Universität des Saarlandes
Fachrichtung 6.2- Informatik

Ausarbeitung zum Fortgeschrittenen-Praktikum

Alice Server Pages

Simon Georg Pinkel

Betreuer: **Guido Tack**

Lehrstuhl für Programmiersysteme, Prof. Dr. Gert Smolka

1 Einleitung

Dynamische Webseiten sind in den letzten Jahren fester Bestandteil des Webs geworden. Seiten mit der Funktionalität von Amazon, Ebay oder Google könnten ohne sie nicht existieren. Serverseitige Übersetzung/Interpretation der Seite hat dabei nicht nur Sicherheitsvorteile, sondern ist bei den riesigen Datenbanken, die hinter Sites wie Google stehen, eine Notwendigkeit geworden.
In dieser Arbeit sollen die Alice Server Pages vorgestellt werden. Es handelt sich dabei um Server Page Technologie, die es erlaubt, dynamische Seiten in Alice[1] zu erstellen. Alice ist eine am Lehrstuhl für Programmiersysteme entwickelte Programmiersprache basierend auf Standard ML, ausgestattet mit einer großen Anzahl an Erweiterungen wie Nebenläufigkeit, Laziness, Constraint Programming oder verteilter Programmierung. Im Rahmen dieser Arbeit soll untersucht werden, in wieweit sich Alice als Skriptsprache für Server Pages eignet. Darauf aufbauend werden mehrere Fragestellungen untersucht:

- Wie lässt sich das Typsystem von Alice im Kontext von dynamischen HTML-Seiten nutzen?
- Wie fügen sich funktionale Programmieridiome in diesen Kontext ein?
- Welche Möglichkeiten ergeben sich durch Alice-spezifische Features wie Nebenläufigkeit, Laziness oder den Component Manager?
- Wie lassen sich die Server Pages am besten in einen Webserver(z.B. Apache [3]) integrieren?

Die folgende Ausarbeitung beschäftigt sich mit dem Aufbau, der Syntax, der Architektur und der Implementierung der Alice Server Pages. Die Arbeit ist in sieben Kapitel unterteilt. Kapitel 2 formuliert die Anforderungen, die die Implementierung erfüllen muss. Der Entwurf, das heißt den Aufbau und die Syntax einer Alice Server Page, wird in Kapitel 3 behandelt und diskutiert. Kapitel 4 gibt Einblick in die Architektur der Implementierung, und Kapitel 5 geht genauer auf die in 4 vorgestellten Module ein. Ähnliche Arbeiten und werden in Kapitel 6 mit den Alice Server Pages verglichen. Kapitel 7 schließlich fasst Kapitel 2 bis 6 zusammen und gibt Antworten auf die Fragen aus Kapitel 1.

2 Anforderungen

2.1 Übersetzung/Interpretation

Um aus Serverseiten, also Webseiten mit statischem HTML und dynamischem Inhalt, pures HTML zu generieren, wird eine Funktionalität benötigt, die die dynamischen Teile der Seite interpretiert, oder in eine ausführbare Form übersetzt. Im Rahmen dieser Arbeit wurde Übersetzung gewählt. Dies bringt Performanzvorteile, da eine häufig angefragte Seite nicht immer wieder übersetzt werden muss.

2.2 Schnittstelle für dynamische Daten

Mit dynamischen Daten seien im Folgenden Formularvariablen und Cookies gemeint. Beide werden bei einer HTTP-anfrage an den Webserver geschickt, und sind

relevanter Bestandteil der Dynamik einer Server Page. Oft werden diese Daten über das Common Gateway Interface (CGI) zur Verfügung gestellt. Dieses ist jedoch sehr unkomfortabel, da zum Beispiel einzelne Formularvariablen erst aus einer einzigen Zeichenkette herausgeparst werden müssen. Es sollte dem Programmierer der Serverseite jedoch möglich sein, auf eine komfortablere Art auf diese Daten zuzugreifen. Des Weiteren ist eine Schnittstelle erstrebenswert, die Gebrauch von der statischen Typprüfung von Alice macht.

2.3 Effizienz

Schließlich sollte die Implementierung schnell genug sein, auch Webseiten mit komplexerem Berechnungsaufwand, wie einer Suche auf einer Datenbank, in möglichst geringer Zeit zu generieren. Es ist nicht direkt Ziel dieser Arbeit, die Effizienz von beispielsweise PHP[4] zu schlagen, sondern vielmehr eine Reaktionszeit zu erreichen, die die Alice Server Pages verwendbar macht.

3 Entwurf

3.1 Übersetzung der Serverseiten

Es gibt in der Welt von Standard ML bereits mehrere Ansätze, ML Dialekte als Websprache nutzbar zu machen. Einer davon sind die ML Server Pages [1], welche als Vorlage für die Alice Server Pages verwendet wurden.

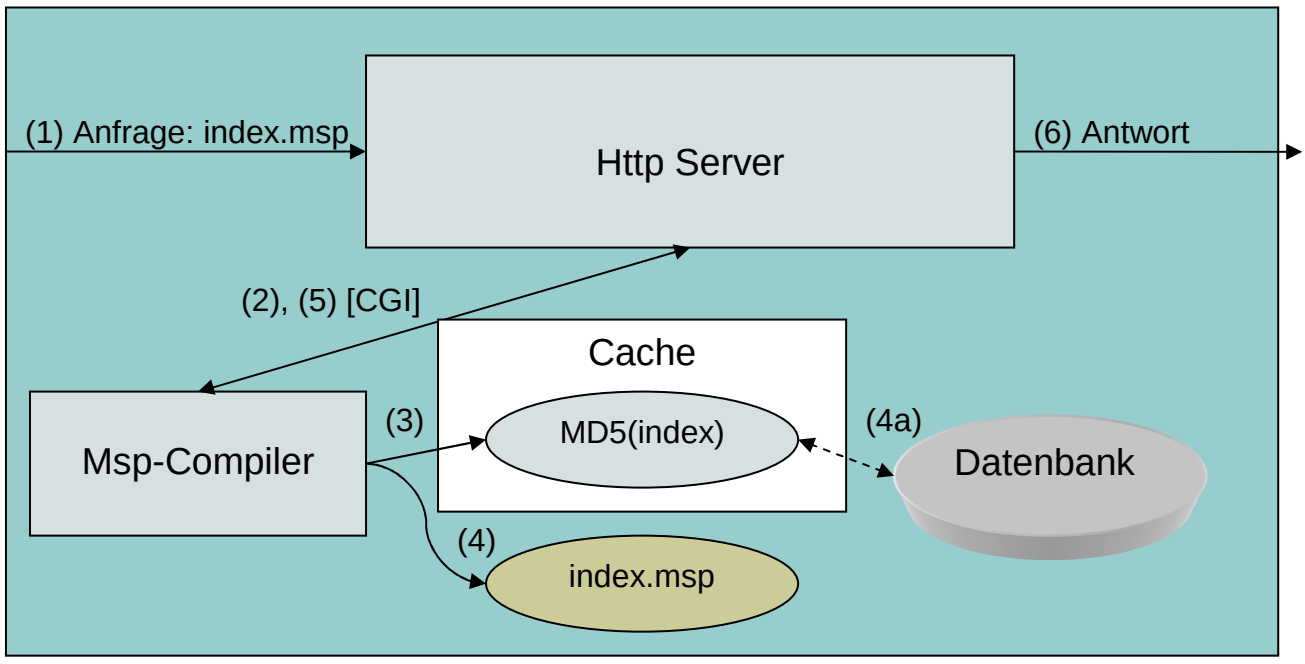

Abbildung 1 Die ML Server Pages

Abbildung 1 zeigt die Vorgehensweise der ML Server Pages bei einer Anfrage einer Seite. Dabei ist insbesondere die Verwaltung eines Caches für bereits kompilierte Server Pages zu beachten.
Die Anfrage (1) an index.msp sorgt dafür, dass der mit „.msp“ verknüpfte Handler, in dem Fall der Msp-Compiler, aufgerufen wird. Dieser erhält die Daten über die

aufzurufende Seite, sowie die dynamischen Daten, über das Common Gateway Interface(2). Der Msp-Compiler überprüft zunächst(3), ob eine *aktuelle* Version von index.msp im Cache vorliegt. Die Kompilate im Cache haben dabei die MD5-Summe des *absoluten* Pfades der Quelldatei als Namen. In diesem Fall wäre der Name zum Beispiel MD5(„/services/http/msp-test/index.msp“).
Falls kein oder nur ein veraltetes Kompilat im Cache ist, wird ein neues generiert(4): „index.msp“ wird ausgelesen und in eine reine ML-quelldatei übersetzt. Das bedeutet konkret, dass alle HTML-fragmente in „`val _ = print ...`“-Anweisungen umgewandelt werden.
Das Kompilat wird anschließend in einer Umgebung aufgerufen, die es erlaubt, auf die gerade gesendeten dynamischen Daten zuzugreifen. Im Fall der ML Server Pages wird dies über eine spezielle Struktur „Msp“ erreicht. Gegebenenfalls kann das Kompilat dann auf eine Datenbank zugreifen(4a).
Der von dem Kompilat generierte HTML-code wird anschließend(5) wieder an den HTTP Server gesendet, der diesen zurück an den anfragenden Browser schickt(6).

CGI impliziert, dass bei jeder Anfrage eines Browsers an eine Seite ein *neuer* Prozess gestartet wird. Dies bringt im Falle von Alice ein Problem mit sich: Alice hat relativ hohe Startupzeiten. Dies würde bedeuten, dass ein Websurfer bei jeder Anfrage an eine Alice Server Page mindestens diese Startupzeit lang warten müsste. Dies ist nicht akzeptabel.
Um dieses Problem zu lösen, wurde das Modell der ML Server Pages ein wenig erweitert (siehe Abbildung 2).

