Gruppe
Die Affenbande

universität ulm

Das 'sopra-listings'-Paket

Dokumentation für das 'sopra-listings'-Paket | Version v1.0.23

1. Oktober 2020

Florian Sihler (florian.sihler@uni-ulm.de)

1 Allgemeines

1.1 Warum, wieso, weshalb?

Dieses LaTeX 2_ε-Paket wurde im Rahmen des Sopras im Wintersemester 2019 und Sommersemester 2020 verfasst und dient als Grundlage für die das Highlight von Codes des *Teams 20*. Diese Dokumentation wurde zusammen mit der `sopra-base.cls` sowie dem Paket `sopra-documentation.sty` kreiert.

Zum Visualisieren der einzelnen Code-Ausschnitte wird das Paket selbst verwendet, welches ebenfalls in hier eingebettet sein sollte:  . Dieses Paket benötigt weder `shell-escape` noch `minted` um zu funktionieren. Es baut auf `listings` auf!

1.2 Abhängigkeiten

Dieses Paket bindet die folgenden Paketen mit ein:

- `fontenc`(`T1, encoding`)
- `inputenc`(`utf8, encoding`)
- `needspace`(`guardspace`)
- `etoolbox`
- `xcolor`
- `pgfkeys`
- `listingsutf8`(≪try≫)
- `listings`(≪fallback≫)
- `accsupp`(≪try≫)

All diese Pakete sollten Teil der gängigen LaTeX-Distribution sein. Hinweis: „≪try≫“ notiert hierbei, dass das Paket nicht vorhanden sein muss, aber eventuell zusätzliche Möglichkeiten bietet. So gestattet es `accsupp` zum Beispiel, die Zeilennummern „unkopierbar“ zu machen und so besser mit den Listings zu arbeiten und `listings` wird nur dann geladen, wenn `listingsutf8` nicht gefunden wird.

Aktuell sind noch einige Dinge wie Optionen und weitere Umgebungen geplant.

1.3 Die Installation

Das Paket wird nicht als `.dtx` ausgeliefert, weswegen sich die folgenden Möglichkeiten ergeben:

- Das Paket kann in dasselbe Verzeichnis wie das Dokument gesetzt werden. In diesem Fall lautet die Einbindungsanweisung:

  ```
  \usepackage{sopra-listings}
  ```

- Das Paket kann in ein Unterverzeichnis/in ein mit dem Dokument ausgeliefertes Verzeichnis gelegt werden. In diesem Fall erfolgt die Angabe durch den (relativen-) Pfad:

  ```
  \usepackage{./Mein/Pfad/zu/sopra-listings}
  ```

- Man kann das Paket (mittels eines Symlinks oder ähnlichem) in einen eigenen *texmf*-Baum ablegen. So kann zum Beispiel auf Linux unter der Verwendung von texlive das Paket hier abgelegt werden: `~/texmf/tex/latex/`. Das Verzeichnis kann erstellt und anschließend mittels `texhash ~/texmf` aktualisiert werden. Nun kann das Paket wie jede andere installierte Paket verwendet werden:

```
\usepackage{sopra-listings}
```

Wichtiger Hinweis: Dieses Paket kommt mit Sprachkonfigurationen, die sich im Unterordner „Languages“ befinden. Die Sprachen stehen nur dann zur Verfügung, wenn sie mit dem Paket platziert werden (nach einer der eben genannten Möglichkeiten) oder sie sich im gleichen Verzeichnis wie die LaTeX 2ε-Datei befinden. Sollte dies nicht gewünscht oder möglich sein, so *muss* \solLanguageSearchPath → p. 7 angepasst werden! Es steht jedem Frei, das Laden der Sprachen mittels noloadlangs zu deaktivieren und seine eigenen Sprachdefinitionen oder Sprachen zu verwenden. Damit dieses Paket allerdings Sinn macht, sollte das \RegisterLanguage → p. 7-Makro verwendet werden.

1.4 Weitere Besonderheiten

Version v1.0.0:

Es wird \ttfamily modifiziert, so dass die „Typewriter“-Schrift auch hier identisch zu der allgemein verwendeten ist.

