Eliza

Sven Eric Panitz

14. Februar 1996

Wir wollen hier die kleine Implementierung des simulierten Psychoanalythikers von Mark P. Jones 1982 in Gofer realisiert annotiert vorstellen.

Ein mit diesem Programm erzeugter Beispieldialog ist:

```
Hi! I'm Eliza. I am your personal therapy computer.
Please tell me your problem.

hello!
How do you...please state your problem.

My name is Peter Hacker.
Names don't interest me.

Oh, I see. Sorry!
Please don't apologise!

I want to write better programs.
Why do you want to write better programs?

Because my programs are very buggy.
Is that the real reason?

Yes I think so.
You seem quite positive.

I don't like my style of programming.
Don't you really like your style of programming?

I think I could write better programs.
Do you really think so?

Maybe I should write my programs in gofer.
You don't seem quite certain.

Maybe C programs run faster.
Why the uncertain tone?

I'm not so sure about this topic.
Did you come to me because you are not so sure about this topic?

I always think of changing the programming language.
Can you think of a specific example?

I have no problem in writung a parser in Haskell.
Everybody has some Problem, don't you think so.
```

```
Yes! I have a problem writung parsers in Cobol.
I'm not sure I understand you fully.

Can You write a parser in Cobol?
Don't you believe that I can write a parser in Cobol?

Cobol is just plain shit.
Oh you are a vulgarian.
Why don't you relax and stop using such words.

Sorry!
Apologies are not necessary.

I don't like Cobol.
Why don't you like Cobol?

Because I can't write a parser in Cobol.
How do you know you can't write a parser in Cobol?

Oh you're right. I never tried to.
What makes you think I am right. you never tried to?

I never tried to write a program in Cobol.
I see.

I will try it. Thank You. Bye bye.
Good Bye! Think about what we were saying.
```

Das Programm wird gegenüber anderen in der Vorlesung vorgestellten Programmen keine sehr komplizierten Konzepte, wie z.B. Monaden oder eigene Typ- und Konstruktorklassen enthalten. Der Text soll auch als Beispiel dienen, wie man Programme im *literate*-Stil dokumentieren kann.

Unser Programm soll abwechselnd eine Zeile aus dem *stdin* lesen und darauf eine Antwort auf das *stdout* erzeugen. Wir brauchen eine *Mainfunktion*, die diesen Dialog realisiert.

```
> main  :: Dialogue
```

Eine sehr bequeme Methode diesen interaktiven Frage- und Antwortprozess in Gofer zu verwirklichen, ist die *Preludefunktion* `interact`.

```
> main   = interact eliza
```

Zur Erinnerung: sie ist im Prelude definiert durch:

```
interact :: (String -> String) -> Dialogue
interact = readChan stdin exit (\x -> appendChan stdout (f x) exit done)
```

Es handelt sich also um eine Funktion, die den Kanal *stdin* öffnet, einen String von dort einliest (dieses können mehrere Zeilen beliebiger Text sein) und dann aus der Eingabe eine Ausgabe erzeugt, die auf den Kanal *stdout* wieder ausgedruckt wird. Um die Ausgabe zu erzeugen, braucht die Funktion ein Methode, wie aus der Eingabe die Ausgabe erzeugt wird. Hierzu hat die Funktion `interact` ein Argument, das einen String in einen String transformiert.

In einer strikten Sprache könnte die Funktion `interact` für unsere Zwecke nicht verwendet werden, da erst alle Eingaben gelesen werden würden, um dann erst die Ausgabe komplett auf den Bildschirm zu schreiben. Da Gofer aber verzögert ausgewertet wird, wird sobald ein Teil der Antwort erzeugt werden konnte, dieser auch ausgedruckt. Damit ist das Frage- und Antwortspiel, das wir benötigen automatisch realisiert.

Auf diese Weise haben wir unser interaktives Programm sehr früh von allen I/O-Operationen befreit.

