Department of Electrical and Computer Systems Engineering

Technical Report
MECSE-34-2004

"OPTMASON": A program for AUTOMATIC DERIVATION of the optical transfer functions of PHOTONIC CIRCUITS from their connection graphs

LN Binh and D Trower

MONASH UNIVERSITY

"OPTMASON": A program for AUTOMATIC DERIVATION of the optical transfer functions of PHOTONIC CIRCUITS from their connection graphs[1]

LN Binh and D. Trower

Department of Electrical and Computer Systems Engineering

Monash University, Clayton Victoria 3168 Australia

Overview

Determining the behavior of a given optical circuit has traditionally been done by solving the field (vector) or intensity (scalar) equations for each optical component simultaneously. This is time consuming and impractical for large circuits; provided some simple requirements are met, a better method using a signal-flow graph approach may be used. (see "Graphical Representation and Analysis of the Z-Shaped Double-Coupler Optical Resonator" by L.N. Binh, N.Q. Ngo, and S.F. Luk, IEEE Journal of Lightwave Technology 1993).

The requirements to be satisfied are:

1) Linearity of all optical components.

2) Time invariance of all optical components.

3) Optical components must be lumped (i.e. not distributed).

In other words, we consider only lumped LTIV optical circuits. Effects such as backscatter of light along the length of an optical fiber, or saturation of an optical amplifier are therefore not considered here (the former is a distributed phenomenon,

---

the latter nonlinear). Such circuits may be graphically represented as optical nodes connected by links. Nodes represent points in the circuit and links the function of the optical components connecting them.

For example, consider the following optical circuit constructed from two 2×2 optical couplers and some lengths of optical fiber:
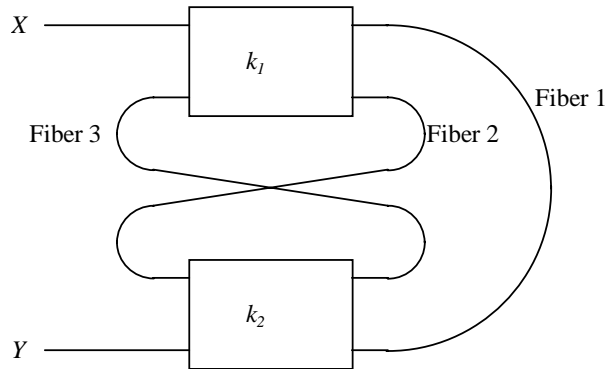
Let us assume that there is no coupling between lightwaves propagating along opposite directions in each of the fibers, and that the fibers are symmetrical w.r.t the direction of light propagation. Then each fiber may be represented by an expression of the form $t.z^{-L}$

Where t is the gain (possibly including a fixed phase factor), L is the length of the fiber, z is the z-transform parameter,

$z = exp(j\beta)$

with $\beta = n_f\omega/c$ is the propagation constant of the guided fundamental mode, and n_f is the effective refractive index of that mode (single mode propagation assumed).

The coupling constants of the various ports of the ith 2×2 optical coupler are as follows: coupling factor $C_i = (1-|k_i|)$ from a port to the port directly opposite coupling factor k_i from a port to the port diagonally opposite. k_i can be imaginary (= $j|k_i|$) if coherent light is being considered, but real if only light intensities are used. The couplers are assumed to be symmetrical.

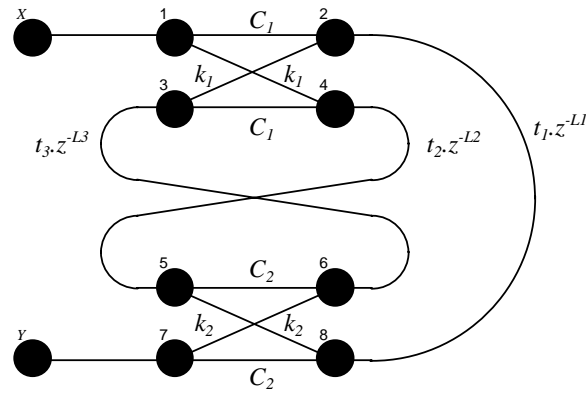The circuit may then be depicted as follows:

Figure 1 Nodes and optical transmission path of a photonic circuit.

This could be called a "photonic connection graph". The nodes are represented by dark circles, and the links by lines. Note that this is <u>NOT</u> a signal flow graph: the links depicted here are bi-directional. To create a signal flow graph from the optical connection graph, it is necessary to double each node and each link, in order to create separate nodes and links for each direction of light propagation. For ex:ample
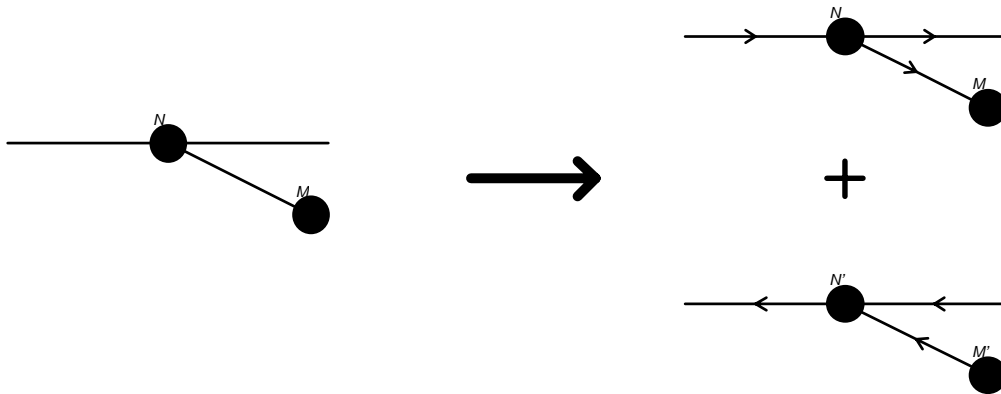
Figure 2: Node and links of bidirectional photonic paths

Once a signal flow graph has been obtained by this doubling method, Mason's rule may be applied to calculate the transfer function between a source node and a sink node (nodes that have only outputs and only inputs respectively; these nodes should not be "doubled"). We take *X* and *Y* above to be the source and sink respectively. Mason's rule depends on the following definitions:

A **forward path** is a connected sequence of directed links going from one node to another (along the link directions), encountering no node more than once.

A **loop** is a forward path that begins and ends on the same node.

The **loop gain** or **path gain** of a loop or path is the product of all the links along that loop or path (N.B. links are labeled with their values; unlabeled links have a default value of unity).

Two loops or paths are said to be **nontouching** if they share no nodes in common.

Then Mason's rule states that the transfer function T from node X to node Y is given by:

$$T = \frac{Y}{X} = \frac{1}{\Delta}\sum_k G_k \Delta_k \qquad\qquad (1)$$

where:

$\Delta = 1 -$ (sum of all individual loop gains)

+(sum of products of loop gains for all possible pairs of nontouching loops)

- (sum of products of loop gains for all possible triples of nontouching loops)

+(sum of products of loop gains for all possible quadruples of nontouching loops)

- …etc.

The summation is over all forward paths from X to Y;

G_k is the path gain of the kth forward path from X to Y.

Δ_k is the same as Δ, but calculated using ONLY those loops not touching said path.

Using OPTMASON

OPTMASON is a Borland Pascal program that accepts a text file description of an optical circuit (in "optical connection graph" form), and generates its transfer function (from source to sink as specified in the input file). Internally the optical connection

graph is converted into a signal flow graph, and then Mason's rule is applied. The resulting expression is simplified by nested grouping-of-terms.

OPTMASON is started from the DOS command line by typing:

```
optmason input_file [output_file] [-d]
```

here "input_file" and "output_file" are the corresponding filenames, and "d" is the number of decimal places to display real numbers to in the output; if omitted, "d" defaults to 3. If the output filename is omitted, output is to the screen (actually to DOS's standard output file, to enable redirection). If the input and output filenames are the same, the input file is not overwritten, but is instead appended (i.e. OPTMASON's output is added to the end of it).

The input text file format for OPTMASON is as follows:

$INPUT = *nodename*

$OUTPUT = *nodename*

nodename: *nodename, nodename,… ; nodename, nodename,...*

nodename: *nodename, nodename,… ; nodename, nodename,...*

nodename: *nodename, nodename,… ; nodename, nodename,...*

$TRANSMITTANCES

[*]*nodename, nodename = expression*

[*]*nodename, nodename = expression*

[*]*nodename, nodename = expression*

Here *nodename* is a label for a node of the optical connection graph. It may be any string of up to 15 characters, but not containing any of the characters ";:,=$*" (double quotes are OK though). e*xpression* is a mathematical expression (see below).

The first line ($INPUT=…) identifies the input (source) node.

The second line ($OUTPUT=…) identifies the output (sink) node.

The input and output nodes may be the same node.

The next section defines the geometry of the optical connection graph. A line of the form

nodename: *nodename, nodename,…* ; *nodename, nodename,…*

is required for each node in the graph except the output node. (If a line for the output node is included, it will be ignored, unless that node is also the input node.)

Each such line begins with the name of the node being defined, followed by a colon ":". The remainder of the line is a list of all the other nodes that it connects to. Since light may travel independently in two directions through a node in an optical connection graph, connections on either "side" (optically speaking) are separated by a semicolon ";". The definition for the input node and any node at the free end of an optical fiber (reflection point) will only include one "side" of this list.

Since links in an optical connection graph are bi-directional, a link from n to m in the definition of a node n must be matched by a corresponding link from m to n in the definition of node m; links to the output node are an exception. NOTE *that only a single link may join any two nodes*. For multiple optical paths between two nodes, intermediate nodes must be inserted.

The geometry definition section is terminated by the "$TRANSMITTANCES" line.

The section following this line defines the values of the links in the optical connection graph. Since all links have a default value of unity in both directions, only the values of links that differ from this need be defined. The format for specifying the value of a link between two nodes n and m (value given by "*expression*") is:

n,m = *expression*

To define the value of the link in the direction n → m only, place an asterix "*" at the start of the line. Note that if a link value is defined twice, the second definition replaces the first (both directions are treated independently).

