Parsing of Context-Free Grammars

Bernd Kiefer

{Bernd.Kiefer}@dfki.de

Deutsches Forschungszentrum für künstliche Intelligenz

- Assignment: Write a regular expression for fully bracketed arithmetic expressions!
- Answer: ?

- Assignment: Write a regular expression for fully bracketed arithmetic expressions!
- Answer:
  - ➤ This is not possible!
  - ➤ Regular expressions can only count *finite* amounts of brackets

Limits of Regular Expressions

- Assignment: Write a regular expression for fully bracketed arithmetic expressions!
- Answer:
  - ➢ This is not possible!
  - ➢ Regular expressions can only count *finite* amounts of brackets
  - ➢ We need a more powerful formal device: *context-free grammars*

Limits of Regular Expressions

- Assignment: Write a regular expression for fully bracketed arithmetic expressions!
- Answer:
  - ➢ This is not possible!
  - ➢ Regular expressions can only count *finite* amounts of brackets
  - ➢ We need a more powerful formal device: *context-free grammars*
  - ➢ Context-free grammars provide a (finite) inventory of named brackets
  - ➢ All regular languages are also context-free, i.e.: for every regular expression, there is a context-free grammar that accepts / derives the same language

A context-free grammar (CFG) consists of:

- The set of terminal symbols $\Sigma = a, b, c, \ldots$ (the words or letters of the language)
- The set of non-terminal symbols $N = A, B, C, \ldots$
- The startsymbol $S \in N$
- The set of productions (rules) P, where $P \ni r = A \rightarrow \alpha$ with $\alpha \in (\Sigma \cup N)^*$ (we use greek letters for strings of $\Sigma \cup N$)

Example: A grammar for arithmetic expressions:

$$\Sigma = \{\texttt{int}, \texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}\} \quad , \quad N = \{E\} \quad , \quad S = E$$

$$P = \{ \; E \rightarrow E\texttt{+}E, \;\; E \rightarrow E\texttt{*}E, \;\; E \rightarrow \texttt{(}E\texttt{)}, \;\; E \rightarrow \texttt{int} \; \}$$

The Language of a CFG

- Given a CFG G, the language $\mathcal{L}(G)$ is defined as the set of all strings that can be *derived* from S
- Given a string α from $(\Sigma \cup N)^*$, derive a new string β:
  - ➤ Choose one of the nonterminals in α, say, A
  - ➤ Choose one of the productions with A on the left hand side
  - ➤ Replace A in α with the right hand side (rhs) of the production to get the derived string β
- If α contains only symbols in Σ, then $\alpha \in \mathcal{L}(G)$
- Example:

  $\alpha =$ `int*`(E); choose $E \rightarrow E$ `+` E; $\beta =$ `int*(`E`+`E`)`

Derivations: Formally

- A string α derives a string β, ($\alpha \underset{G}{\Rightarrow} \beta$) $\alpha, \beta \in (\Sigma \cup N)^*$, if: there are $\gamma, \delta, \eta \in (\Sigma \cup N)^*, A \in N$ such that
  $\alpha = \gamma\, A\, \delta \quad , \quad \beta = \gamma\, \eta\, \delta$ and $A \longrightarrow \eta \in P$
- We write $\alpha \underset{G}{\Rightarrow} \beta$ for a one-step derivation
- $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ is a many-step derivation: $\alpha \underset{G}{\Rightarrow} \alpha_0 \underset{G}{\Rightarrow} \alpha_1 \ldots \underset{G}{\Rightarrow} \beta$
- Language $\mathcal{L}(G)$ generated by G: $\mathcal{L}(G) = \{s \in \Sigma^* | S \overset{*}{\underset{G}{\Rightarrow}} s\}$
- The task of a parser: find one (or all) derivation(s) of a string in Σ^*, given a CFG G

Derivations: Example

- $\Sigma = \{$*john, girl, car, saw, walks, in, the, a*$\}$
- $N = \{$S, NP, VP, PP, D, N, V, P$\}$
- $P = \left\{ \begin{array}{ll} \text{S} \rightarrow \text{NP VP} | \text{N VP} | \text{N V} | \text{NP V} & \text{N} \rightarrow \textit{john, girl, car} \\ \text{VP} \rightarrow \text{V NP} | \text{V N} | \text{VP PP} & \text{V} \rightarrow \textit{saw, walks} \\ \text{NP} \rightarrow \text{D N} | \text{NP PP} | \text{N PP} & \text{P} \rightarrow \textit{in} \\ \text{PP} \rightarrow \text{P NP} | \text{P N} & \text{D} \rightarrow \textit{the, a} \end{array} \right\}$

S $\underset{G}{\Rightarrow}$

john saw the girl in a car

Derivations: Example

- $\Sigma = \{$*john, girl, car, saw, walks, in, the, a*$\}$
- $N = \{$S, NP, VP, PP, D, N, V, P$\}$
- $P = \left\{ \begin{array}{ll} \text{S} \rightarrow \text{NP VP} | \text{N VP} | \text{N V} | \text{NP V} & \text{N} \rightarrow \textit{john, girl, car} \\ \text{VP} \rightarrow \text{V NP} | \text{V N} | \text{VP PP} & \text{V} \rightarrow \textit{saw, walks} \\ \text{NP} \rightarrow \text{D N} | \text{NP PP} | \text{N PP} & \text{P} \rightarrow \textit{in} \\ \text{PP} \rightarrow \text{P NP} | \text{P N} & \text{D} \rightarrow \textit{the, a} \end{array} \right\}$

$\text{S} \underset{G}{\Rightarrow} \text{N VP} \underset{G}{\Rightarrow}$

john saw the girl in a car

Derivations: Example

- $\Sigma = \{$*john, girl, car, saw, walks, in, the, a*$\}$
- $N = \{$S, NP, VP, PP, D, N, V, P$\}$
- $P = \left\{ \begin{array}{llll} \text{S} & \rightarrow \text{NP VP} | \text{N VP} | \text{N V} | \text{NP V} & \text{N} \rightarrow \textit{john, girl, car} \\ \text{VP} & \rightarrow \text{V NP} | \text{V N} | \text{VP PP} & \text{V} \rightarrow \textit{saw, walks} \\ \text{NP} & \rightarrow \text{D N} | \text{NP PP} | \text{N PP} & \text{P} \rightarrow \textit{in} \\ \text{PP} & \rightarrow \text{P NP} | \text{P N} & \text{D} \rightarrow \textit{the, a} \end{array} \right\}$

$\text{S} \underset{G}{\Rightarrow} \text{N VP} \underset{G}{\Rightarrow} \text{john VP} \underset{G}{\Rightarrow}$

john saw the girl in a car

Derivations: Example

- $\Sigma = \{$*john, girl, car, saw, walks, in, the, a*$\}$
- $N = \{$S, NP, VP, PP, D, N, V, P$\}$
- $P = \left\{ \begin{array}{ll} \text{S} \rightarrow \text{NP VP} | \text{N VP} | \text{N V} | \text{NP V} & \text{N} \rightarrow \textit{john, girl, car} \\ \text{VP} \rightarrow \text{V NP} | \text{V N} | \text{VP PP} & \text{V} \rightarrow \textit{saw, walks} \\ \text{NP} \rightarrow \text{D N} | \text{NP PP} | \text{N PP} & \text{P} \rightarrow \textit{in} \\ \text{PP} \rightarrow \text{P NP} | \text{P N} & \text{D} \rightarrow \textit{the, a} \end{array} \right\}$

$$\text{S} \underset{G}{\Rightarrow} \text{N VP} \underset{G}{\Rightarrow} \text{john VP} \underset{G}{\Rightarrow} \text{john V NP} \underset{G}{\Rightarrow}$$

john saw the girl in a car

Derivations: Example

- $\Sigma = \{$*john, girl, car, saw, walks, in, the, a*$\}$
- $N = \{$S, NP, VP, PP, D, N, V, P$\}$
- $P = \left\{ \begin{array}{ll} \text{S} \rightarrow \text{NP VP} | \text{N VP} | \text{N V} | \text{NP V} & \text{N} \rightarrow \textit{john, girl, car} \\ \text{VP} \rightarrow \text{V NP} | \text{V N} | \text{VP PP} & \text{V} \rightarrow \textit{saw, walks} \\ \text{NP} \rightarrow \text{D N} | \text{NP PP} | \text{N PP} & \text{P} \rightarrow \textit{in} \\ \text{PP} \rightarrow \text{P NP} | \text{P N} & \text{D} \rightarrow \textit{the, a} \end{array} \right\}$

$$\text{S} \underset{G}{\Rightarrow} \text{N VP} \underset{G}{\Rightarrow} \text{john VP} \underset{G}{\Rightarrow} \text{john V NP} \underset{G}{\Rightarrow} \text{john saw NP} \underset{G}{\Rightarrow}$$

john saw the girl in a car

Derivations: Example

- $\Sigma = \{$*john, girl, car, saw, walks, in, the, a*$\}$
- $N = \{$S, NP, VP, PP, D, N, V, P$\}$
- $P = \left\{ \begin{array}{ll} \text{S} \rightarrow \text{NP VP} | \text{N VP} | \text{N V} | \text{NP V} & \text{N} \rightarrow \textit{john, girl, car} \\ \text{VP} \rightarrow \text{V NP} | \text{V N} | \text{VP PP} & \text{V} \rightarrow \textit{saw, walks} \\ \text{NP} \rightarrow \text{D N} | \text{NP PP} | \text{N PP} & \text{P} \rightarrow \textit{in} \\ \text{PP} \rightarrow \text{P NP} | \text{P N} & \text{D} \rightarrow \textit{the, a} \end{array} \right\}$

S $\underset{G}{\Rightarrow}$ N VP $\underset{G}{\Rightarrow}$ john VP $\underset{G}{\Rightarrow}$ john V NP $\underset{G}{\Rightarrow}$ john saw NP $\underset{G}{\Rightarrow}$
john saw NP PP $\underset{G}{\Rightarrow}$

john saw the girl in a car

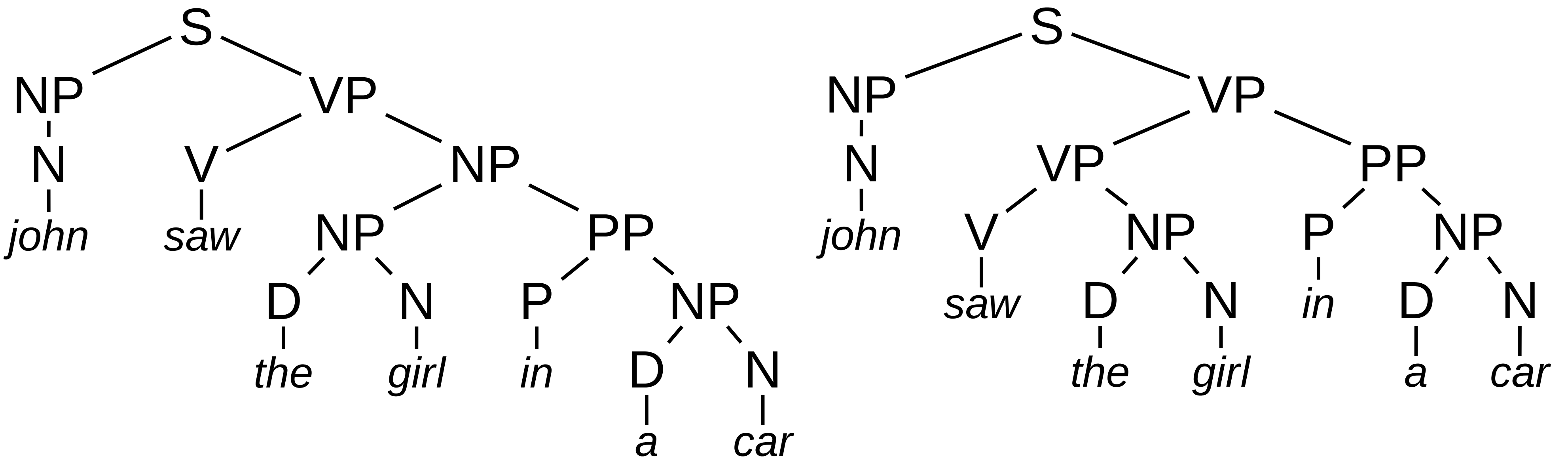
Derivations: Example

- $\Sigma = \{$*john, girl, car, saw, walks, in, the, a*$\}$
- $N = \{$S, NP, VP, PP, D, N, V, P$\}$
- $P = \left\{ \begin{array}{ll} \text{S} \rightarrow \text{NP VP} | \text{N VP} | \text{N V} | \text{NP V} & \text{N} \rightarrow \textit{john, girl, car} \\ \text{VP} \rightarrow \text{V NP} | \text{V N} | \text{VP PP} & \text{V} \rightarrow \textit{saw, walks} \\ \text{NP} \rightarrow \text{D N} | \text{NP PP} | \text{N PP} & \text{P} \rightarrow \textit{in} \\ \text{PP} \rightarrow \text{P NP} | \text{P N} & \text{D} \rightarrow \textit{the, a} \end{array} \right\}$

S $\underset{G}{\Rightarrow}$ N VP $\underset{G}{\Rightarrow}$ john VP $\underset{G}{\Rightarrow}$ john V NP $\underset{G}{\Rightarrow}$ john saw NP $\underset{G}{\Rightarrow}$ john saw NP PP $\underset{G}{\Rightarrow}$ john saw D N PP $\underset{G}{\Rightarrow}$ john saw the N PP $\underset{G}{\Rightarrow}$ john saw the girl PP $\underset{G}{\Rightarrow}$ john saw the girl P NP $\underset{G}{\Rightarrow}$ john saw the girl in NP $\underset{G}{\Rightarrow}$ john saw the girl in D N $\underset{G}{\Rightarrow}$ john saw the girl in a N $\underset{G}{\Rightarrow}$

john saw the girl in a car

Derivation (Parse) Trees



- Encodes many possible derivations
- PP node in the example can be attached to two nodes: the grammar is ambigous
- CF Parsers/Recognizers differ in the way the derivation trees are build

Task: given $s \in \Sigma^*$ and G, is $s \in \mathcal{L}(G)$?

Two ways to go:

- start with the start symbol S and try to derive s by systematic application of the productions:
  top down recognition (goal driven)
- start with the string s and try to reduce it to the start symbol:
  bottom up recognition (data driven)

Idea: Recursively compute all expansions of a nonterminal at some input position

$E \rightarrow S$+S, $E \rightarrow S$ * S,
$S \rightarrow$ (E), $S \rightarrow$ int

expand(S, 0)

S

(int + (int * int))

Idea: Recursively compute all expansions of a nonterminal at some input position

expand(S, 0)

$E \longrightarrow S\texttt{+}S, \quad E \longrightarrow S\,\texttt{*}\,S,$

$S \longrightarrow \texttt{(}E\texttt{)}, \quad S \longrightarrow \texttt{int}$

S

(E)

(int + (int * int))

Idea: Recursively compute all expansions of a nonterminal at some input position

$$E \rightarrow S\texttt{+}S, \quad E \rightarrow S\,\texttt{*}\,S,$$
$$S \rightarrow \texttt{(}E\texttt{)}, \quad S \rightarrow \texttt{int}$$

`expand(`E`, 1)`

S

`(` E `)`

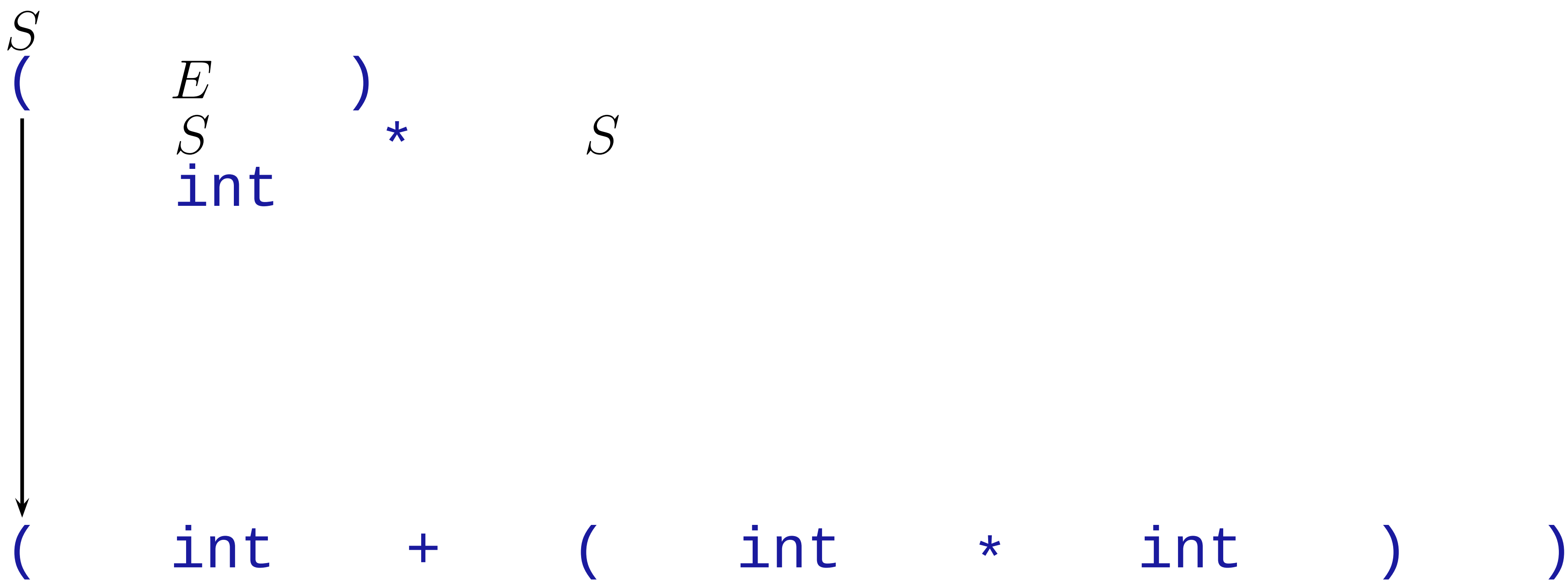
S `*` S

`( int + ( int * int ) )`

Idea: Recursively compute all expansions of a nonterminal at some input position

expand(S, 1)

$E \rightarrow S$ + S, $E \rightarrow S$ * S,
$S \rightarrow$ (E), $S \rightarrow$ int

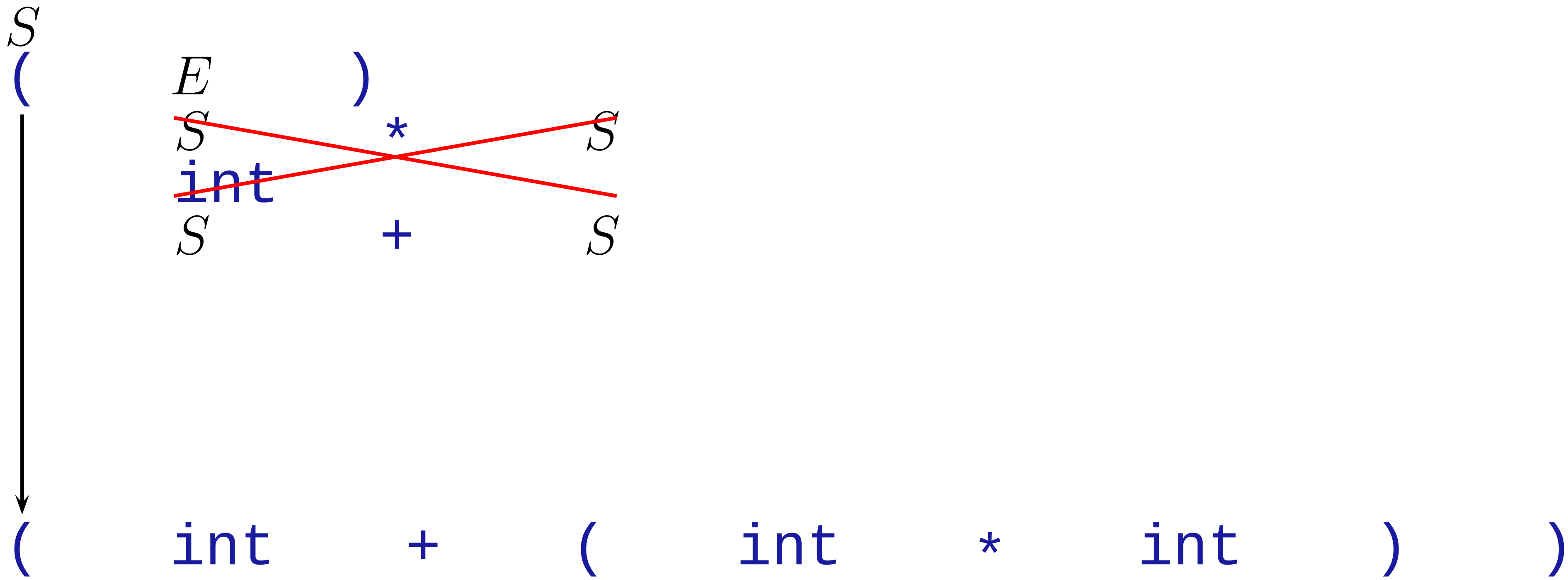


Recursive Descent Parsing

Idea: Recursively compute all expansions of a nonterminal at some input position

`expand(`E`, 1)`

$E \rightarrow S\texttt{+}S, \quad E \rightarrow S\texttt{*}S,$
$S \rightarrow \texttt{(}E\texttt{)}, \quad S \rightarrow \texttt{int}$

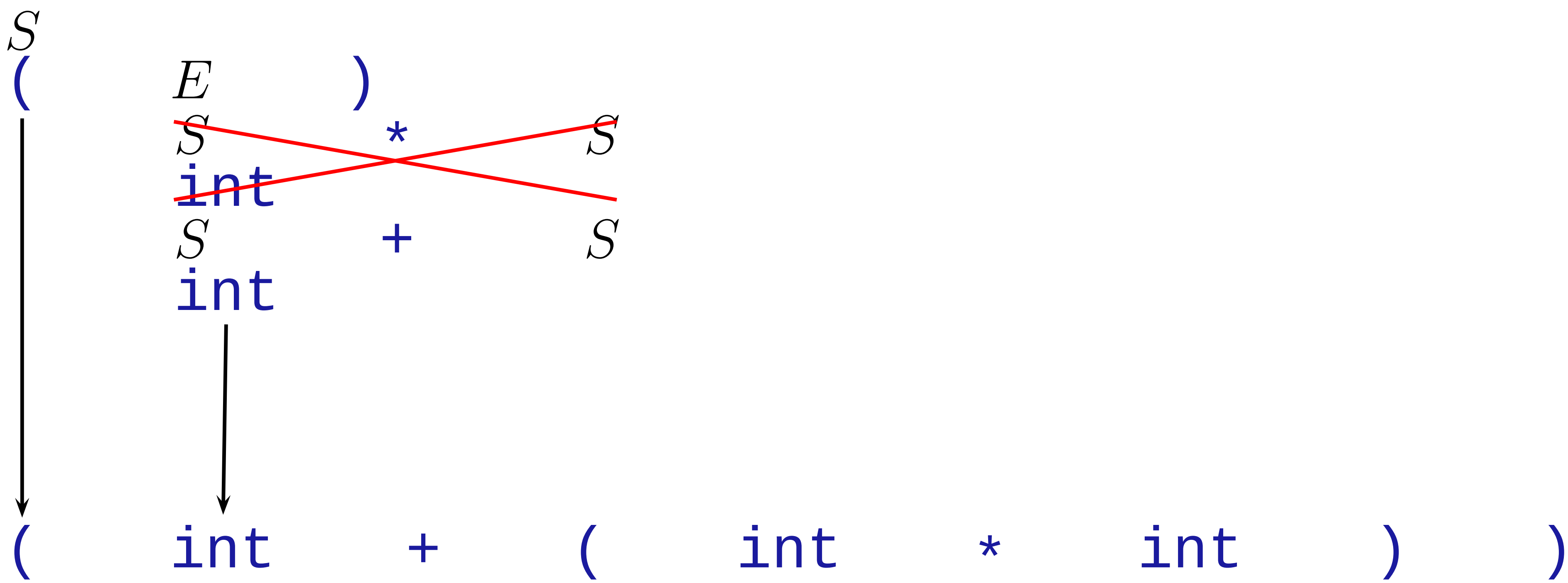


Recursive Descent Parsing

Idea: Recursively compute all expansions of a nonterminal at some input position

$E \rightarrow S\texttt{+}S, \quad E \rightarrow S \texttt{*} S,$
$S \rightarrow \texttt{(}E\texttt{)}, \quad S \rightarrow \texttt{int}$

`expand(`S`, 1)`

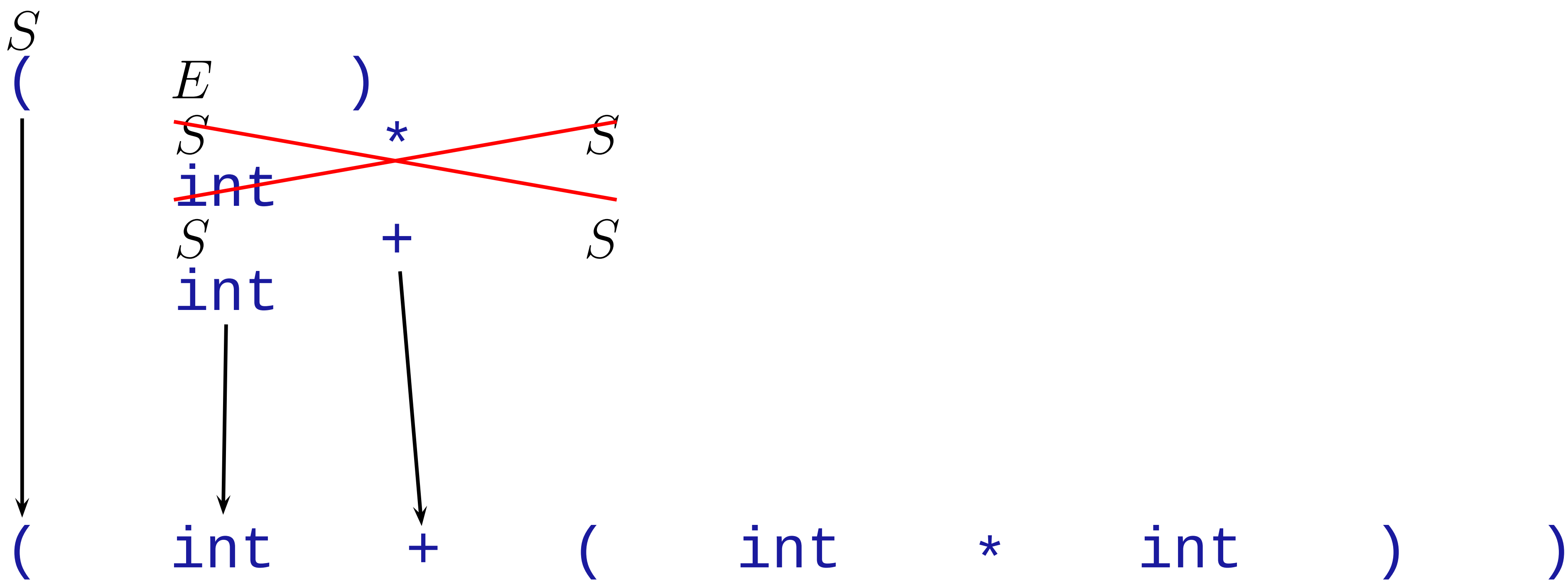


Recursive Descent Parsing

Idea: Recursively compute all expansions of a nonterminal at some input position

expand(S, 1)

