



Top Down Parsing

Jean Mark Gawron

Linguistics

San Diego State University

gawron@mail.sdsu.edu

<http://www.rohan.sdsu.edu/~gawron>



Recognizer

Top down recognizer: basic procedures

recognize(derivation, stack, words): *derivation* is a sequence of grammar elements and *words* a sequence of words. *stack* is a sequence of **parser states**, each a pair of derivation and a word sequence. Try to derive the string from the grammar elements. Return True or False. Calls:

1. *expand_rule*: given nonterminal, remaining derivation, stack, and string, try to derive the first part of the string using any grammar rule for the nonterminal and the remaining string with the remaining derivation. Return True or False.
2. *Match*: given terminal, remaining derivation, stack, and string, return False if given terminal does not match first word in string. If there's a match, try to derive the remaining string with the remaining derivation.

1. grammar element: a terminal or nonterminal in the grammar
2. grammar: a function from nonterminals to productions. We write

grammar[nonterminal]

for the set of productions expanding *nonterminal*. Each expansion is a sequence of grammar elements

3. start: start symbol of grammar
4. *words*: a sequence of words
5. pop(derivation): returns first thing on derivation (= last thing in). Reduces size of derivation by 1.
6. Sequence concatenation. We write:

$L_1 + L_2$

for the result of concatenating sequence L_1 with sequence L_2 .

7. $X := Value$: Sets the variable X to the value $Value$.

```
1 proc
2   recognize([start], [], words)
3 where
4 funct recognize(derivation, stack, words)
5   if succeed_state(derivation, words)
6     then return True
7     else if fail_state(derivation, words)
8       then return backtrack(stack)
9       else next := pop(derivation)
10        if nonterminal(next)
11          then return expand_rule(next, derivation, stack, words)
12          else return match(next, derivation, stack, words)
13        fi
14      fi
15    fi
```

Core Recognition Operations

```
1 func expand_rule(nonterminal, derivation, stack, words)
2   rules := grammar[nonterminal]
3   rhs := pop(rules)
4   new_states := make_states(rules, words)
5   new_derivation := [rhs] + derivation
6   return recognize(new_derivation, new_states + stack, words) end
7
8 func match(terminal, derivation, stack, words)
9   nextWord := pop(words) #May be empty string
10  if terminal == nextWord
11    then return recognize(derivation, stack, words)
12    else return backtrack(stack)
13  fi
14 end
```

Backtracking

```
1 func backtrack(stack)
2   if len(stack) > 0
3     then (next_derivation, new_words) := pop (stack)
4         return recognize(next_derivation, stack, new_words)
5     else return False
6   fi
7 end
```

```
1 funct succeed-state(derivation, words)
2   return empty(derivation) and empty(words)
3 end
4
5 funct fail-state(derivation, words)
6   return (empty(derivation) and not empty(words))
7   or
8   (not empty(derivation) and empty(words))
9 end
```

Recursion in expand_rule

```
1 funct expand_rule(nonterminal, derivation, stack, words)
2   rules := grammar[nonterminal]
3   rhs := pop (rules)
4   new_states := make_states(rules, words)
5   new_derivation := [rhs] + derivation
6   return recognize(new_derivation, new_states + stack, words)
```

Line 6 introduces recursion!

