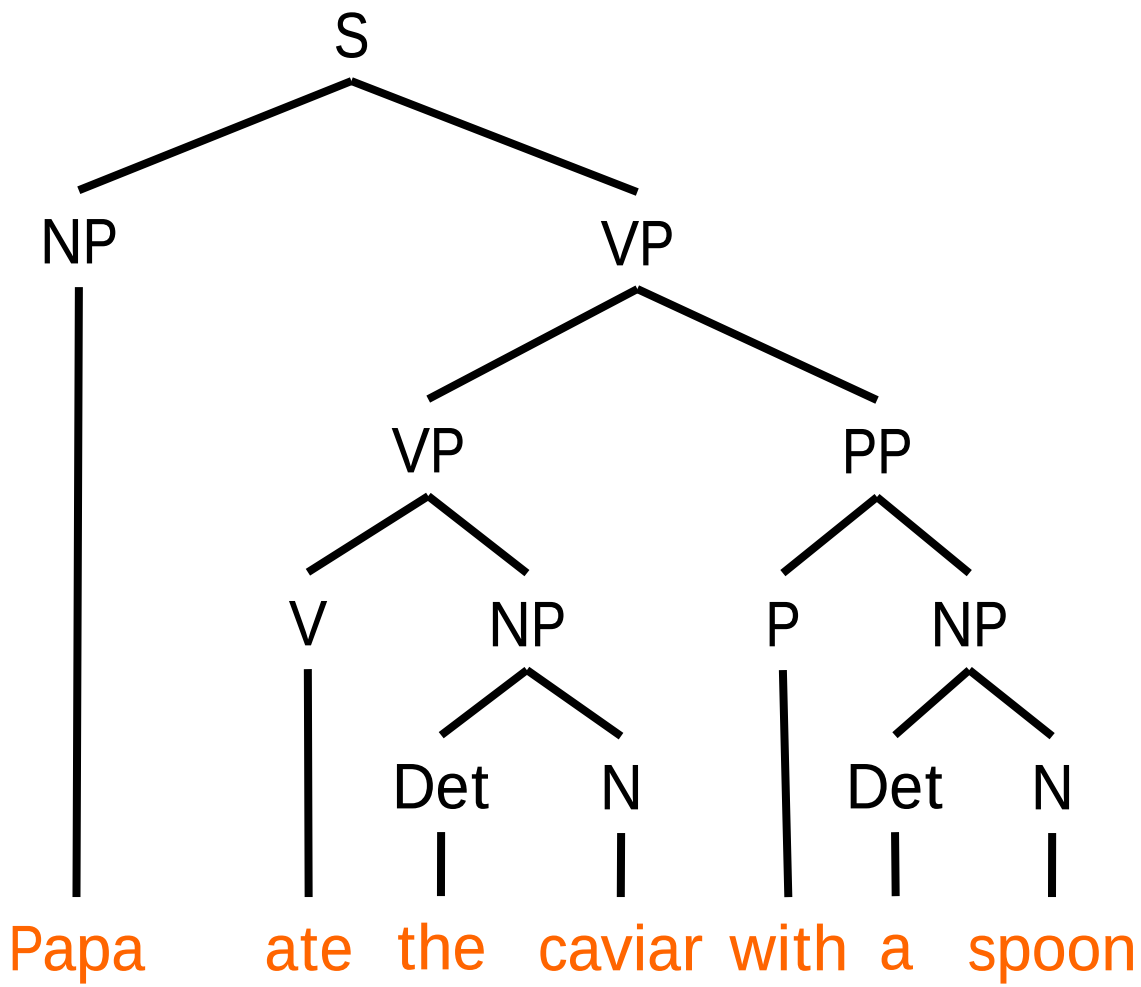
Parsing

What is Parsing?

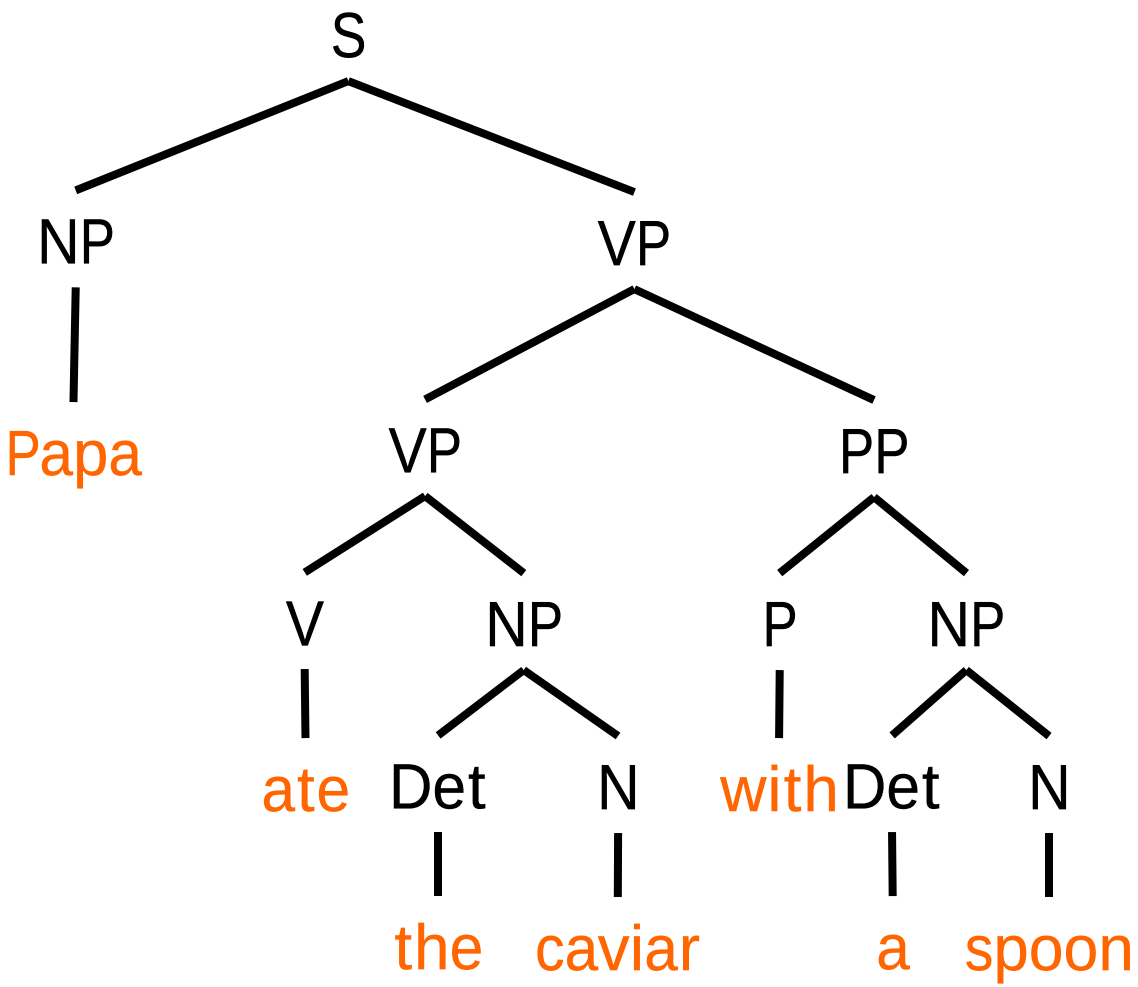
S → NP VP
NP → Det N
NP → NP PP
VP → V NP
VP → VP PP
PP → P NP



NP → Papa
N → caviar
N → spoon
V → spoon
V → ate
P → with
Det → the
Det → a

What is Parsing?

S → NP VP
NP → Det N
NP → NP PP
VP → V NP
VP → VP PP
PP → P NP



NP → Papa
N → caviar
N → spoon
V → spoon
V → ate
P → with
Det → the
Det → a

Programming languages

```
printf ("/charset [%s",
        (re_opcode_t) *(p - 1) == charset_not ? "^" : "");

assert (p + *p < pend);

for (c = 0; c < 256; c++)
  if (c / 8 < *p && (p[1 + (c/8)] & (1 << (c % 8)))) {
     /* Are we starting a range?  */
     if (last + 1 == c && ! inrange) {
        putchar ('-');
        inrange = 1;
     }
     /* Have we broken a range?  */
     else if (last + 1 != c && inrange) {
        putchar (last);
        inrange = 0;
     }

     if (! inrange)
        putchar (c);

     last = c;
  }
```

- Easy to parse.
- Designed that way!

