Neverlang
Reusable and Evolvable DSLs

Walter Cazzola

ADAPT-Lab
Department of Computer Science
Università degli Studi di Milano
e-mail: `cazzola@di.unimi.it`

Domain Specific Languages
Why? General Purpose vs Domain-Specific Languages

We are used to use domain specific languages (DSLs)

- LaTeX to typeset scientific documents
- SQL to query relational databases
- make & ant to build up software systems

. . . but it is missing the culture to write your own DSL

Domain Specific Languages

Why? General Purpose vs Domain-Specific Languages

We are used to use domain specific languages (DSLs)

- LaTeX to typeset scientific documents
- SQL to query relational databases
- make & ant to build up software systems

... but it is missing the culture to write your own DSL

DSL benefits are evident

- problem-tailored solutions
  - i.e., solutions more concise and clear
- domain-oriented solutions
  - i.e., solutions implementable by domain experts

... but to implement them is hard!

- to develop a compiler/interpreter is long, complex and requires some skills;
- existing languages cannot be easily extended or modified; and
- there is a lack of tools easing their development

Domain Specific Languages
Look at the Obstacles

Basically, the main obstacle is

- the traditional approach to programming language implementation

Domain Specific Languages
Look at the Obstacles

Basically, the main obstacle is

- the traditional approach to programming language implementation

Compilers/Interpreters are

Monolithic and Opaque

Domain Specific Languages
Look at the Obstacles

Basically, the main obstacle is

- the traditional approach to programming language implementation

Compilers/Interpreters are

Monolithic and Opaque

Therefore, they are

- hard to extend by changing their code;
- hard to extend over them (layerization, libraries, . . .); and
- hard to reuse in the implementation of other languages

Domain Specific Languages
A Solution: Sectional Compilers

A (sectional) DSL and its compiler/interpreter are:

- fully composed from basic units, i.e., modular language definition;
- easily extensible by plugging new units in; and
- easily built from basic units written to define other sectional DSL.

To defeat the dragon, the knight needs:

- a language for writing the building blocks of the DSL
- a tool for composing the blocks together to form an ad hoc compiler/interpreter.

The Neverlang Model
The Framework Features

Neverlang framework

- It is a compilers/interpreters generator
- It provides a language to define the compiler and a tool that generates it
- It enhances the reusability of the generated compiler/interpreter to ease future modifications of the DSL.

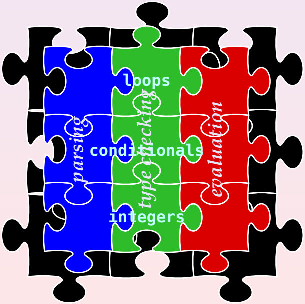
`http://neverlang.di.unimi.it`

The Neverlang Model: The Idea

Glossary

Glossary:

- we call modules with a specific role, the basic units
  - role categories correspond to available dimensions
- a dimension represents a phase of the compilation process, e.g., parsing, type checking, etc.
- a regular slice regards a particular language constructs
  - it is the composition of modules with their roles



Neverlang Syntax By Examples

Syntax Definition

Grammar Centric Approach!

Neverlang Syntax By Examples
Syntax Definition

Grammar Centric Approach!

Syntax is also used for selecting insertion point, where slices are plugged in:

- nonterminals correspond to join points
- semantic actions at nonterminals correspond to advice

Neverlang Syntax By Examples

Syntax Definition (Follows)

Syntax module

- each feature needs one;

```
module Sum {
  reference syntax {
    AddExpr ← Term;
    AddExpr ← AddExpr "+" Term;
  }
}
```

Evaluation module

- any other role refers to the syntax defined in a syntatic role;
- `eval` evaluates the semantic action during the AST visit of the corresponding dimension

```
module Sum {
  role(evaluation) {
    0 .{ eval $0; $0.value = $1.value; }.

    3 .{
      eval $4; eval $5;
      int res = (Integer) $4.value + (Integer) $5.value;
      $3.value = res;
    }.
  }
}
```

Neverlang Syntax By Examples

Syntax Definition (Follows)

