In-source Testing in Answer Set Programming Setup

```
$ mkdir decl24
$ cd decl24
$ python -m venv .env
$ source .env/bin/activate
$ git clone git@github.com:ejgroene/declarative2024.git
$ cd declarative2024
$ pip install .
```

type

or

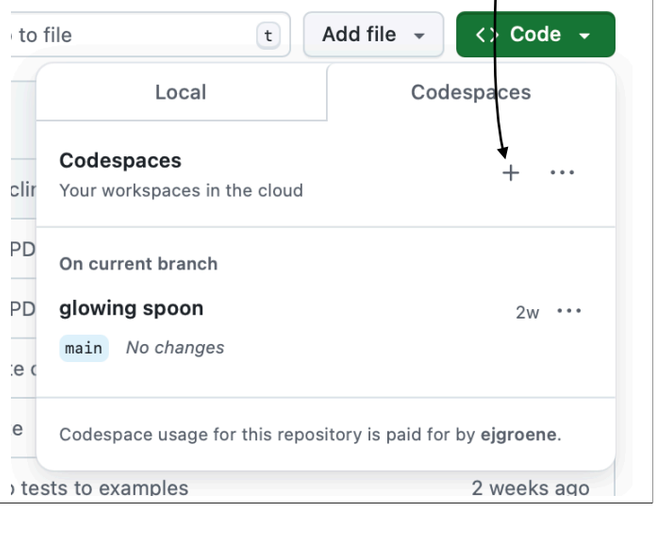
click

Instructions at:

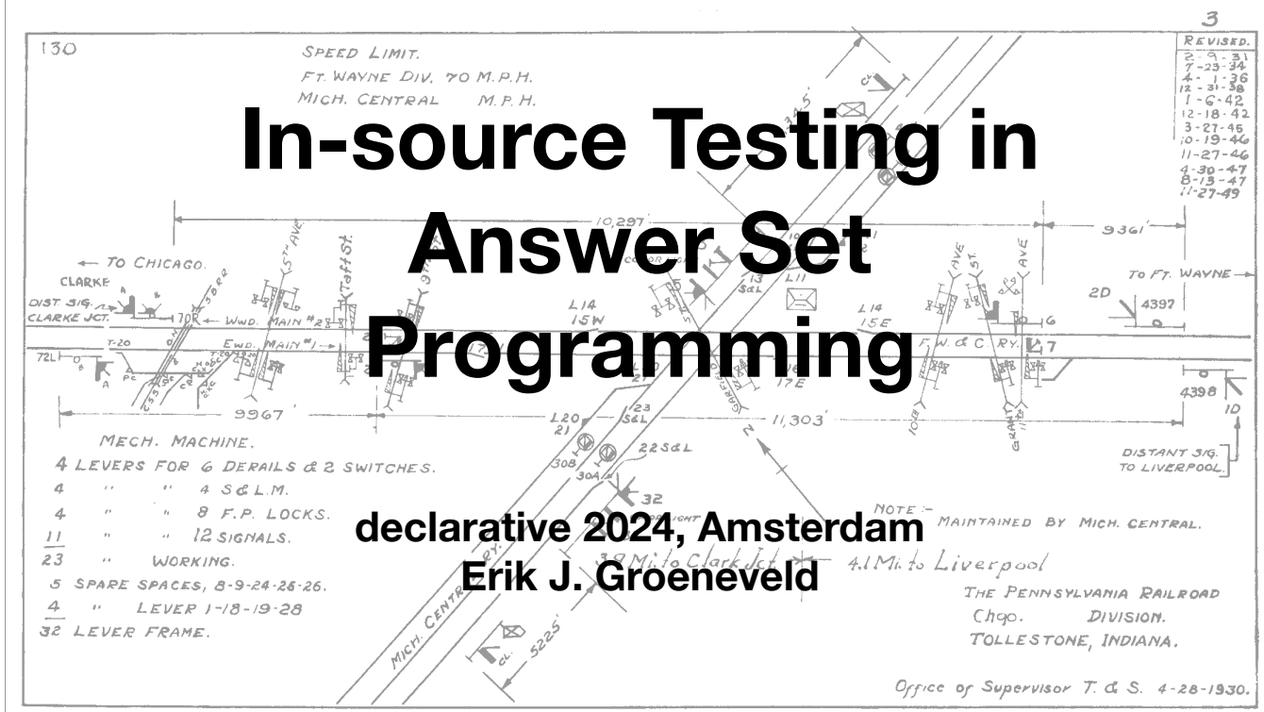
https://github.com/ejgroene/declarative2024

or via:

https://declarative.amsterdam/program



Instructions als via link on program, follow link, README.md.
All examples, challenges and code snippets are there.



In-source Testing in Answer Set Programming

declarative 2024, Amsterdam

Erik J. Groeneveld

Welcome

Contents

- Setup code environment
- Who am I?
- What is In-source Testing
- How to test ASP programs
- Potassco, clingo
- Hands-on
- Questions

Code samples in repo
Invite to join with code.

