Program Analysis

Extracting static and dynamic information from a software system

Program Analysis

- Extracting information, in order to present abstractions of, or answer questions about, a software system
- Static Analysis: Examines the source code
- Dynamic Analysis: Examines the system as it is executing

What are we looking for?

- Depends on our goals and the system
  - In almost any language, we can find out information about variable usage
  - In an OO environment, we can find out which classes use other classes, which are a base of an inheritance structure, etc.
  - We can also find potential blocks of code that can never be executed in running the program (dead code)
  - Typically, the information extracted is in terms of entities and relationships

Entities

- Entities are individuals that live in the system, and attributes associated with them.
- Some examples:
  - Classes, along with information about their superclass, their scope, and where in the code they exist.
  - Methods/functions and what their return type or parameter list is, etc.
  - Variables and what their types are, and whether or not they are static, etc.

Relationships

- Relationships are interactions between the entities in the system.
- Relationships include:
  - Classes inheriting from one another.
  - Methods in one class calling the methods of another class, and methods within the same class calling one another.
  - A method referencing an attribute.

Information format

- Many different formats in use
- Simple but effective: RSF
  `inherit TRIANGLE SHAPE`
- TA is an extension of RSF that includes a schema
  `$INSTANCE SHAPE Class`
- GXL is an XML-like extension of TA. A blow-up factor of 10 or more makes it rather cumbersome

Static Analysis

- Involves parsing the source code
- Usually creates an Abstract Syntax Tree
- Borrows heavily from compiler technology but stops before code generation
- Requires a grammar for the programming language
- Can be very difficult to get right

CppETS

- CppETS is a benchmark for C++ extractors
- It consists of a collection of C++ programs that pose various problems commonly found in parsing and reverse engineering
- Static analysis research tools typically get about 60% of the problems right

Example program

```
#include <iostream.h>
class Hello {
public: Hello(); ~Hello();
};
Hello::Hello()
{ cout << "Hello, world.\n"; }
Hello::~Hello()
{ cout << "Goodbye, cruel world.\n"; }
main() {
  Hello h;
  return 0;
}
```

