Imperial College London

Representing and Learning Grammars in Answer Set Programming

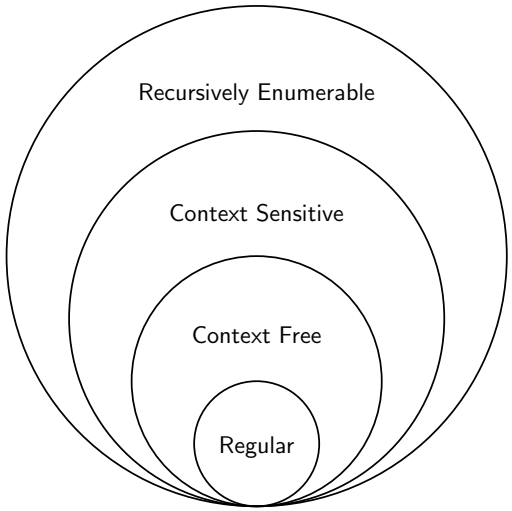
Mark Law[1], Alessandra Russo[1], Elisa Bertino[2], Krysia Broda[1] and Jorge Lobo[3]

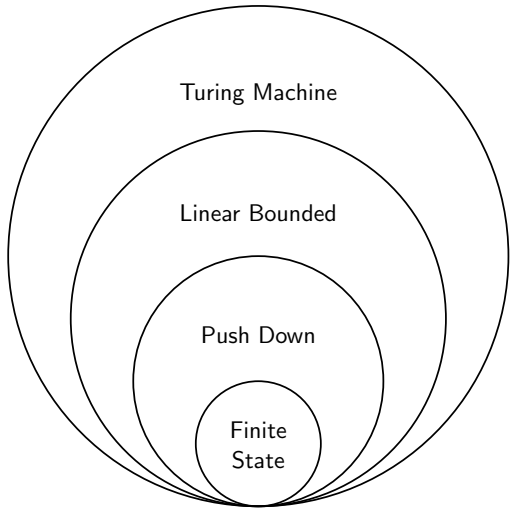
[1]Imperial College London

[2]Purdue University

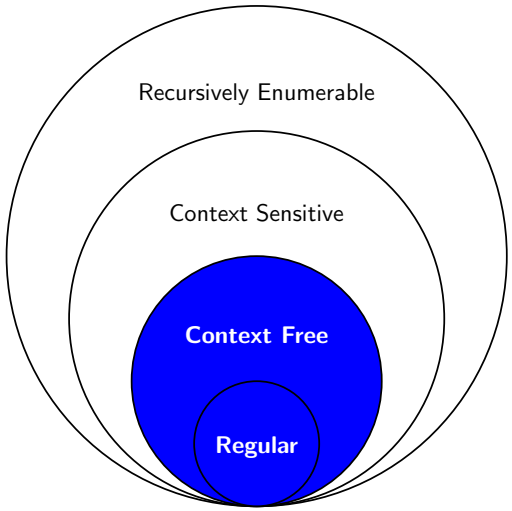
[3]ICREA – Universitat Pompeo Fabra

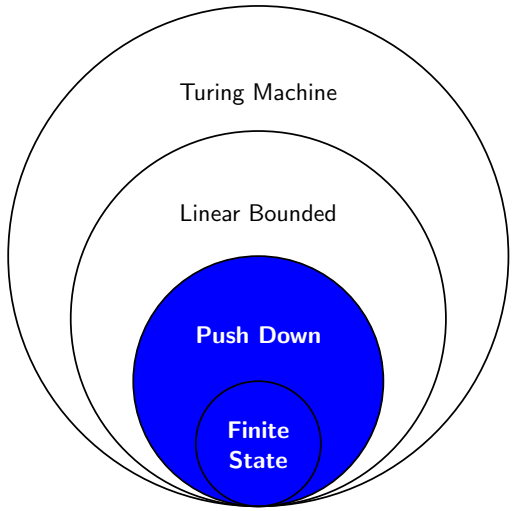
Induction of Grammars and Automata





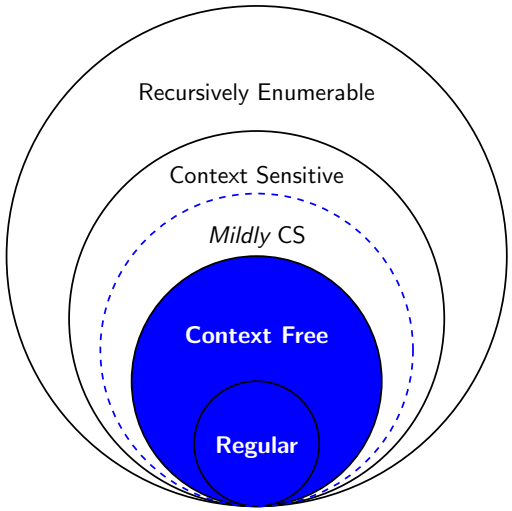
Induction of Grammars and Automata

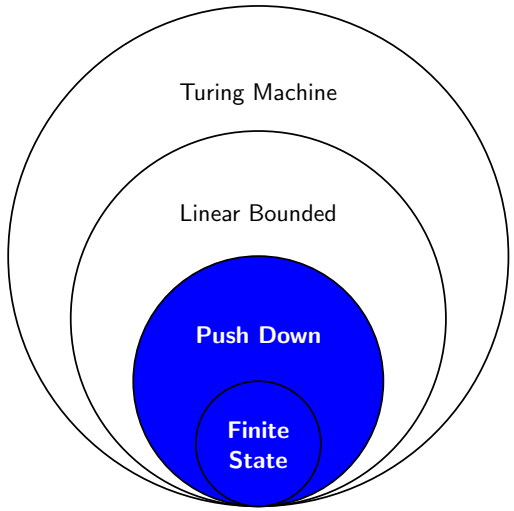




- Previous work on learning grammars has mostly been restricted to learning context-free grammars.

