Scala Compiler Phase and Plug-In Initialization for Scala 2.8

Copyright © 2008, Anders Bach Nielsen

Abstract
This document describes the design and implementation of a more uniform way of handling internal compiler phases and external user supplied phases via plug-ins in the Scala compiler.

Contents

1 Motivation ................................................................. 2
  1.1 Goals ............................................................... 2
  1.2 Detailed Use Cases .................................................. 2
    1.2.1 Assembling compiler internal phases for the JVM target and no plug-ins are loaded. 2
    1.2.2 Assembling compiler internal phases for the MSIL target and no plug-ins are loaded. 3
    1.2.3 Adding one plug-in supplied phase to the JVM target compiler after type checking. 3
    1.2.4 Adding two plug-in supplied phase to the JVM target compiler after type checking. 4
    1.2.5 Adding two plug-in supplied phases, where one has a dependency on the other, to the JVM target compiler. 4
    1.2.6 Adding plug-in supplied phases with dependencies to both plug-in supplied and internal phases to the JVM target compiler. 6
    1.2.7 Two separate phases that should be handled as a single phase. 7
    1.2.8 A cyclic dependency among phases is a fatal error. 8
    1.2.9 Special handling of the parser and terminal phases. 9
    1.2.10 Larger compiler example with three plug-in supplied phases. 9
  1.3 Future Goals That Extend Beyond This Proposal ......................... 12

2 Description ..................................................................... 12
  2.1 The Present Design ...................................................... 12
  2.2 The Proposed Design ..................................................... 12
    2.2.1 Internal Phases .................................................. 13
    2.2.2 External Phases .................................................. 13
    2.2.3 Phases Set ....................................................... 14
    2.2.4 Phase Assembly .................................................. 14
    2.2.5 Constraint System .............................................. 15
  2.3 Constraint Solving Algorithm ......................................... 16
    2.3.1 The phasesSetToDepGraph function .......................... 17
    2.3.2 The removeDanglingNodes function .......................... 18
    2.3.3 The validateAndEnforceHardlinks function .................. 18
    2.3.4 The collapseHardLinksAndLevels function .................. 21
    2.3.5 The compilerPhaseList function ............................. 22
  2.4 Naming Scheme for Internal and External Phases ....................... 23
  2.5 Implications of This Proposal ........................................ 23

3 Implementation .................................................................. 24

A Example implementation .................................................. 25
1 Motivation

This section will cover the motivating goals of creating this proposal and some use cases emphasizing these goals. There will also be a section on goals that extend beyond this proposal and will therefore not be covered by this proposal.

1.1 Goals

- Handling internal compiler phases and external user supplied phases (via plug-ins) in a uniform way.
- The order of the phases in the compiler phase chain is fully determined by the set of phases given to the compiler as internal phases or given via plug-ins. For example, it must not matter in what order the compiler processes the individual plug-ins that supply the external phases.
- Phase ordering is based on a simple, but sufficient powerful and solvable constraint system.
- Ordering of the internal compiler phases should always succeed in a valid compiler phase chain. Adding two plug-ins, which by themselves give a valid compiler phase chain and only have dependencies on phase supplied by that plug-in or internal plug-ins, should also give a valid compiler phase chain.

1.2 Detailed Use Cases

1.2.1 Assembling compiler internal phases for the JVM target and no plug-ins are loaded.

All phases have to declare a list of phase names that should be run before the phase itself in the final compiler phase chain. In this use case there are no external phases, because there are no plug-in supplied. Depending on the command line options and the JVM target the for this target default phases are added to the phases set. This is a set, because a phase may only occur once in the compiler phase chain.

Following these basic runs after constraints in the list of phase names, the compiler phase chain is build from the phases set. In the example below a simple set of phase constraints can be converted into a graph, that in turn can be converted into a list of phases.

Using the list of runs after constraints as the basic ordering constraints, there are some properties that have to be satisfied.

- For instance, the phases liftcode and flatten are only included for the JVM target. They have runs after dependencies on the refchecks phase and the constructors phase, respectively. Because they are only included in the JVM target the uncurry phase and the mixin phase, respectively, need to have multiple dependencies as seen in the example below.

The example shown below, shows the desired transformation steps of a small dependency graph taken from the actual compiler phase graph including refchecks, liftcode and uncurry. When processing refchecks we notice that it has two dependencies to it. One dependency comes from liftcode and the other from uncurry. We now look if any of these phases have more then one outgoing dependency. The
liftcode phase has one outgoing dependency, but the uncurry phase has two, so one of the outgoing dependencies of uncurry has to go to refchecks and the other dependency has to go to some place we have not visited yet. So by this reasoning, we can safely ignore the dependency from uncurry to refchecks (marked red).

- Some internal phase pairs have very strong dependencies, like the phases explicitouter and erasure and also the namer and typer phases. See section 1.2.7 for more information on this problem.

1.2.2 Assembling compiler internal phases for the MSIL target and no plug-ins are loaded.

Like in use case 1 depending on command line options and that the target being MSIL, the needed phases are added to the phases set. Again there are no external phases supplied. There are however some differences to use case 1.

- The phases liftcode and flatten are not included for the MSIL target. This makes the list of **runs after** dependencies of the phases uncurry and mixer special by containing dependencies to phases that are found in the phases set.

  For instance the phase uncurry has to specify that it wants to run after both refchecks and liftcode, as seen in section 1.2.1. (For the JVM target this will force the liftcode phase before uncurry.) For the MSIL target the phase liftcode is not present as shown in the diagram below.

1.2.3 Adding one plug-in supplied phase to the JVM target compiler after type checking.

Like in use case 1 the internal phases are added to the phases set. In this use case a plug-in is loaded that will supply one phase to the phases set with a constraint that it wants to run after the refchecks phase. The original plug-in handling code will still handle loading plug-ins, disabling plug-ins, but will no longer have the responsibility of assembling the compiler chain. The plug-in is loaded and the phase is added to the phases set together with the internal phases.

When assembling the compiler phase chain in this use case, there will be two phases directly after the refchecks phase and the dependencies can not be removed as described in use case 1. A simplified, but similar situation is shown in the example below.
1.2.4 Adding two plug-in supplied phase to the JVM target compiler after type checking.

This use case is very similar to use case 3, however instead of having one internal and one external phase after the refchecks phase, we now have one internal and two external phases after the refchecks phase. A similar situation is shown in the example below.

