Fakultät IV – Elektrotechnik und Informatik
MPGI 1 WS 2013/14
Pepper / Höger / Reicherdt / Zuber / Tutoren

PROJEKTAUFGABE

1 Einleitung

Im Rahmen der Projektaufgabe von MPGI1 implementiert Ihr in Teams einen einfachen Viewer für HTML-Dateien.

HTML[1] ist eine der grundlegenden Sprachen, die zur Beschreibung von WWW-Seiten eingesetzt werden. Mit ihrer Hilfe werden z. B. Texte formatiert und angeordnet, Bilder eingebunden sowie Hyperlinks gesetzt. Da die Implementierung eines Viewers für HTML in seinem vollen Umfang weit über den Rahmen der Veranstaltung MPGI1 hinausginge, beschränken wir uns auf einen sehr kleinen Ausschnitt, den wir TinyHTML nennen, der im Wesentlichen die Beschreibung ausgerichteter Textabsätze erlaubt. Der Viewer liest eine TinyHTML-Datei ein und zeigt den formatierten Text in einem Fenster an. Abbildung 1 zeigt ein Beispiel, in dem rechts- und linksbündige sowie zentrierte und im Blocksatz gesetzte Absätze erkennbar sind.

Die Implementierung des Viewers erfolgt in drei aufeinander aufbauenden Meilensteinen, für die jeweils zwei Wochen Bearbeitungszeit zur Verfügung steht. Die Meilensteine sowie die Eingabesprache TinyHTML werden im Folgenden detailliert beschrieben.

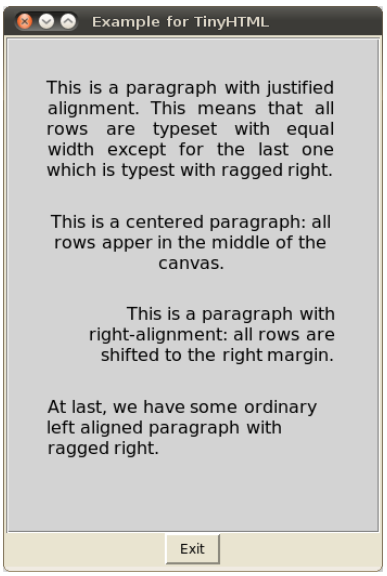

Abbildung 1: Der TinyHTML-Viewer in Aktion.

[1] `http://www.w3.org/standards/webdesign/htmlcss`

2 Die Sprache TinyHTML

TinyHTML ist eine echte Untermenge der Sprache HTML (mit kleinen CSS-Anteilen). Das bedeutet, jede TinyHTML-Datei kann von einem normalen WWW-Browser angezeigt werden. Umgekehrt kann jedoch der TinyHTML-Viewer nur ganz wenige HTML-Dateien anzeigen — nämlich genau jene, die auch TinyHTML-Dateien sind.

Wir beschreiben TinyHTML formal mittels einer kontextfreien Grammatik, die in Abb. 2 gezeigt ist. Wir benutzen dabei die Konvention, dass Terminalsymbole in serifenloser und Nicht-Terminalsymbole in *kursiver* Schrift dargestellt werden. Die in Abb. 2 verwendetet Terminalsymbole sind in den Abb. 3 und 4 definiert. Symbole in `Festbreitenschrift` bezeichnen dabei die Lexeme, wie sie konkret in der Eingabe auftauchen.

Anmerkungen zur Notation In den grammatikalischen Beschreibungen bedeutet * und $^+$ eine Wiederholung des davorstehenden Symbols. Bei A^* darf A auch 0-mal auftreten, bei A^+ muss A mindestens einmal auftreten.

Formal können wir A^* als Abkürzung für

$$R \quad \rightarrow \quad A\ R \ \mid \ \epsilon$$

sowie A^+ als Abkürzung für

$$R \quad \rightarrow \quad A\ R \ \mid \ A$$

auffassen, wobei ϵ das leere Wort bezeichnet.

Anmerkungen zur Lexikalik Die Symbole ␣, ⇆ und ↵ stehen für Leerzeichen, Tabulator und Zeilenende.

2.1 Generelle Eigenschaften der TinyHTML Syntax

Wie aus Abb. 2 ersichtlich ist, besteht ein TinyHTML-Dokument aus sog. *Tags*, die immer paarweise auftreten, z. B. `<body>` und `</body>`. Tags werden immer in Winkelklammern `<`, `>` geschrieben und der schliessende Tag hat denselben Namen wie der öffnende mit einem vorangestellten `/`. Innerhalb des öffnenden und schliessenden Tags können weitere Tags oder Sequenzen von Wörtern stehen.

