4. Natural Language Processing

Features of Language That Make It Both Difficult and Useful

The Problem: English sentences are incomplete descriptions of the information that they are intended to convey:

Some dogs are outside.

$\downarrow$

Some dogs are on the lawn.
Three dogs are on the lawn.
Rover, Tripp, and Spot are on the lawn.

The Good Side: Language allows speakers to be as vague or as precise as they like. It also allows speakers to leave out things they believe their hearers already know.

Features of Language That Make It Both Difficult and Useful (con’t)

The Problem: The same expression means different things in different contexts:

Where’s the water?

- in a chemistry lab, it must be pure
- when you are thirsty, it must be potable
- dealing with a leaky roof, it can be filthy

The Good Side: Language lets us communicate about an infinite world using a finite (and thus learnable) number of symbols.

Features of Language That Make It Both Difficult and Useful (con’t)

The Problem: No natural language program can be complete because new words, expressions, and meanings can be generated quite freely:

I’ll fax it to you.

The Good Side: Language can evolve as the experiences that we want to communicate about evolve.

Features of Language That Make It Both Difficult and Useful (con’t)

The Problem: There are lots of ways to says the same things:

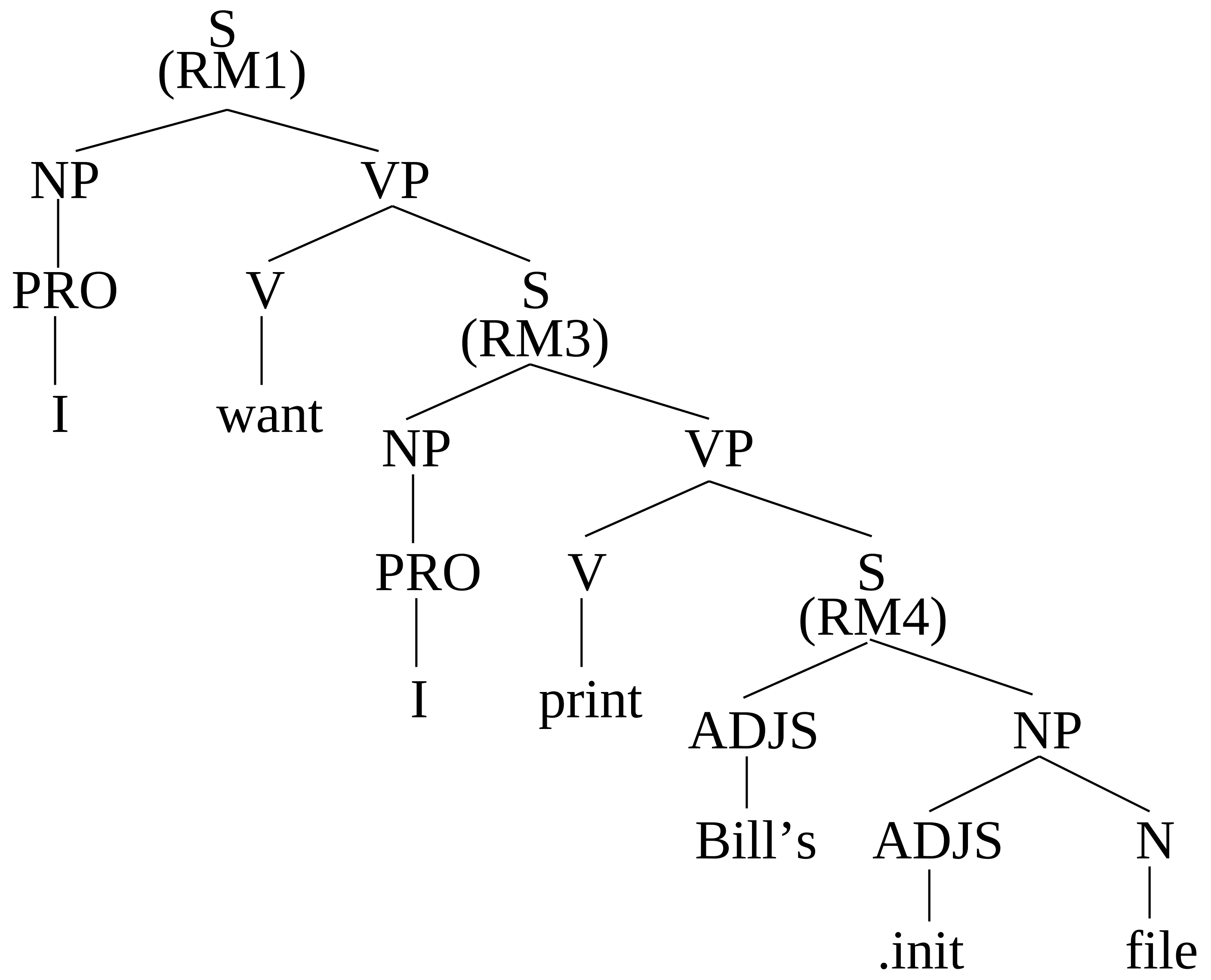
Mary was born on October 11.
Mary’s birthday is October 11.

The Good Side: When you know a lot, facts imply each other. Language is intended to be used by agents who know a lot.

Steps in Natural Language Understanding

- Morphological Analysis
  - individual words are analyzed into their components
- Syntactic Analysis
  - linear sequences of words are transformed into structures that show how words relate to each other
- Semantic Analysis
  - the structures are assigned meanings
- Discourse Integration
  - consider the meaning of a sentence by referring to the preceding sentences
- Pragmatic Analysis
  - reinterpret what was actually meant

The Result of Syntactic Analysis of “I want to print Bill’s .init file.”



A Knowledge Base Fragment

User

isa:	Person
*login-name:	must be <string>

User068

instance:	User
login-name:	Susan-Black

User073

instance:	User
login-name:	Bill-Smith

F1

instance:	File-Struct
name:	stuff
extension:	.init
owner:	User073
in-directory:	/wsmith/

File-Struct

isa:	Information-Object

A Knowledge Base Fragment (con’t)

Printing	
isa:	Physical-Event
*agent:	must be <animate or program>
*object:	must be <information-object>
Wanting	
isa:	Mental-Event
*agent:	must be <animate>
*object:	must be <state or event>
Commanding	
isa:	Mental-Event
*agent:	must be <animate>
*performer:	must be <animate or program>
*object:	must be <or event>
This-System	
instance:	Program

The Result of Semantic Analysis of “I want to print Bill’s .init file.”

