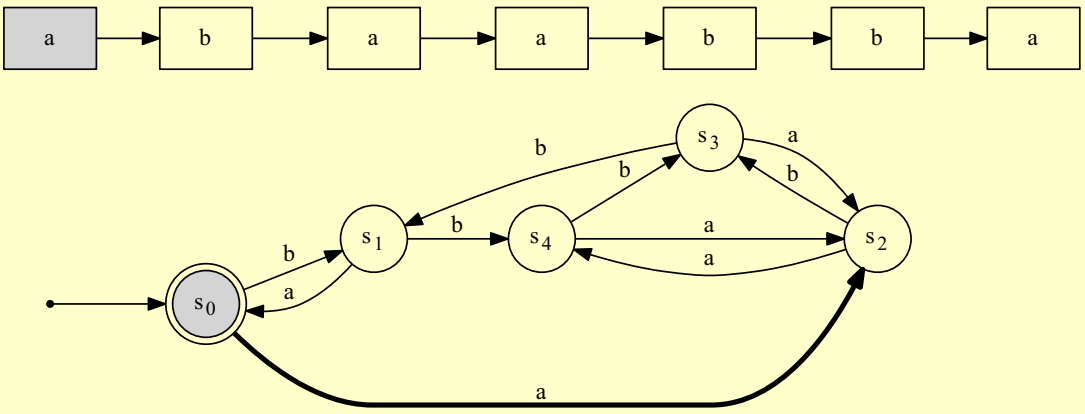
XWizard: Das Online-Informatik-Werkzeug

– Handbuch für Unterrichtende –

Lukas König, Friederike Pfeiffer-Bohnen, Alexander Dorsch



Skript ID-10700

XWizard: **Das Online-Informatik-Werkzeug**

– Handbuch für Unterrichtende –

Contents

1 Was ist XWizard?

XWizard ist ein frei verfügbares (Web-) Tool, mit dem viele Arten von Objekten aus verschiedenen Bereichen der Informatik erzeugt, bearbeitet und für Präsentationen, Veröffentlichungen usw. im PDF-Format aufbereitet werden können. Zu diesen Objekten gehören beispielsweise Turingmaschinen, Kellerautomaten, endliche Automaten, Chomsky-Grammatiken uvm. Zum Bearbeiten dieser Objekte bietet der XWizard eine Vielzahl von Algorithmen (deren Anzahl künftig weiter steigen wird). Die XWizard-PDF-Ausgaben sind meist intuitiv und übersichtlich, können aber auch individuell angepasst werden. Das Tool ist sowohl für den studentischen Gebrauch geeignet als auch (und besonders!) auf die typischen Arbeiten zugeschnitten, die bei Lehrpersonen anfallen, etwa das Erstellen von Aufgaben (das X in XWizard steht für "eXercise" – und auch für "beliebiges"). Dieses Handbuch erklärt die wichtigsten Funktionen von XWizard aus der Sicht eines Lehrenden. Für eine allgemeinere Beschreibung bietet sich das Dokument "XWizard: Handbuch für Studierende" oder ein Blick auf die Hilfeseite der XWizard Webseite an.

Der Leser, den der grundlegende Workflow und die generellen Funktionen von XWizard interessieren, kann den nächsten Abschnitt überspringen und direkt in Abschnitt 3 weiterlesen.

2 Zugang und kurze Geschichte

Aus Sicht der Lehrenden ist, zumindest für Beginner, das Benutzen der **Webversion**, die nur einen Browser benötigt, in den meisten Fällen die einfachste und geeignetste Entscheidung.

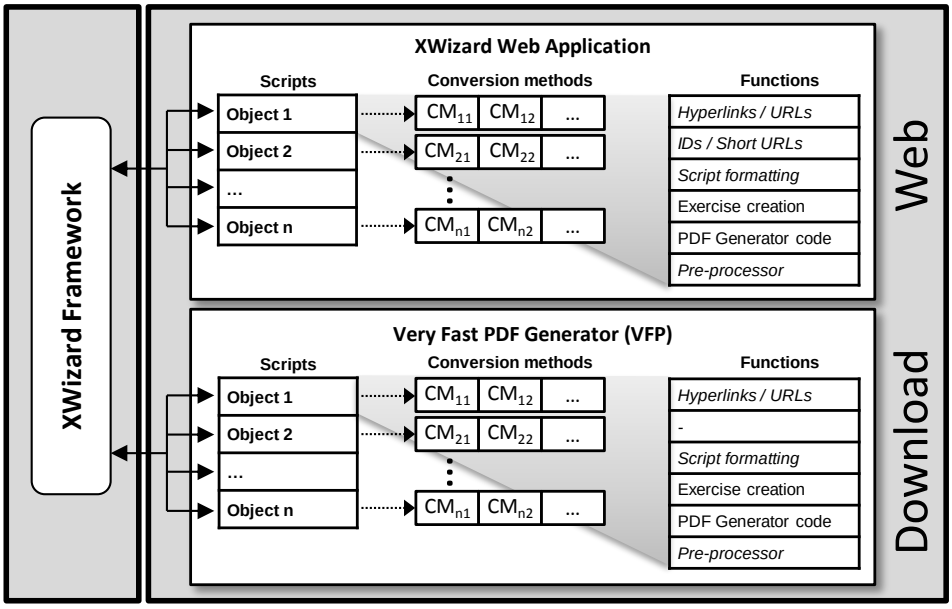
In XWizard kann zwischen englischer und deutscher Ausgabesprache gewählt werden; VFP ist bis jetzt nur in englisch verfügbar.

XWizard und alle dazu gehörenden Implementierungen sind *freie Software*[1]. Sie dürfen für beliebige Zwecke verwendet werden (kommerzielle ausgenommen), und der Quellcode darf studiert, verändert und weiter verbreitet werden, solange das im Einklang mit den unter folgendem Link abrufbaren Rechtsbelehrungen passiert: `http://www.xwizard.de:8080/Wizz?impressum&lang=ger`. (vergleiche Abschnitt 12 dieses Dokuments.)

3 Grundlegender Workflow: Skriptverarbeitung

Die folgende Abbildung stellt die grundlegende Struktur der XWizard Komponenten von Web- und Downloadversion dar (XWizard und VFP; wie zuvor erwähnt sind beide prinzipiell gleich). Die Objekte von XWizard werden durch **Skripte**, die durch **Konversionsmethoden** verändert werden können. Deshalb basieren die Hauptfunktionen von XWizard auf Konversionsmethoden. Jedoch sind Konversionsmethoden (und im Prinzip alles andere auch) in Skripten kodiert, deshalb kann man letztenendes alles auf das Erstellen und Interpretieren von Skripten zurückführen.

[1] `https://en.wikipedia.org/wiki/Free_software`

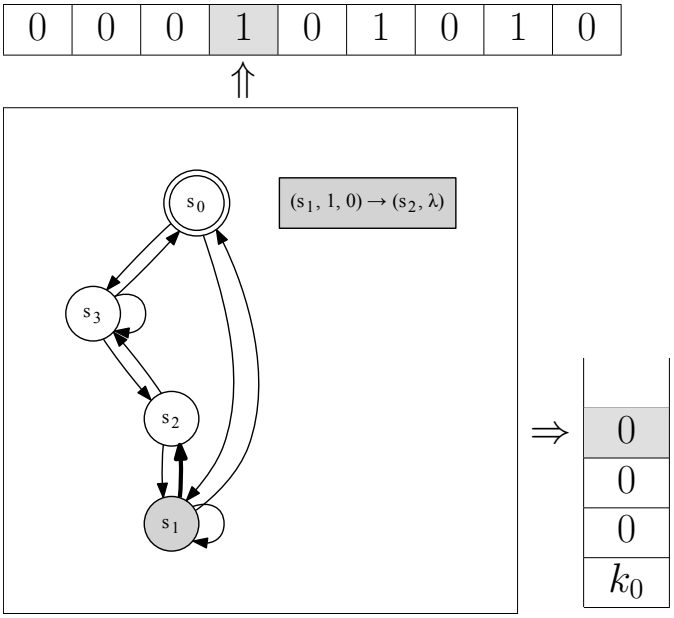


Die grundlegende Funktionsweise des XWizards beruht einfach darauf, ein Skript einzulesen, dieses in ein PDF-Bild zu übersetzen und anzuzeigen. Ein Skript besteht üblicherweise aus drei Teilen (vier, falls man die Ausführung einer Konversionsfunktion mitzählt, siehe unten), wie im folgenden Beispiel eines Kellerautomaten aufgeführt:

```
pda:                                                          ⇐ Skriptpräambel
    (s0, 0, k) => (s1, 0k);
    (s0, 1, k) => (s3, 1k);
    (s1, 0, 0) => (s1, 00);
    (s1, 1, 0) => (s2, lambda);
    (s1, lambda, k) | (s3, lambda, k) => (s0, k);
    (s2, lambda, 0) => (s1, lambda);                          } Hauptteil des Skripts
    (s2, lambda, k) => (s3, bk);
    (s3, 0, 1) => (s3, b);
    (s3, 0, b) => (s3, lambda);
    (s3, 1, 1) => (s3, 11);
    (s3, 1, b) => (s3, b1);
--declarations--
    e=#n#;
    s0=s0;
    F=s0;
    kSymb=k;                                                  } Variablendefinitionen
    inputs=000101010;
    simSteps=3;
--declarations-end--
```

Alle Skripte enthalten die folgenden drei Teile: Eine **Präambel** legt den Typ des Objekts, der durch das Skript definiert wird, fest; der **Hauptteil** definiert die eigentliche Struktur des Objekts; Variablen im Teil **Variablendefinitionen** können dazu genutzt werden, weitere Eigenschaften festzulegen (sowohl der komplette Variablendeklarationsteil, als auch einzelne Variablen können weggelassen werden. In diesem Fall werden die entsprechenden Variablen auf Standartwerte gesetzt.)

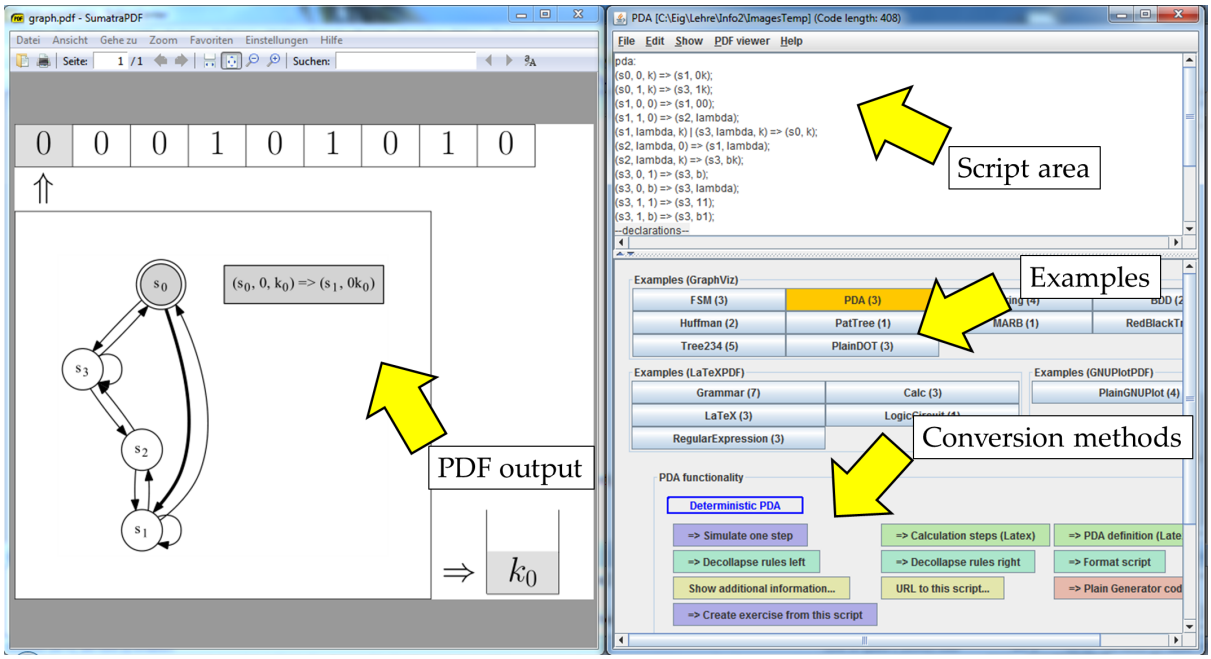
Dieses Beispielskript erzeugt die folgende Abbildung (im Hintergrund werden Graphviz und LaTeX verwendet):



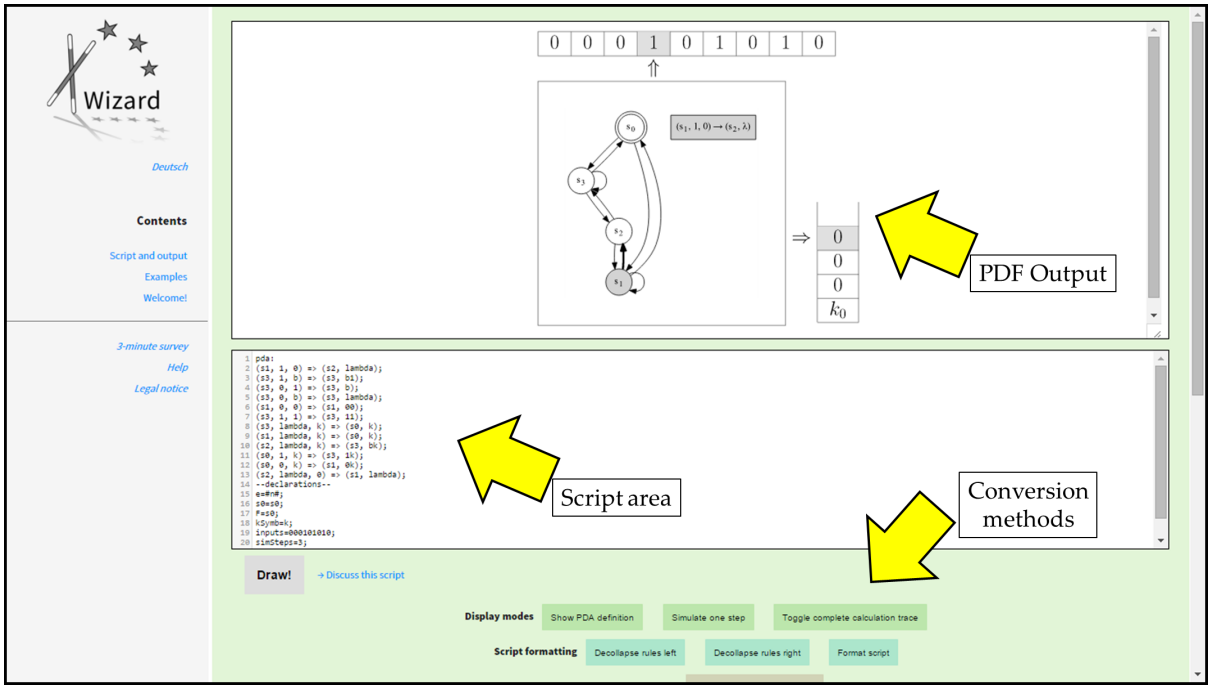
Skript ID-13417

Das Skript kann sowohl in VFP als auch in XWizard eingefügt werden, um die in den Screenshots angezeigte Ausgabe zu erzeugen (Klicken auf den Link oder Scannen des QR-Codes leiten weiter zu XWizard und führen das Skript automatisch aus).

VFP:



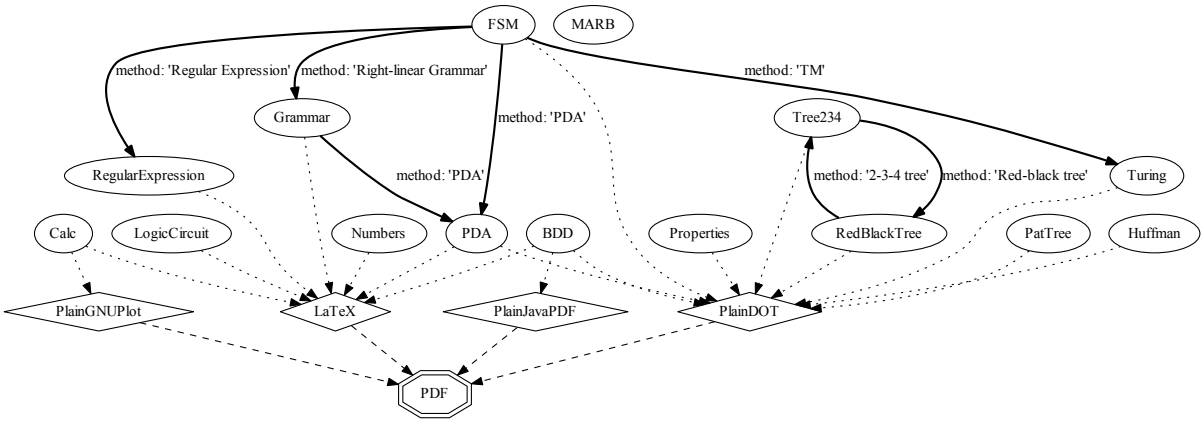
XWizard:



Um die Berechnung des angezeigten Kellerautomaten zu simulieren, kann man die "simSteps"-Variable im Skript erhöhen; ebenso kann man die **Konversionsmethode** anklicken (vergleiche dazu auch die Beschreibung von Konversionsmethoden weiter unten).

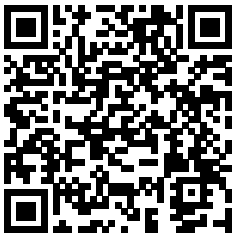
Bei der Verwendung von VFP wird die PDF-Ausgabe automatisch mit Veränderung des Skripts aktualisiert. Bei XWizard wird die Bildausgabe generiert, nachdem der "Draw!"-Knopf auf der Webseite geklickt wurde. Eine PDF kann durch Drücken des "Download PDF"-Links abgerufen werden; dieser erscheint beim Scrollen **unter** die PDF Ausgabe.

Einen Überblick über die verfügbaren Skripttypen zum Zeitpunkt des Verfassens dieses Dokuments ist durch das folgende Diagramm dargestellt(durchgezogene und gepunktete Pfeile bezeichnen Konversionen zwischen verschiedenen Skripttypen durch Konversionsmethoden; gestrichelte Pfeile bezeichnen den PDF-Generierungsprozess; rautenförmige Knoten repräsentieren die direkten PDF-erzeugende Skripttypen – hierbei handelt es sich vor allem um Graphviz und LaTeX (GNUPlot und JavaPDF sind außer Gebrauch, siehe unten).



