FORMAL LANGUAGES

0) FORMAL LANGUAGES CLASSIFICATION

- **DEFINITIONS**:
  - Alphabet = finite non-empty **set of symbols**

$$\Sigma_1 = \{0,1\} \qquad \Sigma_2 = \{\alpha, \beta, \gamma, \dots\} \qquad \Sigma_3 = \{boy, girl, talks, the \dots\}$$

  - String (*word*) = finite **sequence of symbols** taken from an alphabet

$$s_1 = 01010110 \qquad s_2 = the\ girl\ talks$$

  - Length = number of symbols in a string

$$|s_1| = 7 \qquad |s_2| = 3$$

  - Empty string = string of length 0

$$|\varepsilon| = 0$$

  - Alphabet closure = set of **all strings generable** by an alphabet
    - Closure operator (Kleene) → *****

$$\{0,1\}* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$$

    - Positive closure operator → **+** (*no epsilon*)

$$\{0,1\}^+ = \{0, 1, 00, 01, 10, 11, 000, \dots\}$$

⚠ $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

  - Language = a set of **strings generated** by an alphabet (finite if the number of possible strings is finite)

$$L_1 = \{0^n 1^n \mid n \geq 1\} = \{01, 0011, 000111, \dots\} \qquad L_2 = \{\varepsilon\} \qquad L_3 = \emptyset$$

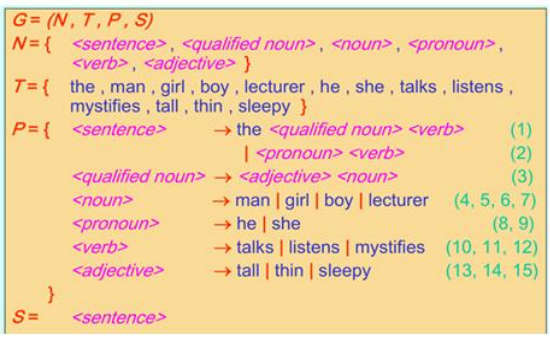
⚠ $L \subseteq \Sigma^*$

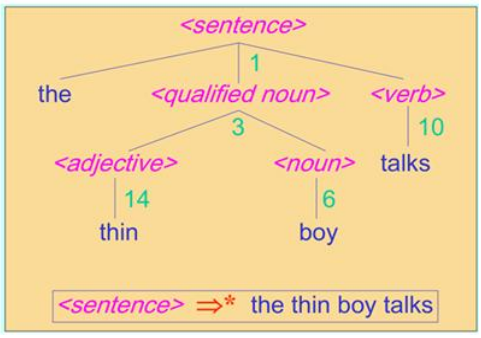
- **GRAMMARS**:
  - Grammar = 4-tuple that describes a language $G = (N, T, P, S)$ where:
    - N → alphabet of **non-terminal** symbols
    - T → alphabet of **terminal** symbols (is the alphabet of the language we want to describe)
      - $\boldsymbol{N \cap T = \emptyset}$
      - $\boldsymbol{V = N \cup T}$ is the alphabet of the grammar
    - P → **productions** = finite set of rules of how to produce the strings of the language
    - S → **start symbol** (that is non-terminal [$S \in N$])
  - Derivation = if $\alpha \to \beta$ is a production of the grammar G
    - if $\sigma = \gamma\alpha\delta$ and $\tau = \gamma\beta\delta$, then $\sigma \Rightarrow \tau$
    - if $\sigma_0 \Rightarrow \sigma_! \Rightarrow \cdots \Rightarrow \sigma_k$ then $\sigma_0 \Rightarrow* \sigma_k$ where "⇒∗" indicates 0 or more repetitions of derivation

The **LANGUAGE PRODUCED** by $G = (N, T, P, S)$ is defined as $\boldsymbol{L(G) = \{w \mid w \in T^*\ ;\ S \Rightarrow* w\}}$

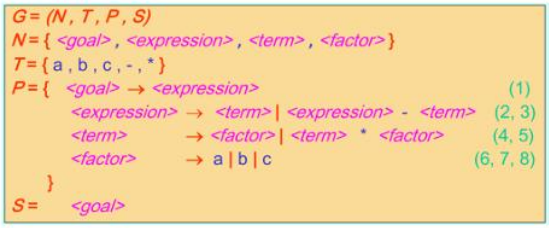
⚠ Equivalent grammars = produce the same language

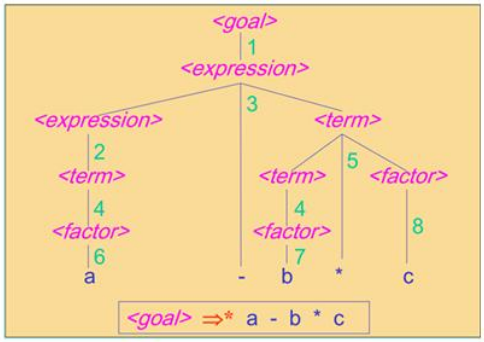
EXAMPLE: starting from the start symbol S = <*sentence*> and applying the productions, it's possible to generate all the strings that compose the language





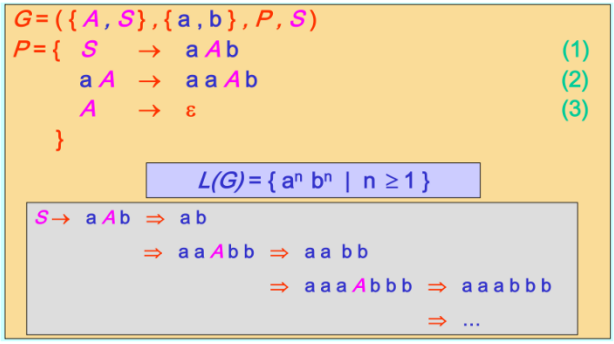
EXAMPLE (same as before, but now we have **recursion** of <*term*>, so it means the language is **not finite**):





GRAMMAR CLASSIFICATION:

- Type 0 = **PHRASE-STRUCTURE** = $P = \{\alpha \to \beta \mid \boldsymbol{\alpha} \in \boldsymbol{V}^+ ;\ \boldsymbol{\alpha} \notin \boldsymbol{T}^+ ;\ \boldsymbol{\beta}\ \in \boldsymbol{V}^*\}$, so α should contain a non-terminal symbol, while β can be any symbol of the alphabet related to the grammar V^* (so including ε [only this type includes ε])

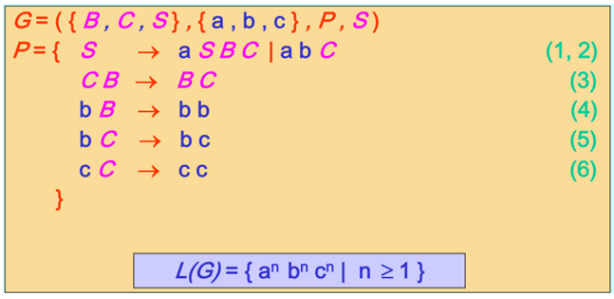


- Type 1 = **CONTEXT-SENSITIVE** = $P = \{\alpha \to \beta \mid \alpha \in V^+ ;\ \alpha \notin T^+ ;\ \beta\ \in V^*;\ |\boldsymbol{\alpha}| \leq |\boldsymbol{\beta}|\}$, so the added condition is that the production must be monotonic (size of β is always >= size of α)

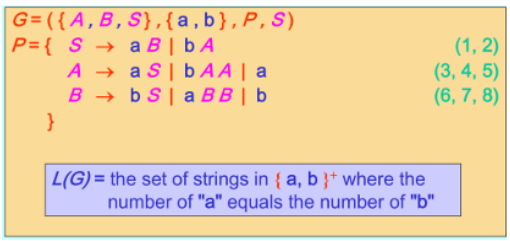
  ⚠ Definition of Chomsky = makes clear the meaning of "context-sensitive":

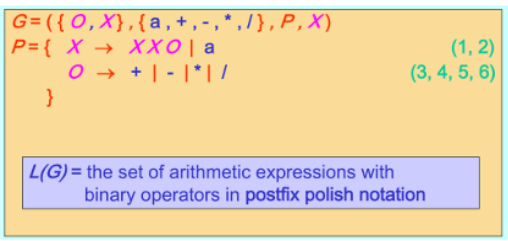
  $$P = \{\alpha A \beta \to \alpha\gamma\beta \mid \alpha, \beta \in V^* ;\ A \in N ; \gamma \in V^+\}$$

  In this definition $\alpha\beta$ is the context that can't be modified by the productions: shows that a production can't modify the surroundings of the non-terminal symbol



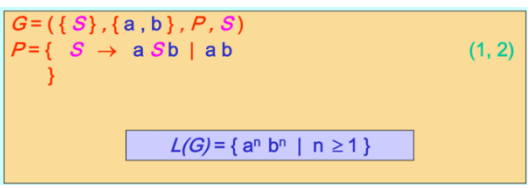
- Type 2 = **CONTEXT-FREE** = $P = \{A \to \beta \mid A \in N\ ;\ \beta\ \in V^+\}$; deletes context (es. programming languages)





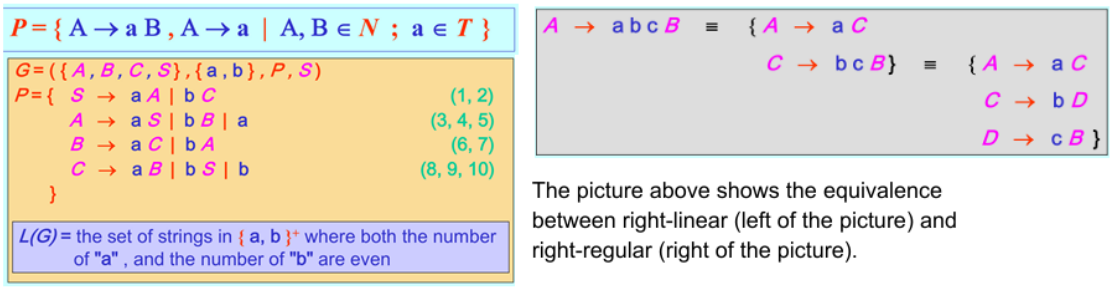
- **LINEAR** = $P = \{A \to xBy\ ,\ A \to x,\ A \to yBx \mid A, B \in N\ ;\ x \in T^+;\ y \in T^*\}$; intermediate between type 2 and type 3 and restrict context-free grammars adding the constraint of having at most 1 non-terminal symbol in the right end side of each production

Example (it's possible to generate the same language starting from different grammars [see the example of type 0 to compare]) →



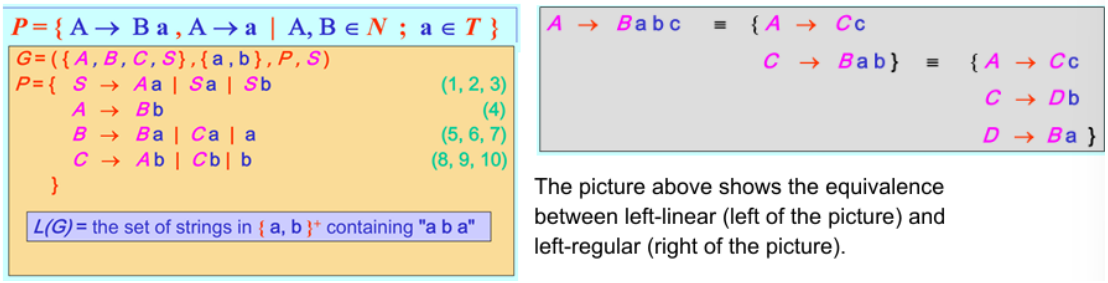
- Type 3 = **REGULAR**:
  - **RIGHT-LINEAR** = $P = \{A \to xB\ ,\ A \to x \mid A, B \in N\ ;\ x \in T^+\}$; at most 1 non-terminal symbol in the right end side of each production (as before), but it must be in the rightmost position
  - **LEFT-LINEAR** = $P = \{A \to Bx\ ,\ A \to x \mid A, B \in N\ ;\ x \in T^+\}$; the non-terminal symbol must be in the leftmost position

  A further simplification that doesn't restrict the possible languages obtainable consists in restricting the domain of x from T^+ that represents the entire closure to T:
  - **RIGHT-REGULAR** = $P = \{A \to xB\ ,\ A \to x \mid A, B \in N\ ;\ x \in T\}$

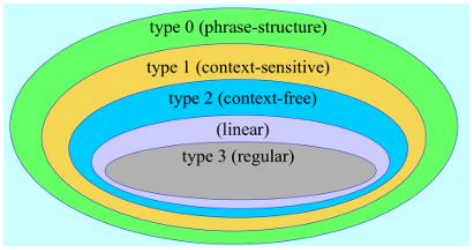


The picture above shows the equivalence between right-linear (left of the picture) and right-regular (right of the picture).

  - **LEFT-REGULAR** = $P = \{A \to Bx\ ,\ A \to x \mid A, B \in N\ ;\ x \in T\}$



The picture above shows the equivalence between left-linear (left of the picture) and left-regular (right of the picture).



LANGUAGE CLASSIFICATION: a language is a type-n if it can be produced by a type-n grammar. Every type is a restriction of another (Type 0 includes all the possible languages, Type 3 is the most specific)

The **membership problem** consists in computing if a string ω belongs to a language L. Complexity is highest with type 0, lowest with type 3; the largest language (Σ^*, that includes all the possible strings) has the lowest complexity (because every string is part of the set)

⚠ Extended grammar = allows ε-productions (empty strings) in Type 1, 2 and 3 grammars (we before said that only Type 0 includes empty strings); for each class C, the corresponding extended is $C' = C \cup \{L \cup \{\varepsilon\} \mid L \in C\}$

1) REGULAR LANGUAGES

REGULAR SET = list of strings (similar to a **REGULAR LANGUAGE**). Sets "axiomatically REGULAR" are:
- $\emptyset$ (empty set)
- $\{\varepsilon\}$ (set with only the empty string)
- $\{a\}$ (set containing any symbol $a \in \Sigma$)

All the other regular sets are combination of them that uses the following axioms:
- Union → union of 2 regular sets is a regular set ($P \cup Q$)
- Concatenation → concatenation of 2 regular sets is a regular set ($PQ = \{x\,y \mid x \in P; y\ \in Q\}$) (es. $P = \{a, ab\}$, $Q = \{abc\} \to PQ = \{aabc, ababc\}$)
- Closure → closure of a regular set is a regular set (es. $P = \{a, ab\} \to P^* = \{\varepsilon, a, ab, aa, aab, aba, abab\}$)

REGULAR EXPRESSION = algebraic expression that represents a regular set/language. The **primitive regular** expressions are:

- φ denoting $\emptyset$ (algebraic zero)
- ε denoting $\{\varepsilon\}$ (algebraic one)
- a denoting $\{a\}$ (where $a \in \Sigma$)

If p and q are regular expressions denoting the sets P and Q, then the following are regular expressions (axioms):
- Union → $p|q$ (algebraic sum)
- Concatenation → pq (algebraic multiplication)
- Closure → p^*, q^* (closures)

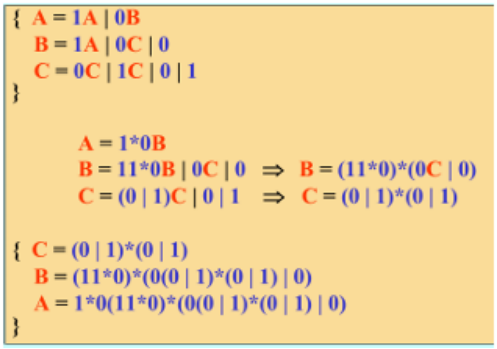
⚠ These are the bases, but others can be derived (es. $p^+ = pp^*$)

Examples (regular expressions):
- ❖ Strings over {0,1} containing 2 ones → $0*10*10$
- ❖ Strings over {0,1} without consecutive equal symbols → $(1|\varepsilon)(01)*(0|\varepsilon)$
- ❖ Set of decimal characters → $digit = 0|1|2|3|4|5|6|7|8|9$
- ❖ Set of strings representing decimal integers → $digit\ digit *= digit^+$
- ❖ Set of alphabetic characters → $A|B|\ldots|Z|a|b|\ldots|z$

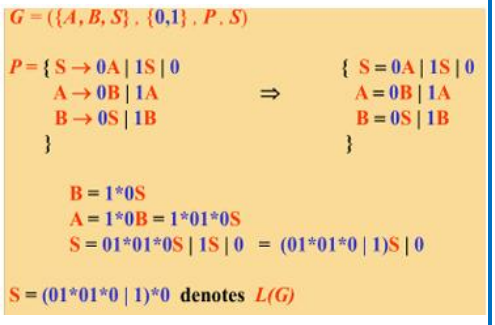
⚠ 2 regular expressions are **equivalent** if they denote the same regular set

Regular expressions **properties**:
- Commutative → $\alpha|\beta = \beta|\alpha$
- Associative → $\alpha|(\beta|\gamma) = (\alpha|\beta)|\gamma$ and also $\alpha(\beta\gamma) = (\alpha\beta)\gamma$
- Distributive → $\alpha(\beta|\gamma) = \alpha\beta|\alpha\gamma$ and also $(\alpha|\beta)\gamma = \alpha\gamma|\beta\gamma$
- Neutral element:
  - $\alpha|\varphi = \alpha$ ($\boldsymbol{\varphi}$ = sum/union neutral element)
  - $\alpha\varepsilon = \varepsilon\alpha = \alpha$ ($\boldsymbol{\varepsilon}$ = multiplication/concatenation neutral element)
- Particular properties (derived from the others):
  - $\alpha\varphi = \varphi\alpha = \varphi$
  - $\alpha|\alpha = \alpha$
  - $\varphi *= \varepsilon *= \varepsilon$
  - $\alpha *= \alpha * \alpha *= (\alpha^*) *= \alpha\alpha * |\varepsilon = \alpha^+|\varepsilon$

EQUATIONS OF REGULAR EXPRESSIONS: if α and β are regular expressions $\boldsymbol{X = \alpha X|\beta}$ is an equation with unknown $\boldsymbol{X}$ as solution $\boldsymbol{X = \alpha * \beta}$ [demonstration → $\alpha X|\beta = \alpha\alpha * \beta|\beta = (\alpha\alpha * |\varepsilon)\beta = \alpha * \beta$].
A set of equations $\{X_1, X_2, \ldots, X_n\}$ is composed by n equations as $\boldsymbol{X_i = \alpha_{i0}|\alpha_{i1}X_1|\ldots|\alpha_{in}X_n}$ where each α_{ij} is a regular expression over any alphabet without unknows; to solve a set of equations, we need to put every equation in the form $\boldsymbol{X_i = \alpha X_i|\beta}$ and **substitute $\boldsymbol{X_i}$ with $\boldsymbol{\alpha * \beta}$** in the next equations (*from first to last*). Then, for every equation (*from last to first*), we need to **solve it and substitute $\boldsymbol{X_i}$** inside the previous equations:



Let's see demonstration of ***Regular sets ≡ Regular languages***, starting from ***Right-linear languages ⊆ Regular sets***: let $G = (\{A_1, A_2, \ldots, A_n\}, T, P, A_1)$ be a right-linear grammar and transform each rule of the grammar $A_i \rightarrow \alpha_{i0}|\alpha_{i1}A_1|\ldots|\alpha_{in}A_n$ to an equation of regular expressions $A_i = \alpha_{i0}|\alpha_{i1}A_1|\ldots|\alpha_{in}A_n$. If we solve the resulting set of equations, we will get the regular expression corresponding to A_1 (which was the S [starting point] of G). Using this regular expression, we can generate all the strings of $L(G)$.

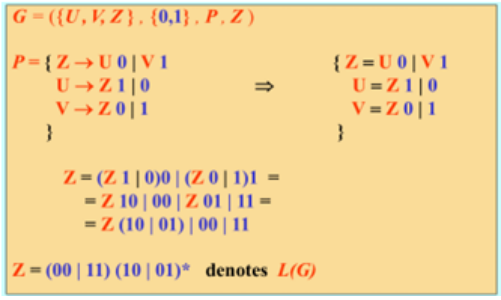


Then we need to verify ***Regular sets ⊆ Right-linear languages*** proving that:

1. The regular sets $\emptyset, \{\varepsilon\}$ and $\{a | a \in \Sigma\}$ can be generated by a right-linear language:
   a. $G_1 = (\{S\}, \Sigma, \emptyset, S) \Rightarrow L(G_1) = \emptyset$
   b. $G_2 = (\{S\}, \Sigma, \{S \to \varepsilon\}, S) \Rightarrow L(G_2) = \{\varepsilon\}$
   c. $G_3 = (\{S\}, \Sigma, \{S \to a | a \in \Sigma\}, S) \Rightarrow L(G_3) = \{a |\ a \in \Sigma\}$

2. The combination of 2 regular sets which are also right-linear languages is still a right-linear language; let's consider $G_1 = (N_1, \Sigma, P_1, S_1)$ and $G_2 = (N_2, \Sigma, P_2, S_2)$ where $N_1 \cap N_2 = \emptyset$:
   a. **Union**: $L(G_1) \cup L(G_2) = L(G_4)$ is a right-linear language because $G_4 = (N_1 \cup N_2 \cup \{S_4\}, \Sigma, P_1 \cup P_2 \cup \{S_4 \to S_1 | S_2\}, S_4)$
   b. **Combination**: $L(G_1)L(G_2) = L(G_5)$ is a right-linear language because $G_5 = (N_1 \cup N_2, \Sigma, P_2 \cup P_5, S_1)$ where $P_5 = \{A \to xB \text{ if } A \to xB \in P_1; A \to xS_2 \text{ if } A \to x \in P_1\}$ (a modified version of P_1)
   c. **Closure**: $L(G_1) *= L(G_6)$ is a right-linear language because $G_6 = (N_1 \cup \{S_6\}, \Sigma, \{S_6 \to S_1 | \varepsilon\} \cup P_6, S_6)$ where $P_6 = \{A \to xB \text{ if } A \to xB \in P_1; A \to xS_6 \text{ if } A \to x \in P_1\}$ (a modified version of P_1)

So now we have verified that ***Regular sets ≡ Right-linear languages***. But that is true also for ***Left-linear languages*** (where we can use $X = \beta a *$ to solve the regular expressions $X = X\alpha | \beta$ [see the image]). So, we can say that ***right-linear languages ≡ left-linear languages*** and these are in fact both different types of **regular languages**.

