

The Dynare Preprocessor

Sébastien Villemot Houtan Bastani

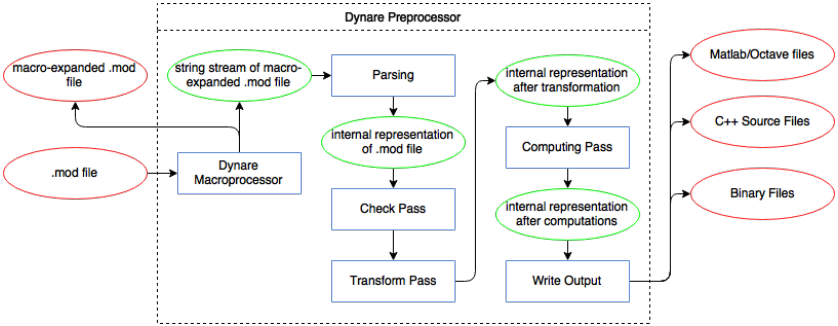
CEPREMAP
CENTRE POUR LA RECHERCHE ECONOMIQUE ET SES APPLICATIONS

1 February 2017



Copyright © 2007–2023 Dynare Team
Licence: Creative Commons Attribution-ShareAlike 4.0

Overview


Dynare Preprocessor
macro-expanded .mod file
string stream of macro-expanded .mod file
Parsing
internal representation after transformation
Matlab/Octave files
C++ Source Files
Binary Files
internal representation of .mod file
Computing Pass
.mod file
Dynare Macroprocessor
Check Pass
internal representation after computations
Transform Pass
Write Output

Outline

1. Invoking the preprocessor
2. Macro processing
3. Parsing
4. Data structure representing a `mod` file
5. Check pass
6. Transform pass
7. Computing pass
8. Writing outputs
9. Proposed Changes

Outline

1. Invoking the preprocessor
2. Macro processing
3. Parsing
4. Data structure representing a `mod` file
5. Check pass
6. Transform pass
7. Computing pass
8. Writing outputs
9. Proposed Changes

Calling Dynare

- Dynare is called from the host language platform with the syntax `dynare «filename».mod`
- This call can be followed by certain options:
  - Some of these options impact host language platform functionality
    → *e.g.* `nograph` prevents graphs from being displayed in MATLAB
  - Some cause differences in the output created by default
    → `notmpterms` prevents temporary terms from being written to the static/dynamic files
  - While others impact the functionality of the macroprocessor or the preprocessor
    → *e.g.* `nostrict` shuts off certain checks that the preprocessor does by default
- MATLAB command line supports syntax completion since Dynare 6.x

Command line options in the mod-file

- Command line options can alternatively be defined in the first line of the .mod file
- Avoids to always have to invoke an option at the command line
  $\rightarrow$ particularly useful for the `nostrict` option during the development phase of a model
- Definition must be
  - a one-line Dynare comment, i.e. begin //
  - the options must be enclosed in `--+ options:` and `+--` and must be whitespace separated
  - As in the command line, if an option admits a value, the equal symbol must not be surrounded by spaces

Example

```
// --+ options: json=compute, stochastic, nostrict +--
```

Outline

1. Invoking the preprocessor
2. Macro processing
3. Parsing
4. Data structure representing a `mod` file
5. Check pass
6. Transform pass
7. Computing pass
8. Writing outputs
9. Proposed Changes

Macro processing

- The Dynare macro language provides a set of macro commands that can be used in `mod` files
- The macro processor employs text expansions/inclusions to transform a `mod` file with macro commands into a `mod` file without macro commands
  $\rightarrow$ result can be stored using `savemacro` option
- The result is fed to the parser
  $\rightarrow$ use `onlymacro` to stop after macro processing before parsing
- Attention: the macro processor only does text substitution; objects computed in later steps are not available yet and cannot be conditioned on for that reason

Outline

1. Invoking the preprocessor
2. Macro processing
3. Parsing
4. Data structure representing a `mod` file
5. Check pass
6. Transform pass
7. Computing pass
8. Writing outputs
9. Proposed Changes