Version v1.0.10:

Der Befehl \lstfs → p. 9 wurde hinzugefügt um die Schriftgrößen von Listings anzupassen.

Version v1.0.12:

Die Paketoption cpalette wurde samt Gegenoption hinzugefügt.

Version v1.0.13:

Die Befehle \lstcolorlet → p. 9 und \lstcolordef → p. 9 wurden zur einfachen Farbmanipulation hinzugefügt.

Version v1.0.14:

Abstände bei Literates korrigiert. Sollten nun auch beim Ersetzen von Ziffern in inline-Listings die korrekte Anzahl an Leerfeldern behalten

Version v1.0.15:

Mit guardspace (noguardspace) → p. 4 wurde eine (vertikale) Abstands-sicherungs-option hinzugefügt.

Version v1.0.17:

Mit autogobble können nun anführende Leerfelder entfernt werden.

Version v1.0.18:

Sichert nun auch die letzte Zeile eines Listings im Falle von guardspace (noguardspace) → p. 4.

Version v1.0.19:

Kursive Schrift für Typewriter Schriftart wird nun sicherer angewendet. Ein paar ungenutzte Komfort-Funktionen wurden entfernt.

Version v1.0.20:

Die Funktion autogobble wurde wieder entfernt, ebenso wurde die Paketstruktur intern überarbeitet.

Version v1.0.21:

Es wurden einige Code-Interne Restrukturierungen durchgeführt.

Version v1.0.22:

Unterstützung für `nofakeminted (fakeminted)` → p. 4 und `noextendednums (extendednums)` → p. 4.

Version v1.0.23:

Unterstützung für `\solblacklistlinenumbers` → p. 10, `\solblacklistisghost` → p. 10 und `\solblacklistispresent` → p. 10.

1.5 Akzeptierte Parameter

Das Paket akzeptiert, so wie die meisten, Argumente. Bei Argumenten mit einer „Counter"-Option wird das jeweils standardmäßig aktive zuerst und das andere in Klammern geschrieben. So wird implizit:

```
\usepackage[noencoding,hlnumbers,loadlangs,guardspace]{sopra-listings}
```

aufgerufen. Während mit:

```
\usepackage[noloadlangs]{sopra-listings}
```

das automatische Laden gewisser Sprachen verhindert wird.

▷ `noencoding (encoding)`

`encoding` lädt die für Umlaute nötigen Pakete. Diese Option kann dann benutzt werden wenn man sie nicht extra laden, oder einfach sicher gehen möchte.

▷ `hlnumbers (nohlnumbers)`

Das Paket versucht automatisch Zahlen zu markieren und zwar nur da, wo es sich auch um eine Zahl handelt. Dieses Verfahren funktioniert gut, kann aber - sofern als störend empfunden, oder es sich doch als unartig herausstellen sollte - mittels `nohlnumbers` deaktiviert werden.

▷ `defaultmode (print)`

Diese Funktionen arbeiten analog zu den gleichnamigen Vertretern in `sopra-base.cls`. Wird `print` gewählt, so wird versucht möglichst Tinte einzusparen (kein Hintegrund bei Codes, ...) und weiter werden gewisse Schlüsselbegriffe fett/kursiv gedruckt um auch im einfarbigen Ausdruck eine (möglichst gute) Unterscheidbarkeit zu erhalten.

▷ `loadlangs (noloadlangs)`

Lädt mittels `\solLoadLanguage` → p. 7 die Sprache `java` (siehe Abschnitt A.1 (Vordefinierte Sprachen) für mehr Informationen). Dies kann durch `noloadlangs` deaktiviert werden. Hinweis: Die Sprachen werden nur geladen, wenn die Konfigurationsdateien gefunden werden. Siehe hierzu `\solLoadLanguage` → p. 7.

▷ `nocpalette (cpalette)`

Bietet Unterstützung für https://github.com/EagleoutIce/color-palettes. In diesem Fall wird das code-highlighting automatisch an die aktuelle Palette angepasst und auch mit dem Ändern der Palette von da an aktualisiert.