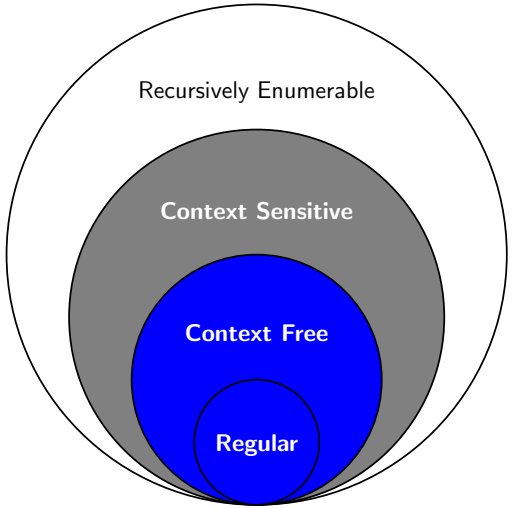
Induction of Grammars and Automata

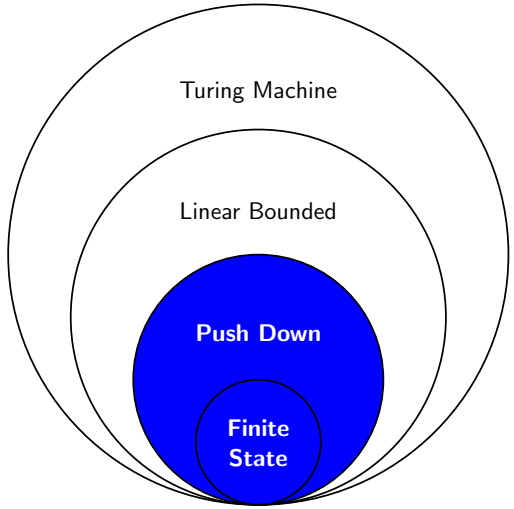




- Previous work on learning grammars has mostly been restricted to learning context-free grammars.
- Some work has considered learning *mildly* context sensitive languages.

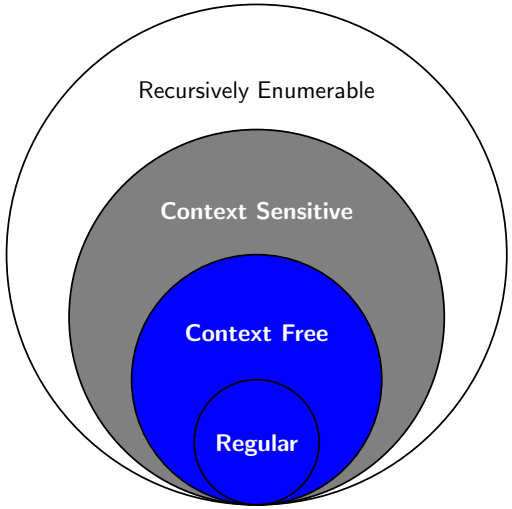
Induction of Grammars and Automata

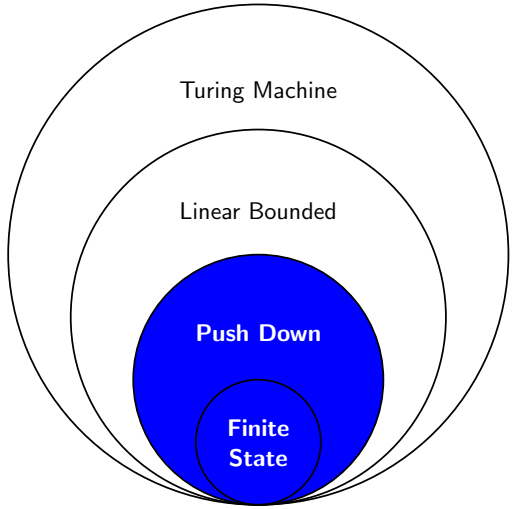




- We propose a new class of context sensitive grammars called Answer Set Grammars (ASGs), which extend CFGs with context sensitive conditions written in ASP.
- ASP conditions can be learned using an existing ASP learner.

Induction of Grammars and Automata

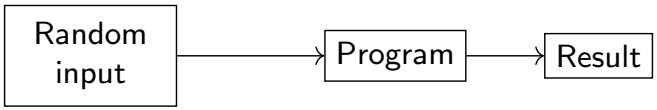




- Unlike other approaches, our learning approach takes an initial grammar as input.