$E \longrightarrow S$+S, $E \longrightarrow S$ * S,
$S \longrightarrow$ (E), $S \longrightarrow$ int

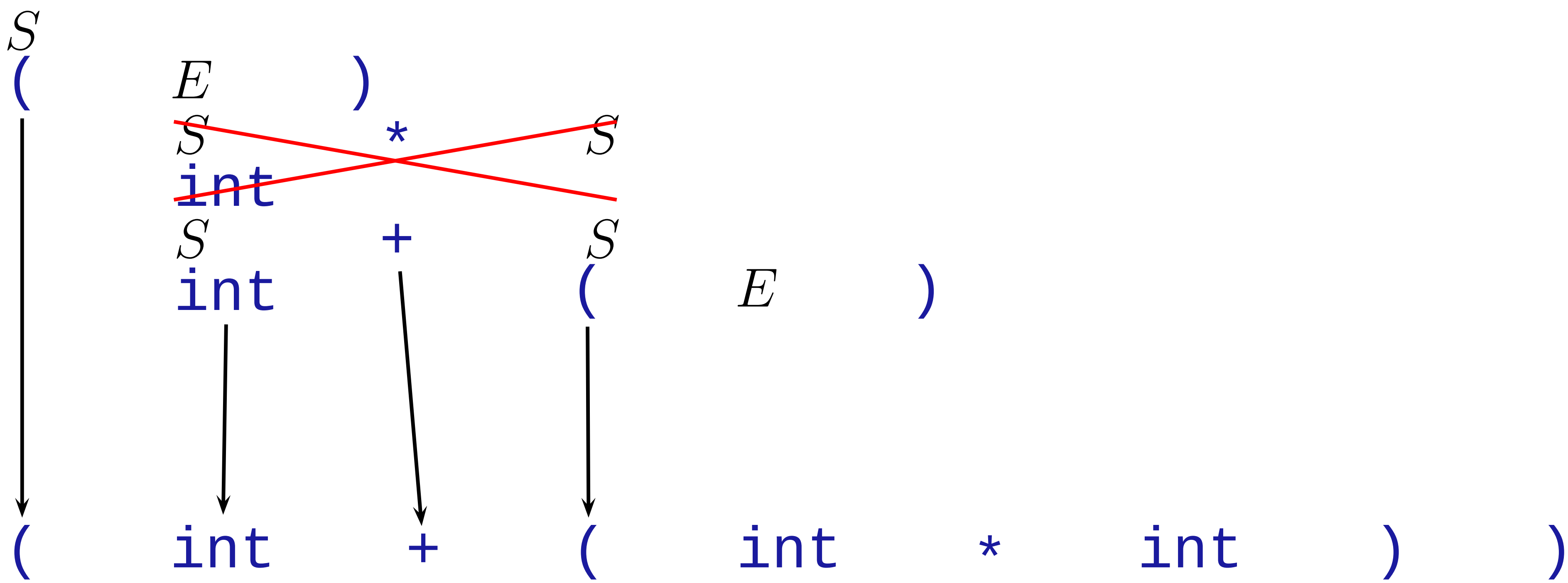


Idea: Recursively compute all expansions of a nonterminal at some input position

$E \rightarrow S\texttt{+}S,\quad E \rightarrow S\,\texttt{*}\,S,$

$S \rightarrow \texttt{(}\,E\,\texttt{)},\quad S \rightarrow \texttt{int}$

expand(S, 3)

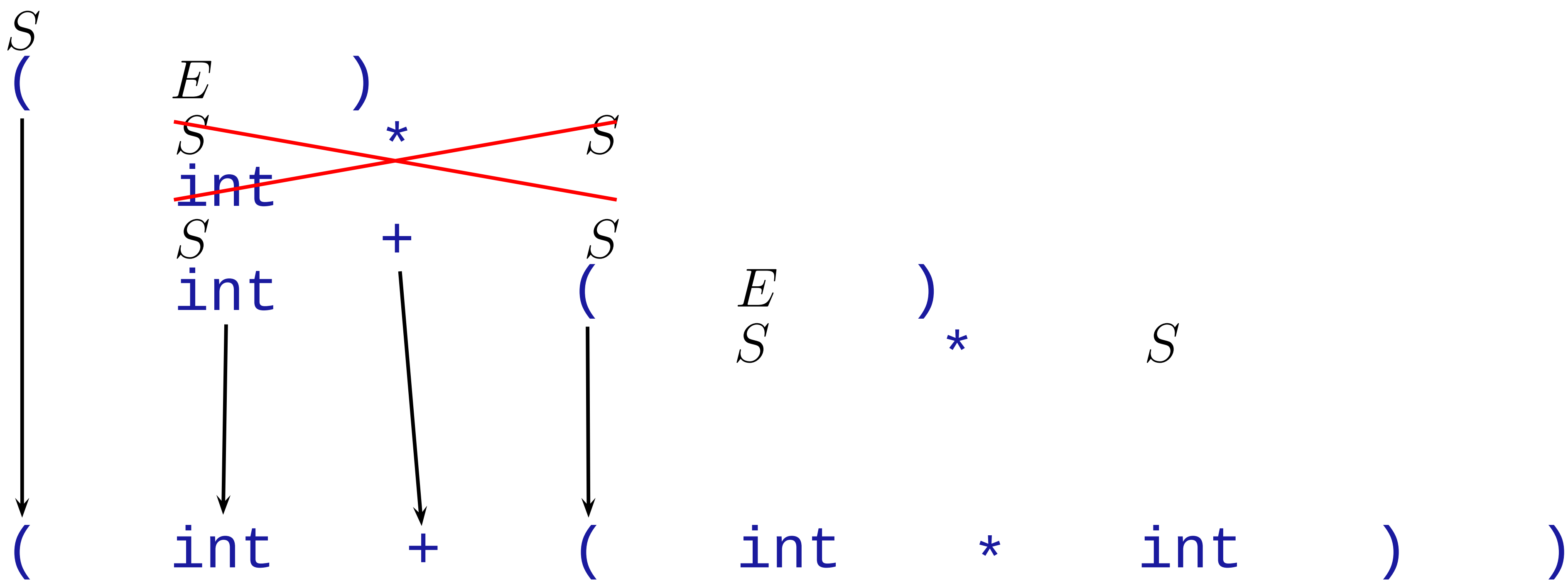


Recursive Descent Parsing

Idea: Recursively compute all expansions of a nonterminal at some input position

`expand(`E`, 4)`

$E \rightarrow S$ `+` S, $E \rightarrow S$ `*` S,
$S \rightarrow$ `(` E `)`, $S \rightarrow$ `int`

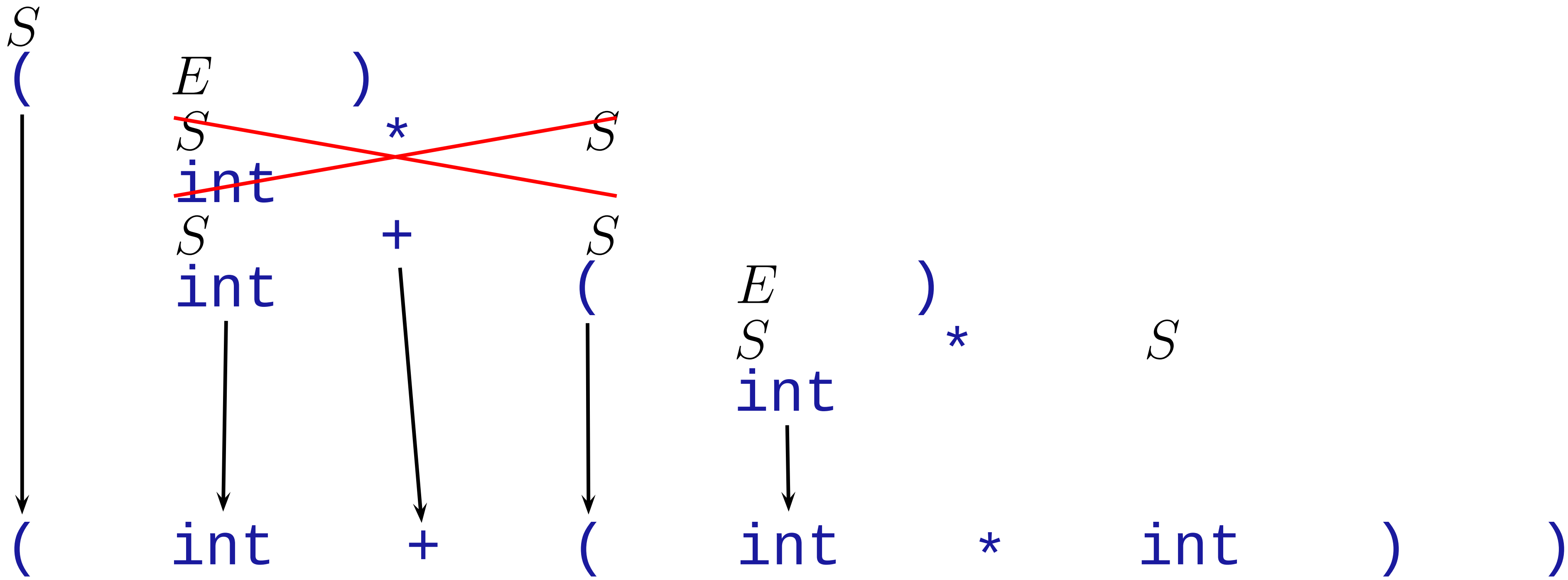


Recursive Descent Parsing

Idea: Recursively compute all expansions of a nonterminal at some input position

expand(S, 4)

$E \to S$ + S, $E \to S$ * S,
$S \to$ (E), $S \to$ int

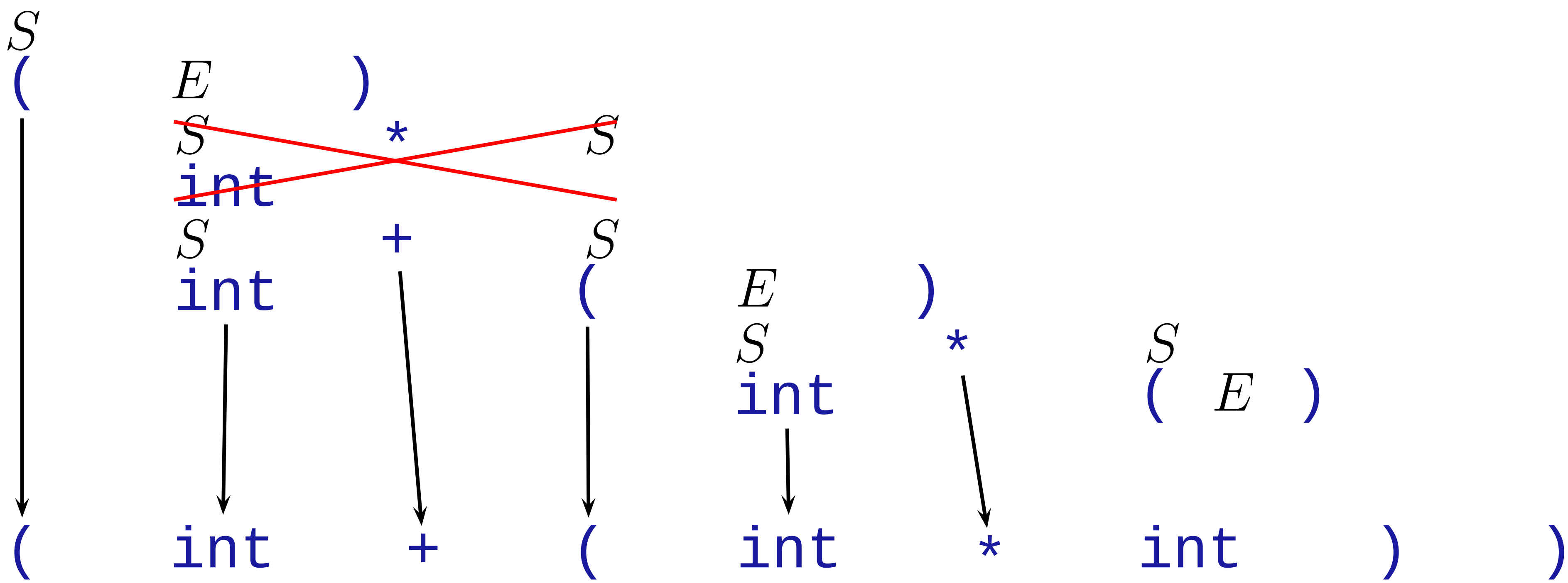


Recursive Descent Parsing

Idea: Recursively compute all expansions of a nonterminal at some input position

$E \longrightarrow S$+S, $E \longrightarrow S$ * S,
$S \longrightarrow$ (E), $S \longrightarrow$ int

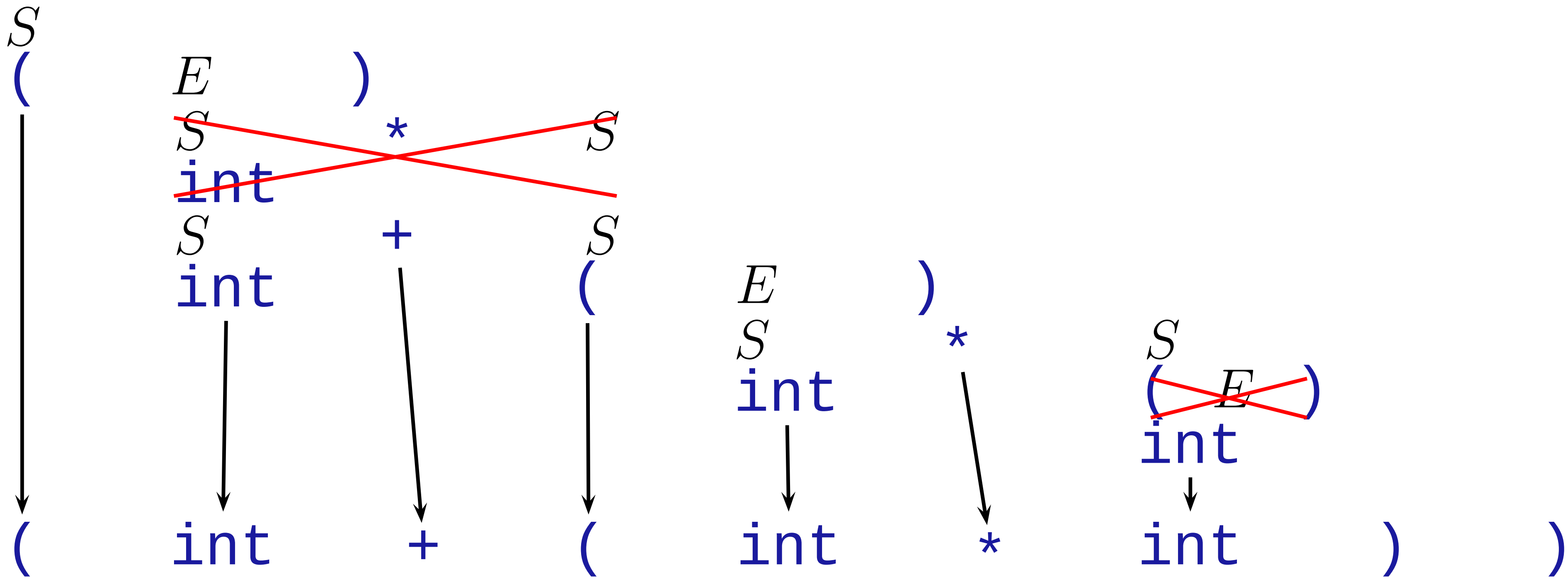
expand(S, 6)



Idea: Recursively compute all expansions of a nonterminal at some input position

`expand(`S`, 6)`

$E \longrightarrow S\texttt{+}S, \quad E \longrightarrow S\texttt{*}S,$
$S \longrightarrow \texttt{(}E\texttt{)}, \quad S \longrightarrow \texttt{int}$

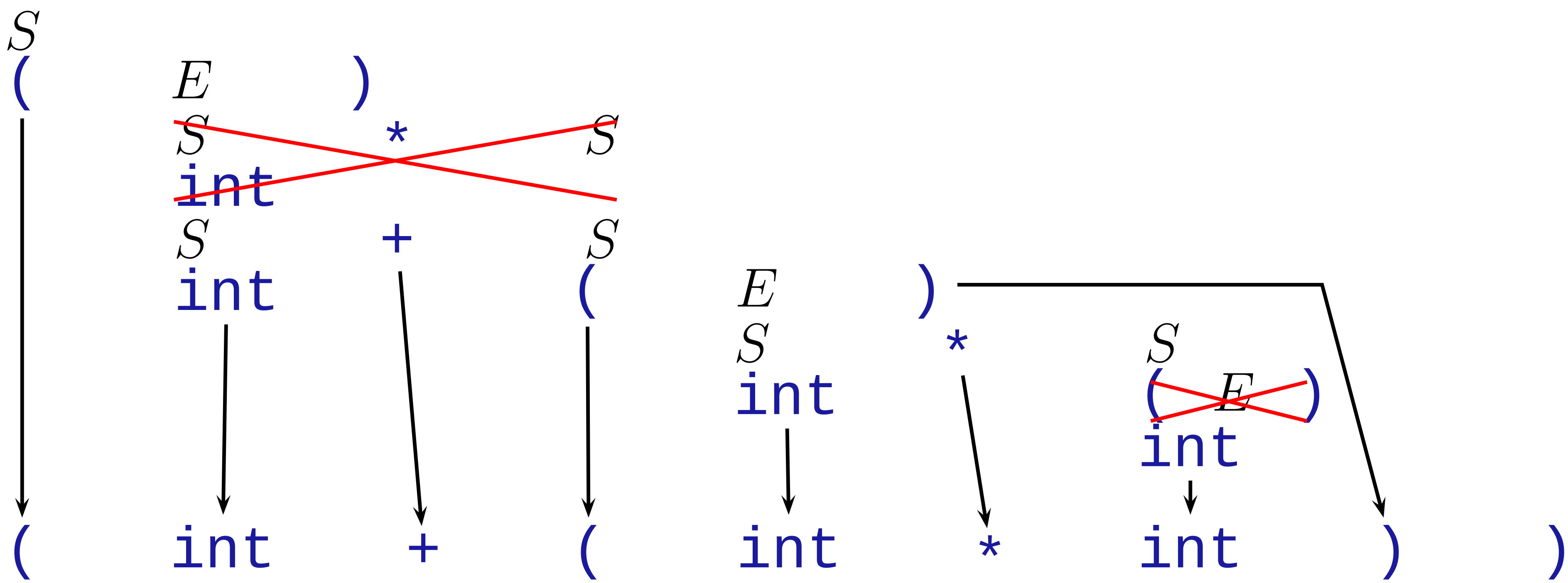


Idea: Recursively compute all expansions of a nonterminal at some input position

expand(S, 3)

$E \longrightarrow S + S, \quad E \longrightarrow S * S,$

$S \longrightarrow (E), \quad S \longrightarrow \texttt{int}$



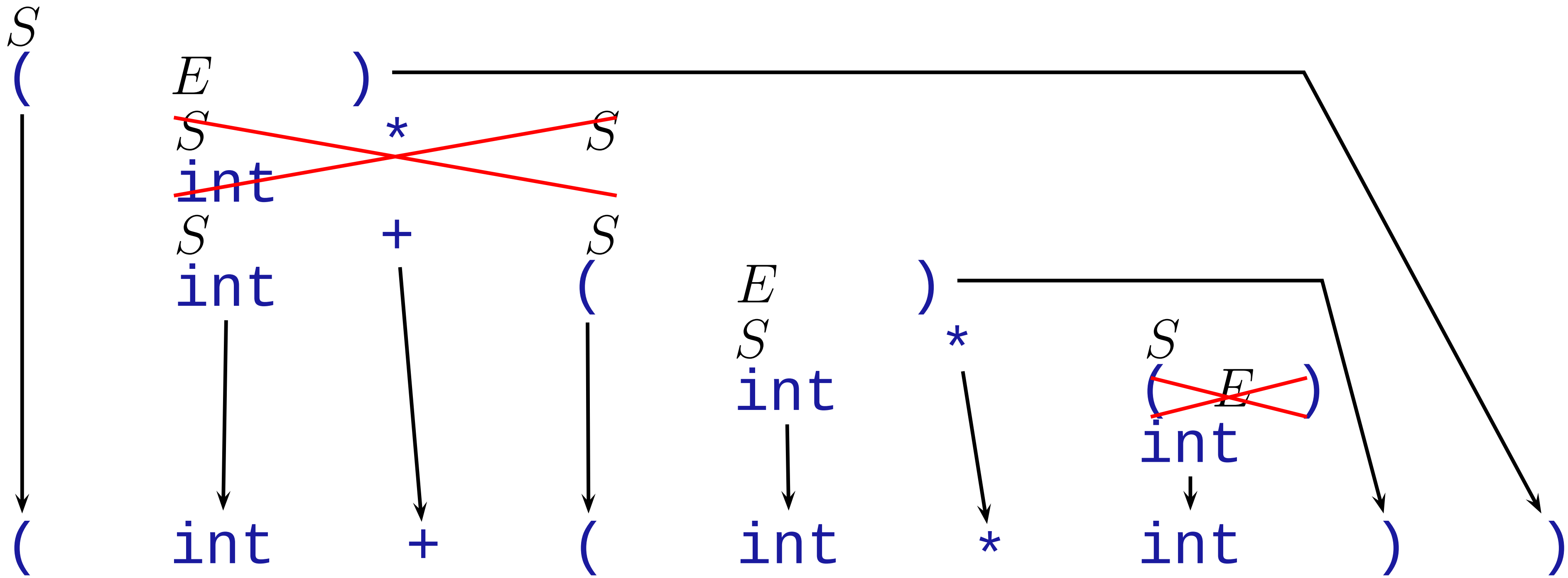
Recursive Descent Parsing

Idea: Recursively compute all expansions of a nonterminal at some input position

`expand(`S`, 0)`

$E \longrightarrow S$ `+` S, $E \longrightarrow S$ `*` S,

$S \longrightarrow$ `(` E `)`, $S \longrightarrow$ `int`



Recursive Descent Parsing II

- expand returns a set of possible end positions
- During expansion of one production
  - Keep track of a set of intermediate positions for the already expanded part
  - Expand the next terminal or nonterminal from every intermediate position
  - If the set of possible intermediate position gets empty, the production fails
- When a production was successfully completed, the intermediate set is added to the set of end positions
- May run into an infinite loop with left-recursive grammars, i.e. grammars where $A \overset{*}{\underset{G}{\Rightarrow}} A\beta$ for some A

Top Down Parsing: Optimizations

- For every nonterminal, compute the set of terminals that can occur in the *first* position
  - ➢ Expand only those nonterminals / productions that are compatible with the current input symbol
- Avoid performing duplicate calls with identical nonterminal/position pair: *memoize* previous calls
  - ➢ use a data structure whose index are tuples consisting of the function arguments
  - ➢ the result of the lookup is the result of a previous call with the same arguments (if available)
  - ➢ The memoized method has to be strictly functional for this to work

Epsilon Reduction

- A CFG may contain productions of the form $A \to \epsilon$
- Construct a CFG G' with the same language as G and at most one epsilon production: $S \to \epsilon$

for all nonterminals A with $A \to \epsilon \in P$:
  mark A as ϵ-deriving and add it to the set Q
while Q is not empty, remove a nonterminal X from Q:
  for all $Y \to \alpha\, X\, \beta \in P$, with α or β not empty, add $Y \to \alpha\, \beta$ to P'
  for all $Y \to X$, if Y is not marked as ϵ-deriving:
    mark Y as ϵ-deriving and add it to Q
if S is ϵ-deriving, add $S \to \epsilon$ to P'
add all non-ϵ productions of P to P'

Chomsky Normal Form

- A context-free grammar is in *Chomsky Normal Form* if:
  (i) it is ϵ-free,
  (ii) all productions have one of two forms:
  - $A \rightarrow a$ with $a \in \Sigma$
  - $A \rightarrow B\,C$ with $B, C \in N$
- Every CFG can be transformed into a CNF grammar with the same language
- Drawback: The original structure of the parse trees must be reconstructed, if necessary
- The original and transformed grammar are said to be *weakly equivalent*

Chomsky Normal Form II

- Convert an arbitrary CFG into CNF:
  - Introduce new nonterminals and productions $A^a \rightarrow a$
  - Replace all occurences of a by A^a
  - Eliminate unary productions $A \rightarrow B$:
    add productions where A is replaced by B in the right hand sides
  - Replace productions with more than two symbols on the right hand side by a sequence of productions:
    $A \rightarrow R_1\, R_2\, \ldots R_n \Rightarrow$
    $$A \rightarrow R_1 A^{(1)}, \quad A^{(1)} \rightarrow R_2 A^{(2)}, \quad \cdots \quad A^{(n-2)} \rightarrow R_{n-1} R_n$$

Chart Parsing

- First algorithm independently developed by Cocke, Younger and Kasami (late 60s)
- Given a string w of length n, use an $n \times n$ table to store subderivations (hence chart or tabular parsing)
- Works for all kinds of grammars: left/right recursive, ambiguous
- Storing subderivations avoids duplicate computation: an instance of *dynamic programming*
- polynomial space and time bounds, although an exponential number of parse trees may be encoded!

