Syntax Analysis - LL(k)

Prof. Licia Sbattella

aa 2007-08

Translated and adapted by L. Breveglieri

SYNTAX ANALYSIS

Given a grammar G, the syntax analyzer or parser reads the source string and, if the string belongs to the language L(G), produces a derivation or a syntax tree for the string; otherwise the parser stops and outputs the configuration at the point where the error has occurred (diagnosis). Possibly the parser tries to go on skipping the substrings containing the errors (that is, tries to repair the error).

Not considering the treatment of errors ... a parser is a language recognizer, enhanced by the ability to produce the derivation of the string; to represent the derivation, the automaton executes moves that are in correspondence with grammar rules, and when a rule is applied the corresponding lablel must be saved into an appropriate data structure.

If the source string is ambiguous, the parser must produce two (or more) derivations or syntax trees (and clearly the parser itself must be a indeterministic automaton).

TOP-DOWN AND BOTTOM-UP ANALYSIS

The same syntax tree corresponds to two or more derivation orders: leftmost, righmost, mixed, ...

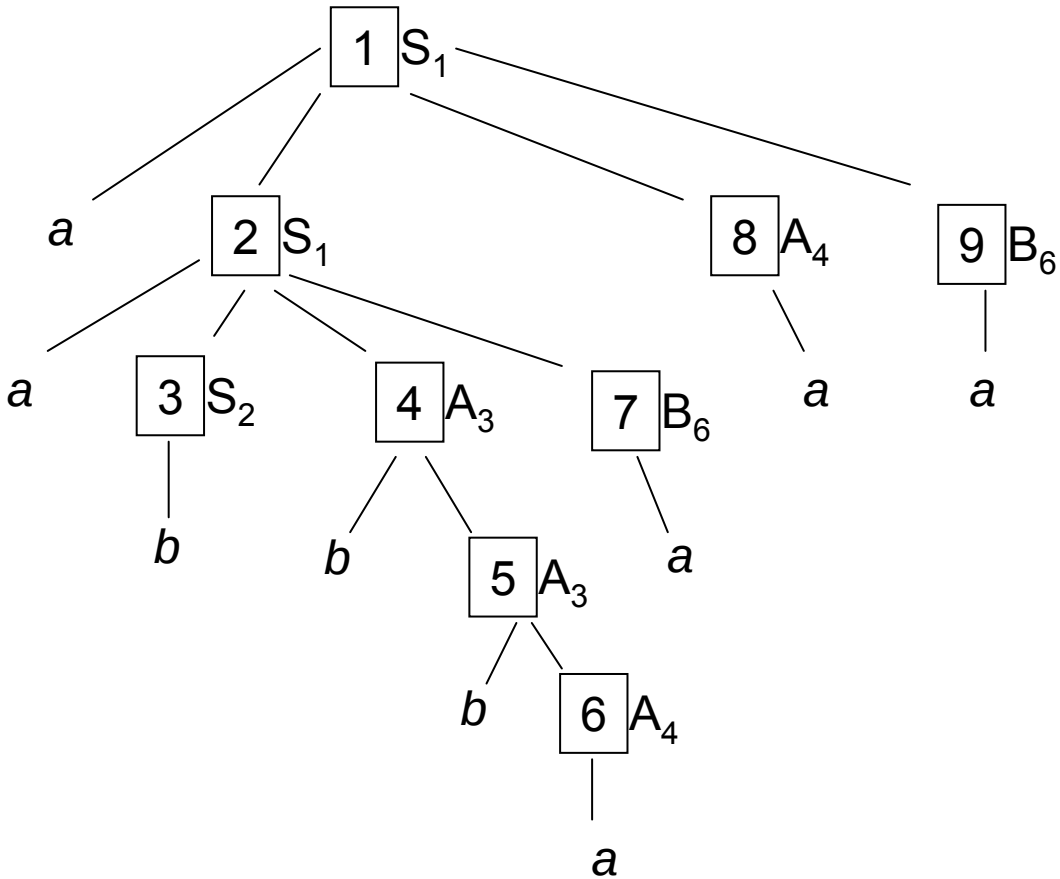
There are two important classes of parsers, depending on the derivation order (leftmost rather than rightmost) and on whether the derivation is constructed starting from the beginning rather than from the end.

TOP-DOWN ANALYSIS: the parser produces the LEFTMOST derivation of the string, and the syntax tree is built starting from the root to the leaves (this kind of analysis is also called RECURSIVE DESCENT syntax analysis) - also indicated with the acronym LL.

BOTTOM-UP ANALYSIS: the parser produces the RIGHTMOST derivation of the string, and the syntax tree is built starting from the leaves to the root (by means of a series of so-called REDUCTION MOVES) - also indicated with the acronym LR.

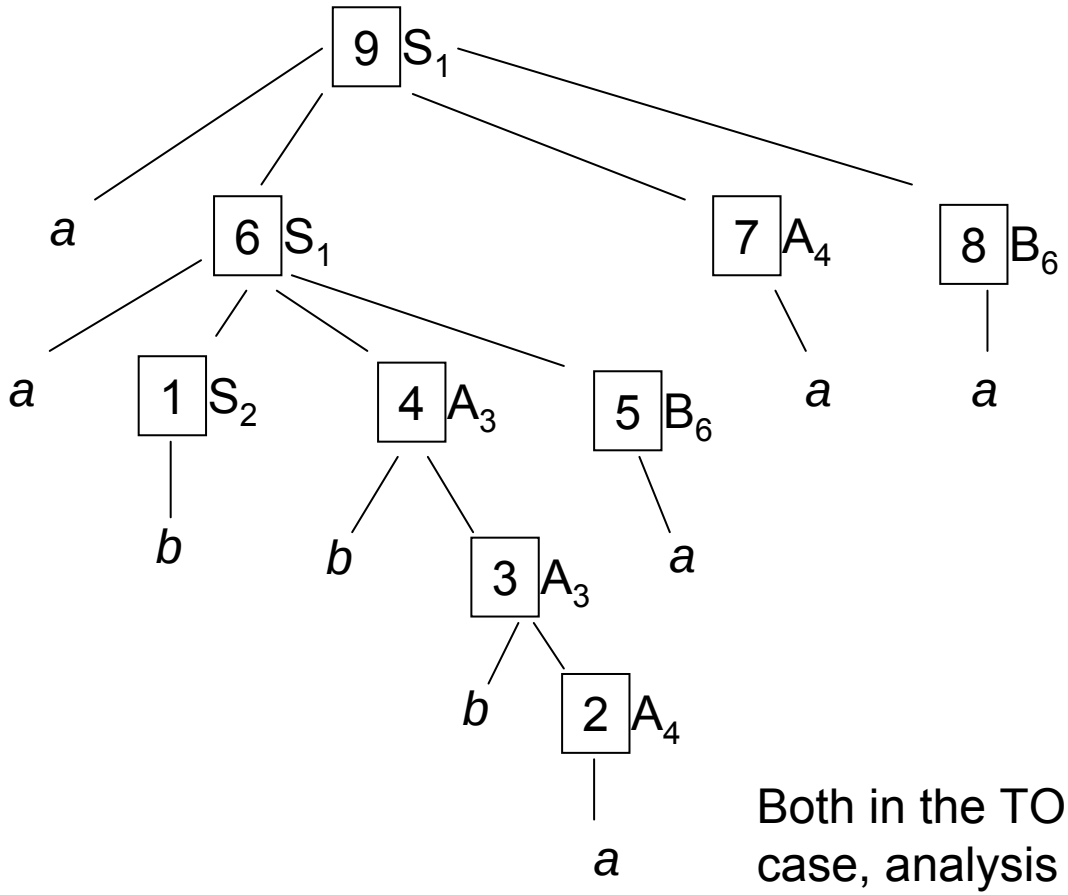
EXAMPLE – browse order of the tree
of the phrase: $a^2 b^3 a^4$

