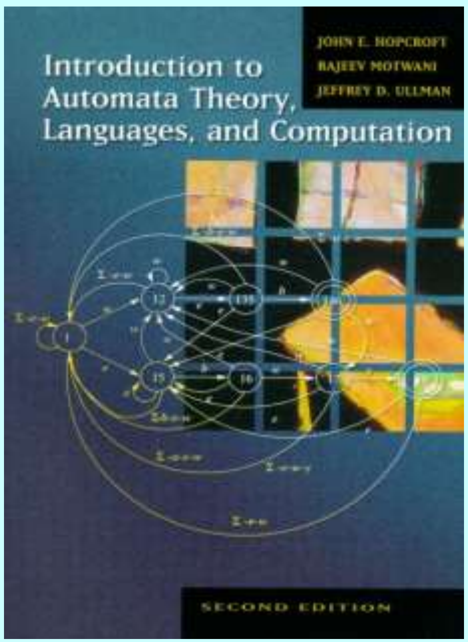
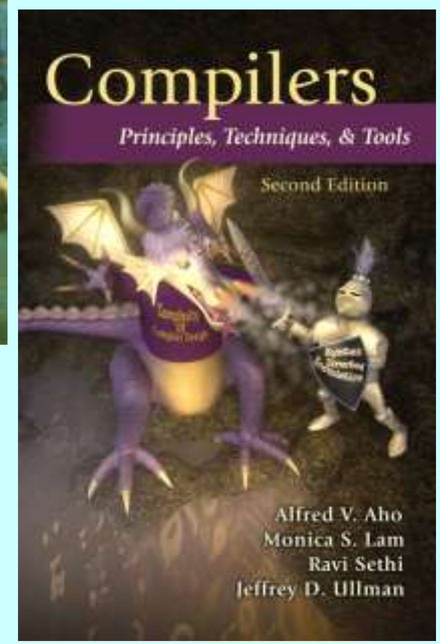
Formal Languages and Compilers



Pieter Bruegel the Elder, *The Tower of Babel*, 1563



http://staff.polito.it/silvano.rivoira | silvano.rivoira@polito.it

FL&C

Cambridge Learner's Dictionary

Definition

language noun

1 COMMUNICATION:

communication between people, usually using words

…She has done research into how children acquire language…

2 ENGLISH/SPANISH/JAPANESE ETC:

a type of communication used by the people of a particular country

…How many languages do you speak?...

3 TYPE OF WORDS:

words of a particular type, especially the words used by people in a particular job

…legal language…technical language…philosophical language…

…pictorial language…the language of business…the language of music…

4 COMPUTERS:

a system of instructions that is used to write computer programs

…Java and Perl are computer programming languages…

See also:

body language, modern languages, second language, sign language

FL&C

Table of Contents

References

- Books
  - J.E. Hopcroft, R. Motwani, J.D. Ullman : *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 2007
    - http://www-db.stanford.edu/~ullman/ialc.html

    (Italian Ed. : *Automi, linguaggi e calcolabilità*, Addison-Wesley, 2009)
    - https://www.pearson.it/opera/pearson/0-6576-automi_linguaggi_e_calcolabilita
  - A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman : *Compilers: Principles, Techniques, and Tools - 2/E*, Addison-Wesley, 2007.
    - http://vig.pearsoned.co.uk/catalog/academic/product/0,1144,0321491696,00.html

    (Italian Ed. : *Compilatori: Principi, tecniche e strumenti – 2/Ed*, Addison-Wesley, 2009)
    - https://www.pearson.it/opera/pearson/0-3479-compilatori
- Development Tools
  - *JFlex* – Scanner generator in Java
    - http://jflex.de/
  - *CUP* – Parser generator in Java
    - http://www2.cs.tum.edu/projects/cup/

Formal Language Classification: definitions (1)

- Alphabet
  - Finite (non-empty) set of symbols
    - $\Sigma_1 = \{0, 1\}$ the set of symbols in binary codes
    - $\Sigma_2 = \{\alpha, \beta, \gamma, \ldots, \omega\}$ the set of lower-case letters in Greek alphabet
    - $\Sigma_3 =$ the set of all ASCII characters
    - $\Sigma_4 =$ **{boy, girl, talks, the, ...}** a set of English terms
- String (word)
  - Finite sequence of symbols chosen from some alphabet
    - $s_1 = 0110001$; $s_2 = \delta\varepsilon\lambda\chi\pi\lambda$; $s_3 =$ **f7$1°Zp](*è** ; $s_4 =$ **the boy talks**
- Length of a string
  - Number of positions for symbols in the string
    - $| 0110001 | = 7$
- Empty string (ε)
  - String of length zero
    - $| \varepsilon | = 0$

FLC: definitions (2)

Alphabet closure

- The set of all strings over an alphabet
  - closure operator (Kleene): *
    - $\{0, 1\}^* = \{\varepsilon , 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$
  - positive closure operator : $^+$
    - $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$
    - $\{0, 1\}^+ = \{0, 1, 00, 01, 10, 11, 000, 001, \dots\}$

Language

- A set of strings over a given alphabet
- $L \subseteq \Sigma^*$
  - $L_1 = \{ 0^n 1^n \mid n \geq 1\} = \{01, 0011, 000111, \dots\}$
  - $L_2 = \{\varepsilon\}$
  - $L_3 = \varnothing$

FLC: grammars (1)

- A grammar is a 4-tuple $G = (N, T, P, S)$
  - N : alphabet of **non-terminal** symbols
  - T : alphabet of **terminal** symbols
    - $N \cap T = \varnothing$
    - $V = N \cup T$: alphabet (vocabulary) of the grammar
  - P : finite set of **rules (productions)**
    - $P = \{ \alpha \rightarrow \beta \mid \alpha \in V^+ ; \alpha \notin T^+ ; \beta \in V^* \}$
  - S : **start** (non-terminal) symbol
    - $S \in N$

FLC: grammars (2)

- Derivation

  let $\alpha \rightarrow \beta$ be a production of G

  - if $\sigma = \gamma\alpha\delta$ and $\tau = \gamma\beta\delta$

    then $\sigma \Rightarrow \tau$ (σ *produces* τ , τ *is derived from* σ)
  - if $\sigma_0 \Rightarrow \sigma_1 \Rightarrow \sigma_2 \ldots \Rightarrow \sigma_k$

    then $\sigma_0 \Rightarrow^* \sigma_k$
- Language produced by $G = (N, T, P, S)$
  - $L(G) = \{ w \mid w \in T^* \; ; \; S \Rightarrow^* w \}$
- Grammars that produce the same language are said equivalent

FLC: example of grammar (1)

G = (*N* , *T* , *P* , *S*)

N = { <sentence> , <qualified noun> , <noun> , <pronoun> , <verb> , <adjective> }

T = { the , man , girl , boy , lecturer , he , she , talks , listens , mystifies , tall , thin , sleepy }

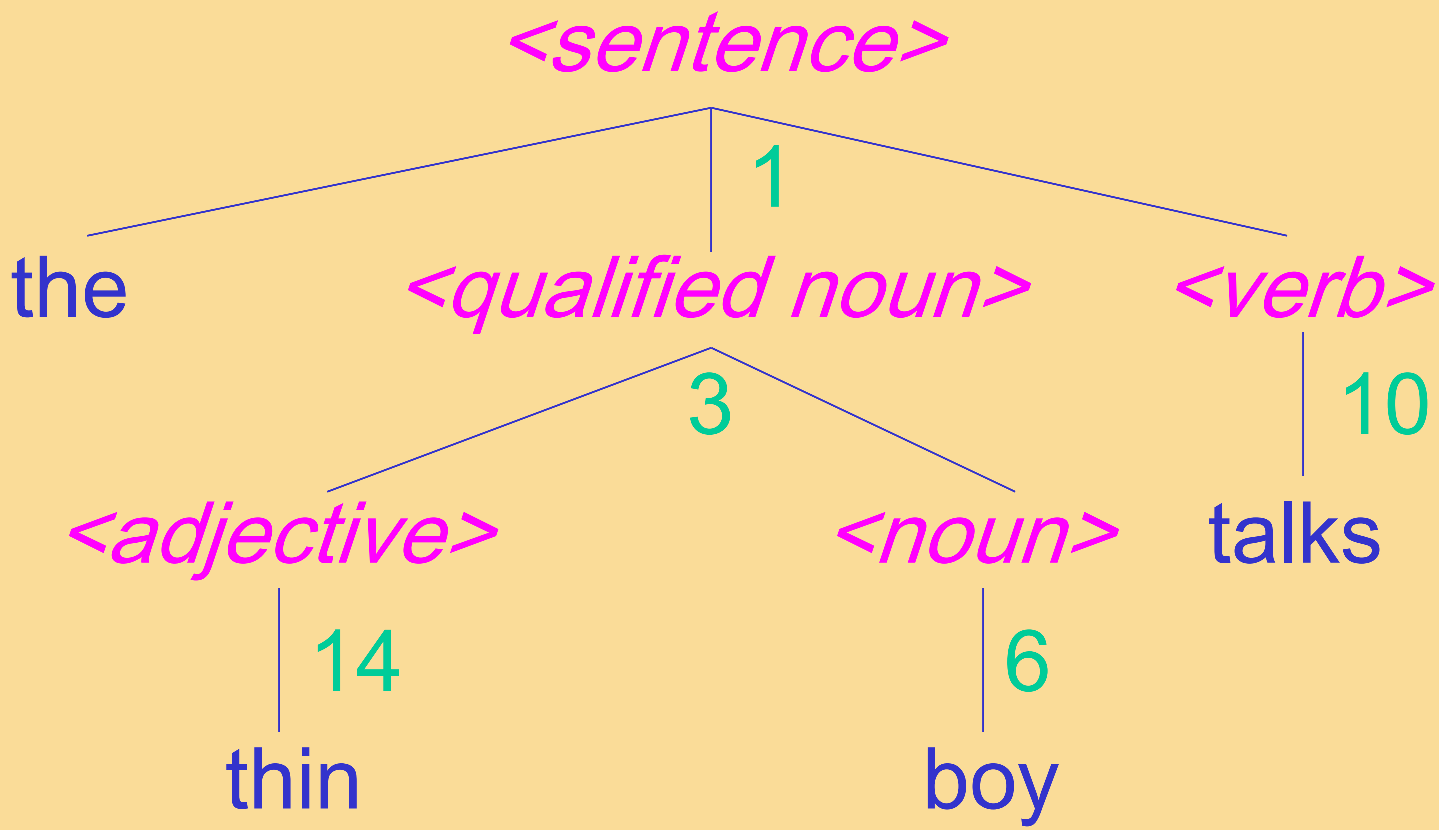
P = {

<sentence>	→ the <qualified noun> <verb>	(1)
	| <pronoun> <verb>	(2)
<qualified noun>	→ <adjective> <noun>	(3)
<noun>	→ man | girl | boy | lecturer	(4, 5, 6, 7)
<pronoun>	→ he | she	(8, 9)
<verb>	→ talks | listens | mystifies	(10, 11, 12)
<adjective>	→ tall | thin | sleepy	(13, 14, 15)

}

S = <sentence>

FLC: example of grammar (2)



<sentence> $\Rightarrow^*$ the thin boy talks

FLC: example of grammar (3)

G = (*N* , *T* , *P* , *S*)

N = { <goal> , <expression> , <term> , <factor> }

T = { a , b , c , - , * }

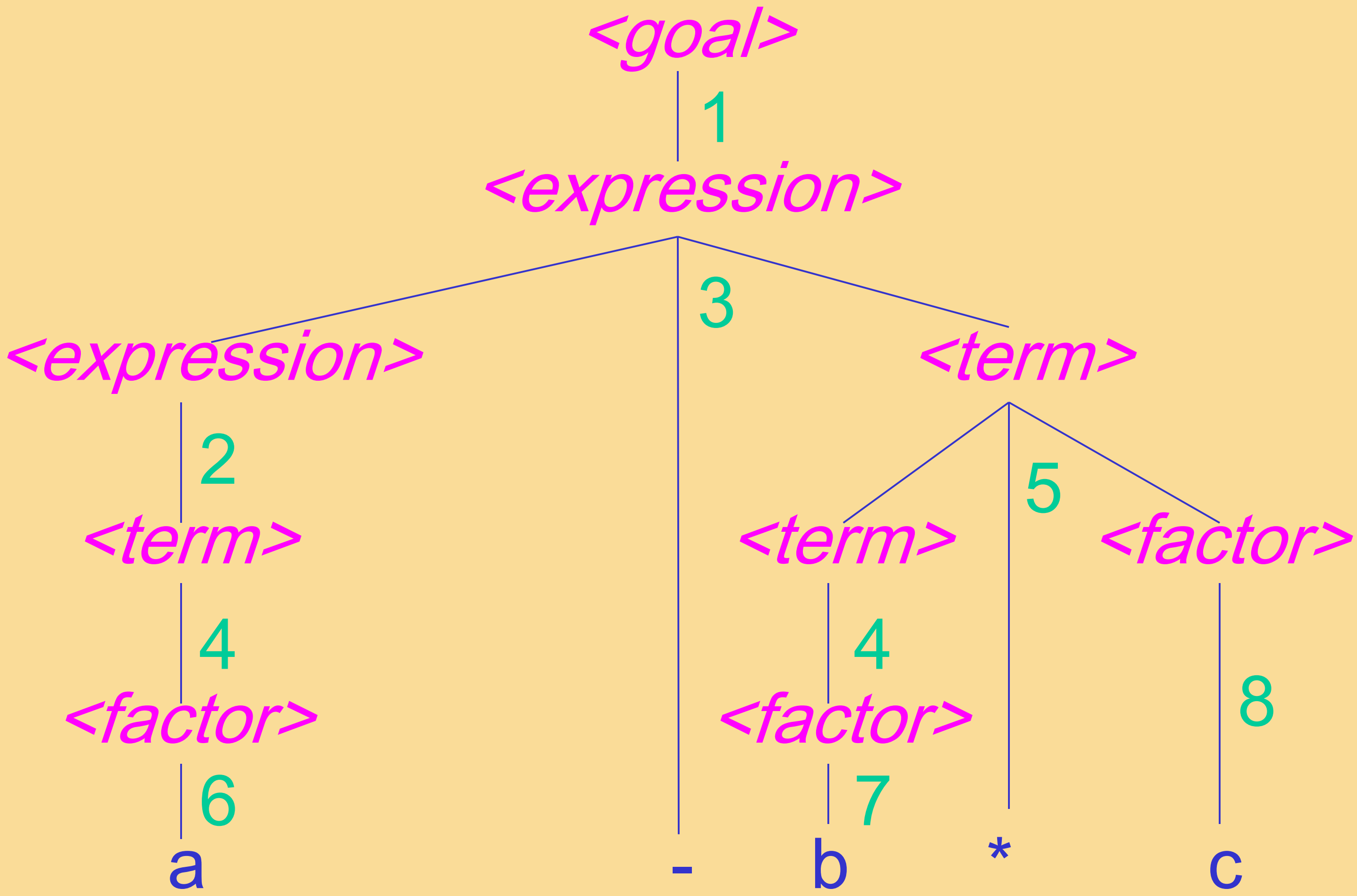
P = {

<goal>	→	<expression>	(1)
<expression>	→	<term> | <expression> - <term>	(2, 3)
<term>	→	<factor> | <term> * <factor>	(4, 5)
<factor>	→	a | b | c	(6, 7, 8)

}

S = <goal>

FLC: example of grammar (4)



<goal> $\Rightarrow^*$ a - b * c

FL&C

FLC: type 0 grammars (phrase-structure)

$$P = \{ \alpha \rightarrow \beta \mid \alpha \in V^+ ;\ \alpha \notin T^+ ;\ \beta \in V^* \}$$

$G = (\{ A , S \} , \{ a , b \} , P , S)$

$P = \{$

S	$\rightarrow$	$a\,A\,b$	(1)
$a\,A$	$\rightarrow$	$a\,a\,A\,b$	(2)
A	$\rightarrow$	ε	(3)

$\}$

$$L(G) = \{ a^n\, b^n \mid n \geq 1 \}$$

$S \rightarrow a\,A\,b \Rightarrow a\,b$

$\Rightarrow a\,a\,A\,b\,b \Rightarrow a\,a\,b\,b$

$\Rightarrow a\,a\,a\,A\,b\,b\,b \Rightarrow a\,a\,a\,b\,b\,b$

$\Rightarrow \ldots$

FLC: type 1 grammars (context-sensitive)

$$P = \{\ \alpha \to \beta \mid \alpha \in V^+;\ \alpha \notin T^+;\ \beta \in V^+;\ |\alpha| \le |\beta|\ \}$$

$G = (\{B, C, S\}, \{a, b, c\}, P, S)$

$P = \{$

S	$\to$	$a\,S\,B\,C \mid a\,b\,C$	(1, 2)
$C\,B$	$\to$	$B\,C$	(3)
$b\,B$	$\to$	$b\,b$	(4)
$b\,C$	$\to$	$b\,c$	(5)
$c\,C$	$\to$	$c\,c$	(6)

$\}$

$$L(G) = \{\ a^n\, b^n\, c^n \mid n \ge 1\ \}$$

FLC: type 2 grammars (context-free) (1)

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

$G = (\{ A, B, S \}, \{ a, b \}, P, S)$

$P = \{ \quad S \to a\,B \mid b\,A$ (1, 2)

$A \to a\,S \mid b\,A\,A \mid a$ (3, 4, 5)

$B \to b\,S \mid a\,B\,B \mid b$ (6, 7, 8)

$\}$

L(G) = the set of strings in $\{ a, b \}^+$ where the number of **"a"** equals the number of **"b"**

FLC: type 2 grammars (context-free) (2)

$G = (\{O, X\}, \{a, +, -, *, /\}, P, X)$

$P = \{ \quad X \rightarrow XXO \mid a$ (1, 2)

$O \rightarrow + \mid - \mid * \mid /$ (3, 4, 5, 6)

$\}$

L(G) = the set of arithmetic expressions with binary operators in **postfix polish notation**

FLC: linear grammars

$$P = \{ A \rightarrow x\,B\,y\,,\ A \rightarrow x \mid A, B \in N\ ;\ x, y \in T^{+} \}$$

$G = (\{ S \}, \{ a, b \}, P, S)$

$P = \{\ S \rightarrow a\,S\,b \mid a\,b$ (1, 2)

$\}$

$L(G) = \{ a^n\, b^n \mid n \geq 1 \}$

FLC: type 3 grammars (right-linear and left-linear)

➢ Right-linear grammars

$$P = \{ A \to x\,B\ ,\ A \to x\ \mid\ A, B \in N\ ;\ x \in T^{+} \}$$

➢ Left-linear grammars

$$P = \{ A \to B\,x\ ,\ A \to x\ \mid\ A, B \in N\ ;\ x \in T^{+} \}$$

FLC: type 3 grammars (right-regular)

$$P = \{ A \rightarrow a\,B ,\ A \rightarrow a \mid A, B \in N \ ;\ a \in T \}$$

$$G = (\{ A, B, C, S \}, \{ a, b \}, P, S)$$

$$P = \{ \quad S \rightarrow a\,A \mid b\,C \qquad (1, 2)$$

$$A \rightarrow a\,S \mid b\,B \mid a \qquad (3, 4, 5)$$

$$B \rightarrow a\,C \mid b\,A \qquad (6, 7)$$

$$C \rightarrow a\,B \mid b\,S \mid b \qquad (8, 9, 10)$$

$$\}$$

L(G) = the set of strings in $\{ a, b \}^+$ where both the number of **"a"** , and the number of **"b"** are even

FLC: type 3 grammars (left-regular)

$$P = \{ A \rightarrow B\,a \,,\, A \rightarrow a \mid A, B \in N \;;\; a \in T \}$$

$G = (\{A, B, C, S\}, \{a, b\}, P, S)$

$P = \{$

$S \rightarrow A\,a \mid S\,a \mid S\,b$ (1, 2, 3)

$A \rightarrow B\,b$ (4)

$B \rightarrow B\,a \mid C\,a \mid a$ (5, 6, 7)

$C \rightarrow A\,b \mid C\,b \mid b$ (8, 9, 10)

