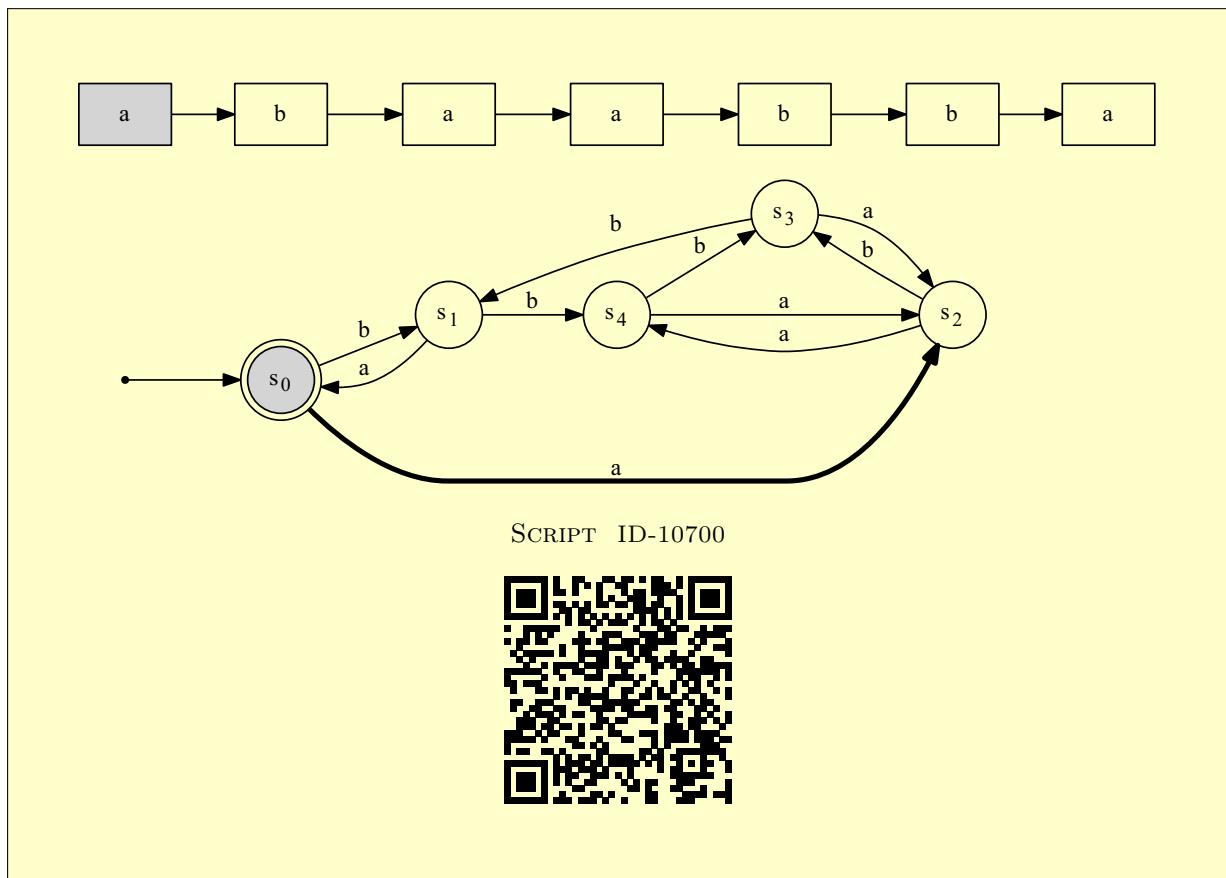




XWizard: The Online Informatics Toolbox

– Handbook for Teachers –

Lukas König, Friederike Pfeiffer-Bohnen



XWizard: The Online Informatics Toolbox

– Handbook for Teachers –

Contents

1	What is XWizard?	3
2	Access and Short History	3
3	Basic Workflow: Script Processing	4
4	Conversion Methods	7
4.1	Conversion methods which create a new script	8
4.2	Conversion methods which create a plain text output	9
4.3	In-Script Application of Conversion Methods (deprecated!)	10
5	The Exercise Mode and Encrypted Scripts	11
5.1	Creating an exercise	11
6	Hyperlinks to XWizard Scripts	16
6.1	Long URLs	16
6.2	Short URLs, Script IDs and the XWizard Database	16
7	PDF Processors and the Conversion Method 'Plain PDF generator code'	17
8	More Complex Objects: Pre-Processors and Sub-Scripts	20
8.1	Sub-Scripts in L ^A T _E X	20
8.2	Pre-Processors	23
8.3	Pre-implemented Examples With Compound Objects	24
9	Simple Animations	28
9.1	Defining basic animations via script	28
9.2	Conversion methods for creating animations	30
10	Advanced Usage: Cool Stuff and Crazy Hacks for Neat Guys	30
10.1	The XWizard Script Language 2.0 – Everything is an Object	31
10.1.1	Informal Examples: the <code>for</code> Loop and the <code>if</code> Statement	31
10.1.2	The XWizard 2.0 Syntax and Semantics	34
Syntax	34	
Semantics	36	
10.1.3	A more advanced Example: Animate to Termination	38
10.1.4	Important Methods (making XWizard Turing-complete)	40
10.2	The XWizard Cache	43
10.3	The XWizard Web Service	44
10.4	L ^A T _E X abbreviations	45
11	Known Bugs, Shortcomings and 'Pitfalls'	45

1 What is XWizard?

XWizard is a free (web) tool for the automatic **visualization, manipulation and PDF generation** of many types of objects from theoretical computer science (such as Turing machines, push-down and finite automata, Chomsky grammars etc.). A broad range of algorithms can be applied to the objects, producing intuitive and customizable views. XWizard is well-suited for students' self-studies, and it is powerful in aiding teachers at the creation of course material such as exercises (the X in XWizard stands for "eXercise" – and also for "anything"). This handbook explains the most important features of XWizard from a teacher's perspective. For more general information, read the document "XWizard: Handbook for Students" or look at the help pages on the XWizard website.

Hint: Readers interested in the basic workflow and the overall functioning of XWizard can skip the next section and move on to Sec. 3.

2 Access and Short History

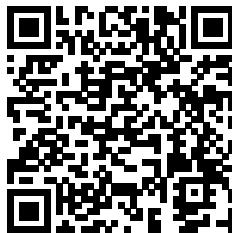
XWizard was created in 2013¹, and the original version was called **Very Fast PDF generator (VFP)**. Its purpose was to simplify the creation of course material, meaning that it was originally solely used by teachers. Today, XWizard is the name of the **web version of VFP**, created in 2015 after students requested an easy-to-use version of VFP for themselves.

XWizard can be accessed via:

www.xwizard.de

or by clicking (or scanning) any of the script links in this document, such as:

SCRIPT ID-10700



 Today, this web version suits well for most purposes. To get the general idea behind XWizard, feel free to play around on the website and apply algorithms to example objects by clicking the "conversion methods" (for example, the conversion method "Simulate one step" of the above script). Note that script IDs such as the one given above can as well be typed into the script field on the website.

The download version VFP, which is still the backbone of XWizard, can be retrieved from:

https://sourceforge.net/projects/xwiz/files/XWizard_VFP.zip

VFP (as opposed to XWizard) has to be installed on a personal computer, and it requires additional software to run². Both versions have essentially the same range of functionality, however, VFP has unlimited computational power while XWizard will interrupt very long or memory-intensive calculations. Nevertheless, **from a teacher's perspective using the web version, which requires only a browser, will in most cases be the easiest and most appropriate choice**, at least for starters. As both versions are mostly interchangeable, only the term XWizard will be used in the following, except where a difference between the two is explicitly addressed, such as this:

 XWizard can be switched between English and German language; so far, VFP is only available in English.

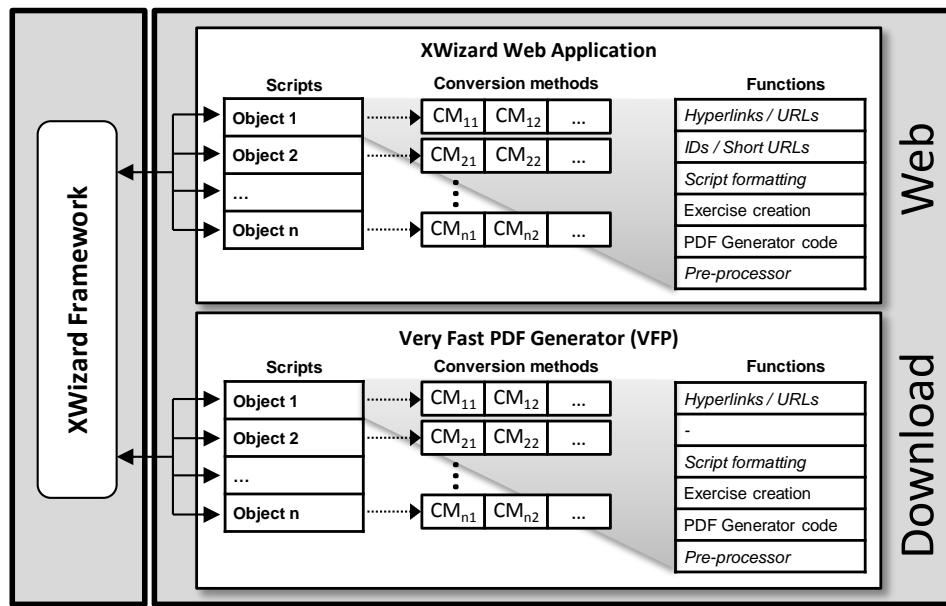
¹By the teaching team of the course "Foundations of Computer Science II" at the Karlsruhe Institute of Technology, taught by Prof. Dr. Hartmut Schmeck, assisted by Lukas König, Friederike Pfeiffer-Bohnen, Micaela Wünsche and Marlon Braun.

²That is, minimally, Java 8, LaTeX (preferably miktex), Graphviz, and SumatraPDF.

XWizard and all implementations related to it are *free software*³. They are allowed to be run for any purpose (except commercial), and the sourcecode can be studied, changed, and further distributed in accordance with this legal notice: <http://www.xwizard.de:8080/Wizz?impressum&lang=eng>.

3 Basic Workflow: Script Processing

The below figure shows the overall structure of the XWizard components for both the web and the download version (XWizard and VFP; as mentioned above, they are basically equal). XWizard objects are defined by **scripts** which can be manipulated by **conversion methods**. Therefore, the main functionality of XWizard is established by the conversion methods. However, conversion methods (and basically everything else) can be encoded into scripts, therefore, in the end, everything boils down to the creation and interpretation of scripts.



XWizard's basic workflow is given by simply processing a script, translating it into a PDF image and displaying this image. A script is usually built up of three parts (four, when including the execution of a conversion method, see below), as illustrated in the following example of a PDA:

```
pda:
  (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);
  (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;
  inputs=000101010;
  simSteps=3;
--declarations-end--
```

← Script preamble

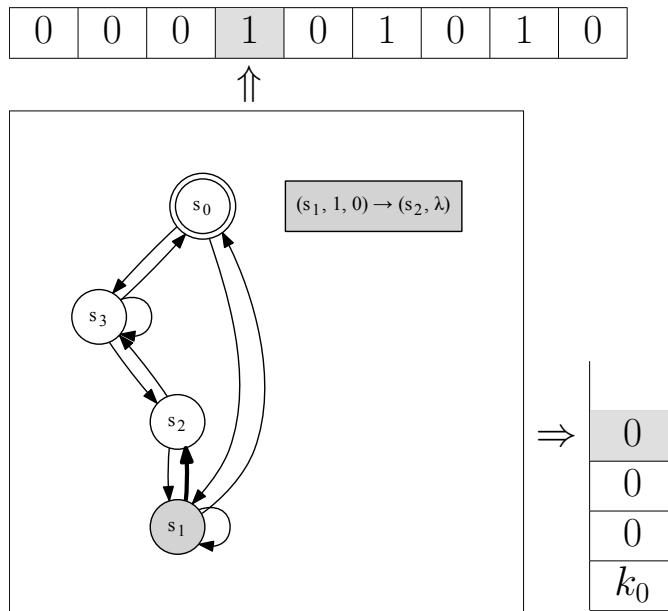
Main script part

Variable declarations

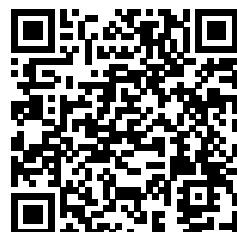
³https://en.wikipedia.org/wiki/Free_software

💡 All scripts include the following three parts: a **preamble** determines the type of object defined by the script; a **main part** defines the actual structure of the object; variables in a **declarations part** can be used to assign many types of additional properties (both the complete declarations part and some of the variables can be omitted, in which case the according variables are set to standard values).

This example script will create the following image (using Graphviz and L^AT_EX in the background):



SCRIPT ID-13417



The script can be pasted into the script areas of both VFP and XWizard to produce the given output, see screenshots below (clicking or scanning the above QR code will open XWizard and automatically execute the script).

VFP:

The screenshot shows the VFP application window. On the left, a PDF viewer displays a stack diagram with states s_0 , s_1 , s_2 , and s_3 . A transition $(s_0, 0, k_0) \Rightarrow (s_1, 0k_0)$ is highlighted. To the right of the stack is a script area containing PDA rules and declarations. Below the stack is a 'PDF output' section showing the result $\Rightarrow k_0$. On the right side of the window is a panel titled 'Examples' listing various examples like FSM, PDA, Huffman, etc. Below it is a 'Conversion methods' panel with several options: Deterministic PDA, Simulate one step, Decollapse rules left, Show additional information..., Create exercise from this script, Calculation steps (Latex), Decollapse rules right, URL to this script, Format script, and Plain Generator code.

