ACRONYM:  
A HIERARCHICAL TREE AND FOREST GROWTH MODEL FRAMEWORK

by

Andrew P. Robinson

and

Alan R. Ek.

Staff Paper Series No. 127

DEPARTMENT OF FOREST RESOURCES

College of Natural Resources
and
Minnesota Agricultural Experiment Station
University of Minnesota
St. Paul, Minnesota
For more information about the Department of Forest Resources and its teaching, research and outreach programs contact the department head at:

Department of Forest Resources  
University of Minnesota  
115 Green Hall  
1530 Cleveland Avenue North  
St. Paul, MN 55108  
Ph: 612-624-3400  
FAX: 612-625-5212  
Email: fr@forestry.umn.edu

The University of Minnesota is committed to the policy that all persons shall have equal access to its programs, facilities, and employment without regard to race, color, creed, national origin, sex, age, marital status, public assistance status, veteran status, or sexual orientation.
ACRONYM:
A HIERARCHICAL TREE AND FOREST GROWTH MODEL FRAMEWORK

by

Andrew P. Robinson

and

Alan R. Ek.

MARCH 1998

Staff Paper Series No. 127

1 Research supported by the College of Natural Resources and the Minnesota Agricultural Experiment Station, University of Minnesota, St. Paul, the McIntire-Stennis Cooperative Forestry Research Program, and the USDA Small Business Innovation Research Program. Published as MAES Paper no. 984420127 of the Minnesota Agricultural Experiment Station.

The authors are Research assistant and Professor, Department of Forest Resources, Univ. of Minnesota, St. Paul.
ABSTRACT

ACRONYM is an extensible tree-level forest growth and dynamics model framework for the synthesis and assessment of models and submodels of forest ecosystem processes. The framework is designed for research in forest ecology and silviculture, development and testing of silvicultural guidelines, forest inventory updates, and long-term projections of forest and landscape dynamics.

The framework accepts modules (submodels) of forest regeneration, tree growth, mortality, silviculture, harvesting, and natural disturbance, each of which may operate at different spatial or temporal scales and levels of resolution. Component modules can range from empirical to process based, as understanding and data permit. Modules can be interchanged with a minimum of reprogramming to allow for comparison of assumptions about processes, different data input protocols, and project-specific reporting requirements.

INTRODUCTION

Individual tree-based forest growth models have been under development since the 1960s, and many variants have been reported in the literature. In that time, several dominant modeling themes have become established (see e.g., Munro 1974). Most forest models operate at either stand or tree level, and output typically includes some stand-level summary statistics and for tree-level models, a tree list on a study plot. These models can usually be divided into components which handle different aspects of tree or forest processes, such as growth, mortality, etc. Finally, these models are either spatial (distance dependent) or aspatial (distance independent). Spatial models consider explicit locations of trees on study plots.

Given these clear modeling traditions it seemed reasonable that there might be a protocol or perhaps even a modeling framework, to deal quickly and efficiently with the common model needs and operations. Such a framework might be used to specify and implement different model components quickly and easily without having to worry about the accounting required. After a general search for such a framework, none was found, although much support was expressed for the concept. Accordingly, this forest growth model simulation framework, ACRONYM, was written.

ACRONYM is an extensible tree-level forest growth and dynamics model framework for the synthesis and assessment of models and submodels of forest ecosystem processes. The framework is designed for research in forest ecology and silviculture, development and testing of silvicultural guidelines, forest inventory updates, and short- or long-term projections of forest and landscape dynamics.

The framework accepts modules (submodels) of forest regeneration, tree growth, mortality, silviculture, harvesting, and natural disturbance, each of which may operate at
different spatial or temporal scales and levels of resolution. Landscape processes are incorporated as plot size increases and spatial and temporal scales are broadened. Component modules can range from empirical to process based, as understanding and data permit. Modules can be interchanged with a minimum of reprogramming to allow for the comparison of assumptions about processes, different data input protocols, and project-specific reporting requirements.

The ACRONYM framework can be applied to both research and management problems. The principle underlying its usage is information sharing, and an agreed protocol for modeling to permit the greatest possible generality. It also has potential as an educational tool in management, modeling or silviculture, with applications encompassing a wide range of silvicultural scenarios.

The main advantage of ACRONYM is its extensibility, which follows from an open programming framework and modularity. This means that with a minimum of programming investment, extant models can be placed in the framework and compared with one another quickly and efficiently. A recent application of the carbon-balance model in Mäkelä (1997) within the framework took two person days to code and debug. The tree definition for this model is included in appendix 3.

ACRONYM is also fast. For example, in generating the initial conditions, and applying empirical growth models for four growth steps to 250,000 trees, sorting the trees at each step, creating stand tables at the beginning and end of the run, and writing the entire tree list to a file took 115 seconds on a 200 MHz Pentium with 64 megabytes of RAM running Windows 95. The application of the annual growth carbon-balance model in Mäkelä (1997) saw processing speeds of up to 850 tree years per second on the same machine.

