# Bachelor’s Thesis

Aarrya Saraf
Supervised by:
Dr. Noam Zeilberger, LIX
Dr. Dale Miller, LIX

# Top-Down and Bottom-Up Approaches to Parsing

# Motivation

x * 7 + 61 * 7

# Motivation

x * 7 + 61 * 7

1. Tokenizing - [Num x, Sym *, Num 7, Sym +, Num 61, Sym *, Num 7]

# Motivation

x * 7 + 61 * 7

1. Tokenizing - [Num x, Sym *, Num 7, Sym +, Num 61, Sym *, Num 7]
2. Parsing - Makes sure we follow the given grammar

# Motivation

x * 7 + 61 * 7

1. Tokenizing - [Num x, Sym *, Num 7, Sym +, Num 61, Sym *, Num 7]
2. Parsing - Makes sure we follow the given grammar

The grammar:

Expr -> Lb Expr Rb | Num | Expr Sym Expr

Num -> Minus Dig Dig* | Dig Dig* | x

Sym -> + | - | / | *

Minus -> -

Rb -> )

Lb -> (

Dig -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

# Motivation

x * 7 + 61 * 7

1. Tokenizing - [Num x, Sym *, Num 7, Sym +, Num 61, Sym *, Num 7]
2. Parsing - Makes sure we follow the given grammar
3. Adjustments and finally evaluation

# Motivation

x * 7 + 61 * 7

1. Tokenizing - [Num x, Sym *, Num 7, Sym +, Num 61, Sym *, Num 7]
2. Parsing - Makes sure we follow the given grammar
3. Adjustments and finally evaluation

(x*7) + (61*7) is 3 operations

However

7*(61+x) is just 2

# Motivation

x * 7 + 61 * 7

1. Tokenizing - [Num x, Sym *, Num 7, Sym +, Num 61, Sym *, Num 7]
2. Parsing - Makes sure we follow the given grammar
3. Adjustments and finally evaluation

(x*7) + (61*7) is 3 operations

However

7*(61+x) is just 2

Lastly say x = 9 and we will have the answer as 490

# Motivation

x * 7 + 61 * 7

1. Tokenizing - [Num x, Sym *, Num 7, Sym +, Num 61, Sym *, Num 7]
2. Parsing - Makes sure we follow the given grammar
3. Adjustments and finally evaluation

# A Good Parser

# A Good Parser

1. Sound

# A Good Parser

1. Sound
2. Complete

# A Good Parser

1. Sound
2. Complete
3. Guaranteed to terminate

# A Good Parser

1. Sound
2. Complete
3. Guaranteed to terminate
4. Fast

# A Good Parser

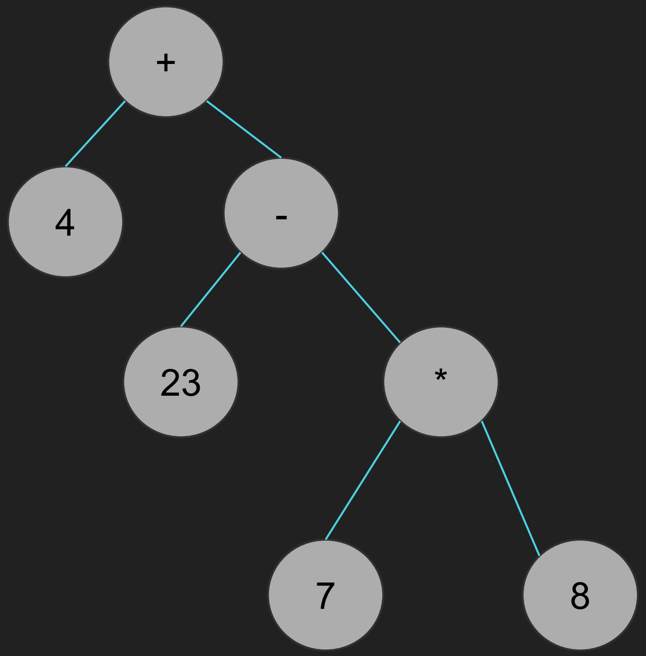
1. Sound
2. Complete
3. Guaranteed to terminate
4. Fast

# Parse Trees

Consider: 4+23-(7*8)

# Parse Trees

Consider: 4+23-(7*8)



# Parse Trees

- Prefer deterministic grammars

# Parse Trees

- Prefer deterministic grammars

- Same parse tree

# Parse Trees

- Prefer deterministic grammars
- Same parse tree
- Super important and precise - hence automated

# A Good Parser

1. Sound
2. Complete
3. Guaranteed to terminate
4. Fast

# Our POV

Start with (●S, N)

Length left to parse

What is left to parse

Current position

# Our POV

Start with (•S, N)

Want (•e, 0)

# Our POV

Start with (●S, N)

Want (●e, 0)

Rule A -> BC

Then

$$\frac{(B, i, k), (C, k, j)}{(A,i,j)}$$

# Our POV

Start with (●S, N)

Want (●e, 0)

Rule A -> a

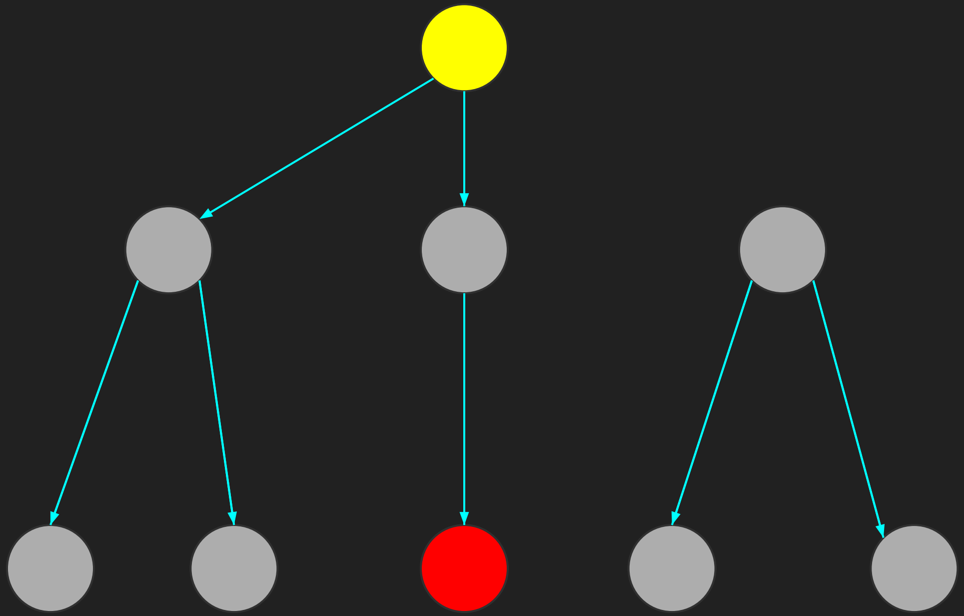
Then

ABCD… (●abcd…, N)