Eine aktuelle Version der Abbildung erhält man über den folgenden Link (beachten Sie, dass sie selbst über ein einfaches XWizard Skript generiert ist):

Skript ID-15812



Für alle dieser Objekttypen gibt es Beispielskripte auf der XWizard Webseite:

`http://www.xwizard.de:8080/Wizz?lang=eng&hide#Examples`

Es sei angemerkt, dass die Skriptsyntax hier nicht näher betrachtet wird; dies kann auf den XWizard Hilfeseiten nachgelesen werden:

`http://www.xwizard.de:8080/Wizz?help&lang=eng&hide`

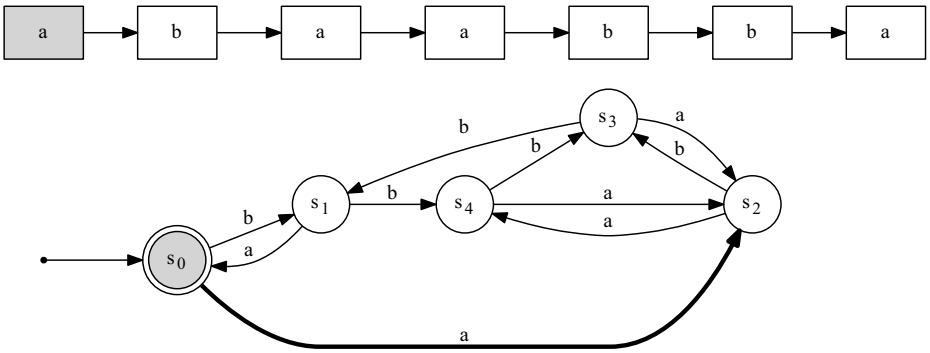
Zusammenfassend zu diesem kleinen Abschnitt kann man sagen, dass der grundlegende Workflow von XWizard dadurch gegeben ist, dass aus einem Inputskript ein PDF-Bild gemacht wird. Nahezu alle GUI-basierenden Abkürzungen oder Vereinfachungen, die im Folgenden beschrieben werden können durch das Erstellen eines zugehörigen Skriptes ersetzt werden; genauer gesagt ist alles, was die GUI im Hintergrund macht ist die Erstellung von entsprechenden Skripten. Der Hauptvorteil dieser Herangehensweise ist, dass jede Aktion archiviert werden kann, indem man den zugehörigen Skript speichert, vergleiche auch Abschnitt 6.

4 Konversionsmethoden

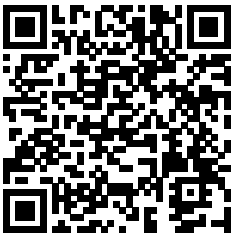
Neben dem Erstellen und Anzeigen von Objekten ist eine Hauptfunktion von XWizard das Anwenden von Algorithmen auf Skripte, also das Visualisieren der schrittweisen Berechnung eines KAs oder der Minimierung eines EAs. Algorithmen werden auf die Objekte durch Konversionsmethoden angewendet, also Methoden, die einen Skript in einen anderen überführen. Konversionsmethoden bieten eine einfache Benutzerschnittstelle für das Anwenden von Algorithmen auf XWizard Objekte.

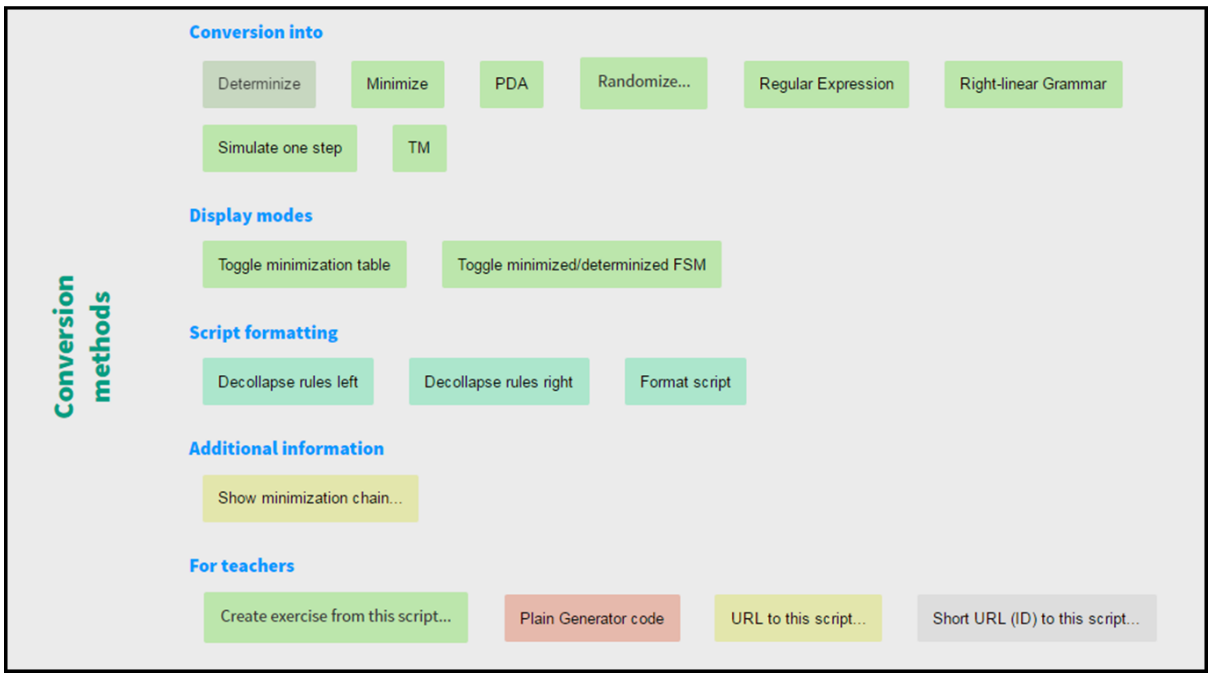
Wenn man eine Konversionsmethode anwendet, dann wird der zu dem aktuellen Objekt gehörende Skript durch das neue von der Konversionsmethode erzeugte Skript ersetzt und das Objekt, das vom neuen Skript definiert wird, wird erzeugt und angezeigt. (Es sei jedoch angemerkt, dass es auch Konversionsmethoden gibt, die einen reinen Text statt einem Skript ausgeben.)

Der einfachste Weg eine Konversionsmethode auf einen Skript anzuwenden ist es auf den zugehörigen Knopf in dem “Konversionsmethodenbereich” der graphischen Schnittstelle zu klicken. So gibt es beispielsweise bei dem endlichen Automaten (EA) die im Folgenden angezeigten Konversionsmethoden;



Skript ID-10700





4.1 Konversionsmethoden, die ein neues Skript anlegen

Konversionsmethoden, die einen Skript in einen anderen überführen sind die üblichsten – und bei weitem häufigsten. Verfügbare Konversionsmethoden werden unter dem Skriptbereich angezeigt; der aktuelle Skript legt fest, welche Konversionsmethoden anwendbar sind und nur diese werden angezeigt. In vorigem Beispiel wurden die folgenden Konversionsmethoden, gruppiert in den blauen Kategorien, für den Skript eines EAs angezeigt:

- **Conversion into** (deutsch: Überführung zu) – Methoden, die ein "neues", semantisch verschiedenes Objekt erzeugen:
  - **Determinize** (deutsch: "Mache deterministisch".) benutzt den allseits bekannten Potenzmengen-Algorithmus, um einen Skript zu erstellen, der einen, zum ursprünglichen äquivalenten, deterministischen EA beschreibt.

  - **Minimize** (deutsch: Minimiere) erzeugt einen äqivalenten EA mit einer minimalen Anzahl an Zuständen, indem die Myhill/Nerode Äquivalenzaussage verwendet wird.
  - **PDA** (deutsch: KA), **Regular Expression** (deutsch: regulärer Ausdruck), **Right-linear Grammar** (deutsch: rechtslineare Grammatik) und **TM** erzeugen entsprechend ein Skript für entweder einen äquivalenten Kellerautomaten, einen regulären Ausdruck, eine rechtslineare Grammatik oder eine Turingmaschine.
  - **Simulate one step** (deutsch: Simulation eines Zeitschrittes) lässt den EA ein weiteres Eingabesymbol abarbeiten oder erfragt einen Input, falls kein Eingabewort gegeben ist.
  - **Randomize...** (deutsch: Randomisiere...) erzeugt, nachdem der Benutzer die Anzahl der Zustände und eine boolsche Zufallsvariable (legt fest, ob der KA deterministisch ist) spezifiziert hat, einen neuen zufälligen EA. (Der erzeugte EA wird immer minimierbar sein, um das Erstellen von Aufgaben zu vereinfachen.)

- **Display modes** (deutsch: Anzeigemodi) – Methoden, die die Ansicht des aktuellen Objekts, nicht jedoch dessen Semantik, verändern:
  - **Toggle minimization table** (deutsch: Minimierungstabelle zuschalten) wechselt zwischen Ansichten, in denen nur der EA, der EA und die Minimierungstabelle sowie nur die Minimierungstabelle angezeigt werden.
  - **Toggle minimized/determined FSM** (deutsch: minimierter/deterministischer EA zuschalten) wechselt zwischen Ansichten, in denen zusätzliche Darstellungen des aktuellen EA in minimierter oder deterministischer Form angezeigt werden.

    
    Das Anzeigen von verschiedenen äquivalenten Versionen des selben EA erlaubt es diese gleichzeitig zu Simulieren, was für Studierende und während der Aufgabenerstellung von Interesse sein kann.

- **Script formatting** (deutsch: Skriptformatierung) – Methoden, die die Skriptdarstellung verändern, nicht jedoch die Ausgabe:
  - Die zwei Methoden **Decollapse rules left** (deutsch: Ausklappen) oder **right** sind für alle Skripte mit Regeln wie `A => B` verfügbar. Diese Regeln können für bessere Lesbarkeit auf der linken Seite (`A | B => C`) und auf der rechten Seite (`A => B | C`) zusammengefasst ("zusammengeklappt") werden. Die "Decollapse" Methoden werden dieses Zusammenfassen links, bzw. rechts rückgängig machen.
  - Die Methode **Format script** fasst die Regeln wo möglich zusammen und nimmt einige Formatierungen vor.

    
    Insbesondere fügen die Methoden **Format script** und **Add declarations to script** (welche davon hängt vom aktuellen Skripttyp ab) den Deklarationsteil, der alle verfügbaren Variablen enthält, zum Skript hinzu.

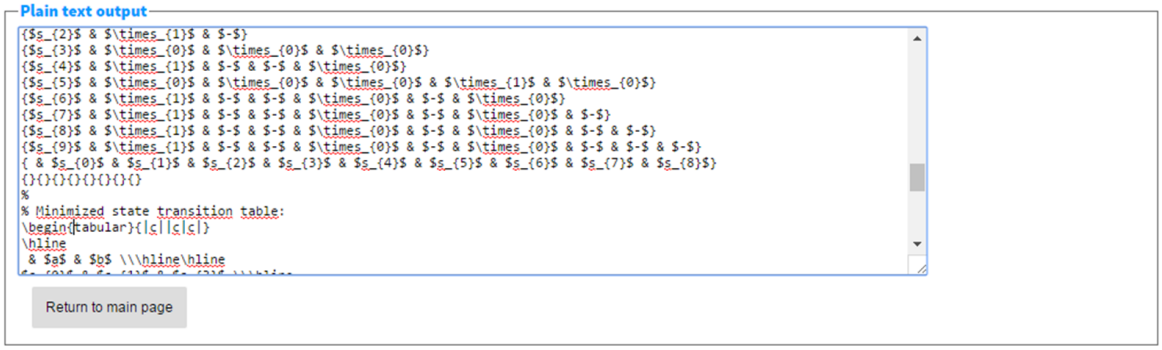
- **Additional information** (deutsch: zusätzliche Informationen) – Als einzige Methode in dieser Kategorie ist **Show minimization chain** (deutsch: Zeige die Minimierungsreihenfolge an) eine reine Textkonversionsmethode (vergleiche Abschnitt 4.2). Sie erzeugt eine Ausgabe, die direkt in ein LaTeX Dokument eingefügt werden kann und Informationen darüber gibt, wie der aktuelle KA zuerst deterministisch gemacht und anschließend minimiert wird.

- **For teachers** (deutsch: für Lehrende) – Die Methoden dieser Kategorie sind für alle Skripte verfügbar. Da sie mit allen Hauptthemen dieses Handbuchs verknüpft sind werden sie genauer in den Abschnitten 5, 6 und 7 beschrieben.

Drei Punkte (...) am Ende des Namens eines Knopfes weisen auf Methoden hin, die die Interaktion mit dem Benutzer erfordern. Manche dieser Methoden benötigen zusätzliche Parametereingaben um ausgeführt werden zu können, diese Methoden werden entsprechenden Parameter vor der Überführung erfragen, vergleiche auch mit der oben erklärten "Randomize..." Methode. Die übrigen Methoden mit Punkten sind jene, die eine einfache Textausgabe erzeugen. Sie werden bloß ein neues Fenster öffnen, um die Ausgabe anzuzeigen und bitten den Benutzer die Ausführung zu bestätigen.

4.2 Konversionsmethoden, die eine einfache Textausgabe liefern

Neben den oben beschriebenen, normalen Konversionsmethoden existieren auch solche, die eine einfache Textausgabe liefern. Als Beispiel dient die Methode “Show minimization chain” eines EA Skriptes, die einen LaTeX Code ausgibt:



In diesem Fall ist die Ausgabe ein LaTeX Code, der Informationen über die Minimierung und Determinierung des aktuellen EA enthält. Ferner sind die Hauptkonversionsmethoden, die eine reine Textausgabe liefern:

- “URL to this script...” und
- “Short URL to this script...”

Diese Methoden erzeugen URLs zu dem aktuellen Skript, um den Austausch von Skripten zwischen Benutzern zu vereinfachen, vergleiche Abschnitte 6.1 und 6.2. Davon abgesehen werden reine Textkonversionsmethoden sehr selten benutzt und können normalerweise durch eine geeignete normale Konversionsmethode ersetzt werden. Beispielsweise kann man den LaTeX Code der Minimierungsreihenfolge erhalten, indem man einen passenden Anzeigemodus auswählt und den “Plain Generator code” verwendet, siehe Abschnitt7.

4.3 Anwendung von Konversionsmethoden per Skript

Das Anwenden von Konversionsmethoden per Skript ist ein fortgeschrittenes Werkzeug, das zu diesem Handbuch für interessierte Leser hinzugefügt wurde. Es wird nicht essentiel benötigt, um den üblichen Workflow von XWizardnachvollziehen zu können, folglich kann der letzte Teil dieses Abschnitts auch übersprungen werden.

Wie oben bereits erwähnt läuft XWizard komplett über Skripte; das bedeutet insbesondere, dass sogar die Anwendung einer Konversionsmethode über einen besonderen Skriptaufruf angestoßen werden kann. Skripte, die diesen Befehl enthalten, werden **conversion scripts** (deutsch: Konversionsskripte) genannt und sie beginnen wie üblich mit den drei Teilen, die das Skript festlegen, der umgewandelt werden soll. Als vierten Teil wird ein **conversion command** (deutsch: Konversionsbefehl) in die letzte Zeile des Skripts geschrieben. Dieser sieht wie folgt aus:

```
**>CM-NAME<**
```

dabei ist `CM-NAME` der englische Name der anzuwendenden Konversionsmethode. Für den Fall, dass die Konversionsmethode Parameter erfordert lautet der Befehl wie folgt:

```
**>CM-NAME[p1, p2, ...]<**
```

dabei sind `p1, p2, ...` die Methodenparameter – diese können in Anführungszeichen gesetzt werden, falls Sonderzeichen, wie beispielsweise Leerzeichen oder Kommata, benötigt werden:

```
["p, a, r, 1", "p, a, r, 2", ...]
```

Die “Simuliere einen Schritt”-Methode kann beispielsweise auf ein Kellerautomaten-Skript angewendet werden, indem man die rot-gefärbte letzte Zeile hinzufügt:

```
pda:
    (s1, 1, 0) => (s2, lambda);
    (s3, 1, b) => (s3, b1);
    (s3, 0, 1) => (s3, b);
    (s3, 0, b) => (s3, lambda);
    (s1, 0, 0) => (s1, 00);
    (s3, 1, 1) => (s3, 11);
    (s3, lambda, k) => (s0, k);
    (s1, lambda, k) => (s0, k);
    (s2, lambda, k) => (s3, bk);
    (s0, 1, k) => (s3, 1k);
    (s0, 0, k) => (s1, 0k);
    (s2, lambda, 0) => (s1, lambda);
--declarations--
    e=#n#;
    s0=s0;
    F=s0;
    kSymb=k;
    inputs=000101010;
    simSteps=3;
    maxNondetCalcDepth=12
--declarations-end--
**>Simulate one step<**   /* This is a conversion command. */
```

Wenn dieses Skript eingegeben wird, so ist das Ergebnis exakt dasselbe, wie für den Fall, dass man das entsprechende Feld zur Konversion des Skriptes angeklickt hat (ohne den Konversionsbefehl).

5 Der Aufgabenmodus und verschlüsselte Skripte

XWizard hat einen besonderen Modus, der **Exercise Mode** (deutsch: Aufgabenmodus) genannt wird (derzeit nur in der Webversion verfügbar). In diesem Modus wird der Benutzer (üblicherweise der Student) dazu aufgefordert die oben auf der Seite angezeigte Aufgabe zu lösen. Um zu der Lösung zu gelangen ist es dem Benutzer gestattet eine vordefinierte Teilmenge der XWizard's Funktionen zu nutzen. Diese Teilmenge kann flexibel vom Aufgabenersteller definiert werden (üblicherweise eine Lehrperson) und sie kann sowohl eine Anpassung der verfügbaren Konversionsmethoden, als auch eine Beschränkung der Skriptnutzung beinhalten. Genauer gesagt unterscheidet sich der Aufgabenmodus in den folgenden Punkten vom normalen Modus:

(1) Die XWizard Webseite stellt eine Frage, die über dem Skriptbereich angezeigt wird und verlangt vom Benutzer, dass er selbige beantwortet (siehe Screenshot)

(2) Ein paar der Konversionsmethoden können verborgen werden, um unerwünschte Abkürzungen auf dem Lösungsweg zu verhindern.

(3) Der Skript kann teilweise oder auch komplett verschlüsselt sein, um das Schummeln durch Verändern des Skripts oder das Lesen der Aufgabendefinition zu verhindern (siehe unten).

(4) Falls die Aufgabe korrekt beantwortet wurde erhält der Benutzer eine "Auszeichung". (Bis jetzt handelt es sich dabei nur um ein Passwort, das dem Benutzer angezeigt wird; zukünftig ist die Einbindung von persönlichen "Portfolios", die es dem Benutzer ermöglicht die Auszeichnungen, Erfahrungspunkte o.Ä. zu sammeln.)

(5) Falls die Aufgabe korrekt beantwortet wurde kann dem Benutzer, falls der Aufgabenersteller dies bereitstellt, eine zusätzliche Erklärung angezeigt werden.