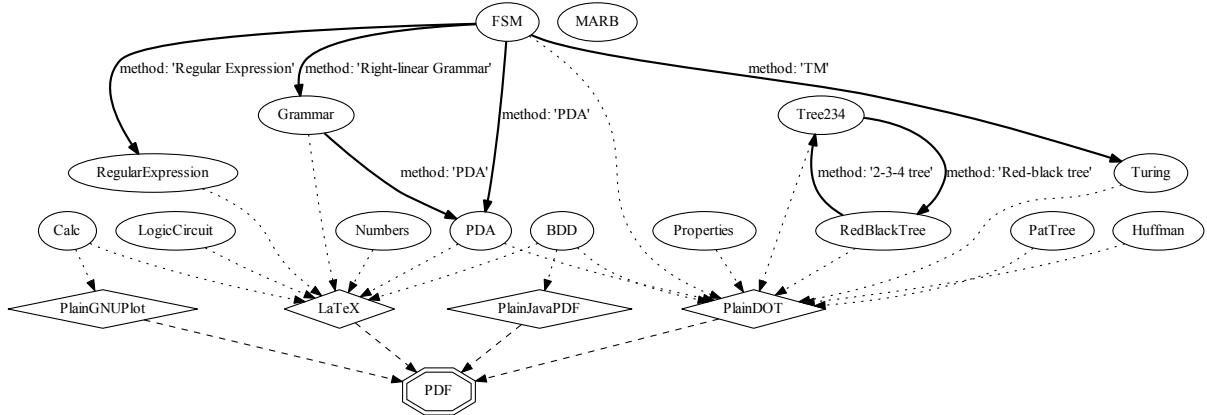
XWizard:

The screenshot shows the XWizard interface. On the left, there's a sidebar with a wizard logo, language selection (Deutsch), contents, and links to 3-minute survey, help, and legal notice. The main area contains a stack diagram and a script area. The stack diagram shows state transitions: $(s_1, 1, 0) \rightarrow (s_2, \lambda)$ and $(s_2, \lambda) \rightarrow (s_3, 1)$. Below the stack is a 'PDF Output' section showing the result $\Rightarrow 0 0 0 k_0$. The script area contains the same PDA rules as the VFP version. To the right is a 'Conversion methods' panel with options: Show PDA definition (Latex), Decollapse rules left, Decollapse rules right, Format script, and Plain Generator code. At the bottom, there are buttons for Draw!, Display modes (Show PDA definition, Simulate one step, Toggle complete calculation trace), Script formatting (Decollapse rules left, Decollapse rules right, Format script), and a link to Discuss this script.

To simulate the calculation of the displayed PDA, the “simSteps” variable in the script can be increased; equivalently, the **conversion method** “Simulate one step” can be clicked (cf. description of conversion methods below).

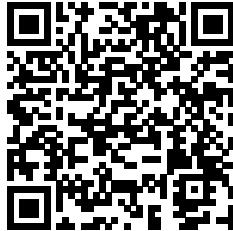
When using VFP, the PDF output is automatically updated each time the script area changes. When using XWizard, the image output is created after clicking the “Draw!” button on the web page. A PDF can be retrieved by clicking the “Download PDF” link; it appears when scrolling down below the PDF output.

An overview of the script types available at creation time of this document is given in the following figure (solid and dotted arrows denote script type transitions by conversion methods; dashed arrows denote the PDF creation process; diamond nodes represent plain PDF generator script types – these are basically the raw Graphviz and L^AT_EX types (GNUPLOT and JavaPDF are obsolete), see below).



A current version of the figure can be retrieved via this link (note that it is itself generated by a simple XWizard script):

SCRIPT ID-15812



For any of these object types, there are example scripts on the XWizard web page:

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

Note that the syntax of scripts will not be discussed here in detail; for this, see the XWizard help pages:

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

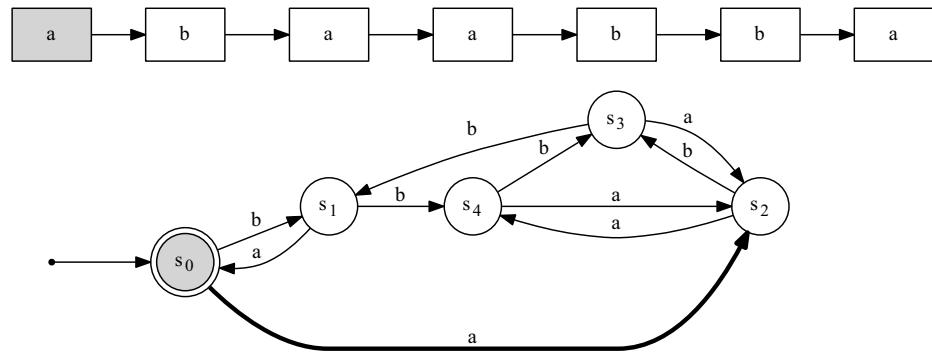
To sum up this short section, the basic workflow of XWizard is given by taking an input script and creating a PDF image from it. Nearly all GUI-based abbreviations and simplifications described below can be substituted by creating an according script; more precisely, all the GUI does is inducing the creation of appropriate scripts in the background. A major benefit of this structure is that every action can be archived by simply storing the according script, cf. Sec. 6.

4 Conversion Methods

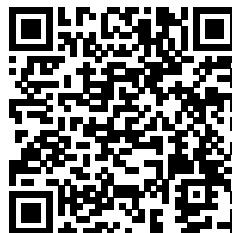
Besides creating and displaying objects, a main functionality of XWizard is to apply algorithms to scripts, e.g., to visualize the stepwise computation of a PDA or to minimize an FSM. Algorithms are applied to objects by using conversion methods, i.e., methods that transform one script into another. Conversion methods provide a simple user interface for applying algorithms to XWizard objects.

 When applying a conversion method, the script belonging to the current object is replaced by the new script created by the conversion method, and the object defined by the new script is created and displayed. (Note, however, that there are also conversion methods that produce a plain text output rather than a new script.)

The easiest way to apply a conversion method to a script is to click the according button in the “Conversion methods” area of the graphical user interface. For example, a finite state machine (FSM), such as the following, comes with a set of conversion methods as shown below.



SCRIPT ID-10700



Conversion methods

Conversion into

Determinize
Minimize
PDA
Randomize...
Regular Expression
Right-linear Grammar

Simulate one step
TM

Display modes

Toggle minimization table
Toggle minimized/determinized FSM

Script formatting

Decollapse rules left
Decollapse rules right
Format script

Additional information

Show minimization chain...

For teachers

Create exercise from this script...
Plain Generator code
URL to this script...
Short URL (ID) to this script...

4.1 Conversion methods which create a new script

Conversion methods which convert one script into another are the usual – and by far most-frequent – ones. Available conversion methods are displayed below the script area; the type of the current script defines which conversion methods are applicable, and only these are displayed. In the above example, the following conversion methods are shown for an FSM script, grouped along the categories printed in blue:

- **Conversion into** – Methods that will create a “new”, i.e., semantically different object:
 - **Determinize** will use the well-known powerset algorithm to create a script that represents a deterministic FSM equivalent to the original one. (This method is not enabled in the above example, because the FSM is already deterministic.)

8

- **Minimize** will create an equivalent FSM with a minimal number of states, using the Myhill/Nerode equivalence statement.
- **PDA, Regular Expression, Right-linear Grammar** and **TM** will create an equivalent push-down automaton, regular expression, right-linear grammar or Turing machine script, respectively.
- **Simulate one step** will let the FSM consume one more character of the given input, or ask the user to enter an input word if none is given.
- **Randomize...** will create a new random FSM, by asking the user first to enter a number (of states) and a boolean value (indicating if the new FSM should be deterministic). (Note that the resulting FSM will always be minimizable, to facilitate the creation of exercises.)
- **Display modes** – Methods that will change the view on the current object, but not its semantics:
 - **Toggle minimization table** will switch between views showing only the FSM, both the FSM and its minimization table, and only its minimization table.
 - **Toggle minimized/determinized FSM** will switch between views showing or hiding an additional view of a minimized and a determinized version of the current FSM.

 Showing different equivalent versions of the same FSM allows to simulate them simultaneously, which can be interesting both for students and at exercise generation.
- **Script formatting** – Methods that will change the script appearance, but not the output:
 - The two methods **Decollapse rules left** or **right** are available for all scripts based on rules such as $A \Rightarrow B$. Such rules can be collapsed on the left side ($A \mid B \Rightarrow C$) and on the right side ($A \Rightarrow B \mid C$) for better readability. The “Decollapse” methods will undo such collapsing on the left or right side, respectively.
 - The **Format script** method will collapse the rules where possible, and do some more formatting.

 Particularly, the methods **Format script** or **Add declarations to script** (which one depends on the current script type) will add the declarations part to a script including all available variables.
- **Additional information** – The single method **Show minimization chain** from this category is a plain text conversion method (cf. Sec. 4.2). It produces an output ready to be copied into a L^AT_EX document, showing information about making the current FSM deterministic first and minimizing it afterwards.
- **For teachers** – The methods from this category are available for all scripts. As they all relate to major topics of this handbook, they are explained in detail in Secs. 5, 6 and 7.

Three dots (...) at the end of a button name indicate methods that require some user interaction. Some methods need parameters to be executed, these methods will ask for those parameters before starting the conversion, cf. the “Randomize...” method explained above. The remaining methods with dots are those that create a plain-text output. They will just open a new window to show their output, asking the user to press a confirmation button.

4.2 Conversion methods which create a plain text output

Besides the above-described, regular conversion methods, there are those which create a plain-text output. For example, the method “Show minimization chain” of an FSM script produces a L^AT_EX code output:

Plain text output

```
($s_{(2)}$ & \$\times_{(1)}\$ & \$\$)\\
($s_{(3)}$ & \$\times_{(0)}\$ & \$\times_{(0)}\$ & \$\times_{(0)}\$)\\
($s_{(4)}$ & \$\times_{(1)}\$ & \$\$ & \$\$ & \$\times_{(0)}\$)\\
($s_{(5)}$ & \$\times_{(0)}\$ & \$\times_{(0)}\$ & \$\times_{(0)}\$ & \$\times_{(1)}\$ & \$\times_{(0)}\$)\\
($s_{(6)}$ & \$\times_{(1)}\$ & \$\$ & \$\$ & \$\$ & \$\times_{(0)}\$ & \$\$)\\
($s_{(7)}$ & \$\times_{(1)}\$ & \$\$ & \$\$ & \$\times_{(0)}\$ & \$\$ & \$\times_{(0)}\$ & \$\$)\\
($s_{(8)}$ & \$\times_{(1)}\$ & \$\$ & \$\$ & \$\times_{(0)}\$ & \$\$ & \$\$ & \$\times_{(0)}\$)\\
($s_{(9)}$ & \$\times_{(1)}\$ & \$\$ & \$\$ & \$\times_{(0)}\$ & \$\$ & \$\$ & \$\$ & \$\$)\\
(& \$s_{(0)}\$ & \$s_{(1)}\$ & \$s_{(2)}\$ & \$s_{(3)}\$ & \$s_{(4)}\$ & \$s_{(5)}\$ & \$s_{(6)}\$ & \$s_{(7)}\$ & \$s_{(8)}\$)\\
(){}{}{}{}{}{}{}{}{}{}\\
%  
% Minimized state transition table:  
\begin{tabular}{|c|c|c|}\hline  
& $a$ & $b$ \\ \hline  
\end{tabular}\hline\hline
```

[Return to main page](#)

In this case, the output is a L^AT_EX code that includes information about the minimization and determinization of the current FSM. Besides this, the main plain-text conversion methods with “straightforward applicability”⁴ are:

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

These methods create URLs to the current script to facilitate script exchange between users, cf. Secs. 6.1 and 6.2. Other than that, plain text conversion methods are used very rarely and can usually be replaced by a more convenient regular conversion method. For example, the L^AT_EX code of the minimization chain can be retrieved by selecting an appropriate display mode and creating the “Plain Generator code”, see Sec.7.

4.3 In-Script Application of Conversion Methods (deprecated!)

 **Important note:** The application of conversion methods via scripts is an advanced feature, explained in detail in Sec. 10. The way it is done in this section is **deprecated and should no longer be used**. It will, however, remain as a feature to allow backward compatibility, particularly, because the XWizard website currently still relies on this mechanism.

As mentioned above, scripts control XWizard completely; this particularly means that even the application of conversion methods can be initiated via a special script command. Scripts containing this command are called **conversion scripts**, and they start as usual with the regular three parts, which determine the script to be converted. As a fourth part, a **conversion command** is written as the last line of the script. It looks like this:

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

where CM-NAME is the English name of the conversion method to be applied or, if the conversion method requires parameters:

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

where p₁, p₂, ... are the method parameters (they can be put in quotes if they are supposed to include special characters such as white spaces or commas: ["p, a, r, 1", "p, a, r, 2", ...]). For example, the “Simulate one step” conversion method can be applied to a PDA script by adding the red-colored last line to it:

⁴Meaning that they can be used by simply clicking a button. There are many more “hidden” plain-text methods that can be used in the XWizard script language 2.0, cf. Sec.10.

```

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. */

```

When entering this script, the result will be exactly the same as after clicking the according button on the script without the conversion command.

5 The Exercise Mode and Encrypted Scripts

XWizard has a distinguished mode called **Exercise Mode** (which is currently only available in the web version). In this mode, the user (typically a student) is asked to solve a task displayed at the top of the page. To get to the solution, the user is allowed to utilize a predefined subset of the XWizard's functions. This subset can be flexibly defined by the creator (typically a teacher) of an exercise, and it can involve both an adjustment of the available conversion methods and a limitation of the script utilization. More precisely, the Exercise Mode differs in the following regards from the regular mode:

- (1) The XWizard website prompts a question, displayed over the script area, and requests the user to answer it (cf. screenshot below).
- (2) Some of the conversion methods may be hidden to prohibit undesired shortcuts to the solution.
- (3) The script may be partially or completely encrypted to prohibit users from cheating by changing it or reading the exercise definition (see below).
- (4) When answered correctly, the user is offered a “badge”. (This is so far just a secret code word, displayed to the user; in future, the implementation of personal “portfolios” is planned, allowing users to collect badges, experience points or similar things.)
- (5) When answered correctly, an additional explanation may be displayed to the user if provided by the creator.

5.1 Creating an exercise

The exercise mode is defined by a variable called “e” (or “**exercise**”) in the declarations area of a script. Therefore, this variable has to be defined according to a specific syntax in order to create an exercise. A conversion method **Create exercise from this script** (which is available for every script type in the “For teachers” category) facilitates this process. When executing the method, the user is asked for the following parameters (all except the last two are string values; optional parameters can be simply left empty):

- **titleString**: A title displayed above the detailed exercise description.

- explanationHTML: A detailed description of the exercise which may contain HTML.
- solutionString: A string which represents the solution to be entered by the user. This parameter is optional, if left empty, the user is not prompted to enter a solution.
- codeToEarn: The “badge”, i. e., a string displayed to the user as a reward if the answer is correct.
- regexForAllowedMethodNames: An optional regular expression (as used in Java) to restrict which conversion methods are displayed. The regular expression is applied to the English method name and displays only matching methods. For example the regular expression

.*inimiz.*

will only display the two methods **Minimize** and **Show minimization chain...**

- regexForAllowedclassNames (*for expert use only, can usually be left empty*): Another optional regular expression to restrict the display of conversion methods. It is applied to the class name of the base class which provides the method.
- regexForAllowedTargetClassNames (*for expert use only, can usually be left empty*): A third regular expression to restrict the display of conversion methods. It is applied to the class name of the target class, i. e., the class of the converted script.
- solExp: Optional explanation to be displayed with the solution after a correct answer has been given (may contain HTML).
- exEncrypt (*boolean*): Set this to true if the code of the exercise (not the complete script code) is to be encrypted (see below).
- encrypt (*boolean*): Set this to true if the complete script code should be encrypted (see below).