- Input: G in Chomsky normal form, input string $w_1, \ldots, w_n$
- Systematically explore all possible sub-derivations bottom-up
- Use an $n \times n$ array $\mathcal{C}$ such that
  - ➤ If nonterminal A is stored in $\mathcal{C}(i,k)$: $A \overset{*}{\underset{G}{\Rightarrow}} w_{i+1}, \ldots, w_k$
  - ➤ Maintain a second table $\mathcal{B}$, such that if $j \in \mathcal{B}(i,k)$: ex. $A \rightarrow B\,C \in P, A \in \mathcal{C}(i,k), B \in \mathcal{C}(i,j)$ and $C \in \mathcal{C}(j,k)$
  - ➤ $\mathcal{B}$ enables us to extract the parse trees
- Implement $\mathcal{C}$ and $\mathcal{B}$ as three-dimensional boolean arrays of size $n \times n \times |N|$ and $n \times n \times n$, respectively

For $i = 1$ to n

    For each $R_j \longrightarrow a_i$, set $\mathcal{C}(i-1, i, j) = \texttt{true}$

For $l = 2$ to n *– Length of new constituent*

    For $i = 0$ to $n - l$ *– Start of new constituent*

        For $m = 1$ to $l - 1$ *– Length of first subconstituent*

            For each production $R_a \longrightarrow R_b R_c$

                If $\mathcal{C}(i, i+m, b)$ and $\mathcal{C}(i+m, i+l, c)$ then

                    set $\mathcal{C}(i, i+l, a) = \texttt{true}$

                    set $\mathcal{B}(i, i+l, i+m) = \texttt{true}$

If $\mathcal{C}(1, n, S)$ is `true`, $w \in \mathcal{L}(G)$

CYK Chart Example

$\mathcal{C}$

$\mathcal{B}$

⓪ *john* ① *saw* ② *the* ③ *girl* ④ *in* ⑤ *a* ⑥ *car* ⑦

CYK Chart Example

$\mathcal{C}$

N						
	V					
		D				
			N			
				P		
					D	
						N

$\mathcal{B}$

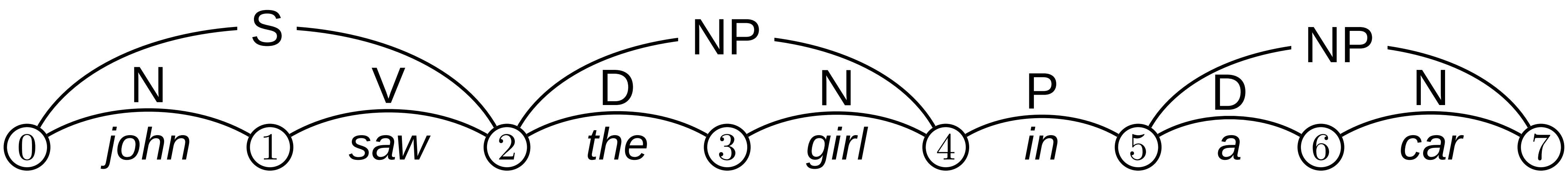
CYK Chart Example

$\mathcal{C}$

N	S					
	V					
		D	NP			
			N			
				P		
					D	NP
						N

$\mathcal{B}$

	1					
			3			
						6



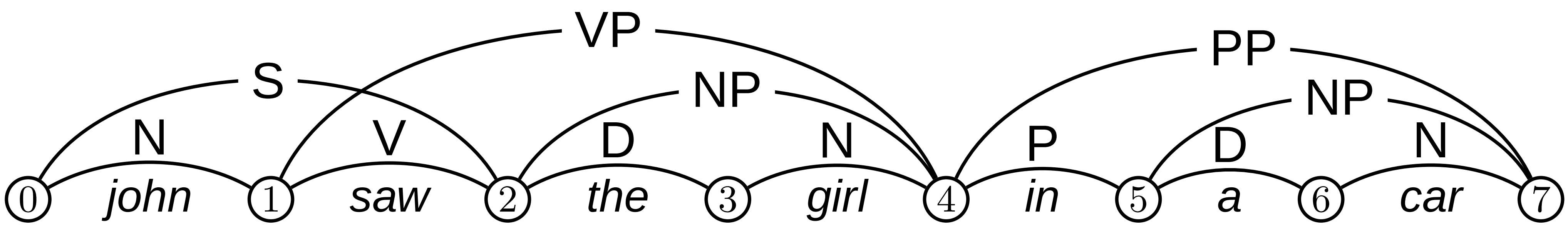
CYK Chart Example

$\mathcal{C}$

N	S					
	V		VP			
		D	NP			
			N			
				P		PP
					D	NP
						N

$\mathcal{B}$

	1					
			2			
			3			
						5
						6



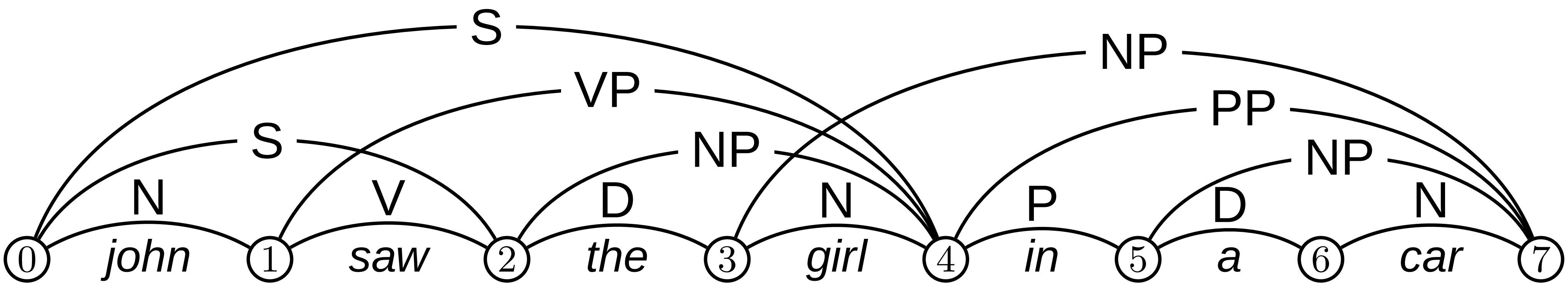
CYK Chart Example

$\mathcal{C}$

N	S		S			
	V		VP			
		D	NP			
			N			NP
				P		PP
					D	NP
						N

$\mathcal{B}$

	1		1			
			2			
			3			
						4
						5
						6



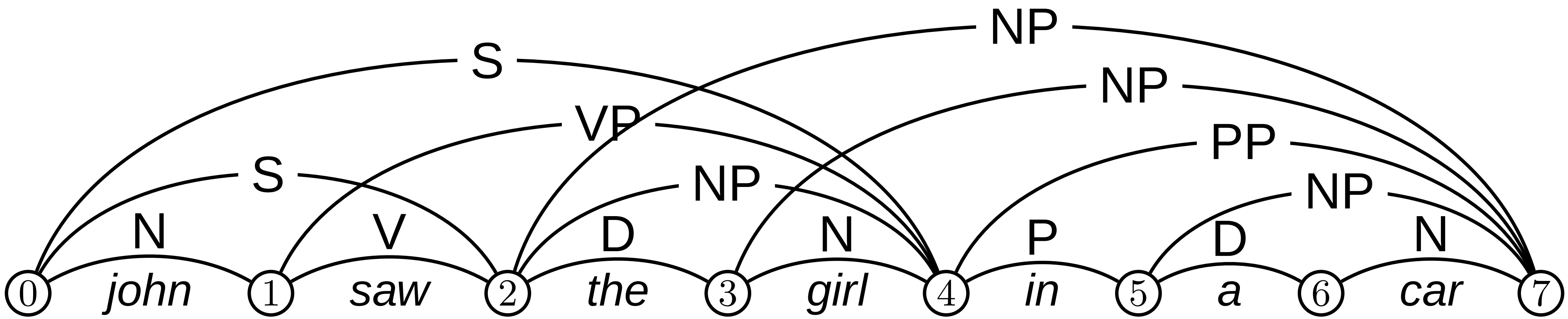
CYK Chart Example

$\mathcal{C}$

N	S		S			
	V		VP			
		D	NP			NP
			N			NP
				P		PP
					D	NP
						N

$\mathcal{B}$

	1		1			
			2			
			3			4
						4
						5
						6



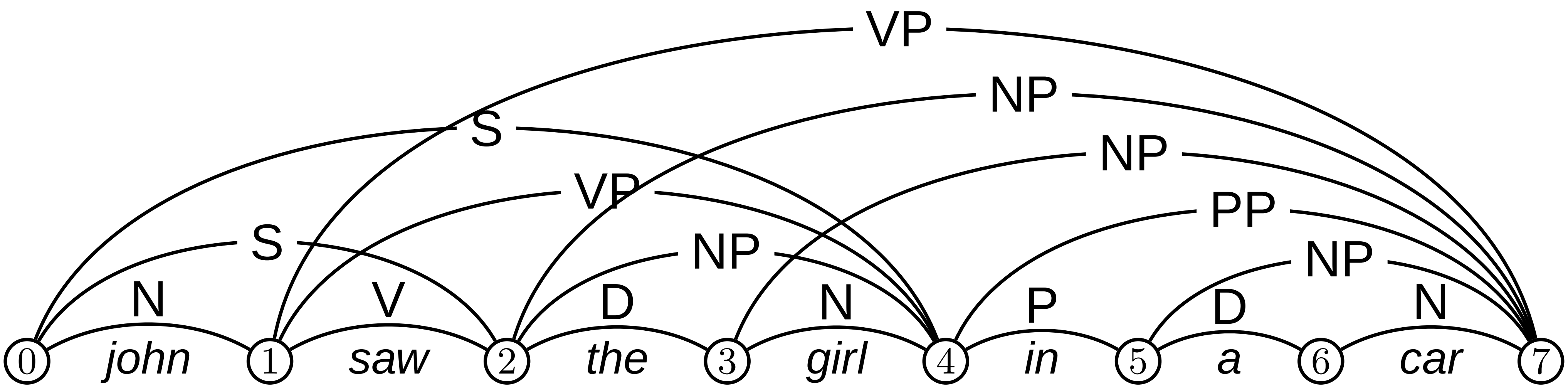
CYK Chart Example

$\mathcal{C}$

N	S		S			
	V		VP			VP
		D	NP			NP
			N			NP
				P		PP
					D	NP
						N

$\mathcal{B}$

	1		1			
			2			2,4
			3			4
						4
						5
						6



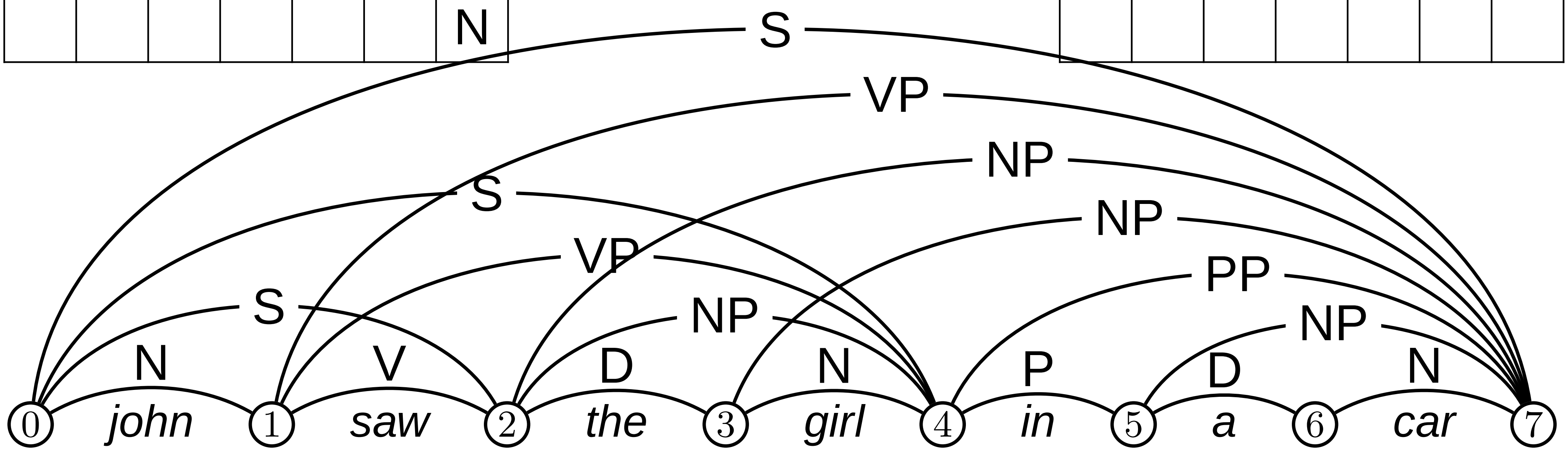
CYK Chart Example

$\mathcal{C}$

N	S		S			S
	V		VP			VP
		D	NP			NP
			N			NP
				P		PP
					D	NP
						N

$\mathcal{B}$

	1		1			1
			2			2,4
			3			4
						4
						5
						6



Example Chart

S →NP VP|N VP|N V|NP V

VP→V NP|V N|VP PP

NP→D N|NP PP|N PP

PP→P NP|P N

N →*john, girl, car*

V →*saw, walks*

P →*in*

D →*the, a*

	1	2	3	4	5	6	7
0	N	S		S			S
1		V		VP			VP(2)
2			D	NP			NP
3				N			NP
4					P		PP
5						D	NP
6							N

$_0$ *john* $_1$ *saw* $_2$ *the* $_3$ *girl* $_4$ *in* $_5$ *a* $_6$ *car* $_7$

Example Chart

S →NP VP|N VP|N V|NP V

VP→V NP|V N|VP PP

NP→D N|NP PP|N PP

PP→P NP|P N

N →*john, girl, car*

V →*saw, walks*

P →*in*

D →*the, a*

	1	2	3	4	5	6	7
0	N	S		S			S
1		V		VP			VP(2)
2			D	NP			NP
3				N			NP
4					P		PP
5						D	NP
6							N

$_0$ *john* $_1$ *saw* $_2$ *the* $_3$ *girl* $_4$ *in* $_5$ *a* $_6$ *car* $_7$

- S →NP VP|N VP|N V|NP V
- VP→V NP|V N|VP PP
- NP→D N|NP PP|N PP
- PP→P NP|P N
- N →*john, girl, car*
- V →*saw, walks*
- P →*in*
- D →*the, a*

	1	2	3	4	5	6	7
0	N	S		S			S
1		V		VP			VP(2)
2			D	NP			NP
3				N			NP
4					P		PP
5						D	NP
6							N

Example Chart

S → NP VP | N VP | N V | NP V

VP → V NP | V N | VP PP

NP → D N | NP PP | N PP

PP → P NP | P N

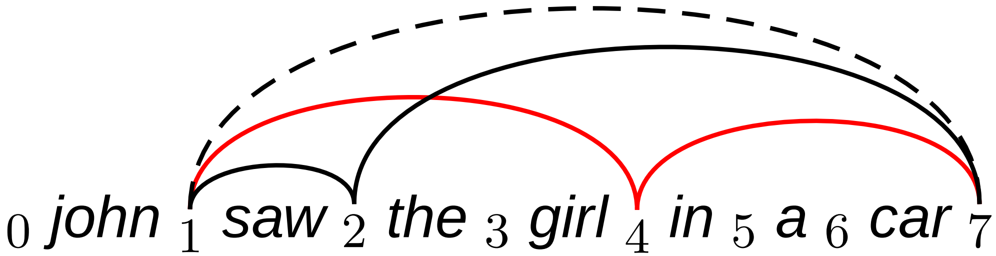
N → *john, girl, car*

V → *saw, walks*

P → *in*

D → *the, a*

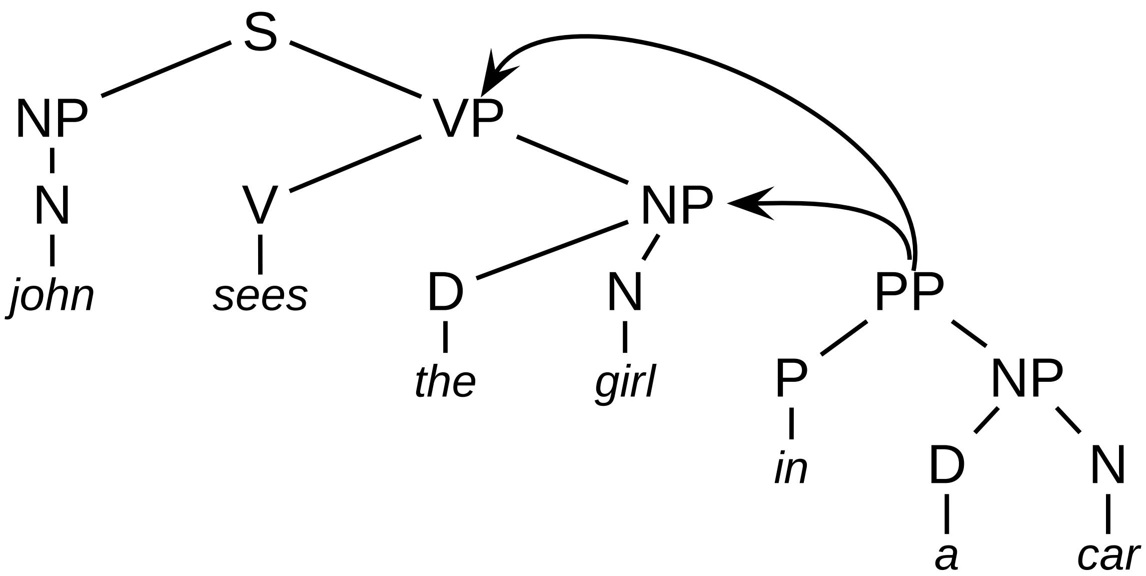
	1	2	3	4	5	6	7
0	N	S		S			S
1		V		VP			VP(2)
2			D	NP			NP
3				N			NP
4					P		PP
5						D	NP
6							N



Encoding Ambiguities

$\Sigma = \{$*john, girl, car, sees, in, the, a*$\}$ $N = \{$S, NP, VP, PP, D, N, V, P$\}$

$$P = \left\{ \begin{array}{ll} \text{S} \rightarrow \text{NP VP,} & \text{N} \rightarrow \textit{john, girl, car} \\ \text{VP} \rightarrow \text{V} | \text{V NP} | \text{V NP PP} & \text{V} \rightarrow \textit{sees} \\ \text{NP} \rightarrow \text{N} | \text{D N} | \text{N PP} | \text{D N PP} & \text{P} \rightarrow \textit{in} \\ \text{PP} \rightarrow \text{P NP} & \text{D} \rightarrow \textit{the, a} \end{array} \right\}$$



Describing Chart Parsing Algorithms

Parsing algorithm should use *original* grammars for comparability

Describing Chart Parsing Algorithms

Parsing algorithm should use *original* grammars for comparability

- They are described using chart items, which consist of
  - a symbol, derived from the grammar
  - a start and end position from $0, \ldots, n$

Describing Chart Parsing Algorithms

Parsing algorithm should use *original* grammars for comparability

- They are described using chart items, which consist of
  - ➤ a symbol, derived from the grammar
  - ➤ a start and end position from $0, \ldots, n$
- The symbol of *complete* items is one of $\Sigma \cup N$

Describing Chart Parsing Algorithms

Parsing algorithm should use *original* grammars for comparability

- They are described using chart items, which consist of
  - ➤ a symbol, derived from the grammar
  - ➤ a start and end position from $0, \ldots, n$
- The symbol of *complete* items is one of $\Sigma \cup N$
- *Incomplete* chart items encode partially filled rules
  - ➤ the symbol is a pair (r, i) of rule and *dot position* if $P \ni r : A \rightarrow \alpha\beta$ with $|\alpha| = i$
  - ➤ write alternatively: $A \rightarrow \alpha \bullet \beta$

Bottom-Up Chart Parsing

- How, when and which chart items are created or combined characterizes a parsing algorithm or parsing strategy
- First: A modified variant of Cocke-Younger-Kasami (CYK) algorithm
- Prerequisites: CFG G, input string $w = a_1, \ldots, a_n$
- Data Structures:
  - ➤ A $n+1 \times n+1$ chart $\mathcal{C}$, where each cell contains a set of (complete or incomplete) chart items
  - ➤ A set of chart items $\mathcal{A}$ (those must still be treated in some way)
- Initialization: add all $(a_i, i-1, i)$ to $\mathcal{A}$ and $\mathcal{C}_{i-1,i}$

Bottom-Up Parsing Algorithm

while $\mathcal{A}$ not empty
  take an (X, i, j) from $\mathcal{A}$ and remove it
  if $X \in \Sigma \cup N$
    for $P \ni r \equiv A \rightarrow X\alpha$ do
      check_and_add$(A \rightarrow X \bullet \alpha, i, j)$
    for $k \in 0, \ldots, i-1$ do
      for all $(A \rightarrow \beta \bullet X\alpha, k, i) \in \mathcal{C}$ do
        check_and_add$(A \rightarrow \beta X \bullet \alpha, k, j)$
  else   – incomplete item: $X \equiv A \rightarrow \beta \bullet Y\alpha$
    for $k \in j+1, \ldots, n$ do
      if $(Y, j, k) \in \mathcal{C}$ *check_and_add*$(A \rightarrow \beta Y \bullet \alpha, i, k)$

check_and_add$(X \equiv A \rightarrow \alpha \bullet \beta, i, j) \equiv$
  if $\beta = \epsilon$ then if $(A, i, j) \notin \mathcal{C}$ add (A, i, j) to $\mathcal{A}$ and $\mathcal{C}$ endif
  else if $(A \rightarrow \alpha \bullet \beta, i, j) \notin \mathcal{C}$ add $(A \rightarrow \alpha \bullet \beta, i, j)$ to $\mathcal{A}$ and $\mathcal{C}$ endif

Bottom-Up: The Details

- How to implement $\mathcal{A}$ and $\mathcal{C}$ efficiently?
- Implementation of the $(n+1)^2$ sets in $\mathcal{C}$:

Bottom-Up: The Details

- How to implement $\mathcal{A}$ and $\mathcal{C}$ efficiently?
- Implementation of the $(n+1)^2$ sets in $\mathcal{C}$:
  - ➢ Operations: add single element, contains element

Bottom-Up: The Details

- How to implement $\mathcal{A}$ and $\mathcal{C}$ efficiently?
- Implementation of the $(n+1)^2$ sets in $\mathcal{C}$:
  - ➤ Operations: add single element, contains element
  - ➤ bit vector of size $|G| := |\Sigma| + |N| + \sum_{P \ni r : A \rightarrow \alpha} |A\alpha|$

Bottom-Up: The Details

- How to implement $\mathcal{A}$ and $\mathcal{C}$ efficiently?
- Implementation of the $(n+1)^2$ sets in $\mathcal{C}$:
  - Operations: add single element, contains element
  - bit vector of size $|G| := |\Sigma| + |N| + \sum_{P \ni r : A \to \alpha} |A\alpha|$
- Implementation of the set $\mathcal{A}$:

Bottom-Up: The Details

- How to implement $\mathcal{A}$ and $\mathcal{C}$ efficiently?
- Implementation of the $(n+1)^2$ sets in $\mathcal{C}$:
  - Operations: add single element, contains element
  - bit vector of size $|G| := |\Sigma| + |N| + \sum_{P \ni r:A \rightarrow \alpha} |A\alpha|$
- Implementation of the set $\mathcal{A}$:
  - Operations: add , get and remove some element

Bottom-Up: The Details

- How to implement $\mathcal{A}$ and $\mathcal{C}$ efficiently?
- Implementation of the $(n+1)^2$ sets in $\mathcal{C}$:
  - Operations: add single element, contains element
  - bit vector of size $|G| := |\Sigma| + |N| + \sum_{P \ni r : A \rightarrow \alpha} |A\alpha|$
- Implementation of the set $\mathcal{A}$:
  - Operations: add , get and remove some element
  - (priority) queue, stack
  - $\mathcal{A}$ is called *agenda* and can be used to implement search strategies
- Keep terminal items separate from the chart for space and time efficiency

Runtime of CYK Algorithm

check_and_add$(X \equiv A \rightarrow \alpha\bullet\beta, i, j) \equiv$

if $\beta = \epsilon$ then if $(A, i, j) \notin \mathcal{C}$ add (A, i, j) to $\mathcal{A}$ and $\mathcal{C}$ endif
else if $(A \rightarrow \alpha\bullet\beta, i, j) \notin \mathcal{C}$ add $(A \rightarrow \alpha\bullet\beta, i, j)$ to $\mathcal{A}$ and $\mathcal{C}$ endif

Runtime of CYK Algorithm

check_and_add$(X \equiv A \rightarrow \alpha \bullet \beta, i, j) \equiv$ all operations $\mathcal{O}(1)$

if $\beta = \epsilon$ then if $(A, i, j) \notin \mathcal{C}$ add (A, i, j) to $\mathcal{A}$ and $\mathcal{C}$ endif

else if $(A \rightarrow \alpha \bullet \beta, i, j) \notin \mathcal{C}$ add $(A \rightarrow \alpha \bullet \beta, i, j)$ to $\mathcal{A}$ and $\mathcal{C}$ endif *check_and_add*$(X \equiv A \rightarrow \alpha\bullet\beta, i, j) \equiv$ all operations $\mathcal{O}(1)$

if $\beta = \epsilon$ then if $(A, i, j) \notin \mathcal{C}$ add (A, i, j) to $\mathcal{A}$ and $\mathcal{C}$ endif

else if $(A \rightarrow \alpha\bullet\beta, i, j) \notin \mathcal{C}$ add $(A \rightarrow \alpha\bullet\beta, i, j)$ to $\mathcal{A}$ and $\mathcal{C}$ endif

while $\mathcal{A}$ not empty

take an (X, i, j) from $\mathcal{A}$ and remove it

if $X \in \Sigma \cup N$

for $P \ni r \equiv A \rightarrow X\alpha$ do

check_and_add$(A \rightarrow X\bullet\alpha, i, j)$

for $k \in 0, \ldots, i-1$ do

for all $(A \rightarrow \beta\bullet X\alpha, k, i) \in \mathcal{C}$ do

check_and_add$(A \rightarrow \beta X\bullet\alpha, k, j)$

else – incomplete item: $X \equiv A \rightarrow \beta\bullet Y\alpha$

for $k \in j+1, \ldots, n$ do

if $(Y, j, k) \in \mathcal{C}$ *check_and_add*$(A \rightarrow \beta Y\bullet\alpha, i, k)$

check_and_add$(X \equiv A \rightarrow \alpha\bullet\beta, i, j) \equiv$ all operations $\mathcal{O}(1)$

if $\beta = \epsilon$ then if $(A, i, j) \notin \mathcal{C}$ add (A, i, j) to $\mathcal{A}$ and $\mathcal{C}$ endif

else if $(A \rightarrow \alpha\bullet\beta, i, j) \notin \mathcal{C}$ add $(A \rightarrow \alpha\bullet\beta, i, j)$ to $\mathcal{A}$ and $\mathcal{C}$ endif

while $\mathcal{A}$ not empty

take an (X, i, j) from $\mathcal{A}$ and remove it max. $|G| \times (n+1)^2$

if $X \in \Sigma \cup N$

for $P \ni r \equiv A \rightarrow X\alpha$ do

check_and_add$(A \rightarrow X\bullet\alpha, i, j)$

for $k \in 0, \ldots, i-1$ do

for all $(A \rightarrow \beta\bullet X\alpha, k, i) \in \mathcal{C}$ do

check_and_add$(A \rightarrow \beta X\bullet\alpha, k, j)$

else – incomplete item: $X \equiv A \rightarrow \beta\bullet Y\alpha$

for $k \in j+1, \ldots, n$ do

if $(Y, j, k) \in \mathcal{C}$ *check_and_add*$(A \rightarrow \beta Y\bullet\alpha, i, k)$

