# The Littlewood-Richardson Rule, Theory and Implementation

Marc A. A. van Leeuwen

*Université de Poitiers, Département de Mathématiques,*
*40 Avenue du Recteur Pineau, 86022 Poitiers, France*
maavl@univ-poiitiers.fr

ABSTRACT

We present the implementation of the Littlewood-Richardson rule in LiE. We describe the mathematical problem it applies to, formulate the rule, and indicate a proof. In a brief historical sketch we indicate some early formulations and partial proofs. We derive a formulation of the rule that can be implemented very efficiently.

*1991 Mathematics Subject Classification:* 05E05, 05E10, 20-04, 20C30, 20G05.
*Keywords and Phrases:* Schur functions, Littlewood-Richardson rule, Robinson-Schensted correspondence, tableaux, pictures, literate programming.
*Note:* Work carried out under project MAS1.4 (Analysis of PDE's :-)
The program contained in this paper is part of the LiE computer algebra package developed at CWI.

**Introduction.**

The main part of this paper consists of a precise rendering of the implementation of the Littlewood-Richardson rule in the computer algebra package LiE. To accomodate uninitiated readers, we give a short introduction to the conventions of the `CWEBx` system for literate programming used for preparing and presenting the source code, and to the relevant aspects of the context provided by the main program of LiE.

A `CWEBx` document consists of a sequence of numbered *sections*, each of which contains a descriptive text, optionally followed by a program fragment. The descriptive text is just a piece of mathematical text, discussing the following program fragment if present, or else making comments not relating to any particular piece of code. The program fragments are pieces of C-code, formatted in a particular way for better readability; this means that they are not a literal representation of the program text seen by the compiler. The most important way in which the presentation is adapted, is that certain program parts are moved to a separate section. In the context where such a code fragment occurs, it is represented by a *module name*, and the same module name heads the actual code fragment in the section it has been moved to: it is preceded there by the descriptive text of that section, and followed by '$\equiv$' to indicate that it is being defined. A module name has the form '$\langle$ Text describing the task performed by this module $n\rangle$', where $n$ is the number of the section where the module is further specified. Apart from this, some minor adaptations to the program text are made in presenting it: fonts and spacing are chosen according to the syntactic structure, and some operators are represented by mathematical symbols, according to the following table.

| *operator* | = | == | != | <= | >= | && | \|\| | ! | -> | ^ |
|---|---|---|---|---|---|---|---|---|---|---|
| *symbol* | $\Leftarrow$ | $=$ | $\neq$ | $\leq$ | $\geq$ | $\wedge$ | $\vee$ | $\neg$ | $\rightarrow$ | $\oplus$ |

Certain identifiers are represented by special symbols as well. Normally this only applied to the identifier `NULL`, which is represented as '$\odot$', but other identifiers can be made to behave similarly. This is introduced by a line (that is not passed to the compiler) of the form

**format** *lambda* $\odot$ $(\lambda)$

which states that the identifier *lambda* is treated similarly to `NULL`, and that it is represented as '$\lambda$'.

Mathematical functions in LiE are implemented in C, and can make use of some functions provided for general use in LiE. These involve mainly the central data types used throughout LiE, the associated memory management functions, ans functions providing some basic operations on these data types. We shall briefly describe these to the extent necessary for understanding their use in the mathematical library. Therefore the data type declarations below do not always precisely match the actual ones in LiE. We start with two integer types that are used throughout LiE

```
typedef long index;      /* for bounds of arrays, and for indices into them */
typedef int entry;       /* for the contents of arrays */
```

Thus a choice concerning the compactness of matrices and polynomials versus the risk of arithmetic overflow of their entries and exponents (for most purposes this is rather unlikely, and even setting **entry** to be **signed char** would be possible) can be made without restricting the overall size of objects. This distinction is maintained as far as possible consistently even in the declarations of local variables. We continue with the main data types, used to represent the objects manipulated by LiE.

```
typedef struct { ... } bigint;      /* none of the fields should be accessed directly */
typedef struct { index ncomp; entry *compon; } vector;
typedef struct { index nrows, ncols; entry **elm; } matrix;
typedef struct { index nrows, ncols; entry **elm; bigint **coef; } poly;
```

The type **bigint** represents integers of (practically) unlimited size, and functions are provided for all arithmetic operations on them. The *ncomp* field of a **vector** gives its number of entries, and the *compon* field points to the first entry (so internally indexing starts at 0, as usual in C; the indices visible to the user of LiE are systematically 1 higher). Similarly a **matrix** has two dimension fields *nrows* and *ncols*, and the actual entries are accessed via *elm*, which points to an array of pointers to the rows of the matrix. In a **poly** there is an additional field *coef* that points to an array of pointers to the **bigint** coefficients of the polynomial; the remaining fields have the same names as those of matrices (for historical reasons), but now *nrows* gives the length, *ncols* the number of indeterminates (size of the exponents), and the exponents are accessed through *elm*. The following functions are used for (de)allocating internal, unpacked, arrays.

```
void *safe_alloc (size_t size);      /* like malloc, but won't return ⊙ */
#define alloc_array(type_arg,size) ((type_arg*) safe_alloc((size) * sizeof(type_arg)))
entry *mkintarray (index n);      /* alloc_array(entry,n) */
void freearr (void *array);
```

The next functions are used for allocating vectors, matrices, and polynomials; deallocation (using *freemem*) is optional, since after each completed command the garbage collector cleans up any unaccessible objects.

```
vector *mkvector (index n);      /* make an uninitialised vector of size n */
matrix *mkmatrix (index r, index c);      /* a matrix with r rows and c columns */
poly *mkpoly (index r, index c);      /* a polynomial of length r in c indeterminates */
```

The following are used for constant values

```
typedef enum { false, true } boolean;
bigint *one;      /* the constant 1 */
poly *poly_null (index n);      /* zero polynomial 0X[0,...,0] in n indeterminates */
poly *poly_one (index n);      /* unit polynomial 1X[0,...,0] in n indeterminates */
```

The following functions of general utility are used.

```
void copyrow (entry *a, entry *b, index n);      /* copy n entries from a to b */
bigint *mult (bigint*, bigint*);      /* multiplication */
poly *Addmul_pol_pol_bin (poly *p, poly *q, bigint *c);      /* p + c * q */
poly *Reduce_pol (poly *p);      /* sort and combine terms with common exponent */
```

Finally, we can use a global accumulation facility when an algorithm builds up a polynomial by repeatedly contributing single terms; using it is more efficient than doing the same using polynomial arithmetic.