Eine Ausnahme bildet hier der `<p>`-Tag, der noch über sog. Attribute verfügt, z. B.:

`style="text-align:right"` `style="text-align:center"`

Unabhängig von seinem Attribut wird der `<p>`-Tag immer mit einem `</p>`-Tag geschlossen.

Bei einem korrekten TinyHTML-Dokument müssen Tags paarweise geschachtelt sein, d. h.

```
<body> <p style="text-align:center"> </body> </p>
```

ist nicht gültig, es muss lauten:

```
<body> <p style="text-align:center"> </p> </body>
```

2.2 Beschreibung der einzelnen Sprachelemente

Neben der Angabe der TinyHTML-Syntax durch die Grammatik in Abb. 2 müssen wir auch noch angeben, was die einzelnen Konstrukte bedeuten, d. h. wie sie angezeigt werden sollen. Wir gehen im Folgenden auf die Elemente einer TinyHTML-Datei ein und beschreiben ihre Bedeutung anhand von Beispielen.

$$
\begin{array}{lcl}
\textit{Document} & \rightarrow & \mathsf{open\_html}\ \textit{Title}\ \textit{Body}\ \mathsf{close\_html} \\
\textit{Title} & \rightarrow & \mathsf{open\_title}\ \mathsf{word}^*\ \mathsf{close\_title} \\
\textit{Body} & \rightarrow & \mathsf{open\_body}\ \textit{Par}^*\ \mathsf{close\_body} \\
\textit{Par} & \rightarrow & \mathsf{open\_p}\ \mathsf{word}^*\ \mathsf{close\_p}
\end{array}
$$

Abbildung 2: Grammatik von TinyHTML.

$$
\begin{array}{ll}
\mathsf{open\_html} = \{\texttt{<html>}\} & \mathsf{close\_html} = \{\texttt{</html>}\} \\
\mathsf{open\_title} = \{\texttt{<title>}\} & \mathsf{close\_title} = \{\texttt{</title>}\} \\
\mathsf{open\_body} = \{\texttt{<body>}\} & \mathsf{close\_body} = \{\texttt{</body>}\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{open\_p} = \{\texttt{<p style="text-align:left">}, \\
\qquad\qquad \texttt{<p style="text-align:right">}, \\
\qquad\qquad \texttt{<p style="text-align:center">}, \\
\qquad\qquad \texttt{<p style="text-align:justify">}\} \\
\mathsf{close\_p} = \{\texttt{</p>}\}
\end{array}
$$

Abbildung 3: TinyHTML-Tags.

$$
\begin{array}{rl}
\mathsf{word} & = \mathsf{word\_char}^+ \\
\mathsf{word\_char} & = \mathsf{letter} \cup \mathsf{digit} \cup \mathsf{other} \\
\mathsf{letter} & = \{\texttt{a}, \ldots, \texttt{z}, \texttt{A}, \ldots, \texttt{Z}\} \\
\mathsf{digit} & = \{\texttt{0}, \ldots, \texttt{9}\} \\
\mathsf{other} & = \{\texttt{!}, \texttt{"}, \texttt{\#}, \texttt{\$}, \texttt{\%}, \texttt{\&}, \texttt{'}, \texttt{(}, \texttt{)}, \texttt{*}, \texttt{+}, \texttt{,}, \texttt{-}, \texttt{.}, \texttt{/}, \texttt{:}, \texttt{;}, \texttt{=}, \texttt{?}, \texttt{@}, \texttt{[}, \texttt{\textbackslash}, \texttt{]}, \texttt{\^{}}, \texttt{\_}, \texttt{`}, \texttt{\{}, \texttt{|}, \texttt{\}}, \texttt{\~{}}\} \\
\mathsf{whitespace} & = \{\textvisiblespace, \leftrightarrows, \hookleftarrow\}
\end{array}
$$

Abbildung 4: Lexikalik von Wörtern und Leerzeichen in TinyHTML.

Dokument und Titel Ein TinyHTML-Dokument wird von den beiden Tags `<html>` und `</html>` eingeschlossen. Innerhalb dieser Tags stehen nacheinander der Titel und Rumpf des Dokuments. Der Titel wird von `<title>`, `</title>` eingeschlossen[2], z. B.

```
<title>Example for TinyHTML</title>
```

Der Titel wird vom Viewer in der Titelleiste des Fensters angezeigt werden (vgl. Abb. 1).

