



MARIO ALVIANO
DeMaCS@UNICAL

Structured Declarative Language

Joint work with
Carmine Dodaro and Ilaria Raffaella Vasile



GitHub Repository
<https://github.com/dodaro/SDL>



Declarative Programming

Expresses the **logic** of a computation
without describing its **control flow**

Declarative Programming

Expresses the **logic** of a computation
without describing its **control flow**

Answer Set Programming
is (at its core) declarative

Declarative Programming

Expresses the **logic** of a computation
without describing its **control flow**

Answer Set Programming
is (at its core) declarative

But is ASP a
good speaker?

Relational Algebra vs Structured Query Language (SQL)

π_{id}

$\sigma_{username = "root" \text{ AND } password = "toor"} user$

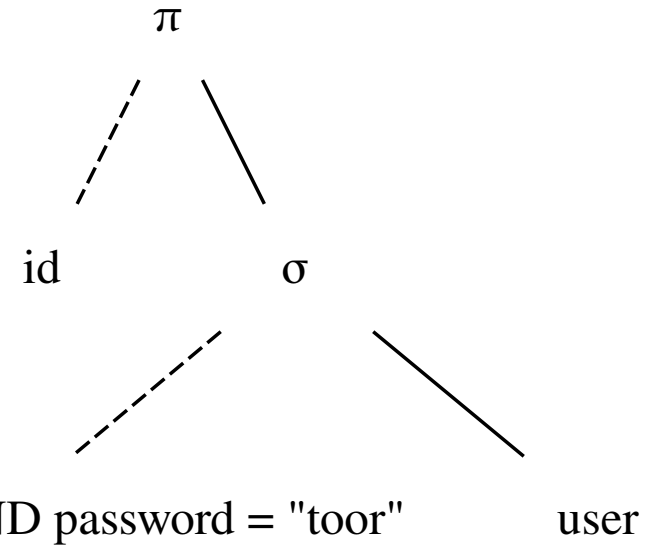
Relational Algebra vs Structured Query Language (SQL)

π_{id}

$\sigma_{username = "root" \text{ AND } password = "toor"} user$

OK in papers (for compactness)

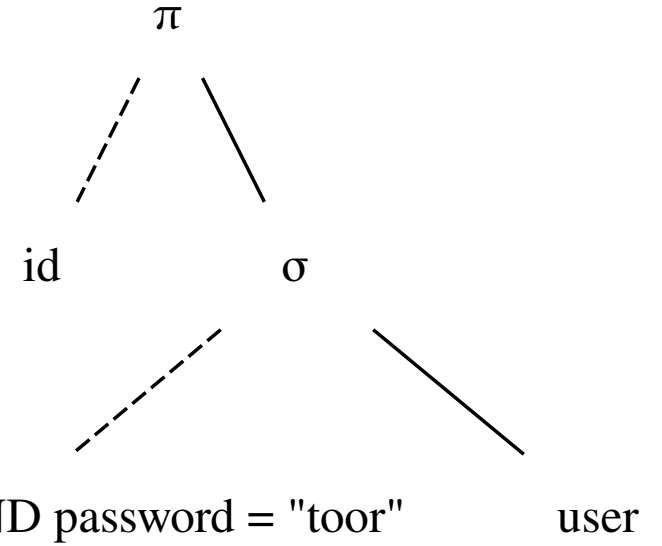
OK for the engine (to optimize)



Relational Algebra vs Structured Query Language (SQL)

π_{id}
 $\sigma_{username = "root" \text{ AND } password = "toor"} user$

OK in papers (for compactness)
OK for the engine (to optimize)



But programmers
prefer to express
themselves this way

```
SELECT id  
FROM user  
WHERE username = 'root' AND password = 'toor'
```

ASP for Papers vs ASP for Programmers

$\{assign(X, C) : color(C)\} = 1 \leftarrow node(X).$

$\perp \leftarrow edge(X, Y), assign(X, C), assign(Y, C).$

$[1@1, C] \leftarrow assign(_, C).$

ASP for Papers vs ASP for Programmers

```
{assign(X, C) : color(C)} = 1 ← node(X).
```

```
⊥ ← edge(X, Y), assign(X, C), assign(Y, C).
```

```
[1@1, C] ⇐ assign(_, C).
```

```
% guess one color for each node
```

```
{assign(X,C) : color(C)} = 1 :- node(X).
```

```
% adjacent nodes must have different colors
```

```
:- edge(X,Y), assign(X,C), assign(Y,C).
```

```
% minimize the number of used colors
```

```
:- assign(_,C). [1@1, C]
```