I love programming

- 1996: Baan R&D (research engineer)
- 1999: Software Engineering Research Centre (SERC)
- 2001: Owner of Seecr (search with Lucene)
- 2024: Independent (finally)

- Programming Languages
  - (GW-)Basic
  - Pascal
  - C/C++
  - Python
  - Clojure
  - Answer Set Programming

- Metaprogramming
  - Efficiency
  - Size reduction
  - Patterns & idioms
  - Metaclasses
  - Integration
  - Extreme Programming
  - Push boundaries

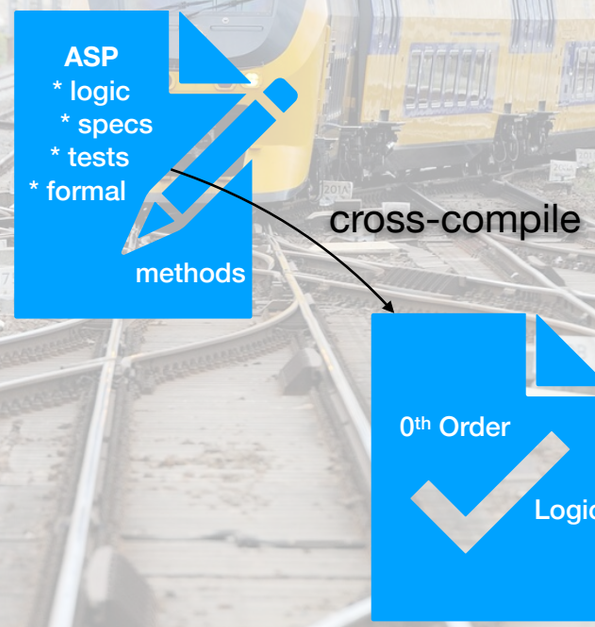
Seecr did Python before it became popular.

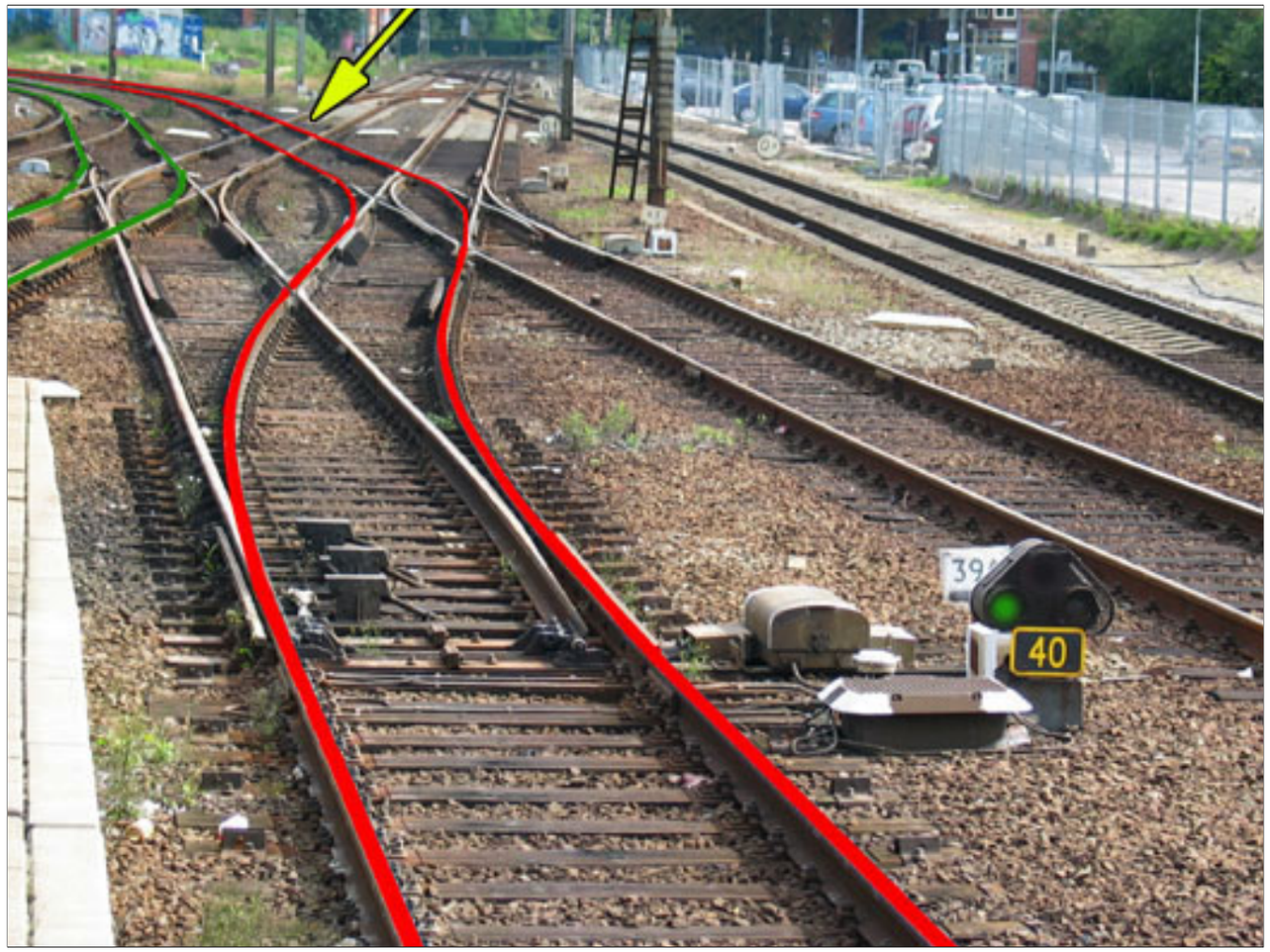
How it started…

- Railway Interlocking
- 0th Order Logic

- Design Automation

- Formal Specification
- **Unit Testing**
- Formal Methods
- => Higher Order Logic





Track, switches, routes and signals.