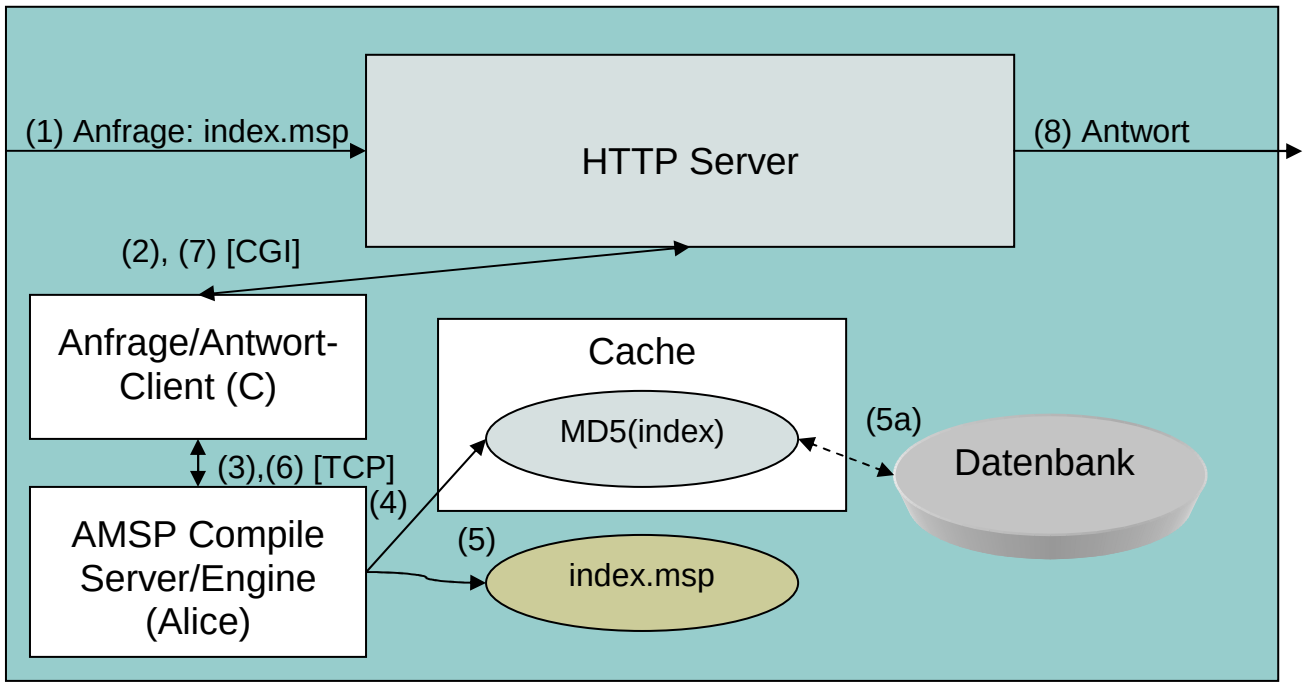

Abbildung 2 Die Alice Server Pages

Der MSP Compiler wurde hierbei ersetzt durch zwei neue Komponenten: Den Anfrage/Antwort-Client und den AMSP Compile Server. Ersterer ist der Prozess, der vom HTTP Server bei einer Anfrage über CGI gestartet wird. Er ist in C implementiert, wodurch niedrige Startupzeiten gewährleistet werden. Die Aufgabe des Anfrage/Antwort-Clients besteht lediglich darin, die Anfrage an die zweite neue Komponente weiterzuleiten, den in Alice implementierten AMSP Compile Server. Bei

diesem handelt es sich um einen persistenten Prozess, ähnlich dem HTTP Server, der sich um die Kompilierung und Cacheverwaltung der Alice Server Pages kümmert, so wie der MSP Compiler bei den ML Server Pages. Sobald die Antwort vom Compile Server berechnet wurde, wird sie an den Anfrage/Antwort Client weitergeleitet, der sie seinerseits wieder an den HTTP Server sendet.
Der AMSP Compile Server muss dabei nicht zwangsläufig auf derselben Maschine laufen wie der HTTP Server, da die Kommunikation zwischen Client und Compile Server über eine einfache TCP-Socket-verbindung erfolgt.
Der Compile Server muss nur einmal, idealerweise zusammen mit dem HTTP Server, gestartet werden.

3.2 Syntax der Serverseiten

3.2.1 Definition & Erläuterung

Eine Alice Server Page ist eine HTML-Webseite, in die Alice Code Fragmente eingebettet sind. Diese Codefragmente sind durch spezielle XML Processing Instructions vom Rest des Codes getrennt und in folgende Typen unterteilt:

`<?amsp`	Alice-programm	beginnt einen Ausschnitt aus einem Alice-programm[1]
`<?amsp=`	Alice-ausdruck	beginnt einen Alice-ausdruck, dessen Wert in einen Wert vom Typ `string` umgewandelt und ausgegeben
`<?amsp$`	word sequence	beginnt einen Alice-ausdruck, bei dessen Wert es sich um eine *word sequence* handeln muss
`<?amsp:fcall`	Ereignisgesteuerter Prozeduraufruf	beginnt einen ereignisgesteuerten Prozeduraufruf

Ein Fragment, unabhängig vom Typ, wird immer durch „?>“ beendet.
Innerhalb eines Code-fragments kann die Prozedur

```
print : string -> unit
```

verwendet werden, um Daten an die Webseite zu senden.
Ein Alice-ausdruck-fragment wandelt den dargestellten Wert in einen String um. Diese Umwandlung findet unter Verwendung des Alice-prettyprinters statt, der auch für die Wertdarstellung im Alice Interpreter verantwortlich ist.
Die bei dem dritten Fragmenttyp erwähnten „word sequences“ werden im nächsten Abschnitt erläutert. Die ereignisgesteuerten Prozeduraufrufe werden in Abschnitt 3.4 behandelt.

3.2.2 Word sequences

Eine Word sequence ist eine Darstellung einer Zeichenkette, die effiziente Konkatenierungen ermöglicht, sofern die Zeichenkette später ausgegeben werden soll.

[1] Definiert wie in Standard ML: Eine Folge von Deklarationen

Normale String-konkatenationen sind teuer, ihre Laufzeit ist linear zu der Länge der übergebenen Strings. Wenn allerdings die Weiterverwendung der Zeichenkette bereits bekannt ist und es sich um eine Ausgabe an einen Datenstrom handelt (ML-Typ `outstream`), kann die Komplexität von Konkatenationen relativ leicht auf konstante Laufzeit reduziert werden: Konkatenationen werden ersetzt durch Baumkonstruktion!
Anstatt Zeichenketten direkt zu konkatenieren, wird stattdessen ein Baum konstruiert. Jede Konkatenation fügt zwei Zeichenketten *t1* und *t2*, als Bäume interpretiert, zu einem neuen Baum *t* zusammen. Diese Operation hat konstante Laufzeit. Sobald die Zeichenkette ausgegeben wird, wird der konstruierte Baum traversiert und die Blätter sukzessive ausgegeben.
Entsprechend ist der Typ der Word sequences aufgebaut:

```
datatype wseq =
    Empty                       (* The empty sequence         *)
  | Nl                          (* Newline                    *)
  | $ of string                 (* A string                   *)
  | $$ of string list           (* A sequence of strings      *)
  | && of wseq * wseq           (* Concatenation of sequences *)
```

Im Quellcode sollte && aus Gründen der Lesbarkeit mit Hilfe von `infix` als Infix-Operator deklariert werden. `Empty`, `Nl` und `$$` dienen nur dazu, die Benutzung der word sequences zu vereinfachen, sind aber nicht notwendig.
Eine Prozedur zur Umwandlung einer word sequence in eine Zeichenkette fällt somit relativ leicht (benutzt aber wieder ineffiziente Konkatenation und sollte nur zu Debug-zwecken eingesetzt werden):

```
fun flatten' Empty acc      = acc
  | flatten' Nl    acc      = "\n" :: acc
  | flatten' ($ s) acc      = s :: acc
  | flatten' ($$ ss) acc    = ss @ acc
  | flatten' (s1 && s2) acc = flatten' s1 (flatten' s2 acc)
fun flatten seq = String.concat(flatten' seq [])

flatten : wseq ->string
```

Ebenso durch einfache strukturelle Rekursion lässt sich die Ausgabeprozedur formulieren (für einen beliebigen `outstream`):

```
fun outputseq _   Empty       = ()
  | outputseq out Nl          = TextIO.output(out,"\n")
  | outputseq out ($ s)       = TextIO.output(out,s)
  | outputseq out ($$ ss)     =
      List.app (fn s=>TextIO.output(out,s)) ss
  | outputseq out (&&(s1,s2)) = (outputseq out s1; outputseq out s2)

outputseq : outstream -> wseq -> unit
```

Eine `print`-Prozedur lässt sich somit einfach mit Hilfe von `outputseq` implementieren:

```
val printseq = outputseq TextIO.stdOut
```

Word sequences wurden den ML Server Pages übernommen. Sie dienen einer effizienteren Verarbeitung der Server Page. Der MSP Compiler, und somit auch die

Alice Server Page Engine, verwenden sie, um statischen mit dynamischem HTML-code zu verknüpfen. Falls eine Alice Server Page viele String-konkatenationen aufweist, sollte der Programmierer word sequences einsetzen, die dafür benötigte Struktur `Amsp` ist bei jeder Alice Server Page importiert.

3.2.3 Beispiel

Zur Erläuterung betrachten wir folgende Alice Server Page:

```
<html>
  <head>
    <title>
      Alice Server Pages
        – Funktionale Programmierung f&uuml;r das Web
    </title>
  </head>
  <body>
    <h1>Alice Server Pages
        – Funktionale Programmierung f&uuml;r das Web</h1>
    <p>Heute ist:
      <?amsp= Date.toString(Date.fromTimeLocal(Time.now())) ?>
    </p>
  </body>
</html>
```

Im Browser wird die Seite wie folgt angezeigt:

Abbildung 3 Eine einfache Server Page

Der Alice-ausdruck wertet zu einer String-repräsentation des aktuellen Datums aus. Den Zusatz `Date.toString` kann man an dieser Stelle nicht weglassen: `Date.fromTimeLocal(Time.now())` wertet zu einem Datum vom Typ `date` aus, bei welchem es sich um einen *abstrakten Datentyp* handelt. Der Wert kann daher vom Alice Pretty Printer nicht ausgegeben werden.
Im Folgenden haben wir die Seite um ein einfaches Programm erweitert:

```
...
  <body>
    <h1>Alice Server Pages
        – Funktionale Programmierung f&uuml;r das Web</h1>
    <p>Heute ist:
      <?amsp= Date.toString(Date.fromTimeLocal(Time.now())) ?>
    </p>
    <ul>
    <?amsp
      (* good ol' fak *)
      fun fak n = if n<2 then 1 else n*fak(n-1)
      (* print the faculty of the first 10 natural numbers *)
      val _ = List.app (fn s => print("<li>" ^ Int.toString s))
              (List.tabulate(10,fak))
    ?>
```

```
    </ul>
  </body>
...
```

Das Programm gibt die Fakultäten der Zahlen `0` bis `9` in einer ungeordneten HTML-Liste(`<ul>`) aus.
Der Programmierer einer Alice Server Page ist bei solchen Programmfragmenten nicht auf die Verwendung lokaler Prozeduren oder den Standardstrukturen beschränkt, der kann auch, wie in Alice üblich mit Hilfe des Schlüsselworts `import` beliebige Alice Programme einbinden. Solche imports müssen allerdings noch vor dem ersten HTML-code in einem Alice-programm-fragment aufgeführt werden. Diese Beschränkung wurde zum einen aus Gründen der Übersichtlichkeit eingeführt. Außerdem dient sie der Effizienz, da sich imports so leichter lokalisieren lassen.
Im obigen Beispiel ist die Prozedur `fak` auf der Webseite lokal definiert. Für eine einfache Prozedur wie `fak` kann dies sinnvoll sein, jedoch ist es grundsätzlich besserer Stil, Funktionalität so wie `fak` hier auszulagern. Wir passen das Beispiel entsprechend an:

```
<?amsp
  import structure Fak from "Fak"
?>
<html>
  <head>
   ...
    <ul>
    <?amsp
      open Fak
      (* print the faculty of the first 10 natural numbers *)
      val _ = List.app (fn s => print("<li>" ^ Int.toString s))
              (List.tabulate(10,fak))
    ?>
    </ul>
  ...
```

Die Struktur `Fak` aus dem Alice-sourcefile „`Fak.aml`“ dessen Kompilat sich in dem Verzeichnis der Server Page befinden muss, sieht folgendermaßen aus:

```
structure Fak =
    struct
        fun fak n = if n<2 then 1 else n*fak(n-1)
    end
```

Eine HTML-seite nicht in purem HTML schreiben zu müssen, sondern sie generieren lassen zu können, ist bereits eine praktische Eigenschaft von dynamischen Webseiten. Wirklich interessant wird es allerdings erst, wenn der Inhalt der Webseite auch von externen Daten, also beispielsweise Variablen aus HTML-Formularen, beeinflusst werden kann. Der folgende Abschnitt erläutert, wie der Zugriff auf diese Daten in einer Alice Server Page funktioniert.