$\}$

L(G) = the set of strings in $\{ a, b \}^+$ containing **"a b a"**

FLC: equivalence among type 3 grammars

- ***right-linear*** and ***right-regular*** grammars are equivalent

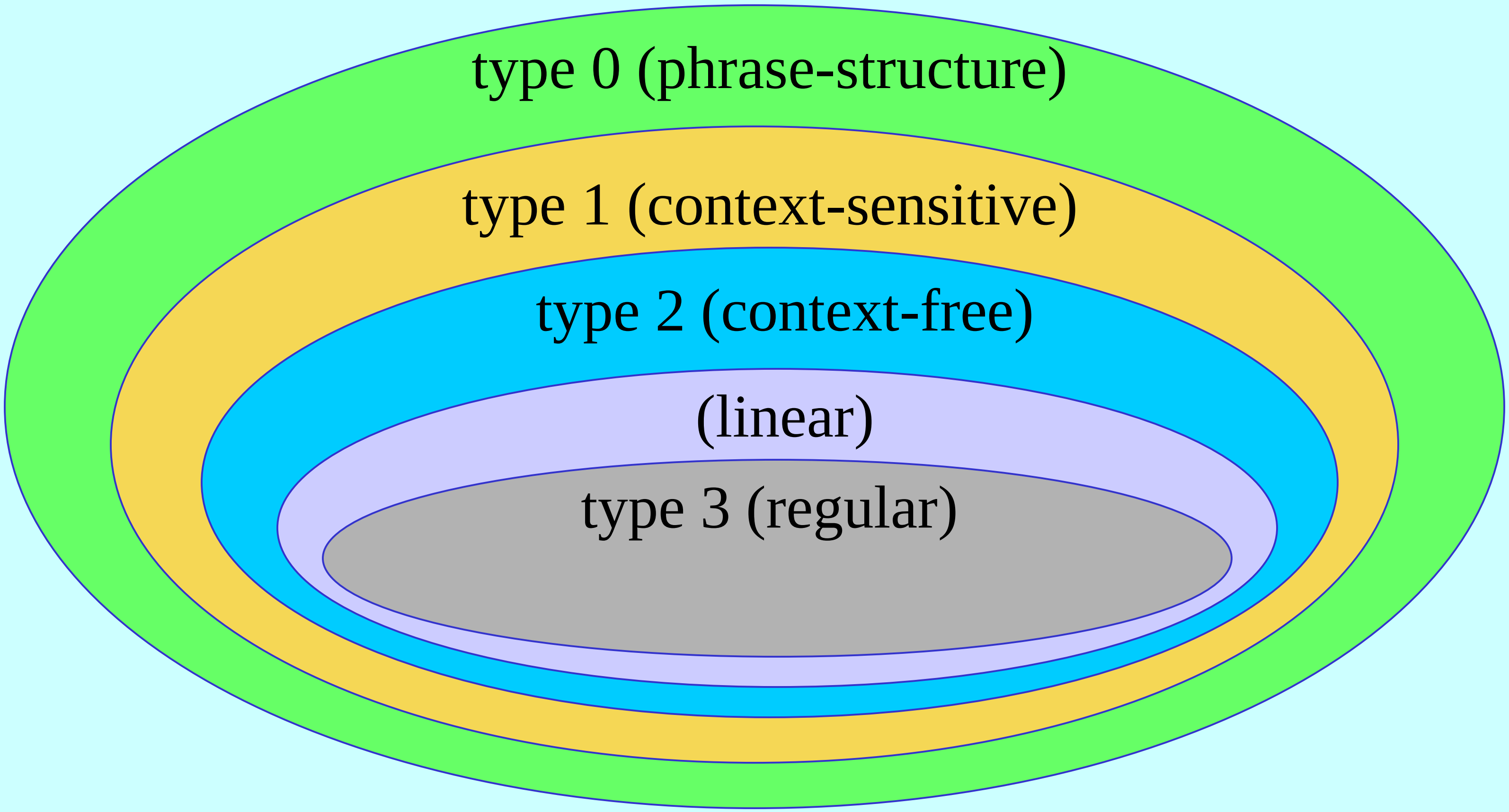
$$A \rightarrow a\,b\,c\,B \;\equiv\; \{A \rightarrow a\,C,\; C \rightarrow b\,c\,B\} \;\equiv\; \{A \rightarrow a\,C,\; C \rightarrow b\,D,\; D \rightarrow c\,B\}$$

- ***left-linear*** and ***left-regular*** grammars are equivalent

$$A \rightarrow B\,a\,b\,c \;\equiv\; \{A \rightarrow C\,c,\; C \rightarrow B\,a\,b\} \;\equiv\; \{A \rightarrow C\,c,\; C \rightarrow D\,b,\; D \rightarrow B\,a\}$$

FLC: language classification (Chomsky hierarchy)

- A *language* is *type-**n*** if it can be produced by a *type-**n** grammar*



Regular Languages: regular sets

- The following sets are ***regular sets*** over an alphabet **Σ**
  - the empty set $\emptyset$
  - the set **{ε}** containing the empty string
  - the set **{a}** containing any symbol **a** $\in$ **Σ**
- If **P** and **Q** are regular sets over **Σ** , the same is true for
  - the union **P** $\cup$ **Q**
  - the concatenation $PQ = \{ xy \mid x \in P ; y \in Q \}$
  - the closures **P*** e **Q***

RL: regular expressions

- The following expressions are ***regular expressions*** over an alphabet **Σ**
  - the expression **φ** , denoting the empty set ∅
  - the expression **ε** , denoting the set **{ε}**
  - the expression **a** , denoting the set **{a}** where **a ∈ Σ**
- If **p** and **q** are regular expressions denoting the sets **P** and **Q** , then also the following are regular expressions
  - the expression **p | q** , denoting the set **P ∪ Q**
  - the expression **p q** , denoting the set **P Q**
  - the expressions **p*** e **q*** , denoting the sets **P*** e **Q***

RL: examples of regular expressions

- the set of strings over **{0,1}** containing two **1**'s
  - **0*10*10***
- the strings over **{0,1}** without consecutive equal symbols
  - **(1 | ε) (01)* (0 | ε)**
- the set of decimal characters
  - **digit = 0 | 1 | 2 | … | 9**
- the set of strings representing decimal integers
  - **digit digit***
- the set of alphabetic characters
  - **letter = A | B | … | Z | a | b | … | z**
- the set of strings representing identifiers
  - **letter (letter | digit)***

RL: regular sets / regular expressions

René Magritte, *This is not a pipe,* 1948

RL: algebraic properties of regular expressions

- two **regular expressions** are ***equivalent*** if they denote the same **regular set**
- $\alpha \mid \beta = \beta \mid \alpha$ (commutative property)
- $\alpha \mid (\beta \mid \gamma) = (\alpha \mid \beta) \mid \gamma$ (associative property)
- $\alpha (\beta \gamma) = (\alpha \beta) \gamma$ (associative property)
- $\alpha (\beta \mid \gamma) = \alpha \beta \mid \alpha \gamma$ (distributive property)
- $(\alpha \mid \beta) \gamma = \alpha \gamma \mid \beta \gamma$ (distributive property)
- $\alpha \mid \varphi = \alpha$
- $\alpha \varepsilon = \varepsilon \alpha = \alpha$
- $\alpha \varphi = \varphi \alpha = \varphi$
- $\alpha \mid \alpha = \alpha$
- $\varphi^* = \varepsilon^* = \varepsilon$
- $\alpha^* = \alpha^* \alpha^* = (\alpha^*)^* = \alpha \alpha^* \mid \varepsilon$

RL: equations of regular expressions

- if α e β are regular expressions, $X = \alpha X \mid \beta$ is an equation with unknown X
- $X = \alpha^* \beta$ is a solution of the equation
  - $\alpha X \mid \beta = \alpha \alpha^* \beta \mid \beta = (\alpha \alpha^* \mid \varepsilon) \beta = \alpha^* \beta = X$
- a set of equations with unknowns $\{X_1, X_2, \dots, X_n\}$ is composed by **n** equations such as:

$$X_i = \alpha_{i0} \mid \alpha_{i1} X_1 \mid \alpha_{i2} X_2 \mid \dots \mid \alpha_{in} X_n$$

where each α_{ij} is a regular expression over any alphabet without the unknowns

FL&C

RL: solution of sets of equations

```
{
 for (int i=1 ; i<n ; i++) {
   put the equation i in the form Xi = α Xi | β;
   substitute Xi with α* β in the equations i+1...n;
 }
 for (int i=n ; i>0 ; i--) {
   // the i-th equation is in the form Xi = α Xi | β
   // where α and β do not contain unknowns
   solve the i-th equation: Xi = α* β;
   substitute Xi with α* β in the equations i-1...1;
 }
}
```

RL: example of solution of sets of equations

$$\begin{cases} A = 1A \mid 0B \\ B = 1A \mid 0C \mid 0 \\ C = 0C \mid 1C \mid 0 \mid 1 \end{cases}$$

$$A = 1^*0B$$

$$B = 11^*0B \mid 0C \mid 0 \quad \Rightarrow \quad B = (11^*0)^*(0C \mid 0)$$

$$C = (0 \mid 1)C \mid 0 \mid 1 \quad \Rightarrow \quad C = (0 \mid 1)^*(0 \mid 1)$$

$$\begin{cases} C = (0 \mid 1)^*(0 \mid 1) \\ B = (11^*0)^*(0(0 \mid 1)^*(0 \mid 1) \mid 0) \\ A = 1^*0(11^*0)^*(0(0 \mid 1)^*(0 \mid 1) \mid 0) \end{cases}$$

RL: right-linear languages $\subseteq$ regular sets

- let $G = (\{A_1, A_2, \ldots, A_n\}, T, P, A_1)$ be a right-linear grammar
- let us transform each rule of the grammar:

$$A_i \rightarrow \alpha_{i0} \mid \alpha_{i1} A_1 \mid \alpha_{i2} A_2 \mid \ldots \mid \alpha_{in} A_n$$

into an equation of regular expressions :

$$A_i = \alpha_{i0} \mid \alpha_{i1} A_1 \mid \alpha_{i2} A_2 \mid \ldots \mid \alpha_{in} A_n$$

- let us solve the resulting set of equations
- the language ***L(G)*** generated by the grammar is denoted by the regular expression corresponding to the symbol A_1

RL: regular expression of a right-linear language

$G = (\{A, B, S\}, \{0,1\}, P, S)$

$$P = \{\ S \rightarrow 0A \mid 1S \mid 0,\quad A \rightarrow 0B \mid 1A,\quad B \rightarrow 0S \mid 1B\ \}$$

$$\Rightarrow \quad \{\ S = 0A \mid 1S \mid 0,\quad A = 0B \mid 1A,\quad B = 0S \mid 1B\ \}$$

$$B = 1^*0S$$

$$A = 1^*0B = 1^*01^*0S$$

$$S = 01^*01^*0S \mid 1S \mid 0 \quad = \quad (01^*01^*0 \mid 1)S \mid 0$$

$S = (01^*01^*0 \mid 1)^*0$ **denotes** $L(G)$

RL: regular sets $\subseteq$ right-linear languages (1)

- the ***regular sets*** **:** $\varnothing$, $\{\varepsilon\}$, $\{ a \mid a \in \Sigma \}$ can be generated by right-linear grammars
  - $G_1 = (\{S\}, \Sigma, \varnothing, S)$ $\Rightarrow$ $L(G_1) = \varnothing$
  - $G_2 = (\{S\}, \Sigma, \{S \rightarrow \varepsilon\}, S)$ $\Rightarrow$ $L(G_2) = \{\varepsilon\}$
  - $G_3 = (\{S\}, \Sigma, \{S \rightarrow a \mid a \in \Sigma\}, S)$

    $\Rightarrow$ $L(G_3) = \{a\}$ where $a \in \Sigma$

RL: regular sets $\subseteq$ right-linear languages (2)

- let $G_1 = (N_1, \Sigma, P_1, S_1)$ and $G_2 = (N_2, \Sigma, P_2, S_2)$ be right-linear grammars where $N_1 \cap N_2 = \varnothing$
- the language $L(G_1) \cup L(G_2) = L(G_4)$ is a right-linear language
  - $G_4 = (N_1 \cup N_2 \cup \{S_4\}, \Sigma, P_1 \cup P_2 \cup \{S_4 \to S_1 \mid S_2\}, S_4)$
- the language $L(G_1)\, L(G_2) = L(G_5)$ is a right-linear language
  - $G_5 = (N_1 \cup N_2, \Sigma, P_2 \cup P_5, S_1)$; $P_5 = \{ A \to x\,B$ if $A \to x\,B \in P_1$

    $A \to x\,S_2$ if $A \to x \in P_1 \}$
- the language $L(G_1)^* = L(G_6)$ is a right-linear language
  - $G_6 = (N_1 \cup \{S_6\}, \Sigma, \{S_6 \to S_1 \mid \varepsilon\} \cup P_6, S_6)$;

    $P_6 = \{ A \to x\,B$ if $A \to x\,B \in P_1$

    $A \to x\,S_6$ if $A \to x \in P_1 \}$

RL: regular sets ≡ right/left-linear languages

- *right-linear languages* ⊆ *regular sets*
- *regular sets* ⊆ *right-linear languages*
- *right-linear languages* ≡ *regular sets*
- *left-linear languages* ≡ *regular sets*
  - the equation of regular expressions **X = X α | β** has the solution **X = β α***
  - it is possible to solve sets of equations corresponding to the rules of a left-linear grammar
  - it is possible to define left-linear grammars that generate any regular set
- *right-linear languages* ≡ *left-linear languages*

RL: regular expression of a left-linear language

$G = (\{U, V, Z\}, \{0,1\}, P, Z)$

$$P = \{ Z \rightarrow U\,0 \mid V\,1,\quad U \rightarrow Z\,1 \mid 0,\quad V \rightarrow Z\,0 \mid 1 \} \quad \Rightarrow \quad \{ Z = U\,0 \mid V\,1,\quad U = Z\,1 \mid 0,\quad V = Z\,0 \mid 1 \}$$

$$Z = (Z\,1 \mid 0)0 \mid (Z\,0 \mid 1)1 =$$
$$= Z\,10 \mid 00 \mid Z\,01 \mid 11 =$$
$$= Z\,(10 \mid 01) \mid 00 \mid 11$$

$Z = (00 \mid 11)\,(10 \mid 01)^*$ **denotes** $L(G)$

RL: deterministic finite automata (DFA)

- A **DFA** is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$
  - Q : finite (non empty) set of **states**
  - Σ : alphabet of **input** symbols
  - δ : **transition** function
    - $\delta : Q \times \Sigma \rightarrow Q$
  - q_0 : **start** state
    - $q_0 \in Q$
  - F : set of **final states**
    - $F \subseteq Q$

RL: an example of DFA

$A = (Q, \Sigma, \delta, q_0, F)$

$Q = \{ q_0, q_1, q_2, q_3 \}$

$\Sigma = \{ 0, 1 \}$

$\delta(q_0, 0) = q_2$	$\delta(q_0, 1) = q_1$
$\delta(q_1, 0) = q_3$	$\delta(q_1, 1) = q_0$
$\delta(q_2, 0) = q_0$	$\delta(q_2, 1) = q_3$
$\delta(q_3, 0) = q_1$	$\delta(q_3, 1) = q_2$

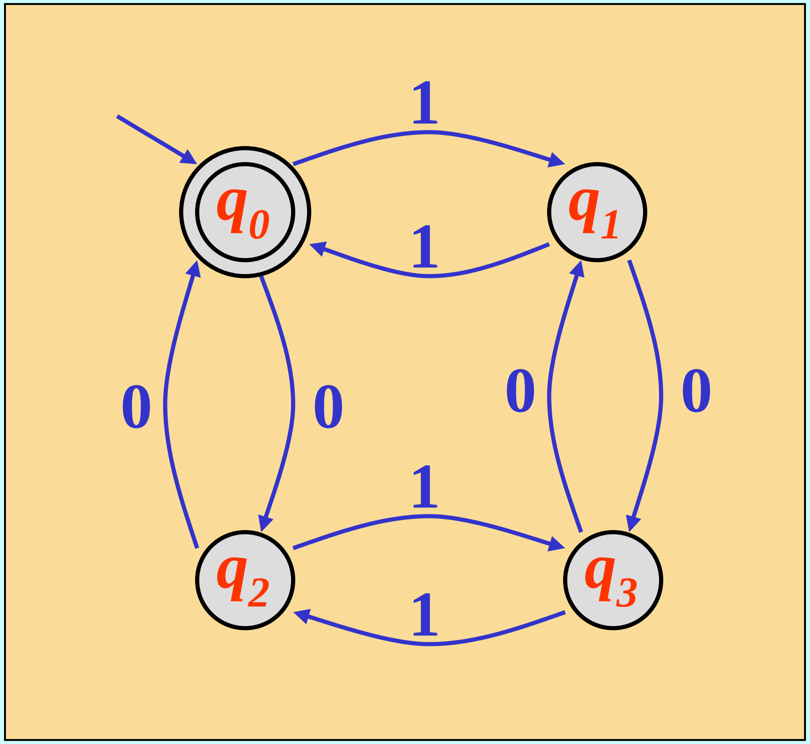
$F = \{ q_0 \}$

RL: transition table / diagram of a DFA

- transition table
  - tabular representation of the transition function
- transition diagram: a *graph* where
  - for each state in the automaton there is a node
  - for each transition $\delta(p, a) = q$ there is an arc from p to q labeled a
  - the start state has an entering non labeled arc
  - the final states are marked by a double circle

RL: representations of a DFA

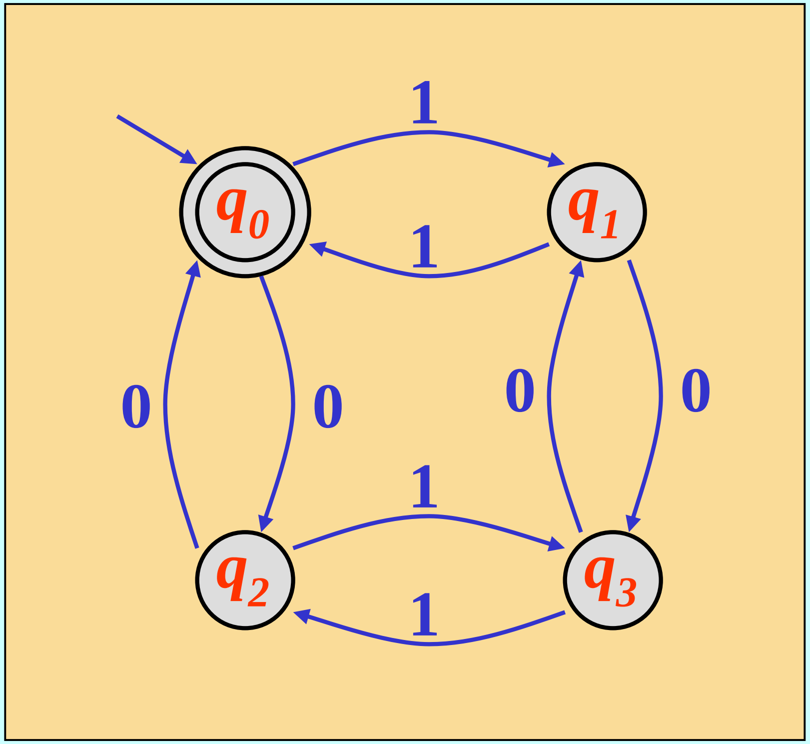
	0	1
$\rightarrow * q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2



RL: the language accepted by a DFA

- The domain of function δ can be extended from $Q \times \Sigma$ to $Q \times \Sigma^*$
  - $\delta(q, \varepsilon) = q$
  - $\delta(q, aw) = \delta(\delta(q, a), w)$ where $a \in \Sigma$; $w \in \Sigma^*$
- Language accepted by $A = (Q, \Sigma, \delta, q_0, F)$
  - $L(A) = \{ w \mid w \in \Sigma^* \ ; \ \delta(q_0, w) \in F \}$