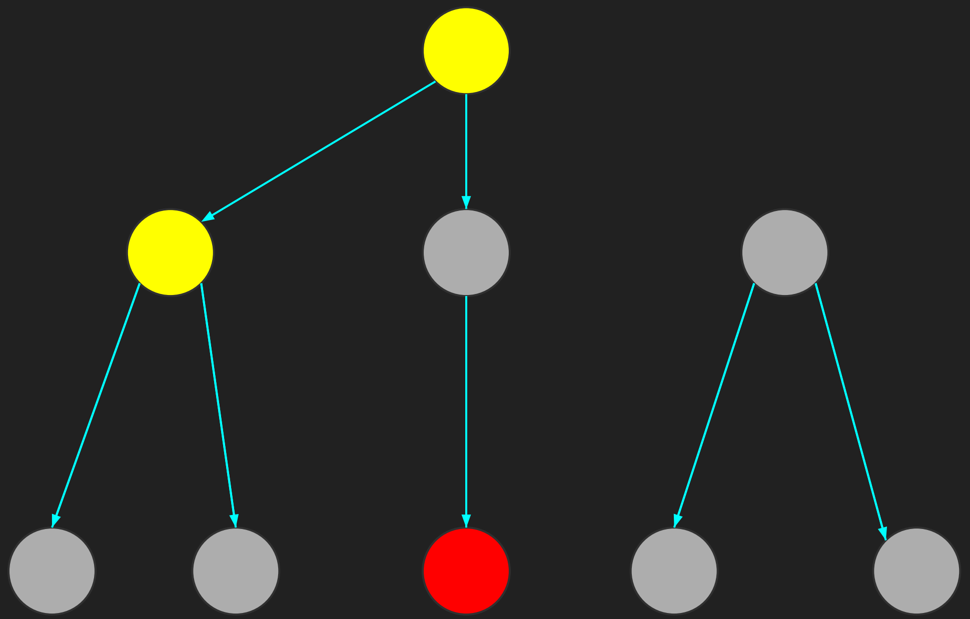
—-------------------------------------

BCD… (a●bcd , N-1)