To define the reflection coefficient at a node r, simply write:

r,r = *expression*

An asterix "*" is optional and has no effect. Reflection coefficients may ONLY be defined at nodes that are optically single-sided (e.g. cut-end of an optical fiber), and may not be defined for the input or output nodes.

Some other things to note about the input file format are:

- The input is case-sensitive for node names and variables within expressions.

- Whitespace (spaces, tabs, and blank lines) are ignored or filtered out.

- The start of the file is ignored up to the line starting with "$INPUT=" so it may be used for a description of the file contents.

- Any line beginning with a semicolon ";" is treated as a comment and ignored.

- Input lines will be truncated beyond 255 characters.

"*expression*"'s have the following form:

  magnitude {mag_expr} ^power < angle {angle_expr} z ^power {power_expr}

magnitude, angle, and *power* are real numbers (scientific notation, e.g. -1.2e+7 is permitted).

"z" may be uppercase or lowercase (the z-transform parameter).

mag_expr, angle_expr, power_expr may be any strings at all provided they do not contain "}". They are intended to be variable names or entire subexpressions.

Any part or parts of the above expression format may be omitted, provided that a meaningful expression results. The following are examples of valid expressions and their corresponding mathematical meaning:

6 $= 6$

2.96E-9{a}^2 $= 2.96 \times 10^{-9} \, a^2$

< {pi.beta} $\qquad$ = exp(j(pi.beta))

-3 < 1.2 $\qquad$ = -3exp(j1.2)

Z^-4{L} $\qquad$ = z^{-4L}

-5.2{x*y/z}^3 < 6.4{beta.pi} z^{L+n} $\qquad$ = -5.2(x*y/z)^{3}exp(j.6.4(beta.pi))z$^{(L+n)}$

Note that although parts of the expression format may be omitted, the ordering of expression components must be strictly adhered to. Also:

- The first "*^power*" is a power of "*mag_expr*" and may only be present if "*mag_expr*" is also.

- The second "*power*" and "*power_expr*" apply to the "z", so can only be present if it is also.

- A single "-" sign is not a number, and cannot precede any of the {} brackets on its own. Use -1{…} to obtain the same effect.

- Expressions in {} brackets are not simplified internally; they are treated as single variables. However when displaying the output, OPTMASON distinguishes two classes of {} expression:

  (1) A string inside {} brackets is treated as a single variable if it is composed only of letters, numbers, and the characters "~#@$%?_", and if it does not begin with a digit (0-9). It may also end with a string of single quotes " ' ". As a single variable, it will appear in summations, products, and power expressions *without brackets around it*. So for correct output, products of two variables contained within {} should include a "." or "*" symbol. e.g. if $t_2 x$ is a product, entering {t2x} may result in the output containing (say) t2x^3 which looks like $t_2 x^3$ when what is desired is $(t_2 x)^3$.

  (2) Any other string inside {} brackets will appear bracketed in the output. In particular, any string containing any of the mathematical symbols
  "[ ] ( ) + - * . / ^ < > = !" will appear bracketed (inside square brackets)

in the output except if it appears on its own (without any multiplying terms) in a summation.

Here are the contents of the input file for the example given earlier:

```
This file is an example of an input file for the sample network included

in the OPTMASON documentation.

The next two lines define the names of the input (source) and output (sink)

nodes:

$INPUT = X

$OUTPUT = Y

; The transfer function calculated by OPTMASON will be T = Y/X

; Here is a description of the network geometry:

X:      1

1:      X    ;  2,4

2:      1,3  ;  8

3:      6    ;  2,4

4:      1,3  ;  5

5:      4    ;  6,8

6:      5,7  ;  3

7:      Y    ;  6,8

8:      5,7  ;  2

; not necessary, but included for completeness:

Y:      7

$TRANSMITTANCES

; Internal optical coupler transmittances:

1,2 = {C1}

3,4 = {C1}

1,4 = {k1}

2,3 = {k1}
```

```
5,6 = {C2}

7,8 = {C2}

5,8 = {k2}

6,7 = {k2}

; We ignore the links from X and Y because they don't contribute to the

; magnitude response of the system, or affect its pole and zero positions.

; ...so they default to values of 1.

; The three main optical fiber transmittances:

2,8 = {t1} z^-1{L1}

4,5 = {t2} z^-1{L2}

3,6 = {t3} z^-1{L3}

; Thats all we need! (simple, isn't it?)
```

Here is OPTMASON's output:

```
T = numerator/denominator

numeratorá=á((k1^2+C1^2)*k2^2+C2^2*k1^2-2*C1^2*C2^2)*t1*t2*t3*z^(-L3-L2-L1)+((C1*C2*k1
^2-C1^3*C2)*k2^2-C1*C2^3*k1^2+C1^3*C2^3)*t1*t2^2*t3^2*z^(-2*L3-2*L2-L1)+C1*C2*t1*z^-L1

denominator = -2*C1*C2*t2*t3*z^(-L3-L2)+C1^2*C2^2*t2^2*t3^2*z^(-2*L3-2*L2)+1
```

<u>The OPTMASON program structure</u>

OPTMASON is written in Borland Pascal (version 7.0). It performs the following basic steps in order to produce the output:

(1) The input file lines $INPUT=… and $OUTPUT=… lines are read, and corresponding input and output (source and sink) nodes are created.

(2) The geometry description section is processed a line at a time. Each new node mentioned in this section actually causes the creation of two nodes (of the same name) in OPTMASON's data structure. Links listed before the semicolon ";" head out from the first of these, and links listed after the semicolon head out from the second. (NB internally, all links associated with a node are <u>outgoing</u>… that's why the output node doesn't need a description line). The destination of a link is

always to the <u>first</u> node with the correct name. Links <u>to</u> the input node are created to ensure correct operation when the input and output nodes are the same (see (3) below). This stage is complete at the end of the file or when a line starting with "$TRANSMITTANCES" is encountered.

(3) The link structure is checked and adjusted: It is verified that every node (except the sink) has at least one outgoing link, that for each link n → m there is a reciprocal link m → n, and that there is at most one link between any two nodes. The following adjustment is also carried out for each link n → m: If the two nodes with the name m are denoted m_1 and m_2 then the link is always n → m_1 at first. If the reciprocal link m → n is of the form m_2 → n then no action need be taken; but if it is of the form m_1 → n, then the n → m link is adjusted to be n → m_2. This eliminates all link loops of the form A → B → A, and results in the correct signal flow graph corresponding to the optical connection graph. This procedure also correctly handles the case where the input and output nodes are the same. (In step (1) it is ensured that the output node precedes the input node in the list of nodes, and so is treated as the first node of that name if the two are the same). If they are not the same, some "phantom" links <u>to</u> the input node will remain, and should be ignored.

(4) The link value description section is parsed one line at a time, and the link values are set (all are already initialized to 1 at link creation). When a link from a node to itself (i.e. a reflection coefficient) is encountered, it is first checked that one of the two corresponding nodes with that name has NO outputs, then a new link with the desired value is created from that node to its pair node. A check is also done to see whether this link already exists, and if so the value is overwritten rather than a new link created.

(5) A complete depth-first search through the signal flow graph is performed recursively, starting at the input node. When the output node is reached, a new path has been found. An entry in the list of paths is created for it and the path gain is computed. Associated with each node is a list of the paths and the loops that it touches (the "touchpath" and "touchloop" lists respectively). The new path is added to the "touchpath" list of each node along it. Associated with each path is

the list of loops that it touches, and to the new path's "touchloop" list is added the contents of each "touchloop" list on each node along the path, checking to ensure that duplicate entries are not made. (NB a path touches any loop that any one of its nodes touches). Similarly, when the search encounters a node that it has been to before a loop has been found. (The last-taken link out of each node is stored during the search, so if this is not nil then the node has been previously traversed. NB this information is also used to follow loops and paths when computing loop/path gain, without disrupting the recursive nature of the search procedure). Associated with each loop is a "touchloop" and a circularly-linked list of nodes in the loop. When a loop has been found, this node list for each loop in the "touchloop" list of the current node is checked, to see if that loop has actually been found before. If not, a new entry for it in the list of loops is created, and placed at the start of this list. The loop gain of the new loop is computed (actually -(loop gain), since this is what Mason's rule uses). Also the following is done: (i) A circular node-list for the loop is created (ii) the loop is added to the "touchloop" list of each node along it (iii) the loop is added to the "touchloop" list of each path in the "touchpath" lists of all the nodes along it (iv) the "touchloop" lists of each of the nodes along the new loop are amalgamated (removing duplicate entries) and stored as the "touchloop" list for this loop. Naturally when performing the computations for a new loop or path, they are all done together on a node-by-node basis rather then being completed sequentially as the above descriptions imply. It would be possible to speed up the operation of this stage of OPTMASON by storing the "touchloop" lists in sorted form (or even in binary tree form, etc) to make searching for duplicate entries and list amalgamation more efficient. The lists actually contain pointers, but these could be converted to "longints" for use as sorting keys.

(6) When the search is complete, the result is a list of all paths and an ordered list of all loops in the signal flow graph, with their associated pathgains and (-loopgains). Also associated with each path is a list of the loops it touches, and associated with each loop is a list of the paths it touches, and a list of the loops below it in the loop-list that it touches. (That is why the loop list order is important; the loops are in no particular order per se). Since this is all the information Mason's rule requires, all the data associated with each node and link is now deleted to free

memory space for further calculation. The node lists for each loop are also deleted.

(7) The Mason's rule denominator Δ is calculated. To explain how this is done, it must first be observed that the rule for calculation of Δ given earlier can be expressed as: Δ = 1+sum of all possible products of (-loopgain) of all nontouching loops   The list of loops can be regarded as a tree structure: the root node is not a loop, but its branch nodes are each of the loops, and every other node in tree is a loop. Further down the tree, a node has as its branch nodes all loops below it in the list (of loops) that do not touch it or any node above it in the tree. Each possible descent through the tree terminating on an arbitrary node (i.e. not necessarily a leaf node) then represents a unique combination of nontouching loops, and the set descents to all possible nodes in the tree represents the set of all possible combinations of nontouching loops. Δ is calculated, therefore, by a recursive depth-first tree search, which at each node adds to delta the running product of that node's (-loopgain) times the (-loopgains) product of the nodes earlier in the tree, before proceeding to search the branches. Associated with each loop is the number of times it has been "deactivated" or touched by loops along the current descent path. The search works by simply searching all loops below the current tree-node in the loop-list that have not been deactivated (touched) yet.