Potential users require an understanding of programming concepts, and some familiarity with C or C++. It is anticipated that reasonably experienced programmers should have no trouble applying their own models to the framework. ACRONYM currently runs only on Windows 95, although extensions to UNIX, OS/2 and Macintosh platforms are possible.

The objective of this paper is to introduce the central concepts for ACRONYM, including the philosophy underpinning the programming style and language choice. A glossary of the main computing terms is included in appendix 1. This report also covers the main aspects of object-oriented programming as it applies to ACRONYM. The model is then introduced in the context of forest ecosystem processes, with its current and possible applications, and requirements for model operation. Finally, directions for future development are discussed.
PROGRAMMING CONSIDERATIONS

Modularity

Complex models are required to understand and predict the behavior of complex systems. Such models are frequently constructed from simpler models, or modules. For example, most individual-tree forest growth models can be considered to be made up of modules of individual tree growth, mortality, and regeneration, etc.

Modules can in turn be made of simpler modules. An individual tree growth prediction module might consist of a maximum potential growth module adjusted by a growth modifier module. The maximum potential growth module might be based on a photosynthesis module, which in turn might rely upon temperature, atmosphere and leaf modules.

Each module can be identified as a function or collection of functions and a set of parameters, typically estimated from experiments, or from literature review. Both the functions and the parameter estimates are typically determined by local conditions and requirements, data availability, and the contemporary statistical tools and ecophysiological theories.

Thus, a model of a complex system can be made of layers of modules, the depth of the layers depending on requirements and resources. The depth need not be uniform within a model. Identifying these layers and the primitives from which they are constructed is the first step toward understanding such models.

Modules communicate with one another through their interface, the sum total of their inputs and outputs. Considering forest growth models, the interface can often be easily standardized. For example, some mortality modules require recent growth as an input and return the status of the tree as output: alive or dead.

This standardization leads to the ideal situation for model flexibility and extensibility. It allows the user to pick and choose between a number of different modules, each with a different set of assumptions, and to be able to insert these into a model as appropriate (see e.g., Reynolds and Acock 1997).

However, by analogy to linear modeling, modules might be considered to have both main effects and interactions. That is, modules will not only affect the overall model prediction but will also likely change the relationship between other modules and the model output. For example, interactions in the form of negative correlations might be expected in modules which have been fit simultaneously as part of a broader model. This possibility of interaction mitigates against the easy exchange of modules, and care is required. Consequently, module-level diagnostics are being developed as part of the larger project in which ACRONYM is embedded.
Object-Oriented Programming

Although object-oriented programming (OOP) has been acknowledged as useful (Lorenz et al. 1989, Jeffers 1991) and implemented at the within-tree level (Salminen et al. 1994), it has yet to be explored as a modeling paradigm at the individual tree level. As far as we are aware, ACRONYM is the first individual tree-level simulation framework written using OOP C++.

According to Eckel (1995), three essential aspects characterize true OOP: data abstraction, inheritance, and polymorphism. Of these three, ACRONYM relies heavily on the first and less on the second. The principle issues within a forest growth model are very simple in computing terms, and focus more on putting the right things in the right places than relying on elegant code manipulation.

Data abstraction is the creation of new data types (e.g., integer, floating point, tree), the gathering together of the distinct data types into collections, and the manipulation of these collections as objects. This feature permits a precise mapping of a computer model to the real-world problem it is designed to address. This in turn facilitates debugging, model assessment, and extension.

Classes and Objects

The essence of OOP is an important paradigm shift from structured programming. For example, instead of coding tree variables in arrays of similar data type (e.g., an array of tree heights, where the height of the first tree is the first element), OOP demands that all the information about each tree be kept in a single data structure - an object.

An object is like an array except the data types within it can differ: an object representing a tree may well comprise several floating-point numerals (e.g., height, diameter at breast height [dbh]), a character vector (species name) and a boolean variable (alive/dead). Objects are created (a process known as "instantiation") from classes, which dictate the attributes (variables) that the objects have, and the processes (functions) which can be applied to the object.

In OOP, objects are defined, or declared, by classes. Classes consist of attributes (e.g., "Height" of class "Tree") and functions (e.g., "Grow" of class "Tree"). So when a class is defined, a new data type is created, with values and processes defined explicitly and uniquely by the coding.