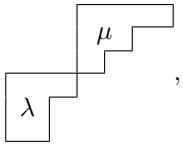
```
void wt_init (index n);      /* initialise accumulation of weights of size n */
void wt_ins (entry *weight, bigint *coef, boolean negate);      /* contribute ±coef X weight */
poly *wt_collect (void);      /* finalise accumulation and return accumulated polynomial */
```

**1. The Littlewood-Richardson rule.** This file describes the implementation in LiE of the Littlewood-Richardson rule, which gives combinatorial method to expand a product of Schur polynomials as a linear combination of Schur polynomials. Some background information concerning the justification of the rule, its various formulations, and its history are also given. Since Schur polynomials in $n$ variables are the characters of irreducible representations of $\mathbf{GL}_n$, this algorithm given here provides an alternative for *tensor* (which uses Klymik's formula) when dealing with groups of type $A_{n-1}T_1$, and a *fortiori* for groups of type $A_{n-1}$.

One can associate to each partition $\lambda$ with $l(\lambda) \leq n$ (here $l(\lambda)$ is the number of non-zero parts of $\lambda$), a Schur polynomial $s_\lambda$, homogeneous of degree $|\lambda|$ in the indeterminates $x_0, \ldots, x_{n-1}$, as follows. A semistandard tableau of shape $\lambda$ is a filling of the Young diagram of $\lambda$ with natural numbers, such that they increase weakly along rows and increase strictly down columns; $s_\lambda$ is the sum over all semistandard tableaux $T$ of shape $\lambda$ with entries $< n$, of $x_0^{a_0} \cdots x_{n-1}^{a_{n-1}}$, where each $a_i$ counts the entries $i$ in $T$. We call $(a_0, \ldots, a_{n-1})$ the weight $\mathrm{wt}(T)$ of $T$, and for $x_0^{a_0} \cdots x_{n-1}^{a_{n-1}}$ we shall write $x^{\mathrm{wt}(T)}$. Although not obvious from this description, these Schur polynomials are invariant under permutations of the indeterminates, and in fact form a linear basis for the space of all such symmetric polynomials. Although $s_\lambda$ depends on the number $n$ of indeterminates, it can be recovered from the corresponding Schur polynomial in any larger number of indeterminates by substituting 0 for all $x_i$ with $i \geq n$; for Schur polynomials associated to partitions $\lambda$ with $l(\lambda) > n$, such a substitution results in 0. The dependency on $n$ can be removed by working in the ring of symmetric functions (see [Macd]), of which the ring of symmetric polynomials in $n$ indeterminates is a quotient; symmetric functions corresponding to Schur polynomials $s_\lambda$ exist for any partition $\lambda$, and are called Schur functions. The Littlewood-Richardson rule can be formulated for symmetric functions, but our implementation assumes symmetric polynomials in $n$ indeterminates; results valid for symmetric functions can be obtained by choosing $n$ sufficiently large.

Each Schur polynomial $s_\lambda$ is the character of an irreducible representation of $\mathbf{GL}_n$: it gives the trace of the image of diagonal matrices with diagonal entries $x_0, \ldots, x_{n-1}$. The weights so occurring as exponents are not expressed using fundamental weight coordinates, as is done elsewhere in LiE; appropriate transformation functions will be defined below. The height partial ordering on weights is generated by relations $(a_0, \ldots, a_{n-1}) \prec (a_0, \ldots, a_i + 1, a_{i+1} - 1, \ldots, a_{n-1})$ for $0 \leq i < n-1$, i.e., a monomial can be raised by replacing a factor $x_{i+1}$ by $x_i$; then $\lambda$ is the highest weight occurring in $s_\lambda$. Since a product $s_\lambda s_\mu$ can be interpreted as the character of a tensor product representation, it can be decomposed as $\sum_\nu c^\nu_{\lambda,\mu} s_\nu$, where the $\nu$ are partitions of $|\lambda| + |\mu|$, and $c^\nu_{\lambda,\mu}$ are non-negative integers, called the Littlewood-Richardson coefficients. Combinatorially this means that the set of all pairs of semistandard tableaux with entries $< n$ and shapes $\lambda$ and $\mu$ can be partitioned into subsets, for each of which the collection of weights defines a Schur polynomial $s_\nu$. Such a partition can be obtained effectively, though inefficiently, by initialising $S$ to the set of all such pairs, and then repeatedly selecting some $p \in S$ with a maximal weight $\nu$, and adjoining to it other elements of $S$, to complete and split off a subset with weights corresponding to $s_\nu$, until $S$ is depleted. The Littlewood-Richardson rule specifies a subset of $S$, whose elements we shall call Yamanouchi tableaux, which for a suitably chosen partitioning of $S$ are precisely the elements with the highest weight within their subset. Thereby the determination of the coefficients $c^\nu_{\lambda,\mu}$ reduces to the enumeration of Yamanouchi tableaux.

The Littlewood-Richardson rule applies to the slightly more general context of the decomposition of skew Schur functions. If $\lambda, \mu$ are partitions with $\mu \subseteq \lambda$ (inclusion of Young diagrams), then the difference set between their Young diagrams is called the skew diagram $\lambda/\mu$, and semistandard skew tableaux are defined by filling a skew diagram with numbers, in the same way as ordinary semistandard tableaux. The skew Schur polynomial $s_{\lambda/\mu}$ is defined as the sum over all semistandard skew tableaux $T$ of shape $\lambda/\mu$ of $x^{\mathrm{wt}(T)}$. It can be seen that the product $s_\lambda s_\mu$ is equal to the skew Schur polynomial $s_{\lambda \uplus \mu}$, where $\lambda \uplus \mu$ is the skew diagram built from $\lambda$ and $\mu$ as follows:



,

so that the decomposition of skew Schur polynomials indeed generalises that of products of Schur polynomials. On the other hand, it can be shown that the coefficient in $s_{\nu/\lambda}$ of $s_\mu$ is equal to the Littlewood-Richardson coefficient $c^\nu_{\lambda,\mu}$, so that no new combinatorial quantities arise in the more general problem.