(8) The Mason's rule numerator is calculated. This involves stepping through each path in turn and adding to the numerator the pathgain times Δ_k for that (the kth) path. Δ_k is calculated the same way Δ is, but the loops touching the current (k^{th}) path are first given a "deactivation level" of 1 so they will not be included.

(9) The results are outputted (in symbolic form) either to the screen (DOS standard redirectable output) or to the specified output file (appending it if it is the same as the input file). All dynamic variables are then disposed of before the program completes execution.

The above functions account for approximately 2/3 of the OPTMASON listing. The other 1/3 of the program is devoted to manipulating the symbolic expressions that result from calculations on link values and on path and loop gains.

OPTMASON expressions are stored in a hierarchical tree-like structure, and all operations on them are performed recursively. The structure corresponding to an expression is built up out of records of type "expression_type". A "format" field in each of these records identifies it as belonging to one of four possible types:

- A "zterm" - a term of the form $z^{num*\{expr\}}$. If expr is absent (nil pointer) then it just represents z^{num}.

- An "anglterm" - a term of the form exp(j.num.{expr}). If expr is absent, just exp(j.num).

- A "magterm" - a term of the form $\{expr\}^{num}$.

- A "magval" - a real number, = num.

In all the above cases "num" is a double-precision real number, and "expr" is a string representing a variable or a mathematical expression; it is what appears inside {} brackets in the input file. No processing is performed on the *contents* of expr. (Actually expr^, since expr is a pointer to the string).

Apart from the "format", "num", and "expr" fields, expression_type also has "sumterm" and "prodterm" field, which point to other expression records, or are nil. They define the structure of a mathematical expression. This is best illustrated graphically: horizontal arrows represent the sumterm pointers, vertical arrows the prodterm pointers, and the absence of an arrow indicates that that pointer is nil. Each letter below represents a single term, i.e. single expression_type record.

```
a ⟶ b ⟶ c

↓              ↓

d → e → f      g → h

↓   ↓              ↓

i   j→k           l→m→n

↓                      ↓
```

o   p
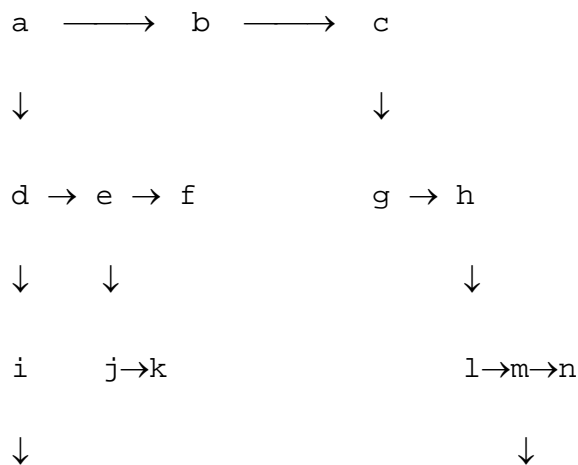
This entire structure corresponds to the expression:

$$(o.i.d+(j+k)e+f)a + b + (g+(l+p.m+n)h)c$$

A boolean function precedes(x,y) is used to determine if the term x "precedes" the term y. Order of precedence is as follows: zterm < anglterm < magterm < magval. If x and y are of the same format, then ascii comparison of $\{expr\}_x$ and $\{expr\}_y$ is used to decide whether x precedes y, and if these are equal, num_x is compared to num_y. The precedes() function always returns "true" if x=y, or if they are both "magval"s (so it should really be called "equal_or_precedes").

The expression tree for an expression is ordered so that if a term x contains a link to a term y (either sumterm or prodterm link), then precedes(x,y) will be true. This has the effect of bringing "zterm"s outside of brackets, and summing "magval"s and "magterm"s together inside brackets - which hopefully results in the most compact and legible output.

"Magval"s are thus the leaves of the expression tree. A nil pointer in an expression or to an expression is treated as a zero. Thus, if a prodterm pointer is nil, the terms it multiplies get canceled. The exception is on "magval" terms, which must therefore terminate every path down through the tree; a nil prodterm pointer is required on "magval" type terms.

Addition of one expression to another is performed as follows: Each term in the top-level sum list (i.e. root, root^.sumterm, root^.sumterm^.sumterm… etc) of the expression being added is compared with each successive term of the top-level sum list of the expression being added to, until the term being added is found to precede() the term it is compared to. If they are addable - which occurs if they are equal or both magvals - then if they are magvals the value of the term being added is just added; otherwise, the addition procedure is called recursively to add their prodterm subexpressions (since ax+bx = (a+b)x). If they are not addable, then the term being added is simply inserted into the top-level sum list of the expression being added to, along with the subexpression pointed to by its prodterm… i.e. along with anything multiplying it.

Multiplication is performed in a similar manner: to create a new expression that is the product of two others, the product expression is set to zero, then every term in the top-level sum list of the first is compared with every term in the top-level sum list of the other. If a pair of such terms can be multiplied directly (i.e. if they are both "magval"s or if they are of the same format and have the same {expr}) then this is done, and both their prodterm subexpressions are multiplied (calling the multiply procedure recursively). Otherwise the prodterm subexpression of the "precede()"ing term is multiplied (recursively) by the other term and its product subexpression. In either case, the result is added to the final product (using the addition procedure described above).

Due to the regular sorted (by the precedes() function) way in which expressions are stored and manipulated, they are automatically simplified by grouping of terms at every stage in the calculation. This is not only efficient, but almost certainly necessary for handling the huge sum-of-product type expressions that might result from calculation of the Mason's rule Δs for large graphs if simplification were not used, without running out of memory.

Finally, the procedure used to output expressions works in a similar manner as those used to manipulate them: Each top-level sum term is handled in turn, first outputting the prodterm subexpression (by calling the output routine recursively), then outputting the term it multiplies, and a "+" if more sumterms remain. At least, that's how it works in principle: In practice, there are exceptions and modifications. In particular, when "+" needs to be output, it is not done right away, but a plus_waiting flag is set instead. Then, when the next item is outputted, the waiting "+" is outputted first, unless then next item to be outputted begins with a minus sign. This eliminates output of the form x+-6y. Similarly, checks are made for multiplication by +/-1 to avoid outputting the numeral. "prodterm" subexpressions involving sums are bracketed, and those that do not are not. An {expr} part of a term that is treated as a single variable (see above) will not have brackets around it, otherwise it will be enclosed in [] brackets, except when it appears in a sum nothing except +1 multiplying it. "prodterm" subexpressions are in general outputted recursively, but where products of multiple single zterms or single anglterms occur, the entire product is displayed within a single $z^{\wedge}(\ldots+\ldots)$ or $\exp(j(\ldots+\ldots))$ to make the output easier to

read; recursion is not used in this case, and the product list is stepped through manually until an incompatible term is found (i.e. that cannot be put within the same brackets).