3.3 Schnittstelle für dynamische Daten

3.3.1 Definition & Erläuterung

In jeder Active Server Page-Implementierung gibt es eine Schnittstelle, die Zugriff auf Formularvariablen und Cookies ermöglicht. Dabei gibt es in beiden Fällen lesenden Zugriff, im Fall von Cookies muss der Programmierer auch in der Lage sein, diese beim anfragenden Browser zu setzen.

In einer Alice Server Page laufen diese Zugriffe folgendermaßen ab: Der Programmierer gibt *zwischen* eventuellen imports und dem Rumpf der Seite (HTML/Alice-code) einen *record-Typ* `formvar` beziehungsweise `cookies` an. Dieser muss folgender Grammatik entsprechen:

start ::= *recordTy*
ty ::= *primTy* `option` | *primTy* `eoption`
| *recordTy* | *ty* * … * *ty*
| *ty* `list` | *ty* `vector` (*)
recordTy ::= { *tyrow* }
tyrow ::= *label* `:` *ty* [`,` *tyrow*]
primTy ::= `int` | `real` | `char` | `string` | `bool` | `file`
(*) nicht erlaubt bei cookies

Abbildung 4 Grammatik der Typen `formvar`/`cookies`

Sobald wenigstens einer der Typen `formvar`/`cookies` spezifiziert wurde, steht dem Programmierer auf Toplevel ein *Wert* (um genau zu sein ein record) `formvar` beziehungsweise `cookies` zur Verfügung, der genau dem vorgegebenen Typ entspricht. Der Typ `file` wird in Abschnitt 3.3.3 erläutert, er wird für Dateiuploads benötigt.
Wie man an der Grammatik erkennen kann, ist eine beliebige Schachtelung durch die Baumkonstruktoren Records, Tupel sowie Vektoren und Listen möglich. Die Blätter des dadurch entstehenden Baums müssen jedoch immer vom Typ `option` oder `eoption` sein. Der Sinn dieser Restriktion wird im nächsten Abschnitt genauer erläutert.

3.3.2 Beispiel

Wir erweitern unser Beispiel aus 3.2.3 um folgendes HTML-formular:

```
...
  </p>
  <form action="index.amsp">
    Ein Zeichen:              <input type="text" name="charvar"><br>
    Eine Zeichenkette:        <input type="text" name="stringvar"><br>
    Eine ganze Zahl:          <input type="text" name="intvar"><br>
    Eine Fliesskommazahl:     <input type="text" name="realvar"><br>
    Ein Wahrheitswert:        <input type="text" name="boolvar"><br>
    <input type="submit" name="button" value="go!">
  </form>
...
```

Damit ein Zugriff auf die Werte in den Eingabefeldern (<input>) möglich ist, muss zwischen imports und dem Rest des Codes ein Typ `formvar` deklariert worden sein. Ein solcher Typ könnte so aussehen:

```
<?amsp
  import structure Fak from "Fak"
  type formvar = {
      charvar   : char eoption,
      stringvar : string option,
      intvar    : int eoption,
      realvar   : real eoption,
      boolvar   : bool eoption
  }
?>
...
```

Ein Alice-ausdruck-fragment bietet sich an, um sich den aktuellen Inhalt des *Werts* `formvar` anschauen zu können:

```
...
  <body>
    <h1>Alice Server Pages
        – Funktionale Programmierung f&uuml;r das Web</h1>
    <p>Heute ist:
      <?amsp= Date.toString(Date.fromTimeLocal(Time.now())) ?>
    </p>
    <p>Inhalt von formvar: <?amsp= formvar ?></p>
...
```

Nun gibt es für jede im Typ `formvar` angegebene Variable `x` vom Typ `t` drei Fälle:

1. In der HTTP-anfrage ist ein Wert für `x` enthalten, und er entspricht eventuellen Formvorgaben[1] von `t`
2. In der HTTP-anfrage ist ein Wert für `x` enthalten, und er entspricht nicht den Formvorgaben von `t`
3. In der HTTP-anfrage ist kein Wert für `x` enthalten

Damit diese Unterscheidung klar im Code abgefangen wird, wird für jeden Bezeichner mit einem primitiven Typ (Nonterminal „`primTy`“ in Grammatik aus Abbildung 4) einer der Typen `option` und `eoption` erzwungen: `option` ist der aus Alice/SML bekannte Optionstyp:

```
datatype 'a option =
      SOME of 'a
    | NONE
```

Problem bei der Verwendung von `option` ist, dass die oben aufgezählten Fälle 2 und 3 zu NONE zusammengefasst würden. Dadurch kann nicht unterschieden werden, ob `x` entweder mit der Anfrage mitgesendet wurde und nur nicht der Formvorgabe entspricht, oder gar nicht erst angegeben wurde. Um diese

[1] Zum Beispiel der Typ `int` spezifiziert eine ganze Zahl; Wenn der Websurfer jedoch gar nichts oder eine Zeichenkette angibt, die nicht einer ganzen Zahl entspricht, entspricht der Wert nicht der Formvorgabe von `int`. Weitere Typen mit Formvorgabe sind `real`, `char` und `bool`.

Unterscheidung möglich zu machen, wurde der Typ `eoption` (extended option) eingeführt:

```
datatype 'a eoption =
      ESOME of 'a
   | EWRONGFORMAT
   | ENONE
```

Die drei Varianten von `eoption` entsprechen genau den drei oben angegebenen Fällen. Für `bool` wurde eine Ausnahmeregelung getroffen: Fälle 2 und 3 führen immer zum Wert `false`. Diese starke Vereinfachung wurde eingeführt, um `bool` für Checkboxen in HTML-formularen verwendbar zu machen (wofür `bool` in der Regel verwendet werden sollte): Diese werden nur, wenn sie „gecheckt“, also angewählt wurden, in der Anfrage mitgesendet.
Es steht dem Programmierer natürlich frei, nur `option` statt `eoption` zu verwenden. Dann werden die beiden Fälle `EWRONGFORMAT` und `ENONE` zu `NONE` zusammengefasst. Umgekehrt kann er natürlich auch `eoption` für einen Typ ohne Formatvorgabe verwenden (der einzige Typ ohne Formatvorgabe ist `string`). Dann tritt der Fall `EWRONGFORMAT` erwartungsgemäß nie auf. Zur Formatüberprüfung beziehungsweise Konvertierung werden die Funktionen `Char.fromString`, `Int.fromString` und `Real.fromString` verwendet (Bei `bool` wurde eine eigene, relaxierte Version von `Bool.fromstring` benutzt).
Abbildung 5 enthält einige Beispiele zur Erläuterung.

Übergebene Werte	Inhalt von `formvar`

`charvar = "c"`
`stringvar = "test"`
`intvar = "234"`
`realvar = "3.14"`
`boolvar = "true"`
	`{ charvar = ESOME #"c",`
	`  stringvar = SOME "test",`
	`  intvar = ESOME 234,`
	`  realvar = ESOME 3.14,`
	`  boolvar = ESOME true }`

`charvar = ""`
`stringvar = ""`
`intvar = "foo"`
`realvar = "3"`
`boolvar = "bar"`
	`{ charvar = EWRONGFORMAT,`
	`  stringvar = SOME "",`
	`  intvar = EWRONGFORMAT,`
	`  realvar = SOME "3.0",`
	`  boolvar = ESOME false }`

`realvar = "foo"`
`boolvar = "1"`
	`{ charvar = ENONE,`
	`  stringvar = NONE,`
	`  intvar = ENONE,`
	`  realvar = EWRONGFORMAT,`
	`  boolvar = ESOME true }`

`boolvar = "on"`	`{ charvar = ENONE,`
	`  stringvar = NONE,`
	`  intvar = ENONE,`
	`  realvar = ENONE,`
	`  boolvar = ESOME true }`

Abbildung 5 Verhaltensbeispiele der Schnittstelle für einfache Typen

Wir könnten zu unserem Typ `formvar` von oben auch ein weiteres Feld `button` hinzufügen. Dieses Feld sollte mit `string option` getypt sein und wäre immer genau dann auf `SOME "go!"` gesetzt, wenn der Button betätigt wurde.
Hier ist Vorsicht geboten:

Die meisten Webbrowser erlauben auch die Betätigung der Entertaste zur Bestätigung von Eingaben in ein Formular. In diesem Fall würde jedoch keine Formularvariable `button` gesendet, da dieser ja nicht betätigt wurde.

Wenn immer nur das am Anfang dieses Abschnitts angegebene Formular verwendet wird, können die Werte in formvar nie ENONE beziehungsweise NONE annehmen. Von dieser Situation kann man im Allgemeinen jedoch nicht immer ausgehen.

Wir können unser Beispiel von oben noch ein wenig erweitern und in die Darstellung der ersten zehn Fakultäten etwas dynamischer gestalten:

```
...
  </p>
  <form action="index.amsp">
    Ein Zeichen:              <input type="text" name="charvar"><br>
    Eine Zeichenkette:        <input type="text" name="stringvar"><br>
    Eine ganze Zahl(n):       <input type="text" name="intvar"><br>
    Eine Fliesskommazahl:     <input type="text" name="realvar"><br>
    Ein Wahrheitswert:        <input type="text" name="boolvar"><br>
    <input type="submit" name="button" value="go!">
  </form>
  <p>Fakult&auml;ten von 0 bis n (sofern 0<=n<=12):</p>
  <ul>
  <?amsp
    open Fak
    (* Try to print the faculty numbers between 0 and n *)
    val _ =
        case #intvar formvar of
            (* n was given; check the domain *)
            ESOME n =>
                if n<0 orelse n>12
                    then print "<li>0<=n<=12 ist nicht erf&uuml;llt!"
                else List.app
                     (fn s => print("<li>" ^ Int.toString s))
                     (List.tabulate(n+1,fak))
          (* the n was not formatted as an integer *)
          | EWRONGFORMAT => print "<li>n war keine ganze Zahl!"
          (* no n was given; gimme! *)
          | ENONE => print "<li>Geben sie eine Zahl 0<=n<=12 an."
  ?>
  </ul>
...
```

Die drei Fälle für eine Formularvariable werden hier im Quellcode abgefangen, und anstatt eine Liste auszugeben, wird gegebenenfalls eine Fehlermeldung angezeigt. Solche Fallunterscheidungen sind häufig Bestandteil von dynamischen Webseiten: Sie untersuchen, in welchem Zustand sich die Seite befindet und passen die Ausgabe entsprechend an. Die Ausgabe erfolgt dabei üblicherweise durch Verwendung von eingebundenen Komponenten, wie der Struktur `Fak` hier. Die eigentliche Funktionalität der Webseite ist somit ausgelagert, nur die Dynamik ist direkt im Quellcode der Server Page enthalten.