5.1 Erstellen einer Aufgabe

Der Aufgabenmodus wird durch die Variable "`e`" (oder "`exercise`") im Deklarationsteil des Skripts definiert. Folglich muss diese Variable auf eine spezielle Art und Weise definiert werden um eine Aufgabe zu erstellen. Eine Konversionsmethode **Create exercise from this script** (deutsch: Erstelle eine Aufgabe

aus diesem Skript) (Diese Variable ist für jeden Skripttyp in der "For teachers" Kategorie verfügbar.) vereinfacht diesen Prozess. Beim Ausführen dieser Methode wird der Benutzer nach den folgenden Parametern gefragt (Alle bis auf die letzten beiden sind Strings; optionale Parameter können einfach leer gelassen werden.):

- titleString: Ein Titel, der über der detaillierten Aufgabenbeschreibung steht.
- explanationHTML: Eine detaillierte Beschreibung der Aufgabe, die HTML-Code enthalten kann.
- solutionString: Ein String, der die Lösung angibt, die der Benutzer eingeben muss. Dieser Parameter ist optional, falls er leer gelassen wurde, der Benutzer wird nicht zur Eingabe einer Lösung aufgefordert.
- codeToEarn: Die "Auszeichnung", also ein Sting der dem Benutzer als Belohnung angezeigt wird, falls die Aufgabe korrekt gelöst wurde.
- regexForAllowedMethodNames: Ein optionaler regulärer Ausdruck (wie er in Java benutzt wird), um die angezeigten Konversionsmethoden einzuschränken. Der reguläre Ausdruck wird auf den englischen Methodennamen angewendet und zeigt nur passende Methoden an. Beispielsweise zeigt der Ausdruck

  `.*inimiz.*`

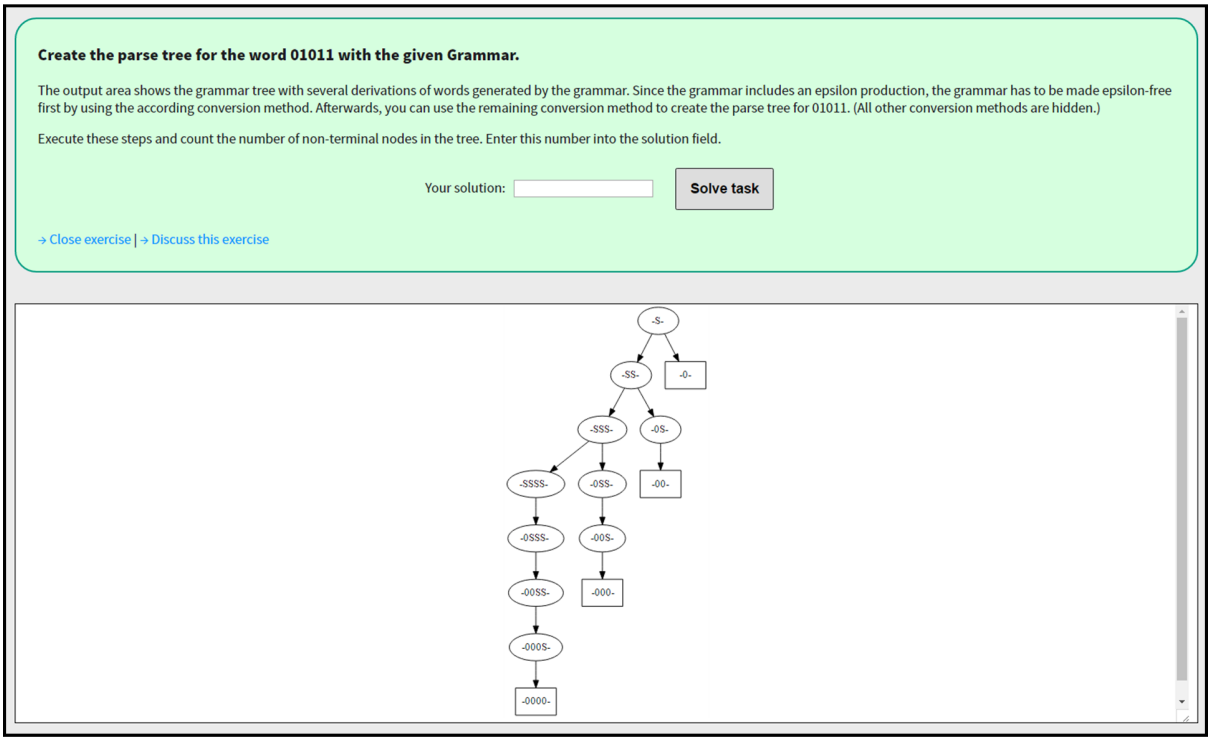
  nur die zwei Methoden **Minimize** und **Show minimization chain...** an.
- regexForAllowedClassNames *(nur für den fortgeschrittenen Gebrauch, kann meist leer gelassen werden)*: Ein weiterer optionaler regulärer Ausdruck, um die angezeigten Konversionsmethoden einzuschränken. Er wird auf den Klassennamen der Basisklasse, die die Methode zur Verfügung stellt, angewendet.
- regexForAllowedTargetClassNames *(nur für den fortgeschrittenen Gebrauch, kann meist leer gelassen werden)*: Ein dritter regulärer Ausdruck, um die angezeigten Konversionsmethoden einzuschränken. Er wird auf den Klassennamen der Zielklasse angewendet, also der Klasse des konvertierten Skripts.
- solExp: Optionale Erklärung, die mit der Lösung angezeigt wird, nachdem die korrekte Antwort gegeben wurde (kann HTML enthalten).
- exEncrypt *(boolean)*: Setzen Sie diesen Parameter auf wahr, falls der Code der Aufgabe (nicht der komplette Skriptcode) verschlüsselt werden soll (siehe unten).
- encrypt *(boolean)*: Setzen Sie diesen Parameter auf wahr, falls der komplette Skriptcode verschlüselt werden soll (siehe unten).

Nach der Eingabe dieser Parameter wird eine Aufgabendefinition zu dem aktuellen Skript hinzugefügt, die XWizard dazu veranlasst in den Aufgabenmodus zu wechseln. (Der Code für den "regular mode" ist "`e=n`" oder "`e=null`".)

Der folgende Skript ist ein Beispiel dafür, wie eine Aufgabe im Deklarationsteil eines Skripts codiert wird. Die Ausgabe davon wird in dem Bildschirmfoto darunter angezeigt. Klicken Sie auf den Skriptlink, um ein Beispiel im Browser ansehen zu können.

```
grammar:
    A => A, A | 0 | epsilon;
    E => A, 1, A;
    S => E, E, E | S, S | 0;
--declarations--
    e=#tit=~Legen Sie den Syntaxbaum für das Wort 01011 mit der gegebenen Grammatik an.~,
    exp=~<P>Im Ausgabebereich wird der Grammatikbaum mit einigen Ableitungen von Wörtern,
    die von der Grammatik erzeugt wurden, angezeigt. Da die Grammatik einen
    Epsilonübergang enthält, muss sie zuerst epsilonfrei gemacht werden, indem man die
    zugehörige Konversionsmethode anwendet. Danach kann man die verbleibenden
    Konversionsmethoden dazu verwenden, den Syntaxbaum für 01011 anzulegen. (Alle
    anderen Konversionsmethoden sind verborgen).</P><P>Führen Sie diese Schritte aus und
    zählen Sie die Anzahl der Nonterminalknoten im Baum. Geben Sie diese Nummer in das
    Lösungsfeld ein.</P>~,
       sol=~6~,
       cod=~parser-guru~,
       met=~.*Epsilon.*|.*Parse.*~,
       cur=~.*~,
       tar=~.*~,
       crypt=~false~,
       excrypt=~false~,#;
    N=S,E,A;
    T=0,1;
    S=S;
    displayMode=2;
    maxdepth=8;
    cutNonTerminalBranches=true;
    cutTerminalDoubleBranches=true;
    maxLengthWords=4;
    multiLetterSymbolsHaveIndex=true;
    parseTreeNum=0;
--declarations-end--
```

In dem Deklarationsteil zeigen die Symbole # und ~ den Beginn und das Ende von Strings an, die Sonderzeichen, wie zum Beispiel "= , ;" etc., enthalten können (jedoch können die Symbole # oder ~ selbst als auch einige reservierte Wörter nicht verwendet werden). Um einen String komplett in den Deklarationsteil einzubinden, kann er in das Klammerkonstrukt `[~(~{ bel. Text }~)~]` eingebunden werden. Das erlaubt es alle Buchstabenkombinationen inklusive untergeordnete Instanzen der Klammerkombinationen selbst einzubinden, bis der gesicherte Teil durch die passende schließende Klammer abgeschlossen wird (vergleiche weitere Erklärungen in Abschnitt 8).



Skript ID-11020

Wie der Beispielskript zeigt ist die Aufgabendefinition bis jetzt als normaler menschenlesbarer Text im Skript gegeben. Das bedeuted insbesondere, dass Studierende in der Lage sind sowohl die Lösung zu lesen, als auch Teile der Aufgabendefinition zu verändern. So können sie beispielsweise die Konversionsmethoden anzeigen lassen, die eigentlich verborgen sein sollten. Da dies unerwünscht ist können **zwei Niveaus der Verschlüsselung** auf den Skript angewendet werden. Das erste Niveau wird durch Setzen des Aufgabenparameters

```
excrypt=true
```

aktiviert, was dazu führt, dass die Aufgabendefinition verschlüsselt wird. Das zweite Niveau verschlüsselt den gesamten Skript und kann aktiviert werden, indem

```
crypt=true
```

gesetzt wird. Die zwei Beispielskripte unten zeigen die zugehörigen Ergebnisse; die Ausgabe auf der Webseite bleibt die selbe wie oben.

Beispielskript mit “excrypt=true”:

```
grammar:
    A => A, A | 0 | epsilon;
    E => A, 1, A;
    S => E, E, E | S, S | 0;
--declarations--
    e=#scrypt:401s3X3o133k2L2R2g202p2z0x3w1Q1G052n0b040u2K0K0l1y2b1k161t0d0Q1J0I
              1K0A0B2m1c0T1B3G3726250T1q2y1h0c3d3f2B1T1D350z3j2c3L3a3u2M0b102s2l
              061a0J3X0e1K1z3Y430X1v142Z0A0N1T0e1z042831301F1r0O2J2x102O450V1x35
              0Y1G2M1E0Q1C283s3f3718400m3X3F170K1m0d3d1O043O2H1O1a1z3G3p082E2k07
              2G2H3S3a1k3D382j071d1p45271h012w460z1m180k332e3u263y2E3i0h3C1z1G0P
              0H0A1v2X1K3K1n3k0427143O3346143R063Y3S3C3f3r2P3h0w3F3x0P1g0F0t3w0V
              3e3M3t0k2k3I3i3F1S2a2y0704440S3M0T033h1e1y3q351d1I0d3T0S01463z0I1Y
              1M1h163x1i2a0K2y0D2d0y1g1X2H011m3O3D2g1w2D0p1J1Z471Z2b3j461V0a2O0L
              0k3g2H3f2P1l32OY3Q3J3g3j081Q443m300d0o1S3E0R342b1k1W2J1o1G262V3i0R
              3D3C3d301n301a1F3o2A2V3z252m0D3y3p1n0M3S1Q0C0a0n3k0r450V2L0t0b102P
              1L2y102Z3I1g0B3p15452R0r1v2u0P1r0Y0p1f101g1f3c1j0p1J0I3W1b46322w3B
              2q172j3Y2m0I471I0z3k0s0R442b233Z1U0C#;
    N=S,E,A;
    T=0,1;
    S=S;
    displayMode=2;
    maxdepth=8;
    cutNonTerminalBranches=true;
    cutTerminalDoubleBranches=true;
    maxLengthWords=4;
    multiLetterSymbolsHaveIndex=true;
    parseTreeNum=0
--declarations-end--
```

Beispielskript mit “crypt=true”:

```
scrypt:401s3X3p1t0a1t2Q2k452j1C2z2a1U0B2h0O0r0k3o3g0X3w3e3x1l2q0S2B1y360v2p3N3E
     2f3G1b1G0i2f2M221U0e1C280W1v2q2w1B0p2T3V1E1e3D1k301j1V14450s1h1o122z3C1f3C
     2h3l0M0A2z2i3A3L3N341f3J1n2p3v2l3e3D0t140a1D071I3M2a3X3M2B3G342L2e090d2q3k
     283H0d2P0a1X2g1X3z3m010f3B2w0a002k3004372Y3l3B2i2l2i1o1D1m1b302d1d1n3U0R02
     0j2e3z1g0R1I082a103j261d0h1z411y1B211U3D3d0o0d3D0825280h021N1K1b3l2x3p1L3a
     0t41322y1j2d2h2m0t303k0y122y1Q2Y2j2N2u26273g1q3B221K2P0x3s0C133J0k0k1i3n0y
     2b3D3Q1g121y0I0c3j2C3H1h3M31373U2M1K2l422B1P3s3A2w1Q141c1X0g1E3b0L3i0r1t3a
     1l0g0s2o2V1o0g3y3l041i3F1G0I2M3D452c1G3k021C2S0U1q2c2h1f0f461k1E0M2r1c200P
     2R1f3R221J402W0W043t3N0J132m1M3A2m1W1C2N3W3g3g2U1D1J1Z2X1t2b3U10303F2n2H3g
     0y1R012W2D352E3X3d2n1Q2A2n1Z331j0h17062W2G1H0Z2S3G110N1b303H3U3j0i3n383H2c
     0B3u1X3x0j2S2E302p400H3F3U011e382N1d2U443G0k1C1B1Z0h2V3s1Q2R1N1B0a2W0W2Y2S
     2m1e193U3n3j2W0n0b0D3U0N3j400V0w1I1G2m2W0O2A0K2z0d0X2g3q2z3y163W3o1p1O2A0Q
     0D1W0e0e40153N3s0N3W1G3110 3G0H3W1O0Y0C0T3q461i1A1Q0j3s1z3R2J1M173v1A2k3022
     01260O3H3F1f1M0W1N3J2H3i392y1P2U3H383R1i3t2v1V2u232O2A0K3d2U3P1G460B151C1N
     3o2K1I0w2y2R0g3h1T0B1x3x1L110b2D222Q2m1h0T0K1K140e1u1u160u2P3B201X1i1a0y1W
     2h3i3q3r08
```

Es sei betont, dass die Verschlüsselung nur dazu dient das Schummeln zu erschweren – nicht es ganz auszuschließen. Die Verschlüsselung wird durch eine Kombination, von den Text zippen und das Ergebnis davon zu alphanumerischen Werten zu konvertieren, erzielt. Da der XWizard Quellcode frei verfügbar ist können ambitionierte Studenten diesen Herunterladen und ihn dazu verwenden einen Skript zu decodieren. Nichtsdestotrotz sollte man sich darüber im Klaren sein, dass nachdem man eine Verschlüsselung in XWizardangewendet hat es **nicht einfach ist dies rückgängig zu machen**, also ist es sinnvoll eine nicht verschlüsselte Version als Sicherung jedes Skripts aufzubewahren.

6 Hyperlinks zu XWizard Skripten

XWizard kann URLs erzeugen, die auf bestimmte Skripte verweisen, um einen Austausch dieser zu vereinfachen. Diese Technik wurde verwendet, um die Skriptverlinkungen in diesem Dokument zu erzeugen. Wenn man einem solchen Link folgt, so öffnet sich die XWizard Webseite und lädt automatisch den zugehörigen Skript. Es gibt zwei verschiedene Typen von URLs ("lange" oder "kurze") die für diesen Zweck genutzt werden können; sie werden im Folgenden erklärt.

Kurz URLs sind im Vergleich zu den langen URLs in den meisten Situationen aufgrund von Zweckmäßigkeit und Sicherheit zu bevorzugen, da lange URLs zu lang werden können, um noch von einem Browser akzeptiert werden zu können. Nichtsdestotrotz enthalten lange URLs, im Gegensatz zu kurzen, die gesamten Informationen des Skripts, was sie unabhängig von der XWizard Datenbank (siehe unten) macht.

6.1 Lange URLs

Jeder Skripttyp bringt die Konversionsmethode **URL to this script...** (deutsch: URL zu diesem Skript ...) mit, die das, was hier als "lange URL" bezeichnet wird, erzeugt. So führt beispielsweise der folgende Skript zu der darunter aufgeführten URL:

```
latex:
    \mbox{XWizard long URL}
--declarations--
    formulaMode=true;
    e=\#n\#;
--declarations-end--
```

```
http://www.xwizard.de:8080/Wizz?template=latex%3A%0D%0A%5Cmbox%7BXWizard+long+
URL%7D%0D%0A--declarations--%0D%0AformulaMode%3Dtrue%3B%0D%0Ae%3D%23n%23%3B%0D%
0A--declarations-end--
```

Diese URL führt auf die XWizard Webseite und überbringt einen Parameter "template", der das gesamte Skript als URL-codierten String enthält. Wird solch ein Link geöffnet, so decodiert XWizard den Skript, kopiert ihn in das Skriptfeld und zeigt das zugehörige Ausgabebild.

Prinzipiell kann jeder Skript in solch eine lange URL umgewandelt werden, jedoch kann es passieren, dass wenn die URL zu lang wird der Browser Teile davon ablehnt was zu fehlerhaften Skripten führt. Des Weiteren können Maleware Blocker Alarm schlagen aufgrund der unüblichen Struktur dieser URLs. Aus diesen Gründen wurden **kurz URLs** eingeführt.

6.2 Kurz URLs, Skript IDs und die XWizard Datenbank

Ein Skript ähnlicher Größe, wie in dem vorigen Abschnitt gezeigt, wie zum Beispiel folgender Skript:

```
latex:
    \mbox{XWizard short URL}
--declarations--
    e=\#n\#;
    formulaMode=true
--declarations-end--
```

kann in einer weitaus einfacheren URL kodiert werden, die generiert wird, wenn die Konversionsmethode **Short URL to this script...** (deutsch: kurze URL zu diesem Skript...) ausgeführt wird.

```
http://www.xwizard.de:8080/Wizz?template=ID-11567
```

Anstatt, dass ein kompletter Skritp in der URL kodiert wird, benutzen kurze URLs die XWizard Datenbank. XWizard (nicht VFP!) legen jeden verarbeiteten Skript mit verschiedenen zugehörigen Informationen in einer mysql Datenbank ab. Jedem Skript wird dazu eine ID in der Datenbank zugewiesen, die man verwenden kann, um den Skript später wieder zu finden. Wenn diese ID an XWizard über den "templates" Parameter eingereicht wird, so wird nach diesem in der Datenbank gesucht und überprüft, ob es "web-free", also ob es öffentlich gezeigt werden kann, ist. Natürlich ist dies nicht für alle Skripte erlaubt; stattdessen wird, wenn die **Short URL to this script...** Methode ausgeführt wird, eine Markierung gesetzt, die den Skript "web-free" macht. Alle anderen Skripte sind geschützt und werden nicht gezeigt, wenn die zugehörige ID geladen wird. Also kann nur der Ersteller der Skriptes entscheiden, ob er es über eine ID öffentlich verfügbar macht oder nicht. Es sei angemerkt, dass alle Skriptverlinkungen in diesem Dokument auf kurz URLs basieren.

Obgleich praktisch stellen kurz URLs eine Falle, wenn man "entscheidende" Skripte erstellt, zum Beispiel Skripte für Klausuraufgaben, da Skripte mit öffentlichen IDs prinzipiell von jedem betrachtet werden können. Trotz der äußerst geringen Wahrscheinlichkeit, dass ein Student beim "raten" zufälligerweise einen für die Klausur relevanten Skript erwischt, sollten entscheidende Skripte nicht öffentlich gemacht werden indem man kurz URLs für sie anlegt.

Eine öffentliche Skript ID kann in das Skriptfeld von XWizard (nicht VFP) eingegeben werden, um den zugehörigen Skript zu erhalten.

Das Hinzufügen von "&lang=ger" oder "&lang=eng" zu einem der beiden URL-Typen legt die Sprache von XWizardfest auf Deutsch oder Englisch. Die Endung "#Output", "#Codebox", "#ConversionMethods", "#Examples" etc. , die ganz ans Ende der URL gesetzt wird sorgt dafür, dass XWizard sofort an den entsprechenden Teil auf der Webseite springt.