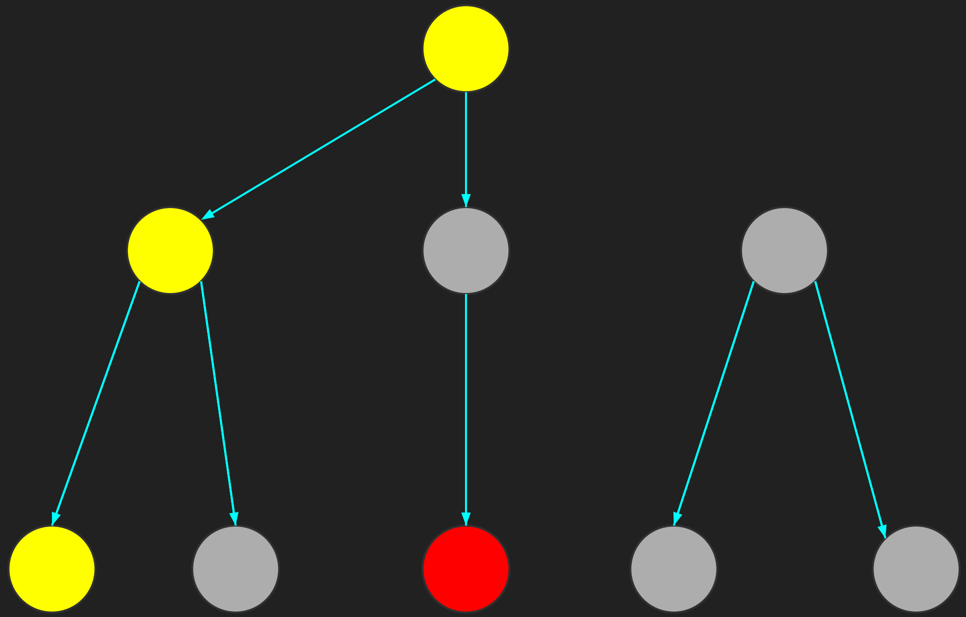
# Top-Down



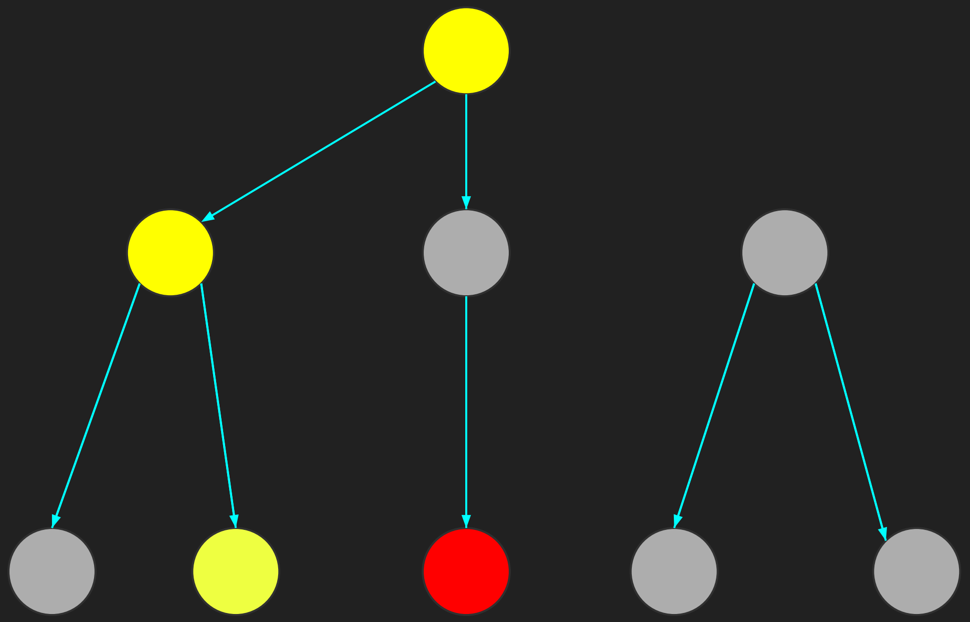
# Top-Down



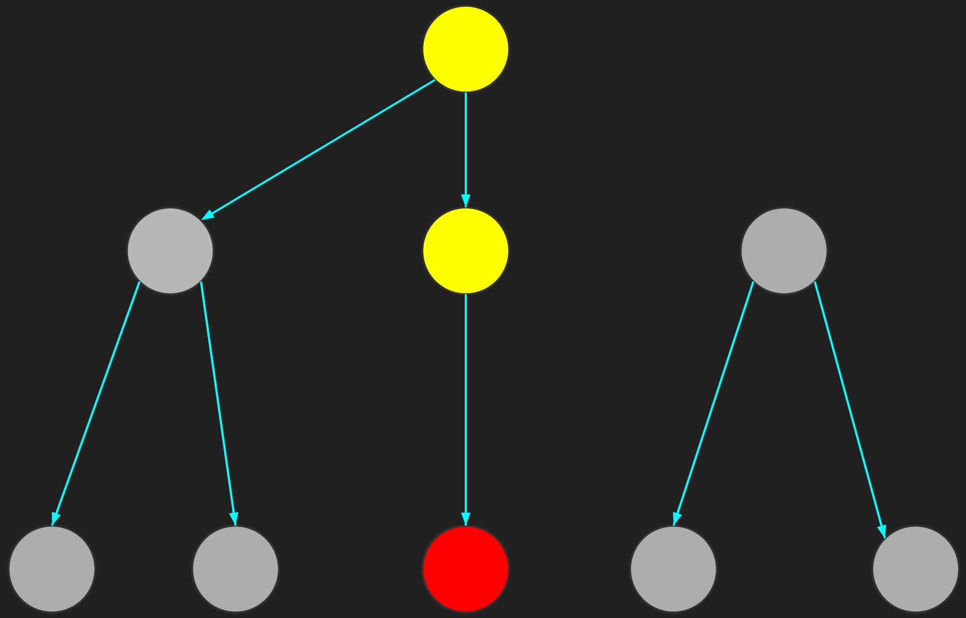
# Top-Down



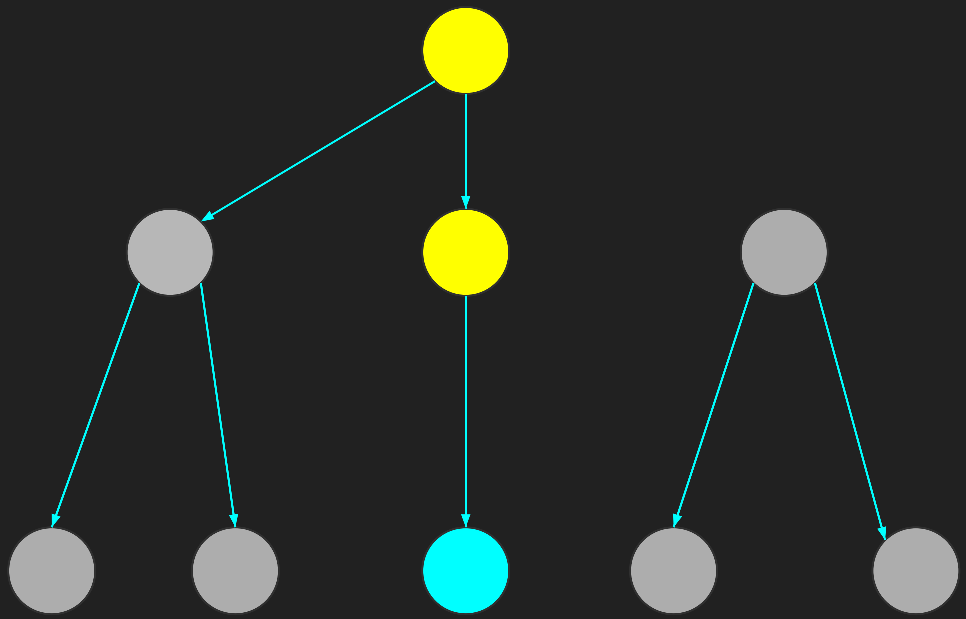
# Top-Down



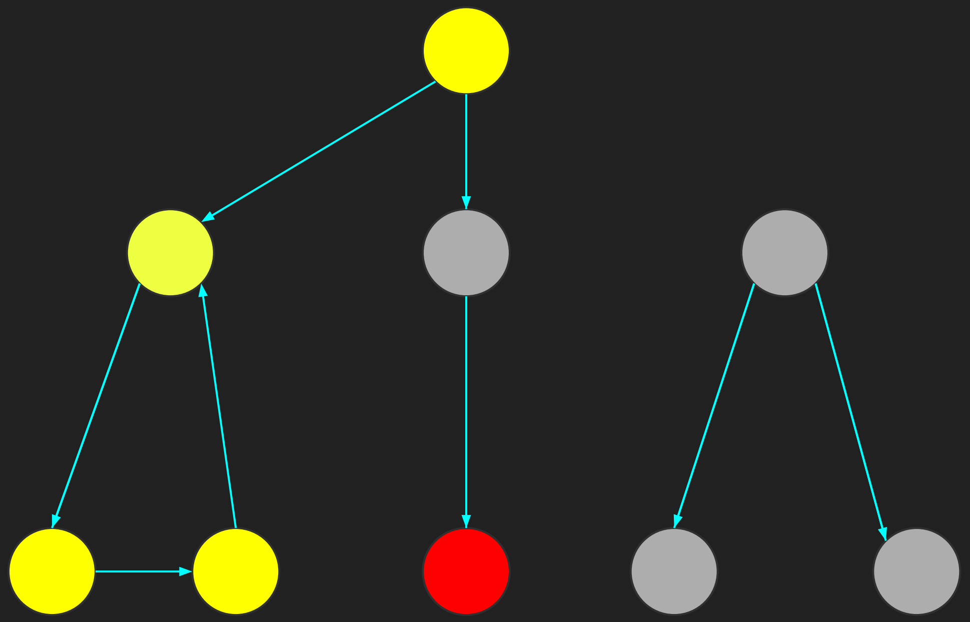