Syntax module

- each feature needs one;

```
module Sum {
  reference syntax {
    AddExpr ← Term;
    AddExpr ← AddExpr "+" Term;
  }
}
```

Evaluation module

- any other role refers to the syntax defined in a syntatic role;
- `eval` evaluates the semantic action during the AST visit of the corresponding dimension

```
module Sum {
  role(evaluation) {
    0 .{ eval $0; $0.value = $1.value; }.
    3 .{
      eval $4; eval $5;
      int res = (Integer) $4.value + (Integer) $5.value;
      $3.value = res;
    }.
  }
}
```

Symmetric approach:

- no base code where aspects are woven into;
- no composition specification until later stages;
- more flexible; and
- it promotes code reuse

The Neverlang Model
Composition Process

Symmetric approach:

- no base code where aspects are woven into;
- no composition specification until later stages;
- more flexible; and
- it promotes code reuse

Composition is Twofold:

1. Composition between roles, which yields slices
2. Composition between slices, which yields the compiler/interpreter

OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

Neverlang at Work

DSL Maintenance: Case study

We want to create an administration utility to define some maintenance tasks on log files:

- the tasks are described by using a DSL;
- the language is interpreted

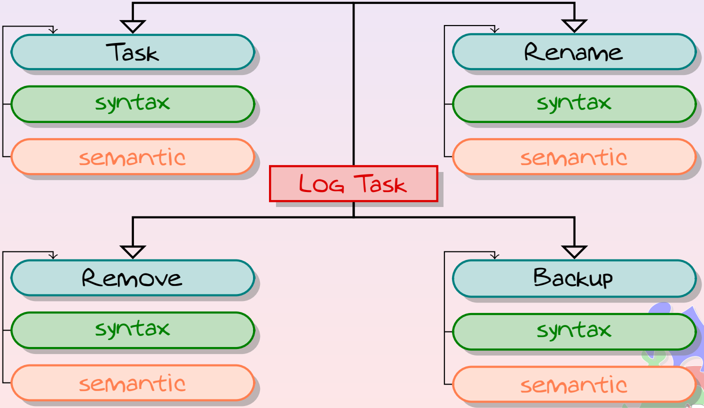
```
task TaskOne {
  remove "application.debug.old"
  rename "application.debug" "application.debug.old"
}

task TaskTwo {
  backup "access.error" "securityLogs"
  backup "system.error" "systemLogs"
}
```

Neverlang is used to realize such a small DSL

Neverlang at Work
Language Definition

To design a DSL with Neverlang, we have to:

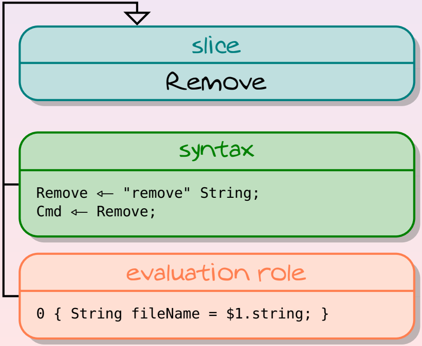
- create each single language feature (slice);
- merge the slices together to build the compiler/interpreter for the language.



Neverlang at Work
Module Composition

Each slice includes:

- the concrete syntax contains a set of grammar rules that defines the DSL syntax (reference syntax in modules);
- the «semantic» roles contain a set of semantic actions – i.e., pieces of Java code;
- the semantic actions are woven to the syntax forming the DSL semantic.



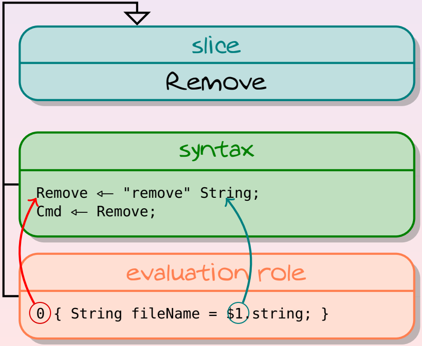
```
Remove ← "remove" String;
Cmd ← Remove;
```

```
0 { String fileName = $1.string; }
```

Neverlang at Work
Module Composition

Each slice includes:

- the concrete syntax contains a set of grammar rules that defines the DSL syntax (reference syntax in modules);
- the «semantic» roles contain a set of semantic actions – i.e., pieces of Java code;
- the semantic actions are woven to the syntax forming the DSL semantic.

