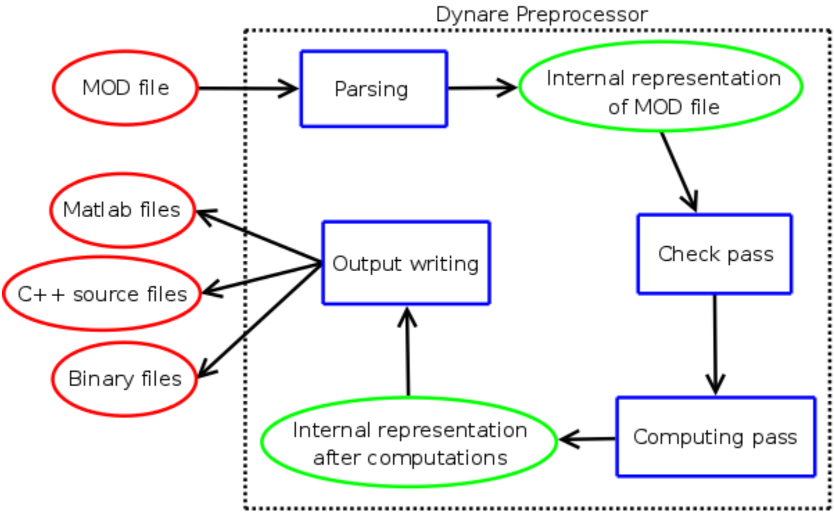
The Dynare Preprocessor

Sébastien Villemot

CEPREMAP

October 19, 2007

General overview



Outline

Outline

Object-oriented programming (OOP)

- Traditional way of programming: a program is a list of instructions (organized in functions) which manipulate data
- OOP is an alternative programming paradigm that uses objects and their interactions to design programs
- With OOP, programming becomes a kind of modelization: each object of the program should modelize a real world object, or a mathematical object (*e.g.* a matrix, an equation, a model...)
- Each object can be viewed as an independent little machine with a distinct role or responsibility
- Each object is capable of receiving messages, processing data, and sending messages to other objects
- Main advantage of OOP is modularity, which leads to greater reusability, flexibility and maintainability

Object-oriented programming (OOP)

- Traditional way of programming: a program is a list of instructions (organized in functions) which manipulate data
- OOP is an alternative programming paradigm that uses objects and their interactions to design programs
- With OOP, programming becomes a kind of modelization: each object of the program should modelize a real world object, or a mathematical object (*e.g.* a matrix, an equation, a model...)
- Each object can be viewed as an independent little machine with a distinct role or responsibility
- Each object is capable of receiving messages, processing data, and sending messages to other objects
- Main advantage of OOP is modularity, which leads to greater reusability, flexibility and maintainability

Object-oriented programming (OOP)

- Traditional way of programming: a program is a list of instructions (organized in functions) which manipulate data
- OOP is an alternative programming paradigm that uses objects and their interactions to design programs
- With OOP, programming becomes a kind of modelization: each object of the program should modelize a real world object, or a mathematical object (*e.g.* a matrix, an equation, a model...)
- Each object can be viewed as an independent little machine with a distinct role or responsibility
- Each object is capable of receiving messages, processing data, and sending messages to other objects
- Main advantage of OOP is modularity, which leads to greater reusability, flexibility and maintainability

- An object is the bundle of:
  - several variables (called its attributes), which modelize the characteristics (or the state) of the object
  - several functions (called its methods) which operate on the attributes, and which modelize the behaviour of the object (the actions it can perform)
- Example: suppose we want to modelize a coffee machine
  - The coffee machine (in real life) is a box, with an internal counter for the credit balance, a slot to put coins in, and a button to get a coffee
  - The corresponding object will have one attribute (the current credit balance) and two methods (one which modelizes the introduction of money, and the other the making of a coffee)

Object
Definition and example

- An object is the bundle of:
  - several variables (called its attributes), which modelize the characteristics (or the state) of the object
  - several functions (called its methods) which operate on the attributes, and which modelize the behaviour of the object (the actions it can perform)
- Example: suppose we want to modelize a coffee machine
  - The coffee machine (in real life) is a box, with an internal counter for the credit balance, a slot to put coins in, and a button to get a coffee
  - The corresponding object will have one attribute (the current credit balance) and two methods (one which modelizes the introduction of money, and the other the making of a coffee)

