Spack

Package manager tool

Javier Cervantes Villanueva
Technical Student
Universidad de Murcia (ES)

CERN

Outline

- Introduction
- How does Spack work?
  - Core elements
  - Installing packages
  - Adding new packages
- Summary and future plans

Introduction

Problem to solve: build package and combinations

Packages

Externals (~300)	MC Generators (~80)	Group's projects
Python, GSL, Boost…	Pythia Herwig HepMCAnalysis …	ROOT GEANT4
About 10 Grid packages		
FTS, WMS, DPM…		

Combinations

Packages, **versions** and **combinations** agreed with experiments

Compilers	Architectures	Operating systems	Build types
• gcc4.9.3, gcc6.2.0 • clang • native	• Intel • arm64	• SLC6 • CentOS7 • Ubuntu16 • Mac 10.11	• Release • Debug

What is Spack?

- **Package manager tool** from SuperComputing
- Written in Python
- Automates all package-related processes:
  - source download
  - checksum verification
  - configuration
  - build
  - installation
  - (very basic) testing

http://github.com/LLNL/spack

What is Spack?

1000+ packages
20+ organizations
110+ contributors

NERSC using Spack on Cori: Cray support.
EPFL (Switzerland) contributing core features.
Fermi, BNL: high energy physics ecosystem.
ANL using Spack on production Linux clusters.
NASA packaging an Ice model code.
ORNL working with us on Spack for CORAL.
Kitware: core features, ParaView, UV-CDAT support

CERN: Building and installing software (ala LCGCMake)
Contributing with packages and core features as well.

Image from: http://llnl.github.io/spack/files/Spack-SC16-Tutorial.pdf

What is Spack?

- Open source (https://github.com/LLNL/spack/)
- Quick adoption by HPC community
- **Many user contributed packages (+700 last year)**

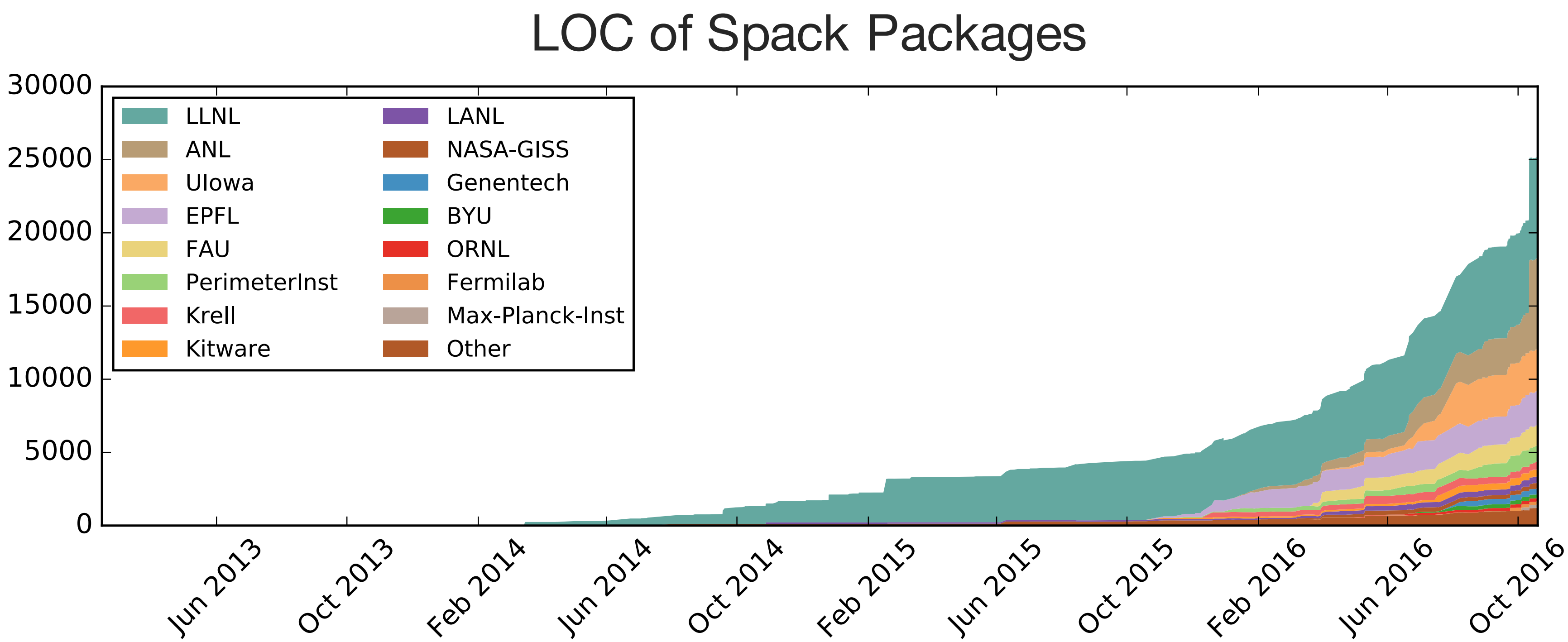


Image from: http://llnl.github.io/spack/files/Spack-SC16-Tutorial.pdf

What is Spack?