▷ `noupshape (upshape)`

Fügt `\upshape` zum Stil hinzu. Dies sorgt dafür, dass sich Listings nichtmehr an kursiv gesetzte Schrift anpassen.

▷ `guardspace` (`noguardspace`)

Mit `guardspace` werden dem Listing jeweils auch zugesichert, dass es die nächste Zeile setzen darf. Dies kann dafür sorgen, dass ein Seitenumbruch erzeugt wird (wenn weniger als eine Zeile des Listings noch auf die Seite passt). Der Zweck dieser Option ist es, bei einer Hintergrundfarbe anfängliche Streifen zu vermeiden. Siehe `\solguard` → p. 10.

▷ `nonuminpar` (`numinpar`)

Kurzform für `\solSetLeftMargin` → p. 8 mit dem Wert „2em“, kann also dazu verwendet werden um die Listings automatisch einzurücken und die Nummern so nicht über den Dokumentenrand hinausragen zu lassen.

▷ `defaultfont` (`nodefaultfont`)

Wenn die Option gesetzt ist, wird die Typewriter-Schriftart auf „AnonymousPro“ geändert, sonst nicht.

▷ `nofakeminted` (`fakeminted`)

Ist die Option gesetzt, so wird eine `env@minted` Umgebung erzeugt, welche es erlaubt dieses Paket wie `env@minted` zu verwenden. In diesem Fall heißt es:

```
\begin{minted}{java}
public static void main(String[] args) { }
\end{minted}
```

Mit `\solmintedmap` → p. 11 kann ein Sprach-Mapping erzeugt werden. So wird beispielsweise automatisch `java` auf die interne Sprache `lJava` abgebildet. Durch `\solsetmintedstyle` → p. 11 kann gewählt werden, welcher der internen Stile für das Listing verwendet werden soll. Zur Verfügung stehen `default`, `nonumber`, `plain` und `plain number`.

▷ `noextendednums` (`extendednums`)

Ist die Option gesetzt, so werden auch Zahlen gehighlighted, die intern Unterstriche verwenden oder die klassischen `0x`-, `0b`- oder `0o`-Präfixe verwenden.

▷ `inlinesize` (`noinlinesize`)

Wenn die Option gesetzt ist, wird in den inline-Befehlen wie `\bjava` die Schriftgröße nicht angepasst (siehe hierfür auch `\lstfs` → p. 9 mit Sternchen). Im Dokument kann diese Konfiguration jederzeit mit `\soladaptinline` aktiviert und mit `\sollockinline` deaktiviert werden.

1.6 Farben

Dieses Paket definiert einige Farben mittels `\colorlet`. Wird `sopra-base` als geladen erkannt, wird versucht passende Farben zu setzen. *Alle folgenden Farben tragen das Präfix `sol@colors@`*: `border`, `background`, `lst@keywordA`, `lst@keywordB`, `lst@keywordC`, `lst@numbers`, `lst@literals`, `lst@comments`, `lst@highlight`, `lst@linenumbers` und zuletzt `lst@command`, wobei die Bedeutung der einzelnen Farben aufgrund ihres Namens hervorgehen sollte (jeder Sprache steht es frei die gleich zu besprechenden Stile, die auf diesen Farben aufbauen, (beliebig) zu verwenden). Beispielsweise kann man die Farbe von Kommentaren wie folgt ändern (`\lstcolorlet` → p. 9):

```
\lstcolorlet{comments}{orange}
```

Wir erhalten(exemplarisch in Java, zuerst standard, danach mit geänderter Farbe):

```
// Vor der Neuzuweisung der Farbe
public static void main(...){ /* ... */ }
```

```
// Nach Neuzuweisung der Farbe
public static void main(...){ /* ... */ }
```

Es ist möglich jede Farbe (nach xcolor) zu verwenden (siehe auch \lstcolordef → p. 9).

