FSA Reference Manual

Table of Contents

1. FSA6: Finite State Automata Utilities Version 6

(manual generated with FSA Utilities version fsa6-266)

FSA6 is a collection of utilities to

- **construct** finite automata (from regular expressions)
- **manipulate** finite automata
- **visualise** finite automata
- **apply** finite automata

FSA6 supports a number of different types of automata:

- recognizers
- weighted recognizers (aka string-to-weight transducers)
- transducers (aka string-to-string transducers)
- weighted transducers (aka string-to-string-weight transducers)

1.1 Functionality

1.1.1 Constructing Finite Automata with Regular Expressions

Many basic regular expression operators are provided, both for acceptors and transducers. Moreover, it is easy to define new regular expression operators. The built-in regular expression operators include:

- Concatenation; Kleene star; Kleene plus; Option; Union
- Complement; Difference; Intersection
- Reversal; Containment; Ignore
- Composition; Cross-product; Domain; Range; Identity; Inversion;
- Interval
- ‘Any’ meta-symbol.
- Arbitrary predicates instead of symbols

- operators to construct weighted automata

1.1.2 Manipulating Finite Automata

Tools are provided to manipulate finite automata. Such manipulations include determinization and minimization (both the classical algorithms for recognizers and the recent algorithms for transducers are provided).

- Determinization. Currently there are three different implementations of this algorithm, depending on how epsilon transitions (jumps) are treated. There is also an implementation of Mohri's determinization algorithm, both for ordinary (string-to-string) transducers and string-to-weight transducers. The implementation is described in a paper in Computational Linguistics, available from
  `http://www.let.rug.nl/~vannoord/papers/`

- Minimization. Three different minimization algorithms are supported. There is also an implementation of Mohri's minimization algorithm, both for ordinary (string-to-string) transducers and string-to-weight transducers.

- Random generation of finite automata, based on the algorithm in Leslie (1995).

- Epsilon-removal.

- Completion and Incompletion: extending a given automaton in order to make the transition table total (typically by adding a sink state and adding transitions to this sink state); and removing transitions leading to sink states.

- Regular manipulations. The various regular expression operators can be applied to automata directly as well.

1.1.3 Applying Finite Automata

- Acceptance. Tools to check a given string for acceptance by a recognizer.

- Transduction. Tools to apply a transducer to a given input string.

- Production. Tools to produce strings of a given recognizer, and pairs of strings for a given transducer.

- Code Generation. Tools to compile finite automata into efficient Prolog, C, C++, and JAVA programs which can be used to check whether a given string is in the language defined by the automaton, or to generate the transduction of a given string w.r.t. a given transducer.

1.1.4 Visualizing Finite Automata

Much attention has been paid to be able to visualize finite state recognizers and finite state transducers. Support includes built- in visualization and interfaces to third party software:

- DOT. The program is able to produce a representation of a finite automaton compatible with the DOT graph visualisation program. DOT (part of AT&T's graphviz) is a tool that automatically figures out how a graph is best displayed (crossing-edges reduction, etc). It can produce e.g. Postscript output. An example is `http://www.let.rug.nl/~vannoord/Fsa/Manual/dot.png`.

- VCG. The program is able to produce a representation of a finite state automaton compatible with the VCG graph visualisation program. VCG is a tool that automatically figures outhow a graph is best displayed (crossing-edges reduction, etc). An example is `http://www.let.rug.nl/~vannoord/Fsa/Manual/vcg.png`. The VCG program is now also known as aiSee.

- daVinci. The program is able to produce a representation of a finite state automaton compatible with the daVinci 1.4 graph visualisation program. This program automatically computes the most optimal way to view the finite-state automaton by minimizing the number of crossing edges. Postscript output can easily be generated from the result. An example is `http://www.let.rug.nl/~vannoord/Fsa/Manual/daVinci.png`.

- TK Widget. The package contains an interface to a TK Widget to browse finite state automata, providing a graphical user-interface for the toolbox. The TK Widget is explained in much more detail below. An example is `http://www.let.rug.nl/~vannoord/Fsa/Manual/dump.png`. Note that the GUI is not an integral part of the toolbox; it makes perfect sense to use the program in batch mode. The graphical user interface is only available under SICStus.

- LaTeX (if you want to be able to use the result you need the pstricks package). An example is `http://www.let.rug.nl/~vannoord/Fsa/Manual/pstricks.png`.

- Postscript (thanks to Peter Kleiweg). An example is `http://www.let.rug.nl/~vannoord/Fsa/Manual/pk.png`.

1.2 How to use the toolbox

There are a number of ways that the toolbox is meant to be used:

- interactively using a command interpreter and/or a graphical user interface. For example, in order to use fsa interactively with the graphical user interface, use the command:

```
% fsa -tk
```

- as a UNIX-like filter. In such cases you use the fsa command with a number of options. For instance:

```
% fsa write=postscript -r '[a,b+,c*,d]' | ghostview -
```

- as a library in your own Prolog program. You can incorporate the FSA program in your own program, just as you can use other Prolog libraries. In order for this to work, you simply need to load the file fsa_library.pl in the installation directory. For example (if you use SWI Prolog, you need 'consult' instead of 'use_module'):

```
% sicstus
SICStus ...
Licensed to ...
| ?- use_module(fsa_library).
...
...
...
yes
| ?- fsa_regex_atom_compile('[a*,b^,{d,e}]',L).
L = fa(r(fsa_preds),3,[0],[1],[trans(0,a,0),trans(0,b,2),
    trans(0,d,1),trans(0,e,1),trans(2,d,1),trans(2,e,1)],[]) ?
yes
| ?- fsa_regex_transduces('{a:b,? -a}*',"ababac",L), atom_codes(Atom,L).
L = [98,98,98,98,98,99],
Atom = bbbbbc ?
yes
| ?-
```

All predicates that are imported have names starting with **fsa**. All module names start with **fsa** as well.

1.3 Examples

The package comes with a number of larger examples These examples include both automata and extended regular expression definitions.

- Examples/Automata

Regular expression operators which allows to input an automaton by listing its transitions, start states, and final states. Contributed by Dale Gerdemann.

- Examples/Booleans

A collection of regular expression operators including boolean operators and various predicates of automata. Contributed by Dale Gerdemann.

- Examples/Bouma

A finite-state automaton of all possible Dutch monosyllabic words. Contributed by Gosse Bouma.

- Examples/DutchWords

Dutch words, taken somewhere from ftp site (see ftp_info.txt). This list of words can then be used to experiment with the option to create perfect hashes (-dict2ph).

- Examples/GerdemannVannoord99

The replace operator as defined in our EACL 99 paper. Also some further examples with longest match and finite-state parsing. Contributed by Gerdemann and van Noord.

- Examples/Graph2Phon

Grapheme to Phoneme conversion for Dutch, as described in Bouma's ACL 2000 paper. Uses the Celex format for phonemes. Contributed by Gosse Bouma.

- Examples/Grimley-Evans

Implementation of the Hopcroft minimization algorithm by Edmund Grimley-Evans, as described in his WIA 97 paper. This code made me re-implement the FSA implementation of Hopcroft's algorithm. Contributed by Grimley-Evans.

- Examples/HMM

HMM's can be seen as a special type of weighted finite automata. This example implements the Baum-Welch training algorithm. Fairly simple-minded implementation.

- Examples/Hyphenation

Code to convert LaTeX hyphenation patterns into a single (large) finite-state transducer that can be used directly to hyphenate a given word. By Gosse Bouma and van Noord, after a suggestion by Lauri Karttunen.

- Examples/KaplanKay94

These examples are taken from Kaplan and Kay, Regular Models of Phonological Rule Systems, Computational Linguistics, 20(3), 1994. Simple examples of transducers, and composition.

- Examples/Karttunen91

These examples are taken from Karttunen, Finite-state Constraints, Proceedings International Conference on Current Issues in Computational Linguistics, Universiti Sains Malaysia, Penang, 1991. Simple examples of transducers, and composition.

- Examples/Karttunen95

Lauri Karttunen, The Replace Operator, ACL 1995, MIT Boston. Fairly complex examples of regular expression operator definitions.

- Examples/Karttunen96

Lauri Karttunen, Directed Replacement, ACL 1996. Includes soundex example from MLTT home page.

- Examples/Karttunen97

Lauri Karttunen, The Replace Operator, 1997. In volume edited by Roche and Schabes.

- Examples/Mohri97

Simple examples of weighted automata.

- Examples/MohriSproat96

Mehryar Mohri and Richard Sproat, An Efficient Compiler for Weighted Rewrite Rules. 34th Annual Meeting of the ACL, Santa Cruz 1996, pages 230-238. This only treats the non-weighted case. Nice example of the power of the regular expression language: their algorithm only takes a few paragraphs in FSA6.

- Examples/MSOL

Nathan Vailette, Logical Specifications of Transducers for NLP. In: FSMNLP 2001, ESSLLI Helsinki. electronically available from
`http://www.let.rug.nl/~vannoord/alp/esslli_fsmnlp` Contributed by Nathan Vailette.

- Examples/Nederhof

These are examples used by Mark-Jan Nederhof while investigating finite-state approximations of context-free grammars. The larger examples were used in my Computational Linguistics paper, The Treatment of Epsilon-moves in Subset Construction, available from `http://www.let.rug.nl/~vannoord/papers/` Contributed by Nederhof.

- Examples/Nerbonne

examples from `http://www.let.rug.nl/~nerbonne/teach.html` material for a course on computational morphology. Simple examples of transducers.

- Examples/OptimalityTheory

Implementation of Lauri Karttunen, The Proper Treatment of Optimality in Computational Phonology. FSMNLP 1998, Ankara. Includes definition of lenient composition operator and syllibification algorithm. Also includes Gerdemann/van Noord (even more proper?) alternative implementation. Contributed by Gerdemann and van Noord.

- Examples/PredModules

Examples of predicate modules; for example using bitvectors to represent sets of symbols, or using types. The bitvector stuff is only available under SICStus.

- Examples/PereiraRiley96

Fernando C. N. Pereira and Michael D. Riley, Speech Recognition by Composition of Weighted Finite Automata, 1996 (on cmp-lg). Also appears as chapter 15 of the volume edited by Roche and Schabes. Simple examples of weighted composition. Definition of their version of the composition operator (filtering our spurious paths).

- Examples/Queens

Solving the N-queens problem with regular expressions, by Dale Gerdemann. Another solution by G. van Noord. Interesting examples of definitions of regular expression operators.

- Examples/Random

Random automata. Used for the experiments documented in my FSMNLP98 paper, on the treatment of epsilon-moves in subset construction.

- Examples/Recognizers

Small examples.

- Examples/RocheSchabes95

Emmanuel Roche and Yves Schabes, Deterministic Part-of-speech Tagging with Finite-state Transducers, Computational Linguistics, 21(2), 1995. Small examples of transducers. Also includes a definition of the local extension operator.

- Examples/RocheSchabes97

Roche and Schabes, Introduction. In: Roche and Schabes (eds), Finite State Language Processing. MIT Press 1997. Includes implementations of is_functional, unambiguous, is_subsequential, build_bimachine, bitransform. Also has simple utils to apply and visualize bimachines.

- Examples/Spell

Implements a simple spell-checker as the combination of a dictionary and strings within Levenshtein distance d of the words in the dictionary (for some fixed d). Interesting application of the priority union operator of Karttunen (1998).

- Examples/Transducers

small stuff, including my attempt to translate Dutch temporal expression into a numerical format (that one is quite large, in fact).

- Examples/twolevel

Definitions to implement twolevel rules in the style of Kimmo. Contributed by Rob Malouf, Gosse Bouma, Gertjan van Noord.

- Examples/Weights

Small stuff, weighted automata.

- Examples/Wordgraphs

Some small acyclic weighted automata.

1.5 References

- Alfred V. Aho and John E. Hopcroft and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms. Addison Wesley, 1974.

- Meera Blattner and Tom Head, Single-Valued a-Transducers. In: Journal of Computer and System Sciences, 15. Pp. 310--327. 1977.

- J.A. Brzozowski. Canonical Regular Expressions and Minimal State Graphs for Definite Events. In: Mathematical Theory of Automata, Polytechnic Press Brooklyn, 1962.

- Gosse Bouma, A Modern CL course using Dutch. In: EACL 99 Postconference Workshop ‘‘Computer and Internet supported Education in Language and Speech Technology’’, June 12 1999, Bergen Norway.

- Jan Daciuk, Incremental Construction of Finite-State Automata and Transducers and their use in the Natural Language Processing. Thesis, Politechnika Gdanska, 1998.

- Jeffrey Friedl. Mastering Regular Expressions. O’Reilly 1997.

- Dale Gerdemann and Gertjan van Noord. Transducers from Rewrite Rules with Backreferences. EACL 1999 Bergen.

- John E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In: Z. Kohavi (editor), The Theory of Machines and Computations. Academic Press 1971.

- John E. Hopcroft and Jeffrey D. Ullman. Introduction to Automata Theory, Languages and Computation. Addison Wesley 1979.

- C. Douglas Johnson. Formal Aspects of Phonological Descriptions. Mouton The Hague, 1972.

- Ronald M. Kaplan and Martin Kay. Regular Models of Phonological Rule Systems. Computational Linguistics 20 (3) 1994.

- Lauri Karttunen. Finite-state Constraints. In: Proceedings International Conference on Current Issues in Computational Linguistics. Unviersiti Sains Malaysia, Penang. 1991.

- Lauri Karttunen. The Replace Operator. 33rd ACL, Boston, 1995.

- Lauri Karttunen and Jean-Pierre Chanod and Gregory Grefenstette and Anne Schiller, Regular Expressions for Language Engineering. Natural Language Engineering 2 (4) 1996.

- Lauri Karttunen, Directed Replacement. ACL 1996 Santa Cruz.

- Lauri Karttunen, The Proper Treatment of Optimality Theory in Computational Phonology. In: FSMNLP 1998, Ankara.

- George Anton Kiraz and Edmund Grimley-Evans, Multi-Tape Automata for Speech and Language Systems: A Prolog Implementation. In D. Wood and S. Yu (eds.), Automata Implementation, Lecture Notes in Computer Science 1436, Springer, 1998.

- Ted Leslie, Efficient Approaches to Subset Construction. University of Waterloo 1995.

- Mehryar Mohri, Compact Representations by Finite-state Transducers. ACL New Mexico 1994.

- Mehryar Mohri, Finite-State Transducers in Language and Speech Processing, Computational Linguistics 23 (2) 1997.

- Mehryar Mohri and Fernando C.N. Pereira and Michael Riley. A rational design for a weighted finite-state transducer library. In: Automata Implementation. WIA ’97. Lecture Notes in Computer Science 1436. Spring Verlag 1998.

- Mehryar Mohri and Richard Sproat. An Efficient Compiler for Weighted Rewrite Rules. ACL 1996 Santa Cruz.

- Gertjan van Noord. FSA Utilities: A Toolbox to Manipulate Finite-state Automata. In: Raymond, Woord, Yu (editors), Automata Implementation. Lecture Notes in Computer Science 1260. Spring Verlag 1997.

- Gertjan van Noord. The Treatment of Epsilon Moves in Subset Construction. In: Computational Linguistics 26 (1) 2000.

- Gertjan van Noord and Dale Gerdemann. An Extendible Regular Expression Compiler for Finite-state Approaches in Natural Language Processing. WIA 1999.

- Gertjan van Norod and Dale Gerdemann. Finite State Transducers with Predicates and Identity. Grammars 4 (3) 2001.

- Emmanuel Roche and Yves Schabes. Deterministic Part-of-speech Tagging with Finite-state Transducers. Computational Linguistics 21 (2) 1995.

- Emmanuel Roche and Yves Schabes. Finite-State Language Processing. MIT Press 1997.

- Bruce Watson, Taxonomies and Toolkits of Regular Language Algorithms. Ph.D. thesis TU Eindhoven. 1996.

- Bruce Watson, Implementing and Using Finite Automata Toolkits. In: Proceedings of the ECAI'96 Workshop Extended Finite State Models of Language. 1996.

1.6 Papers using FSA

- Gosse Bouma, A Modern CL course using Dutch. In: EACL 99 Postconference Workshop ''Computer and Internet supported Education in Language and Speech Technology'', June 12 1999, Bergen Norway.

- Gosse Bouma, A Finite State and Data-Oriented Method for Grapheme to Phoneme Conversion. In: 1st Meeting of the North American Chapter of the Association for Computational Linguistics, 303-310 Seattle 2000.

- Gosse Bouma, A modern Computational Linguistic Course using Dutch. In: Frank van Eynde and Ineke Schuurman, editors, CLIN 1998, Papers from the ninth CLIN Meeting, Amsterdam, 1999. Rodopi Press.

- Gosse Bouma, Finite State Methods for Hyphenation. In: FSMNLP 2001, ESSLLI Helsinki.

- Dale Gerdemann and Gertjan van Noord. Transducers from Rewrite Rules with Backreferences. EACL 1999 Bergen.