Runtime of CYK Algorithm

check_and_add$(X \equiv A \rightarrow \alpha\bullet\beta, i, j) \equiv$ all operations $\mathcal{O}(1)$

if $\beta = \epsilon$ then if $(A, i, j) \notin \mathcal{C}$ add (A, i, j) to $\mathcal{A}$ and $\mathcal{C}$ endif

else if $(A \rightarrow \alpha\bullet\beta, i, j) \notin \mathcal{C}$ add $(A \rightarrow \alpha\bullet\beta, i, j)$ to $\mathcal{A}$ and $\mathcal{C}$ endif

while $\mathcal{A}$ not empty

take an (X, i, j) from $\mathcal{A}$ and remove it max. $|G| \times (n+1)^2$

if $X \in \Sigma \cup N$

for $P \ni r \equiv A \rightarrow X\alpha$ do max. $|G|$

check_and_add$(A \rightarrow X\bullet\alpha, i, j)$

for $k \in 0, \ldots, i-1$ do

for all $(A \rightarrow \beta\bullet X\alpha, k, i) \in \mathcal{C}$ do

check_and_add$(A \rightarrow \beta X\bullet\alpha, k, j)$

else – incomplete item: $X \equiv A \rightarrow \beta\bullet Y\alpha$

for $k \in j+1, \ldots, n$ do

if $(Y, j, k) \in \mathcal{C}$ *check_and_add*$(A \rightarrow \beta Y\bullet\alpha, i, k)$

Runtime of CYK Algorithm

check_and_add$(X \equiv A \rightarrow \alpha \bullet \beta, i, j) \equiv$ all operations $\mathcal{O}(1)$

if $\beta = \epsilon$ then if $(A, i, j) \notin \mathcal{C}$ add (A, i, j) to $\mathcal{A}$ and $\mathcal{C}$ endif

else if $(A \rightarrow \alpha \bullet \beta, i, j) \notin \mathcal{C}$ add $(A \rightarrow \alpha \bullet \beta, i, j)$ to $\mathcal{A}$ and $\mathcal{C}$ endif

while $\mathcal{A}$ not empty

take an (X, i, j) from $\mathcal{A}$ and remove it max. $|G| \times (n + 1)^2$

if $X \in \Sigma \cup N$

for $P \ni r \equiv A \rightarrow X\alpha$ do max. $|G|$

check_and_add$(A \rightarrow X \bullet \alpha, i, j)$

for $k \in 0, \ldots, i - 1$ do max. n times

for all $(A \rightarrow \beta \bullet X\alpha, k, i) \in \mathcal{C}$ do

check_and_add$(A \rightarrow \beta X \bullet \alpha, k, j)$

else – incomplete item: $X \equiv A \rightarrow \beta \bullet Y\alpha$

for $k \in j + 1, \ldots, n$ do

if $(Y, j, k) \in \mathcal{C}$ *check_and_add*$(A \rightarrow \beta Y \bullet \alpha, i, k)$

Runtime of CYK Algorithm

check_and_add$(X \equiv A \rightarrow \alpha{\bullet}\beta, i, j) \equiv$ — all operations $\mathcal{O}(1)$

if $\beta = \epsilon$ then if $(A, i, j) \notin \mathcal{C}$ add (A, i, j) to $\mathcal{A}$ and $\mathcal{C}$ endif
else if $(A \rightarrow \alpha{\bullet}\beta, i, j) \notin \mathcal{C}$ add $(A \rightarrow \alpha{\bullet}\beta, i, j)$ to $\mathcal{A}$ and $\mathcal{C}$ endif

while $\mathcal{A}$ not empty
  take an (X, i, j) from $\mathcal{A}$ and remove it — max. $|G| \times (n+1)^2$
  if $X \in \Sigma \cup N$
    for $P \ni r \equiv A \rightarrow X\alpha$ do — max. $|G|$
      check_and_add$(A \rightarrow X{\bullet}\alpha, i, j)$
    for $k \in 0, \ldots, i-1$ do — max. n times
      for all $(A \rightarrow \beta{\bullet}X\alpha, k, i) \in \mathcal{C}$ do — max. $|G|$ times
        check_and_add$(A \rightarrow \beta X{\bullet}\alpha, k, j)$
  else – incomplete item: $X \equiv A \rightarrow \beta{\bullet}Y\alpha$
    for $k \in j+1, \ldots, n$ do
      if $(Y, j, k) \in \mathcal{C}$ *check_and_add*$(A \rightarrow \beta Y{\bullet}\alpha, i, k)$

Runtime of CYK Algorithm

check_and_add$(X \equiv A \rightarrow \alpha \bullet \beta, i, j) \equiv$ all operations $\mathcal{O}(1)$

if $\beta = \epsilon$ then if $(A, i, j) \notin \mathcal{C}$ add (A, i, j) to $\mathcal{A}$ and $\mathcal{C}$ endif

else if $(A \rightarrow \alpha \bullet \beta, i, j) \notin \mathcal{C}$ add $(A \rightarrow \alpha \bullet \beta, i, j)$ to $\mathcal{A}$ and $\mathcal{C}$ endif

while $\mathcal{A}$ not empty

take an (X, i, j) from $\mathcal{A}$ and remove it — max. $|G| \times (n+1)^2$

if $X \in \Sigma \cup N$

for $P \ni r \equiv A \rightarrow X\alpha$ do — max. $|G|$

check_and_add$(A \rightarrow X \bullet \alpha, i, j)$

for $k \in 0, \ldots, i-1$ do — max. n times

for all $(A \rightarrow \beta \bullet X\alpha, k, i) \in \mathcal{C}$ do — max. $|G|$ times

check_and_add$(A \rightarrow \beta X \bullet \alpha, k, j)$

else – incomplete item: $X \equiv A \rightarrow \beta \bullet Y\alpha$

for $k \in j+1, \ldots, n$ do — max. n times

if $(Y, j, k) \in \mathcal{C}$ *check_and_add*$(A \rightarrow \beta Y \bullet \alpha, i, k)$

CYK – Summary

- Polynomial complexity: $\mathcal{O}(|G|^2 n^3)$
- Explores all possible sub-derivations
- Advantageous for *robust parsing*:
  Extract the biggest/best chunks for ungrammatical input
- That a derivation must start at S is not used at all
  - ➤ Average time is near or equal to the worst case
  - ➤ May lead to poor performance in practice
- Two main steps:
  - ➤ if $(X, i, j) \in \mathcal{C}, X \in \Sigma \cup N$ and $A \rightarrow X\alpha \in P$:
    add $(A \rightarrow X \bullet \alpha, i, j)$ to $\mathcal{C}$
  - ➤ if $(A \rightarrow \beta \bullet Y \alpha, i, j) \in \mathcal{C}$ and $(Y, j, k) \in \mathcal{C}$:
    add $(A \rightarrow \beta Y \bullet \alpha, i, k)$ to $\mathcal{C}$

Predictive Bottom-Up: Earley Parsing

- Described by J. Earley (1970): Predict Top-Down and Complete Bottom-Up
- Initialize by adding the terminal items *and* $(S \rightarrow \alpha, 0, 0)$ for all $S \rightarrow \bullet\alpha \in P$
- Three main operations:

Prediction if $(A \rightarrow \beta \bullet Y\alpha, i, j) \in \mathcal{C}$, add $(Y \rightarrow \bullet\gamma, j, j)$ to $\mathcal{C}$ for every $Y \rightarrow \gamma \in P$

Scanning if $(A \rightarrow \beta \bullet a_{j+1}\alpha, i, j) \in \mathcal{C}$, add $(A \rightarrow \beta a_{j+1} \bullet \alpha, i, j+1)$ to $\mathcal{C}$

Completion if $(Y, i, j), Y \in N$ and $(A \rightarrow \beta \bullet Y\alpha, j, k) \in \mathcal{C}$, add $(A \rightarrow \beta Y \bullet \alpha, i, k)$

Earley Parsing Example

Earley Parsing Example

○ — *john* — ○ — *saw* — ○ — *the* — ○ — *girl* — ○ — *in* — ○ — *a* — ○ — *car* — ○

S → • NP VP

Earley Parsing Example

○ *john* ○ *saw* ○ *the* ○ *girl* ○ *in* ○ *a* ○ *car* ○

S→ • NP VP
NP→ • D N PP
NP→ • N
NP→ • D N
NP→ • N PP

Earley Parsing Example

○ — *john* — ○ — *saw* — ○ — *the* — ○ — *girl* — ○ — *in* — ○ — *a* — ○ — *car* — ○

S → • NP VP
NP → • D N PP
NP → • N
NP → • D N
NP → • N PP
N → • *john*
N → • *girl*
N → • *car*
D → • *the*
D → • *a*

S → • NP VP

NP → • D N PP

NP → • N

NP → • D N

NP → • N PP

N → • *john*

N → • *girl*

N → • *car*

D → • *the*

D → • *a*

Earley Parsing Example

S → • NP VP

NP → • D N PP

NP → • N

NP → • D N

NP → • N PP

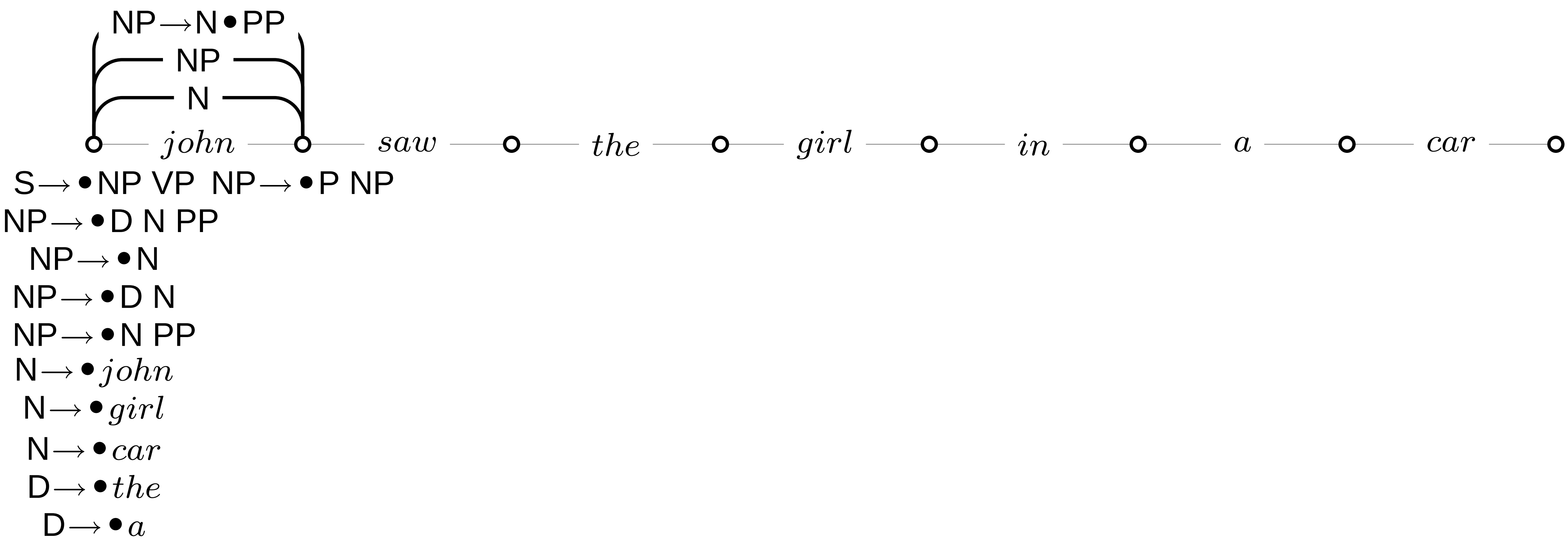
N → • *john*

N → • *girl*

N → • *car*

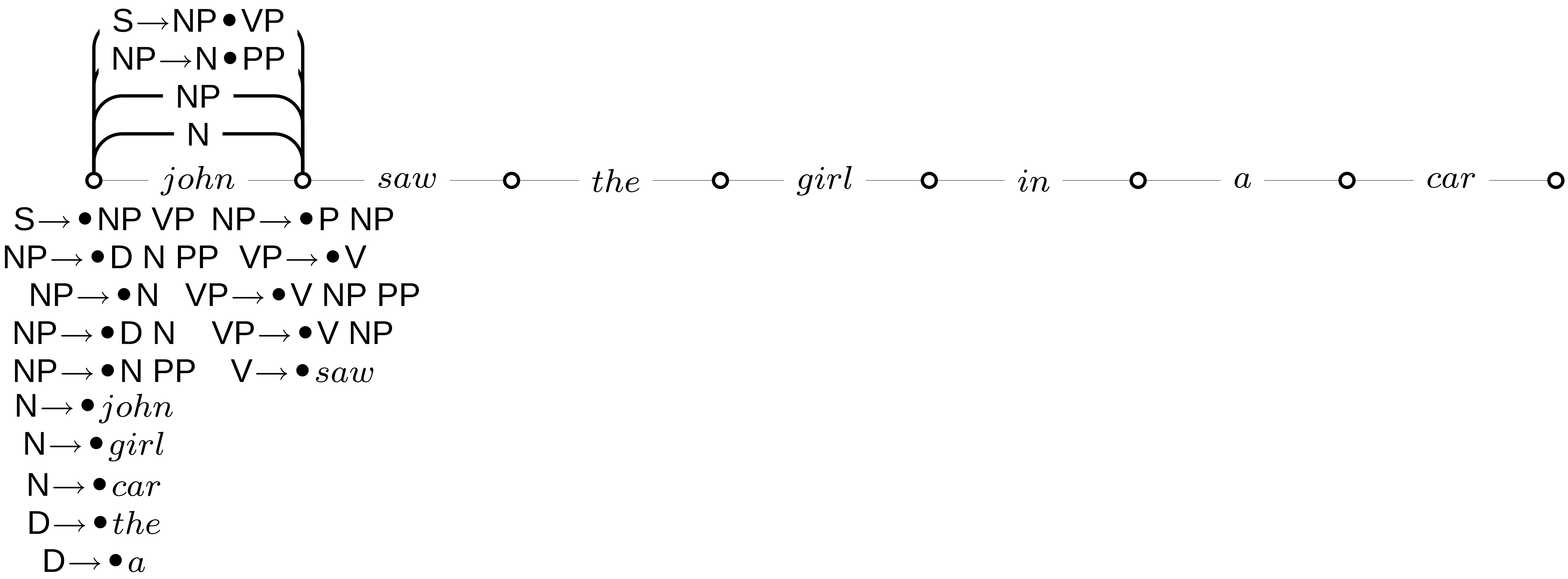
D → • *the*

D → • *a*

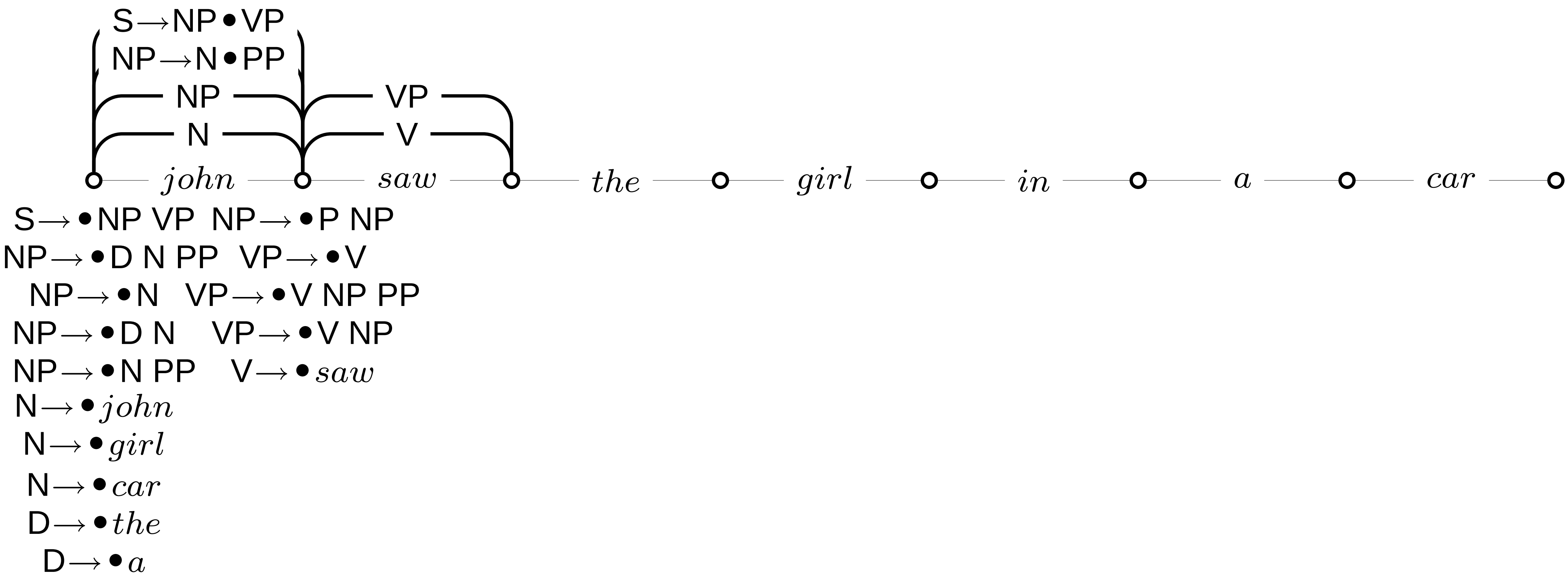


NP→N•PP
NP
N
john
saw
the
girl
in
a
car
S→•NP VP
NP→•P NP
NP→•D N PP
NP→•N
NP→•D N
NP→•N PP
N→•john
N→•girl
N→•car
D→•the
D→•a