Higher order logic

```
normaal_voorwaarde_h(T, Rijweg)  :-
    rijweg_ingesteld(T, Rijweg),
    bgz(T, Rijweg),
    sectie_vrij(T, Rijweg, Sectie)    : rijweg_sectie(Rijweg, Sectie);
    sts_passage(T, Rijweg, Wissel)    : flankzonebewaking(Rijweg, Wissel);
    virtueel_h(T, Rijweg)             : vierdraadsAPB(Rijweg);
    virtueel_d(T, Rijweg)             : rijrichtingskering_zonder_bloksein(Rijweg).
```

variables

sets (such-that)

cross-compile

```
BOOL 190-192-H = (190-192-BGZ * 190-GZ-CS * 190-TP * 191-TP)
BOOL 192-188-H = (189-TP * 190-TP * 191-TP * 192-188-BGZ *
                  192-GZ-CS * 192-TP * A178-TP)
BOOL 192-190-H = (174C-TP * 190-TP * 191-TP * 192-190-BGZ
                  192-GZ-CS * 192-TP)
```

ASP: rule, head, body, conjunction/and
0-orde: straight forward logic, top to bottom, no choice, implicit time
1-orde: variables
2-orde: sets (such-that)
Look at rule: clearly we need tests: 4 conditions, no less!

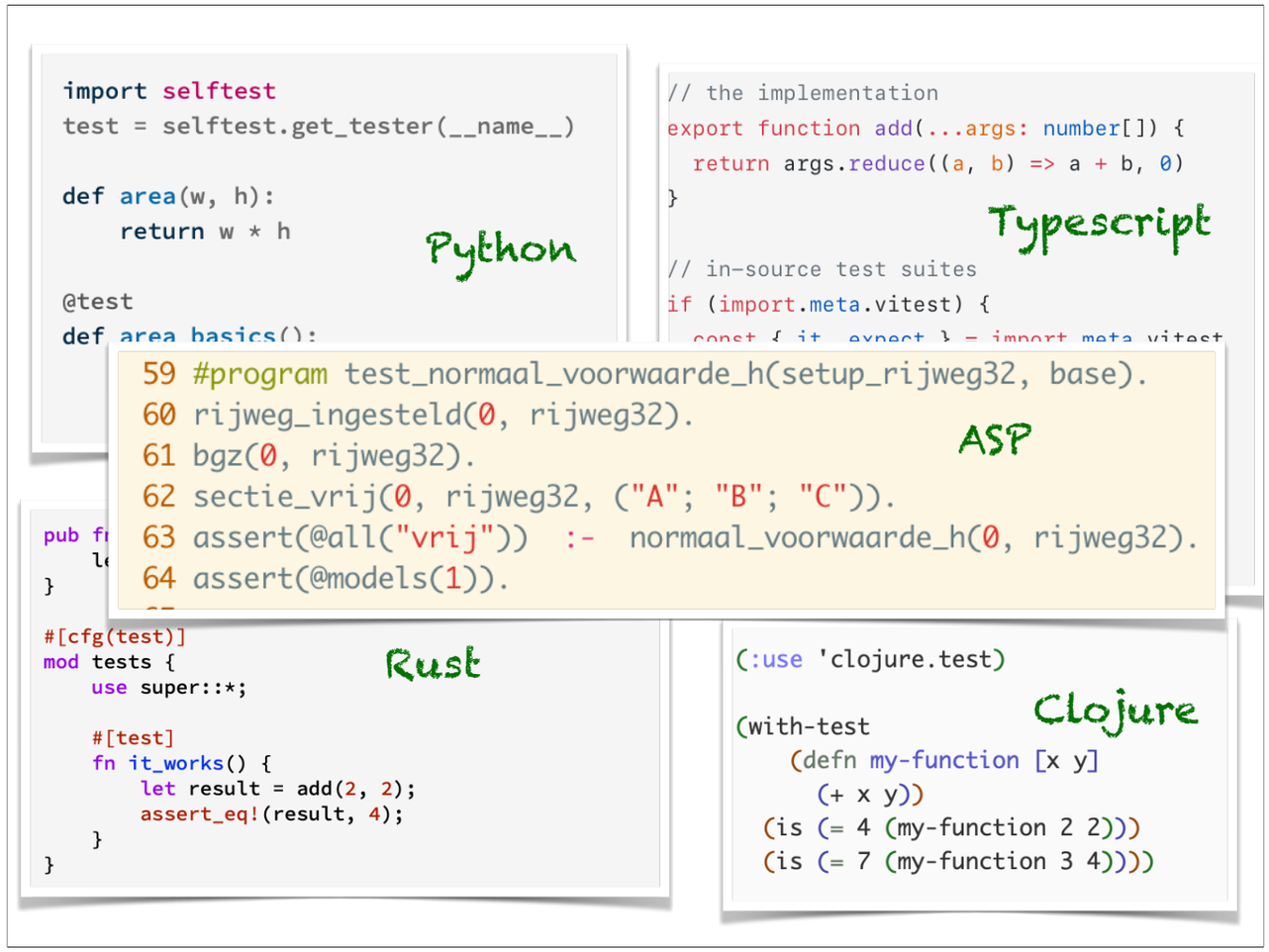
What is In-Source Testing?

