# Genfunlib Developer Documentation

factorial moments!

## Ideas and notes

Genfunlib is a selection of *Mathematica* implementations of generating function-related symbolic methods

Components:

- code in .m files
- developer documentation: this file (and code comments)

Additional information will be found in Andrew MacFie's master's thesis.

In this document, point to external references for mathematical content if possible

non-negativity of power series coefficients: Kauers - Computer Algebra and Power Series with Positive Coefficients

test equality of power series?

## GFeqn → asymptotic

Todo: everything

## Species

Todo: everything

Sage, aldor-combinat, book

Main objects: multisort, weighted, virtual species

Species are represented by expressions built up from predefined species and operations, with no particular head, i.e. not wrapped in `Species`

predefined species

- Empty
- Characteristic
- Cycle
- Partition
- Permutation
- Linear-order
- Set
- Subset

weighted species

- singletons may be weighted with an expression

nonvirtual operations

- sum
- product
- partitional composition
- derivative

pointing
functorial composition
powerset
restriction

virtual operations
multiplicative inverse
combinatorial logarithm
...

toGF(eqn)s
generating series
isomorphism type generating series
cycle index series
partial types
`GFEqns[Spec[...],indeterminate]`?

molecular expansions: see Devmol [Auger]
any generalizations possible?

Areas for `restrictedSum` improvement:

`#≤a&&#≥b&`

`ToGenfunlibSpec`
takes Combstruct grammar as a String and whether it’s labeled and returns spec
for use with http://algo.inria.fr/ecs/
converts upper case symbols to doubled lower case symbols

### Bonus

More counting frameworks that might be convertible to species: ECO, (see “Generating functions for generating trees”), object grammars

test species isomorphism
http://math.stackexchange.com/questions/396004/testing-combinatorial-species-for-isomorphism

complete molecular expansions

semantic spec validity testing, a.k.a. checking for well definedness of a spec
Partial alg: [Bruno]; for MV: valuation computed for each param; two levels of iteration, one for each param, one for each nonterminal; for substitutions, a similar polynomial-checking step is needed
Bruno says the species paper has more on this

# SymbolicMethod

Todo: write Species subpackage, using code from here, then write SM→Species converter in this subpackage

Specification: `Spec[{lhs==rhs,...},labeled?]`
the left hand sides are symbols representing classes, the right hand sides are expressions built with constructions, specification classes, and atomic and neutral classes
Labeled constructions: sum, product, seq, cycle, set, pointing, substitution
`SMPlus, SMTimes, SMSeq, SMCyc, SMSet, SMPointing, SMSub`
Unlabeled constructions: sum, product, seq, cycle, multiset, pointing, substitution
`SMPlus, SMTimes, SMSeq, SMCyc, SMSet, SMMultiset, SMPointing, SMSub`

Syntax for `SMPointing`: `SMPointing[class,paramNumber]`
Syntax for `SMSub`: `SMSub[baseClass,substitutedClass,paramNumber]`

Restrictions:

Number of components in final set/multiset/sequence/cycle object

Option for those heads: `Cardinality→predicateOnIntegers`

Parameter values of final objects

`Restricted[class,{atomicClassNum→func,...}]`, where `func` is a predicate on the integers specifying the allowed set of values for parameter `atomicClassNum`

predicates must by symbolic-friendly (not PrimeQ) -- like in GeneratingFunction

Atomic class: `ZClass[1]`, Neutral class: `EClass`

Additional params:

additional atomic classes

`ZClass[2], ZClass[3],...`

marked by `indeterminate[2],indeterminate[3],...`

Todo: semantic input validation: see Species

# RegularLanguages

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 regular expressions 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 input validation remains a mystery. The result of a successful RegUnion command, for example, is guaranteed to be valid RLR, 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.

Validity-checking rules that return more information than True/False (i.e. return conditions) can have their extra info captured in a Module variable like this:

```
f[]:=Module[{valid,b=3},
(
Print[valid];
)/;(valid=validity[b])
];
```

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.

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

An alternative approach to providing RL functionality would have been to make a *J/Link* interface to brics.