Access to and responsibility for these attributes and functions is controlled by splitting each class into sections; public, private or protected. These labels determine which classes in the program create objects which are permitted to change the values of the attributes of objects of the class in question. Private attributes can only be changed by the object itself. Protected attributes can be changed or accessed by objects of any descendent classes as well. Public attributes and processes can be accessed or changed by any object in the program.
Consider a tree, an instance (object) of the Tree class. An example of the tree being referenced by itself is when the stand sends it a message causing it to grow, that is, to increase its height according to its growth function. An example of being referenced outside the class is the construction of a height-age curve using tree data; it is necessary to see the height (and the age) of each tree but not to alter it. This protocol ensures that the controlling of height always lies with the Tree object, but objects from any other class can observe it. This facility is mostly important for constructing libraries of objects.

It is worth noting that at this point, nothing has been described which could not be implemented in a structured language like FORTRAN 77 (Meissner and Organick 1974). However, the development of these principles is much more fluent in a language specifically designed around them, such as C++.

**Language**

The C++ language was recommended by Lemmon and Chuk (1997) for biological modeling for the following reasons:

1. ANSI standards are being established, so portability will be well defined even if not trivial;
2. many compilers are available for several different platforms;
3. it has a powerful numeric processor;
4. it is well known and has many development tools; and
5. it supports both object-oriented and non object-oriented programming.

The C++ language has already been applied to forestry-related problems. Salminen et al. (1994) described the application of OOP and C++ to the problem of modeling the growth of a single tree for their model LIGNUM. Koesmarno et al. (1994) applied OOP principles and C++ to modeling the change of growth and size-class distribution. Cescatti (1997a,b) programmed a model of heterogeneous crown effects on below-canopy light distribution in C++. Host et al. (1996) translated ECOPHYS from BASIC and C to object-oriented C++ to permit temporal and spatial scaling.

**ACRONYM**

ACRONYM synthesizes modules expressing different forest ecosystem processes at different temporal and spatial scales simultaneously. This allows exploration of complex model interaction and information sharing. The framework is a collection of classes, each of which represents some aspect of the problem of forest growth simulation. These classes can be easily supplemented by further classes to extend the capabilities of the model. The simplest manifestation is described here, and more intricate applications described later in the paper.
Protocol

In ACRONYM, the following declaration protocol has been adopted for clarity.

1) Class names always begin with capitals, objects of classes with lower-case; thus the object "tree" is an instance of class "Tree".

2) Any name which consists of more than one word will have the second and any subsequent words capitalized, e.g., numberOfBoles.

3) Attributes within classes always begin with capital letters, and the functions by which they are accessed outside the class with lower-case letters, thus the height of object "tree" is called "Height", but is referenced from outside the tree object by the function call tree.height(). So the attribute Height is private and the function height() is public.

Stand

The Stand class controls the interaction of the forest, and acts as a container for all the forest objects. All processes which occur to collections of trees, and attributes which apply to collections of trees, are controlled and assessed by the Stand. For example, Stand will send a growth signal to each of the trees in its tree list, and keep a running tally of the changes of such statistics as mean dominant height, basal area, and so on. These are recorded to an output file as the run progresses.

If a stand-level process, for example a disturbance, were required, a Disturbance object would instruct the Stand object as to the location and nature of disturbance, and the Stand object would interpret this and apply the appropriate treatment to the trees in its tree list, as well as any other relevant objects.

This allows the potential for broad-scale processes to be included in the framework. The size of the stand is limited only by available computer memory, and although there is currently only one stand for each model run, it would not be difficult to generalize the framework to create and run numerous stands. These might accept parameters of large-scale processes from, e.g., a geographic information system.

ACRONYM uses dynamic memory allocation, so the only limit to the number of forest objects that can be considered is imposed by how much memory is available. Test runs have successfully involved up to 250,000 unique objects. The forest objects are stored in the Standard Template Library "list" container (Eckel 1995).

ForestBody

The basic class is the forest body, which has among its attributes for example "location", "species", "height", and "diameter". Other attributes and functions can be added for specific inventory or monitoring systems, or particular analyses.
ForestBody is a base abstract class, as there is no instance of "ForestBody" within the model; it is only used as a general class from which other more specific classes are developed. In the simplest version of ACRONYM there are two types or subclasses of forest body: trees (alive) and snags (dead). Both these classes inherit the ForestBody attributes and add some of their own. Other classes, such as Regen (which could be either a seedling or a sprout), Stump, and Debris, are also available.

Tree

The Tree class describes a tree object in terms of its attributes and functions. An example of a function unique to the tree class is Tree::Grow() which instructs the tree object to change its dimensional attributes, such as height, diameter, etc., according to whatever growth function the programmer has coded. A sample growth function is included in appendix 3.

Trees may also reproduce, using simple or complicated functions, depending on the needs of the model. Regeneration components might be taken from the literature (e.g., Ek et al. 1997), or FOREST (Ek and Monserud 1974), SORTIE (Pacala et al. 1993, 1996), or any other forest ecosystem growth simulator, depending on data availability for calibration.