Again the dependency graph is sectioned into levels and each level is then sorted alphabetically according to the phase name.

1.2.5 Adding two plug-in supplied phases, where one has a dependency on the other, to the JVM target compiler.

This use case is very similar to use case 4, however instead of the two plug-in supplied phases having a dependency on the same internal phase, one of the plug-in supplied phases has a dependency on the other plug-in supplied phase and the other phase then has a dependency on an internal phase. This is shown in the example in the diagram below.
The standard behavior would be to divide the dependency graph into sections and then sort each level alphabetically. However, the intention could have been to keep the two plug-in supplied phases together. This is modeled by a *runs before* constraint, that a phase can specify. This can be seen in the diagram below.

A *runs before* constraint from R to F, can without loss of generality, be converted into a *runs after* constraint from F to R. With this conversion we are again able to divide the graph into sections and solve it as before.

The example shown below, can be solved applying the same logic as described above.
1.2.6 Adding plug-in supplied phases with dependencies to both plug-in supplied and internal phases to the JVM target compiler.

This use case builds upon use case 1, use case 3, use case 4 and use case 5 and shows how to handle phases with multiple *runs after* constraints and *runs before* constraints on other plug-in supplied and internal phases. The important phase in this example is T, that with its *runs before* constraint on phase R pushes that phase to the end of the list.
1.2.7 Two separate phases that should be handled as a single phase.

In use case 1 we specified the assembly of the compiler phase chain based on runs after constraints. However, the compiler contains some phase pairs that have very strong dependencies on each other or it simply does not make sense to put a phase in between. In this use case we will focus on the situation that we have two individual phase objects, but one needs to run right after the other and thereby act as one large phase.

The solution to this is to introduce a new runs right after constraint to all phases. In the example below the runs right after constraint is marked as a blue arrow and behaves as a normal runs after constraint.

In the example shown below, we can see the first four phases of the Scala compiler. There is a runs right after constraint between typer and namer. There is in this example a plug-in supplied phase (plugin1) that wants to run after the namer phase. The semantics of the runs after constraint is that it should run somewhere after a specified phase, but as soon as possible. With this in mind it is perfectly valid to change the dependency of phase plugin1 from namer to typer.
If the phase plugin1 had declared a run right after constraint to the namer phase, this would resulted in an unsolvable graph and would therefore have generated an error. Only one phase can come right after another phase. For the same reason we disallow runs before constraints to phases with a runs right after constraint.

A simplification of the dependency graph and to make sure that runs right after constraints are satisfied, nodes with runs right after constraints between them are collapsed into one large node. This is shown in the last step in the example below. The first step in the example below, is to ensure that runs right after constraints really are the only constraint that come a specific node.

1.2.8 A cyclic dependency among phases is a fatal error.

Running the compiler without loading plug-ins it is guaranteed that there are no cyclic dependencies among the internal phases. The only way to introduce cycles in the phase graph is by loading one or more plug-ins. If a cycle is detected in the phase graph the compiler should refuse to start and give a proper error message.

In the example shown below a cycle is detected and marked in pink. A cycle like this would generate a cyclic dependency error and outputting the names of the involved phases.
1.2.9 Special handling of the parser and terminal phases.

There are two very special phase objects in the phase graph. The first phase is the parser phase. This phase will be the head of the compiler phase chain and no other phase is allowed to come before this phase. For this reason *runs before* constraints on the parser phase are not allowed and that dependency is dropped.

The other special phase is the terminal phase, which has to be the last phase in the compiler phase chain. For this reason it is not allowed to run after or run right after the terminal phase. Any phase that specifies a *runs after* or *runs right after* constraint on the terminal phase will have that dependency dropped.

1.2.10 Larger compiler example with three plug-in supplied phases.

This example is implemented in the reference implementation of the algorithm below. In this example the compiler consists of nine internal phases and two plug-ins are loaded adding a total of three phases to the compiler.

<table>
<thead>
<tr>
<th>Phase name</th>
<th>Runs after</th>
<th>Runs right after</th>
<th>Runs before</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Internal phases</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsc::parser</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsc::typer</td>
<td>nsc::parser</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsc::pickler</td>
<td>nsc::typer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsc::liftcode</td>
<td>nsc::pickler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsc::tailcalls</td>
<td>nsc::pickler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsc::liftcode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsc::erasure</td>
<td></td>
<td>nsc::tailcalls</td>
<td></td>
</tr>
<tr>
<td>nsc::cleanup</td>
<td></td>
<td>nsc::erasure</td>
<td></td>
</tr>
<tr>
<td>nsc::jvm</td>
<td>nsc::cleanup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsc::terminal</td>
<td>nsc::jvm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsc::msil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Plug-in supplied phases</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>plug1::optimization1</td>
<td>nsc::typer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>plug2::optimization1</td>
<td>plug1::optimization1</td>
<td>nsc::tailcalls</td>
<td></td>
</tr>
<tr>
<td>plug2::optimization2</td>
<td>plug2::optimization1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsc::tailcalls</td>
<td></td>
<td></td>
<td>nsc::jvm</td>
</tr>
</tbody>
</table>

Using the phases in the table above as the basis for the dependency graph, the following graph will be constructed. This graph contains all the information that is also present in the phases set. As seen in the diagram below there are nodes without phase objects (marked with dotted lines) and nodes with multiple dependencies. There could also be cycles at this stage, but there are non in this example.
The transformation of the dependency graph into a dependency tree is the basis of this proposal. The resulting dependency tree is shown below. This tree is then the basis for generating the compiler phase list.
The resulting compiler phase list of traversing the tree above is shown below. It is ensured that the parser phase is the first and that the terminal phase is the last in this list.

- nsc::parser
- nsc::typer
- nsc::pickler
- plug1::optimization1
- nsc::liftcode
- plug2::optimization1
- nsc::tailcalls
- nsc::erasure
- nsc::cleanup
- plug2::optimization2
- nsc::jvm
- nsc::terminal
1.3 Future Goals That Extend Beyond This Proposal

- Create and add new back ends to the compiler that will replace the JVM or MSIL specific phases using plug-ins.
- Creating internal tools by sub classing the class Global.
- The constraint system should not contain support for the following constraints.
  - A phase should be able to declare that it will replace a given phase and thereby ignoring other constraints declared by this phase and taking over the constraints from the replaced phase.
  - A phase should be able to declare that if included it will remove a given number of phases and removing constraints to these phases.
  - A phase should be able to declare that it only wants to be included if a given phase is also present or else it will be silently dropped.