After the input of these parameters, an exercise definition is added to the current script, and this causes XWizard to enter the exercise mode. (Note that “e=n” or “e=null” is the code for “regular mode”.)

 The following script shows an example of how an exercise is encoded in the declarations part of a script. The output is displayed in the screenshot right below it. Click the script link to see the example in a browser.

```

grammar:
    A => A, A | 0 | epsilon;
    E => A, 1, A;
    S => E, E, E | S, S | 0;
--declarations--
e=#tit=^Create the parse tree for the word 01011 with the given Grammar.^,
exp=^<P>The output area shows the grammar tree with several derivations
of words generated by the grammar. Since the grammar includes an epsilon
production, the grammar has to be made epsilon-free first by using
the according conversion method. Afterwards, you can use the remaining
conversion method to create the parse tree for 01011. (All other
conversion methods are hidden.)</P><P>Execute these steps and count the
number of non-terminal nodes in the tree. Enter this number into the
solution field.</P>^,
sol=~6~,
cod=~parser-guru~,
met=~.*Epsilon.*|.*Parse.*~,
cur=~.*~,
tar=~.*~,
crypt=~false~,
excrypt=~false~,#;
N=S,E,A;
T=0,1;
S=S;
...
--declarations-end--

```

 In the declarations part, the symbols # and ~ indicate the beginnings and endings of strings which may include special symbols such as “= , ;” etc. (however, the symbols # or ~ themselves, respectively, and some reserved words cannot be used unconditionally). To completely secure a string in the declarations part, it can be put within the following bracket combination: [~(~{ some text }~)~]. This allows to include all character combinations including subordinate instances of the bracket combination itself, until the secure part is finished by the matching closing bracket combination (cf. further explanations in Sec. 8).

Create the parse tree for the word 01011 with the given Grammar.

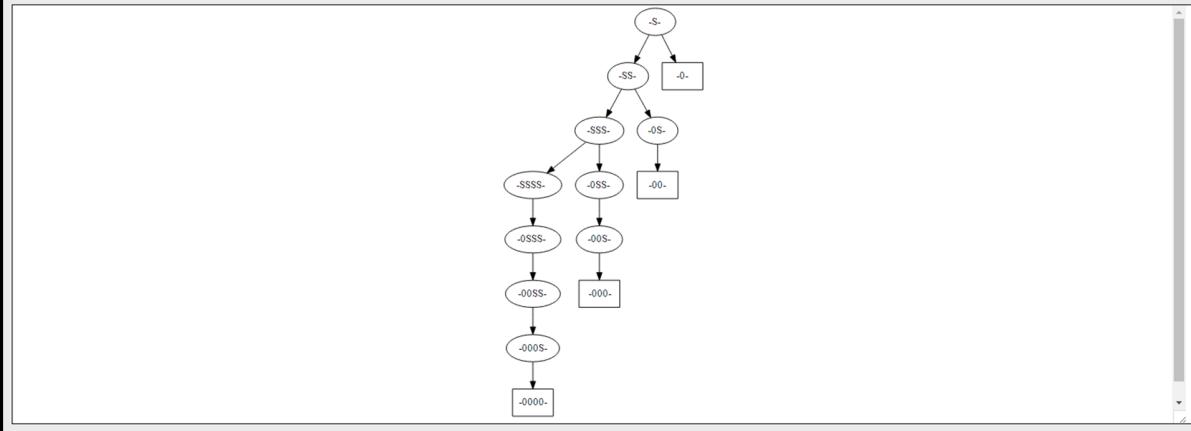
The output area shows the grammar tree with several derivations of words generated by the grammar. Since the grammar includes an epsilon production, the grammar has to be made epsilon-free first by using the according conversion method. Afterwards, you can use the remaining conversion method to create the parse tree for 01011. (All other conversion methods are hidden.)

Execute these steps and count the number of non-terminal nodes in the tree. Enter this number into the solution field.

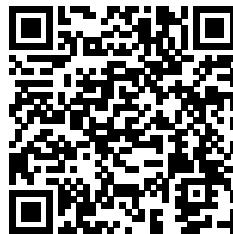
Your solution:

Solve task

[→ Close exercise](#) | [Discuss this exercise](#)



SCRIPT ID-11020



As the example script shows, the exercise definition is, for now, given as plain, human readable text in the script. Particularly, this means that students are able to see the solution as well as change parts of the exercise definition. For example, they can change it to show conversion methods which are supposed to be hidden. As this may be undesired, **two levels of encryption** can be applied to the script. The first level is activated by setting the exercise parameter

`excrypt=true`

which leads to an encryption of the exercise definition. The second level encrypts the whole script, and it can be activated by setting the parameter

`crypt=true`

The two example scripts below show the respective results; the output on the website remains the same as shown above.

Example script with “excript=true”:

```
grammar:  
    A => A, A | 0 | epsilon;  
    E => A, 1, A;  
    S => E, E, E | S, S | 0;  
--declarations--  
e=#scrypt:401s3X3o133k2L2R2g202p2z0x3w1Q1G052n0b040u2K0K011y2b1k161t0d0Q1J0I  
1K0A0B2m1c0T1B3G3726250T1q2y1h0c3d3f2B1T1D350z3j2c3L3a3u2M0b102s21  
061a0J3X0e1K1z3Y430X1v142Z0A0N1T0e1z042831301F1r002J2x1020450V1x35  
0Y1G2M1E0Q1C283s3f3718400m3X3F170K1m0d3d1004302H101a1z3G3p082E2k07  
2G2H3S3a1k3D382j071d1p45271h012w460z1m180k332e3u263y2E3i0h3C1z1G0P  
0H0A1v2X1K3K1n3k042714303346143R063Y3S3C3f3r2P3h0w3F3x0P1g0F0t3w0V  
3e3M3t0k2k3I3i3F1S2a2y0704440S3M0T033h1e1y3q351d1I0d3T0S01463z0I1Y  
1M1h163x1i2a0K2y0D2d0y1g1X2H011m303D2g1w2D0p1J1Z471Z2b3j461V0a200L  
0k3g2H3f2P11320Y3Q3J3g3j081Q443m300d0o1S3E0R342b1k1W2J1o1G262V3i0R  
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--
```

Example script with “crypt=true”:

```
scrypt:401s3X3p1t0a1t2Q2k452j1C2z2a1U0B2h000r0k3o3g0X3w3e3x112q0S2B1y360v2p3N3E  
2f3G1b1G0i2f2M221U0e1C280W1v2q2w1B0p2T3V1E1e3D1k301j1V14450s1h1o122z3C1f3C  
2h310M0A2z2i3A3L3N341f3J1n2p3v213e3D0t140a1D071I3M2a3X3M2B3G342L2e090d2q3k  
283H0d2P0a1X2g1X3z3m01f3B2w0a002k3004372Y313B2i212i1o1D1m1b302d1d1n3U0R02  
0j2e3z1g0R1I082a103j261d0h1z411y1B211U3D3d0o0d3D0825280h021N1K1b312x3p1L3a  
0t41322y1j2d2h2m0t303k0y122y1Q2Y2j2N2u26273g1q3B221K2P0x3s0C133J0k0k1i3n0y  
2b3D3Q1g121y0I0c3j2C3H1h3M31373U2M1K21422B1P3s3A2w1Q141c1X0g1E3b0L3i0r1t3a  
110g0s2o2V1o0g3y31041i3F1G0I2M3D452c1G3k021C2S0U1q2c2h0f461k1E0M2r1c200P  
2R1f3R221J402W0W043t3N0J132m1M3A2m1W1C2N3W3g3g2U1D1J1Z2X1t2b3U10303F2n2H3g  
0y1R012W2D352E3X3d2n1Q2A2n1Z331j0h17062W2G1H0Z2S3G110N1b303H3U3j0i3n383H2c  
0B3u1X3x0j2S2E302p400H3F3U011e382N1d2U443G0k1C1B1Z0h2V3s1Q2R1N1B0a2W0W2Y2S  
2m1e193U3n3j2W0n0b0D3U0N3j400V0w1I1G2m2W002A0K2z0d0X2g3q2z3y163W3o1p102A0Q  
0D1W0e0e40153N3s0N3W1G31103G0H3W100Y0C0T3q461i1A1Q0j3s1z3R2J1M173v1A2k3022  
0126003H3F1f1M0W1N3J2H3i392y1P2U3H383R1i3t2v1V2u23202A0K3d2U3P1G460B151C1N  
3o2K1I0w2y2R0g3h1T0B1x3x1L110b2D222Q2m1h0T0K1K140e1u1u160u2P3B201X1i1a0y1W  
2h3i3q3r08
```

Note that encryption is only meant to make cheating difficult – not to make it impossible. It is achieved by a combination of zipping the text and converting the result to alpha-numeric values. As the XWizard sourcecode is free, ambitious students can download it and use it to decrypt a script. Nevertheless, be aware that after applying encryption in XWizard, **you will not be able to undo this easily**, therefore it makes sense to backup a non-encrypted version of each script.

6 Hyperlinks to XWizard Scripts

XWizard can generate URLs which point to specific scripts, to facilitate the exchange of scripts. This technique has been used to generate the script links in this document. When following such a link, the XWizard website will open and automatically load the according script. There are two types of URLs (“long” or “short”) that can be used for this purpose; they are explained in the following.

 Short URLs are preferable in most situations to long URLs in terms of convenience and security, since long URLs can look ugly and even get too large to be accepted by certain browsers. Nevertheless, long URLs, as opposed to short URLs, carry the whole information about a script which makes them independent of the XWizard database (see below).

6.1 Long URLs

Every script type comes with a conversion method **URL to this script...** which generates what is here called a “long URL” to this script. For example, the following script leads to the URL shown below:

```
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--
```

The URL points to the XWizard website and transmits a parameter “template” which contains the whole script as a URL-encoded string. When such a link is opened, XWizard will decode the script, copy it into the script area and show the according output image.

In principle, every script can be converted into a long URL, however, if the URL gets too large, a browser may reject parts of it which leads to corrupted scripts. Furthermore, malware blockers may produce alerts due to the unusual form of these URLs. To cope with these issues, **short URLs** have been introduced.

6.2 Short URLs, Script IDs and the XWizard Database

A script of similar size as the one shown in the previous section, such as this:

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

can be encoded into this much simpler URL (using the conversion method **Short URL to this script...**):

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

Instead of encoding a complete script within the URL, short URLs make use of the XWizard database. XWizard (not VFP!) stores every processed script along with different types of corresponding information into a mysql database. Each script is assigned an ID in the database which can be used to retrieve the script later. If this ID is passed to XWizard using the “template” parameter, it will be looked up in the database and checked if it is “web-free”. Of course, not all scripts can be looked up by everyone; rather, when executing the **Short URL to this script...** method, a flag is set which makes the script web-free, meaning that it can be looked up from now on. All other scripts are protected and will not be shown when the according ID is tried to be retrieved. Therefore, only the creator of a script can decide to make it publically available via ID. All script links in this document have been created by using this mechanism.

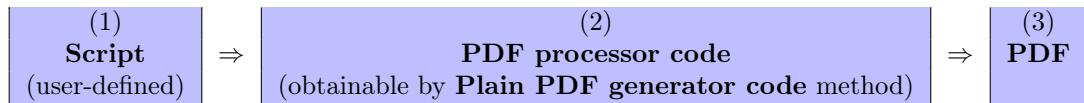
 Although convenient, short URLs set up a pitfall when creating “critical” scripts, e.g., scripts for exam questions, as scripts with public IDs can, in principle, be viewed by everyone. Despite the unlikeliness of students “guessing” an ID which belongs to a script relevant to their exam, critical scripts should not be made public by creating short URLs for them.

 A public script ID can be typed into the script area of XWizard (not VFP) to retrieve the according script.

 Adding “&lang=ger” or “&lang=eng” to any of the two URL types can specify XWizard’s language to German or English. Adding “#Output”, “#Codebox”, “#ConversionMethods”, “#Examples” etc. to the very end of a URL will let XWizard jump immediately to the respective part of the website.

7 PDF Processors and the Conversion Method ‘Plain PDF generator code’

Every XWizard script is associated to a PDF processor which compiles the script into a PDF. The translation process is divided into three steps as follows:



There, the PDF processor code is plain source code in the language of the PDF processor, i.e., it can be copied and compiled outside of XWizard as well, using, e.g., pdflatex or graphviz (which is basically what XWizard does). The current PDF processors used in XWizard are L^AT_EX and Graphviz; overall the following are implemented so far (however, GNUPlot and JavaPDF are deprecated):

- L^AT_EX,
- Graphviz,
- GNUPlot (only available in VFP; GNUPlot has to be installed),
- JavaPDF (implemented, but not yet used in productive mode).

 Technically, an optional fourth step can be executed to convert the resulting PDF into an SVG image. As SVG is nicer to display with HTML, this step is part of the regular workflow on the XWizard website (also, because the animation functionality described in Sec. 9.2 relies on SVG). In VFP, the SVG mode can be turned on or off; the latter can significantly reduce the calculation time. (The XWizard web service also returns an SVG image by default, cf. Sec. 10.3.)

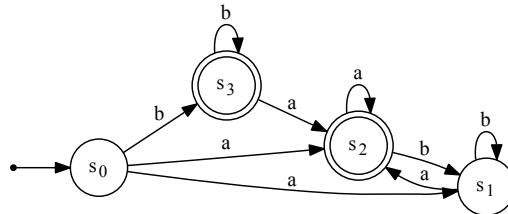
In contrast to students, teachers may be interested in manipulating the PDF processor code – for example, in order to change some specifics in the depiction of an object to point out some peculiarity relevant to the course.

For example, it might be desired to highlight the **nondeterministic transitions** in the FSM depicted below on page 18. However, the FSM script type only allows to define an FSM object in terms of its structural properties, while the actual depiction is left to an internal algorithm. To get access to the actual depiction code, i.e., the raw PDF processor code, XWizard offers for each PDF processor a specific script type which allows to use code directly as accepted by the PDF processor. When using this script type, plain L^AT_EX or Graphviz (or GNUPlot) source code can be inserted as the main script part, to be sent directly to the PDF processor.