- Dale Gerdemann and Gertjan van Noord. Approximation and Exactness in Finite State Optimality Theory. In: FINITE-STATE PHONOLOGY: SIGPHON 2000, Fifth Meeting of the ACL Special Interest Group in Computational Phonology, COLING 2000.

- George Anton Kiraz, Multi-Tiered Nonlinear Morphology Using Multi-Tape Finite Automata: A case study on Syriac and Arabic. In: Computational Linguistics, 26 (1) 2000.

- Edwin Kuipers, Ill-formed Input and Finite-state Devices. MA Thesis, Rijksuniversiteit Groningen, 1997.

- Gertjan van Noord, FSA Utilities: Manipulation of Finite-state Automata implemented in Prolog, in: Proceedings of the First International Workshop on Implementing Automata. London Ontario Canada, 1996.

- Gertjan van Noord. FSA Utilities: A Toolbox to Manipulate Finite-state Automata. In: Raymond, Woord, Yu (editors), Automata Implementation. Lecture Notes in Computer Science 1260. Spring Verlag 1997.

- Gertjan van Noord. The Treatment of Epsilon Moves in Subset Construction. In: Computational Linguistics 26 (1) 2000.

- Gertjan van Noord and Dale Gerdemann. An Extendible Regular Expression Compiler for Finite-state Approaches in Natural Language Processing. WIA 1999.

- Gertjan van Norod and Dale Gerdemann. Finite State Transducers with Predicates and Identity. Grammars X (X) 2001.

- Nathan Vailette, Logical Specifications of Transducers for NLP. In: FSMNLP 2001, ESSLLI Helsinki.

- Markus Walther, Finite-State Reduplication in One-Level Prosodic Morphology. In: 1st Meeting of the North American Chapter of the Association for Computational Linguistics, 296-302 Seattle 2000.

- Markus Walther, One-Level Prosodic Morphology. Marburger Arbeiten zur Linguistic 1, University of Marburg. 64 pp.

- Markus Walther, Temiar Reduplication in One-Level Prosodic Morphology. In: FINITE-STATE PHONOLOGY: SIGPHON 2000, Fifth Meeting of the ACL Special Interest Group in Computational Phonology, COLING 2000.

1.7 Links

- Up-to-date information on the program can be obtained via `http://www.let.rug.nl/~vannoord/Fsa/`. The latest version of the program should be available there too.

- For information on the daVinci program, we refer to its homepage `http://www.informatik.uni-bremen.de/~inform/forschung/daVinci/`.

- For information on dot/GraphViz, we refer to
  http://www.research.att.com:80/sw/tools/graphviz/.

- For information on the VCG program:
  http://www.cs.uni-sb.de/RW/users/sander/html/gsvcg1.html

- A commercial version of VCG (called aiSee) is available from
  http://www.aisee.com/gallery/

- There is lots of interesting material at MLTT Xerox Grenoble:
  http://www.xrce.xerox.com/competencies/content-analysis/fst/.
  Be sure to read the documentation, including a number of nice examples.

- AT&T's FSM toolset for weighted finite-state automata is available from
  http://www.research.att.com/sw/tools/fsm.

- The corresponding lextools package by Richard Sproat is now available too, the url is
  http://www.research.att.com/sw/tools/lextools/

- Grail. Grail is a package of software for symbolic computation with finite machines and regular expressions. It is freely available to students, educators, and anyone who simply wants to use the software for their own amusement or education. The Grail homepage is at http://www.csd.uwo.ca/staff/drraymon/.grail/grail.html
  (somewhat outdated). A version for Linux is available from
  http://http://www.cs.sun.ac.za/~lynette/merlin.html.

- For a web interface to FSA, refer to
  http://www.let.rug.nl/~vannoord/fsademo/.

- For a web interface to FSA3, c.f.:
  http://i12www.ira.uka.de/Visualisierung.endlicher.Automaten/.

- A tutorial for FSA by Gosse Bouma, in Dutch:
  http://www.let.rug.nl/~gosse/tt/fsa.html

- An even simpler tutorial for FSA (in Dutch as well) used for highschool kids (!) is available as
  http://www.let.rug.nl/~vannoord/fsademo/fsademo/klas.html

- Electronic versions of some of the papers mentioned above are available through the cmp-lg archive at http://xxx.lanl.gov:80/cmp-lg/.

- A list of related projects at University of Western Ontario by Darrell Raymond is http://www.csd.uwo.ca/staff/drraymon/.grail/links.html. You can also obtain a copy of Ted Leslie's thesis from that site, which includes the algorithm to generate random automata, and which discusses density of automata related to determinization.

- Finite state Utilities by Jan Daciuk at
  `http://www.pg.gda.pl/~jandac/fsa.html`. Useful tools for dictionary construction and spell checking. Also read his dissertation.

- SICStus Prolog home page: `http://www.sics.se/isl/sicstus.html`.

- Collection of links on Prolog and Regular Expressions:
  `http://www.let.rug.nl/~vannoord/prolog-rx/PrologAndRegex.html`

- Wiese's Little Automata Builder at
  `http://www-ti.informatik.uni-tuebingen.de/~wiese/Automaton/`

- Another Java applet for finite automata at
  `http://www.cs.duke.edu/~rodger/tools/jflap/index.html`

- Interesting papers on Gene Myers' Home Page
  `http://www.cs.arizona.edu/people/gene`

1.8 Copyright

1.9 Acknowledgements

Helpful suggestions, feedback, etc., from a variety of people, too many to list here. Gosse Bouma and Dale Gerdemann were exceptionally influential.

1.10 Author

Gertjan van Noord, `mailto:vannoord@let.rug.nl`

2. Regular Expressions

Regular expressions are the preferred way to specify regular languages and regular relations. The regular expression compiler compiles a regular expression into an equivalent finite-state automaton.

The syntax of regular expressions in FSA is somewhat non-standard. As usual, atomic symbols normally represent themselves. Concatenation is indicated using a Prolog list where each of the elements is a regular expression itself; union is indicated using curly ('set') brackets, and Kleene star uses the *-suffix. For instance:

```
[{a,b,c}*,b,{a,b,c}*]
```

is the set of strings over the alphabet {a,b,c} such that each string contains at least one 'b'. Since a list indicates concatenation, the empty list '[]' indicates the empty concatenation, i.e. the empty string (the language consisting of a single string which is the epsilon string). Furthermore, '{}' represents the empty language.

Such regular expressions define regular languages, but regular expressions can also be used to define weighted regular languages, regular relations and weighted regular relations. A regular relation (transducer) is specified for instance by:

```
[{a:a,b:b,c:c}*,b:a,{a:a,b:b,c:c}*]
```

Where x:y represents a mapping from symbol x to symbol y. Weights can be attached using a double semi-colon:

```
{a::0,b::1,c::2}*
```

This extends to weighted transducers. As an example consider:

```
[{a:a::0,b:b::1,c:c::2}*,b:a::0,{a:a::2,b:b::1,c:c::0}*]
```

btw. this is equivalent to:

```
[{a:a:0,b:b:1,c:c:2}*,b:a:0,{a:a:0,b:b:1,c:c:2}*]
```

Below we provide more detailed documentation on the regular expression syntax, the type coercion performed implicitly by the regular expression compiler, ways to debug regular expressions, the way in which regular expression operators can be added, and detailed documentation on each of the regular expression operators.

2.1. Regular expression syntax

Regular expressions are defined as Prolog terms, and therefore Prolog syntax applies. For detailed information on this, cf. the Prolog manual. The brackets () can always be used to express the desired grouping. The order of precedence of operators is as follows:

```
: /
..
+ * ^
& -
o x xx
! #
```

Operators with the same precedence are interpreted left-to-right. For example, the expression

```
a..z* - b* & c..d*
```

is interpreted as:

```
(((a..z)*) - (b*)) & ((c..d)*)
```

Syntax **restrictions**

These are all due to the use of Prolog syntax. The benefit of using Prolog syntax is that I don't need to implement a parser, and you have flexibility (by using your own operator definitions). However, a few limitations are inherited as well. Here are a few rules of thumb:

Capitals can be used in a regular expression in the Tk entry field, by putting them between quotes:

```
'A'..'Z'
```

At the Regex prompt (after fsa -r) you can use:

```
'A'..'Z'
```

At the command-interpreter you can use:

```
2 |: -r 'A'..'Z'
```

As part of Expr in the fsa -r Expr command, use (this depends on the shell you are using. This example works for bourne sh, csh, and bash):

```
fsa -r "'A'..'Z'"
```

Use space between operators. Use space before and after a question mark (?). Don't use the dot '.' or the vertical bar '|' as (part of) a new operator. Similarly, avoid using the comma ',', the ';', and '->' as (part of a) regular expression operator. It's neither a good idea to use ':-'. Operators can be escaped using (), but hardly ever have to (e.g. the following works, even if o is the binary composition operator!).

```
fsa -r 'o o o'
```

Brackets can be used for grouping as well:

```
fsa -r 'o o o o (o o o)'
```

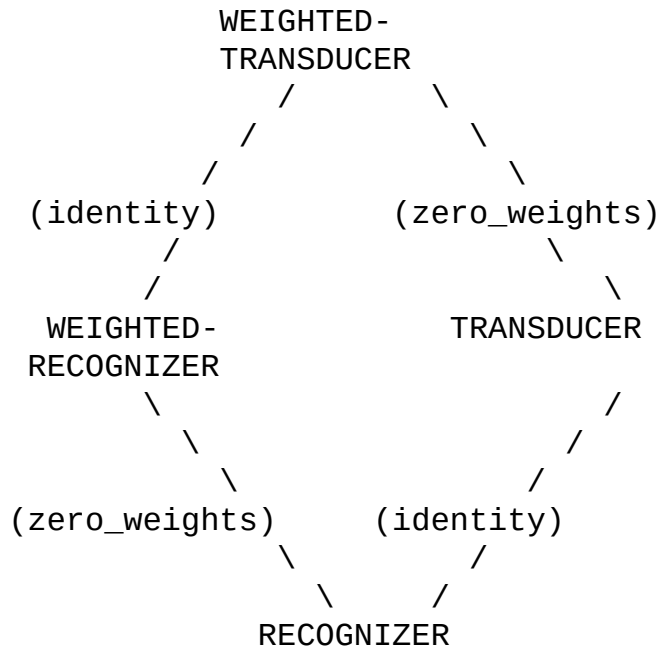
2.2. Type Coercion

FSA6 supports four types of automaton:

- recognizers
- weighted recognizers (aka string-to-weight transducers)
- transducers (aka string-to-string transducers)
- weighted transducers (aka string-to-string-weight transducers)

If automata of different types are combined by a binary regular expression operator such as the concatenation operator, then the types of the automata are compared, and if neccessary the automata are coerced silently. A similar mechanism is used for operations which require a specific type.

Coercion is performed according to the following type hierarchy, where coercion is possible in upward direction, using the operator indicated within brackets.



Consider the following examples:

```
expression:           coerced into:
```

```
{ a, b:c     }          { a:a,   b:c   }
{ a, b::4    }          { a::0,  b::4  }
{ a, b:c:4   }          { a:a:0, b:c:4 }
{ a::0,a:b:4 }          { a:a:0, a:b:4 }
{ a:b, a:b:4 }          { a:b:0, a:b:4 }
```

Some operators which expect a recognizer will temporarily **freeze** their argument automaton if it is of a different type. In that case, the symbol pairs or triples are treated as atomic symbols. This may or may not be what you want. The following operators work in this way:

complete minimize determinize complement

2.3. Spy Points on Regular expressions

The regular expression compiler provides detailed information on the computation time and the size of the resulting automata for certain regular expression operators, namely for those operators Op for which the predicate

```
bb_get(fsa_rx_spy:Op,on).
```

succeeds. So you can set a spy-point to operator concat by the directive:

```
?- bb_put(fsa_rx_spy:concat,on).
```

The special operator spy(Expr) is equivalent to Expr except that it has an associated spy-point.

2.4. Extending the regular expression notation

Using the -aux[file] command line option, or the AuxFile button of the TK Widget, you can load **auxiliary** regular expression operators. The file should be a Prolog source file. It will be loaded into module **fsa_regex_aux**. The syntax of regular expressions can be used in this file (in fact it **must** be used, beware if the file also contains ordinary Prolog code!).

Two relations are important: 1. macro/2 2. rx/2

The first relation is usually defined by unit clauses. It simply states that the regular expression in the first argument is an abbreviation for the regular expression in the second argument. For example:

```
macro(vowels,{a,e,i,u,o}).
```

Such macro’s can be parameterized using Prolog variables; e.g.:

```
macro(brz(Expr),determinize(reverse(determinize(reverse(Expr)))).
```

The relation rx/2 can be used for more complicated operations (operations that are cumbersome or impossible to define in terms of simpler regular expression operators). It defines a relation between the regular expression in the first argument and the finite automaton in the second argument. It is often useful to be able to call the regular expression compiler recursively. This should be done through the predicate fsa_regex:rx/2. The following is equivalent to the first example of macro/2 above:

```
rx( vowels, Fa) :-
    fsa_regex:rx({a,e,i,u,o}, Fa).
```

Consult the Examples directory, for instance in the MohriSproat96, Karttunen95, Karttunen96, Karttunen98, GerdemannVannoord, Queens directories, for some extensive illustrations.

2.5. Combining several auxiliary regular expression operator files

Suppose you want to use the definition of a replace operator in some file **replace.pl** in your analysis of Dutch phonology. In the latter file you can include the definitions from replace.pl by including somewhere at the top of your file the following directives:

```
:- ensure_loaded(replace).    %% loads replace.pl

:- multifile macro/2.
:- multifile rx/2.
```

This only works, if the multifile declarations are also present in the file you are importing. I.e. in this example the file replace.pl should also have the directives

```
:- multifile macro/2.
:- multifile rx/2.
```

3. Regular Expression Operators

This section lists the regular expression operators built-in in FSA.

3.1. ?

The set of one-symbol strings over the universal alphabet, ie. ? can be read as ‘any symbol whatsoever’. It uses the true/1 predicate_declaration from the current predicate module (cf the chapter Predicates on Symbols). If that declaration is not defined then no compilation for this operator is possible, and an error occurs.

3.2. Expr# m[inimize](Expr) mh(Expr) mb(Expr)

Applies minimization to the result of compiling Expr. There are a number of related expressions depending on which minimization algorithm is to be used.

mb uses the algorithm due to Brzozowski, **mh** uses the algorithm by Hopcroft (as described in Aho, Hopcroft and Ullman, 1974).

If Expr is a transducer then it is temporarily treated as a recognizer over pairs of symbols (using the fsa_frozen predicate module).

3.3. A! determinize(A) determinize(A,Algorithm)

Set of strings denoted by A, but moreover the subset construction determinization algorithm is applied to ensure that the automaton is deterministic. The algorithm can be specified as the second argument. There are several variants of the algorithm, which are different with respect to the treatment of epsilon transitions:

- per_graph: first construct efree automaton (jumps taken into account on target side of transitions and on start states)
- per_inverted_graph: first construct efree automaton (jumps taken into account on source side of transitions and and final states)
- per_reachable_graph: as per_inverted_graph, but maintains accessibility
- per_co_reachable_graph: as per_graph, but maintains co-accessibility
- per_subset: compute transitive closure of jumps on the fly for each subset
- per_state: compute transitive closure of jumps on the fly for each state

These variants and some interesting experimental observations are described in a paper I presented at the FSMNLP 98 workshop in Ankara. The paper is available from `http://www.let.rug.nl/~vannoord/papers/`. An improved version of the paper has been published in Computational Linguistics.

By default the algorithm is chosen by a simple heuristic based on the number of states and number of jumps of the input automaton. If A is a transducer then it is temporarily treated as a recognizer over atomic symbols, which happen to be pairs of predicates.

3.4. efree(E) reachable_efree(E) co_reachable_efree(E)

Constructs epsilon-free automaton for the automaton created for E. The first variant is faster, the second and third algorithms yield smaller automata by only taking into account states reachable from the start state, resp. from which a final state is reachable.

3.5. term_complement(E) ‘E

The term complement of E, i.e. the set of all single symbol strings minus those in E; this is equivalent to ? - E.

3.6. ~E complement(E)

The complement of the language denoted by E. E must be a recognizer.

3.7. A-B difference(A,B)

Set of strings denoted by A minus those given by B. A and B must be recognizers.

3.8. $E containment(E)

The language consisting of all strings that have an instance of E as a sub-string: [? *, E, ? *]. Note that the result is a minimal automaton. Since the definition of this operator depends on the ?/0 operator it is only defined if the current predicate module provides a definition of true/1. E can be both a recognizer or a transducer.

3.9. t_determinize(E)

The set of pairs denoted by E, but moreover the determinization algorithm for transducers by Mohri, cf. also Roche and Schabes, is applied to E. NB: this is only guaranteed to terminate if in fact E can be determinized in the appropriate sense. The implementation currently does not check for this. Refer to the Examples/RocheSchabes97 directory for an experimental implementation of that check and various related algorithms.

representation of sequential transducers

Note that in FSA subsequential transducers are represented as ordinary transducers. This implies in particular that instead of output symbols associated with final states, we have a separate transition over epsilon input and final output to a new final state. Similarly, automata which require an initial output to be associated with the start state will give rise to an extra transition from a new start state with epsilon input and the required start output.