2 Description

This section will cover a short description of the present design and the full description of the proposed design changes, including the constraint system and algorithm for solving the constraint system.

2.1 The Present Design

Currently phases in the compiler are ordered statically by a hard coded list structure in class Global (found in Global.scala), where the method builtInPhaseDescriptors returns the list of phase objects. The resulting list from invoking this method is then extended with the phases from the plug-in supplied by the user. This is done in the computePhaseDescriptors method in class Plugins (found in plugins/Plugins.scala). This method is then again called from the method phaseDescriptors in class Global. So all together the process of calculating the phases of the compiler is spread over several files and quite complicated.

A plug-in can supply a number of phases to the compiler and the architecture of the plug-in subsystem already supply some of the functionality that will be added to all internal phases. All phases are a subclass of class SubComponent and all phases supplied by plug-ins are subclasses of class PluginComponent (which is a subclass of class SubComponent).

2.2 The Proposed Design

This section will cover the design of the proposed system for the handling of internal and external phases in a uniform way. It will also cover a description of the constraint system and a algorithm for solving these constraint for generating the compiler phase chain. This design will focus on satisfying the use cases described earlier.

From the use cases above it is possible to identify three kinds of constraints. The first is the \textit{runs after} constraint, the second is the \textit{runs right after} constraint and the third is the \textit{runs before} constraint. It is necessary to declare that a given phase has to \textit{runs after} a list of phase names, however the \textit{runs right after} constraint only needs to declare the name one phase. For symmetry between the \textit{runs after} and the \textit{runs before} constraint it should also be possible to declare a list of phase names in the \textit{runs before} constraint.

- Lets define the \textit{runs after} constraint using the \texttt{runsAfter} variable with the Scala type \texttt{List[String]}. Using this constraint, phase B would declare \texttt{runsAfter=List(A)}, meaning that phase B would like to run somewhere after phase A, but as early as possible.
- Lets define the \textit{runs right after} constraint using the \texttt{runsRightAfter} variable with the Scala type \texttt{Option[String]}. Using this constraint, phase B would declare \texttt{runsRightAfter=Some(A)}, meaning that phase B wants to run right after phase A and no other phases are allowed to come between phase A
and B. From a larger perspective meaning that phase A and B can be regarded as on large phase with one input and one output.

- Let's define the *runs before* constraint using the runsBefore variable with the Scala type List[String]. Using this constraint, phase B would declare runsBefore=List(A), meaning that phase B must run before phase A.

A goal of this work is to handle the inclusion and constraint resolution of internal and external plug-in supplied phases in a uniform way, this also means that the responsibility of phase assembly should not be handled by the plug-in subsystem any more, but by a separate member. For this separate member we create a new PhaseAssembly trait that is mixed into the Global class in the same way as trait Plugins is mixed in.

In the diagram below a schematic overview of the proposed system is shown. A description of the individual components and the modification needed is given below.

### 2.2.1 Internal Phases

All internal phase objects are declared in class Global and will be extended to include the runsAfter and the runsRightAfter variables with the types described above. This will ensure that the internal phases will always succeed in creating a compiler phase chain depending on the compiler target. There is no need for the internal phases to declare a runsBefore variable, because the internal order can be described using only after constraints. For this reason, the default value of runsBefore is given in class SubComponent to be the empty list.

The phase objects that are needed for a given target will be added to the phases set. See section 2.2.3 for more information.

### 2.2.2 External Phases

All external phases are supplied through plug-ins. All these phases inherit from class PluginComponent. Defining the runsRightAfter=None in class PluginComponent will make the changes to the existing plug-ins minimal, but makes it possible for external phases to use the feature by simple overriding the variable if needed.

It is still the responsibility of the plug-in subsystem to load the actual plug-ins from the jar files and also the processing of all the -Xplugin* compiler options available at the command line. The phase assembly code
currently present in the plug-in subsystem will be replaced by code that adds all plug-in supplied phases to the phases set. See section 2.2.3 for more information.

### 2.2.3 Phases Set

The phases set contains all the phase objects that should be taken into consideration when performing compiler phase chain assembly. This phases set would located in the class Global. A proper Scala declaration of such a phases set is shown below.

```scala
protected val phasesSet : HashSet[SubComponent] = new HashSet[SubComponent]()
```

Ensuring that all phase objects are unique with respect to the phase names, the hashCode method in class SubComponent will be overridden to return the hash code of the phase name and not the object itself.

### 2.2.4 Phase Assembly

From a compiler design perspective this component will include the data structures and implementation of the constraint solving algorithm that is needed to implement this proposal. It would be implemented as atrait PhaseAssembly that is mixed into class Global with this self type self: Global =>

A very high level description of the actions performed by this component can be split into several distinct parts. A detailed algorithm that implements this high level description can be found below in the section on the constraint solving algorithm.

1. **Phases set to dependency graph**
   
   From the phase objects in the phases set a dependency graph is created, where the edges in this graph are constructed from the runsAfter, runsRightAfter and runsBefore constraints. All runsBefore constraints are defined as A runsBefore B, but are here turned into dependencies in the form B depends on A. Each node in the dependency graph has both a phase name and a phase object, because all constraints in the phase objects are only given by name.

2. **Remove dangling nodes**
   
   Phases that have named dependencies to phases that are not in the phases set will produce dangling nodes. These dangling nodes have no phase object attached and can be removed.

3. **Validate and enforce hard links**
   
   Phases that declare runs before constraint to phases with a runs right after constraint can give undesirable and even unsolvable graphs. This is checked and also all hard links are checked to see if they are the only edge that is pointing to a node.

4. **Cycles, levels and collapse hard links**
   
   This recursive function will section the graph into levels and at the same time detect any cycles that should be present in the graph. When a hard link is encountered the involved nodes are collapsed into one.

5. **Compiler phase list assembly**
   
   All nodes at increasing levels are extracted from the graph and sorted alphabetically by the phase name and appended to the compiler chain.
2.2.5 Constraint System

The constraint system is implemented as a graph structure where phases are modeled as nodes and the constraints are modeled as directed edges. This section will describe the basic entities of the constraint system and how they are modeled in the dependency graph.

Each node in the dependency graph contains several items of information about the phase object and the dependencies to and from it.

- **phasename**: (type: String)
  The name of the phase object. This is always set.