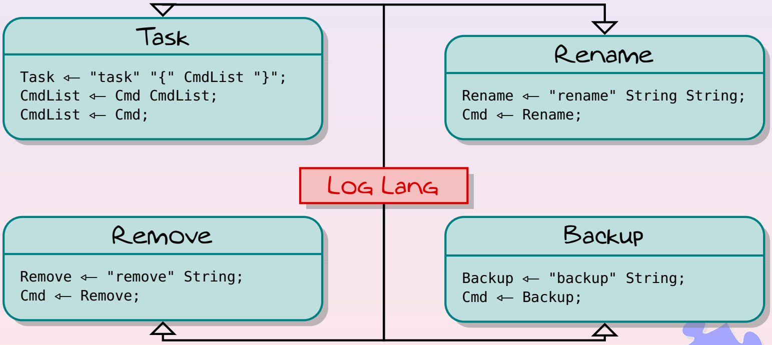


```
Remove ← "remove" String;
Cmd ← Remove;
```

```
0 { String fileName = $1.string; }
```

Neverlang at Work
Slices Composition

The slice composition is syntax driven.



Task

```
Task ← "task" "{" CmdList "}";
CmdList ← Cmd CmdList;
CmdList ← Cmd;
```

Rename

```
Rename ← "rename" String String;
Cmd ← Rename;
```

Log Lang

Remove

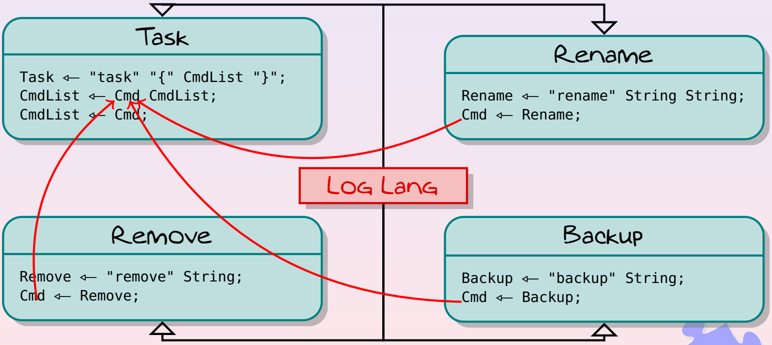
```
Remove ← "remove" String;
Cmd ← Remove;
```

Backup

```
Backup ← "backup" String;
Cmd ← Backup;
```

Neverlang at Work
Slices Composition

The slice composition is syntax driven.



Task

```
Task ← "task" "{" CmdList "}";
CmdList ← Cmd CmdList;
CmdList ← Cmd;
```

Rename

```
Rename ← "rename" String String;
Cmd ← Rename;
```

Remove

```
Remove ← "remove" String;
Cmd ← Remove;
```

Backup

```
Backup ← "backup" String;
Cmd ← Backup;
```

Neverlang at Work

Endemic Slices

The semantic actions could require some supporting code:

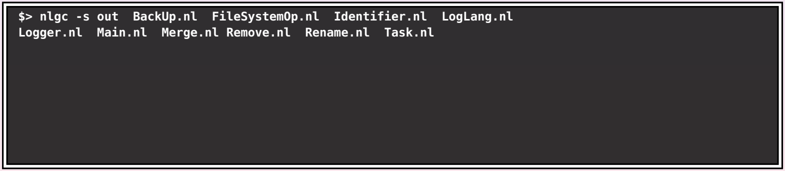
- ancillary structures are defined in the endemic slices;
- fields and methods defined in an endemic slice are accessible by all the others modules.

```
endemic slice FileOpEndemic {
    declare {
        FileOp : neverlang.examples.loglang.utils.FileOp;
    }
}
```