Das Hauptprogramm muß jetzt also in den Funktionsargument von `interact` codiert sein. Hierzu schreiben wir die Funktion `eliza`. In dieser Funktion werden wir eine Sequenz von Operationen auf die Eingabe ausführen:

- Da wir auf jede mit *Return* abgeschlossene Anfrage einen Antwortsatz erzeugen wollen, werden wir die Eingabe in Zeilen splitten mit `lines`.
- Wir werden der Einfachheit halber Satzzeichen nicht berücksichtigen. Daher löschen wir aus allen Eingabezeilen die Satzzeichen mit `map (words . trim)`. `trim` ist dabei eine simple selbstdefinierte Filterfunktion, die Satzzeichen löscht:

```
> trim   = foldr cons "" . dropWhile (`elem` punct)
>          where x `cons` xs | x `elem` punct && null xs = []
>                            | otherwise                 = x : xs
>                punct = [' ', '.', '!', '?', ',']
```

- In einem nächsten Schritt löschen wir leere Sätze, sprich Zeilen, die nur aus *Return* bestehen.
- In `session initial []` werden wir die eigentlichen Antworten auf die Sätze erzeugen. Wir behandel das anschließend.
- Die Ausgabe soll mit einen kurzen Starttext beginnen. Hierzu definieren wir uns einen adequaten Starttext:

```
> starttext = "\n\
>              \Hi! I'm Eliza. I am your personal therapy computer.\n\
>              \Please tell me your problem.\n\
>              \\n"
```

Insgesammt besteht unsere Funktion `eliza` aus der Funktionskonkatenation der soeben vorgestellten Operationen:

```
> eliza   ::  String -> String
> eliza   =  (starttext ++)
>             . session initial []
>             . filter (not.null)
>             . map (words . trim)
>             . lines
```

Die Funktion, die jetzt noch die wesentliche Arbeit zu vollbringen hat, ist die Funktion `session`. Sie erzeugt für die engegebenen Sätze Antworten. Um uns das Programm estwas übersichtlicher zu machen, definieren wir uns ein paar Typsynonyme:

```
> type Word     = String
> type Words    = [Word]
```

Ein Eingabesatz ist demnach eine Liste von `Strings`. In der Funktion `eliza` haben wir die Eingabe schon auf diese Form gebrochen.

Das Prinzip unsres Programmes wird es sein, nach bestimmten Schlüsselwörtern in der Eingabe zu suchen und dann aus einer Tabelle zu diesen Schlüsselwörtern ein Antwortschema auszulesen. Eine Antwort soll dabei dann wieder ein `String` sein. Schlüsselwörter werden eine Liste von Wörtern darstellen. Wir definieren uns die Synonyme:

```
> type Replies  = [String]
> type KeyTable = [(Key, Replies)]
> type Key      = Words
```

Der globale Zustand unseres Programms wird ein Paar sein, das die Schlüsselworttabelle als erste Komponente hat und in der zweiten Komponente Antworten auf wiederholte Benutzereingaben hat.

```
> type State    = (KeyTable, Replies)
```

Einfache Hilfsfunktionen auf den globalen Zustand sind:

```
> newKeyTab                 :: KeyTable -> State -> State
> newKeyTab kt' (kt, rp)     = (kt', rp)

> keyTabOf                  :: State -> KeyTable
> keyTabOf (kt, rp)          = kt
```

Damit können wir angeben, wie unsere Funktion `session` typisiert sein soll. Wie in ihren Aufruf in `eliza` bereits zu sehen war, hat sie drei Argumente. Das erste ist der globale Status, der unsere Tabelle mit den Schlüsselwörtern enthält. Im zweiten Argument vermerken wir den zuletzt abgearbeitete Eingabesatz. Dieses werden wir benutzen, um wiederholte Benutzereingaben zu erkennen und darauf bei unserer Antwort einzugehen. Das dritte Argument sind dann alle noch folgenden Benutzereingaben in Sätze und Wörter bereits aufgesplittet. Ergebnis der Funktion `session` ist ein String, der alle von uns erzeugten Ausgaben auf den Bildschirm enthält. Wir haben insgesamt die folgende Typisierung:

```
> session               :: State -> Words -> [Words] -> String
```

Ein Fall dieser Funktion läßst sich sehr leicht abfangen. Wenn keine Benutzereingaben mehr kommen, dann erzeugen wir auch keine Ausgaben mehr:

```
> session _  _ [] = []
```

Betrachten wir also den Fall, daß noch weitere Benutzereingaben folgen:

```
> session rs prev (l:ls)
```

Wir werden in die Antworten, die wir erzeugen werden als ersten Buchstaben codieren, ob diese Antwort das Programm terminiert. Solche Antworten kann man sinnvoller Weise generieren, wenn der Benutzer *Good Bye* eingegeben hat oder unverschämt geworden ist. Wir werden programmterminierende Antworten daran erkennen, daß die mit dem Zeichen `$` beginnen. Dieses ist natürlich ein schrecklicher Hack. Hier hätte man besser mit unterschiedlichen Konstruktoren gearbeitet.

