boostcon
BOOST LIBRARIES CONFERENCE 2011

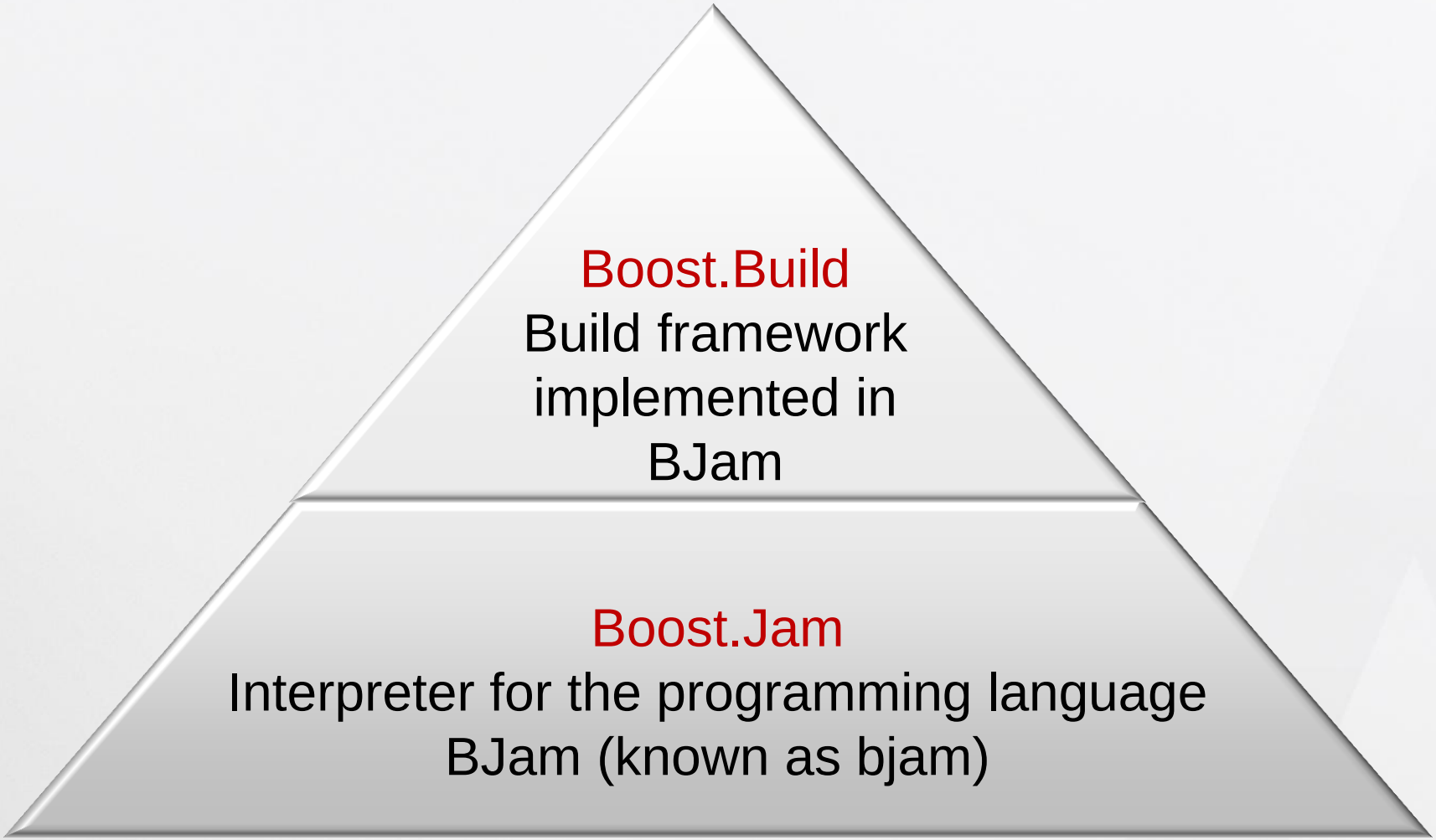
Boost.Jam + Boost.Build

Boris Schaling, May 2011, www.optiver.com

- What‘s the difference of Boost.Jam and Boost.Build?
- How are Jamfiles used to configure a project?
- How does the programming language Bjam look like?
- How does Boost.Build work?

optiver

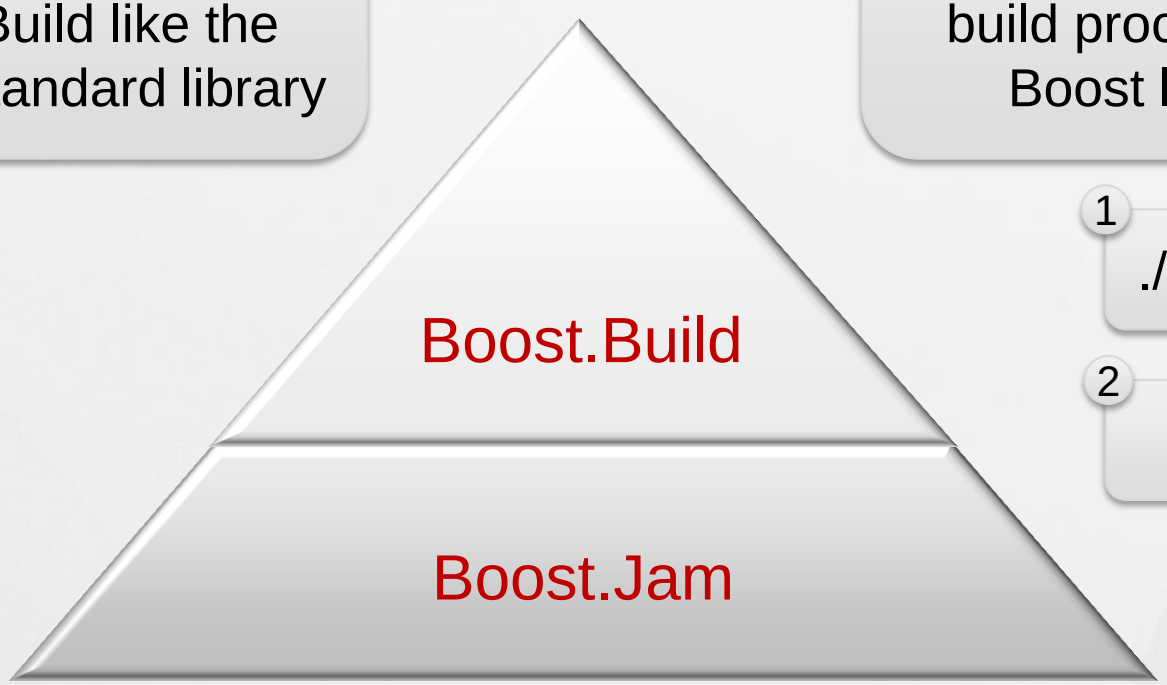
Boost.Jam vs. Boost.Build



Boost.Jam vs. Boost.Build

Boost.Jam is like the Python interpreter and Boost.Build like the Python standard library

This pyramid is the reason for the two-step build process of the Boost libraries

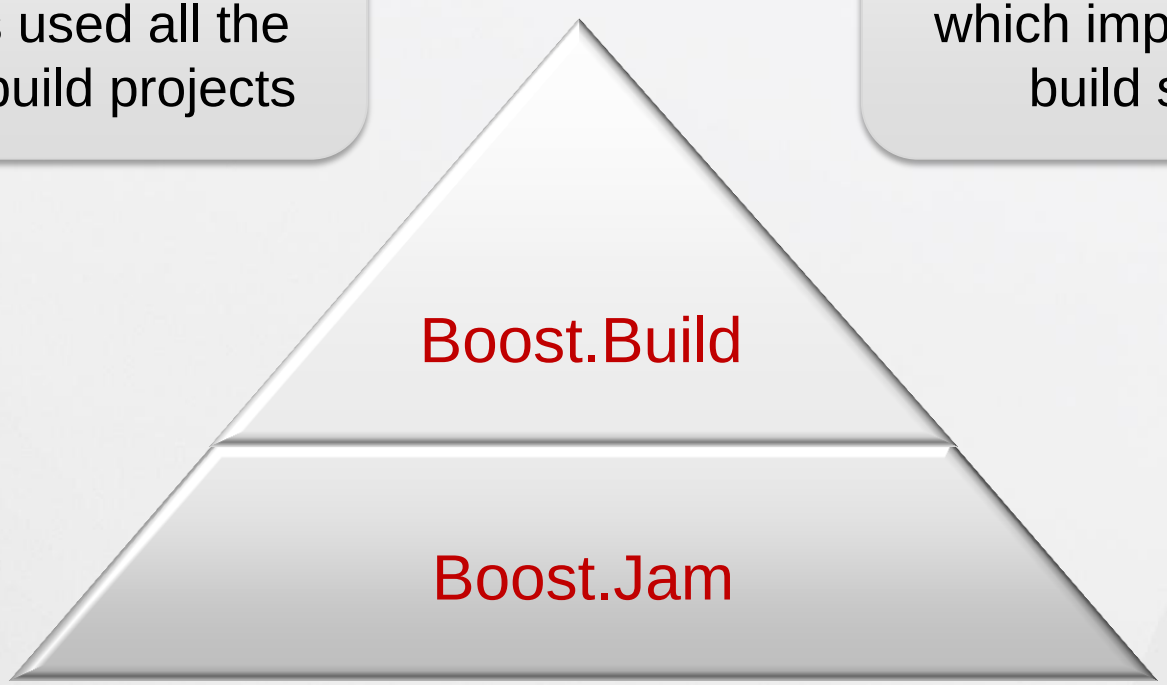


1. ./bootstrap.sh
2. ./bjam

Boost.Jam vs. Boost.Build

Think of Boost.Jam as the executable bjam which is used all the time to build projects

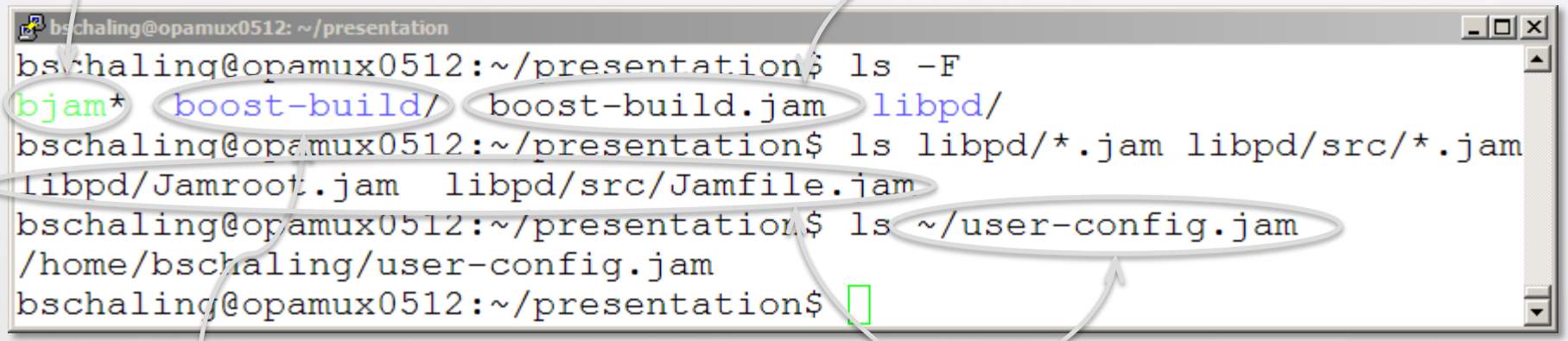
Think of Boost.Build as a bunch of Jamfiles which implement the build system



Project configuration: Structure

1 bjam executable searches for boost-build.jam in current and parent directories

2 boost-build.jam contains path to build system



3 Build system is loaded

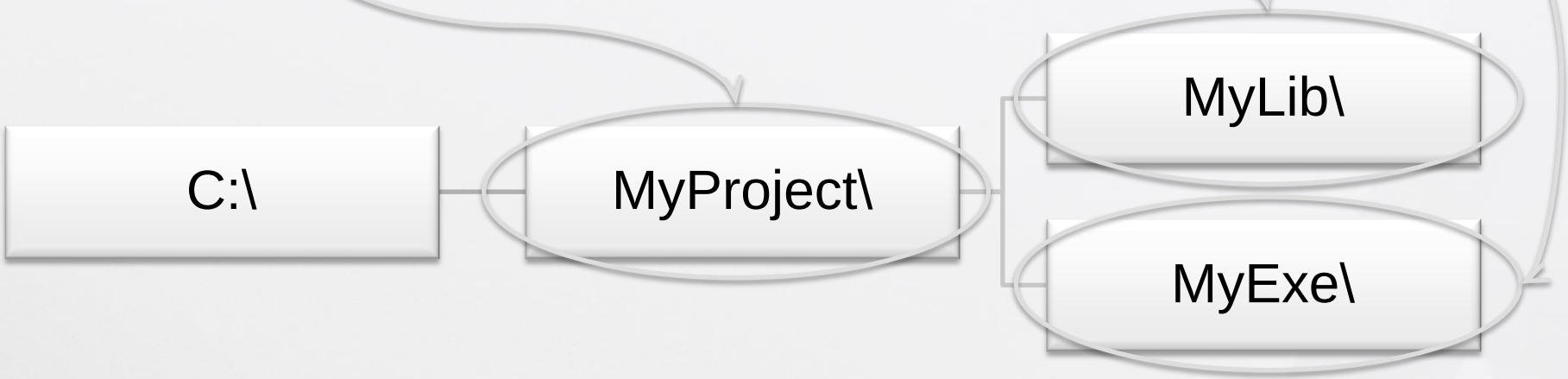
4 Optional configuration files like ~/user-config.jam are loaded