ASP for Papers vs ASP for Programmers

```
{assign(X,C) : color(C)} = 1 ← node(X).
```

```
⊥ ← edge(X,Y), assign(X,C), assign(Y,C).
```

```
[1@1,C] ⇝ assign(_,C).
```

But... they are essentially
the same!

```
% guess one color for each node  
{assign(X,C) : color(C)} = 1 :- node(X).  
  
% adjacent nodes must have different colors  
:- edge(X,Y), assign(X,C), assign(Y,C).  
  
% minimize the number of used colors  
:- assign(_,C). [1@1, C]
```

The SAME... but for Different Purposes!

In a paper, you
want to be concise

The SAME... but for Different Purposes!

In a paper, you
want to be concise

**But a long-standing codebase needs
maintenance, readability, and robustness**

The SAME... but for Different Purposes!

In a paper, you
want to be concise

But a long-standing codebase needs
maintenance, readability, and robustness

Three downsides of long-standing ASP codebases

- 1) Propagating Changes
- 2) Use of Object Variables
- 3) Lack of Semantic Annotations

ASP Example

```
%      cab( cab_id, driver)
% customer(cust_id,  name, title)
%  assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

ASP Example

If driver is moved in a different predicate,
both rules HAVE TO be changed!

```
% cab( cab_id, driver)
% customer(cust_id, name, title)
% assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

ASP Example

If driver is moved in a different predicate,
both rules HAVE TO be changed!

```
% cab( cab_id, driver)
% customer(cust_id, name, title)
% assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

The rules don't use the driver...
but the driver affect the rules!

ASP Example

If driver is moved in a different predicate, both rules HAVE TO be changed!

```
% cab( cab_id, driver)
% customer(cust_id, name, title)
% assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

The rules don't use the driver...
but the driver affect the rules!

Object variables are often shorts and used inconsistently (e.g., C and C')

ASP Example

If driver is moved in a different predicate, both rules HAVE TO be changed!

```
% cab( cab_id, driver)
% customer(cust_id, name, title)
% assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

The rules don't use the driver...
but the driver affect the rules!

Object variables are often shorts and used inconsistently (e.g., C and C')

BTW... is assign(cust, cab) or assign(cab, cust)?
There is no semantic annotation within the syntax!

Structured Declarative Language (SDL)

Like SQL... but mapping to ASP!

Structured Declarative Language (SDL)

Abstraction and Simplification

Problems are expressed in a language
closer to English

Like SQL... but mapping to ASP!

Structured Declarative Language (SDL)

Like SQL... but mapping to ASP!

Abstraction and Simplification

Problems are expressed in a language
closer to English

Qualifying Names for Attribute Access

Dot notation, as in OOP!
No order of attributes,
no object variables

Structured Declarative Language (SDL)

Like SQL... but mapping to ASP!

Abstraction and Simplification

Problems are expressed in a language closer to English

Qualifying Names for Attribute Access

Dot notation, as in OOP!
No order of attributes,
no object variables

Automatic Attribute Tracking

References to records are tracked to ease debugging, and to avoid comparing apples and oranges

Structured Declarative Language (SDL)

Like SQL... but mapping to ASP!

Abstraction and Simplification

Problems are expressed in a language closer to English

Qualifying Names for Attribute Access

Dot notation, as in OOP!
No order of attributes,
no object variables

Automatic Attribute Tracking

References to records are tracked to ease debugging, and to avoid comparing apples and oranges

SDL maps to semantically annotated atoms

```
assign(  
  customer(CUST_ID, NAME, TITLE),  
  cab(CAB_ID, DRIVER)  
)  
vs  
assign(C,C')
```

SDL Example

ASP