Wir werden also jetzt in der Funktion `session` unterscheiden, ob die Antwort, die wir erzeugen, ads Programm terminiert. Ist dieses der Fall, so wird der Ausgabestring beendet mit der letzten Antwort. Ansonsten rufen wir `session` rekursiv mit den weiteren Fragen auf.

```
>     |bye = tail response ++ "\n\n"
>     |otherwise = response ++ "\n\n" ++ session rs' l ls
>       where
>         bye = (head response)=='$'
```

Es bleibt uns nun noch die eigentliche Antwort `response` zu erzeugen. Hierbei werden wir unterscheiden, ob die Eingabe mit der vorherigen identisch ist oder nicht. Bei Wiederholungen erzeugen wir Antworten mit der Funktion `repeated` ansonsten mit der Funktion `answer`.

```
>         (response, rs')
>           | prev == l = repeated rs
>           | otherwise = answer rs l
```

Wie wir sehen werden wir beim Erzeugen einer Antwort auch den globalen Zustand, der unsere Tabelle enthält modifizieren.

Werfen wir einen kurzen Blick auf die Funktion, die Antworten bei wiederholten Fragen generiert:

```
> repeated (kt, (r:rp))      = (r, (kt, rp))
```

Wir bedienen uns dabei der nächsten Antwort im Antwortenpool für wiederholte Eingaben und löschen diese Antwort aus dem Pool.

Die eigentliche Spannung liegt jetzt in der Funktion `answer`, die echte Antworten auf eine Frage generiert und dabei globalen Status der Schlüsselworttabelle ändert. Wir erzeugen also eine Antwort und einen neuen Zustand:

```
> answer                :: State -> Words -> (String, State)
> answer st l            = (response, newKeyTab kt st)
```

Die Listen von Antworten auf wiederholte Eingaben brauchen wir hier nicht. Wir können die Antworten aus der Schlüsselworttabelle generieren:

```
>  where
>    (response, kt)         = ans (keyTabOf st)
```

Hier benutzen wir nun die lokale Funktion `ans` zur Antwortgenerierung. Achtung, die Funktion `ans` ist nicht λ-geliftet. Sie enthält als freie Variable: `l`. Das Argument für `ans` ist die Schlüsselworttabelle. `ans` versucht rekursiv alle Schlüsselworteinträge mit den Wörtern in der Benutzerfrage zu matchen. Dazu nimmt sie sich in einer lokalen Definition den ersten Eintrag der Schlüsselworttabelle und versucht in einer lokalen Definition für dieses Schlüsselwort eine Antwort zu generieren, die `rs` benannt wird. Paßt dieses Schlüsselwort nicht auf die Eingabe, so ist `rs` die leere Liste und wir müssen `ans` für die weiteren Schlüsselwörter aufrufen. Ansonsten muß die gefundene Antwort nur noch mit einer Funktion `makeResponse` für die Ausgabe aufgearbeitet werden.

```
>    e `cons` (r, es)       = (r, e:es)
>    ans (e:es) | null rs   = e `cons` ans es
>               | otherwise = (makeResponse a (head rs), (key,as):es)
>                 where rs           = replies key l
>                       (key,(a:as)) = e
```

Wir haben jetzt unsere Hauptaufgabe auf die Funktion `replies` geschoben. Sie enthält einen Schlüssel und eine Benutzereingabe und gibt eine leere Liste aus, wenn der Schlüssel nicht in der Eingabe vorkommt und eine Liste von Antwortteilen, wenn der Schlüssel in der Eingabe gefunden wurde.

```
> replies                  :: Words -> Words -> [String]
> replies key l             = ( map (conjugate l . drop (length key))
>                               . filter (prefix key . map ucase)
>                               . tails) l
```

