```
-- $Id: paraffins.sa,v 1.5 1995/06/12 08:16:40 borisv Exp borisv $

-- $Log: paraffins.sa,v $
-- Revision 1.5  1995/06/12 08:16:40  borisv
-- Added an option to allow to specify serial or parallel
-- execution.
-- Added some comments
--
-- Revision 1.4  1995/06/09 10:03:06  borisv
-- Made PARTITION_ENUMERATOR a separate class
--
-- Revision 1.3  1995/06/09 01:02:44  borisv
-- Fixed a bug in recursively_partition!
--
-- Revision 1.2  1995/06/09 00:25:02  borisv
-- enum_parttions! is left recursive for now
--
-- Revision 1.1  1995/06/08 18:52:05  borisv
-- Initial revision
--


-- Salishan Paraffins Problem:
-- Given an integer n, output the chemical structure of all
-- paraffin molecules for i<=n, without repetition and in
-- order of increasing size. Include all isomers, but no
-- dupicates. The chemical formula for paraffin molecules is
-- C(i)H(2i+2). Any representation for the molecules could
-- be chosen, as long as it clearly distinguishes among
-- isomers.

-- Solution is based on theory of free and oriented trees
-- (see Knuth, D. The Art of Computer Programming, Vol. 1)

type $PRINTABLE is
    -- supertype of everything

    print;
end;

type $RADICAL < $PRINTABLE  is
    -- supertype for various paraffin radicals

    -- sanity checks
    total_hydrogen:INT; -- total number of hydrogen atoms in a radical
    total_carbon:INT;   -- total number of carbon atoms in a radical
end;

type $MOLECULE < $PRINTABLE is
    -- supertype for various paraffin molecules

    num_neighbors:INT;

    total_hydrogen:INT; -- total number of hydrogen atoms in a molecule
    total_carbon:INT;   -- total number of carbon atoms in a molecule
end;

type $PARAFFINS < $PARTITION_ENUMERATOR, $PRINTABLE is
    -- supertype for all paraffin isomers
    -- ith element of the array contains a list of all
    -- isomers of size i;

    molecules:ARRAY{FLIST{$MOLECULE}};
end;
```

```
type $PARTITION_ENUMERATOR is
    -- supertype for a stand alone enumerator class that
    -- actually does most of dirty work.
    -- enum_partitions! generates all ordered partitions
    -- of size number_of_elements, such that the sum of
    -- all elements in the parttion is sum_of_elements,
    -- and min and max elements are min_element and
    -- max_element respectively. Example: sum_of_elements=7,
    -- number_of_elements=3, min_elemnt=0, max_element=7
    -- (0,0,7),(0,1,6),(0,2,5),(0,3,4),(1,1,5),(1,2,4),
    -- (1,3,3),(2,2,3)

    -- Note: dispatched ites are not yet implemented.
    -- Hence stuff below is commented out.

--    enum_partitions!( sum_of_elements:INT, number_of_elements:INT,
--       min_element:INT, max_element:INT):ARRAY{INT};
end;

----------------------------------------------------------------------------
class PARTITION_ENUMERATOR < $PARTITION_ENUMERATOR is

   -- implementation of a partition enumerator: at each step, returns
   -- a new partition conforming to the global lexicographically
   -- ascending order
    enum_partitions!( sum_of_elements:INT, number_of_elements:INT,
             min_element:INT, max_element:INT):ARRAY{INT} is
        p ::= #ARRAY{INT} (number_of_elements);
        loop
            tmp ::= recursively_partition!(number_of_elements, min_element,
                 max_element,0, sum_of_elements, min_element, p);
            yield p.copy;  -- always return a new array
        end;
    end;

    -- The inner iterator for partition enumeration. It is
    -- called with the first element of the partition set.
    -- Then, it keeps calling itself recursively until it
    -- assigns all partition elements. When the last element
    -- is assigned, it returns true, otherwise false
    -- Although it looks deceptively simple, it is in fact
    -- rather tricky, with complexity coming from the
    -- interactions between recusrive calls and persistent
    -- state of the iters.
    -- It quits when a would be  partition breaks the
    -- lexicographically ascending order.