Wie bereits in 3.2 erwähnt, erlaubt die Grammatik aus Abbildung 4 auch Schachtelungen von Records und Tupeln, und die Konstruktion von Listen und Vektoren. Im Folgenden wird erläutert, wie diese von Formularvariablen gefüllt werden können. Dazu betrachten wir folgenden `formvar`-Typ:

```
type formvar = {
      list1  : int option list,
      list2 : (int option * real option list) list,
      rec   : {
                  foo : {bar: int option, baz: real option},
                  moo : (int option * bool option)
     }
}
```

Eine Baumdarstellung hilft, einen geschachtelten Typen besser zu verstehen. Abbildung 6 ist eine Baumdarstellung von `formvar`. Ein Formular muss Werte für die Blätter dieses Baums liefern, um `formvar` zu füllen.

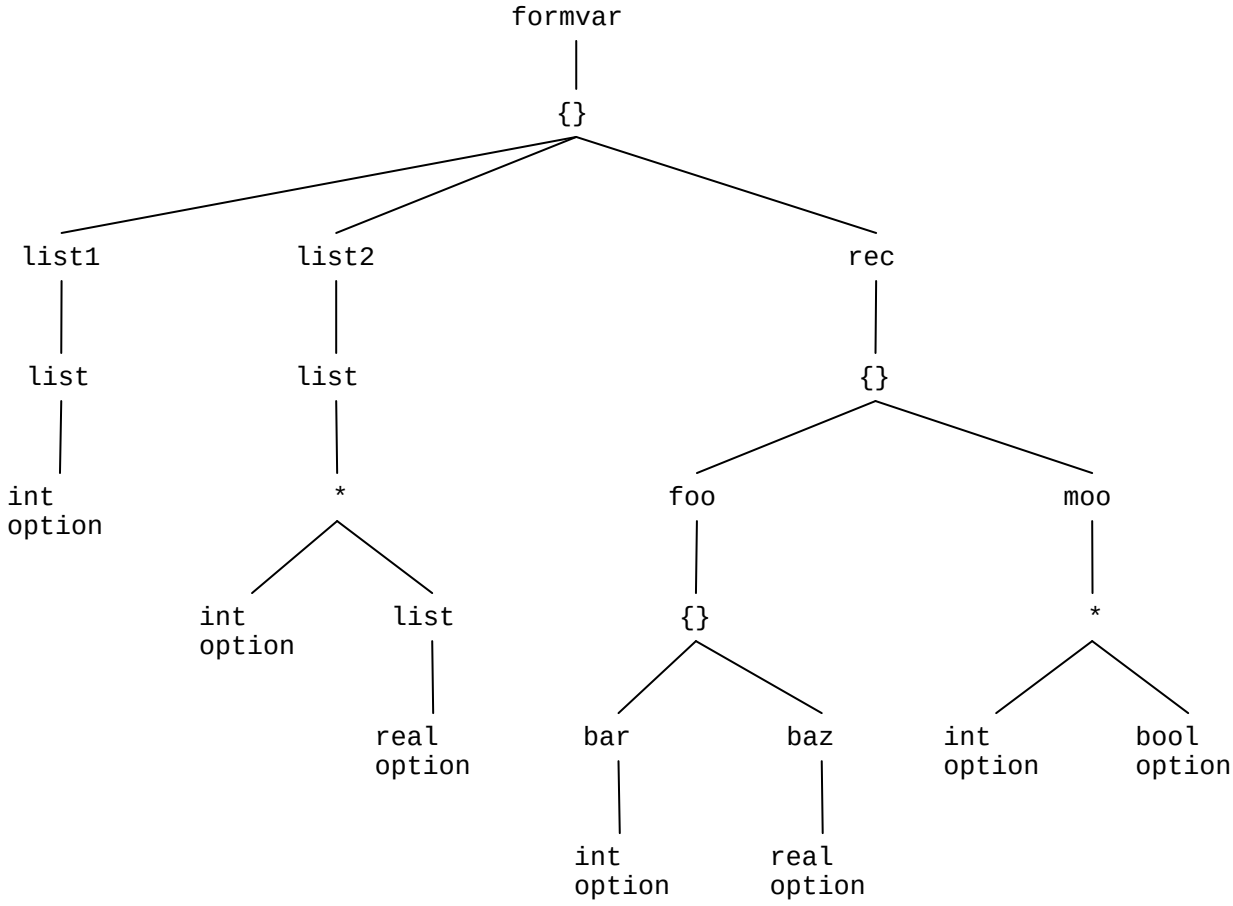

Abbildung 6 Baumdarstellung von `formvar`

Die Blätter eines Baums können über Adressen, die Pfade innerhalb des Baums von der Wurzel bis zu einem Blatt darstellen, identifiziert werden. Um also ein Blatt zu identifizieren, genügt es, in dem Namen einer Variablen in einem HTML-formular die Adresse des Blattes anzugeben. Abbildung 7 zeigt die Grammatik der Adressierung.

start	::=	*name*[*address*]
address	::=	*comp*[*address*]
comp	::=	*recordlabel* | *tupleindex* | *lvindex*
recordlabel	::=	{*label*}
tupleindex	::=	{*index*}
lvindex	::=	[*index*]

Abbildung 7 Grammatik der `formvar`-Adressierung

Beispielsweise würde der Inhalt des Wertes, der unter dem Label `bar` unterhalb von `foo` in dem Record `#rec formvar` abgelegt ist, mit dem Inhalt einer Formularvariablen „`rec{foo}{bar}`“ gefüllt.
Listen- beziehungsweise Vektorindizes werden wie in SML/Alice üblich mit `0`, Tupelindizies mit `1` beginnend durchnumeriert. So wird der zweite Wert in der Liste `#list1 formvar` von der Formularvariablen `list1[1]` gefüllt. Eventuell übersprungene Listen- oder Vektorwerte werden mit `NONE` beziehungsweise `ENONE` belegt. Dies geschieht unabhängig davon, wie viele Einträge in der Liste/in dem Vektor übersprungen wurden. Abbildung 8 enthält einige Beispiele.

Übergebene Werte	Inhalt von `formvar`

`list1[0] = "3"`
`list1[1] = "5"`
`list1[3] = "7"`
`rec{foo}{bar} = "7"`
`rec{moo}{2} = "true"`
	`{ list1 = [SOME 3,SOME 5,NONE,SOME 7],`
	`  list2 = nil,`
	`  rec  = {`
	`  foo = {`
	`  bar = SOME 7, baz = NONE`
	`  },`
	`  moo = (NONE,SOME true)`
	`  }`
	`}`

`rec{foo}{bar} = "1"`
`rec{foo}{baz} = "5.0"`
`rec{moo}{1} = "17"`
	`{ list1 = nil,`
	`  list2 = nil,`
	`  rec  = {`
	`  foo = {`
	`  bar = SOME 1, baz = SOME 5.0`
	`  },`
	`  moo = (SOME 17,SOME false)`
	`  }`
	`}`

`list1[1] = "~3"`
`list2[0]{1} = "-54"`
`list2[0]{2}[0] = "2.0"`
`list2[0]{2}[2] = "~1.0"`
`list2[0]{2}[3] = "-2.0"`
	`{ list1 = [NONE,SOME ~3],`
	`  list2 = [(SOME ~54,`
	`  [SOME 2.0,SOME ~1.0,SOME ~2.0])],`
	`  rec  = {`
	`  foo = {`
	`  bar = NONE, baz = NONE`
	`  },`
	`  moo = (NONE,SOME false)`
	`  }`
	`}`

Abbildung 8 Verhaltensbeispiele der Schnittstelle für komplexe Typen

3.3.3 Der Typ `file`

Webformulare ermöglichen unter anderem auch einen Dateiupload auf den Webserver. Dabei ist es wichtig, das der encoding-type (`enctype`-attribut des HTML-form-tags) auf „`multipart/form-data`“ gesetzt ist („`multipart/mixed`“ wird zurzeit noch nicht unterstützt). Um Dateiuploads mit Hilfe der Alice Server Pages verwalten zu können, gibt es den Typen `file`:

```
type file = {name : string, typ : mime, content : string}
```

Das Feld `name` ist der Dateiname, `content` der Inhalt, und `typ` der MIME-Typ der Datei. Letzterer wird durch folgende Konstruktortypen repräsentiert:

```
datatype application =
    POSTSCRIPT
  | PDF
  | OCTETSTREAM
  | OTHER_APPLICATION of string

datatype text =
     PLAIN
   | HTML
   | OTHER_TEXT of string
datatype image =
    GIF
  | JPEG
  | PNG
  | BMP
  | OTHER_IMAGE of string

datatype mime =
    TEXT of text
  | IMAGE of image
  | APPLICATION of application
  | OTHER of string
```

Dadurch kann der Programmierer mit Hilfe von Pattern Matching den Dateityp bequem abfragen.
Wenn eine Datei hochgeladen werden soll, sollte im Typ `formvar` die entsprechende Formularvariable mit `file option` getypt sein (`file eoption` macht keinen Sinn und ist daher nicht erlaubt).

3.3.4 Manipulation von Cookies

Im Fall von Cookies muss der Programmierer nun noch in die Lage versetzt werden, diese zu setzen, beziehungsweise bereits gesetzte Cookies zu entfernen. Dazu stehen ihm, nach korrekter Definition eines Typs `cookies`, folgende Prozeduren zur Verfügung:

```
setCookies:    cookies -> unit
deleteCookie:  string -> unit
gotCookies:    unit -> bool
```

Wie zu erwarten setzt `setCookies` eine (vorher durch den `cookies`-typ definierte) Reihe von Cookies beim anfragenden Browser. `deleteCookie` löscht ein Cookie, unter Angabe seines Labels/Namens als Zeichenkette.
`gotCookies` liefert `true` genau dann, wenn Cookies vom Client-browser mitgesendet wurden. Der Wert `cookies` muss in diesem Fall jedoch nicht unbedingt Daten enthalten, da die mitgesendeten Cookies nicht der Formvorgabe des Typs `cookies` entsprechen müssen.
Wir erweitern unsere Beispielseite von oben um Cookies. Dabei interpretieren wir das Feld `stringvar` in `formvar` als den Namen des Websurfers und speichern diesen in einem Cookie ab.