OPTMASON.PAS program listing

```
{ OPTMASON.PAS

{ Written in 1996 by Dean Trower

{ for ECS4313 Photonic Signal Processing

{

{ This program uses Mason's rule for signal flow graphs to generate

{ the optical transfer function (in symbolic form) of an arbitrary

{ lumped, linear time-invariant optical circuit, as specified in the

{ input file (file name is a command line parameter).

{

{ See the file OPTMASON.DOC for more information.

}


type term_type = (zterm, anglterm, magterm, magval);

     link_ptr = ^link_type;

     node_ptr = ^node_type;

     path_ptr = ^path_type;

     loop_ptr = ^loop_type;

     looplist_ptr = ^looplist_type;

     pathlist_ptr = ^pathlist_type;

     nodelist_ptr = ^nodelist_type;

     expression_ptr = ^expression_type;

     link_type = record     { record for a signal flow graph link }
```

```
                    dest:node_ptr;        { destination node }

                    val:expression_ptr;        { value }

                    next:link_ptr;

                end;

      expression_type = record    { record for a single term in an expression
}

                          format:term_type;

                          sumterm:expression_ptr;

                          prodterm:expression_ptr;

                          num:double;

                          expr:^string;

                        end;

      looplist_type = record  { record for a single entry in a list of loops
}

                          loop:loop_ptr;

                          next:looplist_ptr;

                        end;

      pathlist_type = record  { record for a single entry in a list of paths
}

                          path:path_ptr;

                          next:pathlist_ptr;

                        end;

      nodelist_type = record  { record for a single entry in a list of nodes
}

                          node:node_ptr;

                          next:nodelist_ptr;

                        end;

      node_type = record       { record for a node of the s.f. graph }
```

```
                           name:string[15];

                           firstlink:link_ptr;  { linked list of OUTPUT links }

                           next:node_ptr;        { i.e. next node in s.f. graph }

                           link_taken:link_ptr; { used during search for
loops/paths}

                           touchloop:looplist_ptr; { list of loops containing node }

                           touchpath:pathlist_ptr; { list of paths containing node }

                       end;

        path_type = record

                        pathgain:expression_ptr;

                        touchloop:looplist_ptr; { list of loops touching path }

                        next:path_ptr;

                    end;

        loop_type = record

                        loopgain:expression_ptr;

                        touchloop:looplist_ptr; { !!!loops later in list only }

                        nodelist:nodelist_ptr; { circularly linked list of nodes
}

                        deactivated:word; { used for calc. of Mason rule deltas }

                        next:loop_ptr;

                    end;
```

```pascal
var firstnode,sourcenode,sinknode:node_ptr;

    firstpath:path_ptr;

    firstloop:loop_ptr;

    numerator,denominator:expression_ptr;

    f:text;

    outfile_name:string;

    precparam:string; { stores the -d command line parameter }

    line:longint;     { counts lines in the input file }

    prec:byte;        { no. of digits to display reals to in the output }

    errpos:integer;   { used while parsing input and cmnd line params }

    heap_state:pointer; { so it can be restored when program ends }


                        { the following variables aren't really global,   }
    path:path_ptr;      { but they are here to prevent them being created }

    loop:loop_ptr;      { many times during recursion.                    }

    node:node_ptr;

    pathentry:pathlist_ptr;

    loopentry,loopentry2:looplist_ptr;

    nodeentry:nodelist_ptr;

    found_duplicate,plus_waiting,addbracket:boolean;

    tmp_exp,mult_result:expression_ptr;




function singleterm(s:string):boolean;
```

```pascal
{ determines if an "expr" string should be treated as a single variable }

var i:integer;

    b:boolean;

begin

  b:=true;

  i:=1;

  while b and (i<=length(s)) do

    begin

      b:=(pos(upcase(s[i]),'~#@$%?_ABCDEFGHIJKLMNOPQRSTUVWXYZ')>0) or

          (pos(copy(s,i,length(s)-i+1),'''''''''''''''''''''''')>0) or

          ((i>1) and (pos(s[i],'0123456789')>0));

      inc(i);

    end;

  singleterm:=b;

end;




procedure writeexpression(var f:text; e:expression_ptr);

{ recursively procedure to display an expression (or output it to file f)}

var t,q:expression_ptr;

begin

  if e=nil then write(f,'0');

  t:=e;

  while t<>nil do   { step through each root-level sumterm }

    begin      { skip over sums of angles inside single exp()       }
```

```
q:=t;     { and sums inside brackets in z^(***) ... done later }

if (t^.format=anglterm) or (t^.format=zterm) then

   while (q^.prodterm<>nil) and (q^.prodterm^.sumterm=nil)

         and (q^.prodterm^.format=t^.format) do q:=q^.prodterm;

if q^.prodterm<>nil then

   if q^.prodterm^.sumterm<>nil then

     begin

       if plus_waiting then write(f,'+');

       plus_waiting:=false;

       write(f,'(');

       writeexpression(f,q^.prodterm);

       write(f,')*');

     end

   else

     begin

       if not((q^.prodterm^.format=magval) and  { don't write "1*" }

              (abs(q^.prodterm^.num)=1)) then    { or "-1*"        }

         begin

           writeexpression(f,q^.prodterm);

           write(f,'*');

         end

       else if q^.prodterm^.num=-1 then write(f,'-')

       else if plus_waiting then write(f,'+');

       plus_waiting:=false;

     end;

if not((t^.format=magval) and (t^.num<0)) and plus_waiting

   then write(f,'+');  { don't write +-number for negative nums }
```

```pascal
        plus_waiting:=false;

      case t^.format of

        magval:  if t^.num=round(t^.num) then write(f,round(t^.num))

                                      else write(f,t^.num:0:prec);

        magterm: if t^.num<>1 then

                    begin       { ie must display the ^...}

                      if singleterm(t^.expr^)

                        then write(f,t^.expr^,'^')

                        else write(f,'[',t^.expr^,']^');

                      if t^.num=round(t^.num) then write(f,round(t^.num))

                                          else write(f,t^.num:0:prec);

                    end

                  else if singleterm(t^.expr^) or

                          (not((t=e) and (t^.sumterm=nil)) and

                          ((t^.prodterm=nil) or

                           ((t^.prodterm^.sumterm=nil) and

                            (t^.prodterm^.format=magval) and

                            (t^.prodterm^.num=1))))

                       then write(f,t^.expr^)

                       else write(f,'[',t^.expr^,']');

        anglterm: begin

                    write(f,'exp(j*');

                    if ((t^.prodterm=nil) or
(t^.prodterm^.format<>anglterm))

                        and

                        ((t^.expr=nil) or singleterm(t^.expr^) or
(t^.num<>1))
```

```
        then addbracket:=false

     else

        begin

           addbracket:=true;

           write(f,'(');

        end;

  q:=t;     { display an entire list of single-anglterm }

  repeat    { products inside the brackets              }

     if (q<>t) and (q^.num>=0) then write(f,'+');

     if (q^.num<>1) or (q^.expr=nil) then

        begin

           if q^.num=-1 then write(f,'-')

           else if q^.num=round(q^.num)

                 then write(f,round(q^.num))

                 else write(f,q^.num:0:prec);

           if (q^.expr<>nil) and (q^.num<>-1)

             then write(f,'*');

        end;

     if q^.expr<>nil then

        if singleterm(q^.expr^) or (q^.num=1)

           then write(f,q^.expr^)

           else write(f,'[',q^.expr^,']');

     q:=q^.prodterm;

  until (q=nil) or (q^.sumterm<>nil) or

        (q^.format<>anglterm);

  if addbracket then write(f,'))')

                else write(f,')');
```

```
                end;

zterm: begin

            write(f,'z');

            if ((t^.prodterm=nil) or (t^.prodterm^.format<>zterm)) and

             ((t^.expr=nil) or (abs(t^.num)=1))

               then addbracket:=false else addbracket:=true;

            if addbracket or (t^.expr<>nil) or (t^.num<>1) then

              begin

                if addbracket then write(f,'^(') else write(f,'^');

                  q:=t;      { display an entire list of single-zterm }

                  repeat     { products using the same z^(...)         }

                    if (q<>t) and (q^.num>=0) then write(f,'+');

                    if (q^.num<>1) or (q^.expr=nil) then

                      begin

                        if q^.num=-1 then write(f,'-')

                        else if q^.num=round(q^.num)

                               then write(f,round(q^.num))

                               else write(f,q^.num:0:prec);

                        if (q^.expr<>nil) and (q^.num<>-1)

                          then write(f,'*');

                      end;

                    if q^.expr<>nil then

                      if singleterm(q^.expr^) or (q^.num=1)

                        then write(f,q^.expr^)

                        else write(f,'[',q^.expr^,']');

                    q:=q^.prodterm;

                  until (q=nil) or (q^.sumterm<>nil) or
```

```pascal
                               (q^.format<>zterm);

                    if addbracket then write(f,')');

                 end;

             end;

       end;

       t:=t^.sumterm;

       if t<>nil then plus_waiting:=true; { next term to be added }

    end;

 end;


procedure dispose_expression(var e:expression_ptr);

{ deletes all the memory allocated to an expression and sets it to nil (=0)}

begin

  if e<>nil then

    begin

      dispose_expression(e^.prodterm);

      dispose_expression(e^.sumterm);

      if (e^.format<>magval) and (e^.expr<>nil)

        then freemem(e^.expr,length(e^.expr^)+1);

      dispose(e);

      e:=nil;

    end;

end;
```

```pascal
procedure copy_expression(var exp1:expression_ptr; exp2:expression_ptr);

{ recursively copys the contents of exp2 to exp1; new exp1 pointer created }

begin

  if exp2=nil then exp1:=nil else

    begin

      new(exp1);

      exp1^:=exp2^;

      if (exp2^.format<>magval) and (exp2^.expr<>nil) then

        begin

          getmem(exp1^.expr,length(exp2^.expr^)+1);

          exp1^.expr^:=exp2^.expr^;

        end

      else exp1^.expr:=nil;

      copy_expression(exp1^.prodterm,exp2^.prodterm);

      copy_expression(exp1^.sumterm,exp2^.sumterm);

    end;

end;




procedure new_scalar(var scalar_expr:expression_ptr; val:double);