1.7 Stile

Zur Hervorhebung der einzelnen Segmente im Listing werden den Farben entlehnte Stile definiert, die ebenfalls frei verändert werden können. Entwicklernotiz, die Stile sollten bei einer kompletten Neudefintion mithilfe des Befehls \sol@list@define@styles definiert werden. Sonst ist das Präfix, das im Folgenden verwendet werden soll in \sol@lst@style@prefix abgespeichert.

Alle im folgenden aufgeführten Befehle tragen (sofern nicht modifiziert) das Präfix sol@styles@lst@ und können (ohne dieses Präfix) mittels \solGet → p. 9 beziehungsweise \solGetStyle → p. 8 abgefragt werden. Einzelne Sprachen können weitere Befehle und Stile fordern und hinzufügen, davon sollte aber nur im Notfall gebraucht gemacht werden (so fügt die Sprache Latex den Stil command hinzu. Für mehr Informationen, siehe: Abschnitt A.1 (Vordefinierte Sprachen))

Es gibt die folgenden Stile: keywordA, keywordB, keywordC, keywordD, numbers, linenumbers, literals, comments, highlight, command und basic, wobei letzterer jedem Stil vorangestellt wird. Die Neudefinition eines Stils sollte nicht leichtfertig getätigt werden und benötigt deswegen etwas mehr Aufwand. Hier ein Beispiel:

```
{\makeatletter
    \sol@list@define@styles{%
        {comments: \color{sol@colors@lst@comments}\scshape}%
    }
}
```

Wir erhalten(exemplarisch in Java):

```
// Vor der Neuzuweisung des Stils
public static void main(...){ /* ... */ }
```

```
// Nach Neuzuweisung des Stils
public static void main(...){ /* ... */ }
```

Mehrere Stile können durch Kommas voneinander abgetrennt in einem Zug definiert werden und überleben Gruppen.

2 Befehle- und Umgebungen

Es gilt zu beachten, dass das Präfix env@ nur auf die Natur einer Umgebung hinweist und nicht zum eigentlichen Bezeichner zuzuordnen ist!

2.1 Die Umgebungen zum Setzen

Diese Umgebungen werden für jede geladene Sprache erstellt. Jede in diesem Paket enthaltene Sprache folgt der Regel, dass sie mit einem „l“ (für language) beginnt und dann ein Großbuchstabe folgt. Möchte man also java aus irgendeinem Grunde manuel in env@lstlisting oder so verwenden, so heißt die Sprache hier lJava.

▷ env@‹Sprache›*[Stildefinitionen]*

Setzt den eingeschlossenen Code in einer durch \solLoadLanguage → p. 7 ‹Sprache›. Also, wichtig: env@‹Sprache› selbst, gibt es nicht.. Beispiel:

```
\begin{java}
// Ein Kommentar!!
public static void main(...){ /* ... */ }
\end{java}
```

Ergibt (Meta-Kommentar: Die Anzeige des LaTeX 2ε-Codes wurde mit `env@latex` vollzogen):

```
// Ein Kommentar!!
public static void main(...){ /* ... */ }
```

Bei `Stildefinitionen` handelt es sich übrigens um Definitionen für `listings`.

▷ `env@‹Sprache›*[Stildefinitionen]`

Analog zu `env@‹Sprache›` → p. 5, allerdings ohne Zeilennummern.

▷ `env@plain‹Sprache›[Stildefinitionen]`

Analog zu `env@‹Sprache›` → p. 5, allerdings ohne Zeilennummern und Hintegrund.

▷ `\c‹Sprache›[Stildefinitionen]{Code}`

Setzt den Code in einer Zeile in der jeweiligen Sprache. Wichtige Bemerkung: Da LaTeX 2ε nicht davon abgehalten werden kann die Zeichen zumindest einfach zu expandieren, müssen Zeichen wie ein Anführungszeichen oder geschwungene Klammern mittels eines Backslash escaped werden. Dies kann weiter dafür sorgen, dass in diesem Fall Leerfelder verloren gehen. Ein solches Leerfeld kann durch „`:ws:`“, ein Literate, forciert werden.
Beispiel:

```
Dies ist ein Test für \cjava{public static void main} – hat er geklappt?
```

Ergibt:
Dies ist ein Test für `public static void main` - hat er geklappt?