RL: how a DFA accepts strings

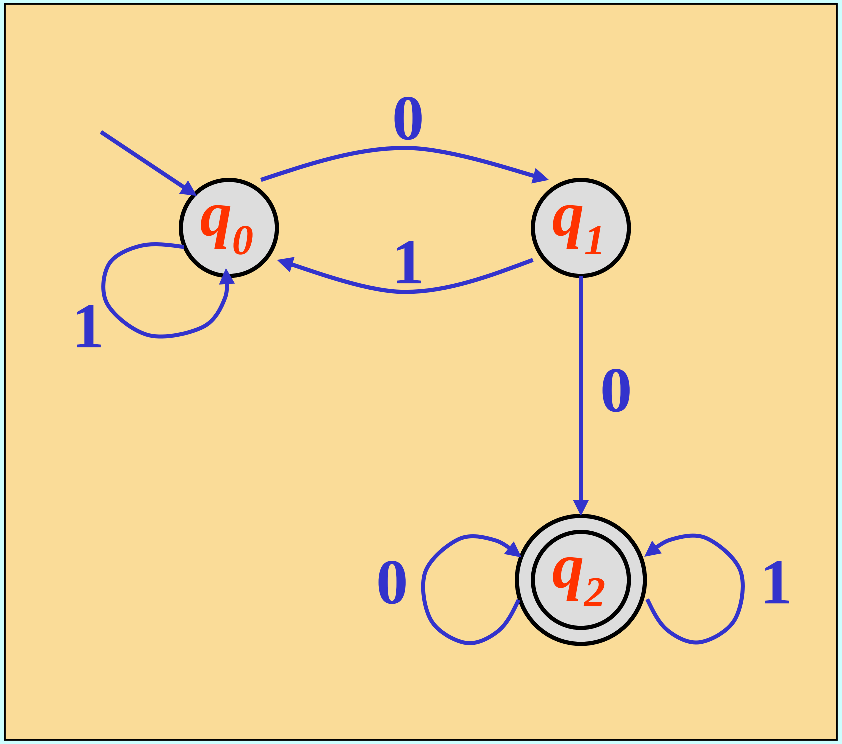


011101 $\in L(A)$

01101 $\notin L(A)$

RL: examples of DFA (1)

- ***L*(*A*) =** the set of all strings over **{0,1}** with at least two consecutive **0**'s

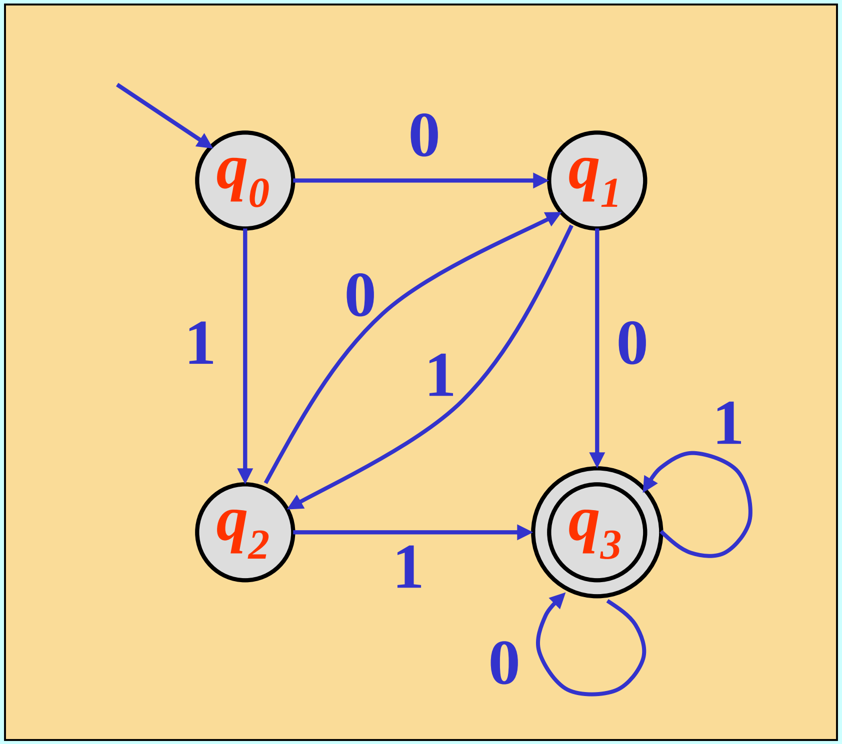


q_0 : strings that do not end in **0**

q_1 : strings that end with only one **0**

RL: examples of DFA (2)

- ***L*(*A*) =** the set of all strings over **{0,1}** with at least two consecutive **0**'s or two consecutive **1**'s



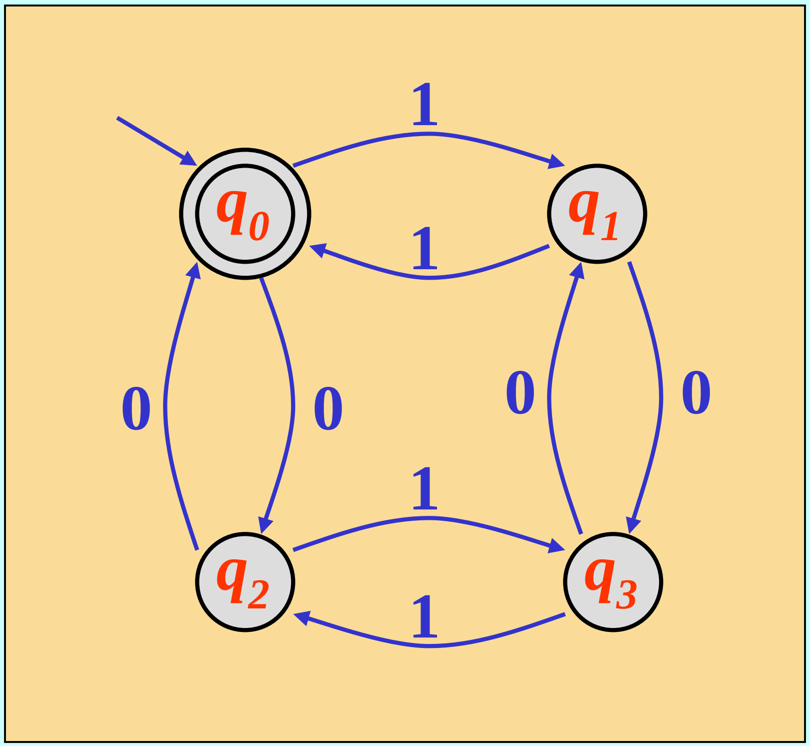
q_0 : strings that do not end in **0** or in **1**

q_1 : strings that end with only one **0**

q_2 : strings that end with only one **1**

RL: examples of DFA (3)

- ***L*(*A*) =** the set of all strings over **{0,1}** having both an even number of **0**'s and an even number of **1**'s



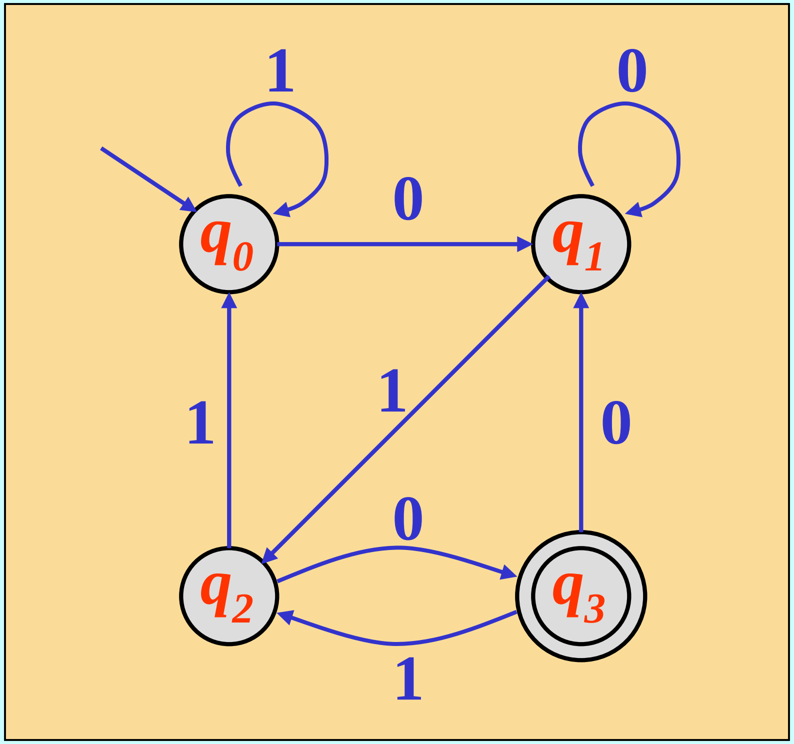
q_1 : strings with even # of **0**'s and odd # of **1**'s

q_2 : strings with odd # of **0**'s and even # of **1**'s

q_3 : strings with odd # of **0**'s and odd # of **1**'s

RL: examples of DFA (4)

- ***L(A)* =** the set of all strings over **{0,1}** ending in " **010** "



q_0 : strings not ending in **0** or in **01**

q_1 : strings ending in **0** but not in **010**

q_2 : strings ending in **01**

RL: examples of DFA (5)

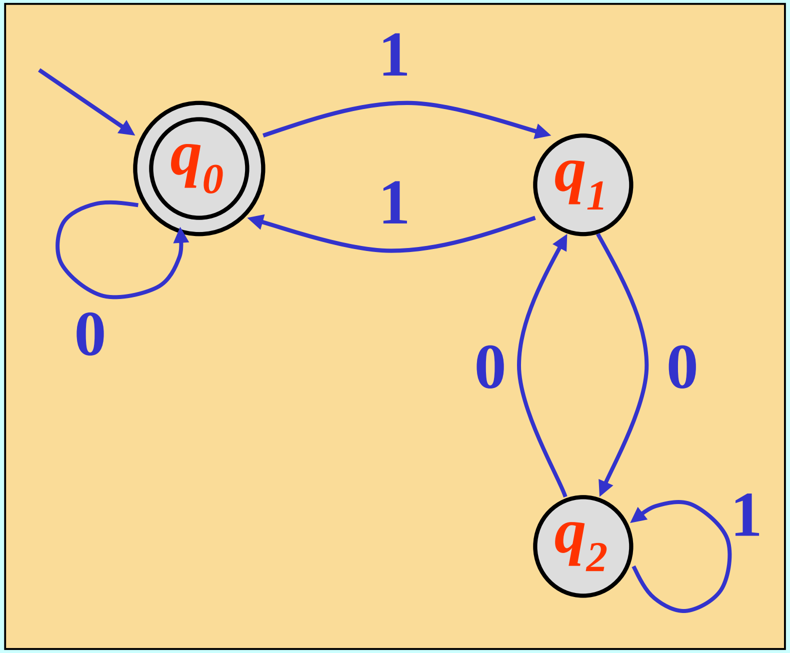
- ***L(A)* =** the set of all strings that represent positive binary integers (***pbi***) multiple of 3

pbi ∈ **{0,1}***

pbi* = (*pbi div **3)** x **3 + (*pbi*** mod **3)**

pbi* 0 := 2** x ***pbi

***pbi* 1 := 2** x ***pbi* +1**



q_0 : integers that give remainder **0** when divided by **3**

q_1 : integers that give remainder **1** when divided by **3**

q_2 : integers that give remainder **2** when divided by **3**

RL: non-deterministic finite automata (NFA)

- An **NFA** is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$
  - Q : finite (non empty) set of **states**
  - Σ : alphabet of **input** symbols
  - δ : **transition** function
    - $\delta: Q \times \Sigma \rightarrow \wp(Q)$
  - q_0 : **start** state
    - $q_0 \in Q$
  - F : set of **final states**
    - $F \subseteq Q$

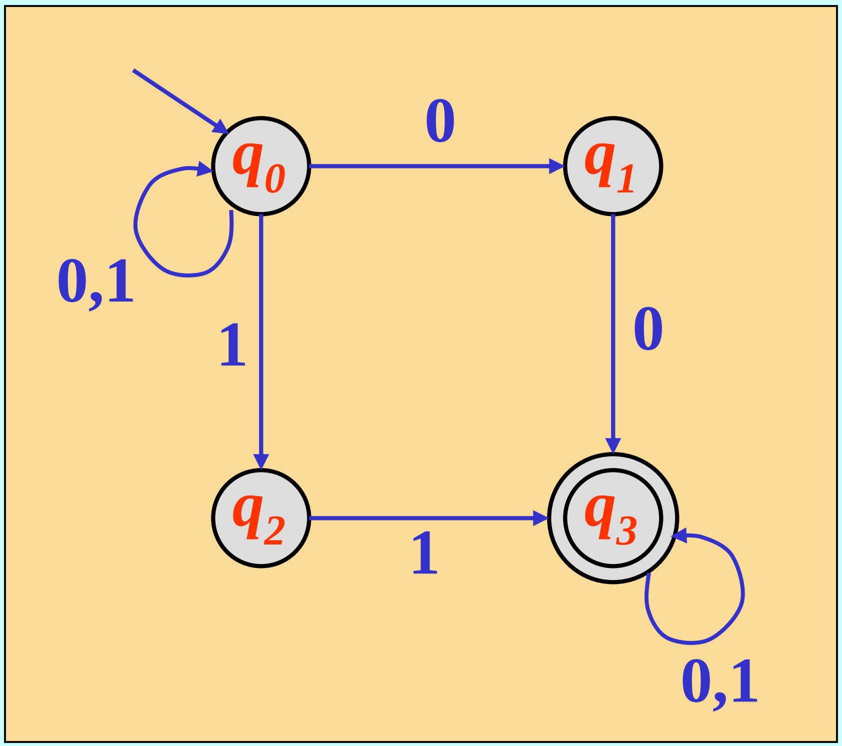
> $\wp(Q)$: *power set* of Q
> **(the set of all subsets)**
> $\|\wp(Q)\| = 2^{\|Q\|}$

RL: the language accepted by an NFA

- The domain of function δ can be extended from $Q \times \Sigma$ to $Q \times \Sigma^*$ to $\wp(Q) \times \Sigma^*$
  - $\delta(q, \varepsilon) = \{q\}$
  - $\delta(q, aw) = \cup_i \delta(p_i, w)$ where $p_i \in \delta(q, a)$
  - $\delta(\{q_1, q_2, \ldots, q_n\}, w) = \cup_j \delta(q_j, w)$
- Language accepted by $A = (Q, \Sigma, \delta, q_0, F)$
  - $L(A) = \{ w \mid w \in \Sigma^* ; \delta(q_0, w) \cap F \neq \varnothing \}$
- a **DFA** is a special case of **NFA**

RL: examples of NFA (1)

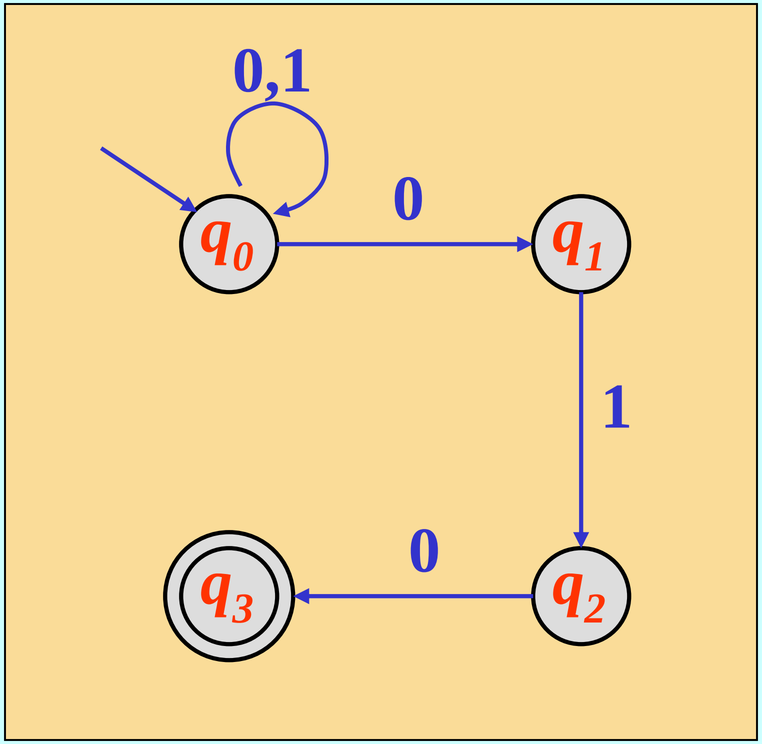
- **$L(A)$ =** the set of all strings over **{0,1}** with at least two consecutive **0**'s or two consecutive **1**'s



(0 | 1)* (00 | 11) (0 | 1)*

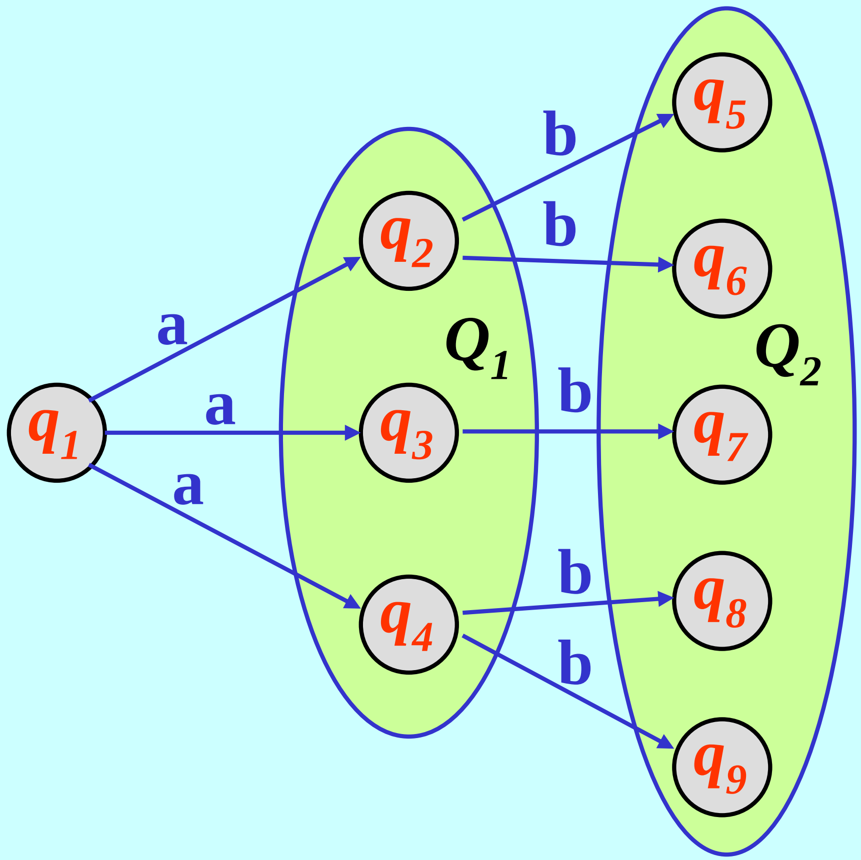
RL: examples of NFA (2)

➢ ***L*(*A*) =** the set of all strings over **{0,1}** ending in " **010** "

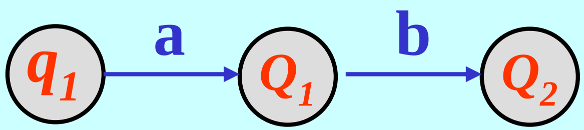


(0 | 1)* 010

RL: equivalence of NFA and DFA (1)



≡



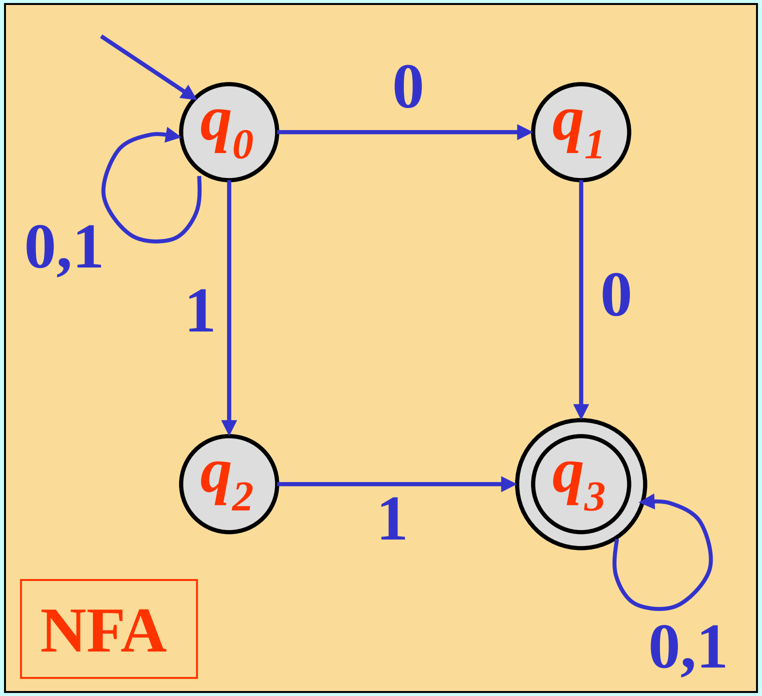
$$Q_1 = \{ q_2 , q_3 , q_4 \}$$

$$Q_2 = \{ q_5 , q_6 , q_7 , q_8 , q_9 \}$$

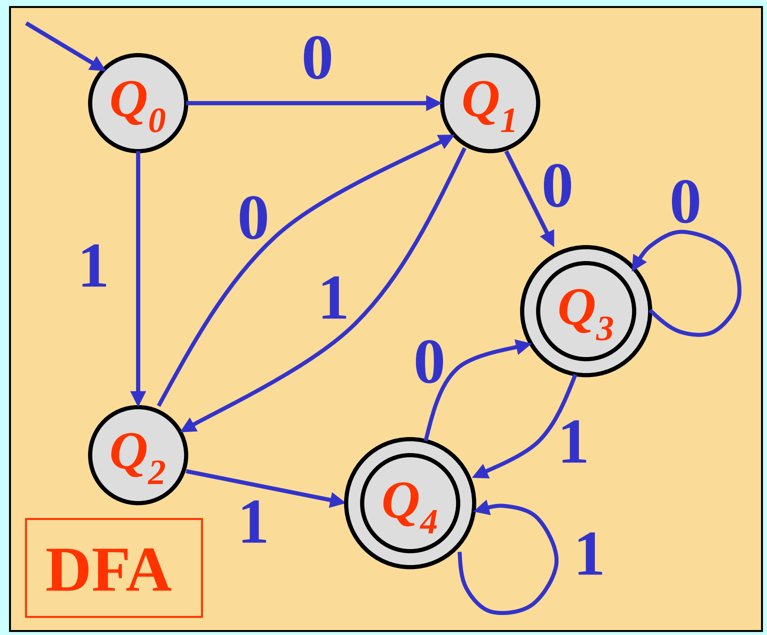
RL: equivalence of NFA and DFA (2)