5 Project Jamfiles are loaded

Project configuration: Structure

Root directory of a project must contain Jamroot[.jam]

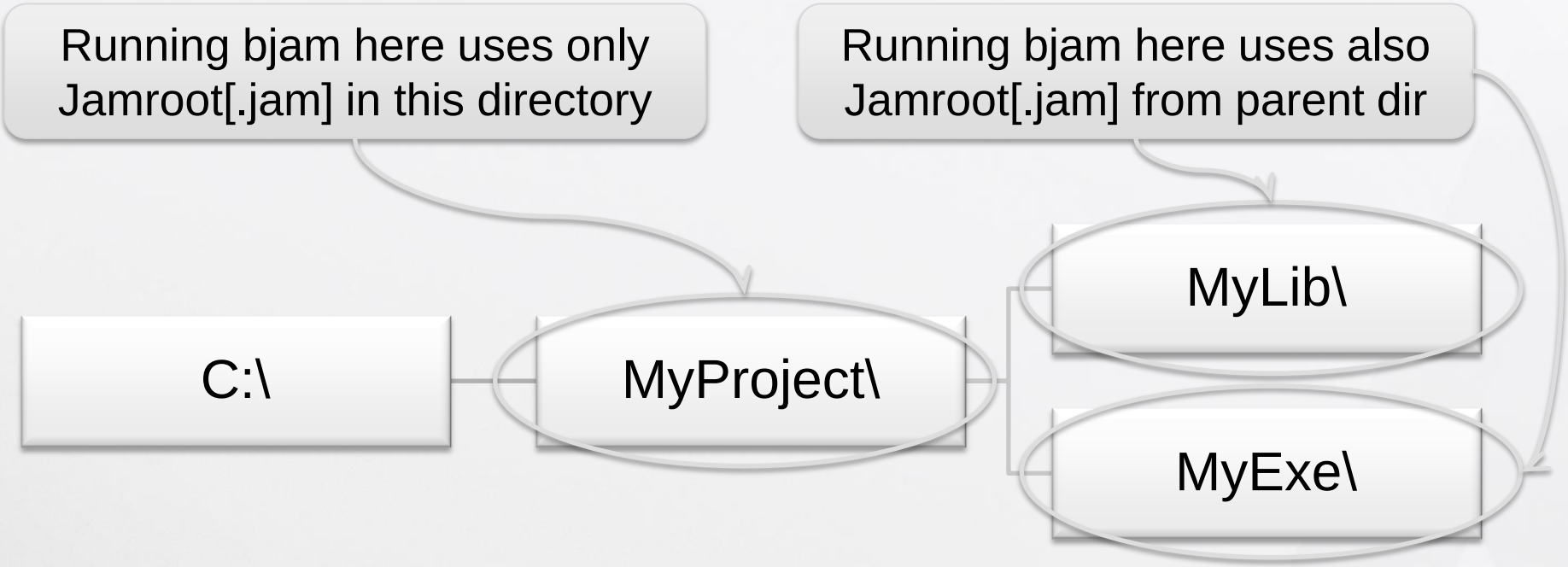
Subdirectories of a project must contain Jamfile[.jam]



A Jamfile in a directory makes the directory a project

If source code is organized in different directories, each directory with a Jamfile is a project

Project configuration: Structure



bjam can be started in any directory as long as it contains a Jamfile

bjam looks for Jamfiles in parent directories until Jamroot[.jam] is found

Project configuration: user-config.jam

bschaling@opamux0512: ~/presentation

```
using gcc ;

option jobs : 8 ;
```

user-config.jam can contain code just like any other Jamfile

Use user-config.jam to define settings for all your projects

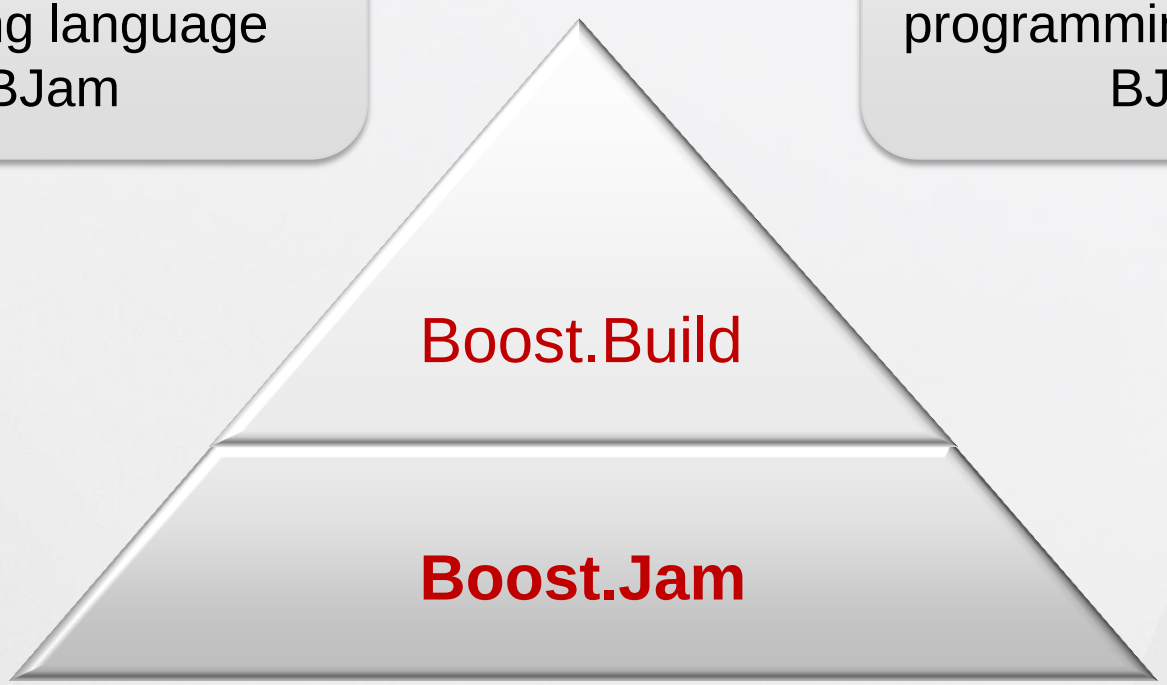
user-config.jam must be in home directory:
- /home/USERNAME on Linux
- C:\Documents and Settings\USERNAME on Windows XP

There is also a site-config.jam to define settings for all users on a system (must be in /etc on Linux and thus is only for sys admins)

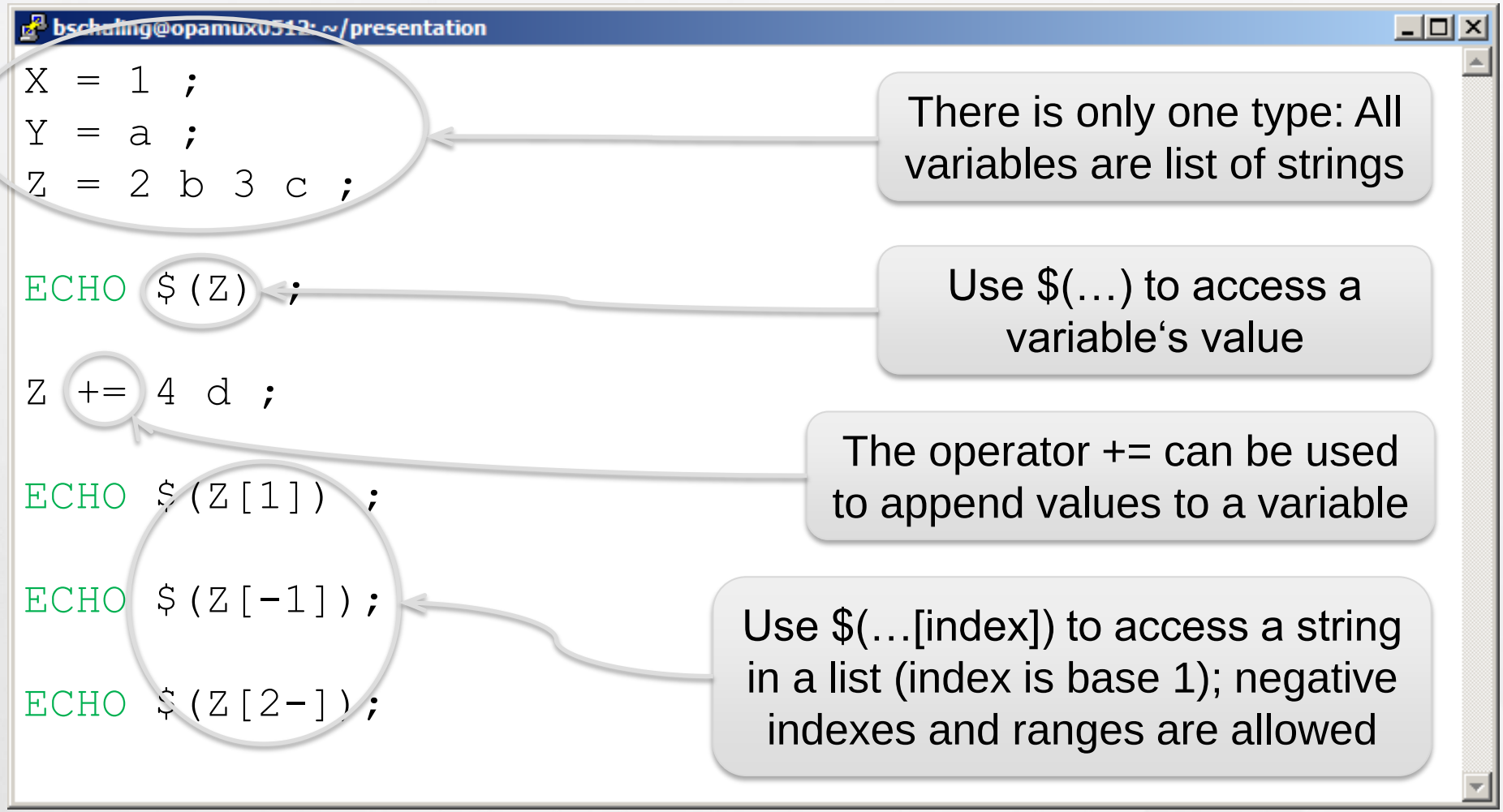
Boost.Jam

The executable bjam is an interpreter for the scripting language BJam

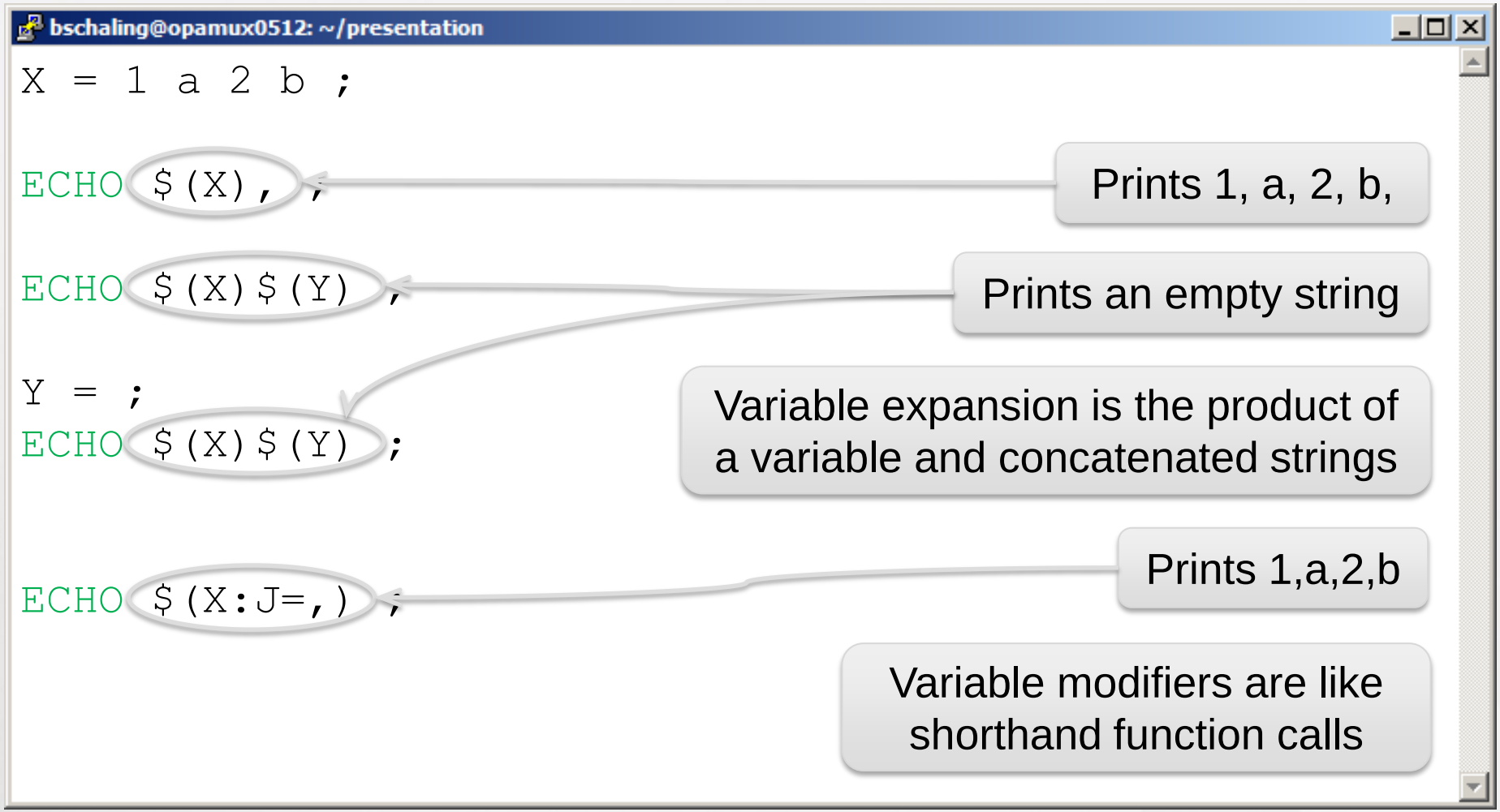
When you create Jamfiles you use the programming language BJam