# Top-Down



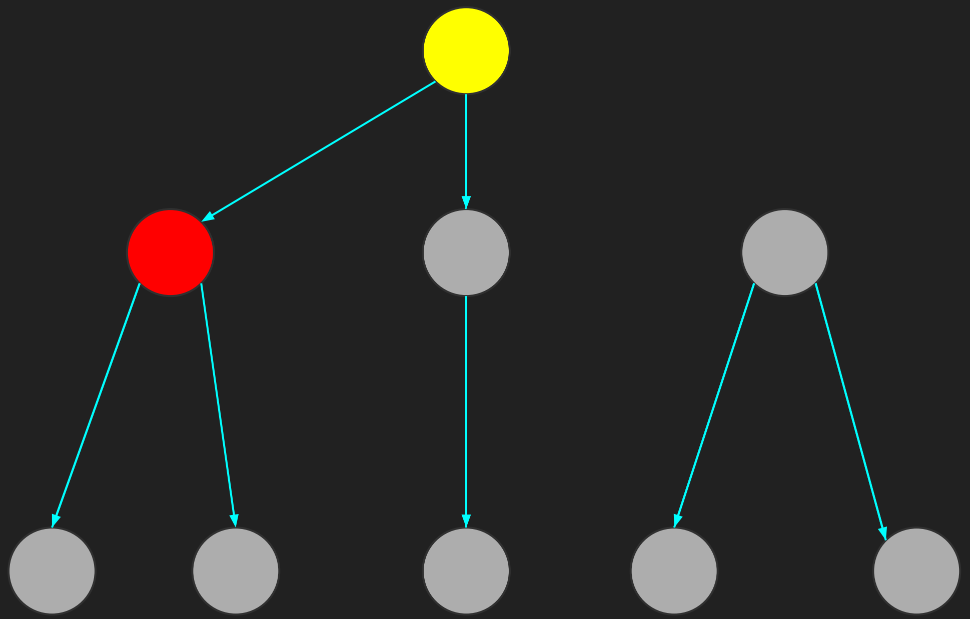
# Top-Down



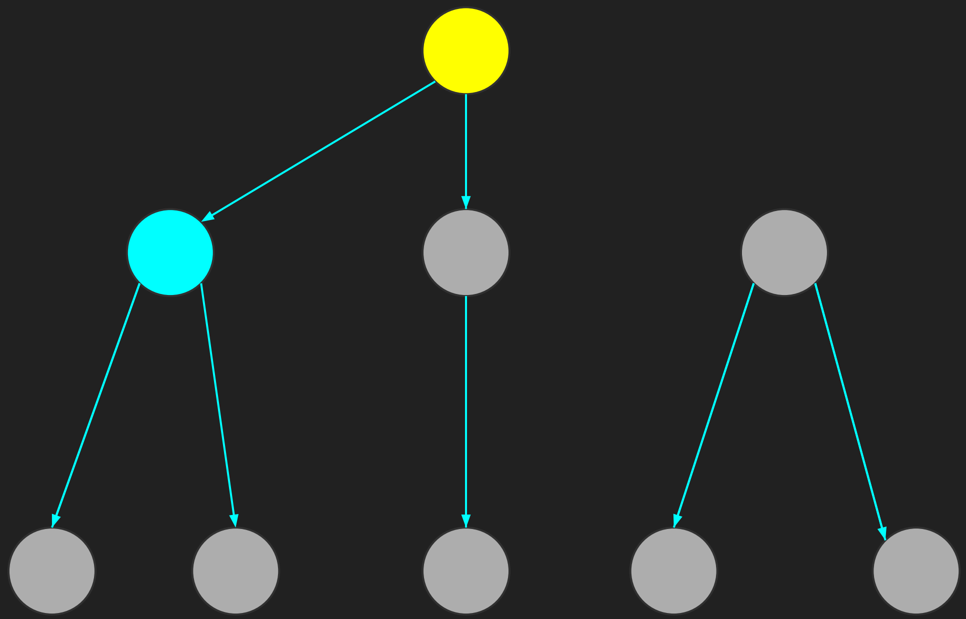
# Top-Down - but what if?



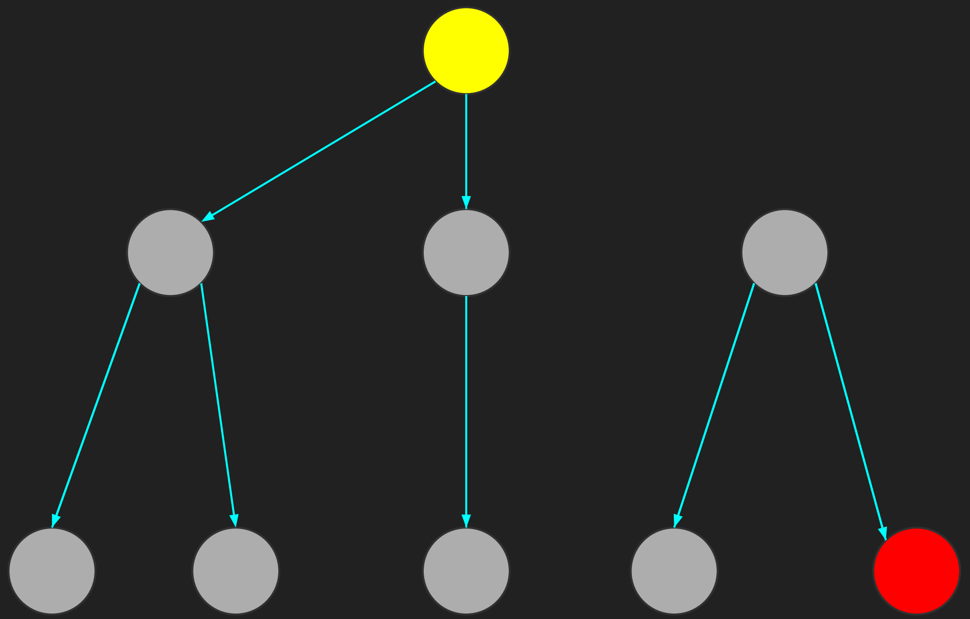
# Top-Down - but what if?