- let $N = (Q_N , \Sigma, \delta_N , q_0 , F_N)$ be an **NFA**
- let us construct a **DFA** $D = (Q_D , \Sigma, \delta_D , \{ q_0 \} , F_D)$
  - $Q_D \subseteq \wp(Q_N)$
  - $\delta_D(S , a) = \cup_i \delta_N(p_i , a)$ where $p_i \in S \in Q_D$
  - $F_D = \{ S \mid S \in Q_D ; S \cap F_N \neq \varnothing \}$
- by construction $L(D) = L(N)$
- **NFA ≡ DFA** (**FA** : finite automata)

RL: constructing a DFA from an NFA (1)

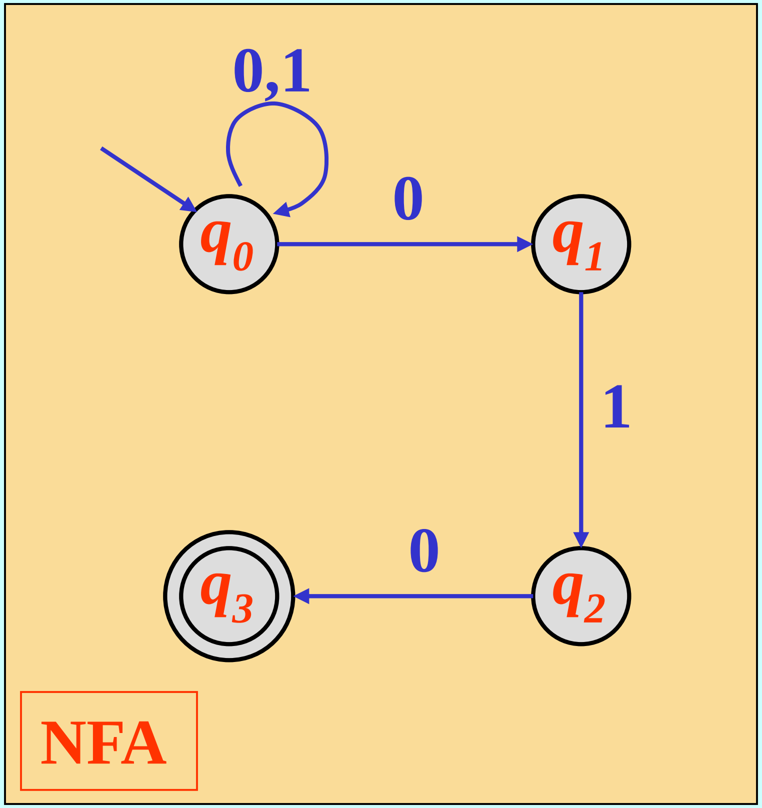


		0	1
Q_0	$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
Q_1	$\{q_0, q_1\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_2\}$
Q_2	$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2, q_3\}$
Q_3	$*\{q_0, q_1, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_2, q_3\}$
Q_4	$*\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_2, q_3\}$

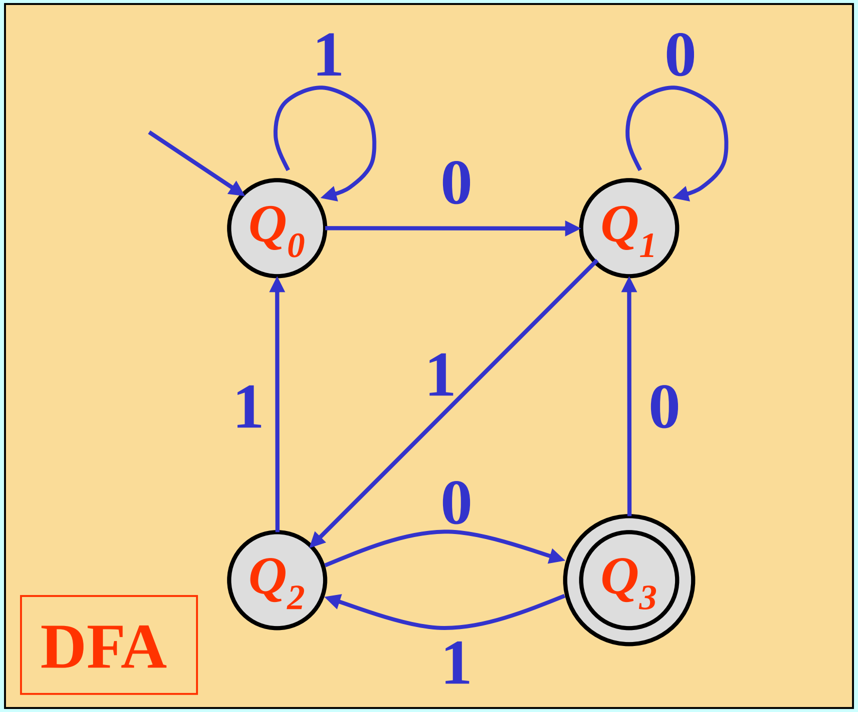


(0 | 1)* (00 | 11) (0 | 1)*

RL: constructing a DFA from an NFA (2)



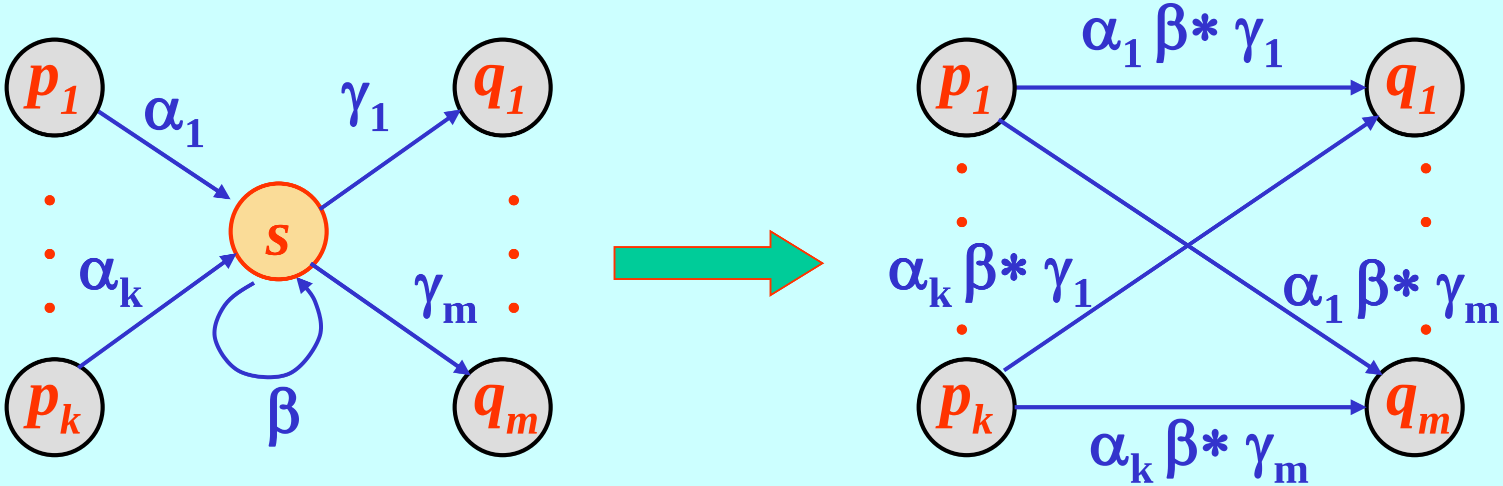
		0	1
Q_0	$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
Q_1	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
Q_2	$\{q_0, q_2\}$	$\{q_0, q_1, q_3\}$	$\{q_0\}$
Q_3	$*\{q_0, q_1, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$



(0 | 1)* 010

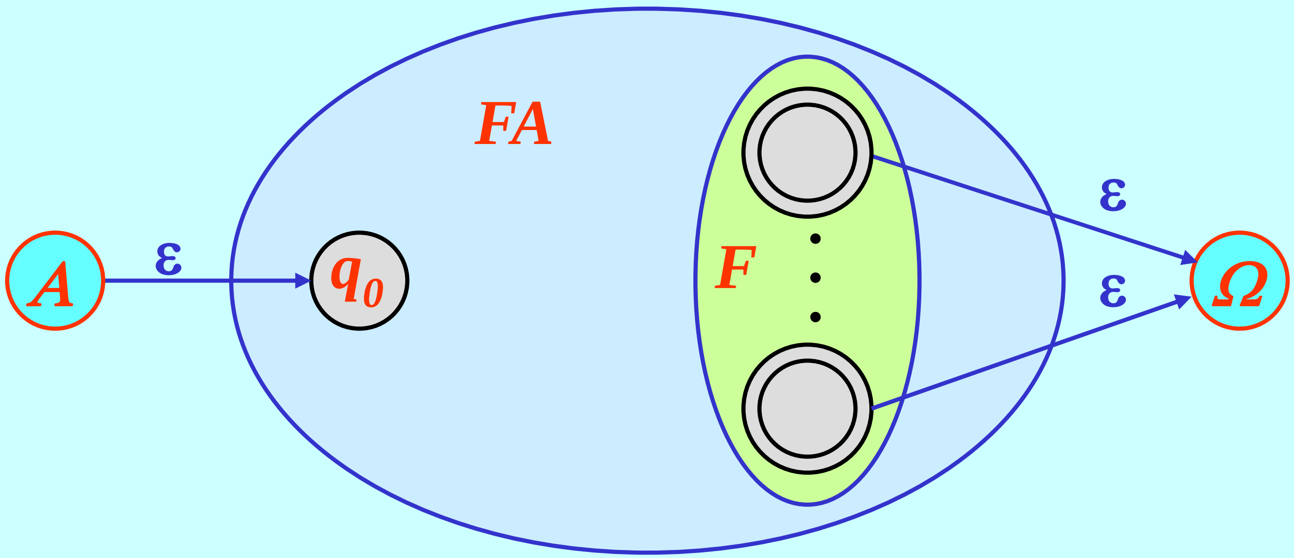
RL: FA languages ⊆ regular sets (1)

- it is possible to ***eliminate*** states in an **FA**
  - maintaining all the paths
  - labeling the transitions with regular expressions



RL: FA languages ⊆ regular sets (2)

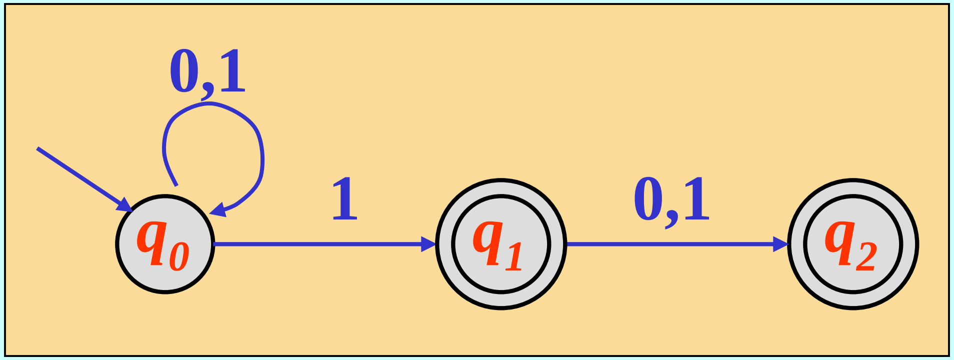
- given a finite state automaton ***FA*** = (***Q*** , ***Σ***, ***δ***, q_0, ***F***) , add an initial state ***A*** and a final state ***Ω***



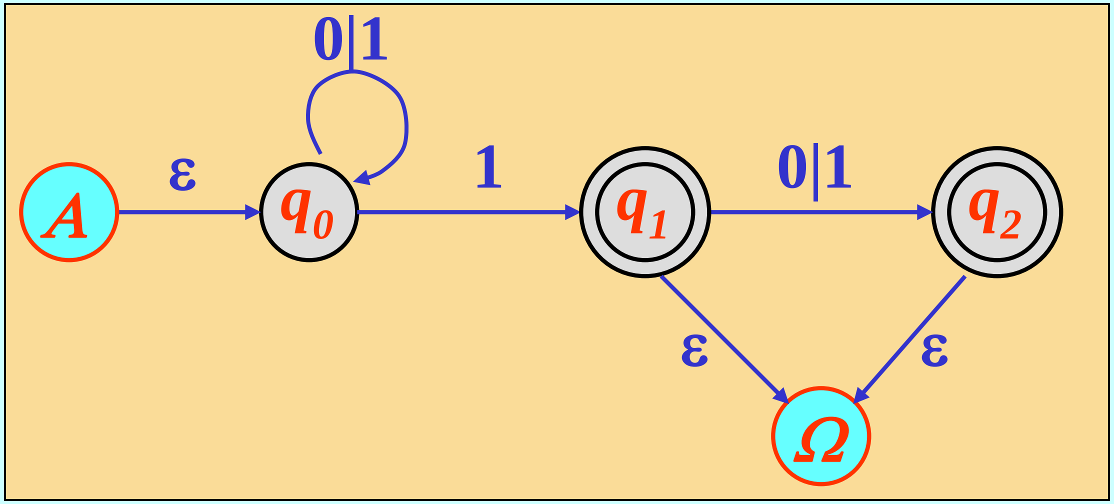
- eliminate all the states in ***FA***
- the union of the labels on the transitions from ***A*** to ***Ω*** gives the regular expression of the language ***L(FA)***

RL: from FA to regular expressions (1)

- $L(A)$ = the set of all strings over **{0,1}** containing a "**1**" in the first or second position from the end

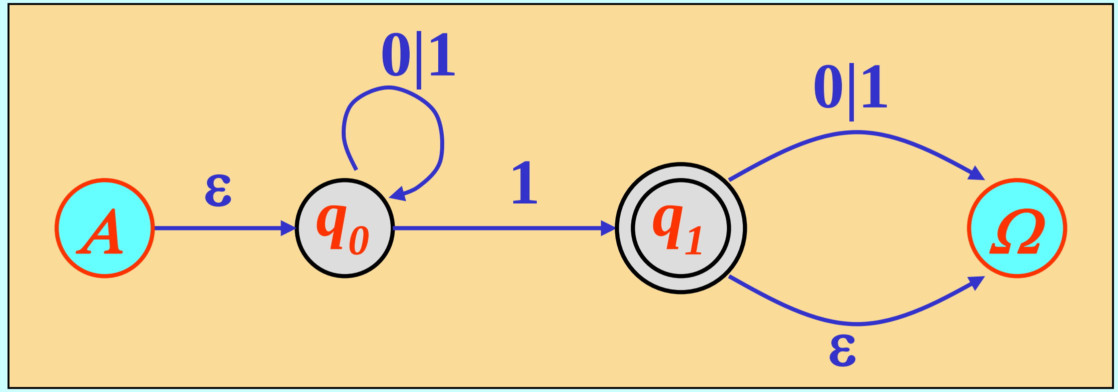


- adding the states A and Ω

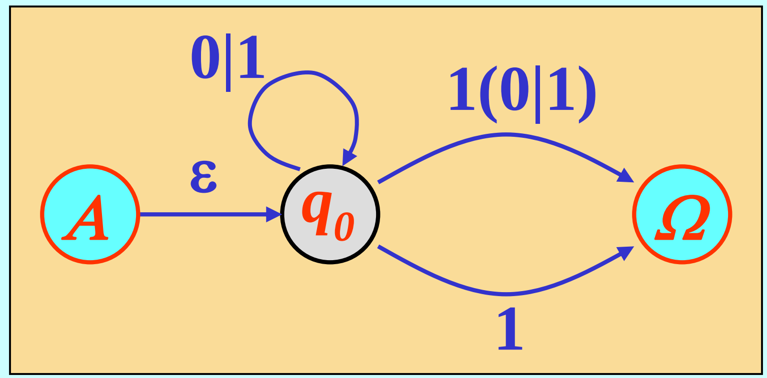


RL: from FA to regular expressions (2)

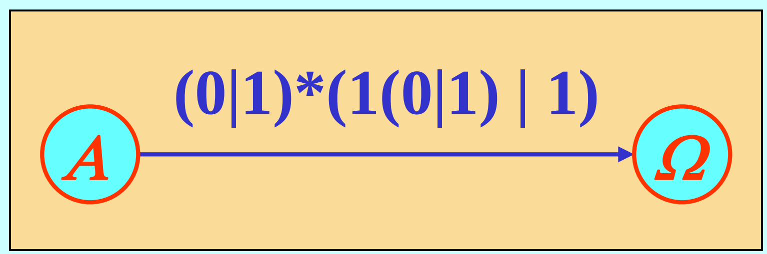
- eliminating q_2



- eliminating q_1

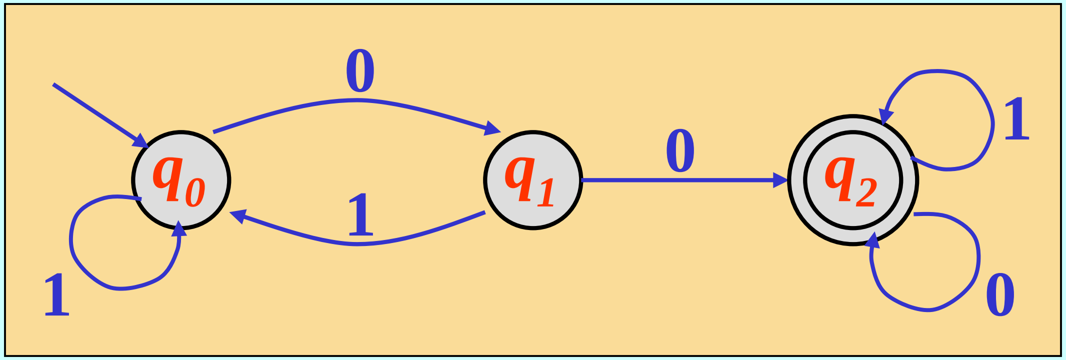


- eliminating q_0

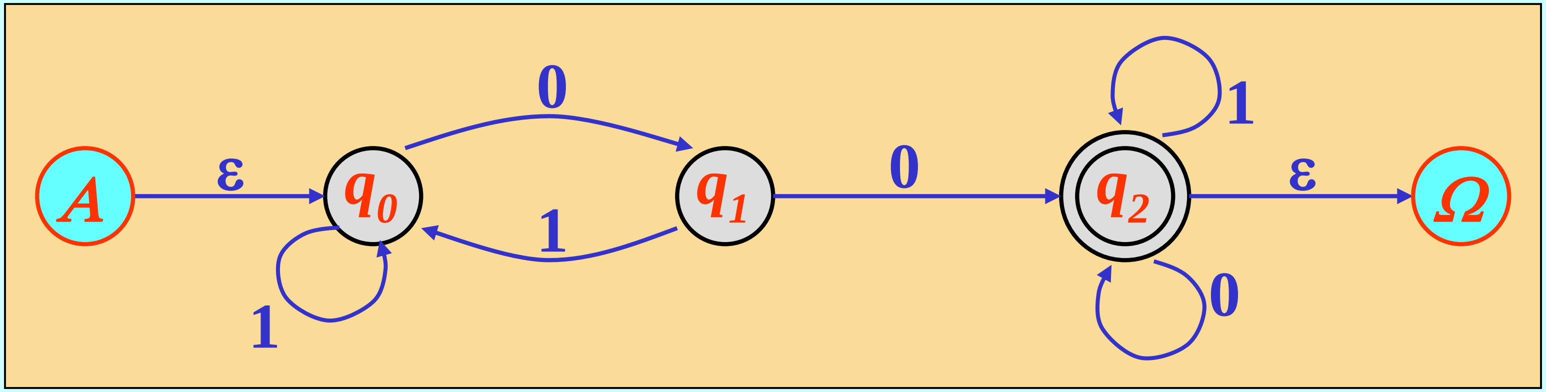


RL: from FA to regular expressions (3)