7 PDF Erzeuger und die Konversionsmethode 'Plain PDF generator code'

Jeder XWizard Skript ist mit einem PDF Verarbeiter verbunden, der den Skript in eine PDF umwandelt. Der Übersetzungsprozess wird in die folgenden drei Schritte unterteilt:

(1)	⇒	(2)	⇒	(3)
Script (user-defined)		**PDF processor code** (obtainable by **Plain PDF generator code** method)		**PDF**

Dort ist der PDF Verarbeiter Code reiner Sourcecode in der Sprache des PDF Verarbeiters, das heißt, dass er kopiert und auch außerhalb von XWizard kompiliert werden kann (was XWizard grundsätzlich auch tut, indem es den Code in einer Datei ablegt und zum Beispiel pdflatex.exe oder dot.exe aus Java laufen lässt). Die wichtigsten PDF Verarbeiter, die in XWizard benutzt werden sind LaTeX und Graphviz; insgesamt sind die folgenden bis jetzt eingebunden:

- LaTeX,
- Graphviz,
- GNUPlot (nur in VFP verfügbar; GNUPlot muss dafür installiert sein),
- JavaPDF (eingebunden, bis jetzt noch nicht im aktiven Modus).

XWizardführt im Gegensatz zu VFP einen vierten Schritt: Es konvertiert die PDF-Ausgabe in das SVG-Format, welches direkt in den HTML-Code der XWizard Webseite eingebettet wird.

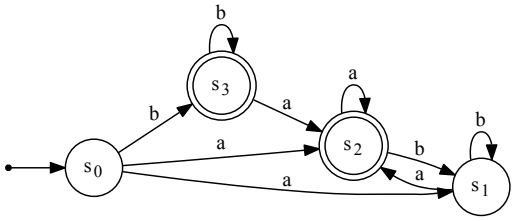
Im Gegensatz zu Studierenden kann es für Lehrende von Interesse sein, den PDF-Verarbeiter-Code öfters zu frisieren – zum Beispiel um gewisse Details in der Darstellung eines Objekts zu ändern, um besonders wichtige Eigenarten für den Kurs hervorzuheben.

So kann es beispielsweise für einen "Basis der Computerwissenschaften" Kurs wünschenswert sein, dass die **nichtdeterministischen Übergänge** in einem EA, dargestellt auf Seite 18, hervorgehoben werden. Jedoch erlaubt es der EA-Skript nur das EA-Objekt in Bezug auf die mathematischen Eigenschaften zu definieren, wohingegen die tatsächliche Abbildung einem internen Algorithmus überlassen wird. Um Zugriff auf den aktuellen Abbildungscode, also den unbearbeiteten PDF-Verarbeitercode, zu erhalten bietet XWizard für jeden PDF-Verarbeiter einen spezifischen Skripttyp, welcher es erlaubt, den Code direkt, wie er von dem PDF-Verarbeiter akzeptiert wird, zu benutzen. Falls dieser Skripttyp verwendet wird, kann reiner LaTeX oder Graphviz (oder GNUplot) Quellcode als Hauptskriptteil eingegeben werden, um es direkt an den PDF-Verarbeiter zu senden.

Es sei angemerkt, dass reine PDF-Verarbeiterskripttypen benutzt werden **können** um den Code an den PDF-Berarbeiter weiterzugeben, aber sie können weit mehr als das. Erstens können Vorprozessor zu dem Code hinzugefügt werden, siehe zum Beispiel Abschnitt 8. Des Weiteren können, wenn man LaTeX PDF verwendet, über die Deklarationen einige Änderungen bezüglich der Interpretationen des Codes erreicht werden, vergleiche "formulaMode=true".

Um aus diesen reinen Skripttypen einen Nutzen ziehen zu können hat jeder XWizard Skript die Konversionsmethode **Plain generator code**. Falls diese verwendet wird wird der Skript, eingebettet in den entsprechenden reinen Skripttyp, in den entsprechenden reinen PDF-Verarbeitercode übersetzt. Nun kann dieser Skript, entsprechend der Sprachregeln dieses PDF-Verarbeiters, wie gewünscht verändert werden.

Diese Vorgehensweise wird am folgenden EA-Skript veranschaulicht. Wie bereits vorgeschlagen kann es wünschenswert sein die nichtdeterministischen Übergänge (die zwei ausgehenden Zustandsübergänge vom Zustand s_0 bezeichnet mit a) in der PDF-Ausgabe hervorzuheben, um ein typisches Merkmal in der Veranstaltung hervorzuheben.



Skript ID-11832

Der normale Skript der den obigen Automaten beschreibt sieht wie folgend aus:

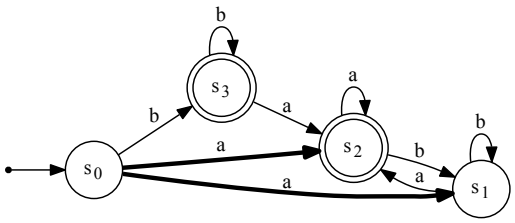
```
fsm:
    (s0, a) | (s1, a) | (s2, a) | (s3, a) => s2;
    (s0, b) | (s3, b) => s3;
    (s1, b) | (s2, b) => s1;
    (s0, a) => s1;
--declarations--
    e=\#n\#;
    simulateToStep=-1;
    input=null;
    s0=s0;
    F=s3,s2
--declarations-end--
```

Offensichtlich gibt es keinen Weg Informationen zum Skript hinzuzufügen, die die Zustandsübergänge hervorheben. Jedoch kann man durch das Ausführen der Methode **Plain generator code** den Skript in unverarbeiteten Graphviz Code umwandeln, welcher wie folgt aussieht (zuerst ohne den in Rot hervorgehobenen Text):

```
dot:
    digraph G {
        rankdir=LR;
        node [shape = point ]; qi
        node [shape = circle];
        s1[label=<s<SUB>1</SUB>>];
        qi -> s0;
        s0[label=<s<SUB>0</SUB>>];
        node [shape = doublecircle];
        s2[label=<s<SUB>2</SUB>>];
        s3[label=<s<SUB>3</SUB>>];
        s3 -> s3 [label="b"];
        s3 -> s2 [label="a"];
        s0 -> s3 [label="b"];
        s0 -> s1 [label="a" ,penwidth=3];
        s0 -> s2 [label="a" ,penwidth=3];
        s1 -> s1 [label="b"];
        s1 -> s2 [label="a"];
        s2 -> s1 [label="b"];
        s2 -> s2 [label="a"];
    }
```

Das Umwandeln eines Skriptes in reinen PDF-Verarbeitercode verändert niemals die PDF-Ausgabe. (Der obere Code wurde jedoch zwecks der besseren Lesbarkeit etwas entschlackt, was die Ausgabe geringfügig verändert.

Basierend auf diesem Skript kann das Hervorheben der nichtdeterministischen Kanten erreicht werden, indem man zum Beispiel den rot gefärbten Text in den reinen Graphviz Code einfügt. Dies führt dazu, dass Graphviz die gewünschten Kanten dicker als die anderen zeichnet. Das Resultat ist wie folgt:



Skript ID-11836

Es können logischerweise alle Arten von Änderungen an diesem Skript vorgenommen werden, solange die Graphviz-Syntax befolgt wird. Das Selbe ist mit Skripten, die auf LaTeXbasieren, möglich. Ein entsprechendes Beispiel wird im nächsten Kapitel gezeigt.

8 Komplexere Objekte: Benutzung von Vorprozessor und Sub-Skripten

Während sowohl Graphviz als auch LaTeX jeweils selbst mächtige Programme sind, macht es manchmal dennoch Sinn diese beiden zu kombinieren, um noch weiter entwickelte Abbildungen entwerfen zu können. Die beiden folgenden Beispiele veranschaulichen die typischen Benutzungsfälle:

- Die Visualisierung der schrittweisen Berechnung eines Kellerautomaten, wie auch dem auf Seite 5, kommt von der Kombination von Graphviz und zwei LaTeX Tabellen. Es wäre sehr schwierig das selbe Ergebnis mit nur einem der beiden Programme zu erzielen.
- Während es sicherlich eine übliche Vorgehensweise ist mittels XWizard PDF-Bilder zu erzeugen und diese in ein externes Dokument zu importieren, so kann es doch praktischer sein, direkt kurze LaTeX Dokumente mit XWizard zu erstellen, die sowohl Text als auch graphische Teile beinhalten.

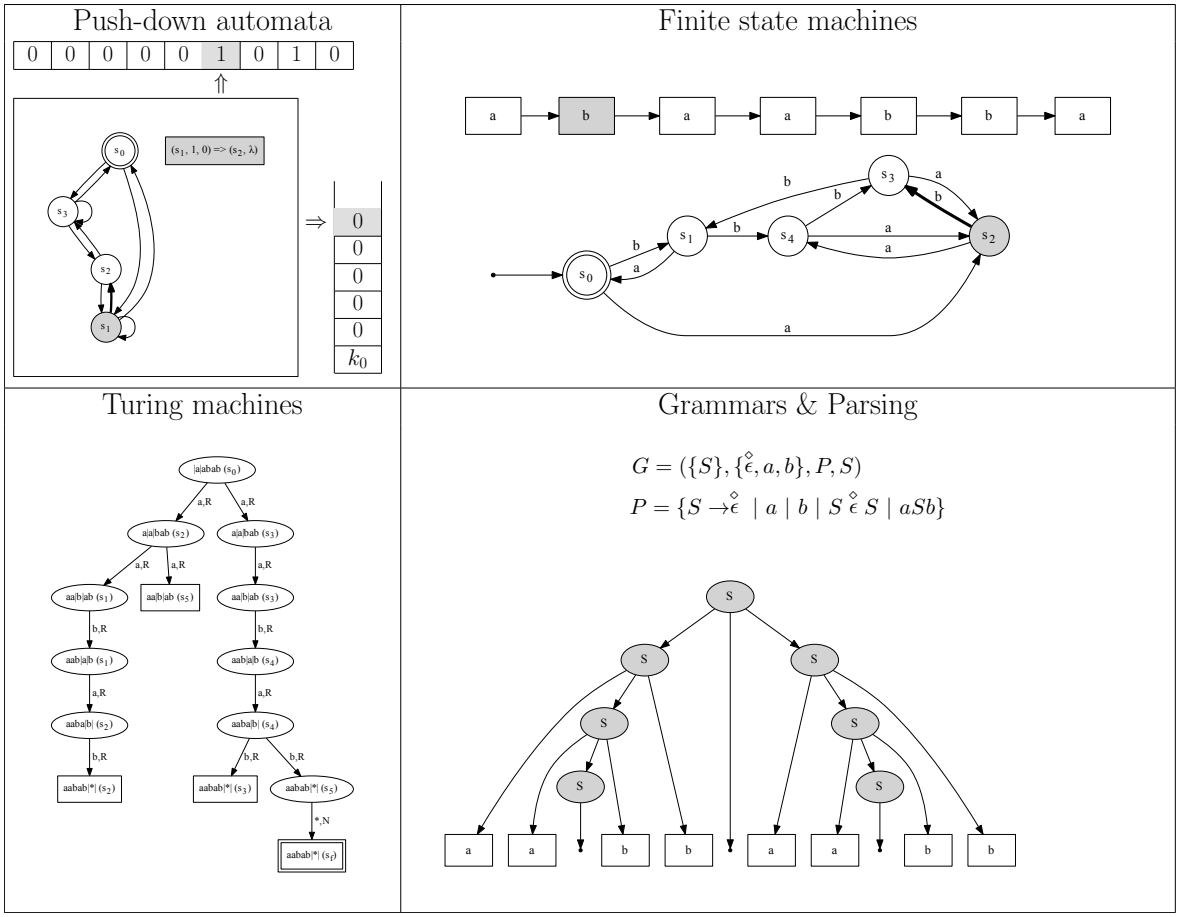
Um diese beiden Anforderungen genüge zu tun, erlaubt XWizard so-genannete "pre-processors" (deutsch: Vorprozessor). Ein Vorprozessor ist ein Skript X, welcher in einem andern Skript Y eingebettet ist. Während der Übersetzung von von Y wird X zuerst in das entsprechende PDF-Bild P_X übersetzt. Da der Vaterskript Y erst anschließend übersetzt wird, kann Y P_X einbinden und kombiniert so den eigenen Inhalt mit der Ausgabe von X.

Für LaTeXist dieser generelle Mechanismus in einer einfach zu nutzenden Struktur eingebunden: dem **Sub-Skript**. Sub-Skripte sind ein Spezialfall der Vorprozessoren, obgleich einfach zu verstehen, und werden deshalb in dem nächsten Abschnitt beschrieben. Der üblichere Vorprozessormechanismus wird anschließend in Abschnitt 8.2 beschrieben.

8.1 Sub-Skripte in LaTeX

Das folgende Bild ist ein komplexes LaTeX Objekt, das vier XWizard Sub-Objekte enthält (ein KA, ein EA, eine Turingmaschine und ein grammatikalisches Objekt).

Some of XWizard's basic object types:



Skript ID-16124

Es kann durch den folgenden XWizard Skript erzeugt werden (der Code des Sub-Objekts, die "Sub-Skripte", sind in Rot hervorgehoben):

```
latex:
    \documentclass[tightpage,preview]{standalone}
    \usepackage{varwidth}\usepackage{amsmath}\usepackage[table]{xcolor}\usepackage{graphicx}\usepackage[space]{grffile}
    \begin{document}
    \huge~\par
    Ein paar der grundelegenden Objekttypen von XWizard :
    \bigbreak
    \begin{tabular}{|c|c|}
    \hline
    Kellerautomaten & Endliche Automaten \\
    @{0.75|
        pda:
            (s1, 1, 0) => (s2, lambda);
            (s3, 1, b) => (s3, b1);
            (s3, 0, 1) => (s3, b);
            (s3, 0, b) => (s3, lambda);
            (s1, 0, 0) => (s1, 00);
            (s3, 1, 1) => (s3, 11);
            (s3, lambda, k) => (s0, k);
            (s1, lambda, k) => (s0, k);
            (s2, lambda, k) => (s3, bk);
            (s0, 1, k) => (s3, 1k);
            (s0, 0, k) => (s1, 0k);
            (s2, lambda, 0) => (s1, lambda);
        --declarations--
            e=#n#;
            s0=s0;
            F=s0;
            kSymb=k;
            inputs=000001010;
            simSteps=5
        --declarations-end--
    }@ &
    @{0.7|
        fsm:
            (s0, a) | (s3, a) | (s4, a) => s2;
            (s0, b) | (s3, b) => s1;
            (s1, a) => s0;
            (s1, b) | (s2, a) => s4;
            (s2, b) | (s4, b) => s3;
        --declarations--
            e=#n#;
            simulateToStep=1;
            input=abaabba;
            s0=s0;
            F=s0
        --declarations-end--
    }@ \\
    \hline
    Turingmaschinen & Grammatiken \& Syntaxbäume \\
    @{0.5|
        turing:
            (s0, a) => (s2, a, R) | (s3, a, R);
            (s0, b) => (s1, b, R) | (s4, b, R);
            (s1, a) => (s2, a, R);
            (s1, b) => (s1, b, R);
            (s2, a) => (s1, a, R) | (s5, a, R);
            (s2, b) => (s2, b, R);
            (s3, a) => (s3, a, R);
            (s3, b) => (s4, b, R);
            (s4, a) => (s4, a, R);
            (s4, b) => (s3, b, R) | (s5, b, R);
            (s5, *) => (sf, *, N);
        --declarations--
            s0=s0;
            F=sf;
            blank=*;
            inputs=aabab;
            runStepsScript=120;
            shortTrace=false
        --declarations-end--
    }@ &
    @{1.5|
        grammar parse(a, a, <>, b, b, <>, a, a, <>, b, b)--48:
            S => a, S, b | <> | S, <>, S | a | b;
        --declarations--
            N=S,A;
            T=a,b,c;
            S=S;
        --declarations-end--
    }@ \\
    \hline
    \end{tabular}
    \end{document}
```

Das Skript basiert auf dem LaTeX Skripttyp (verlgeiche Abschnitt 7) welcher über dem grundlegenden LaTeX Code hinaus die Möglichkeit für das Einbetten von Sub-Skripten bietet. Ein Sub-Skript ist ein

beliebiger XWizard Skript, welcher von der folgenden Klammerkonstruktion eingeschlossen wird:

```
\OB *Beispielskript* \CB
```

Während der Übersetzung werden Sub-Skripte als Vorprozessoren behandelt, diese werden zuerst übersetzt und in PDF-Dateien abgelegt, so wie `file*X*.pdf` für den Sub-Skript `X`. Anschließend wird jeder Sub-Skript `X` in dem LaTeX Code durch den folgenden Code ersetzt:

```
\includegraphics{file*X*.pdf}
```

Infolge dessen wird der nächste Durchlauf von LaTeX die vorkompilierten PDFs an den Positionen der ursprünglichen Sub-Skripte einbinden, also wird die Ausgabe der Sub-Skripte in die Ausgabe des Gesamtskriptes eingegliedert.

Subskripte können naheliegenderweise nur an Positionen in LaTeX plaziert werden an denen `includegraphics` zugelassen ist. Deshalb müssen folgende Importe in der Präambel des LaTeX-Dokuments aufgeführt werden (zukünftig werden diese Importe automatisch beinhaltet sein, bis jetzt aber noch nicht!):

```
\usepackage{graphicx}       % Stellt das includegraphics Makro zur Verfügung.
\usepackage[space]{grffile} % Erlaubt Leerzeichen in der Pfadangabe der
Grafikdateien.
```

Falls die eingebettete Grafikgröße nicht passt, kann der Sub-Skript Code durch einen Parameter skaliert werden, wie zum Beispiel (Beispielhafte Skalierung auf das 0.5-fache.):

```
\OB0.5| *Beispielskript* \CB
```

Was zu dem LaTeX Code

```
\includegraphics[scale=0.5]{file*X*.pdf}
```

führt, welcher wiederum in LaTeX bewirkt, dass das Bild nur halb so groß wie das Original ist. Ein negativer Faktor -0.5 hat

```
\includegraphics[width=0.5\linewidth]{file*X*.pdf}
```

zur Folge, also wird die Bildbreite in Relation zu der aktuellen Zeilenlänge im Dokument angepasst (es gibt keinen besonderen Grund dafür, dass diese Eigenschaft über negative Zahlen eingebunden wurde; es handelt sich lediglich um eine einfache Möglichkeit für diese Funktionalität).

Sub-Skripte verkörpern ein mächtiges Werkzeug, das es erlaubt komplexe Dokumente vergleichsweise einfach zu erstellen. Anzumerken ist, dass ein Sub-Skript selbst ein reiner LaTeX Skript sein kann, welcher wiederum eigene Sub-Sub-Skripte enthalten darf, usw.. Beispielsweise enthält der Skript zu Beginn dieses Abschnitts einen KA-Skript, dieser ist ein Sub-Skript, der in ein LaTeX -Skript mit einem eigenen Sub-Sub-Skript übersetzt wird.