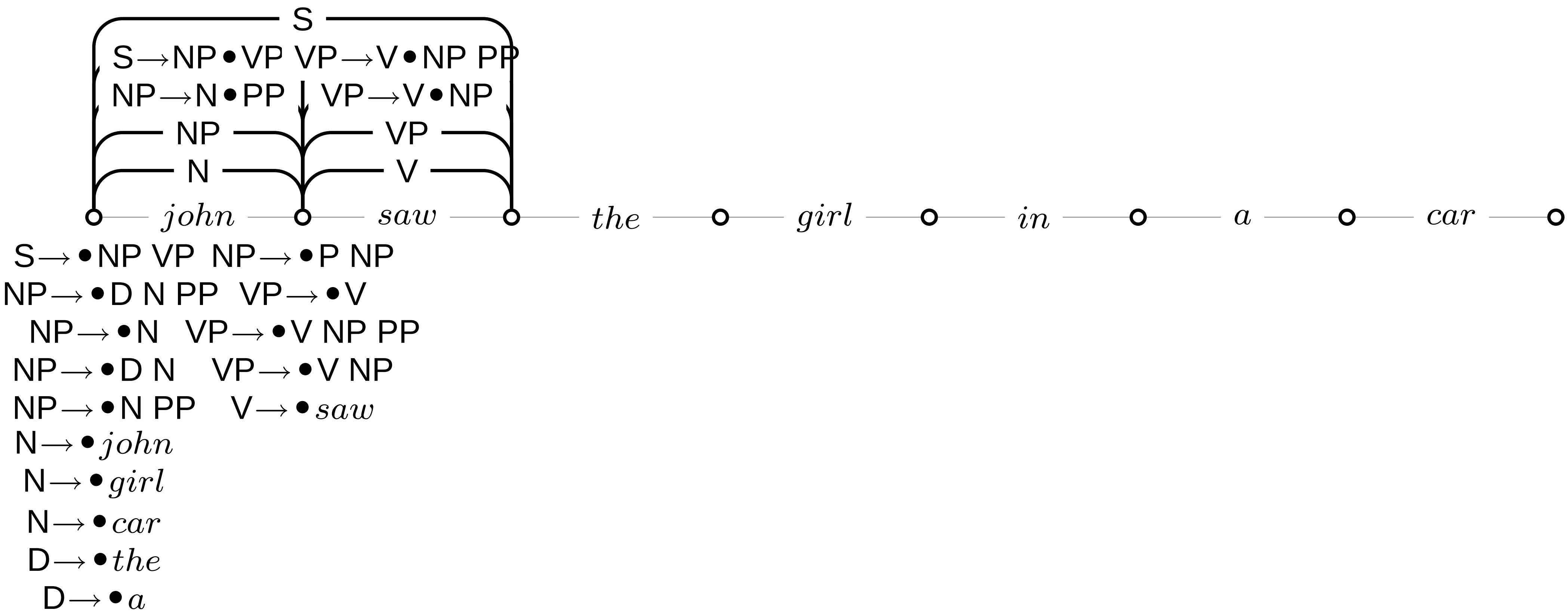
Earley Parsing Example



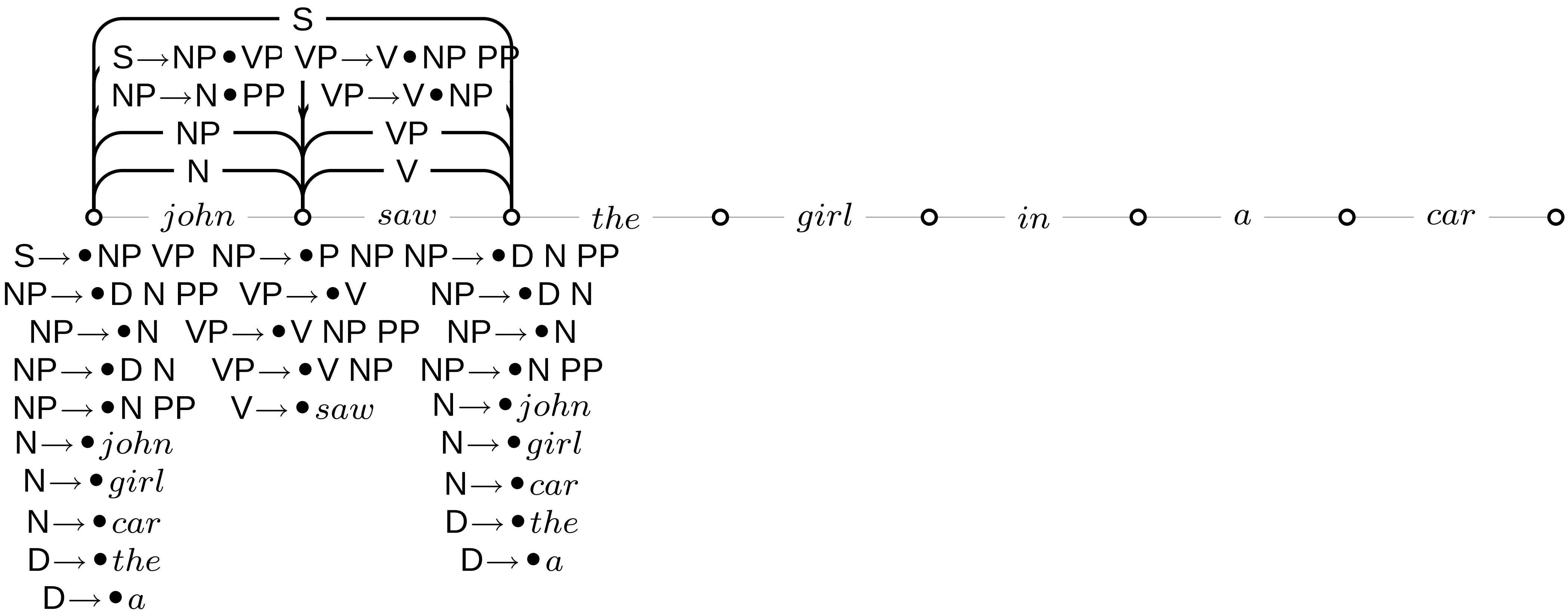
Earley Parsing Example



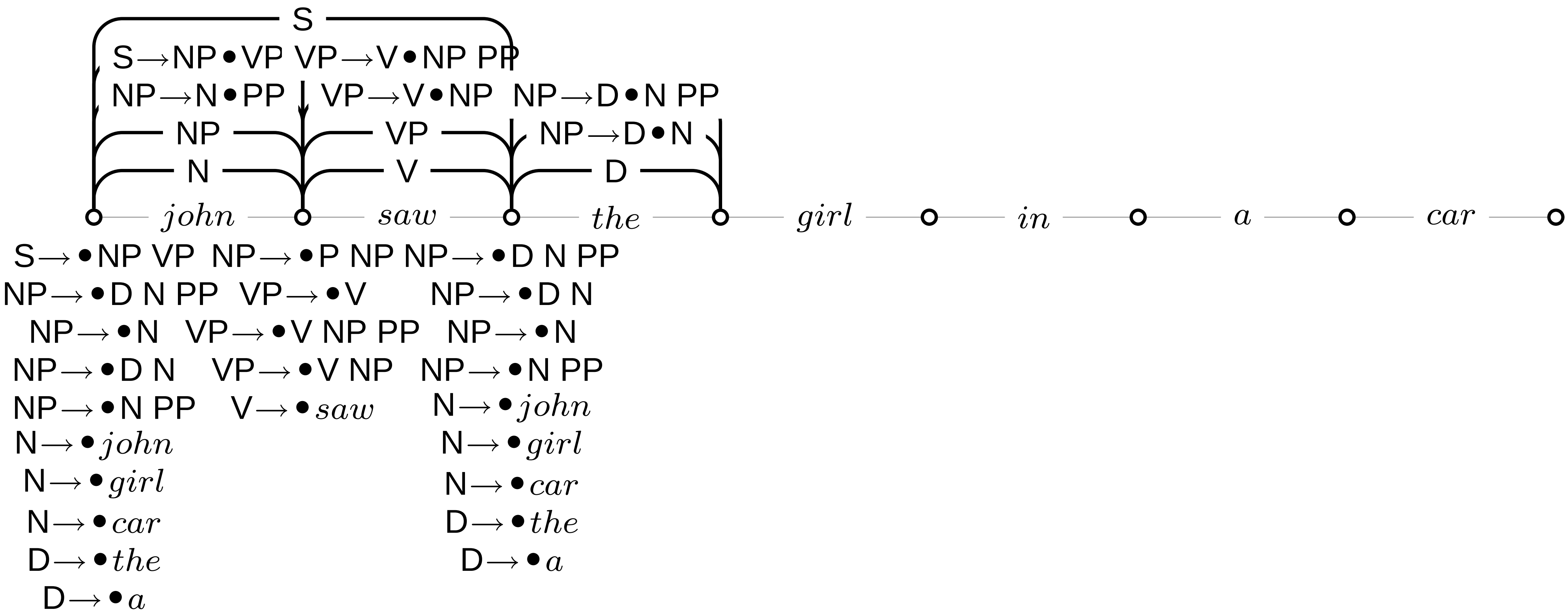
Earley Parsing Example



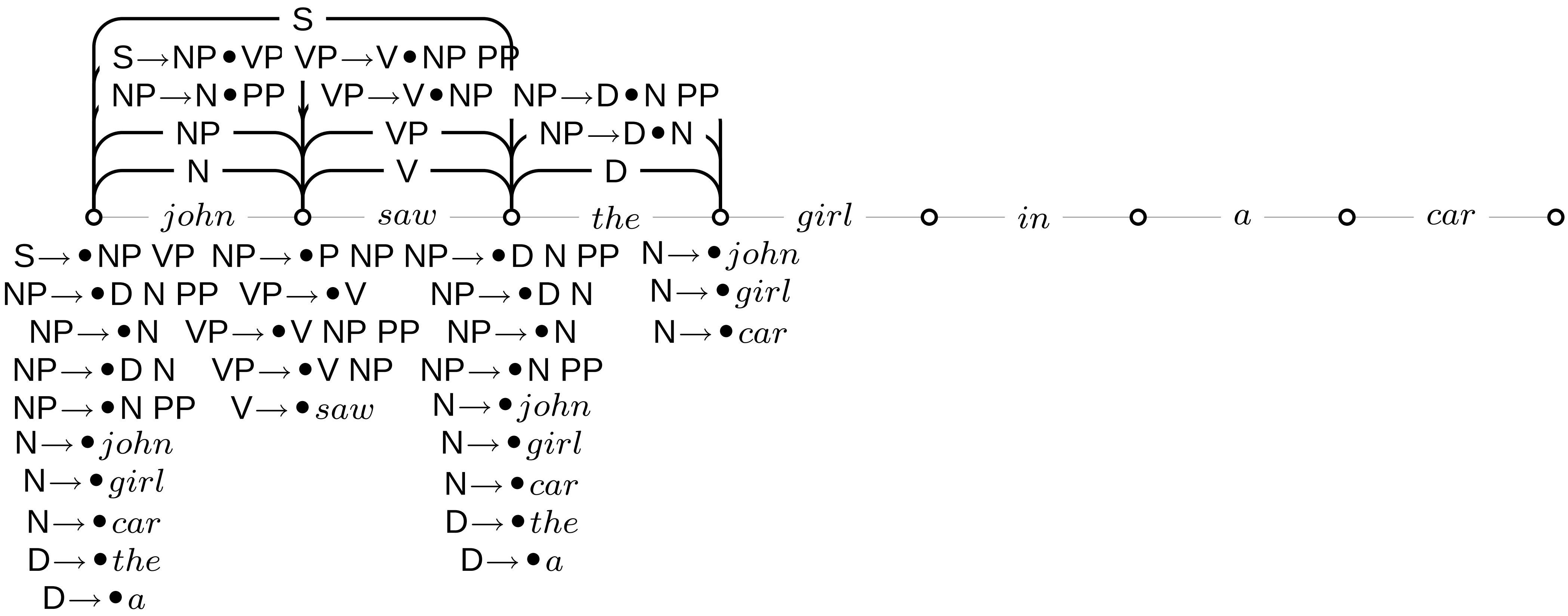
Earley Parsing Example



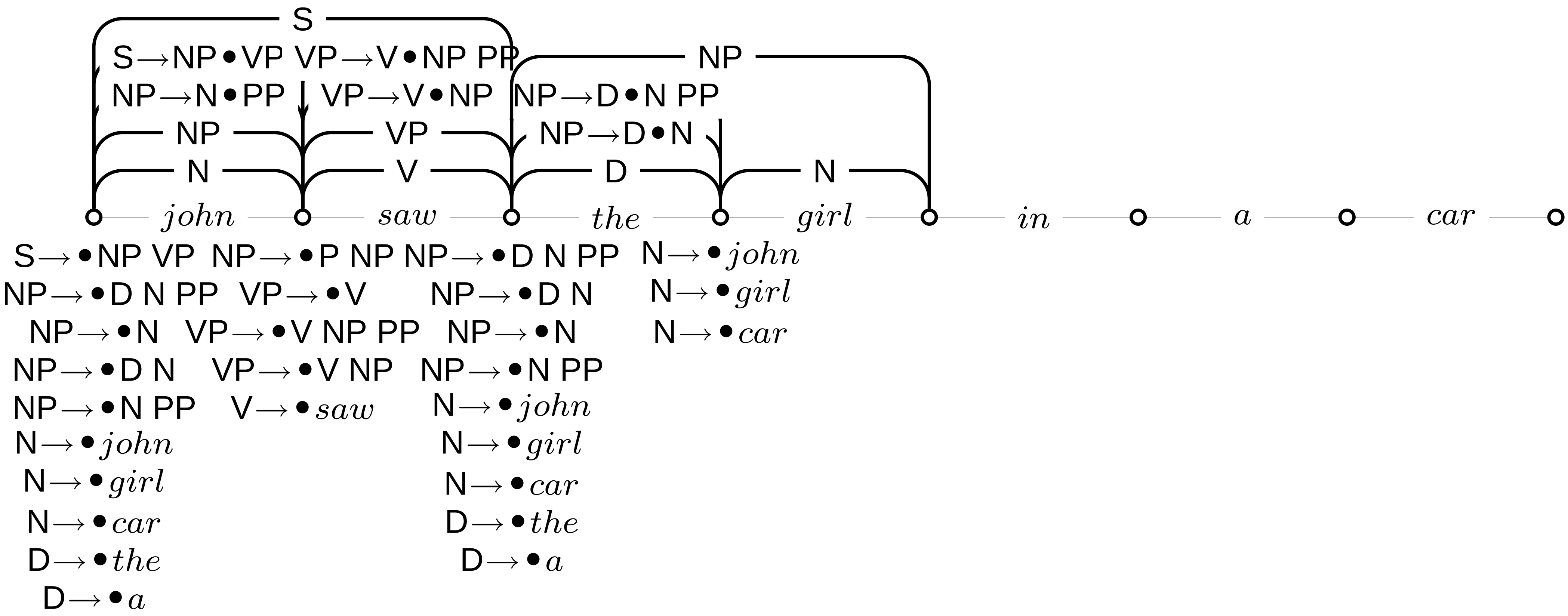
Earley Parsing Example



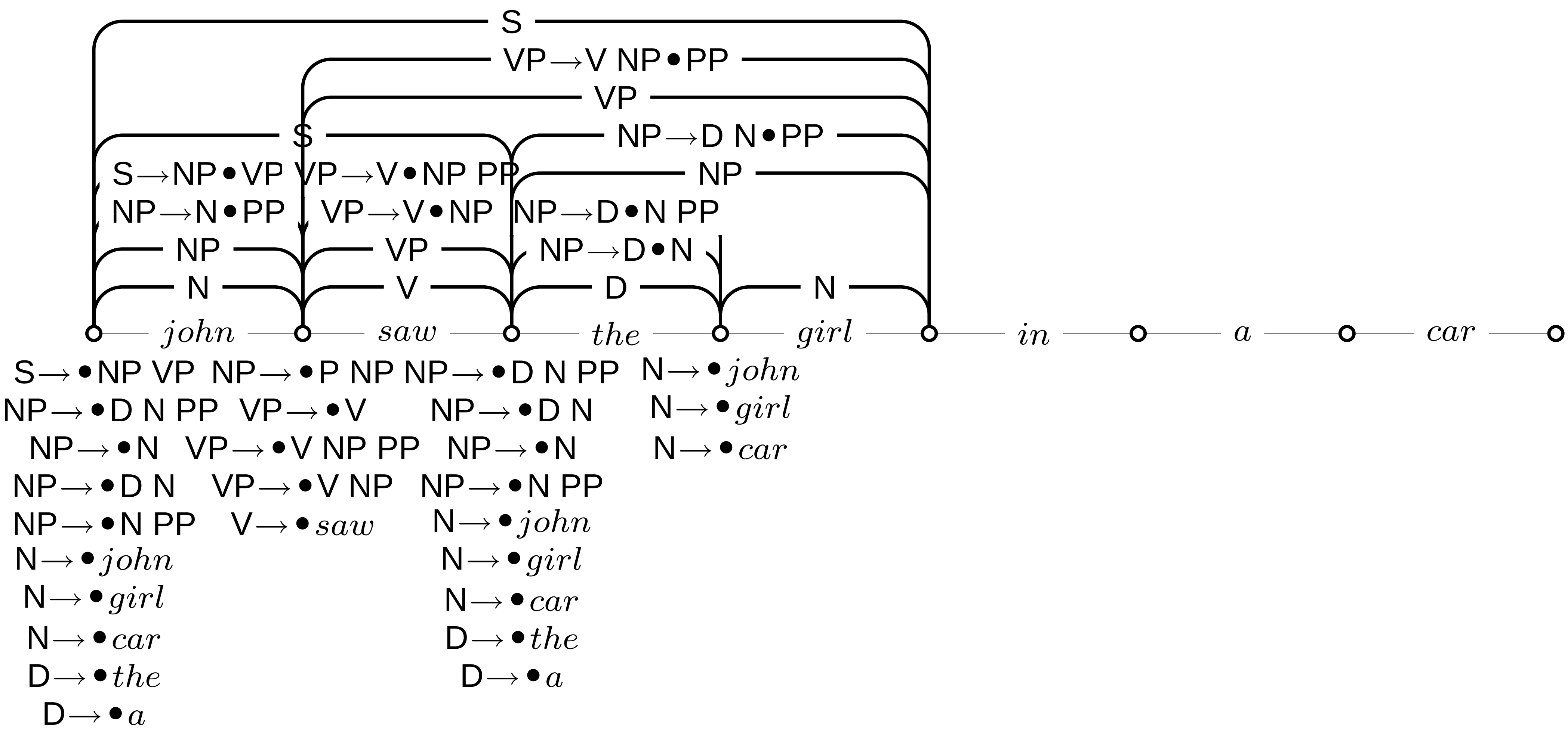
Earley Parsing Example



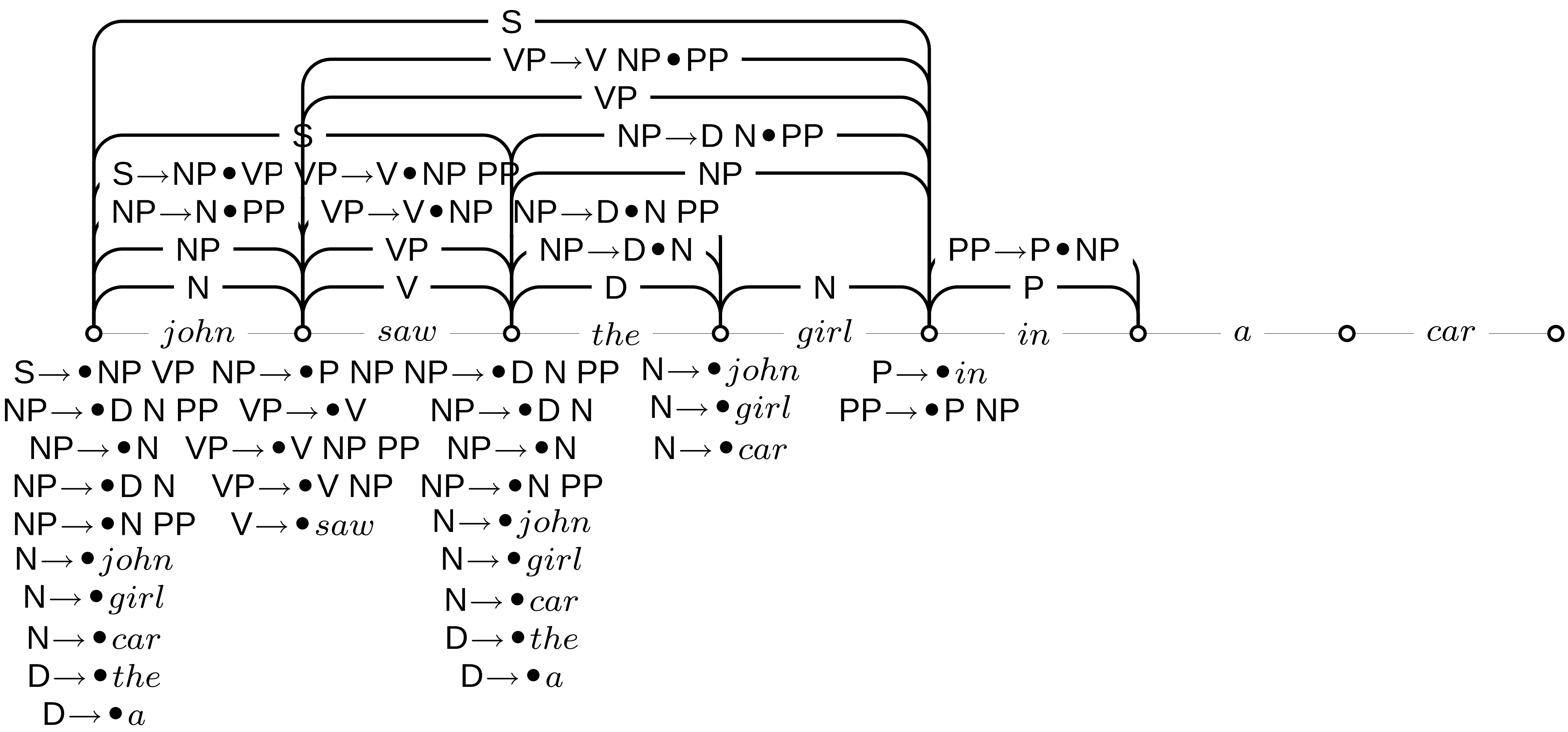
Earley Parsing Example



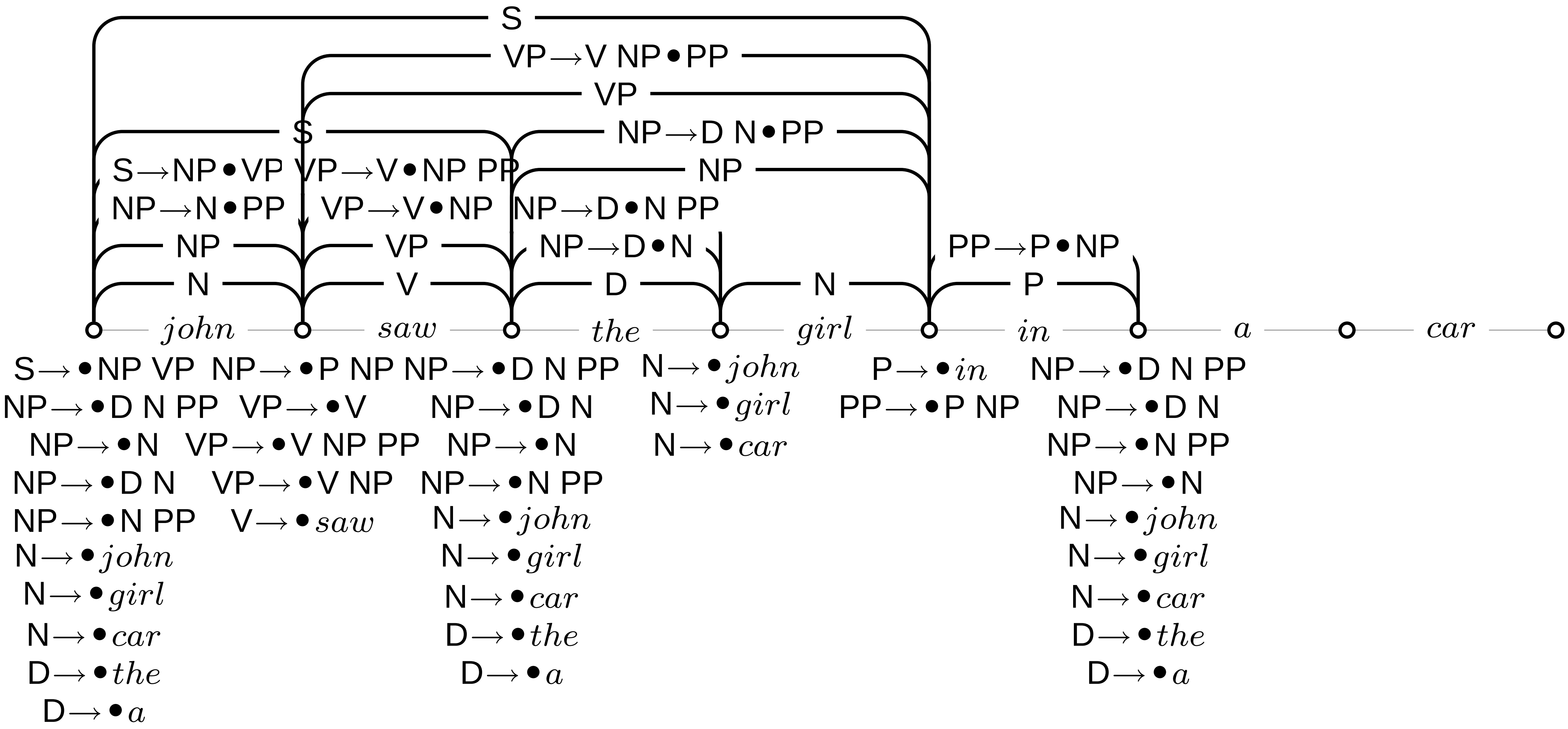
Earley Parsing Example



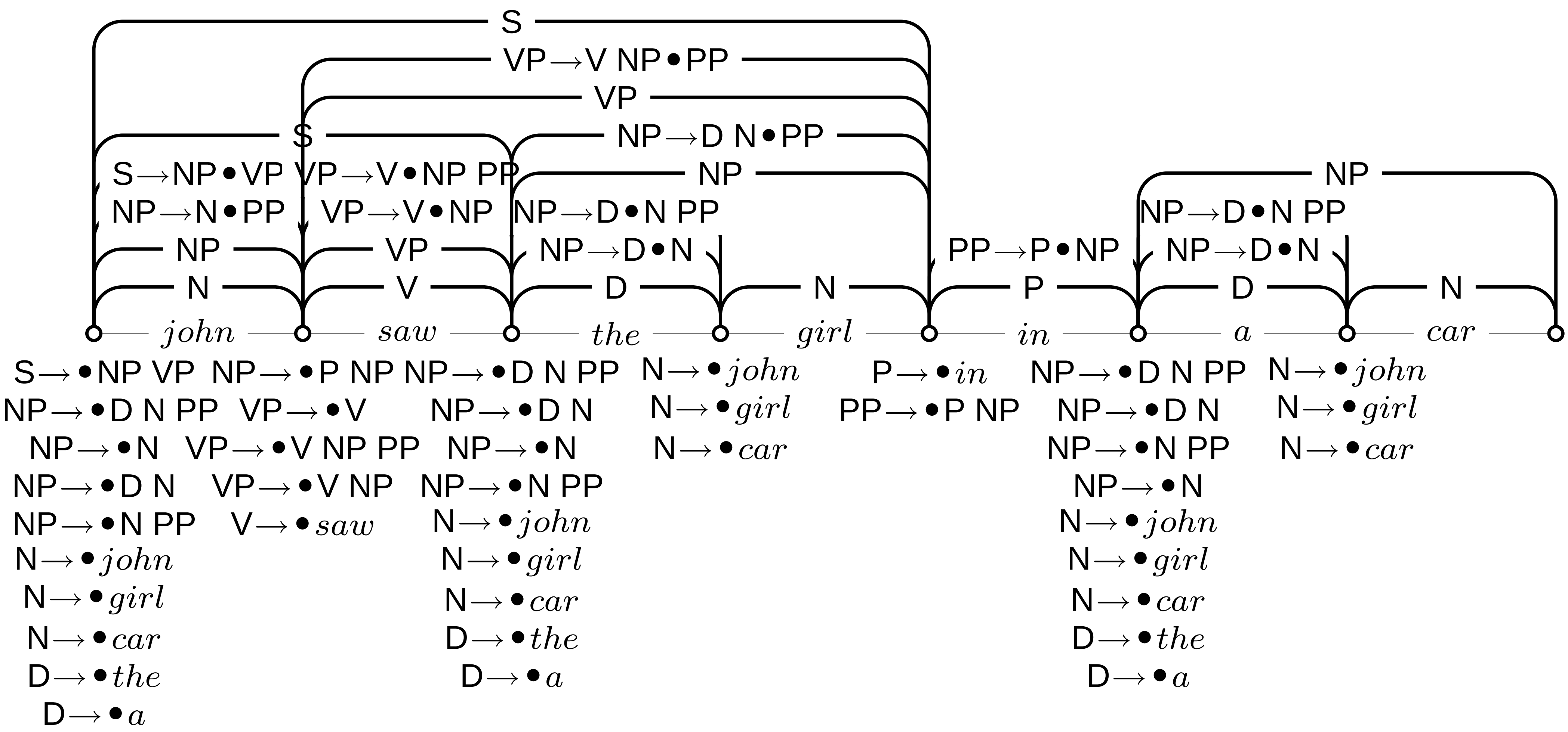
Earley Parsing Example



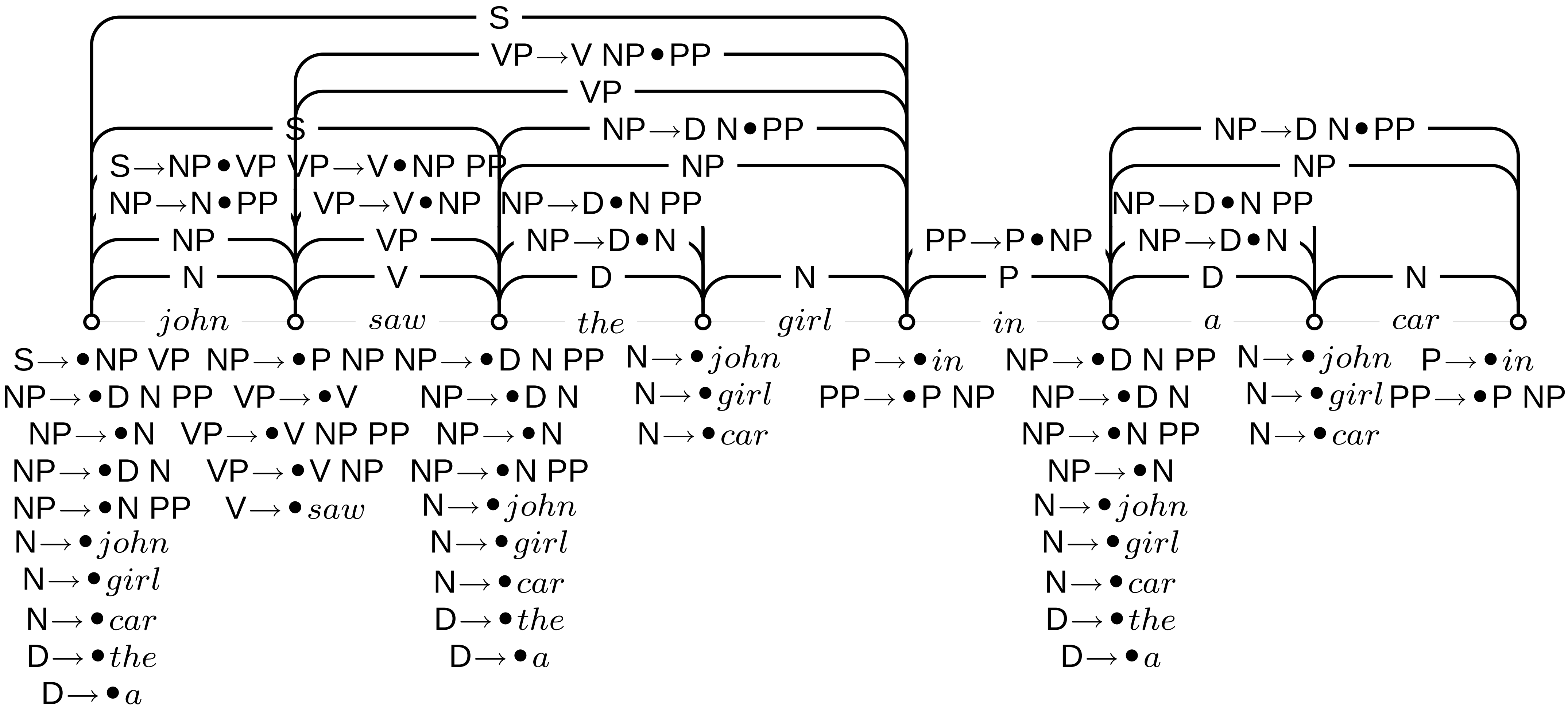
Earley Parsing Example



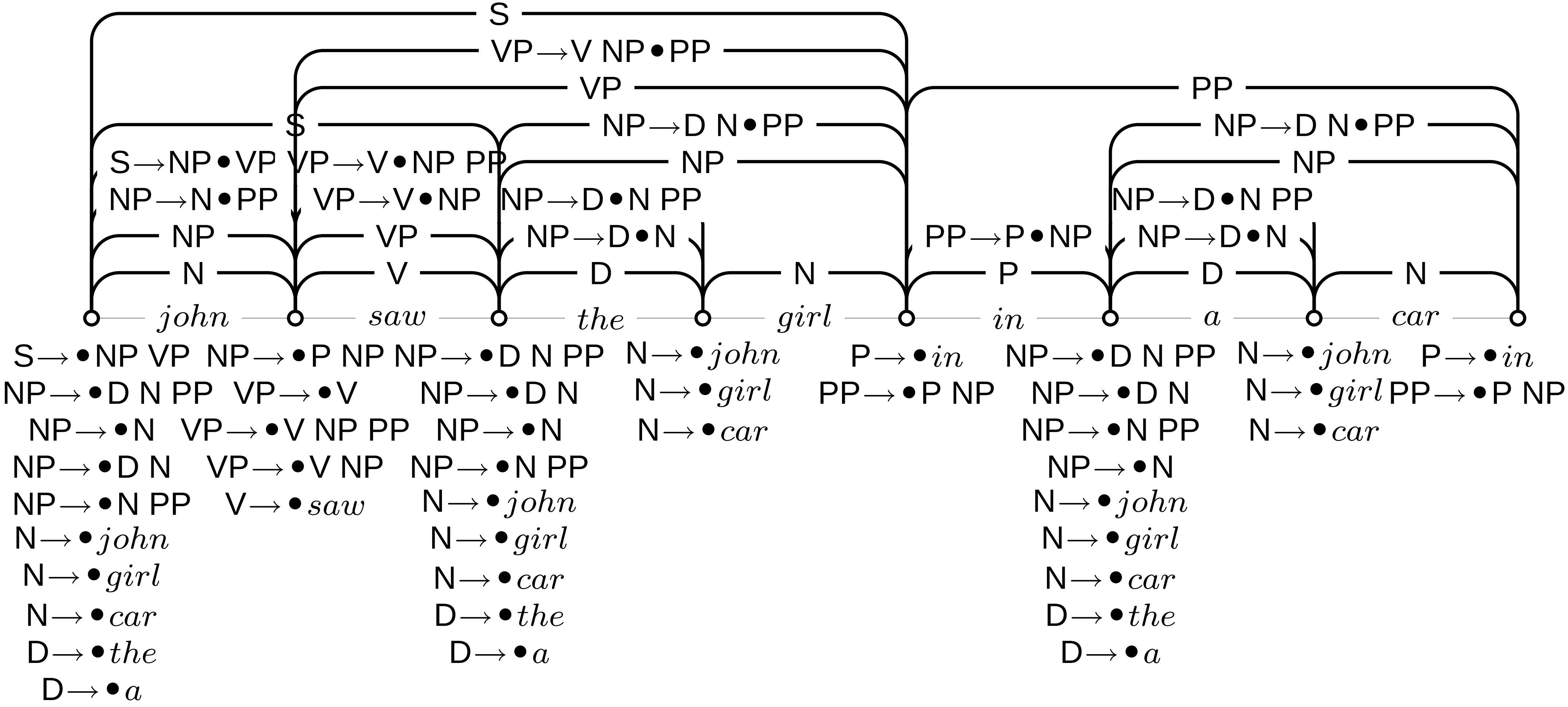
Earley Parsing Example



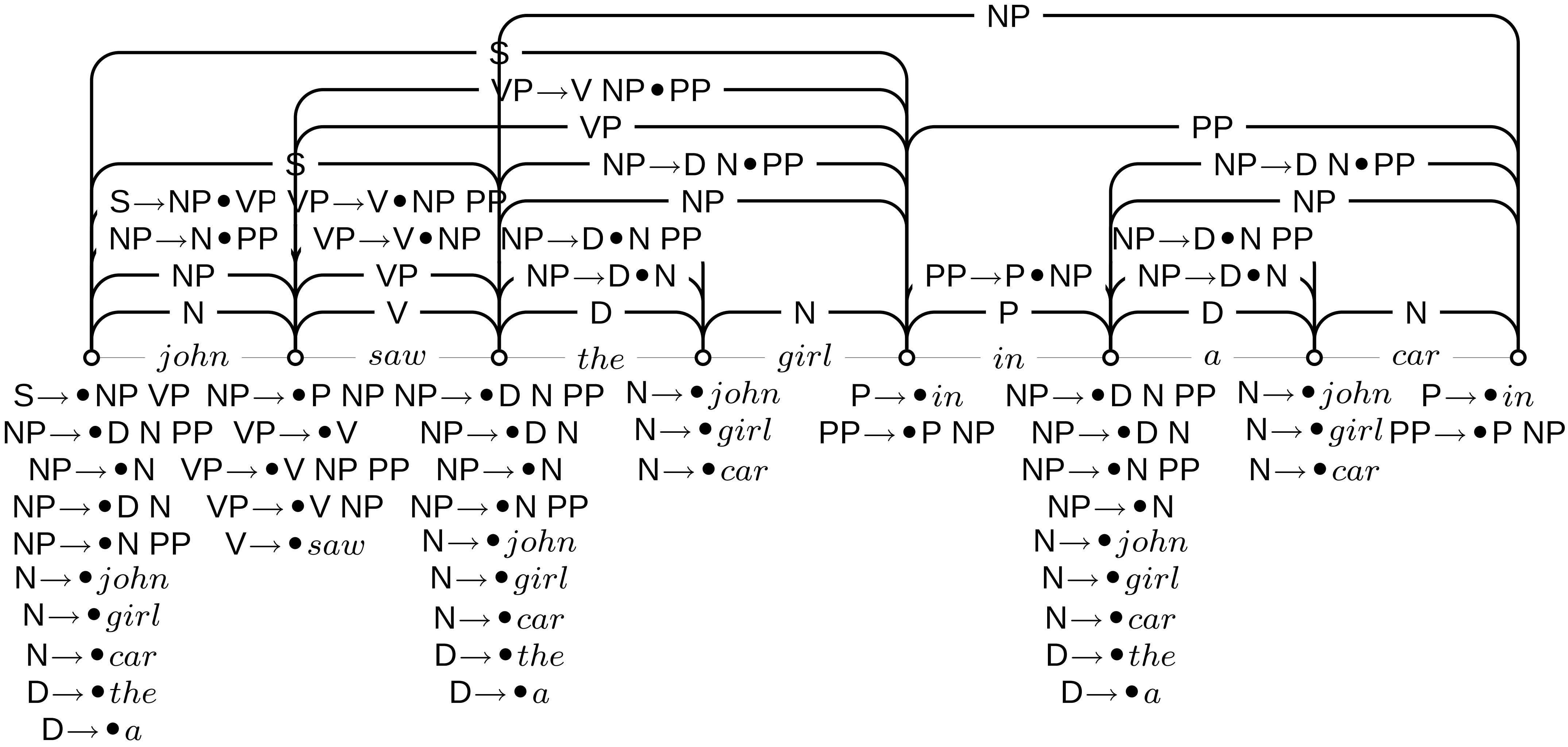
Earley Parsing Example



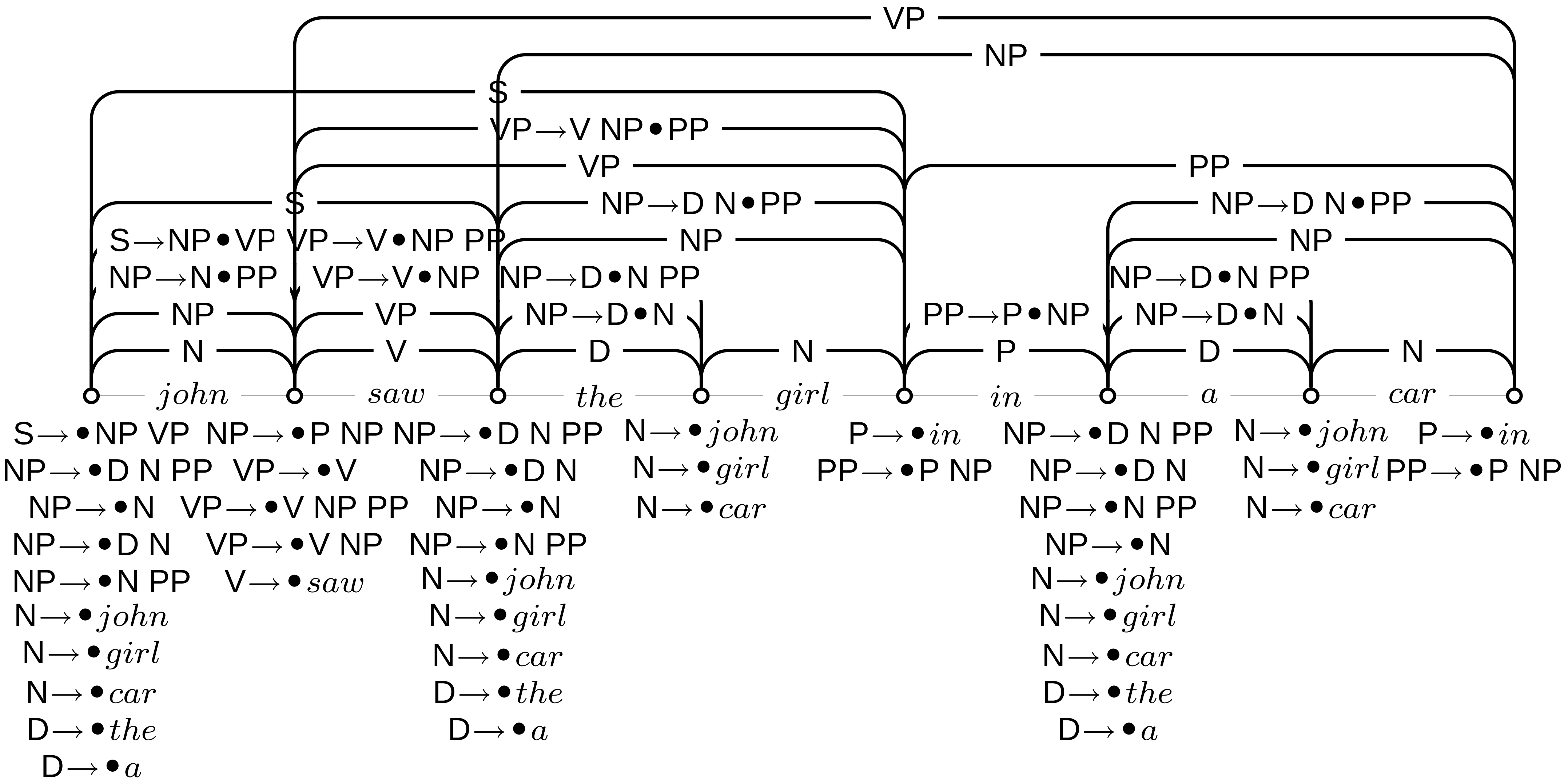
Earley Parsing Example



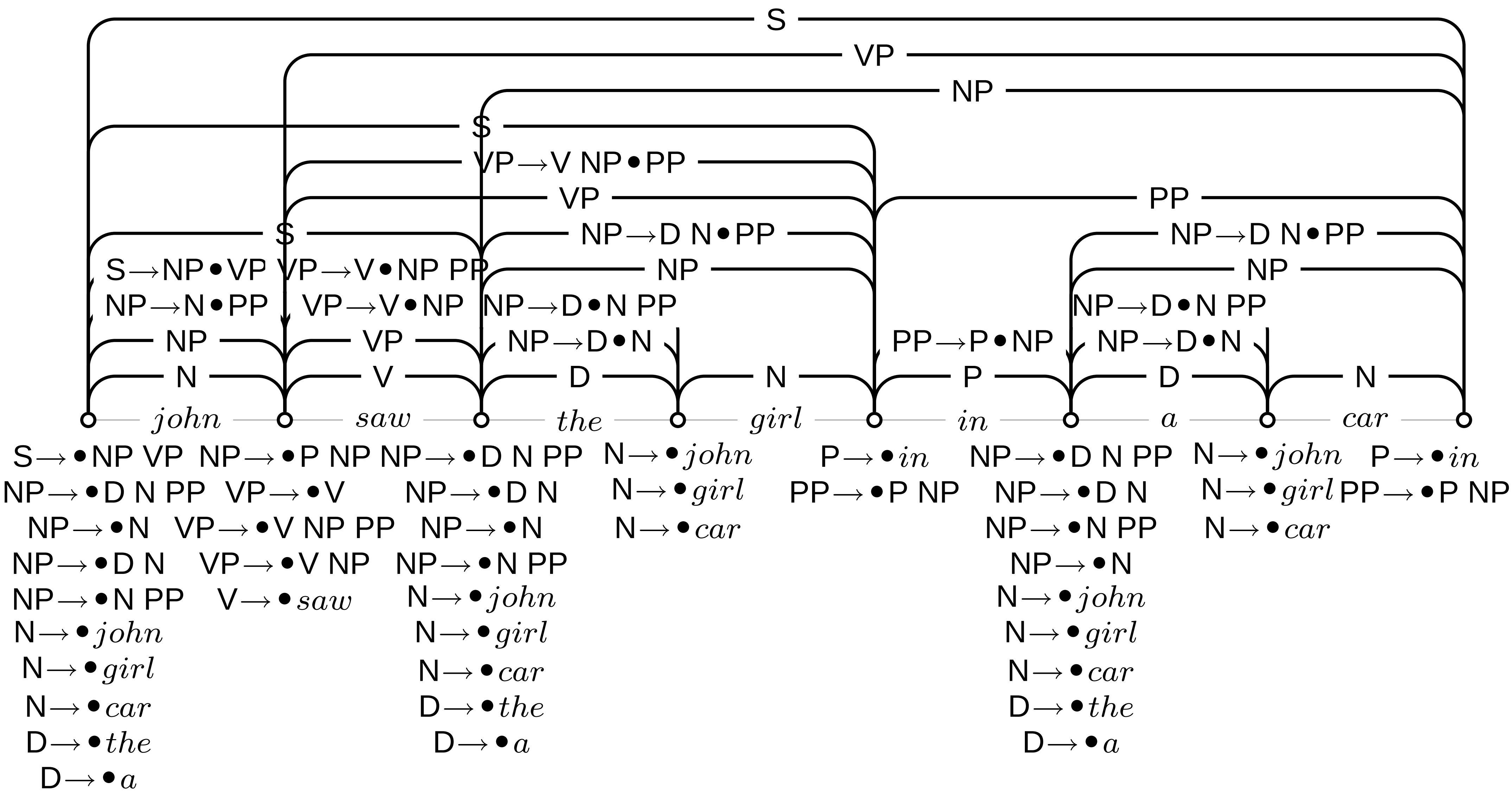
Earley Parsing Example



Earley Parsing Example



Earley Parsing Example



Earley Parsing – Summary