RM1		{the whole sentence}
instance:	Wanting	
agent:	RM2	{I}
object:	RM3	{a printing event}
RM2		{I}
RM3		{a printing event}
instance:	Printing	
agent:	RM2	{I}
object:	RM4	{Bill’s .init file}
RM4		{Bill’s .init file}
instance:	File-Struct	
extension:	.init	
owner:	RM5	{Bill}
RM5		{Bill}
instance:	Person	
first-name:	Bill	

The Result of Pragmatic Analysis of "I want to print Bill's .init file."

Meaning

instance:	Commanding
agent:	User068
performer:	This-System
object:	P27

P27

instance:	Printing
agent:	This-System
object:	F1

This leads to the final result:

```
lpr   /wsmith/stuff.init
```

Syntactic Processing

- Role
  - Constrain the number of constituents that semantics can consider. Since syntax is cheaper than semantics, this is cost effective.
- Components
  - Grammar
  - Parser

A Simple Context-Free Grammar for a Fragment of English

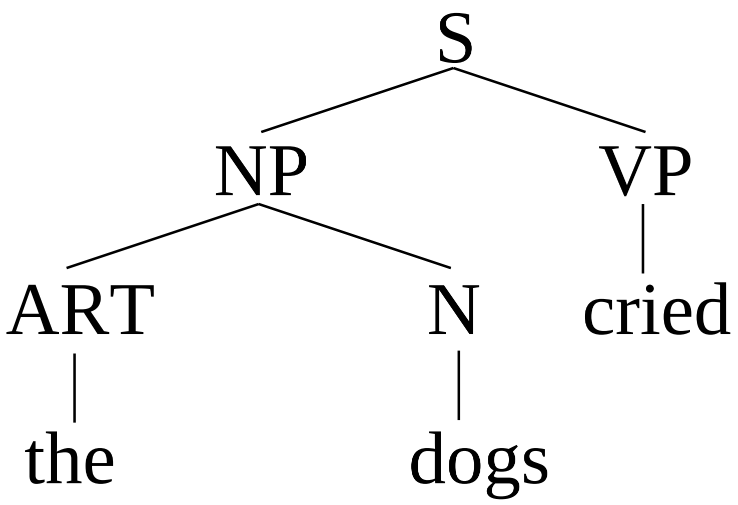
1. $S \rightarrow NP\ VP$
2. $NP \rightarrow ART\ N$
3. $NP \rightarrow ART\ ADJ\ N$
4. $VP \rightarrow V$
5. $VP \rightarrow V\ NP$

Grammar 1.

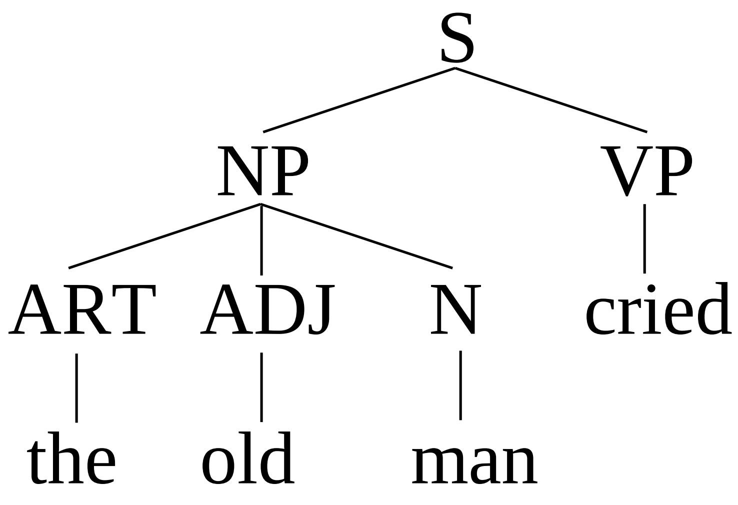
cried: V
dogs: N, V
the: ART
old: ADJ, N
man: N, V

Lexicon

A Parse Tree for Sentences



The tree for
“The dogs cried.”



The tree for
“The old man cried.”

A Top-Down Parser

A parsing algorithm –

uses grammatical rules to search for a combination that generates a tree describing the structure of the input sentence

A top-down parser –

starts with the S symbol and rewrite it into a sequence of terminal symbols that matches the words in the input sentence

A state of parse is a pair of:

- symbol list : the result of operations applied so far
- a number indicating the current position in the sentence

A Top-Down Parser (con’t)

For example, given the Grammar 1 and a sentence:

$_1$The $_2$ dogs $_3$ cried $_4$

a typical parse state would be:

((N VP) 2)

Since the word “dogs” is listed as an N in the lexicon, the next parse state would be:

((VP) 3)

If the first symbol is a nonterminal, like VP, then it is rewritten using a grammar rule.

- Using rule 4 in Grammar 1: the next state is ((V) 3)
- Using rule 5 in Grammar 1: the next state is ((V NP) 3)

A Simple Top-Down Parsing Algorithm

- The algorithm uses a list of possible states:
  - the current state (the first element of the list)
  - backup states (the remaining elements)
- For example, a list of possible states:

(((V) 3) ((V NP) 3) ((ART ADJ N VP) 1))

indicates that the current state consists of the symbol list (V) at position 3, and that there are two possible backup states.

A Simple Top-Down Parsing Algorithm (con't)

The algorithm starts with the initial state ((S) 1) and no backup states.

```
1. Take the first state off the possibilities list and call it C.
    IF the list is empty, THEN fails
2. IF C consists of an empty symbol list and the word
   position is at the end of the sentence, THEN succeeds
3. OTHERWISE, generate the next possible states.
   3.1 IF the first symbol of C is a lexical symbol, AND
            the next word in the sentence can be in that class,
       THEN   – create a new state by removing the first symbol
              – updating the word position
              – add it to the possibilities list.
   3.2 OTHERWISE, IF the first symbol of C is a non-terminal
       THEN   – generate a new state for each rule that can rewrite
                that non-terminal symbol
              – add them all to the possibilities list
```

Top-Down Depth-First Parse of "$_1$ *The* $_2$ *dogs* $_3$ *cried* $_4$"

Step	Current State	Backup States	Comment
1.	((S) 1)		initial position
2.	((NP VP) 1)		rewrite S by rule 1
3.	((ART N VP) 1)		rewrite NP by rules 2&3
		((ART ADJ N VP) 1)	
4.	((N VP) 2)		match ART with *the*
		((ART ADJ N VP) 1)	
5.	((VP) 3)		match N with *dogs*
		((ART ADJ N VP) 1)	
6.	((V) 3)		rewrite VP by rules 4&5
		((V NP) 3)	
		((ART ADJ N VP) 1)	
7.	( )		the parse succeeds as V is matched to *cried*