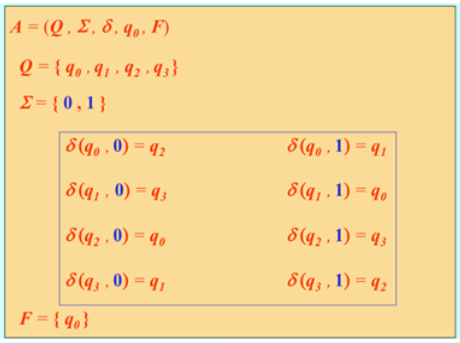


2) FINITE AUTOMATA (FA)

Solving the membership problem using the languages representations we've seen so far is not easy:

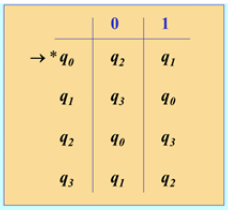
- *Grammars*: we can try to generate all the possible strings with the length of the input string, but the computation time increases with the length of the input string
- *Regular expressions*: compare the input string with the expression, but considering that closures give different choices

With the **FINITE AUTOMATA** (**FA**) the membership problem can be solved more efficiently (linearly depending on the size of the string):

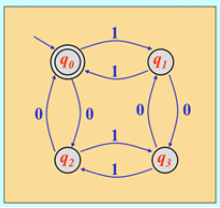


- **DFA** (**Deterministic FA**) → a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$:
  - Q = finite set of **states** (not empty; $Q \neq \emptyset$)
  - Σ = **alphabet** of input symbols
  - δ = transition **function** (define how the machine changes its state according to the new arrived input symbol [δ: $Q \times \Sigma \to Q$])
  - q_0 = **start state**
  - F = set of **finale states**

  There are 2 representations:
  - **Transition table** = tabular representation of δ
  - **Graph** (**transition diagram**) = a node for each state and an arc for each possible transition; the start state (as an entering non labeled arc) and the final states are marked by a double circle



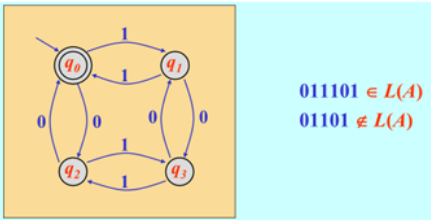
	0	1
→* q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2



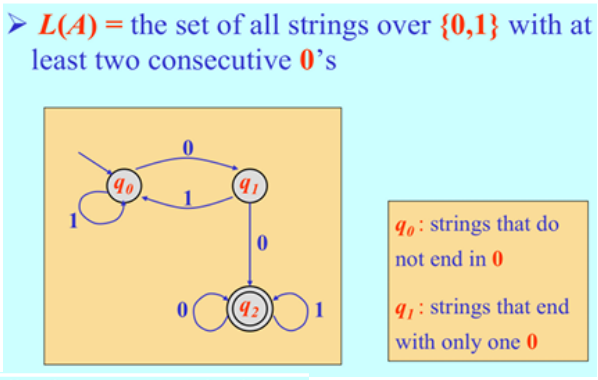
The transition function (δ) can also accept more than 1 symbol at time, so its domain can be extended from $Q \times \Sigma$ to $\boldsymbol{Q} \times \boldsymbol{\Sigma}^*$; let's demonstrate:

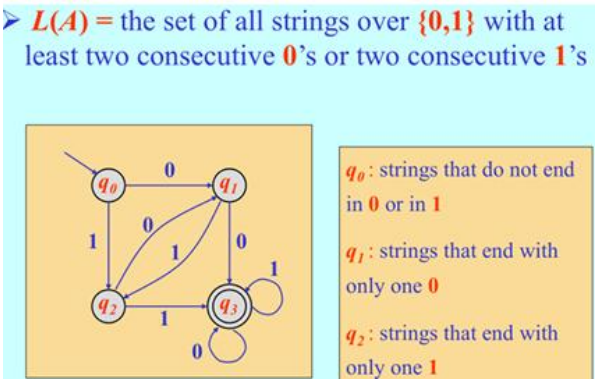
- ❖ if we don't provide any symbol (ε), the state remains '=' → $\delta(q, \varepsilon) = q$
- ❖ we can define δ for a string (aw) of length $n + 1$ (where a = symbol and w = string of length n) → $\delta(q, aw) = \delta(\delta(q, a), w)$ [$a \in \Sigma, w \in \Sigma^*$]

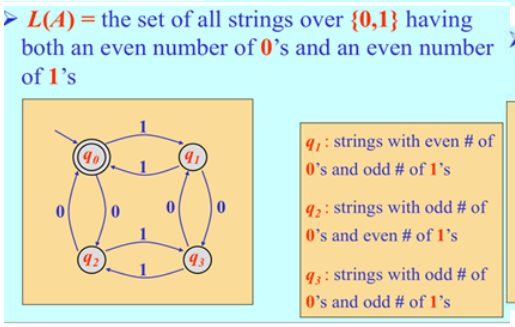
So, the language accepted by $\boldsymbol{A}$ is $L(A) = \{w | w \in \Sigma^*; \delta(q_0, w) \in F\}$

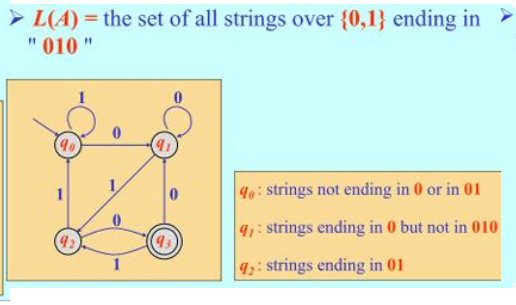


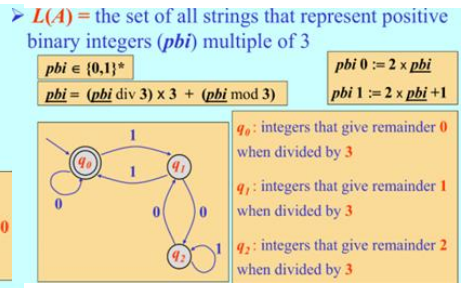
Examples:











⚠ There can be different more DFAs to represent a language. Also, building the graph is not really easy: for this reason, we will define the **NFAs which are easier to design** and let us to transform **regular expressions in DFA**

- **NFA** (**Non-deterministic FA**) → a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$:
  - Q = finite set of **states** (not empty; $Q \neq \emptyset$)
  - Σ = **alphabet** of input symbols
  - δ = transition **function**; $\delta: Q \times \Sigma \rightarrow P(Q)$ which is a set of states (given a state and an input symbol, there can be more than 1 possible future state [powerset of $Q \rightarrow ||P(Q)|| = 2^{||Q||}$])
  - q_0 = **start state**
  - F = set of **finale states**

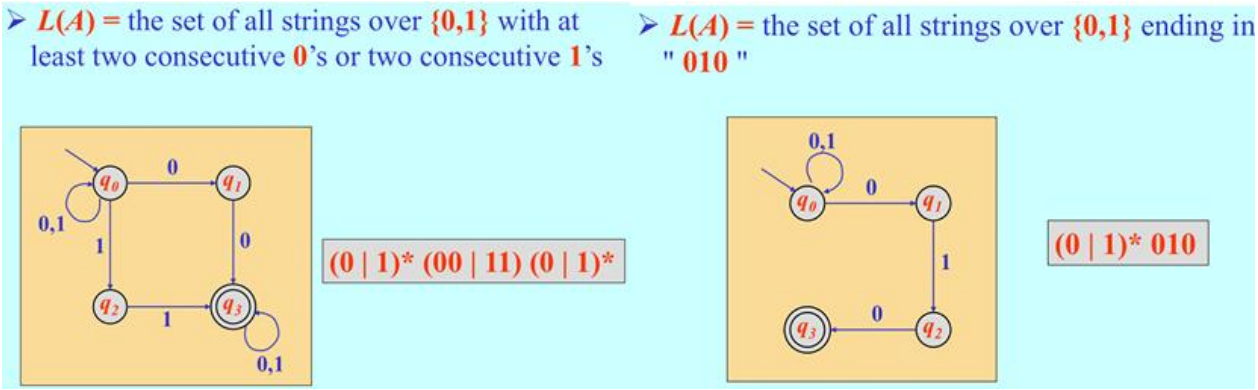
In this case the domain of δ is further extended to $Q \times \Sigma \rightarrow Q \times \Sigma^* \rightarrow P(Q)$ because we can now have different states as output of the transition function (δ):

- ❖ if we don't provide any symbol (ε), we could have as result different states → $\delta(q, \varepsilon) = \{q\}$
- ❖ $\delta(q, aw) = \cup_i (\delta_i(q, a), w)$ where $a \in \Sigma, w \in \Sigma^*$; we can have as result the union of possible future states starting from the q state and adding a symbol a
- ❖ $\delta(\{q_1, q_2, \dots, q_n\}, w) = \cup_j \delta(q_j, w)$

So, the language accepted by ***A*** is $L(A) = \{w | \ w \in \Sigma^*; \delta(q_0, w) \in F \neq \emptyset\}$; this means that the string is accepted if we reach at least 1 final state as a result of applying δ

⚠ A **DFA is a special case of NFA**

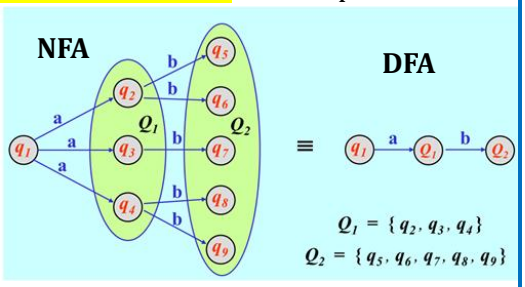
Examples:



In these examples we see that a loop arc with (0,1) (from the node to the node) is (0|1)* **in regular expressions**; also, the sequence is traduced in regular expression (in the first one 00 or 11 becomes (00|11))

⚠ In these examples, we also see that the initial state (q_0) has more than 1 arc for a single symbol (this is the characteristic of NFAs)

⚠ While NFA are easier to design, the **membership problem is more difficult to solve** with respect of DFAs because at each state there are more possibilities for each symbol



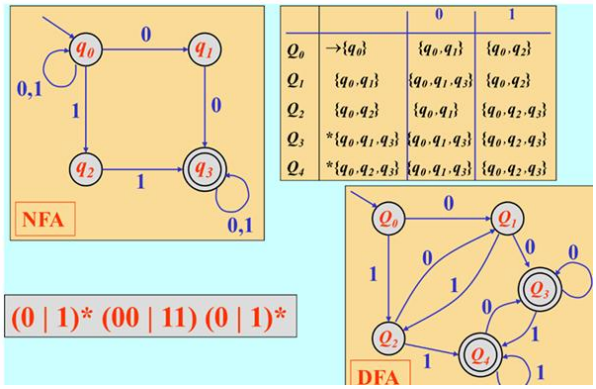
We can solve this issue about difficulty of membership problem through **EQUIVALENCE of NFA and DFA**: starting from an NFA it's possible to build a DFA that describes the same language. We know that the number of states in a DFA is finite, so also the powerset (set of all the subsets) of states must be finite (bigger, but finite): the DFA EQUIVALENT to the NFA has states that represent sets of states of the NFA.

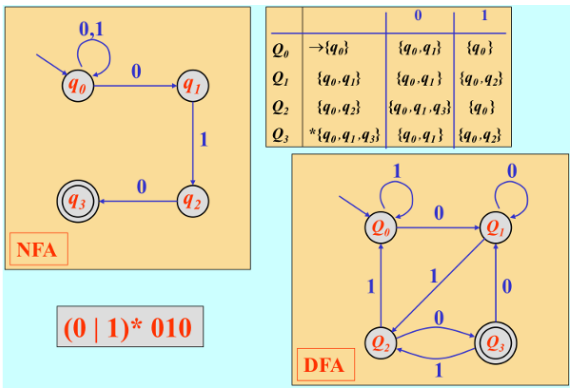
Let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ = NFA and let's **construct a DFA** $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ where:

- $Q_D \subseteq P(Q_N)$
- $\delta_D(S, a) = \cup_i \delta_N(p_i, a)$ where $p_i \in S \in Q_D$
- $F_D = \{S | S \in Q_D; S \cap F_N \neq \emptyset\}$

By construction we see that $L(D) = L(N)$ and so $\boldsymbol{NFA \equiv DFA}$ (so we can use the general $\boldsymbol{FA}$).

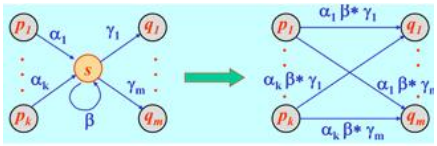
In practice, how we create the equivalent DFA from an NFA? We build a table that represents the transitions from the set of states of the NFA. For example, starting from q_0, using the NFA it's possible to reach $\{q_0, q_1\}$ (that becomes the state Q_1) using 0 as input symbol, or $\{q_0, q_2\}$ (that becomes Q_2) using 1 as input symbol. We continue the algorithm until we have covered all the possible sets of states [remember that the sets of states denoted with capital Q will become the states of the new DFA generated]

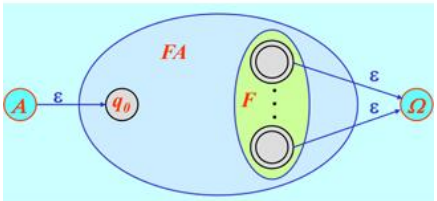




⚠ The ***worst scenario*** is obviously to have in the equivalent DFA 2^n states (with n = states of NFA)

FA languages $\subseteq$ ***Regular sets***: it's possible to eliminate states in an FA by maintaining all the paths and labeling the transitions with regular expressions. Given a finite state automaton $FA = (Q, \Sigma, \delta, q_0, F)$, we can add an initial state A and a final state Ω; after we do that, if we eliminate **all** the states in the middle, we will have only an arc from A to Ω that represents the regular expression of the language $L(FA)$.

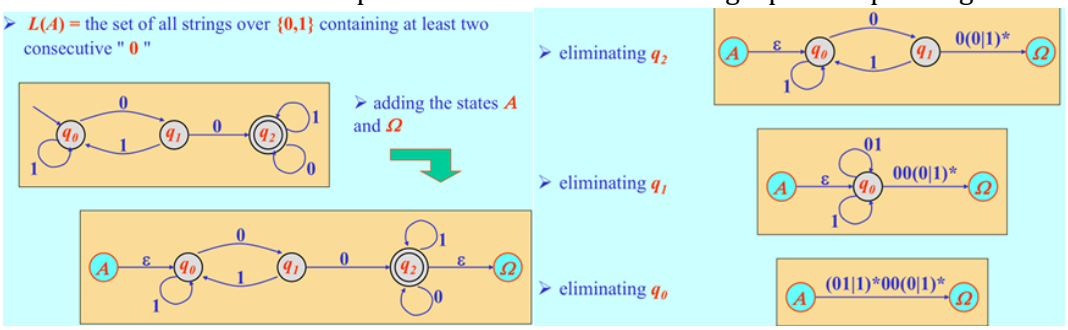


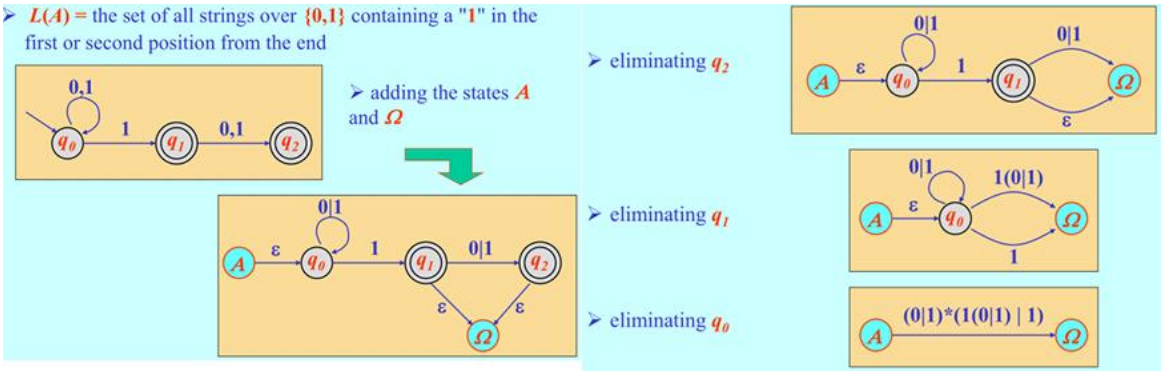


Now it's possible to **transform an FA in a regular expression**:

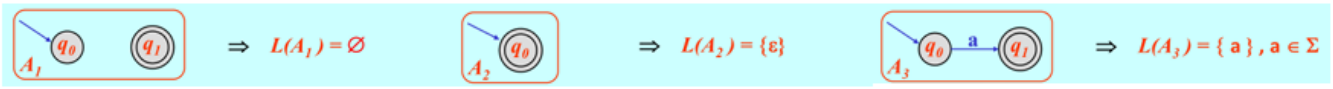
- states with 1 input arc and 1 output arc can be simplified as an arc that correspond to the concatenation of the regexps of the 2 arcs
- 2 arcs with same input and same output state can be simplified with an arc corresponding to the union of the regexps of the 2 arcs
- a state with a recursive arc can be simplified as the closure of the regexp corresponding to the arc

Examples:



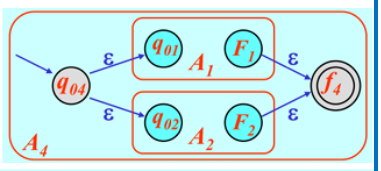


Regular sets $\subseteq$ ***FA languages***: the regular sets $\emptyset$, $\{\varepsilon\}$ and $\{a \mid a \in \Sigma\}$ are accepted by finite state automata and can be represented as shown below:

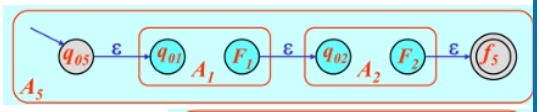


Let $A_1 = \{Q_1, \Sigma, \delta_1, q_{01}, F_1\}$ and $A_2 = \{Q_2, \Sigma, \delta_2, q_{02}, F_2\}$ be *FAs* where $Q_1 \cap Q_2 = \emptyset$:

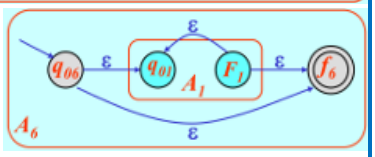
- the language $L(A_1) \cup L(A_2)$ is accepted by an FA A_4, which has a start state q_{04} connected with empty string arcs to q_{01} and q_{02}, and a final state F_4 connected with empty string arcs to F_1 and F_2



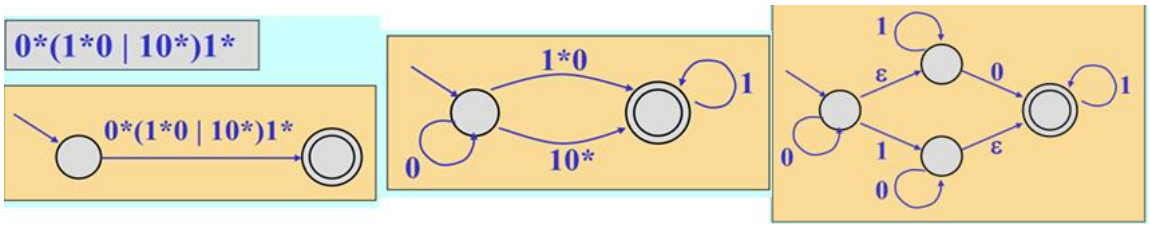
- the language $L(A_1)L(A_2)$ is accepted by an FA A_5, which concatenates the 2 FAs (A_1 and A_2)



- the language $L(A_1)^*$ is accepted by an FA A_6, which represents the closure of $L(A_1)$

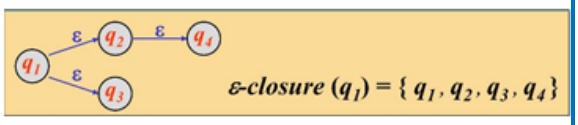


So using the properties above, it's possible to **transform a regexp to FA** substituting unions, concatenations and closures.



The resulting FA is non-deterministic because it can change state without consuming a symbol, with ε-transitions and it's called ***ε*-NFA**. The function ε-closure(q) represents the set of states (including q itself) that can be reached from state q just using the empty string (ε) without using a symbol:

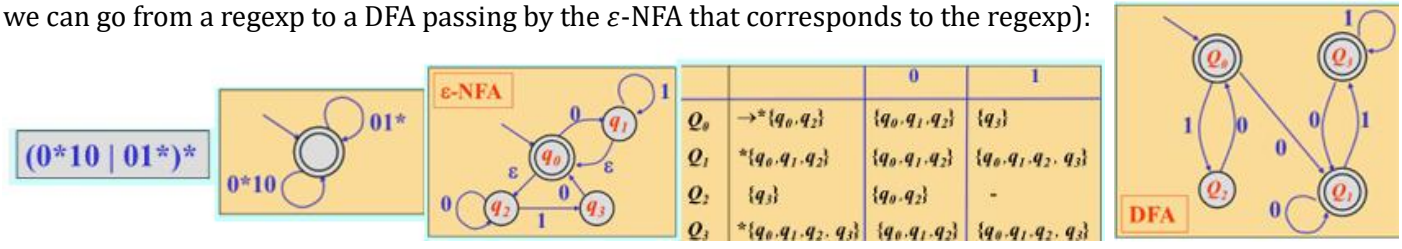
***ε*-closure**$(\{q_1, q_2, \dots, q_n\}) = \cup_i\ \varepsilon\text{-closure}(q_i)$

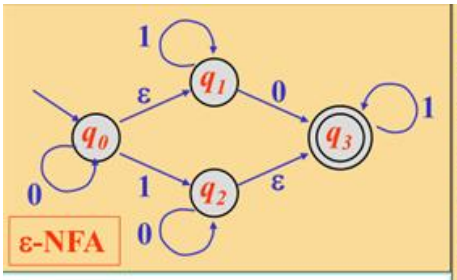


***ε*-NFA ≡ DFA**: let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ be an ***ε*-NFA** and let's construct a DFA $D = (Q_D, \Sigma, \delta_D, \varepsilon\text{-closure}(q_0), F_D)$ where:

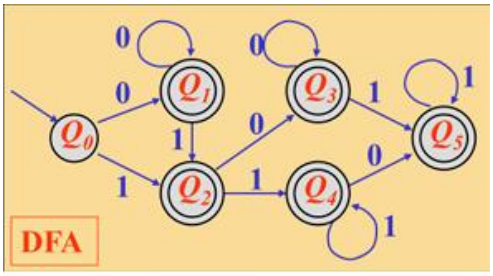
- $Q_D \subseteq P(Q_N)$
- $\delta_D(S, a) = \ \varepsilon\text{-closure}\big(\cup_i\ \delta_N(p_i, a)\big)$ where $p_i \in S \in Q_D$
- ε-closure(q_0) because we consider every element of the ε-closure of q_0 as initial state in the DFA
- $F_D = \{S \mid S \in Q_D; S \cap F_N \neq \emptyset\}$

So, by construction $L(D) = L(N)$. Like before for the NFA, now let's build the corresponding DFA of an ε-NFA (so we can go from a regexp to a DFA passing by the ε-NFA that corresponds to the regexp):





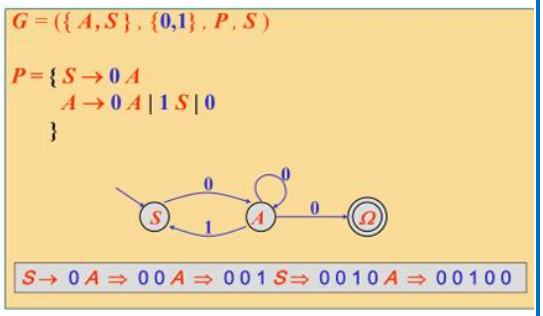
		0	1
Q_0	$\rightarrow\{q_0,q_1\}$	$\{q_0,q_1,q_3\}$	$\{q_1,q_2,q_3\}$
Q_1	$*\{q_0,q_1,q_3\}$	$\{q_0,q_1,q_3\}$	$\{q_1,q_2,q_3\}$
Q_2	$*\{q_1,q_2,q_3\}$	$\{q_2,q_3\}$	$\{q_1,q_3\}$
Q_3	$*\{q_2,q_3\}$	$\{q_2,q_3\}$	$\{q_3\}$
Q_4	$*\{q_1,q_3\}$	$\{q_3\}$	$\{q_1,q_3\}$
Q_5	$*\{q_3\}$	-	$\{q_3\}$



FA languages ≡ Regular sets: to demonstrate this we must verify it for both right-regular and left-regular languages:

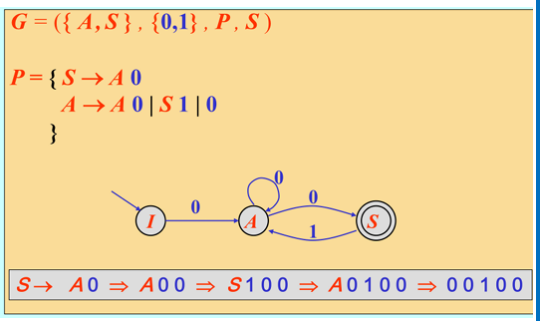
1. Let $G = (N, T, P, S)$ be a **right-regular** grammar and let's construct a FA $A = (Q, T, \delta, S, F)$ where:
   a. $\boldsymbol{Q} = N \cup \{\Omega\}$ with $\Omega \notin N$
   b. $\boldsymbol{F} = \{\Omega\}$
   c. $\boldsymbol{\delta} = \{\delta(A,a) = B$ if $A \rightarrow aB \in P; \delta(A,a) = \Omega$ if $A \rightarrow a \in P\}$. This means that: in 1st case productions $A \rightarrow aB$ correspond to an arc labeled a from A to B; in 2nd case productions $A \rightarrow a$ correspond to an arc labeled a that goes to a final state

   By construction, $L(G) = L(A)$

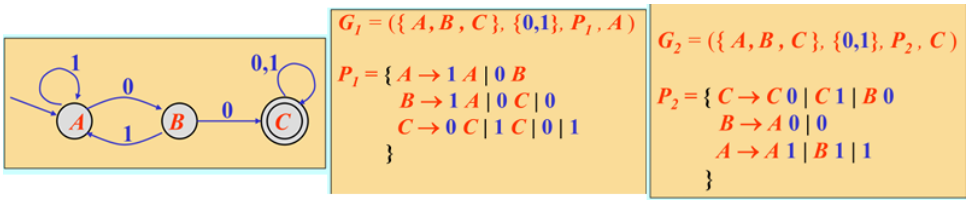


2. Let $G = (N, T, P, S)$ be a **left-regular** grammar and let's construct a FA $A = (Q, T, \delta, I, \{S\})$ where:
   a. $\boldsymbol{Q} = N \cup \{I\}$ with $I \notin N$
   b. $\boldsymbol{F} = \{S\}$
   c. $\boldsymbol{\delta} = \{\delta(B,a) = A$ if $A \rightarrow Ba \in P; \delta(I,a) = A$ if $A \rightarrow a \in P\}$. This means that: in 1st case productions $A \rightarrow Ba$ correspond to an arc labeled a from B to A; in 2nd case productions $A \rightarrow a$ correspond to an arc labeled a that goes from the start state of the automaton to A

   By construction, $L(G) = L(A)$



Example (DFA with its corresponding right-regular grammar and left-regular grammar):



MINIMUM-STATE DFA & minimization → let $DFA = (Q, \Sigma, \delta, q_0, F)$ be a deterministic FA; we define:

- ✓ Distinguishability = 2 states p and q are distinguishable if there's a string $w \in \Sigma^*$ such that $\delta(\boldsymbol{p}, \boldsymbol{w}) \in F$ and $\delta(\boldsymbol{q}, \boldsymbol{w}) \notin F$
- ✓ Equivalence = 2 states p and q are equivalent ($\boldsymbol{p} \equiv \boldsymbol{q}$) if they are non-distinguishable for any string $w \in \Sigma^*$

A DFA is **minimum-state** if it doesn't contain equivalent states, so to minimize a DFA we need to find the equivalent states (to do it, we need to test all the possible strings).