- ***L*(*A*) =** the set of all strings over **{0,1}** containing at least two consecutive " **0** "

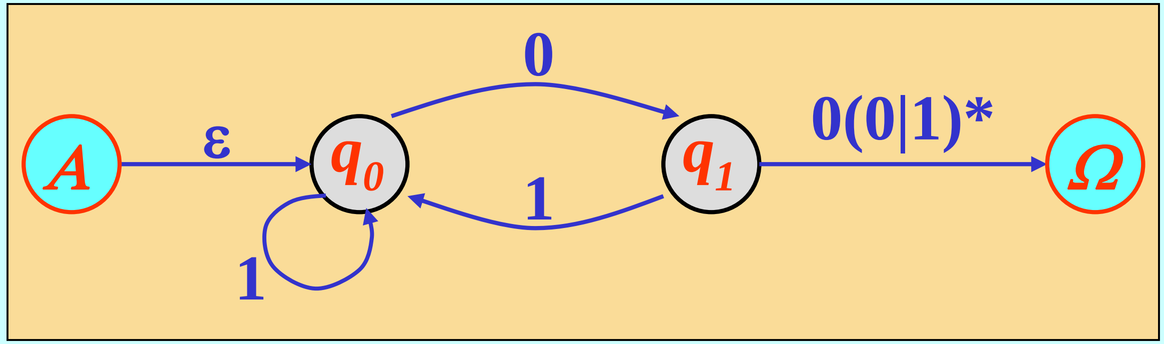


- adding the states ***A*** and ***Ω***

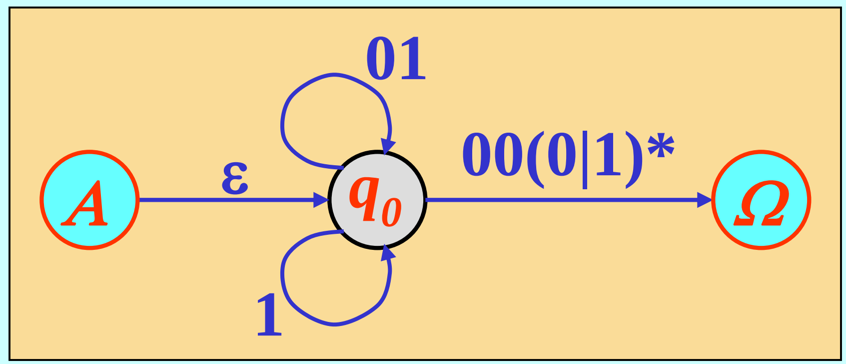


RL: from FA to regular expressions (4)

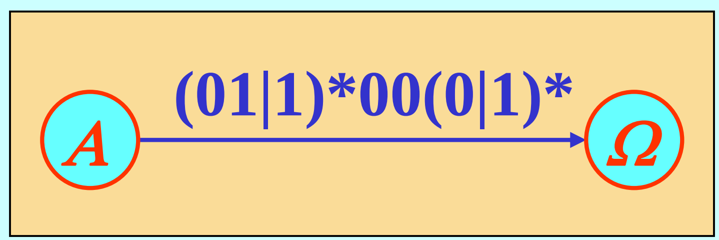
- eliminating q_2



- eliminating q_1

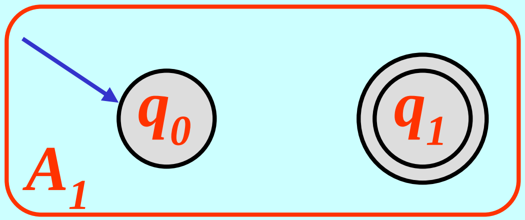


- eliminating q_0

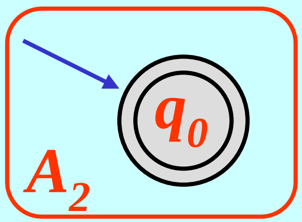


RL: regular sets $\subseteq$ FA languages (1)

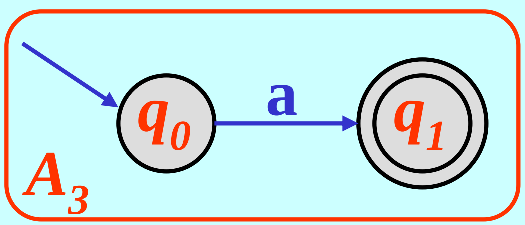
- the ***regular sets*** **:** $\varnothing$, $\{\varepsilon\}$, $\{ \mathtt{a} \}$, $\mathtt{a} \in \Sigma$ are accepted by finite state automata

- 

$\Rightarrow$ $L(A_1) = \varnothing$

- 

$\Rightarrow$ $L(A_2) = \{\varepsilon\}$

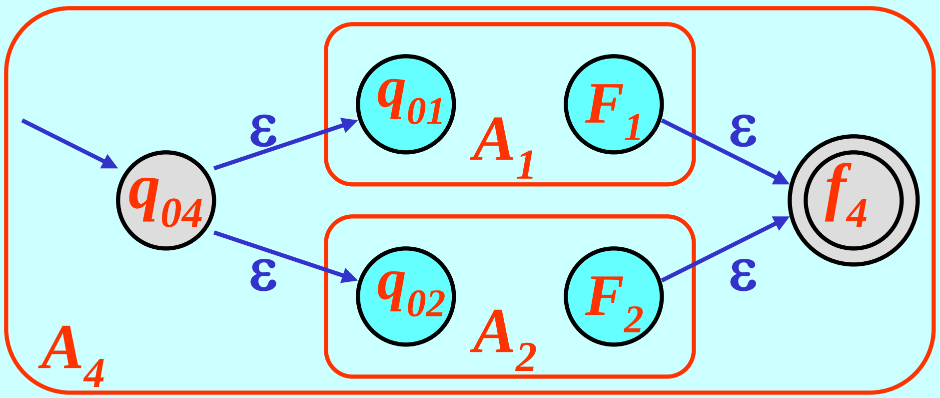
- 

$\Rightarrow$ $L(A_3) = \{ \mathtt{a} \} , \mathtt{a} \in \Sigma$

RL: regular sets ⊆ FA languages (2)

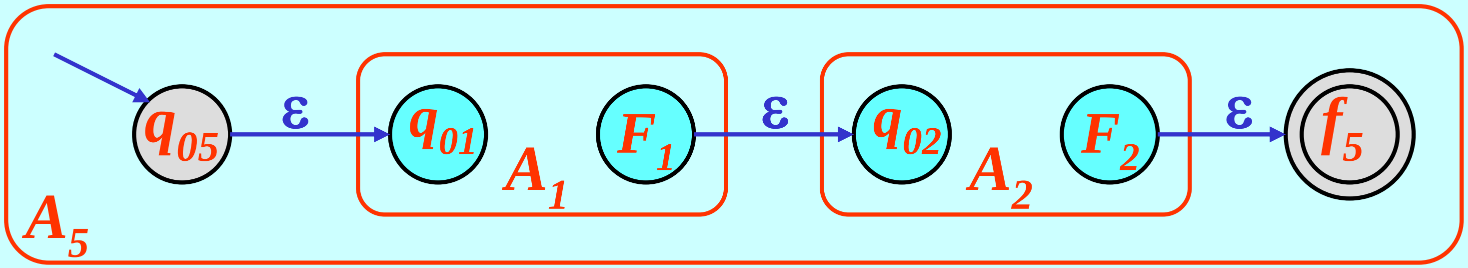
- let $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ be finite state automata

- the language $L(A_1) \cup L(A_2)$ is accepted by a finite state automaton A_4

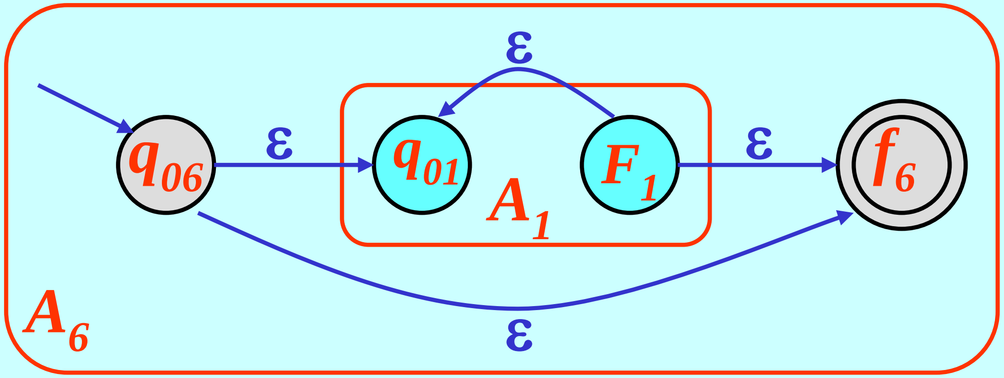


RL: regular sets ⊆ FA languages (3)

- the language $L(A_1)\ L(A_2)$ is accepted by a finite state automaton A_5

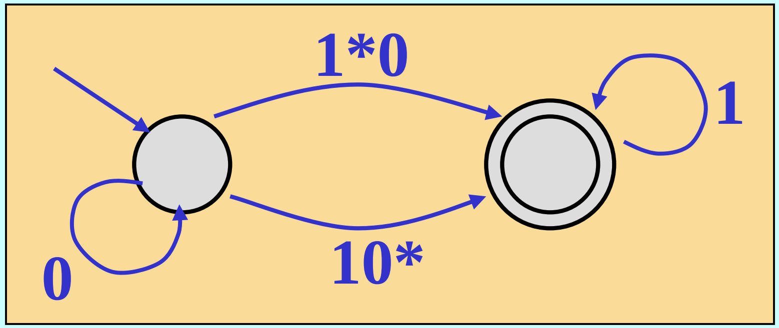


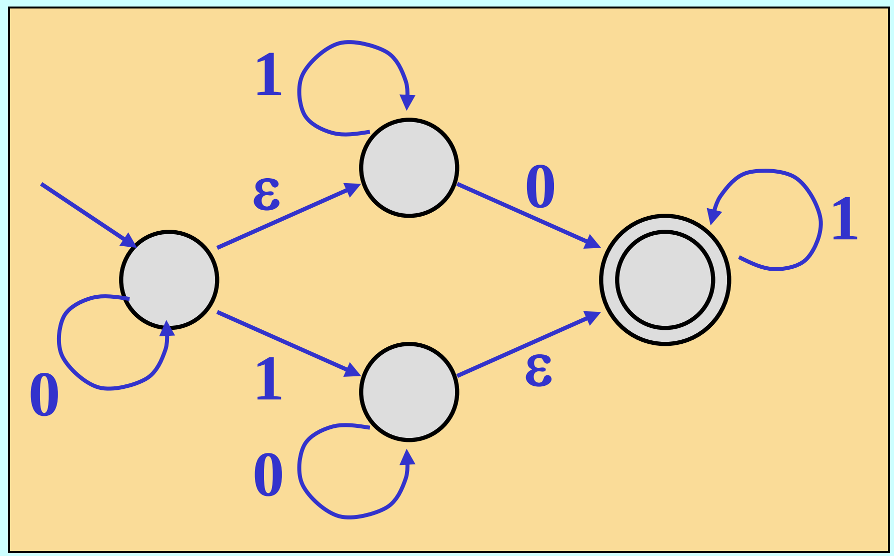
- the language $L(A_1)^*$ is accepted by a finite state automaton A_6



RL: from regular expressions to FA

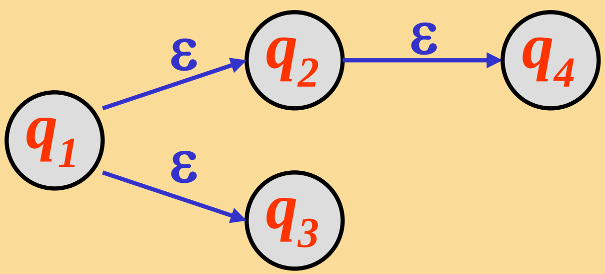
0*(1*0 | 10*)1*





RL: NFA with ε-transitions (ε-NFA)

- in the construction of **FA** from regular expressions, the ***ε-transitions*** make the automata non-deterministic
- the function ***ε-closure (q)*** gives the set of states that can be reached (recursively) from state ***q*** with the empty string



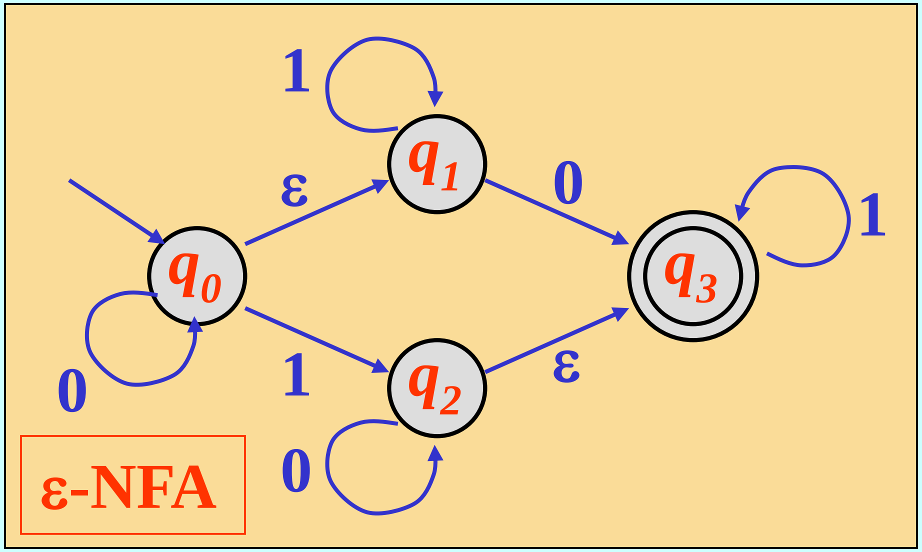
ε-closure $(q_1) = \{ q_1, q_2, q_3, q_4 \}$

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

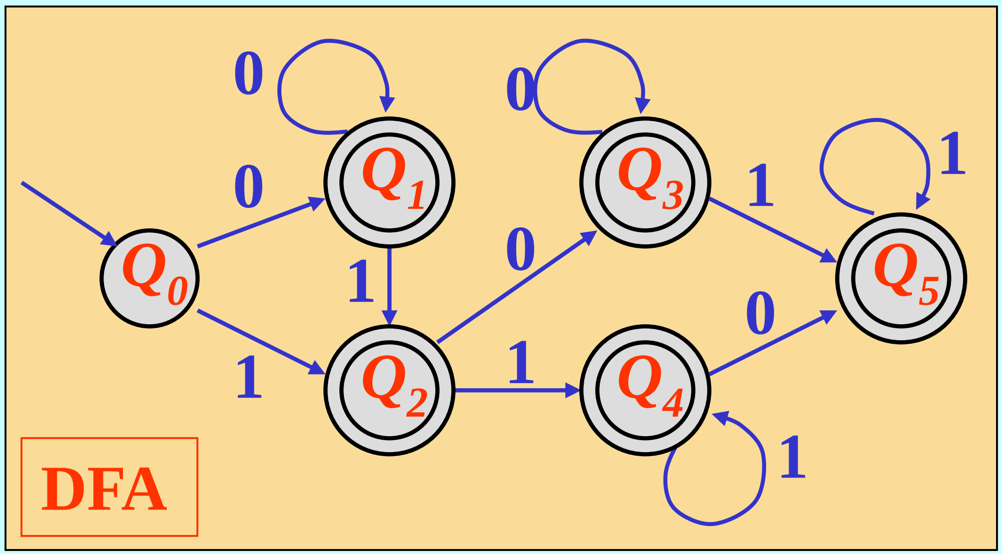
RL: equivalence of ε-NFA and DFA

- let $N = (Q_N , \Sigma, \delta_N , q_0 , F_N)$ be an **ε-NFA**
- let us construct a **DFA** $D = (Q_D , \Sigma, \delta_D , \varepsilon\text{-}closure(q_0) , F_D)$
  - $Q_D \subseteq \wp(Q_N)$
  - $\delta_D(S, a) = \varepsilon\text{-}closure(\cup_i \ \delta_N(p_i, a))$ where $p_i \in S \in Q_D$
  - $F_D = \{ S \mid S \in Q_D \ ; \ S \cap F_N \neq \emptyset \}$
- by construction $L(D) = L(N)$

RL: constructing a DFA from an ε-NFA

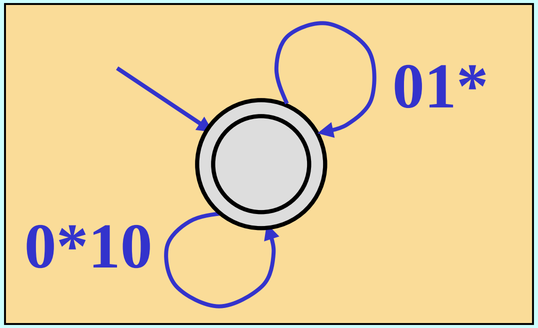


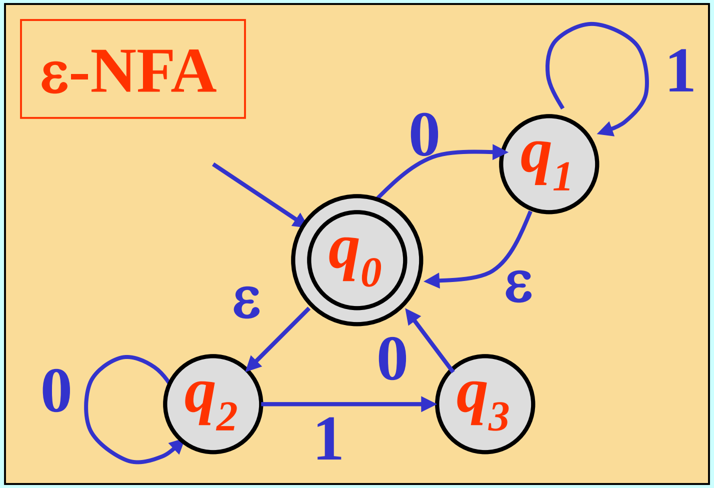
		0	1
Q_0	$\rightarrow\{q_0, q_1\}$	$\{q_0, q_1, q_3\}$	$\{q_1, q_2, q_3\}$
Q_1	$*\{q_0, q_1, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_1, q_2, q_3\}$
Q_2	$*\{q_1, q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_1, q_3\}$
Q_3	$*\{q_2, q_3\}$	$\{q_2, q_3\}$	$\{q_3\}$
Q_4	$*\{q_1, q_3\}$	$\{q_3\}$	$\{q_1, q_3\}$
Q_5	$*\{q_3\}$	-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



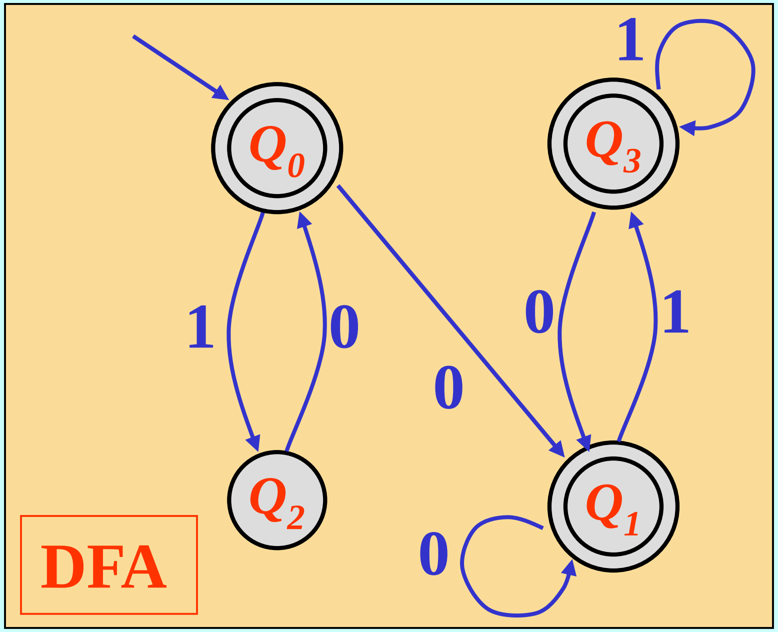
RL: from regular expressions to DFA

(0*10 | 01*)*





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



RL: FA languages ≡ regular languages (1)