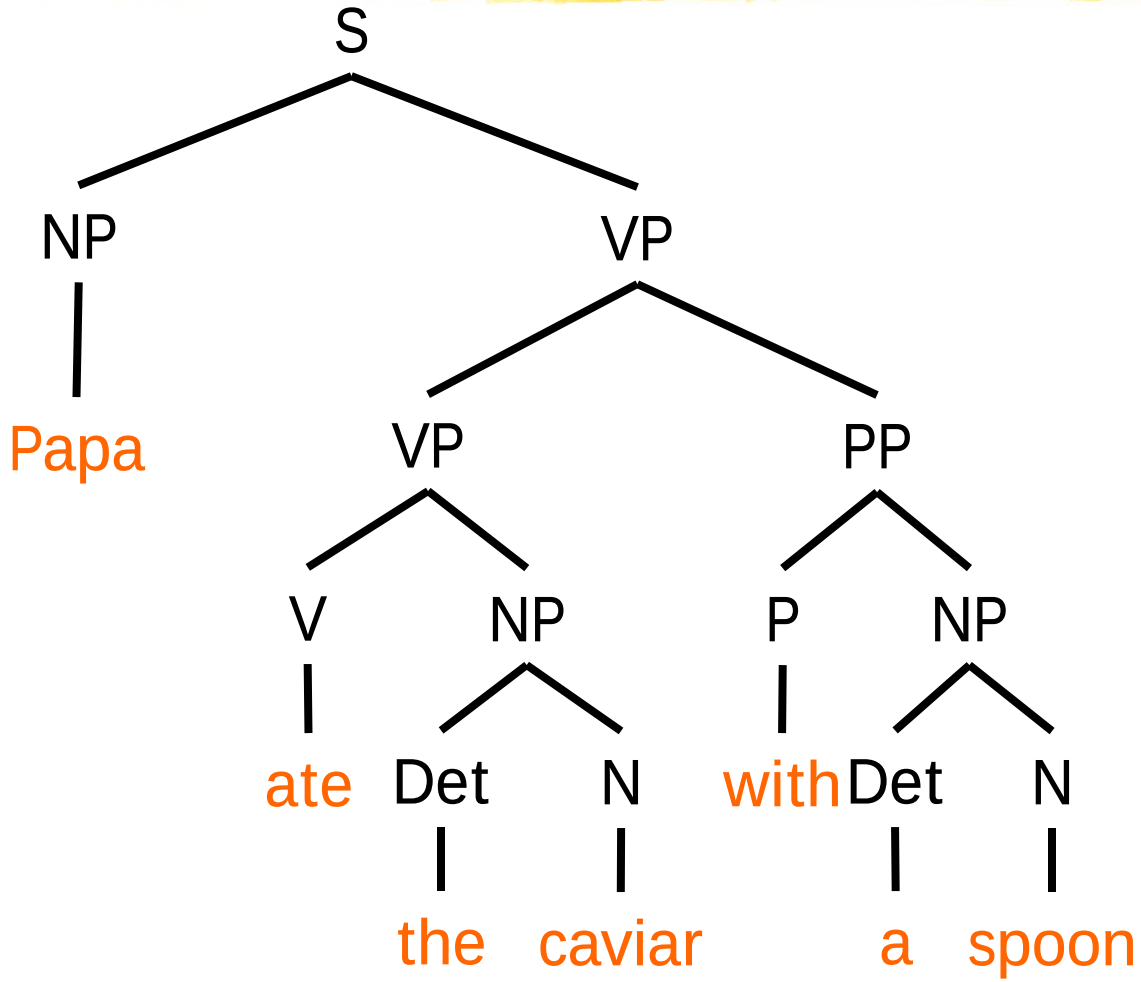
Natural languages

```
printf "/charset %s", re_opcode_t *p - 1 == charset_not ? "^"
: ""; assert p + *p < pend; for c = 0; c < 256; c++ if c / 8 <
*p && p1 + c/8 & 1 << c % 8 Are we starting a range? if last +
1 == c && !  inrange putchar '-'; inrange = 1; Have we broken
a range? else if last + 1 != c && inrange putchar last;
inrange = 0; if !  inrange putchar c; last = c;
```

- No {} () [] to indicate scope & precedence
- Lots of overloading (arity varies)
- Grammar isn’t known in advance!
- Context-free grammar not best formalism

Ambiguity

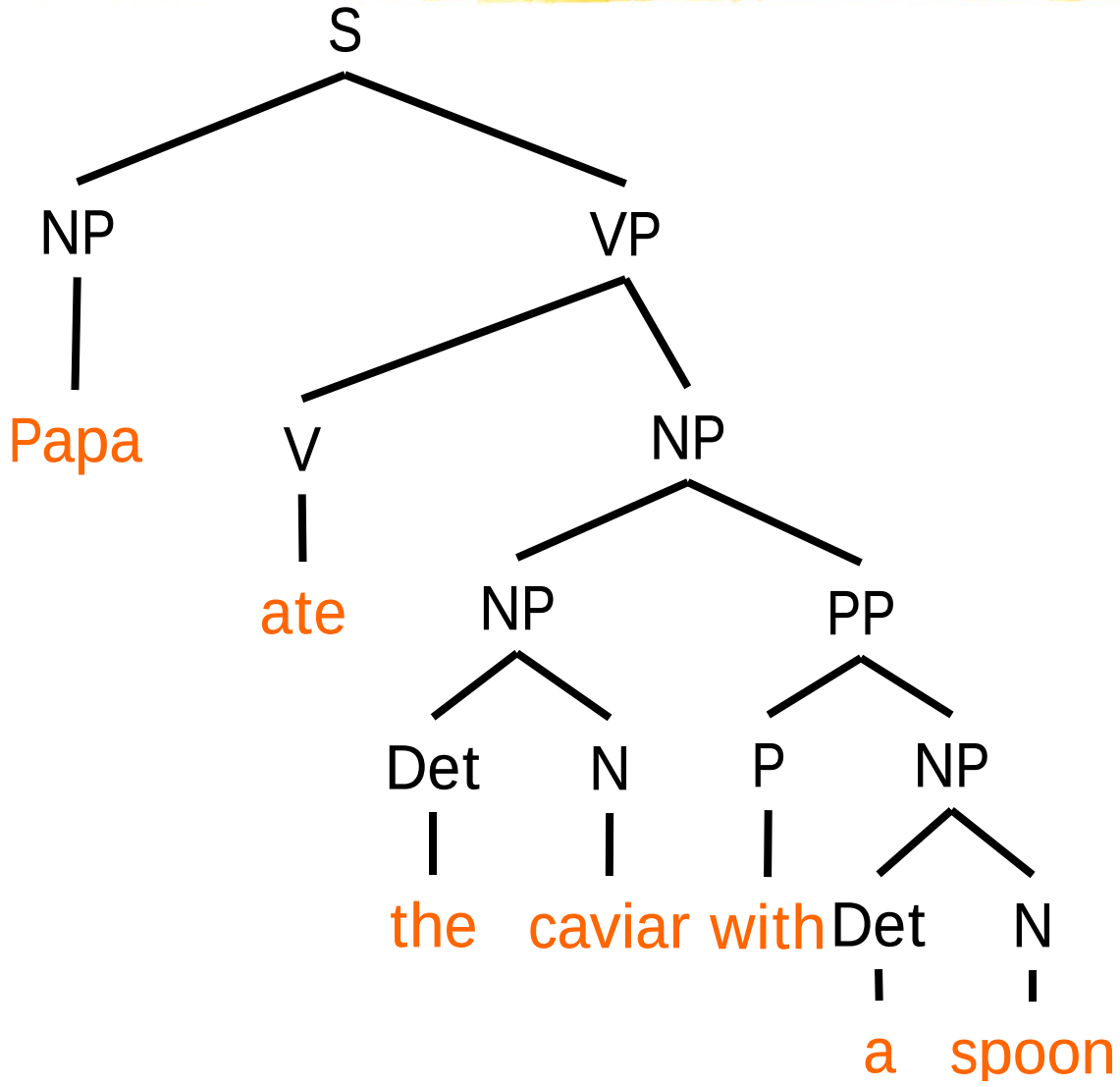
S → NP VP
NP → Det N
NP → NP PP
VP → V NP
VP → VP PP
PP → P NP