## Public (Exported) Symbols with Downvalues

Conversions

**`ToNFA`**

- from **`DFA`**: via Regex
- from **`Regex`**
  - uses nfa *, concat, union
- from **`RRGrammar`**: direct
- from **`Digraph`**: direct
  - LineGraph construction

**`ToDFA`**

- from **`NFA`**: powerset construction, minimize
  - Todo: too slow
  - Remove states from which no end state is accessible, from NFA
  - Create only elements of the powerset that are possible
  - Optimize code

- 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
  - LineGraph construction
- 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

Todo: replace RegStar, RegUnion, RegConcat with grammar versions from contextFree.m, then delete contextFree.m

GFs

**`GeneratingFunction[regex, rules]`**

allow the user to provide a function mapping each letter to a symbol/"weight" in the form of Rules

Todo: add overload that allows more easily specifying that just, for example, $z$ should be used for all letters

## 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 {$\epsilon$}.

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 $\epsilon$.

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 {$\epsilon$}.

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.
Todo: the phrases underlined and bold are not uniformly ahered to

**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

regexes with symbolic parameters ("a" *k* times, etc.)
SE question

`AmbiguousQ`
takes `{Regex,RRGrammar,NFA,Digraph}`
RRGrammar: convert to NFA
NFA: convert to regex (ambiguity preserved by Theorem 2 in Book)
regex: recursive test from Brabrand and Thomsen paper
digraph: iff an accessible state has edges to more than one vertex with the same label
“a | aa” is *1-unambiguous*, “a(*e* | a)” is *0-unambiguous*. 1-unambiguity is the definition used here.
(alternative test: NFA test (see Book and Even papers). Is Book algorithm necessary, would ordinary construction work? Note that “a**” is not considered ambiguous in Book; niether is “a* | b*”. *Our* definition of ambiguity must include *e*.)

# Util

egf2ogf, ogf2egf
method: Laplace transform
could also try manual method with `SeriesCoefficient` and `GeneratingFunction`
Todo: finish implementation

# GFeqn → coefs

Derivatives method

use **Series** on both sides, equate coefs starting from index 0,1,2,...

Todo: implement

Newton iteration method

Todo: finish implementation

# rec2GFeq

see also GeneratingFunctions

"override" **GeneratingFunction**

Todo: areas for improvement:

```
GeneratingFunction[n^k f[n],n,x]
GeneratingFunction[Sum[c[i]*f[n], {i, 0, k}], n, x]
GeneratingFunction[f[n+i],n,x]
GeneratingFunction[Sum[f[n+i], {i, 0, k}], n, x]
GeneratingFunction[1/(n+1) f[n], n, x]
GeneratingFunction[Boole[Divisible[n,2]],n,x]
GeneratingFunction[Boole[Mod[n,2]==0]f[n],n,x]
GeneratingFunction[Boole[n≥1]f[n],n,x]
GeneratingFunction[UnitStep[n-k]f[n],n,x]
```

# GFeq2rec

see also `SeriesCoefficient[DifferentialRoot[lde]]`, `SeriesCoefficient[Root[ae]]` and GeneratingFunctions

“override” SeriesCoefficient

No way to represent known(unknown($z$)) compositions (can’t do “symbolic lists”)

unknown(known($z$), ...) compositions can only be done (for the same reason) for fixed expressions like unknown($k\,z$) or unknown($k\,z$, $j\,z$)

ref/Series: “Series by default assumes symbolic functions to be analytic”

For singular functions, SeriesCoefficient can do rational-power expansions. In that case, the Cauchy product rule for series multiplication doesn’t hold. To use a simplification rule for products of series, we have to determine whether the factors are analytic. Doing that automatically would be an interesting challenge, however, for simplicity, we merely allow the user to specify when the factors should simply be assumed to be analytic. But... built-in functions cannot have new options added.

Also, negative-power expansions can be done, so sometimes that possibility should be ignored.

Current system: global variable called **$FullAnalytic**, which, if true, means that the Cauchy product rule is assumed always applicable.

Todo: recurse over second-argument lists (MV support)

Todo: areas for improvement:

```
SeriesCoefficient[Sum[f[k[i]*x],{i,1,m}],{x,0,n}]
```