- let $G = (N, T, P, S)$ be a right-regular grammar
- let us construct an FA $A = (Q, T, \delta, S, F)$
  - $Q = N \cup \{\Omega\}$ with $\Omega \notin N$
  - $F = \{\Omega\}$
  - $\delta = \{\ \delta(A, a) = B$ if $A \rightarrow a B \in P$

    $\delta(A, a) = \Omega$ if $A \rightarrow a \in P\ \}$
- by construction $L(G) = L(A)$

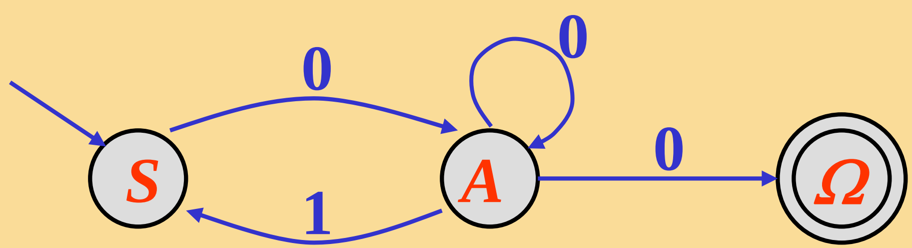
RL: from right regular grammars to FA

$G = (\{ A, S \}, \{0,1\}, P, S)$

$P = \{ S \rightarrow 0\,A$

$A \rightarrow 0\,A \mid 1\,S \mid 0$

$\}$



$S \rightarrow 0\,A \Rightarrow 0\,0\,A \Rightarrow 0\,0\,1\,S \Rightarrow 0\,0\,1\,0\,A \Rightarrow 0\,0\,1\,0\,0$

RL: FA languages ≡ regular languages (2)

- let $G = (N, T, P, S)$ be a left-regular grammar
- let us construct an FA $A = (Q, T, \delta, I, \{S\})$
  - $Q = N \cup \{ I \}$ with $I \notin N$
  - $F = \{S\}$
  - $\delta = \{ \ \delta(B, a) = A$ if $A \rightarrow B\, a \in P$

    $\delta(I, a) = A$ if $A \rightarrow a \in P \ \}$
- by construction $L(G) = L(A)$

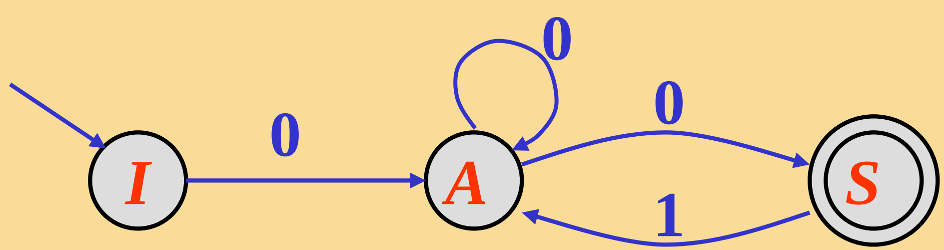
RL: from left regular grammars to FA

$G = (\{ A, S \}, \{0,1\}, P, S)$

$P = \{ S \rightarrow A\,0$

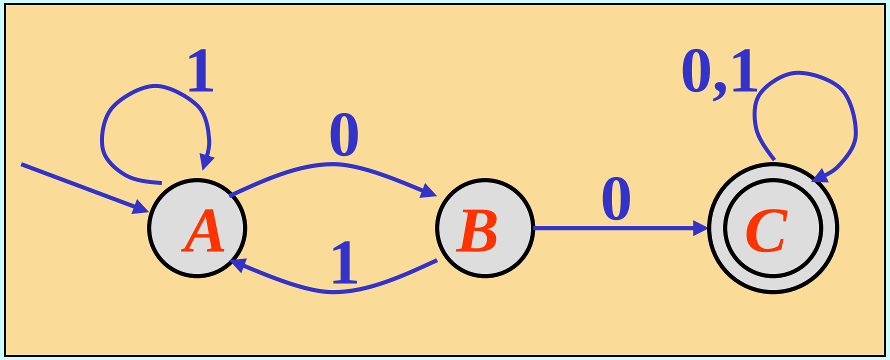
$A \rightarrow A\,0 \mid S\,1 \mid 0$

$\}$



$S \rightarrow A0 \Rightarrow A00 \Rightarrow S100 \Rightarrow A0100 \Rightarrow 00100$

RL: from FA to regular grammars



$G_1 = (\{ A, B, C \}, \{0,1\}, P_1, A)$

$$P_1 = \{ A \rightarrow 1\,A \mid 0\,B$$
$$B \rightarrow 1\,A \mid 0\,C \mid 0$$
$$C \rightarrow 0\,C \mid 1\,C \mid 0 \mid 1$$
$$\}$$

$G_2 = (\{ A, B, C \}, \{0,1\}, P_2, C)$

$$P_2 = \{ C \rightarrow C\,0 \mid C\,1 \mid B\,0$$
$$B \rightarrow A\,0 \mid 0$$
$$A \rightarrow A\,1 \mid B\,1 \mid 1$$
$$\}$$

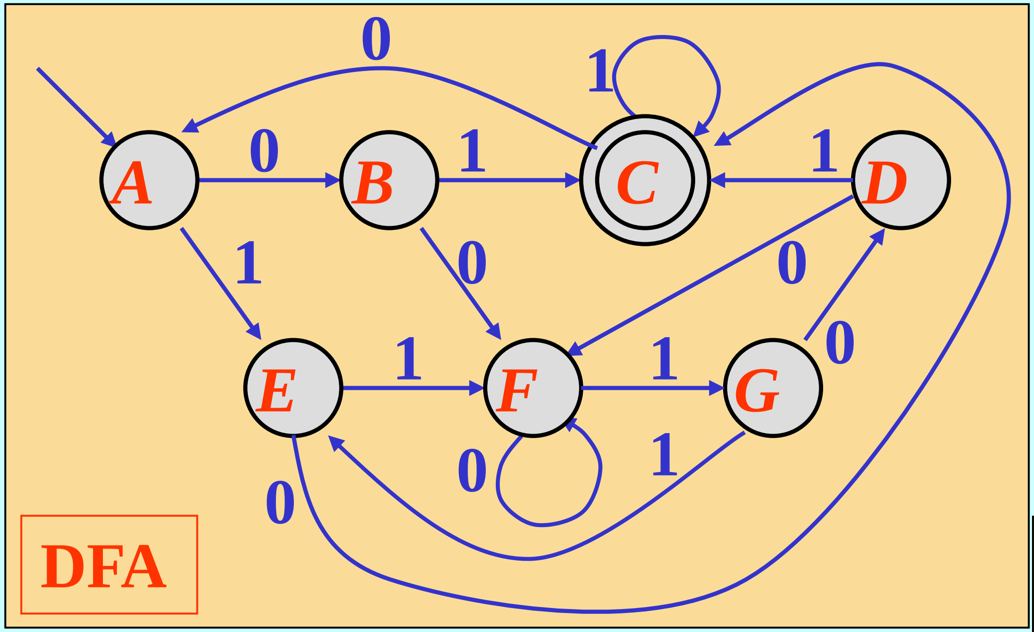
RL: minimum-state DFA

- let $DFA = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite state automaton
- two states p and q of DFA are ***distinguishable*** if there is a string $w \in \Sigma^*$ such that $\delta(p, w) \in F$ e $\delta(q, w) \notin F$
- two states p and q of DFA are ***equivalent*** $(p \equiv q)$ if they are ***non- distinguishable*** for any string $w \in \Sigma^*$
- a DFA is ***minimum-state*** if it does not contain equivalent states

RL: minimization of DFA (1)

- two 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 $p \equiv_{m+1} q$
  - if $p \equiv_m q$ and $m = ||Q|| - 2$ then $p \equiv q$
- the equivalent states can be determined by partitioning the set ***Q*** in classes of ***m-equivalent*** states, for $m = 0, 1, \ldots, ||Q|| - 2$

RL: minimization of DFA (2)

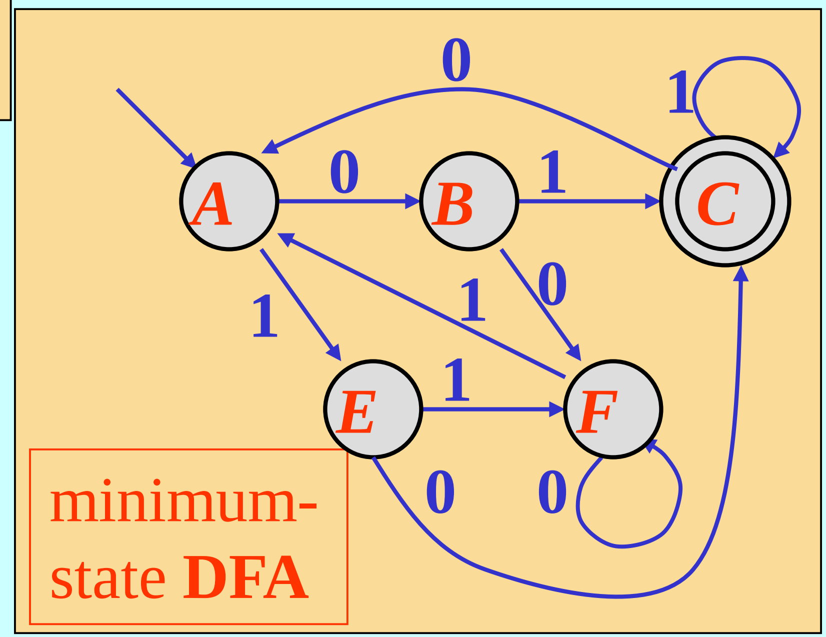


Π_0 : {*C*}, {*A*, *B*, *D*, *E*, *F*, *G*}

Π_1 : {*C*}, {*A*, *F*, *G*}, {*B*, *D*}, {*E*}

Π_2 : {*C*}, {*A*, *G* }, {*F*}, {*B*, *D*}, {*E*}

Π_3 : {*C*}, {*A*, *G* }, {*F*}, {*B*, *D*}, {*E*}

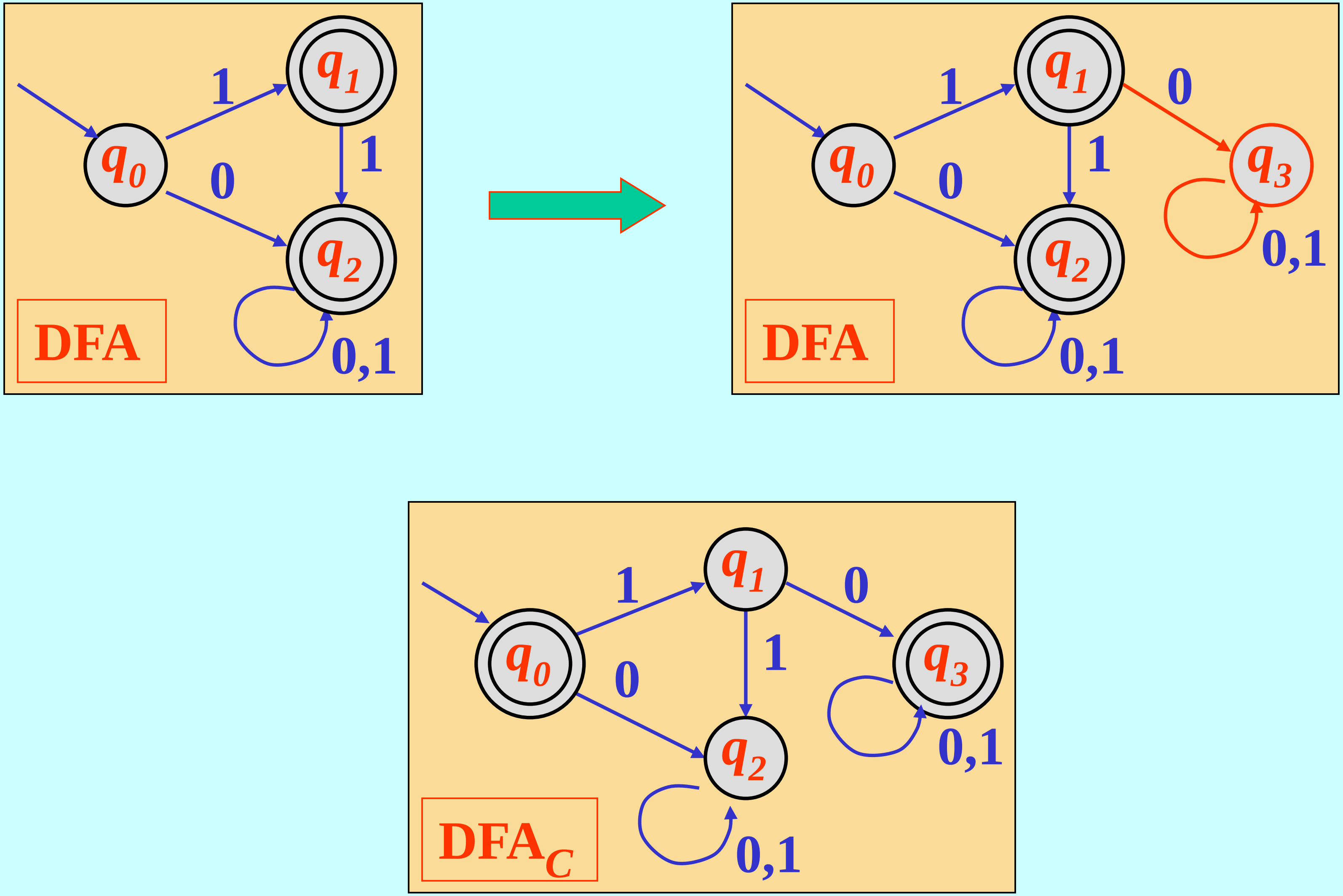


RL: complement of regular languages (1)

- the *complement* of a regular language is a regular language
  - let $DFA = (Q, \Sigma, \delta, q_0, F)$ be a ***completely specified deterministic*** finite state automaton
    - there is a transition on every symbol of Σ from every state
  - the automaton $DFA_C = (Q, \Sigma, \delta, q_0, Q - F)$ accepts the language

    $L(DFA_C) = \Sigma^* - L(DFA) = \neg L(DFA)$

RL: complement of regular languages (2)



RL: intersection of regular languages (1)

- the *intersection* of two regular languages is a regular language
  - $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$
- let $DFA_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$
  $DFA_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$
  - the automaton

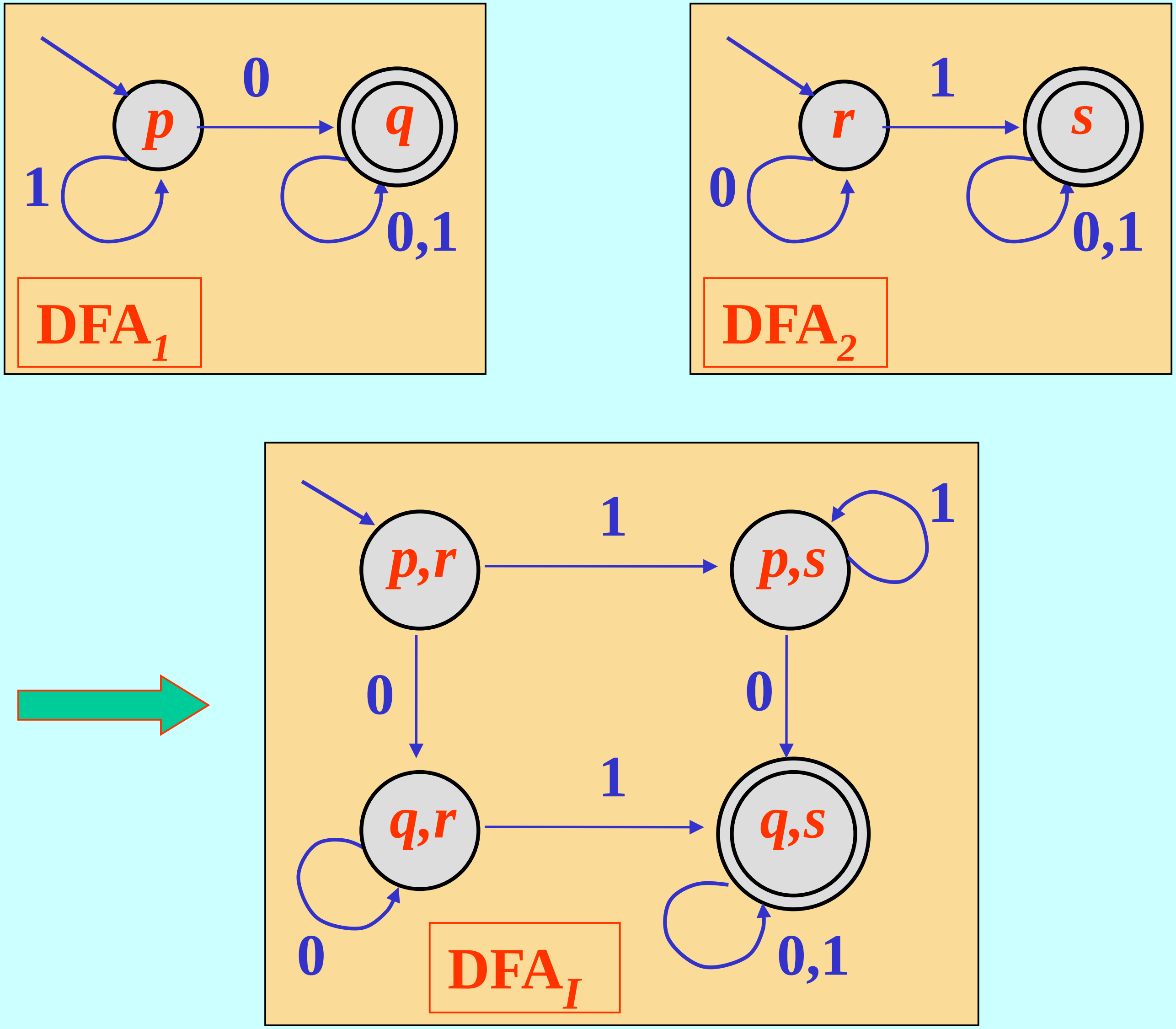
$$DFA_I = (Q_1 \times Q_2, \Sigma, \delta, (q_{01}, q_{02}), F_1 \times F_2),$$

where : $\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a)))$,

accepts the language :

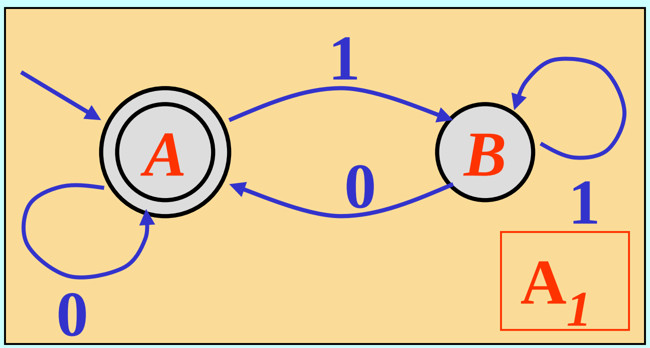
$$L(DFA_I) = L(DFA_1) \cap L(DFA_2)$$

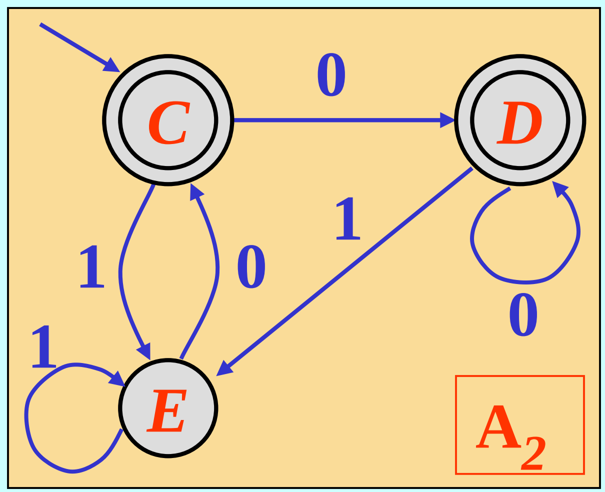
RL: intersection of regular languages (2)