NP → Papa
N → caviar
N → spoon
V → spoon
V → ate
P → with
Det → the
Det → a

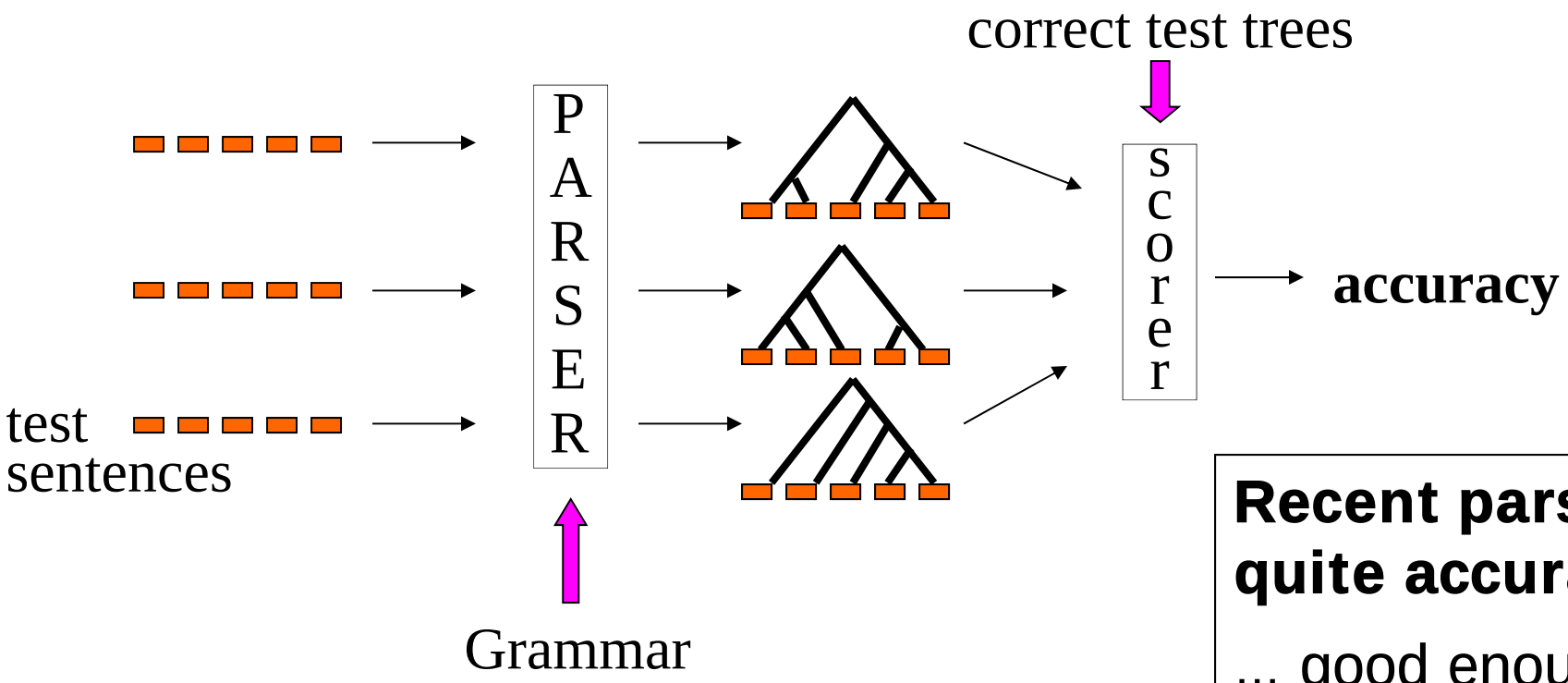
Ambiguity

S → NP VP
NP → Det N
NP → NP PP
VP → V NP
VP → VP PP
PP → P NP



NP → Papa
N → caviar
N → spoon
V → spoon
V → ate
P → with
Det → the
Det → a

The parsing problem

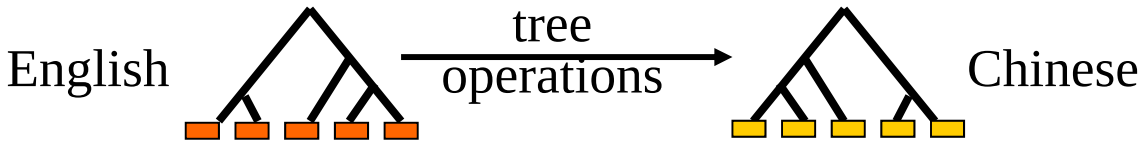


Recent parsers quite accurate

... good enough to help a range of NLP tasks!

Applications of parsing (1/2)

- Machine translation (Alshawi 1996, Wu 1997, ...)



- Speech synthesis from parses (Prevost 1996)

  The government plans to raise income tax.

  The government plans to raise income tax the imagination.

- Speech recognition using parsing (Chelba et al 1998)

  Put the file in the folder.

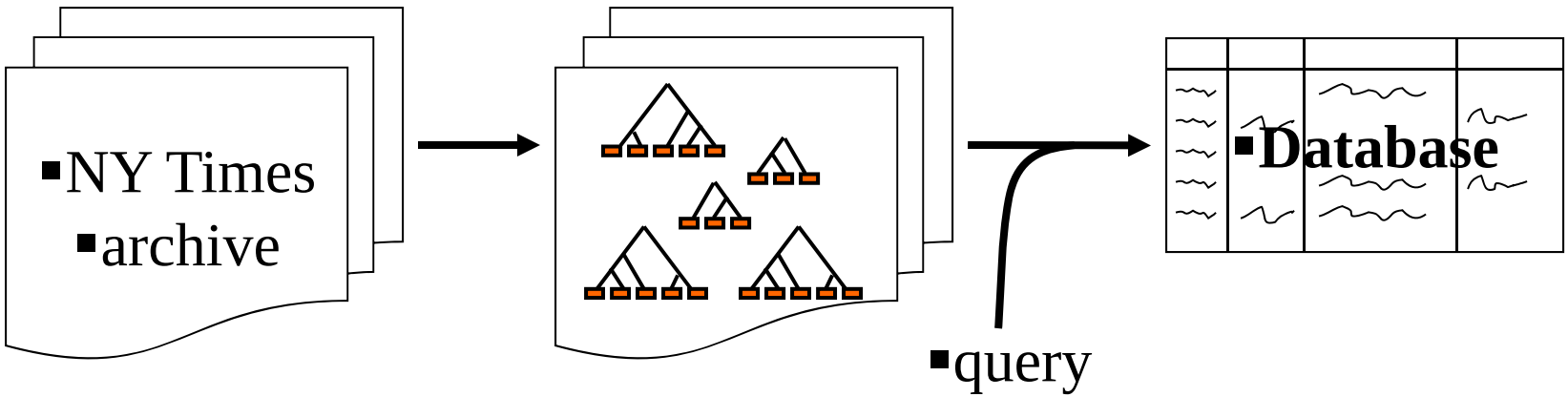
  Put the file and the folder.

Applications of parsing (2/2)

- Grammar checking (Microsoft)
- Indexing for information retrieval (Woods 1997)

  ... washing a car with a hose ... ⟶ vehicle maintenance

- Information extraction (Hobbs 1996)



Parsing for Interpretation

- Most linguistic properties are defined over trees.
- One needs to parse to see subtle distinctions. E.g.:

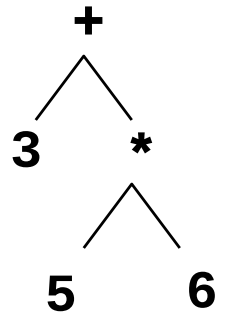
Sara dislikes criticism of her. *(her ≠ Sara)*

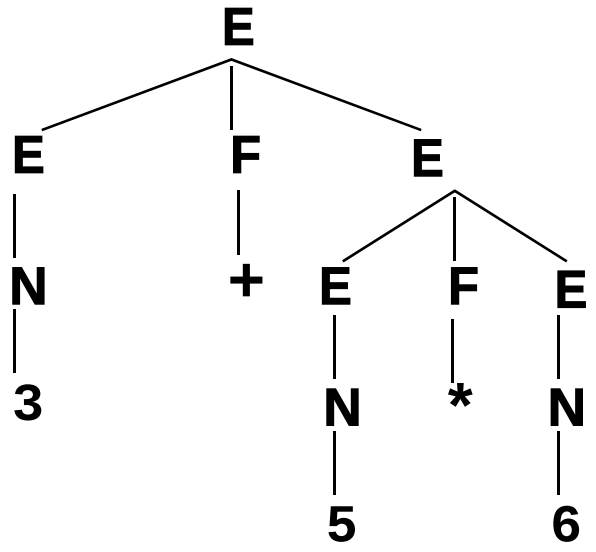
Sara dislikes criticism of her by anyone. *(her ≠ Sara)*

Sara dislikes anyone's criticism of her. *(her = Sara* or *her ≠ Sara)*

Parsing → Compositional Semantics

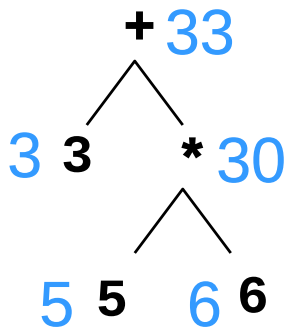
- What is meaning of 3+5*6?
- First parse it into 3+(5*6)

