α_2 einsetzt.

Aufgabe 7 *Implementieren Sie im Modul* `TypeCheck` *den Typcheck für den Lambda-Kalkül. Hierfür ist sinnvoll zunächst eine Funktion*

```
tcheck :: [TypVarName] -> Umgebung -> Expr -> (Type,[Gleichung],[TypVarName])
```

zu implementieren, die eine Liste neuer Typvariablennamen, eine Umgebung Γ *und einen Ausdruck erhält und als Ergebnis den hergeleiteten Typ, die dabei gewonnenen Typgleichungen sowie die nicht verwendeten Typvariablennamen zurückliefert.*

Eine Hauptfunktion `runTCheck :: Expr -> Type` *ist bereits vorgegeben. Diese führt zunächst* `tcheck` *durch und unifiziert die gewonnenen Gleichungen anschließend mit* `unify`. *Schließlich wendet sie die erhaltene Subsitution an und liefert daher den hergeleiteten Typ, d.h. das Ergebnis von* `runTCheck` *ist* $\gamma(\tau)$ *bzw. eine Fehlermeldung im Falle der Untypisierbarkeit.*

Testen Sie Ihre Implementierung u.A. mit den Ausdrücken

- `\x.((\a.\b.\c.c) @ (x @ (\u.u)) @ (x @ (\w.w)))`
- `\x.((\a.\b.\c.c) @ (x @ (\u.\v.u)) @ (x @ (\w.\z.z)))`
- `\x.((\a.\b.\c.c) @ (x @ (\u.\v.(u @ v))) @ (x @ (\w.\z.(z @ w))))`

4 Interpreter

Aufgabe 8 *Implementieren Sie ein Konsolenprogramm, dass einen Quelltext für Lambda-Ausdrücke aus einer Datei einliest, nacheinander die lexikalische, die syntaktische, und die semantische Analyse (Typcheck) durchführt und schließlich das Programm auswertet und den Typ und das Resultat der Auswertung im Interpreter ausdruckt. Fügen Sie eine weitere Option ein, so dass die Ausführung ohne Typcheck direkt durchgeführt wird. Testen Sie das Programm ausgiebig.*

Literatur

[Bar84] H. P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York, 1984.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[Han04] Chris Hankin. *An introduction to lambda calculi for computer scientists*. Number 2 in Texts in Computing. King's College Publications, London, UK, 2004.

[SS12] Manfred Schmidt-Schauß. Skript zur Vorlesung „Grundlagen der Programmierung 2". http://www.informatik.uni-frankfurt.de/~prg2/SS2012/index.html, (Sommersemester 2012). Kapitel 5.

Anhang

A Aufbau eines Happy-Skripts

In diesem Abschnitt werden wir beschreiben, wie eine Parserspezifikation für `Happy` aussieht. Hierfür verwenden wir als Beispiel arithmetische Ausdrücke, mit der (mehrdeutigen!) Grammatik:

$$
\begin{array}{rcl}
\textbf{Expr} & ::= & \textbf{Expr} + \textbf{Expr} \\
 & | & \textbf{Expr} - \textbf{Expr} \\
 & | & \textbf{Expr} * \textbf{Expr} \\
 & | & \textbf{Expr} \; / \; \textbf{Expr} \\
 & | & (\; \textbf{Expr} \;) \\
 & | & \textbf{Zahl}
\end{array}
$$

Die Produktionen für **Zahl** geben wir nicht an, da wir das eigentlich Parsen der Zahl dem Lexer überlassen werden. Wir nehmen an, dass der Lexer einen Tokenstrom vom Typ `[Token]` liefert, wobei `Token` definiert ist als:

```
data Token = TokenInt Int  -- Zahl
           | TokenPlus     -- '+'
           | TokenMinus    -- '-'
           | TokenTimes    -- '*'
           | TokenDiv      -- '/'
           | TokenOB       -- '('
           | TokenCB       -- ')'
```

Jedes Happy-Skript besteht aus bis zu 4 Teilen.

Der erste (optionale) Teil ist ein Block Haskell-Code, der von geschweiften Klammern umschlossen wird. Dieser Block wird unverändert an den Anfang der durch `Happy` generierten Datei gesetzt. Für gewöhnlich stehen hier der Modulkopf und `import`-Befehle.

```
{
module Calc where
import Char
}
```

Der nächste Teil enthält verschiedene Direktiven, die `Happy` unbedingt benötigt:

- `%name` *NAME* bezeichnet den Namen der Parserfunktion. Unter diesem Namen kann der Parser also später aufgerufen werden.
- `%tokentype {` *TYPE* `}` Dies ist der Ausgabetyp des Lexers und damit der Eingabetyp des Parsers.

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

Ein Beispiel ist:

```
%name calculator
%tokentype { Token }
%token
        int        { TokenInt $$ }
        '+'        { TokenPlus }
        '-'        { TokenMinus }
        '*'        { TokenTimes }
        '/'        { TokenDiv }
        '('        { TokenOB }
        ')'        { TokenCB }
```

Der Parser wird somit den Namen `calculator` erhalten, die verwendeten Tokens sind vom Datentyp `Token` und für die Zuweisung der Terminals an die Tokens gilt: Links stehen die Terminals, rechts in geschweiften Klammern die Token.