1. $S \to aSAB$	2. $S \to b$
3. $A \to bA$	4. $A \to a$
5. $B \to cB$	6. $B \to a$



TOP-DOWN PARSER:
it expands the left parts of the production rules by means of the corresponding right parts; the construction terminates when all the non-terminal symbols are transformed in terminal symbols matching the text.

1. $S \to aSAB$	2. $S \to b$
3. $A \to bA$	4. $A \to a$
5. $B \to cB$	6. $B \to a$



BOTTOM-UP PARSER:
the right parts of the grammar rules are reduced to non-terminal symbols; this process starts from the terminal string and goes on applying to the sentential forms that progressively raplace the terminal string to analyze; the process terminates with success if the entire terminal string is reduced to the axiom.

Both in the TOP-DOWN and in the BOTTOM-UP case, analysis stops when an error is found.

THE GRAMMAR AS A NETWORK OF FINITE STATE AUTOMATA

For the presentation of the parsing algorithms, it is convenient to represent the grammar by means of a RECURSIVE NETWORK OF FINITE STATA AUTOMATA.

Given a grammar G, every non-terminal symbol is the left part of one or more alternative rules. Suppose G is EBNF (extended BNF), possibly containing some regular operators (e.g. star, etc): then every non-terminal is expanded by a single rule A → α, where α is a regexp over the terminal and non-terminal alphabets. Hence α defines a regular language, the finite state recognizer of which can be easily built.

$L(G) \quad A \to \alpha$

α may contain B

e $B \to \beta$

B and A may coincide

In this way one gets a set of machines M = { M_0, …, M_m }, which actually is only a notational variant of grammar G. The language, of alphabet Σ – L(M), defined and recognised by the machine network M, is the same as that generated by the grammar G.

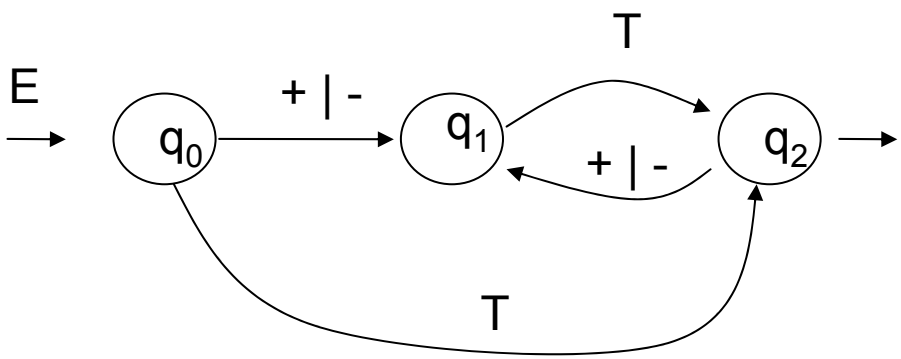
$$q \in Q \quad L(M,q) = \left\{ y \in \Sigma^* \mid \beta \in R(M_i, q) \wedge \beta \overset{*}{\Rightarrow} y \right\}$$

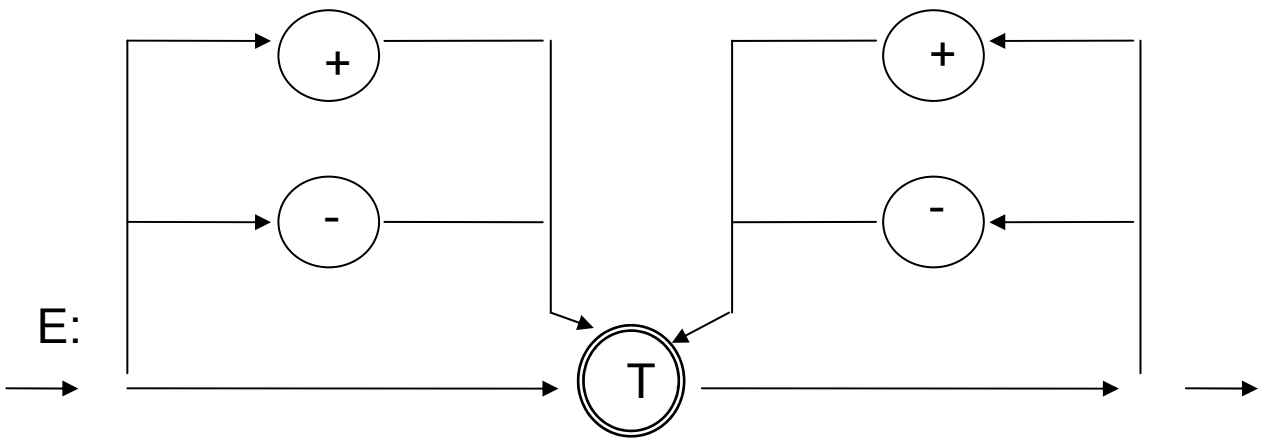
$$L(M, q_{0,0}) \equiv L(M) \equiv L(G) \qquad L(M,q) \equiv L_{M_i}(G), \text{ if } q \in Q_i$$