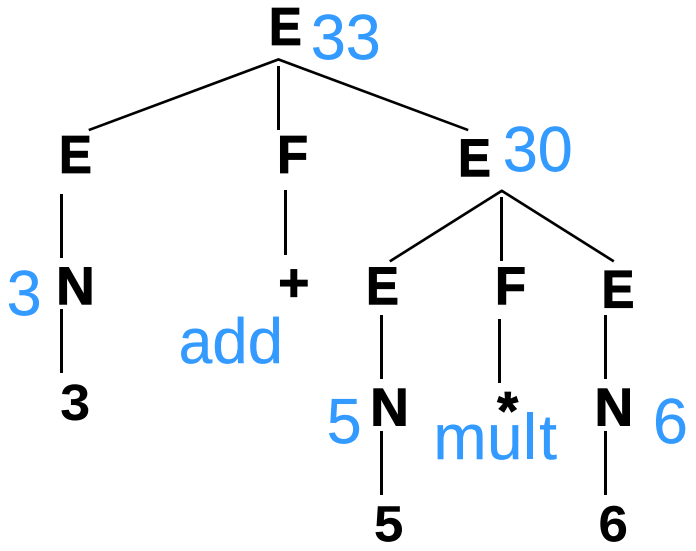




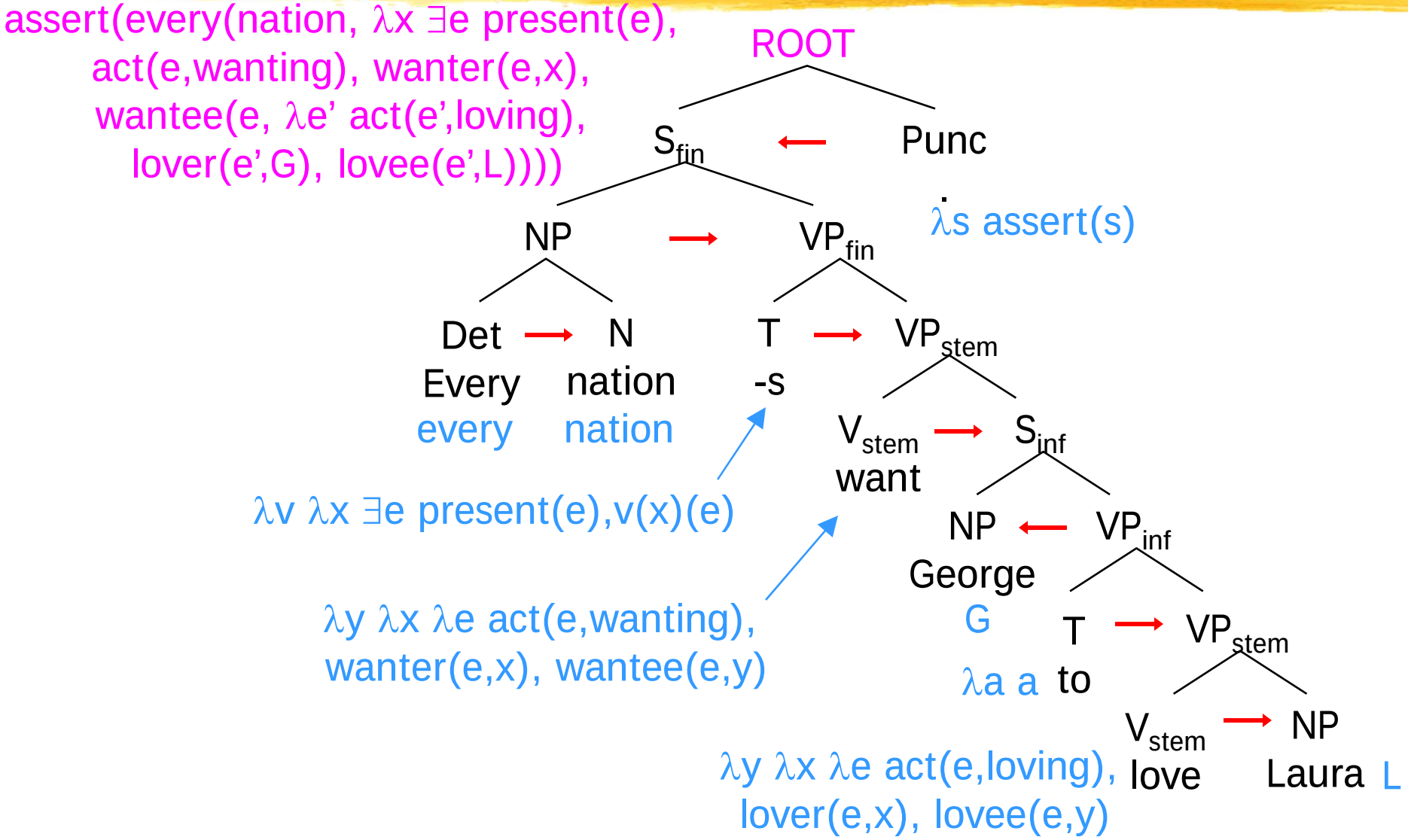
Parsing → Compositional Semantics

- What is meaning of 3+5*6?
- First parse it into 3+(5*6)
- Now give a meaning to each node in the tree (bottom-up)





Parsing → Compositional Semantics



- Now let's develop some parsing algorithms!

- What substrings of this sentence *are* NPs?

0 1 2 3 4 5 6 7 8 9
The government plans to raise income tax are unpopular

- Which substrings *could* be NPs in the right context?
- Which substrings *could* be other types of phrases?
- How would you defend a substring on your list?

0 1 2 3 4 5 6 7

"Papa ate the caviar with a spoon"

- S → NP VP
- NP → Det N
- NP → NP PP
- VP → V NP
- VP → VP PP
- PP → P NP

- NP → Papa
- N → caviar
- N → spoon
- V → spoon
- V → ate
- P → with
- Det → the
- Det → a

0 1 2 3 4 5 6 7

"Papa ate the caviar with a spoon"

First try ... does it work?

- for each constituent on the LIST (Y I Mid)
  - scan the LIST for an adjacent constituent (Z Mid J)
  - if grammar has a rule to combine them (X → Y Z)
    - then add the result to the LIST (X I J)

0 1 2 3 4 5 6 7
"Papa ate the caviar with a spoon"

Second try ...

- initialize the list with parts-of-speech (T J-1 J)

  where T is a preterminal tag (like Noun) for the Jth word
- for each constituent on the LIST (Y I Mid)
  - scan the LIST for an adjacent constituent (Z Mid J)
  - if grammar has a rule to combine them (X → Y Z)
    - then add the result to the LIST (X I J)

- if the above loop added anything, do it again!

  (so that X I J gets a chance to combine or be combined with)

0 1 2 3 4 5 6 7
"Papa ate the caviar with a spoon"

Third try ...

- initialize the list with parts-of-speech (T J-1 J)

  where T is a preterminal tag (like Noun) for the Jth word

- for each constituent on the LIST (Y I Mid)
  - for each adjacent constituent on the list (Z Mid J)
    - for each rule to combine them (X → Y Z)
      - add the result to the LIST (X I J)

        if it's not already there

- if the above loop added anything, do it again!

  (so that X I J gets a chance to combine or be combined with)

0 1 2 3 4 5 6 7

"Papa ate the caviar with a spoon"

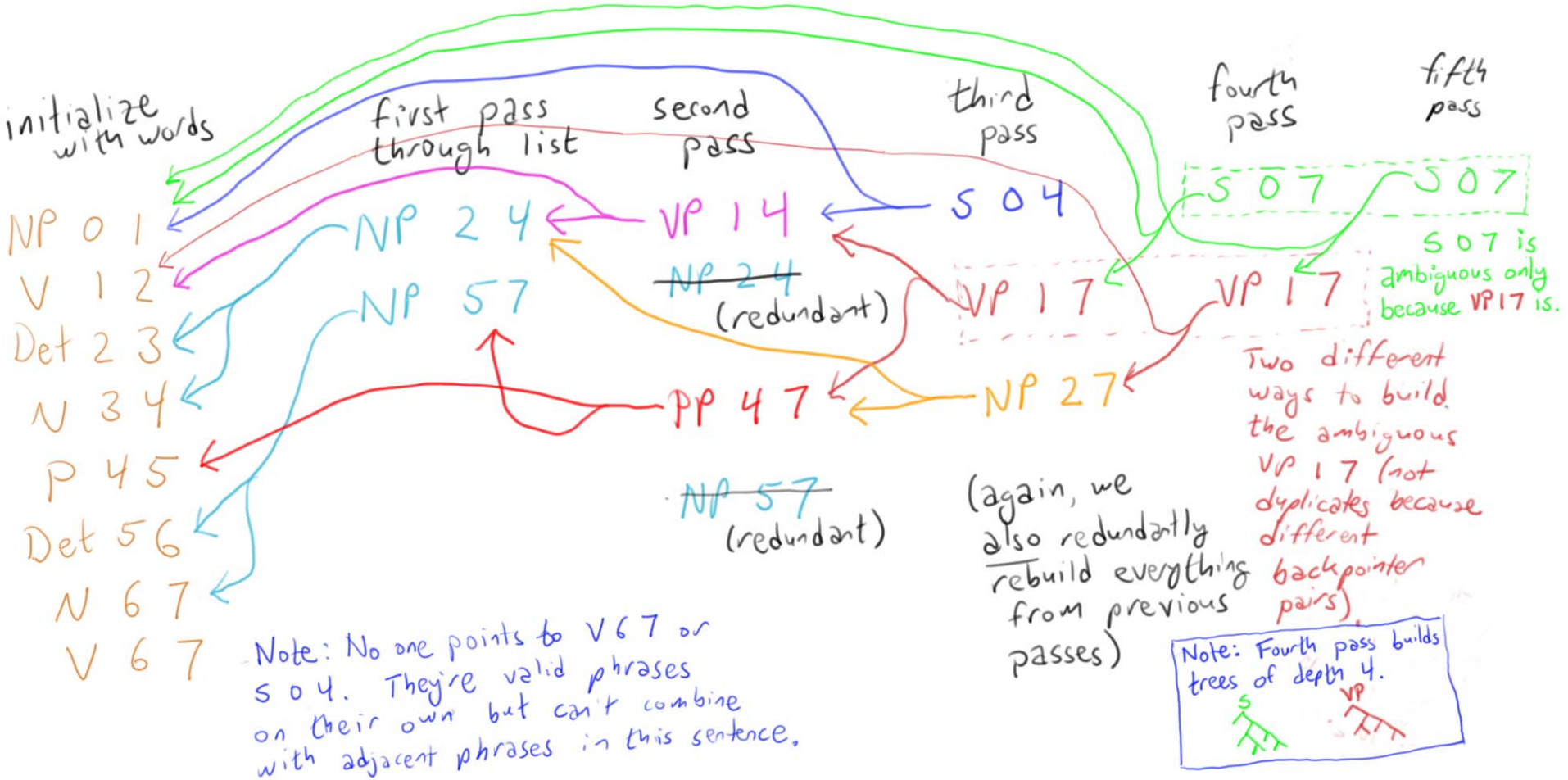
Third try ...