- Community also contributing to the **core development**

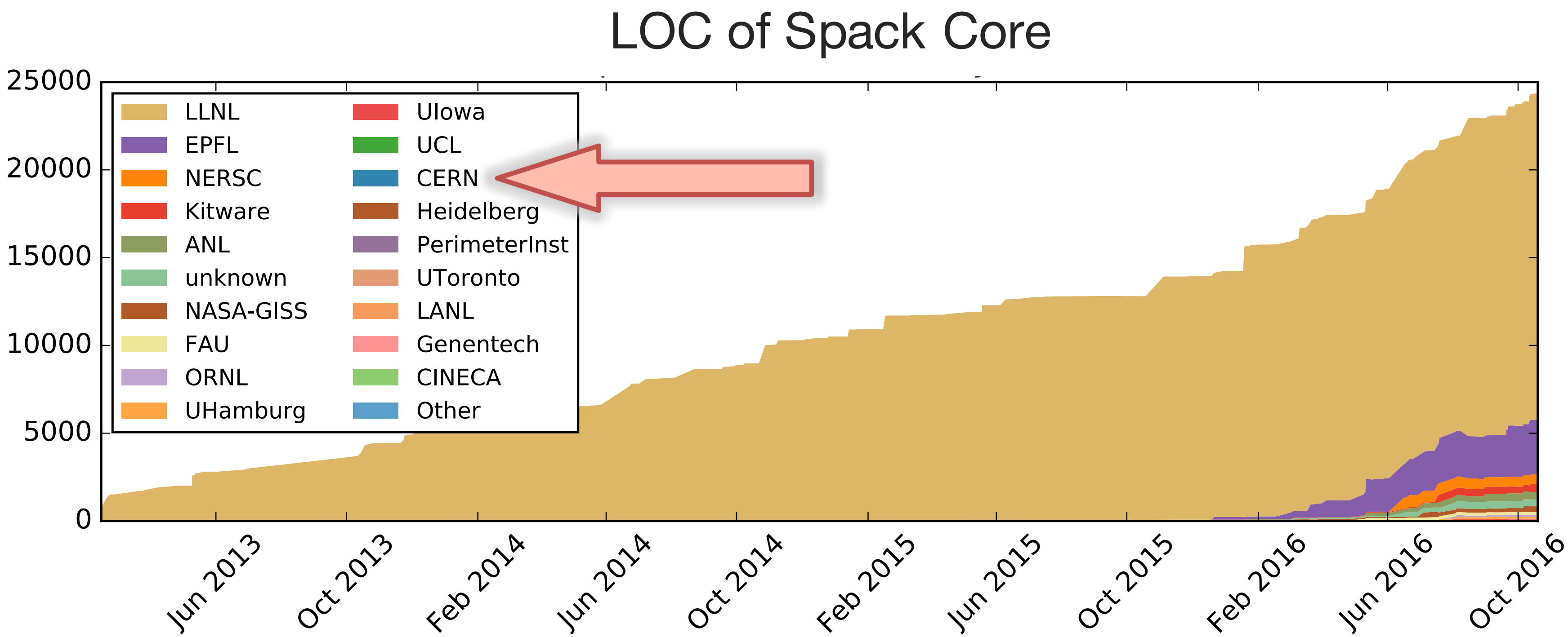


Image from: http://llnl.github.io/spack/files/Spack-SC16-Tutorial.pdf

Influences

- Spack is **not the first tool to automate builds**

- **Influences** from others package managers:
    - Different **package and dependency configurations**
    - **Dynamic linking** with RPATH and environment modules

- **Limitations** of single previous tools:
    - Focus on a **single package** builds with its dependencies up to date.
    - **Not combinatorial versioning** (or not human-readable)
    - Conflicts between different configurations **concern to the user**
    - Multiple versions installed in the **same prefix**

Main features

- Custom packages:
  - By version, platform, compiler, options and dependencies
- Novel **concretization** process
- Modular
- Environment isolation
- Compiler wrappers
- Multiple package installations can coexist
  - `RPATH` to link dependencies
  - Each one has its **own prefix** (in human-readable format)

```
$ spack find ROOT
==> 2 installed packages.
-- linux-x86_64 / gcc@6.2.0 --------     -- linux-x86_64 / gcc@4.9.3 ------------
ROOT@6.08                                ROOT@6.08
```

What Spack isn't?

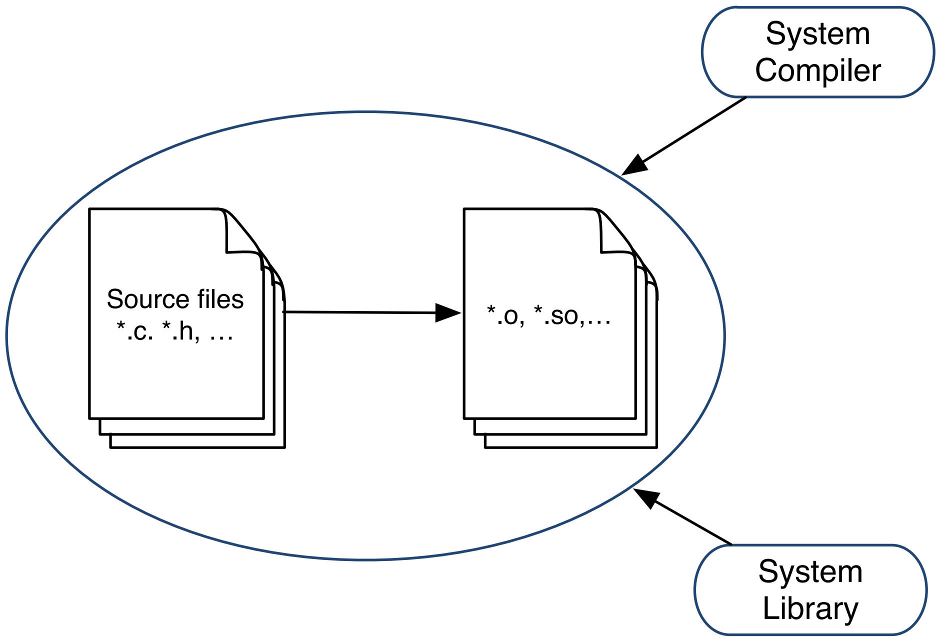
- Spack is NOT a replacement for :
  - CMake
  - GNU Autotools
  - Make
- Spack uses them in a more abstract way
  - Providing core functions that wrap these tools
  - Preparing the environment for us

Top of other tools

Package Manager	• **SPACK** • Manages dependencies • Drive package-level build systems
High Level Build System	• **CMake, Autotools** • Handle library abstractions • Generate Makefiles, etc.
Low Level Build System	• **Make, Ninja** • Handles dependencies among *commands* in a single build