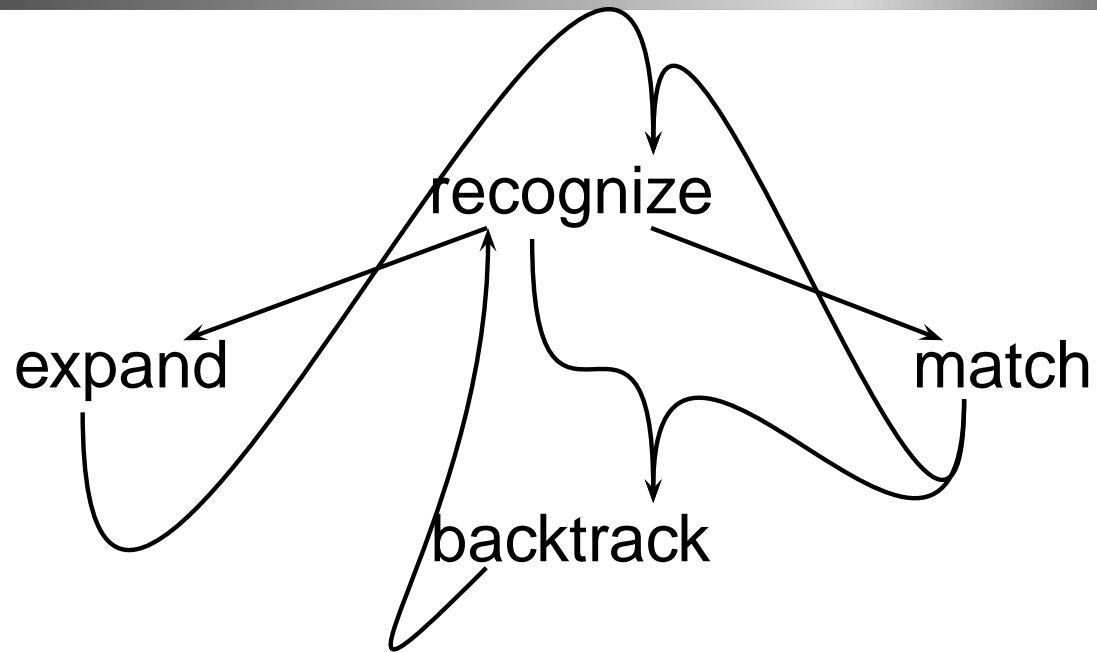
Recursion in match and backtrack

```
1 funct match(terminal, derivation, stack, words)
2   nextWord := pop (words)
3   if terminal == nextWord
4     then return recognize(derivation, stack, words)
5     else return backtrack(stack)
6   fi
7 end
8 funct backtrack(stack)
9   if len(stack) > 0
10    then (next_derivation, new_words) := pop (stack)
11    return recognize(new_derivation, stack, new_words)
12    else return False
13  fi
```

Lines 4 and 11 introduce recursion!

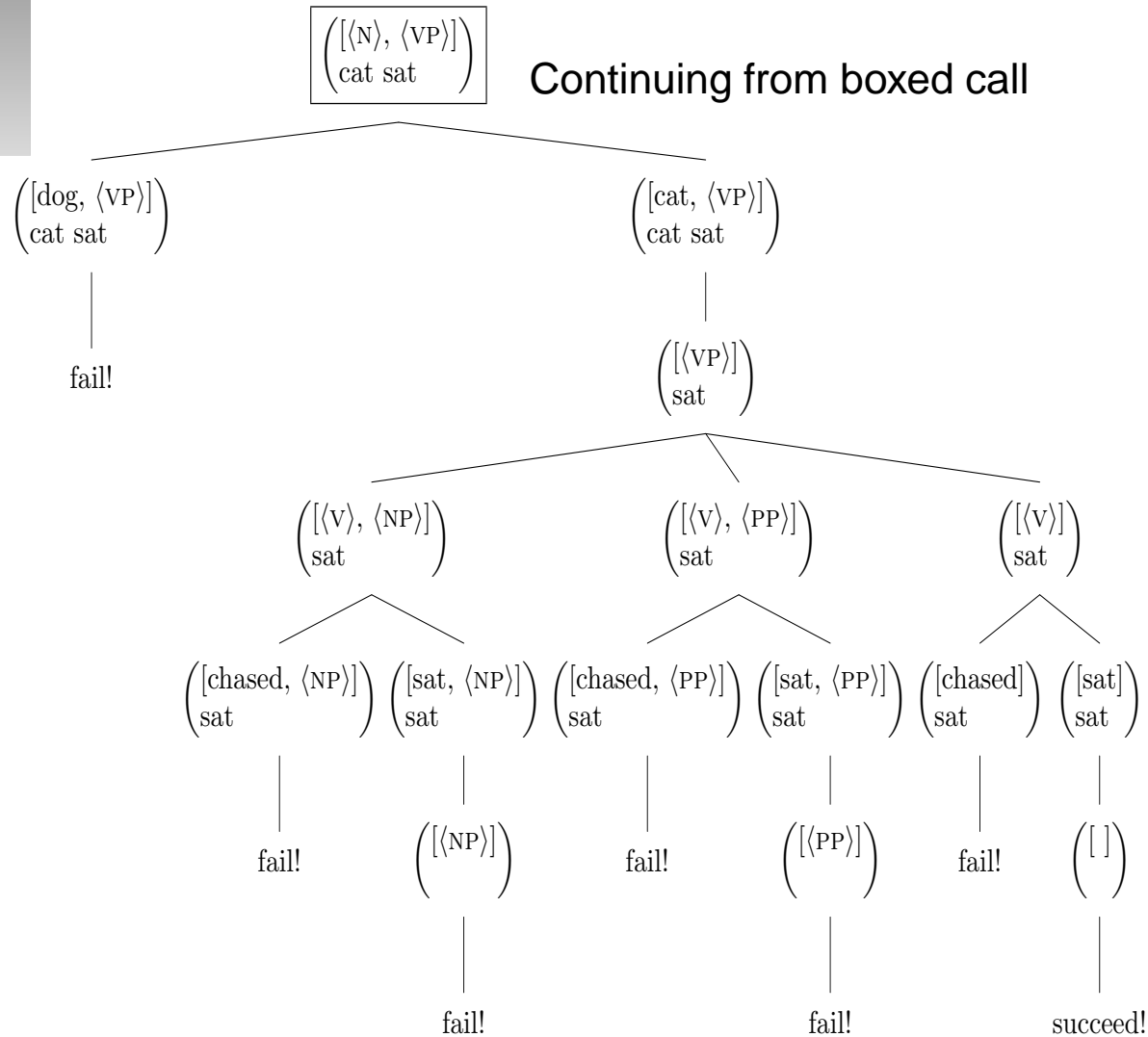
Line 5 introduces **backtracking**.

