genfunlib Developer Documentation

Ideas and notes

computer-readable centralized database of mathematical facts proved by humans, but not yet provable by computers

program that extracts math formulas and text from papers and converts the theorems to computer-processable form

combination of “central database for data such as random mapping results”, the algolib encyclopedia, and building known mathematical facts into a CAS idea to be used in general **Reduce**-type solving

challenges: format for computer-readable math, difference between formulas in first paper and *Mathematica* code that implements them -- which one do we use?

possible components:
code in .m files
data in .m files (if small enough)
user documentation: tutorial, guide, help pages - only a pointer to mathematical background
formal specification?
tests
developer documentation: ?
proofs of correctness?
what about description of new math involved and connection to Notes/publishing?

The User Documentation doesn’t talk about how the implementations compute; Developer Documentation or code-docs do

Programmatic formatting for Mathematica code - possible?
Syntax highlighting for your own functions
Setting Up Mathematica Packages
Making Mathematica packages

User documentation method:
Authoring Using DocumentationTools
Mathematica Development User Guide > Tasks > Mathematica Documentation

mathematical background - point to references, we shouldn't write about that if it isn't necessary

Put Web links to the project on relevant Web pages

Wolfram|Alpha

Package pallettes?

CapitalCase for public symbols, lowerCase for private symbols

blah

GFeq2asymptoticCoef(gdev)
rec2GFeq

"override" **GeneratingFunction**

GFeq2GF(KernelMethod)

GFeq2rec

GFeq2coefs

differentiate eqn, set var to 0, solve

- **GF Frameworks**

  {DFA, Regex, RRGrammar}2Spec?

  (not necessary to obtain GFs)

- **Species**

- **Symbolic Method**

  Spec2GFeq

  Labeled constructions: sum, product, seq, cycle, set, pointing, substitution
  Unlabeled constructions: sum, product, seq, cycle, (set), multiset, pointing, substitution

  Restrictions:

  Number of components in final multiset/sequence/cycle object
  Multiplicity of each structure in the multiset/sequence/cycle (for unlabeled classes)
  Sizes of final objects

  Additional params: additional atomic classes, attribute grammars

  Bonus:implicit specs

- **Regular Languages**

  The authors of *Combinatorica* say, "Our aim in introducting permutation groups into Combinatorica is primarily for solving combinatorial enumeration problems. We make no attempt to efficiently represent permutation groups or to solve many of the standard computational problems in group theory." The situation for this package and automata/-grammar algorithm performance is similar.

  private symbols can't interfere with previously defined symbols in the *Mathematica* session (in "Global`").

  Letters are represented by nonempty **String**s, words are represented by **List**s of letters.

  Options can be used with the input validation scheme. validationRequired just has to be passed *before* any options

  Writing user documention (docs folder) last is OK as long as in-code documentation and this file are written diligently

  - ***Public (Exported) Symbols with Downvalues***

    NFA

    from **DFA**: direct
    from **Regex**
    extract alphabet, then pass to private nonvalidating recursive function
    uses nfa *, concat, union
    from **RRGrammar**: direct
    from **Digraph**: direct

    DFA

    from **NFA**: powerset construction, minimize
    from **Regex**: via NFA

from **RRGrammar**: via NFA
from **Digraph**: via NFA

Regex
from **NFA**: via DFA
from **DFA**: state elimination algorithm
from **RRGrammar**: via DFA
from **Digraph**: via DFA

RRGrammar
from **NFA**: direct
from **DFA**: via NFA
from **Regex**: via NFA
from **Digraph**: via NFA

Digraph
from **NFA**: via DFA
from **DFA**: direct
from **Regex**: via DFA
from **RRGrammar**: via DFA

public rule for Regex <-> RegularExpression

The following take one of **DFA, NFA, Regex, RRGrammar, Digraph**
RegStar
via Regex
RegComplement
via DFA
RegReverse
via Regex

The following take two (of the same kind) of **DFA, NFA, Regex, RRGrammar, Digraph**
RegUnion
via Regex
RegConcat
via Regex
RegIntersection
via DFA
The interesection of the alphabets is taken.

{NFA,DFA,Regex,RRGrammar,Digraph}2GF
allow the user to provide a function mapping each letter to a symbol/"weight"

Disambiguate
takes **{Regex,RRGrammar,Digraph}**
Digraph disambiguation is converting to a DFA and back
AmbiguousQ
takes **{Regex,RRGrammar?,NFA?,Digraph}**
ask on SE for "?" cases

Representation Descriptions

NFA

`NFA[numStates_Integer, alphabet_, transitionMatrix_, acceptStates_?VectorQ, initialState_]`

number of states: integer >=0, where 0 states means null language

alphabet: nonempty sorted list of distinct strings, not containing "". A value of **`{Null}`** implies 0 states.

transition matrix: numStates by alphabet size+1 matrix where entry i,j is a list of (valid) states accessible from state i and letter j = alphabet[j]. "Letter" alphabet size+1 is ϵ

accept states: list of integers between 1 and number of states

initial state: integer between 1 and number of states

DFA

`DFA[numStates_Integer, alphabet_, transitionMatrix_, acceptStates_?VectorQ, initialState_]`

number of states: integer >=0, where 0 states means null language

alphabet: nonempty sorted list of distinct strings, not containing "". A value of **`{Null}`** implies 0 states.

transition matrix: numStates by alphabet size matrix where entry i,j is the (valid) state accessible from state i and letter j.

accept states: list of integers between 1 and number of states

initial state: integer between 1 and number of states

String Regular Expression

string, with wrapping head **`RegularExpression`**, containing [a-z,A-Z,0-9,*,(,),|,] and is a valid *Mathematica* regular expression (POSIX ERE I think)

Empty string accepts just ϵ

`RegularExpression[Null]` for empty language

Symbolic Regular Expression

expression with head **`Regex`** built up from nonempty strings, **`EmptyWord`** and **`star,concat,or`**

`Regex[Null]` for empty language

see **`simplifyRawRegex`** for more info

Right Regular Grammar

`RRGrammar`-wrapped list of rules in the form **`sym_Symbol → RHS`** or **`sym_Symbol[n_Integer] → RHS`**,

where **`RHS`** is either **`EmptyWord`**, a string, **`sym_Symbol`**, where **`sym`** is in a LHS, **`sym_Symbol[n_Integer]`**, where **`sym[n]`** is in a LHS, **`concat[str_String, sym_Symbol]`**, **`concat[str_String, sym_Symbol[n_Integer]]`**, or **`or[args__]`**, where **`args`** is a sequence of those things. Strings cannot be empty.

An empty list corresponds to the null language.

Digraph

`Digraph[graph_, startVertices_, endVertices_, eAccepted_]`

graph: a directed graph, with vertices labeled with nonempty strings

startVertices: list of vertices of graph; if empty: null languge (ϵ may still be accepted)

endVertices: list of vertices of graph; if empty: null language (ϵ may still be accepted)

eAccepted: True if ϵ is accepted, False otherwise
Graph with 0 vertices is null language (ϵ may still be accepted).

ambiguity test via NFA test (see Book and Even papers -- is Book necessary, would ordinary construction work?) or recursive test (see Brabrand and Thomsen)
"a**" is not considered ambiguous in Book, niether is "a* | b*". *our* definition of ambiguity must include *e*.

Bonus: words with occurrences of patterns
Bonus: accept more regex syntax