EXAMPLE – syntax diagrams of the grammar of arithmetic expressions

$$E \to [+|-]T((+|-)T)^*$$
$$T \to F((\times|/)F)^*$$
$$F \to (a|'('E')')$$

automaton expanding E

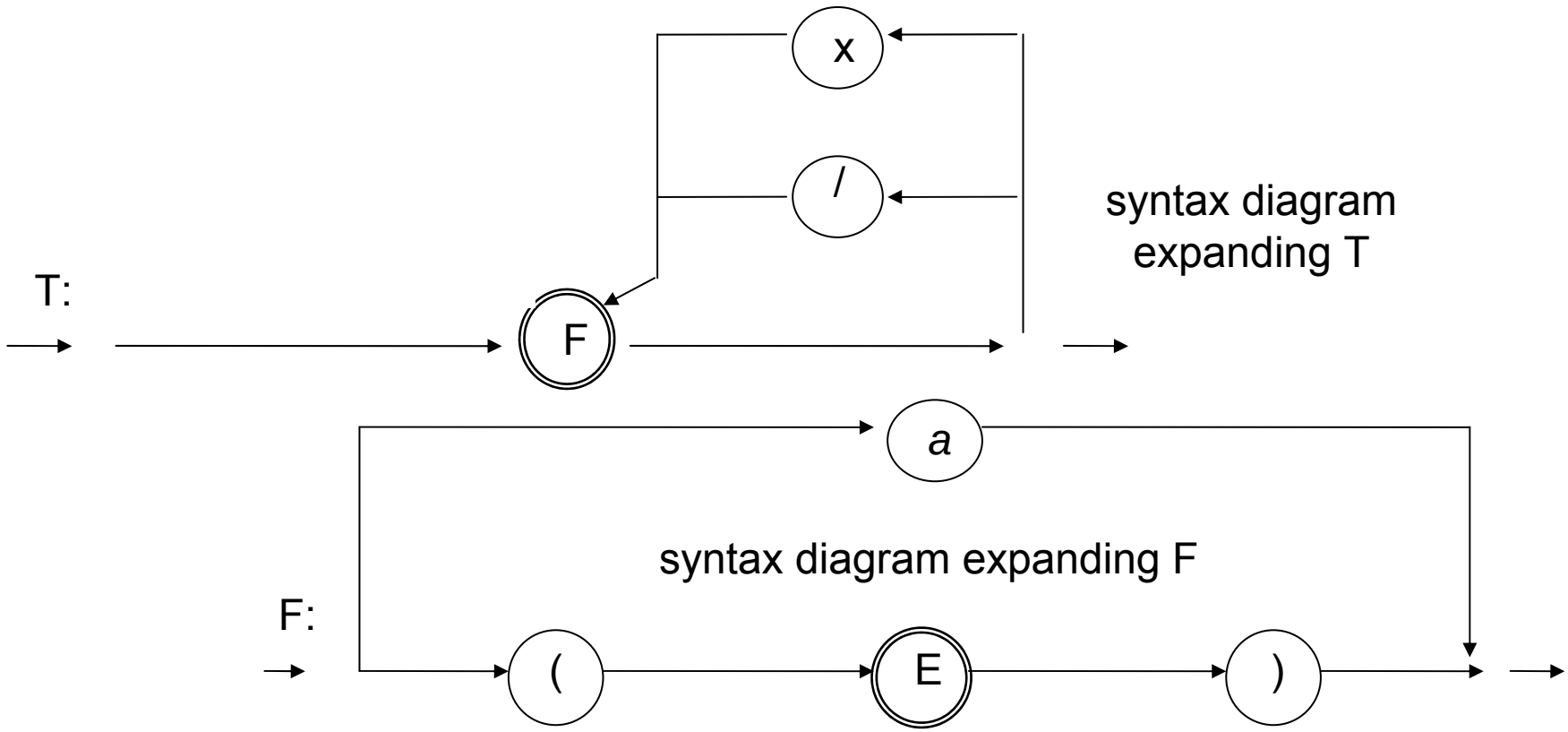


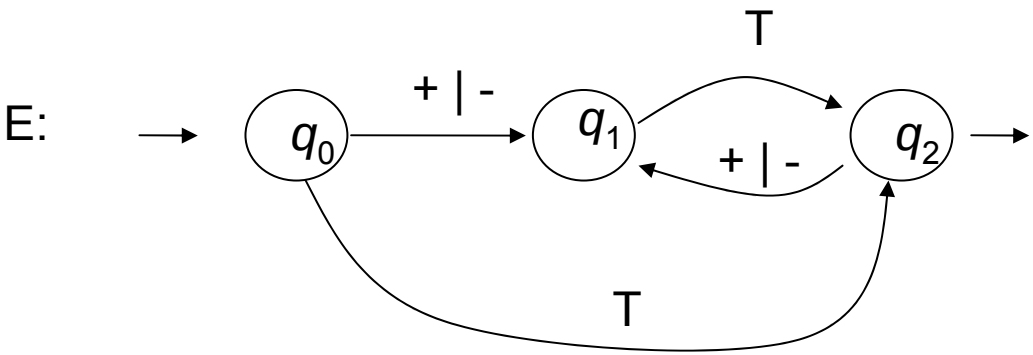


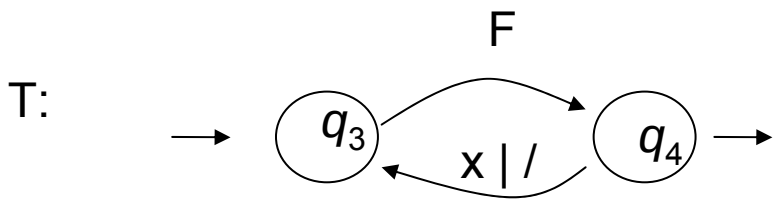
syntax diagram expanding E

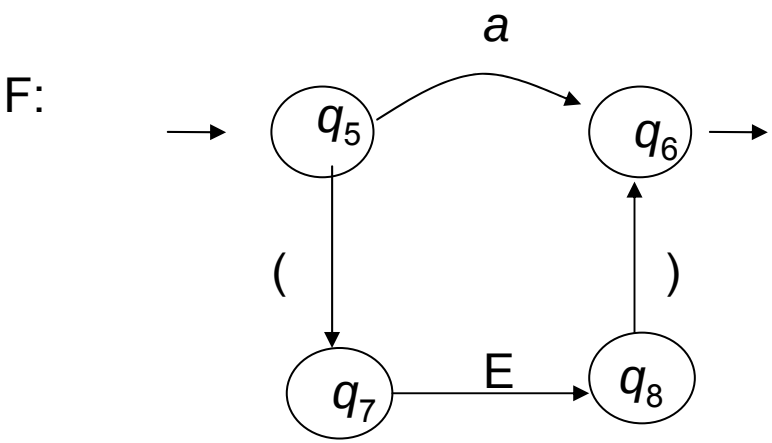
$$E \to [+|-]T((+|-)T)^*$$
$$T \to F((\times|/)F)^*$$
$$F \to (a|'('E')')$$

Given a free extended grammar, construct the finite state recognizers of the regexps that represent the right parts of the production rules. In order to have more efficient parsers, it is advisable to put such finite state machines in deterministic form (which is always possible).









recursive network of deterministic automata representing grammar G

$$E \to [+|-]T((+|-)T)^*$$
$$T \to F((\times|/)F)^*$$
$$F \to (a|'('E')')$$

RECOGNITION

It is immediate to interpret a recursive network of machines as the flow graph of the recursive recognition algorithm for the phrases of the language.
This algorithm realizes a pushdown automaton, in general non-deterministic, with a single internal state and a stack, with acceptance condition by empty stack.