Initialize	1st pass	2nd pass	3rd pass
NP 0 1	NP 2 4	VP 1 4	...
V 1 2	NP 5 7	~~NP 2 4~~	
Det 2 3		PP 4 7	
N 3 4		~~NP 5 7~~	
P 4 5			
Det 5 6			
N 6 7			
V 6 7			

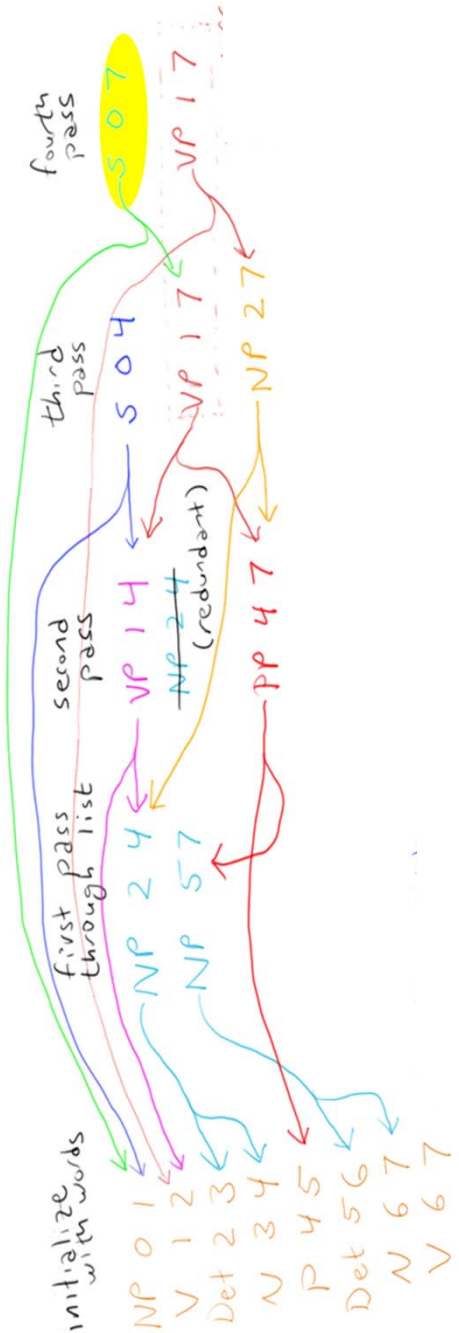
0 1 2 3 4 5 6 7

"Papa ate the caviar with a spoon"

Follow backpointers to get the parse



Turn sideways: See the trees?

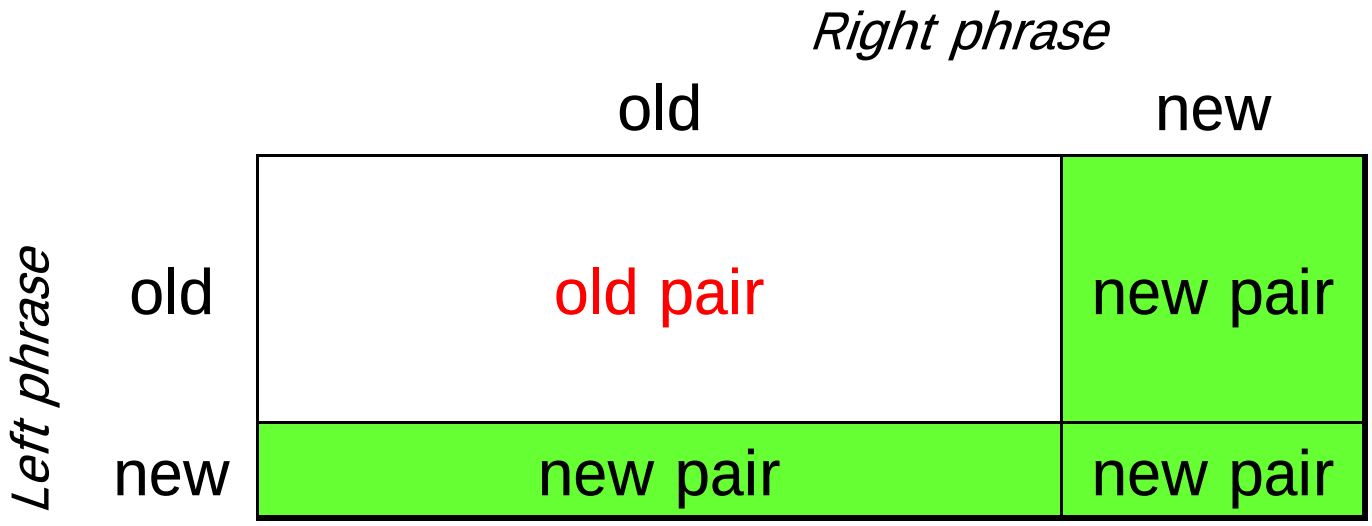


"Papa ate the caviar with a spoon"

Correct but still inefficient ...

We kept checking the same pairs that we'd checked before (both bad and good pairs)

Can't we manage the process in a way that avoids duplicate work?



Correct but still inefficient ...

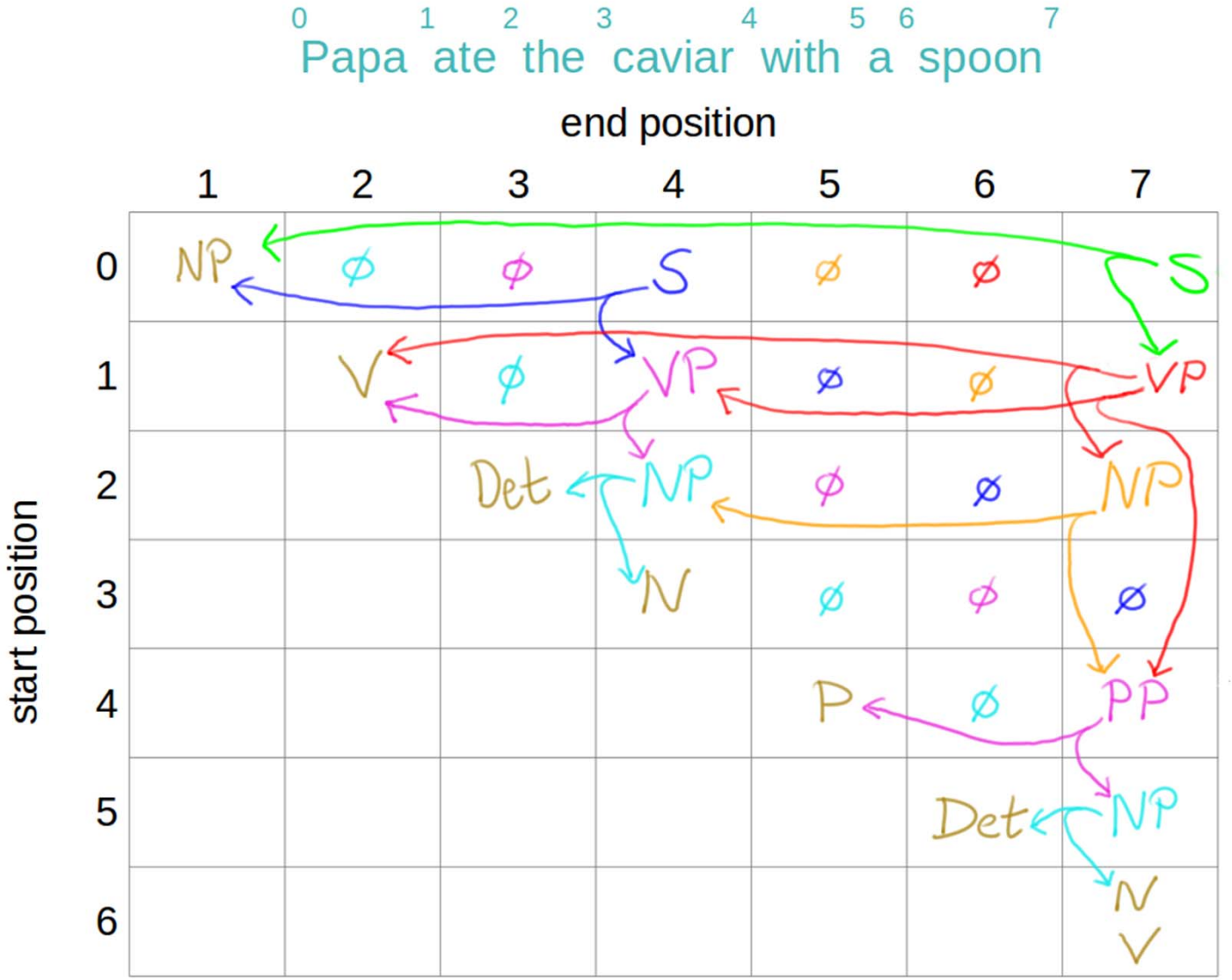
We kept checking the same pairs that we'd checked before (both bad and good pairs)

Can't we manage the process in a way that avoids duplicate work?