- The number of useless items is reduced
- Superior runtime for unambiguous grammars: $\mathcal{O}(n^2)$
- Valid prefix property
- Not all sub-derivations are computed

Left Corner Parsing

- Observation: Earley parsing predicts items that can not succeed
- Idea: predict only items that can also be derived from the leftmost terminal item
- Formalization: left-corner relation
  - ➤ $A >_l B \Longleftrightarrow \exists \beta : A \to B\beta \in P, B \in \Sigma \cup N$
  - ➤ $A >_l^*$ is the transitive closure of $>_l$
- Reformulation of the prediction step:
  - ➤ If $(A \to \beta \bullet Y\alpha, i, j)$ and $(B, j, k) \in \mathcal{C}$, with $B \in \Sigma \cup N$ add $(C \to B \bullet \gamma, j, k)$ if $Y >_l^* C$
- This formulation also avoids the zero-length predictions with the dot in initial position

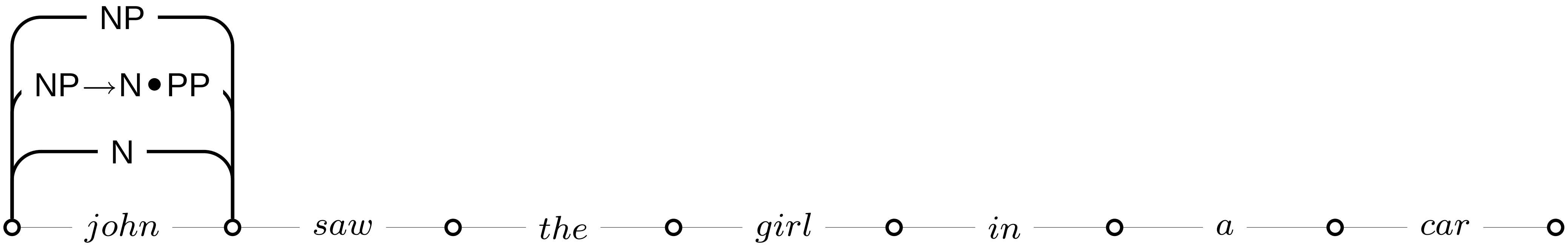
Left-Corner Example

$\circ$ — *john* — $\circ$ — *saw* — $\circ$ — *the* — $\circ$ — *girl* — $\circ$ — *in* — $\circ$ — *a* — $\circ$ — *car* — $\circ$