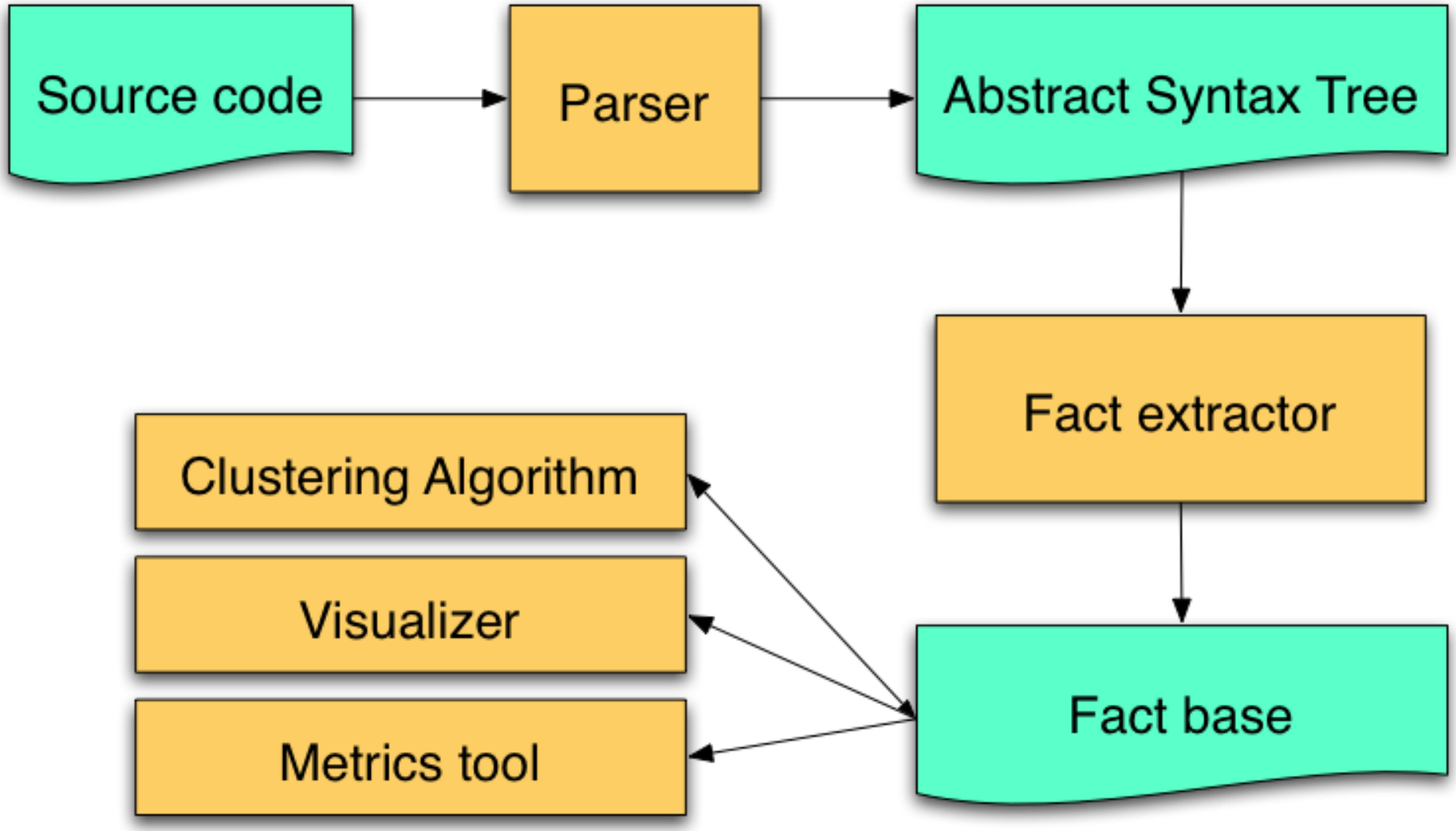
Example Q&A

- How many member methods are in the Hello class?
  Two, the constructor `Hello::Hello()` and destructor `Hello::~Hello()`
- Where are these member methods used?
  The constructor is called implicitly when an instance of the class is created. The destructor is called implicitly when the execution leaves the scope of the instance.

Static analysis in IDEs

- Eclipse displays compilation warnings and errors on the fly, e.g. unused variables
- EiffelStudio automatically creates BON diagrams of the static structure of Eiffel systems
- Rational Rose, as well as some Eclipse plugins, do the same with UML and Java
- Reverse engineers have many other uses for static facts