Language Creation and Evolution
DSL Interpreter Generation

From the slices the interpreter can be generated and used

- the files defining the slices are used to feed the generator;
- the generator creates the classes implementing the interpreter;
- nlg runs the interpreter

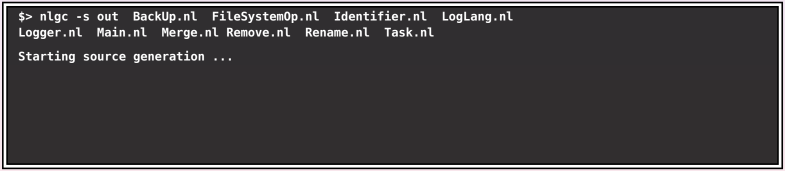


```
$> nlgc -s out  BackUp.nl  FileSystemOp.nl  Identifier.nl  LogLang.nl
Logger.nl  Main.nl  Merge.nl Remove.nl  Rename.nl  Task.nl
```

DSL Interpreter Generation

From the slices the interpreter can be generated and used

- the files defining the slices are used to feed the generator;
- the generator creates the classes implementing the interpreter;
- nlg runs the interpreter



```
$> nlgc -s out  BackUp.nl  FileSystemOp.nl  Identifier.nl  LogLang.nl
Logger.nl  Main.nl  Merge.nl Remove.nl  Rename.nl  Task.nl

Starting source generation ...
```

Language Creation and Evolution

DSL Interpreter Generation

From the slices the interpreter can be generated and used

- the files defining the slices are used to feed the generator;
- the generator creates the classes implementing the interpreter;
- nlg runs the interpreter

```
$> nlgc -s out  BackUp.nl  FileSystemOp.nl  Identifier.nl  LogLang.nl
Logger.nl  Main.nl  Merge.nl Remove.nl  Rename.nl  Task.nl

Starting source generation ...

$> javac out/**/*.java
```

DSL Interpreter Generation

From the slices the interpreter can be generated and used

- the files defining the slices are used to feed the generator;
- the generator creates the classes implementing the interpreter;
- nlg runs the interpreter

```
$> nlgc -s out  BackUp.nl  FileSystemOp.nl  Identifier.nl  LogLang.nl
Logger.nl  Main.nl  Merge.nl Remove.nl  Rename.nl  Task.nl

Starting source generation ...

$> javac out/**/*.java

$> nlg LogLang TaskList.txt
```

DSL Interpreter Generation

From the slices the interpreter can be generated and used

- the files defining the slices are used to feed the generator;
- the generator creates the classes implementing the interpreter;
- nlg runs the interpreter

```
$> nlgc -s out  BackUp.nl  FileSystemOp.nl  Identifier.nl  LogLang.nl
Logger.nl  Main.nl  Merge.nl Remove.nl  Rename.nl  Task.nl

Starting source generation ...

$> javac out/**/*.java

$> nlg LogLang TaskList.txt
Processing TaskList.txt
```

DSL Interpreter Generation

From the slices the interpreter can be generated and used

- the files defining the slices are used to feed the generator;
- the generator creates the classes implementing the interpreter;
- nlg runs the interpreter

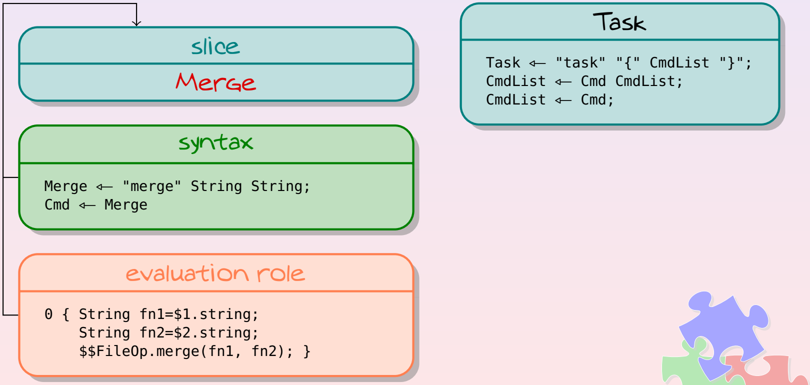
```
$> nlgc -s out  BackUp.nl  FileSystemOp.nl  Identifier.nl  LogLang.nl
Logger.nl  Main.nl  Merge.nl Remove.nl  Rename.nl  Task.nl

Starting source generation ...

$> javac out/**/*.java

$> nlg LogLang TaskList.txt
Processing TaskList.txt
......
Task Executed
```