- **phaseobj**: (type: Option[List[SubComponent]])
  Either no phase object is set (None) or there is a reference to the actual phase object packed in a list (Some(List[SubComponent](phsobj))). If there is a phase object then there is also a phase name, but the constraints are only given by name, so some nodes may have a phase name and no phase object.

- **after**: (type: HashSet[Edge]())
  This is the set of Edge object that point to Node objects that have to come before this Node. So all runsAfter and runsRightAfter constraints given by a phase object are present in this after list.

- **before**: (type: HashSet[Edge]())
  This is the set of Edge object that points to this Node object. So its the opposite of the after list.

- **visited**: (type: Boolean)
  This is used for cycle detection and is set to false per default.

- **level**: (type: Int)
  This is the level of this node in the sectioned graph. The default value is 0.

Each edge (link) in the dependency graph is directed, so it has references to its to and from Node objects and knows if it is created from a runs after constraint (soft link) or created from a runs right after constraint (hard link). All runs before constraints are created as soft edges (soft link) in the graph, but in the opposite direction. This means that if phase A declares to run before phase B, a soft edge from B to A is created.

- **to**: (type: Node)
  This is a reference to the Node object this Edge object is pointing to. This means that the frm node has a dependency on the to node.

- **frm**: (type: Node)
  This is a reference to the Node object this Edge object is pointing from.

- **hard**: (type: Boolean)
  In the dependency graph we have soft and hard dependencies. The soft dependencies are created from the runsAfter and runsBefore (see note above) lists of constraints. The hard dependencies are created from the runsRightAfter constraint, if any.
Below is the interface implementation of the dependency graph in Scala. There are classes implementing the edges and nodes and the container for storing the edges and nodes and aux. method for easy access to the stored information.

class Edge(f: Node, t: Node, h: Boolean) {
    var frm = f
    var to = t
    var hard = h
}

class Node(name:String) {
    var phasename = name
    var phaseobj: Option[List[SubComponent]] = None
    var after = new HashSet[Edge]()
    var before = new HashSet[Edge]()
    var visited = false
    var level = 0
}

class DependencyGraph {
    val nodes = new HashMap[String,Node]()
    val edges = new HashSet[Edge]()

    def getNodeByPhase(name : String) : Node
    def getNodeByPhase(phs : SubComponent) : Node
    def softConnectNodes(frm: Node, to: Node) : Unit
    def hardConnectNodes(frm: Node, to: Node) : Unit
}

2.3 Constraint Solving Algorithm

This algorithm will build a list of phase objects from a set of phase objects with constraints. The position of each phase object in the final list will ensure that all its constraints are satisfied. The algorithm does this by constructing a dependency graph from the set of phase objects. This dependency graph is then verified and a list of phase objects is extracted from the graph, that represent the compiler phase chain that satisfy all given constraints.

The function below is the top most function of this algorithm that transforms the set of phase objects into a ordered list of phase objects so all constraints are satisfied. This function follows the description given about the phase assembly component.

function buildCompilerFromPhasesSet(phasesSet)
// call function to generate dependency graph from phases set
graph <- phasesSetToDepGraph( phaseSet )
// Remove all nodes where phaseobj == null
removeDanglingNodes( graph )
// Enforce hard links (runs right after) and promote nodes down the tree
validateAndEnforceHardlinks( graph )
// Get root node from the graph
rootnode <- graph.getRootNode()
// Test for cycles, assign levels and collapse hard links into nodes
collapseHardLinksAndLevels( graph, rootnode, 1 )
// Assemble compiler phase chain from the graph
return compilerPhaseList( graph )

List of functions called:

• The phasesSetToDepGraph function
• The removeDanglingNodes function
• The validateAndEnforceHardlinks function
• The collapseHardLinksAndLevels function
• The compilerPhaseList function

2.3.1 The phasesSetToDepGraph function

The dependency graph, with the elements described in the section about the constraint system, is build from
the phase objects in the phases set.
It works as follows:

• create a new dependency graph
• for each phase in the phasesSet
  – create a new graph node fromnode from the phase object and add the node to the graph
  – if the phase object has a runs right after constraint, then create a node tonode from the phase
    name of the constraint and create a hard edge between fromnode and tonode. Eventual runs
    after constraints are ignored.
  – if the phase object has no runs right after constraints.
    * for each phasename in phase.after
      · A node tonode is created from phasename and a soft edge from fromnode to tonode is
        created.
    * foreach phasename in phase.before
      · A node tonode is created from phasename and a soft edge from tonode to fromnode is
        created.

A pseudo code version of this algorithm is shown below.

function phasesSetToDepGraph( phaseSet )
// create new graph structure to hold data
graph <- new DependencyGraph
// iterate over all phase objects in the phasesSet
for phs in phasesSet

// create a node from the phase object
fromnode <- graph.getNodeByName(phs)
// test if the phase object has a runs right after constraint
if phs.runRightAfter not eq ""
    // create a node from the phase constraint name and connect with hard edge
tonode <- graph.getNodeByName( phs.runRightAfter )
    graph.hardConnectNodes( fromnode, tonode )
else
    // the phase object did not have a runs right after constraint
    // iterate over all phase constraints in the runs after constraint list
    for phsname in phs.runsAfter
        // create a node from the phase constraint name and connect with soft edge
        tonode <- graph.getNodeByName( phsname )
        graph.softConnectNodes( fromnode, tonode )
    // iterate over all phase constraints in the runs before constraint list
    for phsname in phs.runsBefore
        // create a node from the phase constraint name and connect with soft edge
        tonode <- graph.getNodeByName( phsname )
        graph.softConnectNodes( tonode, fromnode )
return graph

2.3.2 The removeDanglingNodes function

This function will given a dependency graph find all nodes that have no reference to a phase object and remove these nodes from the graph. (See section 1.2.2) These nodes that have no phase object are generated from runs after, runs before or runs right after constraints to phases that are not loaded. It is done as follows:

- create a list L of all nodes in the given graph that have no phase object
- for each node in L
  - remove the node from the graph
  - for each edge in the nodes before list, remove this edge from the graph

Below is shown a pseudo code version of the function.

function removeDanglingNodes( graph )
    // find all nodes with no phase object
dnodes <- filter( lambda node: node.phaseobj == null, graph.nodes )
    // iterate over these nodes and remove them one by one
    for node in dnodes
        graph.nodes.remove( node )
        // also remove all edge objects related to this node object
        for edge in node.before
            graph.edges.remove( edge )
            edge.frm.after.remove( edge )