Recursive Calling Structure

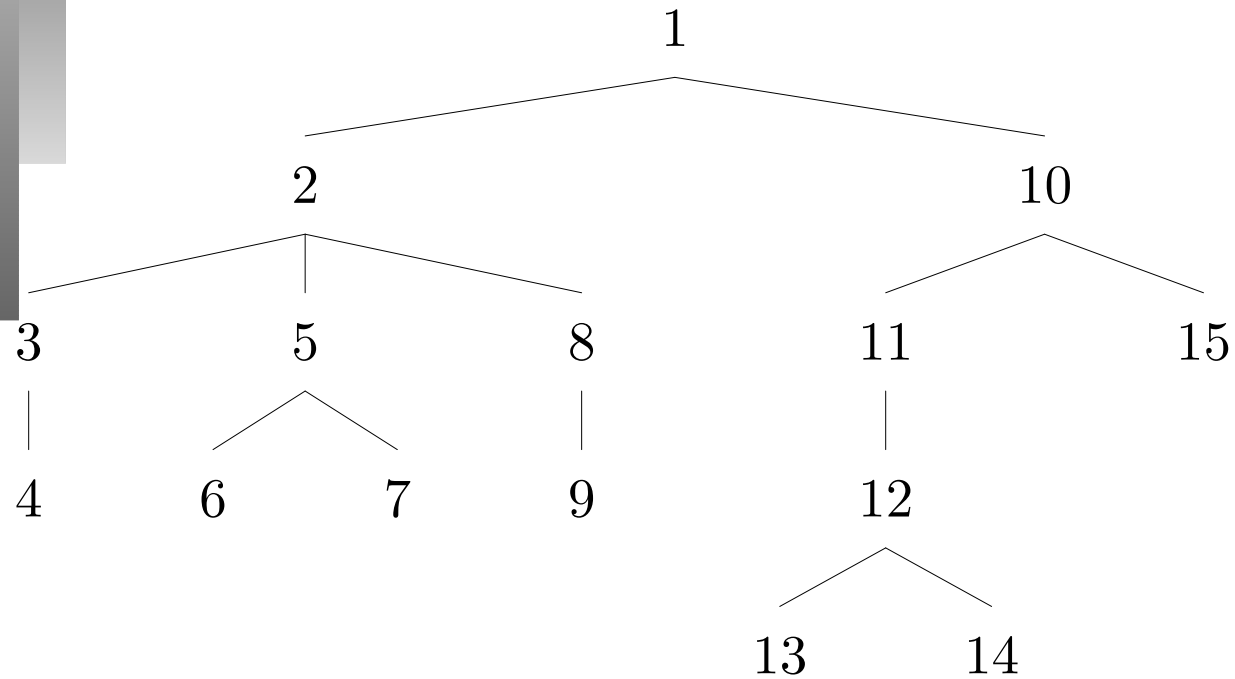


This kind of recursion is called **indirect recursion**.