Um diese Funktionalität zu erreichen, müssen wir einige Änderungen vornehmen. Zum einen definieren wir den Typ `cookies`, zum anderen erweitern wir `formvar` um ein weiteres Feld:

```
<?amsp
  import structure Fak from "Fak"
  type formvar = {
      charvar   : char eoption,
      stringvar : string option,
      intvar    : int eoption,
      realvar   : real eoption,
      boolvar   : bool eoption,
      setName   : bool option
  }
  type cookies = { name : string option }
?>
...
```

Das Feld „`setName`“ stellt den Inhalt einer Checkbox dar, und nur, wenn diese angewählt wurde, soll der Inhalt von `stringvar` in einem Cookie abgespeichert werden. Wir haben also folgende Fallunterscheidungen:

1. `setName = SOME true` und `#stringvar formvar` nicht leer
   → Setze den neuen Namen und gib ihn aus
2. sonst (setName = `SOME false`) →
   i. Es wurden Cookies gesendet (`gotCookies() = true`)
      → gib den Namen des Websurfers aus
   ii. Es wurden keine Cookies gesendet (`gotCookies() = false`)
      → tue nichts

Den Namen verwalten wir in der Service Page unter einem Bezeichner `name`, den wir ganz am Anfang deklarieren:

```
...
  type cookies = { name : string option }
  val stringvar = getOpt(#stringvar formvar,"")
  (* fetch the name of the user *)
  val name =
      if #setName formvar = SOME true andalso stringvar <> ""
          then let val cookies = { name = SOME stringvar }
               in (setCookies cookies; SOME stringvar) end
      else if gotCookies() then #name cookies
           else NONE
?>
...
<h1>Alice Server Pages
        – Funktionale Programmierung f&uuml;r das Web</h1>
<p>Heute ist:
    <?amsp= Date.toString(Date.fromTimeLocal(Time.now())) ?>
</p>
<?amsp
  val _ =
  (* print a greeting message if a name of the user is known *)
      case name of
          SOME s => print("<p>Willkommen zur&uuml;ck, " ^ s ^ "</p>")
        | _      => ()
?>
```

```
...
```

Jetzt müssen wir noch das Formular um die Checkbox erweitern:

```
...
  <form action="index.amsp">
    Ein Zeichen:              <input type="text" name="charvar"><br>
    Eine Zeichenkette(z.B. ihr Name):
      <input type="text" name="stringvar">
      <input type="checkbox" name="setName" value="true">
        als Name speichern<br>
    Eine ganze Zahl:          <input type="text" name="intvar"><br>
    Eine Fliesskommazahl:     <input type="text" name="realvar"><br>
    Ein Wahrheitswert:        <input type="text" name="boolvar"><br>
    <input type="submit" name="button" value="go!">
  </form>
...
```

Auf diese Weise begrüßt die Webseite den Benutzer mit einer kleinen Willkommensnachricht, sofern vorher ein Name definiert wurde.

3.3.5 Die Beispielseite

In den obigen Abschnitten wurde eine Alice Server Page sukzessive erweitert. Dabei wurden immer nur die für die Erweiterung relevanten Code-schnipsel angegeben. Zum besseren Verständnis ist der vollständige Code der Seite hier abgebildet:

```
<?amsp
  import structure Fak from "Fak"
  type formvar = {
      charvar   : char eoption,
      stringvar : string option,
      intvar    : int eoption,
      realvar   : real eoption,
      boolvar   : bool eoption,
      setName   : bool option
  }
  type cookies = { name : string option }
  val stringvar = getOpt(#stringvar formvar,"")
  (* fetch the name of the user *)
  val name =
      if #setName formvar = SOME true andalso stringvar <> ""
          then let val cookies = { name = SOME stringvar }
               in (setCookies cookies; SOME stringvar) end
      else if gotCookies() then #name cookies
           else NONE
?>
<html>
  <head>
    <title>
      Alice Server Pages
        – Funktionale Programmierung f&uuml;r das Web
    </title>
  </head>
  <body>
    <h1>Alice Server Pages
        – Funktionale Programmierung f&uuml;r das Web</h1>
    <p>Heute ist:
      <?amsp= Date.toString(Date.fromTimeLocal(Time.now())) ?>
```

```
    </p>
    <?amsp
      val _ =
      (* print a greeting message if a name of the user is known *)
          case name of
              SOME s => print("<p>Willkommen zur&uuml;ck, "
                              ^ s ^ "</p>")
            | _      => ()
    ?>
    <p>Inhalt von formvar: <?amsp= formvar ?></p>
    <form action="index.amsp">
      Ein Zeichen:           <input type="text" name="charvar"><br>
      Eine Zeichenkette(z.B. ihr Name):
        <input type="text" name="stringvar">
        <input type="checkbox" name="setName" value="true">
          als Name speichern<br>
      Eine ganze Zahl(n):    <input type="text" name="intvar"><br>
      Eine Fliesskommazahl:  <input type="text" name="realvar"><br>
      Ein Wahrheitswert:     <input type="text" name="boolvar"><br>
      <input type="submit" name="button" value="go!">
    </form>
    <p>Fakult&auml;ten von 0 bis n (sofern 0<=n<=12):</p>
    <ul>
    <?amsp
      open Fak
      (* Try to print the faculty numbers between 0 and n *)
      val _ =
          case #intvar formvar of
              (* n was given; check the domain *)
              ESOME n =>
                  if n<0 orelse n>12
                  then print "<li>0<=n<=12 ist nicht erf&uuml;llt!"
                  else List.app
                       (fn s => print("<li>" ^ Int.toString s))
                       (List.tabulate(n+1,fak))
              (* the n was not formatted as an integer *)
            | EWRONGFORMAT => print "<li>n war keine ganze Zahl!"
              (* no n was given; gimme! *)
            | ENONE => print "<li>Geben sie eine Zahl 0<=n<=12 an."
    ?>
    </ul>
  </body>
</html>
```

Die Beispielseite sieht, nachdem das Formular mit einigen Werten gefüllt wurde, so aus:

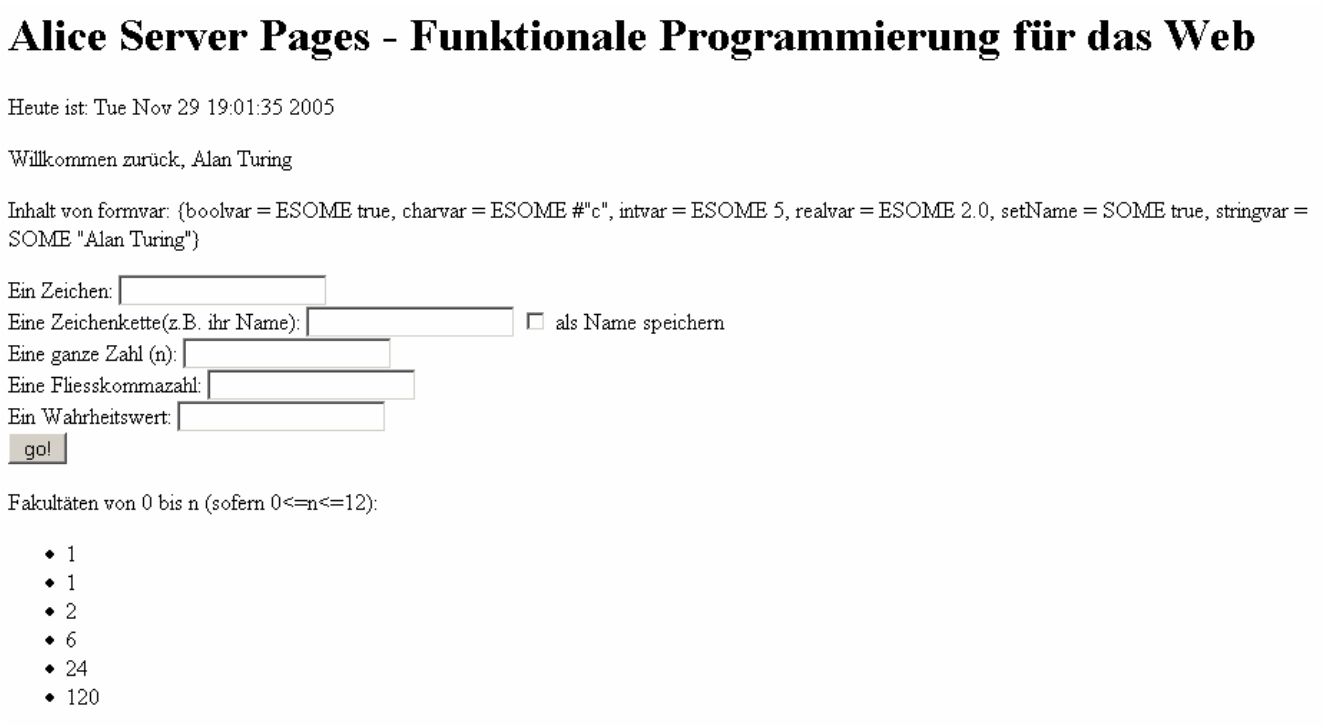

Abbildung 9 Die Beispielseite

3.3.6 Diskussion

3.3.6.1 Eine sehr restriktive Schnittstelle?

Die in den Abschnitten 3.3.1 bis 3.3.5 vorgestellte Schnittstelle legt dem Programmierer einer Server Page einige Restriktionen auf:

1. Der Programmierer muss `formvar`/`cookies` am Anfang der Webseite, zwischen imports und dem eigentlichen Code, deklarieren
2. Er muss sich im Klaren darüber sein, wie viele und welche Variablen er benötigt
3. Er muss im Code der Seite zumindest zwischen korrekter und inkorrekter oder nicht erfolgter Übergabe einer Variablen unterscheiden
4. Durch die strikte Grammatik ist die Verwendung von vorher bereits definierten Typen und deren Verwendung innerhalb von `formvar`/`cookies` nicht möglich

Zunächst sei das wichtigste aller Argumente für die meisten dieser Restriktionen genannt: statische Typprüfung. Durch die Angabe eines Typs `formvar` kann die Alice Server Page Engine (AMSP Engine, siehe 5.2) eine Alice-source erstellen, die eine Deklaration für einen Wert `formvar` enthält. Der Alice first-class Compiler kann dessen typkorrekte Verwendung dann zur Compilezeit überprüfen.
Des Weiteren ist es die Überzeugung des Autors, dass diese Restriktionen dazu beitragen, sauber, lesbar und übersichtlich zu programmieren. Zur Lesbarkeit und Übersichtlichkeit trägt vor allem Restriktion 1 bei. Restriktionen 2 und 3 sollten sowieso für jede saubere Entwicklung selbstverständlich sein. Restriktion 4 trägt wiederum zur Lesbarkeit und zum Codeverständnis bei: Ein Leser kann sich sicher sein, dass es sich bei den von `formvar`/`cookies` angegebenen Werte um externe

Werte handelt, und das alle weiteren Deklaration, sofern sie nicht von diesen Werten abhängen, lokal sind.
Insgesamt ist die Schnittstelle, verglichen mit anderen Active Server-Page-Implementierungen, auf einem recht hohen Niveau. Der Programmierer muss sich nicht um Details des HTTP-Protokolls kümmern, oder umständliche Typkonvertierungen durchführen. Allerdings wird er auch zu mehr Überprüfungen und Schreibaufwand gezwungen, wie die Beispielseite unter 3.3.5 zeigt. Der Programmierer muss sich somit der Zustände, die seine Webseite annehmen kann, zwangsläufig bewusst werden, wenn er die dynamischen Daten ausliest.

3.3.6.2 Besonderheit bei Cookies

Wie dem Hinweis in der Grammatik aus Abbildung 4 zu entnehmen ist, sind Listen- beziehungsweise Vektorkonstruktionen bei Cookies nicht erlaubt. Dies wurde absichtlich so festgelegt. Cookies sind *kleine* Datenpakete, die in der Regel zur Identifikation des anfragenden Benutzers dienen, damit die Webseite benutzerspezifisch gestaltet werden kann. Cookies wurden nicht als Sammlungen von Datensätzen konzipiert. Eine derartige Verwendung ist ineffizient, sie belastet die Bandbreite und den Platz des anfragenden Klienten. Um größere Benutzermodelle abzuspeichern, werden in der Regel serverseitige Datenbanken verwendet, das Cookie beinhaltet nur den Schlüssel zu dem entsprechenden Datensatz.