    recursively_partition!(number_of_elements:INT, min_element:INT,
             max_element:INT, level:INT!, remainder:INT!, prev_element:INT!,
             p:ARRAY{INT}!):BOOL is

        element, largest_element : INT;

        remaining_levels ::= number_of_elements - level-1;

        element := prev_element.max(remainder-max_element*remaining_levels);
        tmp ::= remainder-min_element*remaining_levels;
        largest_element := tmp.min(remainder / (remaining_levels + 1));

        loop
            if remaining_levels > 0 then
                loop
                    if element <= largest_element then
```

```
                        p[level] := element;
                        if recursively_partition!(number_of_elements,
                             min_element, max_element, level+1,
                             remainder-element, element,p) then
                            element := element + 1;
                        end;
                        -- not all elements assigned
                        yield false;
                    else
                        -- we are done, no ascending order possible
                        quit;
                    end;
                end;
                element := element + 1;
            else
                p[level] := remainder;
                -- just fished one more legal partition
                yield true;
            end;
        end;
    end;

end;  -- class PARTITION_ENUMERATOR
----------------------------------------------------------------------------

-- Classes for various paraffin radicals

class RADICAL is
    attr total_hydrogen:INT;  -- total number of hydrogen atoms in a radical
    attr total_carbon:INT;    -- total number of carbon atoms in a radical
end;

class HYDROGEN_RADICAL < $RADICAL is -- hydrogen radical has no neighbors
    include RADICAL;

    print is
        #OUT + "H";
    end;

    create:SAME is
        r ::=new;
        r.total_hydrogen := 1;
        r.total_carbon := 0;
        return r;
    end;

end;  -- class HYDROGEN_RADICAL

class CARBON_RADICAL < $RADICAL is  -- carboniferous radical has 3 neighbors
    include RADICAL;
    attr carbon_neighbors : ARRAY{$RADICAL}; -- radicals attached to C root

    print is
        #OUT + "C(";
        loop
            carbon_neighbors.elt!.print;
        end;
        #OUT + ")";
    end;

    create:SAME is
        r ::= new;
        r.carbon_neighbors := #(3); -- carboniferous radical has 3 neighbors
        r.total_carbon := 1;        -- centroid C atom
```

```
        r.total_hydrogen := 0;
        return r;
    end;

    create(subradicals:ARRAY{$RADICAL}) : SAME is
        -- create a carboniferous radical from 3 other radicals
        r ::= new;
        r.carbon_neighbors := #(3); -- carboniferous radical has 3 neighbor
        r.carbon_neighbors.copy(subradicals);
        r.total_carbon := 1;        -- centroid C atom
        r.total_hydrogen := 0;

        -- Compute the total number of hydrogen and
        -- carbon atoms in the new radical
        loop
            rad ::= r.carbon_neighbors.elt!;
            r.total_hydrogen := r.total_hydrogen + rad.total_hydrogen;
            r.total_carbon := r.total_carbon + rad.total_carbon;
        end;
        return r;
    end;

end;  -- class CARBON_RADICAL
----------------------------------------------------------------------------

-- various paraffin molecules

class MOLECULE is
    attr total_hydrogen:INT;
    attr total_carbon:INT;
end;

class BCP_MOLECULE < $MOLECULE is
    -- ‘‘Bond centered paraffins (BCPs)’’: even-sized double centroid
    -- paraffin molecules. The root corresponds to a carbon-
    -- carbon bond connecting two radicals. The two radicals
    -- of a BCP of size i each have exactly i/2 carbon atoms.
    include MOLECULE;

    const num_neighbors:INT := 2;  -- has 2 radical neighbors
    attr bond_neighbors:ARRAY{$RADICAL};
    print is
        #OUT + "BCP molecule:  "
             + "C(" + total_carbon + ")"
             + "H(" + total_hydrogen + ")"
             + "    (";
        loop
            elt ::= bond_neighbors.elt!;
            if ~void(elt) then
                #OUT + "(";
                elt.print;
                #OUT + ")";
            end;
        end;
        #OUT + ")\n";
    end;

    create:SAME is
        r ::= new;
        r.bond_neighbors := #(r.num_neighbors);
        r.total_hydrogen := 0;
        r.total_carbon := 0;
        return r;
    end;
```

```
    create(radicals:ARRAY{$RADICAL}):SAME is
        -- make a BCP paraffin molecule from 2 radicals
        r ::= new;
        r.bond_neighbors := #(r.num_neighbors);
        r.bond_neighbors.copy(radicals);
        r.total_hydrogen := 0;  -- everything is attached to
        r.total_carbon := 0;    -- a carbon bond

        -- compute the total number of carbon and
        -- hydrogen atoms in the molecule
        loop
            rad ::= r.bond_neighbors.elt!;
            r.total_hydrogen := r.total_hydrogen + rad.total_hydrogen;
            r.total_carbon := r.total_carbon + rad.total_carbon;
        end;

        return r;
    end;
end;  -- class BCP_MOLECULE

class CCP_MOLECULE < $MOLECULE is
    -- ‘‘Carbon centered paraffins (CCPs)’’: odd-sized paraffin molecules
    -- or even-sized single-centroid paraffin molecules
    -- The root is a carbon atom that has 4 subtrees representing
    -- paraffin radicals. Each radical of a CCP of size i is
    -- <= (i-1)/2
    include MOLECULE;


    const num_neighbors:INT := 4;  -- has 4 radical neighbors
    attr carbon_neighbors:ARRAY{$RADICAL};
    print is
        #OUT + "CCP molecule:  "
             + "C(" + total_carbon + ")"
             + "H(" + total_hydrogen + ")"
             + "    (C";
        loop
            elt ::=carbon_neighbors.elt!;
            if ~void(elt) then
                #OUT + "(";
                elt.print;
                #OUT + ")";
            end;
        end;
        #OUT + ")\n";
    end;

    create:SAME is
        r ::= new;
        r.carbon_neighbors := #(r.num_neighbors);
        r.total_hydrogen := 0;
        r.total_carbon := 0;
        return r;
    end;

    create(radicals:ARRAY{$RADICAL}):SAME is
        -- make a CCP paraffin molecule from 4 radicals
        r ::= new;
        r.carbon_neighbors := #(r.num_neighbors);
        r.carbon_neighbors.copy(radicals);
        r.total_hydrogen := 0;
        r.total_carbon := 1;  -- carbon atom is a centroid!