Wie wir sehen, wird die Eingabe in alle Restlisten aufgefächert. Wir erhalten mit `tails` alle Ergebnisse durch iterativ angewendetes `tail`. Da wir bei Schlüsselwörtern nur mit Großbuchstaben rechnen, werden alle Buchstaben auf Großbuchstaben abgebildet. Nun filtern wir aus diesen Resteingabelisten die heraus, die mit der Schlüsselwortsequenz beginnen. Ist dieses geschehen, so entfernen wir die Schlüsselwörter. Übrig bleibt jeweils der Eingabetext, der dem Schlüsselwort folgte. Diesen wollen wir benutzen, um eine Antwort in `makeResponse` zu generieren. Dieses sind also die Eingabeteile, die wir in der Antwort wiederverwenden.

Wir haben drei triviale Hilfsfunktionen benutzt, die hier kurz aufgelistet seien:

```
> []     `prefix` xs        = True
> (x:xs) `prefix` []        = False
> (x:xs) `prefix` (y:ys)    = x==y && (xs `prefix` ys)

> tails []                  = []
> tails xs                  = xs : tails (tail xs)

> ucase                     = map toUpper
```

Wir wollen gerne Teile der Eingabe für die Ausgabe benutzen, um z.B. den folgenden Dialogteil generieren zu können:

```
I don't like my style of programming.
Don't you really like your style of programming?
```

Den Teil der Anfrage, den wir gerne zur Ausgabe benutzen wollen haben wir bereits isoliert. Es ist hingegen noch notwendig, grammatikalisch die zweite in die erste und die erste in die zweite Person umzuwandeln, den der Sprecher hat sich ja geändert. Hierfür diefiniern wir eine simple Funktion `conjugate`. Sie erhält zwei Argumente: die originale Benutzereingabe und das Fragment der Benutzereingabe, das wir eventuell für die Antwort benutzen wollen. Die gesammte Benutzereingabe brauchen wir eventuell, wenn wir keine Reste haben, die wir verwendsen können.

```
> conjugate  :: Words -> Words -> String
> conjugate d
>  = unwords . trailingI . map conj . maybe d  -- d is default input
>      where maybe d xs = if null xs then d else xs
>            conj  w    = head ([m | (w',m)<-conjugates, uw==w'] ++ [w])
>                  where uw = ucase w
```

```
>             trailingI  = foldr cons []
>                   where x `cons` xs | x=="I" && null xs = ["me"]
>                                     | otherwise         = x:xs
```

Wir haben hier eine weitere Tabelle benutzt, in der die Wörter die den Sprecher bzw. den Rezipienten bennenen entsprechend ihres Gegenübers aufgelistet sind:

```
> conjugates
>  = prepare (oneways ++ concat [[(x,y), (y,x)] | (x,y) <- bothways])
>               where oneways  = [ ("me",   "you") ]
>                     bothways = [ ("are",  "am"),     ("we're", "was"),
>  ("you",  "I"),      ("your",  "my"),
>  ("I've", "you've"), ("I'm",   "you're") ]
>                     prepare  = map (\(w,r) -> (ucase w, r))
```

Es bleibt uns nur noch die bisher gefundenen Antwortteile, als da wären die Antwort zu einem Schlüsselwort in der Schlüsselworttabelle sowie die abgetrennte und konjugierte Restanfrage nach dem Schlüsselwort, in eine Gesamtantwort zu vereinen. Wir unterscheiden hierbei drei Arten von Antworten:

- Antworten, deren Eintrag in der Schlüsselworttabelle mit einen Fragezeichen beginnen. Dieses sind Fragen, die wir an den Benutzer stellen, wobei wir Teile der Anfrage benutzen.
- Antworten, deren Eintrag in der Schlüsselworttabelle mit einen Punkt beginnen. Dieses sind Antworten, die Teile der Anfrage benutzen.
- sonstige Antworten. Diese benötigen keine Teile der Anfrage, um die Antwort zu generieren.

```
> makeResponse ('?':cs) us  = cs ++ " " ++ us ++ "?"
> makeResponse ('.':cs) us  = cs ++ " " ++ us ++ "."
> makeResponse cs       us  = cs
```

Damit sind wir im Prinzip fertig. Wir brauchen nur noch unsere Datenbank mit der Schlüsselworttabelle angeben. Wir machen dieses in zwei Schritten: im ersten werden wir alle Schlüsselwörter auslisten und auf eine Liste von Antwortsätzen verweisen. Die Antwortsätze zu Schlüsselwörtern liegen dann als eigener Superkombinator vor. Somit können wir einfach für verschiedene Schlüsselwörter dieselben Antworten benutzen.