2.3.3 The validateAndEnforceHardlinks function

First this function will given a dependency graph find all edges in the graph that are marked as hard and check that the size of the after list in the from nodes of the edge only contains the hard link, if there are more edges in the list there was a before constraint on a phase with a run right after dependency.
Second this function will given a dependency graph find all edges in the graph that are marked as hard. It will then ensure that the to node the edge is pointing to only has this edge in its before list. (See section 1.2.7). It works as follows:

- create list \( L \) of all edges in the given graph that are marked as hard
- for each edge in \( L \)
  - If the size of the edge.frm.after list is bigger then 1, throw a fatal error
- loop until no edges are moved
  - create list \( L \) of all edges in the given graph that are marked as hard
  - for each edge in \( L \)
    * create list \( HE \) of edges marked as hard in the before list of the node object this edge is pointing to
    * if there is more then one element in \( HE \) it will not be possible to generate a compiler phase list and a fatal error is generated
    * if there is only one element in \( HE \), create a list \( E \) of all the edges not marked as hard in the deps list of the node object this edge is pointing to
    * override the before list of the node object this edge is pointing to with the \( HE \) list
    * for each pedge in \( E \)
      - attach the pedge to the before list of the node object this edge is pointing from

Below is shown a pseudo code version of the function.

```python
function validateAndEnforceHardlinks( graph )
    // find all edges in the graph that are marked as hard
    hardlinks <- filter( lambda e: e.hard, graph.edges )
    // iterate over all hard edges found
    for edge in hardlinks
        if edge.frm.after.size > 1
            // Generate fatal error
            do
                // lets assume that no edges are moved until proven otherwise
                rerun <- \textbf{false}
                // find all edges in the graph that are marked as hard
                hardlinks <- filter( lambda e: e.hard, graph.edges )
                // iterate over all hard edges found
                for edge in hardlinks
                    // find all hard edges in the before list of the node this edge is pointing to
                    sanity <- filter( lambda e: e.hard, edge.to.before )
                    if sanity.length > 1
                        // generate fatal error, there is more then one runs right
                        // after constraint on the same node
                        else
                            // find all non hard edges in the deps list of the node
                            // this edge points to
                            promote <- filter( lambda e: not e.hard, edge.to.before )
                            // assign the list of hard edges to the deps list of the node this edge points to
```

19
edge.to.deps <- sanity
// iterate over all non hard edges and attach them to the before list of the node this edge points from
for pedge in promote
  // we have moved an edge - we need to rerun this to check
  rerun <- true
  // move the edge object down along the hard edge
  pedge.to <- edge.frm
  edge.frm.deps.attach(pedge)
// if edges where moved, then rerun to make sure the structure is valid
while rerun

To better understand this little function, let’s look at some diagrams and how this function works on these examples.

The first example is shown below. Here phase B wants to run right after phase A and phase C just wants to run somewhere after phase A, so there is no problem in pushing phase C down to run after phase B. The function does this be first finding all the hard edges (the blue edges). It then inspects the edges one by one and in this example we have only one hard edge. The edge goes from phase B to phase A, meaning that phase B want to run right after phase A. The algorithm now looks at the before list of phase A and finds that there is only one hard edges (this is very good). The algorithm now filters out the non-hard edges and finds the edge to phase C. This edge is now detached from phase A and attached to phase B.

With the same arguments can we also handle the situation shown below.

The first check in the function will disallow situation like the one shown below.
2.3.4 The collapseHardLinksAndLevels function

This is a recursive function that initially is given the graph, the root node and the initial level. This function will visit all nodes in the graph and test if there are any cycles, collapse the hard links and set a level in all nodes. It works as follows:

- the function is called with the graph, a node and a level
- if the node is already visited, then a cycle is found and a fatal error is thrown
- if the level of the node is lower then the level the function is called with, then update the level of the node to the level the function is called with
- find all edges marked as hard (hard links) in the before list of the node and save the list to L
- repeat while size of L is greater then zero
  - foreach edge in L
    * update the phaseobj list of the node with the concatenation of the phaseobj with from the node with the phaseobj list from the node in the edge from pointer
    * update the before list of the node with the before list of the node in the edge from pointer
    * remove the node of the edge from pointer from the graph
    * remove the edge from the graph
    * update all edges in the before list of the node to have its to pointer to point the node
  - find all edges marked as hard (hard links) in the before list of the node and save the list to L
- mark the node as visited
- for each edge in the before list of the node
  - call recursively with the graph, the from node of the edge and level plus one
- mark the node as not visited

Below is shown a pseudo code version of the function.

```plaintext
function collapseHardLinksAndLevels( graph, node, lvl )
  // Simple cycle test, if the node is visited, there is a cycle
  if node.visited == true
    // There is a cycle in the graph, throw fatal error
    return

  // Assign level to node, if the level of the node is lower
  if node.level < lvl
    node.level <- lvl

  // Filter hard edges from nodes.before and create a list from them
  hls <- filter( lambda e: e.hard, node.before )

  // Repeat as long as there are hard links in the before list
  while hls.size > 0
```
// Iterate over the hard edges, there should be only one
for edge in hls
    // Combine the phase object lists
    node.phaseobj <- node.phaseobj.extend( edge.frm.phaseobj )
    // Override node before list with the edge.frm before list
    node.before <- edge.frm.before
    // Remove node and edge from graph
    graph.nodes.remove( edge.frm.phasename )
    graph.edges.remove( edge )
    // Update all edges in the before list to this node
    for e in node.before
        e.to <- node
    // Filter out hard links again
    hls <- filter( lambda e: e.hard, node.before)
    // Mark this node as visited
    node.visited <- true
    // Call function recursively with lvl + 1
    for edge in node.before
        collapseHardLinksAndLevels( graph, edge.frm, lvl + 1 )
    // Mark this node as not visited anymore
    node.visited <- false

2.3.5 The compilerPhaseList function

This function will given a graph that has been assigned levels to each node, transform this graph into a list. This is done by including the phase objects one level at the time. If there are multiple nodes at the same level, they are sorted alphabetically. It works as follows:

- create new empty list L
- create variable lvl and assign 1
- extract all nodes from the graph at lvl and store the list in nodes
- while nodes not empty
  - sort the nodes list alphabetically by the phase names
  - for each node in nodes
    * update L with the phase objects from node
  - increment lvl by one
  - extract all nodes from the graph at lvl and store the list in nodes
- L is now the complete compiler chain