The treatment of identity constraints over predicate pairs is especially tricky. This is mostly hidden in the predicate module declaration of t_determinize/2 preds. Funny things to watch for:

- certain non-functional automata become t_deterministic:

```
t_minimize([ a x {b,d}, d*])
```

This is actually quite useful.

- delayed identity constraints: predicates apply on the input side before the transition containing the target of the identity constraint is encountered. For example:

```
t_minimize({[a:b,?,?,?,?,?,b],[a:c,?,?,?,?,?,c]})
```

This is especially nasty if the number of question marks do not match up:

```
t_minimize({[a:b,?,?,?,?,?,b],[a:c,?,?,?,c]})
```

- the opposite occurs as well: sometimes we have to output symbols satisfying a certain predicate which must be identical to an input symbol which is yet to be encountered! This currently works. Try for instance:

```
t_minimize([a:b,a..f])
```

I think this is quite spectacular.

3.10. t_minimize(E)

The set of pairs denoted by E, but moreover the minimization algorithm for transducers by Morhi is applied to E. Note that this uses the t_determinize operator - for further details check there.

3.11. w_determinize(E)

The set of pairs denoted by E, but moreover the determinization algorithm for string to weight transducers by Mohri is applied to E. Cf. the t_determinize operator for further details.

3.12. wt_determinize(E)

Denotes the weighted transducer given by E, but moreover the determinization algorithm for weighted transducers by Mohri is applied to E. Cf. the t_determinize operator for further details.

3.13. w_minimize(E)

The set of pairs denoted by E, but moreover the minimization algorithm for string to weight transducers by Mohri is applied to E. Thus, E must denote a weighted recognizer. Cf. the t_determinize operator for further details.

3.14. wt_minimize(E)

Denotes the weighted transducer given by E, but moreover the minimization algorithm for string to weight transducers by Mohri is applied to E. Cf. the t_minimize operator for further details.

3.15. words(ListOfAtoms)

Creates minimal automaton for each of the atoms in ListOfAtoms, where each atom is expanded out into a concatenation of the atoms corresponding to its characters; i.e.

```
words([john,peter,mary])
```

is equivalent to

```
{ [j,o,h,n],[p,e,t,e,r],[m,a,r,y] }
```

3.16. perfect_hash(ListOfAtoms)

Creates weighted recognizer implementing a (minimal) perfect hash for the words found in ListOfAtoms. For instance:

```
perfect_hash([john,peter,mary])
```

is equivalent to

```
w_minimize( {[j,o,h,n] :: 0,
             [p,e,t,e,r] :: 2,
             [m,a,r,y] :: 1 } )
```

3.17. A:B pair(A,B) A:B:Weight

A and B are symbols; this is a transducer mapping an A to a B. In addition, A:B:W defines a weighted transducer mapping A to B with associated weight W.

3.18. A::W

Defines a weighted recognizer where A is a recognizer, and W its weight; alternatively defines a weighted transducer where A is a transducer and W its weight.

3.19. A x B cross_product(A,B) A x B x W

The set of pairs (A0,B0) such that A0 is in A and B0 is in B. Both A and B must describe recognizers.

In addition, A x B x W defines a weighted transducer where strings from A are mapped to B, with weight W.

3.20. A xx B sl_cross_product(A,B)

The set of pairs (A0,B0) such that A0 is in A and B0 is in B, moreover the strings A0 and B0 are to be of the same length. Both A and B must be recognizers.

3.21. escape(Sym)

Sym is a symbol. This denotes the language consisting of that symbol. Can be used to overwrite special meaning of some symbols. For instance, escape(?) can be used to denote a literal question mark. Sym should be ground Prolog term, and it is passed through the predicate **regex_notation_to_predicate** of the current predicate module.

3.22. S..T

S and T are one-character atoms or integers. In the first case, denotes the set of symbols from S up to T in ASCII coding. For instance a..e is equivalent to {a,b,c,d,e}. If S and T are integers, represents the set of integers in that interval: for instance 8..11 is equivalent to {8,9,10,11}.

3.23. incomplete(A)

Ensures that all states in the automaton for A are co-accessible, i.e. for each state there is path to a final state.

3.24. coaccessible(A)

Ensures that all states in the automaton for A are co-accessible, i.e. for each state there is path to a final state.

3.25. reachable(A)

This operator ensures that for each state s in the automaton of A there is a path from a start state to s.

3.26. accessible(A)

This operator ensures that for each state s in the automaton of A there is a path from a start state to s.

3.27. complete(A)

Adds transitions and a sink state such that the transition table is total, i.e. there is a transition for every symbol from every state. If A is a transducer then it is temporarily treated as a recognizer over pairs of symbols. If A is a transducer then it is temporarily treated as a recognizer.

3.28. ignore(A,B)

Strings from A interspersed with substrings from B. For instance, ignore([a,a,a],c) contains all strings over the alphabet {a,c} which contain exactly three a's. Both A and B must be recognizers.

3.29. {}

{} denotes the empty language.

3.30. {E1,E2,..,En} union(E1,E2) set([E1,E2,..,En])

Union of the languages denoted by E1,..,En. As a special case, '{}' is the empty language, i.e. a language without any strings. Note that the result is a minimal automaton. E1 .. En can be both transducers or recognizers. If one of them is a transducer, then all of the others are coerced into transducers as well.

3.31. []

[] denotes the empty string (or equivalently the language solely consisting of the empty string).

3.32. [E1,E2,...En] concat(E1,E2)

The concatenation of the languages denoted by E1, E2, .. En. As a special case, [] is the language solely containing of the empty string. Note that the result is a minimal automaton. E1 .. En can be both recognizers and transducers. If one of them is a transducer then all of the others are coerced into transducers as well.

3.33. E* kleene_star(E)

Kleene closure (zero or more concatenations) of the language denoted by E. Note that the result is a minimal automaton. E can be both a recognizer or a transducer.

3.34. E+ [kleene_]plus(E)

Kleene plus (one or more concatentations) of the language denoted by E. Note that the result is a minimal automaton. E can be both a recognizer or a transducer.

3.35. option(E) E^

Union of E with the empty string, i.e. a string from E occurs optionally. The result is a minimal automaton. E can be both a recognizer or a transducer.

3.36. intersect[ion](A,B) A & B

The intersection of the languages denoted by A and B. Produces a minimal automaton. A and B must be recognizers.

3.37. intersect_list([E0,E1,...,En])

This is equivalent to the sequential intersection of E0, E1, .., En, i.e., to the expression

```
E0 & E1 & ... & En
```

3.38. E0 o E1 compose(E0,E1)

The set of pairs (A,C) such that (A,B) is in E0 and (B,C) is in E1. Both E0 and E1 are (coerced into) transducers. Note that the result is a minimal automaton.

Note that in case both E0 and E1 are not same-length transducers, then often the resulting transducer will give rise to ‘spurious’ results in the sense that for a given input the same output is produced several times. See the paper by Pereira and Riley, 1996, for some suggestions to repair this. Obviously, in cases where you can determinize the transducer (with t_determinize) the spurious ambiguities will disappear as well.

3.39. compose_list([E0,E1,...,En])

This is equivalent to the sequential composition of E0, E1, .., En, i.e., to the expression

```
E0 o E1 o ... o En
```

3.40. reverse(E)

set of strings F such that the reversal of F is in E.

3.41. inversion(E) inverse(E) invert(E)

The set of pairs B:A such that A:B is in E. If E is a recognizer, then it is converted to its identity transducer.

3.42. id(E) identity(E)

The set of pairs A:A such that A is in E.

3.43. domain(E)

The set of strings A such A:B is in E.

3.44. weighted_domain(E)

The set of weighted strings A::B such A:C:B is in E.

3.45. range(E)

The set of strings B such that A:B is in E; if E is a weighted transducer then we obtain the set of strings B such that A:B:W in E.

3.46. weighted_range(E)

The set of weighted strings B:W such that A:B:W is in E.

3.47. weights(E)

A recognizer for the weights as defined in the weighed recognizer or weighted transducer E.

3.48. no_weights(E)

If E is a weighted recognizer, then this defines the recognizer obtained by ignoring all weights. If E is a weighted transducer, then this defines a transducer obtained by ignoring all weights.

3.49. cleanup(E)

The cleanup operator attempts to pack several transitions into one. For instance, assume there are two transitions from state p to q over the predicates p1 and p2 respectively. If p3 is a predicate which is true just in case either p1 or p2 is true, then we replace the two transitions by one transition over predicate p3. Note that if E is a sequence transducer, then cleanup does not attempt to create identities or combine one or more transitions involving identities (since that would require global analysis in order to know how identities on input and output side

line up).

3.50. expand_non_overlapping_predicates(E)

This operator constructs an automaton M for the expression E in such a way that all predicates which occur in M have an empty intersection, i.e. the resulting predicates in M are non-overlapping. Note that in the case of a sequence transducer, M is first coerced into a (synchronized) letter transducer.

3.51. subs(E)

E is supposed to be a transducer, or weighted recognizer. The result will be all pairs allowed by E and furthermore all pairs (x,y) such that (x',y) is in E and x' can be formed by substituting one symbol in x.

3.52. del(E)

E is supposed to be a transducer, or weighted recognizer. The result will be all pairs allowed by E and furthermore all pairs (x,y) such that (x',y) is in E and x' can be formed by deleting one symbol in x.

3.53. ins(E)

E is supposed to be a transducer, or weighted recognizer. The result will be all pairs allowed by E and furthermore all pairs (x,y) such that (x',y) is in E and x' can be formed by inserting one symbol in x.

3.54. word(Atom)

Denotes the string Atom, as a concatenation of its individual characters. For instance word(regular) is equivalent to [r,e,g,u,l,a,r].

3.55. convert_pred_module(NewModule,Expr) convert_pred_module(NewDomainMod,NewRangeMod,Expr)

Converts the automaton defined by Expr into an automaton using the pred_module declarations found in NewModule. Note that this is possible only in case the newer module is at least as expressive as the old one. For instance, you can convert an automaton with the **fsa_frozen** predicate module into an automaton with the **fsa_preds** predicate module, but not vice versa. The binary operator is for recognizers, the ternary operator for letter transducers.

3.56. fa(Fa)

Fa is already a finite automaton in appropriate format.

3.57. file(X)

This denotes the finite-automaton read from file X.

3.58. spy(Expr)

Equivalent to Expr, but sets spy-point on compilation of Expr. This implies that for debug level 1 or higher the CPU-time is reported required to compile Expr, as well as the size of the resulting automaton.

3.59. cache(Expr)

Equivalent to Expr, but caches result of compiling Expr, if the flag **regex_cache** is set to **selective**. If that flag has value **off** then there is no caching. If the value is **on** then the regular expression compiler caches every sub-computation.

3.60. random(NrStates,NrSymbols,Den,JDens[,FDens])

A random non-deterministic automaton is constructed consisting of the number of states specified in the first argument, number of symbols in the second argument. The desired density of the automaton is given in the third argument, whereas the fourth argument is the jump density. The final argument is a number between 0 and 1 indicating the likelihood that a state is final. If no fifth argument is specified, then all states are final.

For example, random(20,10,0.1,0.1) will be an automaton with 20 states, 10 symbols, approximately 400 transitions and 40 jumps. Automata generated this way always contain a single start state. They use the fsa_frozen predicate module; transition labels are integers counting from 0 upward.

If you need deterministic automata, consider the det_random/[4,5] regular expression operators.

3.61. det_random(NrStates,NrSymbols,Den,FDens)

A deterministic random automaton is constructed consisting of the number of states specified in the first argument, number of symbols in the second argument. The desired density of the automaton is given in the third argument, whereas the final argument is a number between 0 and 1 indicating the likelihood that a state is final. For instance, det_random(20,10,0.1,0.1) will be an automaton with 20 states, (at most) 10 symbols, approximately 20 transitions, and approximately 2 final states. Note that the generated automaton may contain states which are unreachable from the start state. Automata generated this way always contain a single start

state. They use the fsa_frozen predicate module; transition labels are integers counting from 0 upward.

If you need automata in which all states are reachable, consider the reachable/1 regular expression operator.

3.62. pragma(ListOfExpressions,E)

Equivalent to E, but this expression instructs the regular expression compiler that it should first compile each of the expressions in the list ListOfExpressions, which are supposed to occur repetitively in E. This is typically used in macros which use sub-expressions more than once. For example, consider the following macro (adapted from Kaplan and Kay):

```
macro(p_iff_s(L1,L2),
     if_p_then_s(L1,L2) & if_s_then_p(L1,L2)).
```

The pragma operator can be used in order to ensure that L1 and L2 are compiled only once:

```
macro(p_iff_s(L1,L2), pragma([V1-L1,V2-L2],
     if_p_then_s(V1,V2) & if_s_then_p(V1,V2))).
```

The first argument of pragma thus is a list of pairs V-Term where V is a variable which occurs in the second argument (in typical cases it occurs at least twice). The Term associated with V is compiled only once. For every occurrence of V in the second argument, the result of that compilation is used.

4. Predicates on Symbols

In standard regular expressions, the atomic symbols are normally treated ‘as is’: these symbols represent themselves. In FSA6 the possibility exists to have these atomic symbols stand for arbitrary (user-defined) predicates instead. In order to use this possibility, a collection of declarations must be provided in a module. Such declarations define, for instance, what the conjunction is of two predicates. In a regular expression such as p1 & p2, where p1 and p2 are predicates, the resulting automaton is equivalent to p3 where p3 is the conjunction of p1 and p2.

The global variable **pred_module** defines the name of a module which is the module that is used (by default) to obtain the definitions of these declarations. Recognizers are associated with the name of such a module as well. Transducers have two such predicate module names: one for the domain and one for the range.

Two standard predicate modules are:

```
fsa_preds
fsa_frozen
```

In fsa_preds each predicate is a set of symbols or the complement of a set of symbols. Such sets of symbols are represented by a term

```
in(OrderedList)
not_in(OrderedList)
```

Moreover, in case OrderedList is a list with precisely a single element then the in/1 functor is dropped. The negated sets are useful to provide a treatment of the ‘any symbol’ ?/0 operator. The predicate representing ‘any symbol’ is represented by not_in([]). For example, the expression ‘? - a’ will result in an automaton with a transition over not_in([a]).

The **fsa_frozen** module can be used for cases in which you want to treat symbols ‘as is’. If this module is used, you cannot use the ?/0 any symbol operator, or any of the operators which use this (such as the complement and term_complement operators). This predicate module is used internally for treating transducers temporarily as recognizers; e.g. if you want to determinize a transducer as if it were a recognizer by viewing each transition pair as an atomic unit. It is also used for the representation of weights in weighted automata.

If automata are combined using regular expression operators, then their corresponding modules must be identical. For instance, union of two recognizers implies that both recognizers have the same predicate module. Composition of two transducers imply that the predicate module of the output side of the first transducer is identical to the predicate module of the input side of the second transducer; the resulting transducer will take for its input side the input module of the first transducer, and as its output module it uses the output module of the second transducer.

This section lists the predicates that should be provided by a predicate module. An interesting example is provided by the fsa_preds module. A boring example is provided by the fsa_frozen module. In the Examples directory you will find a sub-directory PredModules which contains various other examples of predicate module declarations.

4.1. true(?Pred)

Pred is a predicate which is true for all symbols. This declaration is used to provide a translation for the ‘any symbol’ operator ?/0. Predicate modules which do not define true/1 cannot employ this operator, and as a consequence cannot use operators which are defined in terms of ?/0. The predicate should succeed at most once.

4.2. regex_atom_to_pred(+Atomic,-Pred)

This predicate translates the regular epxression notation into a predicate. This allows internal and external form of predicates; cf. display_predicate to translate from internal to an external form. Note: ?/0 is treated by the regular expression compiler itself, and uses true/1. The predicate should succeed exactly once.

4.3. evaluate_predicate(+Pred,?Symbol)

This predicate should succeed if Pred is true of Symbol, and fail otherwise.

4.4. conjunction(+P0,+P1,?P)

Predicate P is a predicate that is equivalent to the conjunction of P0 and P1. If the conjunction of P0 and P1 is inconsistent, then conjunction/3 should fail. The predicate should succeed at most once.

4.5. display_predicate(+Pred,-Term)

This predicate is used by the various visualization tools. It allows for the possibility to have an external format of a predicate. The predicate should succeed exactly once.

4.6. prepare_complement_of_set(+Fa,-Term)

Cf. complement_of_set/3. This predicate is used in the complete/1 operator. It computes any information from the finite automaton Fa that is useful later in complement_of_set/3. This computation is then only performed once for each complete/1 operator. Term is an arbitrary term that is passed on to complement_of_set/3. The predicate should succeed exactly once.

4.7. complement_of_set(+SetOfPreds,+Term,-Complements)

Complements is a list of predicates such that the disjunction of that set is equivalent to the complement of the disjunction of SetOfPredicates. Set is some datastructure computed in preparation phase. This definition is used in the complete/1 operator. The predicate should succeed exactly once.