▷ `\b‹Sprache›[Stildefinitionen]{Code}`

Analog zu `\c‹Sprache›` → p. 6, allerdings wird kein Hintergrund und kein Rahmen gesetzt, was dafür sorgt, dass auch Zeilenumbrüche kein Problem sind.

▷ `\i‹Sprache›[Stildefinitionen]{Dateipfad}`

Bindet die Datei in `Dateipfad` in das Dokument ein, wobei `env@‹Sprache›` → p. 5 zum Highlighting verwendet wird.

2.2 Sprachverwaltung

▷ `\solStyles`

Zeigt die Formatierung und Farbe der verschiedenen Stile an und ist wohl nur zu Testzwecken von nutzen. So ergibt `\solStyles`: `basic` `keywordA` `keywordB` `keywordC` *`keywordD`* `numbers` `linenumbers` `literals` `comments` *`highlight`*

▷ \solLanguageSearchPath

Dieser Befehl enthält alle Pfade an denen \solLoadLanguage → p. 7 nach sprachen suchen soll. Dies ist überflüssig, wenn sich die Sprache im texmf-Baum des Systems befindet. Die Pfade können durch \renewcommand aktualisiert werden. Beispiel:

```
\renewcommand{\solLanguageSearchPath}{{OrdnerA/}{OrdnerB/UnterordnerA/}{/home/}}
```

Im lokalen Verzeichnis in dem pdflatex aufgerufen wird, wird immer gesucht! Sollte es mehrere Dateien mit gleichem Namen geben, so hat die Datei im lokalen Verzeichnis die höchste Priorität und anschließend werden die Ordner in der Reihenfolge durchsucht, bis die Datei das erste mal gefunden wurde.

▷ \solLoadLanguage{LanguageName}

Sucht und Lädt Sprachen in solLanguageSearchPath. Es können mehrere Sprachen mittels Kommaseparierung angegeben werden. Der Befehl kann mehrfach aufgerufen werden, erlaubt aber nicht, die selbe Sprache mehrfach zu laden. Die Sprache muss in einer Datei mit dem Namensschema language_‹LanguageName›.cfg definiert sein. Also zum Beispiel language_java.cfg. Wie die Definition einer Sprache auszusehen hat, erklärt \RegisterLanguage → p. 7. Die Sprache lVoid mit void wird immer geladen und kann als Verbatim-Umgebung verwendet werden.

▷ \RegisterLanguage*[Aliasse]*{Sprache}{LanguageName}

Registriert eine Sprache mit dem Namen LanguageName als Sprache. Dieser Aufruf muss von jeder Konfiguration selbst durchgeführt werden.

Im folgenden Beispiel gilt es die Sprache Rubberduck zu definieren und zu laden. In die Konfigurationsdatei schreiben wir:

```
\lstdefinelanguage{lRubberduck}{
    comment=[l]{\#},
    morekeywords = {Quack, new},
    morekeywords = [2]{Duck}
}
\RegisterLanguage{rubberduck}{lRubberduck}
```

Also eine Sprache, die Raute für Kommentare verwendet, die beiden Schlüsselbegriffe Quack und new besitzt sowie noch ein „keywordB“ (so Formal ist die Sprache jetzt ja auch nicht :)): Duck Im Dokument laden wir die Sprache mit:

```
\solLoadLanguage{rubberduck}
```

Nun können wir die Umgebungen wie env@‹Sprache› → p. 5 auch mit rubberduck verwenden:

```
\begin{rubberduck}
Duck jens = new Duck(); # Eine neue Ente
Quack jens ::{
    Quack Quack, Quack Quack
    Quack Quack. # Entisch, es ist so simpel
}
\end{rubberduck}
```

Ergibt:

```
Duck jens = new Duck(); # Eine neue Ente
Quack jens ::{
    Quack Quack, Quack Quack
    Quack Quack. # Entisch, es ist so simpel
```

```
}
```

Die Befehle wie \c<Sprache> →p. 6, also \crubberduck funktionieren natürlich auch: `Duck jens`.