Below is shown a pseudo code version of the function.

function compilerPhaseList( graph )
    // Create new empty list to store the compiler chain
    chain <- new list
    // Create lvl and assign 1
    lvl <- 1
    // Find all nodes in the graph at a specific lvl
    nodes <- filter( lambda n: n.level == lvl, graph.nodes )
2.4 Naming Scheme for Internal and External Phases

Having the ability to load a large number of plug-ins, where each plug-in can supply several named phases, gives a high possibility of name clashes. There is no way to enforce unique names in plug-in supplied phases because they can have inter dependencies. The best solution is to suggest a naming scheme for phase names. This naming scheme composes all names from the package/plug-in name, then a double colon :: and then the actual phase name. So for the internal compiler phase names this would be:

```scala
val phaseName = "nsc::tailcalls"
```

For a plug-in with name unit that supplies a phase with name convert the full phase name would be:

```scala
val phaseName = "unit::convert"
```

By using this naming scheme for all phases there is a namespace for phase names within each plug-in and external phases will never clash with internal phase names.

2.5 Implications of This Proposal

All plug-ins need to be updated to the design in this proposal. This means that all current plug-ins will not be able to load. However the modifications to the existing plug-ins are minor. Current variables and types:

```scala
val runsAfter: String
```

The proposed changes are:

```scala
val runsAfter: List[String]
```

All internal phases in the compiler need to declare there ordering. This means that instead of having the ordering very explicit in the list structure, the ordering is encoded into each phase object. The following signatures will be added to the SubComponent class.

```scala
val runsAfter: List[String]
val runsRightAfter: Option[String]
```

The following items must be added to each phase object to implement the signatures (however they must be adapted to the individual phase object).

```scala
val runsAfter = List[String]("phasename")
val runsRightAfter = None
```

There is no reason to change the names of the phases as suggested in the section above, but it will minimize the chance of name clashes and will also minimize the chance of accidental dependencies.
3 Implementation

An example implementation has been created that follows the algorithm sketched in this proposal and models the compilers internal structure. This implementation can be found in the file sip-00002-10.scala or seen in section A. The example implementation uses the phases from use case in section 1.2.10 as its data.

Running the example code

- Compile the code using `scalac sip-00002-10.scala`
- Run the example using `scala depgraph.DepGraphTest`
- This will generate the console output as seen below and Graphviz dot files that can be compiled into png images. These png images are also shown below.

Console output

```
$ scalac sip-00002-10.scala
$ scala depgraph.DepGraphTest
[dropping depend on node with no phase: nsc::msil]
[promote the dependency of plug2::optimization2: nsc::tailcalls => nsc::erasure]
[promote the dependency of plug2::optimization2: nsc::erasure => nsc::cleanup]
  - nsc::parser
  - nsc::typer
  - nsc::pickler
  - plug1::optimization1
  - nsc::liftcode
  - plug2::optimization1
  - nsc::tailcalls
  - nsc::erasure
  - nsc::cleanup
  - plug2::optimization2
  - nsc::jvm
  - nsc::terminal
$
```

PNG images of the generated dot files showing the graph structure. Internal phases are marked as black circles and plug-in phases are marked as green circles. The hard links (runsRightAfter constraints) are marked as blue arrows.
A Example implementation

/* Package information */
package depgraph

2
/* Simple import of data structures and IO */
import scala.collection.mutable.{HashSet, HashMap}
import java.io.{BufferedWriter, FileWriter}

/* Class made to model the internal compiler phases
* All internal phases inherits from SubComponent */
abstract class SubComponent {
  val phaseName: String
  val runsAfter: List[String]
  val runsBefore: List[String] = List[String]()
  val runsRightAfter: Option[String]
  val internal: Boolean = true
  override def hashCode() = phaseName.hashCode()
}

/*
*/
class Global extends Plugins with PhaseAssembly {
  /* Simple option value to hold the compiler phase chain */
  private var phasesCache : Option[List[SubComponent]] = None

  /* The set of phase objects that is the basis for the compiler phase chain */
  protected val phasesSet : HashSet[SubComponent] = new HashSet[SubComponent]

  /* All the internal phase objects */

  object parser extends {
    val phaseName = "nsc::parser"
    val runsAfter = List[String]()
    val runsRightAfter = None
  } with SubComponent

  object typer extends {
    val phaseName = "nsc::typer"
    val runsAfter = List[String]("nsc::parser")
    val runsRightAfter = None
  } with SubComponent

  object pickler extends {
    val phaseName = "nsc::pickler"
    val runsAfter = List[String]("nsc::typer")
    val runsRightAfter = None
  } with SubComponent

  26
object liftcode extends {
  val phaseName = "nsc::liftcode"
  val runsAfter = List[String]("nsc::pickler")
  val runsRightAfter = None
} with SubComponent

object tailcalls extends {
  val phaseName = "nsc::tailcalls"
  val runsAfter = List[String]("nsc::pickler","nsc::liftcode")
  val runsRightAfter = None
} with SubComponent

object erasure extends {
  val phaseName = "nsc::erasure"
  val runsAfter = List[String]()
  val runsRightAfter = Some("nsc::tailcalls")
} with SubComponent

object cleanup extends {
  val phaseName = "nsc::cleanup"
  val runsAfter = List[String]()
  val runsRightAfter = Some("nsc::erasure")
} with SubComponent

object jvm extends {
  val phaseName = "nsc::jvm"
  val runsAfter = List[String]("nsc::cleanup")
  val runsRightAfter = None
} with SubComponent

object terminal extends {
  val phaseName = "nsc::terminal"
  val runsAfter = List[String]("nsc::jvm","nsc::msil")
  val runsRightAfter = None
} with SubComponent

/* Helper method */
private def computePhaseDescriptors: List[SubComponent] = {
  computeInternalPhases() // Global.scala
  computePluginPhases() // plugins/Plugins.scala
  buildCompilerFromPhasesSet() // PhaseAssembly.scala
}

/* Will add the internal compiler phases to the phases set */
protected def computeInternalPhases() {
  phasesSet += parser
  phasesSet += typer
  phasesSet += pickler
  phasesSet += liftcode

phasesSet += tailcalls
phasesSet += erasure
phasesSet += cleanup
phasesSet += jvm
phasesSet += terminal
}