4.8. determinize_preds(+KeyList0,-KeyList)

This code is required during the construction of deterministic automata, (the subset construction algorithm). Refer to the fsa_determinizer module for more details. In that module you can also find a definition of this predicate provided your predicate module has definitions for negation/2. In that case you can simply define:

```
determinize_preds(U0,U):-
        fsa_determinizer:determinize_preds(U0,U,YourPredModule).
```

This declaration is used in determinize/1 The predicate should succeed exactly once.

4.9. t_determinize_preds(+KeyList0,-KeyList)

This code is required during the construction of deterministic transducers, (the subset construction algorithm for transducers). Refer to the fsa_t_determinizer module for more details. In that module you can also find a definition of this predicate provided your predicate module has definitions for negation/2. In that case you can simply define:

```
t_determinize_preds(U0,U):-
        fsa_t_determinizer:t_determinize_preds(U0,U,YourPredModule).
```

This declaration is used in t_determinize/1 The predicate should succeed exactly once.

4.10. identity(+Pred0,-Pred)

If Pred0 is a predicate that is true of more than a single symbol, then Pred should be bound to `'$@'`(Pred0). If, on the other hand, Pred0 is true only of a single symbol, then Pred should be bound to Pred0. This is used in the computation of the regular operator identity/1. Predicate should succeed exactly once.

4.11. cleanup(+List0,-List)

Used in cleanup/1 operator. List0 is a list of predicates (interpreted as disjunction). List is an equivalent (but shorter) list of predicates (interpreted as disjunction). This predicate is used to translate sets of transitions into smaller sets of transitions. The predicate should succeed exactly once.

4.12. talks_about(+Pred,?Sym)

Used in foreign language interface. If Pred is a predicate which ‘mentions’ Sym then this should succeed. Otherwise it should fail.

5. Formats of Finite State Automata

FSA is capable of reading and writing finite automata in a number of different formats. The defaul format for reading is determined by the global variable **read**. The default format for writing is determined by the global variable **write**.

The following formats are available both for reading and writing:

- fast. Binary format of the **normal** format. Uses library(fastrw). Much faster reading and writing of automata. Drawback: binary files.
- normal. Internal representation (single Prolog term).

- old. Prolog program defining clauses start/1, final/2, trans/3, jump/2. A variant of this was used by FSA2 and FSA3, but it is still useful, for instance, if you want to input automata directly, rather than by means of regular expressions.
- fsm. Format of automata as used in AT&T's fsm library.
- compact. text format, fairly compact. Slow (especially for output).

For writing, the following additional formats are available:

- ps. PostScript.
- vcg. Input to the Xvcg graph visualization tool.
- davinci. Input to the DaVinci graph visualization tool.
- tk. Starts a interactive tcl/tk widget.
- dot. Input for the GraphViz visualization tools dot and dotty.
- pstricks. LaTeX code to be included in a document; requires pstricks macro's.
- latex. LaTeX document; requires pstricks macro's.
- prolog. Prolog program; interface to fsa_compiler module.
- c. C program; interface to fsa_compiler_to_c module.
- java. JAVA program; interface to fsa_java module.
- cpp. C++ program; interface to fsa_cpp module.
- fsm. Format of automata as used in AT&T's fsm library.
- grail. Format of automata as used in the Grail programme.

The **normal** format is the internal format used in FSA6. The module **fsa_data** provides a consistent interface to this format.

Here is a table indicating the relative speed of the standard input and output formats:

format	compact	fast	normal
writing	20	1	4
reading	5	1	4

Here is a table indicating the relative size of the standard input and output formats (measured in bytes):

```
compact    fast     normal
      1     1.8        1.6
```

5.1. The old format

In the **old** format a finite automaton is given as a Prolog program. The automaton is defined by clauses for the predicates:

- start(State)
- final(State)
- trans(State0,Sym,State)
- jump(State0,State)

Note that in this format states can be are arbitrary ground Prolog terms (these will be converted to integers). In the case of transducers, Sym is a pair Left/Right. The empty list [] is used to indicate the empty string. In the case of sequential transducers, Right must be a list of symbols.

5.2. The compact format

The **compact** format fairly closely follows the **normal** format. See the documentation on the **normal** format in the **fsa_data** module for more information. In this format a file is an ordinary text file. The format is intended to be used for machines only, and is not very helpful for human consumption.

- The first line of the file is the string "fsa6". This is used to differentiate the file from the pre-fsa6 compact formats (which can still be read-in).
- The second line is the letter **r** for recognizers or **t** for transducers.
- For recognizers, the third line is the name of the predicate module.
- For transducers, there are two such lines. The first line defines the domain predicate module, the second line the range predicate module.
- The next line is an integer indicating the number of states
- The next line is an ordered sequence of integers, separated by tabs, indicating the start states
- The next line is an ordered sequence of integers, separated by tabs, indicating the final states

- The next lines each represent a transition, until an empty line is encountered. The transitions are ordered. Each transition is a triple State Symbol State. Seperator is the tab again. States are integers. Symbols are readable as Prolog terms (recognizers) or pairs of In/Out, where In is a term and Out is either a single term or a list of terms. If the source state is identical to the source state of the previous line, it can be left out. If the symbol is identical as well, then it can be left out as well.

- The next lines are jumps. Jumps are ordered. Each line consists of two states separated by a tab. If the source state is identical to the source state of the previous line, it can be left out. If the symbol is identical as well, then it can be left out as well.

Example:

```
fsa write=compact -r '[class(a..f),{g,h}]'

fsa6
r
fsa_preds
3
0
1
0       in([a,b,c,d,e,f])       2
2       g       1
        h       1
```

5.3. The fast format

The **fast** format uses the same Prolog term representation as the **normal** format, except that library(fastrw) is used to read and write the Prolog term. This implies that reading and writing of automata in this format is very fast; the disadvantage is that **fast** is a binary format and therefore cannot be (easily) treated by other programs.

5.4. Internal Format of Finite Automata

This module provides a consistent interface to the internal format of finite automata. A finite automaton is a term

```
fa(Symbols,States,Starts,Finals,Transs,Jumps)
```

Symbols is a term r(Sig) (for recognizers) or t(SigD,SigR) for transducers. Weighted automata have r(Sig,fsa_frozen) and t(SigD,SigR,fsa_frozen). Here Sig, SigD, SigR are the predicate modules.

States is an integer indicating the number of states in the automaton.

Starts is an ordered list of integers indicating the start states of the automaton.

Finals is an ordered list of integers indicating the final states of the automaton.

Transs is an ordered list of triples trans(A,B,C) where A and C are integers indicating source and target state, and B is the symbol part. The symbol part is different for the different types of automata:

recognizers: P where P is a predicate

weighted recognizers: P/W where P is a predicate and W is a number

letter transducer: P/Q where P and Q is a predicate or the empty list

sequence transducer: P/Q where P is a predicate or the empty list, and Q is a list of predicates

weighted letter transducer: P/(Q/W) where P and Q is a predicate or the empty list, and W is a number

weighted sequence transducer: P/(Q/W) where P is a predicate or the empty list, and Q is a list of predicates

In addition, transducers allow a term `'$@'`(P) anywhere where a predicate is allowed (P a predicate).

Jumps is an ordered list of pairs jump(A,B) where A and B are integers indicating source and target state. This implies there is an epsilon transition from A to B.

6. Types of transducers

The determinization, minimization and minimum path algorithms for transducers are implemented in a fully general way, i.e., for various types of transducers (‘semirings’).

For each supported type, a number of predicates must be defined in a corresponding module (these are called ‘semiring declarations’):

- zero(Val).
- plus(Val0,Val1,Sum).
- minus(Val0,Val1,Diff).
- minimum(Val0,Val1,MinVal).
- minimum_only(YesNo).

Currently, the following types of transducers are supported:

- fsa_strings (ordinary (string-to-string) transducers)
- fsa_weights (weighted recognizers, aka string-to-weight transducers)
- fsa_weighted_strings (weighted transducers, aka string-to-weighted-string transducer).

The examples directory SemiringModules might contain additional semiring declarations.

6.1. zero(?Val).

The identity element for addition. For strings, this is the empty string; for weights it is 0.

6.2. plus(+Val0,+Val1,?Sum).

Addition. For weights this is number addition, for strings this is concatenation.

6.3. minus(+Val0,+Val1,?Diff).

Inverse of the addition operator.

6.4. minimum(+Val0,+Val1,?Min).

Minimum value of two given values. For weights this is the minumum of two numbers, for strings this is the longest common prefix.

6.5. minimum_only(+YesNo).

YesNo is one of the atoms

- yes
- no

indicating whether we are intested in all outputs associated with a path or only in the minimal output. For weights this is ‘yes’, for strings this is ‘no’.

7. Prolog Code Generation

FSA supports the production of Prolog code on the basis of a finite automaton. Various tricks are employed to make the resulting code efficient (rather than readable), but functionality has an ever higher priority. The functionality is the same as that provided by the fsa_interpreter module, only faster. For pure speed, you should consider using the fsa_compiler_to_c module.

Depending on the type of automaton, the compilation provides the following Prolog predicate:

- recognizer:

```
accepts(?String)
```

- weighted recognizer:

```
w_accepts(+StringIn,?Weight)
```

- transducer:

```
t_accepts(+StringIn,?StringOut)
```

- weighted transducer:

```
wt_accepts(+StringIn,?StringOut,?Weight)
```

accepts/1 can be used to check a given string for acceptance by the automaton, or it can be used to generate all strings accepted by the automaton. Since input can be non-deterministic we check for epsilon-cycles by keeping track of a list of states visited after last consumption of input). There are cases where it would make more sense to pre-compute efree automata first. We provide it anyway.

t_accepts/2 can be used to transduce a given string or to produce pairs of strings, if the length of the input list is known. Since input can be non-deterministic we check for epsilon-cycles by keeping track of a list of states visited after last consumption of input). The predicate uses special meta-notation |N+| for 'output' loops, to indicate that last N characters can be repeated any number of times. The compilation supports unknown symbols, including occurrences of delayed identity constraints (using a queue; trick due to Tamas Gaal was explained to me by Lauri Karttunen, Xerox Grenoble. Refer to Gerdemann and van Noord's paper in Grammars. Try: tminimize({[a:b,?,c],[a,?,d]}).

w_accepts/2 can be used to transduce a given string to the corresponding weight. In case of output loops only the minimum weight is produced. Since input can be non-deterministic we check for epsilon-cycles by keeping track of a list of states visited after last consumption of input.

wt_accepts/3 can be used to produce the transduction and weight for a given input string, or to generate triples of input, output and weight as long as the length of the input string is known.

8. C Code Generation

FSA supports the production of C code on the basis of a finite automaton. Before the automaton is translated into C code, it is determinized. In the case of transducers, Mohri's determinization algorithm is applied. Note however that certain transductions cannot be determinized: in that case the algorithm will not terminate.

The C program will contain definitions of the following functions:

- recognizers:

```
int accepts(char *in)
```

- weighted recognizers

```
int w_accepts(char *in,int *out)
```

- transducers

```
int t_accepts(char *in,char *out)
```

- weighted transducers

```
int wt_accepts(char *in,char *out, int *wout)
```

In each case, the functions return 1 if the string **in** is accepted. Otherwise they return 0. For transducers the resulting string or weight is available in **out** and **wout**.

If the global variable **c_with_main** is set to **on**, then the resulting C program will also contain a main function. This function is defined in such a way that it reads lines from standard input and applies the corresponding accept functions for each line. For recognizers, either **yes** or **no** is printed to standard error. For transducers, the transduction is written to standard output; if the input string is not in the domain of the transducer then **no** is written to standard error.

The representation of the finite-automaton in C is similar to the technique explained on page 43 (table 4.2) of Jan Daciuk's dissertation 'Incremental Construction of Finite-State Automata and Transducers and their use in the Natural Language Processing'. Politechnika Gdanska, 1998.

The special input symbol ^A is used in the representation of the automaton in C to indicate a symbol not otherwise mentioned in the automaton: it will match any such symbol. Similarly, in transducers the symbol ^B is used to indicate an unknown symbol with an associated identity. The corresponding output symbol is also ^B. When a string is transduced this ^B is replaced by the actual input symbol (by means of a queue). An unknown output symbol without an associated identity is represented using the symbol given by the global flag fl_arbitrary_symbol. In the case of final states with multiple outputs a special meta-notation is used using a special symbol given by the global variable fl_multiple_symbol_start which starts a sequence of possible outputs where each output is seperated using a symbol given by the global variable fl_multiple_symbol_sep.

9. C++ Code Generation

FSA supports the production of C++ code on the basis of a finite automaton. Before the automaton is translated into C++ code, it is determinized. In the case of transducers, Mohri's determinization algorithm is applied. Note however that certain transductions cannot be determinized: in that case the algorithm will not terminate.

Further documentation: refer to the fsa_fl module.

10. JAVA Code Generation

FSA supports the production of JAVA code on the basis of a finite automaton. Before the automaton is translated into C code, it is determinized. In the case of transducers, Mohri's determinization algorithm is applied. Note however that certain transductions cannot be determinized: in that case the algorithm will not terminate.

The JAVA program will define a class (named in accordance with the given output file name) which inherits from Applet. The class defines the method:

```
static void main(String argv[])
```

The instance itself is an applet in which you can write strings which are checked against the automaton. The JAVA program starts a graphical user interface in which you can input strings (if the option -w is the single option), or reads lines from standard input and writes the result of applying the automaton to standard output.

```
public void gui();
```

starts a graphical user interface in which you can input strings.

```
public DFA automaton();
```

returns the automaton part of the applet. This DFA class in turn defines the following methods:

```
public boolean Recognizer();
public boolean Transducer();
public boolean WeightedRecognizer();
public boolean WeightedTransducer();
```

As well as:

```
public void filter ()
```

reads lines from standard input and displays the result of running each line through the automaton to standard output.

```
public boolean accepts ( String in )
public String transduces ( String in )
public Integer weighs ( String in )
public StringWeightPair string_weight( String in )
```

The ‘main’ method is provided only if the global variable **java_with_main** is set to on.

In order to be able to run the JAVA code generated by FSA you need a java compiler, as well as the class files which are distributed with FSA. You have to ensure that the JAVA compiler knows where to find these files. For instance, if you have installed FSA in /usr/local/lib then the class files are in /usr/local/lib/fsa/Java. For instance, after running the FSA command:

```
% fsa write=java -r ’[a,b,c,? *]’ z.java
```

you compile the JAVA file with e.g.:

```
% javac  -classpath /usr/local/lib/fsa/Java:\
                    /usr/lib/java/lib/classes.zip:.  z.java
```

Instead of the -classpath option to javac it is preferable to include the relevant class directories in the CLASSPATH environment variable. You can now run the program using one of:

```
% java z -w
% java z
```

The representation of a finite-automaton in JAVA is similar to the technique explained on page 43 (table 4.2) of Jan Daciuk’s dissertation ‘Incremental Construction of Finite-State Automata and Transducers and their use in the Natural Language Processing’. Politechnika Gdanska, 1998, except that instead of the number of transitions we have a boolean flag indicating for each line whether that line is the last transition for the current state.

The special input symbol ^A is used in the representation of the automaton in C to indicate a symbol not otherwise mentioned in the automaton: it will match any such symbol. Similarly, in transducers the symbol ^B is used to indicate an unknown symbol with an associated identity. The corresponding output symbol is also ^B. When a string is transduced this ^B is replaced by the actual input symbol (by means of a queue). An unknown output symbol without an associated identity is represented using the symbol given by the global flag fl_arbitrary_symbol. In the case of final states with multiple outputs a special meta-notation is used using a special symbol given by the global variable fl_multiple_symbol_start which starts a sequence of possible outputs where each output is seperated using a symbol given by the global variable fl_multiple_symbol_sep.

11. Global Variables

This section lists the global variables and documents their effect. Global variables can be set from the command line and the command interpreter using Var=Val. You can also set variables using the Settings menu of the graphical user interface. Note that global variables

use the blackboard primitives; all keys are module expanded using the **fsa** module.