On semistandard skew tableaux $T$ with entries $< n$, we specify partially defined "raising operations" $e_i$ for $0 \leq i < n-1$. One can apply $e_i$ to $T$ if and only if there is a subtableau $T'$ consisting for some $k$ of the rightmost $k$ columns of $T$, such that $i+1$ occurs more often as entry of $T'$ than $i$ does; i.e., if $\mathrm{wt}(T') = (a_0, \ldots, a_{n-1})$ then $a_{i+1} > a_i$. When this condition is satisfied, consider the maximal value of $a_{i+1} - a_i$ occurring for such subtableaux, and let $\bar{T}$ be the smallest one for which this maximum is attained. The leftmost column of $\bar{T}$ must contain an entry $i+1$, and $e_i(T)$ is defined by changing the corresponding entry of $T$ into $i$ (note that $\mathrm{wt}(e_i(T)) \succ \mathrm{wt}(T)$). When none of the $e_i$ can be applied to $T$, then $T$ is a called a Yamanouchi tableau; this happens when $\mathrm{wt}(T')$ is a partition (i.e., weakly decreasing) for all subtableaux $T'$ described above; in particular $\mathrm{wt}(T)$ is a partition. The essence of the Littlewood-Richardson rule is that each Yamanouchi tableau $T$ of shape $\nu/\lambda$ corresponds to an occurrence of $s_{\mathrm{wt}(T)}$ in $s_{\nu/\lambda}$. Whether a given $T$ is a Yamanouchi tableau can be tested by traversing the columns of $T$ from right to left, and test at each stage whether the accumulated weight is a partition. If the entries of each column are taken from top to bottom, one may test after encountering any entry $i+1$ that its accumulated multiplicity does not exceed that of $i$ (if it does, this will still hold at the end of the column); we shall call this order of traversal the column reading order. Traditionally, Yamanouchi tableaux are defined by a similar test, but using instead the row reading order: $T$ is traversed by rows from top to bottom, reading each row from right to left. It can easily be shown that this test is equivalent to the one using the column reading order.

While not needed to formulate the Littlewood-Richardson rule, the operations $e_i$ are important for justifying it. If $S$ is the set of semistandard skew tableaux of shape $\nu/\lambda$ and entries $< n$, a graph with vertex set $S$ can be defined that has edges from $T$ to $e_i(T)$ whenever such an application of $e_i$ is possible; for the purpose of defining isomorphism of graphs, we shall consider such an edge to be oriented, and labelled by the index $i$. This graph defines a partition of $S$ into connected components, each of which obviously contains at least one Yamanouchi tableau. The justification of the Littlewood-Richardson rule is based on the non-trivial fact that if it contains a Yamanouchi tableau of weight $\mu$, then the component is isomorphic to the (connected) graph similarly defined on the set of semistandard tableaux of shape $\mu$ and entries $< n$. In particular the Yamanouchi tableau is unique in its component, its weight $\mu$ is the highest one occurring in the component, and the multiset of all weights in the component corresponds to the Schur polynomial $s_\mu$.

Assuming this fact for the moment, we may associate to each $T \in S$ the pair $(Y, P)$, where $Y$ is the Yamanouchi tableau in the connected component of the graph containing $T$, and $P$ is the tableau of shape $\mathrm{wt}(Y)$ corresponding to $T$ under the graph isomorphism. This defines a bijection from $S$ to the set of all pairs $(Y, P)$ with $Y$ a Yamanouchi tableau of shape $\nu/\lambda$ and $P$ a semistandard tableau of shape $\mathrm{wt}(Y)$ with entries $< n$; moreover the association $T \mapsto P$ preserves weights. In fact, the existence of such a bijection suffices to prove the validity of the Littlewood-Richardson rule. A correspondence of this kind was first described by Robinson [Rob, §5], but he did not give a convincing proof that it was well-defined and bijective. Indeed, complete proofs of the Littlewood-Richardson rule were first given only about 4 decades after its original formulation. Robinson's proof can be completed by a detailed analysis of the graph structure defined on $S$; this is done in [Macd, I §9] (although we note that that proof, despite its considerable length, is still too hasty in concluding surjectivity of the correspondence). A much simpler proof can be given however using Schützenberger's *jeu de taquin* [Schü]: it provides a direct method to obtain $P$ from $T$ (without the need to determine $Y$) by a sequence of small transformations (slides), each of which can be seen to preserve the graph structure, since they commute with the raising operations.

Nowadays, Robinson's bijection is called the Robinson-Schensted correspondence, as it is considered to be essentially equivalent to the bijection defined much later, and by a different algorithm, in [Sche]. The most insightful way to understand both the Littlewood-Richardson rule, and the relations between Robinson's and Schensted's constructions and jeu de taquin, is to use instead of semistandard tableaux the concept of pictures ([JaPe], [Zel], [FoGr]), a type of combinatorial object whose definition combines monotonicity properties (as in the definition of semistandard tableaux) and properties like those used to characterise Yamanouchi tableaux. Indeed, the Robinson-Schensted correspondence has a generalisation to pictures that is very symmetric, and there are two commuting forms of jeu de taquin slides, which transform the picture equivalent of $T$ respectively into the picture equivalents of $P$ and $Y$, see [vLee]. Pictures also provide many ways of obtaining alternative versions of the Littlewood-Richardson rule, such as the one implemented below, which differs in several respects from the form in which the rule is traditionally stated.

**2.** Although not essential for understanding our implementation, let us make a few comments on the intriguing history of the Littlewood-Richardson rule; we try to interpret the old literature in the light of modern insights, which is not at all trivial. The original paper [LiRi] in which Littlewood and Richardson stated their rule (published in 1934), is mainly concerned with symmetric group characters and Schur functions (which term was introduced there); only §8 (out of a total of 16 sections) deals with the multiplication of Schur functions. Semistandard tableaux do not appear explicitly in the paper; Schur functions are defined instead in terms of symmetric group characters and power sum symmetric functions. The rule is stated in §8 as Theorem III, and is inspired by calculations with certain idempotent elements (called Characteristic Units) in the group algebra of the symmetric group, but it is proved only for the case where $l(\mu) \leq 2$.