```
% cab( cab_id, driver)
% customer(cust_id, name, title)
% assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

```
record Cab:          id: int, driver: str;
record Customer:    id: int, name: str, title: str;
record Assign:      customer: Customer, cab: Cab;

guess from Customer exactly 1
  Assign from Cab
    where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
  count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```


SDL Example

A Cab has attributes
id (an integer) and
driver (a string)

```
% cab( cab_id, driver)
% customer(cust_id, name, title)
% assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

```
record Cab:          id: int, driver: str;
record Customer:   id: int, name: str, title: str;
record Assign:     customer: Customer, cab: Cab;

guess from Customer exactly 1
  Assign from Cab
    where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
  count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

SDL Example

ASP

```
% cab( cab_id, driver)
% customer(cust_id, name, title)
% assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

A Cab has attributes
id (an integer) and
driver (a string)

Assign has attributes
customer (being a Customer) and
cab (being a Cab)

```
record Cab:          id: int, driver: str;
record Customer:    id: int, name: str, title: str;
record Assign:      customer: Customer, cab: Cab;

guess from Customer exactly 1
  Assign from Cab
    where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
  count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

SDL Example

A Cab has attributes
id (an integer) and
driver (a string)

Assign has attributes
customer (being a Customer) and
cab (being a Cab)

```
record Cab:           id: int, driver: str;  
record Customer:     id: int, name: str, title: str;  
record Assign:       customer: Customer, cab: Cab;
```

```
guess from Customer exactly 1  
  Assign from Cab
```

Assign one cab to every customer

```
  where Assign.customer == Customer and Assign.cab == Cab;  
deny from Cab having  
  count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

```
ASP  
  
% cab(cab_id, driver)  
% customer(cust_id, name, title)  
% assign(cust_id, cab_id)  
  
% assign one cab to every customer  
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).  
  
% don't assign more than one customer to each cab  
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

SDL Example

```
% cab( cab_id, driver)
% customer(cust_id, name, title)
% assign(cust_id, cab_id)

% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

ASP

A Cab has attributes
id (an integer) and
driver (a string)

Assign has attributes
customer (being a Customer) and
cab (being a Cab)

```
record Cab:           id: int, driver: str;  
record Customer:    id: int, name: str, title: str;  
record Assign:      customer: Customer, cab: Cab;
```

```
guess from Customer exactly 1  
  Assign from Cab
```

Assign one cab to every customer

```
  where Assign.customer == Customer and Assign.cab == Cab;
```

```
deny from Cab having
```

```
  count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

Don't assign more than one customer to each cab

Just two little observations...

```
record Cab:      id: int, driver: str;  
record Customer: id: int, name: str, title: str;  
record Assign:   customer: Customer, cab: Cab;  
guess from Customer exactly 1  
  Assign from Cab  
    where Assign.customer == Customer and Assign.cab == Cab;  
deny from Cab having  
  count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

Just two little observations...

```
record Cab:      id: int, driver: str;  
record Customer: id: int, name: str, title: str;  
record Assign:   customer: Customer, cab: Cab;  
guess from Customer exactly 1 —————  
  Assign from Cab  
    where Assign.customer == Customer and Assign.cab == Cab;  
deny from Cab having  
  count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

Doesn't use the driver... and
has no idea about the driver!

Just two little observations...

```
record Cab:      id: int, driver: str;  
record Customer: id: int, name: str, title: str;  
record Assign:   customer: Customer, cab: Cab;  
guess from Customer exactly 1  
  Assign from Cab  
    where Assign.customer == Customer and Assign.cab == Cab;  
deny from Cab having  
  count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

Doesn't use the driver... and
has no idea about the driver!

Attributes accessed by name,
not by position!

```
record Cab:      id: int, driver: str;  
record Customer: id: int, name: str, title: str;  
record Assign:   customer: Customer, cab: Cab;  
guess from Customer exactly 1  
  Assign from Cab  
    where Assign.customer == Customer and Assign.cab == Cab;  
deny from Cab having  
  count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

```
% cab( cab_id, driver)  
% customer(cust_id, name, title)  
% assign(cust_id, cab_id)  
  
% assign one cab to every customer  
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).  
  