Ist fakeminted →p. 4 nicht gesetzt, so haben die Aliasse aktuell keine Bedeutung. Werden sonst keine Aliasse übergeben, so wird automatisch ein mapping (\solmintedmap →p. 11) von Sprache zu LanguageName erstellt. Andernfalls, wird für jedes (Komma-separierte) Alias, ein Mapping zu LanguageName kreiert, sodass es mit env@minted →p. 11 verwendbar ist.

2.3 Allgemeine Befehle

▷ \thesolversion

Liefert die aktuelle Version des Pakets. So ergibt: \thesolversion: v1.0.23
Hinweis: über \value{solversion} lässt sich die Version als 4-stellige Nummer erhalten: 1023.

▷ \solSetLeftMargin{Länge}

Setzt den Abstand der linken Seite des Listings und kann so dafür sorgen, dass zum Beispiel auch die Zeilennummern nicht über den Dokumentrand hinausragen. Beispiel:

```
\solSetLeftMargin{15pt}
\begin{java}
// Ein Kommentar!!
public static void main(...){ /* ... */ }
\end{java}
```

Ergibt:

```
// Ein Kommentar!!
public static void main(...){ /* ... */ }
```

▷ \solSetRightMargin{Länge}

Setzt den Abstand der rechten Seite des Listings, sonst analog zu \solSetLeftMargin →p. 8.

▷ \solSetNumSep{Länge}

Setzt den Abstand der Zeilennummern zum Code.

▷ \solSetFrameRule{Länge}

Setzt die Dicke der Umrandung (Standard ist hier `0.75pt`, also nicht übertreiben :P).

▷ \T{Text}

Setzt den Text in Schreibmaschinenschrift.

▷ \solGetStyle{StilName}

Expandiert zum Stil von StilName wie ihn das Paket auch selbst verwendet.

▷ `\solGet{StilName}{Text}`

Setzt `Text` im Stil von `StilName` wie ihn das Paket auch selbst verwendet. So zum Beispiel: `\solGet{comments}{Hallo Welt}` ergibt: Hallo Welt.

▷ `\solNumFs{size}`

Ändert die Schriftgröße der Zeilennummern separat.

▷ `\lstfs⟨*⟩[linespread]{size}`

Passt die Schriftgröße des Codes wie auch die der Zeilennummern an. Bei letzteren wird versucht eine geeignete Größe zu finden. Die Variante mit Stern hat *keine* Angabe der `size`, in diesem Fall wird automatisch die aktuelle Schriftgröße gesetzt.

```
\begin{java}
public static int add(int a, int b){ return a + b; }
\end{java}
{\lstfs{5}
\begin{java}
public static int add(int a, int b){ return a + b; }
\end{java}
}
\begin{java}
public static int add(int a, int b){ return a + b; }
\end{java}
{\tiny
\begin{java}
public static int add(int a, int b){ return a + b; }
\end{java}
}
```

Ergibt:

```
public static int add(int a, int b){ return a + b; }
```

```
public static int add(int a, int b){ return a + b; }
```

```
public static int add(int a, int b){ return a + b; }
```

```
public static int add(int a, int b){ return a + b; }
```

▷ `\lstcolorlet{lstcolor}{normalcolor}`

Weißt eine Farbe mittels `\colorlet` zu. Hierbei darf *lstcolor* nur einen der folgenden Werte annehmen: *keywordA*, *keywordB*, *keywordC*, *numbers*, *literals*, *comments*, *highlight* oder *command*.

▷ `\lstcolordef[mode=RGB]{lstcolor}{colorcode}`

Weißt eine Farbe mittels `\definecolor` zu. Hierbei darf *lstcolor* nur einen der folgenden Werte annehmen: *keywordA*, *keywordB*, *keywordC*, *numbers*, *literals*, *comments*, *highlight* oder *command*. Das erste Argument gibt den Modus nach *xcolor* an.