Top-Down Depth-First Parse of "$_1$ *The* $_2$ *old* $_3$ *man* $_4$ *cried* $_5$"

Step	Current State	Backup States	Comment
1.	((S) 1)		initial position
2.	((NP VP) 1)		S rewritten to NP VP
3.	((ART N VP) 1)		NP rewritten by rules 2&3
		((ART ADJ N VP) 1)	
4.	((N VP) 2)		
		((ART ADJ N VP) 1)	
5.	((VP) 3)		
		((ART ADJ N VP) 1)	
6.	((V) 3)		VP rewritten by rules 4&5
		((V NP) 3)	
		((ART ADJ N VP) 1)	
7.	(( ) 4)		
		((V NP) 3)	
		((ART ADJ N VP) 1)	
8.	((V NP) 3)		the first backup is chosen
		((ART ADJ N VP) 1)	

Top-Down Depth-First Parse of “$_1$ *The* $_2$ *old* $_3$ *man* $_4$ *cried* $_5$” (con’t)

Step	Current State	Backup States	Comment
9.	((NP) 4)		
		((ART ADJ N VP) 1)	
10.	((ART N) 4)		looking for ART fails
		((ART ADJ N) 4)	
		((ART ADJ N VP) 1)	
11.	((ART ADJ N) 4)		fails again
		((ART ADJ N VP) 1)	
12.	((ART ADJ N VP) 1)		exploring backup state saved in step 3
13.	((ADJ N VP) 2)		
14.	((N VP) 3)		
15.	((VP) 4)		
16.	((V) 4)		
		((V NP) 4)	
17.	(( ) 5)		success!

A Bottom-Up Chart Parser

- The basic operation in bottom-up parsing is to take a sequence of symbols and match it to the right-hand side of the rules.
- A very simple bottom-up parser
  - formulate the matching process as search
  - the state would consist of a symbol list, starting with the words in the sentence
  - successor states could be generated by exploring all possible ways to :

    -- rewrite a word by its possible lexical categories

    -- replace a sequence of symbols that matches the right-hand side of the rule by its left-hand side symbol

A Bottom-Up Chart Parser (con’t)

A more efficient algorithm : Chart parser

- uses *chart* that allows the parser to store the partial results of the matching it has done so far
- uses *key* for matching by looking for
  - -- rules that start with the key
  - -- rules that have already been started by earlier keys and require the present key either to complete the rule or to extend the rule

A Bottom-Up Chart Parser (con't)

$$\left.\begin{array}{l}
1.\ S \rightarrow NP\ VP \\
2.\ NP \rightarrow ART\ ADJ\ N \\
3.\ NP \rightarrow ART\ N \\
4.\ NP \rightarrow ADJ\ N \\
5.\ VP \rightarrow AUX\ VP \\
6.\ VP \rightarrow V\ NP
\end{array}\right\}\ \underline{\text{Grammar 2.}}$$

- Example: given Grammar 2 and a sentence that starts with ART, the parser uses ART to matched with rules 2 and 3.

  -- use $\circ$ to indicate what has been seen so far

  -- thus,

$$2'.\ NP \rightarrow ART \circ ADJ\ N$$
$$3'.\ NP \rightarrow ART \circ N$$

A Bottom-Up Chart Parser (con't)

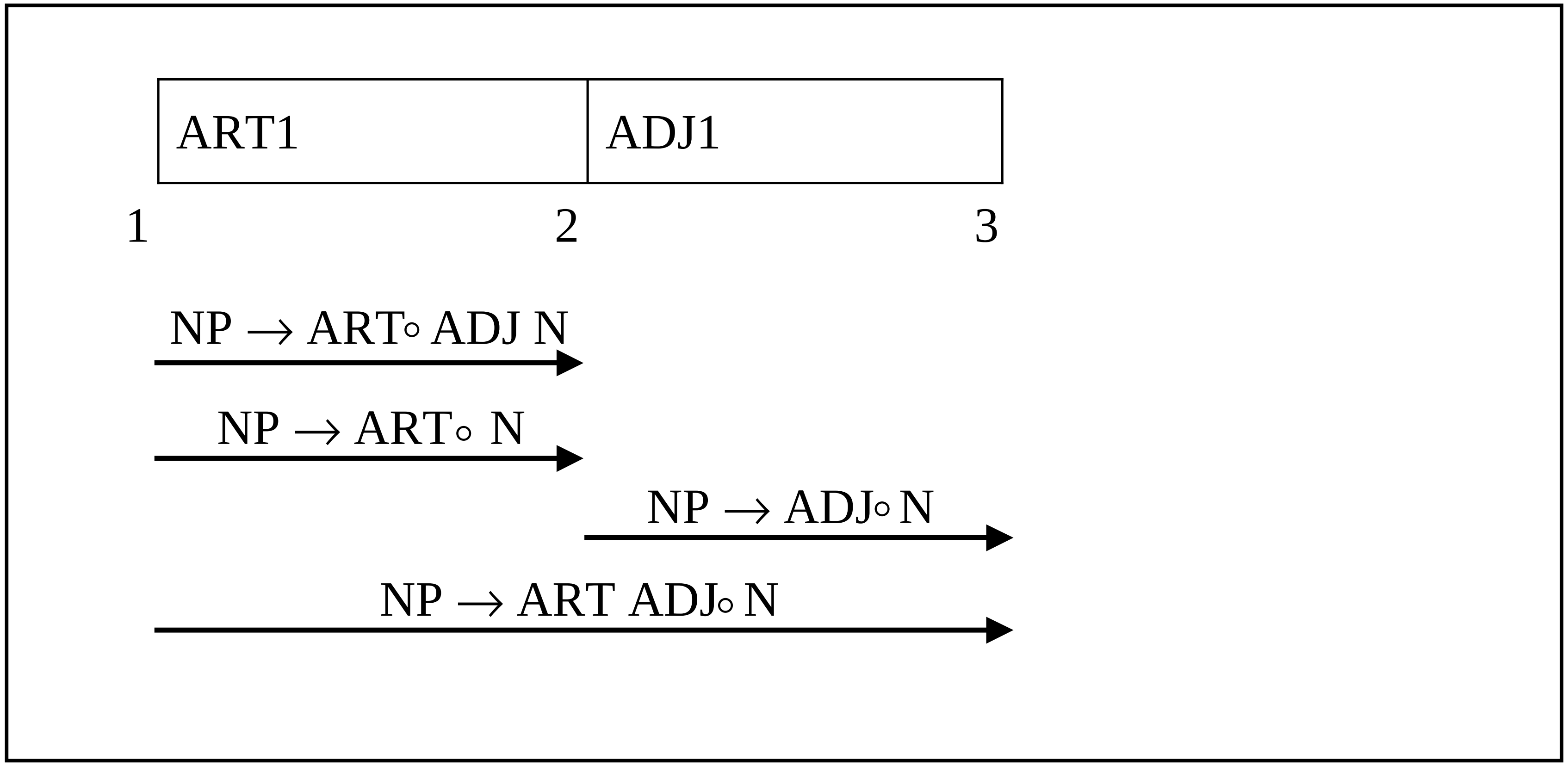
- If the next input key is an ADJ, then rule 4 may be started, and the modified rule 2′ may be extended to:

  2″. NP $\rightarrow$ ART ADJ $\circ$ N

- The chart maintains the record of
  -- all the constituents (such as ART, NP) derived from the sentence so far
  -- "*active arcs*" (rules that have matched partially but are not complete)

A Bottom-Up Chart Parser (con’t)

- The chart after see an ADJ in position 2.



A Bottom-Up Chart Parser (con’t)

- The basic operation of a chart parser involves combining an active arc with a completed constituent.
- The result is
  - -- a new completed constituent, or
  - -- a new active arc that is an extension of the original arc
  - -- new completed constituents are maintained on a list called *agenda* until they are added into the chart
- The process of extension of the arc is shown in the next figure.

The Arc Extension Algorithm

To add a constituent C from position p_1 to p_2:

1. Insert C into the chart from position p_1 to p_2.
2. For any active arc of the form

   $$X \to X_1 \ldots \circ C \ldots X_n$$

   from position p_0 to p_1, add a new active arc

   $$X \to X_1 \ldots C \circ \ldots X_n$$

   from position p_0 to p_2.
3. For any active arc of the form

   $$X \to X_1 \ldots X_n \circ C$$

   from position p_0 to p_1, add a new constituent of type X from p_0 to p_2 to the agenda.

A Bottom-Up Chart Parsing Algorithm

```
Do until there is no input left:
1.  If the agenda is empty, look up the interpretations
    for the next word in the input and add them to the
    agenda.
2.  Select a constituent C from the agenda.
    (Let C is from position $p_1$ to $p_2$.)
3.  For each rule in the grammar of form
            $X \rightarrow C\ X_1 \ldots X_n$
    add an active arc of the form
            $X \rightarrow C \circ X_1 \ldots X_n$
    from position $p_1$ to $p_2$.
4.  Add C to the chart using the arc extension algoritm.
```

An Example of Chart Parsing Algorithm

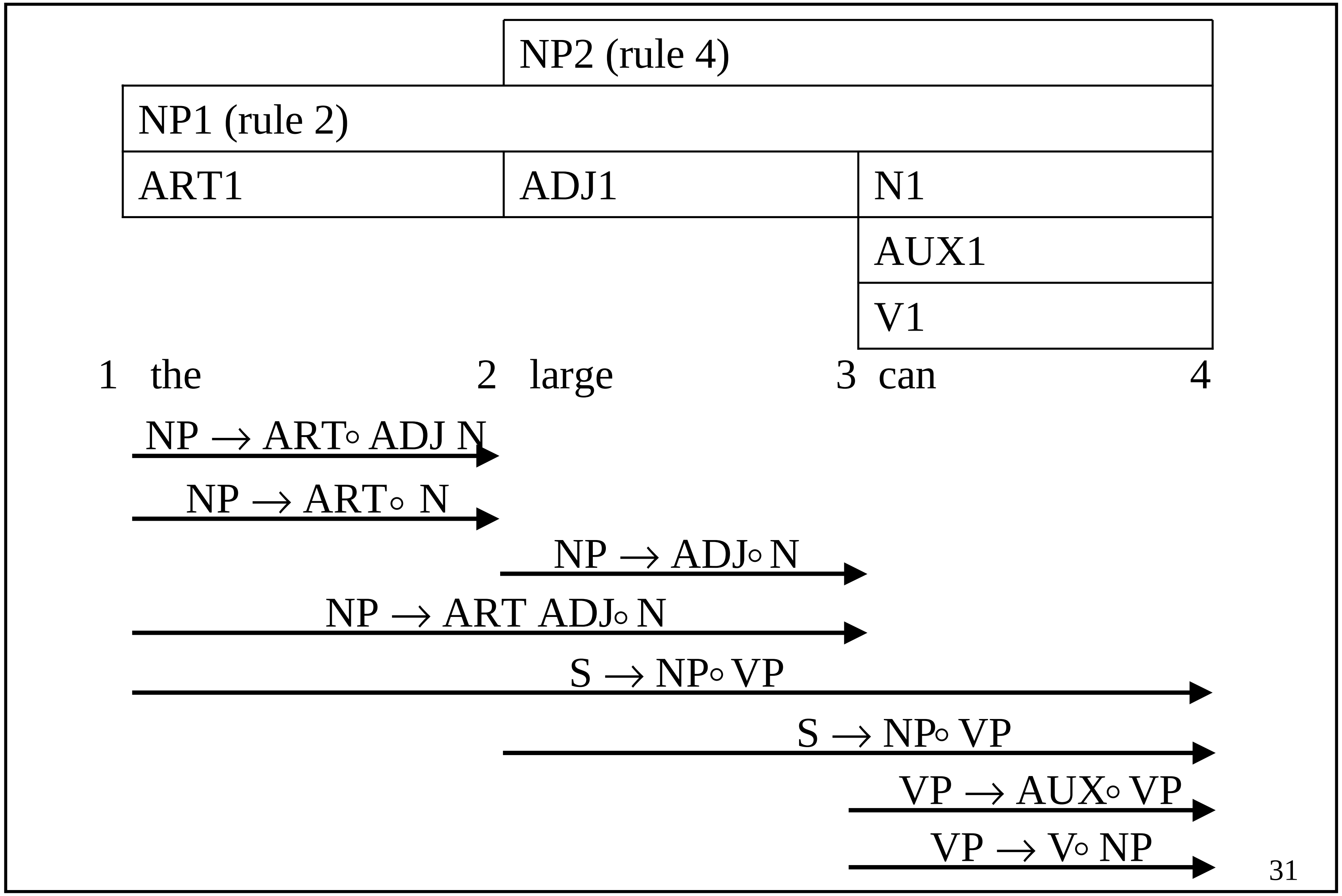
- Consider using the algorithm on the sentence