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:
  1. the lexical analyzer, which recognizes the "words" of the `mod` file (analog to the *vocabulary* of a language)
  2. the syntax analyzer, which recognizes the "sentences" of the `mod` file (analog to the *grammar* of a language)
  3. the parsing driver, which coordinates the whole process and constructs the data structure using the results of the lexical and syntax analyses

Undesired Parsing

- Dynare will try to parse all tokens it recognizes
- Code not recognized by the parser is directly passed to the preprocessor output
  $\rightarrow$ allows using MATLAB/Octave commands directly in the mod-file
- Causes problems when one wants to invoke commands containing recognized tokens

Bypassing the parsing

- The `verbatim` block instructs Dynare not to parse text contained in it
- In the following example, `stoch_simul` would otherwise be recognized as a Dynare command during parsing

Verbatim example

```
verbatim;
rhos = [ 0.8, 0.9, 1];
for iter = 1:length(rhos)
    set_param_value('rho',rhos(iter));
    [info, oo_, options_, M_] = stoch_simul(M_, options_, oo_, var_list_)
    if info(1)~=0
        error('Simulation failed for parameter draw')
    end
end
end;
```

1. Lexical analysis

- The lexical analyzer recognizes the "words" (or lexemes) of the language
- Defined in `DynareFlex.ll`, it is transformed into C++ source code by the program `flex`
- This file details the list of known lexemes (described by regular expressions) and 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
```

2. Syntax analysis

In Dynare

- The `mod` file grammar is described in `DynareBison.yy`, which 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, ...

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

Semantic actions

- So far we have only described how to accept valid `mod` files and reject others
- But validating is not enough: one needs to do something with the parsed `mod` file
- Every grammar rule can have a semantic action associated with it: C/C++ code enclosed by curly braces
- Every rule can return a semantic value (referenced 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 << endl; }
                 | expression_list expression SEMICOLON
                   { cout << $2 << endl; };

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

3. Parsing driver

The class `ParsingDriver` has the following roles:

- It opens the `mod` 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

1. Invoking the preprocessor
2. Macro processing
3. Parsing
4. Data structure representing a `mod` file
5. Check pass
6. Transform pass
7. Computing pass
8. Writing outputs
9. Proposed Changes

The ModFile class

- This class is the internal data structure used to store all the information 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, numerical constants table, external functions table
  - trees of expressions: dynamic model, static model, original model, ramsey dynamic model, steady state model, trend dynamic model, ...
  - the list of the statements (parameter initializations, `shocks` block, `check`, `steady`, `simul`, ...)
  - model-specification and user-preference variables: `block`, `bytecode`, `use_dll`, `no_static`, ...
  - an evaluation context (containing `initval` and parameter values)
- 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 (endogenous, exogenous, local, auxiliary, etc), parameter, external function, ... basically everything that is not recognized as a Dynare keyword
- SymbolTable is a simple class used to maintain the list of the symbols used in the `mod` file
- For each symbol, it stores:
  - its name, tex_name, and long_name (strings, some of which can be empty)
  - its type (an enumerator defined in `CodeInterpreter.hh`)
  - a unique integer identifier (also has a unique identifier by type)

The symbol table (2/3)

Existing types of symbols:

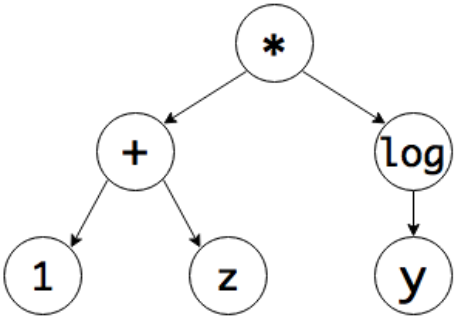
- Endogenous variables
- Exogenous variables
- Exogenous deterministic variables
- Parameters
- Local variables inside model: declared with a pound sign (#) construction
- Local variables outside model: no declaration needed (*e.g.* lhs symbols in equations from `steady_state_model` block, expression outside of model block, ...)