Static analysis pipeline



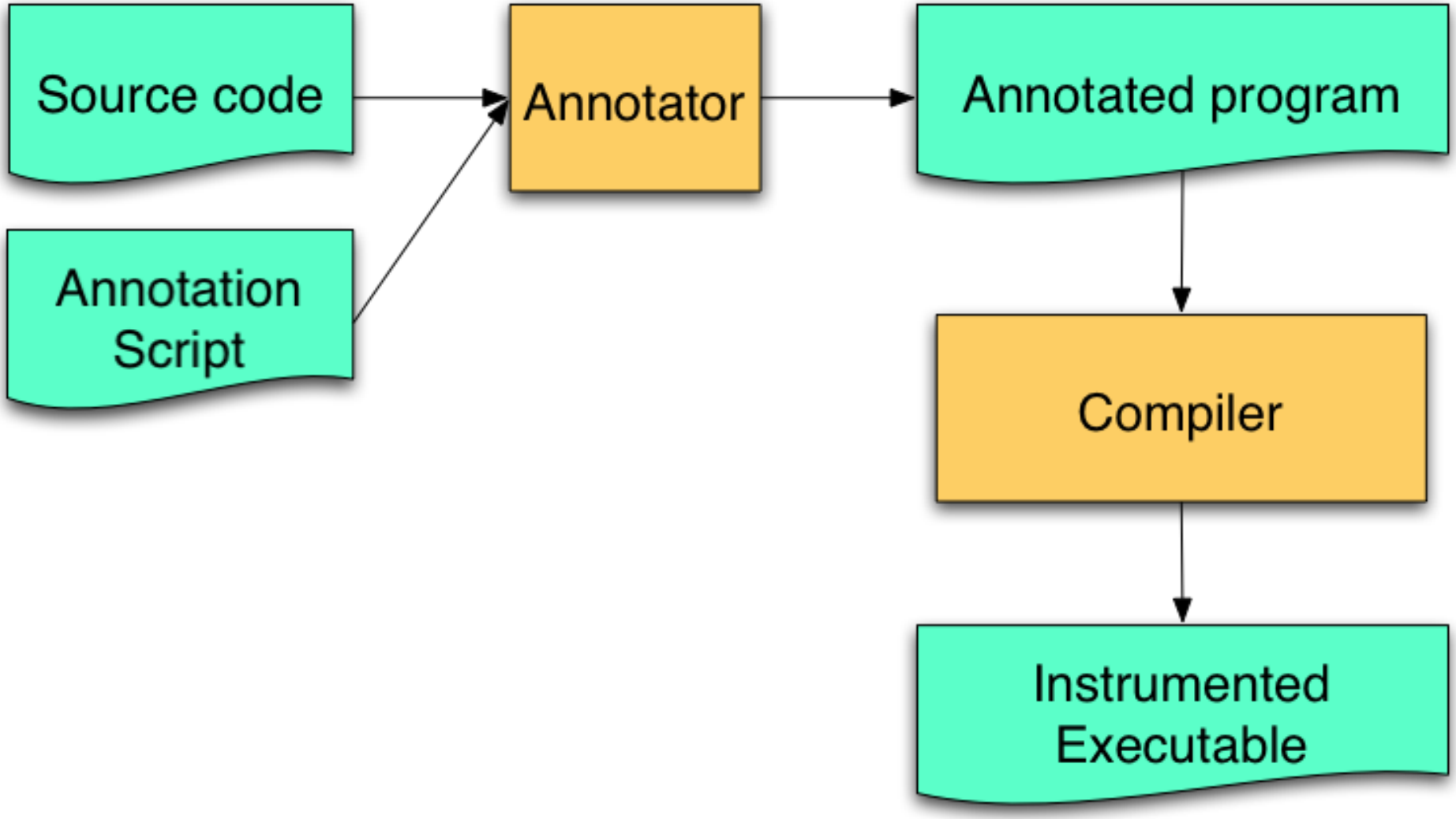
Dynamic Analysis

- Provides information about the run-time behaviour of software systems, e.g.
  - Component interactions
  - Event traces
  - Concurrent behaviour
  - Code coverage
  - Memory management
- Can be done with a profiler or a debugger