# Top-Down - but what if?

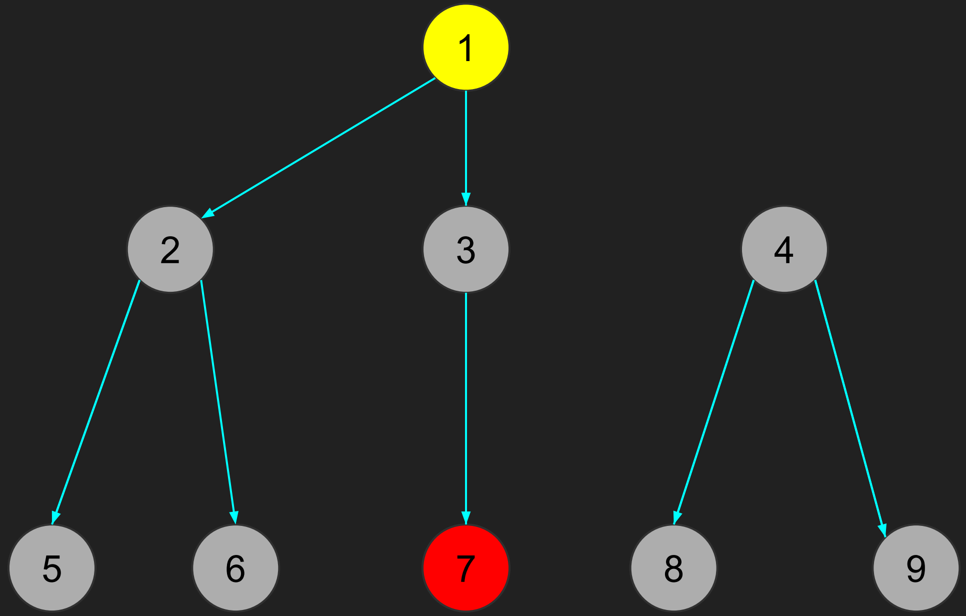


# Top-Down - but what if?



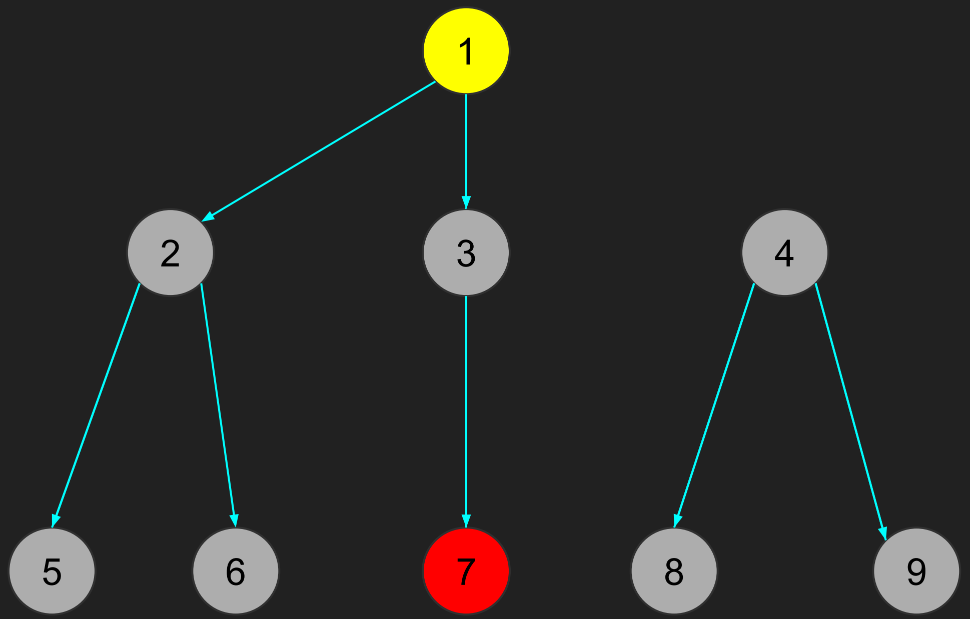
Worst case is O(E) where E is number of edges

# Bottom-Up



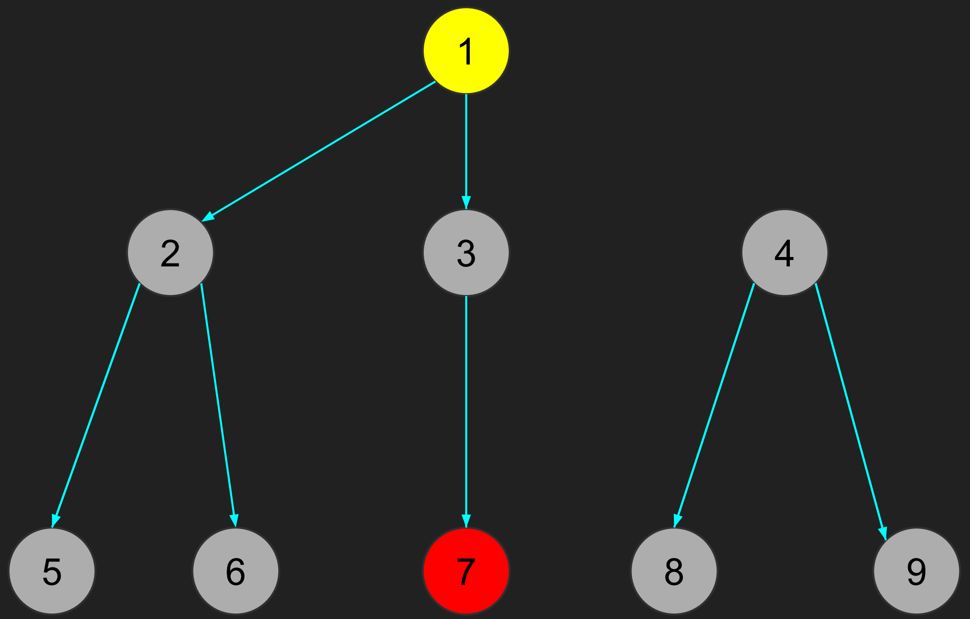

[]

# Bottom-Up



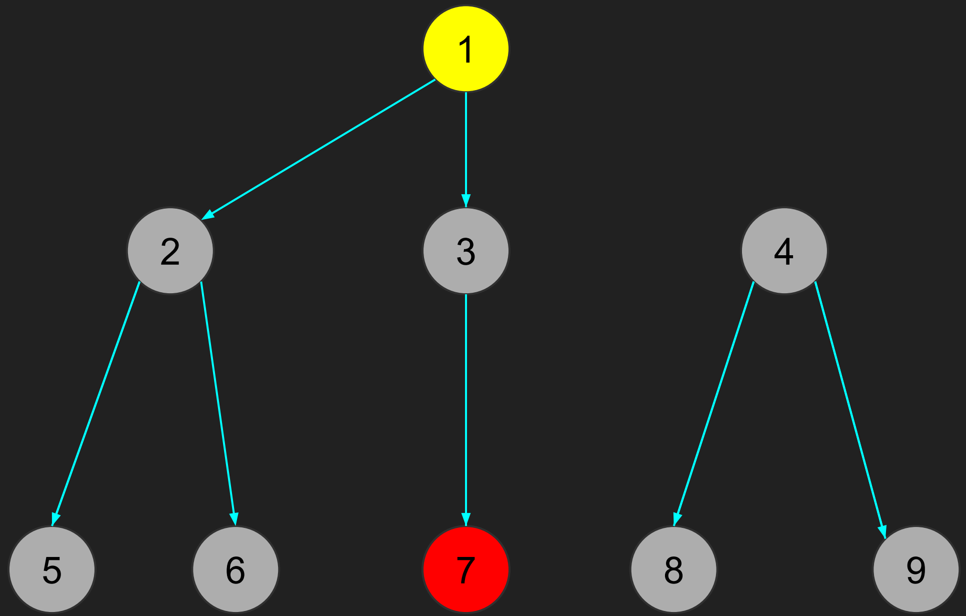

[5, 6, 7, 8, 9]