The growth function can include specifications for non disturbance based mortality. These can be either stochastic or deterministic. If these conditions are met by a tree during its growth phase, it will return a "dead" signal to the stand, which will delete it and replace it with a snag object of the same location and dimensions.

Snag

The Snag class describes a snag object, in terms of its attributes and functions. In the simplest version of ACRONYM snags are like placeholders, to maintain a record of the mortality dynamics of stands.

For a broader ecosystem application, an example of a function which would be unique to snags is Snag::Decay() which might cause the snag object to release nutrients to a local or global soil object. This decay function could be derived from local knowledge or drawn from established models (e.g., Dewar 1991 or Landsberg and Gower 1997).

RunTimeParameters

When ACRONYM is run from the command line (the only interface at this time) a number of options can be chosen for the run. The parameters which relate to the mechanics of the program (namely, verbose and/or diagnostic output, tree list recorded to file on completion, etc.) are chosen in the command line. Those items relevant to the model (different stand-level options, parameters, output requirements and arrangements of starting conditions) are in the input files.
The program is run with the names of the input and output files. The following protocol has been chosen: the input and output files names are constructed of the run name followed by ".in.txt" and ".out.txt", respectively. Thus, for a run named "fortest", the program is run from a command line interface by:

    acronym fortest

ACRONYM will look for "fortest.in.txt" in the same directory, will direct the model run output to "fortest.out.txt", and should the tree list option be selected, will output whatever tree-level information is required in whatever format has been programmed to "fortest.tree.out.txt".

The RunTimeParameters class, like the PseudoRandomNumberGenerator class, poses an implementation problem. Without going in to detail, both classes are accessed by friend functions which call a pointer to the instance of the class, as otherwise there would be a problem with object persistence. This approach was easy to implement, and easy to upgrade should a new approach suggest itself.

**InputControl**

Dissemination of the information from the input file is centralized in the InputControl class. This class reads and interprets the input file and passes the information to each class which requires it. This protocol increases the overhead slightly as the input data will be recorded in two places, but it centralizes the interface for the programmer.

There are two main reasons this class might need to be replaced for any application: to change the nature of the information passed to the program (e.g., add new variables), and to change the format of the input (to read automatically according to an inventory protocol or database management system). In either case, allowing a single class to be responsible for reading and interpreting the data is advantageous, since it is the only place where input adjustments will need to be made.

A file-based tree list input protocol is also available for starting runs with a tree list from an inventory or a database.

**Timer**

The timer class controls the passing of time and reporting within the model run. The class permits different temporal layers by the simple facility of nested loops (for a similar approach see Luan et al. 1996).

**Coordinates**

To facilitate spatial considerations in the model, a specific location for each forest object has been included. The Coordinates class has been constructed to represent the location
of an object, which consists of an x coordinate, a y coordinate, an optional z coordinate for altitude, and a set of functions which can be applied to these coordinate pairs.

**PseudoRandomNumberGenerator**

An extensive review of the history of pseudo-random number generation (PRNG), the techniques available, and diagnostics for testing them can be found in Ripley (1987). The PRNG class used in ACRONYM comprises shift register pseudo-random number generator functions adopted from SORTIE (Pacala et al. 1993).

The code from SORTIE was wrapped in the class PseudoRandomNumberGenerator, and a pointer initialized to point to the sole instance of the class created in the model run. The functions within the class are then accessed by friend functions. In practical terms, this means that to obtain a uniform random number one calls the function uniform(); if boundaries other than (0, 1) are desired, say (a,b) then simply use uniform(a,b). Likewise, the normal() function behaves intuitively. Other distributions may be added as necessary.

**CONCLUSIONS AND FUTURE DIRECTIONS**

ACRONYM will be available on the World Wide Web with periodic updates. The framework is largely complete, but detail is evolving. With use, potential shortcomings will be identified and corrected. For example, several users have suggested that a graphical interface would be helpful for application of the model. The current version of ACRONYM deliberately has no graphics as the complications required to generate the graphics would outweigh some of the model's advantages. ACRONYM is designed to be as simple as is feasible while maintaining the essential features of a tree and forest dynamics simulator.

ACRONYM is intended to support researchers, managers, and educators in the start-up phase of modeling efforts. The flexibility and extensibility allows considerable code and data sharing for model development. For those who have legacy code in other languages, ACRONYM offers the advantage of simple integration of C code, and with the advent of products such as For_C by Cobalt Blue (Lightfoot 1996), even FORTRAN code may be translated to C++ and brought into the framework.

ACRONYM is in use as a research and education tool at the University of Minnesota. It has been implemented by researchers at METLA (the Finnish Forest Research Institute), and it is being explored for application by researchers at the University of Helsinki.