Language Creation and Evolution

Add an operation

To add a Merge operation to the language:

- a new slice for the operation should be created;
- one of its nonterminals must be present in the rest of the grammar definition (a sort of anchor)



slice

Merge

syntax

```
Merge ← "merge" String String;
Cmd ← Merge
```

evaluation role

```
0 { String fn1=$1.string;
    String fn2=$2.string;
    $$FileOp.merge(fn1, fn2); }
```

Task

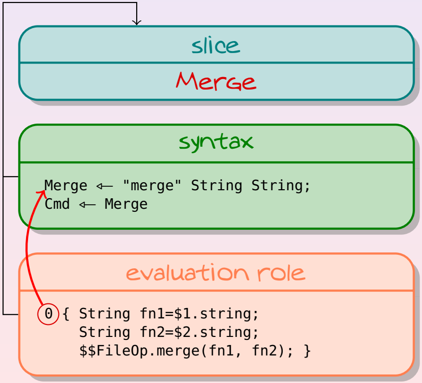
```
Task ← "task" "{" CmdList "}";
CmdList ← Cmd CmdList;
CmdList ← Cmd;
```

Language Creation and Evolution

Add an operation

To add a Merge operation to the language:

- a new slice for the operation should be created;
- one of its nonterminals must be present in the rest of the grammar definition (a sort of anchor)



slice

Merge

syntax

```
Merge ← "merge" String String;
Cmd ← Merge
```

evaluation role

```
0 { String fn1=$1.string;
    String fn2=$2.string;
    $$FileOp.merge(fn1, fn2); }
```

Task

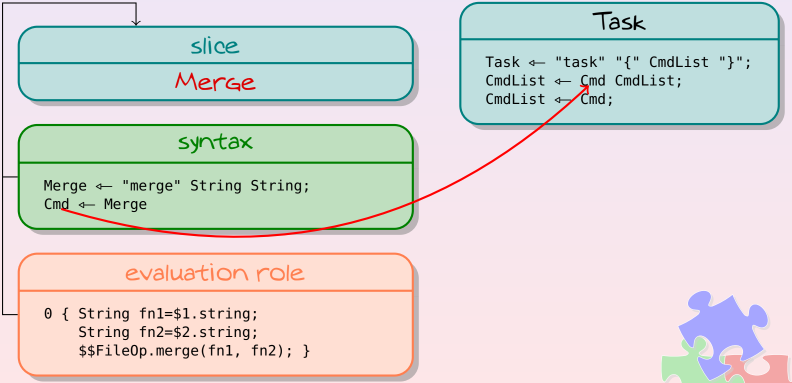
```
Task ← "task" "{" CmdList "}";
CmdList ← Cmd CmdList;
CmdList ← Cmd;
```

Language Creation and Evolution

Add an operation

To add a Merge operation to the language:

- a new slice for the operation should be created;
- one of its nonterminals must be present in the rest of the grammar definition (a sort of anchor)



slice

Merge

syntax

```
Merge ← "merge" String String;
Cmd ← Merge
```

evaluation role

```
0 { String fn1=$1.string;
    String fn2=$2.string;
    $$FileOp.merge(fn1, fn2); }
```