“*The large can can hold the water*”

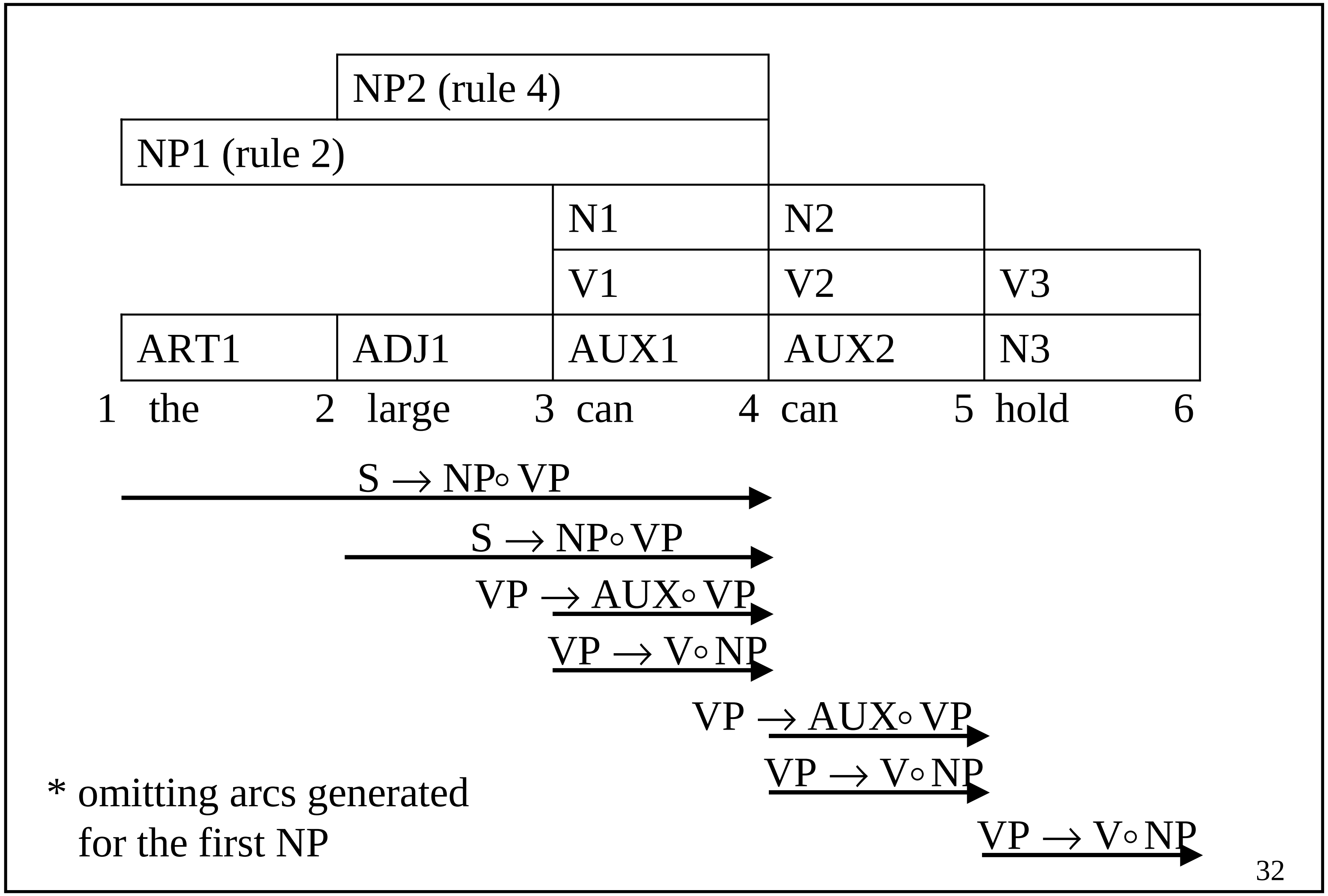
using the following Grammar 2 and following lexicon:

the:	ART
large:	ADJ
can:	N, AUX, V
hold:	N, V
water:	N, V

An Example of Chart Parsing Algorithm (After parsing *the large can*)