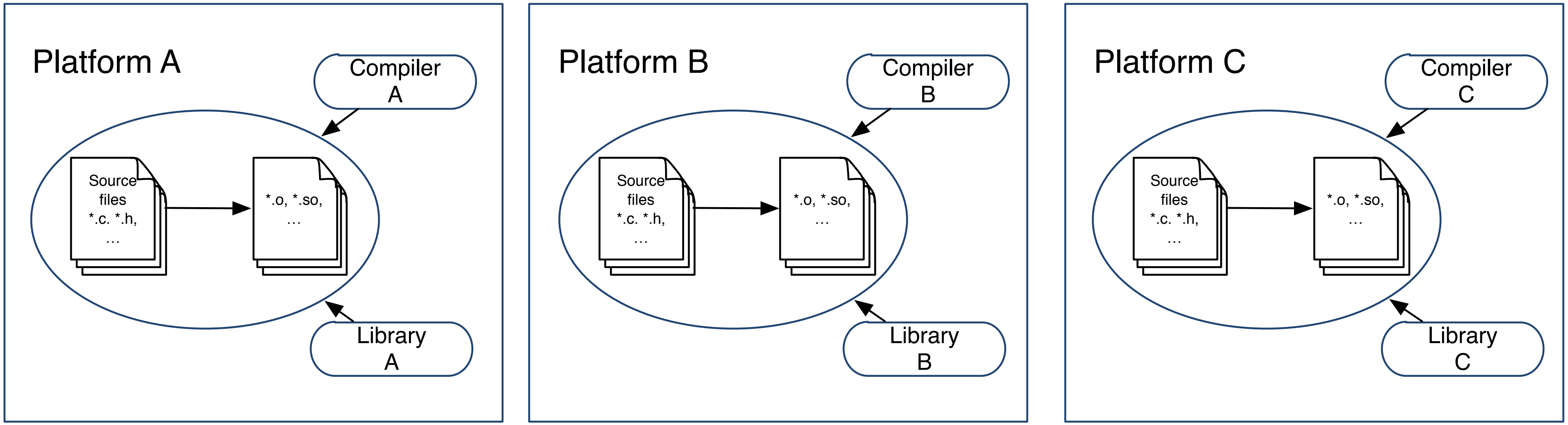
Low Level Build System

- **Creation of executable libraries**
- Allows to build a package **without knowing** how it is done
- Resolves the source file dependencies of the **own package**
- **Knows nothing about package dependencies or platforms**



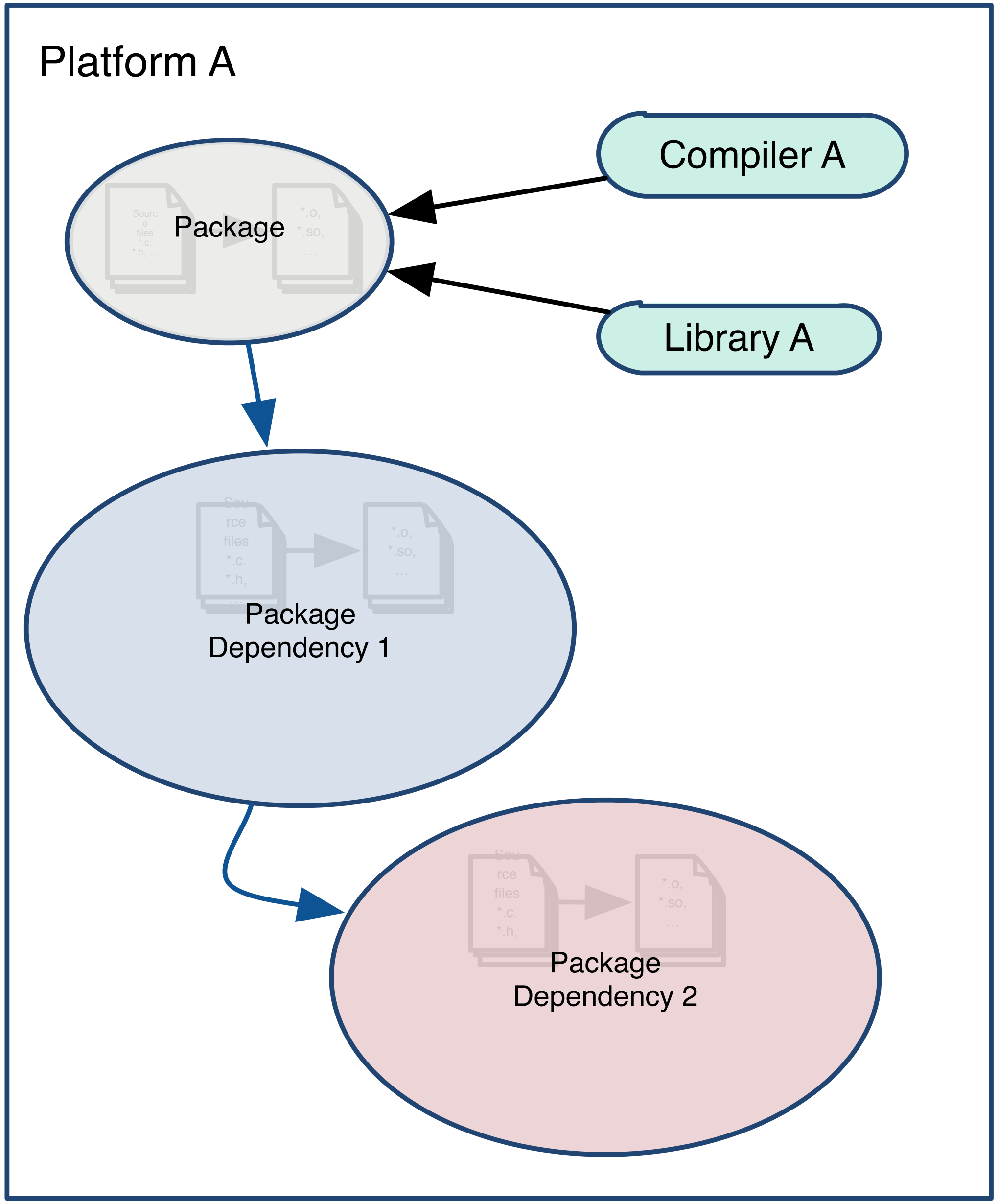
High Level Build System

- Cross platform **discovery of system libraries**
- Automatic discovery and configuration of the toolchain
- Setup build environment