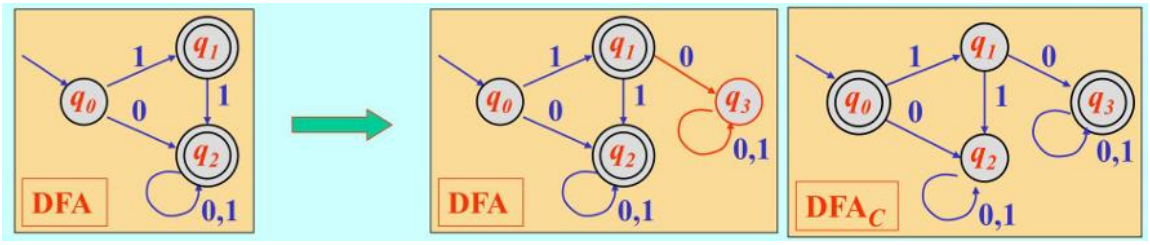
Let's start saying that 2 states p and q of DFA are **m-equivalent** ($p \equiv_m q$) if they are non-distinguishable for all the strings $w \in \Sigma^*$ with $|w| \leq m$:

- $p \equiv_0 q$ if $p \in F; q \in F$ or $p \notin F; q \notin F$
- if $p \equiv_m q$ and for any $a \in \Sigma$, $\delta(p,a) \equiv_m \delta(q,a)$ then $p \equiv_{m+1} q$

The **equivalent states** can be determined by partitioning the set Q in classes of m-equivalent states (for $m = 0, 1, \ldots, ||Q|| - 2$):

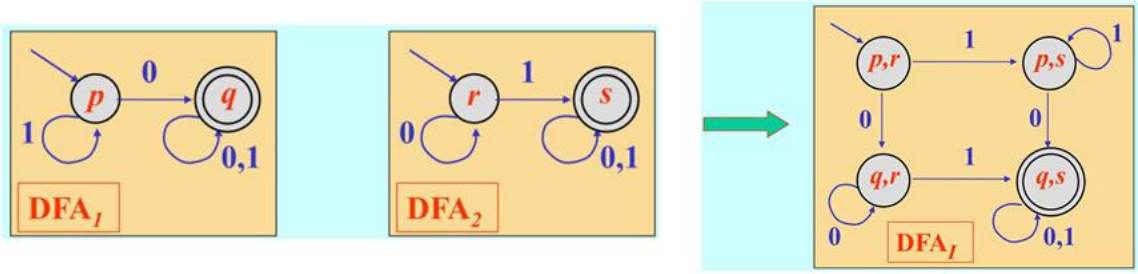
COMPLEMENT & INTERSECTION of a regular language:

- **COMPLEMENT of a regular language is a regular language**: let $DFA = (Q, \Sigma, \delta, q_0, F)$ be a completely specified deterministic FA (there's transition on every symbol of Σ from every state
  The *complement automaton* $DFA_C = (Q, \Sigma, \delta, q_0, Q - F)$ accepts the language $\boldsymbol{L(DFA_C) = \Sigma^* - L(DFA) = \neg L(DFA)}$ [in the example, from q_1 there wasn't the 0):

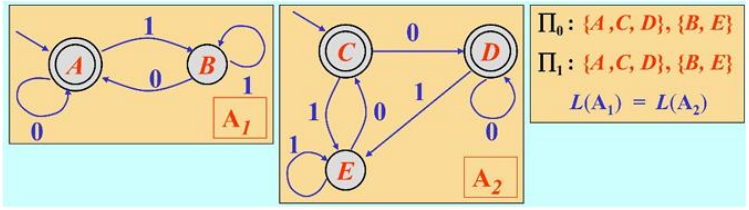


⚠ Before creating the complement, we need to transform the initial DFA to a completely specified DFA!

- **INTERSECTION of 2 regular languages is still a regular language**: let $DFA_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $DFA_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$. The automaton $DFA_I = (Q_1 \times Q_2, \Sigma, \delta, (q_{01}, q_{02}), F_1 \times F_2)$, where $\delta((p,q),a) = (\delta_1(p,a), \delta_2(q,a))$, accepts the language $\boldsymbol{L(DFA_I) = L(DFA_1) \cap L(DFA_2)}$. DFA_I is called product DFA where both DFAs receive the same input symbols and DFA_I reaches its final state when both DFAs reach a final state.



EQUIVALENCE of regular language: to test if 2 regular languages are **equivalent** we just need to verify that the 2 start states are equivalent ($q_{01} \equiv q_{02}$):



⚠ **APPLICATIONS of regular languages** (field of application):

- Pattern matching (finding occurrences of words, phrases, patterns in a text)
- Constructing lexical analyzers [scanners] (in the first stage of the compilation break the source text into lexical elements)
- IDS (intrusion detection systems)
- Functions as `grep`, `egrep`, `matches`, `find` (FA to match strings [especially NFA])

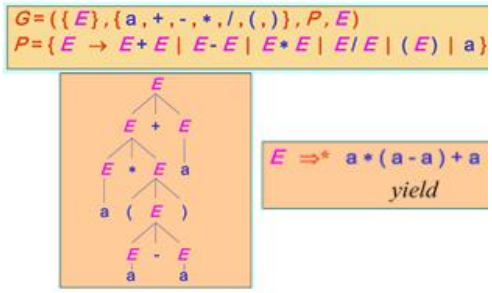
⚠ There are 2 different methods to solve the membership problem using NFAs:

- ✓ **NFA execution with backtracking** (tries all the possibilities)
- ✓ **NFA simulation** (DFA built on the fly considering only states/transition required by the actual input [if the first input symbol is 0, we don't consider the next state for the symbol 1) [used by `egrep`]

3) CONTEXT-FREE LANGUAGES (CFL)

PARSE TREE = for a context-free grammar $G = (N, T, P, S)$ is a tree where:

- S (start) = root
- N = interior node
- Symbol in $N \cup T \cup \{\varepsilon\}$ = leaf
- A = interior node that has children (from left to right) labeled by $X_1, X_2, \dots, X_k$ only if $A \to X_1 X_2 \dots X_k$ is a production in P



The **yield** of a parse tree is a string obtained by concatenating (from left to right) the labels of the leaves. Then we have:

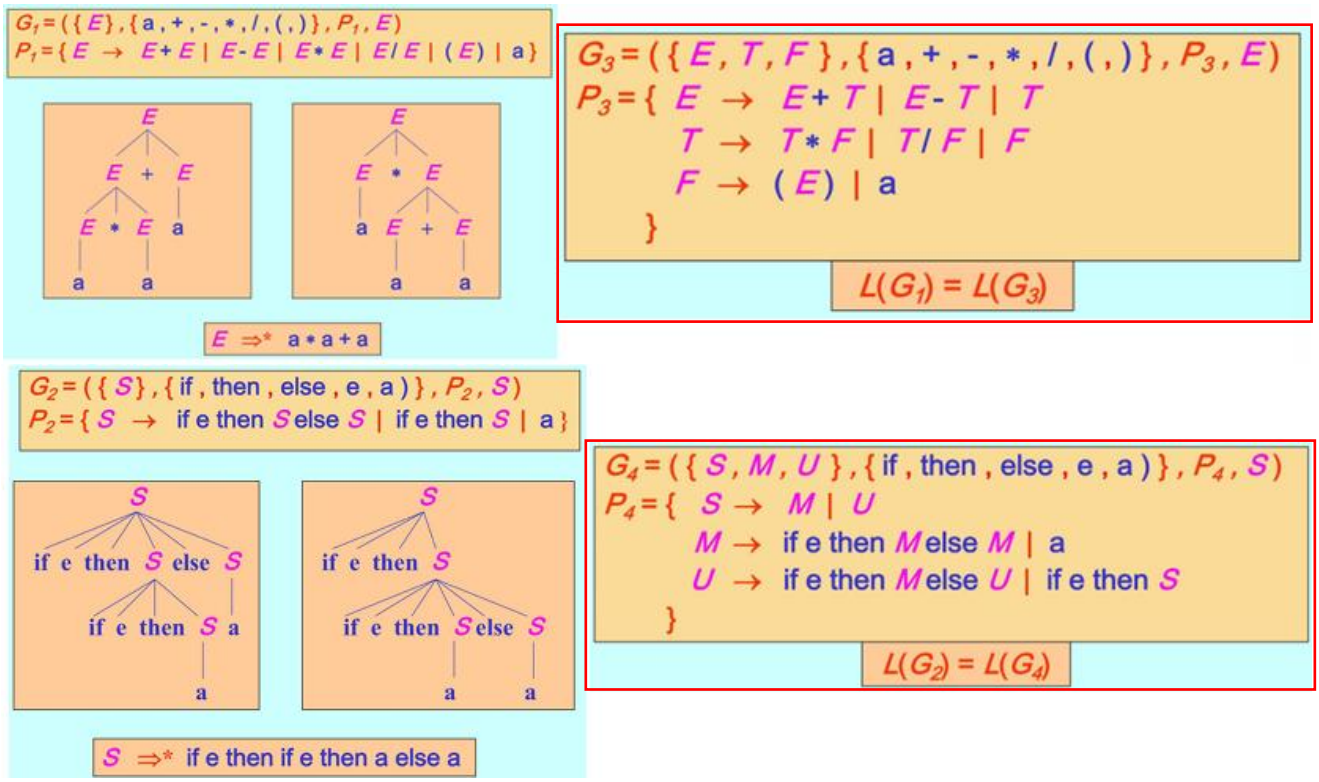
- **leftmost derivation** → the leftmost non-terminal symbol is replaced at each derivation step
- **rightmost derivation** → the rightmost non-terminal symbol is replaced at each derivation step

$E \Rightarrow \underline{E}+E \Rightarrow \underline{E}*E+E \Rightarrow a*\underline{E}+E \Rightarrow a*(\underline{E})+E \Rightarrow a*(\underline{E}-E)+E \Rightarrow a*(a-\underline{E})+E \Rightarrow a*(a-a)+\underline{E} \Rightarrow a*(a-a)+a$

$E \Rightarrow E+\underline{E} \Rightarrow \underline{E}+a \Rightarrow E*\underline{E}+a \Rightarrow E*(\underline{E})+a \Rightarrow E*(E-\underline{E})+a \Rightarrow E*(\underline{E}-a)+a \Rightarrow E*(a-\underline{a})+a \Rightarrow a*(a-a)+a$

⚠ Every string in a CFL has at least 1 parse tree, and each parse tree has just 1 leftmost derivation and just 1 rightmost derivation!

A CFG (Context-Free Grammar) is **AMBIGUOUS** if there's >= 1 string in its language with 2 different parse trees.
A CFL (Context-Free Language) is **inherently ambiguous** if all its grammars are ambiguous.
Example: we can have 2 different parse trees for the grammar (G_1) to have a similar grammar [1st picture]; **to avoid ambiguity we could add some non-terminal symbols and change the transitions** [**2nd picture**]:



⚠ It's impossible to find an algorithm that can always understand if a CFG is ambiguous and produce the corresponding non-ambiguous CFG, but we can do it only manually!

A symbol X is **USEFUL** for a *CFG = (N, T, P, S)* if there is some derivation $S \Rightarrow^* \alpha \boldsymbol{X} \beta \Rightarrow^* w \in \boldsymbol{T}^*$; instead, if X doesn't appear in any intermediate derivation that goes to a string concatenation of terminal symbols it's **USELESS**.
A useful symbol X:

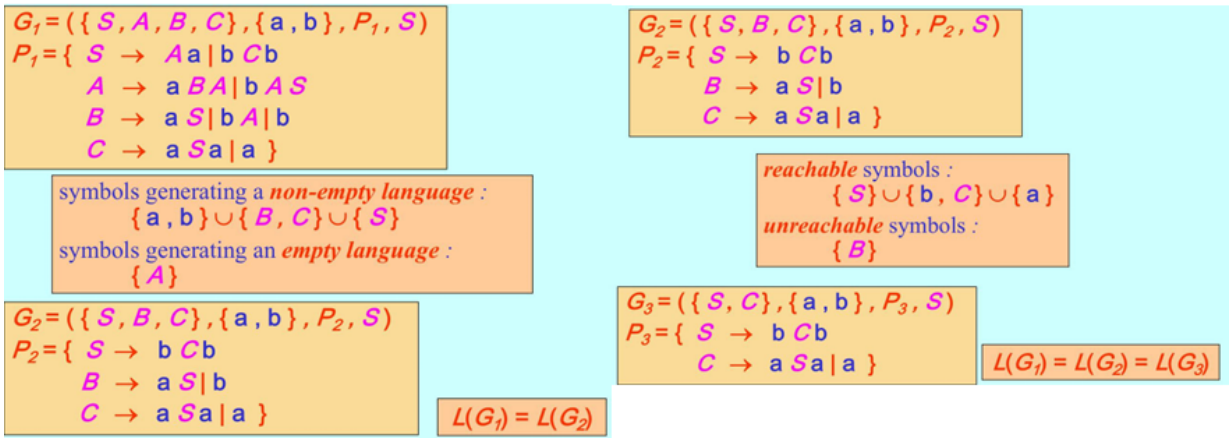
- generates a non-empty language ($X \Rightarrow^* x \in T^*$)
- is reachable ($S \Rightarrow^* \alpha X \beta$)

So, **eliminating useless symbols** from a grammar won't change the generated language, eliminating just:

- symbols generating an empty language
- unreachable symbols

To **find useful symbols in a CFG**, we need to:

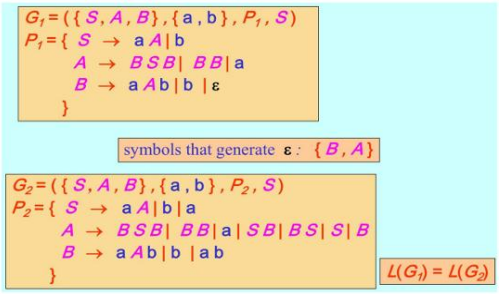
- find symbols **generating non-empty languages**:
  - a terminal symbol always generates a non-empty language
  - if $A \rightarrow \alpha$ and all symbols in α generate a non-empty language, A generates a non-empty language
- find **reachable** symbols:
  - the start symbol S is always reachable
  - if $A \rightarrow \alpha$ and A is reachable, all symbols in α are reachable



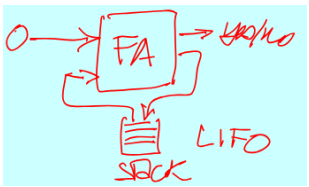
ε-productions in CFG: we said that only type-0 grammars can have ε-productions, but the languages generated by CFGs that contains ε-productions are **extended CFLs**. A CFG G_1 with ε-productions can be transformed into an equivalent CFG G_2 without ε-productions [$\boldsymbol{L(G_2) = L(G_1) - \{\varepsilon\}}$]. After this, we see that if $A \rightarrow X_1 \dots X_n$ is in P_1 and $X_i \Rightarrow^* \varepsilon$, then P_2 contains both:

- $A \rightarrow X_1 \dots X_i \dots X_n$
- $A \rightarrow X_1 \dots X_i X_{i+1} \dots X_n$ (because that symbols could be missing because we have removed ε-productions)

Example: since B generates ε, we apply the rule above → so for all the productions related to B, we need to consider also the possibility that the B is missing in the right side of the production



PDA (**PushDown Automata**) = a 7-tuple $\boldsymbol{P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)}$ that provide a way to solve the membership problem for CFLs by storing an unbounded amount of information in a stack (only **push** and **pop**) [LIFO]:



- $\boldsymbol{Q}$ = finite (non-empty) set of states
- $\boldsymbol{\Sigma}$ = alphabet of input symbols
- $\boldsymbol{\Gamma}$ = alphabet of stack symbols (can be the same as input symbols)
- $\boldsymbol{\delta}$ = transition function = $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \vartheta(\{(p, \gamma) \mid p \in Q; \gamma \in \Gamma^*\})$ → takes as **input = current state + certain symbol + top of the stack**; **returns** a set of **pairs** $(\boldsymbol{p, \gamma})$ that represents the different possibilities at each step (non deterministic)
  At each step there's a pop of 1 symbol from the stack and a push of 0 or more symbols to the stack
- $\boldsymbol{q_0}$ = start state ($q_0 \in Q$)
- $\boldsymbol{Z_0}$ = start stack symbol ($Z_0 \in \Gamma$) [at beginning it's the only symbol in the stack]
- $\boldsymbol{F}$ = set of final states ($F \subseteq Q$)

Transitions of PDA:

- $\boldsymbol{\delta(q, a, X) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots (p_m, \gamma_m)\}}$ (from state q, with a in input and X on top of the stack) consumes a from input string and goes to a state p_i and replaces X with γ_i (the first symbol of γ_i goes on top of stack)
- $\boldsymbol{\delta(q, \varepsilon, X) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots (p_m, \gamma_m)\}}$ (from state q, with X on top of the stack) doesn't consume any input symbol (ε), goes to a state p_i and replaces X with γ_i (the first symbol of γ_i goes on top of stack)

The Instantaneous Configuration of a PDA $\boldsymbol{(q, w, y)}$ represents the **current global state**:

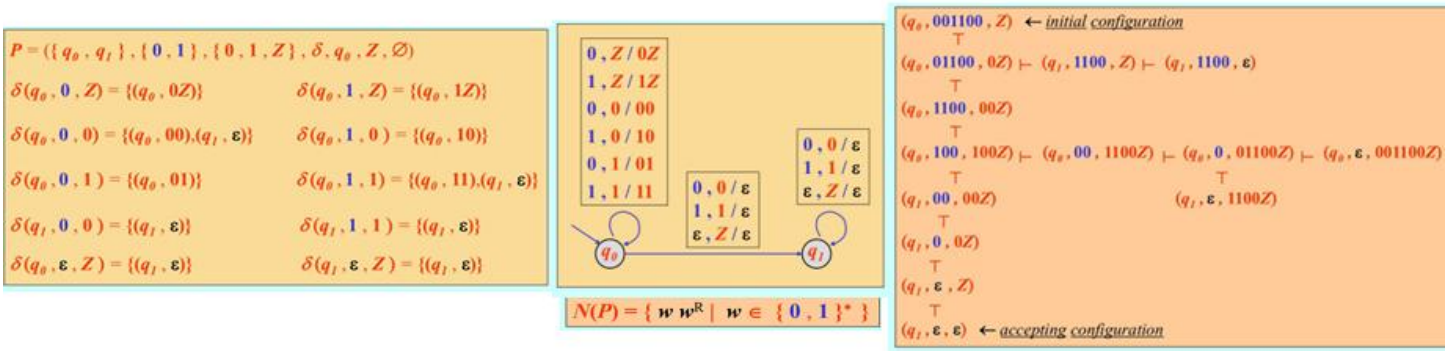
- $\boldsymbol{q}$ = current state
- $\boldsymbol{w}$ = remaining input string (has to be consumed yet)
- $\boldsymbol{\gamma}$ = current stack contents

If $\boldsymbol{\delta(q, a, X) = \{\ldots, (p, \alpha), \ldots\}}$ then $\boldsymbol{(q, aw, X\beta) \vdash (p, w, \alpha\beta)}$ [$\vdash$ = derivability] → it means that, starting from the left configuration, we can go to the right configuration applying the transition δ.

There are 2 possible **language acceptance**:

- Language accepted by **final state**:
  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \rightarrow L(P) = \{w \mid \boldsymbol{w} \in \boldsymbol{\Sigma}^*; (q_0, \boldsymbol{w}, Z_0) \vdash^* (q, \boldsymbol{\varepsilon}, \boldsymbol{\alpha}); \boldsymbol{q} \in \boldsymbol{F}\}$ → this means that a string is accepted by the PDA if, starting from the starting configuration (left), it's possible to reach the final configuration (right) consuming the full string (remains an empty string at the end) ending on a final state
- Language accepted by **empty stack** (*accepting configuration*):
  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \emptyset) \rightarrow L(P) = \{w \mid \boldsymbol{w} \in \boldsymbol{\Sigma}^*; (q_0, \boldsymbol{w}, Z_0) \vdash^* (q, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}); \boldsymbol{q} \in \boldsymbol{Q}\}$ → this means that, if the stack becomes empty, the string is automatically accepted

Example:



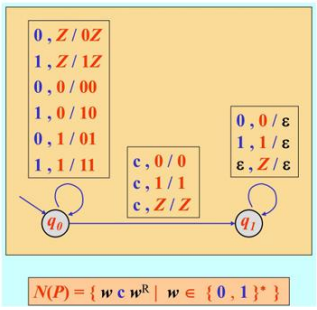
We can see a PDA represented by enumeration of the transitions and by a graph, where the **max n° of transitions** could be 2 states * 2 symbols (+1 because of ε) * 3 stack symbols = 18 different possible transitions.
In the graph we can see that the operations starting from q_0 are all push operations, while the others are all pop operations. What we are doing is building a specular string (3rd image represents all the possible configurations with their sequence when testing the input string "001100").

A PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is **deterministic** (**DPDA**) if:

- $\delta(q, a, X)$ has at most 1 member for any $q \in Q, a \in (\Sigma \cup \{\varepsilon\}), X \in \Gamma$
- if $\delta(q, a, X) \neq \emptyset$ for some $a \in \Sigma$, then $\delta(q, \varepsilon, X) = \emptyset$ → means that, if there's a transition for a certain symbol starting from a certain state, it's impossible to have also a transition starting from the same state with ε as input symbol

The languages accepted by DPDA are properly included ($\subset$) in the languages accepted by PDA (so DPDA can represent less languages than PDA).