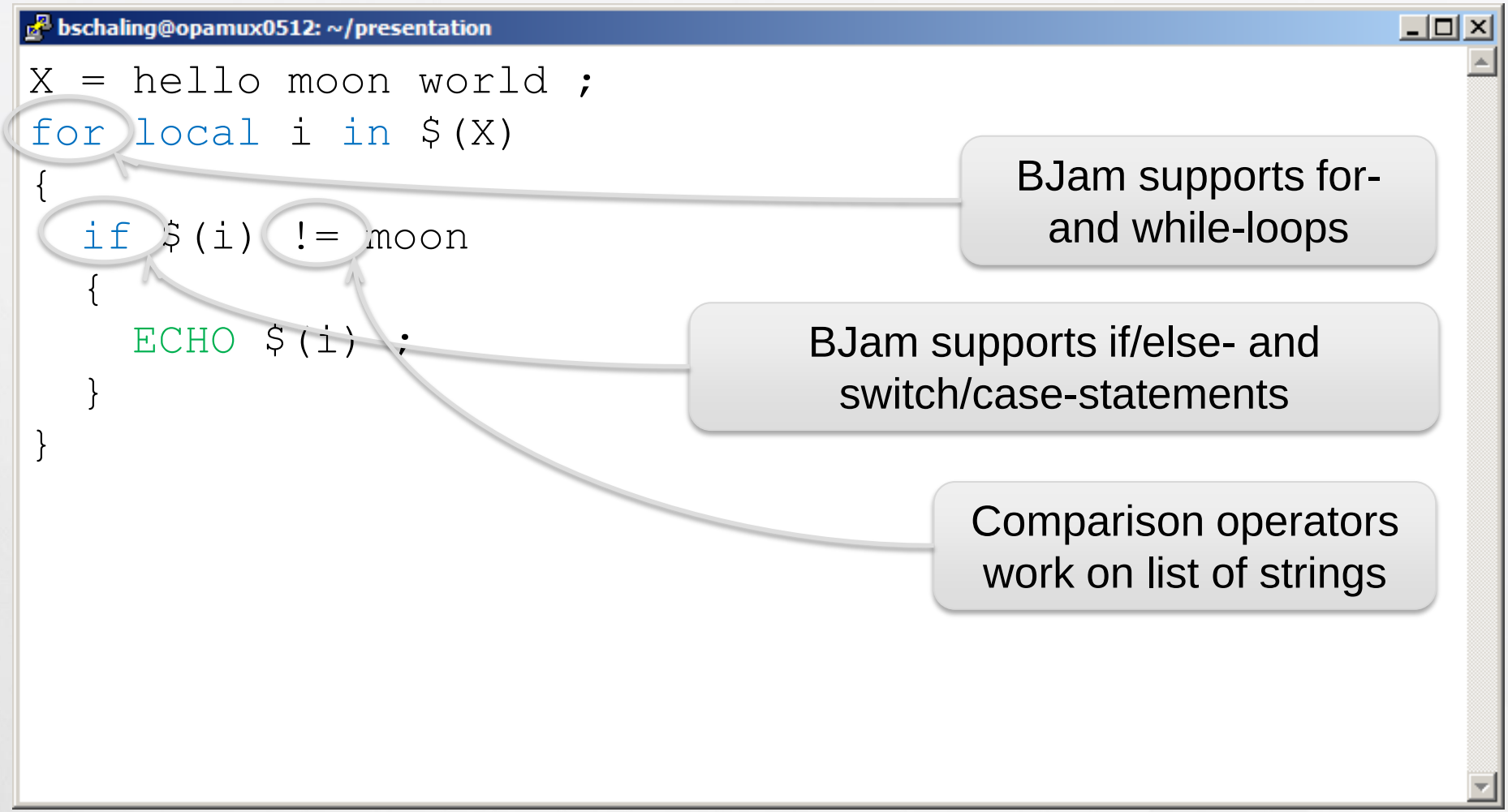
BJam: Variables



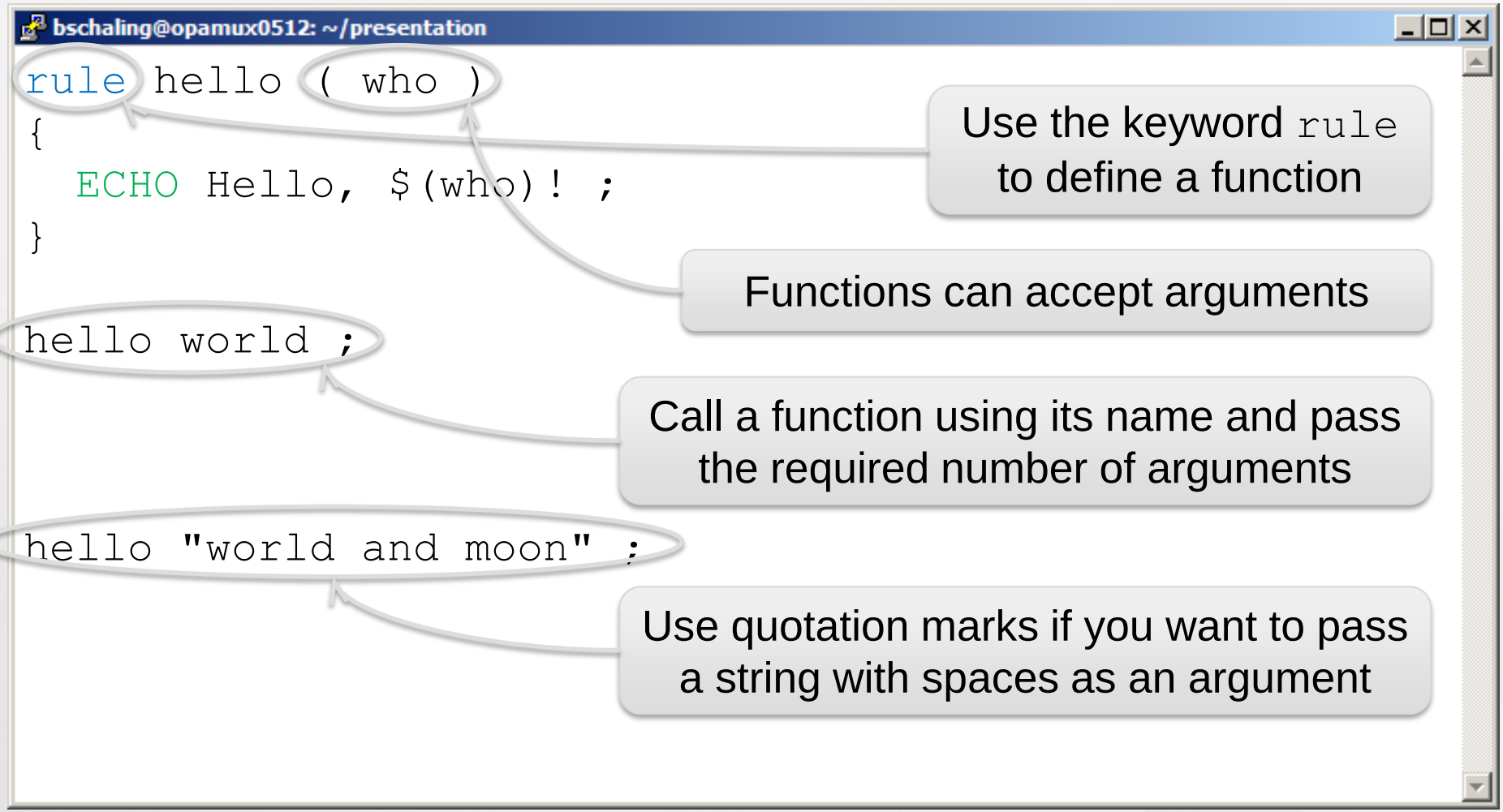
BJam: Variables



BJam: Flow-of-control statements



BJam: Functions



BJam: Functions

bschaling@opamux0512: ~/presentation

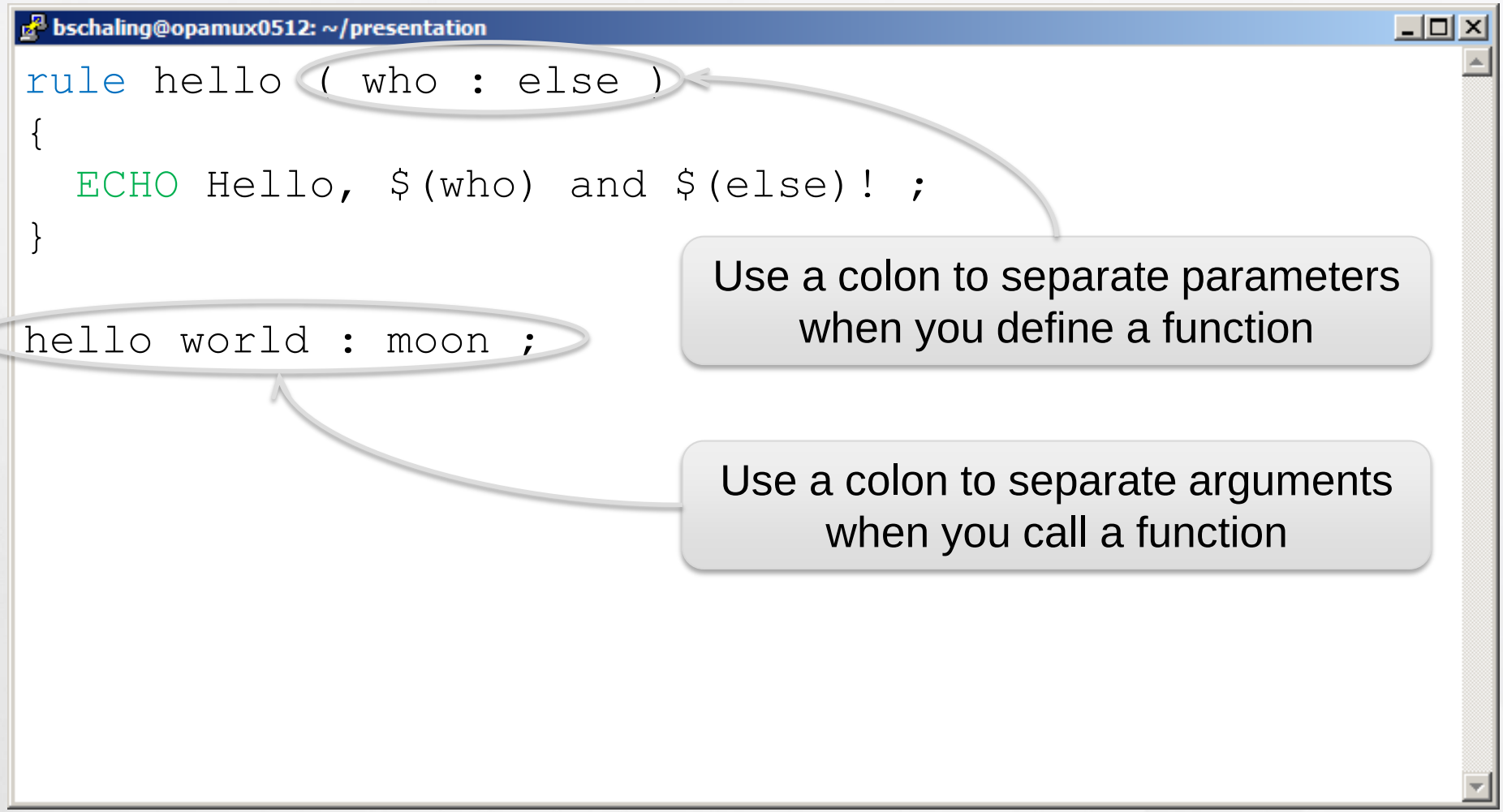
```
rule hello ( who * )
{
  ECHO Hello, $(who:J=" ")! ;
}

hello world and moon ;

hello ;
```

Parameters can be described with a syntax similar to regular expressions (* means arbitrary many, + means minium one and ? means one or zero)