% don't assign more than one customer to each cab  
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```



```

record Cab:      id: int, driver: str;
record Customer: id: int, name: str, title: str;
record Assign:   customer: Customer, cab: Cab;

guess from Customer exactly 1
  Assign from Cab
    where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
  count {Assign.customer from Assign where Assign.cab == Cab} > 1;

```

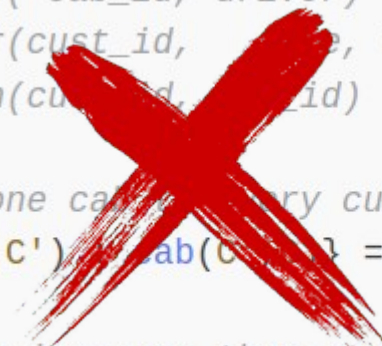
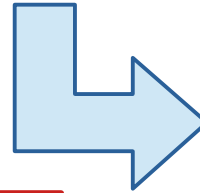
```

% cab( cab_id, driver)
% customer(cust_id, name, title)
% assign(customer_id, cab_id)

% assign one cab to every customer
{assign(C,C') & cab(C,D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.

```

```

{assign(
  customer(ID,NAME,TITLE),
  cab(ID,DRIVER)
) : cab(ID,DRIVER)} = 1 :-
  customer(ID,NAME,TITLE).

:- cab(ID,DRIVER), #count{
  customer(ID,NAME,TITLE) :
  assign(
    customer(ID,NAME,TITLE),
    cab(ID,DRIVER)
  )
} > 1.

```

Structure Instruction

```
record RecordName: Attributes;
```

List of **name: type** pairs

int, str or a record name

Structure Instruction

```
record RecordName: Attributes;
```

List of **name: type** pairs

int, str or a record name

Everything **MUST** be declared
(SDL targets long-standing codebases)

Structure Instruction

```
record RecordName: Attributes;
```

List of **name: type** pairs

int, str or a record name

Everything **MUST** be declared
(SDL targets long-standing codebases)

Acyclicity of structure instructions is
required and checked

Query Instruction

```
show RecordNames;
```

List of record names

Query Instruction

```
show RecordNames;
```

List of record names

We may include more expressive query instructions in the future, but for now... KISS!

Query Instruction

```
show RecordNames;
```

List of record names

We may include more expressive query instructions in the future, but for now... KISS!

By default, we only show SAT/UNSAT...

Query Instruction

```
show RecordNames;
```

List of record names

We may include more expressive query instructions in the future, but for now... KISS!

By default, we only show SAT/UNSAT...

to avoid any ambiguity

```
malvi@pandora:~ [ven giu 07 15:01]
$ echo "a(foo). b(bar). #show a/1. " | clingo --outf=1 -V0
ANSWER
a(foo).
malvi@pandora:~ [ven giu 07 15:01]
$ echo "a(foo). b(bar). #show a(X) : a(X)." | clingo --outf=1 -V0
ANSWER
a(foo). b(bar). a(foo).
```


Model Instructions (several)

Refer records by defining aliases (optional);
prepend **not** to refer the complement

RecordName **as** *alias*

Model Instructions (several)

Refer records by defining aliases (optional);
prepend **not** to refer the complement

RecordName **as** *alias*

Attributes are accessed with dot operations,
as in OOP languages

```
alias.attribute_name  
alias.attribute_name.sub_attribute
```

Model Instructions (several)

Refer records by defining aliases (optional);
prepend **not** to refer the complement

RecordName **as** *alias*

Attributes are accessed with dot operations,
as in OOP languages

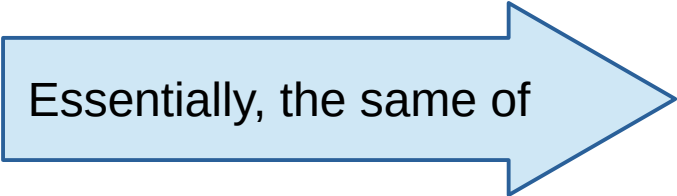
```
alias.attribute_name  
alias.attribute_name.sub_attribute
```

Values (of attributes, or constants)
can be combined in expressions

Model Instructions: Definitions

```
record Node: id: int;  
record Edge: first: Node, second: Node;
```

```
define Edge as self from Edge as other  
where self.first == other.second and self.second == other.first;
```