- Put your test right **between** your code.
  - Same language/file/class/function/compilation unit
  - Runs on **every** import

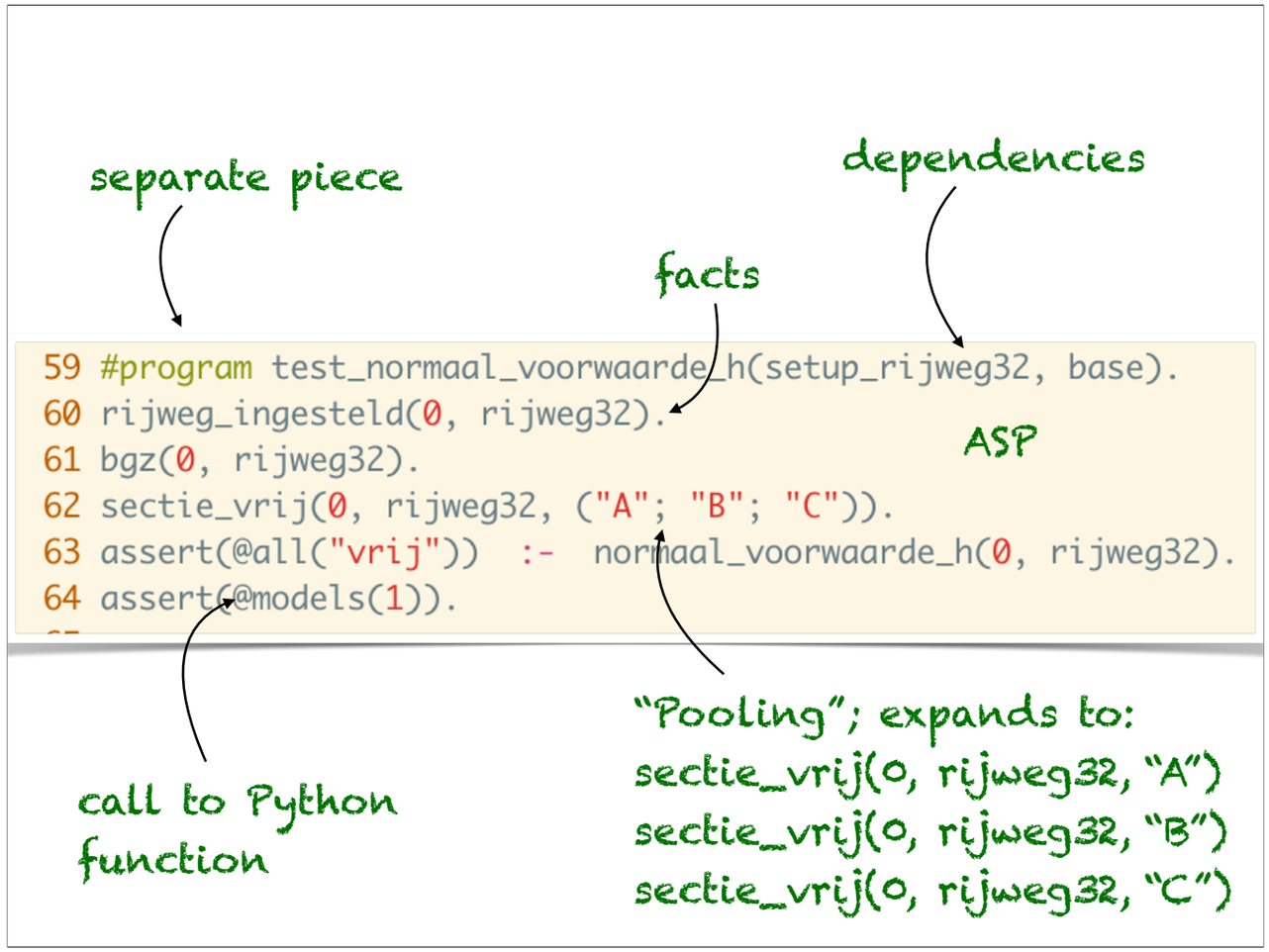
real Python code

```
172
173 def sym2ids(body):
174     """ Create a set of id's for each free symbol in the body """
175     return {mk_id(h, t) for h, t in (get_time(s) for s in body.free_symbols)}
176
177
178 @test
179 def ids_for_symbols():
180     test.eq({'a(0)'}, sym2ids(sym('a')))   # simple symbol
181     test.eq({'a(0)', 'b(0)'}, sym2ids(sympy.And(sym('a'), sym('b'))))  # compound symbol
182
```

Tests also document the behaviour.
Executed on demand: on import
Regardless the context
CONTEXT is the CURRENT software CRISIS



Almost silent until 2024, now, blogs begin to appear
Clojure is a bit more integrated: special function with asserts (not separate tests)
PRESS for ASP example: explain #program, facts and @-callouts



Explain #program, facts and @-callouts
; = pool expansion

Why In-source?