 Note that the plain PDF processor script types **can be** used to just pass code to the PDF processor, but they can do more than that. First, preprocessors may be added to the code, see Sec. 8. Furthermore, some changes to the interpretation of the code can be achieved, e.g., by changing the declarations or by using the L^AT_EX abbreviations, cf. Sec. 10.1.4.

When executing the conversion method **Plain generator code**, the script is converted into the according plain PDF processor code, embedded in the according plain script type. Now, this script can be changed as desired, according to the rules of the language of that PDF processor.

This procedure is illustrated using the following FSM script. As suggested earlier, it might be desired to highlight the non-deterministic transitions (the two outgoing transitions from state s_0 labelled a) in the PDF output in order to point out some peculiarity in a course.



SCRIPT ID-11832



The regular script producing the above automaton looks like this:

```

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--
  
```

Obviously, there is no way of adding information for highlighting transitions to this script. However, by executing the **Plain generator code** method, the script is being converted into raw Graphviz code which looks as follows (without the text highlighted in red):

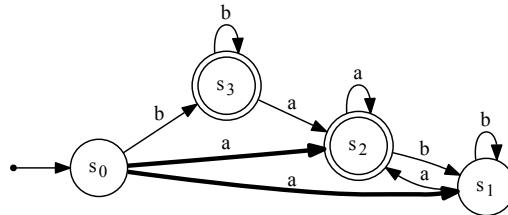
```

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"];
}

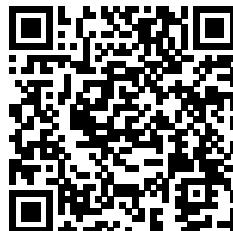
```

 Converting a script into the plain PDF processor code will never change the PDF output. (However, the above code has been cleared out a little for better readability which slightly changes the output.)

Based on this script, the highlighting of non-deterministic edges can be achieved, for example, by adding the red-colored text to the plain Graphviz code. This causes Graphviz to draw the desired edges thicker than the others. This results in:



SCRIPT ID-11836



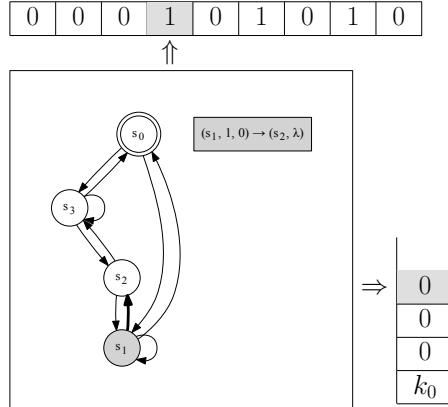
 All kinds of changes can be made to this script as long as the Graphviz syntax is obeyed. The same is possible with scripts based on L^AT_EX. An according example is shown in the next chapter.

8 More Complex Objects: Pre-Processors and Sub-Scripts

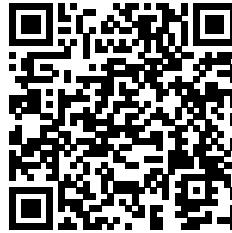
Note that this section provides only a basic introduction of XWizard's pre-processor engine. It is enough to create basic semi-complex objects, but XWizard can do much more. Interested readers can skip this section and directly go to Sec. 10. (Chronologically, this section is about what XWizard could do before the major language extensions in **XWizard-script 2.0**, described in Sec. 10.)

Both Graphviz and L^AT_EX are powerful programs on their own, but, as is shown in the following and particularly in Sec. 10, it makes lots of sense to combine their capabilities to allow for the creation of even more advanced objects.

As a first example, let's revisit the push-down automaton from page 5, depicted below. It has been created as a combination of a Graphviz graph and two L^AT_EX tables. It would be very difficult to achieve the same by using only one of the two programs. Also, embedding "sub-objects" into other objects can be convenient for, e.g., easily including an FSM image into a L^AT_EX document. To facilitate this, XWizard provides so-called "pre-processors". A **pre-processor** is simply an XWizard script X which is embedded into another script Y . During the translation of Y , X is translated first, creating the according PDF image P_X . As the actual father script Y is translated subsequently, Y can import P_X , thus combining its own contents with X 's output. In the PDA example, the Graphviz graph X is a pre-processor embedded within the main L^AT_EX script Y .



SCRIPT ID-13417



This general mechanism is encapsulated in an easy-to-use structure called **sub-script** which we will start with.

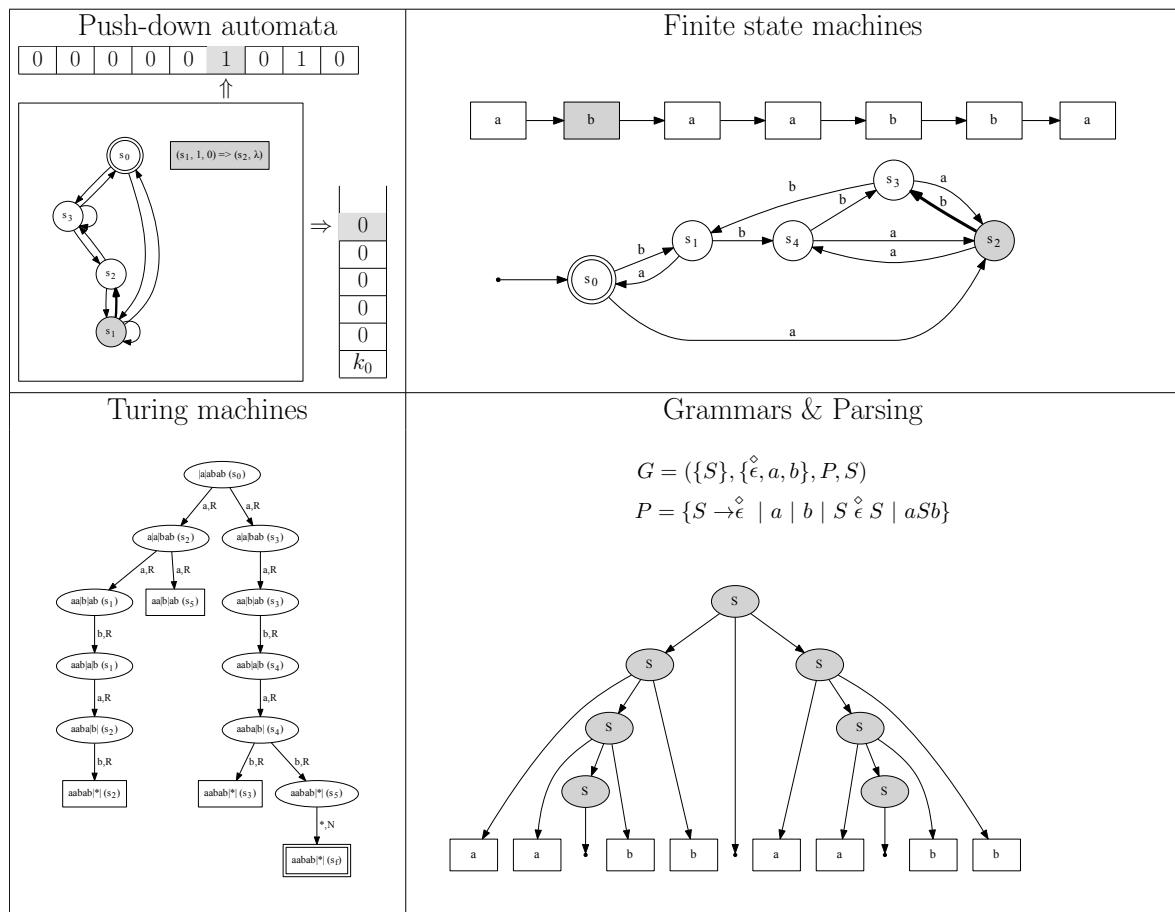
8.1 Sub-Scripts in L^AT_EX

A sub-script is an XWizard script which is embedded into another XWizard script (just as a pre-processor, only using a simpler syntax). In principle, a sub-script can be placed at an arbitrary place inside another script by putting it between `@{` and `}@` (even within a sub-script or a sub-sub-script etc.). At compilation time, sub-scripts are – normally – translated into a PDF image and replaced by code to include the created PDF image at the respective position in a L^AT_EX document. This is the "normal" use-case of sub-scripts as the following example shows.

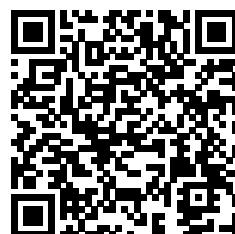
 “Normal” sub-scripts are just regular XWizard scripts which translate to PDF objects. They can be included into plain L^AT_EX scripts only, as shown below. Therefore, L^AT_EX is (so far) the only script type that can handle them. However, sub-scripts translating to plain text can be used in scripts of arbitrary types, cf. Sec. 10.1.

The image below is a complex L^AT_EX object which contains four XWizard sub-objects (a PDA, an FSM, a Turing machine and a grammar object).

Some of XWizard's basic object types:



SCRIPT ID-16124



It can be generated by the following XWizard script (the code of the sub-objects, the sub-scripts, is highlighted in red):

```

latex:
\documentclass[tightpage,preview]{standalone}
\usepackage{varwidth}\usepackage{amsmath}\usepackage{table}\usepackage{xcolor}\usepackage{graphicx}\usepackage[space]{grffile}
\begin{document}
\huge~\par
Some of XWizard's basic object types:
\bigbreak
\begin{tabular}{|c|c|}\hline
Push-down automata & Finite state machines \\
\hline
\end{tabular}
@{0.75|


```

The script is based on the plain L^AT_EX script type (cf. Sec. 7) where sub-scripts can be embedded as follows:

```
@{ *some script* }@
```

On the L^AT_EX level, the document basically consists of a simple tabular with four cells. Each cell contains a sub-script which has to be translated by XWizard before the L^AT_EX run. During the XWizard translation, sub-scripts are translated first and stored in PDF files, say `file*X*.pdf` for sub-script X. Subsequently, each sub-script X in the L^AT_EX code is replaced by (basically) the following code:

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

As a consequence, the subsequent L^AT_EX run will include the pre-compiled PDFs at the positions of the corresponding sub-scripts, thus, inserting the sub-script's output into the overall script's output.

Obviously, sub-scripts may only be placed at positions where `includegraphics` is allowed in L^AT_EX.

 Also, the following package imports have to be provided in the preamble of the L^AT_EX document (Sec. 10.4 describes how this can be replaced by a nicer abbreviation.):

```
\usepackage{graphicx} % Provides the includegraphics makro.  
\usepackage[space]{grffile} % Allows spaces in the graphics file path.
```

If the included graphic's size is unsuitable, the sub-script code may be extended by a scaling parameter, like this (scaling to 0.5 as an example):

```
@{0.5| *some script* }@
```

This will lead to the L^AT_EX code

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

which, in turn, will cause L^AT_EX to scale the image to half its original size. A negative number -0.5 will lead to

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

thus adjusting the image width relative to the current line's width in the document (there is no particular reason for this functionality being encoded by negative numbers; it was just straight forward).

 Note that a sub-script is allowed to be a plain L^AT_EX script itself, which, recursively, allows it to contain its own sub-sub-scripts, and so on. For example, in the long script above, the `pda` script is a sub-script which translates to a L^AT_EX script with an own sub-sub-script.

 Note that, in the context of sub-scripts, the conversion method “Plain generator code” works in a slightly different way than described above. Executing it once will show the plain L^AT_EX script including sub-script notations; executing it twice, will show the actual plain L^AT_EX code where the sub-scripts have been replaced by the “`includegraphics`” commands.

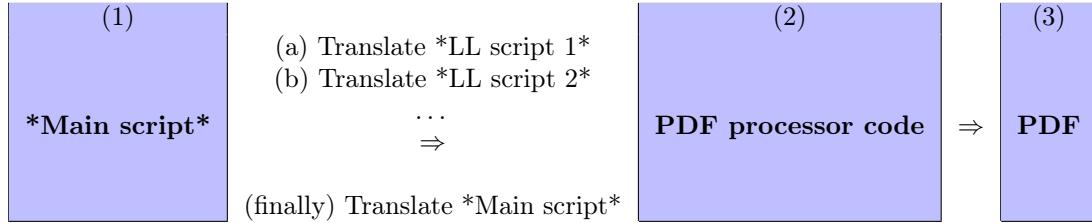
8.2 Pre-Processors

Sub-scripts are implemented on the foundation of the more general **pre-processor** mechanism. Every XWizard script can, in principle, make use of pre-processors in the following way (although only L^AT_EX does so far):

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

There, the declarations part can define an arbitrary number of pre-processors (the according variable names have to start with the word `prep` and can continue arbitrarily). The pre-processor code begins with a file name to store the PDF output in. Behind that, separated by an equals (=) symbol, follows the actual script code which can be an arbitrary XWizard script. Note that this code can, recursively, contain further pre-processors in its own declarations part. The declaration of a pre-processor should therefore always be enclosed in the **securing bracket combination** to ensure its correct interpretation (cf. Sec. 5): `[~(~{ *pre-processor code* }~)~]`

The translation process of a script containing pre-processors works as follows:



When translating a script which includes pre-processors, first the pre-processors will be translated (which, in turn, means that their sub-pre-processors, if any, will be translated before their own main part – i.e., “depth-first”). After all the pre-processors on the different levels have been translated, the main script is translated, as now it can be assumed that the subsequent PDFs exist. Therefore, the main script part can contain code to include the PDF files given in the pre-processors’ codes. The result is a PDF file which may contain sub-images generated by arbitrary XWizard scripts.

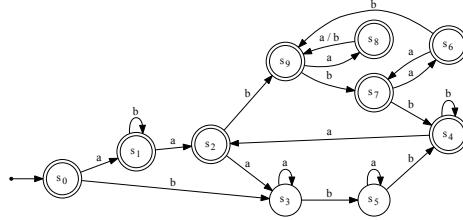
As mentioned before, the main application for the isolated usage of this type of pre-processors (i.e., non-plain-text preprocessors) is the L^AT_EX implementation, where `includegraphics` commands are generated automatically according to the sub-scripts given. Furthermore, simple animations can be created easily using this mechanism as described in Sec. 9. However, many more interesting and much more complex results can be achieved when using this mechanism in its most general form. If sub-scripts can expand (let’s use this word since it’s fairly similar to what happens in T_EX) to plain text (in contrast to just regular objects, i.e., fsm, pda, ...), then they can be used in a way similar to programming in an object-oriented language, cf. Sec. 10.

