1 General Concepts

- Specialized data structures for each DSL level

  L1 DSL level 1

  L2 DSL level 2

  L3 DSL level 3

  L4 DSL level 4

  IR Intermediate representation: for optimizations, hardware specialization, ...

  PP optional: Prettyprinting: Further transformations may be needed for prettyprinting in a specific language

- Collaboration between different groups and users via data structures in program state
- SPL / domain knowledge to be accessible compiler-wide (like a library)
- A central instance keeps track of program state changes: StateManager
- Prettyprinting implemented via internal methods
- Nodes in program state may be annotated
- Clear sectioning of functionality into namespaces, e. g.,
  - `exastencils.core`: Log functionality, `StateManager`, compiler settings
  - `exastencils.core.collectors`
  - `exastencils.datastructures`: Annotations, Program state duplication, `trait Strategy`, `trait Transformation`
  - `exastencils.datastructures.(l1, l2, l3, l4, ir)`
  - `exastencils.parsers`
  - `exastencils.prettyprinting`

2 Transformations

- Carry an identifier
- Are grouped together in Strategies
- Are atomic – either applied completely or not at all
- Are applied to program state in depth search order
- May be applied to a part of the program state
- May contain more than one `case` statement

```
class Transformation (
  name : String,
  function : Function[Node, Transformation.Output[_]],
  recursive : Boolean = true,
  applyAtNode : Option[Node] = None
)

class Output [T <% Node Or List[Node]] (
  val inner : T
)

class TransformationResult (
  val successful : Boolean,
  val matches : Int,
  val replacements : Int
)

class Transformation (
  name : String,
  function : Function[Node, Transformation.Output[_]],
  recursive : Boolean = true,
  applyAtNode : Option[Node] = None
)
```

- `function` denotes the pattern to look for and to replace with
- Different return values possible:
  - Object for replacement (may be the same instance as in the pattern)
  - `List[_]`: replaces one node with multiple nodes (if applicable)
  - `None`: remove node (if applicable)

3 Strategies

- Carry an identifier
- Are applied in transactions
- Are applied by the StateManager
- A standard strategy that linearly executes all transformations is provided
- Custom strategies possible

```
var s = Strategy("example standard strategy")

// replace constant '1' under a certain node with '3'
s += Transformation("t1",
     { case x : Constant if(x.Value == 1)
       => Constant(3)
     }, someProgramStateNode)
```

```
// rename all variables to 'j'
s += Transformation("t2",
     { case x : Variable
       => Variable("j", x.Type)
     })

// duplicate all methods
s += Transformation("t3",
     { case x : FunctionStatement
       => List(x, FunctionStatement(
         x.returntype, x.name + "_", x.parameters, x.body))
     })
s.apply

class ExampleStrategy extends Strategy("example") {
  override def apply = {

    // acquire token
    val token = StateManager.transaction(this)

    // define and apply transformation
    val t1 = Transformation("t1",
      { case x : Constant if(x.value == 1)
        => Constant(3) })
    val result = StateManager.apply(token, t1)

    // end the transaction
    if(result.matches > 10) {
      StateManager.abort(token)
    } else {
      StateManager.commit(token)
    }
  }
}
```

4 Transactions

- Before application, a snapshot of the program state is made
- May be committed or aborted
- Can not run concurrently

5 Collectors

- May be supplied to gather further information during program state traversal
- Enable context-sensitive transformations

6 Annotations

- Allow adding information to object instances
- Optionally carry a value
- May be removed from an object again
- Every node in the program state `extends Annotatable`
- `Annotatable` may also be added to custom classes
- Annotations may be added, removed and checked for existence

```
var s = Strategy("example standard strategy")

// mark all nodes as visited
s += Transformation("mark",
     { case x => x.annotate("visited"); Some(x)
     })

// remove marker
s += Transformation("unmark",
     { case x => x.removeAnnotation("visited"); Some(x)
     })

s.apply
```

7 Collectors

- Are notified for each node visited and left during a transformation
- Report only on the program state (i. e., whatever `extends Node`)
- Allow for checks à la "this node is an indirect subnode of X"
- A stack-based default Collector is provided by the framework
- Custom Collectors need to extend a trait consisting of only 3 methods (`enter()`, `leave()`, `reset()`)
- Collectors are (de)registered via `StateManager`
- Most useful in custom strategies