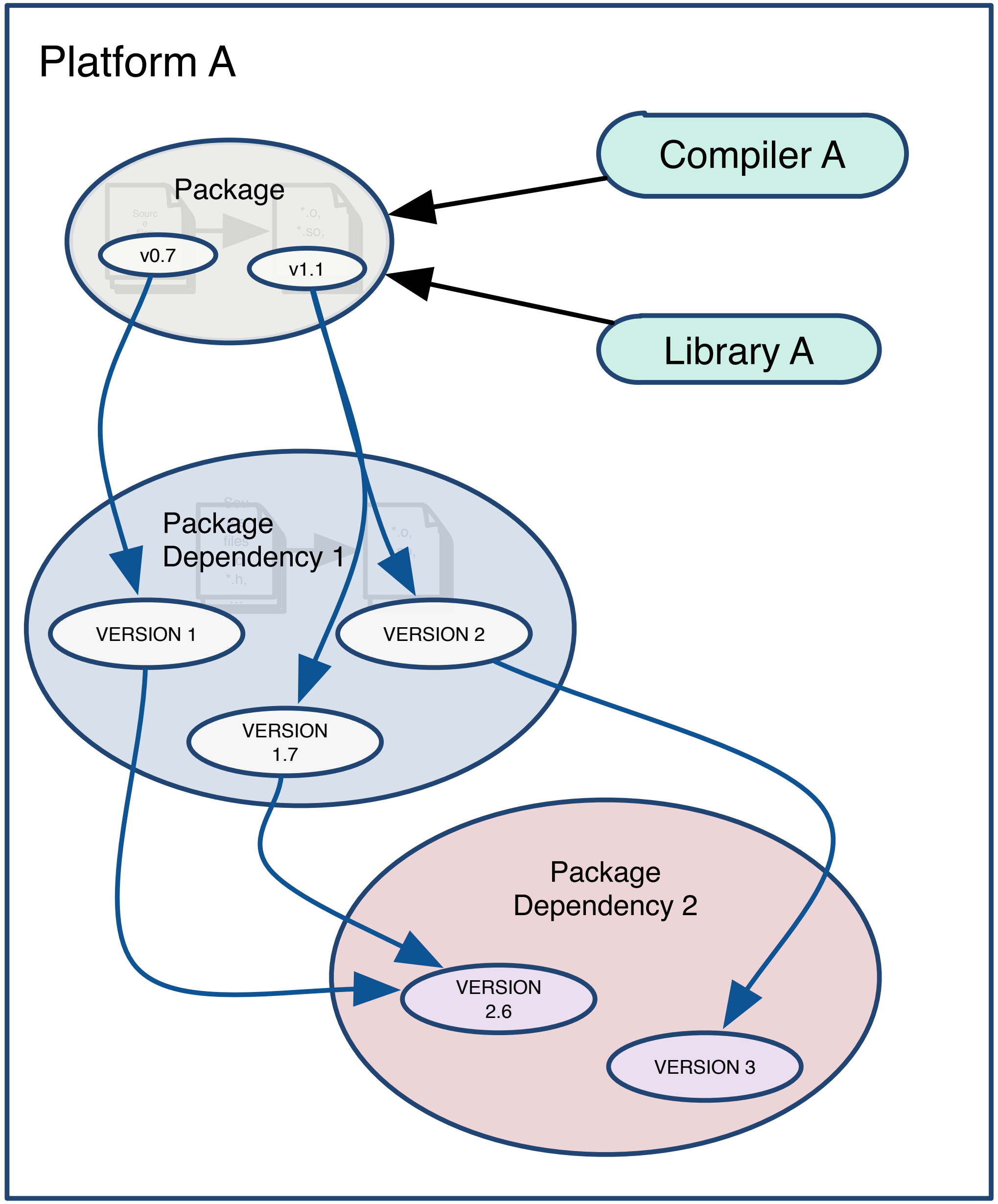
Spack

- Build stacks of packages



Spack

- Build stacks of packages
- Manage dependencies and versions between packages
- Combinatorial versioning in diverse environments



How does Spack work?

Spack 101

- How to install Spack:

```
$ git clone https://github.com/LLNL/spack.git
$ . spack/share/spack/setup-env.sh
```

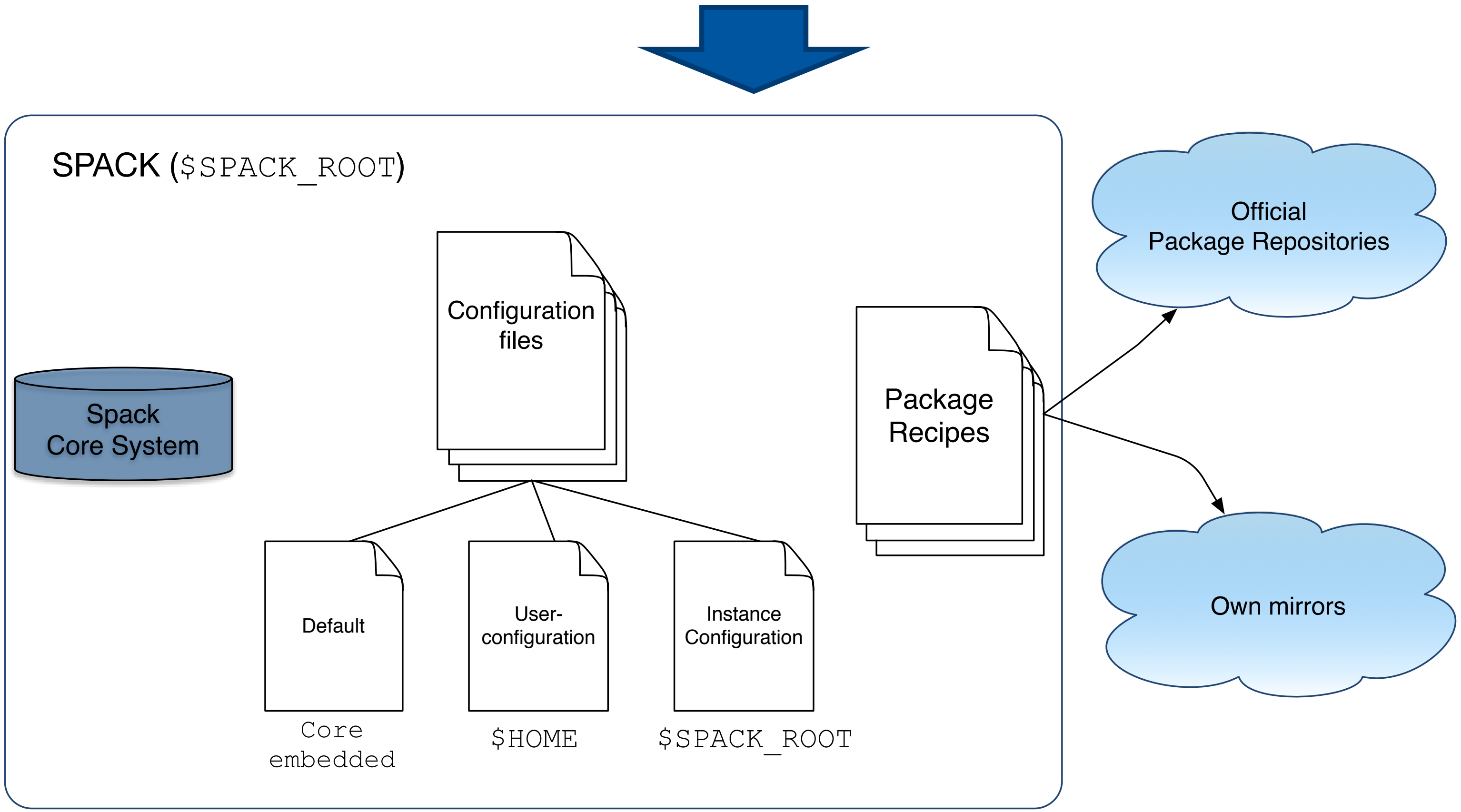
- How to install a package:

```
$ spack install ROOT
```