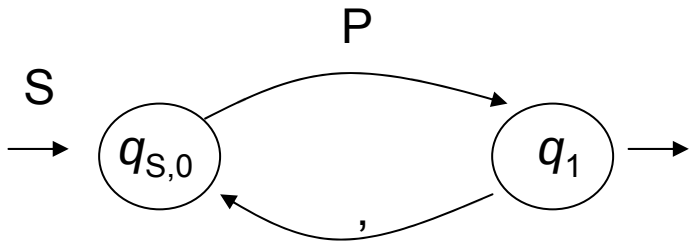
At the beginning, the active machine is that expanding the axiom.

1) If the input string is valid, there must exist a computation that will lead the machine to the final state (possibly invoking recursively other machines in the middle).
2) The automaton, at every step, goes on with the current machine if the symbol is terminal, otherwise, i.e. in the case the current symbol is non-terminal and indicates another machine, it suspends the current computation and goes to the initial state of such new machine (this does not exclude the case of a recursive re-invocation of the current machine).
3) Whenever the automaton moves recursively to a new machine, it pushes the state of the current machine on the top of the stack, for later perusal.
4) When the automaton reaches a final state of the current machine, it restores the state of the most recently suspended machine (popping it from the stack), and, starting from that state, it executes the move labeled by the non-terminal symbol expanded by the just terminated machine.

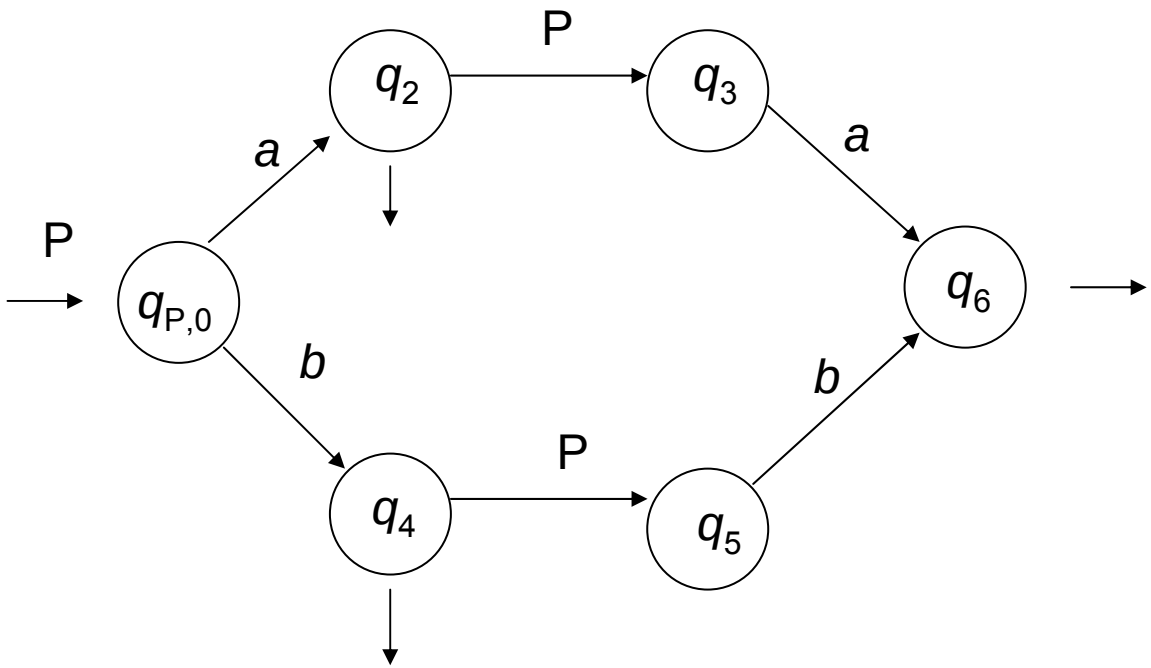
In order not to fall into infinite recursion, the grammar need not contain left recursion.

EXAMPLE – list of palindromes of odd length

$$S \to P(,P)^*$$
$$P \to aPa \mid bPb \mid a \mid b \mid$$



stack	string
$q_{S,0}$	a , b -|
$q_{S,0}$ $q_{P,0}$	a , b -|
$q_{S,0}$ q_2	, b -|
q_1	, b -|
$q_{S,0}$	b -|
$q_{S,0}$ $q_{P,0}$	b -|
$q_{S,0}$ q_4	-|
q_1	



ALGORITHM – recursive descent recognizer, in general non-deterministic

begin

1. the source string is *x* and *cc* is the current input character
2. the stack alphabet is the disjoint union $Q = Q_A \cup Q_B$ ... of the state sets of all the machines of the recursive network, each expanding a specific non-terminal A, B, ...
3. the pushdown automaton itself has only one internal state, which is understood
4. at the beginning the stack contains the symbol $q_{S,0}$ (the initial state of the machine M_S expanding the axiom)
5. transitions: let $s \in q_A$ be the current stack top symbol, which is the current state of the currently active machine M_A:
   a) (shift move) if the move $\delta(s, cc) = s'$ is defined in the currently active machine M_A, read the current input character, pop *s* and push *s*' (to replace *s*)
   b) (call move) if the move $\delta(s, B) = s'$ is defined for a non-terminal B in the currently active machine M_A, suspend the current machine and execute a spontaneous move to push the symbol $q_{B,0}$ that represents the initial state of the new machine M_B, which since now will be the currently active machine
   c) (return move) if *s* is a final state in the currently active machine M_A, pop it, and supposing that state *r* (belonging to the machine M_C) emerges as the new stack top, resume machine M_C and execute the move $\delta(r, B) = s'$ in the current machine M_C, popping state *r* and pushing state *s*' (to replace *r*)
   d) (acceptance move) if *s* is a final state in the currently active machine M_S (the axiom machine) and *cc* = -|, terminate and accept the input string
   e) (reject move) otherwise, terminate and reject the input string

end

DETERMINISTIC RECURSIVE DESCENT ANALYSIS

Many grammars of artificial languages have been designed in order to allow a fast deterministic recognition of the phrases.