It is noteworthy to recall the type of objects enumerated in their formulation of the rule to determine $c^{\nu}_{\lambda,\mu}$: two of tableaux $A, B$ are formed by filling the Young diagrams of $\lambda$ and $\mu$ with symbols, and these symbols are rearranged into a new Young tableau $C$ of shape $\nu$ according to the following rules. The first requirement is that $A$ is kept intact, so the symbols of $B$ are rearranged to form the skew subtableau $T$ of $C$ of shape $\nu/\lambda$. A next requirement amounts to the following: if in $T$ one replaces each symbol by the number of the row of $B$ containing it, then one should obtain a semistandard tableau (of shape $\nu/\lambda$ and weight $\mu$); moreover, in $T$ a fixed ordering is imposed among the symbols constituting a row of $B$, so that no semistandard tableau is obtained more than once. It is proved (in the general case) that the number of arrangements that satisfy these first requirements is the coefficient of $s_\nu$ not in $s_\lambda s_\mu$, but in $s_\lambda h_\mu$, where $h_{(\mu_0,\mu_1,\ldots,\mu_m)} = s_{(\mu_0)}s_{(\mu_1)}\cdots s_{(\mu_m)}$; in other words, semistandard tableaux of shape $\nu/\lambda$ and weight $\mu$ are considered to count occurrences of the Schur function $s_\nu$ in $s_\lambda h_\mu$, rather than of the monomial $x^\mu$ in $s_{\nu/\lambda}$ (nowadays these numbers are well known to be equal). A final restriction is then given (which corresponds to the Yamanouchi condition in our discussion above): if in $B$, each symbol is replaced by the number of the row of $T$ containing it, then one should also obtain a semistandard tableau (of shape $\mu$ and weight $\nu - \lambda$). The ordering criterion just mentioned for symbols forming a row of $B$ was already compatible with this restriction (the symbols of such a row, taken from left to right, must be placed in $T$ in rows with weakly increasing numbers), so the only extra requirement is that symbols forming a column of $B$, taken from top to bottom, must be placed in $T$ in rows with strictly increasing numbers.

If we interpret the rearrangement of symbols from $B$ to form $T$ as a bijection from the squares of $\mu$ to the squares of $\nu/\lambda$, then we get a bijection that can be read off as a semistandard tableau in two directions. Now the concept of pictures mentioned above, which was introduced much later, is such that pictures $\mu \to \nu/\lambda$ are bijections with precisely that property. In fact the bijections allowed according to the description of Littlewood and Richardson differ from pictures only in a minor detail: when some consecutive symbols from a row of $B$ are placed in a single row of $T$, then the original description requires that this happens "without disturbing their order", whereas in a picture, the left-right order is reversed. For the purpose of enumeration, the difference is irrelevant (neither of the associated semistandard tableaux is affected), and all that matters is that a fixed ordering is chosen to avoid overcounting; however, since the left-right ordering is also reversed when symbols from a row of $B$ are placed in *different* rows of $T$, the picture definition is the more natural one. If the original description is modified in this respect to match the definition of pictures, its relation to Yamanouchi tableaux becomes easier to perceive: when the symbols of $T$ are listed in the row reading order, then the set of symbols encountered up to any chosen point, matches that of a Young subtableau of $B$.

Although [LiRi] does not attempt to give a general proof of its Theorem III, it does formulate both a characterisation of the allowed arrangements that matches the definition of Yamanouchi tableaux given above (the term actually used is that reading off the symbols of $T$ in the row reading order should give a "lattice permutation" of those symbols), and the description of an operation that corresponds to our raising operations. These are introduced in the proof for the case where $l(\mu) = 2$, so the entries of the semistandard tableau corresponding to $T$ take only two distinct values; in this situation there is only one kind of raising operation (namely $e_0$) that is involved. The proof is based on the following instance of the Jacobi-Trudi identity: if $\mu = (q, r)$ with $q \geq r > 0$, then $s_\mu = h_\mu - h_{\mu'}$ where $\mu' = (q+1, r-1) \succ \mu$. Since the coefficient of $s_\nu$ in both $s_\lambda h_\mu$ and $s_\lambda h_{\mu'}$ can be determined by counting semistandard tableaux, one can find $c^{\nu}_{\lambda,\mu}$ by subtracting these numbers. The raising operation now maps a subset of the semistandard tableaux of weight $\mu$ bijectively onto those of weight $\mu'$, and the number of remaining ones (those corresponding to lattice permutations) directly gives $c^{\nu}_{\lambda,\mu}$ in this case.

Robinson's paper [Rob] builds forth on these ideas, but is quite difficult to read, due to its extremely obscure formulations, and the fact that all essential argumentation is implicit. However, it appears to proceed in the direction set forth in [LiRi], with one important exception: instead of using the Jacobi-Trudi identity to express $s_\mu$ in terms of $h_{\mu'}$, it uses descending induction on the weight $\mu$, based on the expression $s_\mu = h_\mu - \sum_{\mu' \succ \mu} K_{\mu'\mu} s_{\mu'}$, where $K_{\mu'\mu}$ is the coefficient of $s_{\mu'}$ in $h_\mu$ (nowadays, this is called a Kostka number). For instance, for $\mu = (q, r)$, one has $s_{(q,r)} = h_{(q,r)} - s_{(q+1,r-1)} - \cdots - s_{(q+r,0)}$. The map $T \mapsto (Y, P)$ (claimed without proof to be bijective), is used in the proof as follows. Each semistandard tableau $T$ of shape $\nu/\lambda$ and weight $\mu$ corresponds to an occurrence of $s_\nu$ in $s_\lambda h_\mu$. If the Yamanouchi tableau $Y$ (obtained from $T$ by repeatedly applying raising operations) differs from $T$, it corresponds by the inductive assumption to an occurrence of $s_\nu$ in $s_\lambda s_{\mu'}$, where $\mu' = \mathrm{wt}(Y) \succ \mu$; then $T$ is considered to correspond to that occurrence of $s_\nu$, where $s_{\mu'}$ is viewed as a constituent of $h_\mu$. When different tableaux $T$ give rise to the same $Y$, they are distinguished by the semistandard tableaux $P$ of shape $\mu'$ and weight $\mu$ also associated to them; indeed there are $K_{\mu'\mu}$ such tableaux, matching the number of distinct constituents $s_{\mu'}$ in $h_\mu$. The tableaux that remain, namely those with $Y = T$, must then correspond to the occurrences of $s_\nu$ in $s_\lambda s_\mu$, as claimed.

This would almost suggest that Robinson's proof, apart from the missing argument that the claimed bijection is indeed one, is easy to understand. One should however realise that neither semistandard tableaux nor the numbers $K_{\mu'\mu}$ occur explicitly; rather the decomposition $h_\mu = \sum_{\mu' \succeq \mu} K_{\mu'\mu} s_{\mu'}$ is given in the cryptic form $h_\mu = \sum [\prod S_{rs}^{\lambda_{rs}}](\mu)$, with the following "explanation" (quoted from [Yng]):

> *"$S_{rs}$ where $r < s$ represents the operation of moving one letter from the $s$-th row up to the $r$-th row, and the resulting term is regarded as zero, whenever any row becomes less than a row below it, or when letters from the same row overlap,—as, for instance, happens when $\mu_1 = \mu_2$ in the case of $S_{13}S_{23}$."*