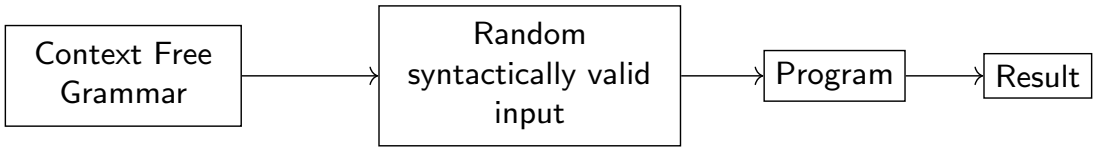
Relevant Applications of ASG Induction

- *Fuzzing* is the process of randomly generating test inputs to programs:



Relevant Applications of ASG Induction

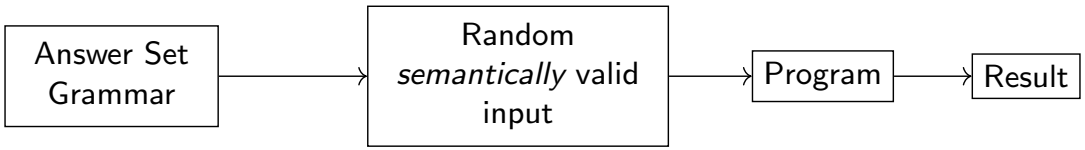
- *Fuzzing* is the process of randomly generating test inputs to programs:



- Previous approaches have used grammar induction to learn to randomly generate *syntactically* valid input.

Relevant Applications of ASG Induction

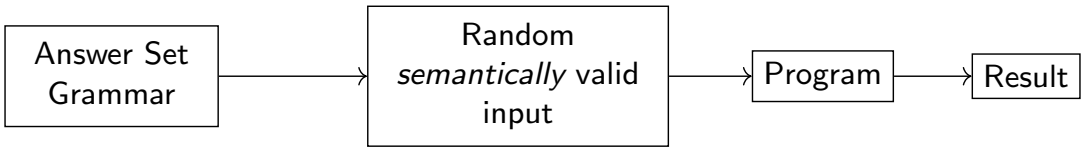
- *Fuzzing* is the process of randomly generating test inputs to programs:



- Given the CFG for the program input, our approach can be used to learn to generate *semantically* valid input.

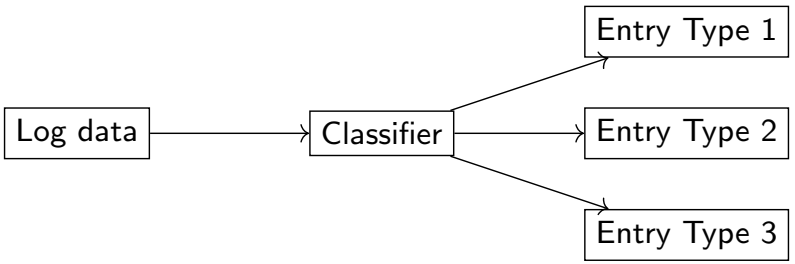
Relevant Applications of ASG Induction

- *Fuzzing* is the process of randomly generating test inputs to programs:



- Given the CFG for the program input, our approach can be used to learn to generate *semantically* valid input.

- Automatic classification of logs:



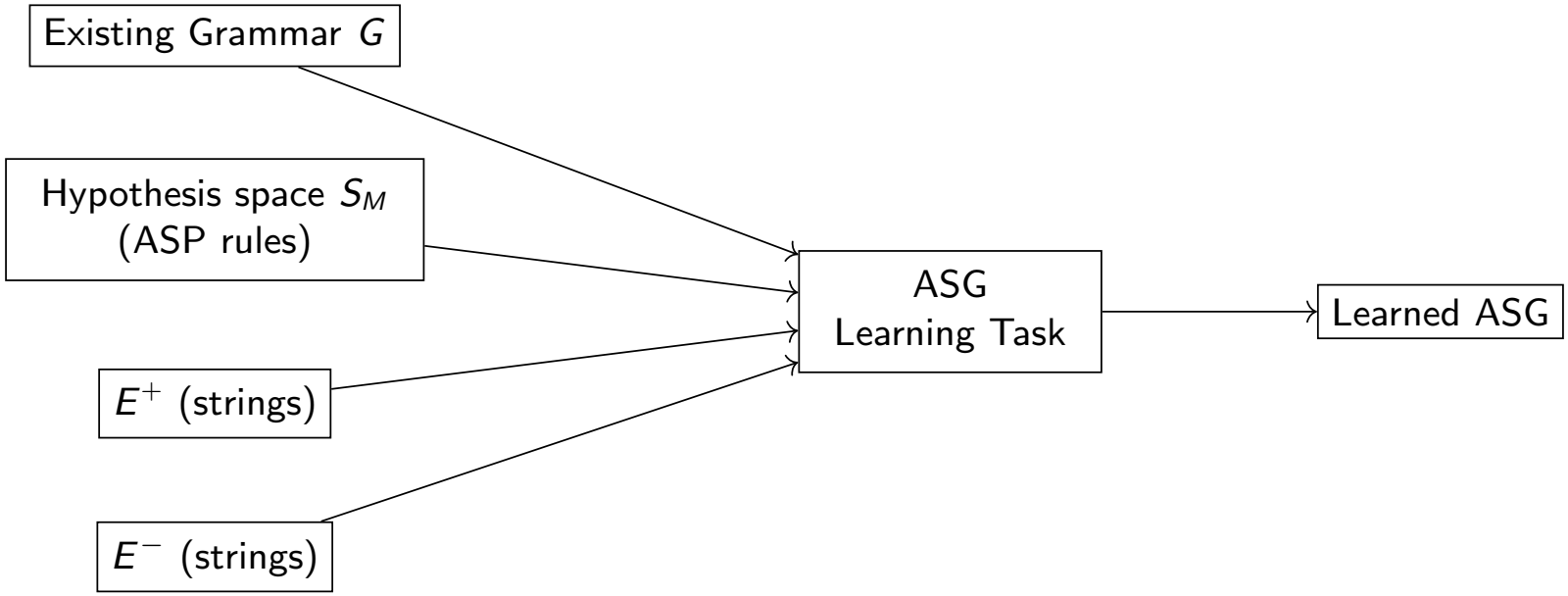
Answer Set Grammars (ASGs)