3.4 Ereignissgesteuerte Prozeduraufrufe

Die ereignissgesteuerten Prozeduraufrufe folgen dem Prinzip der ereignisgesteuerten Abarbeitung von Quellcode. Viele professionelle Programmiersysteme (vergleiche 6.3 oder 6.4) ermöglichen diesen, grundsätzlich imperativen, Programmierstil, und in den Alice Server Pages ist eine einfache Implementierung dieses Konzepts enthalten.
Der Programmierer schreibt zunächst eine Prozedur `h`, die den Typ `unit -> unit` haben muss. Diese wird dann ausgeführt, wenn der Benutzer in einem Webformular auf einen Button geklickt hat, der den Namen „`call_h()`“ hat. Der in 3.2.1 definierte „<`?amsp:fcall`“-tag definiert hierbei, wo welche Prozedur aufgerufen wird:
Innerhalb des tags muss ein Attribut „name“ gesetzt werden, welches den Namen der aufzurufenden Prozedur darstellt. Falls die Prozedur nicht vor dem Tag definiert wurde, liefert die AMSP Engine einen entsprechenden Fehler. Da h selbst, damit sie etwas bewirkt, imperativ sein muss, können Ort, Anzahl und Reihenfolge der Aufrufe eine Rolle spielen.
Betrachten wir zur Erläuterung folgende Bespielseite:

```
<?amsp
type formvar = { n : int option }

(* computing something *)
fun fib n =
    case Int.compare(n,1) of
        LESS    => 0
      | EQUAL   => 1
      | GREATER => if n > 25 then raise Overflow
                   else fib (n-1) + fib (n-2)

fun fak n = if n < 2 then 1 else n * fak (n-1)
```

```
(* displaying result *)
fun display f n =
   (case n of SOME i => print (Int.toString (f i))
            | NONE   => print "Bitte geben sie eine ganze Zahl an!")
   handle Overflow   => print "Die Zahl war zu gross!"

fun displayFak() = display fak (#n formvar)
fun displayFib() = display fib (#n formvar)
?>

<html>
<head>
 <title>Formulargesteuerter Prozeduraufruf</title>
</head>
<body>
 <form action="functest2.amsp">
   n <input type="text" name="n">
   <input type="submit" name="call_displayFak()"
       value="berechne Fakult&auml;t!">
   <input type="submit" name="call_displayFib()"
       value="berechne nte Fibonacci-Zahl!">
 </form>

 <?amsp:fcall name="displayFak" ?>
 <?amsp:fcall name="displayFib" ?>

</body>
```

Falls der Benutzer eine ganze Zahl in das Formularfeld „n“ eingibt und den Button mit dem Namen „`call_displayFak()`“ betätigt, wird die Prozedur displayFak an der Stelle des Tags „`<?amsp:fcall name="displayFak" ?>`“ aufgerufen, und es wird die Fakultät von n an der Stelle des Tags ausgegeben. Entsprechendes gilt für `displayFib`. Das Attribut „`name`“ des fcall-tags ist notwendig, fehlt es, wird zur Compilezeit ein Fehler ausgegeben.

4 Architektur

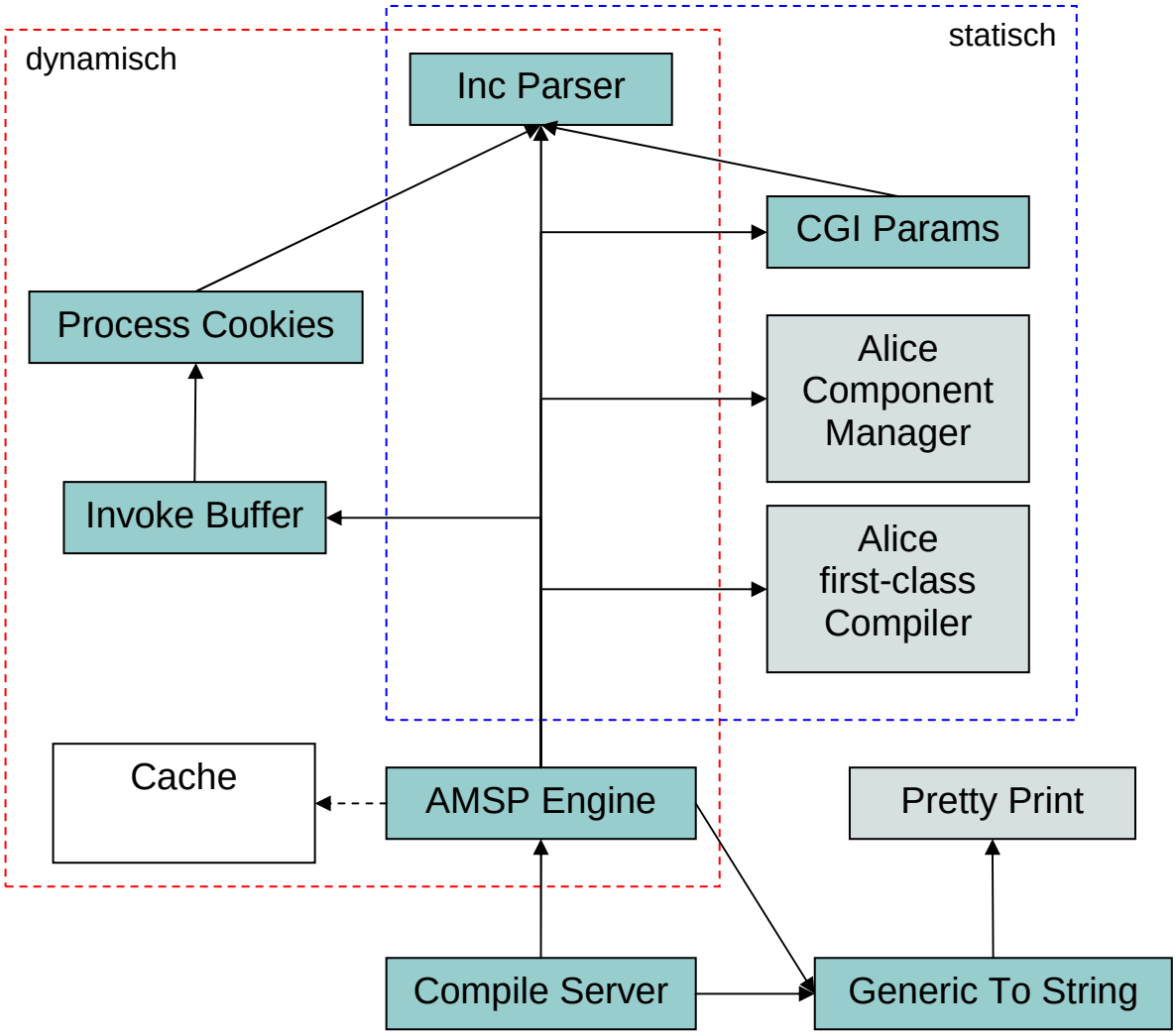

Abbildung 10 Der Compile Server

Die zentrale Komponente der Verarbeitung einer Server Page, also die Generierung einer Alice-quelldatei aus der Server Page, die Kompilierung, und den Aufruf des Kompilats, ist der **Compile Server** (siehe auch 5.1). Er beantwortet Anfragen des Anfrage/Antwort-Clients. Sobald eine Anfrage anliegt, wird die **AMSP Engine** (siehe auch 5.2) darauf angesetzt. Diese überprüft, ob es bereits eine aktuelle Version der angefragten Seite im Cache gibt. Falls ja, ruft sie diese mit Hilfe des **Alice Component Managers** direkt auf, falls nein, erstellt sie unter Zuhilfenahme des **Alice first-class Compilers** zunächst ein neues, aufrufbares Kompilat. Abbildung 11 veranschaulicht den Aufbau der Quelle dieses Kompilats.

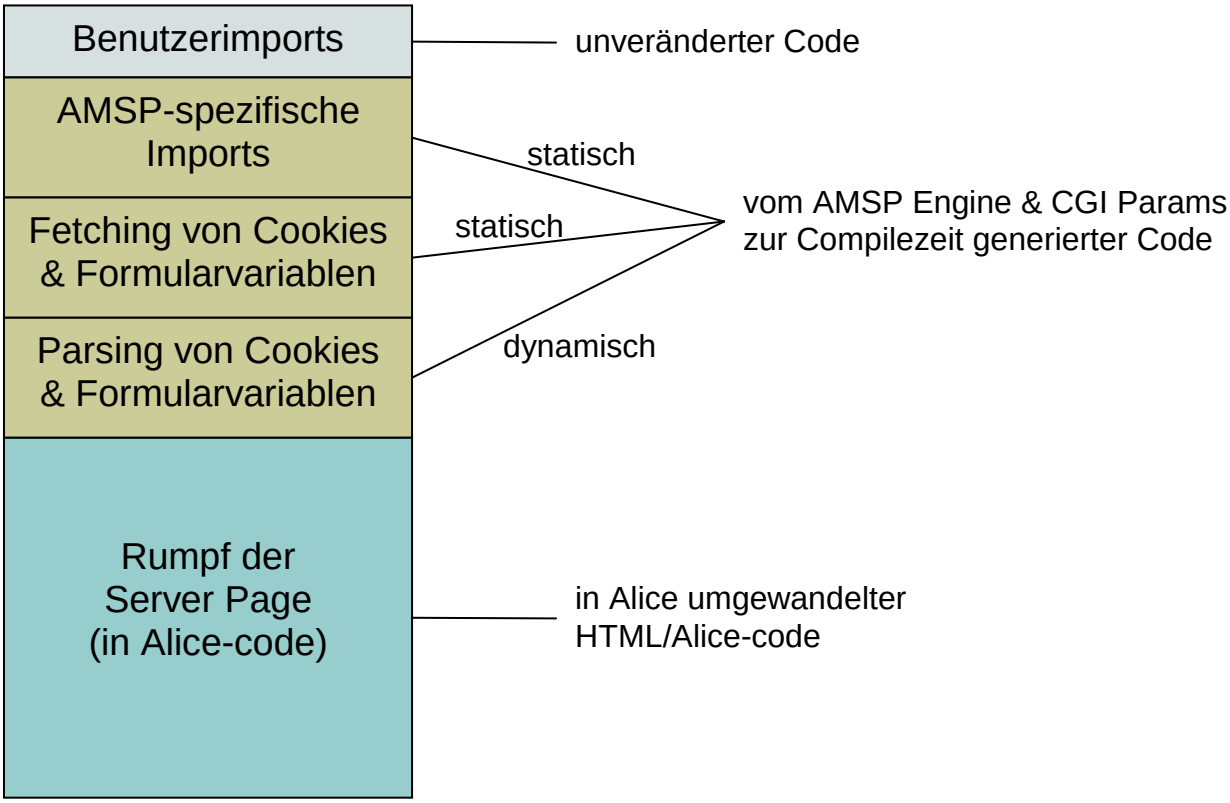

Abbildung 11 Aufbau der verarbeiteten Server Page

Die Server Page wird zunächst in puren Alice-code umgewandelt. Diese Umwandlung wird direkt von der AMSP Engine durchgeführt, dabei werden die HTML-teile in Alice-„`print`“-statements umgewandelt.[1]
Das weitere Parsing auf diesem noch unverarbeiteten Alice-quellcode wird unter anderem von dem **Inc Parser** (Incremental Parser, siehe auch 5.6) zur Verfügung gestellt wird. Dieses Modul enthält Parser-prozeduren, die nur soviel von einer Zeichenkette parsen, wie sie verstehen. So müssen beispielsweise die imports beziehungsweise die Typen `formvar` und `cookies` am Anfang einer Server Page erkannt und geparst werden. Lexen oder Parsen der gesamten Seite ist für die Erstellung der Quelle des Kompilats nicht notwendig.
Zwischen den imports der Server Page (Benutzerimports) und dem Quellcode werden noch imports von verschiedenen AMSP-komponenten gehängt. Unter anderem wird dabei der **Invoke Buffer** importiert. Er ermöglicht die Kommunikation zwischen Compile Server und ausgeführtem Kompilat (siehe auch 5.3). Dazu benötigt er **Process Cookies** (siehe auch 5.5), um die Teile der HTTP response zu generieren, die Cookies setzen.
Aus den Syntaxbäumen der `formvar`/`cookies`-Typen erstellt **Cgi Params** Quellcode, der die Werte `formvar`/`cookies` erzeugt (siehe auch 5.4).
Die Struktur **Generic To String** stellt die Schnittstelle zum Alice **Pretty Printer** dar, welcher für AMSP-Ausdrucksfragmente („<`?amsp=`“, siehe 3.2.1) benötigt wird.
Eine für die fertige Alice-quelle wichtige Struktur ist **Type Convert**[2] (siehe auch 5.7). Type Convert stellt die Typkonvertierungsprozeduren zur Verfügung, die nötig sind,

[1] Dieser Teil der Übersetzung wurde aus den ML Server Pages[1] übernommen
[2] Type Convert wird nur von der Alice-quelle importiert und taucht daher nicht in der Architektur des Compile Server auf (Abbildung 10)

um aus einfachen Zeichenketten diverse Alice-werte zu erzeugen, und wird daher für die Erstellung der Werte `formvar` und `cookies` benötigt.