What is meant appears to be the following. The summation is over certain collections of non-negative numbers $(\lambda_{rs})_{1 \le r < s \le l(\mu)}$, each of which gives rise to a product of formal operators $S_{rs}$. The nature of these operators is vague (it is not stated that they commute, but we must assume they do, as no order for the product is specified, nor is any systematic ordering used in the examples), but they can be applied to Young diagrams (or partitions), with $S_{rs}$ moving a square up from row $s$ to row $r$. They are not simply shape transformations, however, as there appears to be an implicit rule that no square can be moved twice by operators from the same product ($S_{13}$ is distinguished from the product $S_{12}S_{23}$); this might explain why no concern is given to their order. The intention of the formula must be that each non-zero product corresponds to a semistandard tableau of partition shape and weight $\mu$, with each factor $S_{rs}$ corresponding to an entry $s$ in row $r$; presumably that entry was moved down from row $s$ by $S_{rs}$. While the remark of regarding terms as zero can be interpreted as forbidding equal entries in the same column, it is unclear how the monotonicity of rows and columns is deduced (e.g., how is $S_{12}S_{24}$ forbidden for $\mu = (1, 1, 1, 1)$, while $S_{12}S_{23}S_{34}$ is allowed?). After application of a product to $\mu$, the resulting partition $\mu'$ represents a constituent $s_{\mu'}$ of $h_\mu$.

The main function of this weird encoding of tableaux as products of operators, is that it inspired Robinson to associate such products with the sequences of raising operations $e_i$ used in transforming a semistandard skew tableau $T$ into a Yamanouchi tableau $Y$, thus providing the essential ingredient $P$ needed to make the association bijective. Since the nature of the $e_i$ is quite different from the interpretation of the operations $S_{rs}$ (the $e_i$ affect the weight rather than the shape, and they definitely do not commute), it is a small miracle that such an association is possible at all. Yet it suffices to insist on always applying the $e_i$ with smallest possible $i$, to group the longest possible consecutive subsequences $e_r \circ e_{r+1} \circ \cdots \circ e_{s-1}$ together (even though they might have affected different entries), and to associate a factor $S_{rs}$ to each such sequence. Then the products of operators $S_{rs}$ that arise for tableaux $T$ of weight $\mu$ are precisely the ones valid for $\mu$ (i.e., corresponding to semistandard tableaux of partition shape and weight $\mu$), and moreover the factors appear in a fixed order, so that the product uniquely determines the entire sequence of raising operations. Robinson made no attempt to explain this stroke of good fortune, but simply accepted it as obvious without proof.

Robinson's contribution apparently succeeded in keeping the world ignorant for a long time both of a proof of the Littlewood-Richardson rule, and of its absence; his reasoning was reproduced in [Litw] by way of proof. Of the more recent history we shall say only that even when *jeu de taquin* was introduced in [Schü], it was not used to complete Robinson's argument in the way we indicated, but rather to formulate an independent proof, that is cumbersome in its own way because it abstains from using the operations $e_i$. No proof based on the commutation of raising operations and jeu de taquin slides appears to have been published to date.

**3.** Now let us consider the actual implementation of the Littlewood-Richardson rule. We shall not be interested in computing a single coefficient $c^\nu_{\lambda,\mu}$, but rather an entire polynomial $\sum_\nu c^\nu_{\lambda,\mu}X^\nu$. As we shall see, by allowing $\nu$ to vary, we can ensure relative efficiency: contributions to the result will be produced in a constant stream, since the search tree has no dead ends. The traditional way to proceed (as illustrated by examples in [LiRi], [Rob], [Litw]) would be to construct all Yamanouchi tableaux of weight $\mu$ and shape $\nu/\lambda$ (for varying $\nu$), and for each of them contribute a term $X^\nu$ to the result. However, while for humans it might be easier to place fixed symbols in varying places, computers prefer placing varying values in fixed locations. Therefore, it is preferable to construct instead semistandard tableaux of fixed shape $\mu$ that satisfy the following condition (a variation of the Yamanouchi condition): listing the entries of the tableau in the (right to left) row reading order, each initial segment has a weight that, when added to $\lambda$, yields a partition. The partition $\nu$ obtained by adding the weight of the entire tableau to $\lambda$ determines the contribution $X^\nu$ of the tableau to the result.

There are various ways to explain this variation of the traditional formulation of the Littlewood-Richardson rule. The most classical one is to refer to the original formulation of the rule in [LiRi], and to explain that we are recording rearrangements of the symbols of the tableau $B$ of shape $\mu$ into a tableau $T$ of shape $\nu/\lambda$, not by recording the symbols of $T$ according to their location, but by recording for each symbol of $B$ its whereabouts in $T$ (actually only its row number). Another, more modern way of saying the same thing, is to observe that counting pictures $\nu/\lambda \to \mu$ is equivalent to counting pictures $\mu \to \nu/\lambda$, since the inverse of a picture is also a picture. Finally, one may interpret our actions as a computation of the decomposition of the skew Schur polynomial $s_{\mu\uplus\lambda}$ (which equals $s_\lambda s_\mu$) by enumerating all Yamanouchi tableaux of shape $\mu \uplus \lambda$ according to their weight, in the traditional way, but with the slight optimisation that the top-right parts of shape $\lambda$ of these Yamanouchi tableaux are not actually constructed, since they are identical for all of them (all entries in row $i$ must be $i$), and have weight $\lambda$.

**4.** Actually, it is desirable to make one further modification to the rule before starting implementation; the reason for this arises only because we limit our search to tableaux with entries $< n$, in order to get the decomposition for Schur polynomials in $n$ variables rather than for Schur functions. To understand this reason, consider the range of values to try for some entry $t$ encountered while filling the tableau in row reading order. It is bounded below by one more than the entry above $t$, to keep its column strictly increasing (or by 0 if $t$ is in the top row), and bounded above by the entry to the right of $t$, to keep its row weakly increasing (or by $n-1$ if $t$ is at the end of a row). However, there is another upper bound, derived from the fact that it must be possible to extend the column containing $t$ downwards in a strictly increasing way with entries $< n$: if there are $k$ more squares below $t$, then its value should not exceed $n-1-k$. If the column of $t$ is not longer than the one of the right neighbour of $t$, and that neighbour already satisfies the similar bound for its column, then the additional upper bound for $t$ is implied by the original one; in general however, this is not the case, so that both tests must be performed, which complicates the inner loop. If we could arrange all columns to end in the same row, then only one upper bound would be necessary. Indeed this is possible by replacing the shape $\mu$ by $-\mu$, i.e., by rotating it a half turn. This operation is justified by the theory of pictures: jeu de taquin establishes a bijection between pictures $\mu \to \nu/\lambda$ and pictures $-\mu \to \nu/\lambda$ (or if you prefer: its commutation with raising operations ensures that jeu de taquin preserves the Yamanouchi condition, and therefore defines a weight preserving bijection from Yamanouchi tableaux of shape $\mu \uplus \lambda$ to Yamanouchi tableaux of shape $-\mu \uplus \lambda$).