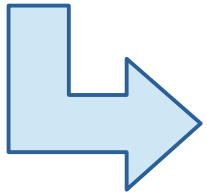


Essentially, the same of

Model Instructions: Definitions

```
record Node: id: int;  
record Edge: first: Node, second: Node;
```

```
define Edge as self from Edge as other  
where self.first == other.second and self.second == other.first;
```

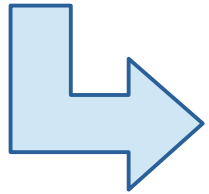


```
edge(node(SelfFirst), node(SelfSecond)) :-  
  edge(node(OtherFirst), node(OtherSecond)),  
  node(SelfFirst) == node(OtherSecond),  
  node(SelfSecond) == node(OtherFirst).
```

Model Instructions: Definitions

```
record Node: id: int;  
record Edge: first: Node, second: Node;
```

```
define Edge as self from Edge as other  
where self.first == other.second and self.second == other.first;
```



```
edge(node(SelfFirst), node(SelfSecond)) :-  
  edge(node(OtherFirst), node(OtherSecond)),  
  node(SelfFirst) == node(OtherSecond),  
  node(SelfSecond) == node(OtherFirst).
```

Essentially, the same of

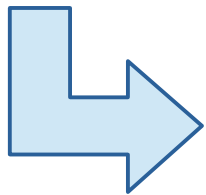
```
edge(OtherSecond, OtherFirst) :-  
  edge(OtherFirst, OtherSecond).
```

```
record Node: id: int;  
record Edge: first: Node, second: Node;  
record In: node: Node;  
record Size: value: int;
```

```
define Size having count {In.node from In} == Size.value;
```

```
record Node: id: int;  
record Edge: first: Node, second: Node;  
record In: node: Node;  
record Size: value: int;
```

```
define Size having count {In.node from In} == Size.value;
```



```
size(Value) :- #count{node(Id) : in(node(Id))} = Value.
```


Model Instructions: Guesses

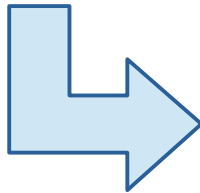
```
record Node: id: int;  
record Edge: first: Node, second: Node;  
record In: node: Node;  
record Size: value: int;
```

```
guess from Node at most 1  
  In where Node == In.node;
```

Model Instructions: Guesses

```
record Node: id: int;  
record Edge: first: Node, second: Node;  
record In: node: Node;  
record Size: value: int;
```

```
guess from Node at most 1  
In where Node == In.node;
```



```
0 <= {  
  in(node(Id')) :  
    node(Id) == node(Id')  
} <= 1 :- node(Id).
```

Model Instructions: Denies

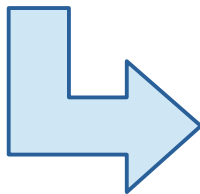
```
record Node: id: int;  
record Edge: first: Node, second: Node;  
record In: node: Node;  
record Size: value: int;
```

```
deny from In as in1, In as in2, not Edge  
where in1.node == Edge.first and  
        in2.node == Edge.second and  
        in1.node < in2.node;
```

Model Instructions: Denies

```
record Node: id: int;  
record Edge: first: Node, second: Node;  
record In: node: Node;  
record Size: value: int;
```

```
deny from In as in1, In as in2, not Edge  
where in1.node == Edge.first and  
      in2.node == Edge.second and  
      in1.node < in2.node;
```



```
:- in(node(Id)), in(node(Id')), not edge(node(F), node(S)),  
   node(Id) == node(F),  
   node(Id') == node(S),  
   node(Id) < node(Id').
```

Model Instructions: Denies with Penalty

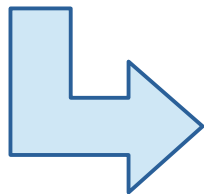
```
record Node: id: int;  
record Edge: first: Node, second: Node;  
record In: node: Node;  
record Size: value: int;
```

```
deny from Node, not In  
where In.node == Node  
or pay 1 at 1;
```

Model Instructions: Denies with Penalty

```
record Node: id: int;  
record Edge: first: Node, second: Node;  
record In: node: Node;  
record Size: value: int;
```

```
deny from Node, not In  
where In.node == Node  
or pay 1 at 1;
```



```
:- node(Id), not in(node(Id')),  
node(Id') == node(Id).  
[1@1, Id, Id']
```

And if you need “assembly”...