# Bottom-Up



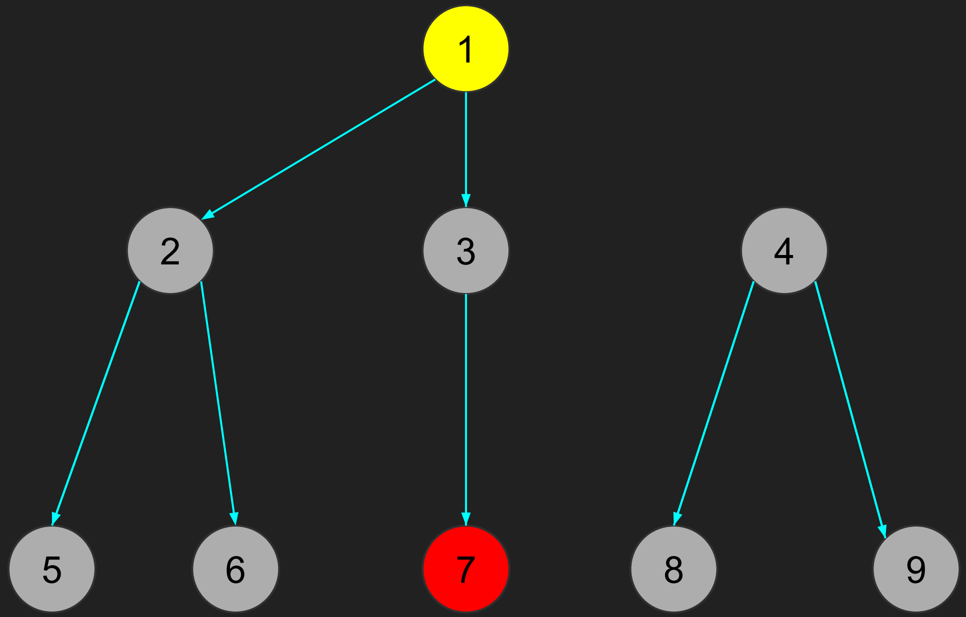

[5, 6, 7, (2,5), (2,6), (3,7), 8, 9, (4,8), (4,9)]

# Bottom-Up



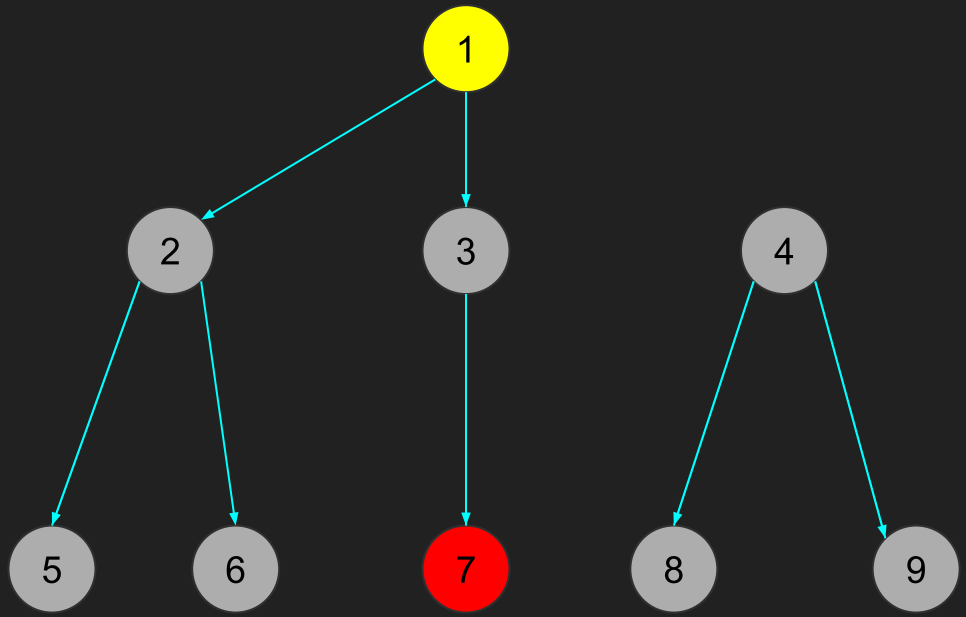

[5, 6, 7, (2,5), (2,6), (3,7), (1,2) (1,3), 8, 9, (4,8), (4,9)]

# Bottom-Up




[5, 6, 7, (2,5), (2,6), (3,7), (1,2) (1,3), (1,5), (1,6), (1,7), 8, 9, (4,8), (4,9)]

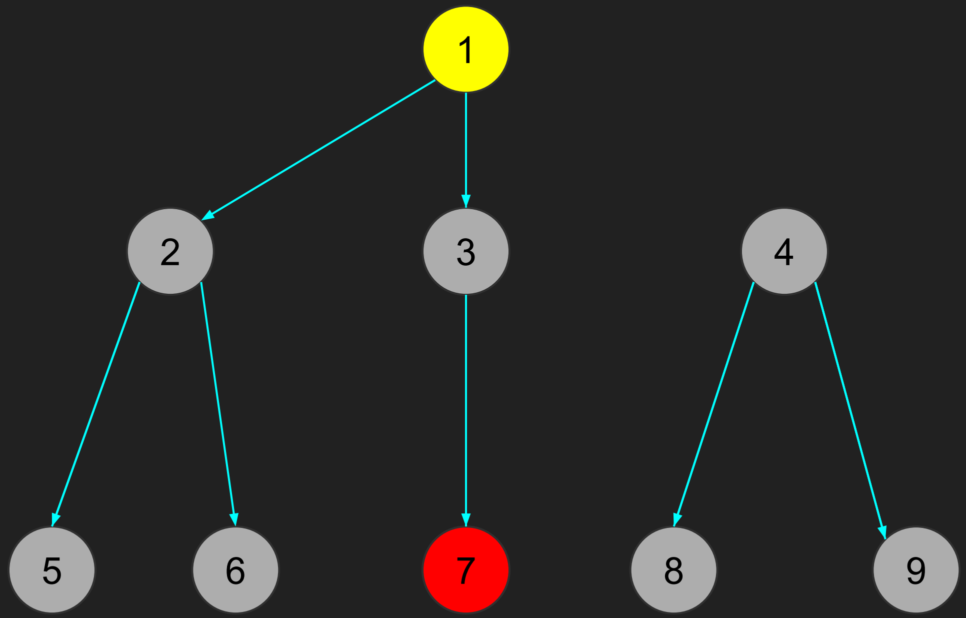
# Bottom-Up




[5, 6, 7, (2,5), (2,6), (3,7), (1,2) (1,3), (1,5), (1,6), (1,7), 8, 9, (4,8), (4,9)]