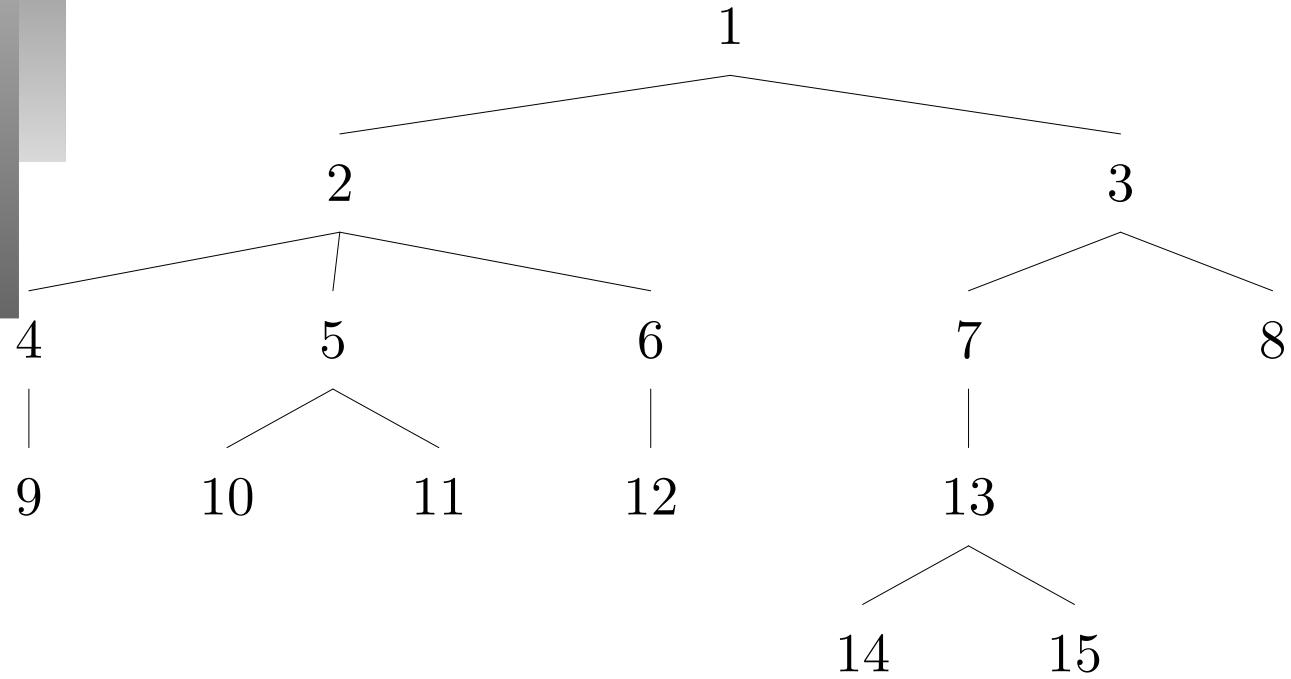
Continuing search space in recognition



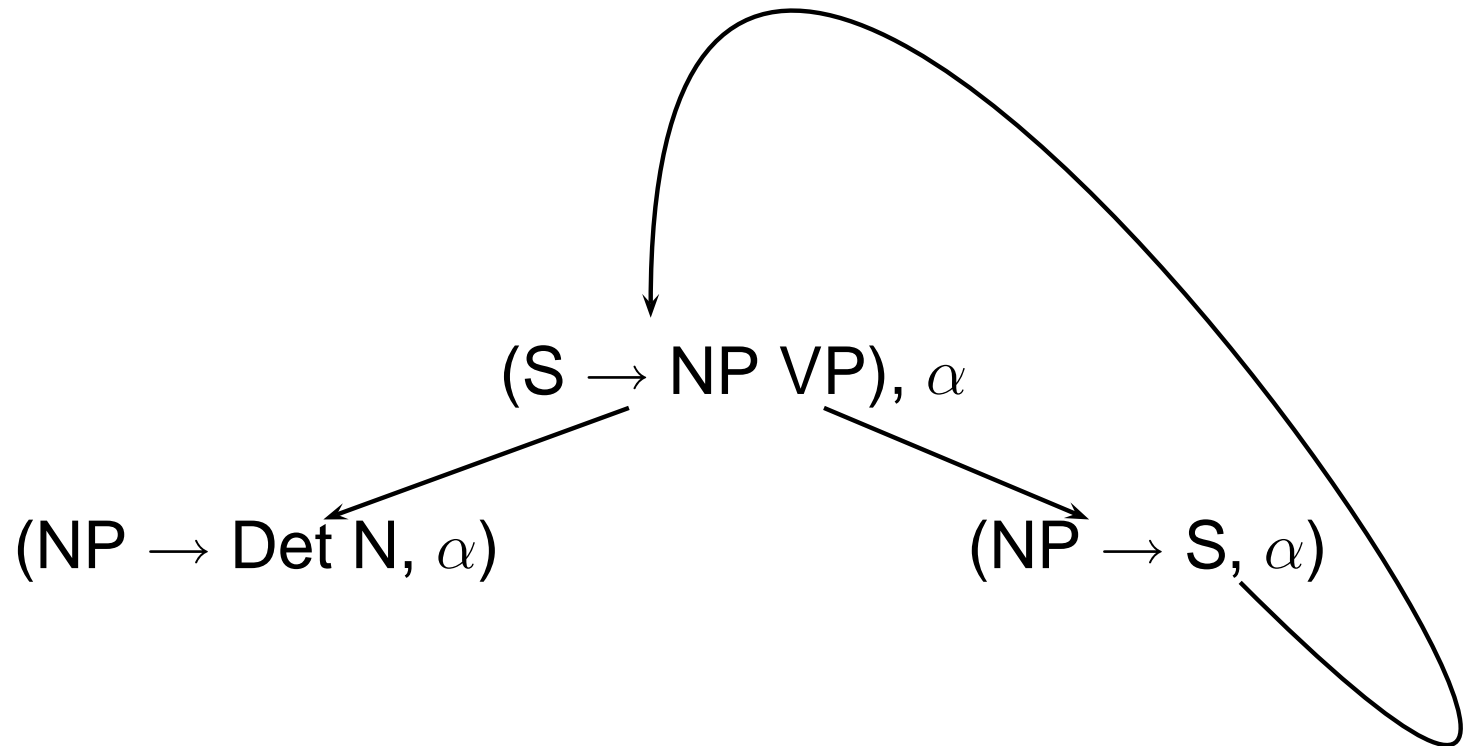
Depth-first search



Breadth-first search



Pitfall of depth-first search



α is the string. What if the search space has loops?

Left Recursion: A problem for top down recognizers/parsers

1. Direct left recursion

$$\begin{aligned} \text{NP} &\rightarrow \text{NP PP} \\ \text{NP} &\rightarrow \text{NP 's } \bar{\text{N}} \end{aligned}$$

2. Indirect left recursion

$$\begin{aligned} \text{S} &\rightarrow \text{NP VP} \\ \text{NP} &\rightarrow \text{S} \end{aligned}$$

Recursion Elsewhere: not a problem

1. Consider

- (a) $S \rightarrow NP VP$
- (b) $NP \rightarrow S'$
- (c) $S' \rightarrow \text{that } S$

2. The recognizer may again visit a state expanding an S in considering rule (c), but it's not the **same state** used for rule (a).

- (a) (S, that John left sucks)
- (b) (NP VP, that John left sucks)
- (c) (S' VP, that John left sucks)
- (d) (that S VP, that John left sucks)
- (e) (S VP, John left sucks)

To go through these steps again, we would have to consume another *that*.