- An *Answer Set Grammar* is an augmented CFG, where each production rule may be *annotated* with ASP.

```
1: start -> as bs cs {
     :- size(X)@1, not size(X)@2.
     :- size(X)@1, not size(X)@3.
   }
2: as -> "a" as { size(X+1) :- size(X)@2. }
3: as ->        { size(0). }
4: bs -> "b" bs { size(X+1) :- size(X)@2. }
5: bs ->        { size(0). }
6: cs -> "c" cs { size(X+1) :- size(X)@2. }
7: cs ->        { size(0). }
```

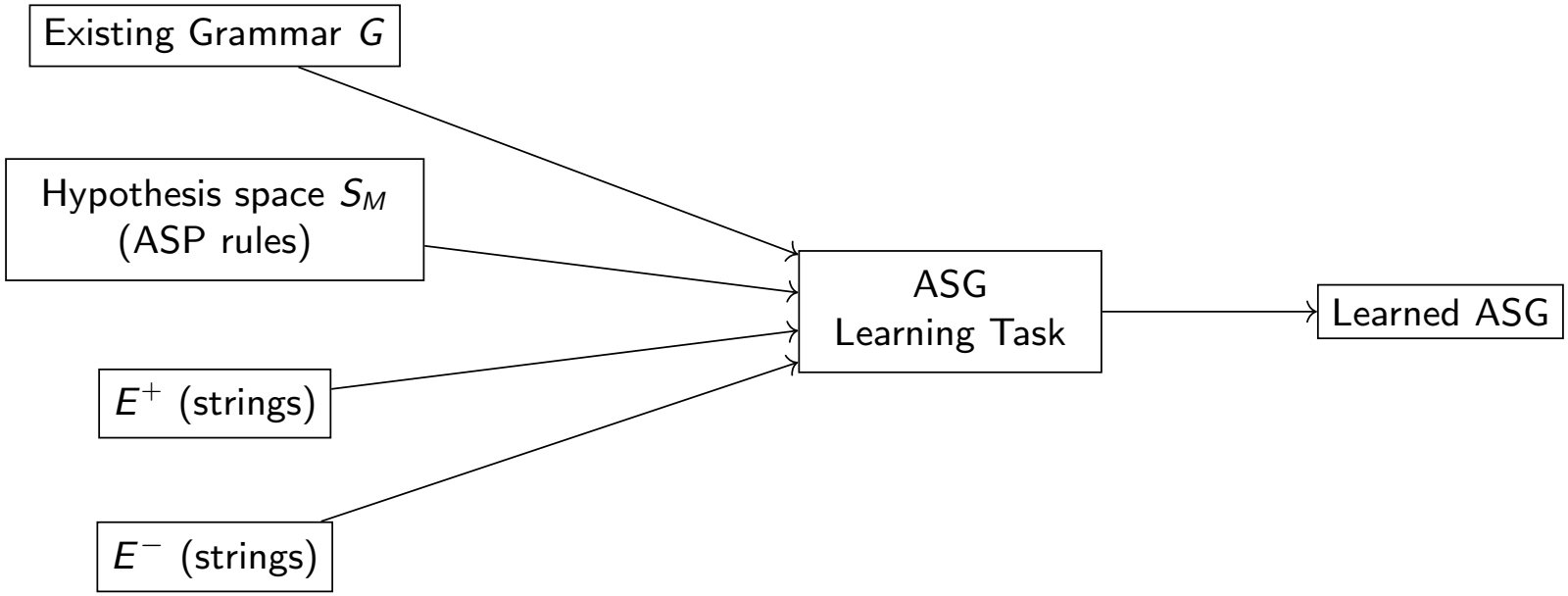
- This grammar represents the context-sensitive language $a^n b^n c^n$.

Learning Answer Set Grammars



- The goal is to find an $H \subseteq S_M$ st when H is added to the annotations of G, the extended ASG accepts every string in E^+ and no string in E^-.

Learning Answer Set Grammars



- The goal is to find an $H \subseteq S_M$ st when H is added to the annotations of G, the extended ASG accepts every string in E^+ and no string in E^-.
- In this work all parse trees are assumed to be bounded by a given maximum depth d.