Rumpf Der Rumpf, eingeschlossen in `<body>` und `</body>`, eines TinyHTML-Documents enthält den eigentlichen Text. Er besteht aus einer Sequenz von Absätzen, die untereinander im Hauptfeld des Viewers angezeigt werden.

Absätze Absätze bilden das Herz eines TinyHTML-Dokuments, da sie den eigentlichen Text enthalten. Es gibt vier Arten von Absätzen, die sich dadurch unterscheiden, wie die Textzeilen ausgerichtet werden:

Linksbündige Absätze

```
<p style="text-align:left">
  Dies ist ein
  linksbündig ausgerichteter Text.
</p>
```

Dies ist ein
linksbündig
ausgerichteter
Text.

Alle Zeilen beginnen links in derselben Spalte, der rechte Rand ist „ausgefranst“.

Rechtsbündige Absätze

```
<p style="text-align:right">
  Dies ist ein
  rechtsbündig ausgerichteter Text.
</p>
```

Dies ist ein
rechtsbündig
ausgerichteter
Text.

Alle Zeilen enden rechts in derselben Spalte, der linke Rand ist „ausgefranst“.

Zentrierte Absätze

```
<p style="text-align:center">
  Dies ist ein
  zentriert ausgerichteter Text.
</p>
```

Dies ist ein
zentriert
ausgerichteter
Text.

Alle Zeilen stehen in der Mitte der Ausgabefläche.

Absätze in Blocksatz

```
<p style="text-align:justify">
  Dies ist ein
  im Blocksatz gesetzter Text.
</p>
```

Dies ist ein
im Blocksatz
gesetzter Text.

Alle Zeilen (bis auf die letzte) sind von gleicher Breite. Dies wird erreicht, in dem der Wortzwischenraum gleichmäßig gestreckt wird.

[2]Wir nehmen hier eine kleine Vereinfachung vor: in HTML muß der Titel eigentlich im Header stehen, der durch `<head>`, `</head>` begrenzt wird. Gängige Browser wie Mozilla Firefox und Internet Explorer sind hier aber liberal und zeigen den Titel auch an, wenn der Header wie bei uns ausgelassen wird.

Zeilenumbrüche Zu beachten ist, dass die Zeilenumbrüche im TinyHTML-Quelltext nichts mit den Zeilenumbrüchen des angezeigten Textes zu tun haben. Diese Zeilenumbrüche werden vom Viewer abhängig von der Größe des Fensters und der Schrift berechnet. Ein und derselbe Quelltext kann also je nach Fenstergröße unterschiedlich angezeigt werden:

```
<p style="text-align:left">
  Dies ist ein linksbündig
  ausgerichteter Text.
</p>
```

Dies ist ein linksbündig ausgerichteter Text.

Dies ist ein linksbündig ausgerichteter Text.

Whitespace Whitespace, d. h. Leerzeichen, Zeilenumbrüche und Tabulator-Zeichen, spielen für die Anzeige des Texts keine Rolle. Das bedeutet, dass die folgenden beiden TinyHTML-Dokumente zu identischer Ausgabe führen:

```
<html><title>Hallo Welt</title>
<body>
<p style="text-align:center">Hallo
Welt!</p></body></html>
```

```
<html>
  <title>Hallo Welt</title>
  <body>
    <p style="text-align:center">
      Hallo Welt!
    </p>
  </body>
</html>
```

Ebenso spielt es keine Rolle, ob Wörter in der Eingabe durch ein oder mehrere Leerzeichen getrennt werden. In der Ausgabe erscheint grundsätzlich nur ein Leerzeichen oder ggf. ein Zeilenumbruch. Die beiden Absätze

```
<p style="text-align:left">
Ein     Text    mit  sehr  vielen
Leereichen.
</p>
```

```
<p style="text-align:left">
Ein Text mit sehr vielen
Leereichen.
</p>
```

führen also zu identischer Ausgabe.

3 Komponenten des Viewers

Wir werden nun die einzelnen Komponenten des Viewers und ihre Verbindungen entwerfen. Im nächsten Abschnitt werden diese Strukturen dann auf die einzelnen Meilensteine verteilt.

Wir nutzen den Signatur-Mechanismus von Opal, um die einzelnen Komponenten und ihre Beziehungen zu beschreiben.