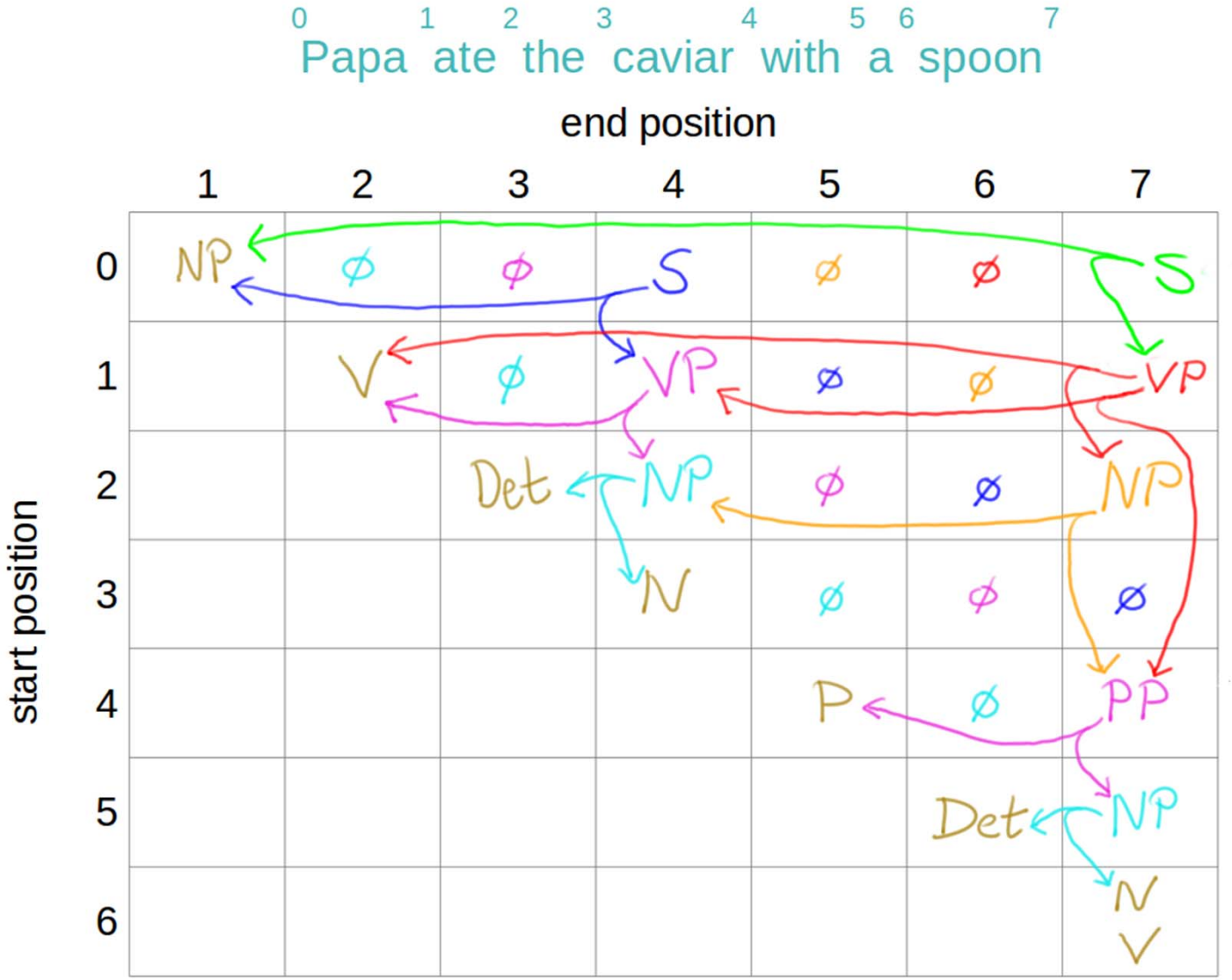
And even finding new pairs was expensive because we had to scan the whole list

Can't we have some kind of index that will help us find adjacent pairs?

Indexing: Store S 0 4 in chart[0,4]

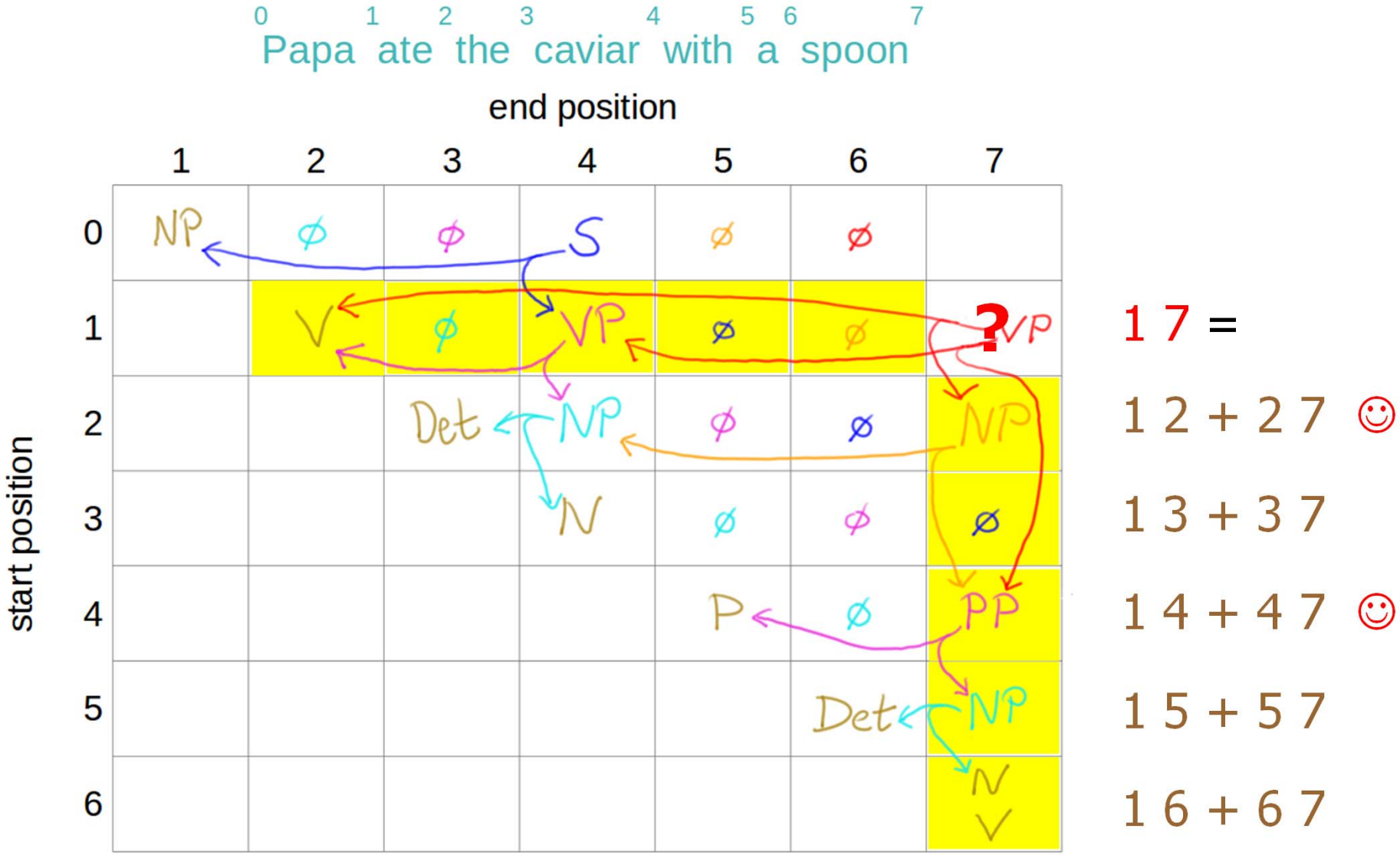


Avoid duplicate work:
Build width-1, then width-2, etc.