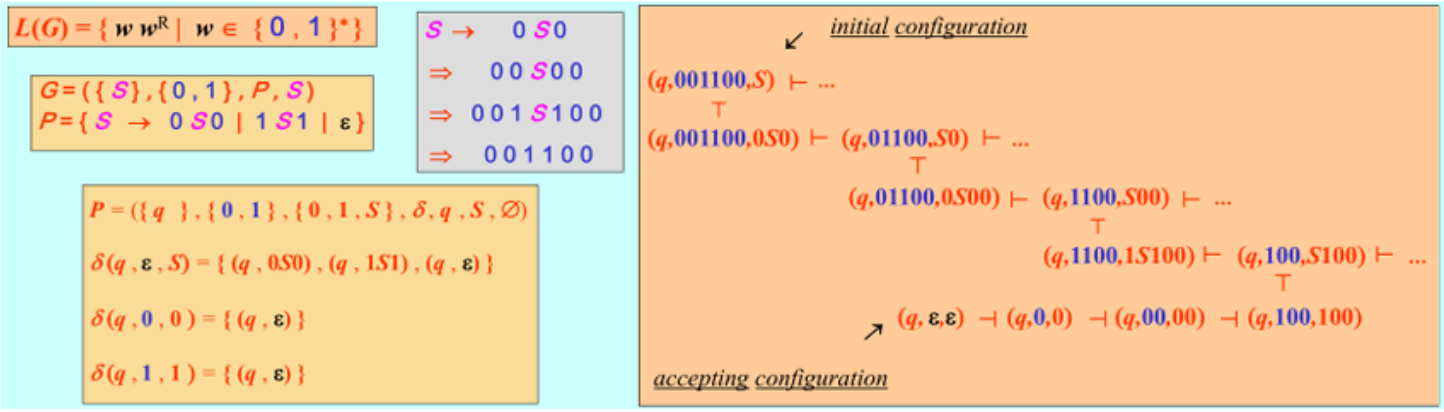
Example: the language $N(P) = \{ww^R \mid w \in (0,1)^*\}$ is not accepted by DPDA; to be representable we need to add a $\boldsymbol{c}$ symbol in the middle, to know when I finished building the 1st half of the string and I can start building the 2nd half



PDA accepted languages ≡ CFL → let $G = (N, T, P, S)$ be a CFG and let's construct a $PDA = (\{q\}, T, \Gamma, \delta, q, S, \emptyset)$:

- $\{\boldsymbol{q}\}$ → there's only 1 state, so we rely only on the stack
- $\boldsymbol{T}$ → input alphabet is the set of terminal symbols of the grammar
- $\boldsymbol{\Gamma} = N \cup T$
- $\boldsymbol{\delta}$ is composed of:
  - $\forall A$: $\delta(q, \varepsilon, A) = (q, \alpha) \rightarrow \alpha \in P$ → when there isn't input symbol, it's possible to substitute A with the string α in the stack
  - $\forall a \in T$: $\delta(q, a, a) = (q, \varepsilon)$ → when on the top of the stack there is a terminal symbol that matches the input symbol, pop of the top of the stack

  With these 2 rules we are doing a leftmost derivation
- $\boldsymbol{S}$ → the initial stack content is the start symbol of the grammar
- $\emptyset$ → accepts by empty stack → the PDA accepts $L(G)$ by empty stack, making a sequence of transitions corresponding to leftmost derivation



Let's conclude with some **observations about CFL**:

- **Properties** → let $G_1 = (Q_1, \Sigma, P_1, S_1)$ and $G_2 = (Q_2, \Sigma, P_2, S_2)$; CFLs are **closed** under:
  - Union: $G_3 = (Q_1 \cup Q_2 \cup \{S_3\}, \Sigma, P_3, S_3)$ with $P_3 = P_1 \cup P_2 \cup \{S_3 \to \boldsymbol{S_1 | S_2}\}$
  - Concatenation: $G_3 = (Q_1 \cup Q_2 \cup \{S_3\}, \Sigma, P_3, S_3)$ with $P_3 = P_1 \cup P_2 \cup \{S_3 \to \boldsymbol{S_1 S_2}\}$
  - Kleene closure: $G_3 = (Q_1 \cup \{S_3\}, \Sigma, P_3, S_3)$ with $P_3 = P_1 \cup \{S_3 \to S_1 S_3 | \varepsilon\}$

  CFLs are **not closed** (result may not be a CFL) under:
  - Complement
  - Intersection

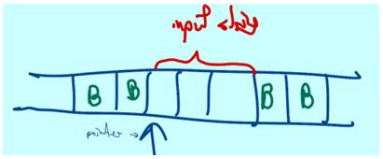
  In a CFL we can decide **membership** of a string with length n using different algos with complexity $O(n^3)$; so, we can't use these high-complexity algorithms with programming languages → we will take a subset of CFLs (which are the deterministic CFLs), so we can parse them more efficiently
- **Deterministic CFL** → are the languages accepted by DPDAs and are only closed under Complement (not closed under union, intersection, concatenation, Kleene closure). Here we can solve the membership problem linearly to the length of the string n
- Applications of CFL:
  - representation of programming languages (C, Java…)
  - construction of parsers (syntax analyzers)
  - description of the structure and semantic contents of documents by means of markup languages (XML, JSON…)

4) TURING MACHINES (TM)

Phrase structure and context sensitive languages are less important, but can be used to study what is machine-computable and to represent natural languages (but with an inefficient parsing). **Phrase structure** (also called **Type0** grammars) can be represented with **Turing machines** to solve the membership problem efficiently.

A **TURING MACHINE** is a 6-tuple $\boldsymbol{M = (Q, \Sigma, \Gamma, \delta, q_0, F)}$ where:

- $\boldsymbol{Q}$ = finite (non-empty) set of states
- $\boldsymbol{\Sigma}$ = alphabet of input symbols ($B \notin \Sigma$), which means that there's a symbol B which represent a blank memory cell and it's not included in the alphabet
- $\boldsymbol{\Gamma}$ = alphabet of tape symbols ($B \in \Gamma$; $\Sigma \subset \Gamma$):
  - The tape extends infinitely to the left and the right
  - The tape initially holds the input string, preceded and followed by an infinite number of B symbols

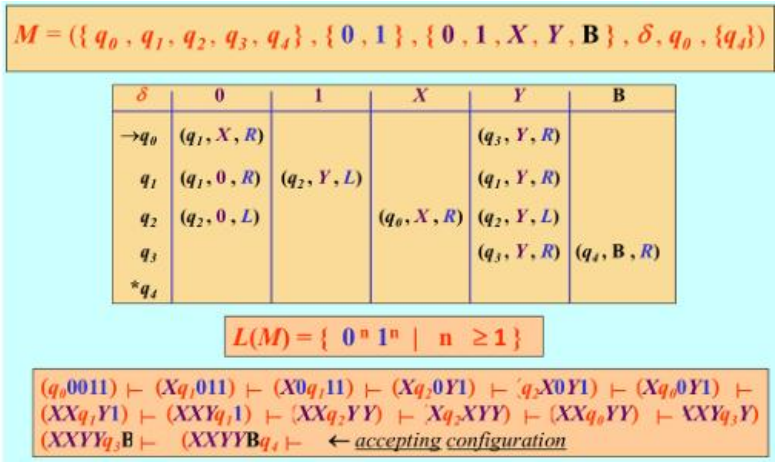


- $\boldsymbol{\delta}$ = transition function = $\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$; given a current state and the tape symbol, δ returns the new state, the symbol that will replace the current symbol on the tape and the position where to move the pointer (left or right) [there could be also no next state, in this case the machine stops + error]
  $\delta(q, X) = (p, Y, L)$: from state q, having X as the current tape symbol goes to state p and replaces X with Y, moving the tape head 1 position left
- $\boldsymbol{q_0}$ = start state
- $\boldsymbol{F}$ = set of final states

The instantaneous configuration of a TM can be represented as $(X_1 \dots X_{i-1} q X_i \dots X_n)$:
- q = current state
- $X_1 \dots X_{i-1} X_i \dots X_n$ = current string on tape
- X_i = current state symbol

If $\delta(q, X_i) = (p, Y, L)$ then $(X_1 \dots X_{i-1} q X_i \dots X_n) \vdash (X_1 \dots X_{i-2} q X_{i-1} Y X_{i+1} \dots X_n)$:
- if $i = 1$, then $(q X_1 \dots X_n) \vdash (pBY X_2 \dots X_n)$
- if $i = n$ and $Y = B$ (blank), then $(X_1 \dots X_{n-1} q X_n) \vdash (X_1 \dots X_{n-2} p X_{n-1})$

The language accepted by a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ is $L(M) = \{w |\ w \in \Sigma^*; (q_0 w) \vdash^* (\alpha q \beta); q \in F\}$ which means that starting from a tape where there's only the input string, the string is accepted if the TM reaches a final state, despite what it's on the tape



RECURSIVE SETS → the languages accepted by TMs are called ***recursively enumerable sets*** and are equivalent to the **Type 0** languages (***phrase structure***). A TM halts when it reaches an accepting state, but in some cases the TM enters a loop; for this reason the membership of a string in a *recursively enumerable set* is **undecidable**.
The subset of TMs that **always halts** are called recursive sets and in this case the membership is decidable

DECIDABILITY & COMPUTABILITY: every computational problem can be reduced to a membership problem:
- **Yes/No questions** → we consider the set of all the possible inputs, dividing it in 2 groups (inputs for yes/ inputs for no), and we consider them as 2 languages for which we need to solve the membership problem
- **Other questions** → es. considering the √ of a number, we consider the language concatenation of the input and the output (`input.output`) where the $output = \sqrt{input}$.
  So, given an `input.output` string, we can solve the membership problem by understanding if output is correct or not (simpler than actually computing the output)

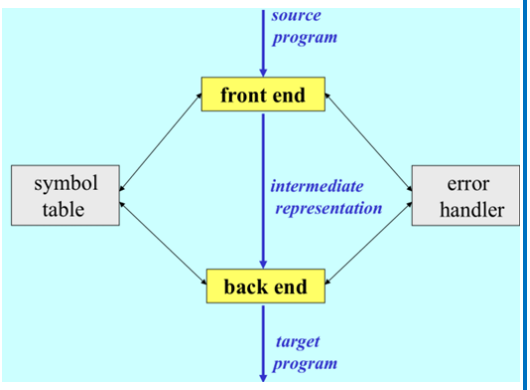
In this way we have an easier membership problem (but both this and the classic membership problem aren't decidable, so we can use this) → we reduce the complexity of understanding if a problem is computable or not

The TM defines the **most general model of computation** (every computable function can be computed by TM).
The TM can be used to classify languages/problems/functions:
- **Not recursively enumerable** = can't be represented by any TM
- **Recursively enumerable/undecidable/uncomputable** = represented by a TM that not always halts
- **Recursive/decidable/computable** = represented by a TM that always halts (algorithm)

⚠ There are languages so complex that can't be represented by a generative grammar (Type 0): there are problems for which it is impossible to solve through an algorithm

COMPILERS

0) COMPILER STRUCTURE

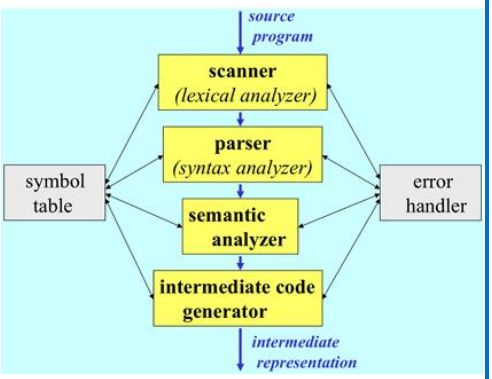


COMPILER = read a source program and generate a target executable; 2 parts:

- **Front-end** → depends on the source language and is not aware of the target language; transforms the source program into an **intermediate representation**
- **Back-end** → depends on the target language and is not aware of the source language, it transforms the intermediate representation into the **target program**

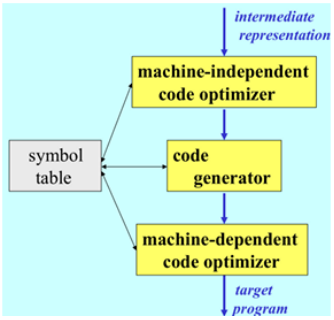
2 components shared in front-end and back-end:

- **Symbol table** = contains identifier + value + type of the symbols used in source program
- **Error handler** = contains error handling functions (called when a error occurs in any stage of compilation)



The **FRONT-END** is composed of:

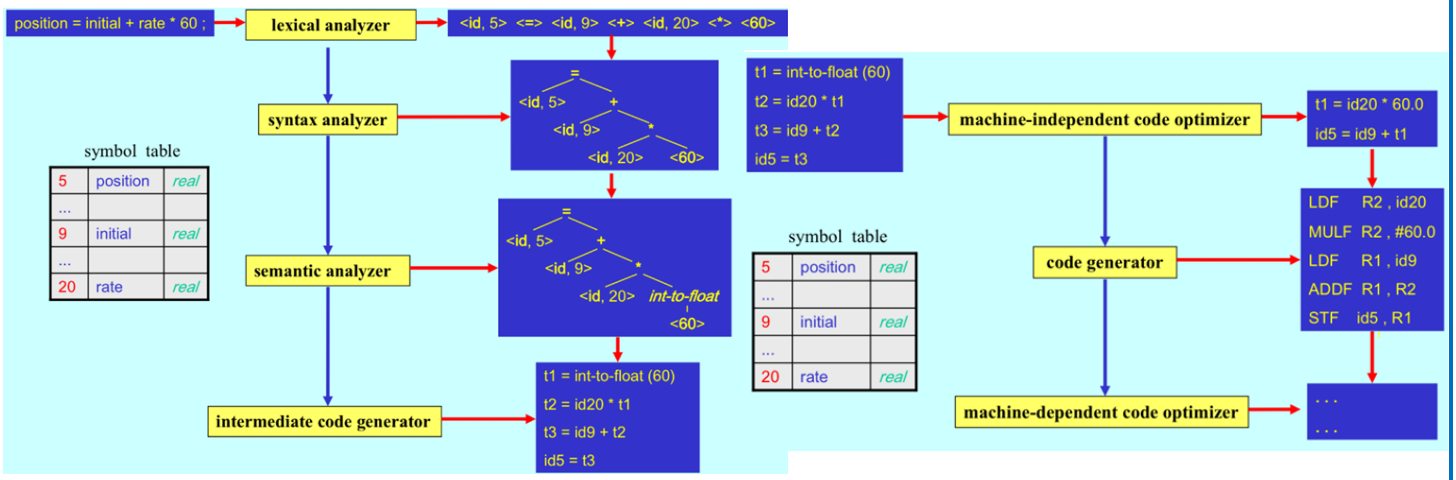
- **Scanner** (**Lexical** Analyzer) → divides the source in tokens (regular grammar)
- **Parser** (**Syntax** Analyzer) → recognizes the syntax constructs of the programming language and works in a pipeline with the scanner. Treats the tokens as symbols of a CFG; if the syntax doesn't correspond to the language, it gives a syntax error
- **Semantic** Analyzer → checks the source program for semantic errors
- **Intermediate Code** Generator → generates the intermediate code



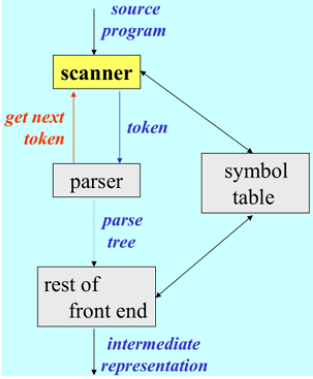
The **BACK-END** is composed of:

- Machine-**independent** Code Optimizer → optimizes the intermediate code
- Code **generator** → translate intermediate representation to target language
- Machine-**dependent** Code Optimizer

Let's an example of use of all these parts:



1) LEXICAL ANALYSIS (SCANNER)



It's **dividing the source program in tokens**; the execution is a pipeline where Parser continuously requests a new token and Scanner provides it. **Scanner** has a buffer with the next words to be scanned (returned as tokens); **Parser** doesn't need access to the symbol table because his role is only to build the parse tree.

We define:

- **TOKEN** = terminal symbol in the grammar for the source language
- **LEXEME** = string of chars in the source program treated as a lexical unit
- **PATTERN** = representation of the set of lexemes associated with a token

TOKEN	PATTERN	SAMPLE LEXEMES
const	the *const* keyword	const
relop	{ < , > , == , <= , >= , != }	<= > ==
id	letter (letter | digit)*	pi counter1 main
num	any numeric constant	3.14 25 6.02E23

After receiving a "`get next token`" command from the Parser, Scanner reads input chars until it can identify a token. It simplifies the job of Parser because it discards irrelevant details (spaces, comments, tabs…) and the Parser is only concerned about tokens (not lexemes).

REGULAR DEFINITION = sequence of definitions $d_1 \to r_1; d_2 \to r_2; \dots; d_n \to r_n$ where:

- d_i = distinct name (TOKEN)
- r_i = regexp over $\Sigma \cup \{d_1, \dots, d_{i-1}\}$ representing a pattern (this means that r_i can contain symbols of the alphabet or other tokens previously defined)

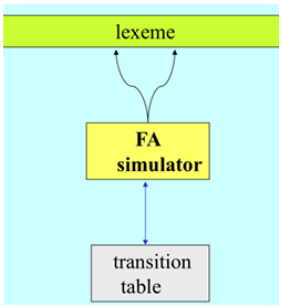
letter → A | B | ... | Z | a | b | ... | z
digit → 0 | 1 | ... | 9
id → letter (letter | digit)*
digits → digit digit*
optional_fraction → . digits | ε
optional_exponent → (E (+ | - | ε) digits) | ε
num → digits optional_fraction optional_exponent

⚠ Every definition generates a different DFA that are merged together; final states of the different DFAs are stored separately

Generation of a Scanner (Lexical Analyzer) [automated] → a **Scanner generator** transforms the specification of a scanner (regular definitions and actions to be executed when a matching occurs) into a program implementing a FA accepting the specified lexemes (as we saw with ***JFlex***)

Structure of a Scanner:

- **Input buffer** = linear buffer that contains the lexeme
- **FA simulator** = reads from the input buffer and returns the token (can recognize errors and activate error procedures); uses 2 pointers: 1st points the start char of the lexeme, 2nd reads each char. For every char, the transition table is queried and changes state according to response; when a final state is reached, the FA simulator generates the token using the position of the 2 pointers
- **Transition table** = given an input symbol, returns the next state of the FA



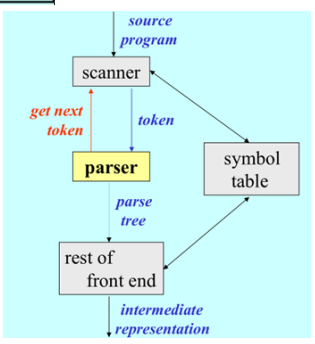
⚠ 2 different simulation (as we saw before) [we see these simplified, in reality longest match is added]:

- DFA simulation: starting from the start state s_0, at every cycle it updates the state until it becomes final
- NFA simulation: starting from the ε-closure of s_0, at every cycle it updates the set of states reached until 1 of the states in the set is final

2 different complexity (r = regexp; x = input string) →

AUTOMATON	SPACE	TIME
NFA	$O(\|r\|)$	$O(\|r\| * \|x\|)$
DFA	$O(2^{\|r\|})$	$O(\|x\|)$

2) SYNTAX ANALYSIS (PARSER)



When the **Parser** obtains a string of tokens from the **Scanner**:

- ❖ verifies that the string can be generated by the grammar for the source language (trying to build a parse tree)
- ❖ reports syntax errors, continuing to process the input to catch all syntax errors

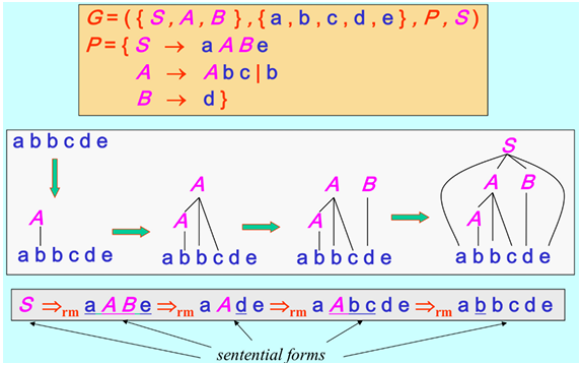
This procedure is the **SYNTAX ANALYSIS**; can be done:

- **BOTTOM-UP** → Parser builds the parse tree **from the bottom** (**leaves**) **to the top** (**root**); most powerful method but not possible for all the CFLs (it's used by ***Cup***)
- **TOP-DOWN** → Parser builds the parse tree **from the top** (**root**) **to the bottom** (**leaves**); easier but less powerful (we can build less languages)

Let's these **PARSING TECHNIQUES**:

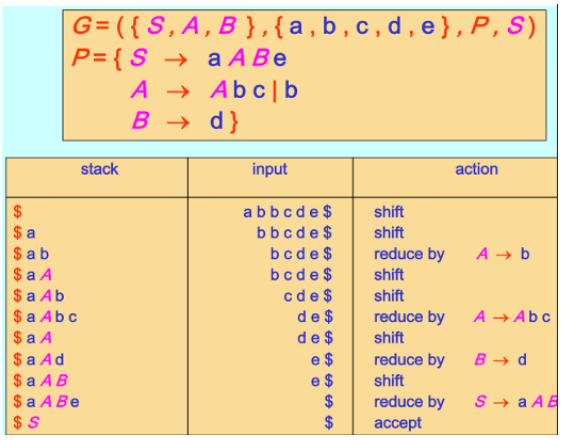
- **BOTTOM-UP**: construct a parse tree for an input string **from leaves to root** (**from left to right**), reducing the input string to S (start symbol of a grammar). At each reduction step, the right side of a production is replaced by its left side symbol (rightmost derivation in reverse). If we reach S, we can conclude that the string is part of the grammar and the parse tree is already built

  ⚠ If a string $\alpha\beta w$ can be produced by a rightmost derivation ($S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$), then $A \to \beta$ is a **HANDLE** of $\alpha\beta w$ ($w \in T^*$ [terminal] because $A \to \beta$ is the last applied rule)



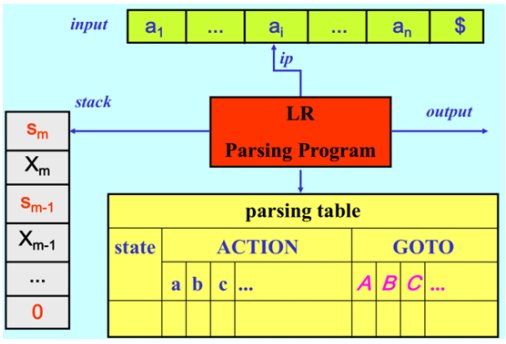
- **SHIFT-REDUCE**: bottom-up parsing can be implemented by a **shift-reduce Parser** that uses a **stack** (to hold grammar symbols) + an **input buffer** (to hold the string to be parsed). This parser **shifts** input symbol on the stack until the *rhs* (right hand side) of a handle β is on top of the stack; then **reduces** β to the left side of the handle until input is empty and stack contains the start symbol.
  If we have more handles that corresponds to the top of the stack or we need to choose between making a reduction or making a shift, the process becomes **non-deterministic** and this situation is called **conflict**

  Example (how the shift-reduce parsing is performed): the reduction is **possible only for handles**, in fact we don't reduce *aAb* to *aAA* (5th row) because the machine recognize that starting from S, it is not possible to arrive to *aAAcd* (as we saw in the bottom-up example) [so $b \to A$ is not a handle for *aAAcd*]. We consider that the machine can recognize if a reduction can be performed because it has full knowledge of the handles.



stack	input	action
$	a b b c d e $	shift
$ a	b b c d e $	shift
$ a b	b c d e $	reduce by A → b
$ a A	b c d e $	shift
$ a A b	c d e $	shift
$ a A b c	d e $	reduce by A → A b c
$ a A	d e $	shift
$ a A d	e $	reduce by B → d
$ a A B	e $	shift
$ a A B e	$	reduce by S → a A B e
$ S	$	accept

  The **actions** that can be performed by a shift-reduce Parser are:
  - **Shift** = the next input symbol is shifted on the top of stack
  - **Reduce** = the rhs of the handle must be located within the stack and must be decided with what non-terminal to replace the rhs of the handle
  - **Accept** = parsing completed
  - **Error** (syntax error)



- **LEFT-RIGHT**: is a strategy for **making parsing decisions** (we see that 4 actions are possible, but how to choose which action take?):
  - parsing program = keeps a *pointer* to the input buffer; can do *push* and *pop* on the stack and uses a parsing table to take the decisions
  - stack = initially contains the state 0 with no symbols and **for every symbol pushed also a state s_i is pushed**
  - parsing table = given current state + next symbol in input buffer, returns the **action** to be performed and the **next state**

  So, if the action is a:
  - ➢ shift → we **push** to stack the read symbol + next state
  - ➢ reduce → we **pop** 2 elements for each symbol of the *rhs* of the handle and than we **push** the *lhs* of the handle + the next state (read from the GOTO part of the parsing table)

  Example:

$G_0 = (\{E, T, F\}, \{\text{id}, +, *, (,)\}, P, E)$

$P = \{$
$E \rightarrow E + T \mid T$ (1, 2)
$T \rightarrow T * F \mid F$ (3, 4)
$F \rightarrow (E) \mid \text{id} \}$ (5, 6)

- si means *shift* and push i
- rj means *reduce* by production numbered j
- acc means *accept*
- *blank* means *error*

state	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

stack	input	action	
0	id + id * id $	s5	
0 id 5	+ id * id $	r6	$F \rightarrow$ id
0 F 3	+ id * id $	r4	$T \rightarrow F$
0 T 2	+ id * id $	r2	$E \rightarrow T$
0 E 1	+ id * id $	s6	
0 E 1 + 6	id * id $	s5	
0 E 1 + 6 id 5	* id $	r6	$F \rightarrow$ id
0 E 1 + 6 F 3	* id $	r4	$T \rightarrow F$
0 E 1 + 6 T 9	* id $	s7	
0 E 1 + 6 T 9 * 7	id $	s5	
0 E 1 + 6 T 9 * 7 id 5	$	r6	$F \rightarrow$ id
0 E 1 + 6 T 9 * 7 F 10	$	r3	$T \rightarrow T * F$
0 E 1 + 6 T 9	$	r1	$E \rightarrow E + T$
0 E 1	$	accept	

But **how to build Parsing Tables?** LR(0) parser is the simplest way for building these tables. An **LR(0) item** of a CFG grammar G is a production of G with a dot (**.**) at some position of the right side:

$A \rightarrow XYZ$ ⟹ $A \rightarrow \cdot XYZ$ | $A \rightarrow X \cdot YZ$ | $A \rightarrow XY \cdot Z$ | $A \rightarrow XYZ \cdot$

- **item** → indicates how much of a production has been recognized at a certain point in the parsing process (recognized part = at left of the dot)
- **dot** → indicates the current position of the parser; when the dot is at the end of an item, the item is completed (all the *rhs* of the production has been recognized)

A **viable prefix** can appear on the stack of a shift-reduce parser where the next action will be a reduce, to check that there won't be other shifted symbols before the reduction. So, we say that the item $\boldsymbol{A \rightarrow \beta_1 \cdot B_2}$ is "**valid**" **for a viable prefix** $\boldsymbol{\alpha\beta_1}$ if there is a derivation $S \Rightarrow^*_{rm} \alpha \boldsymbol{A} w \Rightarrow_{rm} \alpha \boldsymbol{\beta_1 \beta_2} w$ (if $A \rightarrow \beta$ is a "**valid complete**" **item for a viable prefix** $\alpha\beta$, then $S \Rightarrow^*_{rm} \alpha \boldsymbol{A} w \Rightarrow_{rm} \alpha \boldsymbol{\beta} w$ and $A \rightarrow \beta$ is a handle of $\alpha\beta w$)

Recognize viable prefixes using a DFA: sets of viable prefixes are **regular languages**, so it's possible to build a FA that represents them (and that can guide a parser in making parsing decisions).
The **valid LR(0) items** of a CFG grammar are the **states of an NFA recognizing viable prefixes**; so a DFA equivalent to this NFA will have states corresponding to **sets of LR(0) items** and transitions labeled by **symbols in viable prefixes**.
Let's see the pseudo-functions to build this DFA:

- `closure(I)` → finds the set of all LR(0) items that recognize the same viable prefix recognized by `I`. So, the closure of a given item $A \rightarrow \alpha X \beta$ is composed of all the items generated from all the productions that have X in the *lhs* and where the '**.**' is at beginning

- `goto(I,X)` → finds the set of LR(0) items that is reached from the set `I` with the symbol `X` (including their closure)

- `items(G)` → given a CFG grammar $G = (N, T, P, S)$, this function build the collection $C = \{I_0, I_1, \ldots, I_n\}$ of DFA states (`items` combines the `closure` and `goto` with the grammar `G`). Below example:

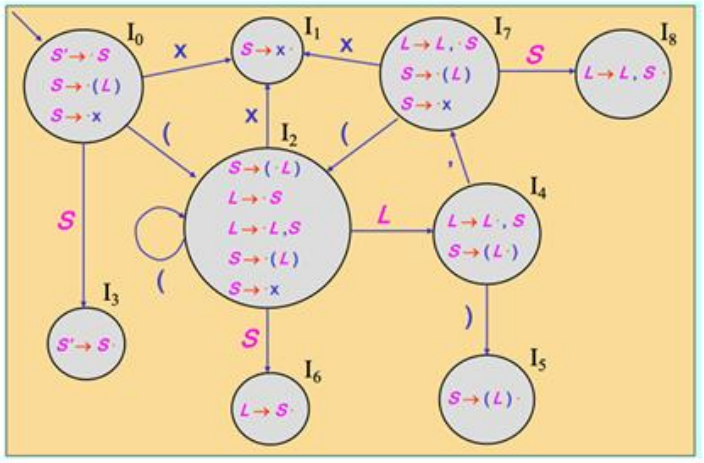
```
Items closure (Items I) ;
 repeat
  for (each item A → α · X β in I )
    for (each production X → γ )
      I = I ∪ { X → · γ } ;
 until ( I does not change ) ;
 return I ;
```

```
Items goto (Items I, Symbol X) ;
 J = ∅ ;
 for (each item A → α · X β in I )
   J = J ∪ { A → α X · β } ;
 return closure (J) ;
```

```
ItemsCollection items (CFG G) ;
 G' = (N ∪ {S'}, T, P ∪ {S' → S}, S') ;
 C = { closure ({S' → · S }) };
 repeat
  for ( each set I in C )
    for ( each item A → α · X β in I )
      C = C ∪ { goto (I, X)} ;
 until ( C does not change ) ;
 return C ;
```

$G_1' = (\{S', S, L\}, \{x, (,), ,\}, P', S')$

$P' = \{$
$S' \rightarrow S$ (0)
$S \rightarrow (L) \mid x$ (1, 2)
$L \rightarrow S \mid L, S \}$ (3, 4)



❖ `lr0Table(G)` → constructs the LR(0) parsing table for the CFG `G`. For every item in `items(G)`:

- if next symbol is a terminal, **shift** + next state will be the `goto()` of the shifted symbol
- if item is complete but the *lhs* is not *S'*, **reduce** for the production related to this item (+ we do it for all the possible next input symbols *a*)
- if item is complete + has *S'* as *lhs*, we **accept** only if the next symbol is **$**
- if transition is labeled by a non-terminal symbol, it will populate the **GOTO part of the table**

⚠ All the other combinations are considered as errors and the string is not accepted

```
void lr0Table (CFG G);
let {I0 , I1 , … , In} be the result of items (G) ;
for ( i = 0 to n )
  if ( A → α · a β is in Ii and a∈T and goto (Ii , a) = Ij )
    set ACTION[i , a] to shift j ;
  if ( A → α · is in Ii and A ≠ S' )
    set ACTION[i , a] to reduce A → α for all a in T∪{$} ;
  if ( S' → S · is in Ii )
    set ACTION[i , $] to accept ;
  if ( goto (Ii , X) = Ij and X∈N ) set GOTO[i , X] to j ;
```

Let's now see the parsing table of the previous grammar example:

state	ACTION					GOTO	
	(	)	x	,	$	S	L
0	s2		s1			3	
1	r2	r2	r2	r2	r2		
2	s2		s1			6	4
3					acc		
4		s5		s7			
5	r1	r1	r1	r1	r1		
6	r3	r3	r3	r3	r3		
7	s2		s1			8	
8	r4	r4	r4	r4	r4		

stack	input	action	
0	(x,(x),x)$	s2	
0(2	x,(x),x)$	s1	
0(2x1	,(x),x)$	r2	S → x
0(2S6	,(x),x)$	r3	L → S
0(2L4	,(x),x)$	s7	
0(2L4,7	(x),x)$	s2	
0(2L4,7(2	x),x)$	s1	
0(2L4,7(2x1	),x)$	r2	S → x
0(2L4,7(2S6	),x)$	r3	L → S
0(2L4,7(2L4	),x)$	s5	
0(2L4,7(2L4)5	,x)$	r1	S → (L)
0(2L4,7S8	,x)$	r4	L → L,S
0(2L4	,x)$	s7	
0(2L4,7	x)$	s1	
0(2L4,7x1	)$	r2	S → x
0(2L4,7S8	)$	r4	L → L,S
0(2L4	)$	s5	
0(2L4)5	$	r1	S → (L)
0S3	$	accept	

The parsing table entries are defined in multiple ways, so it's possible to have **parsing action conflicts**:

- shift/reduce → some entry in the action table contains both a shift and a reduce actions
- reduce/reduce → some entry in the action table contains more reduce actions

⚠ shift/shift is <u>impossible</u> because we would have 2 symbols for a single transition

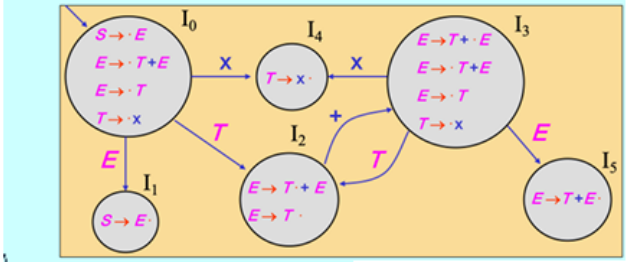
Example (build a **LR(0) parsing table for a grammar**):

$G_2 = (\{S, E, T\}, \{x, +\}, P, S)$

$P = \{ \; S \to E \quad (0)$

$E \to T + E \mid T \quad (1, 2)$

$T \to x \; \} \quad (3)$



state	ACTION			GOTO	
	x	+	$	E	T
0	s4			1	2
1			acc		
2	r2	s3,r2	r2		
3	s4			5	2
4	r3	r3	r3		
5	r1	r1	r1		

shift / reduce conflict

A **grammar** *G* is **LR(0)** if the action table generated by the function `lr0Table(G)` doesn't have conflicts and it's non-ambiguous. If any set of LR(0) items generated by function `items(G)` contains a complete item (originating a reduce action), then:

- no other item in the set is *complete* (avoid reduce/reduce)
- no other item in the set has a *terminal* symbol *immediately at the right of the dot* (avoid shift/reduce)

Summing, a **LR(0) Parser**:

- scans input from left to right
- build a rightmost derivation in reverse
- uses 0 lookahead input symbols in making parsing decisions

→ the class of languages that can be parsed with LR(0) Parsers is a subset of the deterministic CFL

LR(k) parsers [**SLR**]: using LR(0), when we add a reduction to the parsing table, we add it for every possible terminal symbol. If the **`lr0Table(G)`** function knows which input symbols after the dot are valid, can set the reduce action for them only (avoiding several potential conflicts).
For a CFG grammar, given a non-terminal symbol X and a string γ of terminal and non-terminal symbols:

- **`nullable`**(X) is true if X can derive the empty string
- **`nullable`**(γ) is true if each symbol in γ is nullable
- **`FIRST`**(γ) is the set of terminals that can begin strings derived from γ; if X can derive the empty string, then we consider also the first of γ
- **`FOLLOW`**(X) is the set of terminals that can immediately follow X

if (not nullable(X))
then FIRST(X γ) = FIRST(X)
else FIRST(X γ) = FIRST(X) ∪ FIRST(γ)

initialize all *FIRST* and *FOLLOW* to ∅ and all *nullable* to **false** ;
set *FOLLOW(S)* = {$} ;
for (each terminal symbol *z*) set *FIRST(z)* = {*z*} ;
repeat
  for (each production $X \to Y_1 Y_2 \dots Y_k$)
    if ($X \to \varepsilon$ or $Y_1 \dots Y_k$ are all nullable)
      set *nullable(X)* = *true* ;
    for (each i from 1 to k and each j from i+1 to k)
      if (i = 1 or $Y_1 \dots Y_{i-1}$ are all nullable)
        set *FIRST(X)* = *FIRST(X)* ∪ *FIRST(Y_i)* ;
      if (j = i+1 or $Y_{i+1} \dots Y_{j-1}$ are all nullable)
        set *FOLLOW(Y_i)* = *FOLLOW(Y_i)* ∪ *FIRST(Y_j)* ;
      if (i = k or $Y_{i+1} \dots Y_k$ are all nullable)
        set *FOLLOW(Y_i)* = *FOLLOW(Y_i)* ∪ *FOLLOW(X)* ;
until (all *FIRST* , *FOLLOW* and *nullable* do not change)

The algorithm in the image shows how to compute `nullable`, `FIRST` e `FOLLOW` (→): after initialization, for each symbol Y_i in the *rhs* of each production:

- if the production has ε as *rhs* or all the symbols in the *rhs* are `nullable`, **`nullable(X) = true`**
- if `i` is the 1st symbol of the *rhs* or all the symbols up to the analyzed symbol (excluded) are `nullable`, we add **`FIRST(`**Y_i**`)`** to **`FIRST(X)`**
- if `j` correspond to the following symbol of `i` or all the intermediate symbols toward `j` are `nullable`, we add **`FIRST(`**Y_j**`)`** to **`FOLLOW(`**Y_i**`)`**
- if `i` is the last symbol of the *rhs* or all the symbols following `i` are `nullable`, we add **`FOLLOW(X)`** to **`FOLLOW(`**Y_i**`)`** → so we need to consider also what is after the *lhs* of the production when we reach the last non `nullable` symbol

We repeat this until there is no change

$G_2 = (\{S, E, T\}, \{x, +\}, P, S\}$
$P = \{ \; S \to E \quad (0)$
$E \to T + E \mid T \quad (1,2)$
$T \to x \; \} \quad (3)$

	nullable	FIRST	FOLLOW
S	false		$
E	false		
T	false		

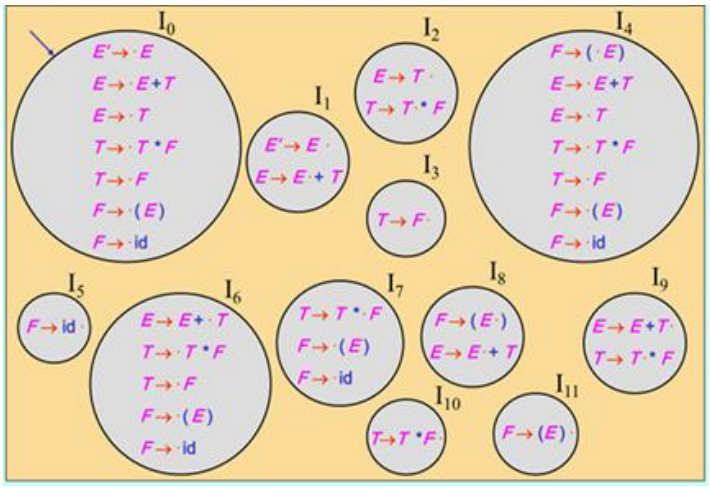
	nullable	FIRST	FOLLOW
S	false		$
E	false		$
T	false	x	+ $

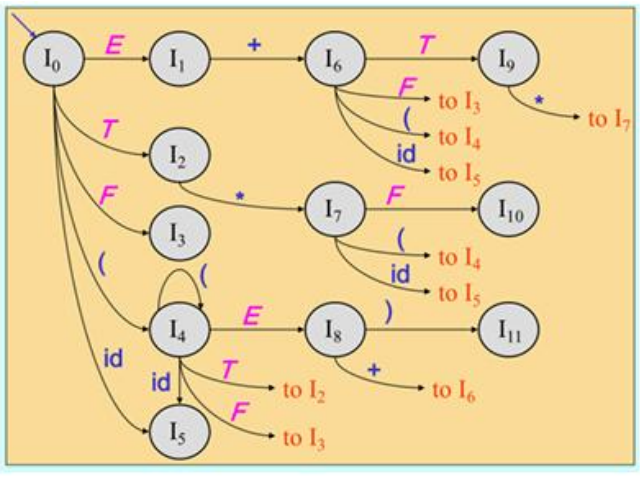
	nullable	FIRST	FOLLOW
S	false	x	$
E	false	x	$
T	false	x	+ $

The function **`slrTable(G)`** constructs the SLR parsing table for the CFG *G*, using the **`FOLLOW`** set of the *lhs* of the item to know for which terminal symbols a reduce has to be done, while the others will be errors

void slrTable (*CFG* G);
let $\{I_0, I_1, \dots, I_n\}$ be the result of items (G) ;
for (i = 0 to n)
  if ($A \to \alpha \cdot a \beta$ is in I_i and $a \in T$ and goto (I_i, *a*) = I_j)
    set ACTION[**i** , *a*] to ***shift j*** ;
  if ($A \to \alpha \cdot$ is in I_i and $A \neq S'$)
    set ACTION[**i** , *a*] to ***reduce*** $A \to \alpha$ for all *a* in *FOLLOW(A)*;
  if ($S' \to S \cdot$ is in I_i)
    set ACTION[**i** , $] to ***accept*** ;
  if (goto (I_i, A) = I_j and $A \in N$) set GOTO[**i** , A] to **j** ;

Example (no longer have conflicts):





$G_0 = (\{E, T, F\}, \{\text{id}, +, *, (,)\}, P, E)$

$P = \{$
$E \rightarrow E + T \mid T$ (1, 2)
$T \rightarrow T * F \mid F$ (3, 4)
$F \rightarrow (E) \mid \text{id}\ \}$ (5, 6)

	nullable	FIRST	FOLLOW
E	false		$ +)
T	false		*
F	false	(id	

	nullable	FIRST	FOLLOW
E	false		$ +)
T	false	(id	$ +) *
F	false	(id	$ +) *

	nullable	FIRST	FOLLOW
E	false	(id	$ +)
T	false	(id	$ +) *
F	false	(id	$ +) *

state	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LR(1) parsers: the **FOLLOW(A)** is the set of terminals that can immediately follow A in any string generated by a grammar G; it takes all the contexts where A can appear and not a specific case. By taking the specific context of A when the rule $A \rightarrow \alpha$ is applied, it could be possible to set a **reduce** $A \rightarrow \alpha$ action for a subset of FOLLOW(A) (avoiding potential conflicts).

A **LR(1) item** of a CFG grammar G is a production of G with a dot at some position of the *rhs*, and a lookahead symbol (terminal or **$**). An LR(1) item $[A \rightarrow \alpha \cdot, \alpha]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is $\boldsymbol{a}$

The item $[A \rightarrow \beta_1 \cdot \beta_2, a]$ is valid for a viable prefix $\alpha\beta_1$ if:

- there's a derivation $S \Rightarrow_{rm}^{*} \alpha \boldsymbol{A} w \Rightarrow_{rm}^{*} \alpha \boldsymbol{\beta_1 \beta_2} w$
- $\boldsymbol{a}$ is the 1st symbol of $\boldsymbol{w}$, or $\boldsymbol{w}$ is ε and $\boldsymbol{a}$ is **$** → so the lookahead symbol is the 1st symbol of next string or is $ (if w = empty string)

The **valid LR(1) items** for a CFG grammar are the **states of an NFA recognizing viable prefixes**; a DFA equivalent to this NFA will have states corresponding to **sets of LR(1) items** and transitions labeled by the **symbols of the viable prefixes**.

Let's see the pseudo-functions to build this DFA:

```
Items closure1 (Items I) ;
 repeat
  for (each item [A → α · X β , a] in I )
   for ( each production X → γ )
    for ( each b ∈ FIRST( β a ) )
      I = I ∪ { [X → · γ , b]} ;
 until ( I does not change ) ;
 return I ;
```

```
Items goto1 (Items I, Symbol X) ;
 J = ∅ ;
 for ( each item [A → α · X β , a] in I )
   J = J ∪ {[A → α X · β , a] } ;
 return closure1 (J) ;
```

```
ItemsCollection items1 (CFG G);
 G' = (N ∪ {S'}, T, P ∪ {S' → S}, S') ;
 C = closure1 ({[S' → · S , $]}) ;
 repeat
  for ( each set I in C )
   for ( each item [A → α · X β , a] in I )
     C = C ∪ { goto1 (I, X)} ;
 until ( C does not change ) ;
 return C ;
```

❖ **closure1(I)** → extends closure(I) considering also the lookahead symbol: we add an item to the closure only if its lookahead symbol is a possible "next symbol" after the *rhs* of input production

❖ **goto1(I,X)** → extends goto(I,X) saying that the lookahead needs to remain the same when adding an item to J

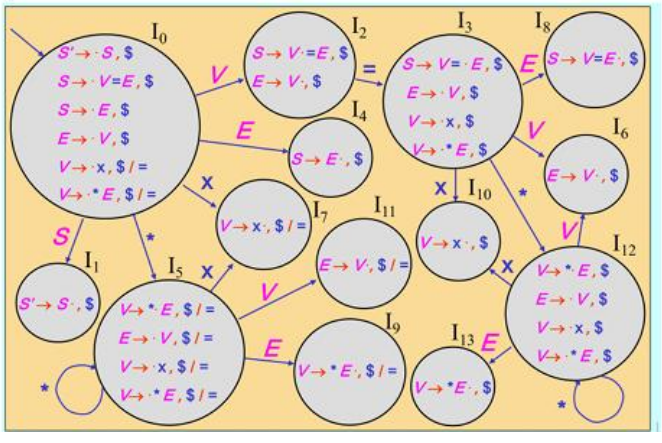
❖ **items1(G)** → extends items(G) [build the collection of DFA states $C = \{I_0, I_1, \ldots I_n\}$ given a CFG grammar $G = (N, T, P, S)$]: we use $ as look-ahead symbol for the start symbol and we add the lookahead symbol in the loop. Example:

$G_3 = (\{S, E, V\}, \{x, *, =\}, P, S)$

$P = \{ S \rightarrow V = E \mid E$
$E \rightarrow V$
$V \rightarrow x \mid *E \}$

$G_3' = (\{S', S, E, V\}, \{x, *, =\}, P', S')$

$P' = \{ S' \rightarrow S$ (0)
$S \rightarrow V = E \mid E$ (1, 2)
$E \rightarrow V$ (3)
$V \rightarrow x \mid *E \}$ (4, 5)



❖ `lr1Table(G)` → build the LR(1) parsing table for the CFG *G*. Example (continuing the above):

void lr1Table (*CFG* G);
let $\{I_0, I_1, \dots, I_n\}$ be the result of items1 (G);
for (i = 0 to n)
if ($[A \to \alpha \cdot a\,\beta, b]$ is in I_i and $a \in T$ and goto1 (I_i, a) = I_j)
set ACTION[i, a] to *shift* j;
if ($[A \to \alpha \cdot, a]$ is in I_i and $A \neq S'$)
set ACTION[i, a] to *reduce* $A \to \alpha$;
if ($[S' \to S \cdot, \$]$ is in I_i)
set ACTION[i, $] to *accept*;
if (goto1 (I_i, A) = I_j and $A \in N$) set GOTO[i, A] to j;

state	ACTION				GOTO		
	x	*	=	$	S	E	V
0	s7	s5			1	4	2
1				acc			
2			s3	r3			
3	s10	s12				8	6
4				r2			
5	s7	s5				9	11
6				r3			
7			r4	r4			
8				r1			
9			r5	r5			
10				r4			
11			r3	r3			
12	s10	s12				13	6
13				r5			

A **grammar** *G* is **LR(1)** if the ACTION table generated by function `lr1Table(G)` doesn't have conflicts (non-ambiguous). If any set of LR(1) items generated by `items1(G)` contains a complete item $[A \to \alpha \cdot, \alpha]$ (originating a reduce action), then:

- no other *complete* item in the set has ***a*** as *lookahead symbol* (avoid reduce/reduce)
- no other item in the set has ***a*** *immediately at the right of the dot* (avoid shift/reduce)

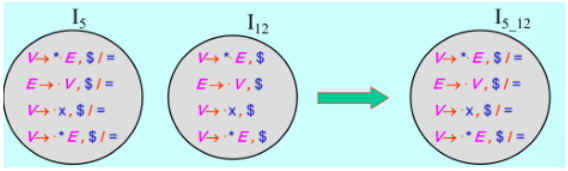
Summing, a **LR(1) Parser**:

- ➢ scans input from left to right (**L**)
- ➢ build a rightmost derivation in reverse (**R**)

→ the class of languages that can be parsed with LR(1) Parsers is exactly the class of deterministic CFLs; so just 1 lookahead symbol is enough