In practice, it is easiest to only virtually rotate $\mu$ into $-\mu$, so that one still fills the shape $\mu$, but reverses the monotonicity conditions on rows and columns (making them weakly respectively strictly *decreasing*), as well as the order in which the tableau is filled (traversing rows from left to right, proceeding from bottom to top, while testing the accumulated weight to remain a partition). It so happens that these fillings are easier to relate to the somewhat deviant definition of pictures in [vLee] than the traditional ones are (in the terminology of that paper, they are row encodings of pictures $\mu \to \nu \setminus \lambda$), which is the real reason that they have been used in LiE from the first time the Littlewood-Richardson rule was implemented; only later was it discovered that in the traditional form one has the complication of two upper bounds.

**5.** The function *LR_tensor_irr* implements the Littlewood-Richardson rule for the multiplication of Schur polynomials $s_\lambda$ and $s_\mu$ in $n$ indeterminates, which corresponds to the tensor product decomposition of irreducible representations of $\mathbf{GL}_n$ or $\mathbf{SL}_n$. The partitions $\lambda$ and $\mu$ are passed to *LR_tensor_irr* as arrays of length $n$ (there may be trailing zeros), which length is passed as third parameter. The result will be a polynomial with exponents of size $n$; therefore it will contain all components that occur in the product of the Schur *functions* parametrised by $\lambda$ and $\mu$ if and only if $n \geq l(\lambda) + l(\mu)$. The exponents occurring in the result will be obtained by gradual modification of $\lambda$, but since $\lambda$ and $\mu$ might be aliases we should not change $\lambda$ during the computation; therefore $\lambda$ is copied into a new array $\nu$ upon entry of *LR_tensor_irr*. By calling *wt_init*$(n)$ we prepare for accumulating vectors of length $n$ into a polynomial, which will be collected by the final call to *wt_collect*.

```
#include "types.h"
  format lambda ⊙ (λ)
  format mu ⊙ (μ)
  format nu ⊙ (ν)
  poly *LR_tensor_irr (entry *λ, entry *μ, index n)
  { index i, j; entry *ν; entry **T;
    if (n = 0) return poly_one(0);
    ⟨ Allocate and initialise local data for LR_tensor_irr 6 ⟩
    wt_init(n);      /* prepare to collect terms with exponents of size n */
    ⟨ Count the Littlewood-Richardson fillings of T, extending the partition ν 8 ⟩
    ⟨ Deallocate local data for LR_tensor_irr 7 ⟩
    return wt_collect( );      /* return sum of all contributed terms */
  }
```

**6.** The semistandard tableau $T$ is constructed in a ragged two-dimensional array of shape $\mu$. We number rows, columns, and entries starting from 0; row numbers and entries are less than $n$. In fact we make $T$ and $\nu$ a bit larger than necessary: we make sure that the positions before and below every square of $\mu$ have a valid entry in $T$, placing sentinel values there that will simplify tests in the main loop, and similarly we define an entry $\nu[-1]$ with a value so large that the condition $\nu[-1] \geq \nu[0]$ places no real restriction on $\nu[0]$. The sentinel values at the bottom of each column are all $-1$, so that by column strictness all actual entries will be non-negative. The sentinel value $n-1-i$ placed before row $T[i]$ puts an upper bound to the entries in that row, ensuring that there is enough space to continue each column strictly increasing upwards with values less than $n$.

To keep indexing into $\nu$ and into each row $T[i]$ natural, we assign to them values pointing at offset 1 into the allocated arrays. Apart from space for the sentinel at $T[i][-1]$, we must also reserve space for the sentinels at the bottom of each column, which is done by computing the size of $T[i]$ based on $\mu[i-1]$ rather than on $\mu[i]$, except for $i = 0$. The row number of the sentinel at the bottom of column $j$ is the smallest $i$ with $\mu[i] \leq j$ (since the last square of $\mu$ in row $i$ is in column $\mu[i]-1$), so of we insert these sentinels by decreasing column number, we can find this row number from the one for the previous column $(j+1)$, by incrementing $i$ zero or more times, so as to make $\mu[i] \leq j$.

```
⟨ Allocate and initialise local data for LR_tensor_irr 6 ⟩ ≡
  { ν ⇐ &mkintarray(n + 1)[1]; copyrow(λ, ν, n); ν[−1] ⇐ λ[0] + μ[0];
    T ⇐ alloc_array(entry*, n + 1);
    for (i ⇐ 0; i ≤ n; ++i)      /* allocate row T[i] and place sentinel before it */
      { T[i] ⇐ &mkintarray(μ[i = 0 ? 0 : i − 1] + 1)[1]; T[i][−1] ⇐ n − 1 − i; }
    for (i ⇐ 0, j ⇐ μ[0] − 1; j ≥ 0; −−j)
    { while (i < n ∧ μ[i] > j) ++i;      /* find first i with μ[i] ≤ j */
      T[i][j] ⇐ −1;      /* place sentinel at bottom of column j */
    }
  }
```

This code is used in section 5.

**7.** In deallocating we should remember that $\nu$ and each row $T[i]$ point at offset 1 from the allocated array, and compensate when calling *freearr*; also that macro should only be called with a variable as argument.

```
⟨ Deallocate local data for LR_tensor_irr 7 ⟩ ≡
  { --ν; freearr(ν); for (i ⇐ 0; i ≤ n; i++) { entry *t ⇐ &T[i][−1]; freearr(t); } freearr(T); }
```

This code is used in section 5.

**8.** To find all legitimate fillings of $T$, we apply a straightforward backtracking technique. The variables $i$ and $j$ record the position in $T$ where we are currently trying to fill in an appropriate number. The piece of code starting at '*recurse*:' and ending with '**goto** *resume*;' represents a recursive function that has been converted to iteration. Using a real recursive function would be a lot less efficient, not just because of the time taken by the function calls, but also because all quantities local to *LR_tensor_irr*, such as $\lambda, \mu, \nu, T$, would have to be passed on unaltered during all the recursive calls, since C does not allow functions local to other ones (which could used the local variables of the outer function). The transformation to iteration is rather simple in this case, since the values that record the state of the computation can be easily returned to their original values after the "recursive call" without use of a separate stack; in fact the tableau $T$ serves as a stack to record values relevant to the outer levels of the recursion. Since there is only one place where the recursion is invoked, there is no need for a return stack either. All that is needed to resume after the recursion is to check if we are completely done, and if not, to go back to the point after the jump that represents the recursive call.