A coffee machine

Class definition

C++ header file (`CoffeeMachine.hh`)

```
class CoffeeMachine {
public:
  int credit;
  CoffeeMachine();
  void put_coin(int coin_value);
  void get_coffee();
};
```

- A class is a template (or a blueprint) of an object
- Collectively, the attributes and methods defined by a class are called members
- A class definition creates a new type (`CoffeeMachine`) that can be used like other C++ types (*e.g.* `int`, `string`, ...)
- In C++, class definitions are put in header files (`.hh` extension)

A coffee machine

Method bodies

C++ source file (`CoffeeMachine.cc`)

```
void CoffeeMachine::put_coin(int coin_value)
{
  credit += coin_value;
  cout << "Credit is now " << credit << endl;
}

void CoffeeMachine::get_coffee()
{
  if (credit == 0)
    cout << "No credit!" << endl;
  else {
      credit--;
      cout << "Your coffee is ready, credit is now " << credit << endl;
  }
}
```

- Methods can refer to other members (here the two methods modify the `credit` attribute)
- Method bodies are put in source files (`.cc` extension)

Constructors and destructors

- In our class header, there is a special method called `CoffeeMachine()` (same name than the class)
- It is a constructor: called when the object is created, used to initalize the attributes of the class

C++ source file (`CoffeeMachine.cc`, continued)

```
CoffeeMachine::CoffeeMachine()
{
  credit = 0;
}
```

- It is possible to create constructors with arguments
- It is also possible to define a destructor (method name is the class name prepended by a tilde, like `~CoffeeMachine`): called when the object is destroyed, used to do cleaning tasks (*e.g.* freeing memory)

Instantiation and method invocation

Program main function

```
#include "CoffeeMachine.hh"

int main()
{
  CoffeeMachine A, B;

  A.put_coin(2);
  A.get_coffee();

  B.put_coin(1);
  B.get_coffee();
  B.get_coffee();
}
```

- Creates two machines: at the end, `A` has 1 credit, `B` has no credit and refused last coffee
- `A` and `B` are called instances of class `CoffeeMachine`
- Methods are invoked by appending a dot and the method name to the instance variable name

Dynamic instantiation with `new`

Program main function

```
#include "CoffeeMachine.hh"

void main()
{
  CoffeeMachine *A;

  A = new CoffeeMachine();

  A->put_coin(2);
  A->get_coffee();

  delete A;
}
```

- Here `A` is a pointer to an instance of class `CoffeeMachine`
- Dynamic creation of instances is done with `new`, dynamic deletion with `delete` (analogous to `malloc` and `free`)
- Since `A` is a pointer, methods are called with `->` instead of a dot

Access modifiers

- In our coffee machine example, all attributes and methods were marked as `public`
- Means that those attributes and methods can be accessed from anywhere in the program
- Here, one can gain credit without putting money in the machine, with something like `A.credit = 1000;`
- The solution is to declare it private: such members can only be accessed from methods within the class

C++ header file (`CoffeeMachine.hh`)

```
class CoffeeMachine {
private:
  int credit;
public:
  CoffeeMachine();
  void put_coin(int coin_value);
  void get_coffee();
};
```

Interface

- The public members of a class form its interface: they describe how the class interacts with its environment
- Seen from outside, an object is a “black box”, receiving and sending messages through its interface
- Particular attention should be given to the interface design: an external programmer should be able to work with an class by only studying its interface, but not its internals
- A good design pratice is to limit the set of public members to the strict minimum:
    - enhances code understandability by making clear the interface
    - limits the risk that an internal change in the object requires a change in the rest of the program: loose coupling
    - prevents the disruption of the coherence of the object by an external action: principle of isolation

Why isolation is important

- Consider a class `Circle` with the following attributes:
  - coordinates of the center
  - radius
  - surface
- If all members are public, it is possible to modify the radius but not the surface, therefore disrupting internal coherence
- The solution is to make radius and surface private, and to create a public method `changeRadius` which modifies both simultaneously
- *Conclusion:* Creating a clear interface and isolating the rest diminishes the risk of introducing bugs

Inheritance (1/2)

Matrices and positive definite matrices

```
class Matrix
{
protected:
  int height, width;
  double[] elements;
public:
  Matrix(int n, int p,
         double[] e);
  virtual ~Matrix();
  double det();
};
```

```
class PositDefMatrix : public Matrix
{
public:
  PositDefMatrix(int n, int p,
                 double[] e);
  Matrix cholesky();
};
```