**ACKNOWLEDGMENTS**

We are indebted to many people for the ideas and encouragement they have contributed to this paper. We would especially like to acknowledge Tom Burk, Sauli Valkonen, Annikki Mäkelä, Pepe Hari, and Risto Sievänen for their encouragement and suggestions.
LITERATURE CITED


APPENDIX 1

Glossary of terms:

*Attribute.* - A variable of an object which refers to some state of the object. Thus, Height is an attribute of a tree object, that is, any object instantiated from the Tree class. If the computer asks a tree what its height is, e.g. by calling tree.Height, it will return the height attribute. Conceptually, an attribute is a noun.

*Class.* - A template by which objects are defined, that is, by which the computer is told what characteristics objects have. It collects and describes all of the functions and attributes that objects of the class will have. The Tree class describes the notion of 'tree' to the computer. When the computer is asked to create a tree or collection of trees in memory, it refers to the Tree class to see exactly what a tree is.

*Data abstraction.* - Creation of new data types, to make programming easier. Older programming language relied on data types such as Integer, Floating point number, character, etc. to do everything. C++ permits the user to define their own data type for manipulation. These data types are called classes.

It is important to recognize that OOP really does not do anything that structured programming won't do. However it makes conceptualization, development, and debugging a great deal easier. For example, an array or list (q.v.) of tree objects can be very easily conceptualized and manipulated.

*Function.* - A body of code that describes anything an object might do. Thus, Grow is a function of a tree object, an object instantiated from the Tree class. If the computer tells a tree to grow, e.g. by calling tree.Grow(), it will change some of its attributes - perhaps Height and Diameter - by whatever algorithms have been coded into the function. Conceptually, a function is a verb.

*Inheritance.* - An advantage of OOP that allows efficient extension and development. Inheritance allows the programmer to define a new class as being an extension of an old one, that is, to have all the functions and attributes of the old class as well as some more.

A simple example will illustrate the point. In ACRONYM the base class is ForestBody, and the Tree and Snag classes both inherit attributes from that class. If it were desirable to create a function which determined the distance between forest objects, say

```c
    distance (object1, object2)
```

then we need only define that function within the ForestBody class, and it is automatically available for Trees and Snags.
**Instantiation.** - Refers to the creation of an object from a class, and is done by a class constructor. This includes locating and reserving a block of memory for the object, writing whatever starting attributes the object has to memory, and returning a pointer to the location in memory of the new object.

**List.** - A special kind of class which acts as a container for objects of classes. The list is analogous to an array, but more flexible, as it can grow or shrink in size (number of objects) dynamically at low cost. This means that the size does not need to be declared beforehand. However, elements cannot be located by an index. The list is a natural choice for a container of tree objects, as the number of trees is therefore unconstrained by programming considerations, and it is rare that only a particular tree would need to be located within the container. The list class comes from the Standard Template Library (Eckel 1995).

**Object.** - A particular instance of a class. Therefore, if we were to consider Integer to be a class, the number 3 is an object of the class Integer. Similarly, considering the Tree class, the objects are trees in memory which have been instantiated according to the Tree specifications.

**Pointer.** - A variable which contains the location in memory of a value instead of the value itself.
APPENDIX 2

ACRONYM Download and usage instructions:

The latest ACRONYM source code, a compiled version for Windows 95, sample inputs and outputs and this document in various formats are bundled into a zipped archive at the following location:

http://prism.fr.umn.edu/~arobinso/Public/ACRONYM/acronym.zip

Please note that the address is case sensitive.

We strongly recommend users try to compile the code as is before making any alterations; it contains simplistic minimalist submodels. The code compiles under Borland C++ V 5.01. After compiling and running the code, and comparing the output with the sample run included, the framework can be used to embed your own models. Should you require assistance in compiling for a different platform please do not hesitate to contact us.

At the time of writing, there is only a draft standard of C++, and it is likely that this will not be a definitive version. ACRONYM relies on the Standard Template Library which will most likely be included in the standard. Although we do not believe major changes will be necessary, we will nonetheless release a new version of ACRONYM when the standard is published.

The principle underlying the development of ACRONYM is to promote the greatest possible sharing of modeling concepts, code, and data. Therefore, if you add any functionality to the framework that does not rely on proprietary information or data we invite you to submit it to the ACRONYM archive, where it will be integrated with proper acknowledgment. Similarly, if you write growth functions for the Tree class we encourage you to submit these to the archive.

Should this seem appropriate or should you have any questions on implementation or recommendations for further development please do not hesitate to contact us at the address on the cover of this document, or

arobinso@forestry.umn.edu

ack@forestry.umn.edu
APPENDIX 3

Sample - the tree class definition, including header file and code from Mäkelä (1997).