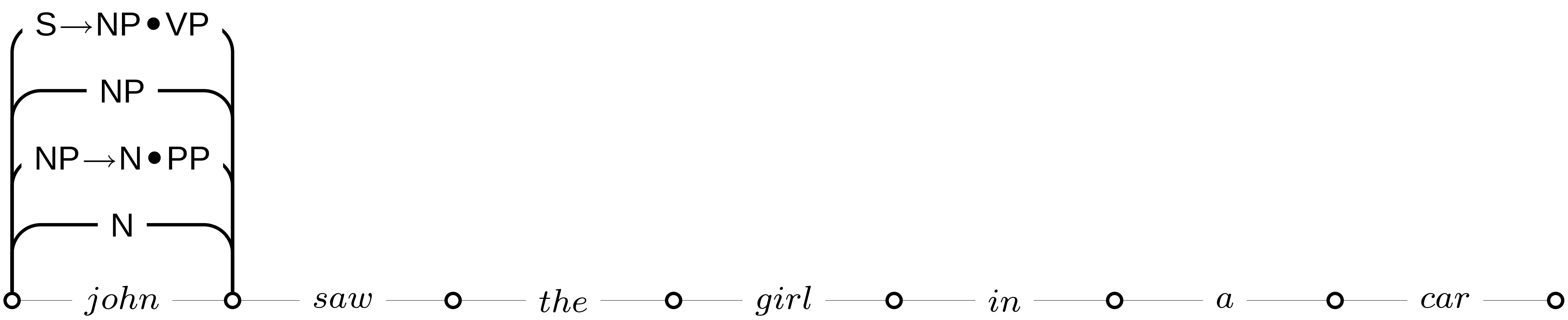
Left-Corner Example

Left-Corner Example

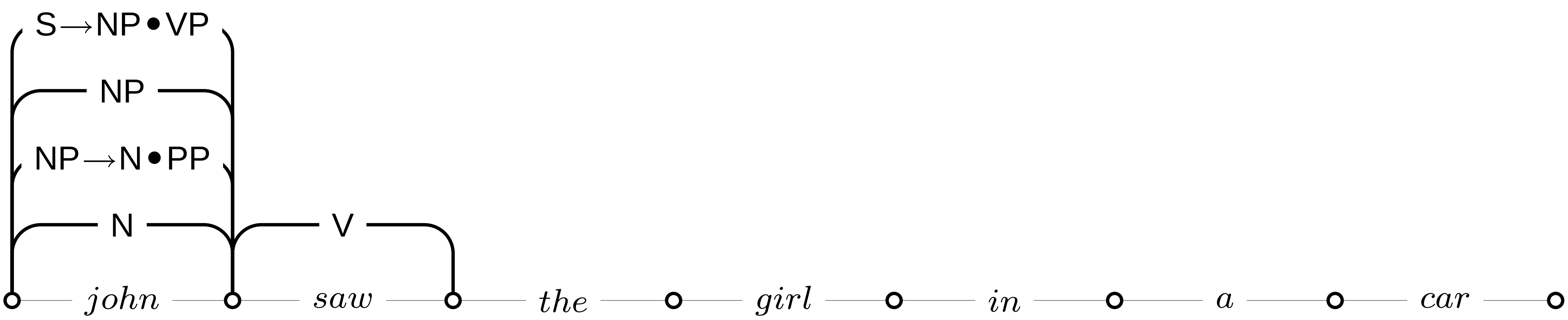
Left-Corner Example

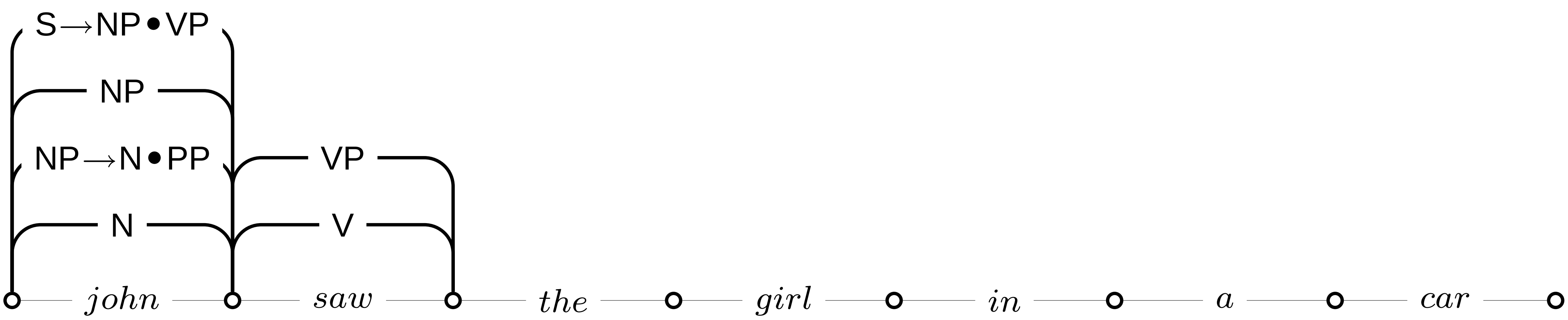


Left-Corner Example



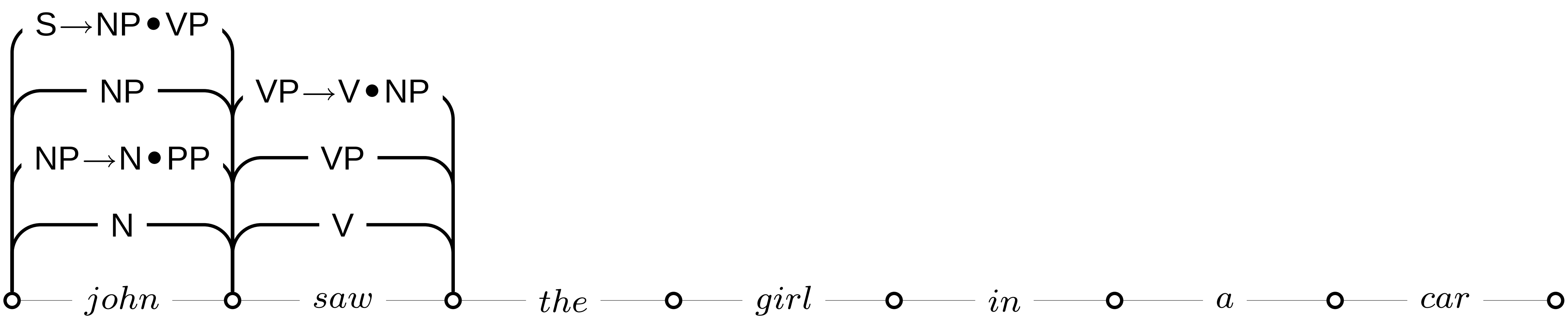
Left-Corner Example



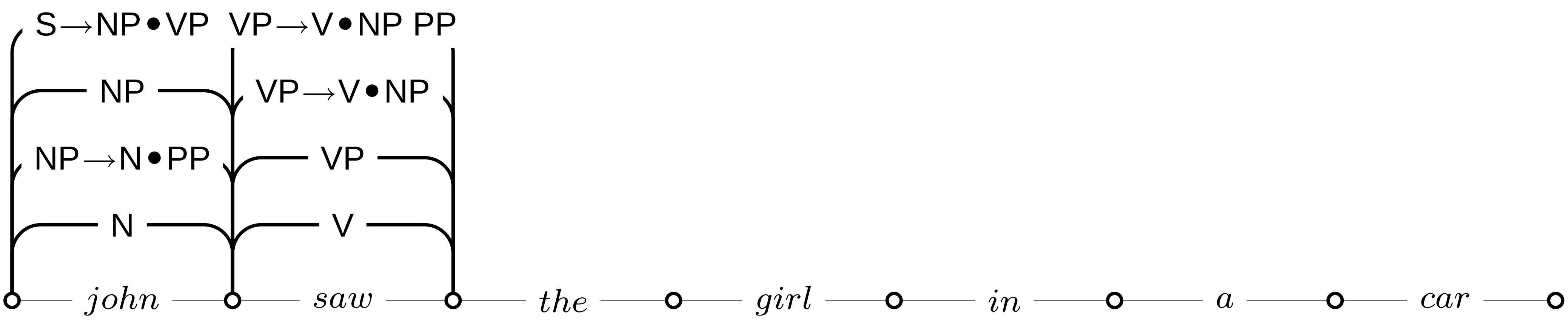


S→NP•VP
NP
NP→N•PP
VP
N
V
john
saw
the
girl
in
a
car

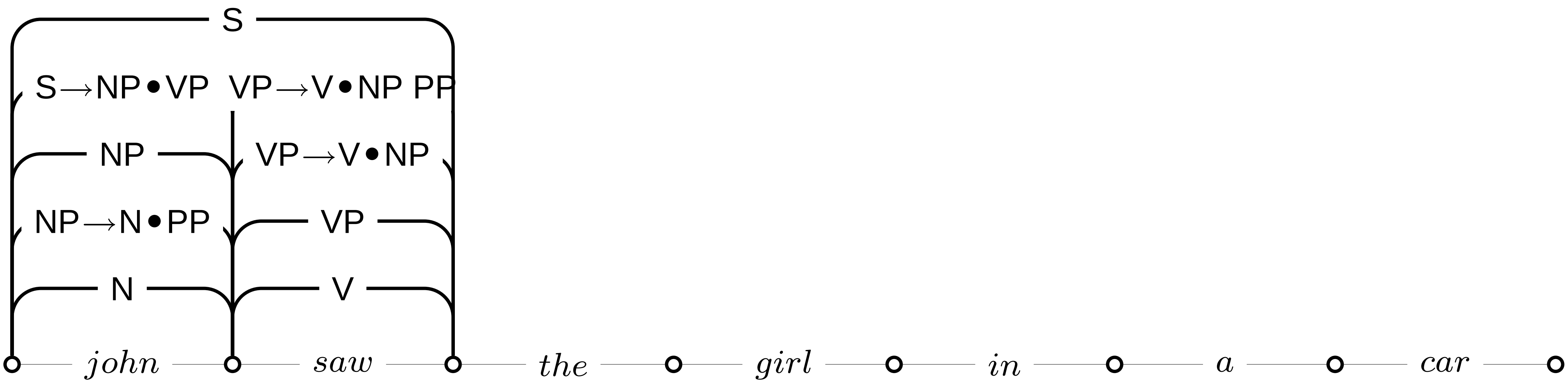
Left-Corner Example



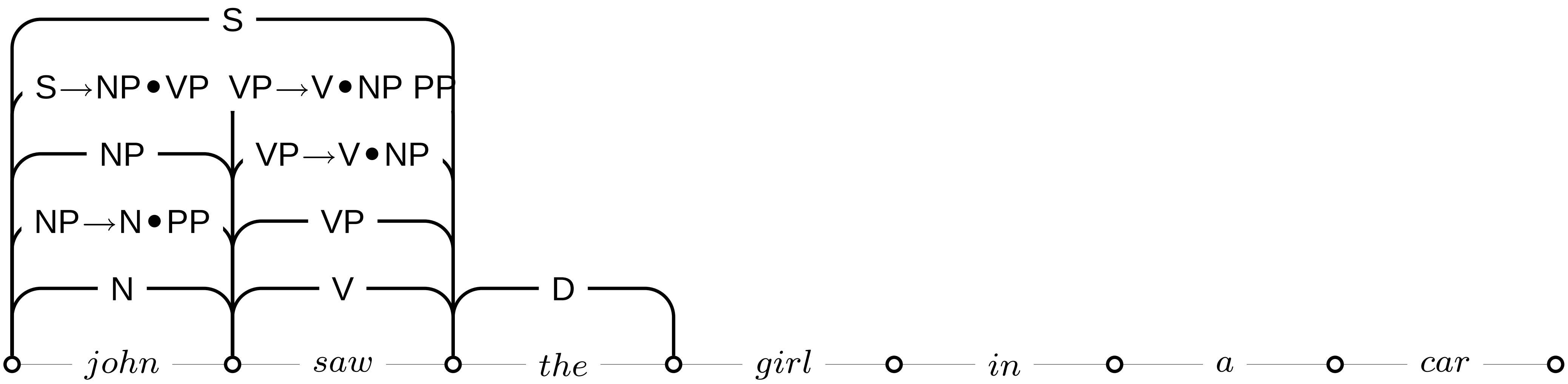
Left-Corner Example



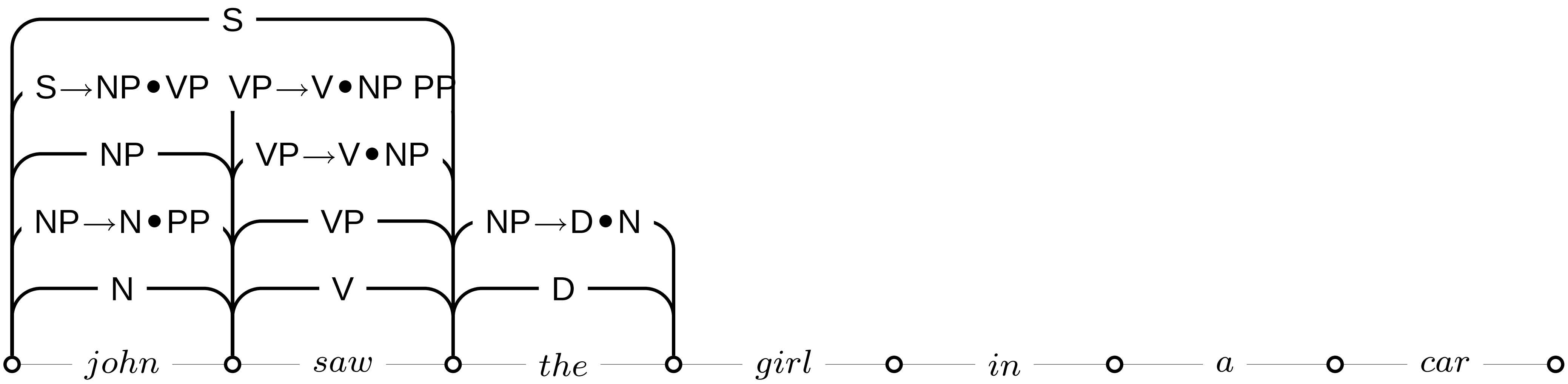
Left-Corner Example



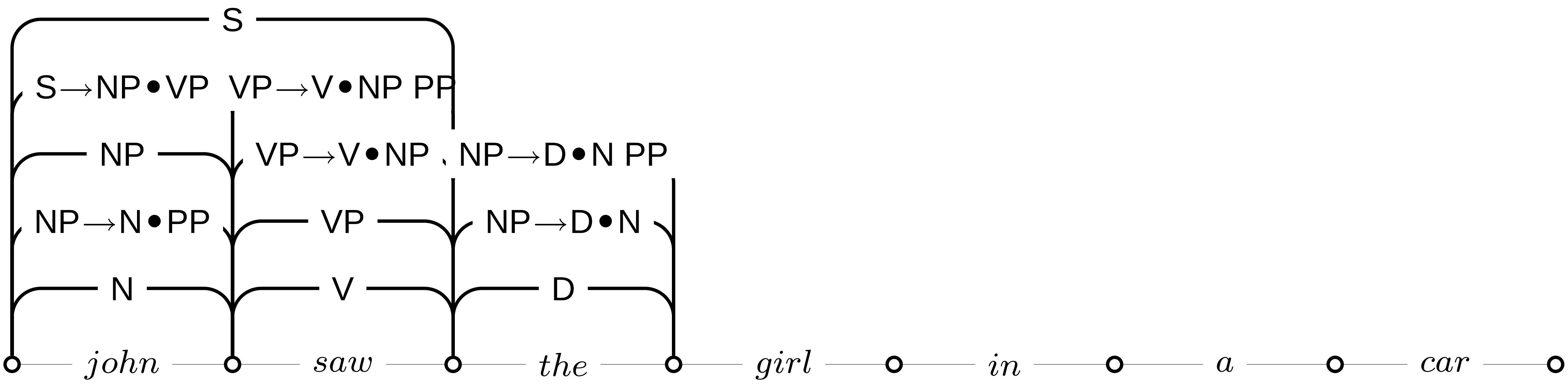
Left-Corner Example



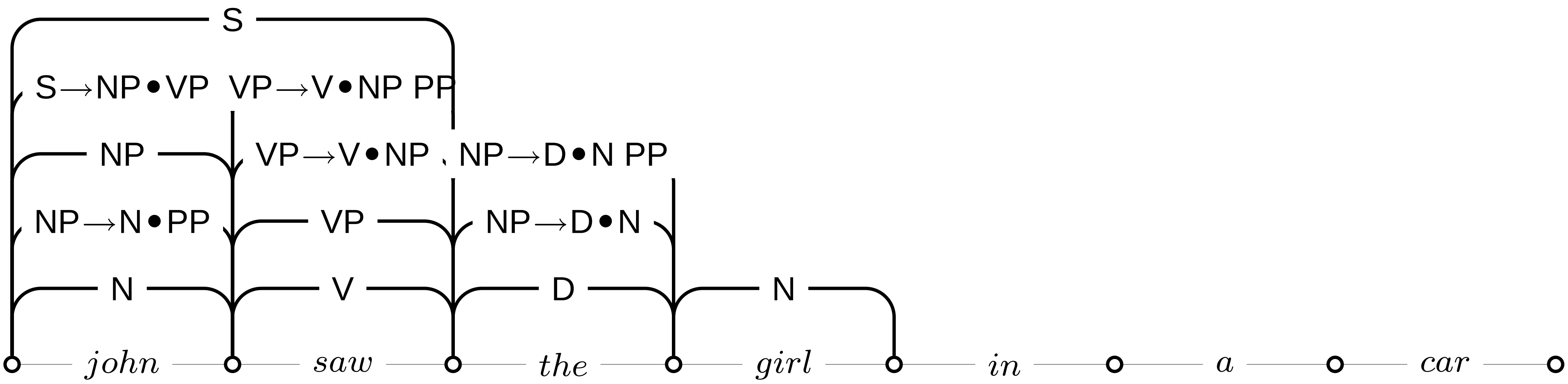
Left-Corner Example

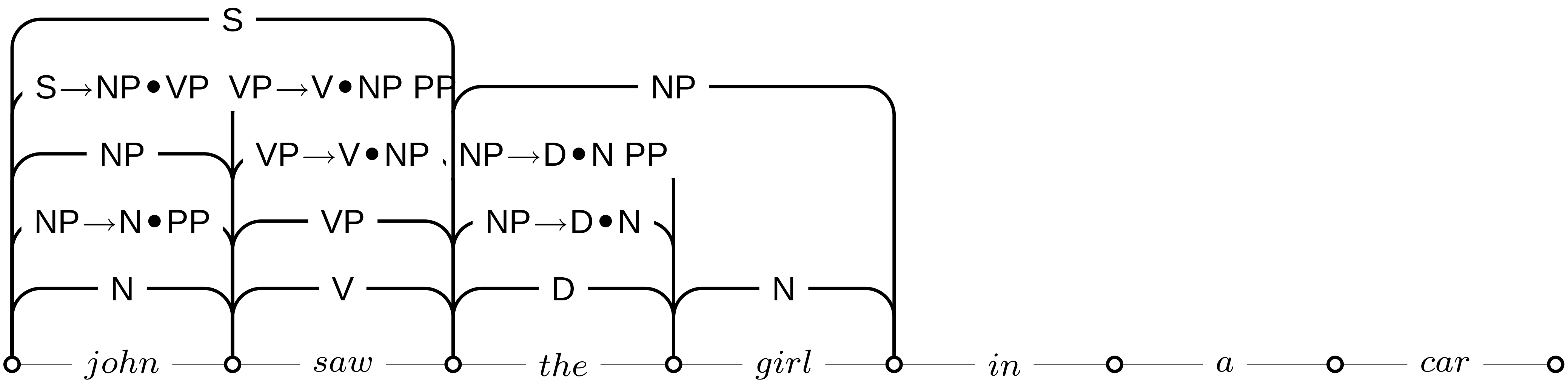


Left-Corner Example

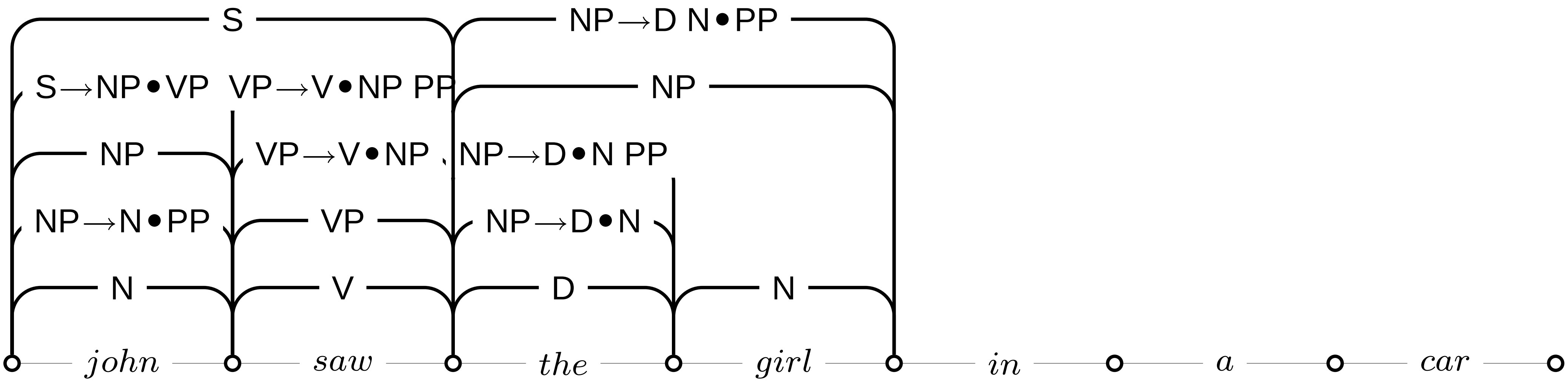


Left-Corner Example

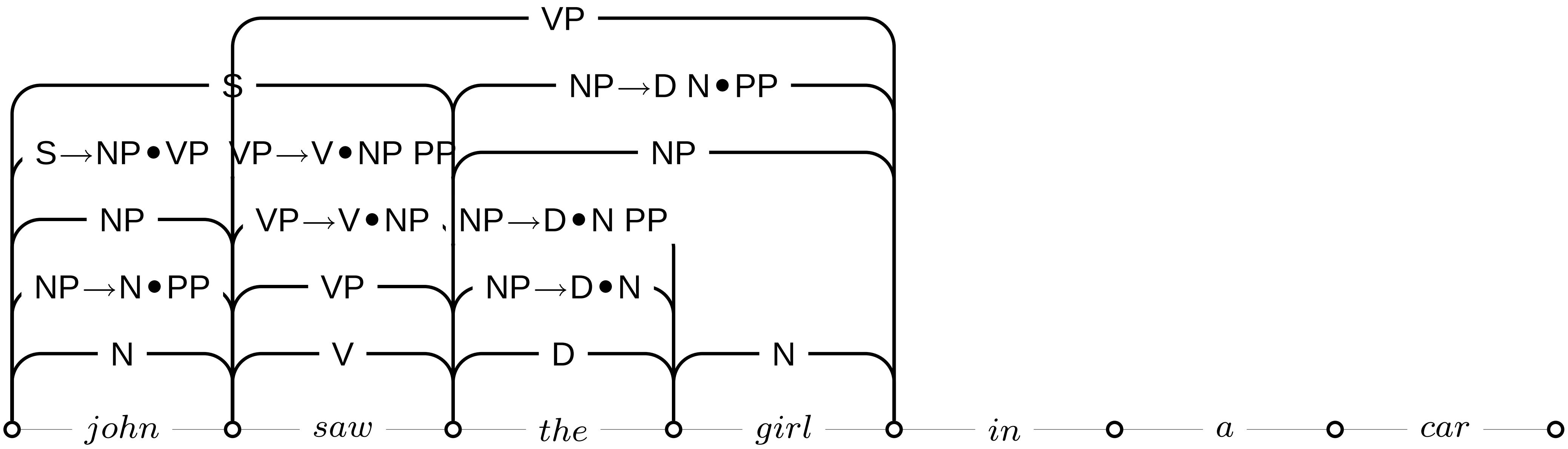



S
S→NP•VP
VP→V•NP PP
NP
NP
VP→V•NP
NP→D•N PP
NP→N•PP
VP
NP→D•N
N
V
D
N
john
saw
the
girl
in
a
car