- External functions
- Trend variables
- Log Trend variables
- Unused Endogenous variables (created when `nostrict` option is passed)

The symbol table (3/3)

- Symbol table filled in:
  - using the `var`, `varexo`, `varexo_det`, `parameter`, `external_function`, `trend_var`, and `log_trend_var` declarations
  - using pound sign (#) constructions ("model-local variables") in the model block
  - on the fly during parsing: local variables outside models or unknown functions when an undeclared symbol is encountered
  - during the creation of auxiliary variables in the transform pass
- Roles of the symbol table:
  - permits parsimonious 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 (1/3)

- 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 non external function node has at most one parent and at most three children (an external function node has as many children as arguments)

Expression trees (2/3)

- A tree node is represented by an instance of the abstract class `ExprNode`
- This class has 5 sub-classes, corresponding to the 5 types of non-external-function nodes:
  1. `NumConstNode` for constant nodes: contains the identifier of the numerical constants it represents
  2. `VariableNode` for variable/parameters nodes: contains the identifier of the variable or parameter it represents
  3. `UnaryOpNode` for unary operators (*e.g.* unary minus, log, sin): contains an enumerator representing the operator, and a pointer to its child
  4. `BinaryOpNode` for binary operators (*e.g.* $+$, $*$, pow): contains an enumerator representing the operator, and pointers to its two children
  5. `TrinaryOpNode` for trinary operators (*e.g.* *normcdf*, *normpdf*): contains an enumerator representing the operator and pointers to its three children

Expression trees (3/3)

- The abstract class `ExprNode` has an abstract sub-class called `AbstractExternalFunctionNode`
- This abstract sub-class has 3 sub-classes, corresponding to the 3 types of external function nodes:
  1. `ExternalFunctionNode` for external functions. Contains the identifier of the external function and a vector of its arguments
  2. `FirstDerivExternalFunctionNode` for the first derivative of an external function. In addition to the information contained in `ExternalFunctionNode`, contains the index w.r.t. which this node is the derivative.
  3. `SecondDerivExternalFunctionNode` for the second derivative of an external function. In addition to the information contained in `FirstDerivExternalFunctionNode`, contains the index w.r.t. which this node is the second derivative.

Classes DataTree and ModelTree

- Class DataTree is a container for storing a set of expression trees
- Class ModelTree is a sub-class container of DataTree, specialized for storing a set of model equations.
- In the code, we use ModelTree-derived classes: DynamicModel (the model with lags) and StaticModel (the model without lags)
- Class ModFile contains:
  - one instance of DataTree for storing all expressions outside model block
  - several instances of DynamicModel, one each for storing the equations of the model block for the original model, modified model, original Ramsey model, the Ramsey FOCs, etc.
  - one instance of StaticModel for storing the equations of model block without lags
- Expression storage is optimized through three mechanisms:
  - symbolic simplification rules
  - sub-expression sharing
  - pre-computing of numerical constants

Constructing expression trees

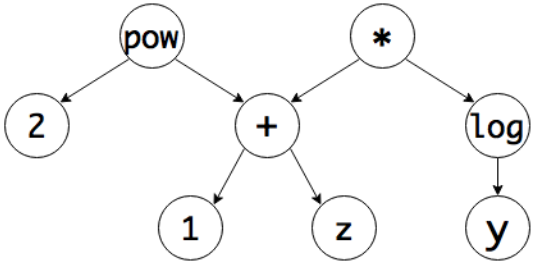
- Class `DataTree` contains a set of methods for constructing expression trees
- Construction is done bottom-up, node by node:
  - one method for adding a constant node (`AddPossiblyNegativeConstant(double)`)
  - one method for a log node (`AddLog(arg)`)
  - one method for a plus node (`AddPlus(arg1, 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
  - when creating the static model from the dynamic model
  - ...

Reduction of constants and symbolic simplifications

- The construction methods compute constants whenever 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
- Attention: this may cause the program to detect issues not directly visible to users like a division by 0