11.1. tkconsol

Boolean flag which determines whether **library(tkconsol)** is used for standard output. Note that support for tkconsol is very experimental. Current value is off. Default value is off. Typical values are [on,off]

11.2. tk_fsa_add_help_menu

Boolean flag which determines whether on-line help information is added to the menu. Since this takes quite a bit of band-width, you might want to turn it off for slow internet connections. Current value is on. Default value is on. Typical values are [on,off]

11.3. fsa_tcl_directory

Path to the directory in which the FSA tcl scripts are installed Current value is none. Default value is none. Typical values are []

11.4. pred_module

Default module for interpreting predicates on symbols. Current value is fsa_preds. Default value is fsa_preds. Typical values are [fsa_preds,fsa_frozen]

11.5. regex

Used internally Current value is []. Default value is []. Typical values are []

11.6. fa

Used internally Current value is []. Default value is []. Typical values are []

11.7. hash_size

Default size for hashes (refer to library **fsa_arrays** for detail). Current value is 65025. Default value is 65025. Typical values are
[500,1000,5000,10000,50000,100000,250000,500000,1000000]

11.8. interactive

This flag can be used to indicate that you want to run FSA interactively, even if you provide a command-line argument which would normally cause non-interactive usage. The value **cmdint** also implies interactivity but in addition the command interpreter is started. Current value is off. Default value is off. Typical values are [on,off,cmdint]

11.9. pstricks_style

Determines what kind of pstricks picture is contructed; at the moment **fancy** and **plain** are equivalent except that **fancy** implies that colors are used. Current value is plain. Default value is plain. Typical values are [fancy,plain]

11.10. v_algorithm

One of **dot** or **tree**. The first uses AT&T's **dot** program (from the GraphViz package) to compute geometry of states. The latter uses a built-in method which works reasonable for small graphs. Current value is tree. Default value is tree. Typical values are [tree,dot]

11.11. v_tree_depth

Is used by the tree algorithm for visualisation. Its effect has been forgotten by the author. Current value is off. Default value is off. Typical values are [on,off]

11.12. v_angle

Angle of edges in visualization of automata on Tk Canvas, as well as for postscript and latex output. A value of 0 implies straight lines between nodes. For larger values the lines that are drawn between nodes will move further away from the straight line. Current value is 0.15. Default value is 0.15. Typical values are [0.1,0.15,0.2,0.3,0.4,0.5,1.0]

11.13. v_xdist

Horizontal distance of states in visualization of automata on Tk Canvas, as well as for postscript and latex output. Current value is 120. Default value is 120. Typical values are [40,60,80,100,120,150,200]

11.14. v_ycoord

Used internally by the **tree** algorithm for visualization. I don't think it matters. Current value is 200. Default value is 200. Typical values are []

11.15. display_unused_states

This boolean variable determines whether states should be visualized which have no outgoing or incoming transitions, and which are neither a start state. Current value is on. Default value is on. Typical values are [on,off]

11.16. fl_arbitrary_symbol

This flag determines which atomic symbol should be used if transducers are compiled to C, C++ or Java, and the transducer contains don't care outputs. The value must be a Prolog atom. For example, if the transducer is given by [c,?,a:?]* and the value of this flag is '?', then the input

```
cxacyacza
```

will be mapped to

```
cx?cy?cz?
```

Current value is ?. Default value is ?. Typical values are [?,0,#]

11.17. symbol_separator

This flags determines which character is used to separate sequences of symbols that are accepted/transduced. For instance, if the value is 32 (for space) then you can type

```
a b a bb b aaa
```

to indicate the sequence of six symbols a, b, a, bb, b, and aaa. If the value is 44 (for comma) the same sequence is written/read as

```
a,b,a,bb,b,aaa
```

As a special case, a value of 0 indicates that a sequence is written without a separator; every single letter is assumed to be a symbol. For instance,

```
ababba
```

represents the sequence of symbols a, b, a, b, b and a.

Current value is 0. Default value is 0. Typical values are [0,32,43,44,45]

11.18. symbol_separator_out

As global variable **symbol_separator**, but only for output. If this variable is undefined, then the value of the global variable **symbol_separator** is used instead. Current value is undefined. Default value is undefined. Typical values are [undefined,0,32,43,44,45]

11.19. symbol_separator_in

As global variable **symbol_separator**, but only for input. If this variable is undefined, then the value of the global variable **symbol_separator** is used instead. Current value is undefined. Default value is undefined. Typical values are [undefined,0,32,43,44,45]

11.20. nr_sol_max

For the **produce** and the **transduce** options this global variable determines how many transductions for each input string should be given at most. Current value is 25. Default value is 25. Typical values are [1,5,10,25,50,100,1000,10000]

11.21. length_max

For the produce options this global variable determines the maximum length of strings that should be produced. In the case of transducer the variable determines maximum length of left string. A value of 0 indicates no restriction (in that case strings are not produced in order of length). Current value is 30. Default value is 30. Typical values are [0,5,10,25,50,100,1000,10000]

11.22. interpreter

This boolean flag indicates whether input automata for **fsa_interpreter** are compiled (by fsa_compiler_to_prolog) or interpreted. Current value is on. Default value is on. Typical values are [on,off]

11.23. debug

A value 0 indicates no continuation messages at all. A value of 1 will give cputime of operation. A level of 2 will give cputime of all intermediate operations too. Finally, level 3 is used for detailed continuation messages Current value is 0. Default value is 0. Typical values are [0,1,2,3,4]

11.24. regex_cache

This global variable determines whether regular expression compilations are cached or not. If the value **selective** is used, then only those operators are cached for which

```
bb_get(fsa_regex_cache:Fun)
```

succeeds. Current value is selective. Default value is selective. Typical values are [on,off,selective]

11.25. determinize_preds_cache

This global variable determines whether during the determinization algorithm the determinize_preds predicate should be cached. This is typically much faster, but requires much more memory in some cases. Current value is on. Default value is on. Typical values are [on,off]

11.26. cleanup_list_cache

This global variable determines whether during the cleanup algorithm the determinize_preds predicate should be cached. This is typically faster, but requires more memory in some cases. Current value is on. Default value is on. Typical values are [on,off]

11.27. set_random

This boolean global variable indicates whether the random generator should start with a new seed or not. If **off** the sequence of randomly generated automata will be the same for different FSA incarnations. Current value is off. Default value is off. Typical values are [on,off]

11.28. w_determinizer_minimum

This flag determines whether the **t_determinizer** applied to transducers using the **fsa_weights** semiring should only consider paths with lowest scores. Current value is on. Default value is on. Typical values are [on,off]

11.29. read

This global variable determines the format of input automata. The formats are explained in **module(fsa_io)**. Current value is normal. Default value is normal. Typical values are [normal,old,fast,compact,fsm,grail]

11.30. write

This global variable determines the format of output automata. The formats are explained in **module(fsa_io)**. Current value is normal. Default value is normal. Typical values are [normal,old,fast,compact,postscript,vcg,davinci,dot,pstricks,latex,prolog,c,count,fsm,java,cpp,grail]

11.31. count

This global variable determines if results are displayed in long or short format for the **count** output format, the **-count** option and the fsa_count predicate. Current value is long. Default value is long. Typical values are [short,long]

11.32. postscript_res

This variable determines which version of postscript output is used. **lowres** is better used for conversion to pngs, **normal** is better used for printing postscript Current value is normal. Default value is normal. Typical values are [normal,lowres]

11.33. no_display_beyond

Integer which determines a maximum number of states for automata that are displayed by any of the visualization tools. Automata with more states are not displayed; in such cases a small automaton is displayed indicating that the maximum was reached. Current value is 50. Default value is 50. Typical values are [30,40,50,100,1000]

11.34. c_with_main

This variable has effect for compilation of automata to C. If **on**, then the resulting C program will contain a main procedure. If **off** no such main procedure will be created. Current value is on. Default value is on. Typical values are [on,off]

11.35. java_with_main

This variable has effect for compilation of automata to JAVA. If **on**, then the resulting JAVA program will contain a main procedure. If **off** no such main procedure will be created. Current value is on. Default value is on. Typical values are [on,off]

11.36. cpp_with_main

This variable has effect for compilation of automata to C++. If **on**, then the resulting C++ program will contain a main procedure. If **off** no such main procedure will be created. Current value is on. Default value is on. Typical values are [on,off]

11.37. to_c_conversion

Boolean variable which has effect for compilation of automata to C, cf the **fsa_compiler_to_c** module for details. If on, the automaton is converted first; otherwise it's assumed the input is already converted. Current value is on. Default value is on. Typical values are [on,off]

11.38. to_java_conversion

Boolean variable which has effect for compilation of automata to JAVA, cf the **fsa_java** module for details. If on, the automaton is converted first; otherwise it's assumed the input is already converted. Current value is on. Default value is on. Typical values are [on,off]

11.39. to_cpp_conversion

Boolean variable which has effect for compilation of automata to C++, cf the **fsa_cpp** module for details. If on, the automaton is converted first; otherwise it's assumed the input is already converted. Current value is on. Default value is on. Typical values are [on,off]

11.40. fl_multiple_symbol_start

If transducers are compiled to C, C++ or Java, then if they contain final states with multiple outputs these multiple outputs are combined into a single output. This single output starts with a special symbol: the value of the f_multiple_symbol_start flag. The single output is constructed by concatenating each of the possible outputs, separated by the value of the fl_multiple_symbol_separator flag. For example, the transducer defined by the expression {a:b,a:c,a:d} will write out the symbol #d|c|b assuming that fl_multiple_symbol_start is # and fl_multiple_symbol_sep is |. Current value is #. Default value is #. Typical values are [#]

11.41. fl_multiple_symbol_separator

If transducers are compiled to C, C++ or Java, then if they contain final states with multiple outputs these multiple outputs are combined into a single output. This single output starts with a special symbol: the value of the f_multiple_symbol_start flag. The single output is constructed by concatenating each of the possible outputs, separated by the value of the fl_multiple_symbol_separator flag. For example, the transducer defined by the expression {a:b,a:c,a:d} will write out the symbol #d|c|b assuming that fl_multiple_symbol_start is # and fl_multiple_symbol_sep is |. Current value is |. Default value is |. Typical values are [|]

12. Command-line Arguments

Usage: fsa [Flag=Val]* [ActionOption]

The fsa program can be started with command-line arguments (options). A command-line consists of a number of global variable assignments, following by (at most one) **action** option, followed by more global variable assignments. If no action option is provided, then the system runs in interactive mode. If the variable **interpreter** is set to **on**, then fsa runs in interactive mode after the action indicated by the action option has been performed. If the flag **interactive** has been set to **cmdint**, then fsa runs the FSA command interpreter. Otherwise, you get the ordinary SICStus Prolog prompt.

Typical actions that can be performed through the use of an action option are:

- regular expression operations such as kleene closure, complementation for given automata
- determinization and minimization of automata
- construction of automaton on the basis of regular expression
- visualization of given automaton
- apply automaton to a string or set of strings

12.1. -aux Aux

Aux is a file containing auxiliary regular expression operator definitions. It is loaded into module fsa_regex_aux, and will be used for compiling regular expressions.

Note that your file with definitions of regular expression operators is consulted with the special Prolog-syntax operators for regular expression notation loaded. Thus you can use * .. & etc. in your definitions. Drawback is that you cannot use operator notation for e.g. the is/2 predicate.

A typical auxiliary definition will be:

```
macro(vowel,{a,e,i,o,u}).
```

A slightly more interesting one:

```
macro(free(Expr), ~ $ Expr).
```

You can also explicitly construct an automaton yourself, e.g.:

```
rx(my_operator(Expr),Fa) :-
    fsa_regex:rx(Expr,Fa0),
    my_operator_definition(Fa0,Fa).
```

so you can call fsa_regex:rx/2 for further compilations.

There can be multiple -aux options.

12.2. -pm File

File is supposed to contain the definition of a predicate module. The file is loaded and moreover the global variable **pred_module** is set to the name of the module defined in that file. There can be multiple -pm options

12.3. -l File

The File is loaded, using use_module/1. There can be multiple -l options.

12.4. -cmd Goal

evaluates Prolog Goal; Goal is parsed as Prolog term. Example:

fsa -cmd ’listing(library_directory),halt’).

There can be multiple -cmd options

12.5. -cmdint

Run interactively with the FSA6 command interpreter.

12.6. -a[ccepts] [In] String

This option can be used to test a given string for acceptance by an automaton read from In (or standard input). Fsa prints ‘yes’ or ‘no’ to standard error. Example:

```
% fsa -r 'a+' | fsa -a aaa
```

Prints ‘yes’.

12.7. -approx [In] String

This option can be used to get all best matches for a given string and an automaton read from In (or standard input). In must contain an automaton.

```
% fsa -r '[a,a]+' | fsa -approx aaa
```

Prints:

```
aa
aaaa
```

12.8. -fsa2fsm In Syms Aut | -fsa2fsm [In [Out]]

If three file names are given, then the automaton read from the first file is converted to an automaton in AT&T’s **fsm** software format. That automaton is written into Aut; Syms will contain a mapping from the integers used in Aut to the actual symbols. If less than three file names are given, then it is assumed that the actual symbols can be ignored and no Syms is written.

12.9. -fsm2fsa [In [Out]]

Converts an automaton in AT&T’s **fsm** software format into fsa5 format. Note that a separate symbol definition file is currently not supported.

12.10. -c[ompile] [In [Out]]

For a given automaton read from file In, a Prolog program is written to Out. For details, see the module fsa_compiler.

12.11. -c[ompile_to_]c [In [Out]]

For a given automaton read from In a C program is written to Out. For details, see module compiler_to_c.

12.12. -java [In] Out

For a given automaton read from In, a JAVA program is written to Out. For details, see module fsa_java.

12.13. -cpp [In] Out

For a given automaton read from In, a C++ program is written to Out. For details, see module fsa_cpp.

12.14. -c++ [In] Out

For a given automaton read from In, a C++ program is written to Out. For details, see module fsa_cpp.

12.15. -compose A B [Out]

The transducers read from A and B are composed, and the result is written to Out. Equivalent to

```
fsa -r 'compose(file(A),file(B))' >Out
```

12.16. -complement [In [Out]]

The complementation operator is applied to the automaton read from In, and the result is written to Out.

12.17. -count [In [Out]]

For the automaton read from In the number of transitions and symbols and some other properties is written to Out. If **-short** is specified, then the output is given as a single line consisting of the number of states, start states, final states, transitions, jumps, and symbols respectively. Otherwise a more elaborate message is printed meant for human consumption.

12.18. -density [In [Out]]

For the automaton read from In various densities are reported to Out. Deterministic density is the number of transition divided by the number of states times the number of symbols; absolute density is the number of transitions dividided by the number of states squared times the number of symbols. Jump density is the number of jumps dividied by the squared number

of states. Deterministic density can be used to characterize the difficulty of determinization. For deterministic densities of around 2, exponential blow-up of the output (and hence processing time) can be expected (Leslie 1995). Jump density can be used to estimate the most efficient subset construction algorithm (van Noord 1998).

12.19. -davinci [In [Out]]

For the automaton read from In a corresponding DaVinci term is written to Out. This can be used to visualize the automaton In using DaVinci:

```
fsa -davinci a.nd > a.davinci
daVinci a.davinci
```

12.20. -vcg [In [Out]]

For the automaton read from In a corresponding vcg term is written to Out. This can be used to visualize the automaton In using (x)vcg:

```
fsa -vcg a.nd | xvcg -
```

12.21. -dot [In [Out]]

For the automaton read from In a corresponding dot term is written to Out. This can be used to visualize the automaton In using dot or dotty:

```
fsa -dot a.nd | dotty -
fsa -dot a.nd | dot -Tps | gv -
fsa -dot a.nd | dot -Tgif | xv -
```

12.22. -d[eterminize] [In [Out]] | -dgraph [In [Out]] -drgraph [In [Out]] -dsubset [In [Out]] | -dstate [In [Out]]

The automaton read from In is determinized and written to Out. FSA6 supports four variants of the determinization algorithm. In the first form, a heuristic is used (based on the jump density) to select the variant of the determinization variant. The other forms indicate the particular variant that is to be used. For details, refer to the **fsa_determinizer** module.

12.23. -efree [In [Out]]

For the automaton read from In an equivalent automaton without any epsilon transitions (jumps) is written to Out.

12.24. -ignore A B [Out]

Equivalent to:

```
fsa -r 'ignore(file(A),file(B))' > Out
```

12.25. -diff[erence] A B [Out]

Equivalent to:

```
fsa -r 'difference(file(A),file(B))' > Out
```

12.26. -aa In | -accept_all In | -raa Regex

The program checks each string read from standard input for acceptance by the automaton read from In. Depending on the type of the automaton, the system reports the transductions for each string, or simply 'yes' or 'no' (for recognizers). In the third form the recognizer is specified by regular expression Regex rather than by an automaton.

12.27. -prolog Goal

Evaluates Prolog goal. Example:

```
fsa -prolog 'listing(user:file_search_path).'
```

12.28. -generate States Syms Dens [JDens]

This option is used to generate random finite automata, using the algorithm of Leslie 1995. States is the number of states, Syms is the number of symbols, Dens is absolute density, and JDens is the jump density.

12.29. -intersect A B [Out]

Equivalent to:

```
fsa -r 'intersect(file(A),file(B))' > Out
```

12.30. -kleene_star [In [Out]]

Equivalent to:

```
fsa -r 'file(In)*' > Out
```

12.31. -minimum_path [In]

Writes the path with the minimum weigth of the automaton read from In.