▷ `\solguard{Zeilen}`

Mit der Paketoption `guardspace` (`noguardspace`) → p. 4 kann so angegeben werden, wie viele Zeilen geschützt werden müssen. Der Standard ist: „.8“. Hinweis: Dies betrifft nur die Zeilen für den Beginn. Die letzte Zeile wird aktuell immer mit `\nobreak` gesichert.

▷ `\solblacklistlinenumbers⟨*⟩{Zeilennummern}`

Eine kommaseparierte Liste an Zeilennummern, welche nicht angezeigt werden sollen. Die Sperre erfolgt lokal, mittels TEX-Gruppen kann das blacklisting demnach kontrolliert werden. Mit `\solblacklistisghost` → p. 10 `\solblacklistispresent` → p. 10 kann kontrolliert werden, ob die ausgeblendeten Zeilennummern aus der Zählung entfernt werden sollen. Dies kann auch über den optionalen Stern geregelt werden (`\solblacklistisghost` → p. 10 ist standard, mit Stern wird `\solblacklistispresent` → p. 10 ausgeführt). Wiederholte Aufrufe fügen die zu versteckenden Zeilen den Bestehenden hinzu.

```
\solblacklistlinenumbers{3}
\begin{java}
public static void main(String[] args){
    System.out.println("Hello World");
    System.out.println("Ghooost!");
}
\end{java}
```

Ergibt:

```
public static void main(String[] args){
    System.out.println("Hello World");
    System.out.println("Ghooost!");
}
```

Im Gegensatz dazu, mit Stern:

```
\solblacklistlinenumbers*{3}
\begin{java}
public static void main(String[] args){
    System.out.println("Hello World");
    System.out.println("Ghooost!");
}
\end{java}
```

Ergibt:

```
public static void main(String[] args){
    System.out.println("Hello World");
    System.out.println("Ghooost!");
}
```

▷ `\solblacklistisghost`

Mit `\solblacklistlinenumbers` → p. 10 ausgeblendete Zeilen werden auch nicht gezählt.

▷ `\solblacklistispresent`

Mit `\solblacklistlinenumbers` → p. 10 ausgeblendete Zeilen werden dennoch gezählt.

2.4 Minted spezifisches

Ist die Paketoption `fakeminted` → p. 4 gesetzt, so existieren weiter die folgenden Befehle und Umgebungen.

▷ `env@minted[lising-Args]{language}`

Setzt den eingeschlossenen Text im Stil von `\solsetmintedstyle` → p. 11, wobei die Sprache wie in `\solminted-map` → p. 11 konfiguriert, abgebildet wird.

▷ `\solmintedmap{from}{to}`

Erstellt oder überschreibt eine Sprachabbildung für `env@minted` → p. 11. Durch `\RegisterLanguage` → p. 7 wird dies automatisch vollzogen (allerdings direkt durch interne Mechanismen).

▷ `\solsetmintedstyle{default/nonumber/plain/plain number}`

Aktualisiert den von `env@minted` → p. 11 zu verwendenden Stil.

A Sprachdefinition

A.1 Vordefinierte Sprachen

Die hier definierten Sprachdateien befinden sich im Unterordner Languages und können auch nur als Basis für eine eigene Sprachdefinition genutzt werden. Sie enthalten schon einige Beispielwerte, so wie ich sie verwende, können aber in der Hinsicht auch komplett auf „Blank“ zurückgesetzt werden. Hinweis: Die Definition der Sprachen verläuft, Überraschung, analog zu Lilly (`https://github.com/EagleoutIce/LILLY`, nur die geforderte Dateinennung und die weiteren Möglichkeiten unterscheiden sich.), also können auch diese Sprachdefinitionen als Beispiel herangezogen werden. Weitere Informationen bietet die Dokumentation zu `listings`.

Java (`lJava`, `java`)

Definiert in diesem Kontext nichts besonderes.

Bash (`lBash`, `bash`)

Definiert in diesem Kontext nichts besonderes.