5 Implementierung

In diesem Abschnitt wird etwas genauer auf die Funktionsweise der in 4 erwähnten Strukturen eingegangen.

5.1 Compile Server

Der Compile Server beantwortet Anfragen des Anfrage/Antwort-Clients. Dazu verwendet er die AMSP Engine, die sich um die Kompilierung und den Aufruf einer Server Page kümmert. Seine relativ einfache Signatur ist wie folgt:

```
signature COMPILE_SERVER =
    sig
      type port = Socket.port

      val myPort : port option ref
      val logOut : TextIO.outstream option ref

      val start  : port option -> port
    end
```

`start` startet den Compile Server. `myPort` sollte vor start des Compile Servers festgelegt werden (Änderung nach start des Servers hat keinen Einfluss mehr bis zum nächsten Neustart) und ist der Port auf dem der Server Anfragen des C-Clients erwartet. `logOut` ist der Ausgabestrom, auf den Fehlermeldungen gelegt werden (standardmäßig werden Fehler in die Datei "`amspcd.error_log`" geschrieben). Fehler, das heißt geworfene Ausnahmen, werden beim Compile Server abgefangen und an den anfragenden Browser in Form eines einfachen Text-dokuments[1] gesendet.

5.2 AMSP Engine

Die AMSP Engine kompiliert Server Pages, falls keine aktuelle Version mehr im Cache liegt. Die beiden wichtigsten Prozeduren sind:

```
val update   : string -> string (* CompileError *)
val invoke   :
    string * string * string * string * string -> wseq * int
```

`update` erwartet den absoluten Pfad einer Server Page, berechnet mit Hilfe von MD5 daraus den Namen eines Kompilats, und überprüft, ob dieses im Cache liegt. Falls nicht, wird dieses mit Hilfe der internen Prozedur `compile` erzeugt. `invoke` erwartet unter anderem den Namen des Kompilats, den cookie- und den querystring (enthalten die rohen Daten der Cookies und Formularvariablen), initialisiert den Invoke Buffer und ruft das Kompilat mit Hilfe des Alice Component

[1] MIME-Typ „`text/plain`“

Managers auf. Die Ausgabe von `invoke` ist die HTTP response der Server Page, welche sowohl die zu setzenden Cookies als auch den Inhalt der generierten HTML-seite enthält. Die zweite Komponente des Tupels ist die Länge der HTTP response in Byte.

5.3 Invoke Buffer

Der Invoke Buffer ist für die Kommunikation zwischen Kompilat und Compile Server verantwortlich. Er ist eine Struktur mit Zustand, der durch den Aufruf eines Kompilats verändert und anschließend von der AMSP Engine ausgelesen wird[1]. Die wichtigsten Prozeduren des Invoke Buffers sind:

```
val print        : string -> unit
val printSeq     : wseq -> unit
val setCookie    : ProcessCookies.cookiedata -> unit
val deleteCookie : string -> unit
```

Die beiden „`print`“-prozeduren dienen der Ausgabe von Zeichenketten an die erstellte HTML-seite aus einem Alice-fragment heraus. Dabei werden die Zeichenketten zunächst an eine Word sequence „`htmlBuffer`“ angehängt, die in einer Referenz im Invoke Buffer gespeichert wird. Beide sind auch auf Toplevel verfügbar.
`setCookie`, nicht zu verwechseln mit `setCookies` aus 3.3.4, erwartet ein Cookie in *roher* Form, wandelt dieses in eine Zeichenkette um und hängt es ebenfalls an eine in einer Referenz gespeicherten Word Sequence `cookieBuffer`. Diese wird, wenn die HTTP response von der AMSP Engine aufgebaut wird, ausgelesen (Cookies müssen in der HTTP response vor dem eigentlichen Inhalt stehen).
`setCookie` wird von `setCookies` aufgerufen, `setCookies` hingegen ist eine (zwangsläufig) zur Kompilierzeit generierte Prozedur, basierend auf dem vom Programmierer definierten Typ `cookies`. `deleteCookie` löscht ein Cookie. Falls das Cookie nicht vorher gesetzt wurde, bewirkt ein Aufruf von `deleteCookie` nichts. Beide Cookieprozeduren sind auf Process Cookies angewiesen.

5.4 CGI Params

CGI Params generiert den Quellcode, der die Werte `formvar` und `cookies` füllt und die Prozedur `setCookies` enthält. Diese Arbeit wird von der Prozedur `buildmain` ausgeführt:

```
val buildMain : string -> wseq
```

Die Erzeugung der Werte `formvar` und `cookies` wird unter Verwendung von Type Convert sowie den geparsten Formular- und Cookiedaten durchgeführt.
`setCookies` generiert für jedes Label im record `cookies` ein Cookie. Falls sich unter einem labil *c* im Typ `cookies` ein komplexeres Objekt *r* bestehend aus geschachtelten Records oder Tupeln befindet, erzeugt `setCookies` eine Stringrepräsentation von *r* und setzt dieses als Inhalt des Cookies *c*.

[1] Durch diese Vorgehensweise kann der Compile Server Anfragen noch nicht nebenläufig beantworten, siehe auch Abschnitt 7.

5.5 Process Cookies

Process Cookies parst gesendete Cookies vom Clientbrowser und erzeugt Cookie-HTTP-header, die Cookies setzen oder vorhandene entfernen. Die wichtigsten Prozeduren sind:

```
val processCookie : cookiedata -> string
val processRemoveCookie :
    { name : string, path : string option } -> string
val parseCookies  : string option -> IncParser.srv StringMap.map
```

`processCookie` erstellt zu einem Record mit Cookiedaten einen HTTP Cookie header. Der Record enthält unter anderem Felder wie den Namen oder den Inhalt des Cookies.
`processRemoveCookie` erstellt zu dem Namen und dem Pfad[1] eines Cookies einen HTTP Cookie Header, der das angegebene Cookie aus dem Client Browser entfernt. Dazu wird das Verfallsdatum des Cookies einfach in die Vergangenheit gesetzt.
`parsecookies` parst, unter Verwendung von Inc Parser, den vom Client Browser empfangenen Cookie String in eine Map-datenstruktur (basiert auf Rot-Schwarzbäumen). Diese bildet Zeichenketten auf entweder wieder eine Zeichnkette oder noch mal eine Map-struktur ab, abhängig davon, ob unter dem Cookie einfache Daten stehen (also Werte vom Typ `int`, `real`, `string`, `char` oder `bool`, in Zeichenkettenform), oder weitere Records/Tupel geschachtelt sind. Diese Struktur wird später im Kompilat unter Zuhilfename von Type Convert gegen den Syntaxbaum des Typs `cookies` gecheckt: Werte, die unter demselben Pfaden in beiden Bäumen erreichbar sind, landen dann im Wert `cookies`.

5.6 Inc Parser

Inc Parser enthält Parserprozeduren und Konstruktortypen für Syntaxbäume. Die Parserprozeduren sind dabei in der Regel so konstruiert, dass sie von einer Zeichenkette nur soviel lesen, wie sie laut ihrer Grammatik erkennen können. Der Rest der Zeichenkette wird unberührt zurückgegeben. Die wichtigsten Parserprozeduren sind:

```
val parseRt  : string -> record_type * string
val parseSrv : string -> srv * string
```

`parseRt` part die Recordtypen `cookies` und `formvar` und gibt den Rest des Quellcodes sowie den Syntaxbaum zurück. Dieser basiert auf folgendem Konstruktortyp:

```
datatype rt =
    Int
  | Real
  | Char
  | String
  | Bool
```

[1] Der Cookiepfad wird bei Cookies der Alice Server Pages nicht gesetzt; Er ist nur deshalb Argument von `processRemoveCookie`, weil Name und Pfad ein Cookie eindeutig identifizieren.

```
    | File
    | Option of rt
    | EOption of rt
    | Vector of rt
    | List of rt
    | Record of (string * rt) list
    | Tuple of rt list
type record_type = {name:string,value:rt}
```

`parseSrv` parst den vom Client Browser übermittelten Cookiestring und wandelt ihn in eine Map-datenstruktur `cookieDict` um:

```
datatype srv =
    R of srv StringMap.map
  | V of string
```

Der Syntaxbaum von `cookies` kann bei Erstellung des Werts `cookies` gegen diese Datenstruktur gecheckt werden. Falls ein Wert gefunden wird, der denselben Pfad im Syntaxbaum als auch in `cookieDict` hat, wird er unter das Blatt im Wert „`cookies`“ abgelegt.

5.7 Type Convert

Type Convert dient der Konvertierung von Zeichenketten in die laut der Grammatik in Abbildung 4 möglichen Alice-werte. Eine etwas interessantere Prozedur aus Type Convert ist `collectValues`:

```
val collectValues :
    string list StringMap.map -> (string -> 'a) -> string -> 'a list
```

`collectValues` baut zu einer Map-datenstruktur *fv*, die die geparsten Formularvariablen enthält, einer Umwandlungprozedur *f* und einem Namen *x* eine Liste von Werte auf. Sie wird benutzt, um Listen- und Vektorwerte aufzubauen. Der Clou bei dieser Konstruktion ist die übergegebene Prozedur *f*: Sie ist wieder eine Typkonvertierungsprozedur, die alle Werte, die in der Liste unter dem Namen *x* liegen, aufbaut.
Das bedeutet beispielsweise für einen Wert des Typs `int option list list`, der unter dem Label `doubleList` abgelegt ist, dass zwei Aufrufe von `collectValues` geschachtelt werden, und der äußere Aufruf eine aufgerufene Version von `collectValues` als *f* erhält. *x* wird dem inneren `collectValues`-Aufruf dabei jeweils als ein teilweise festgelegter Pfad ausgehend von `doubleList` mitgegeben, also zum Beispiel `doubleList[2]`, sofern *fv* eine Variable mit Präfix `doubleList[2]` im Namen enthält.
Dies ist ein praktisches Beispiel für die Verwendung von höherstufigen Funktionen.