Es ist dann nur noch zu beachten, daß wir die Antwortliste für ein Schlüsselwort abarbeiten. Damit ergibt sich, daß wir im Laufe der Sitzung keine Antworten mehr parat haben. Um dieses zu umgehen machen wir aus den Antwortlisten einfach unendlich lange Listen durch Zyklen:

```
> initial :: State
> initial  = ([(words k, cycle rs) | (k,rs) <-respMsgs], cycle repeatMsgs)
```

Es folgen unsere Daten:

```
> respMsgs    = [("SHIT", vulgar),
>                ("FUCK", vulgar),
>                ("FUCKING",           vulgar),
>                ("BYE", bye),
>                ("NO PROBLEM",        noproblem),
>                ("NO PROBLEMS",       noproblem),
```

```
>                 ("NOT HAVE ANY PROBLEM",    noproblem),
>                 ("NOT HAVE ANY PROBLEMS",   noproblem),
>                 ("CAN YOU", canYou),
>   ("CAN I", canI),
>  ("YOU ARE", youAre),
>  ("YOU'RE", youAre),
>  ("I DON'T", iDont),
>  ("I FEEL", iFeel),
>  ("WHY DON'T YOU", whyDont),
>  ("WHY CAN'T I", whyCant),
>  ("ARE YOU", areYou),
>  ("I CAN'T", iCant),
>  ("I AM", iAm),
>  ("I'M", iAm),
>  ("YOU",  you),
>  ("YES", yes),
>  ("NO", no),
>  ("COMPUTER", computer),
>  ("COMPUTERS", computer),
>  ("I WANT", iWant),
>  ("WHAT", question),
>  ("HOW", question),
>  ("WHO", question),
>  ("WHERE", question),
>  ("WHEN", question),
>  ("WHY", question),
>  ("NAME", name),
>  ("BECAUSE", because),
>  ("CAUSE", because),
>  ("SORRY", sorry),
>  ("DREAM", dream),
>  ("DREAMS", dream),
>  ("HI", hello),
>  ("HELLO", hello),
>  ("MAYBE", maybe),
>  ("YOUR", your),
>  ("ALWAYS", always),
>  ("THINK", think),
>  ("ALIKE", alike),
>  ("FRIEND", friend),
>  ("FRIENDS", friend),
>  ("", nokeyMsgs) ]

> canYou     = [ "?Don't you believe that I can",
>  "?Perhaps you would like to be able to",
>  "?You want me to be able to" ]
> canI      = [ "?Perhaps you don't want to",
>  "?Do you want to be able to" ]
> youAre     = [ "?What makes you think I am",
>  "?Does it please you to believe I am",
>  "?Perhaps you would like to be",
>  "?Do you sometimes wish you were" ]
> iDont      = [ "?Don't you really",
>  "?Why don't you",
>  "?Do you wish to be able to",
>  "Does that trouble you?" ]
> iFeel      = [ "Tell me more about such feelings.",
>  "?Do you often feel",
>  "?Do you enjoy feeling" ]
```

```
> whyDont    = [ "?Do you really believe I don't",
>  ".Perhaps in good time I will",
>  "?Do you want me to" ]
> whyCant    = [ "?Do you think you should be able to",
>  "?Why can't you" ]
> areYou     = [ "?Why are you interested in whether or not I am",
>  "?Would you prefer if I were not",
>  "?Perhaps in your fantasies I am" ]
> iCant      = [ "?How do you know you can't",
>  "Have you tried?",
>  "?Perhaps you can now" ]
> iAm      = [ "?Did you come to me because you are",
>  "?How long have you been",
>  "?Do you believe it is normal to be",
>  "?Do you enjoy being" ]
> you      = [ "We were discussing you --not me.",
>  "?Oh,",
>  "You're not really talking about me, are you?" ]
> yes      = [ "You seem quite positive.",
>  "Are you Sure?",
>  "I see.",
>  "I understand." ]
> no      = [ "Are you saying no just to be negative?",
>  "You are being a bit negative.",
>  "Why not?",
>  "Are you sure?",
>  "Why no?",
>                 "Is this definetely your opinion?" ]
> computer     = [ "Do computers worry you?",
>  "Are you talking about me in particular?",
>  "Are you frightened by machines?",
>  "Why do you mention computers?",
>  "What do you think machines have to do with your problems?",