- Reduce test code base **maintenance**
- Automatic and **deterministic** collection of tests (import)
- Automatic **subset** selection
- Easier **refactoring** (move code)
- **Intuitive** test shifting from unit/integration/system
- Test different **environments** (tests part of program)
- **Less framework'ish** in general (more control, less magic)

You really have to try and experience it.

and now: ASP

Answer Set Programming

soda.lp

```
1 % choice
2 beer; wine; soda.
3
4 % rules
5 drunk :- beer.
6 drunk :- wine.
```

```
Solving...
Answer: 1
soda
Answer: 2
wine drunk
Answer: 3
beer drunk
```

party.lp

```
1 % available drinks
2 beverage(wine, 11).
3 beverage(beer,  5).
4 beverage(soda,  0).
5
6 % choice: party or not
7 { party }.
8
9 % when party, drink some
10 { drink(D, P) }  :-  party,  beverage(D, P).
11
12 #show party/0.
13 #show drink/2.
```

- What we need today:
  - facts
  - rules & variables
  - constraints
  - conditional literals
  - aggregates
  - optimisation

explained when we meet them

soda.lp: facts, choice, disjunction
clingo 0 soda.lp. => 1 answer ‘soda’ and not ‘drunk’. (Drunk is not there)
party.lp: set choice: 0 or more
2 choices: party or not, drink some or not. (See also party2.lp)
explain models

```
% facts (atoms)
beer(valdieu, 8).
beer(radler, 0).
pils(gulpener, 6).
ale(kilkenny, 4).

% rule: head :- body
alcoholic(B)  :-  beer(B, _).

% disjunction
beer(B, A)  :-  pils(B, A).
beer(B, A)  :-  ale(B, A).

% conjunction
special(B)  :-  beer(B, A),  A > 5.

% choice
{ drink(radler, 0) }.
{ drink(B, A) }  :-  beer(B, A).

% constraint, it cannot be that...
:-  drink(B, A),  A = 0.

% conditional literals ('such that')
specials(T)  :-  T = { beer(B, A)  :  special(B) }.

% output control
#show drink/2.
#show specials/1.
```

crash course ASP

see Codespace

we'll repeat it when we meet them again

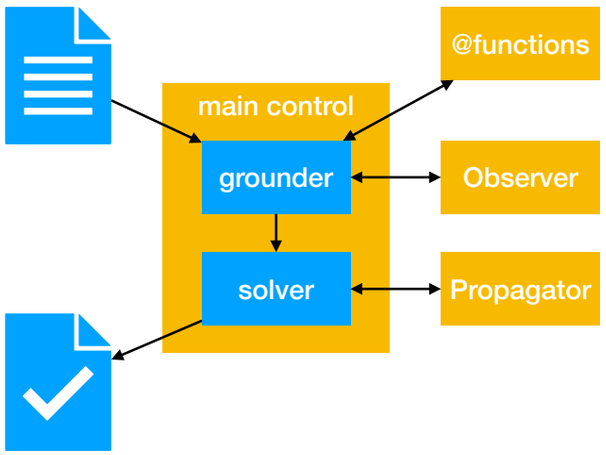
see code base!

not seen yet:

* disjunction
* constraint
* #count aggregate

Potassco

- an ASP implementation by University of Potsdam
- Try it online: https://potassco.org/clingo/run/