Algorithm

1: **procedure** LEARNASG(T)
2: $T_{LAS} = LAS(T, d)$;
3: $H = ILASP(T_{LAS})$
4: **return** H^{ASG};
5: **end procedure**

- $LAS(T, d)$ translates an ASG learning task T and depth d to a learning task for an existing ASP learner (ILASP).
- H^{ASG} is the translation of the ILASP solution H to an ASG solution.

Algorithm

1: **procedure** LearnASG(T)
2: $T_{LAS} = LAS(T, d)$;
3: $H = ILASP(T_{LAS})$
4: **return** H^{ASG};
5: **end procedure**

- $LAS(T, d)$ translates an ASG learning task T and depth d to a learning task for an existing ASP learner (ILASP).
- H^{ASG} is the translation of the ILASP solution H to an ASG solution.

LearnASG is sound and complete.

Complexity – decision problems

- **Bounded-ASG-membership**: deciding whether an ASG accepts a string.
- **Bounded-ASG-satisfiability**: deciding whether an ASG has a non-empty language.

Complexity – decision problems

- **Bounded-ASG-membership**: deciding whether an ASG accepts a string.
- **Bounded-ASG-satisfiability**: deciding whether an ASG has a non-empty language.

Each decision problem is investigated for ASGs with various restrictions on the language of the ASP annotations:

- *Propositional* and *(function-free) first-order* ASGs
- *Horn*, *stratified* and *unstratified* ASGs

Complexity of bounded ASG membership/satisfiability

	Horn	Stratified	Unstratified
Propositional	NP	NP	NP
First-order	EXP	EXP	NEXP

Complexity of bounded ASG membership/satisfiability

	Horn	Stratified	Unstratified
Propositional	NP	NP	NP
First-order	EXP	EXP	NEXP

- "Even loops" can be simulated by duplicate production rules:

```
s -> b {
   p:-not q.
   q:-not p.
}
```

```
s -> b {p.}
s -> b {q.}
```


Complexity of learning – decision problems

- **Bounded-verification**: deciding whether a given hypothesis is a solution of a given learning task.

- **Bounded-satisfiability**: deciding whether a given learning task has any solutions.

Complexity of learning results

	Horn	Stratified	Unstratified
Bounded-verification	DP	DP	DP
Bounded-satisfiability	Σ_2^P	Σ_2^P	Σ_2^P

Complexity of learning results

- If each string has a polynomial number of parse trees:

	Horn	Stratified	Unstratified
Bounded-verification	NP	NP	DP
Bounded-satisfiability	NP	NP	Σ_2^P

Complexity of learning results

- If each string has a polynomial number of parse trees:

	Horn	Stratified	Unstratified
Bounded-verification	NP	NP	DP
Bounded-satisfiability	NP	NP	Σ_2^P

- In the paper, we give a more efficient translation to an ILASP task for this case.

Evaluation

- We evaluated LearnASG on $a^n b^n c^n$, starting from various initial grammars: $a^n b^n c^m$, $a^i b^j c^k$ and $(a|b|c)^*$.
- The more information given as input, the easier the ASG is to learn.

Evaluation

- We evaluated LearnASG on $a^n b^n c^n$, starting from various initial grammars: $a^n b^n c^m$, $a^i b^j c^k$ and $(a|b|c)^*$.
- The more information given as input, the easier the ASG is to learn.
- (Nakamura and Imada 2011) learn $a^n b^n c^n$ from scratch.
  - Faster than LearnASG on the equivalent problem – from $(a|b|c)^*$
  - Slower than LearnASG from $a^i b^j c^k$
  - Cannot take advantage of an existing CFG
  - Their method can only learn MCS languages (whose membership problem must be in P).

Conclusion

- ASGs are a new context-sensitive grammars, which extend CFGs with ASP annotations.
- Presented a method for the ASP annotations.
  - Best suited for tasks where the underlying syntax of a language is known, but (some of) the semantic conditions are unknown.
- Future research directions include:
  - Investigating the relevant applications.
  - Exploring using other formalisms such as CSPs and SMTs in annotations.

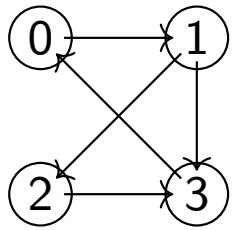
Backup

Going beyond Mildly CS grammars

- Membership of MCS languages is in P.

Going beyond Mildly CS grammars

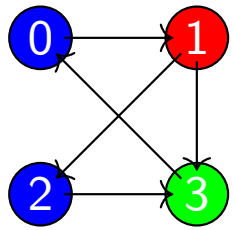
- Membership of MCS languages is in P.
- The graph colouring language consists of strings representing graphs that are 3-colourable.



"0 1 2 3 (0, 1) (1, 2) (1, 3) (2, 3) (3, 0)"

Going beyond Mildly CS grammars

- Membership of MCS languages is in P.
- The graph colouring language consists of strings representing graphs that are 3-colourable.