8.3 Pre-implemented Examples With Compound Objects

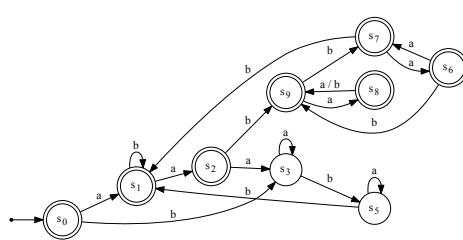
The sub-script mechanism described above is the foundation of several pre-implemented script types. They can be inspected to learn how to use sub-scripts.

For example, FSM scripts can be used to simultaneously show (1) an FSM, (2) a minimized version of it and (3) the according minimization table in a single image. (See figure below; this view is convenient, for example, when creating exam questions, as the difficulty of the question can be estimated more quickly if the additional information is automatically shown while the original FSM’s definition is entered.)

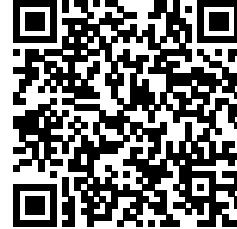
FSM:



Minimized:



SCRIPT 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
s_9	\times_1	\times_2	\times_1	\times_0	\times_2	\times_0	\times_3
s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
				s_8			s_9

This image is created by the FSM script below. Its main part defines the original FSM only, both the minimized FSM and the minimization table are created from it on the fly. The variable declarations “displayMode=1” and “showMinimizedFSM=true” state that these items should be displayed.

```

 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 means don't simulate FSM. */
input=null;
s0=s0;
F=s4,s6,s7,s8,s9,s0,s1,s2;
displayMode=1; /* Show minimization table. */
showMinimizedFSM=true; /* Show minimized FSM. */
showDeterministicFSM=false;
--declarations-end--
  
```

By executing the conversion method “Plain generator code”, the according plain L^AT_EX script can be displayed (this script, as well as the other plain L^AT_EX scripts in this section, are not shown here due to their large sizes). The main script part contains L^AT_EX code which includes the headings and the minimization table (the long code before \begin{document} defines a macro for triangular tables). Furthermore, two sub-scripts of the plain Graphviz type are placed within the L^AT_EX code; they define the graphs for the original and the minimized FSM, respectively.

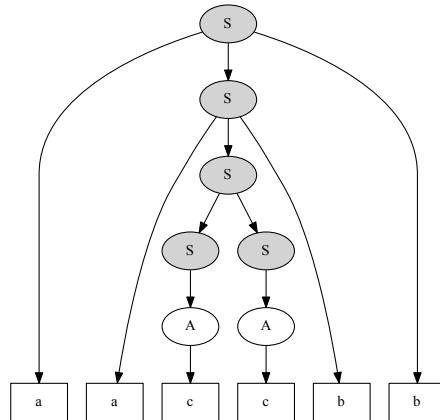
Another example are PDA scripts which, as mentioned above, can be used to create a compound PDF image consisting of two L^AT_EX tables and a Graphviz graph; an according script and image are shown at the beginning of Sec. 3 or online via the following script link (again, the corresponding plain L^AT_EX script can be retrieved using the conversion method “Plain generator code”):

SCRIPT ID-4135



As a last example of a real application, grammar scripts also can produce complex PDF objects. There, a plain Graphviz script (showing a parse tree or different views of the grammar) is embedded within L^AT_EX code showing the Grammar definition:

$$\begin{aligned} G &= (\{A, S\}, \{a, b, c\}, P, S) \\ P &= \{S \rightarrow A \mid SS \mid aSb, \\ &\quad A \rightarrow c \mid AA\} \end{aligned}$$



SCRIPT ID-11087



The script generating this image is shown below, the according plain L^AT_EX script can be retrieved as before via “Plain generator code”.

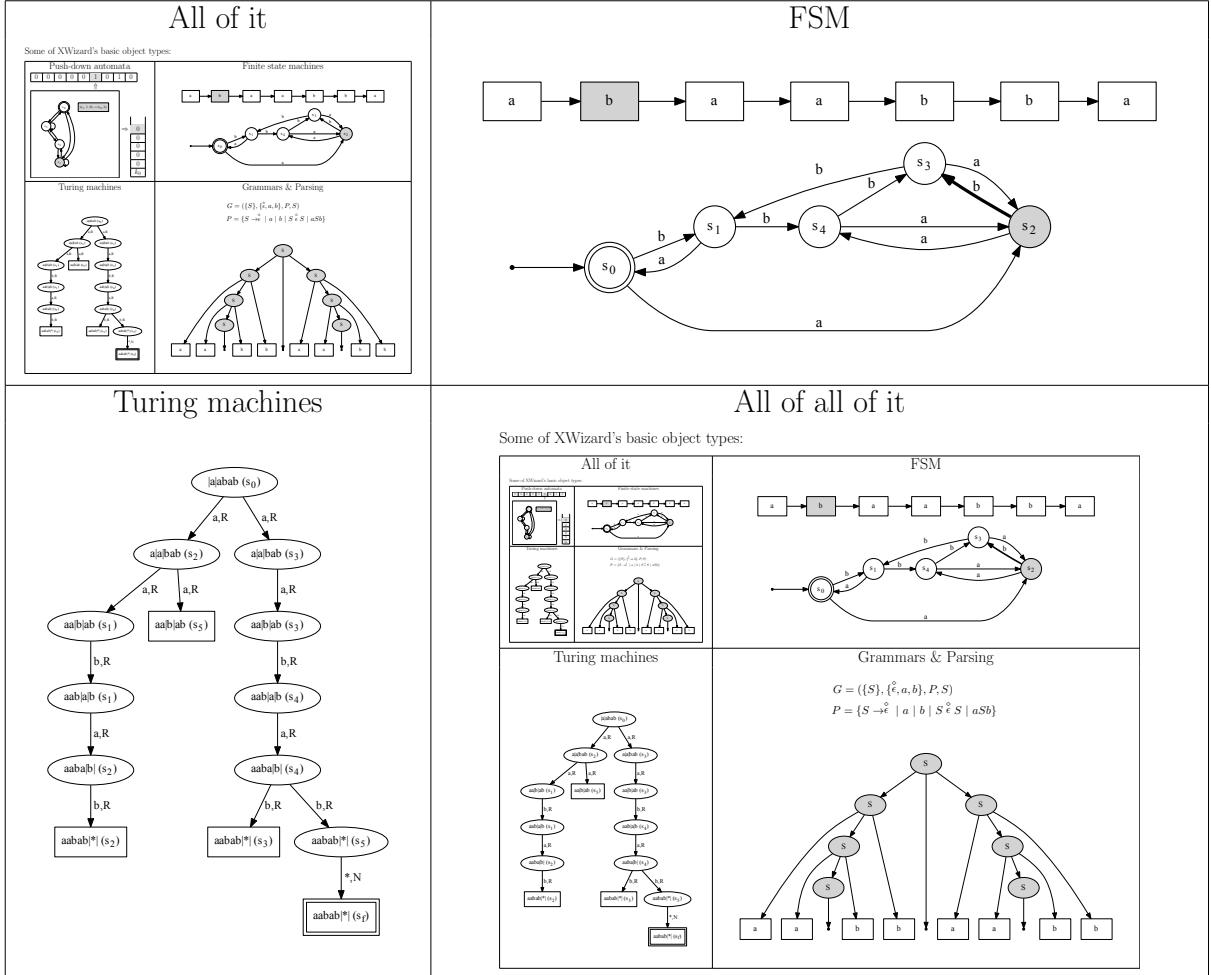
```

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--

```

The following complex and rather artificial example shows how scripts can be embedded recursively to an arbitrary depth within each other:

Some of XWizard's basic object types:



SCRIPT ID-C16107



The translation of the former script and the calculation of the according complex object can take some 10 to 20 seconds. Therefore, the cached version of the object is retrieved when following the above link, which avoids a new calculation – hence the “C” in the ID, cf. Sec. 10.2.

9 Simple Animations

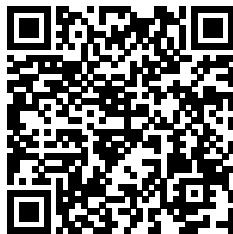
XWizard allows to define animations using features of the SVG image format.

 Animations are therefore only displayed in the SVG output of an object, i.e., particularly when using the XWizard website or the web service, see Sec. 10.3. The current version of VFP also includes an SVG processor (in beta state) to display animations. Conversely, animations in plain PDF are not available; all sub-objects required in the process of creating an animation can be retrieved as separate static PDF files, though.

 XWizard uses the `set` instruction of SVG to create animations, but not the (nicer) `animate` instruction, as the latter is (sadly) deprecated in modern browsers.

So far, animations can be defined as an arbitrary sequence of XWizard objects which are displayed subsequently when the user clicks into the image, like in this example:

SCRIPT ID-C21966



While animations are built upon a quite powerful pre-processing mechanism which offers possibilities to create various types of object sequences to be animated, cf. next section, it is very easy to create basic animations.

9.1 Defining basic animations via script

The general idea behind animations is to assign identifiers, e.g., `x`, `y`, `z`, to XWizard objects. Then, an animation sequence `x->y->z` can be defined in the declarations part as follows:

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

The first object in the sequence, `x`, will be shown when loading the script. Upon clicking into the image, `x` will be replaced by `y` and another click replaces `y` by `z`. Now, the question remains how identifiers can be assigned to objects. Basically, there are two types of objects in XWizard:

- Those implicitly given by the current script; the predefined identifier `this` refers to the object given by the current script.
- Those given by a **pre-processor** in the current script (cf. Secs. 8 and 10.1). To assign an identifier to an object given as a preprocessor, the identifier name has to be placed in front of the pre-processor code, separated by an “equals” sign. There, an identifier can be any alphanumeric string (except `this`). For example, the following assignments are possible:

- In the case of sub-scripts (which can be placed virtually anywhere within a script, cf. Sec. 10):

```
x0=@{ *Any Script* }@
```

- In the case of regular preprocessor definitions (in the declarations part):

```
preprocessor=#x1=@{ *Any Script* }@#
```

or, if there is another identifier `x2` defined:

```
preprocessor=#x1=x2#
```

For example, this script:

```

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

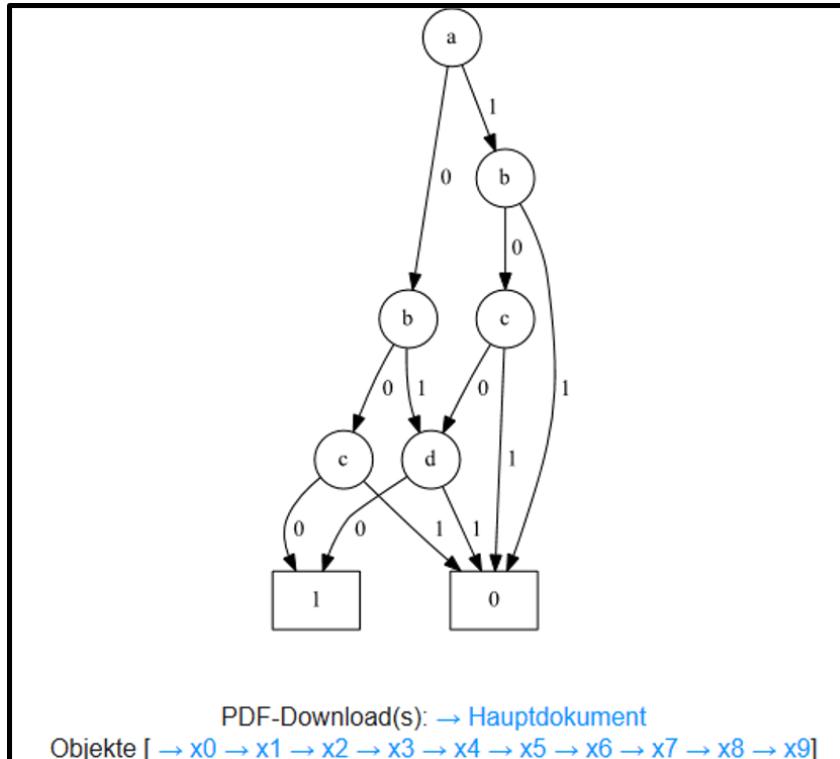
```

will create the following animation containing 11 separate images of “BDD” objects:

SCRIPT ID-21974



There, the download section below the animated image provides not only a link to the PDF of the main document (i.e., `this`), but also links to PDF documents of all the sub-objects, i.e., `x0`, ..., `x9` in this case:



 In addition to explicitly named objects and `this`, the animation sequence may also contain references to individual pages of a multi-page document. For example, if a L^AT_EX script creates a PDF document with five pages, an animation can be created out of these pages like this:

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

XWizard scripts without animations are a special case of those with animations, where the `animate` variable is set to:

```
animate=this;
```

Therefore, this is the default value for `animate`.

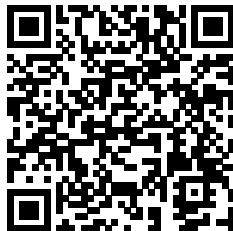
9.2 Conversion methods for creating animations

Several script types provide conversion methods for creating standard animations. For example, finite state machines or push-down automata provide a method which creates an animation of them being **simulated on a given input until termination** (cf. first example in this section; script ID-C21966). It can be created by executing the conversion methods `Animate FSM simulation` or `Animate PDA simulation`, respectively. A similar method is available for binary decision diagrams. These methods insert the following code (in the FSM case; a very similar code in the other cases) into the declarations part of the according script:

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

An example of the effects of this code can be observed here:

SCRIPT ID-22384



The syntax and semantics of this code will be discussed in Sec. 10.1. However, it is important to note that this code will work for any FSM and any input string to simulate it on. This means that, as long as this code is present, the FSM script around can be changed as desired, and the resulting animation will always automatically adjust to contain all simulation steps of this FSM until termination on the given input string. In other words, the only thing the “Animate...” conversion methods do is virtually just inserting these three lines of code into the given script. The same can be achieved by copying them from this page and pasting them into an arbitrary FSM script by hand. All the actual animation semantics comes from XWizard’s pre-processor mechanism.

This may sound like an irrelevant technicality – but it’s not! An important benefit of leaving all the work to pre-processors is that all imaginable animations (well, most) can be created by only altering XWizard scripts. There is no need to go into the Java sources, compile, deploy etc.

So far, these are the only available conversion methods for standard animations, but more and different types are planned – and much more is possible when coding directly without using conversion methods.