Now Query (1,7) in the list

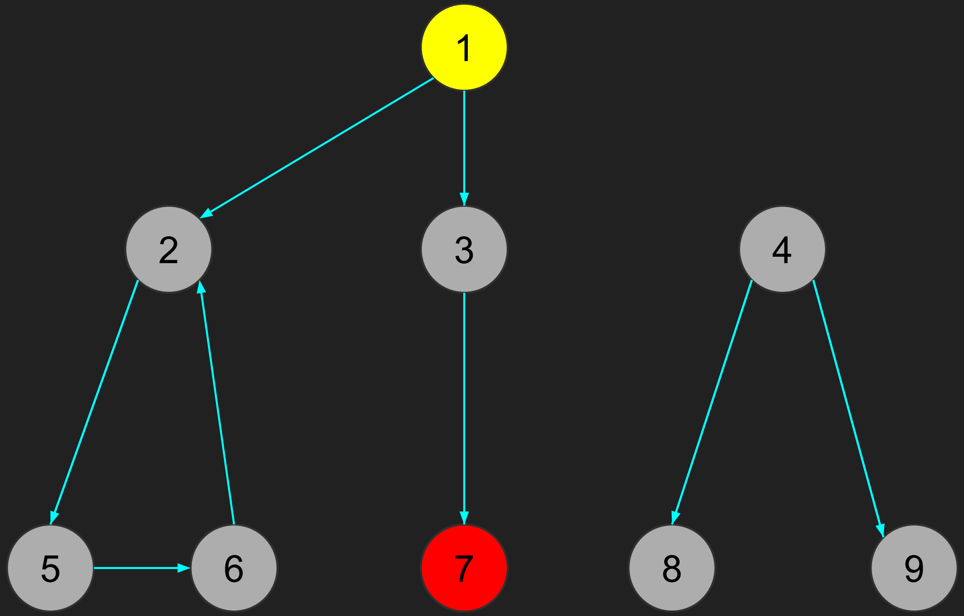
# Bottom-Up




[5, 6, 7, (2,5), (2,6), (3,7), (1,2) (1,3), (1,5), (1,6), (1,7), 8, 9, (4,8), (4,9)]

Now Query (1,7) in the list - O(E) once but then quick

# Bottom-Up - But what if?




[6, 7, (5,6), (2,6), (3,7), (1,2) (1,3), (1,5), (1,6), (1,7), 8, 9, (4,8), (4,9)]

No Difference

# Top-Down: Stack-Based parsing

S -> (SRS | (SR | (RS | (R

R -> )

( ( ) )

STACK - []

# Top-Down: Stack-Based parsing

S -> (SRS | (SR | (RS | (R

R -> )

( ( ) )

▲

STACK - [S, R, S]

# Top-Down: Stack-Based parsing

S -> (SRS | (SR | (RS | (R

R -> )

( ( ) )

▲

STACK - [S, R, S, R, S] and fail

Eventually…

# Top-Down: Stack-Based parsing

S -> (SRS | (SR | (RS | (R

R -> )

( ( ) )

STACK - [S, R]

# Top-Down: Stack-Based parsing

S -> (SRS | (SR | (RS | (R

R -> )

( ( ) )

STACK - [R, R]

# Top-Down: Stack-Based parsing

Key Observations (after fixing a grammar)

- Grammar in Greibach Normal Form for convenience

# Top-Down: Stack-Based parsing

Key Observations  (after fixing a grammar)

- Grammar in Greibach Normal Form for convenience
- Complexity of $O(R^n)$ with n size of input and R the number of rules

# Top-Down: Stack-Based parsing

Key Observations  (after fixing a grammar)

- Grammar in Greibach Normal Form for convenience
- Complexity of O(R^n) with n size of input and R the number of rules
- Cannot have useful error messages  (but this can be helped with look ahead features)

# Idea of Look-Ahead - Recall

S -> (SRS | (SR | (RS | (R

R -> )

( ( ) )

STACK - [S, R, S]

# Idea of Look-Ahead

S -> (SRS | (SR | (RS | (R

R -> )

( ( ) )

STACK - [S, R, S]

We look one step ahead in our set of rules and see if S can parse ( and ), S can parse )).

# Bottom-Up: CKY Algorithm

S -> AB | c

A -> AB | a

B -> BA | b

“ a b a b a ”

0 1 2 3 4 5

| | | | | |
|---|---|---|---|---|
| A | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

# Bottom-Up: CKY Algorithm

S -> AB | c

A -> AB | a

B -> BA | b

" a b a b a "

0 1 2 3 4 5

| | | | | |
|---|---|---|---|---|
| A | | | | |
| | B | | | |
| | | A | | |
| | | | B | |
| | | | | A |

# Bottom-Up: CKY Algorithm

S -> AB | c

A -> AB | a

B -> BA | b

" a b a b a "

0 1 2 3 4 5

| | | | | |
|---|---|---|---|---|
| A | S, A | | | |
| | B | | | |
| | | A | | |
| | | | B | |
| | | | | A |

# Bottom-Up: CKY Algorithm

S -> AB | c

A -> AB | a

B -> BA | b

“ a b a b a ”

0 1 2 3 4 5

| | | | | |
|---|---|---|---|---|
| A | S, A | | | |
| | B | B | | |
| | | A | S, A | |
| | | | B | B |
| | | | | A |

# Bottom-Up: CKY Algorithm

S -> AB | c

A -> AB | a

B -> BA | b

“ a b a b a ”

0 1 2 3 4 5

| | | | | |
|---|---|---|---|---|
| A | S, A | S, A, – | | |
| | B | B | | |
| | | A | S, A | |
| | | | B | B |
| | | | | A |

# Bottom-Up: CKY Algorithm

S -> AB | c

A -> AB | a

B -> BA | b

" a b a b a "

0 1 2 3 4 5

| | | | | |
|---|---|---|---|---|
| A | S, A | S, A | S, A | S, A |
| | B | B | B | B |
| | | A | S, A | S, A |
| | | | B | B |
| | | | | A |

# Bottom-Up: CKY Algorithm (for a fixed grammar)

- O(n^3) for CNF

# Bottom-Up: CKY Algorithm (for a fixed grammar)

- $O(n^3)$ for CNF

- In general, it is $O(n^{(k+1)})$ where k is the maximum number of non terminals in a rule

# Bottom-Up: CKY Algorithm (for a fixed grammar)

- O(n^3) for CNF
- In general, it is O(n^(k+1)) where k is the maximum number of non terminals in a rule
- This implementation is not natural to prolog

# CKY Algorithm - Our implementation

S -> AB | c

A -> AB | a

B -> BA | b

“ a b a b a ”