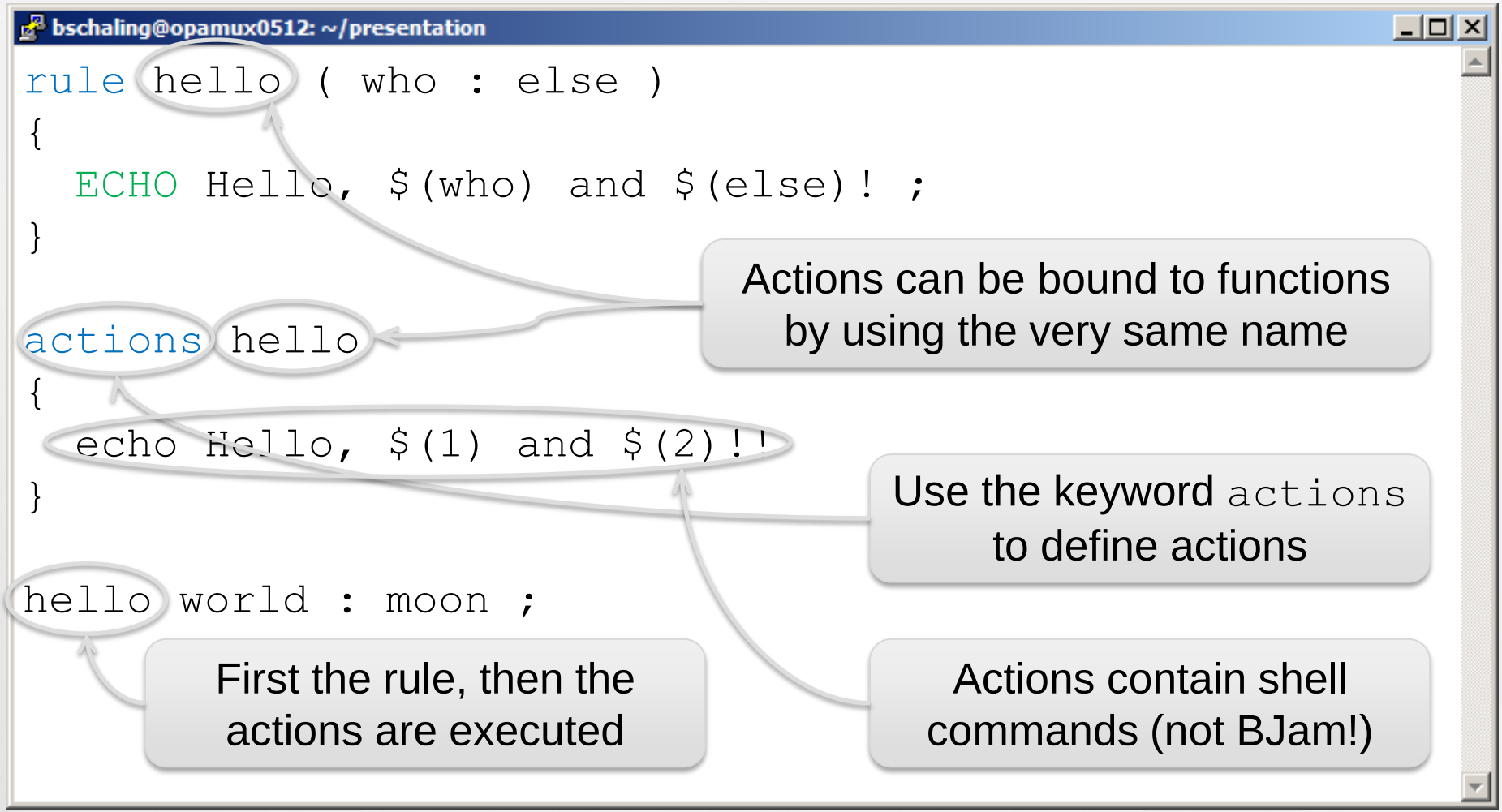
Don‘t use quotation marks if you want to pass a list of strings and the function accepts it

No argument at all needs to be passed – $(who) is then empty

BJam: Functions



BJam: Functions



BJam: Functions

bschaling@opamux0512: ~/presentation

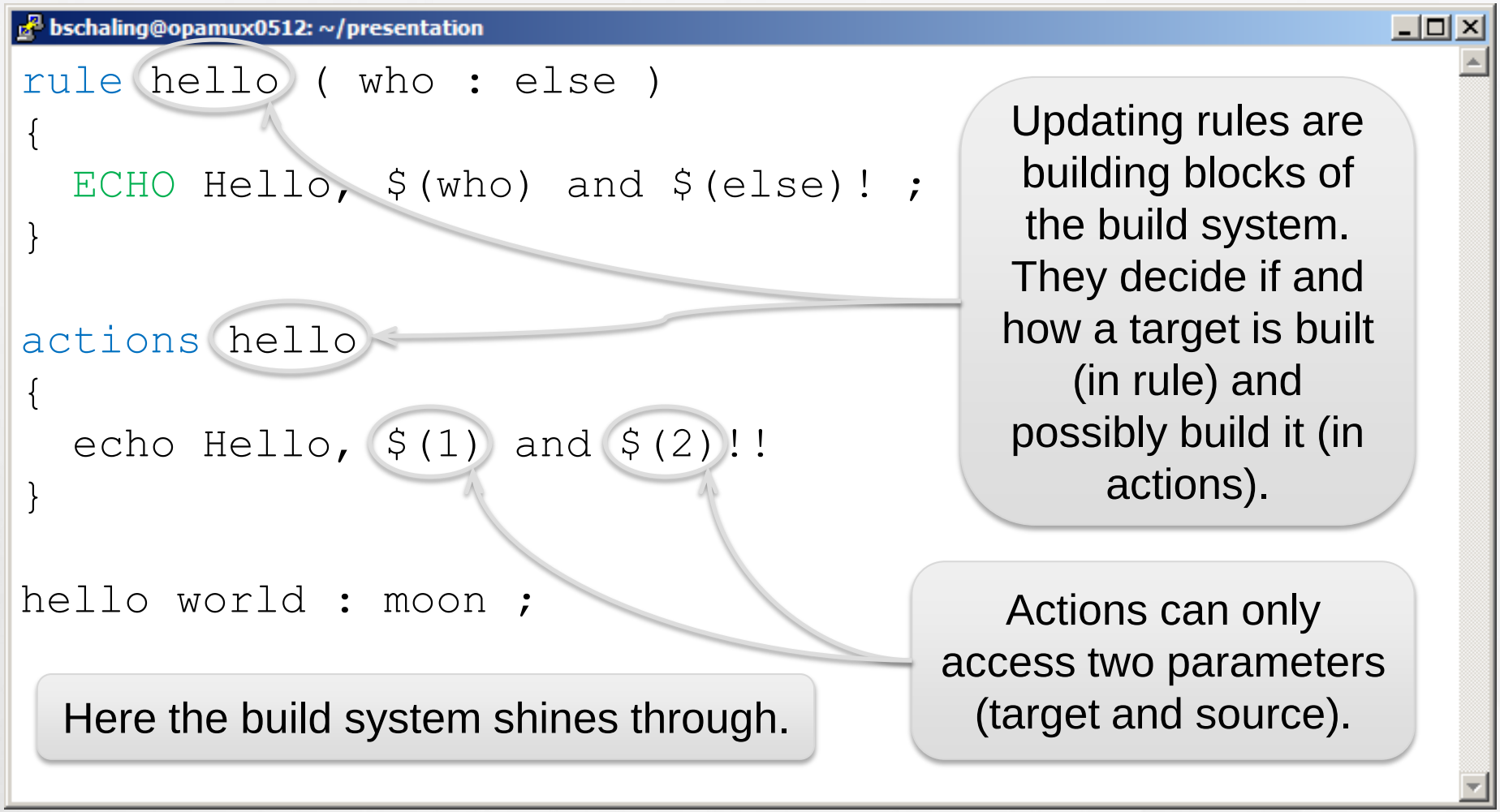
```
rule hello ( who : else )
{
  ECHO Hello, $(who) and $(else)! ;
}

actions hello
{
  echo Hello, $(1) and $(2)!!
}

hello world : moon ;
```

Functions bound to actions are known as updating rules

For updating rules the first parameter is known as target and the second parameter as source

Run with "bjam -f Jamroot.jam world".

BJam: Functions



BJam: Functions

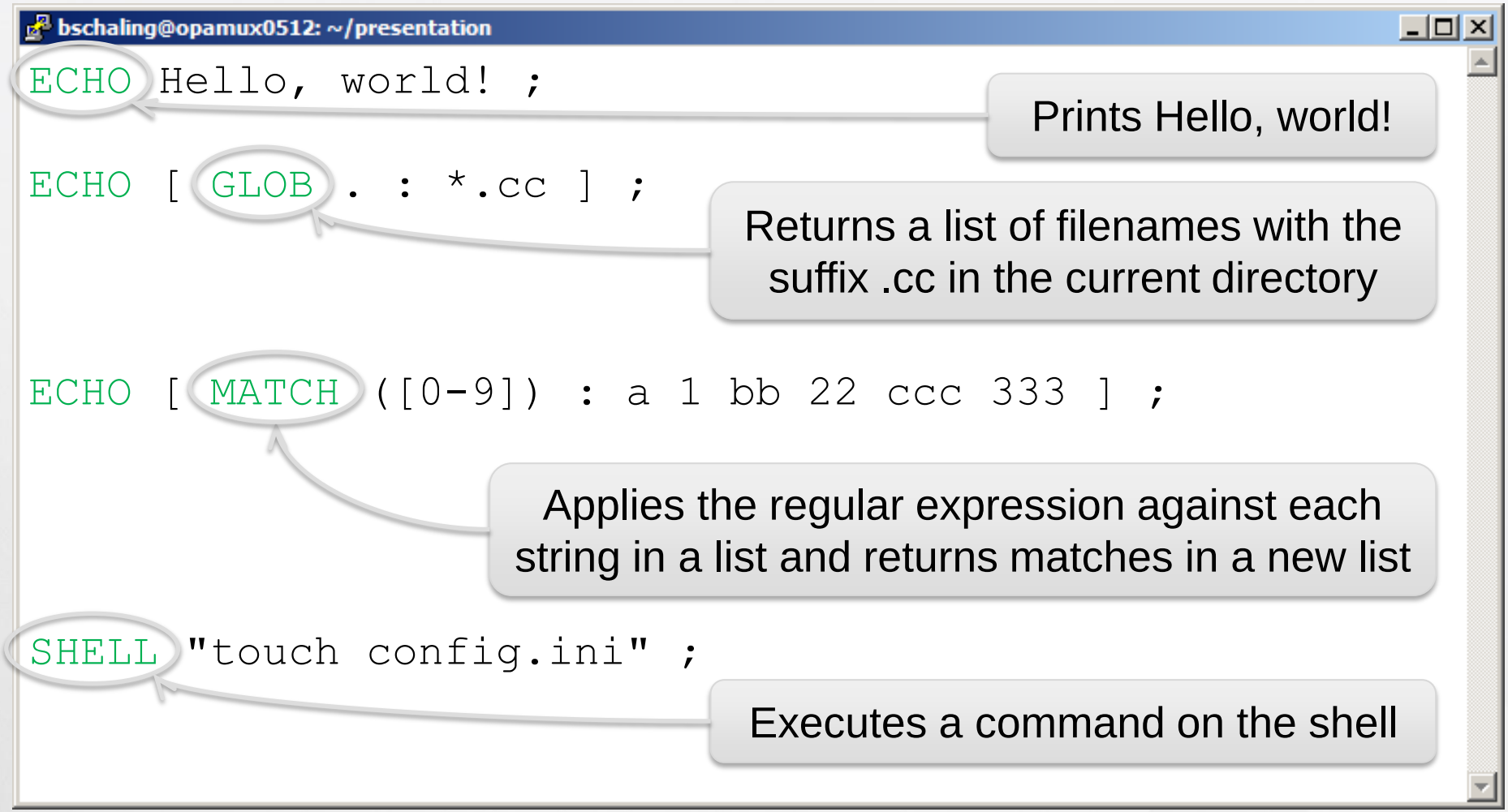
bschaling@opamux0512: ~/presentation

```
rule hello ( who + : else * )
{
  ECHO Hello, $(who:J=,) and $(else[2])! ;
}

actions hello
{
  echo Hello, $(1:J=,) and $(2[2])!!
}

hello world mars : moon sun ;
```

Indexes and modifiers can be used in actions, too.

Allowed number of targets and sources depends on signature only.

BJam: Built-in functions



BJam: Built-in functions

bschaling@opamux0512: ~/presentation

```
ECHO Hello, world! ;

ECHO [ GLOB . : *.cc ] ;
```

All built-in rules are procedual only (not bound to actions).

Use […] for nested function calls.

BJam has three categories of built-in functions:
- Dependency building (eg. DEPENDS)
- Modifying dependency trees (eg. ALWAYS)
- Utility functions (eg. ECHO, GLOB, MATCH and SHELL)

You can ignore any details and especially don‘t need to care about functions of the first two categories.