- `PositDefMatrix` is a subclass (or derived class) of `Matrix`
- Conversely `Matrix` is the superclass of `PositDefMatrix`

Inheritance (2/2)

- `PositDefMatrix` inherits `width`, `height`, `elements` and `det` from `Matrix`
- Method `cholesky` can be called on an instance of `PositDefMatrix`, but not of `Matrix`
- The keyword `protected` means: public for subclasses, but private for other classes
- Type casts are legal when going upward in the derivation tree:
  - a pointer to `PositDefMatrix` can be safely cast to a `Matrix*`
  - the converse is faulty and leads to unpredictable results

Constructors and destructors (bis)

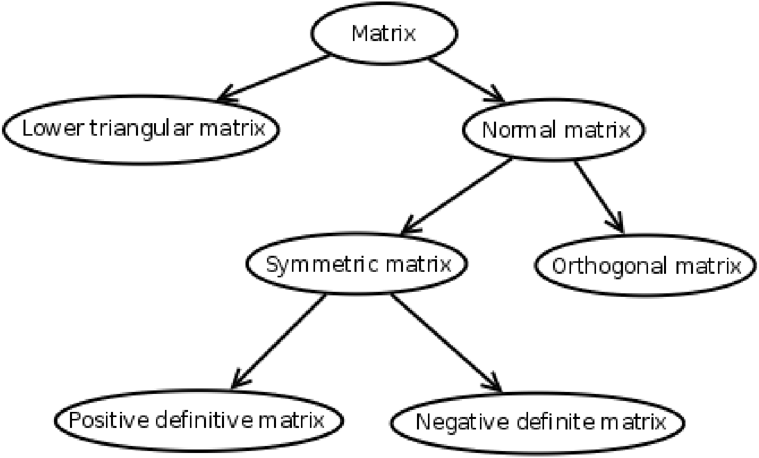
C++ code snippet

```
Matrix::Matrix(int n, int p, double[] e) : height(n), width(p)
{
  elements = new double[n*p];
  memcpy(elements, e, n*p*sizeof(double));
}

Matrix::~Matrix()
{
  delete[] elements;
}

PositDefMatrix::PositDefMatrix(int n, int p, double[] e) :
  Matrix(n, p, e)
{
  // Check that matrix is really positive definite
}
```

- Constructor of `PositDefMatrix` calls constructor of `Matrix`
- Note the abbreviated syntax with colon

Possible derivation tree for real matrices

Arrow means *...is a subclass of...*



Polymorphism (1/3)

- In previous example, determinant computation method uses the same algorithm for both classes
- But for positive definite matrices, a faster algorithm exists (using the cholesky)
- Polymorphism offers an elegant solution:
  - declare `det` as a virtual method in class `Matrix`
  - override it in `PositDefMatrix`, and provide the corresponding implementation
- When method `det` will be invoked, the correct implementation will be selected, depending on the type of the instance (this is done through a runtime type test)

Polymorphism (2/3)

Class headers

```
class Matrix
{
protected:
  int height, width;
  double[] elements;
public:
  Matrix(int n, int p,
         double[] e);
  virtual ˜Matrix();
  virtual double det();
  bool is_invertible();
};
```

```
class PositDefMatrix : public Matrix
{
public:
  PositDefMatrix(int n, int p,
                 double[] e);
  Matrix cholesky();
  virtual double det();
};
```

- Note the `virtual` keyword
- A method has been added to determine if matrix is invertible

Polymorphism (3/3)

C++ code snippet

```
bool Matrix::is_invertible()
{
  return(det() != 0);
}

double PositDefMatrix::det()
{
  // Square product of diagonal terms of cholesky decomposition
}
```

- A call to `is_invertible` on a instance of `Matrix` will use the generic determinant computation
- The same call on an instance of `PositDefMatrix` will call the specialized determinant computation

Abstract classes

- It is possible to create classes which don't provide an implementation for some virtual methods
- Syntax in the header:
  `virtual int method_name() = 0;`
- As a consequence, such classes can never be instantiated
- Generally used as the root of a derivation tree, when classes of the tree share behaviours but not implementations
- Such classes are called abstract classes

Some programming rules (1/2)