Lookahead LR(1) parsers [**LALR**]: **LR(1) parsing tables** can be very large (many states) for grammars generating programming languages; **SLR parsing tables** are smaller but can contain conflicts → **LALR(1) parsing tables** are small as SLR and can express most programming languages (but still without covering all the CFLs)

→ 2 sets of LR(1) items have the same core if they are identical except for the lookahead symbols; a set of LALR(1) items is the **union** of the sets of LR(1) items having the **same core**. This can reduce n° of states

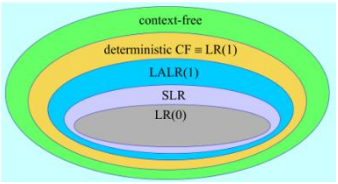


state	ACTION				GOTO		
	x	*	=	$	S	E	V
0	s7_10	s5_12			1	4	2
1				acc			
2			s3	r3			
3	s7_10	s5_12				8	6_11
4				r2			
5_12	s7_10	s5_12				9_13	6_11
6_11			r3	r3			
7_10			r4	r4			
8				r1			
9_13			r5	r5			

Example: in the image we see that the grammar is LALR(1) but it is not SLR because, since ***FOLLOW***`(E) = {=,$}`, in the SLR table we would have ***ACTION***`[2,=] = s3,r3` (a conflict) while in LALR we don't have conflicts

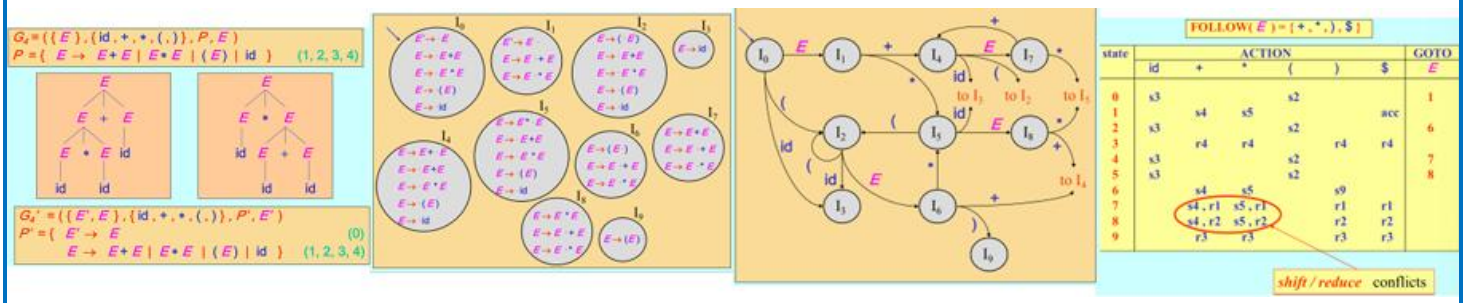
⚠ The merging of states with common cores can't produce a shift/reduce conflict not present in 1 of the original states (shift actions depend only on the core, not the lookahead)

⚠ Merging can produce a reduce/reduce conflict



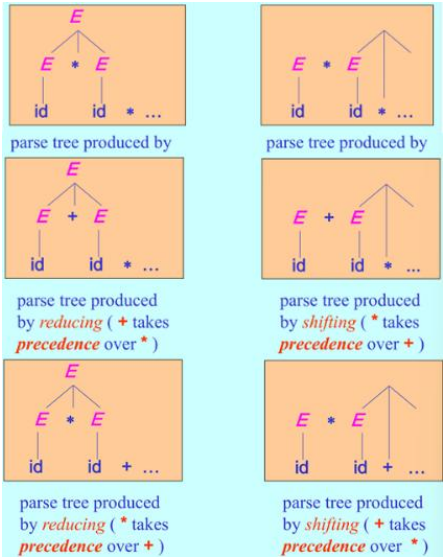
⚠ The class of languages that can be parsed using LALR(1) parsers is a proper subset of the deterministic CFLs

AMBIGUOUS GRAMMARS: aren't LR(k) but can provide shorter and more natural specifications then any equivalent unambiguous grammar. To have a **more efficient parser**, we can use ambiguous grammars implementing disambiguating rules (as precedence and associativity)



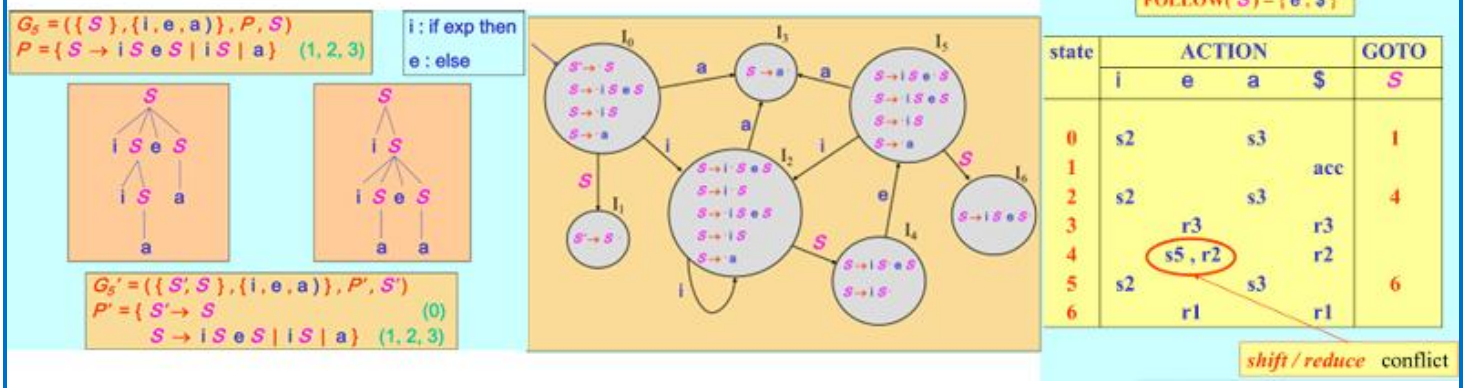
In this case (image above) there are **different conflicts**:

- conflict in **`ACTION[7,+]=s4,r1`** is because of items $E \to E + E \cdot$ and $E \to E \cdot + E$; the top of the stack is $E + E$ and the next input symbol is *+*
- conflict in **`ACTION[8,*]=s5,r2`** is because of items $E \to E * E \cdot$ and $E \to E \cdot * E$; the top of the stack is $E * E$ and the next input symbol is ***
- conflict in **`ACTION[7,*]=s5,r1`** is because of items $E \to E + E \cdot$ and $E \to E \cdot * E$; the top of the stack is $E + E$ and the next input symbol is ***
- conflict in **`ACTION[8,+]=s4,r1`** is because of items $E \to E * E \cdot$ and $E \to E \cdot + E$; the top of the stack is $E * E$ and the next input symbol is *+*



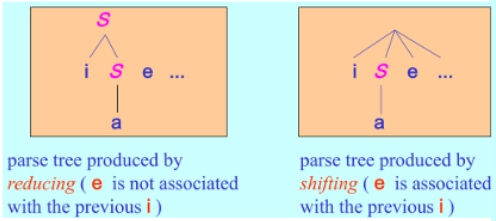
We can solve the conflicts just by defining associativity and precedence directives → we say that * and **+** are left-associative and * takes precedence over **+** (⚠ in some cases it's possible that, by defining such rules, the grammar remains ambiguous)

state	ACTION						GOTO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	



state	ACTION				GOTO
	i	e	a	$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5 , r2		r2	
5	s2		s3		6
6		r1		r1	

The conflict in **`ACTION[4,e]=s5,r2`** is because of items $S \to iS \cdot eS$ and $S \to iS \cdot$; the top of the stack is iS and the next input symbol is *e*: in this case the conflict is resolved choosing the shift.



state	ACTION				GOTO
	i	e	a	$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

Error recovery in LR parsing: "**BLANKS**" in LR parsing tables are **error actions** that stop the parser. Local error recovery mechanisms use a special error symbol to allow parsing to resume; when the error symbol appears in a grammar rule, it can match a sequence of erroneous input symbols:

- the production (3) $\boldsymbol{S \to error}$; $\boldsymbol{E}$ says that the parser can skip to the next '**;**' when encounters a syntax error
- the production (8) $\boldsymbol{E \to (error)}$ says the parser, encountering a syntax error after '**(**', can skip to the next '**)**'

Instead, if $\boldsymbol{A \to error\ \alpha}$ is a grammar production in the construction of the parsing table, the error is considered a terminal symbol and error productions are treated as ordinary productions.

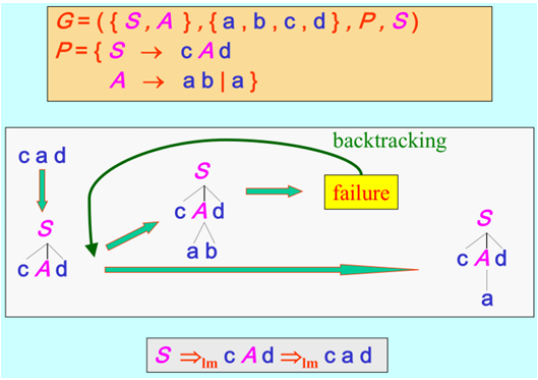
When **encountering an error action** (a blank in the table), the Parser:

- **pop** the stack until reaching a state where the action for error is shift (state with a item $A \to \cdot\ error\ \alpha$)
- **shift** a fictitious `error token` on the stack (as error was found on input)
- **skip** ahead on the input discarding symbols until finding a substring that can be reduced to α (looking if it's possible to continue after the error)
- **reduce** the handle error α (at this point on top of the stack) to A
- **emit** a diagnostic message
- **resume** normal parsing

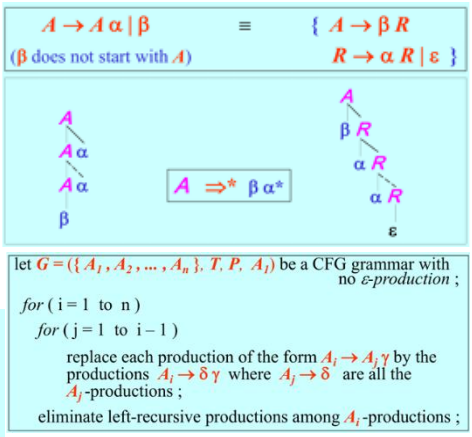
Error rules may introduce both shift/reduce and reduce/reduce conflicts, so they can't be inserted anywhere in LALR grammars. This error mechanism isn't powerful enough to correct all syntax errors

⚠ Constructing a Parser is simple enough to be automated (es. **CUP**)

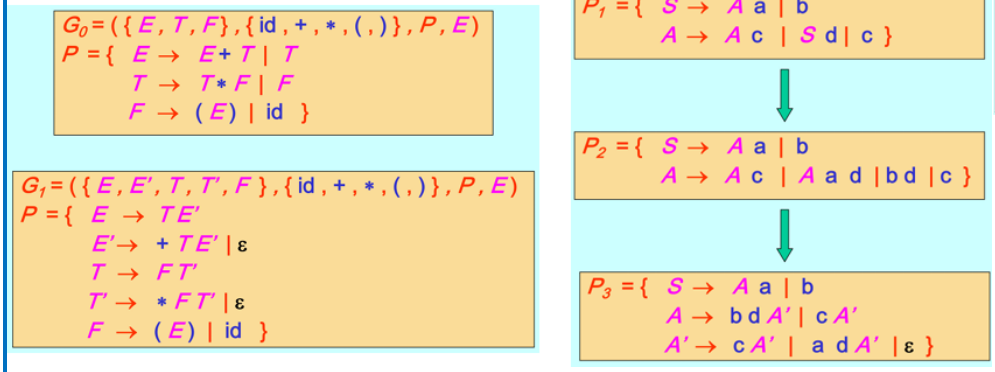
- **TOP-DOWN**: construct a parse tree for an input string **from root to leaves** (**from top to bottom**). This process creates the nodes of the tree in preorder until it obtains the input string; at each creation step, the left side symbol of a production is replaced by its right side (so **leftmost derivation**)



Left-recursive grammars: a production like $A \to A\alpha$ is called a left-recursive production. Grammar is **LEFT-RECURSIVE** if it can generate a derivation $A \Rightarrow^{+} A\alpha$; this grammars can cause a top-down parser to go into an infinite loop ($A \Rightarrow^{+}_{lm} A\alpha \Rightarrow^{+}_{lm} A\alpha\alpha\ldots$)



But left-recursive productions can be replaced by **right-recursive** productions: to do so, the algorithm starts by considering a grammar where all the non-terminal symbols are defined in a certain order and there are not ε-productions. For each non-terminal symbol, the algorithm replaces each production of the form $A_i \to A_j\gamma$ by the productions $A_i \to \delta\gamma$ where $A_j \to \delta$ are all the A_j-productions

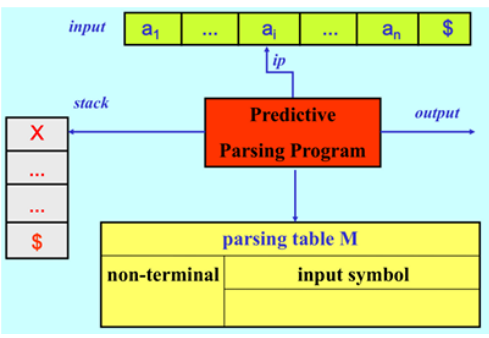


Practically it substitute A_j with the *rhs* of his production

⚠ In top-down parsing, ***backtracking*** can be avoided if it is possible to detect which alternative rule has to be applied, by considering the current input symbol. If the next symbol is **if**, the first rule will be applied

- **NON-RECURSIVE PREDICTIVE**: has a pointer (**ip**) to the current input symbol, a reference to a stack where the symbols are pushed when read, and a parsing table (which helps with decisions and is built on the grammar).

  Predictive parsing program: starts with **push $**, the grammar start symbol (S) on the stack and set **ip** to point to the 1st input symbol. Repeats the following actions until the top of the stack == **$**:



- top of the stack = ***X***; symbol pointed by `ip` = ***a***
- if ***X*** == (terminal symbol or **$**) == ***a*** (so pointed by `ip`) → **pop** ***X*** from the stack and advance `ip` to the next input symbol; if ***X*** != ***a*** → error
- if ***X*** == non-terminal symbol and the table cell **M[*X*, *a*]** is a production where ***X*** is in the *lhs*, **pop** ***X*** from the stack and **push** the *rhs* of the production having on top the 1st symbol of *rhs*; if table `empty`, error

```
push $ onto the stack ;
push the start symbol of the grammar onto the stack ;
set ip to point to the first input symbol ;
repeat
  { let X be the top stack symbol and a the symbol pointed to by ip ;
    if ( X is a terminal or $ )
        if ( X = a )
            { pop X from the stack ;
              advance ip to the next input symbol }
        else error
    else /* X is a non-terminal */
        if ( M[X , a] = X → Y1 Y2 ... Yk )
            { pop X from the stack ;
              push Yk Yk-1 ... Y1 onto the stack, with Y1 on top ;
              output the production X → Y1 Y2 ... Yk ; }
        else error
  }
until ( X = $ ) /* stack is empty */
```

$G_1 = (\{E, E', T, T', F\}, \{\text{id}, +, *, (,)\}, P, E)$

$P = \{$
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid \text{id} \}$

non terminal	input symbol					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

stack	input	output
$ E	id + id * id $	
$ E' T	id + id * id $	E → TE'
$ E' T' F	id + id * id $	T → FT'
$ E' T' id	id + id * id $	F → id
$ E' T'	+ id * id $	
$ E'	+ id * id $	T' → ε
$ E' T +	+ id * id $	E' → +TE'
$ E' T	id * id $	
$ E' T' F	id * id $	T → FT'
$ E' T' id	id * id $	F → id
$ E' T'	* id $	
$ E' T' F *	* id $	T' → *FT'
$ E' T' F	id $	
$ E' T' id	id $	F → id
$ E' T'	$	
$ E'	$	T' → ε
$	$	E' → ε

To build the parsing table: for each production $A \rightarrow \alpha$ in the grammar and for each ***a*** in **FIRST**(α), set the cell **M[*A*, *a*]** to $A \rightarrow \alpha$; then if α is nullable: for each element ***b*** in the **FOLLOW**(A), set **M[*A*, *b*]** to $A \rightarrow \alpha$

$G_1 = (\{E, E', T, T', F\}, \{\text{id}, +, *, (,)\}, P, E)$

$P = \{$
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid \text{id} \}$

	nullable	FIRST	FOLLOW
E	false	(id	$)
E'	true	+	$)
T	false	(id	$) +
T'	true	*	$) +
F	false	(id	$) + *

```
for ( each production A → α )
  for ( each a in FIRST(α) )
        set M[A , a] to A → α ;
  if ( α is nullable )
      for ( each b in FOLLOW(A) )
            set M[A , b] to A → α ;
```

non terminal	input symbol					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

LL(1) grammars: a grammar G is **LL(1)** if its predictive parsing table has no multiply-defined entries: it means that when $A \rightarrow \alpha|\beta$, then:

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
- at most 1 of α and β is **nullable**
- if α is **nullable**, then $\text{FIRST}(\beta) \cap \text{FOLLOW}(\alpha) = \emptyset$

⚠ So, no ambiguous or left-recursive grammars can be **LL(1)**

Summing, a **LL(1) Parser**:

- ➢ scans input from left to right (**L**)
- ➢ build a leftmost derivation in reverse (**L**)
- ➢ uses **1** lookahead input symbol when making parsing decisions

→ the class of languages that can be parsed with LL(1) Parsers is a proper subset of deterministic CFLs (less powerful than the LR(1) because it has fewer languages)

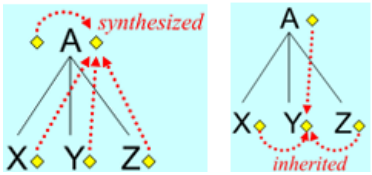
⚠ An LL parser generator transforms the specification of a Parser into a program implementing an LL parser

3) SYNTAX-DIRECTED TRANSLATION

A **Syntax-Directed Definition** (**SDD**) is a **CFG** in which:

- each symbol can have an associated set of **attributes** (numbers, types, table references, strings…)
- each production can have an associated set of **semantic rules** (evaluating attributes, interacting with table…)

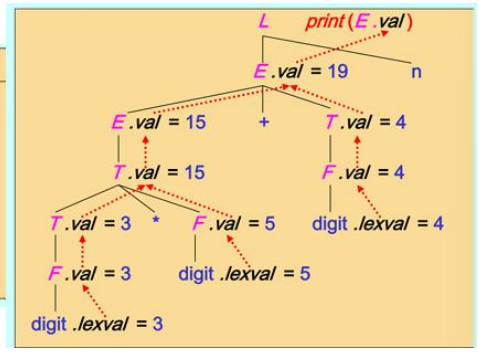
A **semantic rule associated with a production** $A \rightarrow XYZ$ can refer only attributes associated with symbols in that production:



- **synthesized** attributes = evaluated in rules where the associated symbol is on the left side of the production
- **inherited** attributes = evaluated in rules where the associated symbol is on the right side of the production

Example: each of **non-terminals E, T, F** has a single synthesized attribute (named **`val`**); the **terminal digit** has an attribute (**`lexval`**) which is the int value returned by the Scanner:

productions	semantic rules
$L \rightarrow E$ n	print (E .val)
$E \rightarrow E_1 + T$	E .val = E_1 .val + T .val
$E \rightarrow T$	E .val = T .val
$T \rightarrow T_1 * F$	T .val = T_1 .val * F .val
$T \rightarrow F$	T .val = F .val
$F \rightarrow (E)$	F .val = E .val
$F \rightarrow$ digit	F .val = digit .lexval



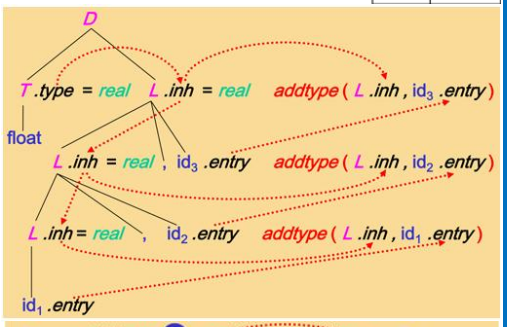
Example2:

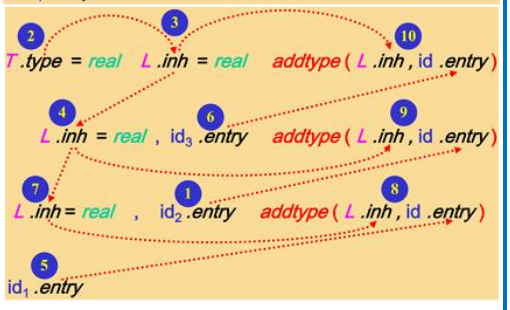
- non-terminal **T** has synthesized attribute (**`type`**)
- non-terminal **L** has inherited attribute (**`inh`**)
- terminal **id** has attribute (**`entry`**) which is the value returned by the scanner (it points to the symbol table entry for the identifier associated with **id**)
- **function** **`addtype(L.inh, id.entry)`** installs the type `L.inh` at the symbol table position `id.entry`

productions	semantic rules
$D \rightarrow T L$	L .inh = T .type
$T \rightarrow$ int	T .type = integer
$T \rightarrow$ float	T .type = real
$L \rightarrow L_1$, id	L_1 .inh = L .inh ; addtype (L .inh , id .entry)
$L \rightarrow$ id	addtype (L .inh , id .entry)

Symbol table	
var1	int
var2	float

The **annotated parse tree** represents the dependencies between the attributes (in the Example2 we see the annotated parse tree for float id_1, id_2, id_3) [img →]





An attribute at a node in an annotated parse tree must be evaluated after the evaluation of all attributes from which its value **depends**: the **dependency relations** in a tree define a **dependency graph**; a topological sort of the dependency graph is a **order of evaluation for the SDD** and any directed acyclic graph has at least one topological sort.

If we associate an order number to every node of the graph, we have a topological sort if, given $\boldsymbol{a}$ as the source node and $\boldsymbol{b}$ as the destination node, we have $\boldsymbol{n(a) < n(b)}$

The **syntax-directed translation** can be performed by:

- creating the parse tree
- visiting the parse tree and evaluating a SDD through topological sort of the dependency graph

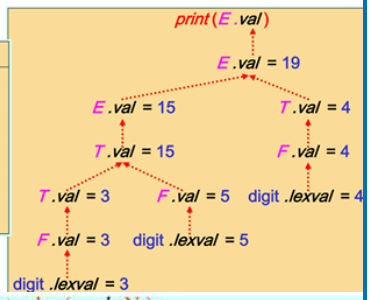
But it has **2 problems**:

1. To visit the parse tree, we need to store it in **memory** (a lot for large programs)
2. Check if the dependency graph of any parse tree contains **cycles** (big time-complexity)

To solve this we can define **classes of SDD** (**S-attributed** and **L-attributed**) so that cycles aren't allowed and translation is performed in connection with parsing (without creating explicitly the tree nodes); so we have **SDD**:

- **S-attributed** → every attribute is **synthesized**: all the semantic rules use only attributes of symbols in the right side of the associated productions.