Method LL(*k*) (L = left, the input string is analyzed starting from the left end, L = left, the reconstructed derivation is of the leftmost type, *k* = length of the lookahead input window).

THERE TWO POSSIBLE IMPLEMENTATIONS FOR THE LL(*k*) DETERMINISTIC RECOGNIZER:

1) a classical pushdown automaton
2) a recursive descent program, where the role of the stack is assumed by the implicit stack used to store the activation records of the syntactic procedures

CONDITIONS FOR THE CONSTRUCTION OF THE LL(1) DET. RECOGNIZER

Consider a grammar G, represented as a recursive network M of finite state machines, (rules $A_i \rightarrow \alpha_i$), and let L(G) over Σ be defined by M. Assume all phrases end by -|.

Start symbols of a state *q* (Ini) = the initial symbols of the terminal strings generated starting from *q*

$$Ini(q) = Ini(L(M,q))$$

LOOKAHEAD SET (Gui) = set (non-empty) of terminal symbols defined for every arc outgoing from a node, as follows:

The union is extended to every state *s* with a ingoing arc labeled by A_h.

1. $q \xrightarrow{a} r \;\; a \in \Sigma \quad Gui(q \xrightarrow{a} r) = \{a\}$

2. $q \xrightarrow{A_k} r \quad Gui(q \xrightarrow{A_k} r) =$

$$\begin{cases} Ini(L(M,q_{0,k})), & \text{if } L(M,q_{0,k}) \text{ is not nullable} \\ Ini(L(M,q_{0,k})) \cup Ini(L(M,r)) & \text{otherwise} \end{cases}$$

3. for the arc $q \rightarrow$ of a final state of A_h

$$Gui(q \rightarrow) = \begin{cases} \bigcup_s Ini(L(s)) & \text{if } A_h \text{ is not axiom} \\ \bigcup_s Ini(L(s)) \cup \{-|\} & \text{otherwise} \end{cases}$$

SUFFICIENT CONDITIONS FOR THE PUSHDOWN AUTOMATON TO BE DET.:

LL(1) CONDITION – a machine A_j (deterministic by hypothesis) in the state $q \in Q_j$ satisfies condition LL(1) if, for every pair of arcs outgoing from that state q, the lookahead sets (Gui) are disjoint.

$$\begin{cases} \{a\} \cap Gui(q \xrightarrow{A_k} r'') = \varnothing & \text{if } q \xrightarrow{a} r' \quad q \xrightarrow{A_k} r''. \\ Gui(q \xrightarrow{A_h} r') \cap Gui(q \xrightarrow{A_k} r'') = \varnothing & \text{if } q \xrightarrow{A_h} r' \quad q \xrightarrow{A_k} r''. \\ Gui(q \rightarrow) \cap Gui(q \xrightarrow{X} r) = \varnothing & \text{if } q \xrightarrow{X} r. \end{cases}$$

The LL(1) condition examines every pair of moves outgoing from each state:

- one move reads a terminal *a* and the other move reads a non-terminal A_k
- both moves read non-terminal symbols
- state *q* is final (and every arc outgoing from *q* must have a lookahead set disjoint from that of $q \rightarrow$)

DEFINITION – a grammar is LL(1) if every state of the machine network that represent the grammar is LL(1)

EXAMPLES – grammars LL(1) and not LL(1)

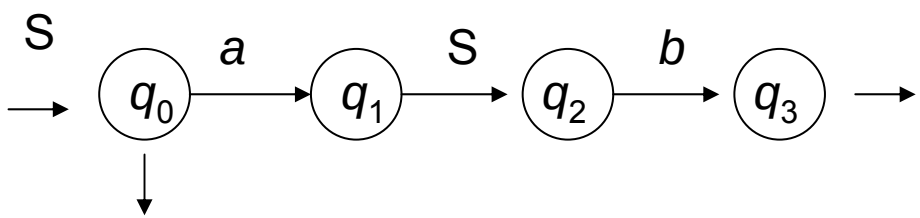
$G_1 = S \to aSb \mid \varepsilon$

$\{a^n b^n \mid n \geq 0\}$

$Gui(q_0 \xrightarrow{a} q_1) = \{a\}$

$Gui(q_0 \to) = \{b, \text{-}|\}$

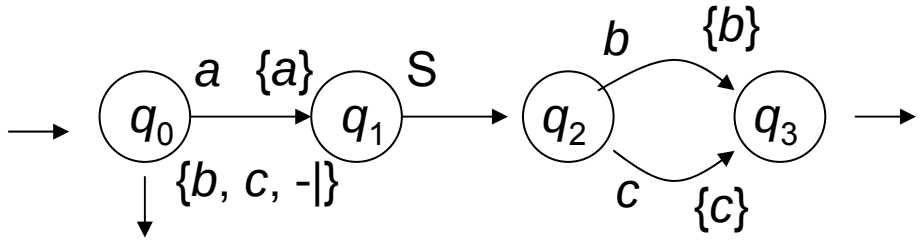
disjoint sets: grammar is $\mathrm{LL}(1)$



