Genfunlib Developer Documentation

Ideas and notes

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, usage messages
 formal specification?
 tests
 developer documentation: this and code comments
 proofs of correctness?

The User Documentation doesn't talk about how the implementations compute; developer documentation does. Extra, additional information will be found in Andrew MacFie's master's thesis.

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 and usage messages for public symbols, lowerCase for private symbols

Subpackages

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**

  This subpackage allows regular languages to be represented by any of the following regular language representations (RLRs): NFA, DFA, regular expression, right regular grammar, or directed graph with labeled vertices. Any RLR can be converted to any other. The following operations on RLRs are supported: union, intersection, complement, reverse, concatenation, star. Generating functions for RLRs can be computed by specifying a weight/marker for each letter in the alphabet.

  Re messages, from the Guidebook: "As a rule of thumb, messages are not generated for "symbolic" input if the function they appear in is used in classical mathematics. A scalar product is used in classical mathematics, so no message was produced in the last case. A table (a list) is not, so *Mathematica* produced a message."
  -The FiniteFields package largely doesn't do input validation. It sometimes performs weakish syntactic validity checks, sometimes performs total semantic validity checks and sometimes sends error messages (on failure all checks result in an expression returning unevaluated).
  -The Splines package does only weak syntactic input validation.
  -*Mathematica* built-in downvalues validate **any** sequence of arguments and send messages on errors.

  How to do efficient and simple validation remains a mystery. The result of a successful RegUnion command, for example, is going to be valid, but when it's passed to another function, it's checked for validity anyway. One option is for all functions to store the validity of their results right before they return them, by setting a downvalue of the "validate" symbol (validate[ret] = True; ret). This system could be altered by making the validate symbol only remember the last *n* such expressions. A somewhat-relevant reference is this.

  If a public function calls another public function, it always passes valid input. One way to avoid unnecessary computation of the validity is to pass an option saying "validation not required"; another is for public functions never to call public functions.

  Currect validation scheme: Public downvalues call validation directly (right in their definition) unless told not to by the validationRequired option; private downvalues don't do validation. Using validationRequired saves some computation at the expense of more complicated code.
  Data representations like DFA[_, _, _, _, _] don't do validation themselves, like RegularExpression and Graph in *Mathematica* built-in rules.

  private symbols are defined before the first public symbol downvalue they're used in

  The authors of *Combinatorica* say, "Our aim in introducing 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 don'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.

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

□ *Public (Exported) Symbols with Downvalues*

Conversions

ToNFA

        from **DFA**: via Regex

        from **Regex**

                extract alphabet, then pass to private nonvalidating recursive function

                uses nfa *, concat, union

        from **RRGrammar**: direct

        from **Digraph**: direct

ToDFA

        from **NFA**: powerset construction, minimize

        from **Regex**: via NFA

        from **RRGrammar**: via NFA

        from **Digraph**: via NFA

ToRegex

        from **NFA**: via DFA

        from **DFA**: state elimination algorithm

        from **RRGrammar**: via DFA

        from **Digraph**: via DFA

ToRRGrammar

        from **NFA**: direct

        from **DFA**: via NFA

        from **Regex**: via NFA

        from **Digraph**: via NFA

ToDigraph

        from **NFA**: via DFA

        from **DFA**: direct

        from **Regex**: via DFA

        from **RRGrammar**: via DFA

Regex <-> RegularExpression conversion:
ToRegex[RegularExpress[...]]
ToRegularExpression[Regex[...]]

        usage string for RegularExpression is joined to built-in one

Operations

The following take one of **DFA, NFA, Regex, RRGrammar, Digraph**

RegStar
    via NFA
RegComplement
    via DFA
    takes alphabet as second parameter
    equals alphabet* \ L(dfa)
RegReverse
    via Regex

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

GFs

GeneratingFunction[regex, rules]
    allow the user to provide a function mapping each letter to a symbol/"weight" in the form of Rules

Bonus

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
     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*.

□ *Representation Descriptions*

NFA
        NFA[numStates_Integer, alphabet_, transitionMatrix_,
         acceptStates_?VectorQ, initialState_]
    number of states: integer >=0, where 0 states means null language
    alphabet: sorted list of distinct strings, not containing "". A value of **{}** means the empty language or {ϵ}.
     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]. The (alphabet size+1) "letter" is ϵ.
            if numStates = 0, transitionMatrix = { }
            if alphabet = { }, transitionMatrix has one column (if there are any rows)
    accept states: list of integers between 1 and number of states
    initial state: integer between 1 and number of states, or Null iff numStates = 0

DFA

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

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

alphabet: sorted list of distinct strings, not containing "". A value of `{}` means the empty language or {ϵ}.

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

> if numStates = 0, transitionMatrix = { }
>
> if alphabet = { }, transitionMatrix = {{ }, { }, ...}

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

initial state: integer between 1 and number of states or Null if numStates = 0

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 **RegexStar,RegexConcat,RegexOr**

Regex[Null] is 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, **RRGrammarConcat[str_String, sym_Symbol]**, **RRGrammarConcat[str_String, sym_Symbol[n_Integer]]**, or **RRGrammarOr[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). empty list means empty language (ϵ may still be accepted)

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

eAccepted: True if ϵ is accepted, False otherwise

Graph with 0 vertices means empty language (ϵ may still be accepted).

Bonus: words with occurrences of patterns

Bonus: accept more regex syntax