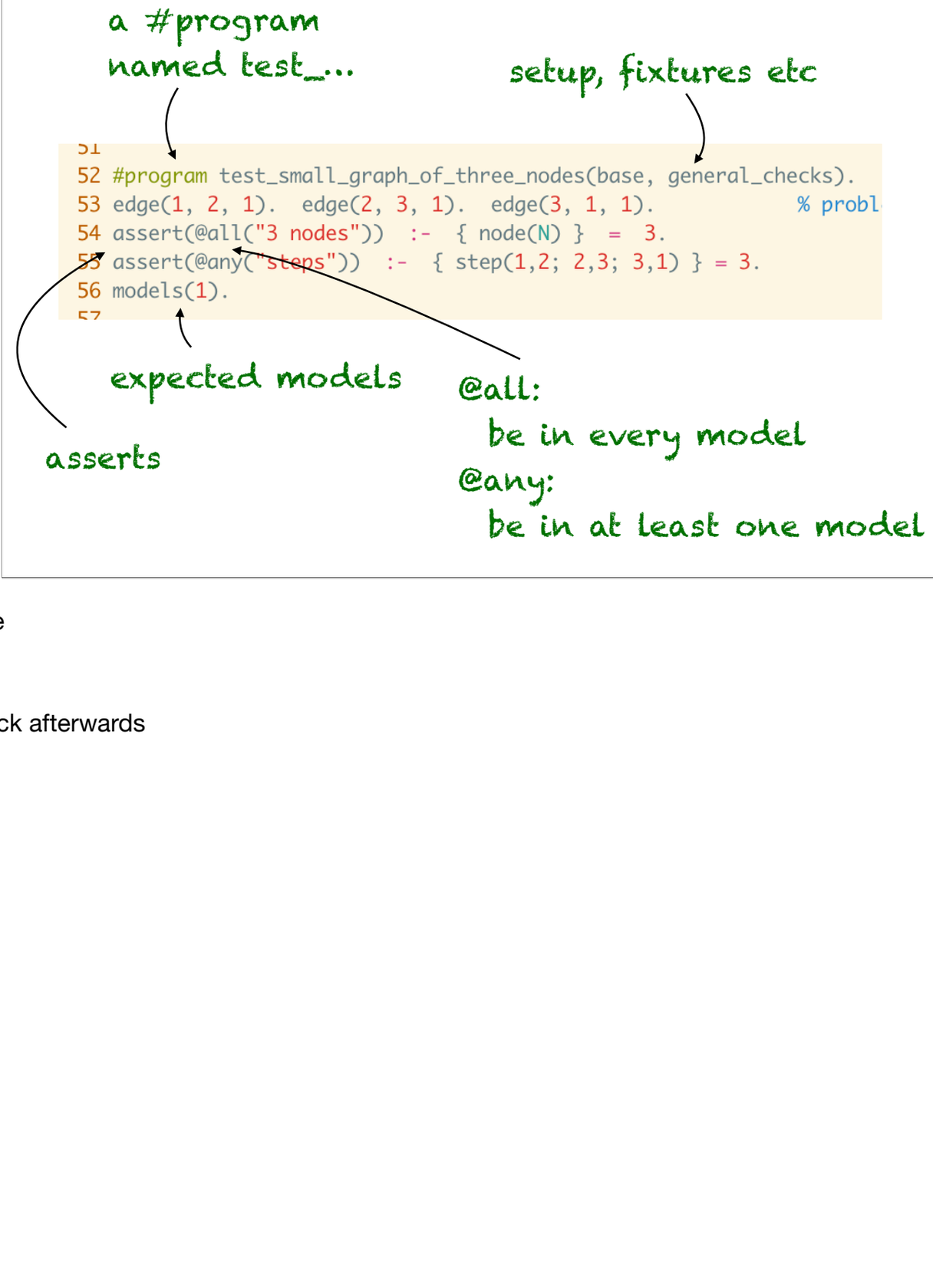
- **API's**
  - C++/Python/Lua
  - Embedded #script
  - Callout @function
  - Intercept
    - Observer
    - Propagator
  - Main control

```
#script (python)
from clingo import Function

def make_atom(name, arg):
    return Function(name.name, [arg])

#end.

beverage(wine, 11).
beverage(beer,  5).

drink(@make_atom(B, P))  :-  beverage(B, P).
```

IF time: show what party.lp gets ground to:

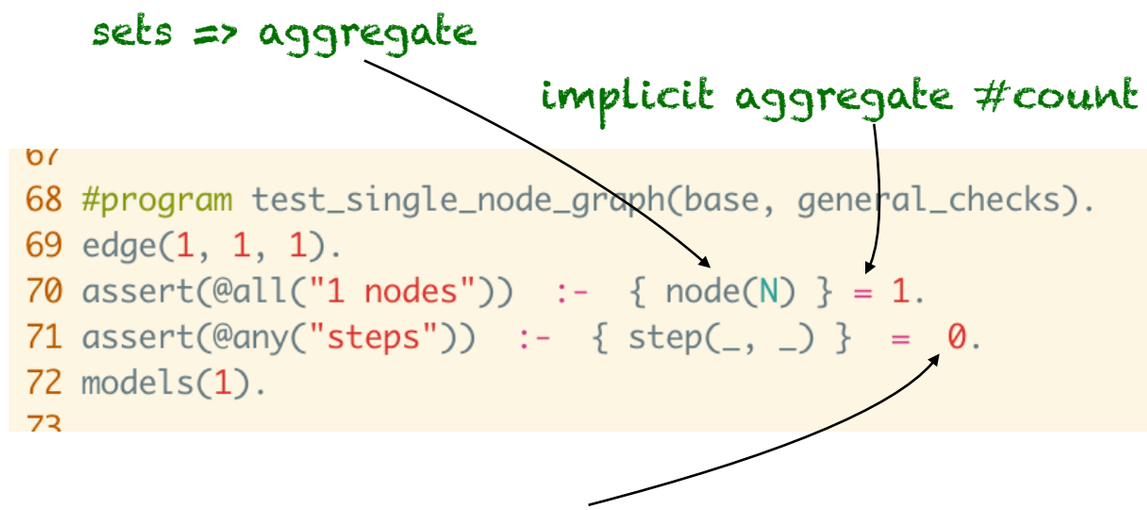
```
beverage(wine,11).
beverage(beer,5).
beverage(soda,0).
{party}.
{drink(wine,11)}:-party.
{drink(beer,5)}:-party.
{drink(soda,0)}:-party.
#show party/0.
#show drink/2.
```

How to test ASP?