$G_2 : S \to aSb \mid aSc \mid \varepsilon$

$\{a^n (b \mid c)^n \mid n \geq 0\}$

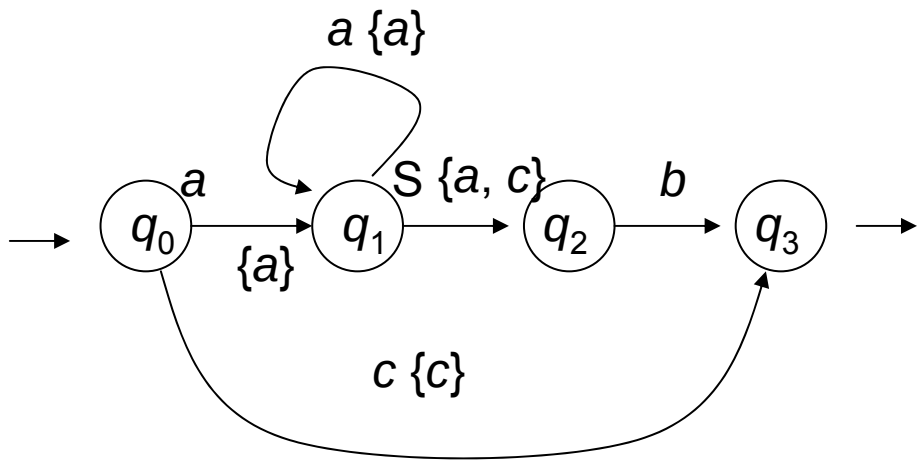
G_2 results $LL(1)$



$G_3 : S \to a^+ Sb \mid c$

$\{a^* a^n c b^n \mid n \geq 0\}$

G_3 violates condition $LL(1)$ in q_1

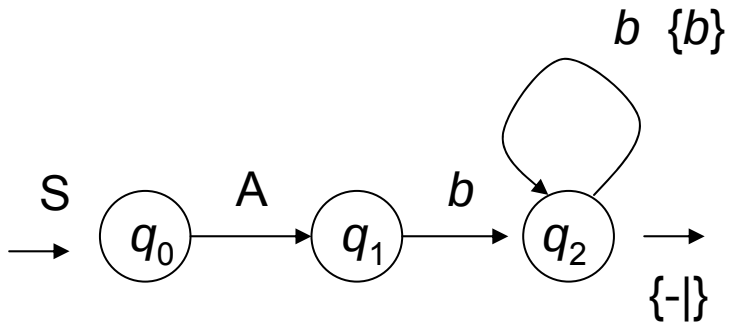


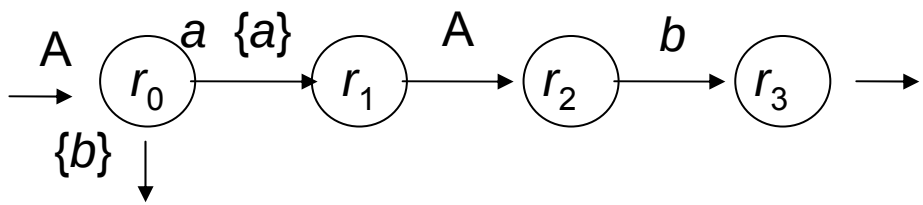
$$G_4: \quad S \to Ab^+$$

$$A \to aAb \mid \varepsilon$$

$$\left\{ a^n b^n b^+ \mid n \geq 0 \right\}$$

condition LL(1) is respected



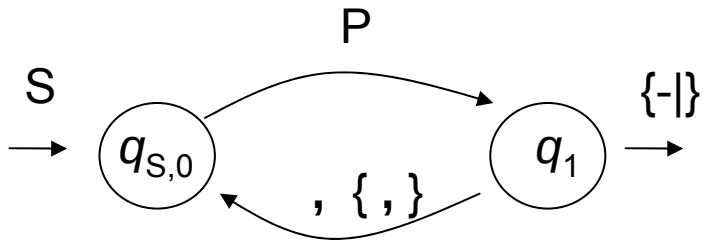


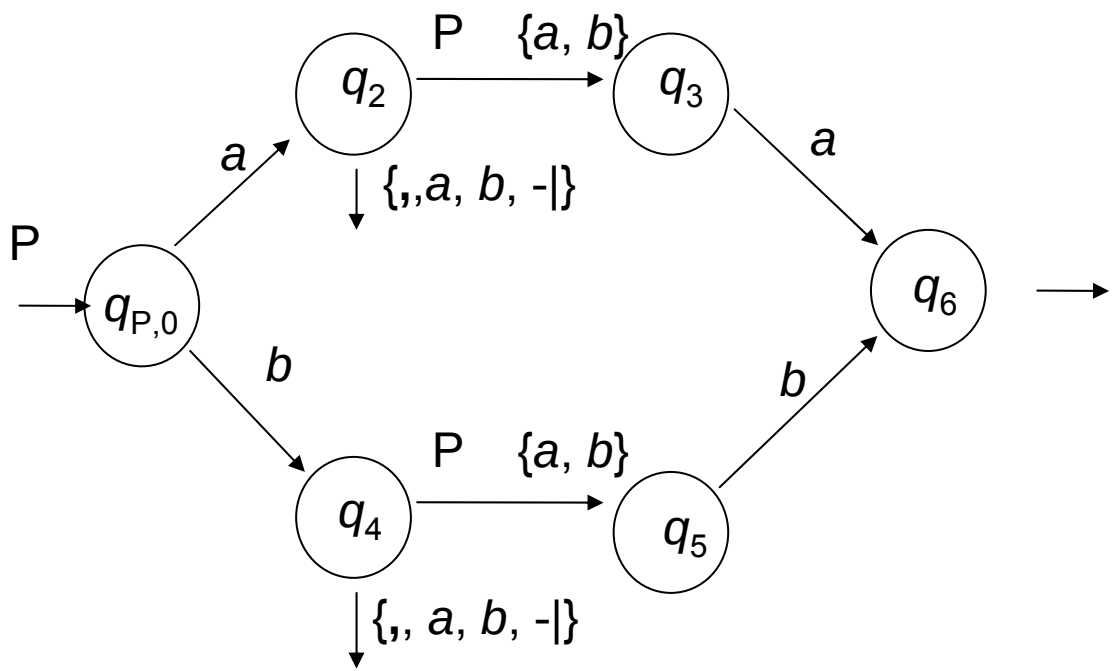
$S \to P(,P)^*$

$P \to aPa \mid bPb \mid a \mid b \mid$

condition LL(1) is not respected in

$q_2 \quad q_4$



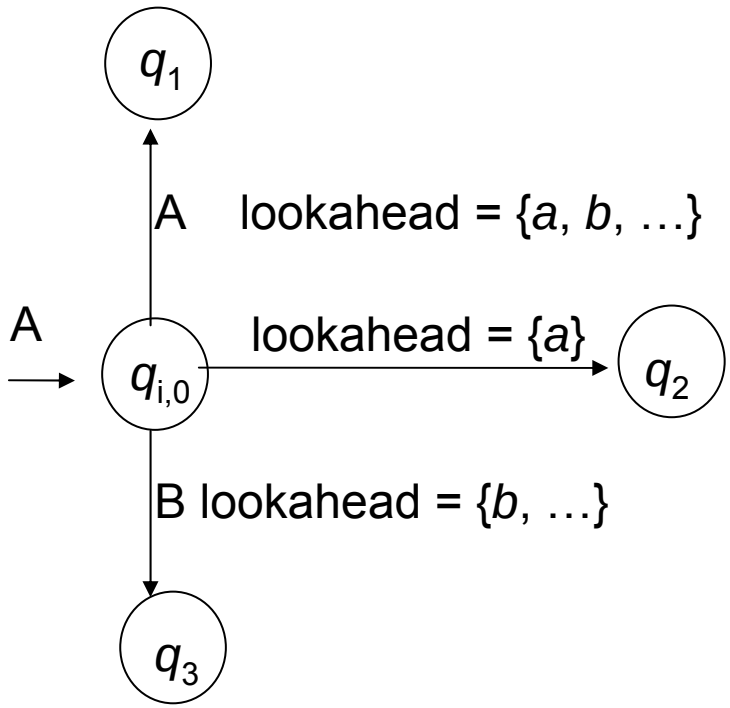


PROPERTY: a left recursive rule is not LL(1) – in fact, the lookahead set of the left recursive alternative contains all the lookahead sets of the other alternatives.

example:

$$A \rightarrow A.... \mid a.... \mid B$$

$$B \rightarrow b....$$



PROPERTY – if the grammar is LL(1) the recognition algorithm is deterministic.