Task

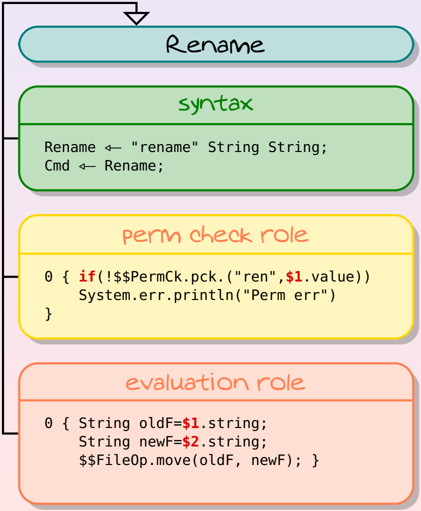
```
Task ← "task" "{" CmdList "}";
CmdList ← Cmd CmdList;
CmdList ← Cmd;
```

Language Creation and Evolution

Add a Compilation phase

To add an additional permission check:

- a new phase in the interpretation process should be defined
- to enrich each slice with a module to be used in the new phase.



Rename

syntax

```
Rename ← "rename" String String;
Cmd ← Rename;
```

perm check role

```
0 { if(!$$PermCk.pck.("ren",$1.value))
    System.err.println("Perm err")
}
```

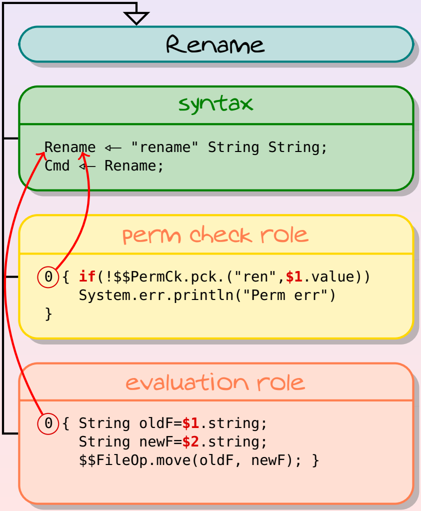
evaluation role

```
0 { String oldF=$1.string;
    String newF=$2.string;
    $$FileOp.move(oldF, newF); }
```

Add a Compilation phase

To add an additional permission check:

- a new phase in the interpretation process should be defined
- to enrich each slice with a module to be used in the new phase.



Rename

syntax

```
Rename ← "rename" String String;
Cmd ← Rename;
```

perm check role

```
0 { if(!$$PermCk.pck.("ren",$1.value))
    System.err.println("Perm err")
}
```

evaluation role

```
0 { String oldF=$1.string;
    String newF=$2.string;
    $$FileOp.move(oldF, newF); }
```

Changing the Back-end

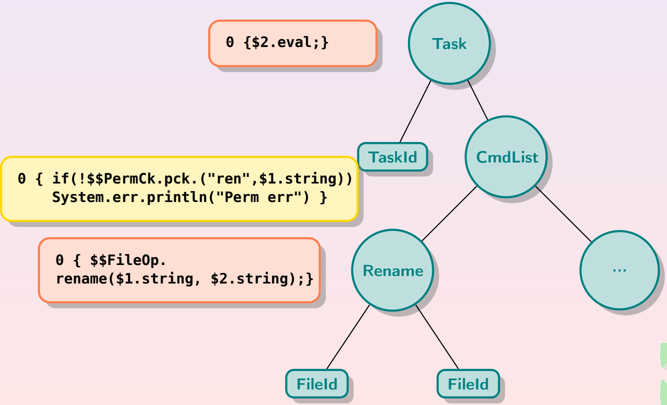
To change the underneath FS:

- as the other slices even the endemic one can be substituted;
- endemic slices represent an interface towards an external library;
- all the previous code can be reused.

```
endemic slice NetworkFileOpEndemic {
    declare {
        FileOp : neverlang.examples.loglang.remote.NetworkFileOp;
    }
}
```

Implementation Details
Generated Interpreter Structure

```
task TaskOne {
  rename File1.log File1.log.old
           ....
}
```