Solution to left-recursion

1. There is a **grammar transformation** which produces a new grammar guaranteed to accept the same set of strings which **eliminates left-recursion** from the grammar.
2. Downside: Greatly expands size of grammar, which slows down parsing.

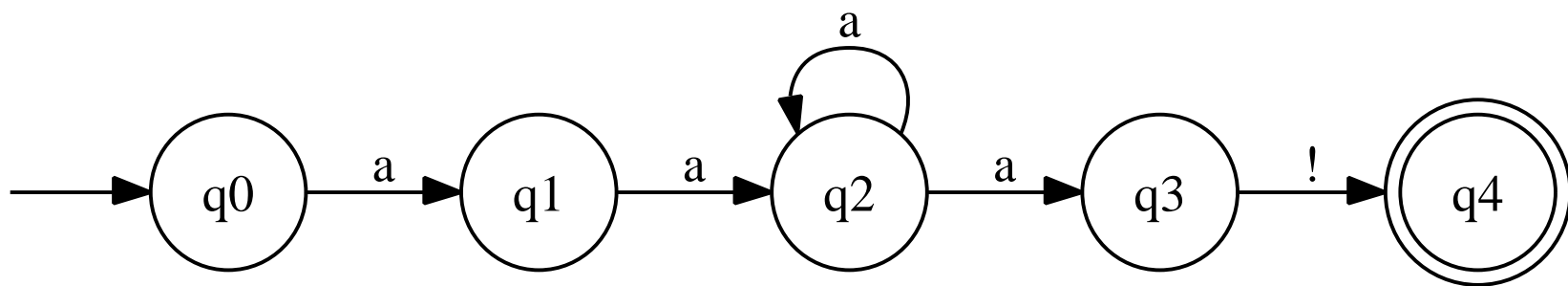
Conclusion

1. This topdown recognizer searches the grammar **depth-first**, expanding daughters left to right.
2. This implementation of a depth-first top-down recognizer is **recursive** (rather than **iterative** [uses loops]).
3. Left recursion in the grammar is a problem for top-down parsers.