Das Symbol $$ ist ein Platzhalter, das den Wert des Tokens repräsentiert. Normalerweise ist der Wert eines Tokens der Token selbst, mit $$ wird ein Teil des Tokens als Wert spezifiziert. Im Beispiel ist der Wert des Tokens `TokenInt` *zahl* die Zahl.

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.

```
%%
Expr :: { Expr }
Expr : Expr '+' Expr { Plus  $1 $3}
     | Expr '-' Expr { Minus $1 $3}
     | Expr '*' Expr { Times $1 $3}
     | Expr '/' Expr { Div $1 $3 }
     | '(' Expr ')'  { $2 }
     | int           { Number $1}
```

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 `$i` wird auf den Wert von i-ten Terminals bzw. Nonterminals 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 Objekt vom Typ `Expr`. Wie wir nun schon sehen, werden die Zahlen nicht mittels der Grammatik geparst, sondern direkt vom Token `TokenInt` bzw.

Terminal `int` übernommen. Dies ist eine Vereinfachung, d.h. wir überlassen das korrekte Parsen der Zahlen dem Lexer (er erstellt ja das Token `TokenInt`).

Der vierte Teil eines `Happy`-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 `happyError` 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.) Der Typ von `happyError` ist dabei

```
happyError :: [Token] -> a
```

Für unser Beispiel müssen auch die Datentypen `Expr` und `Token` irgendwo definiert werden, d.h. entweder in einem der Haskell-Code-Abschnitte der Parserspezifikationsdatei oder in externen Haskell-Dateien, die dann importiert werden.

Der Vollständigkeit halber, der Rest der der Parserspezifikation für unser Beispiel:

```
{
happyError :: [Token] -> a
happyError _ = error "parse error!"

data Token = TokenInt Int
           | TokenPlus
           | TokenMinus
           | TokenTimes
           | TokenDiv
           | TokenOB
           | TokenCB

data Expr   = Plus    Expr Expr
            | Minus   Expr Expr
            | Times   Expr Expr
            | Div     Expr Expr
            | Number  Int
  deriving(Show)

lexer :: String -> [Token]
lexer [] = []
lexer ('+':cs) = TokenPlus : lexer cs
lexer ('-':cs) = TokenMinus : lexer cs
lexer ('*':cs) = TokenTimes : lexer cs
lexer ('/':cs) = TokenDiv : lexer cs
lexer ('(':cs) = TokenOB : lexer cs
lexer (')':cs) = TokenCB : lexer cs
```

```
lexer (c:cs)
 | isSpace c = lexer cs
 | isDigit c = lexNum (c:cs)
 | otherwise = error ("parse error, can't lex symbol " ++ show "c")

lexNum cs = TokenInt (read num) : lexer rest
              where (num,rest) = span isDigit cs
}
```

Mit der so erstellten Parserspezifikation (die Dateien haben die Endung `.y` bzw `.ly` falls es sich um ein literate skript handelt), kann nun mittels `happy` der Parser generiert werden:

```
happy example.y
shift/reduce conflicts:  16
```

Die Meldung der Konflikte sagt uns, dass etwas nicht stimmt. Der erstellte Parser weiß in manchen Situationen nicht was er tun soll. Der Grund hierfür liegt in der Mehrdeutigkeit unserer Grammatik. Wir könnten nun eine eindeutige (aber auch komplizierte Grammatik) benutzen, aber `happy` bietet uns die Möglichkeit Präzedenz und Assoziativität von Operatoren am Ende der Direktiven festzulegen. Hierbei gilt

- `%left` *Terminal(e)* legt fest, dass diese Terminale links-assoziativ sind (d.h. ein Ausdruck $a \otimes b \otimes c$ wird als $(a \otimes b) \otimes c$ aufgefasst).
- `%right` *Terminal(e)* legt fest, dass diese Terminale rechts-assoziativ sind (d.h. ein Ausdruck $a \otimes b \otimes c$ wird als $a \otimes (b \otimes c)$ aufgefasst).
- `%nonassoc` *Terminal(e)* legt fest, dass diese Terminale nicht assoziativ sind (d.h. ein Ausdruck $a \otimes b \otimes c$ kann nicht geparst werden und es tritt ein Fehler auf)

Die Präzedenz der Terminale gegenüber den anderen Terminalen wird durch die Reihenfolge `%left`, `%right` und `%nonassoc` Direktiven festgelegt, wobei „früher“ „weniger Präzedenz“ bedeutet. Nach dem Einfügen der Zeilen

```
%left '+' '-'
%left '*' '/'
```

direkt vor `%%`, hat der Parser keine Konflikte mehr und parst arithmetische Ausdrücke entsprechend der üblichen geltenden Konventionen (Punkt vor Strich usw.).