BJam: Modules

bschaling@opamux0512: ~/presentation

```
module my_module
{
  i = hello ;
  local j = world ;
  rule foo { ECHO $(i) ; }
  local rule bar { ECHO $(j) ; }
}
```

Use keyword `module` to define a module explicitly

A Jamfile automatically defines a module

Local variables and functions are unavailable for other modules

Modules provide a namespace for variables and functions

While every module has a name, a global module without a name exists, too

BJam: Accessing functions in modules

bschaling@opamux0512: ~/presentation

```
module my_module
{
  rule foo { ECHO baz ; }
  local rule bar { ECHO qux ; }
}

IMPORT_MODULE my_module ;
my_module.foo ;


ECHO [ RULENAMES my_module ] ;
```

Local functions unavailable for other modules

Use built-in function IMPORT_MODULE to import non-local functions from a module

Use built-in function RULENAMES to see a module‘s non-local functions

BJam: Accessing variables in modules

bschaling@opamux0512: ~/presentation

```
module my_module
{
  i = foo ;
  local j = bar ;
}

import modules ;
modules.poke my_module : i : goodbye ;
ECHO [ modules.peek my_module : i ] ;
```

Local variables unavailable for other modules

Use the import function to import non-local functions from another Jamfile

Use poke and peek from the module modules to set and get variables from other modules

BJam: Tips and tricks about modules

bschaling@opamux0512: ~/presentation

```
module my_module
{
   rule foo { ECHO my_module ; }
}

IMPORT_MODULE my_module ;
my_module.foo ;

ECHO $(__name__) ;

ECHO [ RULENAMES ] ;
```

Use built-in variable `__name__` to get a module's name

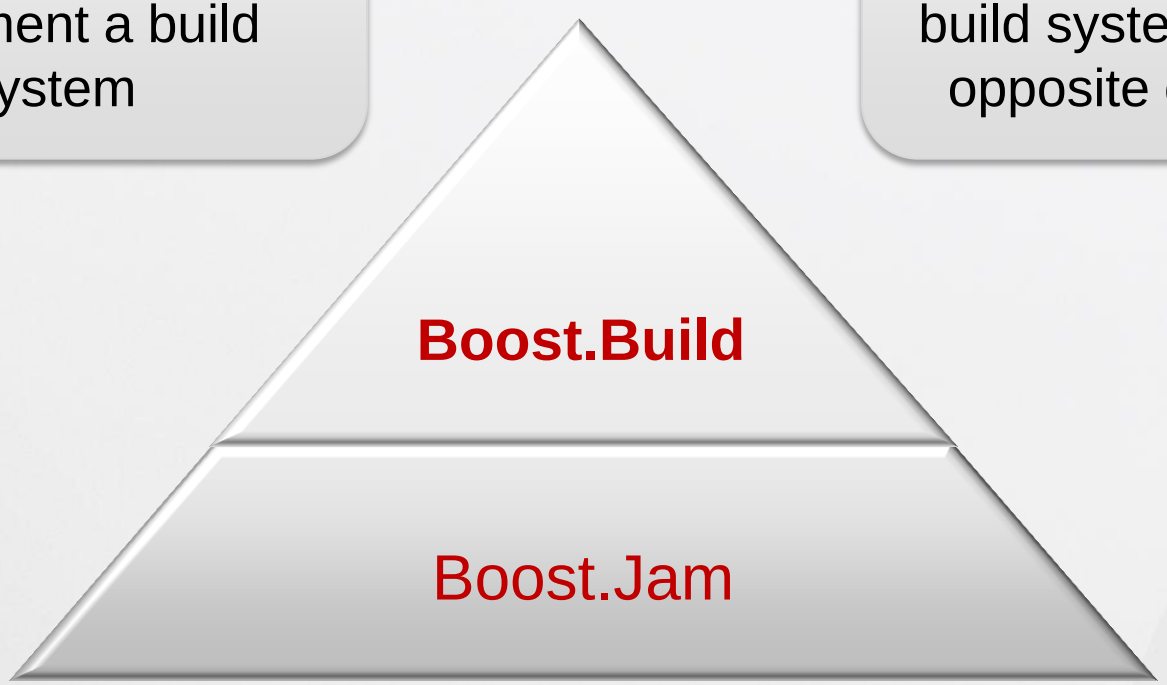
Every Jamfile is a module and has a name

Use built-in function RULENAMES to see functions defined in the global module

Boost.Build

Boost.Build consists of many Jamfiles which implement a build system

Your Jamfiles are executed within this build system (the very opposite of Python)



Boost.Build: Predefined functions

bschaling@opamux0512: ~/presentation

```
lib foo : foo.cc ;
exe bar : bar.cc foo ;
install baz : foo bar ;
```

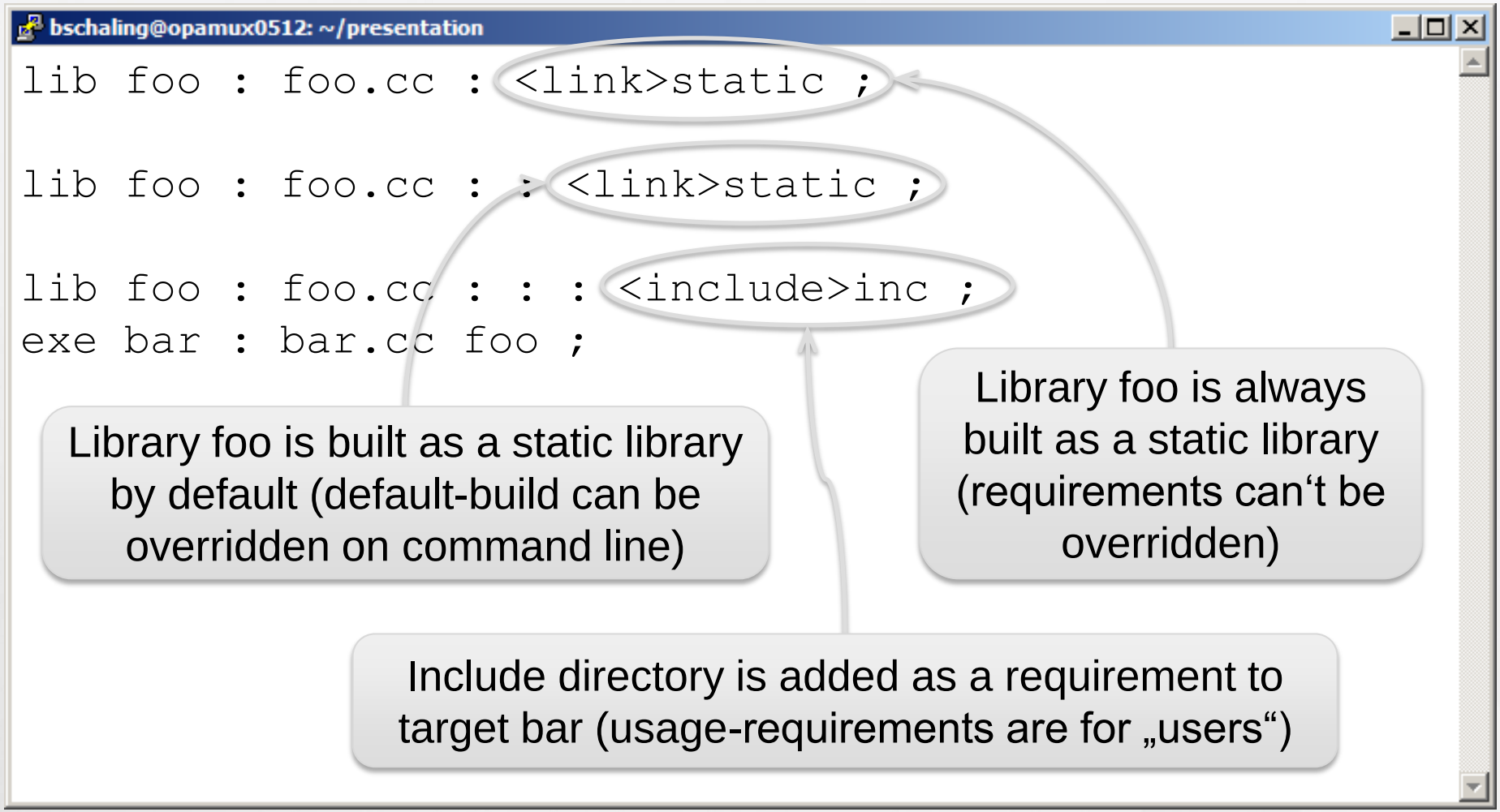
Many predefined functions to build artefacts and define targets and sources

Many predefined functions (including lib, exe and install) use a signature known as common signature:

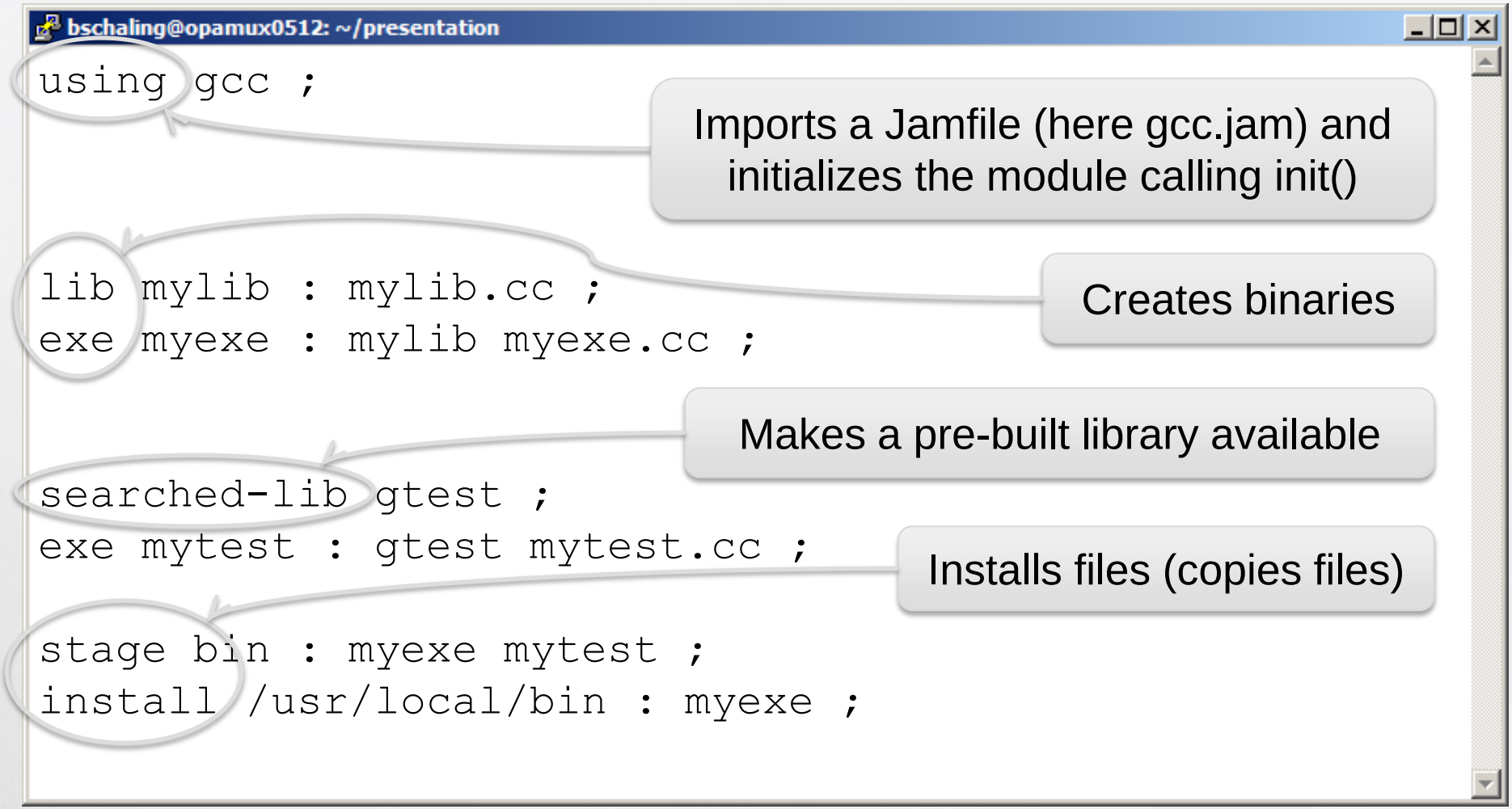
```
rule … ( target : sources + : requirements * :
default-build * : usage-requirements * )
```

Predefined functions like lib, exe and install are no updating rules and are not bound to actions – Boost.Build is on top of Boost.Jam.

Boost.Build: Common signature



Boost.Build: Functions



Boost.Build: Functions

bschaling@opamux0512: ~/presentation

Function reference in Boost.Build documentation:
http://www.boost.org/doc/libs/1_46_1/doc/html/bbv2/reference.html#bbv2.reference.rules

Function reference in Boost.Build tutorial:
http://www.highscore.de/cpp/boostbuild/#rule_reference

Functions are defined in various Jamfiles
in boost-build/build/ and boost-build/tools/

Boost.Build: Features

bschaling@opamux0512: ~/presentation

```
lib foo : foo.cc : <link>static ;
```

`<link>` feature defines whether a static or a shared library is built

Think of a feature as a cross-toolset command line option. Just like command line options they are used to customize the build process.

A feature/value pair like `<link>static` is called a property. Requirements, default-build and usage-requirement parameters are property sets (list of properties).

Boost.Build: Features

bschaling@opamux0512: ~/presentation

```
lib foo : foo.cc : <link>static ;
```

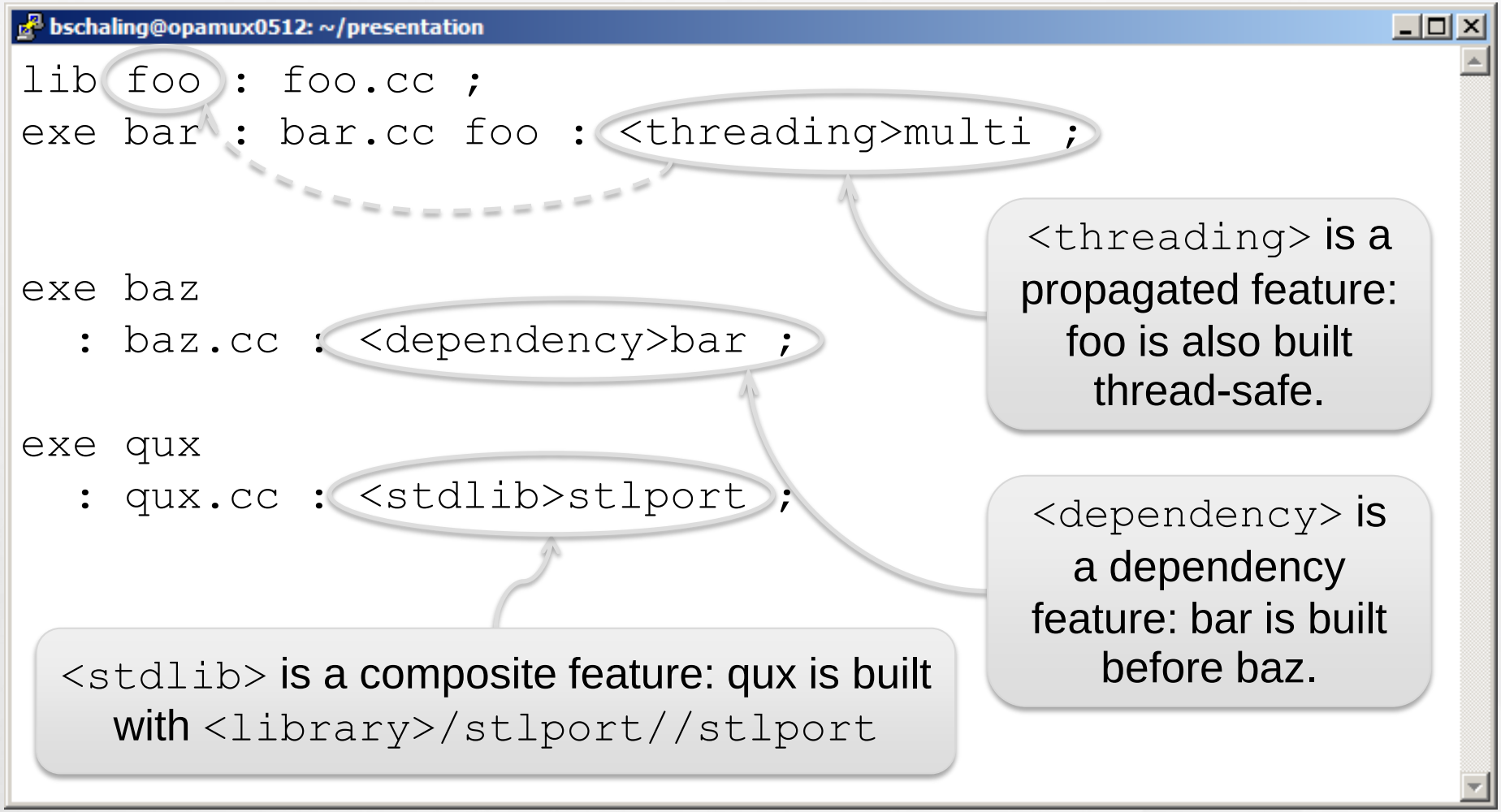
`<link>` is a „non-free“ feature which can be either `static` or `shared`

```
lib bar : bar.cc : <define>WIN32 ;
```