{ creates a new expression pointed to by scalar_expr, with value val }

var e:expression_ptr;

begin

  if val=0 then scalar_expr:=nil else

    begin
```

```pascal
      new(e);

      scalar_expr:=e;

      e^.format:=magval;

      e^.num:=val;

      e^.expr:=nil;

      e^.sumterm:=nil;

      e^.prodterm:=nil;

    end;

  end;


procedure new_link_val(var link_expr:expression_ptr;

                       mag:double; mag_str:string; mag_pwr:double;

                       angl:double; angl_str:string;

                       zcoeff:double; z_str:string);
{ creates a new expression pointed to by link_expr, of the form         }
{ mag*[mag_str]^mag_pwr * exp(j*(angl*[angl_str])) * z^(zcoeff*[z_str])  }
{                                                                       }
{ NB this procedure creates the correct expression tree for precedes()  }
{ as it is now; but if precedes() is altered w.r.t. the precedence of   }
{ the different term formats, this procedure MUST be modified also.     }
var e:expression_ptr;
begin

  if mag=0 then link_expr:=nil else  { anything*0 = 0 }

    begin

      new(e);

      link_expr:=e;

      if zcoeff<>0 then              { include a zterm if present }
```

```
          begin

            e^.format:=zterm;

            e^.num:=zcoeff;

            if z_str='' then e^.expr:=nil else

              begin

                getmem(e^.expr,length(z_str)+1);

                e^.expr^:=z_str;

              end;

            e^.sumterm:=nil;

            new(e^.prodterm);

            e:=e^.prodterm;

          end;

        if angl<>0 then                { include an anglterm if present }

          begin

            e^.format:=anglterm;

            e^.num:=angl;

            if angl_str='' then e^.expr:=nil else

              begin

                getmem(e^.expr,length(angl_str)+1);

                e^.expr^:=angl_str;

              end;

            e^.sumterm:=nil;

            new(e^.prodterm);

            e:=e^.prodterm;

          end;

        if (mag_str<>'') and (mag_pwr<>0) then { include a magterm if present
}
```

```pascal
    begin

      e^.format:=magterm;

      getmem(e^.expr,length(mag_str)+1);

      e^.expr^:=mag_str;

      e^.num:=mag_pwr;

      e^.sumterm:=nil;

      new(e^.prodterm);

      e:=e^.prodterm;

    end;

  e^.format:=magval;           { the magval term is compulsory for }

  e^.num:=mag;                 { nonzero expressions.              }

  e^.expr:=nil;

  e^.prodterm:=nil;

  e^.sumterm:=nil;

  end;

end;
```

```
function precedes(x,y:expression_ptr):boolean;

{ establishes an order-of-precedence among expression components }

begin

  if y=nil then precedes:=true else if x=nil then precedes:=false

  else if (x^.format<>y^.format) then precedes:=(x^.format<y^.format)

  else if x^.format=magval then precedes:=true

  else if (x^.expr=nil) xor (y^.expr=nil) then precedes:=(y^.expr=nil)

  else if (x^.expr=nil) or (x^.expr^=y^.expr^)

         then precedes:=(x^.num>=y^.num)

         else precedes:=(x^.expr^>y^.expr^);

end;

function can_add(x,y:expression_ptr):boolean;

{ true if the expression components pointed to by x,y are directly addable }

{ they are addable if they are equal or if they are both magvals.          }

begin

  if ((x=nil) xor (y=nil)) or (x^.format<>y^.format) then can_add:=false

  else if (x=nil) or (x^.format=magval) then can_add:=true

  else if (x^.num<>y^.num) or

          ((x^.expr=nil) xor (y^.expr=nil)) or

          ((x^.expr<>nil) and (x^.expr^<>y^.expr^))

         then can_add:=false else can_add:=true;

end;
```

```
procedure add(var e1:expression_ptr; e2:expression_ptr);

{ computes e1 <- e1 + e2                              }

{ NB must not have e1,e2 pointing to same expression }

var temp,last:expression_ptr;

begin

  while e2<>nil do  { e2 will step through successive top-level sumterms }

    if e1=nil then  { if e1=0, just set e1 := e2 }

      begin

        copy_expression(e1,e2);

        e2:=nil;

      end

    else

      begin

        temp:=e1;  { step through top-level sumterms of e1 }

        while precedes(temp,e2) and not(can_add(e2,temp)) do

          begin

            last:=temp;

            temp:=temp^.sumterm;

          end;

        if not(precedes(temp,e2)) then { couldn't find one to add to, }

          begin                         { so must insert the e2 term   }

            tmp_exp:=e2^.sumterm; { TEMPORARILY disconnect the remaining }

            e2^.sumterm:=nil;    { sumterms so that only e2 and its     }

            if temp=e1 then      { prodterms get inserted.              }

              begin                      { insert at start of list }

                copy_expression(e1,e2);
```

```
                    e1^.sumterm:=temp;

              end

          else

            begin                        { insert not at start of list }

              copy_expression(last^.sumterm,e2);

              last^.sumterm^.sumterm:=temp;

            end;

          e2^.sumterm:=tmp_exp; { restore remaining sumterms }

      end

    else if e2^.prodterm<>nil then  { can add directly by adding }

          begin                        { prodterm subexpressions    }

              add(temp^.prodterm,e2^.prodterm);

              if (temp^.prodterm=nil) then  { check for a zero after the
}

                begin                        { addition and if there, get
}

                    if temp=e1 then e1:=e1^.sumterm          { rid of it
}

                                  else last^.sumterm:=temp^.sumterm;

                    dispose_expression(temp);

                end;

          end

      else if temp^.format=magval then  { can add directly by adding nums
}

          begin

            temp^.num:=temp^.num + e2^.num;

            if temp^.num=0 then   { check for a zero after addition and }

              begin                { if there, get rid of it.           }
```

```
                if temp=e1 then e1:=e1^.sumterm

                          else last^.sumterm:=temp^.sumterm;

                dispose_expression(temp);

            end;

        end;

    e2:=e2^.sumterm;  { do next term }

  end;

end;
```

```pascal
procedure copy_term(var x:expression_ptr; y:expression_ptr);

{ copies a single expression term x <- y, excluding sumterm/prodterm links }

begin

  new(x);

  x^:=y^;

  x^.sumterm:=nil;

  x^.prodterm:=nil;

  if (y^.format<>magval) and (y^.expr<>nil) then

    begin

      getmem(x^.expr,length(y^.expr^)+1);

      x^.expr^:=y^.expr^;

    end

  else x^.expr:=nil;

end;




function can_multiply(x,y:expression_ptr):boolean;

{ true if the components x,y can multiply to a single component }

begin

  if (x^.format<>y^.format) then can_multiply:=false else

  if x^.format=magval then can_multiply:=true

  else if ((x^.expr=nil) and (y^.expr=nil)) or

          ((x^.expr<>nil) and (y^.expr<>nil) and (x^.expr^=y^.expr^))

        then can_multiply:=true else can_multiply:=false;

end;
```

```
procedure multiply(var product:expression_ptr; e1,e2:expression_ptr);

{ computes product <- e1 * e2 }

var partprod,t1,t2,tmp_ptr:expression_ptr;

begin

  product:=nil;    { start by setting product = 0 }

  t1:=e1;

  while t1<>nil do  { t1 steps through top-level sumterms of e1 }

    begin

      t2:=e2;

      while t2<>nil do  { t2 steps through top-level sumterms of e2 }

        begin

          if can_multiply(t1,t2) then   { multiply t1,t2 directly, then }

            begin                              { multiply their prodterms      }

              copy_term(partprod,t1);

              if t1^.format=magval then partprod^.num:=t1^.num*t2^.num

                                 else partprod^.num:=t1^.num+t2^.num;

              if partprod^.num=0 then   { get rid of zero terms }

                begin

                  dispose_expression(partprod);

                  if t1^.format=magval

                    then partprod:=nil

                    else multiply(partprod,t1^.prodterm,t2^.prodterm);

                end
```

```
            else multiply(partprod^.prodterm,t1^.prodterm,t2^.prodterm);

         end

       else if precedes(t1,t2) then   { t1 precedes t2, so              }

          begin                        { partprod = t1.(t1_prodterm*t2) }

            copy_term(partprod,t1);

            tmp_ptr:=t2^.sumterm;        { temporarily disconnect }

            t2^.sumterm:=nil;           { remaining sumterms      }

            multiply(partprod^.prodterm,t1^.prodterm,t2);

            t2^.sumterm:=tmp_ptr;

          end

        else

          begin       { same as above, but for t2 preceding t1 }

            copy_term(partprod,t2);

            tmp_ptr:=t1^.sumterm;

            t1^.sumterm:=nil;

            multiply(partprod^.prodterm,t1,t2^.prodterm);

            t1^.sumterm:=tmp_ptr;

          end;

        add(product,partprod);          { add the product of the two }

        dispose_expression(partprod);   { terms to the final product }

        t2:=t2^.sumterm;      { get next sumterm from e2 }

      end;

    t1:=t1^.sumterm;          { get next sumterm from e1 }

  end;

end;
```

```pascal
procedure killwhitespace(var s:string);

{ removes spaces and tabs from each line of the input file. }

{ also removes lines starting with ';' - i.e. comments.      }

var i:byte;

begin

  i:=0;

  while i<length(s) do

    if (s[i+1]=' ') or (ord(s[i+1])=9)

      then delete(s,i+1,1)

      else inc(i);

  if (length(s)>0) and (s[1]=';') then s:='';

end;
```

```pascal
procedure error(i:byte; s:string);

{ displays an error message on the screen, cleans up, and exits the program
}

begin

  case i of

    0: writeln('ERROR OPENING FILE',s);

    1: writeln('INVALID INPUT FILE FORMAT: NO "$INPUT=" LINE FOUND');

    2: begin

         writeln('ERROR LINE ',line,': MAXIMUM LENGTH OF NODE NAME IS 15
CHARACTERS:');

         writeln('"',s,'"  IS TOO LONG.');

       end;

    3: writeln('ERROR LINE ',line,': INPUT NODE NAME NOT SPECIFIED');

    4: writeln('INVALID INPUT FILE FORMAT: "$OUTPUT=" LINE MUST FOLLOW
"$INPUT=" LINE');

    5: writeln('ERROR LINE ',line,': OUTPUT NODE NAME NOT SPECIFIED');

    6: begin

         writeln('SYNTAX ERROR IN LINE ',line,' (LINE DISPLAYED UP TO
ERROR):');

         writeln(s);

       end;

    7: writeln('ERROR LINE ',line,': NODE CANNOT CONNECT TO ITSELF
(',s,')');

    8: writeln('INVALID FILE FORMAT: "$TRANSMITTANCES" LINE MUST FOLLOW
LINKS LIST');

    9: writeln('ERROR: NO LINKS DEFINED FOR NODE ',s);

    10: writeln('ERROR: RECIPROCAL LINK FOR LINK ',s,' MISSING');

    11: writeln('ERROR: MULTIPLE LINKS ',s,' NOT PERMITTED');

    12: writeln('ERROR LINE ',line,': NODE NOT FOUND (',s,')');
```

```
    13: writeln('ERROR LINE ',line,': LINK NOT FOUND (',s,')');

    14: writeln('ERROR LINE ',line,': INPUT NODE LINK DESCRIPTION MAY NOT
CONTAIN ";"');

  end;