- Don't repeat yourself (DRY): if several functions contain similar portions of code, factorize that code into a new function
  - makes code shorter
  - reduces the risk of introducing inconsistencies
  - makes easier the propagation of enhancements and bug corrections
- Make short functions
  - often difficult to grasp what a long function does
  - structuring the code by dividing it into short functions makes the logical structure more apparent
  - enhances code readability and maintainability
- Use explicit variable names (except for loop indexes)

Some programming rules (2/2)

- Global variables are evil
  - a global variable can be modified from anywhere in the code (nonlocality problem)
  - creates a potentially unlimited number of dependencies between all portions of the code
  - makes bugs difficult to localize (any part of the code could have created the trouble)
  - to summarize, goes against the principle of modularity
  - in addition, global variables are not thread safe (unless used with locks/mutexes)
- Document your code when it doesn't speak by itself
  - Dynare preprocessor code is documented using Doxygen
  - done through special comments beginning with an exclamation mark
  - run `doxygen` from the source directory to create a bunch of HTML files documenting the code

Outline

Parsing overview

- Parsing is the action of transforming an input text (a `mod` file in our case) into a data structure suitable for computation
- The parser consists of three components:
  - the lexical analyzer, which recognizes the “words” of the `mod` file (analog to the *vocabulary* of a language)
  - the syntax analyzer, which recognizes the “sentences” of the `mod` file (analog to the *grammar* of a language)
  - the parsing driver, which coordinates the whole process and constructs the data structure using the results of the lexical and syntax analyses

Lexical analysis

- The lexical analyzer recognizes the “words” (or lexemes) of the language
- Lexical analyzer is described in `DynareFlex.ll`. This file is transformed into C++ source code by the program `flex`
- This file gives the list of the known lexemes (described by regular expressions), and gives the associated token for each of them
- For punctuation (semicolon, parentheses, ...), operators (+, -, ...) or fixed keywords (*e.g.* `model`, `varexo`, ...), the token is simply an integer uniquely identifying the lexeme
- For variable names or numbers, the token also contains the associated string for further processing
- When invoked, the lexical analyzer reads the next characters of the input, tries to recognize a lexeme, and either produces an error or returns the associated token

Lexical analysis

An example

- Suppose the `mod` file contains the following:

```
model;
x = log(3.5);
end;
```

- Before lexical analysis, it is only a sequence of characters
- The lexical analysis produces the following stream of tokens:

```
MODEL
SEMICOLON
NAME "x"
EQUAL
LOG
LEFT_PARENTHESIS
FLOAT_NUMBER "3.5"
RIGHT_PARENTHESIS
SEMICOLON
END
SEMICOLON
```

Syntax analysis

Using the list of tokens produced by lexical analysis, the syntax analyzer determines which “sentences” are valid in the language, according to a grammar composed of rules.

A grammar for lists of additive and multiplicative expressions

```
%start expression_list;

expression_list := expression SEMICOLON
                 | expression_list expression SEMICOLON;

expression := expression PLUS expression
            | expression TIMES expression
            | LEFT_PAREN expression RIGHT_PAREN
            | INT_NUMBER;
```

- `(1+3)*2; 4+5;` will pass the syntax analysis without error
- `1++2;` will fail the syntax analysis, even though it has passed the lexical analysis

- The `mod` file grammar is described in `DynareBison.yy`
- The grammar is transformed into C++ source code by the program `bison`
- The grammar tells a story which looks like:
  - A `mod` file is a list of statements
  - A statement can be a `var` statement, a `varexo` statement, a `model` block, an `initval` block, ...
  - A `var` statement begins with the token `VAR`, then a list of `NAME`s, then a semicolon
  - A `model` block begins with the token `MODEL`, then a semicolon, then a list of equations separated by semicolons, then an `END` token
  - An equation can be either an expression, or an expression followed by an `EQUAL` token and another expression
  - An expression can be a `NAME`, or a `FLOAT_NUMBER`, or an expression followed by a `PLUS` and another expression, ...

Semantic actions

- So far we have only described how to accept valid `mod` files and to reject others
- But validating is not enough: one need to do something about what has been parsed
- Each rule of the grammar can have a semantic action associated to it: C/C++ code enclosed in curly braces
- Each rule can return a semantic value (referenced to by `$$` in the action)
- In the action, it is possible to refer to semantic values returned by components of the rule (using `$1`, `$2`, ...)