"0 1 2 3 (0, 1) (1, 2) (1, 3) (2, 3) (3, 0)"

	$\lvert E^+ \rvert$	$\lvert E^- \rvert$	Final Time	Total Time
Stratified	2	2	11.5s	23.8s
Unstratified	4	4	90.1s	256.9s

Evaluation: $a^n b^n c^n$

Initial Language	$a^n b^n c^m$ (ASG st CF part is $a^i b^j c^k$)	$a^n b^n c^m$ (CFG)	$a^i b^j c^k$ (CFG)	$(a\|b\|c)*$ (CFG)
Target Constraint	$n = m$	$n = m$	$i = j = k$	All a's before all b's before all c's. Same number of a's, b's and c's
$\|E^+\|/\|E^-\|$	1 / 2	1 / 3	1 / 7	1 / 45
Final/Total Time	0.5s / 1.4s	0.3s / 1.3s	1.1s / 5.1s	1004.0s / 13314.9s

- The target language for each of these tasks is the same ($a^n b^n c^n$), but the information encoded in the initial language decreases from left to right in the table.

Evaluation: $a^n b^n c^n$

Initial Language	$a^n b^n c^m$ (ASG st CF part is $a^i b^j c^k$)	$a^n b^n c^m$ (CFG)	$a^i b^j c^k$ (CFG)	$(a\|b\|c)*$ (CFG)
Target Constraint	$n = m$	$n = m$	$i = j = k$	All a's before all b's before all c's. Same number of a's, b's and c's
$\|E^+\|/\|E^-\|$	1 / 2	1 / 3	1 / 7	1 / 45
Final/Total Time	0.5s / 1.4s	0.3s / 1.3s	1.1s / 5.1s	1004.0s / 13314.9s

- More examples are needed as the information encoded in the initial language decreases.

Evaluation: $a^n b^n c^n$

Initial Language	$a^n b^n c^m$ (ASG st CF part is $a^i b^j c^k$)	$a^n b^n c^m$ (CFG)	$a^i b^j c^k$ (CFG)	(a|b|c)* (CFG)
Target Constraint	$n = m$	$n = m$	$i = j = k$	All a's before all b's before all c's. Same number of a's, b's and c's
$\|E^+\|/\|E^-\|$	1 / 2	1 / 3	1 / 7	1 / 45
Final/Total Time	0.5s / 1.4s	0.3s / 1.3s	1.1s / 5.1s	1004.0s / 13314.9s

- More examples are needed as the information encoded in the initial language decreases.
- Although the final experiment shows that it is possible to learn the whole language from scratch, the method performs better when the CFG is given as input.

Evaluation: $a^n b^n c^n$

Initial Language	$a^n b^n c^m$ (ASG st CF part is $a^i b^j c^k$)	$a^n b^n c^m$ (CFG)	$a^i b^j c^k$ (CFG)	$(a\|b\|c)*$ (CFG)
Target Constraint	$n = m$	$n = m$	$i = j = k$	All a's before all b's before all c's. Same number of a's, b's and c's
$\|E^+\|/\|E^-\|$	1 / 2	1 / 3	1 / 7	1 / 45
Final/Total Time	0.5s / 1.4s	0.3s / 1.3s	1.1s / 5.1s	1004.0s / 13314.9s

- (Nakamura and Imada 2011) presented a method that can learn $a^n b^n c^n$ from scratch, which takes 18s.

Evaluation: $a^n b^n c^n$

Initial Language	$a^n b^n c^m$ (ASG st CF part is $a^i b^j c^k$)	$a^n b^n c^m$ (CFG)	$a^i b^j c^k$ (CFG)	$(a\|b\|c)*$ (CFG)
Target Constraint	$n = m$	$n = m$	$i = j = k$	All a's before all b's before all c's. Same number of a's, b's and c's
$\|E^+\|/\|E^-\|$	1 / 2	1 / 3	1 / 7	1 / 45
Final/Total Time	0.5s / 1.4s	0.3s / 1.3s	1.1s / 5.1s	1004.0s / 13314.9s

- (Nakamura and Imada 2011) presented a method that can learn $a^n b^n c^n$ from scratch, which takes 18s.
  - Their method is faster at learning from scratch, but cannot take an initial grammar.
  - Their method can only learn MCS grammars.

Answer Set Grammars

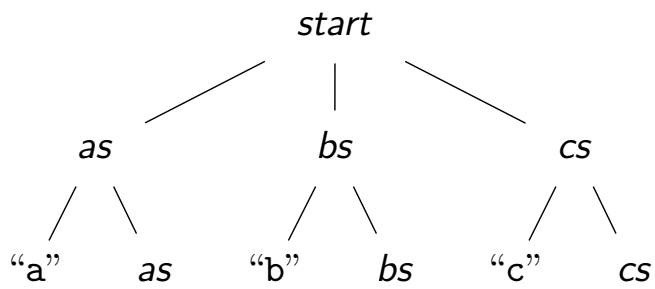
- An *Answer Set Grammar* (ASG) is an augmented CFG, where each production rule may be *annotated* with ASP constraints.
- A string s is a member of an ASG G, if there is at least one parse tree of G for s st the program $G[PT]$ is satisfiable.

```
1: start -> as bs cs {
     :- size(X)@1, not size(X)@2.
     :- size(X)@1, not size(X)@3.
   }
2: as -> "a" as { size(X+1) :- size(X)@2. }
3: as ->        { size(0). }
4: bs -> "b" bs { size(X+1) :- size(X)@2. }
5: bs ->        { size(0). }
6: cs -> "c" cs { size(X+1) :- size(X)@2. }
7: cs ->        { size(0). }
```


Answer Set Grammars

- An *Answer Set Grammar* (ASG) is an augmented CFG, where each production rule may be *annotated* with ASP constraints.
- A string s is a member of an ASG G, if there is at least one parse tree of G for s st the program $G[PT]$ is satisfiable.