  close(f);                 { close input/output file if open }

  release(heap_state);    { release all dynamic variables }

  halt;                     { ABORT PROGRAM!!! }

end;
```

```
procedure findnode(var node:node_ptr; nm:string);

{ returns a pointer to the FIRST node in the list with name 'nm' }

{ if no node is found, TWO new nodes are created with this name, }

{ and are added to the START of the node list.                   }

var i:byte;

begin

  if length(nm)>15 then error(2,nm);

  node:=firstnode;

  while (node<>nil) and (node^.name<>nm) do node:=node^.next;

  if node=nil then

    for i:=1 to 2 do

      begin

        new(node);

        node^.next:=firstnode;

        firstnode:=node;

        node^.name:=nm;

        node^.firstlink:=nil;

        node^.link_taken:=nil;

        node^.touchloop:=nil;

        node^.touchpath:=nil;

      end;

end;
```

```
procedure getsourcesink;

{ parses the first 2 meaningful lines of the input file }

var s:string;

    i,l:byte;

begin

  line:=0;

  repeat                              { find the $INPUT line }

    readln(f,s);

    inc(line);

    killwhitespace(s);

  until eof(f) or (copy(s,1,7)='$INPUT=');

  if eof(f) then error(1,'');

  i:=8;                              { get the input node name }

  l:=0;

  while (i+l<=length(s)) and (pos(s[i+l],';:,=$*')=0) do inc(l);

  if i+l<=length(s) then error(6,copy(s,1,i+l))

  else if l=0 then error(3,'')

  else if l>15 then error(2,copy(s,i,l));

  new(firstnode);                    { create a node for it }

  firstnode^.next:=nil;

  firstnode^.name:=copy(s,i,l);

  firstnode^.firstlink:=nil;

  firstnode^.link_taken:=nil;

  firstnode^.touchloop:=nil;

  firstnode^.touchpath:=nil;

  sourcenode:=firstnode;
```

```
   repeat                          { find the $OUTPUT line }

     readln(f,s);

     inc(line);

     killwhitespace(s);

   until eof(f) or (s<>'');

   if copy(s,1,8)<>'$OUTPUT=' then error(4,'');

   i:=9;                           { get the output node name }

   l:=0;

   while (i+l<=length(s)) and (pos(s[i+l],';:,=$*')=0) do inc(l);

   if i+l<=length(s) then error(6,copy(s,1,i+l))

   else if l=0 then error(5,'')

   else if l>15 then error(2,copy(s,i,l));

   new(firstnode^.next);           { create a node for it, at the START    }

   node:=firstnode^.next;          { of the node list (which then contains }

   node^.next:=nil;                { exactly two nodes)                    }

   node^.name:=copy(s,i,l);

   node^.firstlink:=nil;           { NB names of sourcenode and sinknode }

   node^.link_taken:=nil;          { may be the same.                    }

   node^.touchloop:=nil;

   node^.touchpath:=nil;

   sinknode:=node;

end;
```

```
procedure getlinks;

{ parses the geometry description section of the input file }

var outputnode:node_ptr;

    link:link_ptr;

    s,nm:string;

    i,l,j:byte;

begin

  repeat                { get a line }

    readln(f,s);

    inc(line);

    killwhitespace(s);

    if (s<>'') and (s[1]<>'$') then { skip blank lines/$TRANSMITTANCES line
}

      begin

        i:=1;              { get the name of the node that the line defines }

        l:=0;

        while (i+l<length(s)) and (pos(s[i+l],';:,=$*')=0) do inc(l);

        if (l=0) or (s[i+l]<>':') or (i+l=length(s))

          then error(6,copy(s,1,i+l));

        if l>15 then error(2,copy(s,i,l));

        nm:=copy(s,i,l);

        findnode(node,nm);         { find the node or create it }

        if node<>sinknode then     { ignore geometry defn. for output node }

          repeat                   { NB if input=output name, the INPUT    }

            i:=i+l+1;              { node will be found.                    }

            l:=0;         { find each node name in geometry list in turn }

            while (i+l<=length(s)) and (pos(s[i+l],';:,=$*')=0) do inc(l);
```

```
                if ((l=0) and not((i+l<=length(s)) and (s[i+l]=';'))) or

                   ((i+l<=length(s)) and (pos(s[i+l],':=$*')>0)) or

                   ((i+l<length(s)) and (s[i+l]=';') and

                    ((node^.next=nil) or (node^.name<>node^.next^.name)))

                  then error(6,copy(s,1,i+l));   { syntax check }

                if l>15 then error(2,copy(s,i,l));

                nm:=copy(s,i,l);

                if nm=node^.name then error(7,nm);

                findnode(outputnode,nm);     { find the listed node, and     }

                new(link);                   { create a link (value=1) to it }

                link^.next:=node^.firstlink;

                node^.firstlink:=link;

                link^.dest:=outputnode;

                new_scalar(link^.val,1);     { switch to 2nd node of same name }

                if (i+l<length(s)) and (s[i+l]=';') then     { when ';' found }

                   if node=sourcenode then error(14,'')

                                    else node:=node^.next;

              until (i+l>length(s)) or ((i+l=length(s)) and (s[i+l]=';'));

        end;             { ignore ';' if it is the last thing on the line }

     until (s[1]='$') or eof(f);

   if not(eof(f)) and (copy(s,1,15)<>'$TRANSMITTANCES') then error(8,'');

end;        { terminate at end of file or at "$TRANSMITTANCES" line }
```

```
procedure correctlinks;

{ checks that (1) outgoing links exist, (2) reciprocal links exist, and   }

{ (3) that no multiple links exist.  Shifts link destinations from a node }

{ to its mirror node (i.e. same point in optical cct, opposite direction  }

{ of light travel) as required so that no A -> B -> A type links exist.    }

var destnode:node_ptr;

    link,otherlink:link_ptr;

begin

  node:=firstnode;

  while node<>nil do     { step through each node in turn }

    begin

      link:=node^.firstlink;

      if link<>nil then

        while link<>nil do    { step through each link on that node in turn }

          begin

            destnode:=link^.dest;

            if (destnode<>sinknode) then  { check for reciprocal link }

              begin                          { except from sink node     }

                otherlink:=destnode^.firstlink;

                while (otherlink<>nil) and
(otherlink^.dest^.name<>node^.name)

                  do otherlink:=otherlink^.next;

                if destnode<>sourcenode then  { if reciprocal link found }

                  if otherlink<>nil then link^.dest:=destnode^.next

                  else    { on first node of pair, shift original link to
}

                    begin { 2nd...otherwise continue check on 2nd same-name
}
```

```
                         otherlink:=destnode^.next^.firstlink;            { node
}

                      while (otherlink<>nil) and

                          (otherlink^.dest^.name<>node^.name)

                        do otherlink:=otherlink^.next;

                      if otherlink=nil

                        then error(10,node^.name+' -> '+destnode^.name);

                    end  { err if no reciprocal link found }

                  else   { link goes to source node => error unless names of }

                    if otherlink=nil  { source, sink are the same - then just}

                      then error(10,node^.name+' -> '+destnode^.name)

                      else if sourcenode^.name=sinknode^.name   { shift it to
}

                        then link^.dest:=sinknode;               { the sink
}

             end;

         otherlink:=link^.next;  { check for 2nd link to same node }

         while (otherlink<>nil) and

             (otherlink^.dest^.name<>link^.dest^.name)

           do otherlink:=otherlink^.next;

         if (otherlink=nil) and (node^.next<>nil) { may need to continue
}

             and (node^.next^.name=node^.name)     { search on 2nd same-
}

           then otherlink:=node^.next^.firstlink; { name node
}

         while (otherlink<>nil) and

             (otherlink^.dest^.name<>link^.dest^.name)

           do otherlink:=otherlink^.next;
```

```pascal
           if (otherlink<>nil) then    { error if duplicate link found }

              error(11,node^.name+' <--> '+link^.dest^.name);

          link:=link^.next;    { do next link }

      end

  else if (node=sourcenode) or ((node^.next<>nil) and

          (node^.next^.firstlink=nil) and
(node^.next^.name=node^.name))

          then error(9,node^.name);  { no links from sourcenode or from }

    node:=node^.next;                  { a pair of same-name node => err! }

  end;              { check next node... }

end;
```

```pascal
procedure getlinkvals;

{ read the input file to obtain expressions for links (=1 if not listed) }

var s:string;

    node1,node2,tmp_node:node_ptr;

    link:link_ptr;

    nm1,nm2:string[15];

    oneway:boolean;

    i,l:byte;

    mag_val,angl_num,mag_pwr,zcoeff:double;

    mag_expr,angl_expr,z_expr:string;

begin

  repeat              { get each remaining line of the input file in turn }

    readln(f,s);

    inc(line);

    killwhitespace(s);

    if s<>'' then     { ignore blanks, comments }

      begin

        i:=1;

        l:=0;

        if s[1]='*' then    { check for * at start of line, denoting }

          begin             { a one-way only link assignment.        }

            oneway:=true;

            i:=2;

            if length(s)=1 then error(6,s);

          end

        else oneway:=false;   { get first node name and the comma }
```

```pascal
while (i+l<length(s)) and (pos(s[i+l],';:,=$*')=0) do inc(l);

if (s[i+l]<>',') or (length(s)=i+l) or (l=0)

  then error(6,copy(s,1,i+l)) else

if l>15 then error(2,copy(s,i,l)) else nm1:=copy(s,i,l);

i:=i+l+1;

l:=0;                  { get second node name }

while (i+l<length(s)) and (pos(s[i+l],';:,=$*')=0) do inc(l);

if (s[i+l]<>'=') or (length(s)=i+l) or (l=0)

  then error(6,copy(s,1,i+l)) else

if l>15 then error(2,copy(s,i,l)) else nm2:=copy(s,i,l);

i:=i+l+1;

l:=0;

node1:=firstnode;    { find both nodes, check that they exist }

node2:=node1;

while (node1<>nil) and (node1^.name<>nm1) do

  begin

    if (node2^.name<>nm2) then node2:=node1;

    node1:=node1^.next;

  end;

while (node2<>nil) and (node2^.name<>nm2) do node2:=node2^.next;

if (node1=nil) then error(12,nm1);

if (node2=nil) then error(12,nm2);

mag_val:=1;            { set defaults for any part of the    }

mag_pwr:=0;            { expression not included in the input }

angl_num:=0;

zcoeff:=0;

mag_expr:='';
```

```
angl_expr:='';

z_expr:='';             { (1) look for a number }

while (i+l<=length(s)) and (pos(s[i+l],'+-0123456789eE.')>0) do
inc(l);

if l>0 then

  begin

    val(copy(s,i,l),mag_val,errpos);

    if errpos>0 then error(6,copy(s,1,i+errpos));

    i:=i+l;

    l:=0;

  end;                  { (2) look for a magterm expr }

if (i<length(s)) and (s[i]='{') then

  begin

    inc(i);

    while (i+l<length(s)) and (s[i+l]<>'}') do inc(l);

    if s[i+l]<>'}' then error(6,s);

    mag_expr:=copy(s,i,l);

    mag_pwr:=1;

    i:=i+l+1;