10 Advanced Usage: Cool Stuff and Crazy Hacks for Neat Guys

The functionality described in this section is not so crazy, actually, but it is the type of things that are usually wrapped up by a developer to be used as a nice-to-look-at package by regular users. Nevertheless, everything in this document can be performed on the regular XWizard script level, there is no need to dive into Java etc. Therefore, all the stuff here is considered standard XWizard functionality – as opposed to the contents of the “XWizard Developers’ Handbook”. **So, don’t be scared and continue reading!**

10.1 The XWizard Script Language 2.0 – Everything is an Object

The script language described so far can – plain chronologically – be seen as the first version of the XWizard script language. It already includes a basic support for pre-processors (cf. Sec. 8.2), but only as a means to embed the image of an XWizard object into another XWizard object. Furthermore, it includes the possibility of encoding the execution of a conversion method inside the script it's supposed to be executed on – however, in a rather clumsy, hardly generalizable way (cf. Sec. 4.3).

The XWizard script language 2.0 combines these two properties by allowing conversion methods to be executed on sub-scripts within another script. Furthermore, as conversion methods can return not only another script, but also a plain-text output as result (depending on the method type, cf. Sec. 4), it makes sense to allow conversion methods to be executed on arbitrary parts of a script (not only parts that encode a regular XWizard object). Put plainly, (nearly) everything can be used as an “object”. This allows for the implementation of programming structures such as “for loops” or recursive method calls which work on arbitrary strings. Overall, arbitrary computable functions can be solved using this mechanism, meaning that XWizard is Turing-complete.

This sounds good – and it is good; nevertheless, plain XWizard script is probably not a very convenient way of computing complex functions. It is not efficient, and non-trivial functions tend to get quite cryptic. It has never been a developmental objective to make the language so expressive to allow for arbitrary computations – it rather happened as a side effect of providing a set of convenient functions in a clean way. (Basically, it was the result of cleaning up the first pre-processor mechanism described in Sec. 8.2.) Therefore, use these functions as desired, but please don't complain if some things are not as nice as in, say, Java or Python...

If desired, a nicer language (such as lua) could easily be implemented within XWizard, in a similar way, lua is implemented in [Lua^{AT}EX](#).

As first informal examples, let's look at how a “for loop”-like behavior and an `if` statement can be implemented using plain-text conversion methods.

10.1.1 Informal Examples: the for Loop and the if Statement

The following code snippet is a valid FSM script:

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

When ignoring the “`for`” part (that is, `@{sX | }@.for[X, 1, 4]`) and the preamble, the remaining string `(s0, a) => s5;` is just plain old classic fsm code which creates a transition from state s_0 to state s_5 when reading a on the input tape. The part `@{sX | }` looks like a sub-script (cf. Sec. 8.2), but it is not a valid script of one of the classic types.

Internally, plain-text parts are handled as scripts of a special type called `DummyRepresentable`. These are objects of the same super type as regular scripts (i.e., `RepresentableAsPDF`), so technically, plain text does not really differ from classic scripts.

However, `for` is the name of a **plain-text conversion method** that can be applied to an arbitrary string. This string, here “`sX |` ”, is called the method's **body** (more precisely, the body of a **method chain** consisting, in this case, of the single method `for`; in general several or zero methods can be applied to a body, see below). The body is given within the bracket combination `@{ *body* }@`, and the method to be applied is given in Java-like notation by a dot, followed by the method's name, followed by a list of parameters in square brackets (just like in the case of the deprecated in-script conversion methods, cf. Sec. 4).

As in Java, a sequence of methods can be applied to an object like this:

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

 Furthermore, if the ***body*** represents a regular XWizard script, all regular conversion methods of this script type can be applied to it. For example, the determinization method can be applied to an FSM script, and the minimization method to the resulting FSM like this:

```
@{ *some FSM script* }@.det.min
```

There **det** and **min** are abbreviations of the full method names **Determinize** and **Minimize**, respectively. The full names can always be used (including white spaces, if present), abbreviations are available for some of the most important methods only.

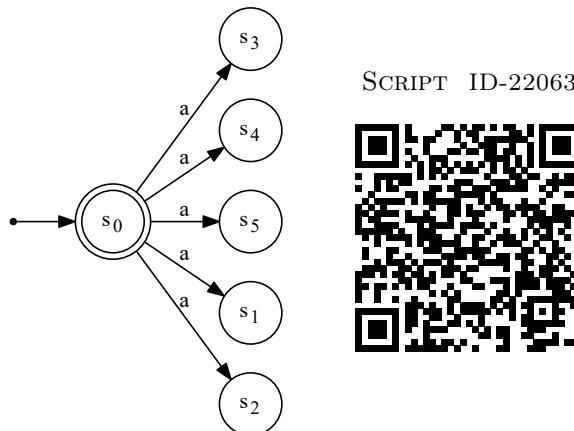
Therefore, the meaning of `@{sX | }@.for[X, 1, 4]` is more or less the following: Apply to the object given by “`sX |` ” the conversion method `for` with the three parameters `X, 1, 4`, and replace `@{sX | }@.for[X, 1, 4]` by the method’s returned result. In the `for` case, the result is given by copying the string within the brackets four times, as the loop variable `X` runs from 1 to 4, and replacing in the string all occurrences of `X` by the current value of the variable. In other words, the string “expands” (let’s borrow this term from `TeX`, although it’s not quite the same) to:

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

and the complete snippet becomes after compilation:

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

Therefore, the resulting object looks something like this:



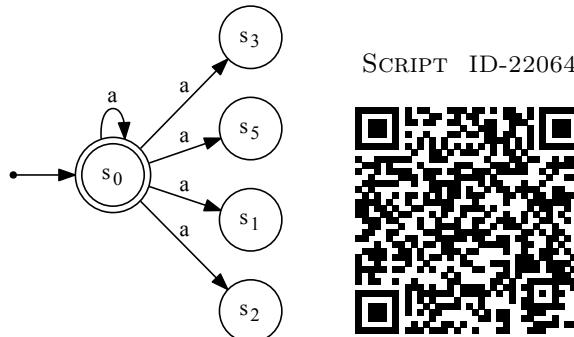
Besides just using a plain loop variable `X` in the string, standard arithmetic expressions such as `X - 1`, `X * 5` etc. can be applied to it if put like this:

```
~{ *expression* }~
```

For example, `fsm: (s0, a) => @{s~{X-1}~ | }@.for[X, 1, 4] s5;` would expand to

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

and hence result in the following object:



The loop variable can be any alpha-numeric string, and it may even contain certain special characters, particularly “#”. As all occurrences of this string will be replaced in the parenthesized string, it is good practice (and will be practiced from now on) to use variable names such as #X, #Y etc. to make undesired replacements unlikely. Note that this will never collide with L^AT_EX’s notation of # or ## as macro parameter prefixes, as XWizard’s compiler runs **before** a potential L^AT_EX run, removing all # characters during expansion. (Except, of course, if a macro is **created** within a for method; then the loop variable might replace parts of the parameter names – which can be avoided by just using a different loop variable.)

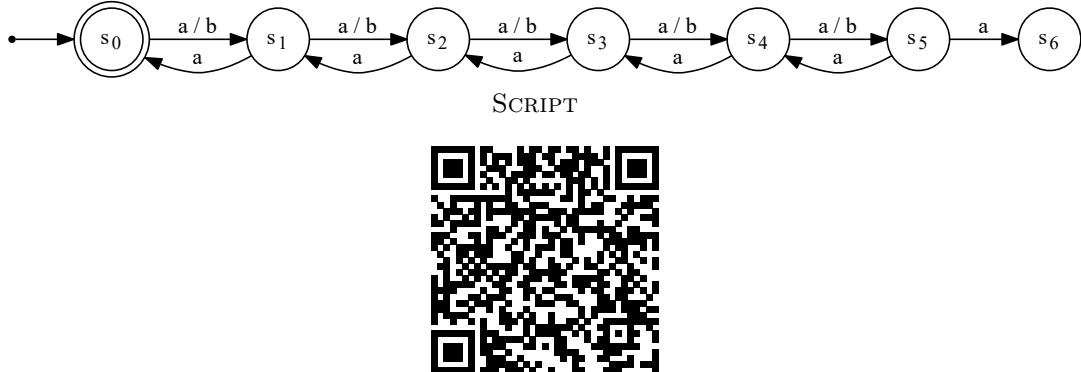
As a final example of the **for** loop, let’s look at the following more complex FSM script:

```
fsm:
(s0,a)=>s1;
@{(s#v, a) => s~{#v-1}~ | s~{#v+1}~; (s~{#v-1}~, b) => s#v;}@.for[#v, 1, 5]
```

There, the body contains two separate transition definitions, cf. single-underlined parts, one of which transitions from the state numbered #v when reading *a* to both the state numbered #v-1, and the state numbered #v+1 (cf. double-underlined parts), and the other of which transitions from the state numbered #v-1 when reading *b* to the state numbered #v. As the variable #v loops from 1 to 5, the **for** method expands to (with line breaks included for nicer readability):

```
(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;
```

This results in transitions between states numbered from 0 to 6, and the whole script including the middle line creates this object:



Besides **for** loops, conditional decisions in an **if-then-else** manner can be interesting in the creation of scripts. XWizard provides an **if** conversion method implemented with the following syntax:

```
@{ *body* }@.if[ *expression* ]
```

The body can be an arbitrary string, representing, as a whole, the ***then case***, or it can contain two parts like this:

```
@( *then case* )@ @( *else case* )@
```

The **if** method expands to the ***then case*** if the ***expression*** expands to the string **true**. Otherwise, it expands to the ***else case***, which, if not explicitly given, is defined to be the empty string.

Typical methods that can be used in the ***expression*** include:

- **smeq[x, y]**, resulting in **true** iff $x \leq y$.
- **sm[x, y]**, resulting in **true** iff $x < y$.

- `greq[x, y]`, resulting in `true` iff $x \geq y$.
- `gr[x, y]`, resulting in `true` iff $x > y$.
- `eq[x, y]`, resulting in `true` iff $x = y$.
- `neq[x, y]`, resulting in `true` iff $x \neq y$.
- Less typical, but nice: `isPrime` which expands to `true`, iff its body is a prime number.

Note that these methods (like all methods) have to be applied to an object; except for the `isPrime` method, this can be any object as it has no influence on the expansion; usually `this` will do fine:

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

An example of a simple IF statement is:

```
@{Yes, 2 is smaller than or equal to 3}@.if[this.smeq[2, 3]]
```

For such expressions to work, conversion method parameters are allowed to contain calls to other (plain-text) conversion methods. For example, something like this:

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

 is a valid expression (where `this.states` expands to the number of states, e.g., of the current FSM). It can have unexpected results, though, if the loop itself is involved in the creation of the actual “`this`” object, as has been the case in the three above examples of the `for` method. Then, the states created by the loop obviously cannot be counted by `this.states`. The evaluation order of sub-scripts and pre-processors is explained in the next section (this is the part that can actually get a little crazy, as methods can be applied to all parts of a script, even within the declarations).

10.1.2 The XWizard 2.0 Syntax and Semantics

The XWizard script language 2.0 is quite powerful, but admittedly it can get a little confusing in the beginning, and sometimes there may be unexpected outcomes such as the one described in the box above.⁵ In the following, a semi-formal overview of the syntax and semantics of the XWizard language is given, without going too deep into details. The XWizard developer’s handbook is planned to provide more comprehensive information about implementation details.

Syntax The new parts of the XWizard language that are described in this section all rely on the concept of **pre-processors** and **sub-scripts** (i.e., **inscript pre-processors**), as explained in Sec. 8.2.

More precisely, they rely on the more general concept allowing plain-text method calls to pre-processors, as shown in the examples of the last section. The distinction between “pre-processors” and “sub-scripts” then becomes rather arbitrary as in the context of plain-text methods these terms merge into each other. This is due to that fact that sub-scripts can be used even in the middle  of a regular pre-processor definition (i.e., a pre-processor definition using a `prep` variable in the declarations part; the animation code in Sec. 10.1.3 is an example for this). Then, additional pre-processors may be created automatically as the final pre-processor definition is expanded. From now on, “pre-processor” will be used as the most general term, while “sub-script” will only refer to expressions used in the middle of a script like this:

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

⁵The reasons for this are two-fold. Firstly, since XWizard scripts can include code of other programs, namely L^AT_EX and Graphviz, its constructs should be fairly unique to not collide with the syntax of these programs – which would lead to a lot of escaping. Therefore, e.g., bracket combinations that include an `@` symbol have been chosen as they virtually do not appear in non-XWizard code. The second reason is that the XWizard language grew rather organically, without a formal syntax or semantics, up to a point where the complexity became barely treatable. After a major restructuring of the code, it once again became treatable, and there are (quite certainly) no major bugs in the current version; most of the unintuitive effects were eliminated by this as well, and the expansion process is quite clear and simple now. Lukas König is currently trying to find some free time to actually provide a formalization of the XWizard semantics.

The idea of pre-processors is basically, to allow creating XWizard objects inside of other XWizard objects which can be referenced at different places or manipulated by applying methods to them. Therefore, pre-processors can be given names which can be used as identifiers to refer to the according object like a variable in common programming languages. As explained in Sec. 9, every regular pre-processor definition includes an identifier name `x` like this: `prep=#x=@{ *some object* }@#`. In the case of subscripts, an identifier name can be simply put in the beginning of the sub-script code, followed by an equals sign. The syntax of sub-scripts is therefore (omitting the optional “scale” in the beginning of regular scripts, cf. Sec. 8.2):

$$\begin{array}{c}
 \text{*identifier name*} = \text{*object reference*} \text{*method sequence*} \\
 \underbrace{\quad\quad\quad}_{\text{optional}} \quad \underbrace{\quad\quad\quad}_{\text{@{*identifier name*}@ empty or}} \quad \underbrace{\quad\quad\quad}_{\text{.*method name*[p1, p2, ...]}} \\
 \text{or actual object} \qquad \qquad \qquad \text{*method sequence*}
 \end{array}$$

There, `*identifier name*` can be any alphanumeric string (including `this` which is, however, forbidden **before** the equals sign; as part of a method parameter, an `*identifier name*` can be given without the surrounding brackets as well). An “actual object” is given by

```
@{ *script* }@
```

where `*script*` can be a regular XWizard object or just any plain string. According to the actual type of `*script*`, the available methods for the leftmost method in the `*method sequence*` are changing (for example, an FSM script will only allow FSM methods to be called on it); in turn, the leftmost method’s return value determines the available methods for the next method in the chain and so on. Every conversion method available for a certain script type can be called by using its plain English name (including blanks). However, for the most important methods abbreviations are available, such as `det` for `Determinize`, `min` for `Minimize` and so on, cf. Sec. 10.1.4.