- ROOT and all its dependencies are installed within the default Spack directory (`$SPACK_DIR/opt`)

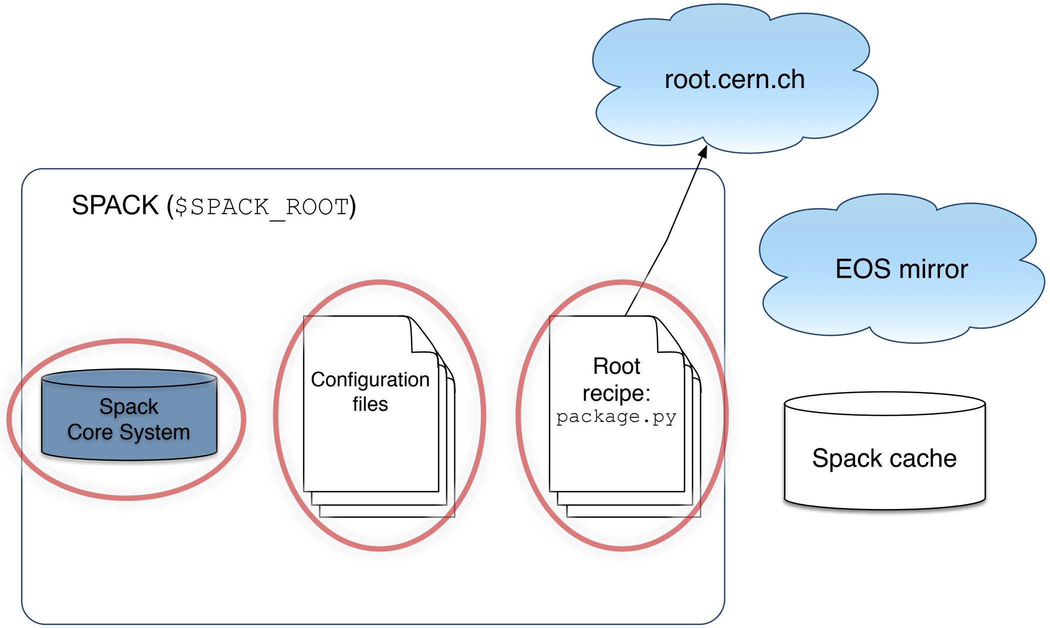
Core elements

```
$ git clone https://github.com/LLNL/spack.git
```



Core elements

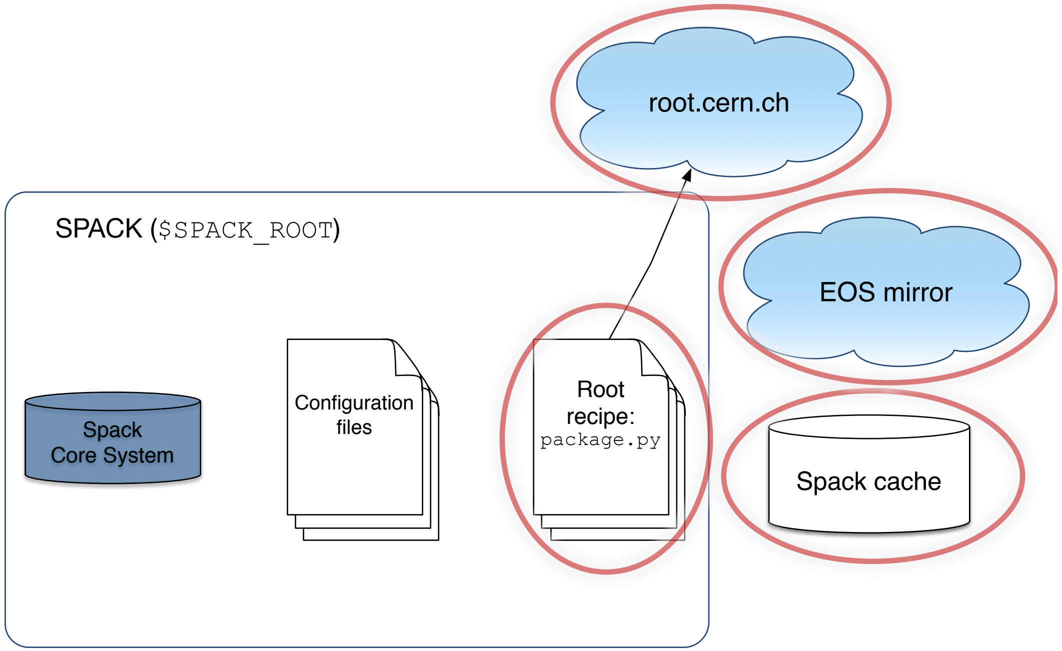
```
$ spack install ROOT
```



1. Start the Spack process
2. Take recipe to build and install ROOT
3. If anything else specified, Spack takes:
   - From its recipe
     - URL to download ROOT
     - Preferred ROOT version or latest one
     - Dependencies of ROOT to build and install
   - From configuration files:
     - Version and other parameters to build each dependency of ROOT
     - Environment setup (compilers, libraries, prefix…)