Besonders anzumerken ist im Kontext der Sub-Skripte, dass die Konversionsmethode “Plain generator code” auf eine etwas andere Art, als beschrieben arbeitet. Das einmalige Ausführen zeigt den reinen LaTeX -Skript mit Sub-Skriptnotation an; das zweimalige Ausführen zeigt den aktuellen reinen LaTeX -Code an, bei dem die Sub-Skripte durch die “includegraphics” Befehle ersetzt wurden.

8.2 Vorprozessoren

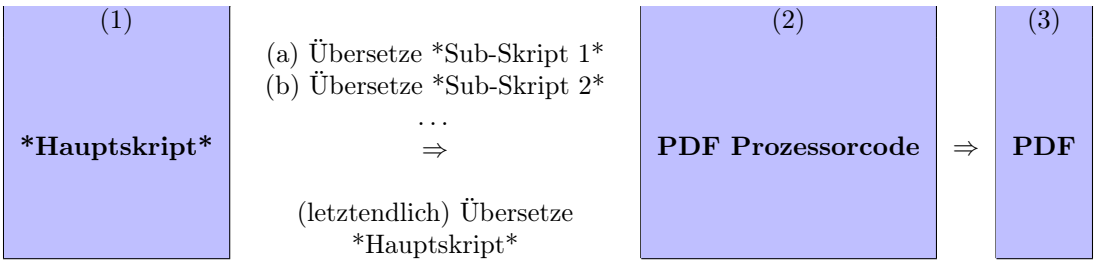
Sub-Skripte wurden auf Basis der allgemeineren Mechanismen der emphVorprozessoren implementiert, welche mit jedem Skripttyp (nicht nur reinem LaTeX) benutzt werden können. Bei der Entstehung dieses Dokuments jedoch war die einzige Anwendung der Vorprozessoren die oben beschriebene Sub-Skripteinbindung. Der restliche Abschnitt wurde für den interessierten Leser verfasst und zeigt zukünftig mögliche Anwendungen auf.

Jeder XWizard Skript kann von Vorprozessoren auf die folgende Art und Weise profitieren:

```
*Main script*
    ...
    * Import filename1.pdf *
    ...
    * Import filename2.pdf *
    ...
--Declarations--
    preprocessor1 = [~(~{filename1.pdf|*sub-script 1*}~)~];
    preprocessor2 = [~(~{filename2.pdf|*sub-script 2*}~)~];
    ...
--Declarations-end--
```

Dabei können im Deklarationsteil eine beliebige Anzahl an Vorprozessoren definiert werden (die entsprechend der Variablennamen mit dem Wort `preprocessor` und beliebiger Endung benannt werden). Der Vorprozessorcode beginnt mir einen Dateinamen, in dem die PDF-Ausgabe abgelegt wird. Dahinter durch ein `|` Symbol abgetrennt folgt der eigentliche Skriptcode, der ein beliebiger XWizard Skript sein kann. Es ist anzumerken, dass dieser Code selbst wieder weitere Vorprozessoren in dem eigenen Deklarationsteil enthalten kann. Die Deklaration eines Vorprozessors sollte immer in der **sicheren Klammerumgebung** eingeschlossen sein (vergleiche Abschnitt 5) um die Korrekte Interpretation `[~(~{ *Vorprozessorcode* }~)~]` gewährleisten zu können.

Der Übersetzungsprozess eines Skripts mit Vorprozessoren funktioniert folgendermaßen:



Bei der Übersetzung eines Skripts mit Vorprozessoren werden zuerst die Vorprozessoren übersetzt (was wiederum bedeutet, dass deren Sub-Vorprozessoren, falls vorhanden, wieder vor dem Hauptteil übersetzt werden – also "die tiefsten zuerst"). Nachdem alle Vorprozessoren der verschiedenen Niveaus übersetzt wurden, wird der Hauptskript übersetzt, dabei kann dieser davon ausgehen, dass alle vorhergehenden PDFs existieren. Also kann der Skripthauptteil Code enthalten, der die PDF-Dateien der Vorprozessorcodes einbindet. Das Ergebnis ist eine PDF-Datei, die Teilbilder, die aus beliebigen XWizard Skripten generiert wurden, enthält.

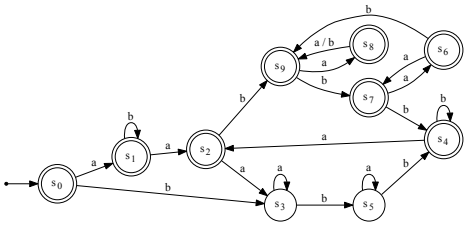
Wie bereits erwähnt ist die derzeit einzige Anwendung der Vorprozessoren im Moment die Implementierung von Sub-Skripten ins LaTeXÄllerdings kann man sich viele interessante Vorgänge überlegen, die in Zukunft mit diesem Mechanismus eingebunden werden können.

8.3 Vorimplementierte Beispiele mit komplexen Objekten

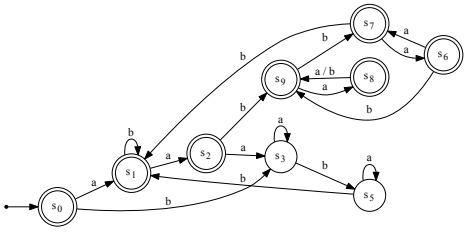
Der oben beschriebene Subskriptmechanismus ist die Grundlage für einige vorimplementierte Skripttypen. Diese können inspiziert werden, um den Umgang mit Subskripten zu lernen.

Beispielsweise können EA Skripte benutzt werden um **simultan** (1) einen EA, (2) eine minimierte Version des selbigen und (3) die zugehörige Minimierungstabelle in einer einzigen PDF Ausgabe anzuzeigen. (Siehe Abbildung unten; diese Ansicht ist sehr praktisch, wenn Klausuraufgaben erstellt werden, da der Schwierigkeitsgrad der Aufgabe schneller eingeschätzt werden kann, da die zusätzlichen Informationen automatisch angezeigt werden, während die usrprüngliche EA Definition eingegeben wird.

FSM:



Minimized:



SKRIPT ID-13363

Minimization table:

s_1	$\times_1$							
s_2	$\times_1$	$\times_1$						
s_3	$\times_0$	$\times_0$	$\times_0$					
s_4	$\times_1$	–	$\times_1$	$\times_0$				
s_5	$\times_0$	$\times_0$	$\times_0$	$\times_1$	$\times_0$			
s_6	$\times_1$	$\times_2$	$\times_1$	$\times_0$	$\times_2$	$\times_0$		
s_7	$\times_1$	$\times_2$	$\times_1$	$\times_0$	$\times_2$	$\times_0$	$\times_3$	
s_8	$\times_1$	$\times_2$	$\times_1$	$\times_0$	$\times_2$	$\times_0$	$\times_4$	$\times_3$
s_9	$\times_1$	$\times_2$	$\times_1$	$\times_0$	$\times_2$	$\times_0$	$\times_4$	$\times_3$	$\times_4$
	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8

Diese Abbildung wurde durch den nachfolgenden EA-Skript erzeugt. Der Hauptteil definiert nur den ursprünglichen EA, sowohl der minimierte EA als auch die Minimierungstabelle werden nebenher erzeugt. Die Variablendeklarationen “displayMode=1” und “showMinimizedFSM=true” geben an, dass diese Bestandteile angezeigt werden sollen.

```
fsm:
    (s0, a) | (s1, b) => s1;
    (s0, b) | (s2, a) | (s3, a) => s3;
    (s1, a) | (s4, a) => s2;
    (s2, b) | (s6, b) | (s8, a) | (s8, b) => s9;
    (s3, b) | (s5, a) => s5;
    (s4, b) | (s5, b) | (s7, b) => s4;
    (s6, a) | (s9, b) => s7;
    (s7, a) => s6;
    (s9, a) => s8;
--declarations--
    e=#n#;
    simulateToStep=-1;          /* -1 bedeutet,dass der EA nicht simuliert wird. */
    input=null;
    s0=s0;
    F=s4,s6,s7,s8,s9,s0,s1,s2;
    displayMode=1;              /* Zeige die Minimierungstabelle. */
    showMinimizedFSM=true;      /* Zeige den minimierten EA. */
    showDeterministicFSM=false;
--declarations-end--
```

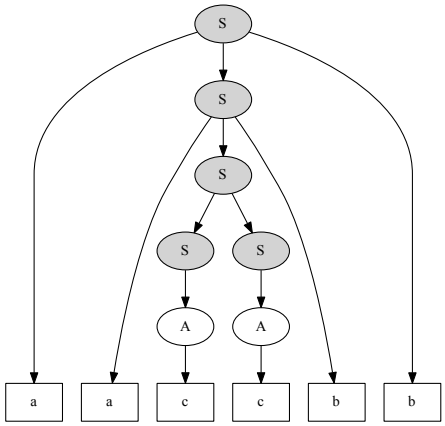
Beim Ausführen der Konversionsmethode “Plain generator code” kann der zugehörige reine LaTeX-Skript angezeigt werden (dieser Skript wird, wie die anderen reinen LaTeXSkripte, in diesem Abschnitt aufgrund ihrer Größe nicht aufgeführt). Der Skripthauptteil enthält LaTeXCode welcher die Überschriften und die Minimierungstabelle enthält (der lange Code in der Präambel definiert ein Makro für dreidimensionale Tabellen). Des Weiteren sind zwei Sub-Skripte vom reinem Typ Graphviz in dem LaTeXCode eingebaut; sie definieren entsprechend die Graphen für den ursprünglichen und den minimierten EA.

Ein weiteres Beispiel sind KA-Skripte, welche wie oben bereits erwähnt dazu benutzt werden können ein verzweigtes PDF-Bild zu erzeugen, welches aus zwei LaTeX -Tabellen und einem Graphviz Graphen besteht; ein entsprechender Skript und eine Abbildung werden zu Beginn des Abschnitts 3 gezeigt, alternativ ist sie unter dem folgenden Skriptlink abrufbar (wieder kann der zugehörige reine LaTeX -Skript über die Konversionsmethode "**Plain generator code**" erhalten werden):

Skript ID-4135

Als letztes Beispiel einer realen Anwendung können auch Grammatikskripte vielschichtige PDF-Objekte erzeugen. Dabei ist ein reiner Graphviz-Skript (zeigt einen Syntaxbaum oder verschiedene Darstellungen der Grammatik) in einen LaTeX -Code eingebunden und veranschaulicht die Grammatikdefinition:

$$G = (\{A, S\}, \{a, b, c\}, P, S)$$
$$P = \{S \to A \mid SS \mid aSb,$$
$$A \to c \mid AA\}$$



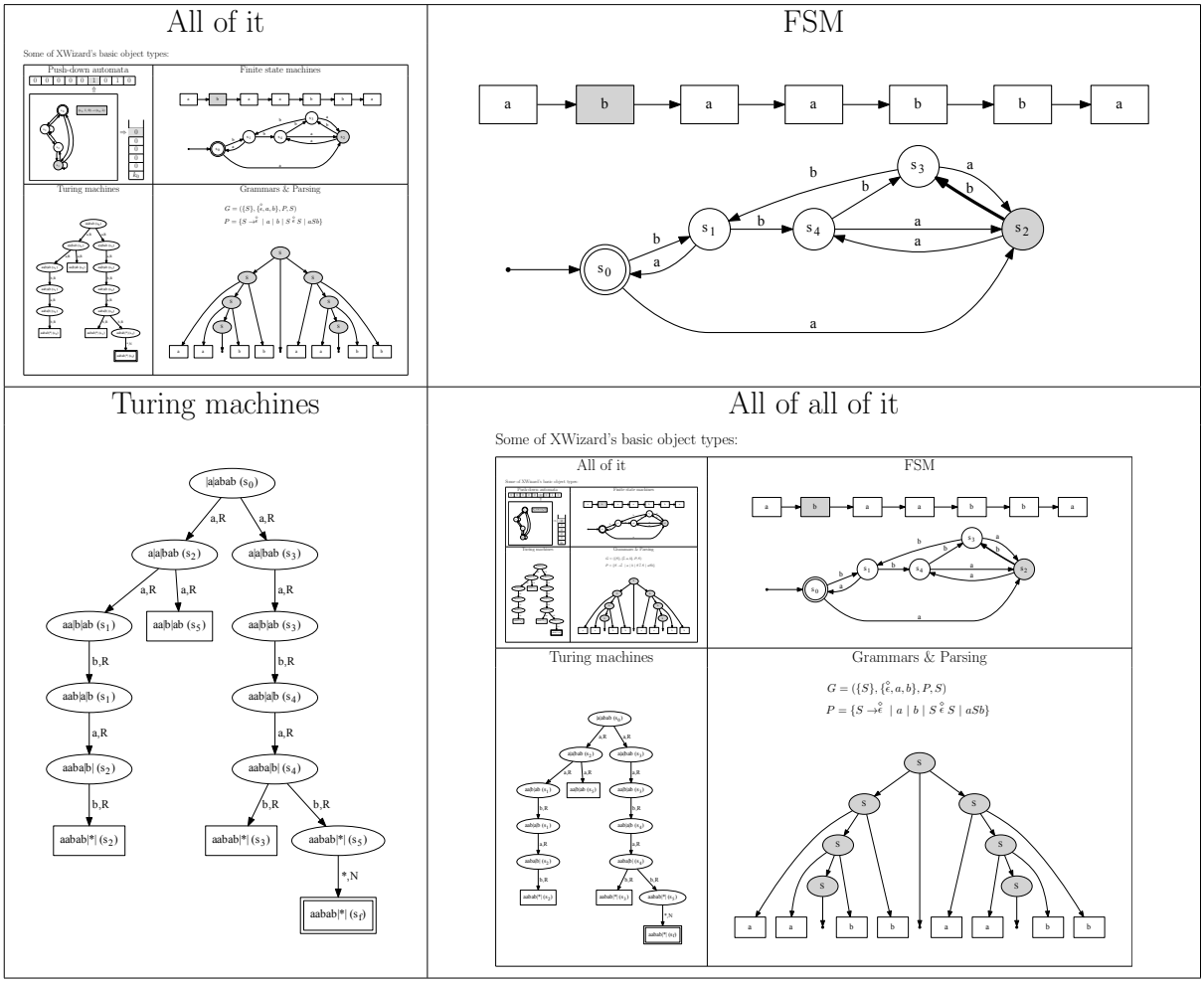
Skript ID-11087

Der Skript, der diese Abbildung erzeugt wird im Folgenden aufgeführt, der zugehörige reine LaTeX -Skript kann wie vorhin über "Plain generator code" erhalten werden.

```
grammar parse(a, a, <>, b, b, <>, a, a, <>, b, b)--48:
    S => a, S, b | <> | S, <>, S | a | b;
--declarations--
    N=S,A;
    T=a,b,c;
    S=S;
--declarations-end--
```

Das folgende komplexe und eher künstliche Beispiel zeigt, wie Skripte rekursiv bis in eine beliebige Tiefe ineinander verschachtelt werden können:

Some of XWizard's basic object types:



Skript ID-C16107

Die Übersetzung des vorigen Skriptes und die Berechnung des zugehörigen komplexen Objekts kann zwischen 10 und 20 Sekunden dauern. Deshalb kann die zwischengelagerte (engl. cached) Version des Objekts erhalten werden, wenn man obigem Link folgt. Dies verhindert eine erneute Berechnung – daher das "C" in der ID, vergleiche Abschnitt 10.2.

9 Einfache Animationen

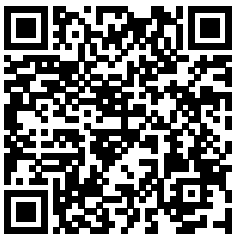
XWizard erlaubt es Animationen mittels Eigenschaften des SVG Bildformats zu definieren.

Animationen werden also nur in der SVG Ausgabe eines Objektes angezeigt, zum Beispiel insbesondere dann, wenn man die XWizard Webseite oder den Webservice benutzt, vergleiche Abschnitt 10.3. Allerdings beinhaltet die aktuelle Version des VFP ebenfalls einen SVG Verarbeiter (noch im Betastadium), um Animationen wiederzugeben. Dagegen sind Animationen in reinem PDF noch nicht verfügbar; alle Teilobjekte die für das Erstellen einer Animation benötigt werden, können jedoch als separate, nicht veränderbare PDF Datei heruntergeladen werden.

XWizard benutzt die `set` Anleitung von SVG, um Animationen zu erstellen und nicht die (schönere) `animate` Anleitung, da die letztere in modernen Browsern (leider) nicht angezeigt wird.

Bis jetzt können Animationen als beliebige Abfolge von XWizard Objekten definiert werden, die nacheinander angezeigt werden, wenn der Benutzer auf das Bild klickt, siehe auch folgendes Beispiel:

Skript ID-C21966



Während die Animationserstellung auf einem relativ komplexen Vorprozessormechanismus aufgebaut ist, welcher großartige Möglichkeiten für verschiedene Animationstypen bietet (diese werden im Detail im nächsten Abschnitt erklärt), ist es sehr einfach grundlegende Animationen zu erstellen.