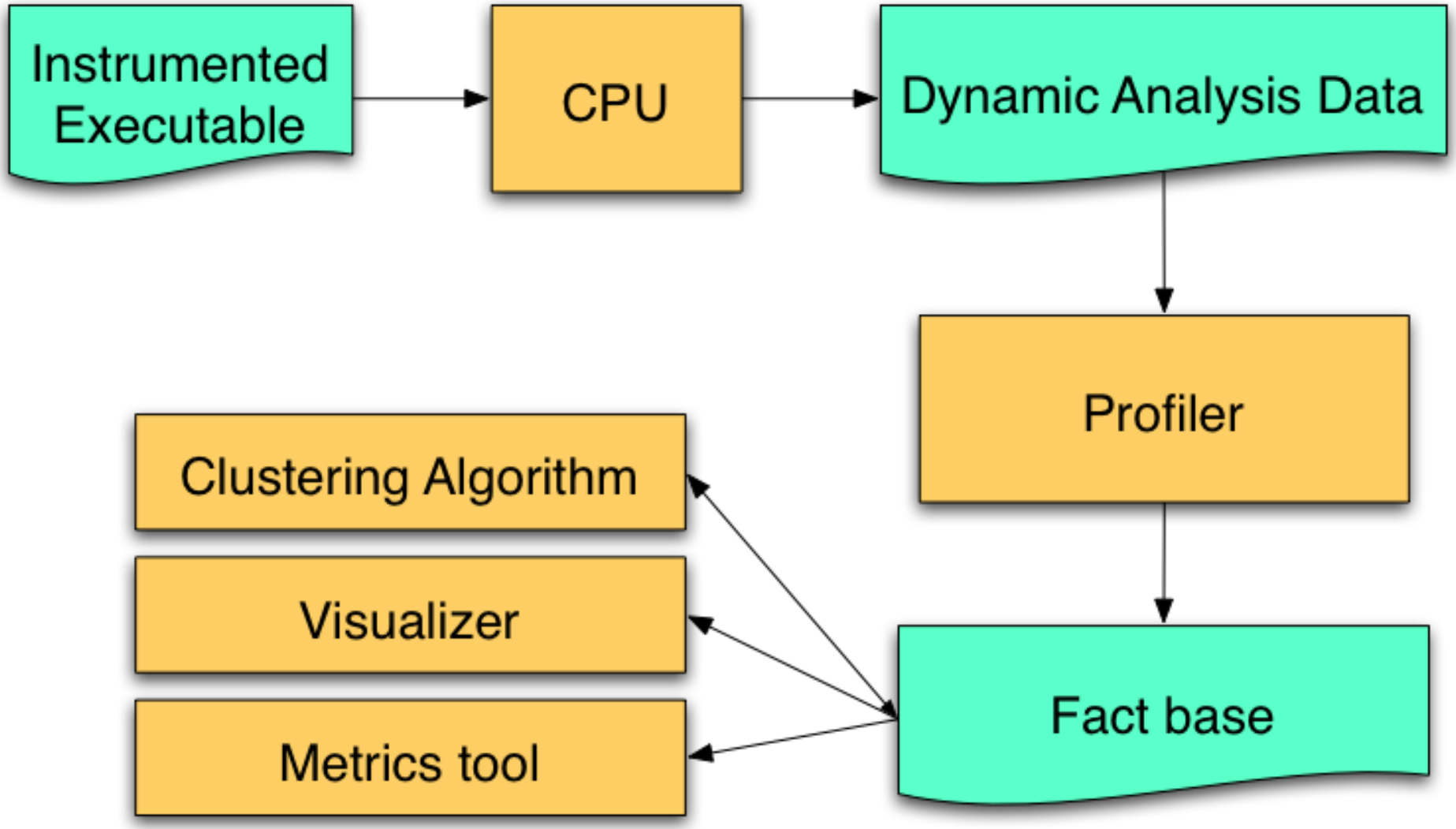
Instrumentation

- Augments the subject program with code that transmits events to a monitoring application, or writes relevant information to an output file
- A profiler can be used to examine the output file and extract relevant facts from it
- Instrumentation affects the execution speed and storage space requirements of the system

Instrumentation process



Dynamic analysis pipeline



Non-instrumented approach

- One can also use debugger log files to obtain dynamic information
- Disadvantage: Limited amount of information provided
- Advantage: Less intrusive approach, more accurate performance measurements

Dynamic analysis issues

- Ensuring good code coverage is a key concern
- A comprehensive test suite is required to ensure that all paths in the code will be exercised
- Results may not generalize to future executions