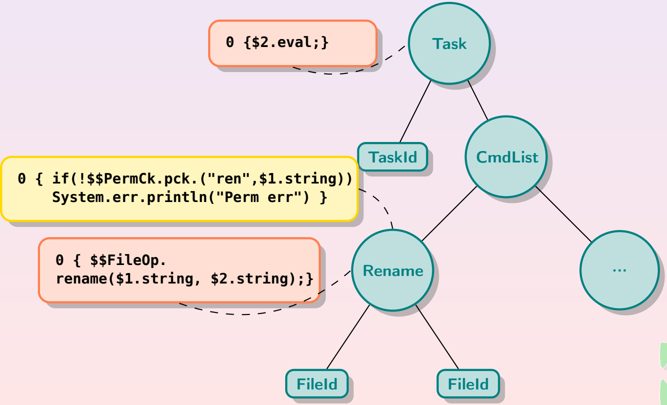
```
0 {$2.eval;}
```

```
0 { if(!$$PermCk.pck.("ren",$1.string))
    System.err.println("Perm err") }
```

```
0 { $$FileOp.
rename($1.string, $2.string);}
```

Implementation Details
Generated Interpreter Structure

```
task TaskOne {
  rename File1.log File1.log.old
            ....
}
```



```
0 {$2.eval;}
```

```
0 { if(!$$PermCk.pck.("ren",$1.string))
    System.err.println("Perm err") }
```

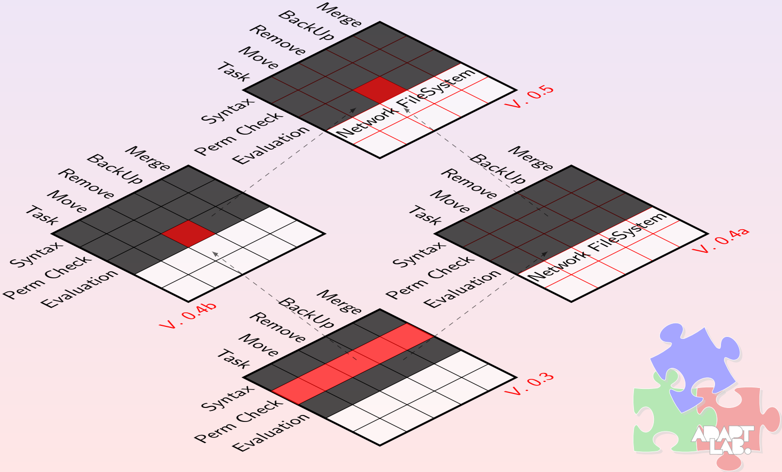
```
0 { $$FileOp.
rename($1.string, $2.string);}
```

Conclusions & Future Work

Neverlang Benefits

Neverlang can be effectively exploited:

- to easily maintain the language during the evolution of the domain
- to create new languages by reusing part of already defined programming languages.



Conclusions (Cont'd)

Our approach aims to bring aspect-oriented architecture in compiler/interpreter design effectively.

Main benefits:

- modular definition of any programming language
- easy to define a new DSL as a variant of an existing language
- thanks to symmetric composition, a wider spectrum for code reuse

Conclusions (Cont'd)

Our approach aims to bring aspect-oriented architecture in compiler/interpreter design effectively.

Main benefits:

- modular definition of any programming language
- easy to define a new DSL as a variant of an existing language
- thanks to symmetric composition, a wider spectrum for code reuse

More Details on Thursday, 4th @ 14:15

References

- Walter Cazzola.
  Domain-Specific Languages in Few Steps: The Neverlang Approach.
  In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, Proceedings of the 11th International Conference on Software Composition (SC'12), Lecture Notes in Computer Science 7306, pages 162–177, Prague, Czech Republic, 31st of May-1st of June 2012. Springer.
- Walter Cazzola and Davide Poletti.
  DSL Evolution through Composition.
  In Proceedings of the 7th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'10), Maribor, Slovenia, on 23rd of June 2010. ACM.
- Walter Cazzola and Edoardo Vacchi.
  Neverlang 2: Componentised Language Development for the JVM.
  In Walter Binder, Eric Bodden, and Welf Löwe, editors, Proceedings of the 12th International Conference on Software Composition (SC'13), Lecture Notes in Computer Science 8088, pages 17–32, Budapest, Hungary, 19th of June 2013. Springer.