Core elements

```
$ spack install ROOT
```



4. Checks whether source of ROOT exists in the system
5. Otherwise, downloads it from:
   - An own mirror
   - The official URL (specified in the recipe)
6. Build and install ROOT with all its dependencies
   - By default in `$SPACK_ROOT/opt`
   - If specified, in `$install_prefix`

Core elements

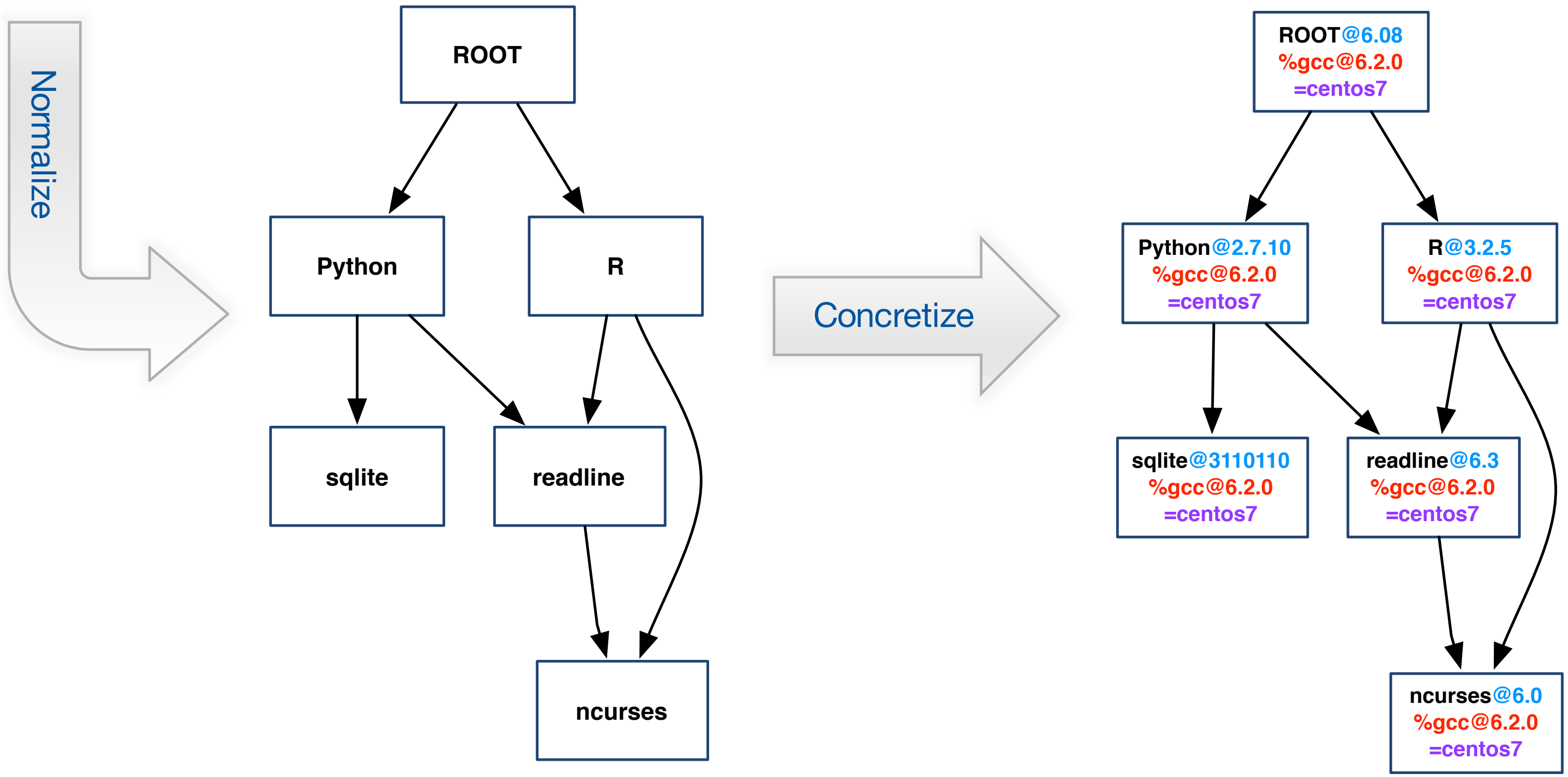
*Spec: build specification

```
$ spack install ROOT                                        unconstrained
$ spack install ROOT@6.06                                   @ custom version
$ spack install ROOT@6.06 %gcc@4.9.3                        % custom compiler
$ spack install ROOT@6.06 cppflags=\"-O3\"                  setting compiler flags
$ spack install ROOT@6.06 ^python@3.5.2 %gcc@4.9.3          ^ dependency information
```

- Different installed ROOT versions can coexisting at the same time
- Each expression is a spec for a particular configuration
  - Constraints are optional
  - Different dependency graphs are generated
- Spec syntax is recursive
  - Full control over the combinatorial build space

Core elements

```
$ spack install ROOT                                    unconstrained
```