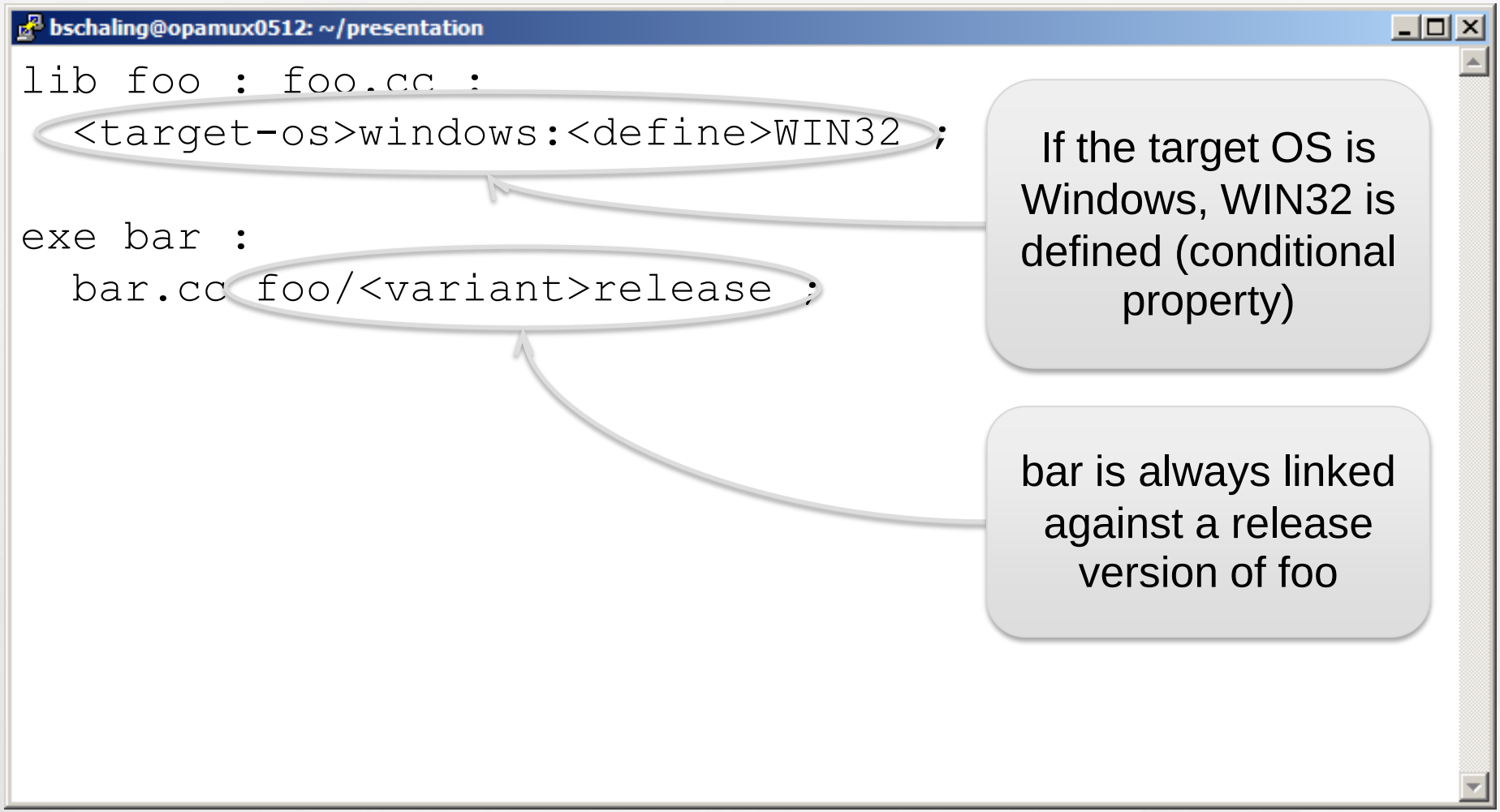
`<define>` is a „free“ feature which can be set to any value

Features have attributes which define how they are used. The most important attributes are free, propagated, dependency and composite.