    l:=0;               { (3) if found one, look for associated ^num }

    if (i<length(s)) and (s[i]='^') then

      begin

        inc(i);

        while (i+l<=length(s)) and (pos(s[i+l],'+-0123456789eE.')>0)

          do inc(l);

        if l=0 then error(6,copy(s,1,i));

        val(copy(s,i,l),mag_pwr,errpos);
```

```
          if errpos>0 then error(6,copy(s,1,i+errpos));

          i:=i+l+1;

          l:=0;

        end;

    end;                  { (4) look for an angle term identifier '<' }

if (i<length(s)) and (s[i]='<') then

   begin

      inc(i);            { (5) if found '<' look for a number }

      while (i+l<=length(s)) and (pos(s[i+l],'+-0123456789eE.')>0)

         do inc(l);

      if l>0 then

         begin

            val(copy(s,i,l),angl_num,errpos);

            if errpos>0 then error(6,copy(s,1,i+errpos));

            i:=i+l;

            l:=0;

         end              { (6) ... then look for an anglterm expr }

      else if (i<length(s)) and (s[i]='{') then angl_num:=1;

      if (i<length(s)) and (s[i]='{') then

         begin

            inc(i);

            while (i+l<length(s)) and (s[i+l]<>'}') do inc(l);

            if s[i+l]<>'}' then error(6,s);

            angl_expr:=copy(s,i,l);

            i:=i+l+1;

            l:=0;

         end;
```

```
          end;                 { (7) look for a zterm identifier 'z' or 'Z' }

      if ((i<=length(s)) and (upcase(s[i])<>'Z')) then
error(6,copy(s,1,i))

      else if i=length(s) then zcoeff:=1 else if i<length(s) then

        begin          { if it's not the last char. on the line, it must }

          inc(i);      { be followed by ^something                       }

          if s[i]<>'^' then error(6,copy(s,1,i)) else

          if i=length(s) then error(6,s) else inc(i);

          while (i+l<=length(s)) and (pos(s[i+l],'+-0123456789eE.')>0)

            do inc(l);     { (8) look for a number (zterm .num field) }

          if l>0 then

            begin

              val(copy(s,i,l),zcoeff,errpos);

              if errpos>0 then error(6,copy(s,1,i+errpos));

              i:=i+l;

              l:=0;

            end            { (9) look for a zterm expr }

          else if (i<length(s)) and (s[i]='{') then zcoeff:=1;

          if i=length(s) then error(6,s);

          if (i<length(s)) and (s[i]='{') then

            begin

              inc(i);

              while (i+l<length(s)) and (s[i+l]<>'}') do inc(l);

              if s[i+l]<>'}' then error(6,s);

              if i+l<length(s) then error(6,copy(s,1,i+l+1));

              z_expr:=copy(s,i,l);

            end;
```

```
            end;

        if node1<>node2 then  { is it a reflection coefficient? }

          begin       { NO: then if it involves source or sink, may have }

            if sourcenode^.name<>sinknode^.name then          { to swap }

              begin     { if oneway is set, order is already explicit }

                if not(oneway) and (node1=sinknode) or (node2=sourcenode)

                  then begin

                        tmp_node:=node1;  { swap nodes so that oneway }

                        node1:=node2;      { direction is appropriate, }

                        node2:=tmp_node;  { and set oneway. (since     }

                        oneway:=true;      { source/sink only have one }

                      end;                 { way links)                }

                if (node1=sourcenode) or (node2=sinknode) then oneway:=true;

              end;                    { find the desired link }

          link:=node1^.firstlink;

          while (link<>nil) and (link^.dest^.name<>nm2) do
link:=link^.next;

          if (link=nil) and (node1^.next<>nil) and (node1^.next^.name=nm1)

            then link:=node1^.next^.firstlink;

          while (link<>nil) and (link^.dest^.name<>nm2) do
link:=link^.next;

          if link=nil then error(13,nm1+' -> '+nm2) else  { link absent? }

            begin                  { set its value }

              dispose_expression(link^.val);

              new_link_val(link^.val,

                        mag_val,mag_expr,mag_pwr,

                        angl_num,angl_expr,
```

```
                         zcoeff,z_expr);

              end;

         if not(oneway) then    { if setting value for both directions, }

           begin                { find the reciprocal (other way) link   }

             link:=node2^.firstlink;

             while (link<>nil) and (link^.dest^.name<>nm1)

               do link:=link^.next;

             if (link=nil) and (node2^.next<>nil) and
(node2^.next^.name=nm2)

               then link:=node2^.next^.firstlink;

             while (link<>nil) and (link^.dest^.name<>nm1)

               do link:=link^.next;

             if link=nil then error(13,nm2+' -> '+nm1) else { absent? }

               begin               { set its value }

                 dispose_expression(link^.val);

                 new_link_val(link^.val,

                              mag_val,mag_expr,mag_pwr,

                              angl_num,angl_expr,

                              zcoeff,z_expr);

               end;

           end;

         end

       else  { from earlier: names are the same so its a reflection coeff }

         begin

           if (node1=sourcenode) or (node1=sinknode)

             then error(13,nm1+' -> '+nm1);   { source or sink not allowed
}
```

```
node2:=node1^.next;    { get node and its same-name counterpart }

if (node1^.firstlink<>nil) and    { reflection already set? }

   (node1^.firstlink^.dest=node2) then

  begin                      { if so, replace it }

    dispose_expression(node1^.firstlink^.val);

    new_link_val(node1^.firstlink^.val,

               mag_val,mag_expr,mag_pwr,

               angl_num,angl_expr,

               zcoeff,z_expr);

  end

else if (node2^.firstlink<>nil) and  { already set, other way? }

        (node2^.firstlink^.dest=node1) then

  begin                       { replace existing link }

    dispose_expression(node2^.firstlink^.val);

    new_link_val(node2^.firstlink^.val,

               mag_val,mag_expr,mag_pwr,

               angl_num,angl_expr,

               zcoeff,z_expr);

  end

else if (node1^.firstlink=nil) and (node2^.firstlink<>nil) then

  begin       { create new link on 1st node and set value }

    new(link); { if the 1st node is the input node for the }

    link^.next:=nil;                       { same-name pair }

    node1^.firstlink:=link;

    link^.dest:=node2;

    new_link_val(link^.val,

               mag_val,mag_expr,mag_pwr,
```

```pascal
                              angl_num,angl_expr,

                              zcoeff,z_expr);

                   end

           else if (node2^.firstlink=nil) and (node1^.firstlink<>nil) then

             begin          { otherwise create new link on 2nd node... }

                new(link);

                link^.next:=nil;

                node2^.firstlink:=link;

                link^.dest:=node1;

                new_link_val(link^.val,

                              mag_val,mag_expr,mag_pwr,

                              angl_num,angl_expr,

                              zcoeff,z_expr);

                   end

             else error(13,nm1+' -> '+nm1);  { neither node was the input    }

          end;  { of a cut-off fiber end. So reflections aren't allowed!!! }

       end;

  until eof(f);   { keep getting lines until end of input file }

end;
```

```
procedure searchfrom(root:node_ptr);

{ performs a recursive search from "root" node to find loops and paths }

begin

  if root^.link_taken<>nil then { hit a node already passed: loop found }

    begin

      found_duplicate:=false;        { CHECK IF LOOP ALREADY EXISTS:        }

      loopentry:=root^.touchloop;   { compare with existing loops at node }

      while not(found_duplicate) and (loopentry<>nil) do  { check each one }

        begin        { find the current node in its node list }

          nodeentry:=loopentry^.loop^.nodelist;

          while nodeentry^.node<>root do nodeentry:=nodeentry^.next;

          node:=root;

          repeat    { follow both loops a step at a time until we cycle, or
}

            node:=node^.link_taken^.dest;     { we get to 2 different nodes
}

            nodeentry:=nodeentry^.next;

          until (node=root) or (node<>nodeentry^.node); { if we cycled,loop
}

          if node=nodeentry^.node then found_duplicate:=true; { isn't new
}

          loopentry:=loopentry^.next;  { check next loop at node }

        end;

      if not(found_duplicate) then    { CREATE NEW LOOP: }

        begin

          new(loop);                        { create loop entry }

          loop^.next:=firstloop;

          firstloop:=loop;
```

```
loop^.touchloop:=nil;

new_scalar(loop^.loopgain,-1);     { -loopgain actually stored }

loop^.nodelist:=nil;                    { will * one link at a time }

loop^.deactivated:=0;                   { activate loop- used later }

nodeentry:=nil;

node:=root;          { along each node on the loop do 5 things: }

repeat with node^ do

  begin

    new(loopentry);     { (1) add loop to list of loops at node }

    loopentry^.loop:=loop;

    loopentry^.next:=touchloop;

    touchloop:=loopentry;

    if nodeentry=nil then    { (2) add node to nodelist for loop }

      begin

        new(loop^.nodelist);

        nodeentry:=loop^.nodelist;

      end

    else

      begin

        new(nodeentry^.next);

        nodeentry:=nodeentry^.next;

      end;

    nodeentry^.node:=node;

    loopentry:=touchloop;     { (3) add all loops at node to      }

    while loopentry<>nil do  {      touchloop list for this loop }

      begin                         { (but check for duplicates first) }

        loopentry2:=loop^.touchloop;
```

```
        while (loopentry2<>nil) and

            (loopentry2^.loop<>loopentry^.loop)

          do loopentry2:=loopentry2^.next;

        if loopentry2=nil then        { add loop to list }

          begin

            new(loopentry2);

            loopentry2^.loop:=loopentry^.loop;

            loopentry2^.next:=loop^.touchloop;

            loop^.touchloop:=loopentry2;

          end;

        loopentry:=loopentry^.next;

      end;

  pathentry:=touchpath;      { (4) add this loop to touchloop }

  while pathentry<>nil do   {     list of all paths at node  }

    begin

      loopentry2:=pathentry^.path^.touchloop;

      while (loopentry2<>nil) and (loopentry2^.loop<>loop)

        do loopentry2:=loopentry2^.next;

      if loopentry2=nil then   { check for existing entry }

        begin                  { none, so add to list }

          new(loopentry2);

          loopentry2^.loop:=loop;

          loopentry2^.next:=pathentry^.path^.touchloop;

          pathentry^.path^.touchloop:=loopentry2;

        end;

      pathentry:=pathentry^.next;

    end;    { (5) multiply loopgain by value of outgoing link }