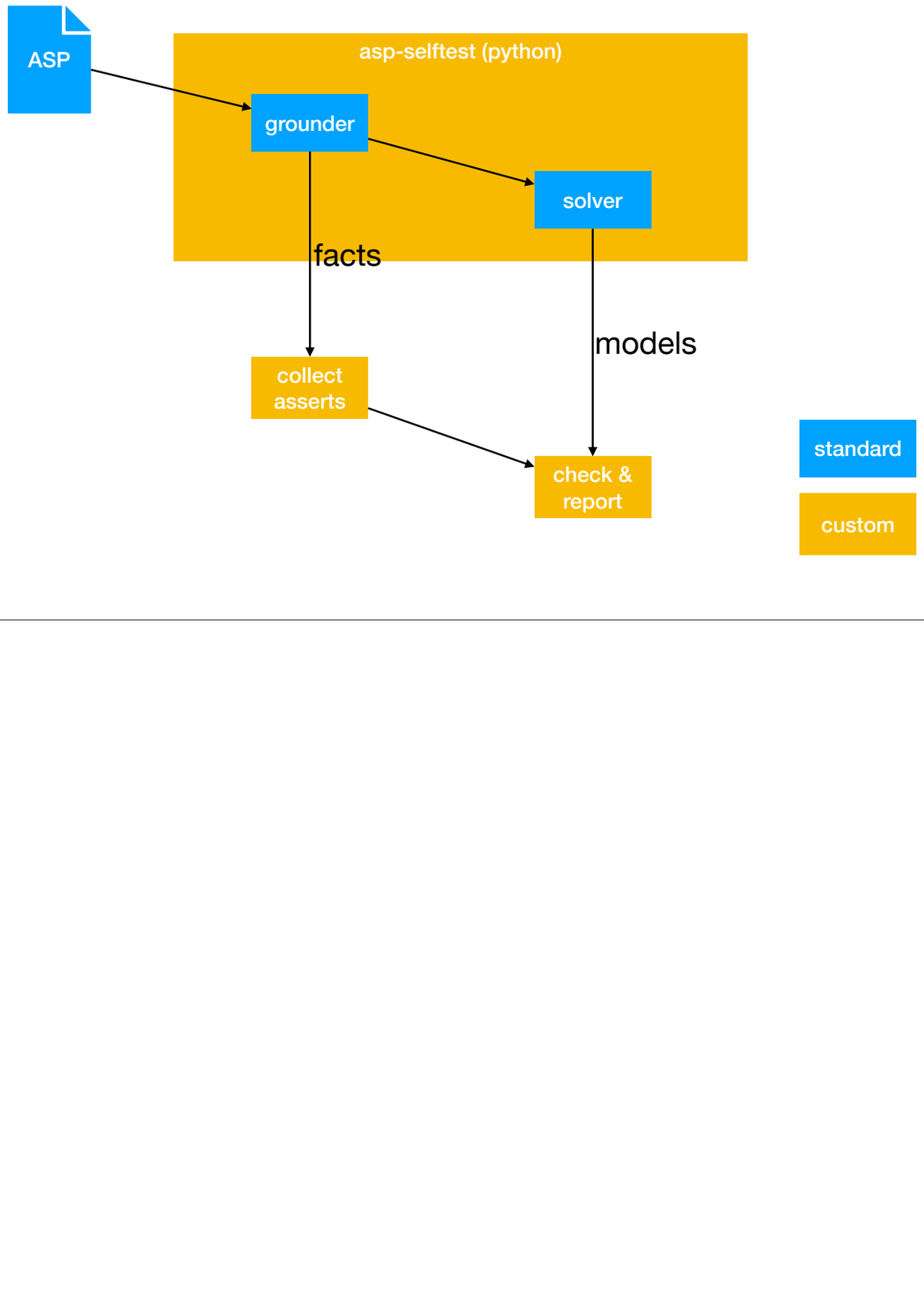
an ACTUAL test from the main challenge
a DIFFERENT test than before
@all/@any are Python call-outs.
They register the assert so it can be check afterwards