/* Getter method for the compiler phases chain */
def phaseDescriptors : List[SubComponent] = {
  if (phasesCache.isEmpty) {
    phasesCache = Some(computePhaseDescriptors)
  }
  phasesCache.get
}

/* Class make to model the Plug-in supplied phases */
/* All external phases inherits from PluginComponent */
abstract class PluginComponent extends SubComponent {
  final override val internal = false
  val runsRightAfter: Option[String] = None
}

/* Trait made to model the behavior of the plugins */
trait Plugins { self: Global =>

  /* Example plugin phases */
  object plugin1 extends {
    val phaseName = "plug1::optimization1"
    val runsAfter = List[String]("nsc::typer")
  } with PluginComponent

  object plugin2 extends {
    val phaseName = "plug2::optimization1"
    val runsAfter = List[String]("plug1::optimization1")
    override val runsBefore = List[String]("nsc::tailcalls")
  } with PluginComponent

  object plugin3 extends {
    val phaseName = "plug2::optimization2"
    val runsAfter = List[String]("plug2::optimization1", "nsc::tailcalls")
    override val runsBefore = List[String]("nsc::jvm")
  }
```scala
/* Add plug-in supplied phase objects to the phases set */
def computePluginPhases() {
  phasesSet += plugin1
  phasesSet += plugin2
  phasesSet += plugin3
}

/* Trait made to separate the constraint solving from the rest of the compiler */
trait PhaseAssembly { self: Global =>

  /* Aux datastructure for solving the constraint system */
  /* Simple edge with to and from refs */
  class Edge(f: Node, t: Node, h: Boolean) {
    var frm = f
    var to = t
    var hard = h
  }

  /* Aux datastructure for solving the constraint system */
  /* Simple node with name and object ref for the phase object, */
  /* also sets of in and out going dependencies */
  class Node(name: String) {
    var phasename = name
    var phaseobj: Option[List[SubComponent]] = None
    var after = new HashSet[Edge]()
    var before = new HashSet[Edge]()
    var visited = false
    var level = 0
  }

  /* Aux datastructure for solving the constraint system */
  /* The dependency graph container with helper methods for node and edge creation */
  class DependencyGraph {
    val nodes = new HashMap[String, Node]()
    val edges = new HashSet[Edge]()

    /* Given a phase object, get the node for this phase object. If the */
    /* node object does not exist, then create it. */
  }
```

29
def getNodeByPhase(phs : SubComponent) : Node = {
    var node : Node = getNodeByPhase(phs.phaseName)
    node.phaseobj match {
        case None =>
            node.phaseobj = Some(List[SubComponent](phs))
            case _ => null
            node
    }

    // Given the name of a phase object, get the node for that name. If the
    // node object does not exits, then create it.
    */
    def getNodeByPhase(name : String) : Node = {
        var node : Node = null
        this.nodes.get(name) match {
            case None =>
                node = new Node(name)
                nodes += (name->node)
            case Some(n) =>
                node = n
            }
            node
        }

        // Connect the frm and to nodes with an edge and make it soft.
        // Also add the edge object to the set of edges, and to the dependency
        // list of the nodes
        */
        def softConnectNodes(frm: Node, to: Node) {
            var e = new Edge(frm, to, false)
            this.edges += e
            frm.after += e
            to.before += e
        }

        // Connect the frm and to nodes with an edge and make it hard.
        // Also add the edge object to the set of edges, and to the dependency
        // list of the nodes
        */
        def hardConnectNodes(frm: Node, to: Node) {
            var e = new Edge(frm, to, true)
            this.edges += e
            frm.after += e
            to.before += e
        }
    }

    // Given the entire graph, collect the phase objects at each level, where the phase
names are sorted alphabetical at each level, into the compiler phase list
private def compilerPhaseList(graph : DependencyGraph) : List[SubComponent] = {
    var chain : List[SubComponent] = Nil

    var lvl = 1
    var nodes = graph.nodes.values.filter(n => (n.level == lvl)).toList
    while(nodes.size > 0) {
        nodes = nodes.sort((n1,n2) => (n1.phasename compareTo n2.phasename) < 0)
        for( n <- nodes ) {
            chain = chain ::: n.phaseobj.get
            lvl = lvl + 1
            nodes = graph.nodes.values.filter(n => (n.level == lvl)).toList
        }
    }
    return chain
}

/* Test if there are cycles in the graph, assign levels to the nodes
 and collapse hard links into nodes
*/
private def collapseHardLinksAndLevels(graph : DependencyGraph, node : Node, lvl: Int) : Unit = {
    if (node.visited) {
        println("Cyclic dependency detected, abort!")
        System.exit(0)
    }
    if (node.level < lvl) node.level = lvl
    var hls = Nil ++ node.before.filter( e => e.hard )
    while( hls.size > 0 ) {
        for( hl <- hls ) {
            node.phaseobj = Some(node.phaseobj.get ++ hl.frm.phaseobj.get)
            node.before = hl.frm.before
            graph.nodes -= hl.frm.phasename
            graph.edges -= hl
            for( edge <- node.before ) edge.to = node
            hls = Nil ++ node.before.filter( e => e.hard )
        }
    }
    node.visited = true
    for( edge <- node.before ) {
        collapseHardLinksAndLevels( graph, edge.frm, lvl + 1)
    }
    node.visited = false
}

/* Find all edges in the given graph that are hard links. For each hard link we
private def validateAndEnforceHardlinks(graph : DependencyGraph) : Unit = {
    var hardlinks = graph.edges.filter(e => e.hard)
    for(hl <- hardlinks) {
        if (hl.frm.after.size > 1) {
            println("error: phase " + hl.frm.phasename + " want to run right after " + hl.to.phasename + ", but some phase has declared to run before " + hl.frm.phasename)
            System.exit(0)
        }
    }
    var rerun = true
    while(rerun) {
        rerun = false
        hardlinks = graph.edges.filter(e => e.hard)
        for(hl <- hardlinks) {
            var sanity = Nil ++ hl.to.before.filter(e => e.hard)
            if (sanity.length == 0) {
                println("This is not supposed to happen!")
                System.exit(1)
            } else if (sanity.length > 1) {
                println("Multiple phases want to run right after the same phase")
                println("Phases:")
                for (edge <- sanity) {
                    println(" - " + edge.frm.phasename)
                }
                System.exit(1)
            } else {
                var promote = hl.to.before.filter(e => (!e.hard))
                hl.to.before.clear
                sanity foreach (edge => hl.to.before += edge)
                for (edge <- promote) {
                    rerun = true
                    println("[promote the dependency of " + edge.frm.phasename + ": " + edge.to.phasename + "] => " + hl.frm.phasename)
                    edge.to = hl.frm
                    hl.frm.before += edge
                }
            }
        }
    }
}
private def removeDanglingNodes(graph : DependencyGraph) : Unit = {
  var dnodes = graph.nodes.values.filter(n => (n.phaseobj match {case None => true case Some(n) => false }))
  for(node <- dnodes) {
    println("[dropping depend on node with no phase: " + node.phasename + "]")
    graph.nodes -= node.phasename
    for(edge <- node.before) {
      graph.edges -= edge
      edge.frm.after -= edge
    }
  }
}