9.1 Definition von grundlegenden Animationen per Skript

Die allgemeine Idee hinter Animationen ist es Objektkennungen, wie zum Beispiel `x`, `y`, `z` XWizard Objekten zuzuordnen. Anschließend kann eine Animationsabfolge `x->y->z` im Deklarationsteil wie folgt definiert werden:

```
animate=x->y->z;
```

Das erste Objekt in der Abfolge `x` wird angezeigt, sobald das Skript geladen wird. Beim Anklicken des Bildes wird nun `x` durch `y` ersetzt, ein erneutes Anklicken ersetzt wiederum `y` durch `z`. Es bleibt die Frage, wie die Objektkennungen den jeweiligen Objekten zugeordnet werden können. Prinzipiell gibt es zwei Arten von Objekten in XWizard:

- Diejenigen, die implizit durch den aktuellen Skript gegeben sind; die vordefinierte Objektkennung `this` bezieht sich auf das vom Objekt gegebene aktuelle Skript.
- Diejenigen, die durch einen **Vorprozessor** im aktuellen Skript gegeben sind (vergleiche Abschnitte 8 und 10.1). Um den als Vorprozessor gegebenen Objekten eine Objektkennung zuzuweisen muss der Name vor dem Vorprozessorcode eingegeben werden und durch ein "Gleichheits-" Zeichen separiert werden. Dort kann eine Objektkennung ein alphanumerischer String (allerdings nicht `this`) sein. Zum Beispiel sind die folgenden Zuweisungen möglich:
  - Im Falle von Subskripten (die überall in einem Skript eingebaut werden können – was viel Verrücktes zulässt, vergleiche Abschnitt 10):

    ```
    x0=\OB *Beliebiger Skript* \CB
    ```

  - Im Falle von üblichen Vorprozessordefinitionen (im Deklarationsteil):

    ```
    preprocessor=\#x1=\OB *Beliebiger Skript* \CB\#
    ```

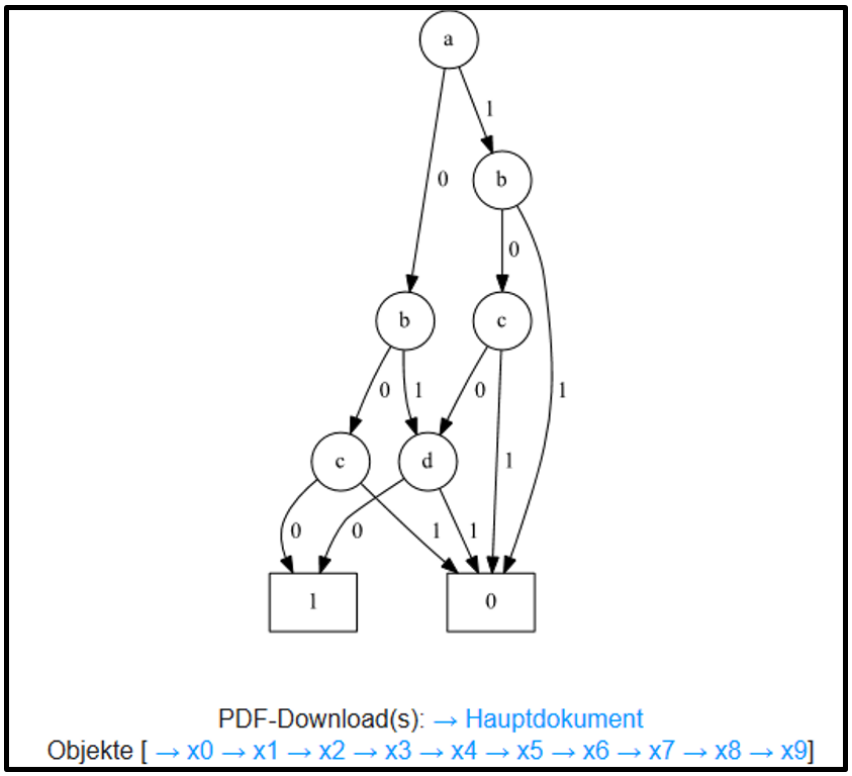
Als Beispiel für die übliche Vorprozessordefinition das folgende Skript:

```
bdd:1000101010
--declarations--
preprocessor=\#x0=\OB bdd:0000101010\CB\#;
preprocessor=\#x1=\OB bdd:1100101010\CB\#;
preprocessor=\#x2=\OB bdd:1010101010\CB\#;
preprocessor=\#x3=\OB bdd:1001101010\CB\#;
preprocessor=\#x4=\OB bdd:1000001010\CB\#;
preprocessor=\#x5=\OB bdd:1000111010\CB\#;
preprocessor=\#x6=\OB bdd:1000100010\CB\#;
preprocessor=\#x7=\OB bdd:1000101110\CB\#;
preprocessor=\#x8=\OB bdd:1000101000\CB\#;
preprocessor=\#x9=\OB bdd:1000101011\CB\#;
animate=\#this->x0->x1->x2->x3->x4->x5->x6->x7->x8->x9\#;
--declarations-end--
```

Dieses legt die folgende Animation an, die 11 separate Bilder von "BDD" Objekten enthält:

SKRIPT ID-21974

Dort bietet der Abschnitt zum Herunterladen nicht nur einen Link zur PDF des Hauptdokuments (z.B. `this`), sondern auch Verlinkungen zu PDF Dokumenten aller Subobjekte, wie zum Beispiel `x0`, ..., `x9`, in diesem Fall:



Zusätzlich zu den explizit benannten Objekten und `this` kann die Animation Referenzverlinkungen zu individuellen Seiten mehrseitiger Dokumente enthalten. Beispielsweise kann, falls ein LaTeX Skript ein PDF Dokument mit fünf Seiten anlegt, eine Animation aus diesen Seiten auf die folgende Art und Weise erstellt werden:

```
animate=#page1->page2->page3->page4->page5#;
```

Offensichtlich sind XWizard Skripte ohne Animationen Spezialfälle derjenigen mit Animationen, bei denen die `animate` Variable einfach auf den folgenden Wert gesetzt wurde:

```
animate=this;
```

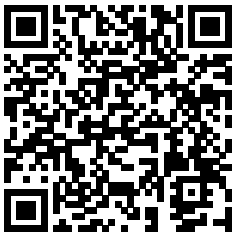
9.2 Konversionsmethoden zur Animationserstellung

Konversionsmethoden zur Erstellung von Standartanimationen werden von verschiedenen Skripttypen angeboten. Für endliche Automaten oder Kellerautomaten kann beispielsweise eine Animationsabfolge ihrer **Simulation bis zum Schluss** (vgl. erstes Beispiel in diesem Abschnitt; Skript ID-C21966) erzeugt werden, indem man den entsprechenden Konversionsknopf `Animate FSM simulation` oder `Animate PDA simulation` anklickt. Ein ähnlicher Knopf ist für BDDs verfügbar. Diese Methoden fügen den folgenden Code in den Deklarationsteil des zugehörigen Skriptes ein (im EA Fall; in anderen Fällen einen sehr ähnlichen Code):

```
prep0=\#x0=this.sim\#;
\OB prepA=\#xA=x~{A-1}~.sim\#;\CB.for[A, 1, x0.inputLength]
animate=this\OB->xB\CB.for[B, 0, x0.inputLength];
```

Ein Beispiel der Auswirkungen dieses Codes kann unter

Skript ID-22384



beobachtet werden. Die Semantik dieses Codes wird in Abschnitt 10.1 erläutert. Dennoch ist es wichtig hier kurz anzumerken, dass dieser Code für alle EAs und jegliche Eingabestrings simulierbar ist. Das bedeutet, dass solange dieser Code präsent ist, das umgebende EA Skript wie gewünscht verändert werden kann und die daraus resultierende Animation automatisch angepasst wird und alle Simulationsschritte dieses EAs bis zur Terminierung des gegebenen Eingabestrings enthält. Anders ausgedrückt ist das Einfügen dieser drei Codezeilen in das gegeben Skript, das Einzige das "Animate..." Konversionsmethoden tun. Das Selbe kann erreicht werden, indem man diesen Code von dieser Seite kopiert und per Hand in ein Skript einfügt. Jegliche Animationsmagie wird durch den Vorprozessormechanismus von XWizardausgeführt.

Es mag sich zwar anhören wie eine bedeutungslose technische Feinheit – ist es aber nicht! Ein wichtiger Vorteil den Vorprozessoren die ganze Arbeit zu überlassen liegt darin, dass alle vorstellbaren Animationen (jedenfalls die meisten) erzeugt werden können, indem man lediglich die XWizard Skripte ändert. Es ist nicht von Nöten in die Javaquellen einzusteigen, zu kompiliern, zu entfalten usw.

Bis jetzt gibt es nur Konversionsmethoden für die Standartanimationen, allerdings kommen bald mehr dazu – und vor allem auch verschiedene Typen von Animationen!

10 Erweiterte Nutzungsmöglichkeiten: Coole Kniffe und verrückte Hacks für den geschickten Nutzer

Die Funktionsweise, die in diesem Kapitel beschrieben wird, ist eigentlich nicht so verrückt, allerdings handelt es sich dabei um die Art von Dingen, die von einem Entwickler verpackt werden, um dem Normalnutzer als nett anzusehende Bausteine zu dienen. Dennoch kann alles aus diesem Dokument auf dem normalen XWizard Skriptlevel ausgeführt werden. Es besteht keine Notwendigkeit sich in die Tiefen von Java zu begeben. Folglich wird jeglicher hier aufgeführte Stoff als Standard XWizard Funktionalität betrachtet – im Gegensatz zu den Inhalten des "XWizard Handbuch für Entwickler". Lassen Sie sich also nicht abschrecken und lesen Sie weiter!

10.1 Die XWizard Skriptsprache 2.0

Die bis jetzt beschriebene Skriptsprache kann – rein chronologisch – als die erste Version der XWizard Skriptsprache betrachtet werden. Sie beinhaltet bereits grundlegende Unterstützung für sogenannte "Vorprozessoren" (vgl. Abschnitt 8.2), allerdings nur als Behelfsmöglichkeit, um das Bild eines XWizard Objekts in ein anderes XWizard Objekt einzubinden. Des Weiteren beinhaltet sie die Möglichkeit die Ausführung einer Konversionsmethode innerhalb des Skripts, auf dem sie ausgeführt werden soll, zu verschlüsseln – allerdings in einer relativ plumpen, eher weniger verallgemeinerbaren Art und Weise.

Die XWizard Skriptsprache 2.0 vereint diese zwei Eigenschaften, indem sie Konversionsmethoden erlaubt auf Subskripten innerhalb eines anderen Skriptes ausgeführt zu werden. Weiterhin macht es Sinn, dass Konversionsmethoden an beliebigen Teilen eines Skriptes (nicht nur Teilen, die ein gewöhnliches XWizard Objekt codieren) ausgeführt werden können, da Konversionsmethoden ein anderes Skript oder reinen Text als Ergebnis ausgeben können (vgl. Abschnitt. **??**). Dies erlaubt die Einbindung von Programmierstrukturen, wie zum Beispiel "for Schleifen", in Skripten. Werfen wir nun einen Blick darauf, wie beispielsweise eine "for Schleife" mittels rein textueller Konversionsmethoden realisiert werden können.

10.1.1 Informelle Beispiele: Die `for` Schleife und die `if` Bedingung

Der folgende Codeschnipsel ist ein funktionierendes EA Skript:

```
fsm: (s0, a) => @{sX | }@.for[X, 1, 4] s5;
```

Wenn man den "`for`" Teil (also `@{sX | }@.for[X, 1, 4]`) und die Präambel weglässt, besteht der verbleibende String `(s0, a) => s5;` nur noch aus dem reinen klassischen XWizard Code. Dieser bewirkt einen Übergang von Zustand s_0 zu Zustand s_5, wenn auf dem Eingabeband ein a gelesen wird. Der Teil `@{sX | }@` sieht wie ein Subskriptcode aus (vgl. Abschnitt. 8.2), allerdings ist es kein valider Skript.

Technisch gesehen ist es das – da intern diese reinen Textteile als Skripte eines speziellen Typs mit Namen codeDummyRepresentable behandelt werden. Diese Feinheit kann hier aber außer Acht gelassen werden.

Jedoch ist `for` der Name einer **Klartextkonversionsmethode**, welche auf einen beliebigen String angewendet werden kann. Der String, hier "`sX | `", wird der **Rumpf** der Methode genannt. Der Hauptteil steht innerhalb der Klammerung `@{ *Rumpf* }@` und die angewendete Methode wird auf eine Java-ähnliche Schreibweise durch einen Punkt, gefolgt von dem Methodennamen und anschließend einer Liste von Parametern in eckigen Klammern (genau wie in dem Fall der in Skripten enthaltenen Konversionsmethoden, vgl. Abschnitt. **??**) angegeben.

Wie in Java kann eine Abfolge von Methoden folgendermaßen auf ein Objekt angewendet werden:

`@{ *Rumpf* }@.method1[...].method2[...].method3[...]...`

. Weiterhin können alle normalen Konversionsmethoden dieses Skripttyps auf den `*Rumpf*` angewendet werden, falls dieser ein normales XWizard Skript darstellt. So kann beispielsweise die Determinierungsmethode, ebenso wie anschließend die Minimierungsmethode, auf ein EA Skript angewendet werden. Das sieht wie folgt aus:

`@{*einige Zeilen EA Skript* }@.det.min`

Es sei angemerkt, dass das, was wir zuvor als "Subskripte" bezeichnet haben, lediglich ein Spezialfall dieses Syntax mit einer leeren Abfolge von Methoden ist.

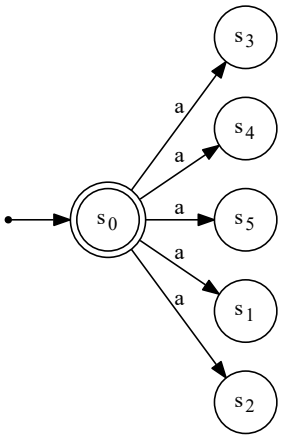
Also ist die Bedeutung von `@{sX | }@.for[X, 1, 4]` mehr oder weniger: Wende auf das durch "`sX | `" gegebene Objekt die Konversionsmethode `for` mit den Parametern `X, 1, 4` an und ersetze `@{sX | }@.for[X, 1, 4]` durch die von der Methode zurückgegebenen Ergebnisse. Im Fall des `for` ist das Resultat dadurch gegeben, dass der String innerhalb der Klammern viermal kopiert wird, da die Schleifenvariable `X` von `1` bis `4` läuft und im String alle aufgeführten `X` durch den aktuellen Wert der Variablen ersetzt werden. In anderen Worten "entwickelt" sich der String (wir übernehmen diesen Ausdruck von TEX, obwohl es nicht der selbe ist) zu:

```
s1 | s2 | s3 | s4 |
```

und der komplette Schnipsel wird nach dem Kompilieren zu:

```
fsm: (s0, a) => s1 | s2 | s3 | s4 | s5;
```

Das resultierende Objekt sieht also ungefähr so aus:



Skript ID-22063

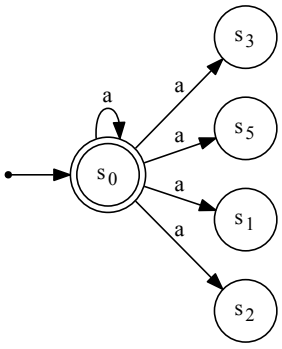
Neben der Benutzung der einfachen Schleifenvariable `X` im String können standardisierte arithmetische Ausdrücke, wie zum Beispiel $\texttt{X} - 1$, $\texttt{X} * 5$ usw., auf ihn angewendet werden, indem man folgendermaßen vorgeht:

```
~{ *Ausdruck* }~
```

Beispielsweise würde sich `fsm: (s0, a) => @{s~{X-1}~ | }@.for[X, 1, 4] s5;` zu

```
fsm: (s0, a) => s0 | s1 | s2 | s3 | s5;
```

entwickeln, woraus sich folgendes Objekt ergäbe:



Skript ID-22064

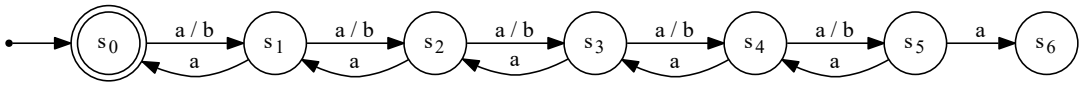
Die Schleifenvariable kann irgendein alphanumerischer String sein und sie kann sogar bestimmte Sonderzeichen, insbesondere "`#`", enthalten. Da jede Aufzählung dieses Strings im umgebenden dazu führt, dass es dort ersetzt wird, ist es eine gute Übung (die von jetzt an auch durchgeführt wird), Variablennamen wie beispielsweise `#X, #Y` zu verwenden, um unerwünschte Ersetzungen zu vermeiden. Es sei angemerkt, dass diese niemals mit den LATEXNotationen `#` oder `##` als Vorsilbenparameter eines Makros kollidieren, da der XWizardCompiler **vor** einem möglichen LATEX Durchlauf greift und alle `#` Zeichen während der Entwicklung entfernt.

Als letztes Beispiel der `for` Schleife betrachte man das folgende komplexere EA Skript: `fsm:(s0,a)=>s1;@{(s#v, a) =`

Dort enthält der Methodenrumpf zwei separate Übergangsdefinitionen, siehe unterstrichene Teile. Einer davon geht von dem Zustand mit Nummer `#v` über in die Zustände mit den Nummern `#v-1` und `#v+1` (siehe doppelt unterstrichene Teile), wenn ein a gelesen wird und der andere, der von dem Zustand mit der Nummer `#v-1` zu dem Zustand mit der Nummer `#v` übergeht, wenn ein b gelesen wird. Da die Variable `#v` von `1` bis `5` läuft, erweitert sich die `for` Methode zu (die Zeilenumbrüche dienen der besseren Lesbarkeit):

```
(s1, a) => s0 | s2;
(s0, b) => s1;
(s2, a) => s1 | s3;
(s1, b) => s2;
(s3, a) => s2 | s4;
(s2, b) => s3;
(s4, a) => s3 | s5;
(s3, b) => s4;
(s5, a) => s4 | s6;
(s4, b) => s5;
```

Das führt zu Übergängen zwischen den mit `0` bis `6` nummerierten Zuständen und das gesamte Skript, inklusive der zweiten Zeile, erzeugt das Objekt:



Skript

Neben `for` Schleifen können bedingte Entscheidungen in Form von `if-then-else` Abfragen für das Erstellen von Skripten von Interesse sein. XWizard stellt eine `if` Konversionsmethode zur Verfügung, die mit folgendem Syntax aufgerufen wird:

```
@{ *Rumpf* }@.if[ *Ausdruck* ]
```

Der Rumpf kann ein beliebiger String sein, der als Ganzes den `*then Fall*` repräsentiert, oder er kann zwei Teile enthalten:

```
@( *then Fall* )@ @( *else Fall* )@
```

. Die `if` Methode springt zu dem `*then Fall*`, wenn der `*expression*` den String `true` ergibt. Andernfalls springt es zum `*else Fall*` welcher, falls nicht explizit aufgeführt, als leerer String definiert ist.

Die typischen Methoden, die in dem `*Ausdruck*` verwendet werden können umfassen:

- `smeq[x, y]`, ergibt `true` $\Leftrightarrow$ `x` $\leq$ `y`.
- `sm[x, y]`, ergibt `true` $\Leftrightarrow$ `x` $<$ `y`.
- `greq[x, y]`, ergibt `true` $\Leftrightarrow$ `x` $\geq$ `y`.
- `gr[x, y]`, ergibt `true` $\Leftrightarrow$ `x` $>$ `y`.
- `eq[x, y]`, ergibt `true` $\Leftrightarrow$ `x` $=$ `y`.
- `neq[x, y]`, ergibt `true` $\Leftrightarrow$ `x` $\neq$ `y`.

Es sei angemerkt, dass diese Methoden (wie alle Methoden) auf ein Objekt angewendet werden müssen, das in diesem Fall jedoch irgendein Objekt sein kann, da es keinen Einfluss auf die Entwicklung hat; meist wird `this` den Zweck erfüllen:

```
this.smeq[x, y]
```

Ein Beispiel einer sehr einfachen `IF` Bedingung ist:

```
@{Ja, 2 ist kleiner oder gleich 3}@.if[this.smeq[2, 3]]
```

Damit diese Ausdrücke funktionieren, muss es den Konversionsmethodenparametern erlaubt sein, Signale zu anderen (rein textuellen) Konversionsmethoden zu enthalten. So ist beispielsweise:

```
@{ *irgendein Ausdruck* }@.for[#v, 1, this.states]
```

ein gültiger Ausdruck (wobei `this.states` sich zu der Anzahl der Zustände ausdehnt, zum Beispiel die des aktuellen EA). Dies kann allerdings zu unerwarteten Ergebnissen führen, wenn die Schleife selbst in der Bearbeitung des "`this`" Objekts eingebunden ist, wie es in den drei obigen Beispielen der `for` Methode der Fall ist. Dann können die Zustände, die durch die Schleife erzeugt wurden, offensichtlich nicht durch `this.states` gezählt werden. Die Ausführungsreihenfolge für Subskripte und Vorprozessoren wird im nächsten Abschnitt erklärt (das ist eigentlich der Teil, in dem es etwas verrückt werden kann, da Methoden auf alle Skiptteile angewendet werden können, selbst in den Deklarationen).