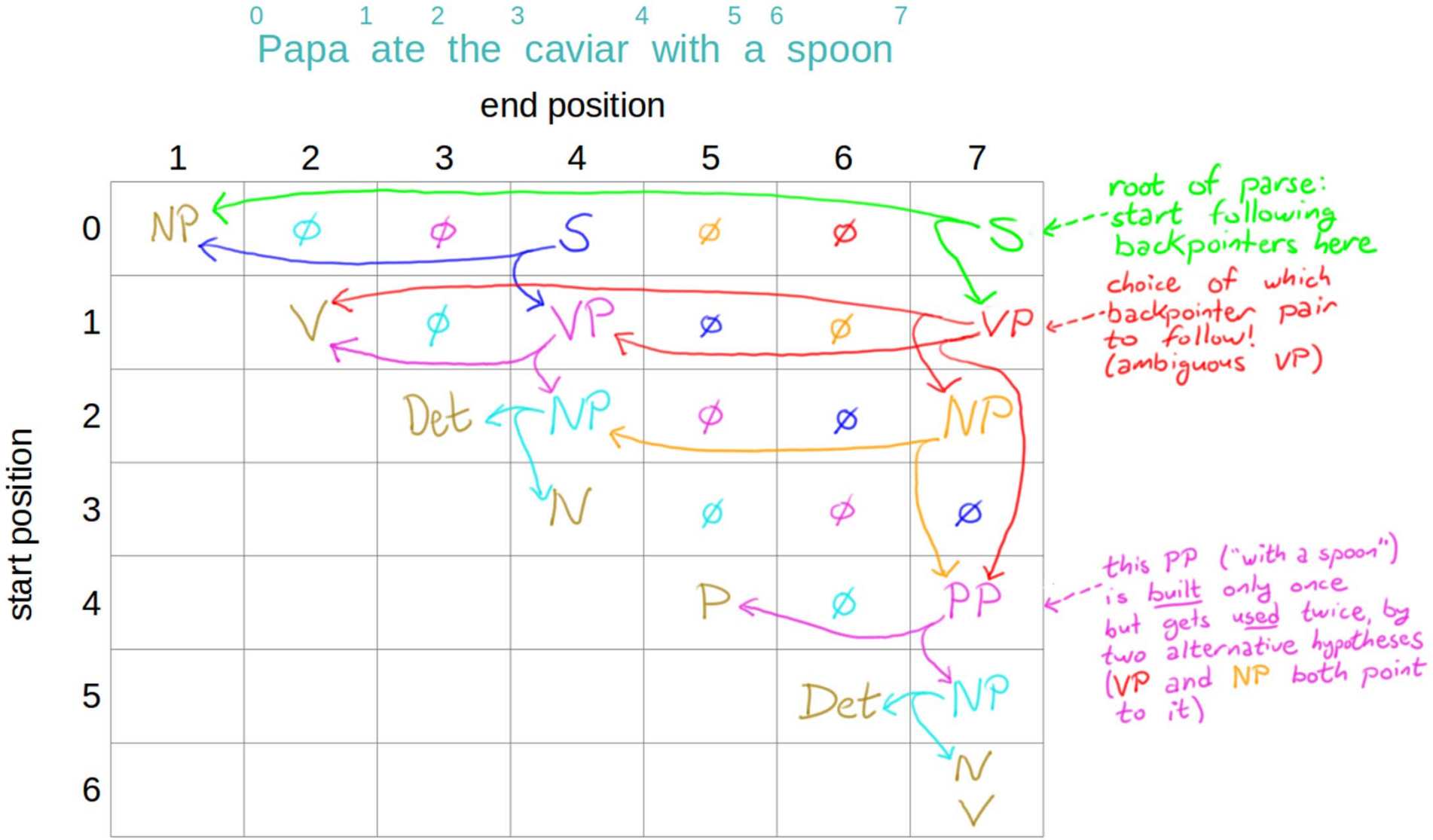


How do we build a width-6 phrase?

(*after* building and indexing *all* shorter phrases)



Avoid duplicate work:
Build width-1, then width-2, etc.



CKY algorithm, recognizer version

- Input: string of n words
- Output: yes/no (since it's only a recognizer)
- Data structure: n × n table
  - rows labeled 0 to n-1
  - columns labeled 1 to n
  - cell [i,j] lists constituents found between i and j

- Basic idea: fill in width-1 cells, then width-2, …

CKY algorithm, recognizer version

- for J := 1 to n
  - Add to [J-1,J] all categories for the Jth word
- for width := 2 to n
  - for start := 0 to n-width // this is I
    - Define end := start + width // this is J
    - for mid := start+1 to end-1 // find all I-to-J phrases
      - for every nonterminal Y in [start,mid]
      - for every nonterminal Z in [mid,end]
      - for all nonterminals X
      - if X → Y Z is in the grammar
      - then add X to [start,end]

Loose ends to tie up (no slides)

- What's the space?
  - Duplicate entries?
  - How many parses could there be?
  - Can we fix this?
- What's the runtime?

CKY algorithm, recognizer version

- for J := 1 to n
  - Add to [J-1,J] all categories for the Jth word
- for width := 2 to n
  - for start := 0 to n-width // this is I
    - Define end := start + width // this is J
    - for mid := start+1 to end-1 // find all I-to-J phrases
      - for every nonterminal Y in [start,mid]
      - for every nonterminal Z in [mid,end]
      - for all nonterminals X
      - if X → Y Z is in the grammar
      - then add X to [start,end]

Alternative version of inner loops

- for J := 1 to n
  - Add to [J-1,J] all categories for the Jth word
- for width := 2 to n
  - for start := 0 to n-width // this is I
    - Define end := start + width // this is J
    - for mid := start+1 to end-1 // find all I-to-J phrases
      - for every rule X → Y Z in the grammar
        if Y in [start,mid] and Z in [mid,end]
        then add X to [start,end]

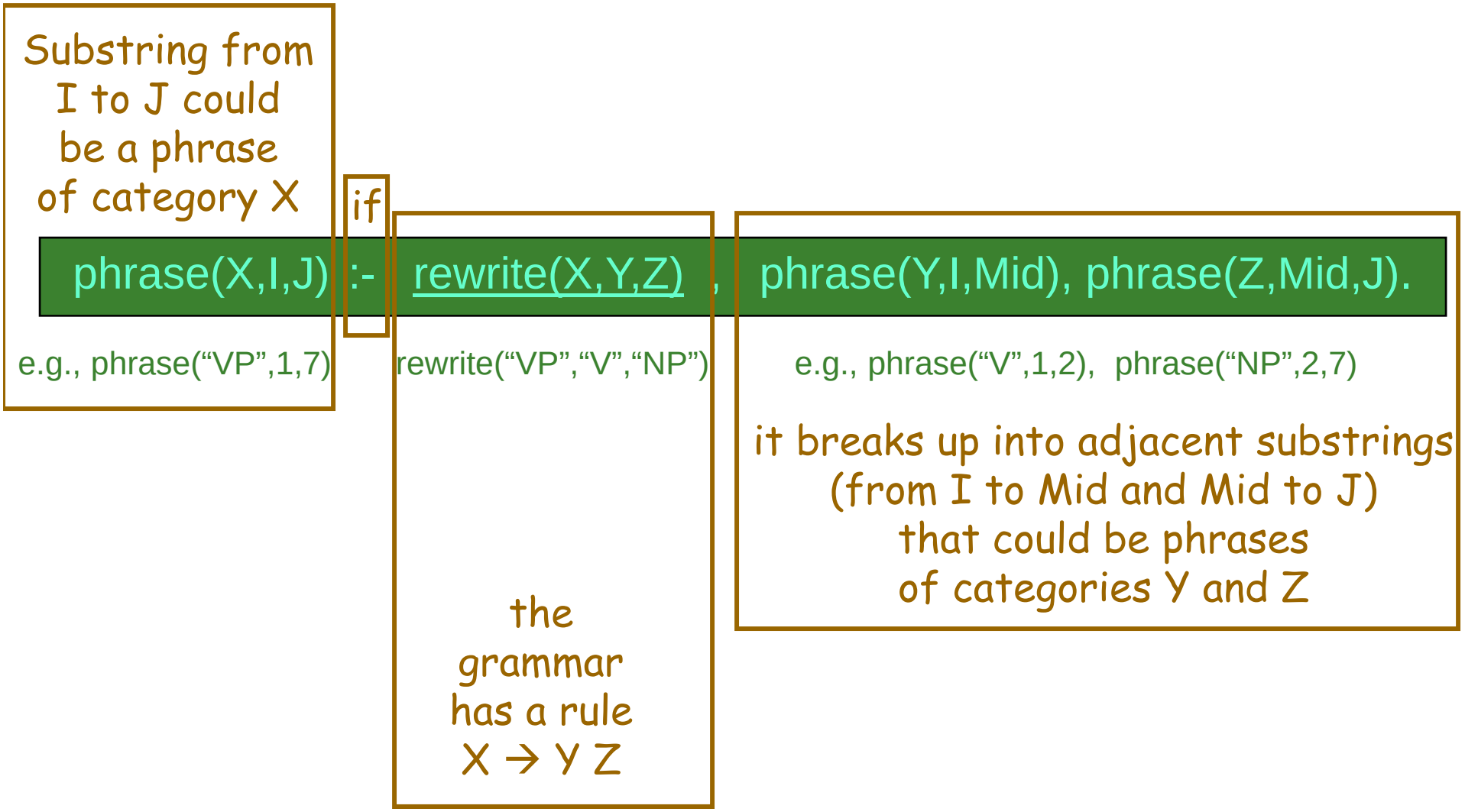
Extensions to discuss (no slides)

- In Chinese, no spaces between the words!?
- Incremental (left-to-right) parsing?
- We assumed rules like X → Y Z or X → word.
  - Grammar in "Chomsky Normal Form." What if it's not?
- How do we choose among parses?

Chart Parsing in Dyna

```
phrase(X,I,J)  :-  rewrite(X,W), word(W,I,J).
phrase(X,I,J)  :-  rewrite(X,Y,Z), phrase(Y,I,Mid), phrase(Z,Mid,J).
goal           :-  phrase(start_symbol, 0, sentence_length).
```

Understanding the Key Rule



("an X can be made of a Y next to a Z")

Chart Parsing in Dyna

```
phrase(X,I,J)  :-  rewrite(X,W), word(W,I,J).
```

- "A word is a phrase" (if grammar allows)

```
phrase(X,I,J)  :-  rewrite(X,Y,Z), phrase(Y,I,Mid), phrase(Z,Mid,J).
```

- "Two adjacent phrases are a phrase" (if grammar allows)

```
goal           :-  phrase(start_symbol, 0, sentence_length).
```

- "A phrase that covers the whole sentence is a parse"
  (achieves our goal by showing that the sentence is grammatical)

```
start_symbol := "S".
sentence_length := 7.
```

Alternatively:

```
sentence_length max=  J for word(_,_,J).
```

Chart Parsing in Dyna

```
phrase(X,I,J)  :-  rewrite(X,W), word(W,I,J).
phrase(X,I,J)  :-  rewrite(X,Y,Z), phrase(Y,I,Mid), phrase(Z,Mid,J).
goal           :-  phrase(start_symbol, 0, sentence_length).
```