Static vs. Dynamic

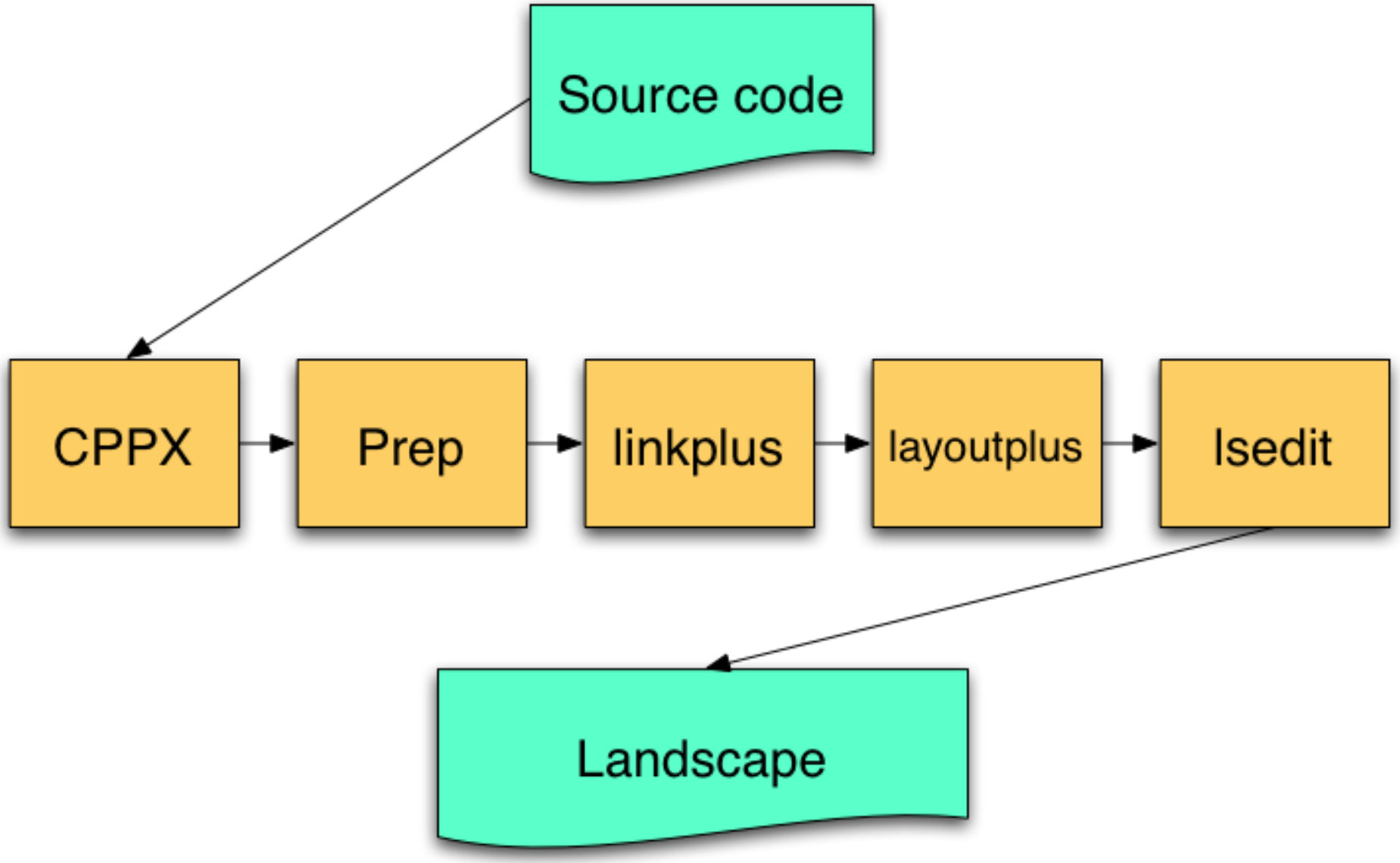
- Reasons over all possible behaviours (general results)
- Conservative
- Challenge: Choose good abstractions

- Observes a small number of behaviours (specific results)
- Precise and fast
- Challenge: Select representative test cases

SWAGKit

- SWAGKit is used to generate software landscapes from source code
- Based on a pipeline architecture with three phases
  - Extract (cppx, bfx, javex)
  - Manipulate (prep, linkplus, layoutplus)
  - Present (lsedit)
- Currently usable for programs written in C/C++ and Java

The SWAGKit Pipeline



CPPX

- C/C++ fact extractor based on gcc
- Extracts facts from one source file at a time
- Facts represent program information in TA format, e.g. `$INSTANCE x integer`
- Can pass normal gcc parameters using the -g option
- In the assignment, we will see two other fact extractors, bfx and javex. They extract facts from compiled code, C and Java respectively.

Prep

- Prep is a series of scripts written in Grok
- Function is to “clean up” facts from cppx so they are in a form which can be usable by the rest of the pipeline.