In fact, in the disjunction of the lookahead sets of the arcs outgoing from the same state makes the conditions *a*, *b*, *c*, and *d* of step 5 in the following algorithm, mutually exclusive.

ALGORITHM – deterministic recursive descent - the pushdown automaton is:

begin

1. the source string is *x* and *cc* is the current input character
2. the stack alphabet is the disjoint union $Q = Q_A \cup Q_B$ ... of the state sets of all the machines of the recursive network, each expanding a specific non-terminal A, B, ...
3. the pushdown automaton itself has only one internal state, which is understood
4. at the beginning the stack contains the symbol $q_{S,0}$ (the initial state of the machine M_S expanding the axiom)
5. transitions: let $s \in q_A$ be the current stack top symbol, that is the current state of the currently active machine M_A:
   a) (shift move) if the move $\delta(s, cc) = s'$ is defined in the currently active machine M_A, read the current input character, pop *s* and push *s*' (to replace *s*)
   b) (call move) if the move $\delta(s, B) = s'$ is defined for a non-terminal B in the currently active machine M_A, suspend the current machine and execute a spontaneous move to push the symbol $q_{B,0}$ that represents the initial state of the new machine M_B, which since now will be the currently active machine
   c) (return move) if *s* is a final state in the currently active machine M_A, pop it, and supposing that state *r* (belonging to the machine M_C) emerges as the new stack top, resume machine M_C and execute the move $\delta(r, B) = s'$ in the current machine M_C, popping state *r* and pushing state *s*' (to replace *r*)
   d) (acceptance move) if *s* is a final state in the currently active machine M_S (the axiom machine) and *cc* = -|, terminate and accept the input string
   e) (reject move) otherwise, terminate and reject the input string

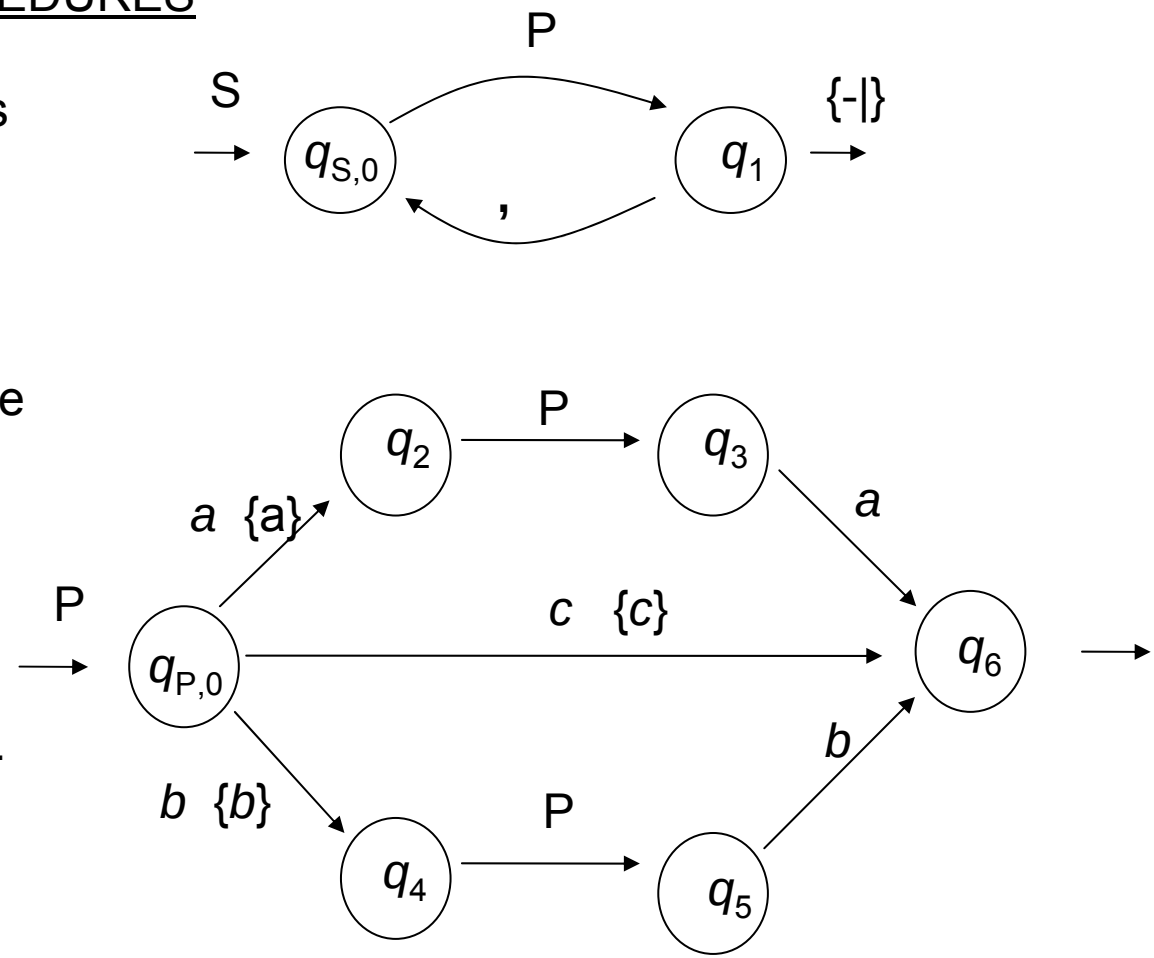
end

HOW TO IMPLEMENT THE PARSER AS A PROGRAM CONSISTING OF A SET OF RECURSIVE PROCEDURES

The parser is implemented as a set of *recursive syntactic procedures*, corresponding to the rules of the grammar. Each procedure is modeled according to the block scheme associated with one of the machines of the network representing the grammar, and is named after the non-terminal symbols associated with that machine.