- We also need a sentence:

```
word("Papa",0,1).
word("ate",1,2).
word("the",2,3).
word("caviar",3,4).
word("with",4,5).
word("a",5,6).
word("spoon",6,7).
```

- We also need a grammar:

```
rewrite("NP","Papa").
rewrite("N","caviar").
…
```

```
rewrite("S","NP","VP").
rewrite("NP","Det","N").
rewrite("NP","NP","PP").
…
```

Procedural Algorithms

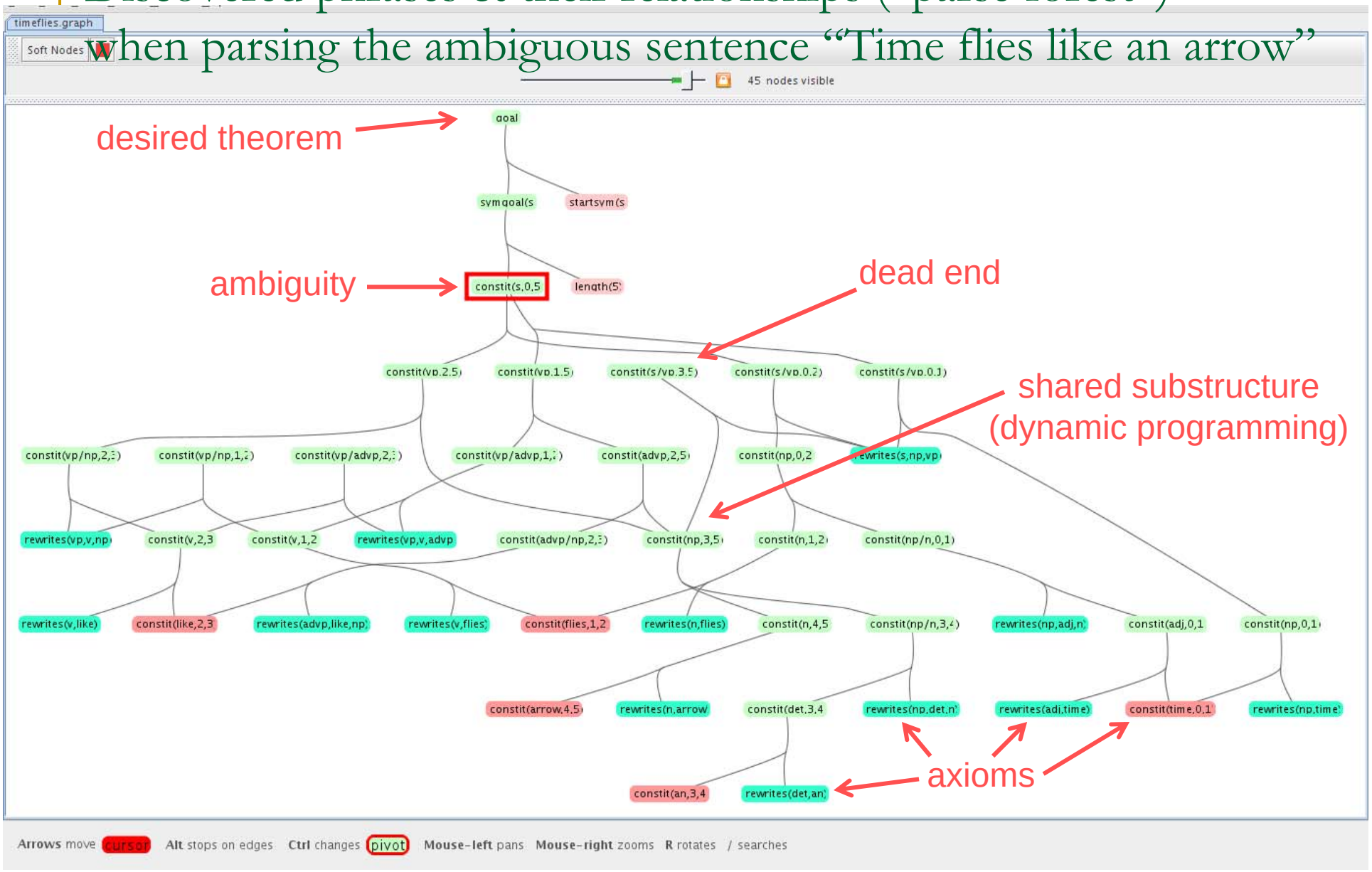
- The Dyna program runs fine.
- It nicely displays the abstract structure of the algorithm.

- But Dyna is a declarative programming language that hides the details of the actual execution from you.
- If you had to find the possible phrases by hand (or with a procedural programming language), what steps would you go through?

This picture assumes a slightly different version of the Dyna program, sorry

Discovered phrases & their relationships ("parse forest")

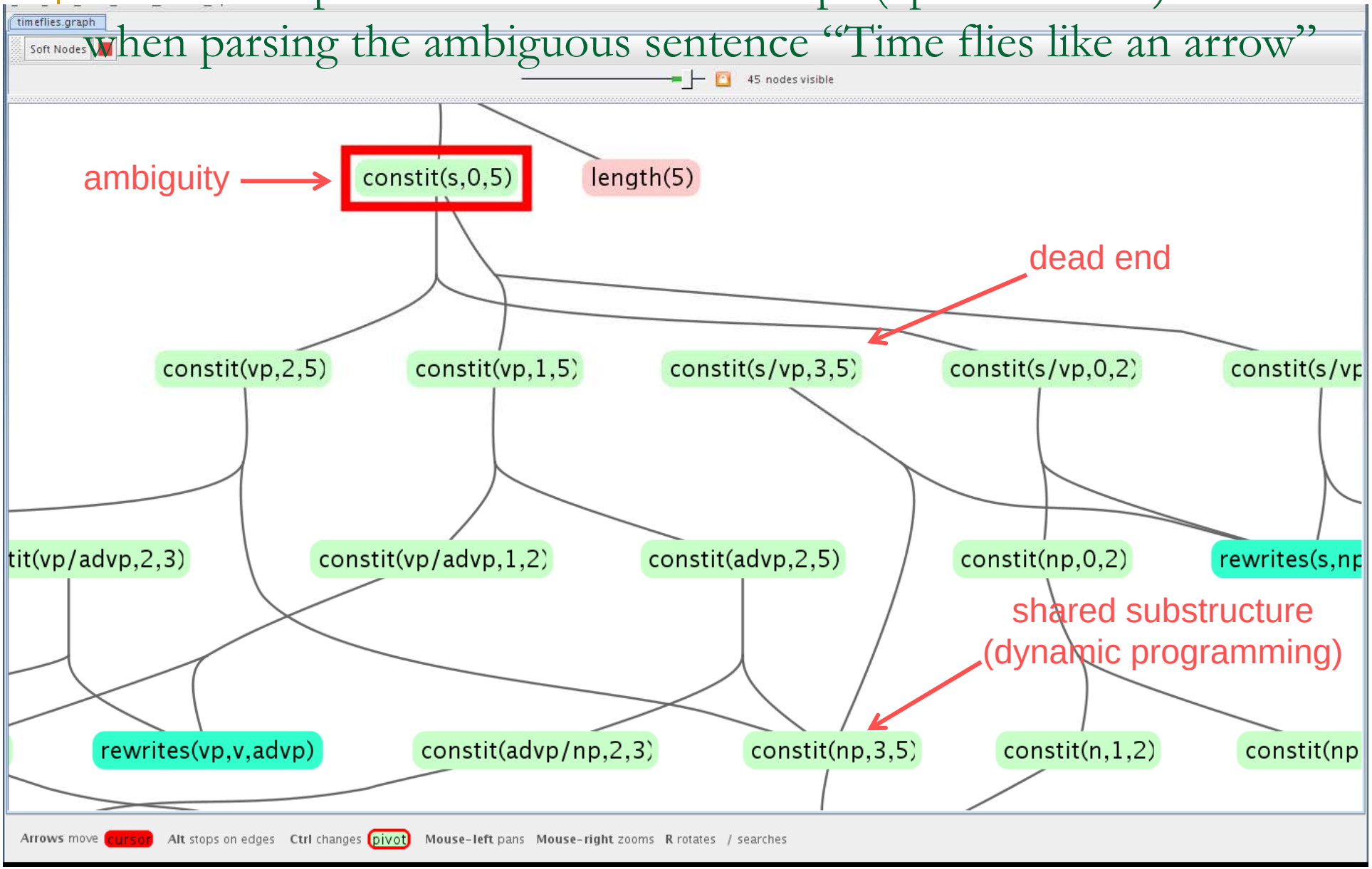
when parsing the ambiguous sentence "Time flies like an arrow"



This picture assumes a slightly different version of the Dyna program, sorry

Discovered phrases & their relationships ("parse forest")

when parsing the ambiguous sentence "Time flies like an arrow"



Procedural algorithms like CKY are just strategies for running this declarative program

```
phrase(X,I,J)  :-  rewrite(X,W), word(W,I,J).
phrase(X,I,J)  :-  rewrite(X,Y,Z), phrase(Y,I,Mid), phrase(Z,Mid,J).
goal           :-  phrase(start_symbol, 0, sentence_length).
```

- And we'll look at further such strategies later, e.g.
  - magic sets / Earley's algorithm / left-to-right parsing,
  - agenda-based forward chaining,
  - backward chaining with memoization,
  - coarse-to-fine search,
  - probability-guided search and pruning, …