/* Tree.h

*/

#ifndef TREE
#define TREE

#include "acronym.h"
#include "forestbody.h"
#include "InputControl.h"

// CLASS DEFINITION

/** The tree class describes an object which has a location, a species label, a height, and a collection of other state variables. It is capable of growing, and being killed. */

class Tree : public ForestBody // inherits the ForestBody attributes
{
private:
    // ATTRIBUTES See Appendix 1 of Mäkelä (1997)
    /// A measure of crown mass
    float FolageWeight;
    /// A measure of fine root mass
    float FineRootWeight;
    /// A measure of sapwood mass in the stem
    float StemSapwoodWeight;
    /// A measure of sapwood mass in the branches
    float BranchSapwoodWeight;
    /// A measure of sapwood mass in the transport roots
    float RootSapwoodWeight;
    /// Length of the crown in metres
    float CrownLength;
    /// Length of the clear bole in metres
    float BoileLength;
    /// Length of the branches in metres
    float BranchLength;
    /// Length of the roots in metres
    float RootLength;
    /// Stem Sapwood area
    float StemSapwoodArea;
    /// Branch Sapwood area
    float BranchSapwoodArea;
    /// Root Sapwood area
    float RootSapwoodArea;
    /// Crown surface area
    float CrownSurfaceArea;
    /// Stem Heartwood volume
    float StemHeartwoodWeight;
    // Pointer to species-level information
    InputControl* SpeciesData;

public:
    // CONSTRUCTORS
    Tree(); // Null constructor, we don't use it but <list> does
    Tree(Coordinates, int, InputControl &);
// REPORTERS
float treeBasalArea() const { return pow(Diameter, 2) / 4 * PI; }
float height() const { return Height; }
float foliageWeight() const { return FoliageWeight; }
float fineRootWeight() const { return FineRootWeight; }
float branchSapwoodWeight() const { return BranchSapwoodWeight; }
float rootSapwoodWeight() const { return RootSapwoodWeight; }
float crownLength() const { return CrownLength; }
float boleLength() const { return BoleLength; }
float branchLength() const { return BranchLength; }
float rootLength() const { return RootLength; }
float stemSapwoodArea() const { return StemSapwoodArea; }
float branchSapwoodArea() const { return BranchSapwoodArea; }
float rootSapwoodArea() const { return RootSapwoodArea; }
float crownSurfaceArea() const { return CrownSurfaceArea; }
float stemHeartwoodWeight() const { return StemHeartwoodWeight; }
float volume() const {
    return (StemHeartwoodWeight + StemSapwoodWeight) / rho_s();
}

// DUMP TREE TO SINGLE LINE
void treeHeader(fstream &outFile) {
    outFile << setw(10) << "Foliage Weight" << setw(10) << "Foliage Weight"
    << setw(10) << "Bole Length" << setw(10) << "Bole Length"
    << setw(10) << "Crown Length" << setw(10) << "Crown Length"
    << setw(10) << "Height" << setw(10) << "Height"
    << setw(10) << "Volume" << setw(8) << "Species"
    << endl;
}

friend ostream & operator << (ostream & out, const Tree & t) {
    return out << setw(10) << setprecision(4) << t.foliageWeight()
    << setw(10) << setprecision(4) << t.boleLength()
    << setw(10) << setprecision(4) << t.crownLength()
    << setw(10) << setprecision(4) << t.height()
    << setw(10) << setprecision(4) << t.volume()
    << setw(8) << setprecision(4) << t.speciesID() << endl;
}

// PROCESS
bool grow(Tree & t, float, float, float); // instructs tree to grow, returns life status

// GROW PARAMETERS
float phi_s() const { return SpeciesData->speciesList(SpeciesID).parameters().phi_s(); }
float phi_c() const { return SpeciesData->speciesList(SpeciesID).parameters().phi_c(); }
float phi_b_prime() const { return SpeciesData->speciesList(SpeciesID).parameters().phi_b_prime(); }
float phi_t_prime() const { return SpeciesData->speciesList(SpeciesID).parameters().phi_t_prime(); }
float c_b() const { return SpeciesData->speciesList(SpeciesID).parameters().c_b(); }
float c_t() const { return SpeciesData->speciesList(SpeciesID).parameters().c_t(); }
float rho_s() const { return SpeciesData->speciesList(SpeciesID).parameters().rho_s(); }
float rho_b() const { return SpeciesData->speciesList(SpeciesID).parameters().rho_b(); }
float rho_t() const { return SpeciesData->speciesList(SpeciesID).parameters().rho_t(); }
float alpha_s() const { return SpeciesData->speciesList(SpeciesID).parameters().alpha_s(); }
float alpha_b() const { return SpeciesData->speciesList(SpeciesID).parameters().alpha_b(); }
float alpha_t() const { return SpeciesData->speciesList(SpeciesID).parameters().alpha_t(); }
float alpha_r() const { return SpeciesData->speciesList(SpeciesID).parameters().alpha_r(); }
float two_z() const { return SpeciesData->speciesList(SpeciesID).parameters().two_z(); }
float xi() const { return SpeciesData->speciesList(SpeciesID).parameters().xi(); }
float y() const { return SpeciesData->speciesList(SpeciesID).parameters().y(); }
float r_1() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().r_1(); }
float r_2() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().r_2(); }
float s_f() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().s_f(); }
float s_r() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().s_r(); }
float d_s_0() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().d_s_0(); }
float d_b_0() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().d_b_0(); }
float d_t_0() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().d_t_0(); }
float d_s_1() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().d_s_1(); }
float d_b_1() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().d_b_1(); }
float d_t_1() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().d_t_1(); }
float psi_s() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().psi_s(); }
float psi_c() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().psi_c(); }
float psi_b_prime() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().psi_b_prime(); }
float psi_t_prime() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().psi_t_prime(); }
float a_n() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().a_n(); }
float p_0() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().p_0(); }
float a_sigma() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().a_sigma(); }
float k() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().k(); }
float q() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().q(); }
float p() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().p(); }
float a_q() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().a_q(); }
float m_0() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().m_0(); }
float m_1() const
    { return SpeciesData->speciesList( SpeciesID ).parameters().m_1(); }
};