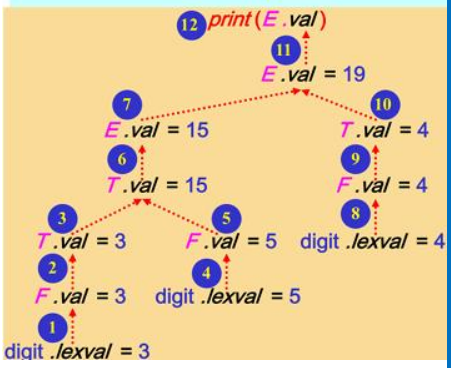
productions	semantic rules
$L \rightarrow E$ n	print (E .val)
$E \rightarrow E_1 + T$	E .val = E_1 .val + T .val
$E \rightarrow T$	E .val = T .val
$T \rightarrow T_1 * F$	T .val = T_1 .val * F .val
$T \rightarrow F$	T .val = F .val
$F \rightarrow (E)$	F .val = E .val
$F \rightarrow$ digit	F .val = digit .lexval



S-attributed SDD can be evaluated in any **bottom-up order** since they are independent from the evaluation order of their "siblings", but rely only on their child nodes.

```
void postorder ( node N ) ;
    for ( each child C of N , from left to right )
        postorder (C) ;
    evaluate the attributes and semantic rules
    associated with node N ;
```

The evaluation order of the function **`postOrder(rootNode)`** corresponds to the order in which a bottom-up parser creates nodes in a parse tree: this function recursively traverses the tree from the last leftmost child, evaluates it, and then proceeds to its sibling and its children, continuing this process until it reaches the **`rootNode`** → the order in which the nodes are traversed is referred to as the ***topological order***



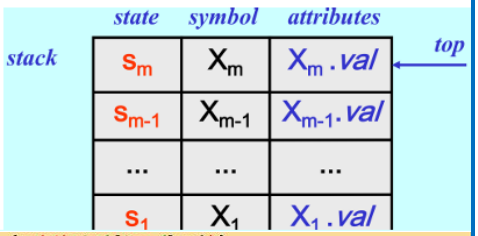
- **Syntax-Directed Translation scheme** (**SDT**) → SDD with the actions of each semantic rule **embedded** at some positions on the right side of the associated production. An SDT executes each action when all the grammar symbols to the left of the action are processed; if all actions are at the end of the *rhs*, we have a "postfix" SDT.
  The action $\boldsymbol{a}$ in the rule $A \rightarrow X\{\boldsymbol{a}\}Y$ should be performed in:
  - bottom-up, as soon as this occurrence of X appears on the top of the parsing stack
  - top-down:
    - if Y is non-terminal, just before attempting to expand this occurrence of Y
    - if Y is terminal, just before checking for Y on the input

How a **S-attributed SDD** can be **transformed into a postfix SDT**? Replace the action with a marker symbol and create an empty production rule for that marker symbol, associating the action at the end:

$$A \rightarrow X\{\boldsymbol{a}\}Y \quad \text{becomes} \quad A \rightarrow X\boldsymbol{M}Y \quad \text{with} \quad \boldsymbol{M} \rightarrow \boldsymbol{\varepsilon}\{\boldsymbol{a}\}$$

Synthesized attributes can be placed along with the grammar symbols **on the parser stack**:

- when a handle β is on top of the stack, all the synthesized attributes of β have been evaluated
- when the reduction of β occurs, the associated actions can be executed




```
L → E n     { print ( stack [top - 1].val ) }
E → E + T   { n_top = top – 3 + 1 ;
              stack [n_top].val = stack [top - 2].val + stack [top].val ;
              top = n_top }
E → T
T → T * F   { n_top = top - 3 + 1 ;
              stack [n_top].val = stack [top - 2].val * stack [top].val ;
              top = n_top }
T → F
F → ( E )   { n_top = top - 3 + 1 ;
              stack [n_top].val = stack [top - 1].val ;
              top = n_top }
F → digit
```

Example (img): we handle the stack by updating the top value when a reduction occurs, and we access the stack elements to calculate the value of the synthesized attribute for the *lhs* of the production

- **L-attributed** → any production $A \rightarrow X_1 X_2 \ldots X_n$ has:
  - **synthesized** attributes (2^{nd},3^{rd},4^{th},5^{th} productions of the example)
  - **inherited** attributes $X_i.a$ computed in terms of:

productions	semantic rules
$D \rightarrow T\ L$	L .inh = T .type
$T \rightarrow$ int	T .type = integer
$T \rightarrow$ float	T .type = real
$L \rightarrow L_1$, id	L_1 .inh = L .inh ; addtype (L .inh , id .entry)
$L \rightarrow$ id	addtype (L .inh , id .entry)

- inherited attributes associated to A (4th and 5th productions of the example)
- inherited or synthesized attributes associated with symbols $X_1, X_2 \dots X_{i-1}$ located at the left (**L**) of the symbol X_i (1st production of the example)

How a **L-attributed SDD** can be **transformed into a SDT**?

- place the actions that compute an inherited attribute for a symbol X immediately **before** X
- place the actions that compute a synthesized attribute at the **end** of the production

$D \rightarrow T \ \{L.inh = T.type\} \ L$
$T \rightarrow$ int $\{T.type = integer\}$
$T \rightarrow$ float $\{T.type = real\}$
$L \rightarrow \{L_1.inh = L.inh\} \ L_1$, id $\{addtype(L.inh, id.entry)\}$
$L \rightarrow$ id $\{addtype(L.inh, id.entry)\}$

A bottom-up parsers is aware of the production that is using only when performs a **`reduction`**; this parser can **execute actions** associated with a production only when they are at the end of the production, but actions that compute inherited attributes are not at the end of the production.
So, we can transform an L-attributed SDD where **all actions are at the end** of productions: if we have $A \rightarrow X \ \{\boldsymbol{Y.i = X.s}\} \ Y$ (where $X.s$ is a synthesized attribute and $Y.i$ is an inherited attribute defined by a ***copy rule***), the value $X.s$ is already on the parser stack before any reduction to Y is performed; so we just need to retrieve $X.s$ from 1 position before Y on the stack (in this way the rule can be replaced retrieving the symbol directly from the stack):

$D \rightarrow T \ \{L.inh = T.type\} \ L$
$T \rightarrow$ int $\{T.type = integer\}$
$T \rightarrow$ float $\{T.type = real\}$
$L \rightarrow \{L_1.inh = L.inh\} \ L_1$, id $\{addtype(L.inh, id.entry)\}$
$L \rightarrow$ id $\{addtype(L.inh, id.entry)\}$

$D \rightarrow T \ L$
$T \rightarrow$ int $\{stack[top].val = integer\}$
$T \rightarrow$ float $\{stack[top].val = real\}$
$L \rightarrow L$, id $\{addtype(stack[top-3].val, stack[top].val)\}$
$L \rightarrow$ id $\{addtype(stack[top-1].val, stack[top].val)\}$

stack	input	production	action
$	float id_1, id_2, id_3 $		
$ float	id_1, id_2, id_3 $	$T \rightarrow$ float	stack[top].val = real
$ T	id_1, id_2, id_3 $		
$ T id_1	, id_2, id_3 $	$L \rightarrow$ id	addtype(stack[top - 1].val, stack[top].val)
$ T L	, id_2, id_3 $		
$ T L ,	id_2, id_3 $		
$ T L , id_2	, id_3 $	$L \rightarrow L$, id	addtype(stack[top - 3].val, stack[top].val)
$ T L	, id_3 $		
$ T L ,	id_3 $		
$ T L , id_3	$	$L \rightarrow L$, id	addtype(stack[top - 3].val, stack[top].val)
$ T L	$	$D \rightarrow T L$	
$ D	$	accept	

⚠ However, reaching into the parser stack for an attribute value works only if the grammar allows the position of the attribute value to be predicted

$S \rightarrow a \ A \ \{C.i = A.s\} \ C$ (1)
$S \rightarrow b \ A \ B \ \{C.i = A.s\} \ C$ (2)
$C \rightarrow c \ \{C.s = f(C.i)\}$ (3)

$S \rightarrow a \ A \ \{C.i = A.s\} \ C$
$S \rightarrow b \ A \ B \ \{C.i = A.s\} \ C$
$C \rightarrow c \ \{C.s = f(C.i)\}$

$S \rightarrow a \ A \ C$
$S \rightarrow b \ A \ B \ M \ C$
$C \rightarrow c \ \{stack[top].val = f(stack[top-1].val)\}$
$M \rightarrow \varepsilon \ \{stack[top].val = stack[top-2].val\}$

In this SDT the value $A.s$ can be either 1 or 2 positions in the stack before C (1st and 2nd productions). To place the value of $A.s$ always 1 position before C, we can insert just before C in rule (2) a new marker non-terminal symbol with a synthesized attribute $\boldsymbol{M}.s$ having the same value of A.s

But if we have an **L-attributed** translation with a rule $A \rightarrow X \ \{\boldsymbol{Y.i = f(X.s)}\} \ Y$ (where $X.s$ is a synthesized attribute and $Y.i$ is an inherited attribute **not** defined by a ***copy rule***), the value $X.s$ isn't already on the parser stack before any reduction to Y is performed because $\boldsymbol{Y.i}$ **isn't just a copy of** $\boldsymbol{X.s}$. In this case we can insert just before Y a new marker non-terminal symbol $\boldsymbol{M \rightarrow \varepsilon \ \{M.s = f(M.i)\}}$ with:

$S \rightarrow a \ A \ \{C.i = f(A.s)\} \ C$
$C \rightarrow c \ \{C.s = g(C.i)\}$

$S \rightarrow a \ A \ M \ C$
$C \rightarrow c \ \{stack[top].val = g(stack[top-1].val)\}$
$M \rightarrow \varepsilon \ \{stack[top].val = f(stack[top-1].val)\}$

- an inherited attribute $M.i = X.s$
- a synthesized attribute $M.s$ to be copied in $Y.i$ and to be evaluated in a new rule

⚠ The introduction of markers makes it possible to evaluate L-attributed translations during bottom-up parsing, but a **LR(1)** grammar **may not remain** LR(1) after introduction of markers. Instead, **LL(1)** grammars **remain** LL(1) even when markers are introduced: so, since LL(1) grammars are a proper subset of the LR(1), L-attributed translation scheme based on an LL(1) grammar can be parsed bottom-up

4) SEMANTIC ANALYSIS

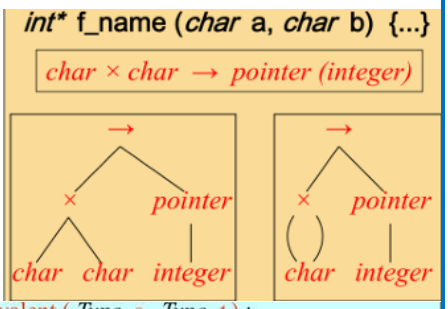
The **SEMANTIC ANALYSIS** checks the source program for semantic error and collects type information for the code-generation phase:

- **type** checks = the type of a construct must match the expected by its context

- **name-related and uniqueness** checks = objects must be declared exactly once
- **flow-of-control** checks = statements (as break and continue) that cause flow of control to leave a construct must have a place where to go

A **TYPE EXPRESSION** `T` denotes the type of a language construct that can be:

- **basic type** (`int`, `real`, `char`, `bool`, `void`, …, `type_error`)
- **type constructor**:
  - Array → $array(indexset, T)$
  - Cartesian product → $T_1 \times \ldots \times T_n$
  - Record → $record((name_1 \times T_1) \times \ldots \times (name_n \times T_n))$
  - Pointer → $pointer(T)$
  - Function → $T_1 \times \ldots \times T_n \rightarrow T$

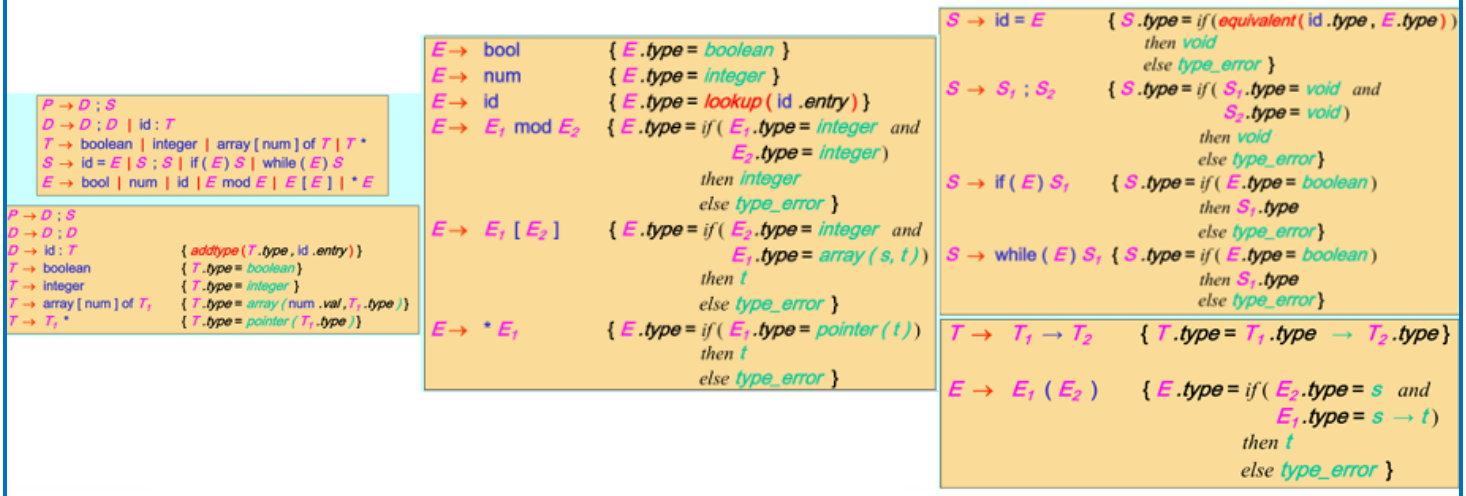


⚠ Type expression can be represented by tree or direct acyclic graph (DAG) with type constructors as **interior nodes** and basic types or type names as **leaves** [1st representation is better (sx)]

⚠ The function `equivalent` is used to check if 2 type expressions are equivalent (works for basic types, array, cartesian product, pointer, function; it's also extendable to classes)

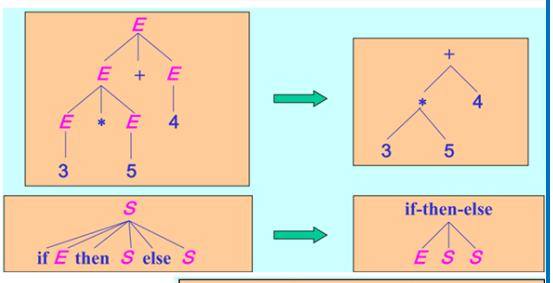
```
boolean equivalent ( Type s , Type t ) ;
  if ( s and t are the same basic type ) return true
  else if ( s = array (s1 , s2) and t = array (t1 , t2) )
       return (equivalent (s1 , t1 ) and equivalent (s2 , t2 ))
  else if ( s = s1 × s2 and t = t1 × t2 )
       return (equivalent (s1 , t1 ) and equivalent (s2 , t2 ))
  else if ( s = pointer (s1) and t = pointer (t1) )
       return equivalent (s1 , t1 )
  else if ( s = s1 → s2 and t = t1 → t2 )
       return (equivalent (s1 , t1 ) and equivalent (s2 , t2 ))
  else if ...
  else return false
```

Example (the recursive production $D \rightarrow D; D$ for the creation of a list is wrong and should be $LD \rightarrow SC\ D \mid D$;):



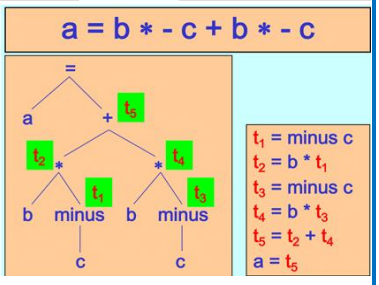
5) INTERMEDIATE CODE GENERATION

The **SYNTAX TREE** is a condensed **parse tree** where operators and keywords replace their non-terminal parent nodes.



The **three-address code** is a linearized representation of a **syntax tree** in which explicit names correspond to interior nodes. It's built from 2 concepts:

- **ADDRESS**
  - Source-program name [var1, x]
  - Constant [2, 3.5, 'c']
  - Compiler-generated tmp name [t_1, t_2]



- **INSTRUCTION**
  - Assignment → $x = y,\ x = op_1\ y,\ x = y\ op_2\ z$ (where x, y, z are addresses; op_1 is a unary operator; op_2 is a binary operator)
  - Indexed assignment → $x = y[i], x[i] = y$ (not allowed $a[2] = b[3]$, so split in $t_1 = b[3]$ and $a[2] = t_1$)
  - Address and pointer assignment → $x = \&y,\ x =^* y,\ *x = y$
  - Unconditional jump → $goto\ L$
  - Conditional jump → $if\ \ x\ \ goto\ L,\ \ if\ (x\ relop\ y)\ goto\ L$
  - Procedure call → $p(x_1, \dots, x_n)$
  - Procedure return → $return\ y$

Objects can have 2 representations:
- Quadruplet $(op, arg_1, arg_2, result)$
- Triple (op, arg_1, arg_2) [result is referred by its position]

⚠ Triple is more compact, but, in the case of code reordering, it is necessary to analyze every line of code

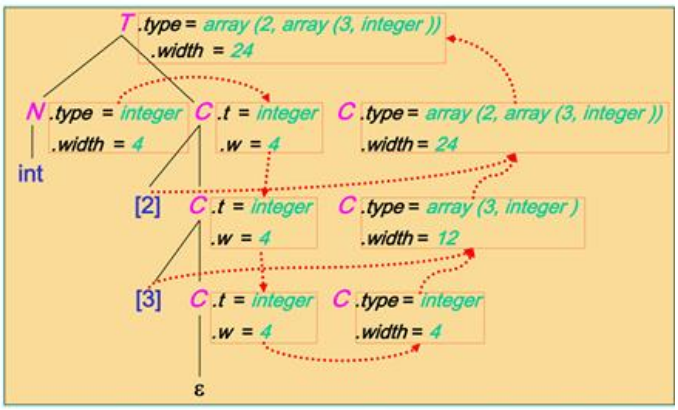
t_1 = minus c
t_2 = b * t_1
t_3 = minus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

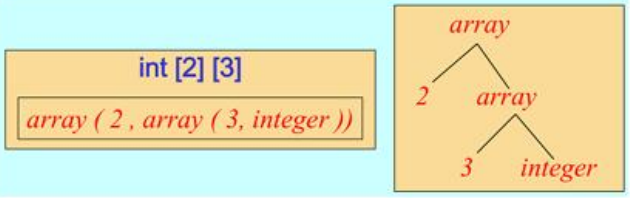
	op	arg_1	arg_2	result
(0)	minus	c		t_1
(1)	*	b	t_1	t_2
(2)	minus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		a

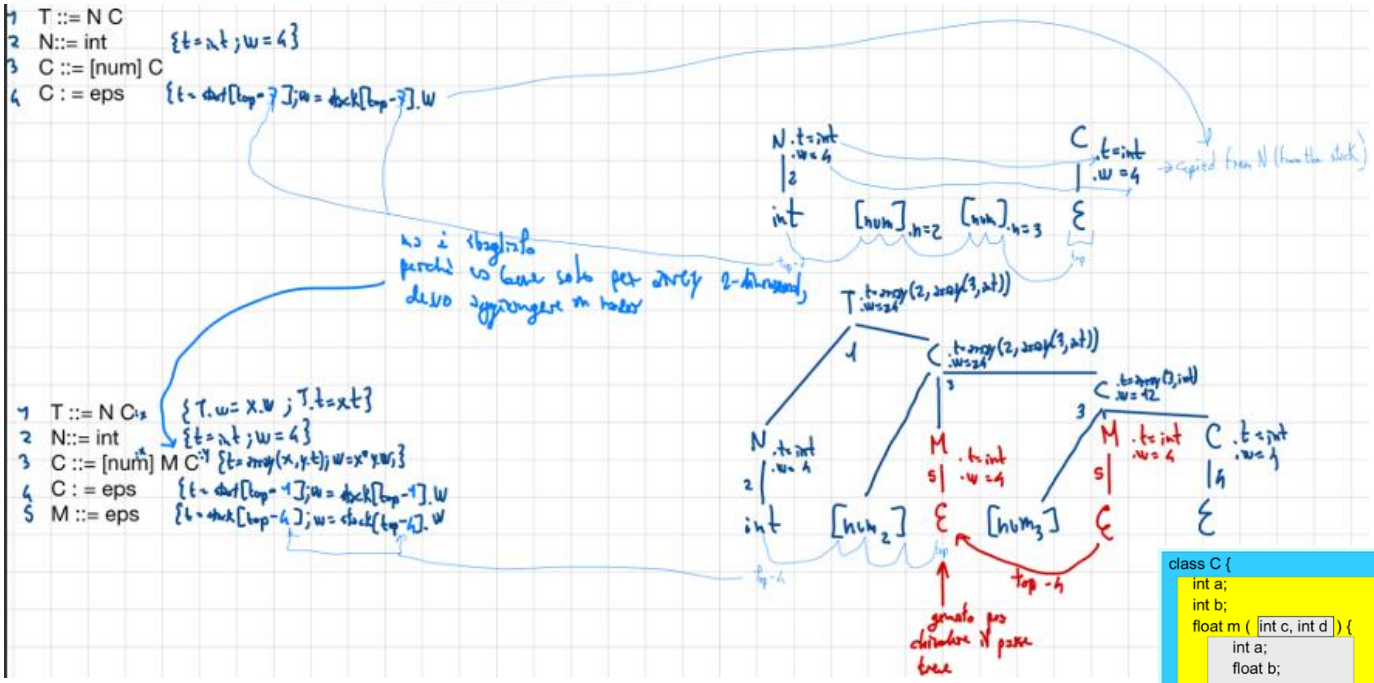
	op	arg_1	arg_2
(0)	minus	c	
(1)	*	b	(0)
(2)	minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

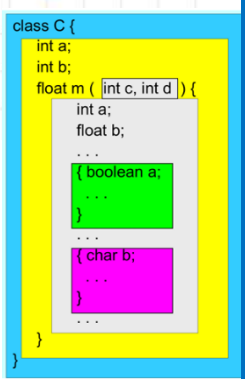
Example (shows a parser that computes the size in memory of a type, starting from the type definition. It can be also defined in an easier way using the stack, as shown below in the handwritten example):

T →	N	{C.t = N.type ; C.w = N.width}
	C	{T.type = C.type ; T.width = C.width}
N →	int	{ N.type = integer ; N.width = 4 }
N →	real	{ N.type = real ; N.width = 8 }
C →	ε	{ C.type = C.t ; C.width = C.w}
C →	[num]	{ C_1.t = C.t ; C_1.w = C.w}
	C_1	{ C.type = array (num .val , C_1.type) ; C.width = num .val * C_1.width}