6 verwandte Projekte

Die im Folgenden vorgestellten Server Page Implementierungen verwenden, mit Ausnahme der ML Server Pages, nur *imperative* Programmiersprachen[1]. Die Alice Server Pages bieten somit jedem Programmierer, der den funktionalen Stil bevorzugt, eine bequeme Alternative.

6.1 ML Server Pages

Da die Alice Server Pages auf den ML Server Pages basieren und viele Techniken übernommen haben, liegt die Frage nahe, was sich eigentlich im Vergleich zu den ML Server Pages verbessert hat.
An dieser Stelle wäre zunächst die bequemere Schnittstelle zu Formularvariablen und Cookies (siehe 3.3) zu nennen, die dem Programmierer zum einen umständliche Typkonvertierungen abnimmt, zum anderen Sammlung und hierarchische Anordnung der Werte in Records/Tupel/Listen/Vektoren ermöglicht. Datei-uploads sind bei den ML Server Pages besonders schwierig zu handhaben, da die Schnittstelle hierfür sehr low-level gehalten ist. Die typkorrekte Verwendung der Daten wird durch die statische Typprüfung sichergestellt. Die Restriktion, die Schnittstelle am Anfang einer Server Pages definieren zu müssen, sorgt für Übersichtlichkeit und erhöht die Lesbarkeit des Codes.
Des Weiteren erlaubt die Architektur der Alice Server Pages zum einen eine Auslagerung des Compile Servers auf einen anderen physikalischen Rechner als den des HTTP-servers, um dessen Workload gegen etwas mehr traffic einzutauschen. Zum anderen ist ein Austausch der HTTP-server-schnittstelle bei den Alice Server Pages relativ einfach zu realisieren (in diesem Fall muss der C-Client ausgetauscht werden), während die ML Server Pages nur für CGI ausgelegt sind.

6.2 PHP

PHP[4] hat sich in den letzten Jahren als eine gute Server Page-lösung für kleinere, oft auch private Projekte, entwickelt. Die Tatsache, dass die PHP-programmiersprache zum einen nicht getypt ist, zum anderen dem Programmierer viele Freiheiten erlaubt, machen PHP allerdings für professionelle, industrielle Entwicklung unzureichend. Im Vergleich zu den Alice Server Pages bietet PHP eine sehr große Library, die unter anderem Schnittstellen zu vielen Datenbanksystemen, aber auch externen Systemen wie .Net oder SAP bietet.
PHPs Zugriffe auf Formularvariablen laufen jedoch, unter default-Einstellung, relativ low-level ab und setzen Kenntnisse des HTTP-Protokolls voraus.
Allerdings fasst PHP Daten, sofern möglich beziehungsweise erkennbar, in Datenstrukturen (arrays) zusammen und erlaubt eine bequeme, wenn auch unsichere (ungetypte), Weiterverarbeitung.

[1] Dies galt im Zeitraum der Implementierung der Alice Server Pages; Da ASP.Net sprachunabhängig ist, ist eine Erweiterung des .Net-Frameworks um eine funktionale Sprache nicht auszuschliessen.

6.3 ASP.Net

ASP.Net[5] ist die Active Server Page-Implementierung von Microsoft. ASP.Net ist sprachunabhängig, und unterstützt bereits über 20 verschiedene Programmiersprachen. Konkret bedeutet das, dass Code-fragmente innerhalb einer Seite in einer anderen Sprache formuliert sein können wie eingebundene Komponenten. Des Weiteren bietet ASP.Net eine Reihe von automatischen Komfort- und Perfomanzfeatures, wie eine Makefile-funktionalität bei der Kompilierung einer Webseite (eingebundene, externe Komponenten werden ebenfalls überprüft und kompiliert), Caching von Kompilaten und Output (bereits berechnete HTML-seiten werden nicht neu berechnet, wenn nicht nötig) oder Sessionmanagement auch über mehrere verschiedene Webserver.
Durch die Anbindung an das .Net-Framework bietet ASP.Net außerdem eine äußert reiche Bibliothek an Klassen (über 4500), die alle Arten von Funktionalität oder Schnittstellen zur Verfügung stellt.
Eine ASP.Net-seite wird in der Regel ereignissgesteuert ausgewertet. Das bedeutet, dass der Programmierer zu einer Reihe von Ereignissen Handler schreiben kann, die genau dann aufgerufen werden, wenn das Ereignis eintritt
So gibt es feste Ereignisse wie „PageLoad“, welches beim Aufbau einer Seite aufgerufen wird. Es können aber auch benutzerdefinierte Ereignisse verwendet werden: Durch die Betätigung eines Buttons in einem Formular kann, ebenso wie bei den ereignisgesteuerten Prozeduraufrufen der Alice Server Pages (3.4), ein solches Ereignis dann gefeuert werden.
In einer ASP-seite kann auf verschiedene Arten auf dynamische Daten zugegriffen werden. Die ASP.Net Server Controls sind dabei die professionelle Lösung dieses Problems. Dabei handelt es sich um HTML-tags, entweder HTML-intrinsische oder spezielle ASP-tags, die ein Attribut „runat=server“ enthalten. Formular- oder Textfelder, die mit Hilfe solcher Tags definiert wurden, können anschließend im Code über spezielle Objekte gelesen und manipuliert werden.
Über so genannte „Validatoren“ können Formulareingaben bereits auf Clientseite überprüft werden, sodass beispielsweise eine syntaktisch falsche E-Mailadresse oder Telefonnummer zurückgewiesen und aus den beim Server ankommenden Daten ausgeschlossen werden kann.
Mit einer derart fortgeschrittenen Active Server Page-Implementierung können die Alice Server Pages natürlich nur schwer konkurrieren. Die Alice Server Pages geben Fans des funktionalen Programmierstils und solchen, die Alice-spezifische Features wie Constraint Programmierung im Zusammenhang mit dynamischen Webseiten nutzen wollen, die Möglichkeit dazu, und bietet eine einfache Handhabung von dynamischen Daten auf hohem Niveau.

6.4 SAPs Business Server Pages (BSP)

Die Business Server Pages[6] sind eine der Active Server Page-implementierungen von SAP. Die Business Server Pages sind, wie viele Programmierkonzepte im SAP System, ereignisgesteuert. So gibt es Ereignisse wie „OnInitialization“, welches immer vor Aufbau der Seite aufgerufen wird, oder „OnInputProcessing“, welches genau dann gefeuert wird, wenn eine Formularvariable einen speziellen Namen hat (vergleiche 3.4 beziehungsweise 6.3).
Die Business Server Pages haben mit den Alice Server Pages gemein, dass Formularvariablen bei beiden separat deklariert werden müssen, bevor ein Zugriff im

Quellcode möglich ist. Wie bei SAP üblich, geschieht dies innerhalb des SAP Systems, und für jedes dieser „Seitenattribute“ gibt es ein Textfeld für eine Kurzbeschreibung.
Wie ASP.Net bietet auch das SAP-system eine reiche Bibliothek, die vor allem auf betriebswirtschaftliche Zwecke ausgelegt ist. Beispielsweise enthält das Modul „CRM“ (Customer Relationship Management) Datenbanktabellen und SAP-Programmierobjekte (Programme, Funktionsgruppen, BAPIs, Klassen etc.), die auf die Verarbeitung von Kundenanfragen, der Beantwortung dieser Anfragen, dem Anstoß eines Workflows etc. ausgelegt sind.

7 Zusammenfassung und Ausblick

Die Alice Server Pages bieten funktionale Programmierung für dynamische Webseiten. Die Fragen aus Abschnitt 1 können folgendermaßen beantwortet werden:

- Wie lässt sich das Typsystem von Alice im Kontext von dynamischen HTML-Seiten nutzen?

  Die statische Typprüfung von Alice sichert die korrekte Verwendung dynamischer Daten.

- Wie fügen sich funktionale Programmieridiome in diesen Kontext ein?

  Bei der Implementierung der Typkonvertierung von Formularvariablen in Alice-werte konnten Rekursion und höherstufige Funktionen eingesetzt werden.

- Welche Möglichkeiten ergeben sich durch Alice-spezifische Features wie Nebenläufigkeit, Laziness oder den Component Manager?

  Der Component Manager von Alice ermöglicht den Aufruf einer Alice-komponente innerhalb eines laufenden Alice-prozesses, wodurch der Zeitaufwand für den Startup eines weiteren Alice-prozesses vermeidet wird. Der Alice First-Class Compiler wird zur Kompilierung der verarbeiteten Alice Server Pages verwendet. Dabei werden die Fehlermeldungen des Compilers an den C-Client und somit an den anfragenden Webserver weitergeleitet. Dies impliziert jedoch ein schwieriges Problem: Die Zeilen- und Spaltenangaben der Kompilierfehler beziehen sich nicht auf die ursprüngliche Quelle, sondern auf die verarbeitete Version, und sind somit nicht korrekt beziehungsweise irreführend.
  Nebenläufigkeit wird zwar in der aktuellen Implementierung im Compile Server eingesetzt, allerdings nur zur Begrenzung der Laufzeit des Aufrufs einer Server Page, um beispielsweise deadlocks abzufangen. Der Compile Server kann momentan noch nicht nebenläufig implementiert werden[1], dies ist eine Limitierung der aktuellen Implementierung.

- Wie lassen sich die Server Pages am besten in einen Webserver(z.B. Apache [3]) integrieren?

[1] Damit der Compile Server Anfragen nebenläufig beantwortet, muss erst die Kommunikation zwischen aufgerufener Webseite (Kompilat) und Compile Server (siehe 5.3) umgestellt werden

Das Konzept des Compile Servers ermöglicht eine Integration der Server Pages in prinzipiell jeden Webserver. Für CGI existiert dabei bereits Client, der die Anfragen des Webservers an den Compile Server weiterleitet. Da der Compile Server auf einen anderen physikalischen Rechner wie den Webserver ausgelagert werden kann, muss der workload des Webservers nicht mit dem des Alice-prozesses zusätzlich belastet werden. Für die Anbindung an weitere Schnittstellen außer CGI muss nur der C-Client ausgetauscht werden, der Compile Server kann weiter verwendet werden.

Abgesehen von der noch ausstehenden Nebenläufigkeit des Compile Servers und den inkorrekten Ortsangaben von Kompilierfehlern gibt es noch weiteres Ergänzungspotenzial für die Alice Server Pages. So bietet Alice momentan bezüglich dynamischer Webseiten, verglichen mit anderen Active Server Page-implementierungen (siehe 6), noch relativ wenige Komponenten. Wünschenswert wären weitere Datenbankschnittstellen (für SQLite und MySQL existieren bereits APIs), Sessionmanagement, E-Mail-unterstützung, Packroutinen oder auch Anbindungen/Schnittstellen an andere Programmiersysteme wie .Net oder SAP R/3.

Bibliographie

[1] Alice Programming Language
http://www.ps.uni-sb.de/alice/

[2] ML Server Pages
http://ellemose.dina.kvl.dk/~sestoft/msp/index.msp

[3] Apache HTTP Server Project
http://httpd.apache.org/

[4] PHP Hypertext Preprocessor
http://www.php.net/

[5] ASP.Net
http://www.asp.net

[6] Business Server Pages
http://www.sap.com/index.epx