At the point where recursion is invoked, the position $(i, j)$ is that of the entry whose value has just been determined, so the first action is to increment this to the position of the next entry to be determined. This is usually achieved by incrementing $j$, but if this makes $j = \mu[i]$, we move to $T[i-1][0]$, unless already $i = 0$, in which case $T$ has been completely filled, and we can contribute $X^\nu$ to the result polynomial by calling *wt_ins*. After returning from the recursion we reverse these actions to restore the original values of $(i, j)$. Initially, we must set $(i, j)$ to a value such that incrementation will make it point the the bottom left square of $\mu$; the index of the sentinel to the left of that square satisfies this requirement. We allow $\mu$ to be the empty partition, in which case $(i, j) = (0, -1)$ will cause incrementation to move to the branch calling *wt_ins* directly. One might observe that, since the code of module 6 has just been executed, the initialisation of $(i, j)$ can be achieved by simply decrementing $i$, unless already $i = 0$; however, we believe this would not be in the spirit of structured or literate programming, and prefer to execute some slightly more elaborate code (which is executed only once for every call of *LR_tensor_irr*) that determines the correct initial values independently of any previous actions.

```
⟨ Count the Littlewood-Richardson fillings of T, extending the partition ν 8 ⟩ ≡
  { j ⇐ −1; for (i ⇐ n − 1; i > 0 ∧ μ[i] = 0; --i) { }      /* move to initial position */
  recurse:      /* recursive starting point; */
    if (++j ≥ μ[i] ∧ (j ⇐ 0, --i < 0))      /* move to next empty position, if any */
      wt_ins(ν, one, false);      /* if not, T is full; contribute ν once */
    else ⟨ In a loop find the values that may be put into T[i][j], each time adjusting ν appropriately;
           instead of a recursive call say 'goto recurse; resume:'; afterwards return values to their original
           state 9 ⟩
    if (j = 0) j ⇐ ++i < n ? μ[i] : 0;      /* return to end of row below if necessary */
    if (--j ≥ 0) goto resume;      /* do return jump unless empty row is reached */
  }
```

This code is used in section 5.

**9.** We come to the heart of our algorithm, which consists of a loop that finds all numbers $k$ that may be placed at position $T[i][j]$. Those numbers must satisfy $T[i+1][j] < k \leq T[i][j-1]$ and $\nu[k] < \nu[k-1]$ (so that $\nu$ will remain a partition after incrementing $\nu[k]$); here use is made of the sentinels stored in $T$ and $\nu$. We consider values in the interval given by the first condition in increasing order; at any point let $k'$ be one less then the next value of $k$ tried, so $k'$ is $T[i+1][j]$ initially, and equal to the last value of $k$ already found otherwise, and let $prev = \nu[k']$. Then the condition $\nu[k] < \nu[k-1]$ can be met by incrementing $k$ until $\nu[k] \neq prev$. If $prev = \nu[T[i][j-1]]$, then no $k \leq T[i][j-1]$ will meet the requirement, and we might as well stop looking immediately; otherwise, the condition $k \leq T[i][j-1]$ will be met for the smallest $k$ with $\nu[k] \neq prev$, and therefore does not need to be explicitly tested. For this reason the condition for continuing the outer **while** loop is $prev > \nu[T[i][j-1]]$.

A special argumentation shows that we can use a **do-while** loop rather than just a **while** loop, in other words that we can always find at least one valid value for $k$, which also justifies our claim about not having any dead ends in the search tree. This is just the statement of [vLee, Proposition 2.6.1], but for convenience we shall reproduce its proof, specialised to our current situation. Using the value $T[i+1][j-1]$, one deduces that the interval of values tried for $k$ is not empty, and we shall see that the first value in this interval is always a valid choice (so if desired, one could modify the code below so that the outer loop is entered by jumping into its body to the point after the initial **while** loop, provided the initialisation of $k$ is also adapted). If $T[i][j]$ is at the bottom of a column of $\mu$, then the value $k = 0$ is always valid, since we can always increase the first part of a partition. Otherwise, let $k'$ be the value of the entry $T[i+1][j]$, let that entry be the $l$-th (consecutive) one with value $k'$ in row $T[i+1]$, and let the value that $\nu$ had immediately after it was taken into account be $\nu'$. Then $\nu'[k'] \geq \nu'[k'+1] + l$, since $\nu'[k']$ was incremented $l$ times consecutively. Now by the monotonicity conditions on the rows and columns, the only places where entries $k'+1$ could have been inserted into $T$ since then are the $l-1$ positions to the left of $T[i][j]$. But even if all these entries are $k'+1$, then still $\nu[k'] > \nu[k'+1]$, and so it is admissible to increment $\nu[k'+1]$ once more; therefore $k = k'+1$ is always a valid possibility.

```
⟨ In a loop find the values that may be put into T[i][j], each time adjusting ν appropriately; instead of a
      recursive call say 'goto recurse; resume:'; afterwards return values to their original state 9 ⟩ ≡
{ index k ⇐ T[i+1][j]; entry prev ⇐ ν[k];
  do
  { while (ν[++k] = prev) { }      /* find next k with ν[k] < ν[k'] */
    ++ν[T[i][j] ⇐ k]; goto recurse;      /* insert k into T and extend partition ν; recurse */
  resume: prev ⇐ --ν[k ⇐ T[i][j]];      /* restore k and ν; set prev ⇐ ν[k] */
  } while (prev > ν[T[i][j-1]]);      /* if so, there are still corners of ν to try */
}
```

This code is used in section 8.