RL: equivalence of regular languages

- it is possible to test if two regular languages are the same
  - $DFA_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$; $DFA_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$
  - let us find the equivalence states in the set $Q_1 \cup Q_2$
  - if $q_{01} \equiv q_{02}$ then $L(DFA_1) = L(DFA_2)$





Π_0 : {A, C, D}, {B, E}

Π_1 : {A, C, D}, {B, E}

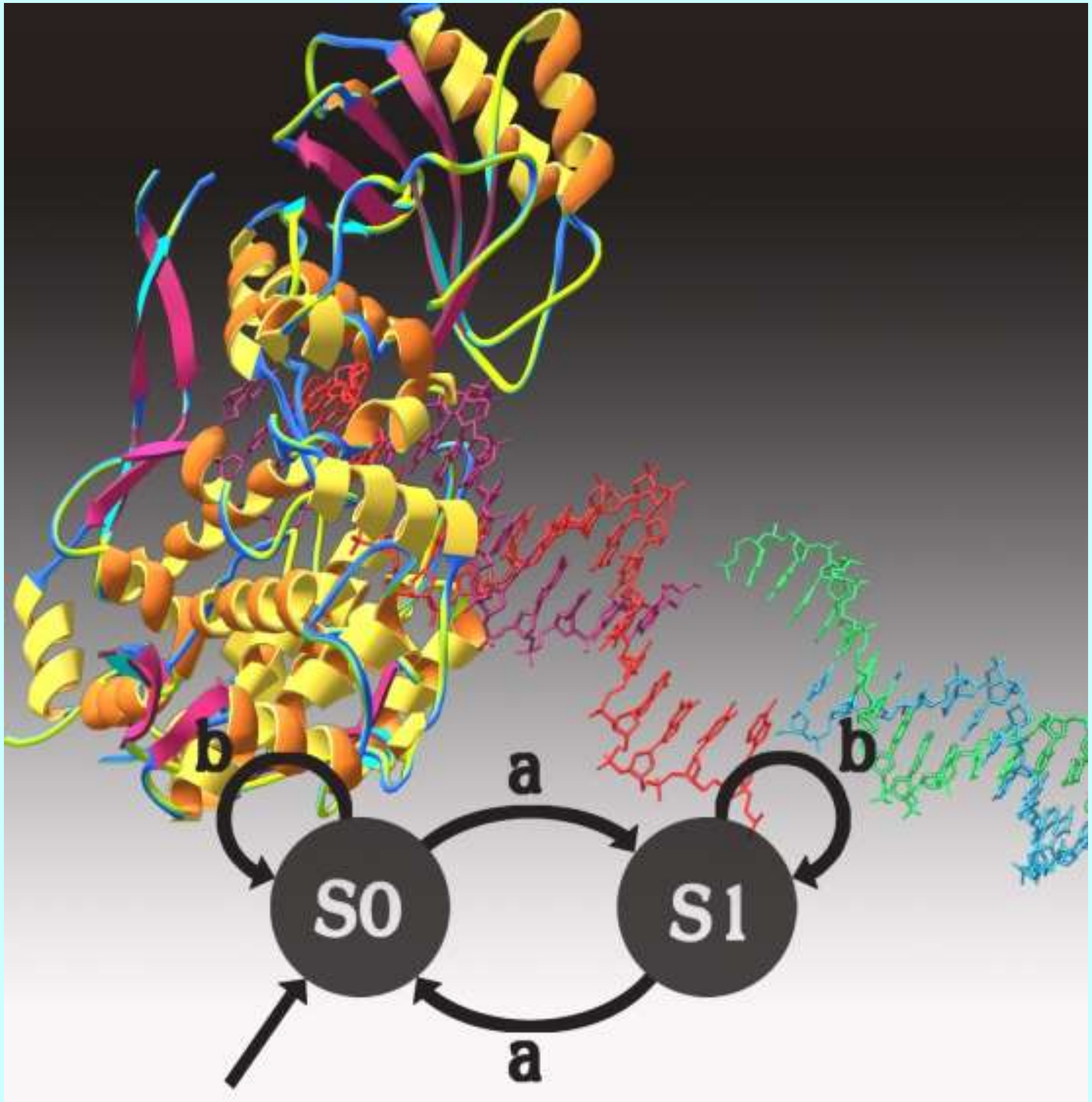
$L(A_1) = L(A_2)$

FL&C

RL: applications of regular languages

- Finding occurrences of words, phrases, *patterns* in a text
  - software for *editing* , *word processing* , …
- Constructing lexical analyzers (*scanners*)
  - compiler components that break the source text into lexical elements
    - identifiers, keywords, numeric or alphabetic constants, operators, punctuation, …
- Designing and verifying systems that have a finite number of distinct states
  - digital circuits, communication protocols, programmable controllers, …

RL: Molecular realization of an automaton



An input DNA molecule (green/blue) provides both data and fuel for the computation. Software DNA molecules (red/purple) encode program rules, and the restriction enzyme FokI (colored ribbons) functions as the automaton's hardware.

Ehud.Shapiro@weizmann.ac.il

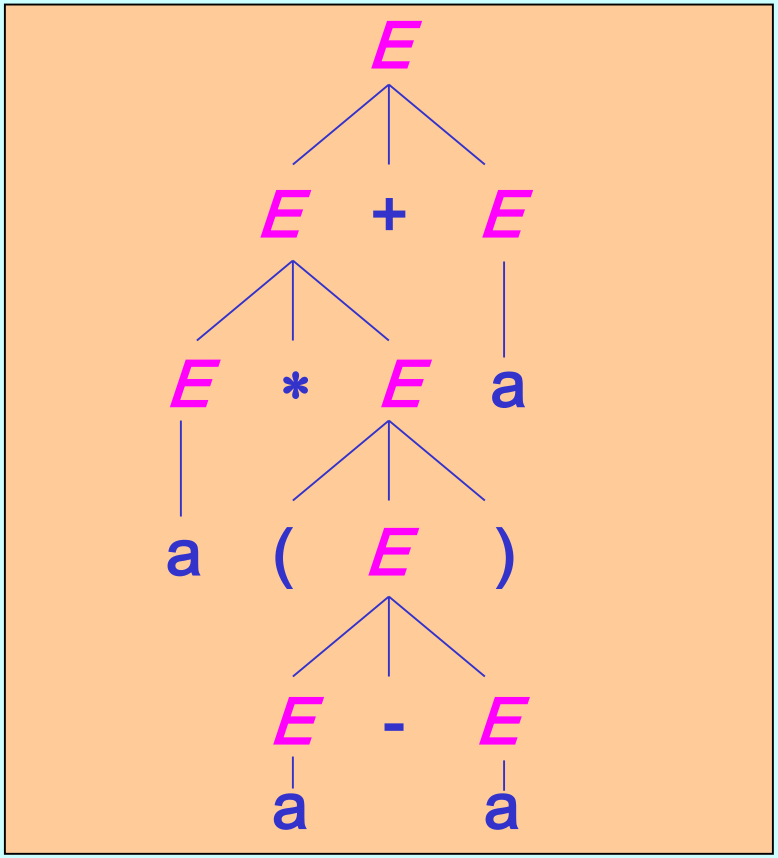
Context-Free Languages: parse trees (1)

- a *parse tree* for a context-free grammar (*CFG*) $G = (N, T, P, S)$ is a tree where
  - the root is labeled by the start symbol S
  - each interior node is labeled by a symbol in N
  - each leaf is labeled by a symbol in $N \cup T \cup \{\varepsilon\}$
  - an interior node labeled by A has children (from left to right) labeled by $X_1, X_2, \ldots, X_k$ only if $A \rightarrow X_1 \ X_2 \ \ldots \ X_k$ is a production in P
- yield of a parse tree
  - string obtained by concatenating (from left to right) the labels of the leaves

CFL: parse trees (2)

$G = (\{E\}, \{a, +, -, *, /, (,)\}, P, E)$

$P = \{E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid a\}$



$E \Rightarrow^* a * (a - a) + a$

yield

CFL: leftmost / rightmost derivations

- leftmost derivation
  - the leftmost non-terminal symbol is replaced at each derivation step
    - $E \Rightarrow \underline{E} + E \Rightarrow \underline{E} * E + E \Rightarrow a * \underline{E} + E \Rightarrow a * (\underline{E}) + E \Rightarrow a * (\underline{E} - E) + E \Rightarrow a * (a - \underline{E}) + E \Rightarrow a * (a - a) + \underline{E} \Rightarrow a * (a - a) + a$
- rightmost derivation
  - the rightmost non-terminal symbol is replaced at each derivation step
    - $E \Rightarrow E + \underline{E} \Rightarrow \underline{E} + a \Rightarrow E * \underline{E} + a \Rightarrow E * (\underline{E}) + a \Rightarrow E * (E - \underline{E}) + a \Rightarrow E * (\underline{E} - a) + a \Rightarrow \underline{E} * (a - a) + a \Rightarrow a * (a - a) + a$

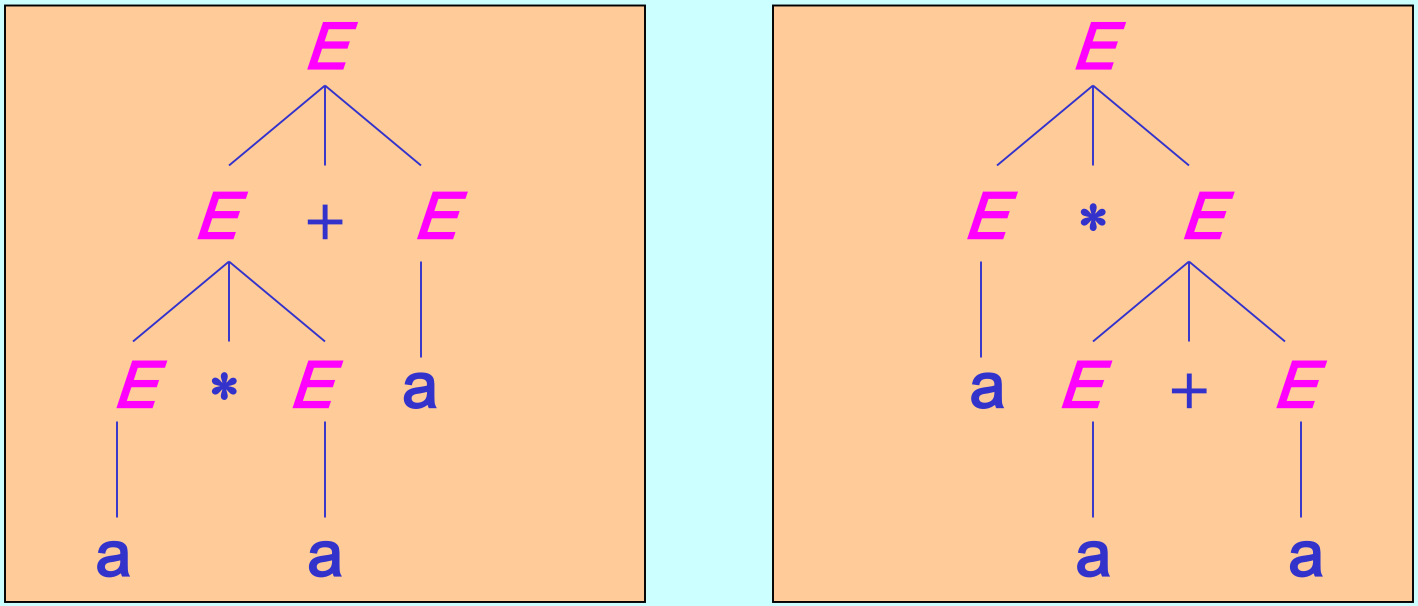
CFL: ambiguity

- every string in a CFL has at least one parse tree
- each parse tree has just one leftmost derivation and just one rightmost derivation
- a ***CFG*** is ***ambiguous*** if there is at least one string in its language having two different parse trees
- a ***CFL*** is ***inherently ambiguous*** if all its grammars are ambiguous

CFL: ambiguous grammars (1)

$G_1 = (\{ E \}, \{ a, +, -, *, /, (,) \}, P_1, E)$

$P_1 = \{ E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid a \}$

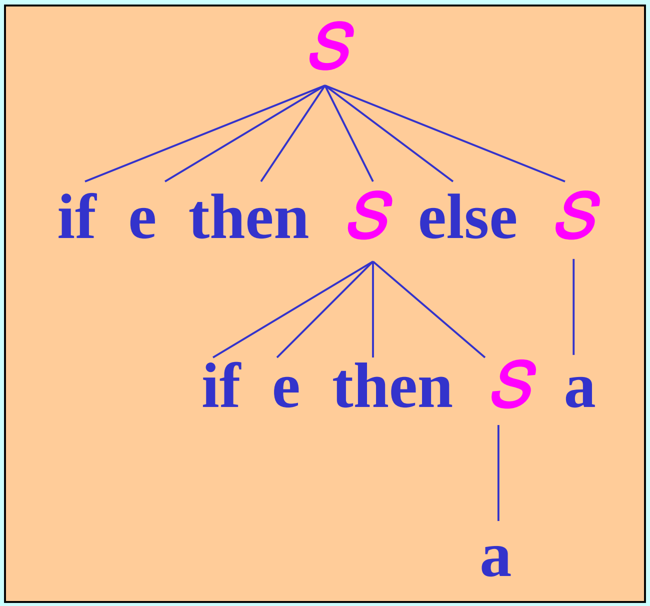


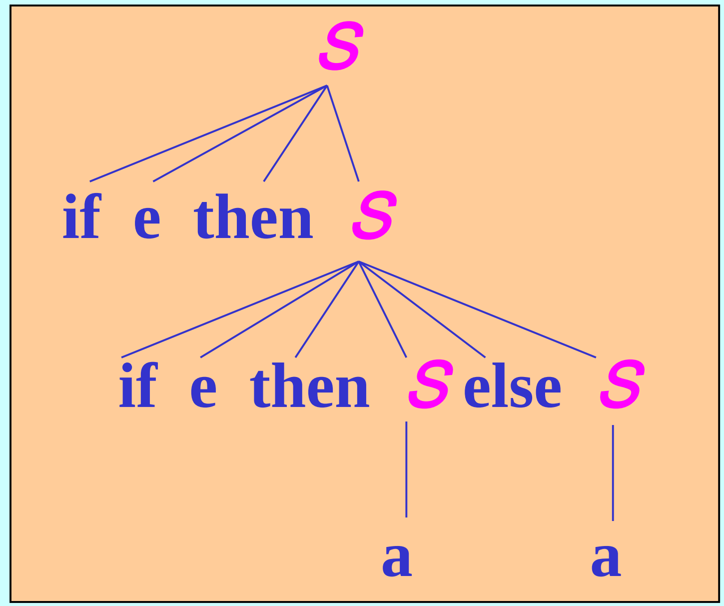
$E \Rightarrow^* a * a + a$

CFL: ambiguous grammars (2)

G_2 = ({ S } , { if , then , else , e , a) } , P_2 , S)

P_2 = { S → if e then S else S | if e then S | a }





S ⇒* if e then if e then a else a

CFL: equivalent non-ambiguous grammars

$G_3 = (\{E, T, F\}, \{a, +, -, *, /, (,)\}, P_3, E)$

$P_3 = \{\; E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid a$

$\}$

$L(G_1) = L(G_3)$

$G_4 = (\{S, M, U\}, \{\text{if}, \text{then}, \text{else}, e, a\}, P_4, S)$

$P_4 = \{\; S \rightarrow M \mid U$

$M \rightarrow \text{if } e \text{ then } M \text{ else } M \mid a$

$U \rightarrow \text{if } e \text{ then } M \text{ else } U \mid \text{if } e \text{ then } S$

$\}$

$L(G_2) = L(G_4)$

CFL: eliminating useless symbols in CFG

- a symbol X is useful for a $CFG = (N, T, P, S)$ if there is some derivation $S \Rightarrow^* \alpha X \beta \Rightarrow^* w \in T^*$
  - a useful symbol X generates a *non-empty language*:
    $X \Rightarrow^* x \in T^*$
  - a useful symbol X is *reachable*:
    $S \Rightarrow^* \alpha X \beta$
- eliminating useless symbols from a grammar will non change the generated language

1. eliminate symbols generating an empty language
2. eliminate unreachable symbols

CFL: finding the useful symbols in CFG

- finding symbols generating *non-empty languages*
  - every symbol of ***T*** generates a non-empty language
  - if $A \rightarrow \alpha$ and all symbols in α generate a non-empty language, then ***A*** generates a non-empty language
- finding *reachable* symbols
  - the start symbol ***S*** is reachable
  - if $A \rightarrow \alpha$ and ***A*** is reachable, all symbols in α are reachable

CFL: symbols generating an empty language

$G_1 = (\{S, A, B, C\}, \{a, b\}, P_1, S)$

$P_1 = \{\ S \rightarrow A\,a \mid b\,C\,b$

$A \rightarrow a\,B\,A \mid b\,A\,S$

$B \rightarrow a\,S \mid b\,A \mid b$

$C \rightarrow a\,S\,a \mid a\ \}$

symbols generating a ***non-empty language*** :

$\{a, b\} \cup \{B, C\} \cup \{S\}$

symbols generating an ***empty language*** :

$\{A\}$

$G_2 = (\{S, B, C\}, \{a, b\}, P_2, S)$

$P_2 = \{\ S \rightarrow b\,C\,b$

$B \rightarrow a\,S \mid b$

$C \rightarrow a\,S\,a \mid a\ \}$

$L(G_1) = L(G_2)$

CFL: unreachable symbols

$G_2 = (\{S, B, C\}, \{a, b\}, P_2, S)$

$P_2 = \{ \; S \rightarrow b\,C\,b$

$B \rightarrow a\,S \mid b$

$C \rightarrow a\,S\,a \mid a \; \}$

reachable symbols :

$\{S\} \cup \{b, C\} \cup \{a\}$

unreachable symbols :

$\{B\}$

$G_3 = (\{S, C\}, \{a, b\}, P_3, S)$

$P_3 = \{ \; S \rightarrow b\,C\,b$

$C \rightarrow a\,S\,a \mid a \; \}$

$L(G_1) = L(G_2) = L(G_3)$

CFL: ε-productions in CFG

- according to the Chomsky classification, only type 0 grammars can have *ε-productions*
- anyway the languages generated by CFG's that contain *ε-productions* are CFL
  - a CFG G_1 with *ε-productions* can be transformed into an equivalent CFG G_2 without *ε-productions* :

    $$L(G_2) = L(G_1) - \{\varepsilon\}$$

  - if $A \rightarrow X_1 \dots X_i \dots X_n$ is in P_1 and $X_i \Rightarrow^* \varepsilon$, then P_2 will contain $A \rightarrow X_1 \dots X_i \dots X_n$ and $A \rightarrow X_1 \dots X_{i-1}\, X_{i+1} \dots X_n$

CFL: eliminating ε-productions in CFG

$G_1 = (\{S, A, B\}, \{a, b\}, P_1, S)$

$P_1 = \{ \; S \rightarrow aA \mid b$

$\quad A \rightarrow BSB \mid BB \mid a$

$\quad B \rightarrow aAb \mid b \mid \varepsilon$

$\}$

symbols that generate **ε** : $\{B, A\}$

$G_2 = (\{S, A, B\}, \{a, b\}, P_2, S)$

$P_2 = \{ \; S \rightarrow aA \mid b \mid a$

$\quad A \rightarrow BSB \mid BB \mid a \mid SB \mid BS \mid S \mid B$

$\quad B \rightarrow aAb \mid b \mid ab$

$\}$

$L(G_1) = L(G_2)$

CFL: pushdown automata (PDA)

- A **PDA** is a 7-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
  - Q : finite (non empty) set of **states**
  - Σ : alphabet of **input** symbols
  - Γ : alphabet of **stack** symbols
  - δ : **transition** function
    - $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \{(p, \gamma) \mid p \in Q ; \gamma \in \Gamma^*\}$
  - q_0 : **start** state ($q_0 \in Q$)
  - Z_0 : **start** stack symbol ($Z_0 \in \Gamma$)
  - F : set of **final states** ($F \subseteq Q$)

CFL: transitions of PDA (1)

- $\delta(q\,,a\,,X) = \{\,(p_1\,,\gamma_1)\,,(p_2\,,\gamma_2)\,,\ldots\,,(p_m\,,\gamma_m)\,\}$
  - from state q , with a in input and X on top of the stack:
    - consumes a from the input string
    - goes to a state p_i and replaces X with γ_i
      - the first symbol of γ_i goes on top of the stack
- $\delta(q\,,\varepsilon\,,X) = \{\,(p_1\,,\gamma_1)\,,(p_2\,,\gamma_2)\,,\ldots\,,(p_m\,,\gamma_m)\,\}$
  - from state q , with X on top of the stack:
    - no input symbol is consumed
    - goes to a state p_i and replaces X with γ_i
      - the first symbol of γ_i goes on top of the stack