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 9 tables which keep track of the already-constructed nodes: one table by type of node (constants, variables, unary ops, binary ops, trinary ops, external functions, first deriv of external functions, second deriv of external functions, local variables)
- 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 constants: 0, 1, 2, -1, `NaN`, ∞, $-\infty$, and π
  - these constants are used in many places (in simplification rules, in derivation algorithm...)
  - sub-expression sharing algorithm ensures that these 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 (`shocks`, `mshocks`, ...)
  - computing tasks (`steady`, `check`, `perfect_foresight_solver`, ...)
- Each type of statement has its own class (*e.g.* `InitValStatement`, `PerfectForesightSolverStatement`, ...)
- 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 values and parameter initializations
- Used during equation normalization (in the block decomposition), for finding non-zero entries in the Jacobian
- Used in testing that trends are compatible with a balanced growth path, for finding non-zero cross partials of equations with respect to trend variables and endogenous variables

Outline

1. Invoking the preprocessor
2. Macro processing
3. Parsing
4. Data structure representing a `mod` file
5. Check pass
6. Transform pass
7. Computing pass
8. Writing outputs
9. Proposed Changes

Error checking during parsing

- Some errors in the `mod` file can be detected during parsing:
  - syntax errors
  - use of undeclared symbols 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 shock declarations for the same variable
- But some other checks can only be done when parsing is completed...

Check pass

- The check pass is implemented through the method `ModFile::checkPass()`
- Performs many checks. Examples include:
  - check there is at least one equation in the model (except if doing a standalone BVAR estimation)
  - checks for coherence in statements (*e.g.* options passed to statements do not conflict with each other, required options have been passed)
  - checks for coherence among statements (*e.g.* if `osr` statement is present, ensure `osr_params` and `optim_weights` statements are present)
  - checks for coherence between statements and attributes of `mod` file (*e.g.* `use_dll` is not used with `block` or `bytecode`)

Outline

1. Invoking the preprocessor
2. Macro processing
3. Parsing
4. Data structure representing a `mod` file
5. Check pass
6. Transform pass
7. Computing pass
8. Writing outputs
9. Proposed Changes

Transform pass (1/2)

- The transform pass is implemented through the method `ModFile::transformPass(bool nostrict)`
- It makes necessary transformations (notably to the dynamic model, symbol table, and statements list), preparing the `ModFile` object for the computing pass. Examples of transformations include:
  - creation of auxiliary variables and equations for leads, lags, expectation operator, differentiated forward variables, etc.
  - detrending of model equations if nonstationary variables are present
  - decreasing leads/lags of predetermined variables by one period
  - addition of FOCs of Lagrangian for Ramsey problem
  - addition of `dsge_prior_weight` initialization before all other statements if estimating a DSGE-VAR where the weight of the DSGE prior of the VAR is calibrated

Transform pass (2/2)

- It then freezes the symbol table, meaning that no more symbols can be created on the `ModFile` object
- Finally, checks are performed on the transformed model. Examples include:
  - same number of endogenous variables as equations (not done in certain situations, *e.g.* Ramsey, discretionary policy, where checks are done in MATLAB)
  - correspondence among variables and statements, *e.g.* Ramsey policy, identification, perfect foresight solver, and perfect foresight solver are incompatible with deterministic exogenous variables
  - correspondence among statements, *e.g.* for DSGE-VAR with `bayesian_irf` option, the number of shocks must be equal to the number of observed variables

Outline

1. Invoking the preprocessor
2. Macro processing
3. Parsing
4. Data structure representing a `mod` file
5. Check pass
6. Transform pass
7. Computing pass
8. Writing outputs
9. Proposed Changes