**10.** A function *LR_tensor* is also available to multiply linear combinations of Schur polynomials, which corresponds to computing the tensor product decomposition for reducible representations of $\mathbf{GL}_n$ or $\mathbf{SL}_n$. As usual, this is realised by linear combination of the results obtained by calling the function for the irreducible case.

```
poly *LR_tensor (poly *p, poly *q)
{ index i, j, n ⇐ p→ncols; poly *res ⇐ poly_null(n);
  for (i ⇐ 0; i < p→nrows; ++i)
    for (j ⇐ 0; j < q→nrows; ++j) res ⇐
          Addmul_pol_pol_bin(res, LR_tensor_irr(p→elm[i], q→elm[j], n), mult(p→coef[i], q→coef[j]));
  return res;
}
```

**11. Weight transformations.** As we have said before, the Littlewood-Richardson rule does not use the fundamental weight coordinates used elsewhere in LiE to represent weights, but coordinates that we shall call "partition coordinates", since dominant weights are represented by partitions. We shall now describe the conversion routines between these coordinate systems. If $\lambda$ is a weight vector of size $n$ in partition coordinates, then the corresponding weight for $\mathbf{GL}_n$ or $\mathbf{SL}_n$, as function on the torus consisting of diagonal matrices with diagonal entries $x_0, \ldots, x_{n-1}$, is simply $x^\lambda$. For $\mathbf{SL}_n$ the product $x_0 x_1 \cdots x_{n-1}$ equals 1, so weight vectors that differ only by a constant added to all components represent (in partition coordinates) the same $\mathbf{SL}_n$-weight. The $n-1$ fundamental weights for $\mathbf{SL}_n$ (which has type $A_{n-1}$) are the products $\omega_i = x_0 x_1 \cdots x_i$ for $0 \leq i < n-1$. It follows that the coefficient of $\omega_i$ in the weight $\lambda = (\lambda_0, \ldots, \lambda_{n-1})$ is $\lambda_i - \lambda_{i+1}$. Therefore the function *From_Part_v*, which implements the conversion from partition to fundamental weight coordinates for $\mathbf{SL}_n$, simply replaces a vector of length $n$ by the $n-1$ differences of its consecutive entries; thereby all vectors that represent the same $\mathbf{SL}_n$-weight are mapped to the same value.

```
vector *From_Part_v (entry *λ, index n)
{ index i; vector *result ⇐ mkvector(n − 1); entry *res ⇐ result→compon;
  for (i ⇐ 0; i < n − 1; ++i) res[i] ⇐ λ[i] − λ[i + 1];
  return result;
}
matrix *From_Part_m (entry **λ, index n_rows, index n)
{ index i, j; matrix *result ⇐ mkmatrix(n_rows, n − 1); entry **res ⇐ result→elm;
  for (i ⇐ 0; i < n_rows; ++i)
    for (j ⇐ 0; j < n − 1; ++j) res[i][j] ⇐ λ[i][j] − λ[i][j + 1];
  return result;
}
```

**12.** Polynomials are like matrices, except that the bounds are available in the object itself, and that each coefficient has to be copied, and its reference count adjusted. Since the mapping of exponents is not injective, it is necessary to reduce the polynomial computed, in order to combine terms with equal exponents.

```
poly *From_Part_p (poly *p)
{ index i, j, n_rows ⇐ p→nrows, n ⇐ p→ncols; poly *result ⇐ mkpoly(n_rows, n − 1);
  entry **λ ⇐ p→elm; entry **res ⇐ result→elm;
  for (i ⇐ 0; i < n_rows; ++i)
  { result→coef[i] ⇐ p→coef[i]; setshared(p→coef[i]);      /* copy coefficient */
    for (j ⇐ 0; j < n − 1; ++j) res[i][j] ⇐ λ[i][j] − λ[i][j + 1];
  }
  return Reduce_pol(result);
}
```

**13.** In converting from fundamental weight coordinates to partition coordinates we increase the number of entries by 1; this gives us a degree of freedom, which we fix by choosing the partition such that its last part is always zero. Then the partition coordinate $\lambda_i$ of a vector with fundamental weight coordinates $(v_0, \ldots, v_{n-1})$ is $\sum_{k=i}^{n-1} v_k$; note that this implies $\lambda_n = 0$.

```
vector *To_Part_v (entry *wt, index n)
{ index i ⇐ n; vector *result ⇐ mkvector(n + 1); entry *λ ⇐ result→compon; entry sum ⇐ 0;
  while (λ[i] ⇐ sum, --i ≥ 0) sum +⇐ wt[i];
  return result;
}
matrix *To_Part_m (entry **wt, index n_rows, index n)
{ index i; matrix *result ⇐ mkmatrix(n_rows, n + 1); entry **λ ⇐ result→elm;
  for (i ⇐ 0; i < n_rows; ++i)
    { index j ⇐ n; entry sum ⇐ 0; while (λ[i][j] ⇐ sum, --j ≥ 0) sum +⇐ wt[i][j]; }
  return result;
}
poly *To_Part_p (poly *p)
{ index i, n_rows ⇐ p→nrows, n ⇐ p→ncols; entry **wt ⇐ p→elm;
  poly *result ⇐ mkpoly(n_rows, n + 1); entry **λ ⇐ result→elm;
  for (i ⇐ 0; i < n_rows; ++i)
  { index j ⇐ n; entry sum ⇐ 0; result→coef[i] ⇐ p→coef[i]; setshared(p→coef[i]);
    while (λ[i][j] ⇐ sum, --j ≥ 0) sum +⇐ wt[i][j];
  }
  return Reduce_pol(result);
}
```

## 14. References and Index.


[FoGr] S. Fomin and C. Greene, "A Littlewood-Richardson Miscellany", *Europ. J. Combinatorics* **14**, (1993), 191–212.

[JaPe] G. D. James and M. H. Peel, "Specht series for skew representations of symmetric groups", *Journal of Algebra* **56**, (1979), 343–364.

[vLee] M. A. A. van Leeuwen, "Tableau algorithms defined naturally for pictures", *Discrete Mathematics* **157**, (1996), 321–362.

[Litw] D. E. Littlewood, *The theory of group characters, 2nd ed.*, Clarendon press, Oxford, 1950.

[LiRi] D. E. Littlewood and A. R. Richardson, "Group characters and algebra", *Phil. Trans. A* **233**, (1934), 99–141.

[Macd] I. G. Macdonald, *Symmetric Functions and Hall Polynomials*, Oxford Mathematical Monographs, Clarendon press, Oxford, 1979.

[Rob] G. de B. Robinson, "On the representations of the symmetric group", *American Journal of Math.* **60**, (1938), 745–760.

[Sche] C. Schensted, "Longest increasing and decreasing subsequences", *Canadian Journal of Math.* **13**, (1961), 179–191.

[Schü] M. P. Schützenberger, "La correspondance de Robinson", pp. 59–113 in *Combinatoire et Représentation du Groupe Symétrique* (D. Foata, ed.), Lecture Notes in Mathematics 579, 1976.

[Yng] A. Young, "Quantitative substitutional analysis, Part IV", *Proceedings of the London Mathematical Society (2)* **31**, (1930), 253–272.

[Zel] A. V. Zelevinsky, "A Generalisation of the Littlewood-Richardson Rule and the Robinson-Schensted-Knuth Correspondence", *Journal of Algebra* **69**, (1981), 82–94.