Bevor wir ins Detail gehen, skizzieren wir grob die einzelnen Schritte, die der Viewer ausführt, um ein Dokument anzuzeigen:

1. Einlesen der Eingabedatei
2. Lexikalische Analyse des Eingabetextes aus der Datei
3. Syntaktische Analyse der Token-Sequenz und Erzeugen des abstrakten Syntaxbaums

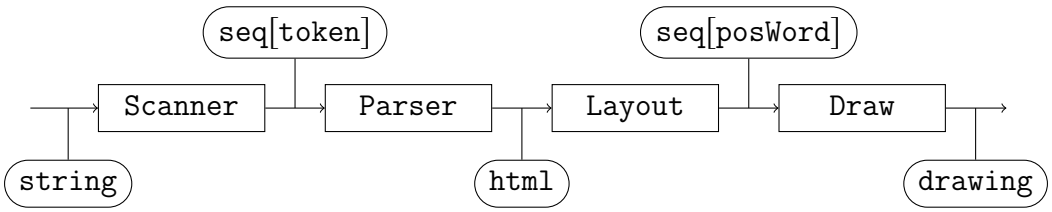

Abbildung 5: Datenfluß im TinyHTML-Viewer. Abgerundete Boxen bezeichnen Datentypen, eckige Boxen Strukturen.

4. Umwandlung eines jeden Absatzes in eine Sequenz von Wörtern (Denotations) mit zugehöriger Ausgabeposition — unter Berücksichtigung der Formatierung — auf der Anzeigefläche
5. Anzeigen eines Fensters
6. Ausgabe der Wörter des Dokuments an den berechneten Positionen
7. Warten, dass der Benutzer den Viewer beendet.

Im Folgenden ordnen wir nun den einzelnen Stationen ein oder mehrere OPAL-Strukturen zu und entwickeln ihre Schnittstellen. Die Strukturen und ihre Verbindung sind in Abb. 5 gezeigt.

3.1 Lexikalische Analyse: `Scanner` und `Token`

Die Struktur `Scanner` stellt eine Funktion zur Verfügung, die einen Eingabetext in eine Sequenz von Token umwandelt:[3]

```
SIGNATURE Scanner

  IMPORT Token ONLY token

  FUN scan : string → seq[token]
```

Die Struktur `Token` definiert den Datentyp der Tokens, also der OPAL-Repräsentation der Lexeme (siehe Vorlesung):

```
SIGNATURE Token

  TYPE token == ...
```

3.2 Syntaktische Analyse: `Parser` und `Syntax`

Die Struktur `Parser` implementiert den TinyHTML-Parser, der eine Sequenz von Tokens in einen abstrakten Syntaxbaum transformiert. Der Parser liefert den abstrakten Syntaxbaum sowie die verbleibende Eingabe. Im Falle einer korrekten Eingabedatei ist das zweite Ergebnis immer immer die leere Sequenz (vgl. Vorlesung).

[3]Wir beschränken uns hier und in weiteren Signaturen auf die essentiellen Importe.

```
SIGNATURE Parser

  IMPORT Syntax COMPLETELY
         Token  ONLY token

  FUN parse : seq[token] → html × seq[token]
```

Die Datentypen für den abstrakten Syntaxbaum sind in der von `Parser` importierten Struktur `Syntax` definiert:

```
SIGNATURE Syntax

  TYPE html      == html(title : title, body : body)
                    fail
  TYPE title     == title(content : seq[string])
                    fail
  TYPE body      == body(content : seq[par])
                    fail
  TYPE par       == par(align : alignment, content : seq[string])
                    fail
  TYPE alignment == left right center justify
```

3.3 Berechnung des Layouts: `Layout`

Die Funktion `layout` aus der Struktur `Layout` berechnet nun für jedes Wort einer Sequenz von Absätzen die Position, an der es auf der Anzeigefläche platziert wird. Dazu wird der Datentyp `posWord` eingeführt, der ein Wort zusammen mit seiner Position und Breite speichert.

```
SIGNATURE Layout

  IMPORT Syntax ONLY par

  FUN layout : metrics × seq[par] → seq[posWord]

  TYPE posWord == posWord(text : string, pos : point, width : real)
```

Die Funktion `layout` benötigt zusätzlich einige Maße der Schriftart, z. B. Höhe der Wörter, um das Layout berechnen zu können. Diese sind im Argument des Typs `metrics` zu finden. Die bereitgestellte Struktur `WinFontMetrics` stellt Funktionen bereit, um für eine bestimmte Schriftart diese Angaben zu bekommen.

3.4 Zeichnen der Wörter: `Draw`

Die Funktion `draw` der Struktur `Draw` nimmt schliesslich eine Sequenz positionierter Wörter und erstellt ein `drawing`, das mit Hilfe eines `canvasEditor`s aus OPAL WINDOWS gezeichnet werden kann.