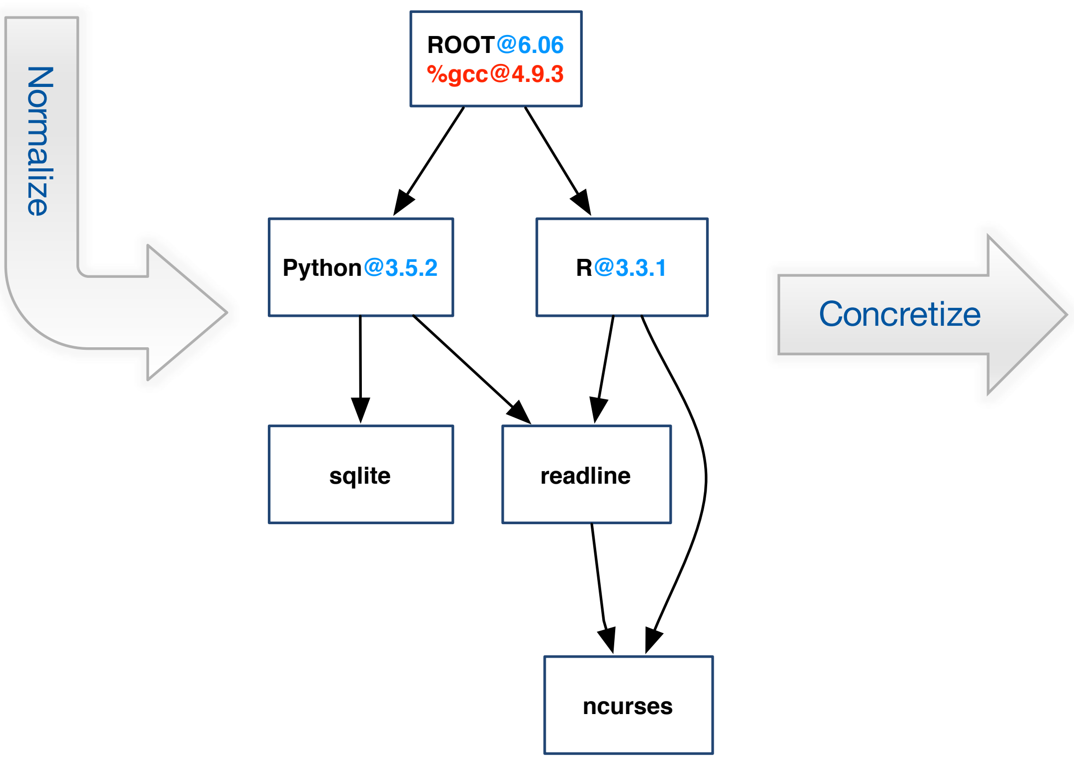


Abstract, normalized spec with some dependencies.

Concrete spec is fully constrained and can be passed to install.

Core elements

```
$ spack install ROOT@6.06 %gcc@4.9.3 ^python@3.5.2 ^R@3.3.1 ^ dependency information
```



Abstract, normalized spec with some dependencies.

Concrete spec is fully constrained and can be passed to install.

Package recipes

- Specify all needed details to build and install a package
- Define the download url to use depending on the version
- Apply patches, check platforms and declare different conditions
- CMake, Autotools, Make… can be used
- Special packages (Python and R packages) use their corresponding parent package to be built

ROOT recipe - example

```
class Root(Package):
    """A modular scientific software framework."""

    homepage = "http://root.cern.ch"

    # Git repository
    version('HEAD', git='http://root.cern.ch/git/root.git',
            tag='HEAD')
    version('8.04', git='http://root.cern.ch/git/root.git',
            tag='8.04')

    variant('graphviz', default=False, description='Enable graphviz support')

    # Dependencies
    depends_on('xrootd')
    ...

    # Conditional dependencies
    depends_on('vc', when='@6.0:')
    ...
    if not 'mac' in plt:
        depends_on('davix')

    # R-packages dependencies
    depends_on('r-rcpp')
    ...
```

Metadata

Dependency information

ROOT recipe - example

```
def install(self, spec, prefix):
    cmake_args = [source_directory]
    cmake_args.extend([
        'CMAKE-Arguments'])
    ...

    with working_dir(build_directory, create=True):
        cmake(*cmake_args)
        make()
        make("install")
```

Using CMake

Declare a new directory to make the build

Configure

Build

Install

Build orchestration and monitoring

Jenkins

- Similar to the approach used with LCGCMake
  - 34m 7.20s only ROOT
  - 2h 21m ROOT + Dependencies

```
==> Successfully installed ROOT
  Fetch: 1m 11.41s.  Build: 32m 55.80s.  Total: 34m 7.20s.
[+] /mnt/build/jenkins/workspace/spack_experimental/BUILDT
28/spack/opt/spack/linux-centos7-x86_64/gcc-4.9.3/ROOT-HEA
[description-setter] Could not determine description.
[PostBuildScript] - Execution post build scripts.
[lcgapp-centos7-x86-64-28] $ /bin/bash -x /tmp/hudson55375
[WS-CLEANUP] Deleting project workspace...[WS-CLEANUP] don
Finished: SUCCESS
```

CDash

Build Name	Update	Configure		Build		Test		
	Files	Error	Warn	Error	Warn	Not Run	Fail	Pass
ROOT@HEAD%gcc@4.9.3~graphviz arch=linux-centos7-x86_64-vsbuxcq						0	2	148
python@2.7.10%gcc@4.9.3~tk~ucs4 arch=linux-centos7-x86_64-46nff4l						0	0	7
zlib@1.2.8%gcc@4.9.3 arch=linux-centos7-x86_64-dcwy4kn						0	0	1

Reasons to consider Spack

- Out-of-the-box
- Modular
- Scalable
- Big community support (and growing)
- User-friendly
- Good approach to distribute our software (Root, Geant,…)

- Still on development (alpha), frequent changes in the core:
  - Lack of conventions of how configuration files should look like
  - Low performance with big graphs of dependencies
  - Testing and monitoring very basic

What about LCGCMake?

- Most part of the mentioned features are already provided by LCGCmake
- Solid workflow based on LCGCMake
- Strong expertise using it
- Self made, adapted to ours needs

Why this interest in Spack now?

Interest in Spack

- Spack covers most part of LCGCMake features
- HSF community is converging to Spack
  - New contributions are rapidly spread
  - No need to remake existing work
- Scalable environment
- Support provided by the community
  - Option to contribute with desired changes
  - Less amount of work

Spack future plans in SFT

Goals to achieve

- Short-term:
  - Test of concept
- Mid-term
  - Build our group projects using Spack
    - ROOT
    - GEANT
  - Collaborate with the community and other HEP organizations on the development
- Long-term
  - Eventual migration from LCGCMake to Spack

Backup

Spack prerequisites

- Python 2.6 or higher
- A C/C++ compiler
- `git` and `curl` commands

Same result, different way

Option 1

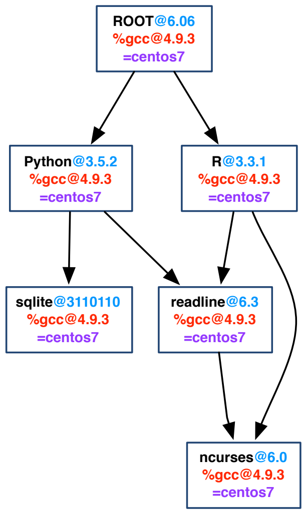
```
$ spack install ROOT@6.06 ^python@3.5.2 ^R@3.3.1 %gcc@4.9.3   ^ dependency information
```

Option 2

```
$ spack install ROOT
```

packages.yaml

```
packages:
    ROOT:
          compiler: [gcc@4.9.3]
          version: [6.06]
    python:
          version: [3.5.2]
    R:
          version: [3.3.1]
```