>  "Don't you think computers can help people?",
>  "What is it about machines that worries you?" ]
> iWant      = [ "?Why do you want",
>  "?What would it mean to you if you got",
>  "?Suppose you got",
>  "?What if you never got",
>  ".I sometimes also want" ]
> question     = [ "Why do you ask?",
>  "Does that question interest you?",
>  "What answer would please you the most?",
>  "What do you think?",
>  "Are such questions on your mind often?",
>  "What is it that you really want to know?",
>  "Have you asked anyone else?",
>  "Have you asked such questions before?",
>  "What else comes to mind when you ask that?" ]
> name      = [ "Names don't interest me.",
>  "I don't care about names --please go on." ]
> because     = [ "Is that the real reason?",
>  "Don't any other reasons come to mind?",
>  "Does that reason explain anything else?",
>  "What other reasons might there be?" ]
> sorry      = [ "Please don't apologise!",
>  "Apologies are not necessary.",
>  "What feelings do you have when you apologise?",
>  "Don't be so defensive!" ]
```

```
> dream      = [ "What does that dream suggest to you?",
>  "Do you dream often?",
>  "What persons appear in your dreams?",
>  "Are you disturbed by your dreams?" ]
> hello      = [ "How do you...please state your problem." ]
> maybe      = [ "You don't seem quite certain.",
>  "Why the uncertain tone?",
>  "Can't you be more positive?",
>  "You aren't sure?",
>  "Don't you know?" ]
> your      = [ "?Why are you concerned about my",
>  "?What about your own" ]
> always     = [ "Can you think of a specific example?",
>  "When?",
>  "What are you thinking of?",
>  "Really, always?" ]
> think      = [ "Do you really think so?",
>  "?But you are not sure you",
>  "?Do you doubt you" ]
> alike      = [ "In what way?",
>  "What resemblence do you see?",
>  "What does the similarity suggest to you?",
>  "What other connections do you see?",
>  "Cound there really be some connection?",
>  "How?" ]
> friend     = [ "Why do you bring up the topic of friends?",
>  "Do your friends worry you?",
>  "Do your friends pick on you?",
>  "Are you sure you have any friends?",
>  "Do you impose on your friends?",
>  "Perhaps your love for friends worries you." ]
>
> repeatMsgs = [ "Why did you repeat yourself?",
>  "Do you expect a different answer by repeating yourself?",
>  "Come, come, elucidate your thoughts.",
>  "Please don't repeat yourself!",
>                 "This does't help. Stop repeating yourself!",
>                 "$I have enough of you. Good Bye!" ]
>
> nokeyMsgs    = [ "I'm not sure I understand you fully.",
>  "What does that suggest to you?",
>  "I see.",
>  "Can you elaborate on that?",
>  "Say, do you have any psychological problems?"]
>
> bye      = [ "$Good Bye! Think about what we were saying."]
>
> vulgar = [ "Oh you are a vulgarian.\nWhy don't you relax and stop using "++
>                 "such words.",
>                 "You are again using a very harsh language.",
>                 "What do you want to proof by this sort of language?",
>                 "Do you think being so destructive helps you?",
>                 "$I think we better stop now.\n"++
>                 "Come back, when you have relaxed a bit."]
>
> noproblem    = ["Everybody has some Problem, don't you think so.",
>                 "To have no problems would be very boring\n"++
>                 "and that would be a very big problem, wouldn't it?",
>                 "Think about the problems you had as a child!"]
```