```

```
        -- compute the total number of carbon and
        -- hydrogen atoms in the molecule
        loop
            rad ::= r.carbon_neighbors.elt!;
            r.total_hydrogen := r.total_hydrogen + rad.total_hydrogen;
            r.total_carbon := r.total_carbon + rad.total_carbon;
        end;

        return r;
    end;
end;  -- class CCP_MOLECULE
----------------------------------------------------------------------------

class PARAFFINS < $PARAFFINS is
    include PARTITION_ENUMERATOR;

    attr radicals:ARRAY{FLIST{$RADICAL}};   -- paraffin radicals from 0 to n/2
    attr molecules:ARRAY{FLIST{$MOLECULE}}; -- paraffin molecules from 0 to n

    print is
        -- compute the total number of isomers
        total_molecules ::= 0;
        loop
            i ::= 1.upto!(molecules.size-1);
            total_molecules := total_molecules + molecules[i].size;
        end;

        #OUT + "All paraffin isomers with up to "
             + (molecules.size-1)   -- compiler bombs without parens!
             + " carbon atoms\n\n";
        #OUT + "Total Molecules: " + total_molecules + "\n";

        loop
            molecule_list ::= molecules.elt!(1);
            loop
                molecule_list.elt!.print;
            end;

        end;
    end;

    --------------------------------------------------------

    gen_radicals_of_size(size:INT) is
        -- generate all radicals of a specified size
        enum_partitions_for_rads(size-1, 3, 0, size-1, size);
    end;

    enum_partitions_for_rads(sum:INT, number:INT, min:INT, max:INT, i:INT) is
        -- generates partitions representing subradicals
        -- and then actually creates new radicals based
        -- on these partitions
        loop
            p ::= enum_partitions!(sum, number, min, max);
            enum_subrads_for_rads(p, i);
        end;
    end;

    enum_subrads_for_rads(p:ARRAY{INT}, i:INT) is
        -- given a partition representing subradical
        -- sizes for a new radical, generate all such
        -- new radicals and append them to the radical list.
        -- enum_rad_tuple takes a partition and also
        -- a "continuation", a function to be applied
```

```
        -- to each new possible radical
        enum_rad_tuples(p, #ROUT(make_and_append_rad(_, i)));
    end;

    make_and_append_rad(triple:ARRAY{$RADICAL}, i:INT) is
        -- make a radical from the array of subradicals and
        -- append it to the appropriate radical list list.
        -- i is the number of carbon atoms in the radical
        -- that also determines the list in the array of
        -- radical list radicals.

        radicals[i] := radicals[i].push(#CARBON_RADICAL(triple));
    end;

    --------------------------------------------------------

    gen_ccps_of_size(size:INT) is
        -- generate all CCPs of a specified size
        enum_partitions_for_ccps(size-1,4,0,(size-1)/2, size);
    end;

    enum_partitions_for_ccps(sum:INT, number:INT, min:INT, max:INT, i:INT) is
        -- generates partitions representing radicals of a CCP
        -- and then actually creates new CCP molecules  based
        -- on these partitions
        loop
            enum_rads_for_ccps(enum_partitions!(sum, number, min, max), i);
        end;
    end;

    enum_rads_for_ccps(p:ARRAY{INT}, i:INT) is
        -- given a partition representing CCP radical
        -- sizes for new CCP molecules, generate all such
        -- new molecules and append them to the list of molecules.