Overview of the computing pass

- Computing pass implemented in `ModFile::computingPass()`
- Creates static model from dynamic one (by removing leads/lags)
- Determines which derivatives to compute
- Then calls `DynamicModel::computingPass()`, which computes:
  - leag/lag variable incidence matrix
  - symbolic derivatives w.r.t. endogenous, exogenous, and parameters, if needed
  - equation normalization + block decomposition
  - temporary terms
  - computes equation cross references, if desired
- NB: analogous operations for static model are performed by `StaticModel::computingPass()`
- Asserts that equations declared `linear` are indeed linear (by checking that Hessian $== 0$)
- Finally, calls `Statement::computingPass()` on all statements

Model Variables

- In the context of class `ModelTree`, a variable is a pair (symbol, lag)
- The symbol must correspond to a variable of type endogenous, exogenous, deterministic exogenous variable, or parameter
- The `SymbolTable` class keeps track of valid symbols, while the `variable_node_map` keeps track of model variables (symbol, lag pairs stored in `VariableNode` objects)
- After the computing pass, the `DynamicModel` class writes the leag/lag incidence matrix:
  - `max_lag + max_lead + 1` rows (usually 3): the first row indicates $t-1$ (if applicable), the second row t, and the third row $t+1$ (if applicable)
  - one column for every endogenous symbol in order of declaration; NB: includes endogenous auxiliary variables created during the transform pass
  - elements of the matrix are either 0 (if the variable does not appear in the model) or correspond to the variable's column in the Jacobian of the dynamic model

Static versus dynamic model

- The static model is simply the dynamic model without leads and lags
- 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

Example

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

Which derivatives to compute?

- In deterministic mode:
  - static Jacobian w.r.t. endogenous variables only
  - dynamic Jacobian w.r.t. endogenous variables only
- In stochastic mode:
  - static Jacobian w.r.t. endogenous variables only
  - dynamic Jacobian w.r.t. endogenous, exogenous, and deterministic exogenous variables
  - dynamic Hessian w.r.t. endogenous, exogenous, and deterministic exogenous variables
  - possibly dynamic 3rd derivatives (if `order` option ≥ 3)
  - possibly dynamic Jacobian and/or Hessian w.r.t. parameters (if `identification` or analytic derivs needed for `estimation` and `params_derivs_order` > 0)
- 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::computeJacobian()`, `ModelTree::computeHessian()`, `ModelTree::computeThirdDerivatives()`, and `ModelTree::computeParamsDerivatives()`
- Simply call `ExprNode::getDerivative(deriv_id)` on each equation node
- Use of polymorphism:
  - for a constant or variable node, derivative is straightforward (0 or 1)
  - for a unary, binary, trinary op nodes and external function nodes, recursively calls method `computeDerivative()` on children to construct derivative

Derivation algorithm (2/2)

Optimizations

- Caching of derivation results
  - method `ExprNode::getDerivative(deriv_id)` memorizes its result in a member attribute (`derivatives`) the first time it is called
  - the second time it is called (with the same argument), it simply returns the cached value without recomputation
  - caching is useful because of sub-expression sharing
- Efficiently finds symbolic derivatives equal to 0
  - consider the expression $x + y^2$
  - without any computation, you know its derivative w.r.t. z is zero
  - each node stores in an attribute (`non_null_derivatives`) the set of variables which appear in the expression it represents ($\{x, y\}$ in the example)
  - this set is computed in `prepareForDerivation()`
  - when `getDerivative(deriv_id)` is called, immediately returns zero if `deriv_id` 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
- These terms are variables containing expressions shared by several equations or derivatives
- Using these terms greatly enhances the computing speed of the model residual, Jacobian, Hessian, or third derivative

Example

The equations:

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

Can be optimized in:

```
T1=x+y^2;
residual(0)=T1-z^3;
residual(1)=3*T1+1;
```

Temporary terms (2/2)

- Expression storage in the preprocessor implements maximal sharing, but this 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 model

- 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` which stores the objective function (`i.e.` 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

1. Invoking the preprocessor
2. Macro processing
3. Parsing
4. Data structure representing a `mod` file
5. Check pass
6. Transform pass
7. Computing pass
8. Writing outputs
9. Proposed Changes

Output overview

- Implemented in `ModFile::writeOutputFiles()`
- If `mod` file is `mymodel.mod`, all created filenames will begin with `mymodel`
- Main output file is `mymodel.driver`, containing:
  - general initialization commands
  - symbol table output (from `SymbolTable::writeOutput()`)
  - lead/lag incidence matrix (from `DynamicModel::writeDynamicMFile()`)
  - 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:
  - for the static model (residuals, temporary terms, derivatives)
  - for the dynamic model (residuals, temporary terms, derivatives)
  - one for the auxiliary variables in the dynamic model (if relevant)
  - one for the steady state file (if relevant)
  - for the planner objective and Lagrange multipliers (static residuals and derivatives, if relevant)
  - one each for the static and dynamic parameter derivatives (if required)

Model output files

Three possible output types:

- MATLAB/Octave mode: static and dynamic files in MATLAB
- Julia mode: static and dynamic files in Julia
- 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

1. Invoking the preprocessor
2. Macro processing
3. Parsing
4. Data structure representing a `mod` file
5. Check pass
6. Transform pass
7. Computing pass
8. Writing outputs
9. Proposed Changes

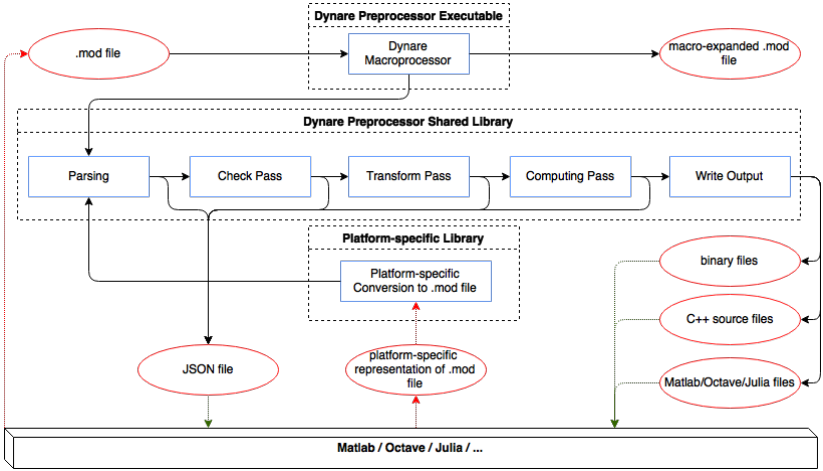
Proposed changes with addition of Julia support (1/2)

1. Julia output is provided upon parsing of `mod` file, everything else done in Julia
   - Pros: very few changes to the preprocessor
   - Cons: repeated code (same checks, transformations, computations done in preprocessor and Julia); potential code divergence/two parallel projects
2. Dump preprocessor altogether: do everything with Julia
   - Pros: simple to distribute, move away from C++ (no contributions, requires more expertise)
   - Cons: MATLAB/Octave users must also download Julia, a big project, speed (?)

Proposed changes with addition of Julia support (2/2)

3. Create libraries out of the preprocessor
   - Pros: Dynare interaction similar across HLPs, preprocessor used as is
   - Cons: difficult for outsiders to contribute, big project, not much benefit in speed when compared to...
4. Write `mod` file from HLP then call preprocessor; option to output JSON file representing `ModFile` object at every step of the preprocessor
   - Pros: Dynare interaction similar across HLPs, preprocessor used as is, minimal amount of work, easy incremental step, allows users to support any given HPL given the JSON output
   - Cons: unnecessary processing when certain changes made in host language, keeps defaults of current preprocessor, speed (?)
5. Other ideas?

Using HLP `mod` file objects (1/2)



Using HLP `mod` file objects (2/2)

- Allows interactivity for all HLPs; requires only
  - A definition of a mod file class in the HLP
  - A library function that converts an HLP mod file object to a `mod` file
- Allows users to use Dynare with any HPL. Standard JSON output can be read in any HPL; user can use it construct desired HPL objects and work with model in their language of preference
- Easy first step
- No divergence of codebase: don't need to repeat code (checks, transformations, etc.) across platforms
- Creates `mod` files that can be used on other host language platforms
- Adds one more HLP library to distribute
- Need to design/implement classes that will store processed Dynare `mod` file in various HLPs