/**
 * This template is used by the sorting algorithm in stand for calculating
 * mean dominant height. It reports which of a pair of trees is the thinner.
 */

template <class Tree>
struct thinner : binary_function<Tree, Tree, bool> {
    bool operator() (const Tree& x, const Tree& y) const
    { return x.diameter() < y.diameter(); }
};

#endif // TREE
// Tree.cpp (Describes a tree object, attributes and functions)
#include "tree.h"

// CONSTRUCTORS

/** After declaring the attributes of the tree object, the processes which
 the tree is capable of can be declared. The constructor creates (instantiates)
a tree object from the class tree. The parameters are currently set at
simple defaults.

All references come from: Mäkelä, A. 1997. A carbon balance model of growth
and self-pruning in trees based on structural relationships. Forest Science
43(1): 7-24. */

inline Tree::Tree(Coordinates newLocation, int newSpeciesID, InputControl &inputs)
{ SpeciesID = newSpeciesID;
  SpeciesData = &inputs;
  Location = newLocation;

  // Conversions - see equations 6a-c and 12a-c
  float phi_t = phi_t_prime() * c_t();
  float phi_b = phi_b_prime() * c_b();
  float xi_prime = xi();

  // These starting conditions come from P. 14
  FoliageWeight = 0.1;
  BoleLength = 0;

  // Equations 5c, 5b, 5a
  CrownLength = pow((foliageWeight() / xi()), pow(two_z(), -1));
  RootLength = c_t() * (crownLength() + boleLength());
  BranchLength = c_b() * crownLength();

  // Common sense :-).
  Height = crownLength() + boleLength();

  // Equations 6a, 6b, 6c
  StemSapwoodWeight = rho_s() * alpha_s() * (phi_s() * boleLength() + phi_c() * crownLength()) * foliageWeight();
  BranchSapwoodWeight = rho_b() * alpha_b() * phi_b() * crownLength() * foliageWeight();
  RootSapwoodWeight = rho_t() * alpha_t() * phi_t * (boleLength() + crownLength()) * foliageWeight();

  // For volume
  StemHeartwoodWeight = 0;

  // Equation 3
  FineRootWeight = alpha_r() * foliageWeight();

  // Equation 2 (in three parts)
  StemSapwoodArea = alpha_s() * foliageWeight();
  BranchSapwoodArea = alpha_b() * foliageWeight();
  RootSapwoodArea = alpha_t() * foliageWeight();

  // Equation 4
  CrownSurfaceArea = pow((foliageWeight() / xi_prime), pow(two_z() / 2, -1));
}
inline bool Tree::grow( Tree &tree, 
float stocking, 
float leafAreaIndex, 
float crownCoverage ) 
{
    bool alive = true;

    // Conversions - see equations 6a-c and 12a-c
    float phi_t = tree.phi_t_prime()*tree.c_t();
    float phi_b = tree.phi_b_prime()*tree.c_b();
    float psi_t = tree.psi_t_prime()*tree.c_t();
    float psi_b = tree.psi_b_prime()*tree.c_b();
    float x_i_prime = tree.xi();

    // Equation 27 determines average tree photosynthesis
    float averagePhotosynthesis = tree.p_0() / stocking *
        (1 - exp(-tree.k() * tree.leafAreaIndex()));

    // Equation 29 calibrates the average photosynthesis by foliage weight
    float sigma_co = averagePhotosynthesis / tree.foliageWeight();

    // Equation 9a reduces photosynthesis by crown length, a proxy for age.
    float sigma_c = sigma_co*(1 - tree.a_sigma() * tree.crownLength());