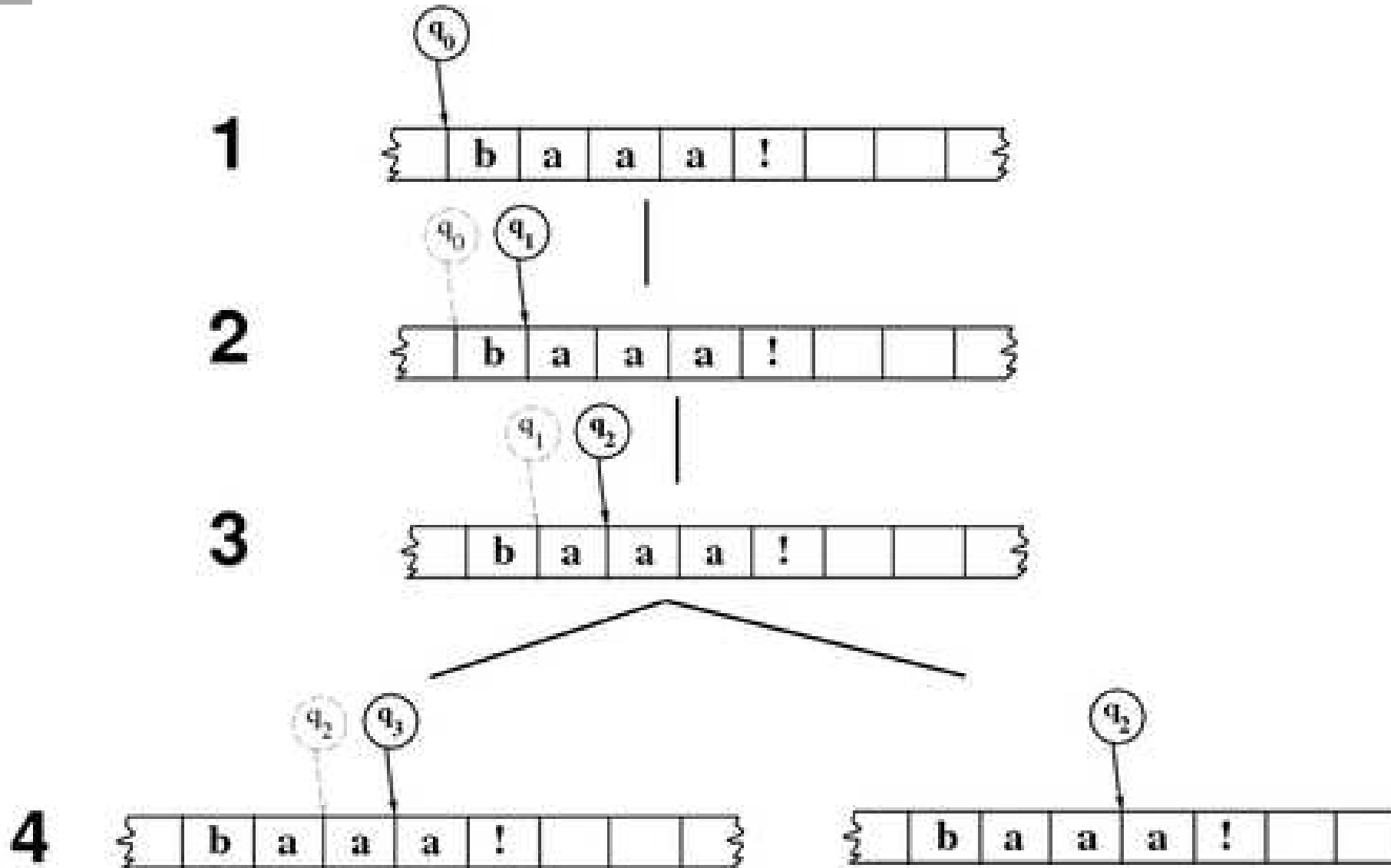


Parser

Nondeterministic Recognizers



Nondeterministic Recognition



Nondeterminism Strategy

- **Backtracking:** Backtrack (resume computation at a prior choice point)
- **Computational state:** Need explicit representation of prior computational state

$\langle q_2, a! \rangle$

- **Agenda:** Keep a list of computational states not yet explored, updated at each choice point.

Nondeterminism in Transducers

Given input string s_i

- Recognition: Determine whether there is a string related to s_i by regular relation R .
- Transduction: Find the set of strings related to input string s_i by regular relation R .
- states

state = $\langle \text{machine-state}, \text{tape-pos}_1, \text{related-string} \rangle$

init = $\langle \mathbf{s}_0, 0, \epsilon \rangle$

Recognition

```
1  proc
2    recognize([(s0, 0, ε)], tape)
3  where
4  funct recognize(agenda, tape) ≡
5    current_search_state := pop (agenda)
6    while True do
7      if accept_state(current_search_state)
8        then return True
9        else extend(agenda, new_states(current_search_state, tape))
10      if agenda
11        then current_search_state := pop (agenda)
12        else return False
13      fi
14    fi
15  od
16  .
```

All related strings

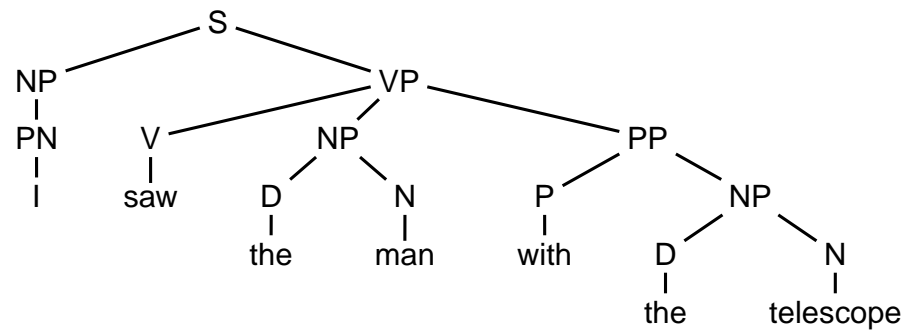
```
1  proc
2    transduce([(s0, 0,  $\epsilon$ )], tape)
3  where
4  funct transduce(agenda, tape)  $\equiv$ 
5    related_strings := [];
6    while agenda
7      do
8        current_search_state := pop (agenda);
9        if accept_state(current_search_state)
10       then push (related_string(current_search_state), related_strings)
11       else extend (agenda, new_states(current_search_state, tape))
12     fi
13  od;
14  return related_strings
15  .
```