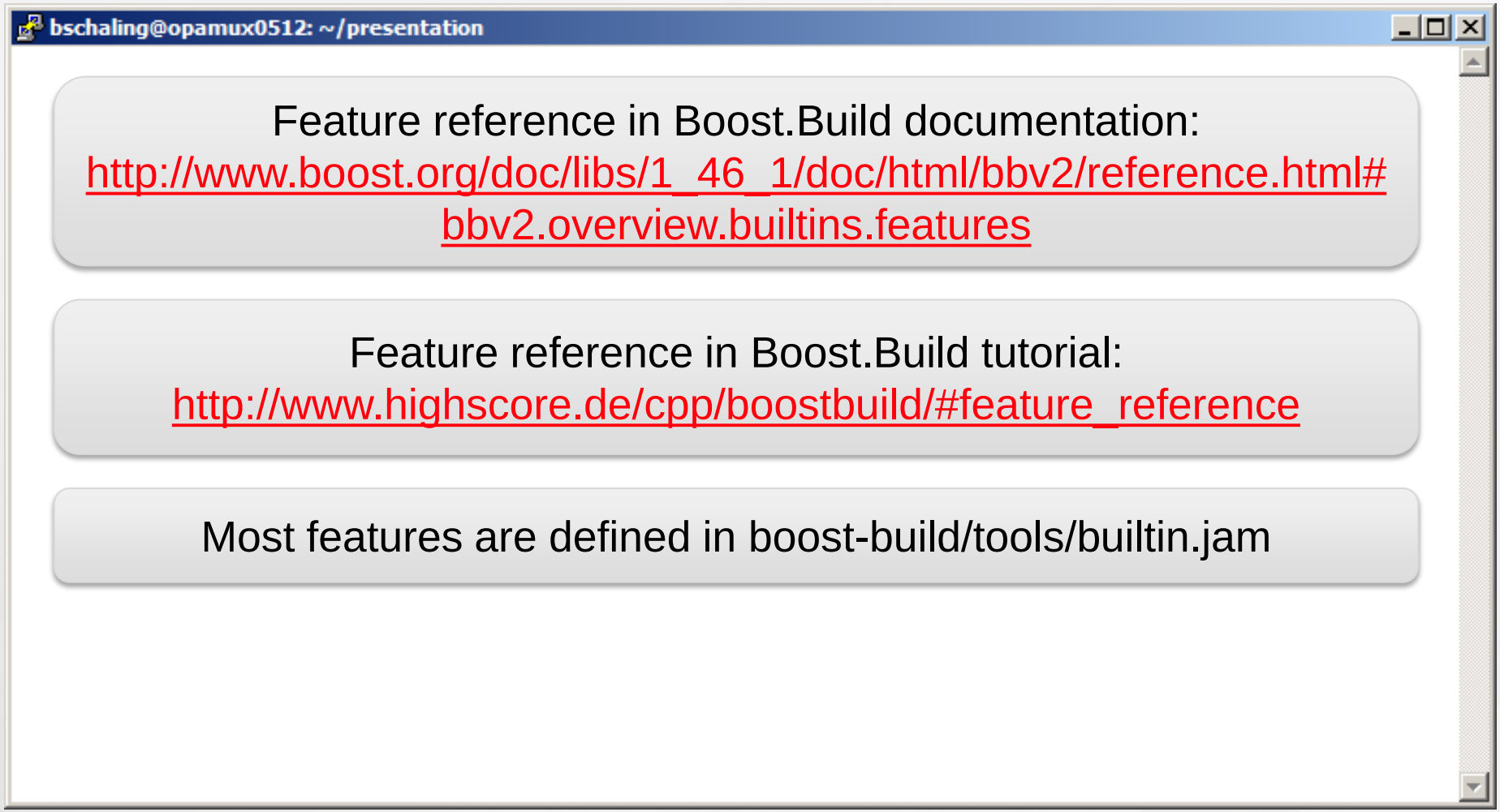
Boost.Build: Features



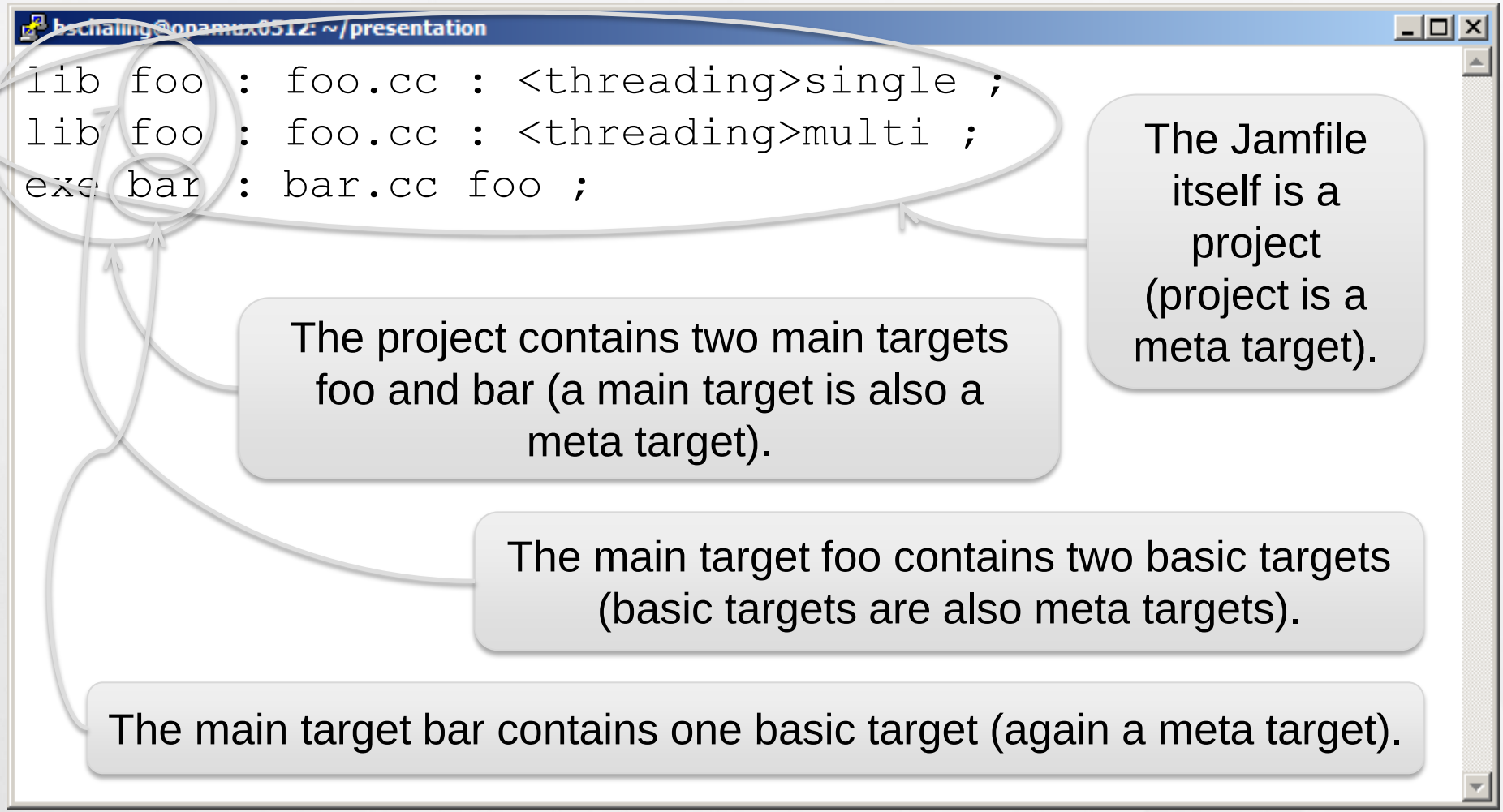
Boost.Build: Features



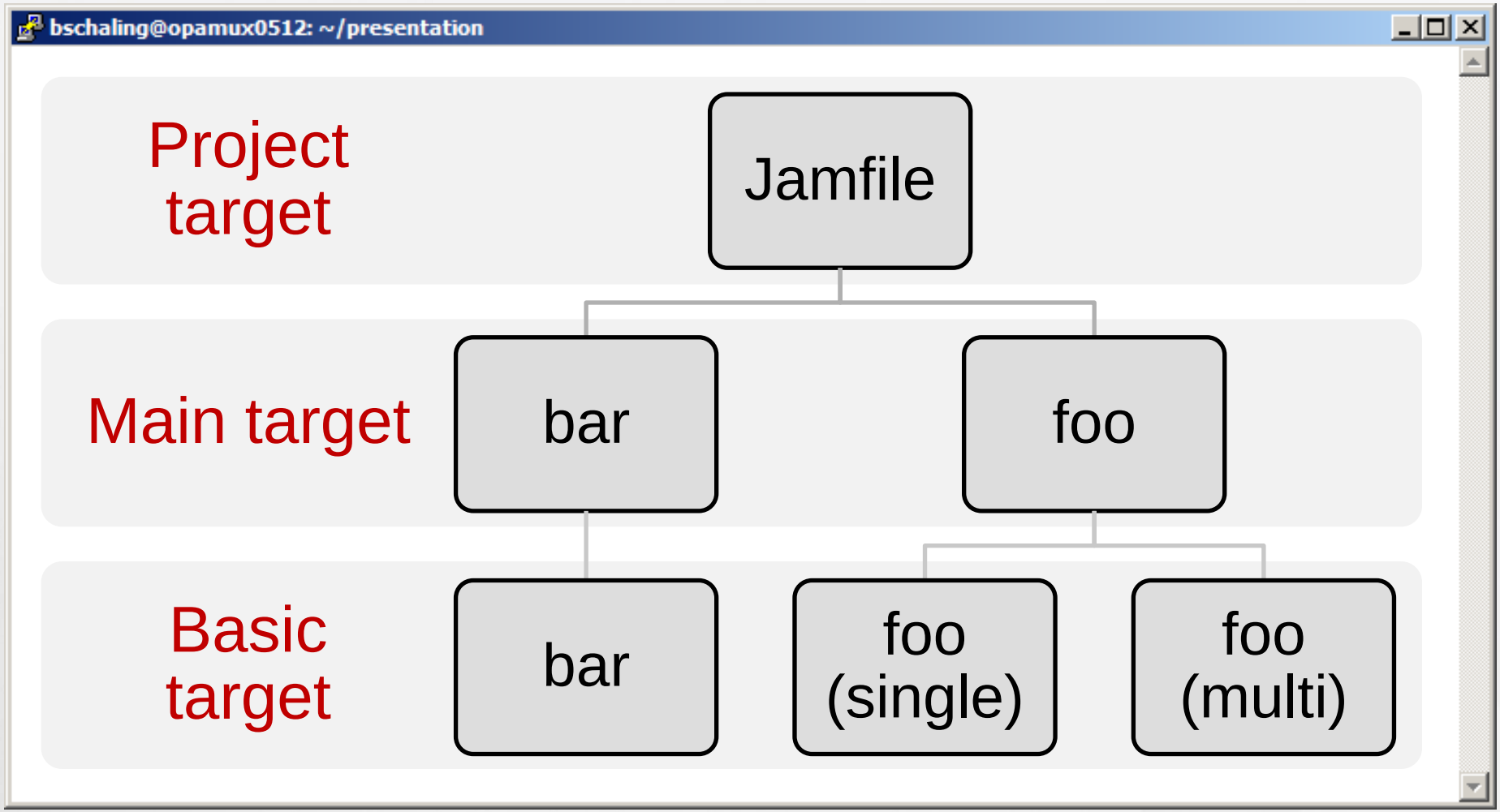
Boost.Build: Features



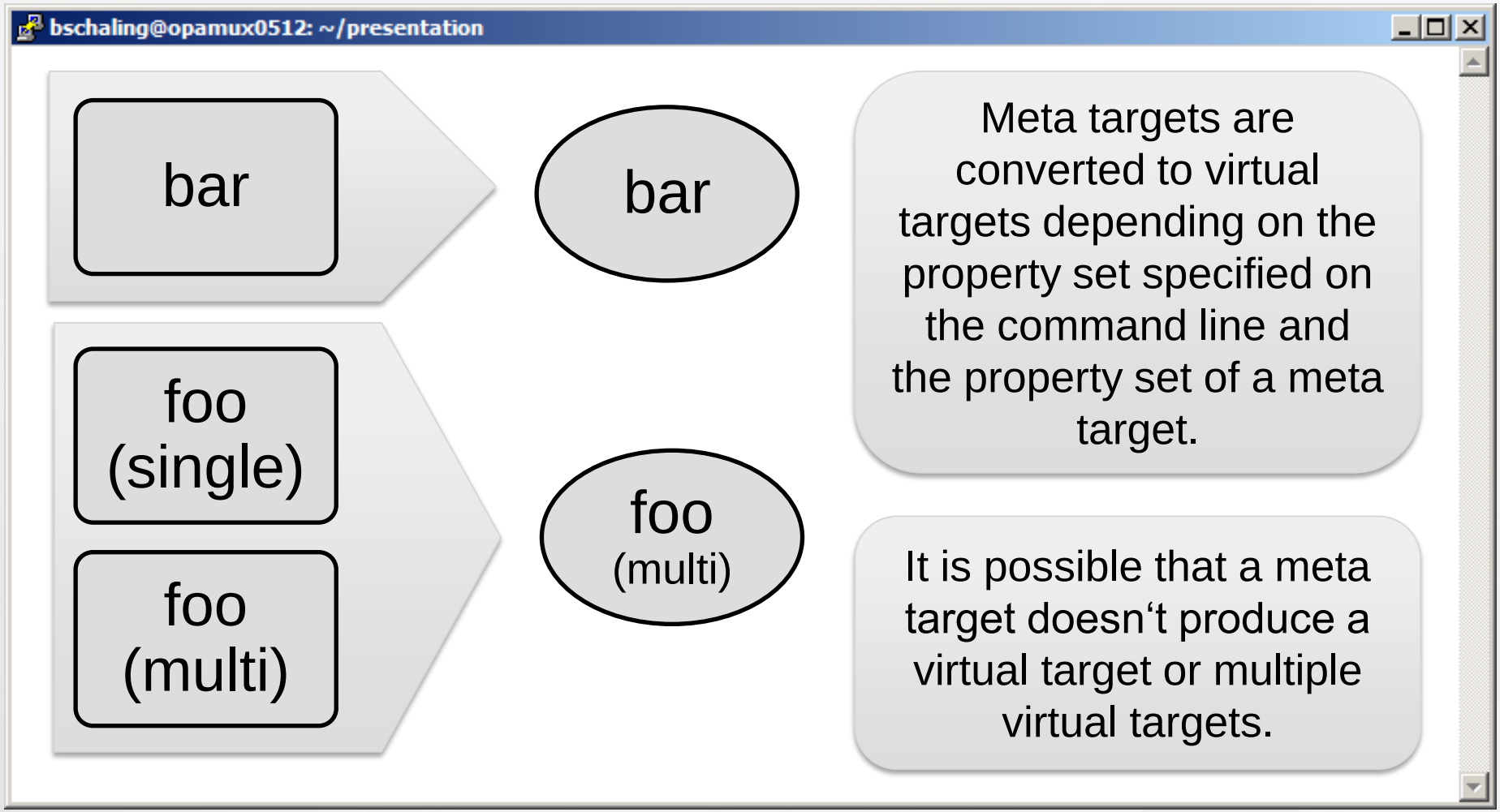
Boost.Build: Meta targets



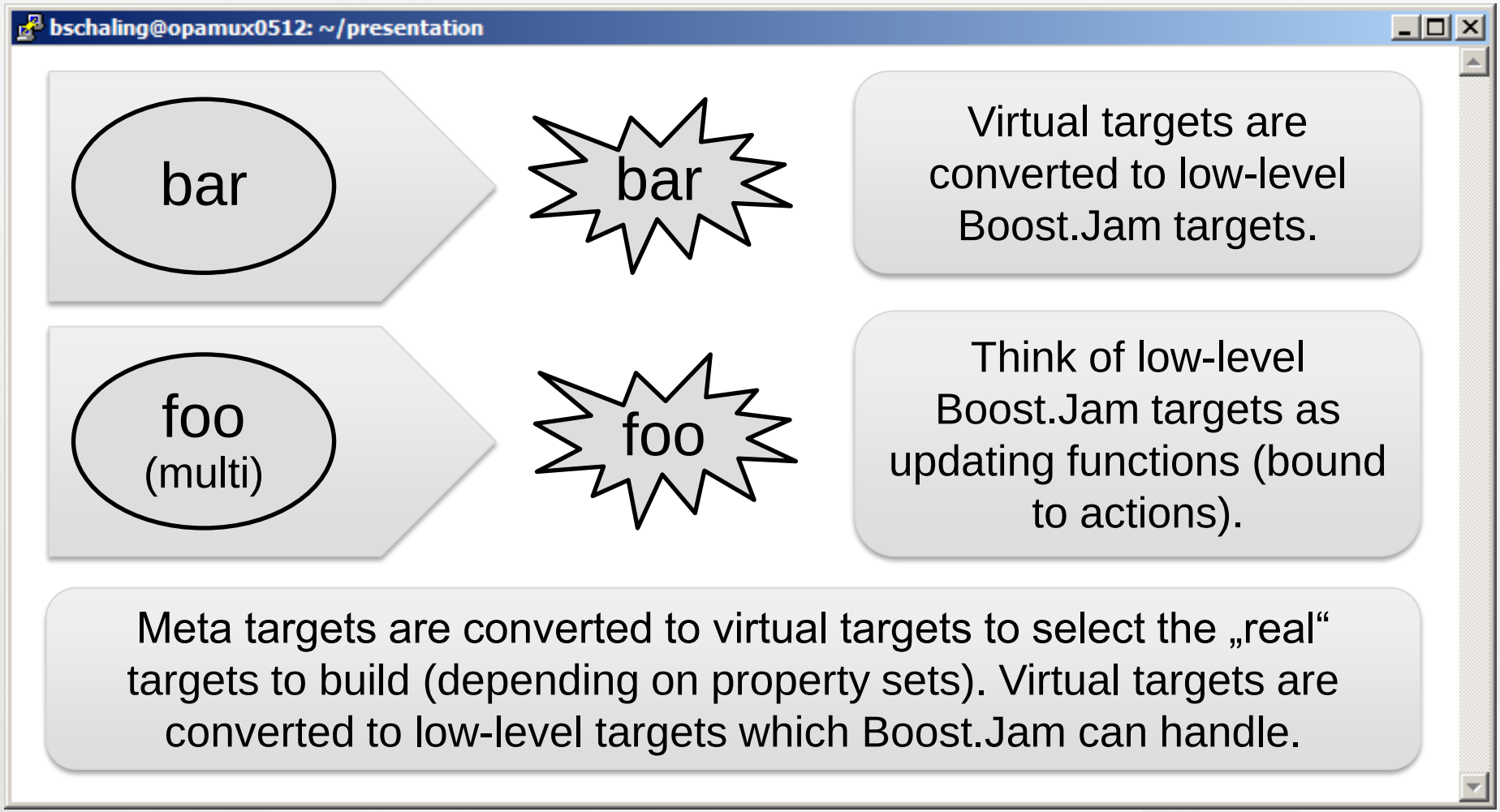
Boost.Build: Meta targets



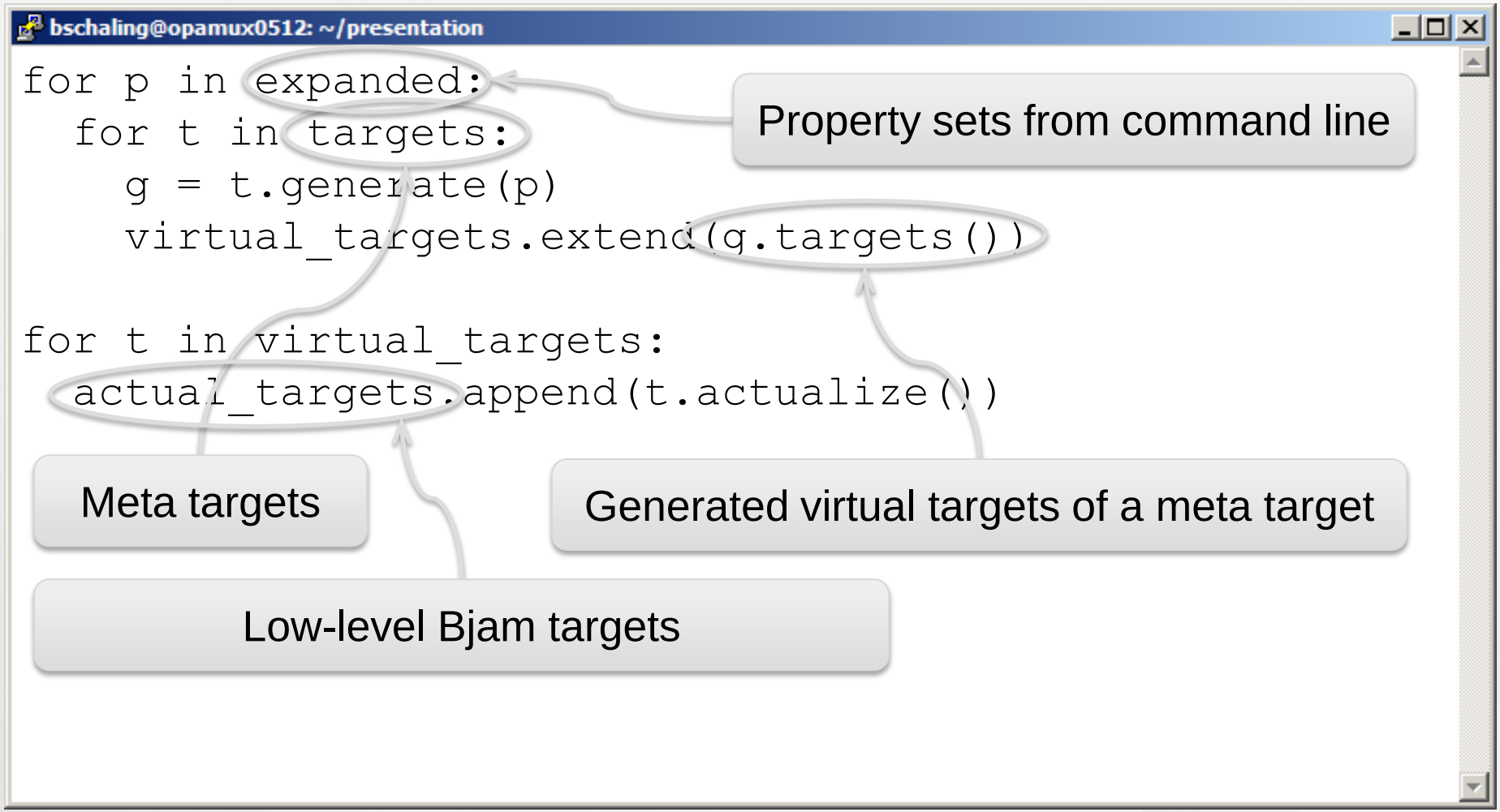
Boost.Build: Virtual targets



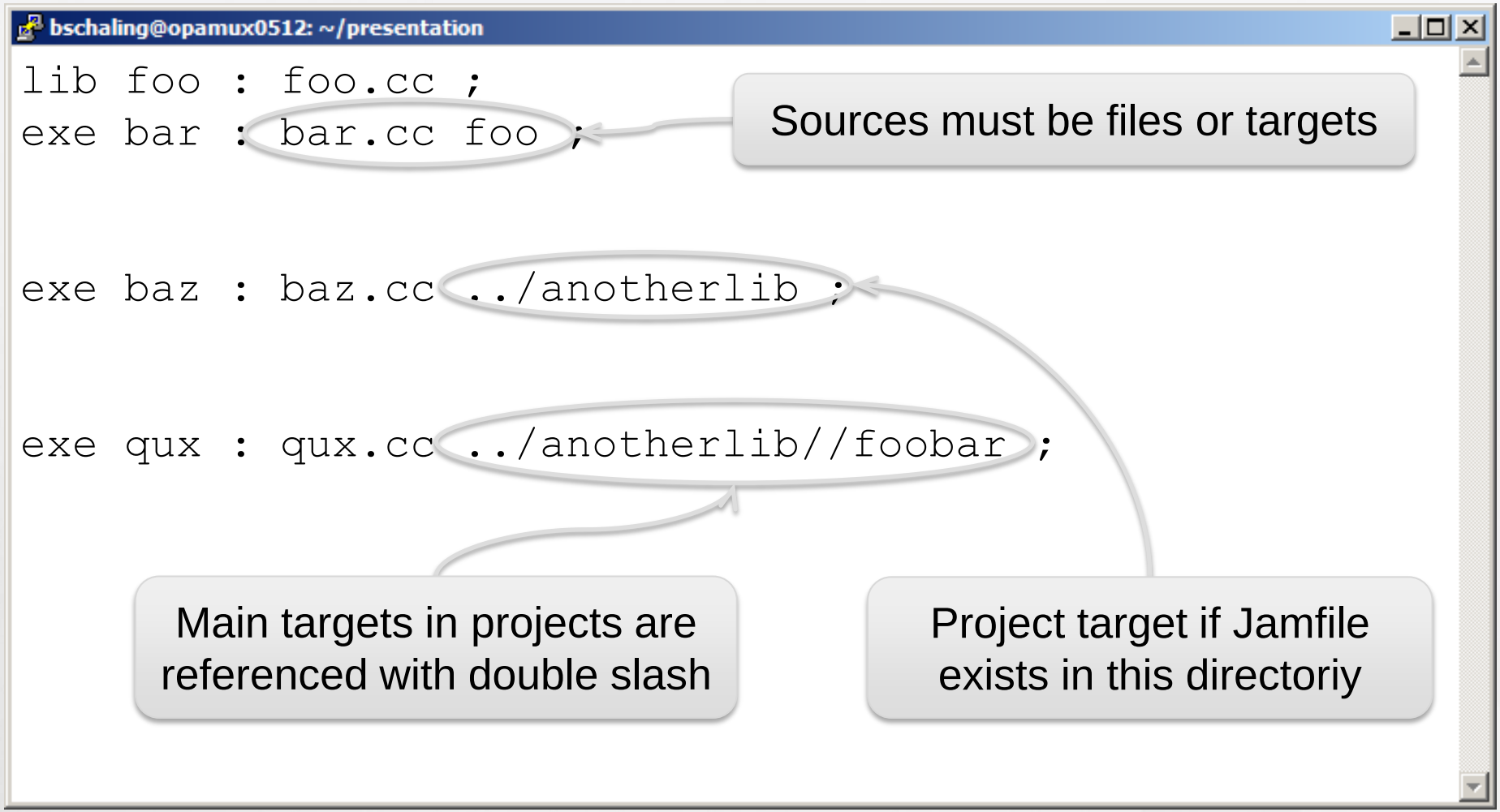
Boost.Build: Boost.Jam targets



Boost.Build: Main loop in Python port



Boost.Build: Targets as sources



Boost.Build: Features as build criteria

bschaling@opamux0512: ~/presentation

```
exe foo : foo.cc : <build>no ;
```

Main target foo never generates a virtual target

```
exe bar : bar.cc : <target-os>linux:<build>no ;
```

Main target bar generates no virtual target if project is built for Linux (conditional property)

Boost.Build: Generators

bschaling@opamux0512: ~/presentation

```
import type ;
import generators ;

type.register FE : fe ;
generators.register-standard mygen.fe2c : FE : C ;

actions mygen.fe2c
{
  command-line-fe-generator $(1) $(2)
}
```

Generators are the extension mechanism provided by Boost.Build to easily define new meta targets – without understanding all the details of meta targets.

Boost.Build: Generators

bschaling@opamux0512: ~/presentation

```
import type ;
import generators ;

type.register FE : fe ;
generators.register-standard mygen.fe2c : FE : C ;

actions mygen.fe2c
{
  command-line-fe-generator $(1) $(2)
}
```

Type FE registered for files with suffix fe

The assumption is that this generator is defined in a Jamfile called mygen.jam

Standard generator registered which converts one FE file to one C file via the actions mygen.fe2c

Boost.Build: Generators

bschaling@opamux0512: ~/presentation

```
import mygen ;

lib parser : json.fe ;

c parser.c : json.fe ;
```

The generator is automatically used to generate a C file from the FE file as Boost.Build can then use the C file to generate a LIB file.

A registered type can be used as a function (lower-case letters) to explicitly convert a FE file to a C file.

Boost.Build: Generators

bschaling@opamux0512: ~/presentation

```
import type ;
import generators ;

type.register FE : fe ;
generators.register-composing mygen.fe2c : FE : C ;

actions mygen.fe2c
{
  command-line-fe-generator $(1) $(2)
}
```

A composing generator is defined like a standard generator. It does not generate a target file per source file but a target file for all source files (composing them).

Boost.Build: Generators