`drawing`s erlauben es bspw. einfache gemoetrische Figuren wie Kreise und Linien in unterschiedlichen Stilen zu zeichnen aber auch die positionierte Ausgabe von Text.

```
SIGNATURE Draw

  IMPORT Layout ONLY posWord

  FUN draw : seq[posWord] → drawing
```

3.5 Einstellungen: `Settings`

In der Struktur `Settings` sind Konfigurationseinstellungen wie Größe des Fensters, genutzte Schriftart usw. abgelegt.

3.6 Hauptstruktur: `Viewer`

Die Struktur `Viewer` beinhaltet den Eintrittspunkt in das Programm. Hier werden die einzelnen Schritte nacheinander ausgeführt und mit den jeweiligen Eingaben versorgt, wie in Abb. 5 gezeigt.

4 Meilensteine

Wir beschreiben nun die einzelnen Meilensteine. In Abschnitt 3 haben wir skizziert, welche Schritte der Viewer durchführen muss, um ein TinyHTML-Dokument anzuzeigen. Es liegt nahe, die Meilensteine anhand dieser Aufgaben anzuordnen. Damit ergeben sich die folgende

Vorläufige Einteilung

1	Aufgabe:	Syntaxanalyse
	Strukturen:	`Token`, `Scanner`, `Syntax`, `Parser`

2	Aufgabe:	Layoutberechnung
	Strukturen:	`Layout`

3	Aufgabe:	Anzeigen des Texts
	Strukturen:	`Draw`, `Viewer`

Allerdings hat diese Abfolge einen gravierenden Nachteil: Nach dem ersten Meilenstein haben wir einen abstrakten Syntaxbaum des Dokuments, den wir aber noch nicht anzeigen können. Ebenso haben wir nach dem zweiten Meilenstein eine Sequenz positionierter Wörter, die wir noch nicht anzeigen können. Dies wird erst im dritten Meilenstein möglich. Dadurch ist es viel schwerer, Fehler in den ersten beiden beiden Meilensteinen zu finden, bevor wir den dritten zur Verfügung haben.

Aus diesem Grund ist es besser, die o. g. Reihenfolge umzudrehen und wir erhalten die

Endgültige Einteilung

1	Aufgabe:	Anzeigen des Texts
	Strukturen:	`Draw`, `Viewer`

2	Aufgabe:	Layoutberechnung
	Strukturen:	`Layout`

3	Aufgabe:	Syntaxanalyse
	Strukturen:	`Token`, `Scanner`, `Syntax`, `Parser`

Nun haben wir zwar im ersten Meilenstein noch keine Sequenz positionierter Wörter. Wir können aber eine oder mehrere handgeschriebene Sequenzen einsetzen, um die Anzeige zu testen.

Analog fehlt der Layoutberechnung eine Sequenz von Absätzen, so dass wir auch hier händisch geschriebene Sequenzen zum Testen einsetzen.

Dadurch, dass wir die Anzeige als erstes implementieren, können wie die nachfolgenden Meilensteine viel einfacher testen, da wir bereits in der Lage sind, Texte auf dem Bildschirm anzuzeigen

Im Folgenden werden nun die einzelnen Meilensteine beschrieben. Es werden jeweils der Abgabezeitpunkt, die zu hauptsächlich zu bearbeitenden Strukturen, evtl. Vorgaben, die genaue Aufgabenstellung sowie hilfreiche Strukturen aus der Bibliotheca Opalica angegeben.

4.1 Meilenstein 1: Anzeigen des Texts

Abgabefrist: 12.01.2014, 23:55 Uhr

Strukturen: `Draw`, `Settings`, `Viewer`

Vorgaben: –

Aufgabenstellung:

Implementierung der Struktur `Viewer`
Implementiert die Struktur `Viewer`, die mit Hilfe von OPAL WINDOWS ein Fenster auf dem Bildschirm anzeigt. Dieses Fenster hat wie in Abb. 1 einen Button zum Beenden des Programms sowie eine Zeichenfläche (`WinCanvas`) zum Anzeigen des Texts.
Die Struktur `Viewer` enthält das Hauptkommando des Viewers des Typs `com`[`void`]. Wird dieses Kommando ausgeführt, soll das Fenster angezeigt werden. Durch Betätigung des „Beenden“-Buttons soll das Programm beendet werden.