Parsing CFGs

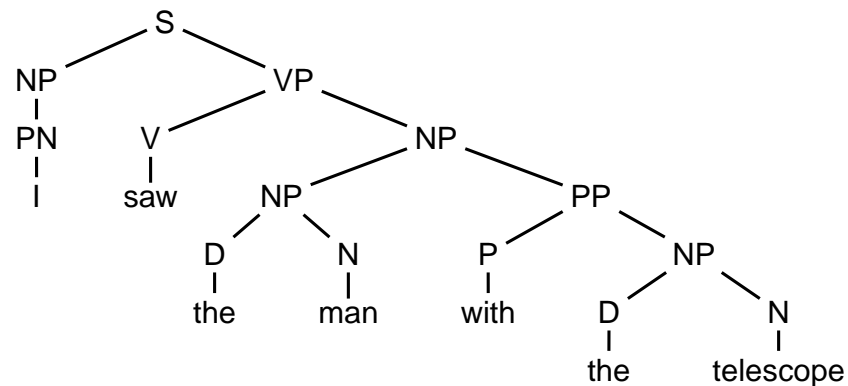
Inherent non-determinism

1. Nonterminals may have more than one expansion $S \rightarrow NP VP$
 $S \rightarrow VP$
2. More than one way of accepting a string

(1) a.



b.



recognize *and non-determinism*

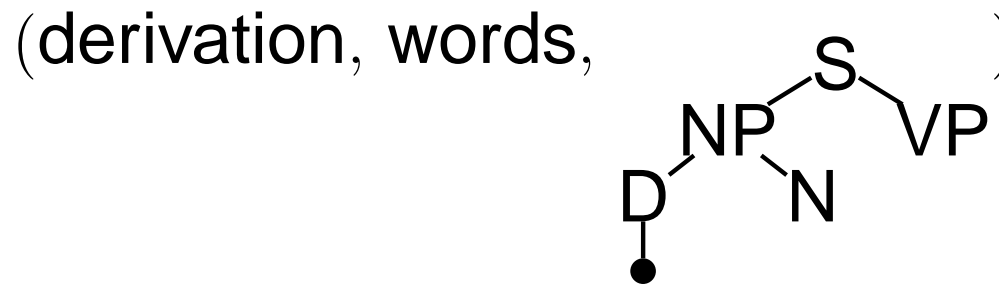
1. A pair of a derivation and a string served as computational state.
2. Backtracking was explicitly managed in *backtrack*:

```
funct backtrack(stack)  
    if len(stack) > 0  
        then (next_derivation, new_words) := pop (stack)  
            return recognize(new_derivation, stack, new_words)  
        else return False  
    fi
```

3. The stack was the agenda: List of search alternatives not yet tried.

Changing to a parser

- State now consists of a triple of a derivation, words, and a **tree**.



- Derivation elements are pairs of grammar elements and tree addresses

$[(D, (1, 1)), (N, (1, 2)), (VP, (2,))]$

- Agenda remains a list of states

```
1 proc
2   parses = []
3   parse([(S, 0)], [], words, S)
4 where
5 funct parse(derivation, stack, words, tree) ≡
6   if succeed_state(derivation, words)
7     then push (tree, parses)
8     return backtrack(stack)
9   else if fail_state(derivation, words)
10    then return backtrack(stack)
11    else next := pop (derivation)
12      if nonterminal(next)
13        then return expand_rule(next, derivation, stack, words, tree)
14        else return match(next, derivation, stack, words, tree)
15      fi
16    fi
17  fi
18  .
```

Redefining backtrack

```
1 funct backtrack(stack)
2   if len(stack) > 0
3     then (next_derivation, new_words, new_tree) := pop(stack)
4         return parse(new_derivation, stack, new_words, new_tree)
5     else return parses
6   fi
```