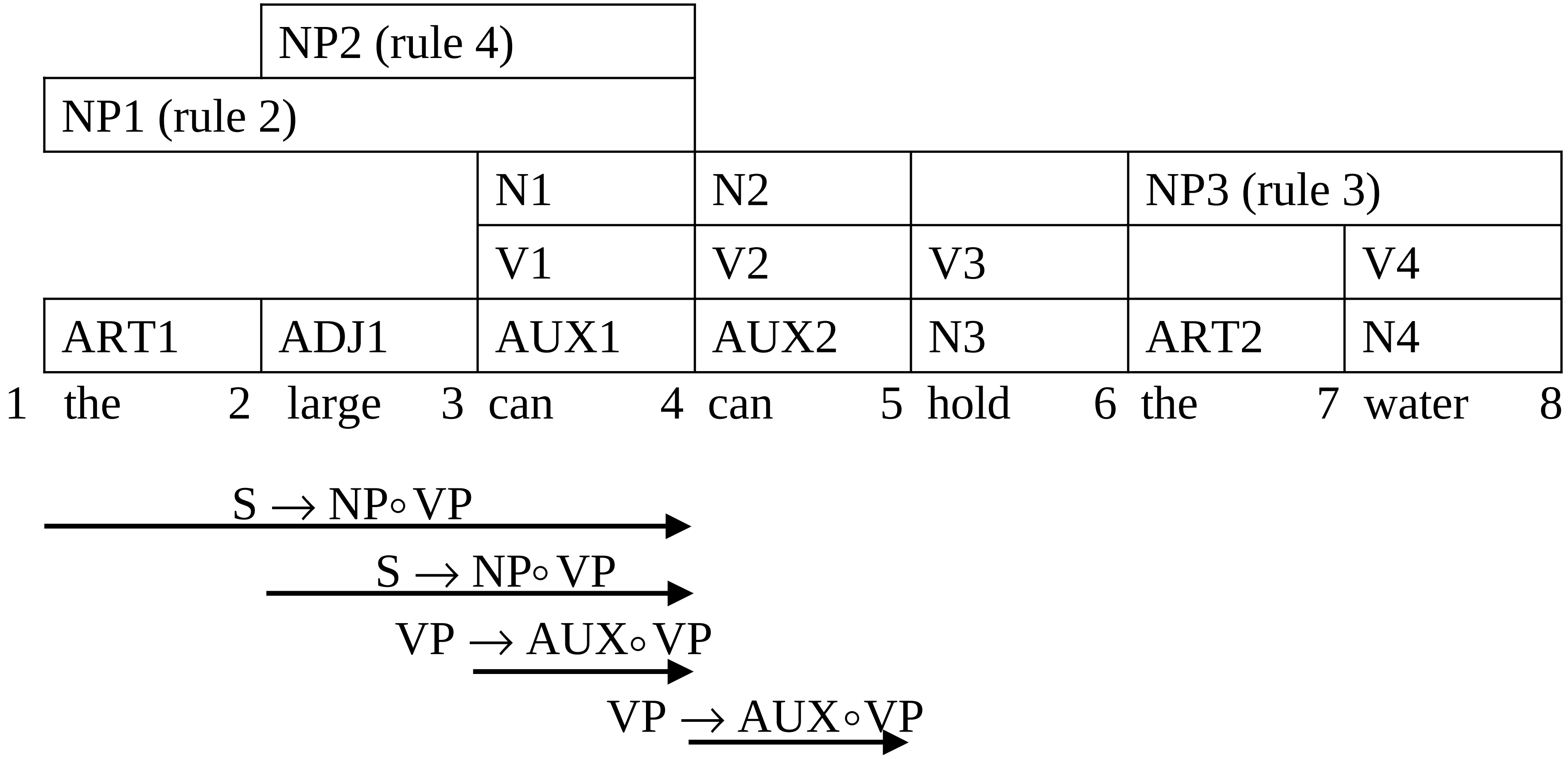


An Example of Chart Parsing Algorithm (After adding *hold*)



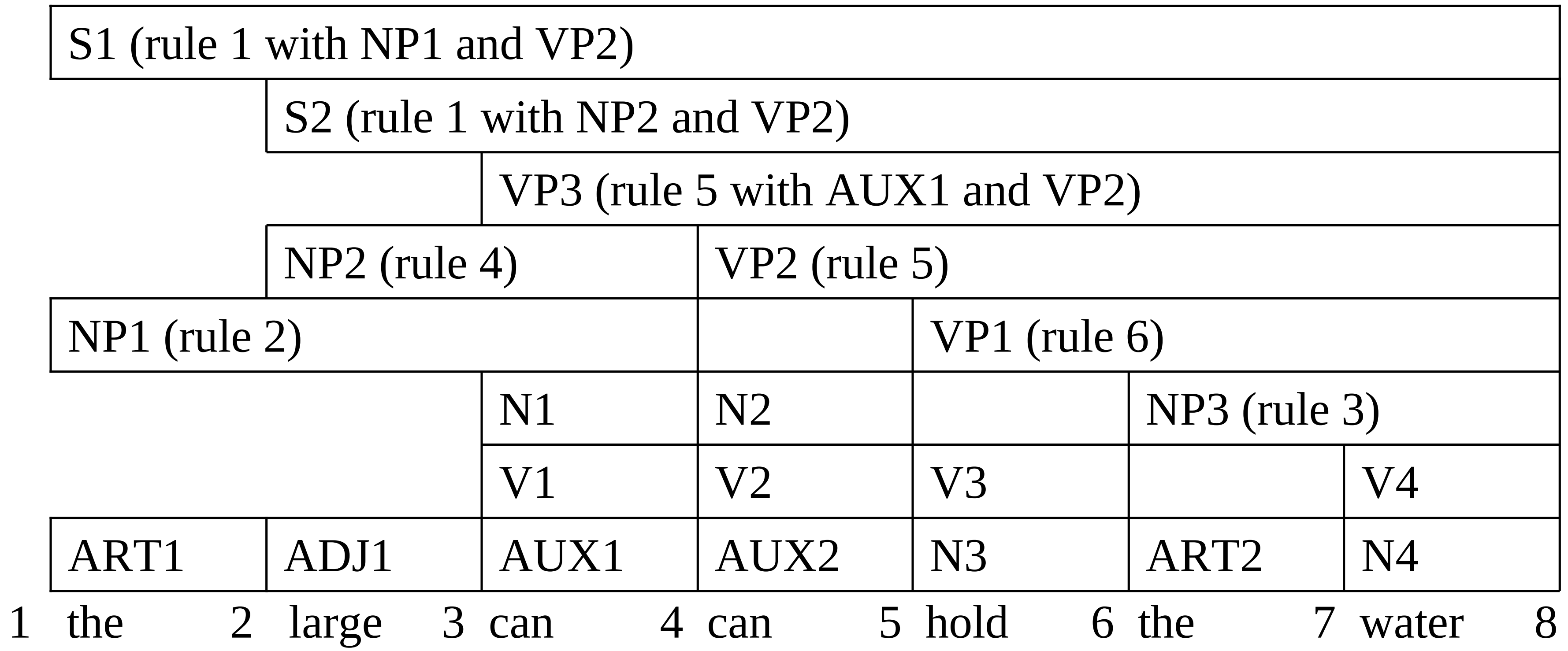
* omitting arcs generated for the first NP

An Example of Chart Parsing Algorithm (After all the NPs are found)



* omitting all but the crucial active arcs

An Example of Chart Parsing Algorithm (The final chart)



S1 (rule 1 with NP1 and VP2)
S2 (rule 1 with NP2 and VP2)
VP3 (rule 5 with AUX1 and VP2)
NP2 (rule 4)
VP2 (rule 5)
NP1 (rule 2)
VP1 (rule 6)
N1
N2
NP3 (rule 3)
V1
V2
V3
V4
ART1
ADJ1
AUX1
AUX2
N3
ART2
N4
1 the 2 large 3 can 4 can 5 hold 6 the 7 water 8

Transition Network Grammar

- Transition network grammar is another useful formalism.
- It consists of *nodes* and *labeled arcs* :