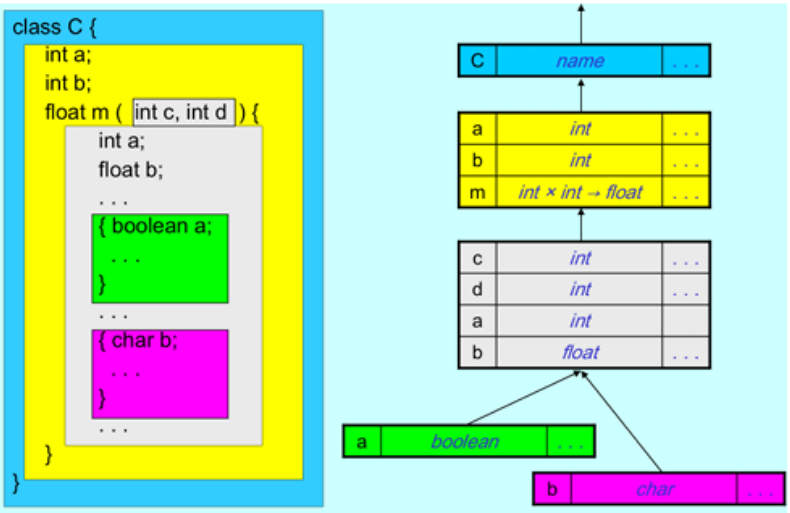






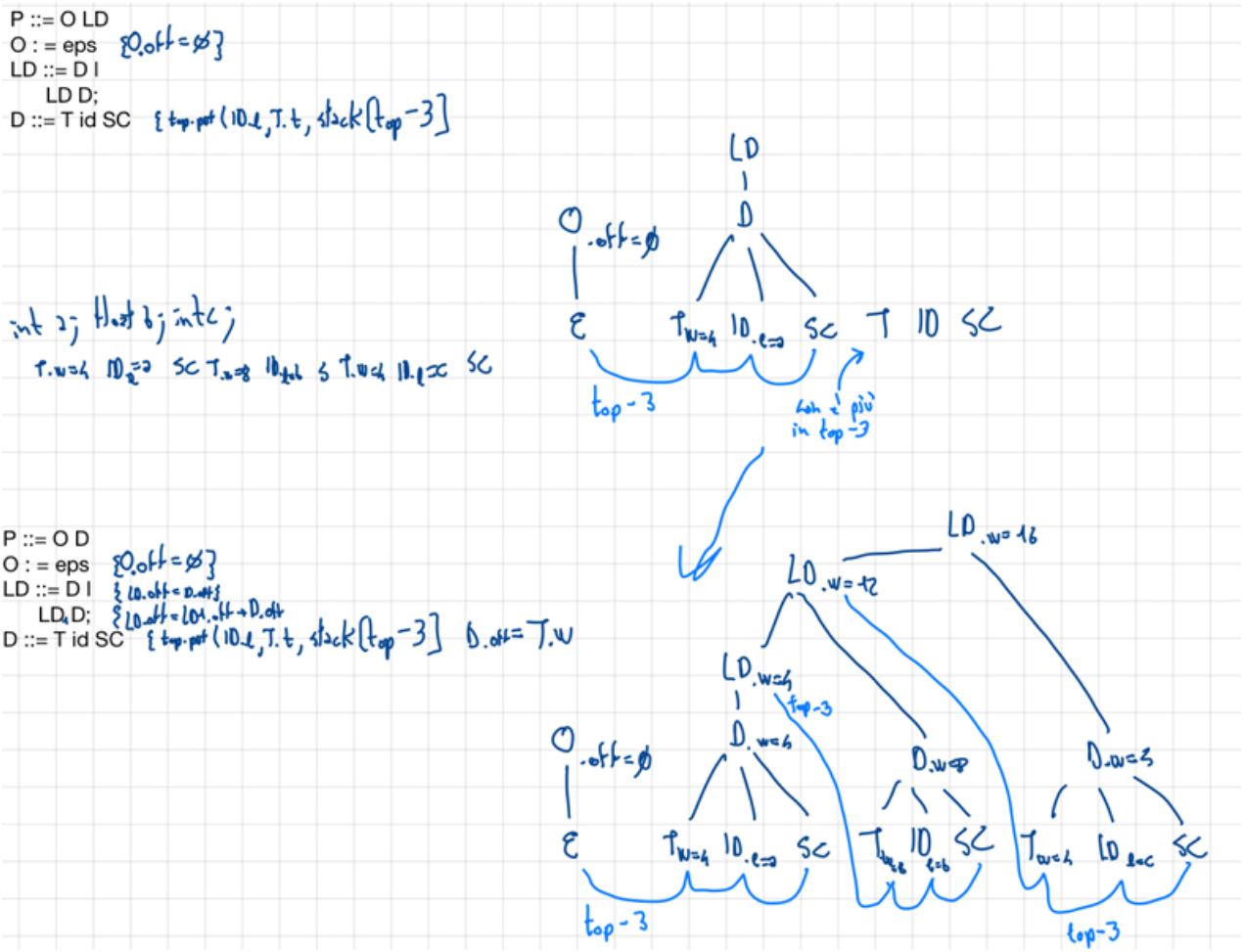
Scope of Declarations → the **scope** of a declaration of an identifier **x** is the region of a program **in which the uses of x refer to this declaration** (la classica definizione di ***scope***) and is determined by where the declaration of **x** appears in the program and by keywords (**`public`**, **`private`**, **`protected`**). If there are multiple declarations in a nested environment that allows redeclarations, we take the declaration in the scope of the **most-closely nested** declaration of **x**

To enforce this behaviour, **SYMBOL TABLE**s can be used (a data structure to hold info about source-program constructs, where the info is collected incrementally in the **analysis** phase and used in the **synthesis** phase to generate the code).
Its **entries** can contain various info about an identifier (string, type, memory address, ...). So, Multiple declarations ("redeclarations") of the same identifier can be supported by **using a separate symbol table for each scope** (the "most-closely nested" rule can be implemented by chaining the symbol tables):




```
public class Env {
    Hashtable <String, Symbol>  table ;
    Env prev ;
    // Create a new symbol table
    public  Env ( Env p ) {
            table = new Hashtable <String, Symbol> ( ) ;
            prev = p ;
    }
    // Put a new entry in the current table
    public  boolean  put ( String s, Symbol sym ) {
            if ( table.containsKey ( s ) ) return false ;
            table.put ( s, sym ) ;
            return true ;
    }
    // Get an entry for an identifier by searching the chain of tables
    public  Symbol  get ( String s ) {
            for ( Env e = this ;  e != null ;  e = e.prev ) {
               Symbol  found =  e.table.get ( s ) ;
               if ( found != null )  return  found ;
            }
            return null ;
    }
}
```

We can manage the storage of our code assigning a memory address to an identifier with a type: "next" address is calculated with respect to the identifiers previously stored in the symbol table (see handwritten example):



In this example, we use the stack to **access the 1st free address which is always in top-3** (another solution would be to use a global variable **`offset`** to track, but global variables are bad!)

```
P → { offset = 0 } D
D → D D
D → T id ;    { top.put ( id .lexeme , T .type , offset ) ;
                offset = offset + T .width }
```

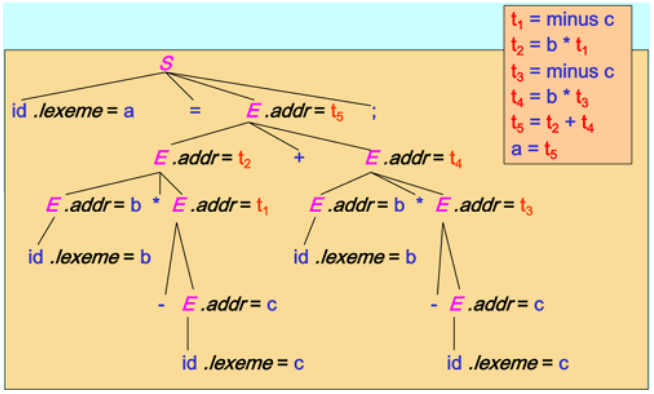
In this case, there is a production that adds **record types** (like structs) $T \rightarrow record\{D\}$, since a field *x* in a record type doesn't conflict with othere uses of *x* outside (because each record type has its own symbol table; so in this context, **`offset`** refers to the data area of its symbol table). In the grammar beside, we see that when a new record type is recognized, a new **`Env`** gets created.

$T \rightarrow$ record { { *Env.push* (top) ; top = *new Env* (top) ; *Storage.push* (offset) ; offset = 0 }
D } { *T.type* = *record* (top) ; *T.width* = offset ; top = *Env.pop* ( ) ; offset = *Storage.pop* ( ) }

Translation of assignment statements → 2 functions:

- **`gen(3-address instruction)`** = constructs a 3-address instruction and appends it to the sequence generated
- **`top.get(id.lexeme)`** = retrieves the entry for `id.lexeme` in the data area of the current (`top`) symbol table

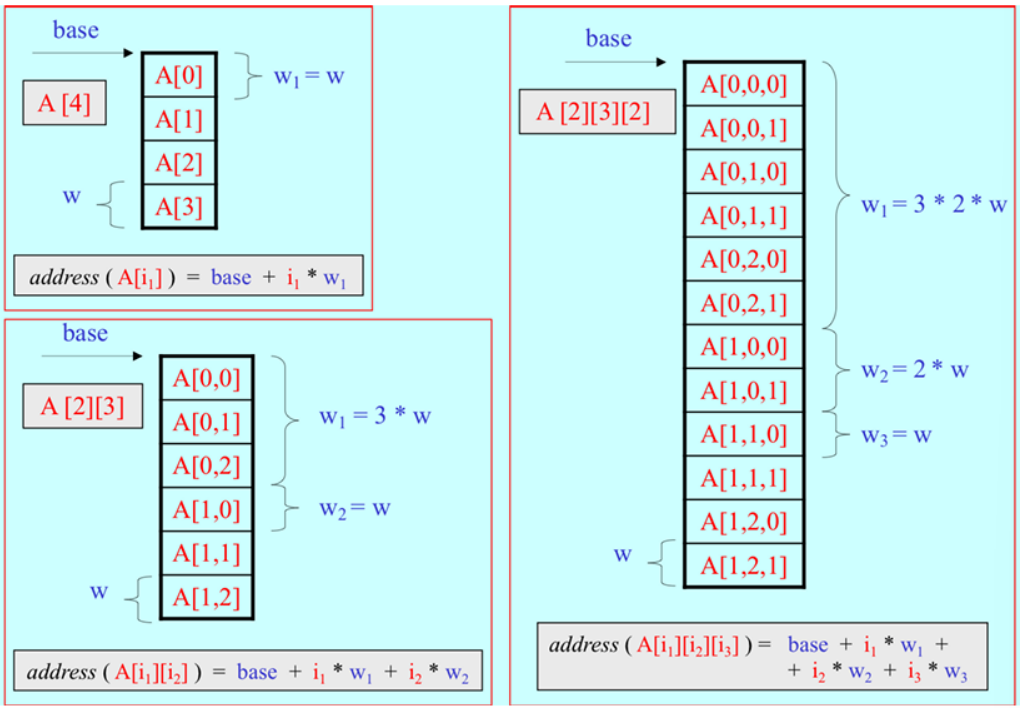
$S \rightarrow$ id = *E* ; { *gen* (*top.get* (id *.lexeme*) "=" *E .addr*) }
$E \rightarrow E_1 + E_2$ { *E .addr* = *new Temp* ( ) ; *gen* (*E .addr* "=" E_1 *.addr* "+" E_2 *.addr*) }
| - E_1 { *E .addr* = *new Temp* ( ) ; *gen* (*E .addr* "=" "minus" E_1 *.addr*) }
| (E_1) { *E .addr* = E_1 *.addr* }
| id { *E .addr* = *top.get* (id *.lexeme*) }



ARRAY:

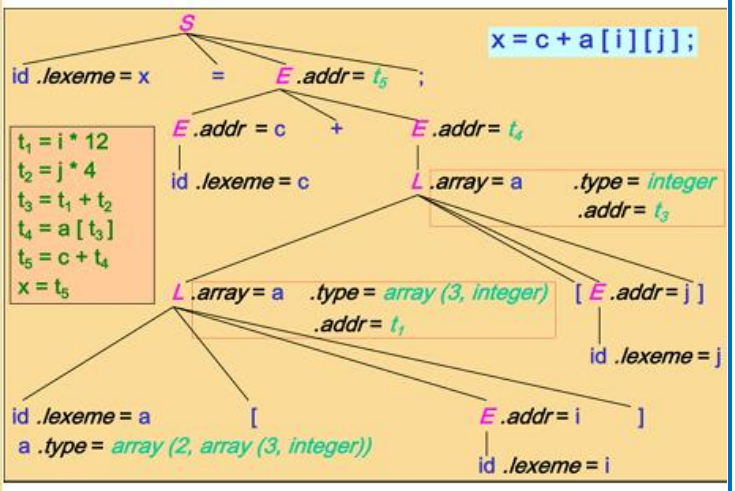
- **Addressing** array elements → $address(A[i_1][i_2] \dots [i_k]) = base + i_1 * w_1 + i_2 * w_2 + \dots + i_k * w_k$; *i* = index; w_k = calculated as follows:

$$\begin{cases} for\ 1 \le j \le k-1 \rightarrow w_j = n_{j+1} * n_{j+2} * \dots * n_k * w \\ for\ j = k \rightarrow w_k = w \end{cases}$$



- **Translating** array references → we need to add some attributes:
  - **`L.addr`** = sum of the $i_j * w_j$ in the address computation
  - **`L.array`** = pointer to the symbol-table entry for the array (es. `int A[2][3]`)
    - **`L.array.base`** = base address of the array
    - **`L.array.type`** (es. `array(2, array(3, int))`)
      - **`L.array.type.elem`** = type of the array elements (es. `array(3, int)`)
  - **`L.type`** = type of the sub-array generated by L
    - **`L.type.width`** = width of the sub-array generated by L (es. w_1, w_2, w_3 depends on the context)
    - **`L.type.elem`** = type of elements of sub-array generated by L (es. `array(2, array(3, int))` oppure `array(3, int)` oppure `int` a seconda del livello interno)

```
S → id = E ;        { gen ( top.get ( id .lexeme ) "=" E .addr ) }
  | L = E ;         { gen ( L .array.base "[" L .addr "]" "=" E .addr ) }
E → E1 + E2         { E .addr = new Temp ( ) ;
                      gen ( E .addr "=" E1 .addr "+" E2 .addr ) }
  | id              { E .addr = top.get ( id .lexeme ) }
  | L               { E .addr = new Temp ( ) ;
                      gen ( E .addr "=" L .array.base "[" L .addr "]" ) }
L → id [ E ]        { L .array = top.get ( id .lexeme ) ;
                      L .type = L .array.type.elem ;
                      L .addr = new Temp ( ) ;
                      gen ( L .addr "=" E .addr "*" L .type.width ) }
  | L1 [ E ]        { L .array = L1 .array ;
                      L .type = L1 .type.elem ;
                      L .addr = new Temp ( ) ;
                      t = new Temp ( ) ;
                      gen ( t "=" E .addr "*" L .type.width )
                      gen ( L .addr "=" L1 .addr "+" t ) }
```



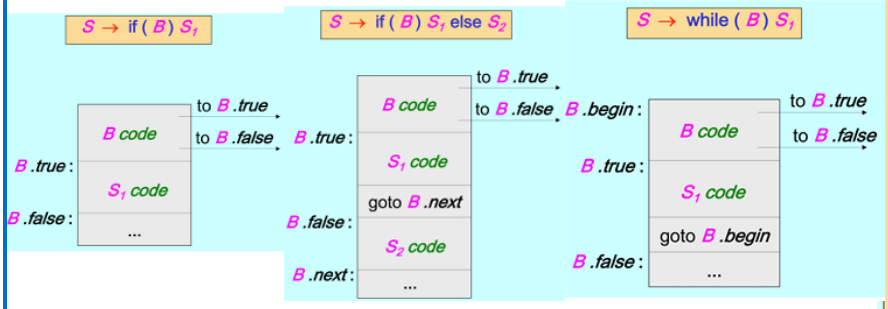
Type Conversion → types are a ***widen hierarchy*** where every type is extension of the other. 2 functions:

- `widen(a,t,w)` generates type conversions to widen an address `a` of type `t` into an address of type `w`
- `max(t1,t2)` returns the max of 2 types in the widening hierarchy


```
Addr widen ( Addr a , Type t , Type w ) ;
  if ( t = w ) return a
  else if ( t = integer and w = float )
       { temp = new Temp ( ) ;
         gen (temp "=" float (a));
         return temp }
  else if ...
  else error
```

```
E → E1 + E2  { E .type = max ( E1 .type , E2 .type ) ;
               a1 = widen ( E1 .addr , E1 .type , E .type ) ;
               a2 = widen ( E2 .addr , E2 .type , E .type ) ;
               E .addr = new Temp ( ) ;
               gen ( E .addr "=" a1 "+" a2 ) }
```

Translation of flow control statements → we need to build the intermediate representation with the approach that we use in assembler: by assigning a label to a significant line and referencing to it when we do a `goto`

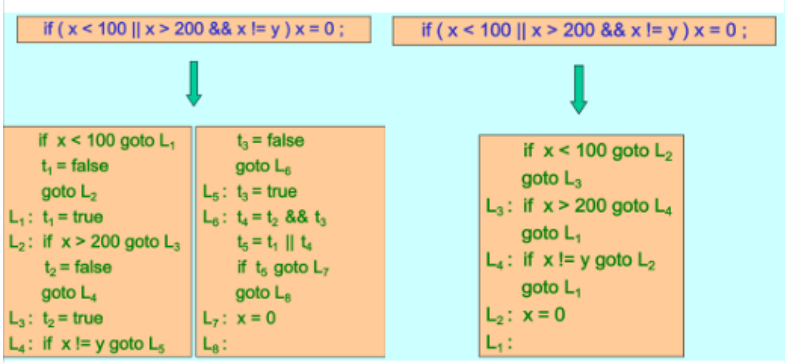



```
S → id = E ;        { gen ( top.get ( id .lexeme ) "=" E .addr ) }

S → S S

S → if (            { B .true = new Label ( ) ; B .false = new Label ( ) }
    B )             { gen ( B .true ) }
    S               { gen ( B .false ) }

S → if (            { B .true = new Label ( ) ; B .false = new Label ( ) ;
    B .next = new Label ( ) }
    B )             { gen ( B .true ) }
    S else          { gen ( "goto" B .next ) ; gen ( B .false ) }
    S               { gen ( B .next ) }

S → while (         { B .begin = new Label ( ) ; B .true = new Label ( ) ;
    B .false = new Label ( ) ; gen ( B .begin ) }
    B )             { gen ( B .true ) }
    S               { gen ( "goto" B .begin ) ; gen ( B .false ) }
```

Translation of boolean expressions:

- **AND** (`&&`) and **OR** (`||`) are left associative
- **NOT**(`!`) has precedence on **AND**, which has precedence on **OR**

```
B → B || B | B && B | ! B | ( B ) | E rel E | true | false

rel.op ∈ { < , <= , == , != , > , >= }
```




```
B →          { B1 .true = B .true ; B1 .false = new Label ( ) }
   B1 ||     { B2 .true = B .true ; B2 .false = B .false ; gen ( B1 .false ) }
   B2

B →          { B1 .true = new Label ( ) ; B1 .false = B .false }
   B1 &&     { B2 .true = B .true ; B2 .false = B .false ; gen ( B1 .true ) }
   B2

B → !        { B1 .true = B .false ; B1 .false = B .true }
   B1

B → E1 rel E2  { gen ( "if " E1 .addr rel.op E2 .addr "goto" B .true ) ;
                 gen ( "goto" B .false ) }

B → true     { gen ( "goto" B .true ) }

B → false    { gen ( "goto" B .false ) }
```

⚠️ The semantic definition of the programming language determines what part of an expression evaluate (es. if we have `(true || a>1)` the 2nd expression should not be evaluated)

BACKPATCHING → jump instructions must often be **generated before** the jump target has been determined (*forward references*). If **labels** **`B.true`** and **`B.false`** are passed as inherited attributes, a 2nd stage of the translation is needed to **bind** labels to instruction addresses.
A complementary approach (**BACKPATCHING**) passes lists of jumps **`B.truelist`** and **`B.falselist`** as synthesized attributes.
When a jump to an undetermined target is generated, the target of the jump is **temporarily left unspecified** and the jump is put on a **list of jumps** having the same target (jump instructions in the list will be completed when the proper target can be determined). Different **functions** used in this approach:

- **`makelist(i)`** creates a list of jumps containing only the index **`i`** into the sequence of instructions; returns a pointer to the new created list
- **`merge(p1,p2)`** concatenates the lists pointed by **`p1`** and **`p2`**; returns a pointer to the concatenated list
- **`backpatch(p,i)`** inserts **`i`** as the target label for each of the instructions on the list pointed by **`p`**

Example (backpatching for **boolean expressions**):

$B \rightarrow B_1 \,||\, M\, B_2$ { *backpatch* (B_1 .*falselist* , M .*instr*) ;
B .*truelist* = *merge* (B_1 .*truelist* , B_2 .*truelist*) ;
B .*falselist* = B_2 .*falselist* }

$B \rightarrow B_1$ && $M\, B_2$ { *backpatch* (B_1 .*truelist* , M .*instr*) ;
B .*truelist* = B_2 .*truelist* ;
B .*falselist* = *merge* (B_1 .*falselist* , B_2 .*falselist*) }

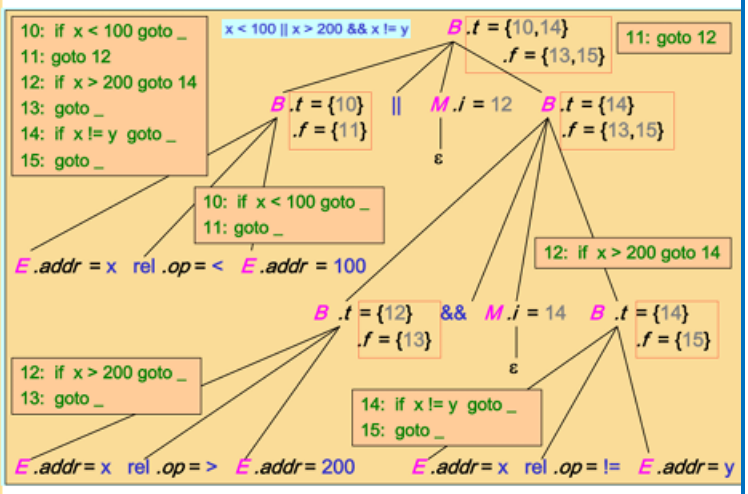
$B \rightarrow$! B_1 { B .*truelist* = B_1 .*falselist* ;
B .*falselist* = B_1 .*truelist* }

$B \rightarrow E_1$ rel E_2 { B .*truelist* = *makelist* (nextinstr) ;
B .*falselist* = *makelist* (nextinstr + 1) ;
gen ("if " E_1 .*addr* rel.*op* E_2 .*addr* "goto _") ;
gen ("goto _") }

$B \rightarrow$ true { B .*truelist* = *makelist* (nextinstr) ;
gen ("goto _") }

$B \rightarrow$ false { B .*falselist* = *makelist* (nextinstr) ;
gen ("goto _") }

$M \rightarrow \varepsilon$ { M .*instr* = nextinstr }



Example (backpatching for **flow control statements**):

$S \rightarrow$ if (B) $M\, S_1$ { *backpatch* (B .*truelist* , M .*instr*) ;
S .*nextlist* = *merge* (B .*falselist* , S_1 .*nextlist*) }

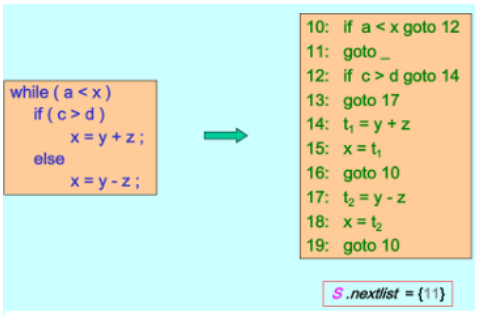
$S \rightarrow$ if (B) $M_1\, S_1\, N$ else $M_2\, S_2$ { *backpatch* (B .*truelist* , M_1 .*instr*) ;
backpatch (B .*falselist* , M_2 .*instr*) ;
temp = *merge* (S_1 .*nextlist* , N .*nextlist*) ;
S .*nextlist* = *merge* (temp , S_2 .*nextlist*) }

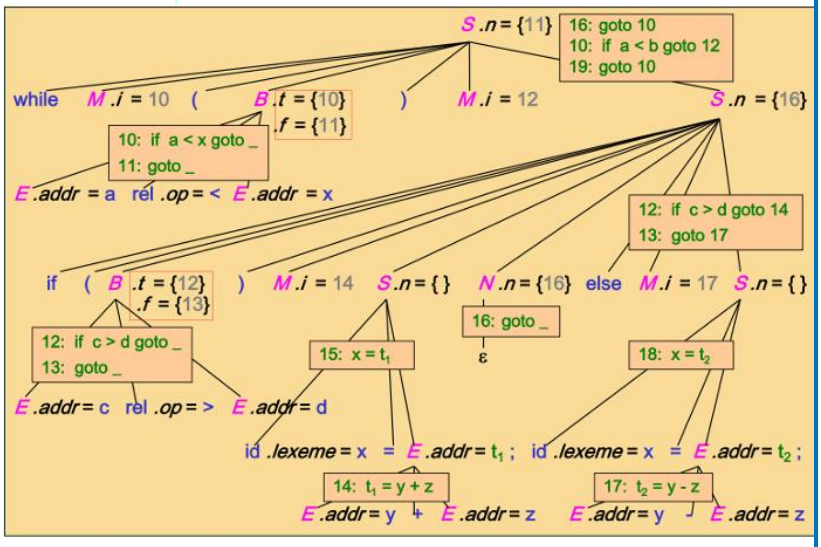
$S \rightarrow$ while M_1 (B) $M_2\, S_1$ { *backpatch* (S_1 .*nextlist* , M_1 .*instr*) ;
backpatch (B .*truelist* , M_2 .*instr*) ;
S .*nextlist* = B .*falselist* ;
gen ("goto" M_1 .*instr*) }

$S \rightarrow \{ L \}$ { S .*nextlist* = L .*nextlist* }
$S \rightarrow$ id = E ; { S .*nextlist* = *null* ;
gen (*top.get* (id .*lexeme*) "=" E .*addr*) }

$L \rightarrow L_1\, M\, S$ { *backpatch* (L_1 .*nextlist* , M .*instr*) ;
L .*nextlist* = S .*nextlist* }

$L \rightarrow S$ { L .*nextlist* = S .*nextlist* }
$M \rightarrow \varepsilon$ { M .*instr* = nextinstr }
$N \rightarrow \varepsilon$ { N .*nextlist* = *makelist* (nextinstr) ;
gen ("goto _") }





FINE.