Semantic actions
An example

A simple calculator which prints its results

```
%start expression_list
%type <int> expression

expression_list := expression SEMICOLON
                   { cout << $1; }
                 | expression_list expression SEMICOLON
                   { cout << $2; };

expression := expression PLUS expression
              { $$ = $1 + $3; }
            | expression TIMES expression
              { $$ = $1 * $3; }
            | LEFT_PAREN expression RIGHT_PAREN
              { $$ = $2; }
            | INT_NUMBER
              { $$ = $1; };
```

Parsing driver

The class `ParsingDriver` has the following roles:

- Given the `mod` filename, it opens the file and launches the lexical and syntaxic analyzers on it
- It implements most of the semantic actions of the grammar
- By doing so, it creates an object of type `ModFile`, which is the data structure representing the `mod` file
- Or, if there is a parsing error (unknown keyword, undeclared symbol, syntax error), it displays the line and column numbers where the error occurred, and exits

Outline

The `ModFile` class

- This class is the internal data structure used to store all the informations contained in a `mod` file
- One instance of the class represents one `mod` file
- The class contains the following elements (as class members):
  - a symbol table
  - a numerical constants table
  - two trees of expressions: one for the model, and one for the expressions outside the model
  - the list of the statements (parameter initializations, shocks block, `check`, `steady`, `simul`, ...)
  - an evaluation context
- An instance of `ModFile` is the output of the parsing process (return value of `ParsingDriver::parse()`)

The symbol table (1/3)

- A symbol is simply the name of a variable, of a parameter or of a function unknown to the preprocessor: actually everything that is not recognized as a Dynare keyword
- The symbol table is a simple structure used to maintain the list of the symbols used in the `mod` file
- For each symbol, stores:
    - its name (a string)
    - its type (an integer)
    - a unique integer identifier (unique for a given type, but not across types)

The symbol table (2/3)

Existing types of symbols:

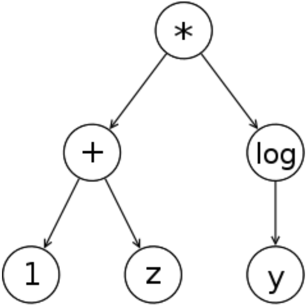
- Endogenous variables
- Exogenous variables
- Exogenous deterministic variables
- Parameters
- Local variables inside model: declared with a pound sign (#) construction
- Local variables outside model: no declaration needed, not interpreted by the preprocessor (*e.g.* Matlab loop indexes)
- Names of functions unknown to the preprocessor: no declaration needed, not interpreted by the preprocessor, only allowed outside model (until we create an interface for providing custom functions with their derivatives)

The symbol table (2/3)

- Symbol table filled in:
  - using the `var`, `varexo`, `varexo_det`, `parameter` declarations
  - using pound sign (#) constructions in the model block
  - on the fly during parsing: local variables outside models or unknown functions when an undeclared symbol is encountered
- Roles of the symbol table:
  - permits parcimonious and more efficient representation of expressions (no need to duplicate or compare strings, only handle a pair of integers)
  - ensures that a given symbol is used with only one type

Expression trees (1/2)

- The data structure used to store expressions is essentially a tree
- Graphically, the tree representation of $(1 + z) * \log(y)$ is:



- No need to store parentheses
- Each circle represents a node
- A node has at most one parent and at most two children

Expression trees (2/2)

- In Dynare preprocessor, a tree node is a represented by an instance of the abstract class `ExprNode`
- This class has 5 sub-classes, corresponding to the 5 types of nodes:
  - `NumConstNode` for constant nodes: contains the identifier of the numerical constants it represents
  - `VariableNode` for variable/parameters nodes: contains the identifier of the variable or parameter it represents
  - `UnaryOpNode` for unary operators (*e.g.* unary minus, log, sin): contains an integer representing the operator, and a pointer to its child
  - `BinaryOpNode` for binary operators (*e.g.* $+$, $*$, pow): contains an integer representing the operator, and pointers to its two children
  - `UnknownFunctionNode` for functions unknown to the parser (*e.g.* user defined functions): contains the identifier of the function name, and a vector containing an arbitrary number of children (the function arguments)

Classes `DataTree` and `ModelTree`

- Class `DataTree` is a container for storing a set of expression trees
- Class `ModelTree` is a sub-class of `DataTree`, specialized for storing a set of model equations (among other things, contains symbolic derivation algorithm)
- Class `ModFile` contains:
  - one instance of `ModelTree` for storing the equations of model block
  - one instance of `DataTree` for storing all expressions outside model block
- Expression storage is optimized through three mechanisms:
  - pre-computing of numerical constants
  - symbolic simplification rules
  - sub-expression sharing