    // Equations 24 a-f prepare to determine rf, specific foliage growth rate
    float b_s = tree.rho_s() * tree.alpha_s() * tree.phi_s() * tree.alpha_t() * phi_t;  // 24a
    float b_c = tree.rho_s() * tree.alpha_s() * tree.phi_c() + tree.rho_b() * tree.alpha_b() * phi_b + tree.rho_t() * tree.alpha_t() * phi_t;  // 24b

    // The article has an error in the following equation: r_b should be rho_b.
    float a_1 = tree.rho_s() * tree.alpha_s() + tree.phi_s() + tree.psi_c() * tree.d_s_0() + tree.rho_b() * tree.alpha_b() * phi_b + tree.rho_d() * tree.alpha_d() * psi_t + psi_t * tree.d_t_0();  // 24c
    float a_2 = tree.rho_s() * tree.alpha_s() + tree.psi_s() * tree.d_s_0() + tree.rho_t() * tree.alpha_t() * psi_t + tree.d_t_0();  // 24d
    float a_s = tree.rho_s() * tree.alpha_s() + tree.psi_s() * tree.d_s_1() + tree.rho_t() * tree.alpha_t() * psi_t + tree.d_t_1();  // 24e
    float a_c = tree.rho_s() * tree.alpha_s() + tree.psi_c() * tree.d_s_1() + tree.rho_b() * tree.alpha_b() * psi_b + tree.rho_d() * tree.alpha_d() * psi_t + tree.d_t_1();  // 24f

    // Equation 23 (in several parts for niceness) determines rf VIA u_s. See // "The Rate of Self-Pruning" p 13.
    float s = min(pow((tree.a_q() * crownCoverage), tree.q()), 1.0);  // 31
    float P_n = pow(tree.y(), -1) * (sigma_c - tree.r_1()) * (1 + tree.alpha_r()) - tree.r_2() * (b_s * tree.boleLength() + b_c * tree.crownLength());
    float r_0_upper = P_n - tree.s_f() - tree.alpha_r() * tree.s_r() - a_s * tree.boleLength() - a_c * tree.crownLength();
    float r_f_lower = 1 + tree.alpha_r() + b_s * tree.boleLength() + ((tree.two_z_0 + 1) / tree.two_z_0) * b_c * tree.crownLength();
    float u_s_multiplier = (a_1 + a_2 * tree.boleLength() / tree.crownLength());
// This value obtained by straightforward algebra from Equations 23, 25, 26.
float u_s = s * r_0_upper / u_s_multiplier;
float r_f = (r_0_upper - u_s_multiplier * u_s) / r_f_lower;
tree.FoliageWeight *= (1 + r_f); // Equation 14
// Equation 19

// Equations 5c, 5b, 5a  Updating dimensions

tree.CrownLength = pow(tree.FoliageWeight() / tree.xi(), pow(two_z(), -1));
tree.RootLength = tree.c_t() * (tree.crownLength() + tree.boleLength());
tree.BranchLength = tree.c_b() * tree.crownLength();
// Common sense :-).

tree.Height = tree.crownLength() + tree.boleLength();

// Equations 6a, 6b, 6c  Updating dimensions

tree.StemSapwoodWeight = tree.rho_s() * tree.alpha_s() *
    (tree.phi_s() * tree.boleLength() + tree.phi_c() * tree.crownLength())
    * tree.FoliageWeight();
tree.BranchSapwoodWeight =
    tree.rho_b() * tree.alpha_b() * phi_b * tree.crownLength() * tree.FoliageWeight();
tree.RootSapwoodWeight =
    tree.rho_t() * tree.alpha_t() * phi_t *
    (tree.boleLength() + tree.crownLength()) * tree.FoliageWeight();
// Equation 3

tree.FineRootWeight = tree.alpha_r() * tree.FoliageWeight();
// Equation 2 (in three parts)

tree.StemSapwoodArea = tree.alpha_s() * tree.FoliageWeight();
tree.BranchSapwoodArea = tree.alpha_b() * tree.FoliageWeight();
tree.RootSapwoodArea = tree.alpha_t() * tree.FoliageWeight();
// Equation 4

tree.CrownSurfaceArea =
    pow(tree.FoliageWeight() / xi_prime(), pow(two_z(), 2, -1));
// Equation 21a

tree.StemHeartwoodWeight +=
    tree.rho_s() * tree.alpha_s() *
    (tree.d_s_0() * tree.boleLength() / tree.crownLength() + tree.d_s_1() *
    tree.FoliageWeight())
    * (tree.psi_s() * tree.boleLength() + tree.psi_s() * tree.crownLength());

/* Mortality function would be faster if we did this first, but it's better
at the end to permit some measure of tree growth to enter the equation later for
comparisons purposes. */
float probabilityOfDeath = tree.m_0() + tree.m_1() * pow(crownCoverage, tree.p());
if (uniform() < probabilityOfDeath) alive = false;
return alive;
}