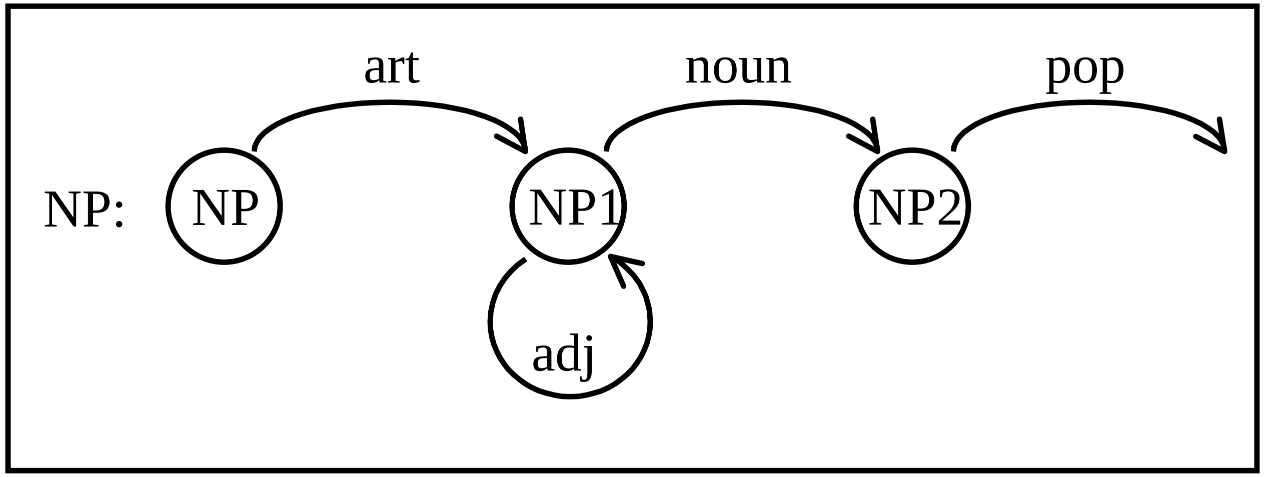
  -- initial or start state is the first node of the network

  -- start at the initial state, and traverse an arc if the current word in the sentence is in the category on arc

  -- if the arc is followed, the current word is updated to the next word

  -- a phrase is a legal one if there is path from the initial state to a *pop arc*

Transition Network Grammar (con't)



≡ NP → ART NP1
NP1 → ADJ NP1
NP1 → N

<u>transition network 1.</u>

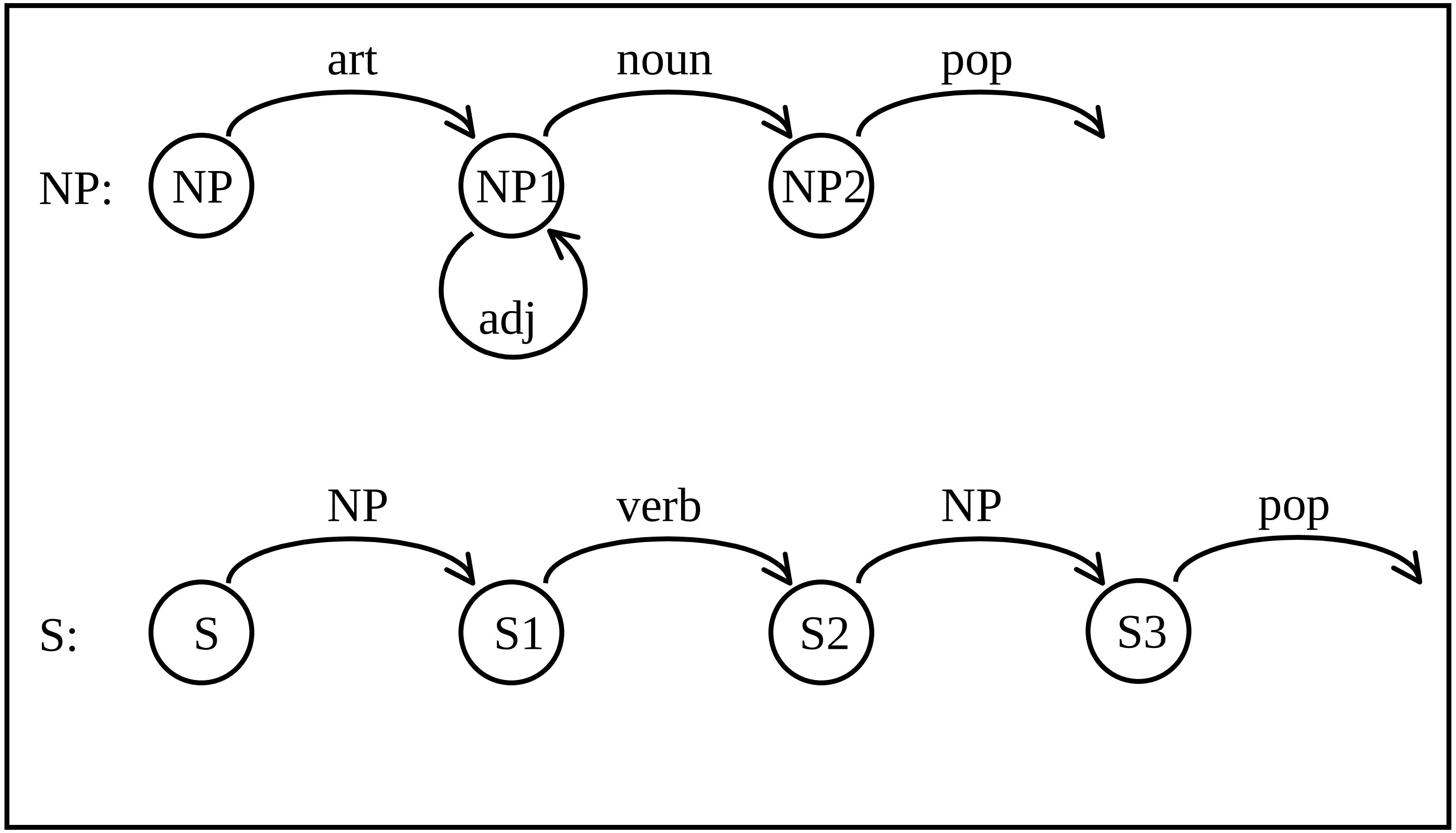
- The transition network 1 can be used to parse the NP "*a purple cow*".

Recursive Transition Network (RTN)

- A recursive transition network (RTN) is a transition network that allows arc labels to refer to other networks as well as word categories.
- RTN is more powerful than a simple transition network
- In a RTN,
  -- uppercase labels refer to networks
  -- lowercase labels refer to word categories.
- Transition network 2 shows an example of RTN that can be used to parse a sentence

“*The purple cow ate the grass*”.