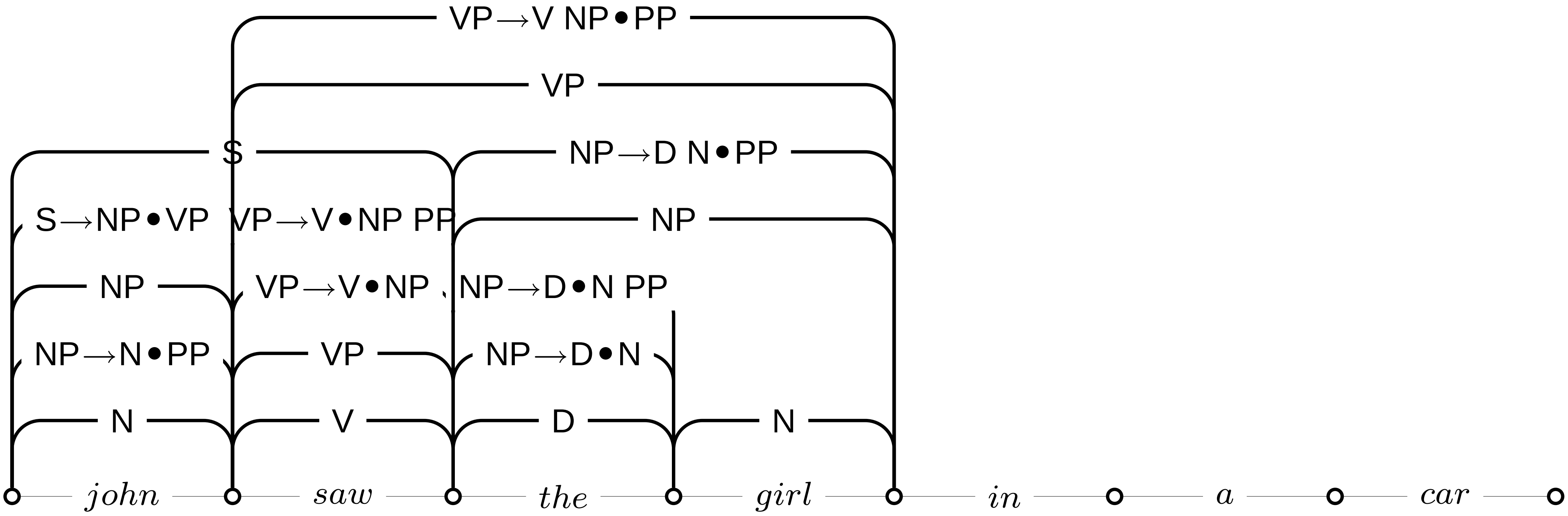
Left-Corner Example



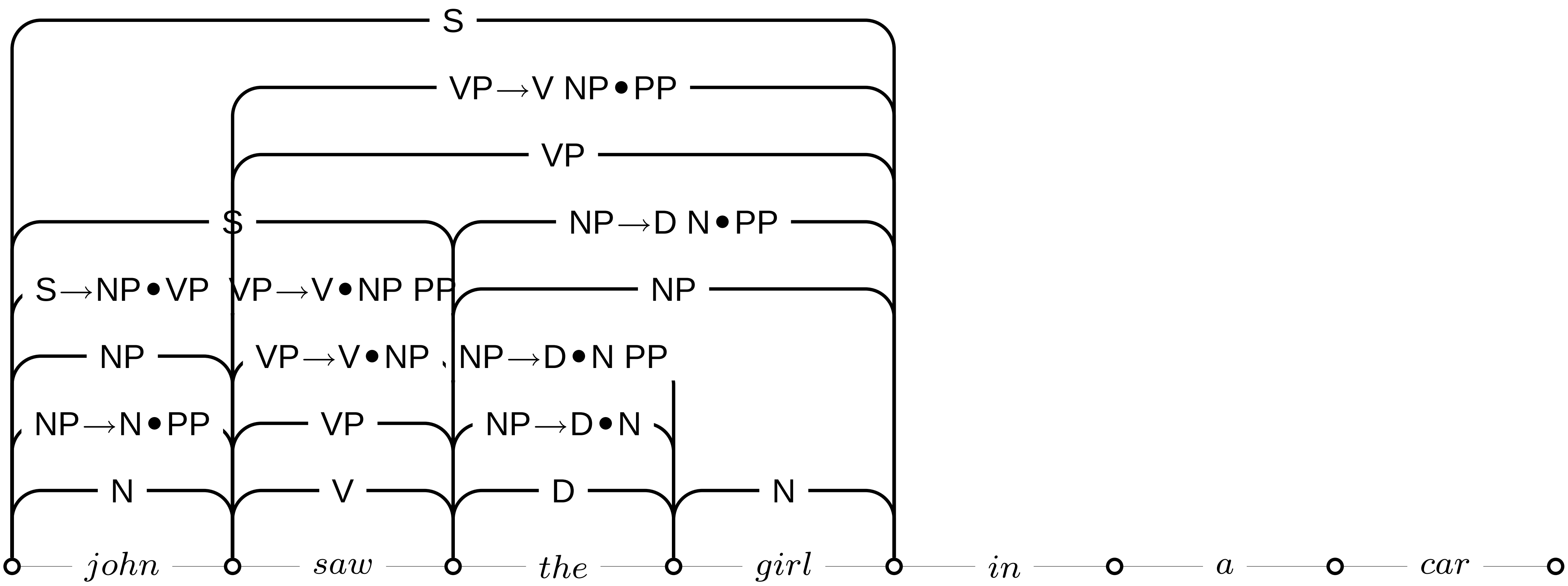
Left-Corner Example



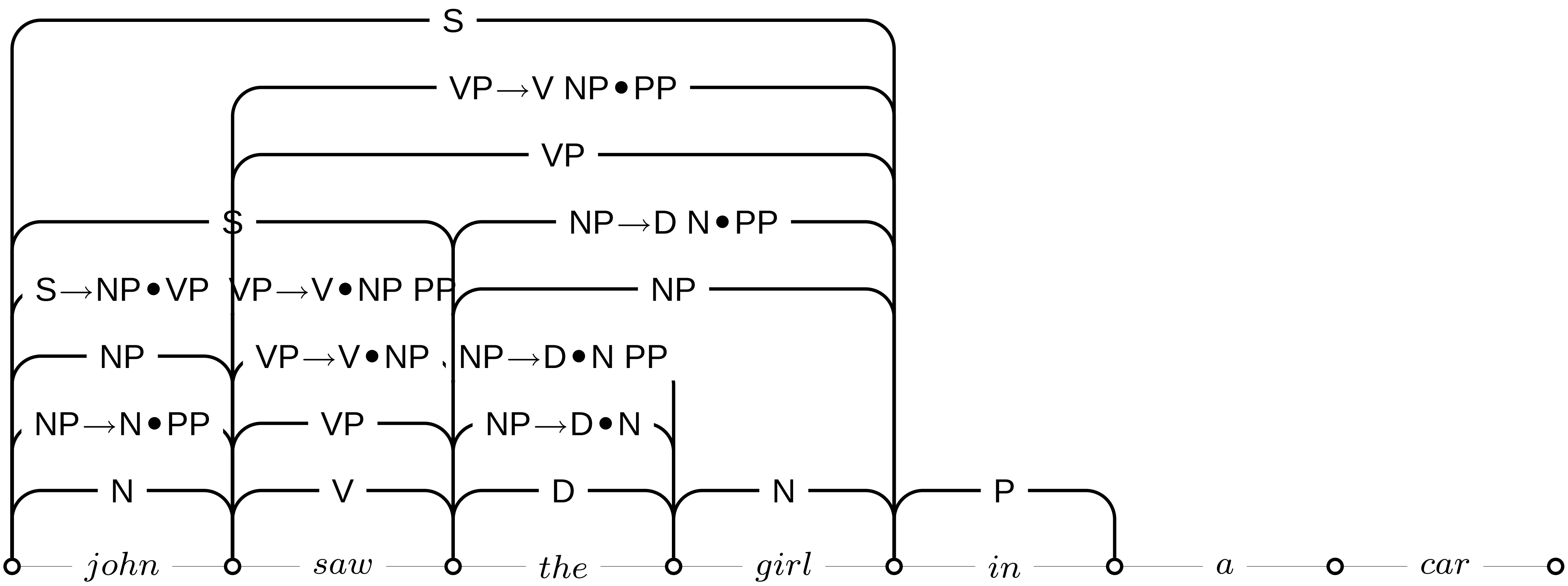
Left-Corner Example



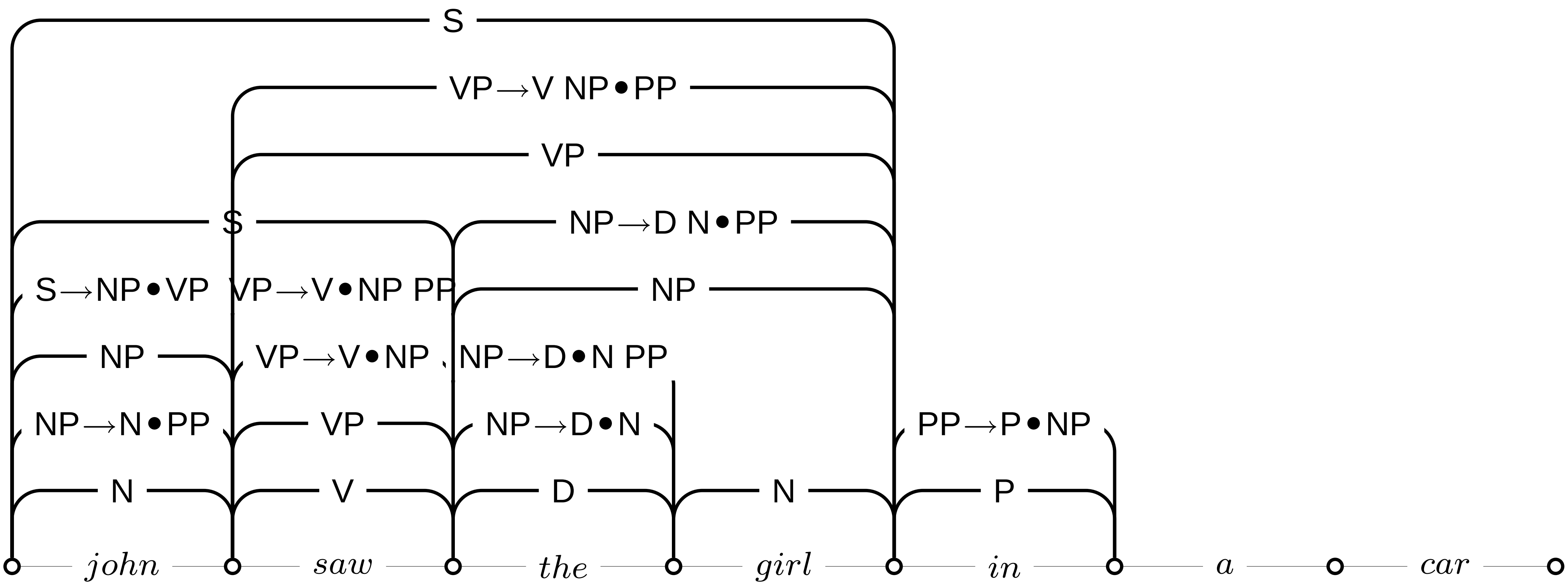
Left-Corner Example



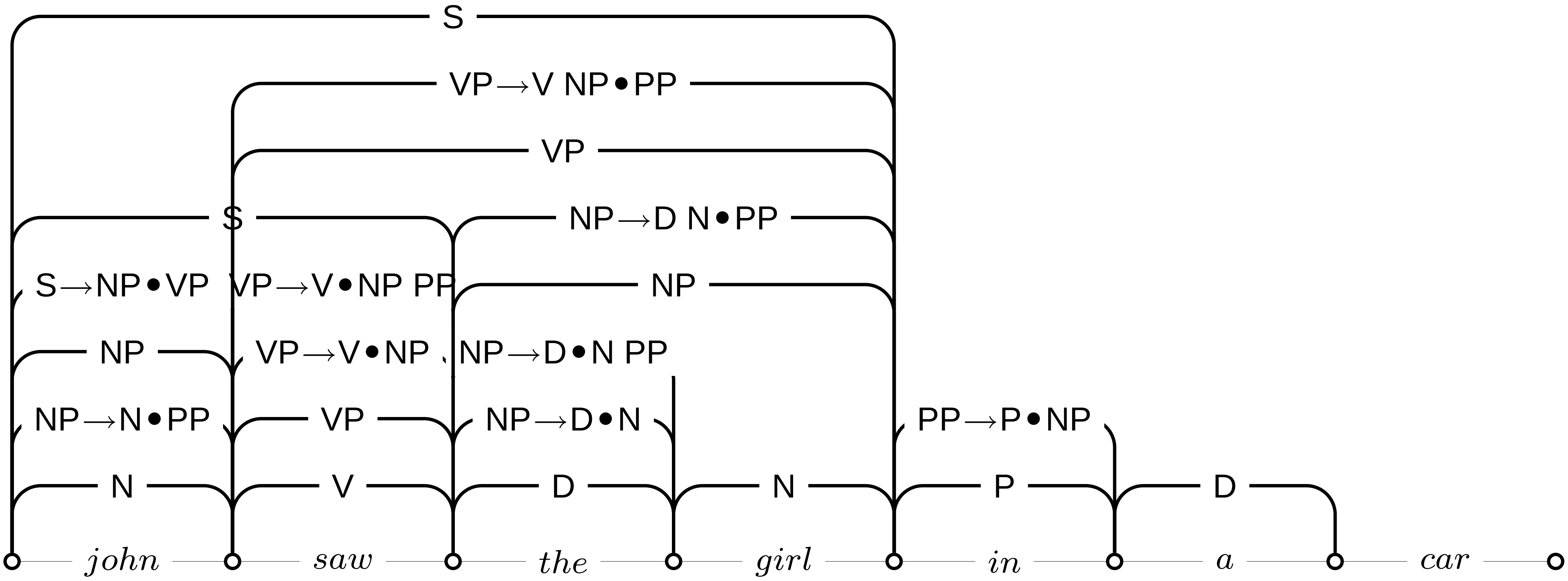
Left-Corner Example



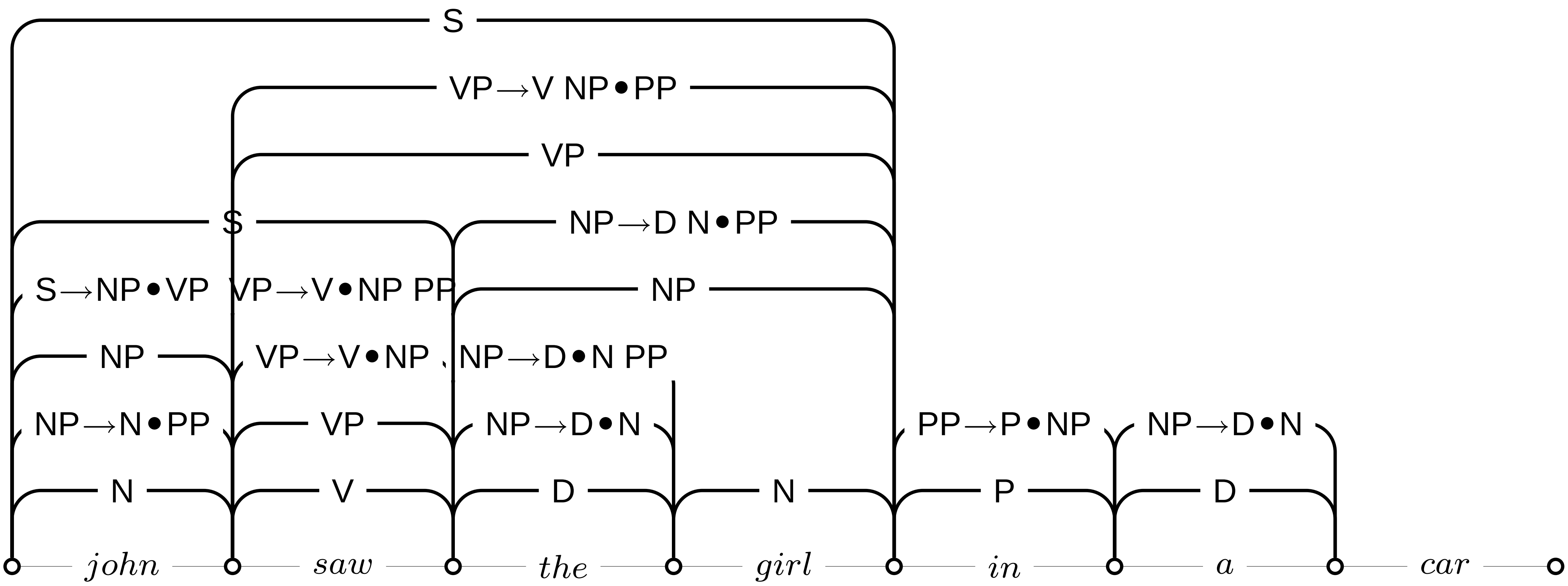
Left-Corner Example



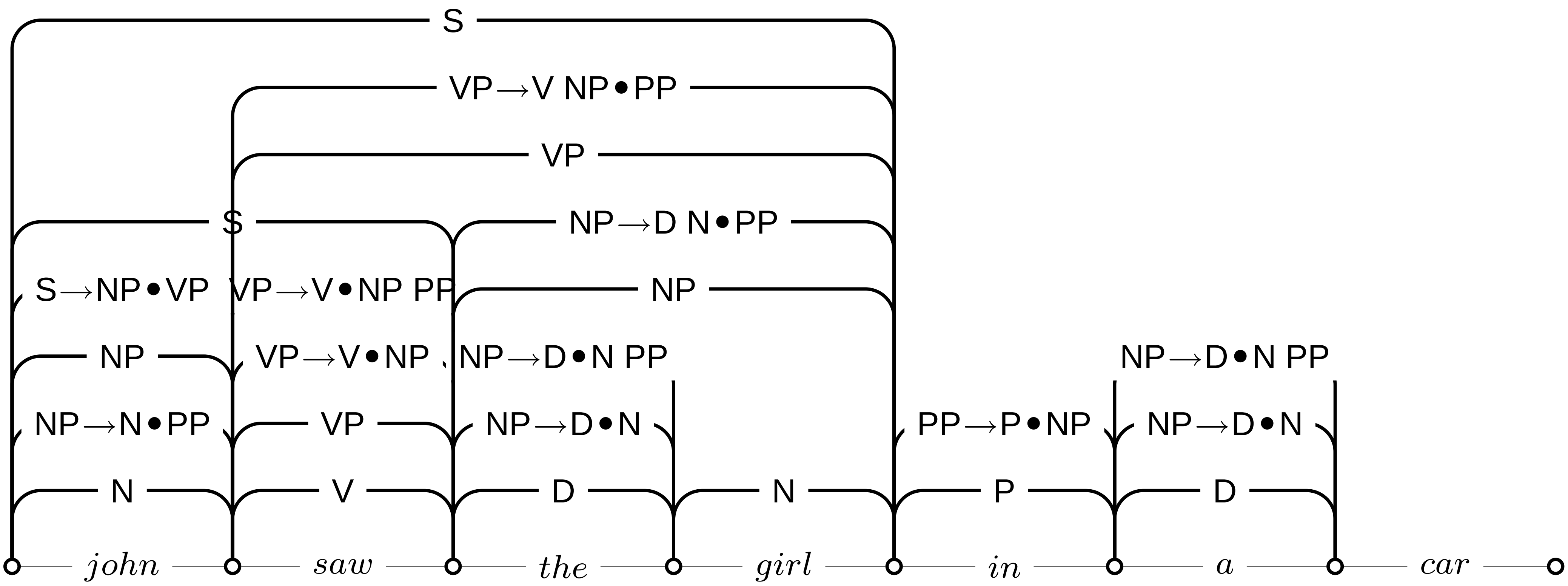
Left-Corner Example



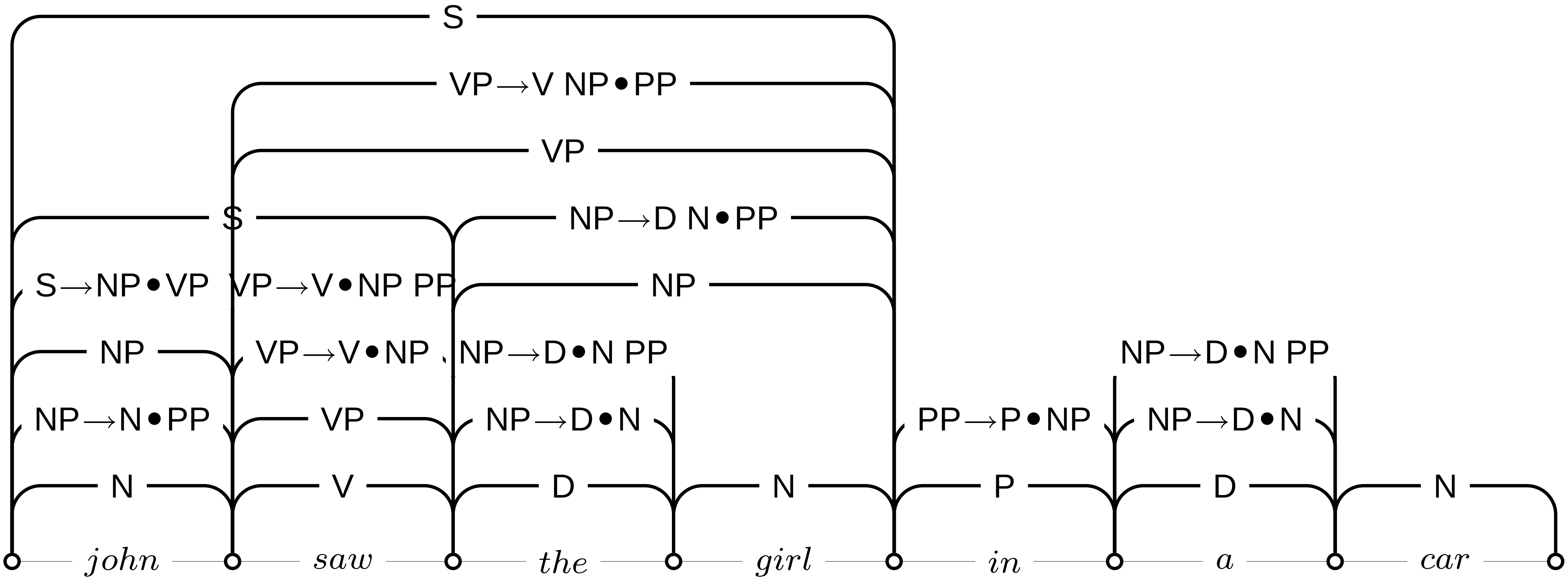
Left-Corner Example



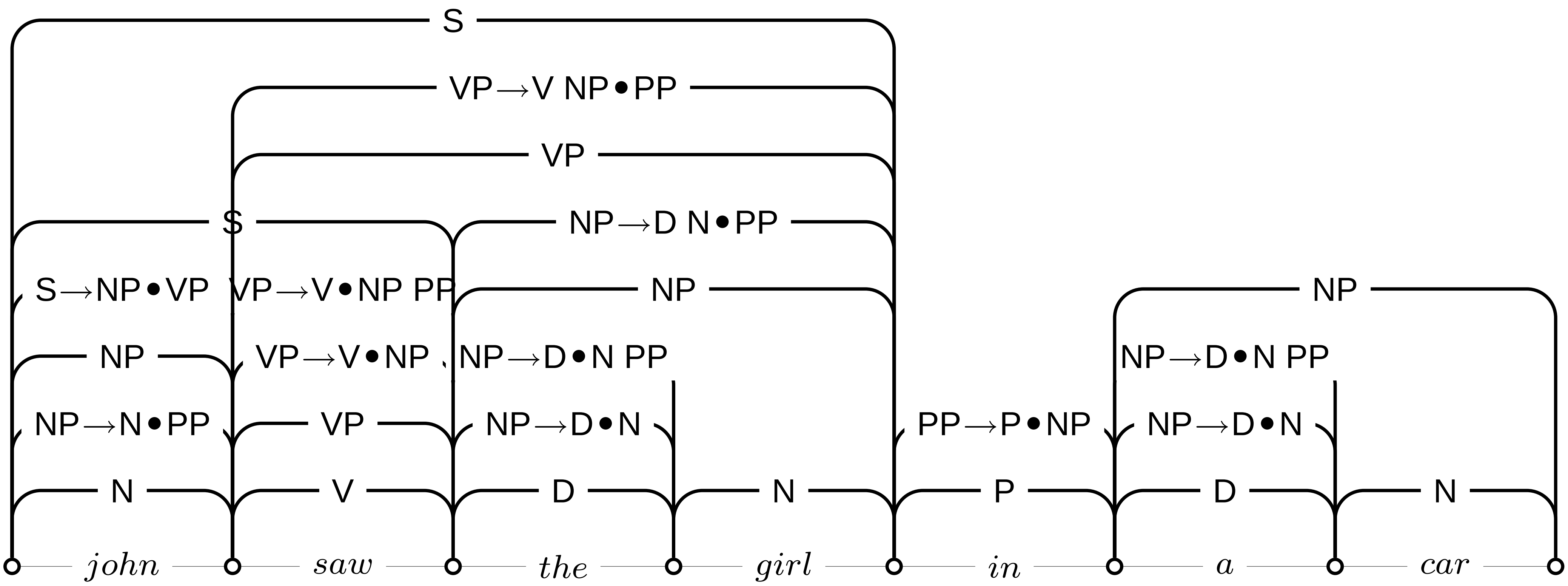
Left-Corner Example



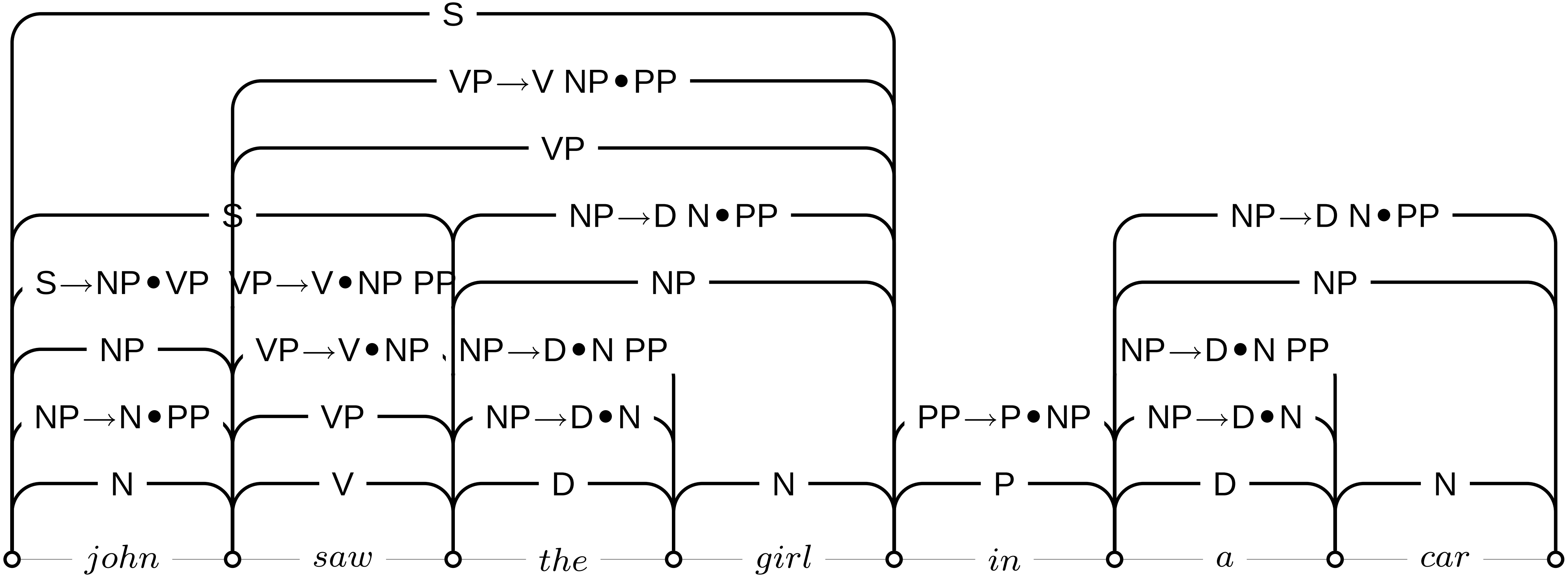
Left-Corner Example



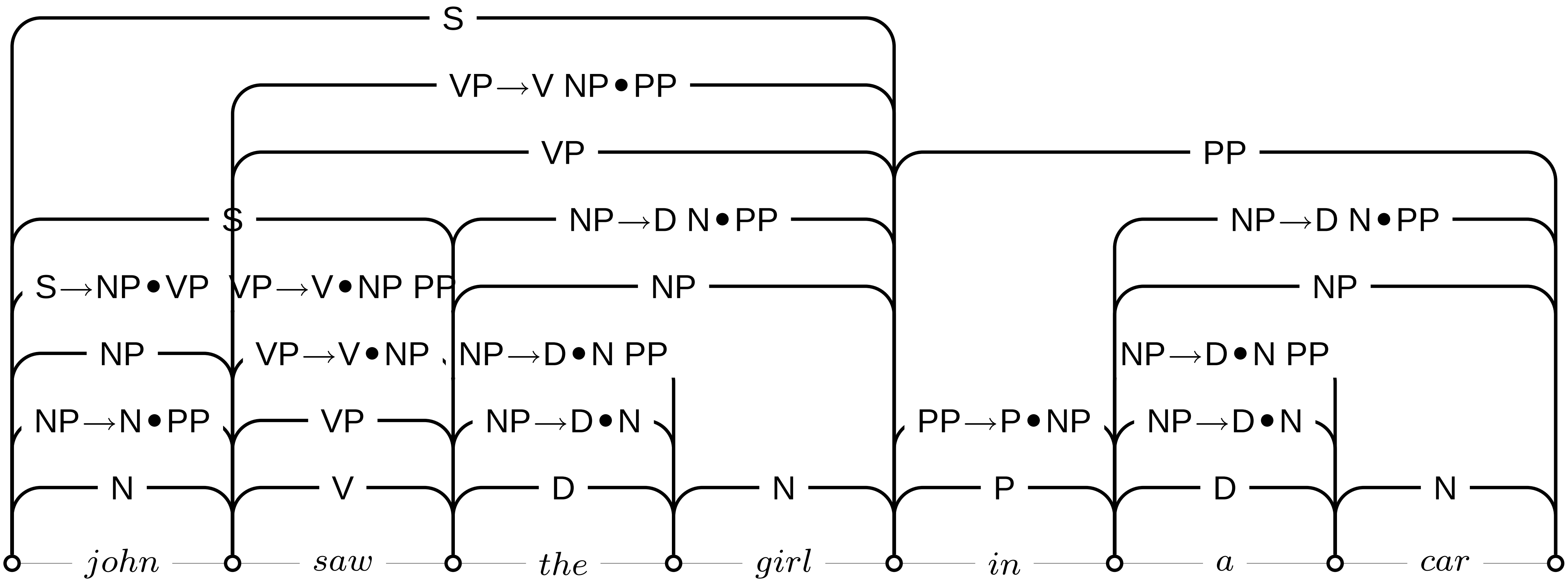
Left-Corner Example



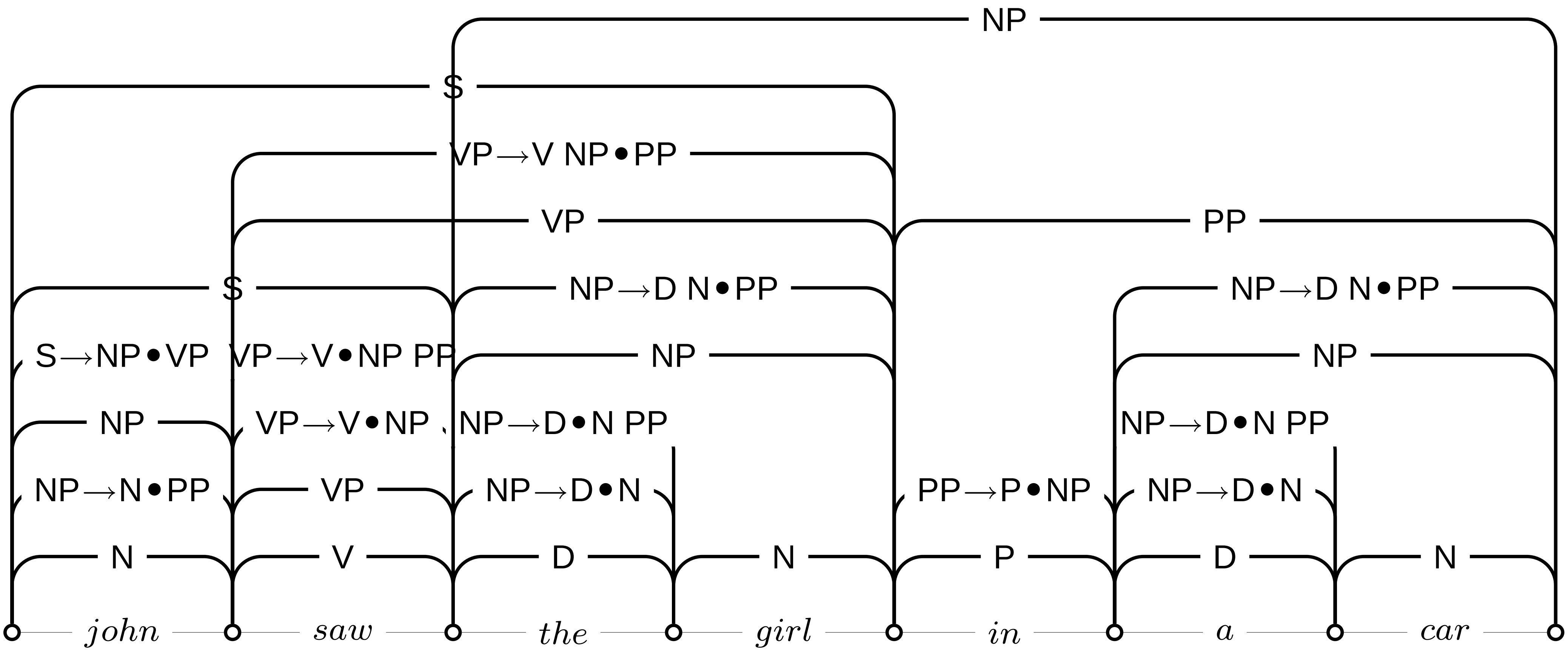
Left-Corner Example



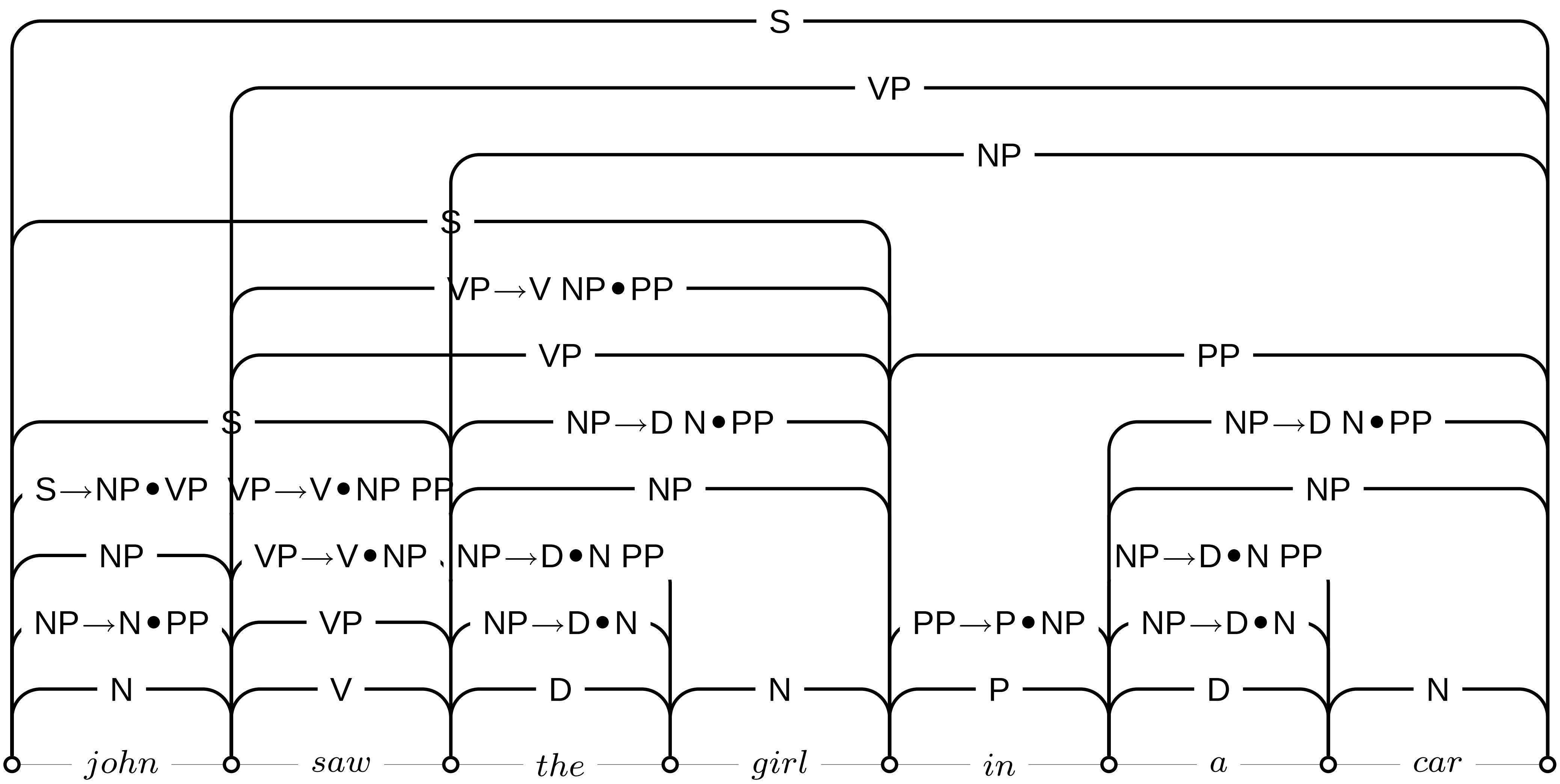
Left-Corner Example



Left-Corner Example



Left-Corner Example



Context-Free Parsing: Summary

- Context-free grammars provide you with a finite set of infinitely embeddable brackets
- Two main approaches to CF recognition: top down (goal-driven) and bottom-up (data driven)
- Storing sub-derivations for re-use (*dynamic programming*) in a chart lead to a polynomial algorithm with worst case n^3
- The chart offers a compact (polynomial size) storage for a possibly exponential number of results
- Earley and Left Corner Parsing improve the average runtime over the naïve CYK algorithm, and have a better worst case complexity for some classes of context-free grammars