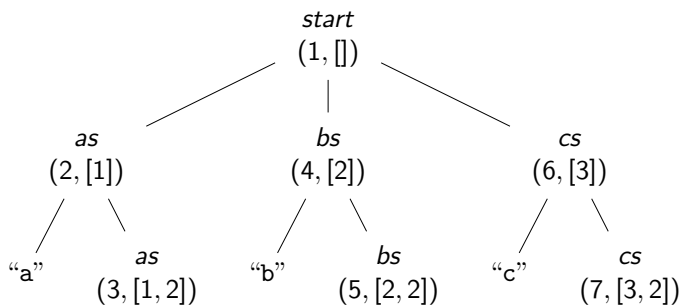
```
1: start -> as bs cs {
     :- size(X)@1, not size(X)@2.
     :- size(X)@1, not size(X)@3.
   }
2: as -> "a" as { size(X+1) :- size(X)@2. }
3: as ->        { size(0). }
4: bs -> "b" bs { size(X+1) :- size(X)@2. }
5: bs ->        { size(0). }
6: cs -> "c" cs { size(X+1) :- size(X)@2. }
7: cs ->        { size(0). }
```



Answer Set Grammars

- An *Answer Set Grammar* (ASG) is an augmented CFG, where each production rule may be *annotated* with ASP constraints.
- A string s is a member of an ASG G, if there is at least one parse tree of G for s st the program $G[PT]$ is satisfiable.

```
1: start -> as bs cs {
     :- size(X)@1, not size(X)@2.
     :- size(X)@1, not size(X)@3.
   }
2: as -> "a" as { size(X+1) :- size(X)@2. }
3: as ->        { size(0). }
4: bs -> "b" bs { size(X+1) :- size(X)@2. }
5: bs ->        { size(0). }
6: cs -> "c" cs { size(X+1) :- size(X)@2. }
7: cs ->        { size(0). }
```

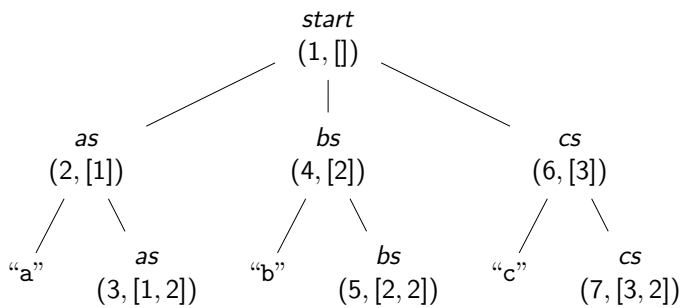


Answer Set Grammars

- An *Answer Set Grammar* (ASG) is an augmented CFG, where each production rule may be *annotated* with ASP constraints.
- A string s is a member of an ASG G, if there is at least one parse tree of G for s st the program $G[PT]$ is satisfiable.

G[PT]:

```
:- size(X)@[1], not size(X)@[2].
:- size(X)@[1], not size(X)@[3].
size(X+1)@[1] :- size(X)@[1, 2].
size(0)@[1, 2].
size(X+1)@[2] :- size(X)@[2, 2].
size(0)@[2, 2].
size(X+1)@[3] :- size(X)@[3, 2].
size(0)@[3, 2].
```

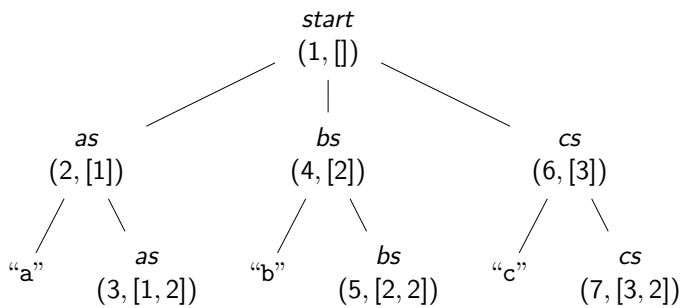


Answer Set Grammars

- An *Answer Set Grammar* (ASG) is an augmented CFG, where each production rule may be *annotated* with ASP constraints.
- A string s is a member of an ASG G, if there is at least one parse tree of G for s st the program $G[PT]$ is satisfiable.