Grok

- A simple scripting language
- A relational algebraic calculator
- Powerful in manipulating binary relations

Grok Script (1)

```
cat := {"Garfield", "Fluffy"}
mouse := {"Mickey", "Nancy"}
cheese := {"Roquefort", "Swiss"}
animals := cat + mouse
food := mouse + cheese
animalsWhichAreFood := animals ^ food
animalsWhichAreNotFood := animals - food
animalsWhichAreFood
animals - food
#food
mouse <= food
```

Grok Script (2)

```
chase := cat X mouse
chase
eat := chase + mouse X cheese
eat
```

Grok Scripts (3)

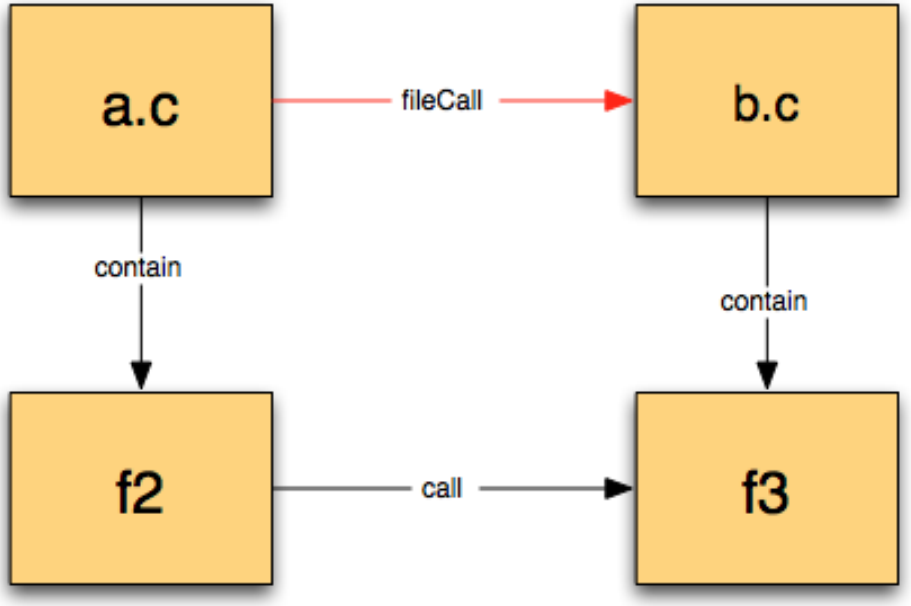
```
{"Mickey"} . eat
eat . {"Mickey"}
eater := dom eat
food := rng eat
chasedBy := inv chase
topOfFoodChain := dom eat - rng eat
bottomOfFoodChain := rng eat - dom eat
bothEatAndChase := eat ^ chase
eatButNotChase := eat - chase
chaseButNotEat := chase - eat
secondOrderEat := eat o eat
anyOrderEat := eat +
```

A more real example

Factbase rawFacts.rsf

```
contain a.c f1
contain a.c f2
contain b.c f3
contain b.c f4
call f1 f2
call f2 f3
call f3 f4
```

We need to compute call relations between files



A bigger real example

```
containFacts := $1
getdb containFacts
d := dom contain
r := rng contain
e := ent contain
roots := d - r
leaves := r - d
toKeep := roots + leaves
toDelete := e - toKeep
cc := contain+
delset toDelete
delrel contain
contain := cc
relToFile contain $2
```

Input: A nested partition of a set of objects
Output: A flattened version of the original partition

linkplus

- Function is to link all facts into one large graph
  - Combines facts residing in separate files
  - Resolves inter-compilation unit relationships
  - Merges header files together
  - Does some cleanup to shrink final graph
- Usage: `linkplus list-of-files-to-link`
- Produces `out.ln.ta`

layoutplus

- Adds
  - Clustering of facts based on contain.rsf (created manually or from a clustering algorithm)
  - Layout information so that graph can be displayed
  - Schema information
- Usage:

  `layoutplus contain.rsf out.ln.ta`
- Produces `out.ls.ta`

lsedit

- View software landscape produced by previous parts of the pipeline
- Can make changes to landscape and save them
- Usage: `lsedit out.ls.ta`