Constructing expression trees

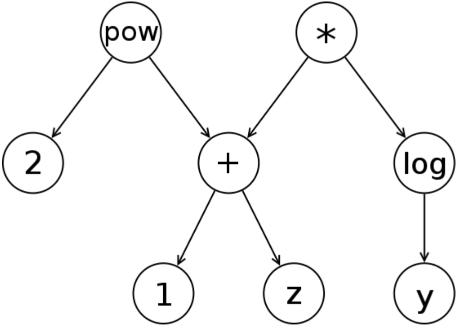
- Class `DataTree` contains a set of methods for constructing expression trees
- Construction is done bottom-up, node by node:
  - one method for adding a constant node (`AddPossiblyNegativeConstant(double)`)
  - one method for a log node (`AddLog(arg)`)
  - one method for a plus node (`AddPlus(arg1, arg2)`)
- These methods take pointers to `ExprNode`, allocate the memory for the node, construct it, and return its pointer
- These methods are called:
  - from `ParsingDriver` in the semantic actions associated to the parsing of expressions
  - during symbolic derivation, to create derivatives expressions
- Note that `NodeID` is an alias (typedef) for `ExprNode*`

Reduction of constants and symbolic simplifications

- The construction methods compute constants whenever it is possible
  - Suppose you ask to construct the node $1 + 1$
  - The `AddPlus()` method will return a pointer to a constant node containing 2
- The construction methods also apply a set of simplification rules, such as:
  - $0 + 0 = 0$
  - $x + 0 = x$
  - $0 - x = -x$
  - $-(-x) = x$
  - $x * 0 = 0$
  - $x/1 = x$
  - $x^0 = 1$
- When a simplification rule applies, no new node is created

Sub-expression sharing (1/2)

- Consider the two following expressions: $(1+z) * \log(y)$ and $2^{(1+z)}$
- Expressions share a common sub-expression: $1+z$
- The internal representation of these expressions is:



Sub-expression sharing (2/2)

- Construction methods implement a simple algorithm which achieves maximal expression sharing
- Algorithm uses the fact that each node has a unique memory address (pointer to the corresponding instance of `ExprNode`)
- It maintains 5 tables which keep track of the already constructed nodes: one table by type of node (constants, variables, unary ops, binary ops, unknown functions)
- Suppose you want to create the node $e_1 + e_2$ (where e_1 and e_2 are sub-expressions):
  - the algorithm searches the binary ops table for the tuple equal to (address of e_1, address of e_2, op code of +) (it is the search key)
  - if the tuple is found in the table, the node already exists, and its memory address is returned
  - otherwise, the node is created, and is added to the table with its search key
- Maximum sharing is achieved, because expression trees are constructed bottom-up

Final remarks about expressions

- Storage of negative constants
  - class `NumConstNode` only accepts positive constants
  - a negative constant is stored as a unary minus applied to a positive constant
  - this is a kind of identification constraint to avoid having two ways of representing negative constants: (-2) and $-(2)$
- Widely used constants
  - class `DataTree` has attributes containing pointers to one, zero, and minus one constants
  - these constants are used in many places (in simplification rules, in derivation algorithm...)
  - sub-expression sharing algorithm ensures that those constants will never be duplicated

List of statements

- A statement is represented by an instance of a subclass of the abstract class `Statement`
- Three groups of statements:
  - initialization statements (parameter initialization with $p = \ldots$, `initval`, `histval` or `endval` block)
  - shocks blocks
  - computing tasks (`check`, `simul`, ...)
- Each type of statement has its own class (*e.g.* `InitValStatement`, `SimulStatement`, ...)
- The class `ModFile` stores a list of pointers of type `Statement*`, corresponding to the statements of the `mod` file, in their order of declaration
- Heavy use of polymorphism in the check pass, computing pass, and when writing outputs: abstract class `Statement` provides a virtual method for these 3 actions

Evaluation context

- The `ModFile` class contains an evaluation context
- It is a map associating a numerical value to some symbols
- Filled in with `initval` block, and with parameters initializations
- Used during equation normalization (in the block decomposition), for finding non-zero entries in the jacobian

Outline

Error checking during parsing

- Some errors in the `mod` file can be detected during the parsing:
  - syntax errors
  - use of undeclared symbol in model block, initval block...
  - use of a symbol incompatible with its type (*e.g.* parameter in initval, local variable used both in model and outside model)
  - multiple shocks declaration for the same variable
- But some other checks can only be done when parsing is completed