        -- enum_rad_tuple takes a partition and also
        -- a "continuation", a function to be applied
        -- to each new possible radical
        enum_rad_tuples(p, #ROUT(make_and_append_ccp(_, i)));
    end;

    make_and_append_ccp(quad:ARRAY{$RADICAL}, i:INT) is
        -- make a CCP from the array of subradicals and
        -- append it to the appropriate list
        molecules[i] := molecules[i].push(#CCP_MOLECULE(quad));
    end;

    --------------------------------------------------------

    gen_bcps_of_size(size:INT) is
        -- generate all BCPs of a specified size
        enum_partitions_for_bcps(size, 2, size/2, size/2, size);
    end;

    enum_partitions_for_bcps(sum:INT, number:INT, min:INT, max:INT, i:INT) is
        -- generates partitions representing radicals of a BCP
        -- and then actually creates new BCP molecules  based
        -- on these partitions

        loop
            enum_rads_for_bcps(enum_partitions!(sum, number, min, max), i);
        end;
    end;

    enum_rads_for_bcps(p:ARRAY{INT}, i:INT) is
```

```
        -- given a partition representing BCP radical
        -- sizes for new BCP molecules, generate all such
        -- new molecules and append them to the list of molecules.
        -- enum_rad_tuple takes a partition and also
        -- a "continuation", a function to be applied
        -- to each new possible radical
           enum_rad_tuples(p, #ROUT(make_and_append_bcp(_, i)));
    end;

    make_and_append_bcp(pair:ARRAY{$RADICAL}, i:INT) is
        -- make a BCP from a pair of radicals and
        -- append it to the appropriate list
        molecules[i] := molecules[i].push(#BCP_MOLECULE(pair));
    end;

    --------------------------------------------------------

    enum_rad_tuples(p:ARRAY{INT}, apply_to_each: ROUT{ARRAY{$RADICAL}}) is
        -- enumarates all possible radical tuples with size
        -- represented by a partition p. Bound routine
        -- apply_to_each is applied to each legal radical tuple.
        radical_tuple:ARRAY{$RADICAL};

        radical_tuple := #(p.size);
        recursively_enumerate(p, apply_to_each, radical_tuple, 0,
             radicals[p[1]], p[1]);
    end;

    recursively_enumerate(p:ARRAY{INT},apply_to_each:ROUT{ARRAY{$RADICAL}},
         radical_tuple:ARRAY{$RADICAL}, level:INT,
         remainder:FLIST{$RADICAL}, prev_element:INT) is
             -- the inner recursive procedure for radical tuple enumaration.

        levels_radical_list:FLIST{$RADICAL};

        if p[level] = prev_element then
            levels_radical_list := remainder;
        else
            levels_radical_list := radicals[p[level]];
        end;

        loop
            radical_tuple[level] := levels_radical_list.elt!;
            if level < p.size-1 then
                recursively_enumerate(p, apply_to_each, radical_tuple,
                     level+1, levels_radical_list, p[level]);
            else
                apply_to_each.call( radical_tuple );
            end;
        end;
    end;

    --------------------------------------------------------

    create(n:INT):SAME is
        -- Create all paraffin isomers of size up to n

        tmp1, tmp2 : ARRAY{INT};

        r ::= new;
        r.radicals := #(n/2+1);
        r.molecules := #(n+1);

        -- initialize a list of radicals
```

```
        -- hydrogen is a radical of size 0 (no C atoms)
        r.radicals[0] := r.radicals[0].push(#HYDROGEN_RADICAL);

        loop
            i ::= 1.upto!(n/2);
            r.gen_radicals_of_size(i);
        end;

        -- generate paraffin molecules
        loop
            i ::= 1.upto!(n);
            -- CCPs are generated for both even and odd
            -- molecule sizes
            r.gen_ccps_of_size(i);
            if (i%2 = 0) then
                -- even size molecule: need bond centered paraffins as well
                r.gen_bcps_of_size(i);
            end;
        end;
        return r;
    end;

end;  -- class PARAFFINS

----------------------------------------------------------------------------
class MAIN is
    main(argv : ARRAY{STR}) is
        i, size : INT;
        p : $PARAFFINS;


        if argv.size /= 3 then
            #OUT + "usage: paraffins <-s for serial, -p for parallel> <number of carbon it
oms>\n";
            return;
        end;

        size := #(argv[2]);

        if argv[1] = "-p" then
            #OUT + "Parallel Version\n";
            p := #PAR_PARAFFINS(size);
        else
            #OUT + "Serial Vesrion\n";
            p := #PARAFFINS(size);
        end;

        p.print;
    end;
end;
```