Json (`lJson`, `json`)

Definiert in diesem Kontext nichts besonderes.

latex (`lLatex`, `latex`)

Definiert command als zusätzlichen Stil für Befehle.

XML (`lXml`, `xml`)

Definiert in diesem Kontext nichts besonderes.

Rubberduck (`lRubberduck`, `rubberduck`)

Definiert in diesem Kontext nichts besonderes. Auch keine echte Sprache :P.

A.2 Literates

Im Kontext verschiedener Programmiersprachen kam bald der Wunsch auf verschiedene Symbole entsprechend einfach Setzen zu können. Die Ersetzungsregeln werden grundlegend geladen, wenn das Paket geladen wird, allerdings können einzelne Programmiersprachen (so wie Java mit seinen Escape-Sequenzen) noch eigene definieren. Die Ersetzungsregeln ermöglichen es, neben Umlauten auch Symbole einzubinden. Die Ersetzungsregeln werden nicht über eine Liste gehandhabt und sind ebenso vielfältig wie es die Bedürfnisse erfordern. Im Folgenden eine Auflistung aller wichtigen Ersetzungsregeln:

:bs: „\“	**:ws:** „ “	**:float:** „f“	**:bcmd:** „\“
:bmath: „$“	**:cdots:** „···“	**:exp:** „e“	
:emath: „$“	**:cdot:** „·“	**:yields:** „→“	**:star:** „*“
:dollar: „$“	**:ldots:** „...“	**:lan:** „⟨“	
:space: „ “	**:c:** „“	**:ran:** „⟩“	**:percent:** „%“

Hinweis: Ja, `:c:` expandiert zu nichts. Es kann als Trenner verwendet werden. Allerdings sollte dies nicht notwendig sein, da das Paket von Version `1.0.14` an automatische Maßnahmen ergreift: `14 + 3 = 17 * (4 - 3)` → `$ 1↩ 7$... !`

B Beispiele und Tests

Die Codes hierzu finden sich am Ende des Quellcodes der Dokumentation:

So gilt: `public static void main(...)` definiert die `main`-Methode... Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

```
// Dies ist sehr guter Code
public static void main(String[] args){
    System.out.println("Hallo Welt");
    System.out.print(2+3*4-5)
}
```

```
1  import java.util.Scanner;
2
3  /**
4   * @file ean.java
5   * @author Florian Sihler, Raphael Straub
6   * @brief Das Programm berechnet anhand den ersten 12 Stellen einer EAN die zugehörige ↩
        Prüfsumme
7   * @version 1.0
8   * @date 2.11.2018
9   */
10
11 /**
12  * @class ean
13  * @brief Die Hauptklasse des Programms, in ihr 'lebt' der gesamte Code
14  */
15 public class ean {
16     ///@brief der Einstiegspunkt des Programms
17     public static void main(String[] args){
18         Scanner scanner = new Scanner(System.in); //Initialisierung
19         int sum = 0; boolean toggle = false;
20         for(int i = 0; i < 12; i++){ //Einlesen der 12 Zahlen
21                                      /* abwechselnd *1 und *3 */
22             sum += (scanner.nextInt() * ((toggle)?3:1)); //Summieren der Zahl
23             toggle = !toggle; //Invertieren des Multiplikators
24         }
25         System.out.println( (10 - (sum % 10)) % 10 );
26         // Gibt die Prüfsumme als den Abstand zum nächsten Vielfachen von 10
27         scanner.close();
28     }
29 }
```

```
<?xml version="1.0" encoding="UTF-8"?>
<collection>
    <book category="Roman">
        <!-- Ich bin ein Kommentar -->
        <title>Tolles Buch</title>
        <authorlist>
            <author>Netter Mensch</author>
            <author>Noch netterer Mensch</author>
```

```
        </authorlist>
        <description />
        <date> heute, wann sonst? </date>
    </book>
    <book category="Gute␣Bücher">
        ...
    </book>
</collection>
```

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

```
Einfach nur eine Verbatim-Ausgabe. 1234 Hallo42, Hallo 42, Hallo 42
```