After the closing bracket `@{ *script* }@` of a sub-script `*script*`, arbitrary many stars `**...` can be appended like this: `@{ *script* }@***.m1.m2...`. The number of stars determines the priority of the script during expansion (the more, the higher), meaning that the regular expansion order (left to right, inner to outer) can be controlled by giving a lower-priority script one or more stars. Details follow in the “Semantics” paragraph.

Method parameters (if any) are put in square brackets after the method name and separated by commas. A parameter can be anything from a simple constant to a large text fragment, and it can itself contain subsequent method calls. Simple parameters can just be put plainly as in:

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

Long strings which can include white space, commas and most other special symbols, can be put in quotation marks as in:

```
setLongText["a long text, please", simplePar2]
```

Even more complex strings can be put within the secure bracket combination:

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

Where `strange`]par {@})1 is the string interpreted as the actual parameter value. Nested method calls can be put just as anywhere else as:

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

where `x` would have to be set as the identifier of some object elsewhere.

Identifiers can be used plainly as in `[...x.inputLength...]` within method parameters. At other places in the script, they have to be put in brackets: `@{x}@.inputLength`.

Hint: if a parameter is supposed to be the **string var** although an identifier **var** is already defined (rather than the **value** of **var**), an empty sub-script or the method **idd** can be used as follows:



```
var=@{varValue}@          /* Parameter is "varValue" */
@{...}@.myMethod[var]      /* Parameter is "varValue" */
@{...}@.myMethod[@{var}@]    /* Parameter is "var" */
@{...}@.myMethod[@{@}@var]  /* Parameter is "var" */
@{...}@.myMethod[@{var}@*.idd] /* Parameter is "var" */
```

The star in the last line lets **var** be evaluated before the assignment has been performed. The method **idd** marks the result as plain text. (The same could be achieved by `[@{@}"{var}"@@*]` – not by just `[@"{var}"@]`, though.)

Besides sub-scripts, explicit pre-processor definitions can be given in the declarations part using the **prep** variables. Most of the syntax described above applies there, too. So, for example, the following would be valid pre-processor definitions:

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

This is so far as much as there is to be said about the XWizard script syntax. As mentioned earlier, there is no formal grammar defining the language of all correct XWizard scripts, so this semi-formal description has to suffice for now (more details will be given in the Developer's handbook; and of course, the full truth can be found in the Java sources).⁶ The next paragraph describes how the expansion of sub-scripts and the evaluation of pre-processors work.

Semantics The following semantics description, too, is semi-formal and supposed to give a broad overview of what is happening only. Basically, the translation of an XWizard script is a rather simple process. A script *S*, possibly containing pre-processors, sub-script parts and declarations, is subject to the following sequence of actions:

- 1.) Cut out the script preamble from *S* (for example `fsm:`).
- 2.) Set the declared variables to **preliminary** values. Meaning:
 - set default values for all variables, and
 - overwrite the values of those that are already completely given in the declarations part of *S*. Particularly, regular pre-processor variables are evaluated now if they can be interpreted completely. (What “completely” means is somewhat subtle, but not really important here; basically it means that for every snippet `varname==sth;` XWizard tries to assign `*sth*` to the variable `varname`. If the variable exists, and its type matches what is given by `*sth*`, the variable is assigned the according value; nevertheless, the value can be overwritten later if a new assignment of `varname` occurs, possibly after some expansion.)
- 3.) As long as *S* contains sub-scripts, repeatedly do the following:
 - i.) Find the first sub-script *S'* to process as follows. Let *n* be the **highest number of stars** appended to any of the sub-scripts of *S*. (In all example scripts so far, *n* was zero.) Of all the sub-scripts with *n* stars, *S'* is the inner-most sub-script of the left-most top-level sub-script in *S*.

⁶This rough syntax description is certainly not unambiguous, but let's be realistic – the number of people who actually came so far as to this page in reading this document must be so small (and hence my appreciation for them so big) that I'll gladly answer all their questions via email.

- ii.) If S' is a plain-text script or not within the declarations part of S , replace S' with its respective expanded result (possible sub-sub-scripts in S' are not expanded yet if S' is a plain-text script; otherwise the star-based priority would be corrupted).

 In the non-plain-text case, each script type decides individually how to expand sub-scripts. For now, only the L^AT_EX-based `\includegraphics` expansion is implemented, cf. Sec. 8.2. Non-plain-text methods obviously only make sense outside the declarations.

 If S' contains the identifier `this`, the respective object is defined to be S without any sub-scripts. If the resulting script becomes syntactically incorrect due to cutting out the sub-scripts, it is translated “as far as possible” – which depends on the lower-level script processor.

Note that by default, identifier definitions expand to the object they are assigned. So assigning `x` some object will always lead to the object occurring at the position of the assignment. The method `nil`, which always expands to the empty string, can be used to cut out these objects after the assignment:

```
@{test}@          /* Expands to 'test'. */
x=@{test}@        /* Expands to 'test' and defines x to be 'test'. */
@{x=@{test}@}.nil /* Expands to '' and defines x to be 'test'. */
@{x}@            /* Expands to 'test' with any of the 2 above lines. */
*anything*.nil    /* Always expands to ''.
```

- iii.) Look if there are new variables that are now completely defined in the declarations part and set them like before.

After the expansion process has terminated, no sub-scripts are left in S and all variables from the declarations part have been set to their final values. Then, the declarations part is cut out of S , and only the main part of the final script is given to the actual individual script processor (such as FSM or PDA).

 Note that, when talking about scripts “expanding to plain text”, this only makes statements about the **last method of a chain**. For example, in a chain like:

```
@script@m1.m2.m3.m4
```

only `m4` will determine if the whole code expands to a script or to plain text. All the in-between methods `m1` to `m3` may, principlly, switch between regular scripts and plain text.

What do we actually need the star for? There are many situations where it makes sense to allow for controlling the expansion order (for example, look at the two method definitions for `fac` and `fib` in Sec. 10.1.4). But it can be made very obvious at the example of for loops which would not work without it when containing nested sub-scripts. Within a L^AT_EX script, the following FSM sub-script might be expected to produce the images of three FSMs, each transitioning from `s0`, when reading `a`, to a different target state `s0`, `s1` or `s2`:

```
@{fsm: (s0, a) => sv;}@.for[v, 0, 2]
```

However, this is not what happens. As `for` is a plain-text method, the result will be the three **scripts** of the respective FSMs:

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

 The first idea coming to mind – putting additional brackets around the body to tell XWizard that the result is a script – would indeed lead to subsequently expanding the scripts into images:

```
@{@{fsm: (s0, a) => sv;}@}@.for[v, 0, 2]
```

But this time, the result would be three times the same automaton where the `(s0, a)` transition points to `sv`. This is due to the fact that XWizard expands sub-scripts in a depth-first order, meaning that `fsm: (s0, a) => sv;` will be processed, before the for loop on the next-higher level had a chance of replacing `v` by the actual looped values. Using the star as follows:

```
@{@{fsm: (s0, a) => sv;}@}@*.for[v, 0, 2]
```

XWizard can be forced to first expand the outer sub-script using the `for` method to:

```
@{fsm: (s0, a) => s0;}@{@{fsm: (s0, a) => s1;}@{@{fsm: (s0, a) => s2;}@}
```

Afterwards, these sub-scripts are translated to the three **different** automata as desired. The three different outcomes can be observed via script ID-23643.

After this semi-formal, moderately fuzzy description, the XWizard semantics should be fairly clear. Try, as a self-test, to figure out what this extra-cryptic FSM script involving the identifier `x` and a loop variable named `#x` does:

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

The solution can be looked up here:

SCRIPT ID-23645



10.1.3 A more advanced Example: Animate to Termination

The XWizard language 2.0 allows to express quite complex statements which, for example, facilitate the creation of sophisticated animations. To demonstrate this, let's once again look at the code snippets produced by the “Animate” commands mentioned in Sec. 9.2. In general, these snippets look very similar

for all the different script types that provide such methods. In the FSM case, the `Animate Simulation` method creates the following code which, when inserted into the declarations part of a script, creates an animation of the current FSM, simulated on the current input word (cf. script ID-C22384):

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

The code may seem a little overwhelming at first sight, but when looking more closely it can well be understood with the knowledge of the XWizard syntax and semantics provided above. First we observe that the top line is plain pre-processor code which just assigns to the identifier `x0` the current FSM (`this`) simulated for one step (using the `sim` method). So, just with this pre-processor we would be able to create an animation of only the first simulation step by assigning the `animate` variable as follows:

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

Now, we would like to have so many additional pre-processors `x1`, `x2`, `x3`, ..., `xn` that all the steps `n` to be simulated are captured, each pre-processor representing the FSM being simulated one step further than the last. As (ϵ -free) FSMs run for as many steps as the number of symbols in their input word (+/- one or so), we need the indexes `i` of the identifiers `xi` to run from 1 to the length of the input word. The method `inputLength` provides this length, and a `for` loop can now be used to create the code for the required pre-processors. The middle line

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

accomplishes this. The loop variable `A` runs in the desired range, which makes the string it is invoked on

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

expand to:

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

up to pre-processor `prepn`, if `n` is the input length. Put more nicely with line breaks, we get:

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

We invoke the method `inputLength` on `x0`, and not on `this`, because for `this` no input (`input=null`) might have been defined. In this case, the first method in the top line of the animation code would have to be different from the other ones inside the loop. Rather than the parameter-less `sim` method it would have to be, for example, `sim[*string*]` where the parameter `*string*` provides the word to simulate on.

This is exactly what we wanted as, when counting the first pre-processor `x0=this.sim`, we now have all the objects to create the animation from the initial state to termination. The only code still missing is the animation code itself:

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

up to `xn`. This code is created by the bottom line of the animation code:

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

The loop variable `B` runs from 0 to `n=x0.inputLength`, which makes the inner string the loop method operates on

`->xB`

expand to

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

When including the part before the loop, `animate=this`, and the part after the loop, `;`, we get the correct animation line as desired:

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

Overall, the three lines of code expand to (line breaks included for readability):

```
prep0=#x0=this.sim#;           /* Top line */
prep1=#x1=x0.sim#;           /* Middle line */
prep2=#x2=x1.sim#;
prep3=#x3=x2.sim#;
:
animate=this->x0->x1->x2->...; /* Bottom line */
```

When looking, more generally, on all the different script types providing this type of animations, the code can be generated in always the same manner, only depending on three parameters: The method names `m1`, `m2`, `m3`, for:

-  (1) the first call on `this` (performing the first step in whatever type of change is desired to be animated), put as `this.m1` in the top line;
- (2) the other calls to the subsequently created objects (performing the rest of the steps to be animated), put as `xA=x~{A-1}~.m2` in the middle line;
- (3) the plain-text method returning the maximum number of objects to be created, put as `x0.m3` in the middle and bottom lines, as third parameter of the `for` loops.

 **Debugging hint 1:** It can help to click one of the script formatting methods (i.e., `Format script` or `Add declarations to script`). This will expand the pre-processors, at least as far as they are syntactically “tolerable” (which is far more than what would be considered “correct” by an actual parser).

 **Even better debugging hint 2:** The method `Stepwise script expansion` creates a `LATEX` document with all the steps in the script translation process from the raw script given by the user to the final script which is subsequently translated by the subordinate script processor.

 **Another debugging hint 3:** The “preprocessor tree”, representing the nesting hierarchy of all the sub-scripts in the final script, can be retrieved by the method `prepTree`. It may also help to just look at the pre-processors of a script using the methods `get[prep]` (for only the explicitly named pre-processors of the respective sub-script), `get[preph]` (for all pre-processors, even the “hidden” ones created automatically, of the respective sub-script) or `get[prepa]` (for all pre-processors created in the current run, not only the ones belonging to the sub-script which `get` is called upon).

10.1.4 Important Methods (making XWizard Turing-complete)

Tab. 1 lists the most important plain-text methods generally applicable to all scripts (or at least to all plain-text scripts) in XWizard, available at writing time of this document. New methods may have been added in the meanwhile, but this list is supposed to be kept fairly up-to-date. Non-plain-text methods are not covered here as their functioning usually depends on the script type they work on.⁷

An important method, from a technical point of view, is the `newMethod` method. It can be used to define new customized methods within the script. For example, the following code can be used to define a new conversion method `fak` such that `@{n}@.fak` expands to the factorial $n!$ (using the simple recursive rule that $1! = 1$ and $n! = n \cdot (n - 1)!$ for $n > 1$):

```
@{@{
@{(1)@                                /* THEN case. */
@(@{@{@#0#}@.sub[1].fak}@.mult[#0#])@ /* Recursive ELSE case. */
}@*.if[this.smeq[#0#, 1]]}@**.newMethod[fak, 0]
```

⁷ Meaning they are far to many and far to complex, and they are described in full detail on the XWizard website: <http://www.xwizard.de:8080/Wizz?help&lang=eng>.

Table 1: Important plain-text methods in XWizard.