$$S \rightarrow P(,P)^*$$
$$P \rightarrow aPa \mid bPb \mid c$$

EXAMPLE



HOW TO IMPLEMENT THE PARSER AS A PROGRAM CONSISTING OF A SET OF RECURSIVE PROCEDURES

```
procedure S
begin
1. call P
2. while (not cc = '-|') do
            if cc = ',' then
                        cc := next
                        call P
            else error
            end if
   end while
3. if cc = '-|' then
            accept and terminate
4. else error
   end if
end
```

```
procedure P
begin
1. if cc = 'a' then
            cc := next
            call P
            if cc = 'a' then
                        cc := next
            else error
            end if
2. else if cc = 'b' then
            cc := next
            call P
            if cc = 'b' then
                        cc := next
            else error
            end if
3. else if cc = 'c' then
            cc := next
4. else error
   end if
end
```

OBSERVATIONS

IN PROGRAM THAT IMPLEMENTS THE RECURSIVE DESCENT PARSER:

1) function "next" shifts the read head on the input string and reads the next char
2) if an arc is labeled by a terminal symbol, the next character is read (calling "next")
3) if an arc is labeled by a non-terminal symbol, the corresponding procedure is called
4) if the current character *cc* is not in the lookahead set, an error occurs
5) as the axiomatic rule contains a star operator and the corresponding machine contains a loop, the syntactic procedure containa a *while ... do* structure
6) in the case of an error, the parser terminates immediately and silently, although in the practice the parser should emit a warning
7) comparing the characters expected in a given state (i.e., those included in the lookahead sets labeling the outgoing arcs) with the current character, it is possible to emit a simple warning message (of the kind: the expected chars are ... but the current char is ...)
8) a well designed parser should go on with the computation also after the first error, so that the entire input string is analyzed in one compilation step, even if it contains several errors, as it is unacceptable for the user to stop after each error

USING LOOKAHEAD OF LENGTH GREATER THAN 1

In order to obtain a deterministic parser, when the LL(1) condition is not met, it is possible to examine not only the current input character, but also the subsequent one (in practice, the lookahead length is increased to 2). If in doing so the choice of the move becomes uniquely determined, then the current state satisfies the so-called LL(2) condition.

By generalizing the above observation, one obtains the LL(k) condition, where the length of the lookahed is a window of $k \geq 1$, characters.

In the practice, the LL(k) parsers use lookaheads of different lengths: each state uses the value of k that is strictly necessary to make it deterministic.

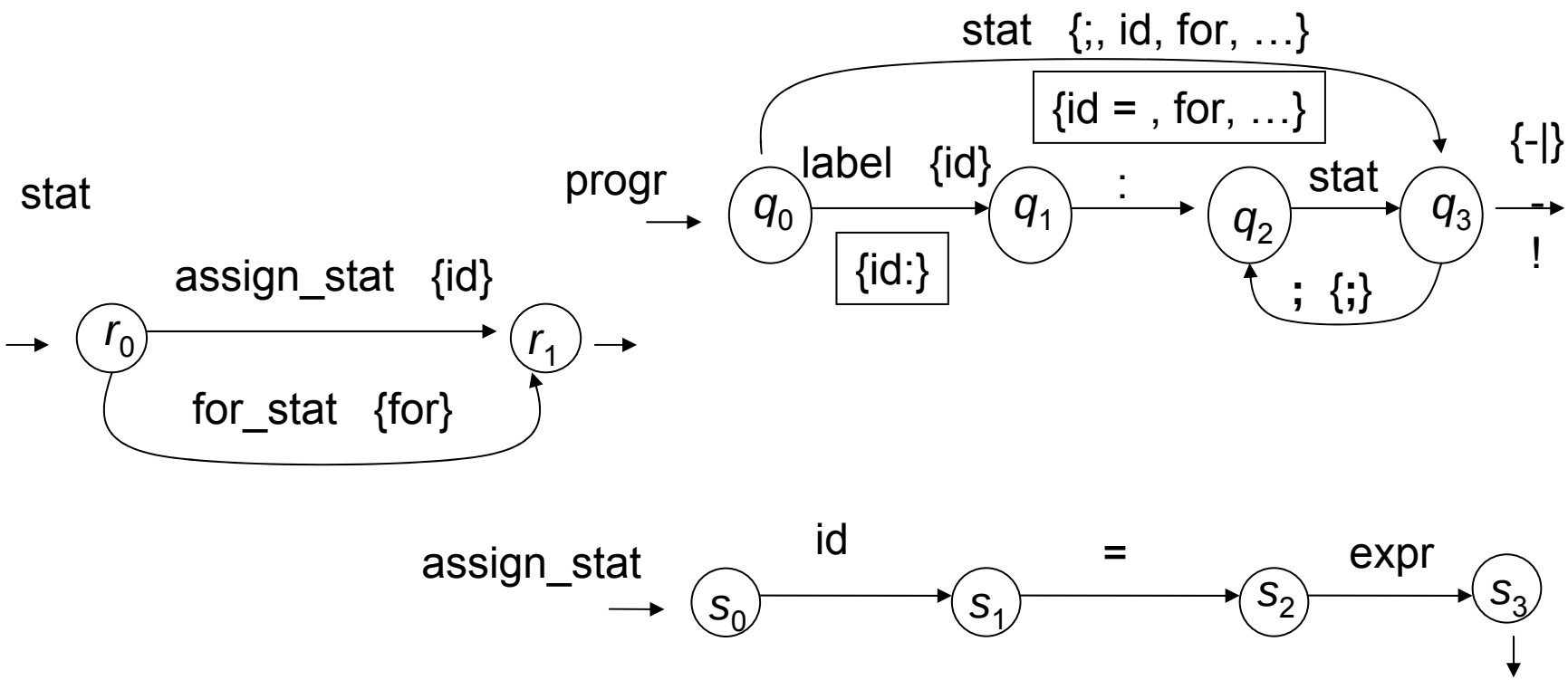
There exist also parsers that, when a lookahead of fixed finite length does not suffice, can look ahead in the input tape for an unbounded number of positions, as far as a terminal character is found that allows to solve deterministically the choice of the move to undertake.

EXAMPLE

$progr \to [label :] stat(;stat)^*$ $\quad stat \to assign\_stat \mid for\_stat \mid ....$

$assign\_stat \to id = expr$ $\quad for\_stat \to for...$

$label \to id$ $\quad expr \to ...$



Bibliography

- S. Crespi Reghizzi, *Linguaggi Formali e Compilazione*, Pitagora Editrice Bologna, 2006
- Hopcroft, Ullman, *Formal Languages and their Relation to Automata*, Addison Wesley, 1969
- A. Salomaa – *Formal Languages*, Academic Press, 1973
- D. Mandrioli, C. Ghezzi – *Theoretical Foundations of Computer Science*, John Wiley & Sons, 1987
- L. Breveglieri, S. Crespi Reghizzi, *Linguaggi Formali e Compilatori: Temi d'Esame Risolti,* web site (eng + ita)