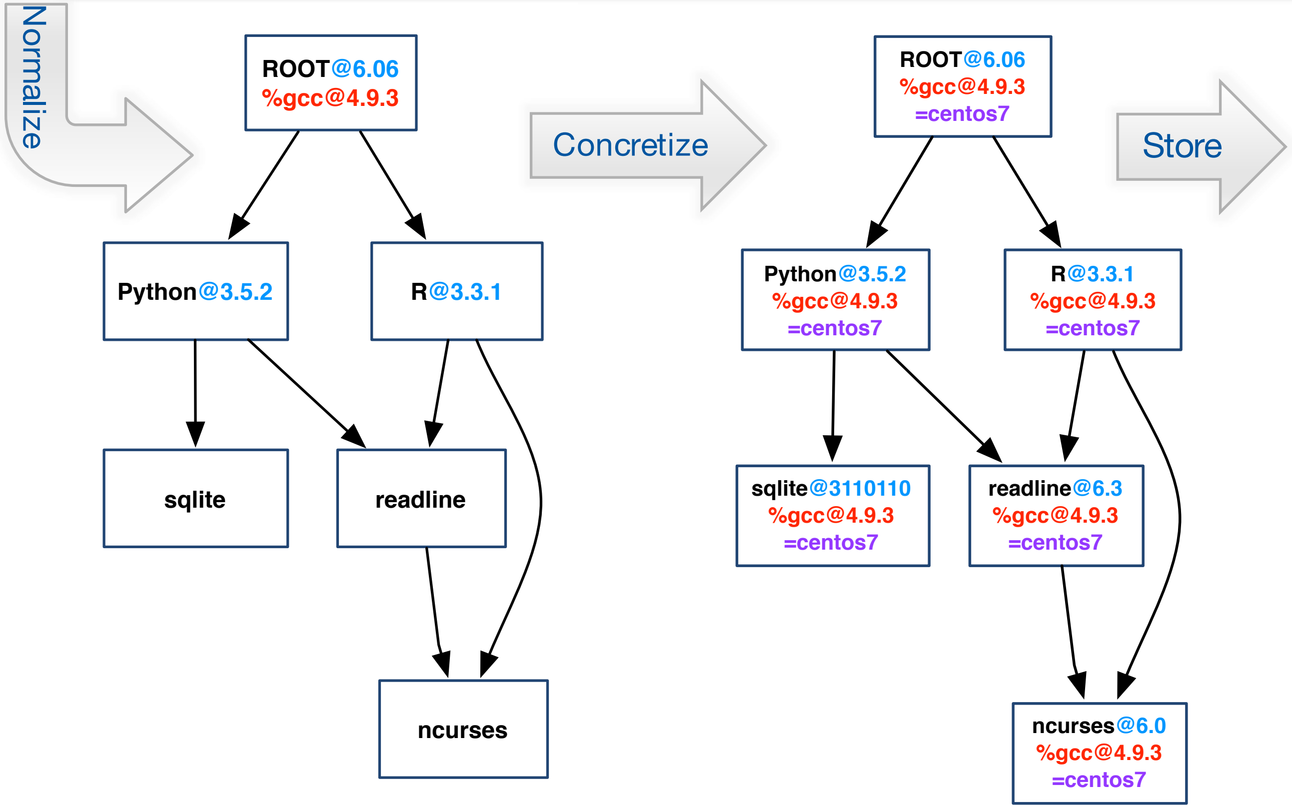


spec.yaml


```
spec:
- ROOT:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.3
    dependencies:
      Python: kszrtkpbzac3ss2ixcjkcorlaybnptp4
      R: bah5f4h4d2n47mgycej2mtrnrivvxy77
    hash: 33hjjhxi7p6gyzn5ptgyes7sghyprujh
    variants: {}
    version: 6.06
- Python:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.3
    dependencies:
      sqlite: teesjv7ehpe5ksspjim5dk43a7qnowlq
      readline: aa4ar6ifj23yijqmdabeakpejcli72t3
    hash: kszrtkpbzac3ss2ixcjkcorlaybnptp4
    variants: {}
    version: 3.5.2
- sqlite:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.3
    dependencies: {}
    hash: teesjv7ehpe5ksspjim5dk43a7qnowlq
    variants: {}
    version: 3110110
...
```

Core elements

```
$ spack install ROOT@6.06 ^python@3.5.2 ^R@3.3.1 %gcc@4.9.3   ^ dependency information
```



spec.yaml


```yaml
spec:
- ROOT:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.3
    dependencies:
      Python: kszrtkpbzac3ss2ixcjkcorlaybnptp4
      R: bah5f4h4d2n47mgycej2mtrnrivvxy77
    hash: 33hjjhxi7p6gyzn5ptgyes7sghyprujh
    variants: {}
    version: 6.06
- Python:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.3
    dependencies:
      sqlite: teesjv7ehpe5ksspjim5dk43a7qnowlq
      readline: aa4ar6ifj23yijqmdabeakpejcli72t3
    hash: kszrtkpbzac3ss2ixcjkcorlaybnptp4
    variants: {}
    version: 3.5.2
- sqlite:
    arch: linux-x86_64
    compiler:
      name: gcc
      version: 4.9.3
    dependencies: {}
    hash: teesjv7ehpe5ksspjim5dk43a7qnowlq
    variants: {}
    version: 3110110
...
```


Abstract, normalized spec with some dependencies.

Concrete spec is fully constrained and can be passed to install.

Detailed provenance is stored with the installed package

Configuration files

- User or Instance level
- Files:
  - `compilers.yaml`
    - Contains information of available compilers
    - Defines path of different compilers
    - Defines custom environment (location of LD_LIBRARY_PATH)
    - Ej: Points to compilers provided by AFS/CVMFS

Configuration files

- User or Instance level
- Files:
  - `packages.yaml`
    - Toolchain
    - Defines every single option to build a package

Configuration files

- User or Instance level
- Files:
  - `mirrors.yaml`
    - Defines areas to look for source packages
    - Mirrors locally or remotely located
    - Ej: Points to EOS tarfiles area as a mirror

Configuration files

- User or Instance level
- Files:
  - `repos.yaml`
    - Location of package recipes

Configurable at different levels

```
$ spack install python [constraints]
```