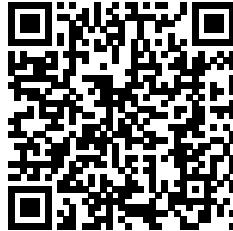
Method	Description	Note
<code>obj.for[#a, nb, ne]</code>	Runs variable <code>#a</code> from <code>i = nb</code> to <code>ne</code> , copying <code>obj</code> and replacing each occurrence of <code>#a</code> by <code>i</code> .	
<code>x.if[exp]</code> <code>@{@(x)@ @@(y)@}@.if[exp]</code>	Both versions expand to <code>x</code> if <code>exp = true</code> .	<code>exp = false</code> yields <code>y</code> , if given, or else the empty string.
<code>smeq[x, y]</code> <code>sm[x, y]</code> <code>greq[x, y]</code> <code>gr[x, y]</code> <code>eq[x, y]</code> <code>neq[x, y]</code>	Expands to <code>true</code> if <code>x ≤ y</code> , and to <code>false</code> otherwise. Expands to <code>true</code> if <code>x < y</code> , and to <code>false</code> otherwise. Expands to <code>true</code> if <code>x ≥ y</code> , and to <code>false</code> otherwise. Expands to <code>true</code> if <code>x > y</code> , and to <code>false</code> otherwise. Expands to <code>true</code> if <code>x = y</code> , and to <code>false</code> otherwise. Expands to <code>true</code> if <code>x ≠ y</code> , and to <code>false</code> otherwise.	These methods do not interact with the object they are called on. Easiest usage: call on <code>this</code> . Non-integer parameters <code>x, y</code> cause exception.
<code>x.add[y]</code> <code>x.sub[y]</code> <code>x.mult[y]</code> <code>x.div[y]</code> <code>x.mod[y]</code>	Expands to the integer value of <code>x + y</code> . Expands to the integer value of <code>x - y</code> . Expands to the integer value of <code>x · y</code> . Expands to the integer value of <code>x/y</code> . Expands to the integer value of <code>x mod y</code> .	Non-integer parameters <code>y</code> or non-integer objects <code>x</code> cause exception. The result of <code>x.div[y]</code> is rounded down to next integer.
<code>x.id</code>	Expands to the script represented by <code>x</code> . This is <code>x</code> itself, if <code>x</code> is plain text. Otherwise, it's the script of the object represented by <code>x</code> . Can be used to retrieve the script after the application of plain-text methods, e.g., <code>*FSM*.det.min.id</code> .	Can have side effects for non-plain-text scripts, though, e.g., cutoff of inner declaration parts. Then, <code>idd</code> can be used, see below.
<code>x.idd = @"{x.id}"@</code> <code>x.nil</code>	Same as <code>id</code> , but puts result in plain-text tags. Expands to the empty string.	To avoid further processing.
<code>x.get[var]</code>	Retrieves the value of variable <code>var</code> .	E.g., from the decl. part of <code>x</code> .
<code>x.prepTree</code>	Retrieves the nesting tree of all sub-scripts of <code>x</code> .	
<code>b.newMethod[nam, n]</code>	Uses <code>b</code> as “body” to create a new plain-text method named <code>nam</code> with <code>n</code> parameters. The body can be an arbitrary script containing sub-scripts etc. The new method can be used subsequently as <code>x.nam[p1, ..., pn]</code> . It expands to the body <code>b</code> where every occurrence of the parameter pattern <code>#i#</code> is replaced by <code>p_i</code> . (<code>x</code> is considered zeroth parameter <code>p₀</code> .)	Should be prioritized higher than <code>b</code> using stars. <code>nam</code> can be called recursively within its own body. This is the key mechanism providing Turing-completeness.
<code>b.newMethodD[nam, n, d]</code>	Same as above, but <code>d</code> sets parameter pattern.	Default: <code>#n#</code>
<code>b.sethard[c]</code>	Lets every future occurrence of <code>b</code> be expanded to <code>c</code> instead of regular expansion: <code>@{1.fib}@.sethard[1]</code>	Can be used, e.g., for dynamic programming. Expands to <code>c</code> .

There, it is crucial to define a correct expansion order. While it can be quite subtle to make it right on every level within the method body (the example wouldn't work without the additional brackets around `@{#0#}@.sub[1].fak` in the third line), it is always necessary to let `newMethod` expand before any sub-script in the body. I. e., `newMethod` has to be given one more star than the most stars used in the body (no sub-script at all counts as “-1 stars”, so `newMethod` needs no star in that case). Another example is the following method `fib` such that `@{n}@.fib` expands to the n 'th Fibonacci number (in a very inefficient manner, using the naive recursive rule that $fib(n) = n$ for $n \leq 1$ and $fib(n) = fib(n - 1) + fib(n - 2)$ otherwise; it's just a proof of concept with exponential runtime):

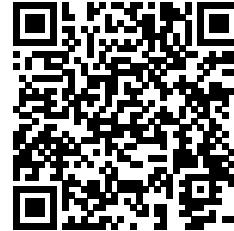
```
@{@{
@(#0#)@
/* fib(n)=n for n<2 */
@{@{@{@{@#0#}@*.sub[1].fib}@*}@.add[@{@{@#0#}@*.sub[2].fib})@ /* Rec. expansion */
}@**.if[this.smeq[#0#, 1]]}@***.newMethod[fib, 0]
```

The correct expansion order is crucial here, too. These subtleties are not discussed here, though, as they are too technical for the “Teacher’s Handbook”. The following scripts show how the new methods `fac` and `fib` can be used:

SCRIPT ID-23844



SCRIPT ID-23830



Both these two methods serve as a proof of concept only, but what they prove as a side effect is how confusing complex functions tend to get in XWizard. Therefore, the `newMethod` method’s main application area will (similar to `\newcommand` in L^AT_EX) probably be mostly among the simpler use cases, for example to avoid double code as in:

```
@{This is a text I will use slightly differently #0# times}@.newMethod[cp, 0]
```

The snippet

```
@{100}@.cp
```

will then expand to:

```
This is a text I will use slightly differently 100 times
```

Without going too much into detail note that the Fibonacci function actually can be made efficient in the same way as is done via the concept of **dynamic programming** in many programming languages. For this, the method `@{}@.sethard[...]` can be used to store the already calculated values of `fib` and reuse them when they are needed in future. Doing so, the exponential runtime of the above approach becomes (nearly) linear. The `fib` method then looks like this:

```
💡 @{@{
@(#0#)@
@{@{@{@#0#.fib}@.sethard[
@{@{@{@#0#}@*.sub[1]}@*.fib}@*}@.add[@{@{@#0#}@*.sub[2]}@*.fib])@
}@**.if[this.smeq[#0#, 1]]}@***.newMethod[fib, 0]
```

The underlined part is new, the parameter of the `sethard` method is basically the same code as before in the non-dynamic case. (It only requires some additional brackets – for the usual “technical” reasons.)

Tab. 2 shows some abbreviations for methods that would otherwise be very clumsy to use. Also, some important script-specific plain-text methods are listed.

Table 2: Method name abbreviations and script-specific plain-text method names.

Script type	English method name	Abbreviation	Note
FSM	Simulate one step	sim	
	Determinize	det	
	Minimize	min	
	Randomize	rand	
	Randomize (seed)	randD	
	Regular Expression	regex	
	-	states	Retrieves the number of states.
PDA	-	inputLength	Retrieves the length of the input word.
	Simulate one step	sim	
	-	states	Retrieves the number of states.
BDD	-	maxSteps	Retrieves the number of steps till termination.
	Simplify one step	simp	
	Truth table (Latex)	truthTableLatex	
	Truth table (JavaPDF)	truthTableJava	
-	-	max	Retrieves the steps required for simplification.

Some final technical notes regarding the methods `id` and `idd`. They are, functionally, similar and very simple (both basically return the script of the object they are called upon), but one subtlety has to be considered:

- If used on the top level of, e.g., a L^AT_EX script, `*object*.id` will print the script text of `*object*`. However, the compiler will cut off the declarations part as it will mistake it for declarations belonging to the top-level L^AT_EX script. The following method calls will therefore expand to the text of a random FSM's script, but without the declarations part:



```
latex:%varm%
\begin{verbatim}
@{fsm:@.rand[4, false].id
\end{verbatim}
```

In general, the tags `@{"`` and ``"}@` can be used to mark verbatim parts that should not be treated as pre-processors or declarations. These tags are put around the script if using `*object*.idd`, therefore, hiding the declarations during the translation of the L^AT_EX script, and preserving them for the final output.

10.2 The XWizard Cache

To avoid long calculation times, for example when creating complex animations, the generated XWizard object including the actual SVG image can be stored in the database and reloaded from there if desired. Doing this can extremely speed up the generation of objects as the actual calculation is omitted. This temporary storage of objects (temporary as the image might get obsolete with new developments in XWizard) is called **cache**. Not every script's output is stored in the cache for performance reasons, though, using the cache has to be requested. The request is formulated by putting a capital “C” in front of the respective **script** or **script ID** (like this: ID-C22384 – or this: Cbdd:10110001). This leads to finally displaying the respective object as usual, but it has three additional consequences:

- (1) If the script's output is not yet in the cache, it will be compiled and the output will be stored in the cache (showing the output afterward, as always).
- (2) If the script is already in the cache, it will **not** be recalculated, but directly loaded from the cache.
- (3) If a script which is already in the cache is retrieved **without** the leading “C”, it will be recalculated and the object will be updated in the cache.



Several examples in this document use the leading “C” to quickly process the respective scripts.

Try using the “C” on some of the other IDs. It is also possible to use the “C” on a full script. In this case it is put at the very beginning, for example in the case of an FSM script: Cfsm: ...

 Don't forget that the cached version of a specific script might get obsolete if the XWizard implementation changes. On the other hand, the cache may also be used as a way of conservation of objects that should **not** be changed by new implementations. (This works only as long as the respective object is not recalculated; a method of preserving a script from being recalculated when loaded without the "C" is planned on being implemented in future.)

 When using the cache, the PDF documents of the script and its sub-scripts are not generated – and they are not cached either. Therefore, to retrieve the PDF documents, a cached script has to be recalculated first, using the Draw! button.

 As the cache requires database access, it is available in VFP only via the button "Call Web Service" which accesses the Web Service (cf. Sec. 10.3) rather than working locally where there is no database.

10.3 The XWizard Web Service

The XWizard web service is called `DeDescriptor`, and it consists of a single Java method:

```
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;
```

The `DeDescriptor` service is available via `http://www.xwizard.de:8080/services/DeDescriptor`. Calling the above method will translate the script (which may be a script ID) given by the first parameter into an SVG object and the according SVG code will be returned. If one of the middle two parameters are `true`, the plain SVG code will be embedded into a HTML DIV which will contain a link to the XWizard website to load the given script and/or a field showing the script text below the actual graphic. The last parameter determines the language (English or else German) to use in the returned text.

 Note that so far the web service will always try to use the cache, so technically, there is no need to use the leading C, cf. Sec. 10.2. This is, however, not a fixed decision yet and may be changed in future. Therefore, it makes sense to specifically add the leading C when cache usage is desired, particularly for scripts that are supposed to endure for a long time, such as embedded scripts, cf. Sec. ??

The web service can be called in many different ways. An example of how to do it via Javascript is given in the next section. The following code can be used to call the web service from Java.

```
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/DeDescriptor";
            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[] {"ID-10700", "false", "false", "false"}));
            // Do something with retrieved svgString.
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

For more technical details please check out <https://sourceforge.net/projects/xwiz>.

10.4 L^AT_EX abbreviations

An abbreviation scheme has been introduced to make creating L^AT_EX scripts more pleasant. Using the scheme embeds the script text into pre-fabricated L^AT_EX code such that only the part between `\begin{document}` and `\end{document}` has to be explicitly written in the script. A L^AT_EX script which uses the scheme looks like this:

```
latex: %*docclass* | *packages1* | *packages2* | ... | *packagesn*%
      *LaTeX code in the document body*
```

The resulting L^AT_EX code will look something like this:

```
\documentclass[...]{...}
\usepackage{...}
:
\usepackage{...}
\begin{document}
*LaTeX code in the document body*
\end{document}
```

There, the `documentclass` and `usepackage` parameters are determined by the parameters in the abbreviation scheme. The following values are allowed so far:

Scope	Abbreviation	Effect
First parameter (docclass)	<code>artlet</code>	<code>\documentclass[letter]{article}</code>
	<code>var</code>	<code>\documentclass[varwidth, border=15pt]{standalone}</code> <code>\usepackage{varwidth}</code>
	<code>varm</code>	<code>\documentclass[varwidth=\maxdimen, border=15pt]{standalone}</code> <code>\usepackage{varwidth}</code>
	<code>tight</code>	<code>\documentclass[tightpage]{standalone}</code>
Other parameters (packages)	<code>gra</code>	<code>\RequirePackage{graphicx}</code> <code>\RequirePackage{space}{grffile}</code>
	<code>ger</code>	<code>\usepackage[ngerman]{babel}</code> <code>\usepackage[utf8]{inputenc}</code> <code>\usepackage[T1]{fontenc}</code>
	<code>geo</code>	<code>\usepackage[a3paper, margin=1in]{geometry}</code>
	<code>relsize</code>	<code>\usepackage{relsize}</code>
	<code>hyperref</code>	<code>\usepackage{hyperref}</code>
	<code>qrcode</code>	<code>\usepackage{qrcode}</code>
	<code>loop</code>	<code>\usepackage{forloop}</code>
	<code>etoolbox</code>	<code>\usepackage{etoolbox}</code>
	<code>ulem</code>	<code>\usepackage{ulem}</code>
	<code>ams</code>	<code>\usepackage{amsmath,amsfonts,amssymb}</code>
	<code>*otherwise*</code>	<code>*otherwise*</code> is added to preamble (e.g., <code>\usepackage{*mypack*}</code>).

...and, sorry, no, these abbreviations cannot be adapted in any way so far. If you don't like one of them, you'll have to avoid it... But, more often than not, these few commands can extremely shorten the creation of a L^AT_EX script.



Maybe there will be LuaL^AT_EX support in future...

11 Known Bugs, Shortcomings and 'Pitfalls'

This section lists some known bugs, possibly unintuitive things and problematic issues we are aware of, to look out for when using XWizard.

- **Exam questions and critical scripts should not be made publicly available.** When doing so, everybody can load the script by typing its ID into XWizard's script area (although somebody guessing correctly the script belonging to his or her exam seems highly unlikely).
- **The error messaging and debugging systems are still being improved.** So far, the plain java exception trace is displayed if something goes wrong during compilation. Using the Format script or Plain generator code buttons can give some debugging clues if the script compiles well. Furthermore, the pre-processors can be listed (method `get [prep]`) and a sub-script tree can be displayed (method `preptree`). The most recent debugging feature shows the stepwise

expansion of the script it is called on (method `stepwise[true/false]`). When called with the parameter `true`, the currently available pre-processors are shown for each step. In general, debugging is recommended to be done with VFP, not the web version.

- **Code completion and similar features.** The text editor used in the web version is quite powerful and technically allows for such features. They are currently under construction. VFP has some functionality in that area.
- **The only strange bug:** In the web version, an error may occur during script translation (very rarely!) although the script is correct – particularly when many people are working simultaneously with XWizard. The problem is being reviewed, but it has no obvious reason and may well remain for some more time. (The property of “occurring rarely” is a good thing from a users’ perspective, but it certainly does not make bug fixing easier.) **A simple workaround is to just click Draw! one more time; usually it will work then.**
- **A minor issue with the “back” button of the browser:** After using it, downloading the PDF will not work properly. Clicking the “Draw!” button will reestablish this functionality.
- **A minor issue with short URLs:** Scripts created by a conversion method cannot be made web-free immediately, i.e., the `Short URL to this script` method will not work. Either click the “Draw!” button or execute the `Short URL to this script` method again to reestablish this functionality. (Yeah, it’s a little thing, could be fixed in a minute... but it hasn’t been so far.)
- **A technical issue, affecting experts and hackers only:** So far, all identifiers in XWizard are global (except `this`). This is a little unsatisfactory, as it leads to scripts possibly having different effects in different contexts, breaking the semi-object-oriented paradigm introduced. For the future it might be desirable to create scopes for identifiers, “getters” (`this.x, x.y etc.`) returning references to identifiers of a sub-object and – possibly – private vs. public identifiers.
- To be continued.

Have fun with XWizard!

July 17, 2017