An Example of RTN



transition network 2 (RTN)

Top-Down Parsing with RTN

- The algorithm for parsing with RTNs represents a parse state as follows:
  - -- current node : the node at which you are located in the network
  - -- current position : a pointer to the next word to be parse
  - -- return points : a stack of nodes in other networks where you will continue if you *pop* from the current network

Top-Down Parsing with RTN

- At each node, you can leave the current node and traverse an arc in the following cases:

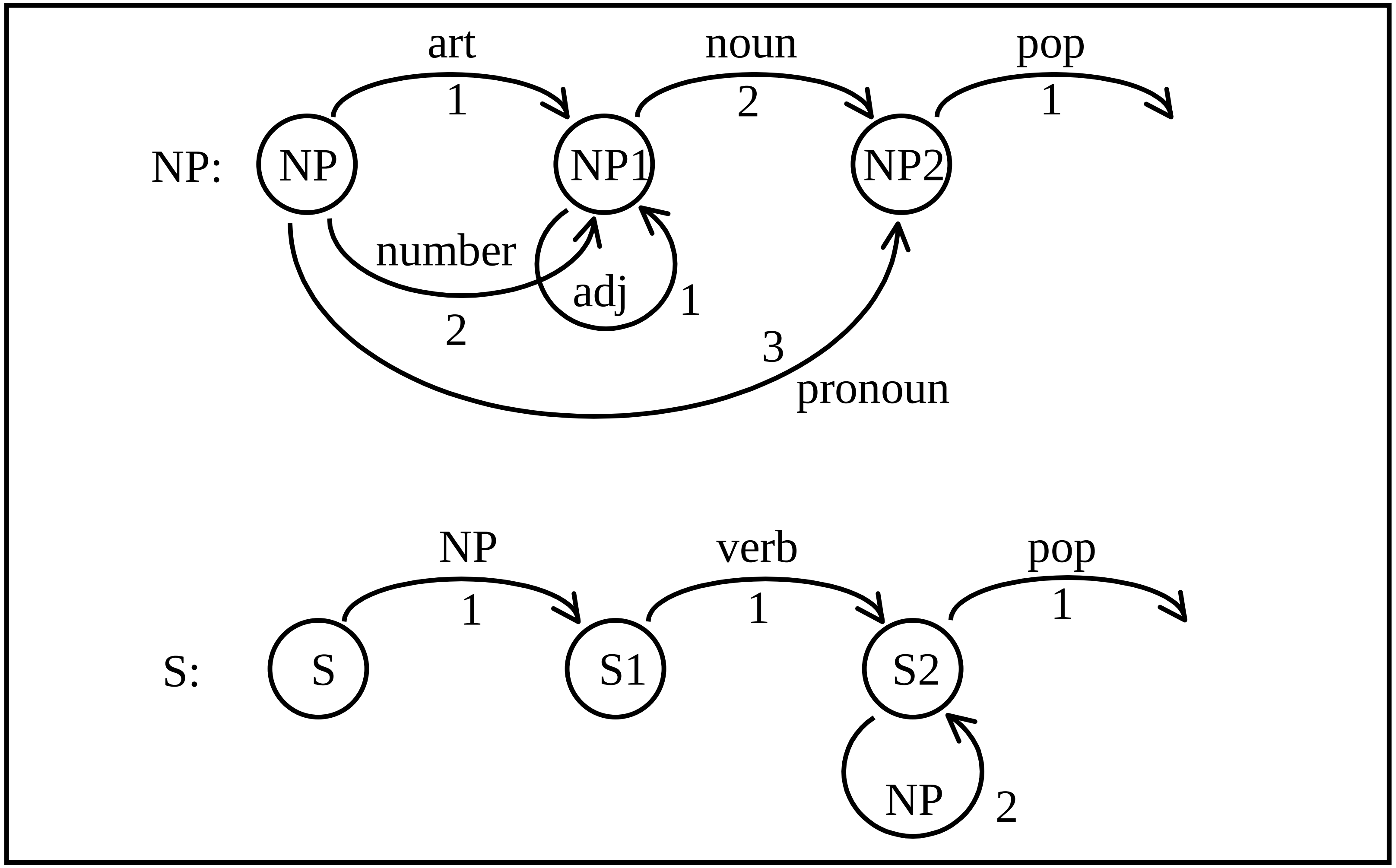
  Case 1: IF arc is word category and next word in the sentence is in that category,
  THEN (1) update *current position* to start at the next word
  (2) update *current node* to the destination of the arc.

  Case 2: IF arc is a push arc to a network N,
  THEN (1) add the destination of the arc onto return points;
  (2) update current node to the starting node in the network N.

  Case 3: IF arc is a pop arc and *return points* list is not empty,
  THEN (1) remove first return point and make it *current node*

  Case 4: IF arc is a pop arc, *return points* list is empty and there are no words left
  THEN (1) parse completes successfully

Another Example of RTN



transition network 3 (RTN)

A Trace Using Transition Network 3 of Parsing "$_1$ *The* $_2$ *old* $_3$ *man* $_4$ *cried* $_5$"

Step	Current Node	Current Position	Return Points	Arc to be Followed	Comments
1.	(S,	1,	NIL)	S/1	initial position
2.	(NP,	1,	(S1))	NP/1	followed push arc to NP network, to return to S1
3.	(NP1,	2,	(S1))	NP1/1	followed arc NP/1(*the*)
4.	(NP1,	3,	(S1))	NP1/2	followed arc NP1/1(*old*)
5.	(NP2,	4,	(S1))	NP2/1	followed arc NP1/2(man) since NP1/1 is not applicable
6.	(S1,	4,	NIL)	S1/1	the pop arc gets us back to S1
7.	(S2,	5,	NIL)	S2/1	followed arc S2/1(cried)
8.					Parse succeeds on pop arc from S2

A Top-Down RTN Parse with Backtracking for “$_1$ *One* $_2$ *saw* $_3$ *the* $_4$ *man* $_5$”

Step	**Current State**	**Arc to be followed**	**Backup States**
1.	(S,1,NIL)	S/1	NIL
2.	(NP,1,(S1))	NP/2(&NP/3 for backup)	NIL
3.	(NP1,2,(S1))	NP1/2	(NP2,2,(S1))
4.	(NP2,3,(S1))	NP2/1	(NP2,2,(S1))
5.	(S1,3,NIL)	no arc can be followed	(NP2,2,(S1))
6.	(NP2,2,(S1))	NP2/1	NIL
7.	(S1,2,NIL)	S1/1	NIL
8.	(S2,3,NIL)	S2/2	NIL
9.	(NP,3,(S2))	NP/1	NIL
10.	(NP1,4,(S2))	NP1/2	NIL
11.	(NP2,5,(S2))	NP2/1	NIL
12.	(S2,5,NIL)	S2/1	NIL
13.	Parse succeeds		