ASP Test Idioms

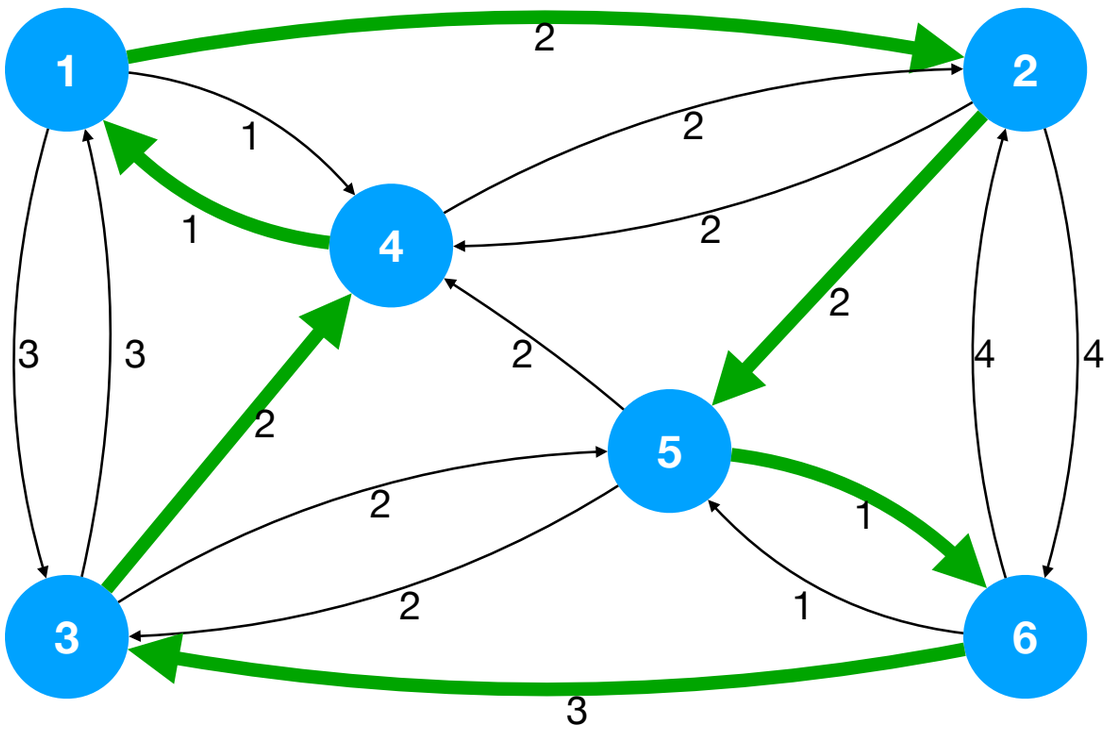


aggregates: #sum, #count, #minimize operate on sets.
not step(…) would be optimised away

Design


ASP
asp-selftest (python)
grounder
solver
facts
models
collect asserts
check & report
standard
custom

Hamiltonian Path



Hands-On (after the break)

```
%%%%%%%% Problem Instance %%%%%%%%

% edge(From, To, Cost). We use facts.
edge(1 ,2 ,2). edge(2 ,4 ,2). edge(3 ,1 ,3). edge(4 ,1 ,1). edge(5 ,3 ,2). edge(6 ,2 ,4).
edge(1 ,3 ,3). edge(2 ,5 ,2). edge(3 ,4 ,2). edge(4 ,2 ,2). edge(5 ,4 ,2). edge(6 ,3 ,3).
edge(1 ,4 ,1). edge(2 ,6 ,4). edge(3 ,5 ,2).               edge(5 ,6 ,1). edge(6 ,5 ,1).


assert("6 nodes")  :-  { node(N) } = 6.
assert("incident in", N)  :-  node(N),  edge(_, N, _).
assert("incident out", N)  :-  node(N),  edge(N, _, _).
assert("valid costs", S, E) :- edge(S, E, C),  C > 0,  C < 10.
assert("minimal cost")  :-  #sum { C, A, B  :  step(A, B),  edge(A, B, C) } < 12.


%%%%%%%% Problem Encoding %%%%%%%%

%%%% Preparation %%%%
% Infer nodes from edges. We use a simple disjunctive rule: head :- body.
node(N)  :-  edge(N, _, _).    % variable N,  wildcard _
node(N)  :-  edge(_, N, _).    % disjunction/or


%%%% Generation %%%%
% Choose an arbitrary step. We use conditional literal ("such that") + choice
step(A, B)  :  edge(A, B, _).

% if you have one step, choose a connected one, but not back.
step(B, C)  :  edge(B, C, _),  C <> A   :-   step(A, B).

% Path to given node via step's. We use a disjunctive rule with conjunction
```

hamiltonian-cycle-1.lp

We can’t fix the test ‘steps’ because:
- first we need to understand the problem, so
- run the code with clingo 0
- there are 3 models with 3 paths
- our test asserts 1 specific path
- we cannot differentiate models

hamiltonian-cycle-2.lp

We number of steps issues a warning:
- first understand the problem:
- Clingo expands this rule for every node N.
- The rule gets instantiated (grounded) for every node N
- but the head remains the same every time
- so we get a disjunction!
- this is usually not intended, so it warns about it
- fix it by introducing N in the head, for example:

assert(@all(“number of steps”, N)) :- …

```
37 assert(@all("number of steps"))    :-  { step(A, B) } = S,  S = { node(N) }.
```

```
hamiltonian-cycle-3.lp
1/2 challenges:

1. relate steps and cost
- there must be a model with specific costs and steps:
  assert(@any("steps and costs"))  :-
       cost(11),  { step(1,2; 2,5; 5,6; 6,3; 3,4; 4,1) } = 6
```

```
37 assert(@all("number of steps"))    :-  { step(A, B) } = S,  S = { node(N) }.
```

```
hamiltonian-cycle-3.lp
2/2 challenges:

1. The challenge is to change the program so it accepts
   single node graphs.

Solution:
 next slide
```

Solution:

1. only choose a step when more than 1 node:
```
step(A, B)  :  edge(A, B, _)  :-  { node(_) } > 1.
```

2. adjust the constraint on path:
```
:-  node(N),  step(_, _),  not path(N).
```

3. relax the ‘no self reference’ assert by adding:
```
assert(@all("no self reference"))  :-  node_count(1).
```

4. allow for costs of zero in ‘cycle costs’:
```
assert(@all("cycle cost”))  :-
            node_count(N),  cost(S),  0 <= S < 100.
```

5. only checks steps when more than 1 node:
```
assert(@all("number of steps", N)) :-
           node_count(N),  N = { step(A, B) },  N > 1.
```