G[PT]:

```
:- size(X)@[1], not size(X)@[2].
:- size(X)@[1], not size(X)@[3].
size(X+1)@[1] :- size(X)@[1, 2].
size(0)@[1, 2].
size(X+1)@[2] :- size(X)@[2, 2].
size(0)@[2, 2].
size(X+1)@[3] :- size(X)@[3, 2].
size(0)@[3, 2].
```

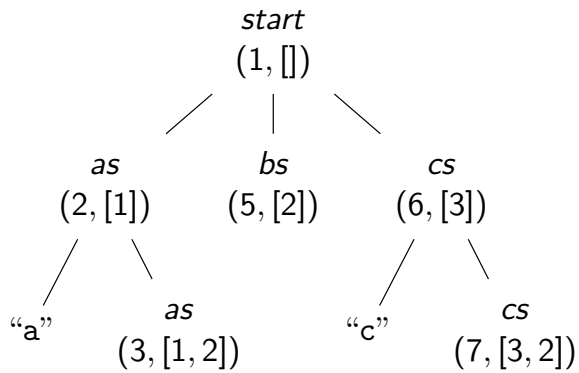


$G[PT]$ is satisfiable, hence "abc" $\in \mathcal{L}(G)$.

Answer Set Grammars

- An *Answer Set Grammar* (ASG) is an augmented CFG, where each production rule may be *annotated* with ASP constraints.
- A string s is a member of an ASG G, if there is at least one parse tree of G for s st the program $G[PT]$ is satisfiable.

```
1: start -> as bs cs {
     :- size(X)@1, not size(X)@2.
     :- size(X)@1, not size(X)@3.
   }
2: as -> "a" as { size(X+1) :- size(X)@2. }
3: as ->        { size(0). }
4: bs -> "b" bs { size(X+1) :- size(X)@2. }
5: bs ->        { size(0). }
6: cs -> "c" cs { size(X+1) :- size(X)@2. }
7: cs ->        { size(0). }
```

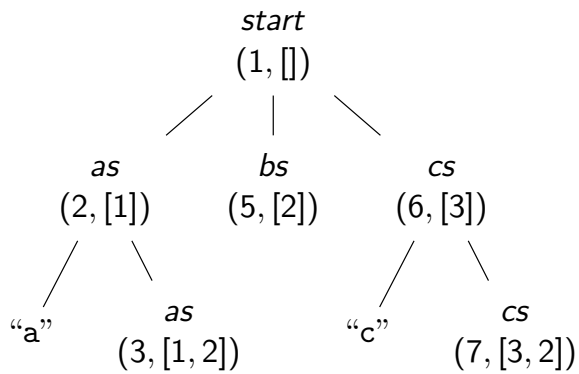


Answer Set Grammars

- An *Answer Set Grammar* (ASG) is an augmented CFG, where each production rule may be *annotated* with ASP constraints.
- A string s is a member of an ASG G, if there is at least one parse tree of G for s st the program $G[PT]$ is satisfiable.

G[PT]:

```
:- size(X)@[1], not size(X)@[2].
:- size(X)@[1], not size(X)@[3].
size(X+1)@[1] :- size(X)@[1, 2].
size(0)@[1, 2].
size(0)@[2].
size(X+1)@[3] :- size(X)@[3, 2].
size(0)@[3, 2].
```

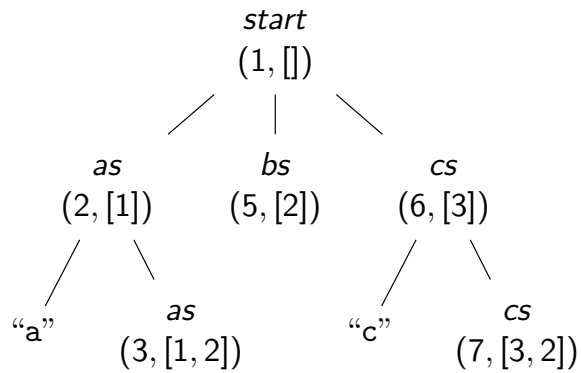


Answer Set Grammars

- An *Answer Set Grammar* (ASG) is an augmented CFG, where each production rule may be *annotated* with ASP constraints.
- A string s is a member of an ASG G, if there is at least one parse tree of G for s st the program $G[PT]$ is satisfiable.

```
G[PT]:

    :- size(X)@[1], not size(X)@[2].
    :- size(X)@[1], not size(X)@[3].
    size(X+1)@[1] :- size(X)@[1, 2].
    size(0)@[1, 2].
    size(0)@[2].
    size(X+1)@[3] :- size(X)@[3, 2].
    size(0)@[3, 2].
```



$G[PT]$ is unsatisfiable, hence "ac" $\in \mathcal{L}(G)$.

Answer Set Grammars

- An *Answer Set Grammar* (ASG) is an augmented CFG, where each production rule may be *annotated* with ASP constraints.
- A string s is a member of an ASG G, if there is at least one parse tree of G for s st the program $G[PT]$ is satisfiable.

```
1: start -> as bs cs {
     :- size(X)@1, not size(X)@2.
     :- size(X)@1, not size(X)@3.
   }
2: as -> "a" as { size(X+1) :- size(X)@2. }
3: as ->        { size(0). }
4: bs -> "b" bs { size(X+1) :- size(X)@2. }
5: bs ->        { size(0). }
6: cs -> "c" cs { size(X+1) :- size(X)@2. }
7: cs ->        { size(0). }
```

G represents $a^n b^n c^n$, which is a *Context Sensitive Grammar* (CSG).

Nakamura, K. and Imada, K. 2011.

Towards incremental learning of mildly context-sensitive grammars.
In *Machine Learning and Applications and Workshops (ICMLA), 2011 10th International Conference on*.
Vol. 1. IEEE, 223–228.