Implementierung der Struktur `Draw`
Implementiert die Struktur `Draw`, mittels der eine Sequenz positionierter Wörter (aus `Layout`) in ein `drawing` umgewandelt werden kann.
Ergänzt Eure Struktur `Viewer` derart, dass eine Testsequenz (wie in Abschnitt 4 beschrieben) unter Nutzung der Struktur `Draw` auf der Zeichenfläche angezeigt wird.

Implementierung der Struktur `Settings`
Nutzt die Struktur `Settings`, um Einstellungen, wie etwa die Fenstergröße oder Farben, zu hinterlegen.

Compilation
Sorgt dafür, dass Euer Programm mittels des Compilers `ocs` übersetzt werden kann. Das schliesst insb. das Verfassen einer `SysDefs`-Datei ein.

Hinweis
Im Verzeichnis `examples/Graphics` der Opal-Installation befinden sich drei Beispiele, die OPAL WINDOWS benutzen und als gute Hilfestellung dienen können.

Bibliotheca Opalica	Ein-/Ausgabe	`Com`, `ComCompose`
	OPAL WINDOWS	`WinButton`, `WinCanvas`, `WinCanvasEditor`, `WinConfig`, `WinEmitter`, `WinView`, `WinWindow`

4.2 Meilenstein 2: Layoutberechnung

Abgabefrist: 26.01.2014, 23:55 Uhr

Strukturen

`Layout`, `Syntax`

Vorgaben

`WinFontMetrics`

Aufgabenstellung

Implementierung der Struktur `Layout`
Programmiert die Struktur `Layout`, mit deren Hilfe sich eine Sequenz von Absätzen (aus `Syntax`) in eine Sequenz positionierter Wörter umwandeln läßt. Ihr müßt dabei die Ausrichtungen der einzelnen Absatztypen (siehe Abschnitt 2.2) sowie die Breite des Ausgabefensters berücksichtigen. Mit der bereitgestellten Struktur `WinFontMetrics` könnt Ihr die Grösse von Wörtern auf dem Bildschirm sowie den Zeilenabstand für eine bestimmte Schriftart ermitteln.
Hinweis: Ihr dürft beim Layout ignorieren, dass die vertikale Ausdehnung Eures Fensters mglw. zu klein ist, um den gesamten positionierten Text anzuzeigen. Ihr müßt also kein „Scrolling“ implementieren, sondern überschüssiger Text darf weggelassen werden.

Integration in `Viewer`
Integriert die Struktur `Layout` in `Viewer`, so dass jetzt statt einer Testsequenz positionierter Wörter eine Testsequenz aus Absätzen verwendet wird, die Ihr mittels der `Layout`-Struktur positioniert.

Bibliotheca Opalica	Sequenzen	`Seq`, `SeqMap`, `SeqMapEnv`, `SeqOfSeq`, `SeqReduce`, `SeqZip`
	Paare	`Pair`

4.3 Meilenstein 3: Syntaxanalyse

Abgabefrist: 09.02.2014, 23:55 Uhr

Strukturen

`Token`, `Scanner`, `Syntax`, `Parser`

Vorgaben

–

Aufgabenstellung

Implementierung des Scanners: `Token` *und* `Scanner`
Entwerft einen geeigneten Datentyp für die Tokens der TinyHTML-Sprache und implementiert einen Scanner, der eine `string` in eine Sequenz dieser Tokens umwandelt.

Implementierung des Parsers: `Syntax` *und* `Parser`
Implementiert einen Recursive-Descent-Parser, der aus der Tokensequenz einen abstrakten Syntaxbaum erstellt oder einen Fehler zurückgibt.

Integration in `Viewer`
Zum Schluss muss `Viewer` so erweitert werden, dass die Eingabedatei gelesen wird. Der Name der Datei soll dabei auf der Kommandozeile angegeben werden, bspw.

```
$ ./viewer example.html
```

Als weitere Schritte müssen der Scanner und der Parser in `Viewer` aufgerufen werden, so dass schliesslich die Sequenz von Absätzen aus aus der Eingabedetei generiert wird.
Beachtet dabei, dass bei der Syntaxanalyse Fehler auftreten können, bei denen das Programm mit einer Fehlermeldung abgebrochen werden muss.

Hinweis: Vergesst nicht, dass das Fenster noch den Titel des TinyHTML-Dokumentes anzeigen soll.

Bibliotheca Opalica		
	Kommandozeile	`ProcessArgs`
	(Datei-)Ein-/Ausgabe	`BasicIO`, `File`
	Scanner	`String`, `Char`