Check pass

- The check pass is implemented through method `ModFile::checkPass()`
- Does the following checks:
  - check there is at least one equation in the model (except if doing a standalone BVAR estimation)
  - check there is not both a `simul` and a `stoch_simul` (or another command triggering local approximation)
- Other checks could be added in the future, for example:
  - check that every endogenous variable is used at least once in current period
  - check there is a single `initval` (or `histval`, `endval`) block
  - check that `varobs` is used if there is an estimation

Outline

Overview of the computing pass

- Computing pass implemented in `ModFile::computingPass()`
- Begins with a determination of which derivatives to compute
- Then, calls `ModelTree::computingPass()`, which computes:
  - leag/lag variable incidence matrix
  - symbolic derivatives
  - equation normalization + block decomposition (only in `sparse_dll` mode)
  - temporary terms
  - symbolic gaussian elimination (only in `sparse_dll` mode) *(actually this is done in the output writing pass, but should be moved to the computing pass)*
- Finally, calls `Statement::computingPass()` on all statements

The variable table

- In the context of class `ModelTree`, a variable is a pair (symbol, lead/lag)
- The symbol must correspond to an endogenous or exogenous variable (in the sense of the model)
- The class `VariableTable` keeps track of those pairs
- An instance of `ModelTree` contains an instance of `VariableTable`
- Each pair (`symbol_id`, lead/lag) is given a unique `variable_id`
- After the computing pass, the class `VariableTable` writes the leag/lag incidence matrix:
    - endogenous symbols in row
    - leads/lags in column
    - elements of the matrix are either 0 or correspond to a variable ID, depending on whether the pair (symbol, lead/lag) is used or not in the model

Static versus dynamic model

- The static model is simply the (dynamic) model from which the leads/lags have been omitted
- Static model used to characterize the steady state
- The jacobian of the static model is used in the (Matlab) solver for determining the steady state
- No need to derive static and dynamic models independently: static derivatives can be easily deduced from dynamic derivatives

Example

- suppose dynamic model is $2x \cdot x_{-1} = 0$
- static model is $2x^2 = 0$, whose derivative w.r.t. to x is $4x$
- dynamic derivative w.r.t. to x is $2x_{-1}$, and w.r.t. to x_{-1} is $2x$
- removing leads/lags from dynamic derivatives and summing over the two partial derivatives w.r.t. to x and x_{-1} gives $4x$

Which derivatives to compute ?

- In deterministic mode:
  - static jacobian (w.r. to endogenous variables only)
  - dynamic jacobian (w.r. to endogenous variables only)
- In stochastic mode:
  - static jacobian (w.r. to endogenous variables only)
  - dynamic jacobian (w.r. to all variables)
  - possibly dynamic hessian (if `order` option ≥ 2)
  - possibly dynamic 3rd derivatives (if `order` option ≥ 3)
- For ramsey policy: the same as above, but with one further order of derivation than declared by the user with `order` option (the derivation order is determined in the check pass, see `RamseyPolicyStatement::checkPass()`)

Derivation algorithm (1/2)

- Derivation of the model implemented in `ModelTree::derive()`
- Simply calls `ExprNode::getDerivative(varID)` on each equation node
- Use of polymorphism:
  - for a constant or variable node, derivative is straightforward (0 or 1)
  - for a unary or binary op node, recursively calls method `getDerivative()` on children to construct derivative of parent, using usual derivation rules, such as:
    - $(log(e))' = \frac{e'}{e}$
    - $(e_1 + e_2)' = e_1' + e_2'$
    - $(e_1 \cdot e_2)' = e_1' \cdot e_2 + e_1 \cdot e_2'$
    - ...