12.32. -kleene_plus [In [Out]]

Equivalent to:

```
fsa -r 'file(In)+' > Out
```

12.33. -reverse [In [Out]]

Equivalent to:

```
fsa -r 'reverse(file(In))' > Out
```

12.34. -inverse [In [Out]]

Equivalent to:

```
fsa -r 'inverse(file(In))' > Out
```

12.35. -domain [In [Out]]

Equivalent to:

```
fsa -r 'domain(file(In))' > Out
```

12.36. -range [In [Out]]

Equivalent to:

```
fsa -r 'range(file(In))' > Out
```

12.37. -cleanup [In [Out]]

Equivalent to:

```
fsa -r 'cleanup(file(In))' > Out
```

12.38. -identity [In [Out]]

Equivalent to:

```
fsa -r 'identity(file(In))' > Out
```

12.39. -option [In [Out]]

Equivalent to:

```
fsa -r 'option(file(In))' > Out
```

12.40. -union A B [Out]]

Equivalent to:

```
fsa -r 'union(file(A),file(B))' > Out
```

12.41. -concat A B [Out]]

Equivalent to:

```
fsa -r 'concat(file(A),file(B))' > Out
```

12.42. -m[inimize] [In [Out]] | -mb [In [Out]] | -mh [In [Out]]

Minimizes the automaton read from In. The first version uses the default minimization algorithm (by Brzozwski). The other options explicitly require the algorithms by, respectively, Brzozowski or Hopcroft. Refer to the fsa_regex module and the fsa_minimizer module for details.

12.43. -t_m[inimize] [In [Out]]

Applies the minimization algorithm for transducers (by Mohri) to the transducer read from In. Refer to the fsa_t_determinizer module for details.

12.44. -produce [In [Out]]

For the recognizer read from In strings accepted by In are written to Out.

12.45. -sample [In [Out]]

For the automaton read from In strings or string pairs accepted by In are written to Out, using a sampling procedure based on the weights in In. If In is not a weighted automaton, then uniform weights are assumed. The size of the sample is determined by the flag nr_sol_max. Examples:

```
fsa -r '[a::0.3*,b::5*]' |
      fsa nr_sol_max=10000 -sample |sort |uniq -c | sort -nr
```

produces:

```
5331 []
2519 a
1028 aa
 456 aaa
 187 aaaa
 172 b
 108 aaaaa
  73 ab
  32 aaaaaa
  28 aab
  24 aaaaaaa
  17 aaab
   8 aaaaaaaa
   5 bb
   3 abb
   3 aaaab
   3 aaaaab
   2 aaaaaaaaa
   1 aabb
```

```
fsa -r '[a*,b*]' |
       fsa nr_sol_max=100 -sample |sort |uniq -c | sort -nr
```

produces:

```
32 []
16 b
11 a
 9 bb
 5 abb
 5 ab
 3 bbbb
 3 bbb
 3 aa
 2 bbbbb
 2 abbbb
 2 aaa
 1 bbbbbbb
 1 abbbbbb
 1 abbb
 1 aab
 1 aaabb
```

```
1 aaaabbb
1 aaaaaab
```

12.46. -r[egex] [Regex] [Out]

The automaton described by regular expression Regex are written to Out. If Regex is not specified, it is read from standard input.

12.47. -tk [File] | -tk [-r Regex]

Starts the graphical user interface on the automaton in File, or the automaton defined by the regular expression Regex.

12.48. -postscript [In [Out]]

Produces postscript version of the automaton read from In.

12.49. dict2ph [In [Out]]

A minimal string-to-weight transducer will be written to Out, transducing each of the lines read from In into its rank in alphabetic ordering; in other words, the transducer computes a perfect hash for the keys read from In. For more info, see module **fsa_dict**.

12.50. dict2m [In [Out]]

A minimal recognizer will be written to Out, recognizing each of the lines read from In.

12.51. -pstricks_tex [In [Out]] | -pstricks_picture [In [Out]]

Produces LaTeX code using PsTricks macro's for the automaton read from In. In the first variant a self-contained LaTeX document is produced; in the second variant a LaTeX picture is produced to be included in another document.

12.52. -copy [In [Out]]

Copies the automaton from In to Out. Useful to convert between different formats using the **read** and **write** global variables:

```
fsa read=fast write=normal -copy a.nd b.nd
```

12.53. -prefix_tree [In [Out]]

A so-called prefix tree will be written to Out, which is a weighted recognizer of all the lines read from In, where the shape of the recognizer is the corresponding trie, and the weights are derived from the counts that each transition is used in In. Cf. Carrasco and Oncina, 1999.

12.54. -dict [In [Out]]

A so-called trie will be written to Out, which is a deterministic recognizer of all the lines read from In, where no states have more than a single incoming transition.

12.55. -t_d[eterminize] [In [Out]]

The determinization algorithm for transducers (by Mohri) is applied to string-to-string transducer In and yields Out. For details, refer to module t_determinizer.

12.56. -w_d[eterminize] [In [Out]]

The determinization algorithm for transducers (by Mohri) is applied to string-to-weight transducer In and yields Out. For details, refer to module t_determinizer.

12.57. -w_m[inimize] [In [Out]]

The minimization algorithm for transducers (by Mohri) is applied to string-to-weight transducer In and yields Out. For details, refer to module t_determinizer.

12.58. -identical In1 In2

Reports to standard output whether the two automata read from In and In2 are identical. The algorithm used to determine this first determines both automata, and then renames the states of the automata in a canonical way. The automata are identical in case their canonical representations are identical. Note that two non-identical automata might still be equivalent in the sense that they define the same language or the same relation.

13. The Command Interpreter

The FSA command interpreter provides line-based interaction with the FSA functionality. The command interpreter provides a history mechanism, escape to the operating system, escape to Prolog and on-line help information. All startup options for fsa are also available as commands in the command interpreter.

13.1. Syntax

A command is typed in by the user as one line of text; it's tokenized as a sequence of 'words', where spaces and tabs are treated as separators. Each word is treated as an atomic (Prolog atom or Prolog integer), unless it is written within { and }. In the latter case the 'word' is parsed as a Prolog term (in the latter case spaces and tabs are not interpreted as separators).

```
|: flag jan jan(a,b,c)
```

is equivalent to the Prolog goal

```
?- fsa_globals:fsa_global_set(jan,'jan(a,b,c)')
```

wherease

```
|: flag jan {jan(a,b,c)}
```

is equivalent to the Prolog goal

```
?- fsa_globals:fsa_global_set(jan,jan(a,b,c))
```

Variables occuring in such terms have scope over the full command-line!

13.2. Alias and History

The command-interpreter has an alias mechanism which subsumes a history mechanism as well. All occurences of $word are replaced by the definition of the alias word. The alias command itself can be used to define aliases:

```
19 |: alias hallo ! cat hallo
20 |: $hallo
```

so command number 20 will have the same effect as typing

```
33 |: ! cat hallo
```

and if this command had indeed been typed as command number 33 then typing

```
35 |: $33
```

gives also the same result. The special meaning of $ can be turned off by prefixing it with another $, e.g.:

```
|: cd $$HOME
```

Moreover, if no alias has been defined, then it will apply the last command that started with the name of the alias:

```
66 |: parse john kisses mary
67 |: $parse
```

will have the same meaning (in this order) if the macro parse is not defined.

13.3. Prolog goals

It is also possible to issue Prolog commands; however some restrictions apply.

```
39 |: p {member(X,[X|T])}
```

Note that this may succeed, but ’yes’ or ’no’ and variable bindings will NOT be printed.

13.4. Starting and Stopping the command interpreter

If no action option is provided, then the system runs in interactive mode. If the variable **interpreter** is set to **on**, then fsa runs in interactive mode after the action indicated by the action option has been performed. If the flag **interactive** has been set to **cmdint**, then fsa runs the FSA command interpreter. The command interpreter can be started any time from the Prolog prompt using the command r/0:

```
| ?- r.
*** Welcome to the FSA Command Interpreter (type ? for help) ***
5 |:
```

You can stop the command interpreter using the **p** command:

```
5 |: p
*** execution interrupted ***

yes
| ?-
```

You can quit FSA entirely using the **quit** command. Note that you can also use the command interpreter together with the graphical user interface.

13.5. p[rolog]

Stops the command interpreter.

13.6. % Words

ignores Words (comment). Note that there needs to be a space after %.

13.7. fc Files

fcompiles(Files).

13.8. um Files

use_module(Files).

13.9. el Files

ensure_loaded(Files).

13.10. c Files

compile(Files).

13.11. rc Files

reconsult(Files).

13.12. ld Files

load(Files).

13.13. libum Files

for each File, use_module(library(File)).

13.14. librc Files

for each File, reconsult(library(File)).

13.15. libc Files

for each File, compile(library(File)).

13.16. libel Files

for each File, ensure_loaded(library(File)).

13.17. libld Files

for each File, load(library(File)).

13.18. version

displays version information.

13.19. quit

quits FSA.

13.20. b

break; enters Prolog prompt at next break level.

13.21. d

debug/0.

13.22. nd

nodebug/0.

13.23. p [Goal]

without Goal: quits command interpreter -- falls back to Prolog prompt with Goal: calls Goal. Normally you will need {} around the Goal. For example:

```
4 |: p { member(X,[a,b,c]), write(X), nl }
```

13.24. ! Command

Command is executed by the shell. Note that the space between ! and Command is required.

13.25. alias [Name [Val]]

No args: lists all aliases; one arg: displays alias Name; two args: defines an alias Name with meaning Val.

13.26. help [Module [Class [Key]]]

displays help on Module-Class-Key; use ? to get help on commands only.

13.27. ? [Cmd]

displays help on Cmd; without Cmd prints list of commands.

13.28. spy [Module] Pred

set spypoint on Module:Pred; Pred can either be Fun or Fun/Ar.

13.29. cd [Dir]

change working directory to Dir; without argument cd to home directory.

13.30. pwd

print working directory.

13.31. ls

listing of directory contents

13.32. <any FSA startup option>

Any valid option you can give to the fsa command is a valid command for the command interpreter. For instance:

```
|: -d a.nd a.d

|: -m a.d a.m

|: -aux File

|: -tk

|: -r [[a,b]+,c]+
```

14. The Graphical User Interface

This section discusses the graphical user interface for FSA. It’s mostly extremely obvious. So this is a kind of ‘If you click on the help button, a help text will be displayed’ explanation. Note that currently the graphical user interface is available only under SICStus Prolog. Take a look at `http://www.let.rug.nl/~vannoord/Fsa/Manual/dump.png` if you

would like to know what the graphical user interface looks like.

MENU

The menu consists of a number of menu buttons. Really. The actions associated with the menu buttons are:

- **File**

[Load]: Loads a file that is assumed to contain an automaton.

[LoadAux]: Loads a file that is assumed to contain auxiliary regular expression macro's and operators.

[ReconsultAux]: Reconsults a file that is assumed to contain auxiliary regular expression macro's and operators. This allows tracing of your code.

[SaveAs]: Saves the automaton and the associated geometry information in a file. Such a file can be read-in using the Load button, or as the startup file. Various formats are available under various [SaveAsX] buttons.

[Revert] Redraws the current automaton without the current layout; a new layout will be computed.

[Redraw] Redraws the current automaton with the current layout.

[Close] Halts the graphical user interface, but FSA continues.

[Quit] Halts.

- **Settings** A number of global variables can be set via this menu. Help is available on-line.
- **Operations** A number of unary operations on the current regular expression can be fired through this menu. The operations are a subset of those allowed in regular expressions.
- **Produce** Produces a number of example strings (pairs of strings) accepted by the current finite automaton.
- **Visualization** Interface to a number of external visualization tools. These only work if you have the tools installed (really!), and the appropriate commands are in your PATH (yes, no magic here either).
- **Help** Well. What do you **think** this menubutton would do?

Now let's consider some of the other widgets maintained by the graphical user interface:

- **Regex**

If a regular expression is typed in the field, then after hitting <CR> the corresponding automaton will be visualised on the canvas.

The [Expand User Macro's] and [Expand All Macro's] can be used to expand all the macro's of the current regular expression.

- **String**

If a sequence of symbols (separated by whatever the symbol_separator flag requires) is typed in this field, then (after hitting <CR>, or after pushing the 'Submit' button) the system **runs** the current automaton on the input you provided. The actual way in which the automaton is run depends on the value displayed in the radio-button available to the right of the 'Submit' button.

- **Canvas**

The large canvas contains a picture of the current regular expression (or automaton read-in from a file). Note that you can drag states to alter the layout interactively. If you point your mouse to a label of an edge, then the corresponding edge will become red temporarily (this is useful for large labels). Also note that for states P and Q all edges from P to Q are combined in a single edge. Start states are green, final states are red and have a sunken relief. If a state is both a start state and a final state, then it is green with a sunken relief.

- **ToolBar**

The tool bar at the bottom consists of the following sub parts:

[EdgeAngle]: Text field should contain a real. Does a redraw using the current (typically) new angle variable upon <CR>. Does not require re-computation of layout.

[Xdistance]: Text field should contain an integer. Re-computes and re-draws using the current (typically new) distance of states parameter.

[Quality]: Re-computes and re-draws using the current (typically new) parameters.

[DisplaySigma]: Displays internal representation of alphabet and symbols list of current automaton.

[DisplayFa]: Displays internal representation of current automaton.

[CountFa]: Provides numerical information of current automaton.

[ClearCache]: Clears the cache of the regular expression compiler.

[ZoomIn]: This does the opposite of ZoomOut.

[ZoomOut]: This does the opposite of ZoomIn.

Finally, the [Interp] button can be used to get the name of the current Tcl/Tc interpreter. This is of interest only for development work.

- **TkConsol** (experimental)

As an experimental feature, you can include a widget displaying standard input and standard output. If you want to try out this new feature, you have to set the global variable **tkconsol** to **on**. E.g.:

```
fsa tkconsol=on -tk
```

Note that this is currently not very robust.

The following global variables are relevant for this module:

- tkconsol
- v_angle
- v_xdist
- no_display_beyond

14.1. tk_fsa_file(+File)

Starts a Tcl/Tk widget for the automaton read from File

14.2. tk_fsa(+Fa)

Starts a Tcl/Tk widget for the automaton Fa

14.3. tk_regex(+Atom)

Atom is an atom, converted to regular expression and compiled into automaton. A Tcl/Tk widget is started for that automaton.

14.4. tk_rx(+Expr)

Atom is a regular expression and compiled into automaton. A Tcl/Tk widget is started for that automaton.

15. Exported Predicates

In addition to the graphical user interface and the FSA command interpreter, it is also possible to use the toolkit as a library to your program. You can incorporate the FSA program in your own program, just as you can use other Prolog libraries. In order for this to work, you simply need to load the file fsa_library.pl in the installation directory. For example:

```
% sicstus
SICStus ...
Licensed to ...
| ?- use_module(fsa_library).
...
...
...
yes
| ?- fsa_regex_atom_compile('[a*,b^,{d,e}]',L).
L = fa(r(fsa_preds),3,[0],[1],[trans(0,a,0),trans(0,b,2),
    trans(0,d,1),trans(0,e,1),trans(2,d,1),trans(2,e,1)],[]) ?
yes
| ?- fsa_regex_transduces('{a:b,? -a}*',"ababac",L), atom_codes(Atom,L).
L = [98,98,98,98,98,99],
Atom = bbbbbc ?
yes
| ?-
```

Most predicates that are imported have names starting with **fsa**. All module names start with **fsa** as well. Below, we list the predicates exported by the FSA library. The modules which efficiently implement datastructures such as arrays, hashes, maps and sets are documented separately; these modules are supposed to be useful independently from their use in the FSA toolkit.

15.1. fsa_load_aux_file(+File)

File is assumed to contain auxiliary regular expression operators. It is loaded in module fsa_regex_aux and will be used for compiling regular expressions.

Note that your file with definitions of regular expression operators is compiled with the special Prolog-syntax operators for regular expression notation loaded. Thus you can use * .. & etc. in your definitions. Drawback is that you cannot use operator notation for e.g. the is/2 predicate.

A typical auxiliary definition will be:

```
macro(vowel,{a,e,i,o,u}).
```

A slightly more interesting one:

```
macro(free(Expr), ~ $ Expr).
```

You can also explicitly construct an automaton yourself, e.g.:

```
rx(my_operator(Expr),Fa) :-
    fsa_regex:rx(Expr,Fa0),
    my_operator_definition(Fa0,Fa).
```

so you can call fsa_regex:rx/2 for further compilations.

15.2. fsa_reconsult_aux_file(+File)

File is assumed to contain auxiliary regular expression operators. It is reconsulted in module fsa_regex_aux and will be used for compiling regular expressions. Normally you want to use fsa_load_aux_file instead. Use this predicate if you need to debug your Prolog definitions in File.

15.3. fsa_regex_atom_compile_file(+RegexAtom,+File)

RegexAtom is parsed as a regular expression. This expression is compiled to a finite automaton which is written to File.

15.4. fsa_regex_atom_compile(+RegexAtom,+Fa)

RegexAtom is parsed as a regular expression. This expression is compiled to a finite automaton Fa.

15.5. fsa_regex_read_compile_file(File)

A regular expression is read-in from standard input. The expression is compiled and the resulting automaton is saved in file File.

15.6. fsa_regex_read_compile(-Fa)

A regular expression is read-in from standard input. The expression is compiled into an automaton Fa.

15.7. fsa_regex_compile_file(+Expr,+File)

The regular expression Expr (ground Prolog term) is compiled into an automaton. The automaton is saved into File.