bschaling@opamux0512: ~/presentation

```
import mygen ;

c parser : json.fe xml.fe ;
```

The composing generator will create one C file from both FE files. A standard generator creates two C files – one per FE file.

Boost.Build: Generators

bschaling@opamux0512: ~/presentation

```
import type ;
import generators ;
import toolset ;
import feature ;

type.register FE : fe ;
generators.register-standard mygen.fe2c : FE : C ;
feature.feature param : : free ;
toolset.flags mygen.fe2c PARAM <param> ;

actions mygen.fe2c
{
  command-line-fe-generator $(1) $(2) $(PARAM)
}
```

A free feature <param> is defined

PARAM will be set to the feature‘s value

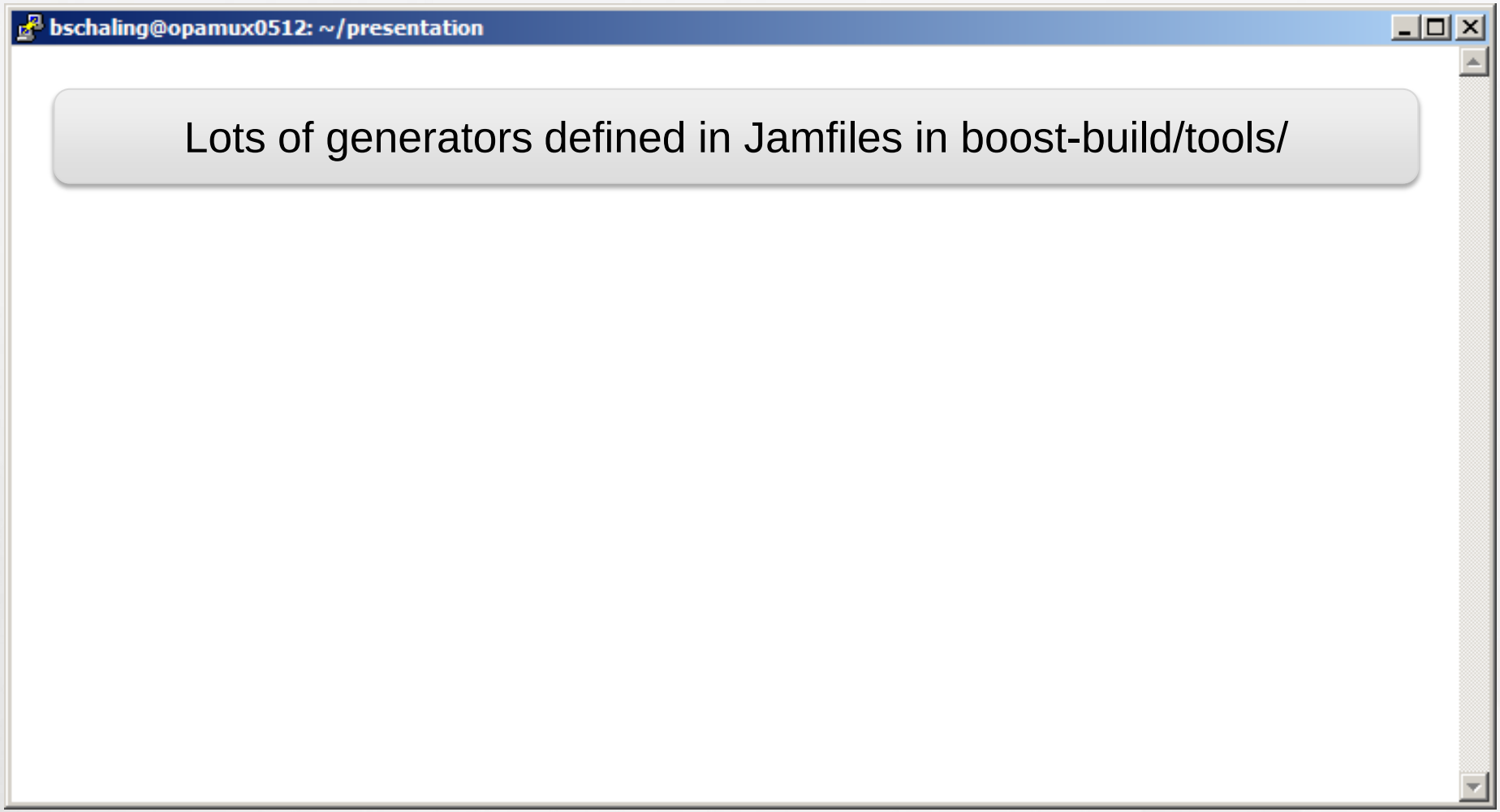
Boost.Build: Generators

bschaling@opamux0512: ~/presentation

```
import mygen ;

c parser.c : json.fe : <param>FOO ;
```

`FOO` will be passed as a parameter to the command-line-fe-generator to customize the build process

Boost.Build: Generators



Boost.Build: Useful modules

bschaling@opamux0512: ~/presentation

```
import sequence ;

ECHO [ sequence.reverse a b c d ] ;

import numbers ;

ECHO [ numbers.increment 19 ] ;
```

Module sequence provides functions like filter, transform, reverse, less, merge, join, unique, max-element to process lists of strings

Module numbers provides functions like check, increment, decrement, range, less, log10 to process numbers

Boost.Build: Useful modules

bschaling@opamux0512: ~/presentation

```
import os ;

ECHO [ os.environ PATH ] ;

import path ;

ECHO [ path.pwd ] ;
```

Module os provides functions like environ, executable-path, home-directories, expand-variable, on-windows, on-unix for OS-specific tasks

Module path provides functions like pwd, glob, glob-tree, exists, is-rooted, has-parent, parent, basename for path manipulation

Boost.Build: Useful modules

bschaling@opamux0512: ~/presentation

```
import regex ;

ECHO [ regex.split a,b,c , ] ;
```

Module regex provides functions like split, split-list, match, transform, escape, replace, replace-list to support regular expressions

Module set provides functions like difference, intersection and equal to process sets

```
import set ;

ECHO [ set.difference a b : b c ] ;
```

Boost.Build: Useful modules

bschaling@opamux0512: ~/presentation

```
import string ;

ECHO [ string.join a b c : , ] ;
```

Module string provides functions like whitespace-chars, whitespace, chars, join, words, is_whitespace to process strings

```
using boost : 1.46 ;

import boost ;
boost.use-project ;
```

Module boost makes Boost libraries automatically available