10.1.2 XWizard 2.0 Syntax und Semantik

Die XWizard Sprache 2.0 ist ziemlich mächtig, allerdings kann es am Anfang etwas unübersichtlich sein und manchmal kann es zu unerwarteten Ergebnissen, wie dem in der obigen Box beschriebenen, kommen.[2] Im Folgenden wird eine semiformale Übersicht über Syntax und Semantik der XWizard Sprache gegeben, ohne jedoch zu tief ins Detail zu gehen. Es ist geplant im XWizard Entwicklerhandbuch umfassendere Informationen über die Implementierungsdetails zu geben.

Syntax Die neuen Teile der XWizard Sprache, die in diesem Abschnitt beschrieben sind, gründen alle auf dem Konzept der **Vorprozessoren** und **Subskripte** (z.B. **inscript pre-processors**), wie in Abschnitt 8.2 beschrieben.

Genauer gesagt gründen sie auf dem allgemeineren Konzept, das Aufrufe von Vorprozessoren durch Klartextmethoden erlaubt, wie in Beispielen des letzten Abschnitts gezeigt wurde. Die Unterscheidung zwischen "Vorprozessoren" und "Subskripten" wird dann eher der Vergangenheit angehören, da im Kontext von Klartextmethoden ebendiese Begriffe vermischt werden. Von nun an wird der Begriff "Vorprozessor" als allgemeinerer Term verwendet, wohingegen "Subskript" nur noch auf Ausdrücke hinweist, die in der Skriptmitte verwendet werden, siehe auch:

```
@{ *Rumpf* }@.for[#v, 1, this.states]
```

(im Gegensatz zu dem expliziten Definieren von Vorprozessoren im Deklarationsteil mittels einer `prep` Variablen).

Die grundlegende Idee hinter Vorprozessoren ist es, das Erstellen von XWizard Objekten innerhalb anderer XWizard Objekte zu ermöglichen, um auf erstere an verschiedenen Stellen Bezug zu nehmen, oder diese durch Methodenanwendungen zu verändern. Deswegen können Vorprozessoren benannt werden, welche als Objektkennung verwendet werden können, um auf das zugehörige Objekt wie auf eine Variable in üblichen Programmiersprachen zugreifen zu können. Im Falle von Subskripten kann die Objektkennung mit anschließendem Gleichheitszeichen einfach vor den Subskriptcode gesetzt werden. Der allgemeinste Syntax von Subskripten ist:

[2] Dafür gibt es zwei Gründe. Erstens müssen die Konstrukte von XWizard einzigartig sein, damit es keine Überschneidungen mit der Syntax des Codes anderer beinhalteter Programme, wie LaTeX und Graphviz, gibt. Deswegen wurden zum Beispiel Klammerkombinationen mit einem `@` Zeichen gewählt, da diese so gut wie nicht in XWizard fremdem Code vorkommen. Der zweite Grund ist, dass die XWizard Sprache relativ organisch gewachsen ist, also ohne formale Syntax oder Semantik. Dadurch entstand ein hoher Komplexitätsgrad, der nur schwer händelbare Ausmaße angenommen hat – das ist der aktuelle Zustand. Er ist zwar noch pflegbar und es gibt (hoffentlich) keine echten Fehler in der aktuellen Version, allerdings könnten tatsächlich manche nicht–intuitiven Effekte behoben werden, wenn man die klassische Methode des Kompilerbaus verwenden würde. Lukas König versucht gerade etwas freie Zeit zu finden, um dies zu bewerkstelligen ... (Es sei jedoch angemerkt, dass dadurch neben der Freakiness auch etwas Coolness verloren gehen kann...)

$$\underbrace{\texttt{*Objektkennungsname*} =}_{optional}\ \underbrace{\texttt{*Objektreferenz*}}_{*Objektkennungsname* \text{ oder echtes Objekt}}\ \underbrace{\texttt{*Methodenabfolge*}}_{leer \text{ oder } *Methodenname* [p1,p2,...]}$$

Dabei kann `*Objektkennungsname*` ein alphanumerischer String sein (inklusive `this`, welches jedoch **vor** dem Gleichheitszeichen verboten ist). Ein "aktuelles Objekt" ist gegeben durch

```
@{ *Skript* }@
```

, wobei `*Skript*` ein normales XWizard Objekt oder einfach ein reiner String sein kann. Je nach dem tatsächlichen Typ des `*Skriptes*` können die am weitesten links stehende Methoden in der `*Methodenabfolge*` wechseln (beispielsweise wird ein EA Skript nur EA Methoden zum ausführen erlauben); die Ausgabe der am weitesten links stehenden Methode wiederum wird die nächsten verfügbaren Methoden in der Abfolge festlegen usw.. Jede für einen bestimmten Skripttyp verfügbare Konversionsmethode kann durch ihren englischen Namen und das Ersetzen der Leerzeichen durch Gedankenstriche "-" aufgerufen werden. Jedoch sind für die meisten Methoden Abkürzungen verfügbar, wie zum Beispiel `det` für `Determinize`, `min` für `Minimize` und so weiter, siehe auch Abschnitt 10.1.4.

Methodenparameter (falls vorhanden) werden in eckige Klammern hinter dem Methodennamen, getrennt durch Kommata, eingefügt. Ein Parameter kann alles sein, von einer einfachen Konstanten bis zu einem langen Textausschnitt, und kann selbst weitere Methodenaufrufe enthalten. Einfache Parameter können direkt eingefügt werden, wie in:

```
for[#x, 1, 4]
```

. Lange Strings mit Leerzeichen, Kommata und anderen Sonderzeichen, können in Anführungszeichen gesetzt werden, wie in:

```
setLongText[Ëin langer Text, danke", simplePar2]
```

. Noch längere Strings können in die sichere Klammerkombination gesetzt werden:

```
setVeryComplexText[[~(~{strange ]]par {@}}1}~)~], "long par 2", simplePar3]
```

. Wobei `strange ]]par {@}}1` der String ist, der als aktueller Parameterwert interpretiert wird. Eingebundene Methodenaufrufe können einfach irgendwo anders untergebracht werden:

```
method[this.smeq[2, 3], @{fsm:}@.rand[5, true].minimize.states, x.inputLength]
```

, wobei zuvor `x` an beliebiger Stelle als Objektkennung für ein Objekt gesetzt werden müsste.

Neben Subskripten können explizite Vorprozessordefinitionen in dem Deklarationsteil angegeben werden, indem die `prep` Variablen angegeben werden. Der Großteil des oben erklärten Syntax findet dort auch Anwendung. So wäre zum Beispiel auch die folgenden Definitionen gültige Vorprozessordefinitionen:

```
prep1=[~(~{x0=this.sim}~)~];
prep2=[~(~{x1=x0.sim}~)~];
prep3=[~(~{x2=x1.sim}~)~];
prep4=[~(~{x3=x2.sim}~)~];
prep5=[~(~{x4=x3.sim}~)~];
prep6=[~(~{x5=x4.sim}~)~];
```

Das ist soweit alles, was über den XWizard Syntax gesagt werden sollte. Wie bereits erwähnt, gibt es keine formale Grammatikdefinition für alle korrekten XWizard Skripte, so dass diese halbformale Beschreibung für den aktuellen Stand genügen muss (mehr Details können dem Entwicklerhandbuch entnommen werden und natürlich können die exakten Regelungen in den Java Quellen nachgeschlagen werden.[3]

[3]Diese grobe Syntaxbeschreibung ist sicherlich nicht eindeutig. Realistisch betrachtet ist aber die Anzahl der Leser, die bis hier gekommen ist, so gering, dass diese meine große Bewunderung genießen und ich Ihnen daher alle Ihre Fragen gerne per E-Mail beantworte.

Semantik Als letzter Abschnitt folgt eine semiformale Beschreibung der Semantik, die darauf abzielt, einen breiten Überblick über das zu geben, was passiert. Mehr Details werden im Entwicklerhandbuch folgen. Im Grunde ist die Auswertung der Vorprozessoren ein relativ einfacher Prozess. Ein Skript S, das möglicherweise Vorprozessoren, Subskriptteile und Deklarationen enthält, ist Teil der folgenden Abarbeitungskette:

1.) Skriptpräambel von S ausschneiden

2.) Die deklarierten Variablen auf die **vorläufigen** Werte setzen. Bedeutung:

- Alle Variablen auf Standartwerte setzen und
- Die Werte überschreiben, die bereits komplett in dem Deklarationsteil von S gegeben sind. Darunter fallen alle Variablen, die keine `@{...}@` Teile enthalten; insbesondere werden jetzt normale Vorprozessorvariablen evaluiert, falls diese komplett interpretiert werden können.

3.) So lange sich S ändert, führe Folgendes aus:

i.) Ersetze **im gesamten Skript** nur die Subskripte, die sich **zu normalem Text entwickeln**, mit dem entsprechend entwickelten Resultat.

Es sei angemerkt, dass Subskriptteile in Klartext, die **keine Methodenaufrufe** aber **Objektkennungsdefinitionen enthalten**, aus dem Skript geschnitten werden (das heißt, sie entwickeln sich zum leeren String, genauso, wie bei der Verwendung der Methode `nil`). Die Objektkennung kann benutzt werden, um solche Subskripte tatsächlich im Skript auftauchen zu lassen. Wenn das Subskript tatsächlich an der Position seiner Definition auftauchen soll, muss die Methode `id` verwendet werden:

```
@{test}@                /* Entwickelt sich zu 'test'.  */
x=@{test}@            /* Entwickelt sich zum leeren Wort und definiert x als
'test'.  */
x=@{test}@.id         /* Entwickelt sich zu 'test' und definiert x als 'test'.  */
@{x}@                   /* Entwickelt sich zu 'test' mit jeder der obigen
Zeilen.  */
*anything*.nil /* Entwickelt sich immer zum leeren Wort.  */
```

ii.) Prüfe, ob es neue Variablen gibt, die im Deklarationsteil jetzt komplett definiert sind und belege sie wie zuvor.

iii.) Ersetze **alle** Subskripte, **die sich nicht im Deklarationsteil des Skriptes befinden** durch die entsprechenden entwickelten Werte.

In dem Fall, dass kein Klartext vorliegt, entscheidet jedes Skript einzeln, wie die Subskripte entwickelt werden. Bis jetzt sind nur auf LaTeX basierende `\includegraphics` Erweiterungen implementiert, vgl. dazu Abschnitt 8.2. Es sei angemerkt, dass Methoden ohne Klartext nur außerhalb von Deklarationen Sinn machen.

iv.) Prüfe, ob es neue Variablen gibt, die im Deklarationsteil jetzt komplett definiert sind und belege sie wie zuvor.

Es sei angemerkt, dass, wenn über das "Entwickeln" von "Skripten zu Klartext" gesprochen wird, nur Aussagen über **die letzte Methode einer Abfolge** gemacht werden. Beispielsweise bei einer Abfolge wie dieser:

```
@{Skript}@.m1.m2.m3.m4
```

Nur `m4` wird festlegen, ob sich der gesamte Code zu einem Skript oder einem Klartext entwickelt. Alle Methoden dazwischen, also `m1` bis `m3`, können prinzipiell zwischen normalen Skripten und Klartext wechseln.

Nach diesen semiformalen, halb unklaren Beschreibungen sollte die XWizard Semantik ziemlich klar sein. Finden Sie nun in einem Selbsttest heraus, was dieser extra rätselhafte EA Skript mit der Objektkennung `x` und einer Schleifenvariable mit dem Namen `#x` macht:

```
fsm:
x=@{1}@
(s@{x}@, @{x}@) => s@{#x}@.for[#x, x, x.add[x].add[x].add[x]];
--declarations--
s0=s@{x}@.id;
F=s@{x}@.id;
--declarations-end--
```

Die Lösung kann unter

Skript ID-22347

nachgeschlagen werden.

Anzumerken ist, dass `@{x}@` sich entwickelt zu `1`, auch wenn die Methode `id` nicht auf darauf angewendet wird (wie im Beispiel des Deklarationsteils). In früheren Versionen war dies jedoch an bestimmten Positionen im Skript nicht der Fall, sodass dies dadurch behoben werden musste, dass `id` aufgerufen werden musste. `id` gibt den Wert des normalen Texts zurück. auf dem er aufgerufen wurde, erzwingt allerdings die Zuweisung von Objektkennungen, wie im Beispiel zuvor `x`. Dies ist nun nicht mehr notwendig, aber es schadet auch nicht.

10.1.3 Ein fortgeschrittenes Beispiel: Animation bis zur Beendigung

Die XWizard Sprache 2.0 erlaubt es ziemlich komplexe Aussagen auszudrücken, die beispielsweise das Erzeugen von ausgeklügelten Animationen vereinfachen. Um dies zu veranschaulichen betrachten wir noch einmal die Codeschnipsel, die von den "Animate" Befehlen erzeugt wurden und in Abschnitt 9.2 erwähnt wurden. Im Allgemeinen sehen diese Schnipsel für alle verschiedenen Skripttypen, die solche Methoden zur Verfügung stellen, sehr ähnlich aus. Im Falle von EAs, erzeugt die Methode `Animate Simulation` den folgenden Code, der, wenn er in den Deklarationsteil eines Skriptes eingefügt wird, eine Animation des aktuellen EA, basierend auf dem Eingabewort, erstellt (vgl. Skript ID-C22384):

```
prep0=\#x0=this.sim\#;
\OB prepA=\#xA=x~{A-1}~.sim\#;\CB.for[A, 1, x0.inputLength]
animate=this\OB->xB\CB.for[B, 0, x0.inputLength];
```

Der Code kann am Anfang etwas erdrückend wirken, aber beim genaueren Hinsehen kann er mit dem zuvor zur Verfügung gestellten Wissen über XWizard Syntax und Semantik verstanden werden. Zuerst kann man sehen, dass die erste Zeile ein reiner Vorprozessorcode ist, der der Objektkennung `x0` lediglich den aktuellen EA (`this`) zuordnet, auf der ein Schritt simuliert wurde (benutzt wird dafür die Methode `sim`). Also könnte man nur mit diesem Vorprozessor eine Animation des ersten Simulationsschrittes anlegen, indem man die Variable `animate` wie folgt belegt:

```
animate=this->x0;
```

. Nun wollen wir so viele zusätzliche Vorprozessoren `x1`, `x2`, `x3`, ..., `xn` anlegen, damit alle `n` Schritte, die simuliert werden sollen, erfasst werden können. Dabei stellt jeder Vorprozessor den EA dar, der einen Schritt weiter simuliert wurde, als der jeweils vorherige. Da alle (ϵ-freien) EAs so viele Schritte brauchen, wie die Anzahl der Symbole im Eingabewort ($+/-$ eins oder so), müssen die Indizes `i` der Objektkennungen `xi` von `1` bis zur Wortlänge des Eingabewortes laufen. Die Methode `inputLength` gibt diese Größe zurück und eine `for` Schleife kann jetzt dazu verwendet werden, den Code für die benötigten Vorprozessoren anzulegen. Dies wird durch die mittlere Zeile

```
@{prepA=#xA=x~{A-1}~.sim#;}@.for[A, 1, x0.inputLength]
```

bewerkstelligt. Die Schleifenvariable `A` läuft über den gewünschten Bereich, was den String, auf dem es aufgerufen wurde, zu

```
prepA=#xA=x~{A-1}~.sim#;
```

verändert. Dieser entwickelt sich weiter zu:

```
prep1=#x1=x0.sim#;prep2=#x2=x1.sim#;prep3=#x3=x2.sim#;...
```

bis zum Vorprozessor `prepn`, falls `n` die Inputlänge ist. Mit zusätzlichen Zeilenumbrüchen verschönert erhalten wir:

```
prep1=#x1=x0.sim#;
prep2=#x2=x1.sim#;
prep3=#x3=x2.sim#;
  ⋮
```

Wir rufen die Methode `inputLength` auf `x0` und nicht auf `this` auf, weil es möglich ist, dass für `this` keine Eingabe (`input=null`) definiert ist. In diesem Fall müsste die erste Methode in der ersten Zeile der Animation verschieden von den anderen innerhalb der Schleife sein. Statt der parameterfreien Methode `sim` müsste der Vorzug zum Beispiel `sim[*String*]` gegeben werden. Dabei gibt der Parameter `*String*` das Wort an, auf dem simuliert werden soll.

Das ist genau das, was wir erreichen wollten, als wir den ersten Vorprozessor `x0=this.sim` aufgezählt haben. Wir haben nun alle Objekte, um die Animation von Beginn bis zum Schluss anzulegen. Der einzige noch fehlende Code ist der Animationscode selbst:

```
animate=this->x0->x1->x2->...;
```

bis `xn`. Dieser Code wird von der letzten Zeile des Animationscodes erzeugt:

```
animate=this@{->xB}@.for[B, 0, x0.inputLength];
```

Die Schleifenvariable `B` läuft von `0` bis zu `n=x0.inputLength`. Das sorgt dafür, dass sich der innere String, auf dem die Schleife läuft, von

```
->xB
```

zu

```
->x0->x1->x2->...
```

entwickelt, während die Methode angewendet wird. Wenn man den Teil vor der Schleife `animate=this` einschließt und ebenso den Teil danach `;` erhalten wir die gewünschte korrekte Animation:

```
animate=this->x0->x1->x2->...;
```

Insgesamt entwickeln sich die drei Codezeilen zu (Zeilenumbrüche wurden der Lesbarkeit halber eingefügt):

```
prep0=#x0=this.sim#;           /* erste Zeile    */
prep1=#x1=x0.sim#;             /* mittlere Zeile */
prep2=#x2=x1.sim#;
prep3=#x3=x2.sim#;
  ⋮
animate=this->x0->x1->x2->...; /* letzte Zeile   */
```

Tabelle 1: **Wichtige Methoden in reinem Text in XWizard.**

Method	Description	Note
`obj.for[#a, nb, ne]`	Runs variable `#a` from `i` = `nb` to `ne`, copying `obj` and replacing each occurrence of `#a` by `i`.	
`x.if[exp]` `@{@(x)@ @(y)@}@.if[exp]`	Both versions expand to `x` if `exp` = `true`.	`exp` = `false` yields `y`, if given, or else the empty string.

`smeq[x, y]`
`sm[x, y]`
`greq[x, y]`
`gr[x, y]`
`eq[x, y]`
`neq[x, y]`
	Expands to `true` if `x` $\leq$ `y`, and to `false` otherwise.
	Expands to `true` if `x` $<$ `y`, and to `false` otherwise.
	Expands to `true` if `x` $\geq$ `y`, and to `false` otherwise.
	Expands to `true` if `x` $>$ `y`, and to `false` otherwise.
	Expands to `true` if `x` $=$ `y`, and to `false` otherwise.
	Expands to `true` if `x` $\neq$ `y`, and to `false` otherwise.
		These methods do not interact with the object they are called on. Easiest usage: call on `this`. Non-integer parameters `x, y` cause exception.

`x.add[y]`
`x.sub[y]`
`x.mult[y]`
`x.div[y]`
`x.mod[y]`
	Expands to the integer value of `x` + `y`.
	Expands to the integer value of `x` − `y`.
	Expands to the integer value of `x` · `y`.
	Expands to the integer value of `x`/`y`.
	Expands to the integer value of `x` mod `y`.
		Non-integer parameters `y` or non-integer objects `x` cause exception. The result of `x.div[y]` is rounded down to next integer.