15.8. fsa_regex_compile(+Term,-Fa) rx(+Term,-Fa)

Term is a regular expression. It is compiled into the automaton Fa. The first form is typically used for a new regular expression compilation, whereas the second form is used for embedded compilations (called from user definitions). The only difference is that during debugging the depth of recursion is set to zero for the first form.

15.9. copy_fa(+File0,+File1).

The automaton in File0 is copied to File1. Useful to convert between different formats, by setting the **read** and **write** global variables.

15.10. fsa_read_file([+Format,]+File,?Fa)

Fa is read from File. If Format is unspecified the value of the global variable **read** is taken as the input format.

15.11. fsa_write_file([+Format,]+File,+Fa)

Fa is written to File. If Format is unspecified the value of the global variable **write** is taken as the input format.

15.12. fsa_states_number(?Fa,?Integer)

The number of states in Fa is Integer.

15.13. fsa_states_set(+Fa,?States)

States is an ordered list of integers: all states in Fa.

15.14. fsa_state(+Fa,?State)

State is a state in Fa.

15.15. fsa_start_states(?Fa,?StartStates)

StartStates is the ordered list of start states of Fa.

15.16. fsa_start_state(+Fa,?StartState)

StartState is a start states of Fa.

15.17. fsa_final_states(?Fa,?FinalStates)

FinalStates is the ordered list of final states of Fa.

15.18. fsa_final_state(+Fa,?FinalState)

FinalState is a final states of Fa.

15.19. fsa_transitions(?Fa,?Trans)

Trans is the ordered list of transitions of Fa.

15.20. fsa_transition(+Fa,?P,?Sym,?Q)

In Fa there is a transition from P to Q with symbol(pair) Sym.

15.21. fsa_jumps(?Fa,?Jumps)

Jumps is the ordered list of jumps of Fa.

15.22. fsa_jump(+Fa,?P,?Q)

In Fa there is a jump from P to Q.

15.23. fsa_construct([[+Symbols,]+NumberStates,]+Starts,+Finals,+Trans,+Jumps,-Fa)

Predicate to construct a finite automaton on the basis of lists of start states, final states, transitions and jumps. These lists need not be ordered. It's somewhat more efficient to specify the number of states, if known. It's even more efficient if you also know the symbols data-structure you want for Fa.

15.24. fsa_components(?Symbols,?Length,?Starts,?Finals,?Trans,?Jumps,?Fa)

Predicate to construct an automaton on the basis of its components, or to query the components of a given automaton. The difference with fsa_construct/7 is that Starts, Finals, Trans and Jumps must be sorted already.

15.25.
fsa_construct_rename_states([+Symbols,]+Starts,+Finals,+Trans,+Jumps,-Fa)

Predicate to construct a finite automaton on the basis of lists of start states, final states, transitions and jumps. These lists need not be ordered. Moreover, state names can be arbitrary Prolog terms. These state names will be renamed to integers. Symbol list is computed on the basis of Trans. Sigma is determined by the current default predicate module (i.e. by flag *pred_module*).. It's more efficient if you also know the symbols data-structure you want for Fa. Some checking on these symbols is performed nevertheless.

15.26. fsa_copy_except(+Key,?Fa0,?Fa1,?Part0,?Part1)

This predicate unifies Fa0 and Fa1 except for the part specified by Key. Part must be one of the atoms symbols, states, start_states, final_states, transitions, jumps. Part0 and Part1 are the corresponding parts in Fa0 and Fa1.

15.27. fsa_type(+Fa,?Type)

Type is the type of the automaton Fa, where type is one of recognizer, w_recognizer, transducer(Sub), w_transducer(Sub). Sub is one of letter or sequence.

15.28. fsa_compile_to_prolog(+Fa)
fsa_compile_to_prolog(+FileIn,+FileOut)

In the first variant, a Prolog program is written to standard output for the automaton Fa. In the second variant, the Prolog program is written to FileOut on the basis of the automaton read in from FileIn.

15.29. fsa_compile_to_c(+Fa)
fsa_compile_to_c(+FileIn,+FileOut)

In the first variant, a C program is written to standard output for the automaton Fa. The C program will read lines from standard input and report for each line whether it is a string accepted by Fa. In the second variant, the C program is written to FileOut on the basis of the automaton read in from FileIn.

15.30. fsa_compile_to_c_fa(+Fa,+FileOut)

A C program is written to FileOut for the automaton Fa. The C program will read lines from standard input and report for each line whether it is a string accepted by Fa.

15.31. fsa_cpp(+FileIn,+FileOut)

A C++ program is written to FileOut for the recognizer read from FileIn. The C++ program will read lines from standard input and apply the automaton to each line.

15.32. fsa_java(+FileIn,+FileOut)

A JAVA program is written to FileOut for the recognizer read from FileIn. The JAVA program will read lines from standard input and apply the automaton to each line.

15.33. fsa_global_set(+Key,?Val)

Predicate to set the global variable with name **Key** to **Val**.

15.34. fsa_global_get(+Key,?Val)

Predicate to query the value of the global variable with name **Key**. If the value is undefined then **Val** is unified to a default value. These default values are available as the third argument of the **fsa_global_decl** predicate.

15.35. fsa_global_decl(?Key,?Help,?Default,?Typical,Val^Goal)

Key is a global variable with default value **Default**. Some typical values are given in the list **Typical**. **Help** is a string explaining the variable. Val^Goal can be used to check that Val is an appropriate value for this flag.

15.36. fsa_global_list[-List]

List will be unified with a keylist of all the global variables with their associated values. If no argument is given, then this list is written to standard output

15.37. fsa_version

FSA version information is displayed on standard error. Note that the version information is available through the **fsa_version** global variable.

15.38. fsa_host_prolog(?Atom)

Atom is an atom indicating the current Prolog system. At the moment Atom is one of **sicstus**, **yap**, or **swi**.

15.39. fsa_dict_to_perfect_hash(+ListOfStrings,-Fa)

A string-to-weight transducer Fa will be constructed implementing the perfect hash for the ListOfStrings; i.e. the transducer maps each string to its rank (in alphabetic order), and does not accept any string not listed in ListOfStrings. The transducer is (w_)minimal.

15.40. fsa_dict_to_perfect_hash_file(+FileIn,+FileOut)

FileIn is assumed to contain a set of strings: each line is a string. A string-to-weight transducer will be written to FileOut implementing the perfect hash for the set of strings read from FileIn: i.e. the transducer maps each string to its rank (in alphabetic order), and does not accept any string not listed in FileIn. The transducer is (w_)minimal.

15.41. fsa_dict_to_fsa(+ListOfStrings,-Fa)

A minimal recognizer Fa will be constructed recognizing exactly the strings in ListOfStrings

15.42. fsa_dict_to_fsa_file(+FileIn,+FileOut)

FileIn is assumed to contain a set of strings: each line is a string. A minimal automaton recognizing exactly those strings is written to FileOut

15.43. fsa_dict_to_trie_file(+FileIn,+FileOut)

FileIn is assumed to contain a set of strings: each line is a string. A deterministic automaton recognizing exactly those strings is written to FileOut; the automaton has the form of a trie.

15.44. fsa_dict_to_trie(+ListOfStrings,-Fa)

Fa is a deterministic automaton recognizing exactly each of the strings in ListOfStrings; the automaton has the form of a trie.

15.45. fsa_regex_accepts(+Atom,+String)

Succeeds if String is accepted by the regular expression in Atom. For example:

```
fsa_regex_accepts('[{a,b}*,b,a,b,{a,b}*]',"abbbbabababa").
```

15.46. fsa_regex_transduces(+Atom,+String0,?String)

String is a transduction of String0 according to the regular expression in Atom. Example:

```
fsa_regex_transduces('a:b',"a",L).

L = [98] ?
```

15.47. fsa_regex_transduces_w(+Atom,+String0,?Weight)

String is a transduction of String0 according to the regular expression in Atom. Example:

```
fsa_regex_transduces_w('[a:3,b:1*]',"abbb",L).

L = 6 ?
```

15.48. fsa_accepts(+String,+Fa)

This predicate can be used both to recognize a given string or to produce a string according to Fa. This is why we use dif/2 below. We prefer functionality over efficiency here; note that the compiler-to-prolog implements the same functionality.

Making this code faster could be done for instance by indexing on source state and symbol. For deterministic automata, use the compiler-to-c for a fast and compact recognizer.

Since input can be non-deterministic we check for epsilon-cycles (the fifth argument of accepts/6 is a list of states visited after last consumption of input). Again, there are cases where it would make more sense to pre-compute efree automata first. But if that's the case you could do it, right?

15.49. fsa_transduces(+StringIn,?StringOut,+Fa)

StringOut is a transduction of StringIn according to transducer Fa.

This predicate employs a meta-notation in cases where loop-checking encounters a cycle. In that case the notation

```
|N+|
```

is written into the output string indicating that the previous N symbols could be repeated here as many times as desired. For example, consider the simple regular expression mapping an a to one or more b's:

```
a:(b+)
```

If a transduction is request for input string a, then the following outputs occur:

```
b
bb|1+|
```

In the second output string the meta-notation indicates that the second b could be repeated multiple times.

Many of the same remarks wrt fsa_accepts/2 apply here: works for non-instantiated input (lists with variable elements work OK, but variable length lists typically don't). Also takes care of identity constraints in the transducer, including delayed identity constraints and too early identity constraints, by using a non-proper implementation of queues which allow dequeue-ing before enqueue-ing! Examples to try, using the fsa_preds predicate module:

```
t_minimize([a:b,class(a..f)])

t_minimize({[a:b,?,?,?,?,?,b],[a:c,?,?,?,?,?,c]})
```

I think this is neat.

15.50. fsa_transduces_w(+String,?Weight,+Fa)

Weight is the weight assigned to String by Fa.

Similar to fsa_accepts/2 and fsa_transduces/3 above. However, we assume that there are no identity constraints. Loop-checking for [] input, but no meta-notation in output: we simply produce the minimum in such cases. That seems to be appropriate in most applications (?).

15.51. fsa_regex_approx_accepts(+String,+Regex,-Recipe)

String is a Prolog string, and Regex is an atom that will be parsed as a regular expression. The system will match String approximately to that regular expression, returning each of the matches which require a minimal number of substitutions, insertions, deletions, and transpositions. A match is given by a recipe which is a list of Prolog terms as follows:

```
P:d        deletion at position P
P:i(Pred)  insertion of symbol for which Pred is true, at P
P:s(Pred)  substitution of symbol for which Pred is true, at P
P:t        transposition at position P
```

where P refers to the position in the sequence of symbols extracted from String where the corresponding edit operation takes place.

15.52. fsa_approx_accepts(+String,+Fa,-Recipe)

String is a Prolog string, and Fa is a finite automaton. The system will match String approximately to this Fa, returning each of the matches which require a minimal number of substitutions, insertions, deletions, and transpositions. A match is given by a recipe which is a list of Prolog terms as follows:

```
P:d        deletion at position P
P:i(Pred)  insertion of symbol for which Pred is true, at P
P:s(Pred)  substitution of symbol for which Pred is true, at P
P:t        transposition at position P
```

where P refers to the position in the sequence of symbols extracted from String where the corresponding edit operation takes place.

15.53. fsa_regex_approx_transduces(+String0,+Regex,-String)

String0 and String is a Prolog string, Regex is an atom that will be parsed as a regular expression denoting a string to string transducer. The system will match String0 approximately to the domain of the regular expression, and return each of the transductions of these approximate matches.

15.54. fsa_approx_transduces(+String0,+Fa,-String)

String0 and String is a Prolog string, Fa a string to string transducer. The system will match String0 approximately to the domain of Fa, and return each of the transductions of these approximate matches.

15.55. fsa_regex_approx_transduces_w(+String0,+Regex,-Weight)

String0 is a Prolog string, Regex is an atom that will be parsed as a regular expression denoting a weighted recognizer. The system will match String0 approximately to the domain of the regular expression, and return each of the weights of these approximate matches.

15.56. fsa_regex_approx_transduces_wt(+String0,+Regex,-String,-Weight)

String0 and String is a Prolog string, Regex is an atom that will be parsed as a regular expression denoting a weighted transducer. The system will match String0 approximately to the domain of the regular expression, and return each of the transductions of these approximate matches (a String and a Weight).

15.57. fsa_approx_transduces_w(+String0,+Fa,-Weight)

String0 is a Prolog string, Fa a weighted recognizer. The system will match String0 approximately to the domain of Fa, and return each of the weights of these approximate matches.

15.58. fsa_approx_transduces_wt(+String0,+Fa,-String,-Weight)

String0 and String is a Prolog string, Fa a weighted transducer. The system will match String0 approximately to the domain of Fa, and return each of the transductions of these approximate matches (a String and a Weight).

15.59. fsa_minimum_path_file(+InFile)

Reports on standard output the minimum weight path in the transducer read from InFile.

This is implemented by a generalization of Dijkstra's algorithm to find the minimum weight path in a given transducer. The algorithm for transducers are implemented in a fully general way, i.e., for various types of transducers (cf. the fsa_semiring module).

The implementation is more general than the predicates provided by e.g. the SICStus libraries for graphs. And **much** more efficient, even though the agenda is not maintained as a heap (the latter decision was caused by the fact that it would be hard to implement such a heap efficiently for the various types of transducers with their corresponding 'minimum' definitions).

15.60. fsa_minimum_path(+Fa[,-Path])

Reports on standard output (if Path is not present) or instantiates Path to the minimum weight path in the transducer Fa.

15.61. fsa_minimum_path_array(+Fa,-Array,+Flag)

Array will be instantiated to an UpdatableFsaArray (cf. module fsa_arrays) indicating for each state in the transducer Fa the minimum cost from that state to a final state.

15.62. fsa_davinci(+File0,+File) fsa_davinci(+Fa)

In the first variant, a representation accepted by the daVinci graph visualization program is written to File on the basis of the automaton read from File0. In the second form, the representation for Fa is written to standard output.

15.63. fsa_dot(+File0,+File) fsa_dot(+Fa)

In the first variant, a representation accepted by the dot / GraphViz graph visualization program is written to File on the basis of the automaton read from File0. In the second form, the representation for Fa is written to standard output.

15.64. fsa_vcg(+File0,+File) fsa_vcg(+Fa)

In the first variant, a representation accepted by the vcg graph visualization program is written to File on the basis of the automaton read from File0. In the second form, the representation for Fa is written to standard output.

15.65. fsa_pstricks_picture(+File0,+File)

A piece of LaTeX code with pstricks macro's which produces a picture of the automaton read from File0 is written to File. This LaTeX code is supposed to be included in a full LaTeX document. The global variable pstricks_style influences the result.

15.66. fsa_pstricks_tex(+File0,+File)

A standalone LaTeX document with pstricks macro's which produces a picture of the automaton read from File0 is written to File. The global variable pstricks_style influences the result.

15.67. fsa_postscript(+File0,+File)

Postscript code which produces visualization of automaton read from File0 is written to File. The Postscript macro's are due to Peter Kleiweg. The global variable postscript_res can be set to indicate whether output is meant to be displayed on the screen, or printed.

15.68. fsa_visualization(+Format,+Fa)

Starts an external visualization program visualizing Fa. Format indicates what program is to be used and must be one of:

- vcg
- dot_ghostview (dot -Tps | gv)
- pstricks_ghostview (latex ; dvips ; gv)
- dotty
- davinci

16. fsa_array: Non-updatable Arrays (127+32 trees)

This module provides a non-updatable array data-structure. Accessing individual items in the array is very efficient. The arrays are implemented using O'Keefe's N+K trees, with N=127 and K=32.

NB. Array indices start at 0: so 0 refers to the first element of the array.

Here's an overview of the predicates provided:

- fsa_array_new/[1,2] create a new non-updatable array
- fsa_array_access/[3,4] access a value in a non-updatable array
- fsa_array_get/3 get a value in a non-updateble array
- fsa_array_to_list/2 conversion of array -> list

The N+K tree data-structure is described in The Craft of Prolog, by Richard A. O'Keefe, MIT Press, 1990, chapters 4.5. and 4.6. It is similar to the tries described in section 2 of Tarjan and Yao, Storing a Sparse Table, CACM, 1979, 606-611; also refer to Knuth, The Art of Computing Vol 3.

16.1. List of Predicates

This section lists the predicates defined by this module.

16.1.1. fsa_array_new(-FsaArray[,?Size])

Initializes FsaArray as a new array. In this implementation of arrays the optional second argument is not used.

16.1.2. fsa_array_access(+Index,?Val[,?Default],+FsaArray)

Val is unified with the Index'th entry of FsaArray. This predicate thus subsumes setting and reading of a value in the array. Remember that you can't change values of an array (except by further instantiation). For the 4-ary form, if the Index'th entry was not yet defined, then Val is unified with Default (and not with the Index'th entry).

16.1.3. fsa_array_get(+Index,?Val,+FsaArray)

Val is unified with the Index'th entry of FsaArray. That entry must not be variable. This predicate is different from fsa_array_access/3 in that it can fail.

17. fsa_m_array: Mutable Arrays

This module provides a mutable array datastructure. The arrays are implemented using O'Keefe's N+K trees, with N=127 and K=32.

NB. Array indices start at 0: so 0 refers to the first element of the array.