private def phasesSetToDepGraph(phsSet : HashSet[SubComponent]) : DependencyGraph = {
  val graph = new DependencyGraph()
  for(phs <- phsSet) {
    var fromnode = graph.getNodeByPhase(phs)
    phs.runsRightAfter match {
      case None =>
        for(phsname <- phs.runsAfter) {
          if (! (phsname equals "terminal")) {
            var tonode = graph.getNodeByPhase(phsname)
            graph.softConnectNodes(fromnode, tonode)
          } else {
            println("[after depends on terminal not allowed, dropping depend: " + fromnode.phasename + " => "+ phsname + "]")
          }
        }
        for(phsname <- phs.runsBefore) {
          if (! (phsname equals "parser")) {
            var tonode = graph.getNodeByPhase(phsname)
            graph.softConnectNodes(tonode, fromnode)
          } else {
            println("[before depends on parser not allowed, dropping depend: " + phsname + " => " + fromnode.phasename + "]")
          }
        }
      case Some(phsname) =>
        if (! (phsname equals "terminal")) {
          var tonode = graph.getNodeByPhase(phsname)
          graph.hardConnectNodes(fromnode, tonode)
        } else {
          println("[right after depends on terminal not allowed, dropping depend: " + fromnode.phasename + " => "+ phsname + "]")
        }
    }
  }
}

/* Given the phases set, will build a dependency graph from the phases set
 * Using the aux. method of the DependencyGraph to create nodes egdes
 */
private def phasesSetToDepGraph(phsSet : HashSet[SubComponent]) : DependencyGraph = {
  val graph = new DependencyGraph()
  for(phs <- phsSet) {
    var fromnode = graph.getNodeByPhase(phs)
    phs.runsRightAfter match {
      case None =>
        for(phsname <- phs.runsAfter) {
          if (! (phsname equals "terminal")) {
            var tonode = graph.getNodeByPhase(phsname)
            graph.softConnectNodes(fromnode, tonode)
          } else {
            println("[after depends on terminal not allowed, dropping depend: " + fromnode.phasename + " => "+ phsname + "]")
          }
        }
        for(phsname <- phs.runsBefore) {
          if (! (phsname equals "parser")) {
            var tonode = graph.getNodeByPhase(phsname)
            graph.softConnectNodes(tonode, fromnode)
          } else {
            println("[before depends on parser not allowed, dropping depend: " + phsname + " => " + fromnode.phasename + "]")
          }
        }
      case Some(phsname) =>
        if (! (phsname equals "terminal")) {
          var tonode = graph.getNodeByPhase(phsname)
          graph.hardConnectNodes(fromnode, tonode)
        } else {
          println("[right after depends on terminal not allowed, dropping depend: " + fromnode.phasename + " => "+ phsname + "]")
        }
    }
  }
}
/* Method called from computePhaseDescriptors in class Global */
def buildCompilerFromPhasesSet(): List[SubComponent] = {

    // Add all phases in the set to the graph
    val graph = phasesSetToDepGraph(phasesSet)

    // graphToDotFile(graph, "depgraph1.dot")

    // Remove nodes without phaseobj
    removeDanglingNodes(graph)

    // graphToDotFile(graph, "depgraph2.dot")

    // Validate and Enforce hardlinks / runsRightAfter and promote nodes down the tree
    validateAndEnforceHardlinks(graph)

    // graphToDotFile(graph, "depgraph3.dot")

    // test for cycles, assign levels and collapse hard links into nodes
    collapseHardLinksAndLevels(graph, graph.getNodeByPhase("nsc::parser"), 1)

    // graphToDotFile(graph, "depgraph4.dot")

    // assemble the compiler
    return compilerPhaseList(graph)
}

/* This is a helper method, that given a dependency graph will generate a graphviz dot * file showing its structure.
* Plug-in supplied phases are marked as green nodes and hard links are marked as blue edges.
*/
private def graphToDotFile(graph: DependencyGraph, filename: String): Unit = {
    var sbuf = new StringBuffer()
    var extnodes = new HashSet[Node]()
    var fatnodes = new HashSet[Node]()
    sbuf.append("digraph G {
"
    for (edge <- graph.edges) {
        sbuf.append("" + edge.frm.phasename + "(" + edge.frm.level + ")" + "\"->\"" + edge.to.phasename
        if (! edge.frm.phaseobj.get.first.internal) {
            extnodes += edge.frm
        }
    }
edge.frm.phaseobj match { case None => null case Some(ln) => if(ln.size > 1) fatnodes += edge.frm
edge.to.phaseobj match { case None => null case Some(ln) => if(ln.size > 1) fatnodes += edge.to }
if (edge.hard) {
  sbuf.append(" [color=\"#0000ff\"]\n")
} else {
  sbuf.append(" [color=\"#000000\"]\n")
}
}
for(node <- extnodes) {
  sbuf.append("\n" + node.phasename + "(" + node.level + ")" + "\n [color=\"#00ff00\"]\n")
}
for(node <- fatnodes) {
  sbuf.append("\n" + node.phasename + "(" + node.level + ")" + "\n [color=\"#0000ff\"]\n")
}
sbuf.append("\n")
var out = new BufferedWriter(new FileWriter(filename))
out.write(sbuf.toString)
out.flush()
out.close()

/**
 * Test object that will create a new object from the Global class
 * and call the method phaseDescriptors to get the list of phase objects
 * and print the phase names to stdout
 *
*/
object DepGraphTest extends Application {
  val global = new Global()
  var compilerchain = global.phaseDescriptors
  for(phase <- compilerchain) {
    println(" - " + phase.phasename)
  }
}