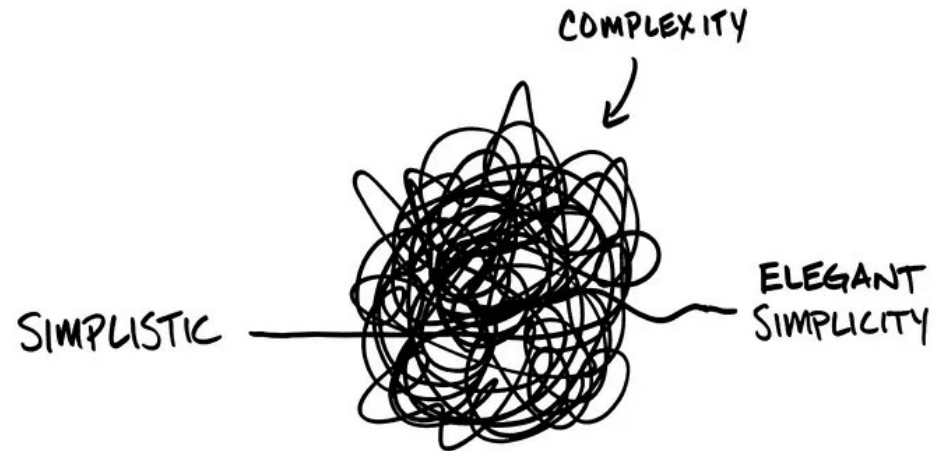
```
@asp_block $  
  to_be | not_to_be :-  
    plain, hold, asp, code.  
  
  possible.  
  :- suggested.  
$
```

We don't do any syntactic or semantic check in

@asp_block \$... \$

Summing UP

ASP syntax is not suitable for long-standing codebases
(it doesn't even look like a programming language)



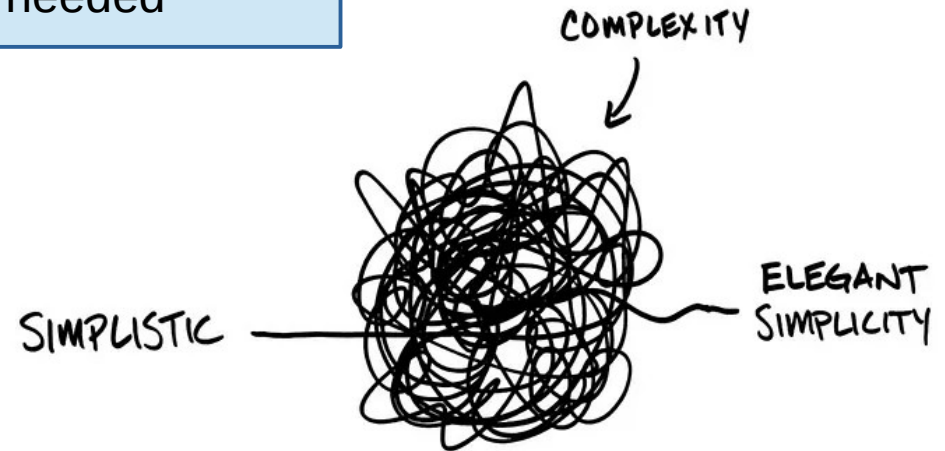
BEHAVIOR GAP

Image Credits to Hill Investment Group

Summing UP

ASP syntax is not suitable for long-standing codebases
(it doesn't even look like a programming language)

SDL is a programming language of higher level,
enforcing a proper structure of records, and
introducing dependencies only if explicitly needed



BEHAVIOR GAP

Image Credits to Hill Investment Group

Summing UP

ASP syntax is not suitable for long-standing codebases
(it doesn't even look like a programming language)

SDL is a programming language of higher level,
enforcing a proper structure of records, and
introducing dependencies only if explicitly needed

On the long run, programmers may code
in SDL without thinking at all to ASP
(as SQL programmers never rarely think
to relational algebra)



BEHAVIOR GAP

Image Credits to Hill Investment Group

Questions