Here's an overview of the predicates provided:

MutableFsaArray:

- fsa_m_array_new/[1,2] create a new mutable array
- fsa_m_array_get/3 lookup a value from a mutable array
- fsa_m_array_put/[3,5] update a value in a mutable array

The N+K tree data-structure is described in The Craft of Prolog, by Richard A. O'Keefe, MIT Press, 1990, chapters 4.5. and 4.6.

17.1. List of Predicates

This section lists the predicates defined by this module.

17.1.1. fsa_m_array_new(-MutableFsaArray,[+Size])

Initializes MutableFsaArray as a new mutable array.

17.1.2. fsa_m_array_get(+Index,?Val[,?Default],+MutableFsaArray)

Val is unified with the Index'th entry in MutableFsaArray. The predicate **succeeds** if that entry has not yet been set, without binding Val (first form); or it binds Val to Default (second form).

17.1.3. fsa_m_array_put(+Index,?Val,+MutableFsaArray) fsa_m_array_put(+Index,?ValOld,?ValDefault,?Val,+MutableFsaArray)

The Index'th entry in MutableFsaArray is updated to Val (using the SICStus built-in update_mutable/create_mutable). ValOld will be bound to the old value, or to ValDefault if no value existed.

18. fsa_u_array: Updatable Arrays (15+16 trees)

This module provides an updatable array datastructure. The arrays are implemented using O'Keefe's N+K trees, with N=15 and K=16.

NB. Array indices start at 0: so 0 refers to the first element of the array.

Here's an overview of the predicates provided:

- fsa_u_array_new/[1,2] create a new updatable array

- fsa_u_array_get/[3,4] lookup a value from an updatable array
- fsa_u_array_put/[4,5] update a value in an updatable array

The N+K tree data-structure is described in The Craft of Prolog, by Richard A. O'Keefe, MIT Press, 1990, chapters 4.5. and 4.6.

18.1. List of Predicates

This section lists the predicates defined by this module.

18.1.1. fsa_u_array_new(-UpdatableFsaArray[,?Size])

Initializes UpdatableFsaArray as a new mutable array. In this implementation the optional Size argument is not used.

18.1.2. fsa_u_array_get(+Index,?Val[,?Default,]+UpdatableFsaArray)

Val is unified with the Index'th entry in UpdatableFsaArray.

18.1.3. fsa_u_array_put(+Index[,?OldVal],?Val,+UpdatableFsaArray0,?UpdatableFsaArray)

The Index'th entry in UpdatableFsaArray0 is updated to Val; UpdatableFsaArray is the resulting new array. OldVal is unified with the old value of Index.

19. fsa_hash: Non-updatable Hashes (N+K trees)

This module provides a non-updatable hash datastructure, on top of the fsa_array module.

Here's an overview of the predicates provided:

- fsa_hash_new/[1,2] create a new non-updatable hash
- fsa_hash_access/[3,4] access a value in a non-updatable hash
- fsa_hash_get/3 get a value in a non-updatable hash

The hash function is taken from library(terms). The default size of the hashes is determined by the global variable **hash_size**.

19.1. List of Predicates

This section lists the predicates defined by this module.

19.1.1. fsa_hash_new(-FsaHash[,Size])

Initializes a new FsaHash with size Size; or default size if there is no second argument. The default size is given by the global variable **hash_size**.

19.1.2. fsa_hash_access(+Key,?Val[,?Default],+FsaHash)

Unifies Val with the value associated with Key in FsaHash. Note that keys must be ground Prolog terms. For the 4-ary form, if Key had no associated value, then Default is unified with Val (and Key is not added to the table).

19.1.3. fsa_hash_to_keylist(+HashedFsaArray,-Keylist)

Keylist is a list of all the Key-Value pairs in HashedFsaArray.

20. fsa_m_hash: Mutable Hashes

This module provides a mutable hash datastructure on top of the fsa_hash datastructure.

Here's an overview of the predicates provided:

- fsa_m_hash_new/[1,2] create a new mutable hash
- fsa_m_hash_get/[3,4] lookup a value from a mutable hash
- fsa_m_hash_put/[3,5] update a value in a mutable hash

20.1. List of Predicates

This section lists the predicates defined by this module.

20.1.1. fsa_m_hash_new(-MutableFsaHash[,Size])

Initializes a new MutableFsaHash with size Size; or default size if there is no second argument. The default is determined by the global variable **hash_size**.

20.1.2. fsa_m_hash_get(+Key,?Val,+MutableFsaHash) fsa_m_hash_get(+Key,?Val,?Default,+MutableFsaHash)

Val is unified with the value associated with Key in MutableFsaHash. If no such key exists in MutableFsaHash already, then the predicate **succeeds** without binding Val (in the first form) or unifies Val and Default (second form).

20.1.3. fsa_m_hash_put(+Key,?Val,+MutableFsaHash) fsa_m_hash_put(+Key,?OldVal,?Default,?Val,+MutableFsaHash)

The value associated with Key in MutableFsaHash is updated to Val (using the SICStus built-in update_mutable). OldVal is unified with the old value (it it existed) or with Default (if it didn't exist).

21. fsa_u_hash: Updatable Hashes

This module provides an updatable hash data-structure on top of the updatable array datastructure.

Here's an overview of the predicates provided:

- fsa_u_hash_new/[1,2] create a new updatable hash
- fsa_u_hash_get/3 lookup a value from an updatable hash
- fsa_u_hash_put/[4,6] update a value in an updatable hash

The hash function is taken from library(terms). The default size of the hashes is determined by the global variable **hash_size**.

21.1. List of Predicates

This section lists the predicates defined by this module.

21.1.1. fsa_u_hash_new(-UpdatableFsaHash[,Size])

Initializes a new UpdatableFsaHash with size Size; or default size if there is no second argument. The default size is determined by the global variable **hash_size**.

21.1.2. fsa_u_hash_get(+Key,?Val,+UpdatableFsaHash)

Val is unified with the value associated with Key in UpdatableFsaHash. If no such key exists in UpdatableFsaHash already, then the predicate fails.

21.1.3. fsa_u_hash_put(+Key,?Val,+UpdatableFsaHash0,?UpdatableFsaHash) fsa_u_hash_put(+Key,?OldVal,?Default,?Val,+UpdatableFsaHash0,?UpdatableFsaHash)

The value associated with Key in UpdatableFsaHash0 is updated to Val, resulting in the new hash UpdatableFsaHash. OldVal is unified with the old value (it it existed) or with Default (if it didn't exist).

22. set_bbbtree: Balanced Binary Trees: Sets

This module implements sets using bounded balanced binary trees. It is adapted from the Mercury version. The original is available from `http://www.cs.mu.oz.au/research/mercury/`. That implementation is based on 'Functional Pearls: Efficient sets -a balancing act' by Stephen Adams, J. Functional Programming 3 (4): 553-561, Oct 1993.

22.1. List of Predicates

This section lists the predicates defined by this module.

22.1.1. set_bbbtree__init(?Bbbtree)

Bbbtree is initialized as an empty set.

22.1.2. set_bbbtree__empty(?Bbbtree)

Succeeds if **Bbbtree** is the empty set.

22.1.3. set_bbbtree__non_empty(?Bbbtree)

Succeeds if **Bbbtree** is a non-empty set.

22.1.4. set_bbbtree__size(+Bbbtree,?Integer)

Integer is the cardinality of the set **Bbbtree**.

22.1.5. set_bbbtree__is_member(+El,+Bbbtree,?Bool)

Bool is the atom **yes** if **El** is an element of **Bbbtree**. Otherwise it is the atom **no**.

22.1.6. set_bbbtree__member(?El,+Bbbtree)

El is an element of **Bbbtree**. Can be used to enumerate all elements of **Bbbtree**.

22.1.7. set_bbbtree__least(+Bbbtree,?El)

El is the least element occurring in **Bbbtree**, using the standard ordering of terms.

22.1.8. set_bbbtree__largest(+Bbbtree,?El)

El is the largest element occurring in **Bbbtree**, using the standard ordering of terms.

22.1.9. set_bbbtree__singleton_set(?BbbTree,?El)

Bbbtree is a set with single element **El**.

22.1.10. set_bbbtree__equal(+BbbtreeA,+BbbtreeB)

BbbtreeA and **BbbtreeB** are the same sets.

22.1.11. set_bbbtree__insert(+BbbtreeA,+El,-BbbtreeB[,?New])

BbbtreeB is the result of inserting **El** in **BbbtreeA**. The optional fourth argument is the atom **yes** if **El** is not an element of *BbbtreeA*; otherwise it is the atom **no**.

22.1.12. set_bbbtree__insert_list(+BbbtreeA,+List,-BbbtreeB)

BbbtreeB is the result of inserting each of the elements in **List** to **BbbtreeA**.

22.1.13. set_bbbtree__delete(+BbbtreeA,+El,-BbbtreeB

BbbtreeB is the result of removing the element **El** from **BbbtreeA**. The predicate **succeeds** if **El** is not an element of **BbbtreeA** (cf. set_bbbtree__remove).

22.1.14. set_bbbtree__delete_list(+List,+BbbtreeA,-BbbtreeB)

BbbtreeB is the result of deleting each of the elements of **List** from **BbbtreeA**. The elements are **not** required to be contained in **BbbtreeA** (cf. set_bbbtree__remove_list).

22.1.15. set_bbbtree__remove(+BbbtreeA,+El,-BbbtreeB

BbbtreeB is the result of removing the element **El** from **BbbtreeA**. The predicate **fails** if **El** is not an element of **BbbtreeA** (cf. set_bbbtree__delete)

22.1.16. set_bbbtree__remove_list(+List,+BbbtreeA,-BbbtreeB)

*BbbtreeB is the result of deleting each of the elements of **List** from **BbbtreeA**. The elements **are** required to be contained in **BbbtreeA** (cf. set_bbbtree__delete_list).

22.1.17. set_bbbtree__remove_least(+BbbtreeA,?Least,-BbbtreeB)

BbbtreeB is the result of removing the least element **Least** from **BbbtreeA**, in the standard ordering of terms.

22.1.18. set_bbbtree__remove_largest(+BbbtreeA,?Largest,-BbbtreeB)

BbbtreeB is the result of removing the largest element **Largest** from **BbbtreeA**, in the standard ordering of terms.

22.1.19. set_bbbtree__list_to_set(+List,-Bbbtree)

Bbbtree is a set containing precisely all elements of **List**.

22.1.20. set_bbbtree__sorted_list_to_set(+SortedList,?Bbbtree)

Bbbtree is the set containing precisely the elements of **SortedList**.

22.1.21. set_bbbtree__sorted_list_to_set_len(+SortedList,?Bbbtree,+Len)

Bbbtree is the set containing precisely the elements of **SortedList**. **Len** is the length of **SortedList**.

22.1.22. set_bbbtree__to_sorted_list(+Bbbtree,?SortedList)

SortedList is a sorted list of the elements of the set **Bbbtree**.

22.1.23. set_bbbtree__union(+BbbtreeA,+BbbtreeB,-BbbtreeC)

BbbtreeC is the union of **BbbtreeA** and **BbbtreeB**.

22.1.24. set_bbbtree__power_union(+Bbbtrees,-BbbtreeC)

Bbbtrees is a set of sets. **BbbtreeC** is the union of all of of these sets.

22.1.25. set_bbbtree__intersect(+BbbtreeA,+BbbtreeB,-BbbtreeC)

BbbtreeC is the intersection of **BbbtreeA** and **BbbtreeB**.

22.1.26. set_bbbtree__power_intersect(+Bbbtrees,-BbbtreeC)

Bbbtrees is a set of sets. **BbbtreeC** is the set containing the elements which occur in each of **Bbbtrees**

22.1.27. set_bbbtree__difference(+BbbtreeA,+BbbtreeB,-BbbtreeC)

BbbtreeC is the set **BbbtreeA** minus all elements of **BbbtreeB**.

22.1.28. set_bbbtree__subset(+BbbtreeA,+BbbtreeB)

BbbtreeA is a subset of **BbbtreeB**.

22.1.29. set_bbbtree__superset(+BbbtreeA,+BbbtreeB)

BbbtreeA is a superset of **BbbtreeB**.

23. map_bbbtree: Balanced Binary Trees: Maps

This module implements maps using bounded balanced binary trees. It is adapted from set_bbbtree, which itself is adapted from the Mercury version. The original of that version is available from `http://www.cs.mu.oz.au/research/mercury/`. That implementation is based on 'Functional Pearls: Efficient sets -a balancing act' by Stephen Adams, J. Functional Programming 3 (4): 553-561, Oct 1993.

A map is a set of key/value pairs, such that each key is associated with at most one value. Keys are required to be ground. The typical operations on maps such as lookup the value of a given key are O(log n) where n is the number of pairs in the map. A potentially more efficient implementation of maps is provided by the **fsa_hash**, **fsa_m_hash** and **fsa_u_hash** modules.

23.1. List of Predicates

This section lists the predicates defined by this module.

23.1.1. map_bbbtree__init(?Bbbtree)

Initializes **Bbbtree** as an empty map.

23.1.2. map_bbbtree__empty(?Bbbtree)

Bbbtree is an empty map.

23.1.3. map_bbbtree__size(+Bbbtree,?Size)

Size is the number of pairs in map **Bbbtree**.

23.1.4. map_bbbtree__get(+Key,?Val,+Bbbtree)

Val is the value associated with **Key** in the map **Bbbtree**. This predicate fails if **Key** is not a key of **Bbbtree**.

23.1.5. map_bbbtree__least(+Bbbtree,?Least,?Val)

Key is the least key in **Bbbtree** (using the standard order ordering of terms). Its value is **Val**.

23.1.6. map_bbbtree__largest(+Bbbtree,?Largest,?Val)

Key is the largest key in **Bbbtree** (using the standard order ordering of terms). Its value is **Val**.

23.1.7. map_bbbtree__put(+Key,?Val,+Bbbtree0,-Bbbtree)

Bbbtree is the same map as **Bbbtree0**, except that **Key** is now associated with **Val**.

23.1.8. map_bbbtree__put_list(+Bbbtree0,+KeyValList,-Bbbtree

Bbbtree is the same map as **Bbbtree0**, except that each of the key-value pairs in **KeyValList** are in **Bbbtree**.

23.1.9. map_bbbtree__delete(+Bbbtree0,+Key,-Bbbtree)

Bbbtree is the result of removing **Key** and its associated value from **Bbbtree0**. Succeeds if **Key** was not a key of **Bbbtree0** (cf map_bbbtree__remove).

23.1.10. map_bbbtree__delete_list(+Keys,+Bbbtree0,-Bbbtree)

Bbbtree is the result of deleting all keys **Keys** with associated values from **Bbbtree0**. These keys are not required to exist in **Bbbtree0** (cf map_bbbtree__remove_list).

23.1.11. map_bbbtree__remove(+Bbbtree0,+Key,-Bbbtree)

Bbbtree is the result of removing **Key** and its associated value from **Bbbtree0**. Fails if **Key** was not a key of **Bbbtree0** (cf map_bbbtree__delete).

23.1.12. map_bbbtree__remove_list(+Keys,+Bbbtree0,-Bbbtree)

Bbbtree is the result of removing all keys **Keys** with associated values from **Bbbtree0**. These keys are required to exist in **Bbbtree0** (cf map_bbbtree__delete_list).

23.1.13. map_bbbtree__remove_least(+Bbbtree0,?Key,?Val,-Bbbtree)

Key is the least key in **Bbbtree0** (using standard ordering of terms). Its value is **Value**. **Bbbtree** is the same map as **Bbbtree0** except that **Key** is removed.

23.1.14. map_bbbtree__remove_largest(+Bbbtree0,?Key,?Val,-Bbbtree)

Key is the largest key in **Bbbtree0** (using standard ordering of terms). Its value is **Value**. **Bbbtree** is the same map as **Bbbtree0** except that **Key** is removed.

23.1.15. map_bbbtree__list_to_map(+KeyValList,-Bbbtree)

Bbbtree is the map for the key-value pairs given as a list in **KeyValList**.

23.1.16. map_bbbtree__sorted_list_to_map(+SortedKeyValueList,-Bbbtree)

SortedKeyValueList is a sorted list of key value pairs; **Bbbtree** is the corresponding map.

23.1.17. map_bbbtree__sorted_list_to_map_len(+SortedKeyValueList,-Bbbtree,+Len)

SortedKeyValueList is a sorted list of key value pairs; **Bbbtree** is the corresponding map. **Len** is the lenth of the list.

23.1.18. map_bbbtree__to_sorted_list(+Bbbtree,?SortedKeyValList)

SortedKeyValList is a sorted list of the key-value pairs in the map **Bbbtree**.

24. help: The Help System

The help module provides support to create both on-line and off-line documentation on Prolog programs. Documentation must be defined by the hook predicate help_info/4. Documentation on a per module basis is provided if a help_info(module,Module,TitleString,DescriptionString) definition is given for Module. In that case the system also checks for Module:help_info/4 definitions.

The module supports production of the help information on standard output, (which can be converted into html format), and there also is an interface to a graphical user interface based on library(tcltk).