0 1 2 3 4 5

```
[]

Infer ((S, X, Y), L):-
        member ((A, X, Z), L),
        member ((B, Z, Y), L).
infer((S, X, X+1), L):-
        member((‘c’, 0, 1), L)
```

# CKY Algorithm - Our implementation

S -> AB | c

A -> AB | a

B -> BA | b

" a b a b a "

0 1 2 3 4 5

```
[(A, 0, 1)]
Infer ((S, X, Y), L):-
        member ((A, X, Z), L),
        member ((B, Z, Y), L).
infer((S, X, X+1), L):-
        member(('c', 0, 1), L)
```

# CKY Algorithm - Our implementation

S -> AB | c

A -> AB | a

B -> BA | b

“ a b a b a ”

0 1 2 3 4 5

[(A, 0, 1), (B, 1, 2), (A, 2, 3), (B, 3, 4), (A, 4, 5)]

```
Infer ((S, X, Y), L):-
        member ((A, X, Z), L),
        member ((B, Z, Y), L).
infer((S, X, X+1), L):-
        member((‘c’, 0, 1), L)
```

# CKY Algorithm - Our implementation

S -> AB | c

A -> AB | a

B -> BA | b

“ a b a b a ”

0 1 2 3 4 5

```
[(A, 0, 1), (B, 1, 2), (A, 2, 3), (B, 3, 4), (A, 4, 5), (S, 0, 2), (A, 0, 2)...]
Infer ((S, X, Y), L):-
        member ((A, X, Z), L),
        member ((B, Z, Y), L).
infer((S, X, X+1), L):-
        member((‘c’, 0, 1), L)
```

# CKY Algorithm - Our implementation

- Idea is same as before!

# CKY Algorithm - Our implementation

- Idea is same as before!

- By changing the nature of inference, we have made the algorithm more free

# Outlook

- Combining Bottom-Up and Top-Down - Earley’s Algorithm

# Combining Bottom-Up and Top-Down - Earley's Algorithm

- runs in $O(n^3)$ for all

# Combining Bottom-Up and Top-Down - Earley's Algorithm

- runs in O(n^3) for all

- No obvious and simple implementation in Prolog (to me at least)

# Outlook

- Combining Bottom-Up and Top-Down - Earley's Algorithm

- A better Parsing Algorithm and connections to Matrices

# A better Parsing Algorithm and connections to Matrices

- Parsing a CFG = Boolean Matrix Multiplication

# A better Parsing Algorithm and connections to Matrices

- Parsing a CFG = Boolean Matrix Multiplication

- Theoretical lower bound unknown

# A better Parsing Algorithm and connections to Matrices

- Parsing a CFG = Boolean Matrix Multiplication

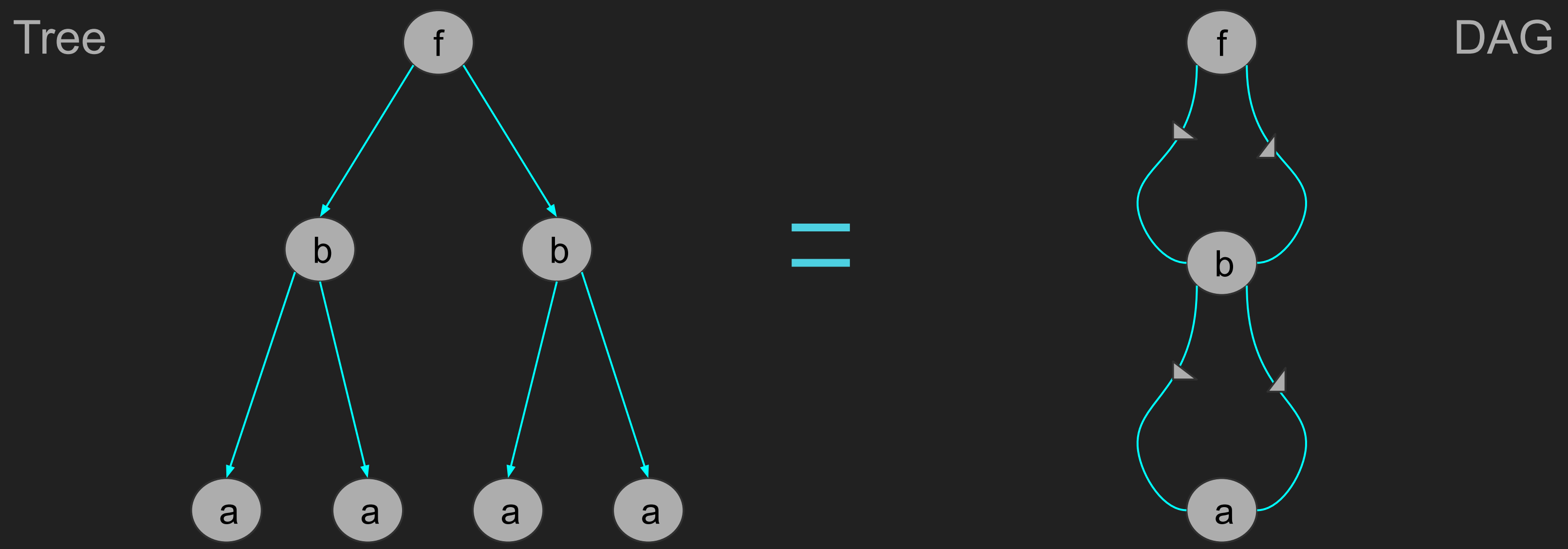
- Theoretical lower bound unknown

- Current best is ~ O(n^2.37)

# Outlook

- Combining Bottom-Up and Top-Down - Earley’s Algorithm
- A better Parsing Algorithm and connections to Matrices
- Term sharing and common subformulae elimination

# Term Sharing




Much easier to do with a Bottom-Up Implementation

# Common SubFormulae Elimination

A(B(C), D(B(C)))  = A(K, D(K)), K = B(C)

Again much easier with Bottom-Up Approaches!

# Conclusions

- Analyzed top down and bottom up approaches

# Conclusions

- Analyzed top down and bottom up approaches

- Found a new prolog implementation - generalising the algorithm

# Conclusions

- Analyzed top down and bottom up approaches
- Found a new prolog implementation - generalising the algorithm
- Based on this work we can do a better logical analysis of Earley’s algorithm

# Conclusions

- Analyzed top down and bottom up approaches
- Found a new prolog implementation - generalising the algorithm
- Based on this work we can do a better logical analysis of Earley’s algorithm
- This may lead to better algorithms

# Conclusions

- Analyzed top down and bottom up approaches
- Found a new prolog implementation - generalising the algorithm
- Based on this work we can do a better logical analysis of Earley’s algorithm
- This may lead to better algorithms
- Also helpful in CSFE and Term-Sharing

# Thank You!