`x.id`
 	
`x.idd` = `@"{x.id}"@`
`x.nil`
	Expands to the script represented by `x`. This is `x` itself, if `x` is plain text. Otherwise, it's the script of the object represented by `x`. Can be used to retrieve the script after the application of plain-text methods, e. g., `*FSM*.det.min.id`.
	Same as `id`, but puts result in plain-text tags.
	Expands to the empty string.
		Can have side effects for non-plain-text scripts, though, e. g., cutoff of inner declaration parts. Then, `idd` can be used, see below.
		To avoid further processing.

`x.get[var]`	Retrieves the value of variable `var`.	E. g., from the decl. part of `x`.
`x.prepTree`	Retrieves the nesting tree of all sub-scripts of `x`.	

`b.newMethod[nam, n]`
 	
`b.newMethodD[nam, n, d]`
	Uses `b` as "body" to create a new plain-text method named `nam` with `n` parameters. The body can be an arbitrary script containing sub-scripts etc. The new method can be used subsequently as `x.nam[p1, ..., pn]`. It expands to the body `b` where every occurrence of the parameter pattern `#i#` is replaced by `pi`. (`x` is considered zeroth parameter `p0`.)
	Same as above, but `d` sets parameter pattern.
		Should be prioritized higher than `b` using stars. `nam` can be called recursively within its own body. This is the key mechanism providing Turing-completeness.
		Default: `#n#`

`b.sethard[c]`	Lets every future occurrence of `b` be expanded to `c` instead of regular expansion: `@{1.fib}@.sethard[1]`	Can be used, e. g., for dynamic programming. Expands to `c`.

Wenn man von einem allgemeineren Blickwinkel auf die verschiedenen Skripttypen schaut, die diesen Animationstyp unterstützen, kann der Code immer auf die selbe Art und Weise erzeugt werden. Diese ist lediglich von drei Parametern abhängig: Den Methodennamen `m1`, `m2`, `m3`. Diese stehen für:

(1) Der erste Aufruf von `this` (führt den ersten Schritt für jegliche zu animierende Veränderung aus) wird als `this.m1` in die erste Zeile eingefügt;

(2) Die anderen Aufrufe auf die nacheinander erzeugten Objekte (führen die restlichen zu animierenden Schritte durch), werden als `xA=x~{A-1}~.m2` in die mittlere Zeile eingefügt;

(3) Die klartextuelle Methode, die die maximale Anzahl an anzulegenden Objekten zurückgibt, wird als `x0.m3` in die mittlere und letzte Zeile eingefügt, sowie als letzter Parameter in `for` Schleifen.

Hinweis: Zur Fehlerbehebung kann es hilfreich sein, eine der Methoden zur Skriptformatierung anzuklicken (z.B. `Format script` oder `Add declarations to script`). Das wird die Vorprozessoren auffächern, zumindest soweit wie sie syntaktisch "zulässig" sind (was weit mehr ist als das, was von einem echten Parser als "korrekt" betrachtet werden würde.)

10.1.4 Wichtige Methoden

Tabelle 1 führt die wichtigsten Methoden, die zum Zeitpunkt des Verfassens dieses Dokuments verfügbar sind, in reinem Text auf, und die im Allgemeinen auf alle Skripte in XWizardanwendbar sind (oder zumindest auf alle reinen Textskripte). Es kann sein, dass mittlerweile neue Methoden hinzugefügt wurden,

Tabelle 2: **Abkürzungen von Methodennamen und skriptspezifische Klartextmethodennamen.**

Skripttyp	englischer Methodenname	Abkürzung	Anmerkung(en)
EA	`Simulate one step`	`sim`	
	`Determinize`	`det`	
	`Minimize`	`min`	
	`Randomize`	`rand`	
	`Randomize (seed)`	`randD`	
	`Regular Expression`	`regex`	
	-	`states`	Gibt die Anzahl der Zustände zurück.
	-	`inputLength`	Gibt die Länge des Eingabewortes zurück.
KA	`Simulate one step`	`sim`	
	-	`states`	Gibt die Anzahl der Zustände zurück.
	-	`maxSteps`	Gibt die Anzahl der Schritte bis zur Beendigung der Abarbeitung zurück.
BDD	`Simplify one step`	`simp`	
	`Truth table (Latex)`	`truthTableLatex`	
	`Truth table (JavaPDF)`	`truthTableJava`	
	-	`max`	Gibt die zur Vereinfachung benötigten Schritte aus.

aber diese Liste sollte ziemlich aktuell sein. Methoden, die sich nicht in reinem Text befinden sind hier nicht abgedeckt, da ihre Funktionalität von dem Skripttyp abhängt, auf dem sie arbeiten.[4] Folgende Tabelle zeigt jedoch einige Abkürzungen für Methoden, die ansonsten sehr schwerfällig benutzbar wären. Außerdem sind einige wichtige skriptspezifische Klartextmethoden in der Tabelle aufgeführt: Tabelle 2

Die Methoden `id` und `idd` sind, was die Funktionsweise angeht, sehr einfach gestrickt (beide geben im Grunde das Skript des Objektes wieder, auf das sie aufgerufen wurden), aber sie haben wichtige und nicht-triviale Auswirkungen:

- Wie oben bereits erwähnt musste `id` aufgerufen werden, um eine Objektkennung an manchen Stellen im Skript korrekt aufzufächern. Das ist zwar nicht mehr notwendig, aber es zeigt, wie `id`, wie alle Methoden, die sofortige Auswertung von Objektkennungen erzwingt.
- Nehmen wir nun an, dass ein LaTeX Skript dazu benutzt wird vier EA Objekte mittels einer for Schleife anzuzeigen, so legt der folgende unbefangene Code keinen EA Graphen an, gibt aber die wirklichen Skripttexte aus: `@{fsm:...}@.for[#a, 0, 3]`. Das passiert, weil die `for` Methode die Subskripttags "verschlingt", was den Compiler dazu bringt die Skripte als Klartext zu behandeln. Das Setzen in doppelte oder dreifache Subskripttags hilft nicht weiter, da XWizard alle verschachtelten Subskripttags ignoriert. Das Verwenden der `id` Methode als Puffer zwischen doppelten Tags per: `@{@{fsm:...}@.id}@.for[#a, 0, 3]` erhält die Subskripttags (ebenso wie auch jeder Methodenaufruf) und erzeugt das gewünschte Resultat.
- Wenn man `*Objekt*.id` auf der obersten Ebene eines (beispielsweise) LaTeX Skriptes benutzt, dann gibt es den Skripttext von `*Objekt*` zurück. Jedoch wird der Compiler den Deklarationsteil entfernen, da er ihn für Deklarationen der obersten Ebene des LaTeX Skriptes hält. Im Allgemeinen können die Tags `@"{` und `}"@` dafür verwendet werden, die wörtlichen Teile, die nicht als Vorprozessoren oder Deklarationen behandelt werden sollen, zu markieren. Das Skript wird in diese Tags verpackt, wenn `*Objekt*.idd` verwendet wird.

10.2 Der XWizard Cache

Um lange Berechnungszeiten vermeiden zu können, beispielsweise wenn eine aufwendige Animation erstellt wird, wird das erzeugte XWizard Objekt inklusive des eigentlichen SVG Bildes in der Datenbank zwischengelagert und von dort geladen, falls gewünscht. Wenn das gemacht wird, kann die Erzeugung von Objekten signifikant beschleunigt werden, da die eigentliche Berechnung weggelassen wird. Der zeitweise Speicher von Objekten (zeitweise, da das Bild mit neuen Entwicklungen von XWizardvielleicht unnötig wird) wird **Cache** genannt. Allerdings wird aus Gründen der Performance nicht jede Skriptausgabe im Cache gespeichert, stattdessen muss für die Benutzung des Caches eine Anfrage gestellt werden. Die Anfrage wird gestellt, indem ein großes "`C`" vor den entsprechenden **Skript** oder die entsprechende **Skript**

[4] Was bedeutet, dass es viel zu viele gibt und sie viel zu komplex sind.

ID geschrieben wird (so: ID-C22384 – oder so: `Cbdd:10110001`). Das führt dazu, dass die zugehörigen Objekte wie gewöhnlich geladen werden, allerdings hat es drei weitere Konsequenzen:

(1) Wenn eine Skriptausgabe bis jetzt noch nicht im Cache ist, wird sie kompiliert und die Ausgabe wird dort abgelegt (die Ausgabe wird anschließend wie immer angezeigt).

(2) Falls der Skript bereits im Cache ist, wird er **nicht** berechnet sondern direkt aus dem Cache geladen.

(3) Wenn ein Skript bereits im Cache ist und **ohne** das führende "`C`" aufgerufen wird, wird es neu berechnet und das im Cache befindliche Objekt wird aktualisiert.

Einige Beispiele in diesem Dokument benutzen das führende "`C`", um die entsprechenden Skripte schnell abzuarbeiten. Verwenden Sie das "`C`" zum Probieren auf verschiedenen anderen IDs. Es ist ebenfalls möglich das "`C`" auf einen ganzen Skript anzuwenden. In diesem Fall wird es ganz an den Anfang geschrieben. Im Falle eines EA Skriptes sieht dies beispielsweise so aus: `Cfsm: ...`

Vergessen Sie bitte nicht, dass die im Cache gespeicherte Version eines speziellen Skriptes überflüssig werden kann, wenn sich die Einbindung in XWizard ändert. Andererseits kann der Cache auch genutzt werden, um Objekte, die **nicht** durch neue Implementierungen verändert werden sollen zu sichern (das funktioniert nur solange das entsprechende Objekt nicht neu berechnet wird; eine Möglichkeit zu verhindern, dass Skripte beim Laden mit dem Parameter "`C`" nicht neu berechnet werden ist für die baldige Implementierung geplant.

Wenn der Cache benutzt wird, werden die PDF Dokumente des Skripts und dessen Subskripte nicht generiert – zudem werden sie auch nicht im Cache abgelegt. Deshalb muss ein im Cache abgelegtes Skript zuerst neu berechnet werden, wenn man das PDF Dokument erhalten will.

Offensichtlich ist der Cache im VFP (noch) nicht verfügbar, da dafür ein Datenbankzugriff benötigt wird.

10.3 Der XWizard Webservice

Der XWizard Webservice wird `DeScriptor` genannt und besteht aus einer einzelnen Java Methode:

```
public java.lang.String retrieveSVGFromScript(
       java.lang.String script,
       java.lang.Boolean withURL,
       java.lang.Boolean withScripttext,
       java.lang.Boolean languageEnglish) throws java.rmi.RemoteException;
```

Der `DeScriptor` Service ist über die Adresse `http://www.xwizard.de:8080/services/DeScriptor` abrufbar. Das Aufrufen der oben aufgeführten Methode übersetzt das Skript (was eine Skript ID sein kann) entsprechend dem ersten Parameter in ein SVG Objekt und der zugehörige SVG Code wird zurückgegeben. Falls einer der beiden mittleren Parameter `true` ist, wird der reine SVG Code in ein HTML `DIV` eingebunden, der einen Link zu der XWizard Webseite enthält, um das entsprechenden Skript und/oder den Skripttext unter der der eigentlichen Grafik zu laden. Der letzte Parameter legt die Sprache fest (Englisch oder andernfalls Deutsch), die in dem zurückgegebenen Text benutzt werden sollte.

Es sei angemerkt, dass der Webservice bis jetzt immer versuchen wird den Cache zu benutzen, also ist es nicht von Nöten das führende `C` zu verwenden, vgl. Abschnitt 10.2. Das ist jedoch noch nicht endgültig festgelegt und kann sich in Zukunft ändern. Also ist es durchaus sinnvoll das führende `C` zu benutzen, wenn der Cache verwendet werden soll. Insbesondere gilt dies für Skripte, die eine lange Zeit fortdauern sollen, wie beispielsweise eingebettete Skripte, vgl. Abschnitte 10.4

Der Webservice kann auf verschiedene Arten aufgerufen werden. Ein Beispiel, wie es per Javascript funktioniert, wird im kommenden Kapitel gegeben. Der folgende Code kann dazu verwendet werden den Webservice von Java aufzurufen.

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
public class TestClient {
    public static void main(String[] args) {
        try {
            String endpoint = "http://www.xwizard.de:8080/services/DeScriptor";
            Service service = new Service();
            Call call = (Call) service.createCall();
            call.setTargetEndpointAddress(new java.net.URL(endpoint));
            call.setOperationName("retrieveSVGFromScript");
            call.setTimeout(10000000);
            String svgString = ((String) call.invoke(
                            new Object[] {ÏD-10700", "false", "false", "false"}));
            // Do something with retrieved svgString.
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Für genauere technische Details schauen Sie bitte unter `https://sourceforge.net/projects/xwiz` nach.

10.4 Das Einbinden von XWizard Objekten in beliebige Webseiten

10.5 LaTeX Abkürzungen

Ein Abkürzungsschema wurde eingeführt, um das Anlegen von LaTeX Skripten vergnüglicher zu machen. Das Verwenden des Schemas hat zur Folge, dass das Skript in vorgefertigten LaTeX Code eingebettet wird, sodass nur der Teil zwischen `\begin{document}` und `\end{document}` explizit im Skript geschrieben werden muss. Ein LaTeX Skript, das das Schema verwendet, sieht folgendermaßen aus:

```
latex: %*docclass* | *packages1* | *packages2* | ... | *packagesn*%
*LaTeX Code in dem Dokumentenrumpf*
```

Der resultierende LaTeX Code wird in etwa so aussehen:

```
\documentclass[...]{...}
\usepackage{...}
⋮
\usepackage{...}
\begin{document}
*LaTeX Code in dem Dokumentenrumpf*
\end{document}
```

Dort werden die `documentclass` und `usepackage` Parameter von den Parametern im Abkürzungsschema festgelegt. Die folgenden Werte sind bis jetzt erlaubt:

Bereich	Abkürzung	Effekt
Erster Parameter (docclass)	`artlet`	`\documentclass[letter]{article}`
	`tight`	`\documentclass[tightpage]{standalone}`
Andere Parameter (packages)	`gra`	`\RequirePackage{graphicx}` `\RequirePackage[space]{grffile}`

	`ger`	`\usepackage[ngerman]{babel}`
		`\usepackage[utf8]{inputenc}`
		`\usepackage[T1]{fontenc}`

	`geo`	`\usepackage[a3paper, margin=1in]{geometry}`
	`relsize`	`\usepackage{relsize}`
	`hyperref`	`\usepackage{hyperref}`
	`qrcode`	`\usepackage{qrcode}`
	`loop`	`\usepackage{forloop}`

... und so weiter. Leider können die Abkürzungen bist jetzt noch nicht angepasst werden. Wenn Sie keine einzige davon mögen, dann müssen Sie das Abkürzungsschema insgesamt vermeiden und das komplette LaTeX Dokument wie gewohnt schreiben... Meistens verkürzen diese Befehle die Erstellung von LaTeX Skripten jedoch extrem.

Möglicherweise gibt es dafür in Zukunft einen LuaLaTeX Support...

11 Bekannte Fehler, Mängel und 'Fallen'

In diesem Abschnitt werden einige bekannte Fehler und Fallen aufgeführt, an die man denken sollte, wenn man XWizard benutzt.

- Prüfungsaufgaben und sensible Skripte sollten nicht öffentlich verfügbar gemacht werden. Falls dies geschieht, so kann jeder die Skripte über die Eingabe der ID in den XWizard's Skriptbereich laden (auch wenn es sehr unwahrscheinlicht ist, dass jemand den korrekten Skript für die eigene Klausur erwischt)
- Die Fehlermeldungen werden derzeit noch verbessert. Bis jetzt wird die java exception-trace angezeigt, falls etwas schief ging.
- Bis jetzt hat das Skriptfeld keine Textvorschläge oder ähnliche Besonderheiten; diese sind in Bearbeitung.
- In der Webversion von XWizard kann (in seltenen Fällen) ohne ersichtlichen Grund eine Fehlermeldung angezeigt werden – insbesondere wenn viele Personen gleichzeitig mit XWizard arbeiten. Wir arbeiten an der Lösung des Problems. **Als Übergangslösung kann man die ausgeführte problemauslösende Aktion wiederholen; in der Regel funktioniert es beim zweiten Mal.** (Wir sind uns bewusst, dass dies noch nicht zufriedenstellend ist.)
- Nach dem Benutzen des "zurück" Knopfes des Webbrowsers funktioniert das Herunterladen der PDF nicht mehr richtig. Um die Funktion wieder herzustellen muss zuerst der "Draw!" Knopf angeklickt werden.
- Nachdem man eine Überführungsmethode verwendet hat kann man den erzeugten Skript nicht gleich im Web verfügbar machen, die "Short URL to this script" Methode funktioniert nicht sofort. Als Übergangslösung kann man entweder zuerst den "Draw!" Knopf drücken, oder man führt die "Short URL to this script" Methode zwei mal aus.
- Fortsetzung folgt.

12 Legal Note

Die folgenden rechtlichen Hinweise finden Sie auch in einer aktuellen Fassung unter:

`http://www.xwizard.de:8080/Wizz?impressum&lang=ger`

Easy Agent Simulation und das darin enthaltene Teilprogramm Very Fast PDF Generator (auch PDF-XWizard oder XWizard-Desktopversion genannt) sowie XWizard, die Web-Version des letztgenannten, sind Open-Source-Programme; sämtliche Quelltexte, insbesondere Programmcode in Java, SQL, XML, HTML, LaTeXCode, GraphViz, XWizard-SCRIPT usw., ob nativ oder automatisch generiert, sind geschützt durch die Creative Commons by-nc-sa-Lizenz.

Alle Quelltexte sowie Javadoc für die meisten Java-Klassen können bei Sourceforge heruntergeladen werden:

- EAS (inklusive VFP) auf Sourceforge: `https://sourceforge.net/projects/easyagentsimulation`
- XWizard auf Sourceforge: `https://sourceforge.net/projects/xwiz`

Kurzgefasst dürfen Sie:

- Teilen – das Material in jedwedem Format oder Medium vervielfältigen und weiterverbreiten,
- Bearbeiten – das Material remixen, verändern und darauf aufbauen.

Der Lizenzgeber kann diese Freiheiten nicht widerrufen solange Sie sich an die Lizenzbedingungen halten.

Unter folgenden Bedingungen:

- Namensnennung – Sie müssen angemessene Urheber- und Rechteangaben machen, einen Link zur Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden. Diese Angaben dürfen in jeder angemessenen Art und Weise gemacht werden, allerdings nicht so, dass der Eindruck entsteht, der Lizenzgeber unterstütze gerade Sie oder Ihre Nutzung besonders.
- Nicht kommerziell – Sie dürfen das Material nicht für kommerzielle Zwecke nutzen.
- Weitergabe unter gleichen Bedingungen – Wenn Sie das Material remixen, verändern oder anderweitig direkt darauf aufbauen, dürfen Sie Ihre Beiträge nur unter derselben Lizenz wie das Original verbreiten.

Keine weiteren Einschränkungen – Sie dürfen keine zusätzlichen Klauseln oder technische Verfahren einsetzen, die anderen rechtlich irgendetwas untersagen, was die Lizenz erlaubt.

Detaillierte Lizenzbestimmungen (Deutsch): `http://creativecommons.org/licenses/by-nc-sa/3.0/de`

Detaillierte Lizenzbestimmungen (unported): `http://creativecommons.org/licenses/by-nc-sa/3.0/deed.en`

Viel Spaß mit XWizard!

Dieses Dokument wurde am 28. Juni 2017 erstellt.