```

```
              multiply(mult_result,loop^.loopgain,link_taken^.val);

              dispose_expression(loop^.loopgain);

              loop^.loopgain:=mult_result;

              node:=link_taken^.dest

           end;

        until node=root;  { cycle through nodes until back at the start }

        nodeentry^.next:=loop^.nodelist;  { nodelist itself should loop }

     end;                               { i.e. it's circularly linked }

  end

else if root=sinknode then { REACHED SINKNODE: NEW PATH FOUND }

  begin

    new(path);              { create path entry }

    path^.next:=firstpath;

    firstpath:=path;

    path^.touchloop:=nil;

    new_scalar(path^.pathgain,1);

    node:=sourcenode;       { along each node on the path do 3 things: }

    repeat with node^ do

      begin

        new(pathentry);         { (1) add path to list of paths at node }

        pathentry^.path:=path;

        pathentry^.next:=touchpath;

        touchpath:=pathentry;

        loopentry:=touchloop;    { (2) add all loops at node to loop }

        while loopentry<>nil do  {     list for this path (but check }

          begin                  {        for duplicates first)         }

              loopentry2:=path^.touchloop;
```

```
            while (loopentry2<>nil) and

                  (loopentry2^.loop<>loopentry^.loop)

               do loopentry2:=loopentry2^.next;

            if loopentry2=nil then      { add loop to list }

              begin

                new(loopentry2);

                loopentry2^.loop:=loopentry^.loop;

                loopentry2^.next:=path^.touchloop;

                path^.touchloop:=loopentry2;

              end;

            loopentry:=loopentry^.next;

          end;

       if link_taken=nil then node:=nil else { (3) multiply pathgain by }

          begin                                 {    value of outgoing link }

            multiply(mult_result,path^.pathgain,link_taken^.val);

            dispose_expression(path^.pathgain);

            path^.pathgain:=mult_result;

            node:=link_taken^.dest

          end;

     end;

  until node=nil;  { step through nodes on the path until none remain }

 end

else with root^ do       { continue depth-first search }

  begin

    link_taken:=firstlink;   { mark the way we went }

    while link_taken<>nil do

      begin      { don't follow 'phantom' links TO the source node }
```

```
            if link_taken^.dest<>sourcenode then searchfrom(link_taken^.dest);

            link_taken:=link_taken^.next;    { go a different way... }

        end;

    end;

end;
```

```
procedure deletenodedata;

{ frees memory used for storing node information (no longer needed) }

var nextentry:pointer;

    link:link_ptr;

begin

  loop:=firstloop;           { remove each loop's nodelist }

  while loop<>nil do

    begin

      nodeentry:=loop^.nodelist;

      while nodeentry<>nil do

        begin

          nextentry:=nodeentry^.next;

          if nextentry=loop^.nodelist then nextentry:=nil;

          dispose(nodeentry);

          nodeentry:=nextentry;

        end;

      loop^.nodelist:=nil;

      loop:=loop^.next;

    end;

  node:=firstnode;                { remove all data for each node: }

  while node<>nil do

    begin

      pathentry:=node^.touchpath;    { remove the touchpath list }

      while pathentry<>nil do

        begin

          nextentry:=pathentry^.next;
```

```
        dispose(pathentry);

        pathentry:=nextentry;

      end;

    loopentry:=node^.touchloop;    { remove the touchloop list }

    while loopentry<>nil do

      begin

        nextentry:=loopentry^.next;

        dispose(loopentry);

        loopentry:=nextentry;

      end;

    link:=node^.firstlink;         { remove the links }

    while link<>nil do

      begin

        nextentry:=link^.next;

        dispose_expression(link^.val);

        dispose(link);

        link:=nextentry;

      end;

    nextentry:=node^.next;         { remove the node itself }

    dispose(node);

    node:=nextentry;

  end;

end;
```

```pascal
procedure addloopcombination(var delta:expression_ptr;

                                 running_product:expression_ptr;

                                 current_loop:loop_ptr);

{ recursively adds to delta the sum of all possible products of }

{ nontouching loop gains, down the list from 'current_loop'.    }

{ 'running product' contains the product of those loops already }

{ used... i.e. the ones up the list from current_loop that were }

{ taken on the way to getting to current_loop.                  }

var tempexpression:expression_ptr;

begin

  while current_loop<>nil do  { step through all possible branches    }

    begin                     { and descend to them in turn if active }

      if current_loop^.deactivated=0 then  { skip deactivated ones, i.e. }

        begin                              { those already 'touched'     }

          multiply(tempexpression,current_loop^.loopgain,running_product);

          add(delta,tempexpression); { add to delta the tree descent so far
}

          loopentry:=current_loop^.touchloop;  { deactivate touching loops }

          while loopentry<>nil do              { further down the list     }

            begin

              inc(loopentry^.loop^.deactivated);

              loopentry:=loopentry^.next;

            end;        { recursively add to delta all terms downbranch }

          addloopcombination(delta,tempexpression,current_loop^.next);

          dispose_expression(tempexpression);

          loopentry:=current_loop^.touchloop;  { reactivate loops }

          while loopentry<>nil do              { (or clear deactivations }
```

```
            begin                            { set by current loop)      }

                dec(loopentry^.loop^.deactivated);

                loopentry:=loopentry^.next;

            end;

        end;

    current_loop:=current_loop^.next;   { do next branch }

    end;

end;
```

```
procedure calcdelta(var delta:expression_ptr);

{ calculates the Mason rule delta, using all loops not already deactivated }

var running_product:expression_ptr;

begin

  new_scalar(delta,1);

  new_scalar(running_product,1);

  addloopcombination(delta,running_product,firstloop);

  dispose_expression(running_product);

end;
```

```
procedure calctransferfunction;

{ calculates the numerator and denominator of the signal flow }

{ graph transfer function using the Mason's rule equation     }

var delta:expression_ptr;

begin

  calcdelta(denominator);        { calculate denominator = delta }

  new_scalar(numerator,0);       { set numerator = 0 }

  path:=firstpath;

  while path<>nil do             { for each path in turn: }

    begin

      loopentry:=path^.touchloop;    { deactivate each loop that touches it }

      while loopentry<>nil do

        begin

          loopentry^.loop^.deactivated:=1;

          loopentry:=loopentry^.next;

        end;

      calcdelta(delta);                { find delta for this reduced loop set }

      multiply(mult_result,delta,path^.pathgain);

      dispose_expression(delta);

      dispose_expression(path^.pathgain);

      add(numerator,mult_result); { numerator <- numerator + delta*pathgain
}

      dispose_expression(mult_result);

      loopentry:=path^.touchloop;    { reactivate all the loops     }

      while loopentry<>nil do         { deactivated by current path }

        begin

          loopentry^.loop^.deactivated:=0;
```

```
                loopentry:=loopentry^.next;

            end;

        path:=path^.next;    { do next path }

      end;

  end;
```

```pascal
{ ######################### MAIN PROGRAM ############################ }

begin

  writeln;

  writeln;

  prec:=3;   { default to 3 decimal places in the output }

  errpos:=0;

  outfile_name:=paramstr(2);

  precparam:=paramstr(3);

  if outfile_name[1]='-' then  { if outfile is omitted, -d may be the }

    begin                      { second command line parameter        }

      precparam:=outfile_name;

      outfile_name:='';

    end;

  if precparam<>'' then        { set precision if parameter present }

    val(copy(precparam,2,length(precparam)-1),prec,errpos);

  if (paramcount=0) or (paramcount>3) or

     ((paramcount=3) and (precparam[1]<>'-')) or (errpos<>0) then

    begin    { if the parameters weren't right, show instructions and quit }

      writeln('OptMason generates the transfer function of an optical
network using Mason''s');

      writeln('rule for signal-flow graphs.');

      writeln;

      writeln('Written by Dean Trower, 1996.  This program may be freely
distributed.');

      writeln;

      writeln;

      writeln('USAGE:  optmason  input_file  [output_file]  [-d]');
```

```
      writeln;

      writeln('d is the number of decimal places that numbers are displayed
to in the output.');

      writeln('If the output file is omitted, output is to the screen (or
standard output).');

      writeln('If the output file has the same name as the input file, the
output is appended');

      writeln('to the input file.');

      writeln;

      halt;

    end;

  mark(heap_state);    { record the state of memory, to restore it when done
}

  firstnode:=nil;      { init lists and miscellaneous }

  firstpath:=nil;

  firstloop:=nil;

  plus_waiting:=false;

  assign(f,paramstr(1));  { attempt to open input file }

  reset(f);

  if IOResult<>0 then error(0,paramstr(1));

  getsourcesink;            { parse input, output definitions }

  getlinks;                 { parse geometry description }

  correctlinks;             { create signal flow graph geometry, and check }

  getlinkvals;              { parse transmittance expressions }

  close(f);                 { close the input file }

  writeln('SIGNAL FLOW GRAPH CREATED');  { this always goes to the SCREEN }

  searchfrom(sourcenode);  { recursively find all loops and paths }

  deletenodedata;     { get rid signal flow graph, now no longer necessary }
```

```pascal
    writeln('PATH AND LOOP INFORMATION COMPUTED');  { displays to SCREEN }

    calctransferfunction;     { apply Mason's rule to loop and path info }

    writeln('TRANSFER FUNCTION T COMPUTED');   { displays to SCREEN }

    assign(f,outfile_name);   { create or append the desired output file }

    if paramstr(1)=outfile_name then append(f) else rewrite(f);

    if IOresult<>0 then error(0,outfile_name);

    writeln(f);  { NB a file name of '' get assigned to DOS standard output }

    writeln(f,'T = numerator/denominator');   { output the results }

    writeln(f);

    write(f,'numerator = ');

    writeexpression(f,numerator);

    writeln(f);

    writeln(f);

    write(f,'denominator = ');

    writeexpression(f,denominator);

    writeln(f);

    close(f);                       { close the output file }

    release(heap_state);            { deallocate all memory that got used }

end.
```