Derivation algorithm (2/2)

Optimizations

- Caching of derivation results
  - method `ExprNode::getDerivative(varID)` memorizes its result in a member attribute the first time it is called
  - so that the second time it is called (with the same argument), simply returns the cached value without recomputation
  - caching is useful because of sub-expression sharing
- Symbolic *a priori*
  - consider the expression $x + y^2$
  - without any computation, you know its derivative w.r.t. to z is zero
  - each node stores in an attribute the set of variables which appear in the expression it represents ($\{x, y\}$ in the example)
  - that set is computed in the constructor (straigthforwardly for a variable or a constant, recursively for other nodes, using the sets of the children)
  - when `getDerivative(varID)` is called, immediately returns zero if `varID` is not in that set

Derivation algorithm (2/2)

Optimizations

- Caching of derivation results
  - method `ExprNode::getDerivative(varID)` memorizes its result in a member attribute the first time it is called
  - so that the second time it is called (with the same argument), simply returns the cached value without recomputation
  - caching is useful because of sub-expression sharing
- Symbolic *a priori*
  - consider the expression $x + y^2$
  - without any computation, you know its derivative w.r.t. to z is zero
  - each node stores in an attribute the set of variables which appear in the expression it represents ($\{x, y\}$ in the example)
  - that set is computed in the constructor (straigthforwardly for a variable or a constant, recursively for other nodes, using the sets of the children)
  - when `getDerivative(varID)` is called, immediately returns zero if `varID` is not in that set

Temporary terms (1/2)

- When the preprocessor writes equations and derivatives in its outputs, it takes advantage of sub-expression sharing
- In Matlab static and dynamic output files, equations are preceded by a list of temporary terms
- Those terms are temporary variables containing expressions shared by several equations or derivatives
- Doing so greatly enhances the computing speed of model residual, jacobian or hessian

Example

The equations:

```
residual(0)=x+y^2-z^3;
residual(1)=3*(x+y^2)+1;
```

Can be optimized in:

```
T01=x+y^2;
residual(0)=T01-z^3;
residual(1)=3*T01+1;
```

Temporary terms (2/2)

- Expression storage in the preprocessor implements maximal sharing...
- ...but it is not optimal for the Matlab output files, because creating a temporary variable also has a cost (in terms of CPU and of memory)
- Computation of temporary terms implements a trade-off between:
  - cost of duplicating sub-expressions
  - cost of creating new variables
- Algorithm uses a recursive cost calculation, which marks some nodes as being "temporary"
- *Problem*: redundant with optimizations done by the C/C++ compiler (when Dynare is in DLL mode) $\Rightarrow$ compilation very slow on big models

The special case of Ramsey policy

- For most statements, the method `computingPass()` is a no-op...
- ...except for `planner_objective` statement, which serves to declare planner objective when doing optimal policy under commitment
- Class `PlannerObjectiveStatement` contains an instance of `ModelTree`: used to store the objective (only one equation in the tree)
- During the computing pass, triggers the computation of the first and second order (static) derivatives of the objective

Outline

Output overview

- Implemented in `ModFile::writeOutputFiles()`
- If `mod` file is `model.mod`, all created filenames will begin with `model`
- Main output file is `model.m`, containing:
  - general initialization commands
  - symbol table output (from `SymbolTable::writeOutput()`)
  - lead/lag incidence matrix (from `ModelTree::writeOutput()`)
  - call to Matlab functions corresponding to the statements of the `mod` file (written by calling `Statement::writeOutput()` on all statements through polymorphism)
- Subsidiary output files:
  - one for the static model
  - one for the dynamic model
  - and one for the planner objective (if relevant)
  - written through `ModelTree` methods: `writeStaticFile()` and `writeDynamicFile()`

Model output files

Three possibles modes for `ModelTree` (see `mode` attribute):

- Standard mode: static and dynamic files in Matlab
- DLL mode:
  - static and dynamic files in C++ source code (with corresponding headers)
  - compiled through `mex` to allow execution from within Matlab
- Sparse DLL mode:
  - static file in Matlab
  - two possibilities for dynamic file:
    - by default, a C++ source file (with header) and a binary file, to be read from the C++ code
    - or, with `no_compiler` option, a binary file in custom format, executed from Matlab through `simulate` DLL
    - the second option serves to bypass compilation of C++ file which can be very slow

Outline

Future work (1/2)

Enhancements, optimizations

- Refactor and reorganize some portions of the code
- Create a testsuite (with unitary tests)
- Separate computation of temporary terms between static and dynamic outputs
- Enhance sub-expression sharing algorithm (using associativity, commutativity and factorization rules)
- Add many checks on the structure of the `mod` file

Future work (2/2)

Features

- Add precompiler macros (#include, #define, #if)
- Add handling for several (sub-)models
- Add indexed variables and control statements (if, loops) both in models and command language
- Add sum, diff, prod operators
- For unknown functions in the model: let user provide a derivative, or trigger numerical derivation
- Generalize binary code output
- Generalize block decomposition ?