CFL: transitions of PDA (2)

- *instantaneous configuration* of a PDA: (q, w, γ)
  - q : current state
  - w : remaining input string
  - γ : current stack contents
- transition:
  - if $\delta(q, a, X) = \{ \ldots, (p, \alpha), \ldots \}$, then

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

CFL: languages accepted by PDA

➢ Language accepted by *final state* by the PDA

$$P = (Q , \Sigma , \Gamma , \delta , q_0 , Z_0 , F)$$

- $L(P) = \{ w \mid w \in \Sigma^* \; ; \; (q_0 , w , Z_0) \vdash^* (q , \varepsilon , \alpha) \; ; \; q \in F \}$

➢ Language accepted by *empty stack* by the PDA

$$P = (Q , \Sigma , \Gamma , \delta , q_0 , Z_0 , \varnothing)$$

- $N(P) = \{ w \mid w \in \Sigma^* \; ; \; (q_0 , w , Z_0) \vdash^* (q , \varepsilon , \varepsilon) \}$

CFL: example of PDA

$P = (\{ q_0 , q_1 \} , \{ 0 , 1 \} , \{ 0 , 1 , Z \} , \delta, q_0 , Z , \varnothing)$

$\delta(q_0 , 0 , Z) = \{(q_0 , 0Z)\}$

$\delta(q_0 , 1 , Z) = \{(q_0 , 1Z)\}$

$\delta(q_0 , 0 , 0) = \{(q_0 , 00),(q_1 , \varepsilon)\}$

$\delta(q_0 , 1 , 0) = \{(q_0 , 10)\}$

$\delta(q_0 , 0 , 1) = \{(q_0 , 01)\}$

$\delta(q_0 , 1 , 1) = \{(q_0 , 11),(q_1 , \varepsilon)\}$

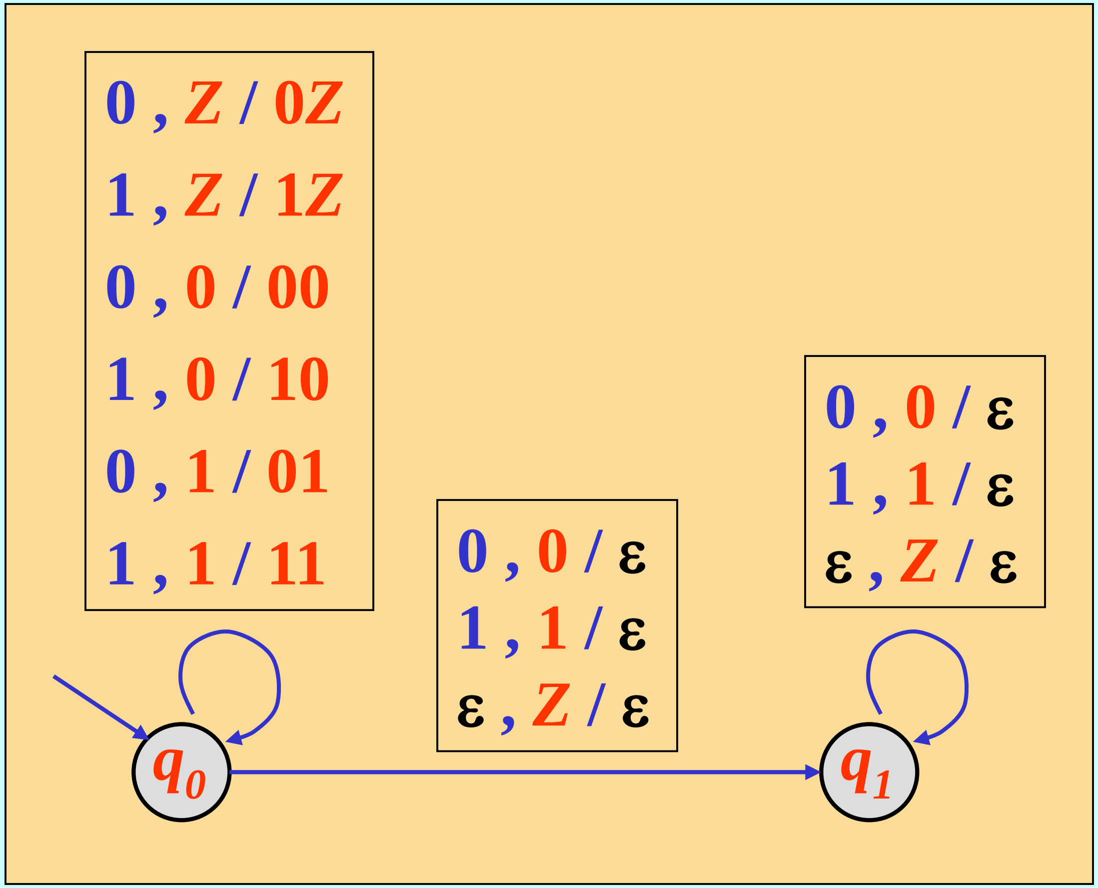
$\delta(q_1 , 0 , 0) = \{(q_1 , \varepsilon)\}$

$\delta(q_1 , 1 , 1) = \{(q_1 , \varepsilon)\}$

$\delta(q_0 , \varepsilon , Z) = \{(q_1 , \varepsilon)\}$

$\delta(q_1 , \varepsilon , Z) = \{(q_1 , \varepsilon)\}$

CFL: graphical notation for PDA



$N(P) = \{ w\, w^R \mid w \in \{0, 1\}^* \}$

CFL: configuration sequences of PDA

$(q_0, 001100, Z)$ ← *initial configuration*

$\top$

$(q_0, 01100, 0Z) \vdash (q_1, 1100, Z) \vdash (q_1, 1100, \varepsilon)$

$\top$

$(q_0, 1100, 00Z)$

$\top$

$(q_0, 100, 100Z) \vdash (q_0, 00, 1100Z) \vdash (q_0, 0, 01100Z) \vdash (q_0, \varepsilon, 001100Z)$

$\top$ (below $(q_0, 100, 100Z)$) $\qquad\qquad$ $\top$ (below $(q_0, 0, 01100Z)$)

$(q_1, 00, 00Z)$ $\qquad\qquad$ $(q_1, \varepsilon, 1100Z)$

$\top$

$(q_1, 0, 0Z)$

$\top$

(q_1, ε, Z)

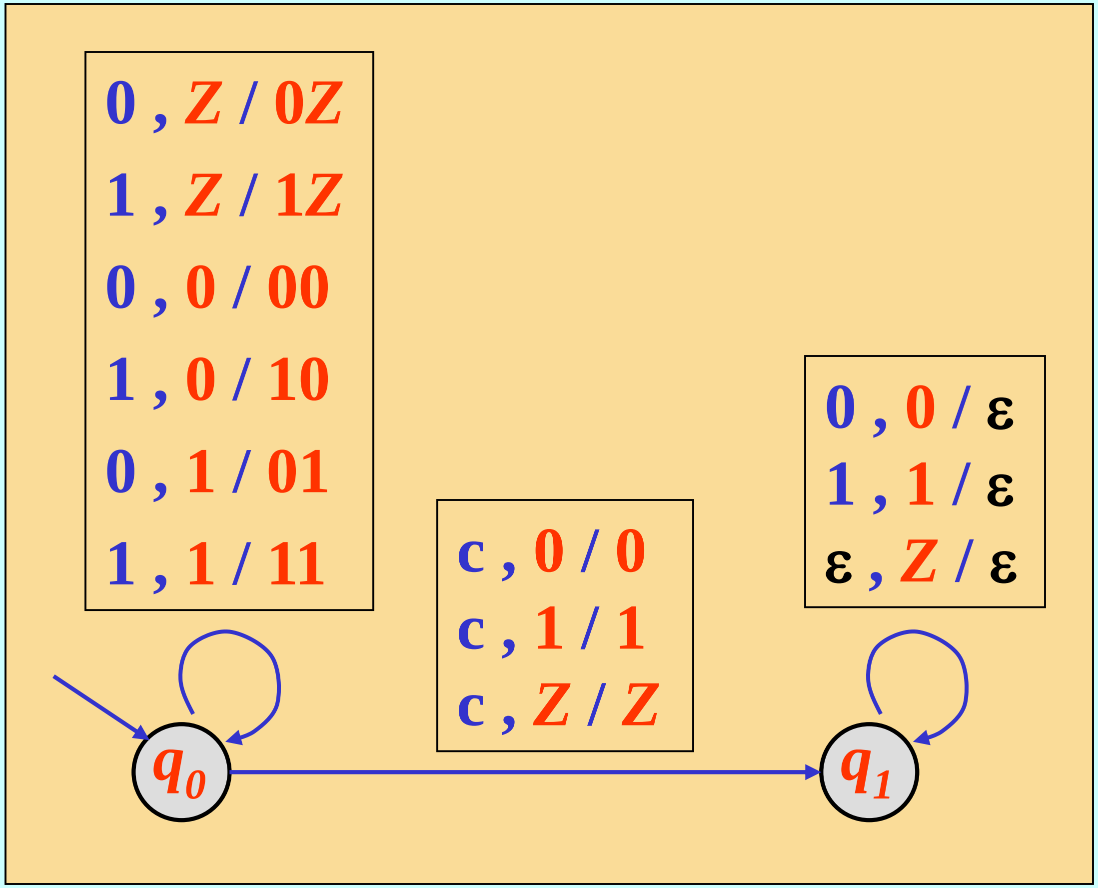
$\top$

$(q_1, \varepsilon, \varepsilon)$ ← *accepting configuration*

CFL: deterministic pushdown automata (DPDA)

- A **PDA** $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is *deterministic (DPDA)* if:
  - $\delta(q, a, X)$ has at most one member for any $q \in Q$, $a \in (\Sigma \cup \{\varepsilon\})$, $X \in \Gamma$
  - if $\delta(q, a, X) \neq \varnothing$ for some $a \in \Sigma$, then $\delta(q, \varepsilon, X) = \varnothing$
- the languages accepted by DPDA are *properly included* ($\subset$) in the languages accepted by PDA
  - the language $\{ w w^R \mid w \in \{0, 1\}^* \}$ is not accepted by DPDA

CFL: example of DPDA



$N(P) = \{ w \, c \, w^R \mid w \in \{0, 1\}^* \}$

CFL: PDA languages ≡ CFL

- let $G = (N, T, P, S)$ be a context-free grammar
- let us construct a $PDA = (\{q\}, T, \Gamma, \delta, q, S, \varnothing)$
  - $\Gamma = N \cup T$
  - $\delta = \{ \ \delta(q, \varepsilon, A) = \{ (q, \alpha) \}$ for each $A \rightarrow \alpha \in P \}$

    $\delta(q, a, a) = \{ (q, \varepsilon) \}$ for each $a \in T$

    $\}$
- ***PDA*** accepts ***L(G)*** by *empty stack*, making a sequence of transitions corresponding to a *leftmost derivation*

CFL: from CFL to PDA (1)

$L(G) = \{ w\, w^R \mid w \in \{0, 1\}^* \}$

$S \rightarrow 0\,S\,0$

$\Rightarrow 0\,0\,S\,0\,0$

$\Rightarrow 0\,0\,1\,S\,1\,0\,0$

$\Rightarrow 0\,0\,1\,1\,0\,0$

$G = (\{S\}, \{0, 1\}, P, S)$

$P = \{ S \rightarrow 0\,S\,0 \mid 1\,S\,1 \mid \varepsilon \}$

$P = (\{q\}, \{0, 1\}, \{0, 1, S\}, \delta, q, S, \varnothing)$

$\delta(q, \varepsilon, S) = \{ (q, 0S0), (q, 1S1), (q, \varepsilon) \}$

$\delta(q, 0, 0) = \{ (q, \varepsilon) \}$

$\delta(q, 1, 1) = \{ (q, \varepsilon) \}$

CFL: from CFL to PDA (2)

↙ *initial configuration*

$(q,001100,S) \vdash \ldots$

$\top$

$(q,001100,0S0) \vdash (q,01100,S0) \vdash \ldots$

$\top$

$(q,01100,0S00) \vdash (q,1100,S00) \vdash \ldots$

$\top$

$(q,1100,1S100) \vdash (q,100,S100) \vdash \ldots$

$\top$

↗ $(q,\varepsilon,\varepsilon) \dashv (q,0,0) \dashv (q,00,00) \dashv (q,100,100)$

accepting configuration

CFL: properties of CFL

- the CFL's are *closed* under the operations:
  - ***union***
  - ***concatenation***
  - ***Kleene closure***
- the CFL's are *not closed* under the operations:
  - ***complement***
  - ***intersection***
- it is possible to decide membership of a string ***w*** in a CFL by algorithms (Cocke-Younger-Kasamy, Earley, …) with complexity $O(n^3)$, where $n = |w|$

FL&C

CFL: deterministic context free languages

- the *deterministic* CFL's (the languages accepted by ***DPDA***) are *closed* under the operations:
  - ***complement***
- the *deterministic* CFL's are *not closed* under the operations:
  - ***union***
  - ***intersection***
  - ***concatenation***
  - ***Kleene closure***
- it is possible to decide membership of a string ***w*** in a *deterministic* CFL by algorithms with complexity O(n) , where n = |***w***|

CFL: applications of context free languages

- Representation of programming languages
  - grammars for Algol, Pascal, C, Java, …
- Construction of syntax analyzers (*parsers*)
  - compiler components that analyze the structure of a source program and represent it by means of a parse tree
- Description of the structure and the semantic contents of documents (*Semantic Web*) by means of *Markup Languages*
  - XML (*Extensible Markup Language*) , RDF (*Resource Description Framework*) , OWL (*Web Ontology Language*) , …

Turing Machines

- A **TM** is a 6-tuple $M = (Q , \Sigma, \Gamma, \delta, q_0 , F)$

  - Q : finite (non empty) set of **states**
  - Σ: alphabet of **input** symbols (**B** $\notin \Sigma$)
  - Γ: 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
  - δ: **transition** function
    - $\delta: \ Q \times \Gamma \rightarrow Q \times \Gamma \times \{ L , R \}$ (L = *left* , R = *right*)
  - q_0 : **start** state ($q_0 \in Q$)
  - F : set of **final states** ($F \subseteq Q$)

TM: transitions of a TM (1)

- $\delta(q, X) = (p, Y, L)$
  - from state q , having X as the current tape symbol:
    - goes to state p and replaces X with Y
    - moves the tape head one position *left*
- $\delta(q, X) = (p, Y, R)$
  - from state q , having X as the current tape symbol:
    - goes to state p and replaces X with Y
    - moves the tape head one position *right*

TM: transitions of a TM (2)

- *instantaneous configuration* of a TM:

$$(X_1 \ldots X_{i-1}\ q\ X_i\ \ldots X_n)$$

  - q : current state
  - $X_1 \ldots X_{i-1} X_i \ldots X_n$: current string on tape
  - X_i : current tape symbol

TM: transitions of a TM (3)

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

TM: language accepted by a TM

- Language accepted by a TM

$$M = (Q\ , \Sigma, \Gamma, \delta, q_0\ , F)$$

- $L(M) = \{\ w \mid w \in \Sigma^* ; (q_0 w) \vdash^* (\alpha\ q\ \beta) ; q \in F\ \}$

FL&C

TM: example of TM

$M = (\{ q_0, q_1, q_2, q_3, q_4 \}, \{ 0, 1 \}, \{ 0, 1, X, Y, B \}, \delta, q_0, \{q_4\})$

δ	0	1	X	Y	B
$\rightarrow q_0$	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	(q_4, B, R)
$*q_4$					

$L(M) = \{ 0^n 1^n \mid n \geq 1 \}$

$(q_0 0011) \vdash (Xq_1 011) \vdash (X0q_1 11) \vdash (Xq_2 0Y1) \vdash (q_2 X0Y1) \vdash (Xq_0 0Y1) \vdash$
$(XXq_1 Y1) \vdash (XXYq_1 1) \vdash (XXq_2 YY) \vdash (Xq_2 XYY) \vdash (XXq_0 YY) \vdash (XXYq_3 Y)$
$(XXYYq_3 B) \vdash (XXYYBq_4) \vdash$ ← *accepting configuration*

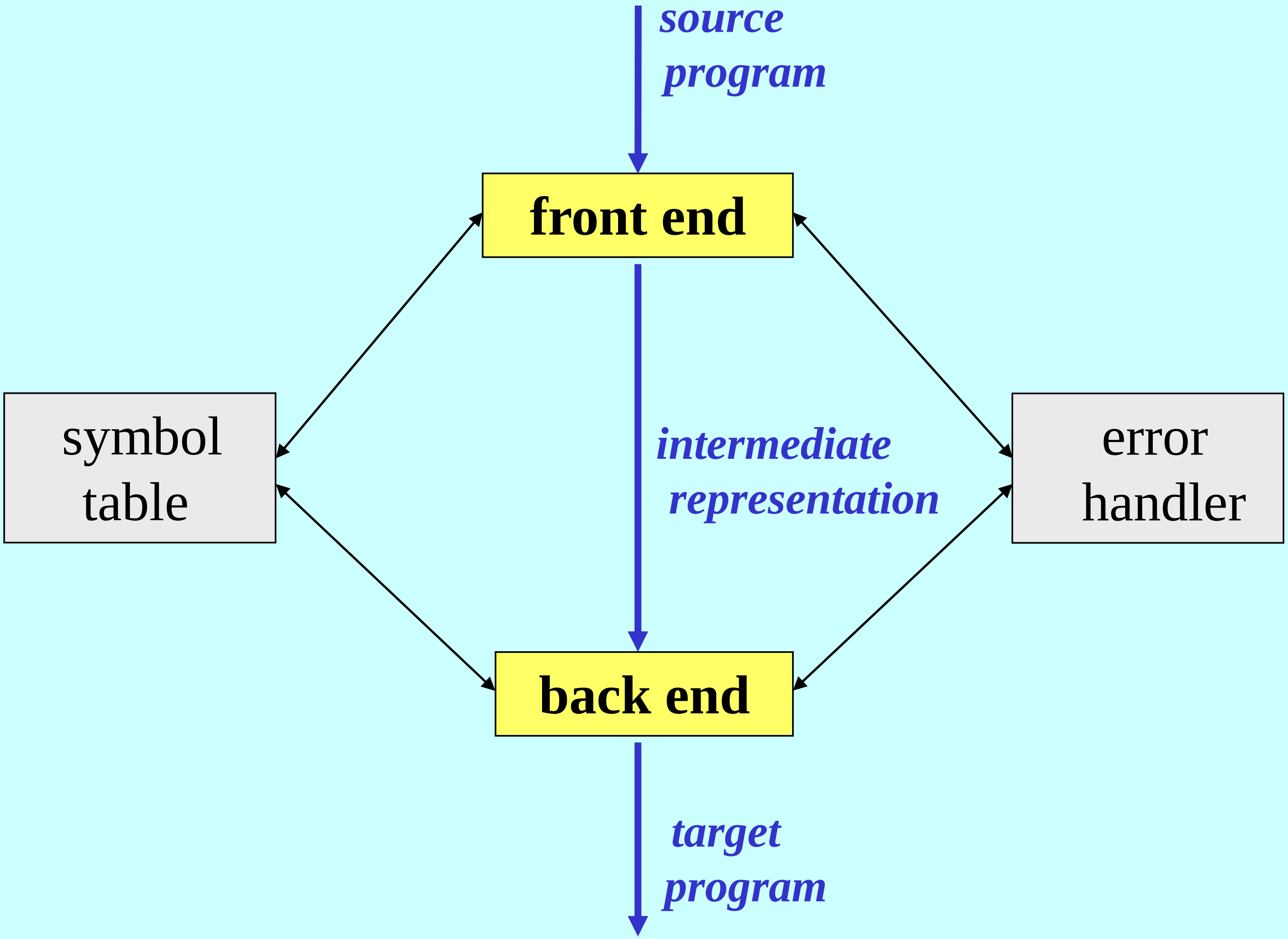
TM: recursive sets

- the languages accepted by *TM's* are called ***recursively enumerable sets*** and are equivalent to the ***type 0 languages*** (*phrase structure*)
- Halting problem
  - a TM always ***halts*** when it is in an accepting state
  - it is not always possible to require that a TM ***halts*** if it does not accept
- the *membership* of a string in a *recursively enumerable set* is ***undecidable***
- the languages accepted by TM's that always ***halt*** are called ***recursive sets***
- the *membership* of a string in a *recursive set* is ***decidable***

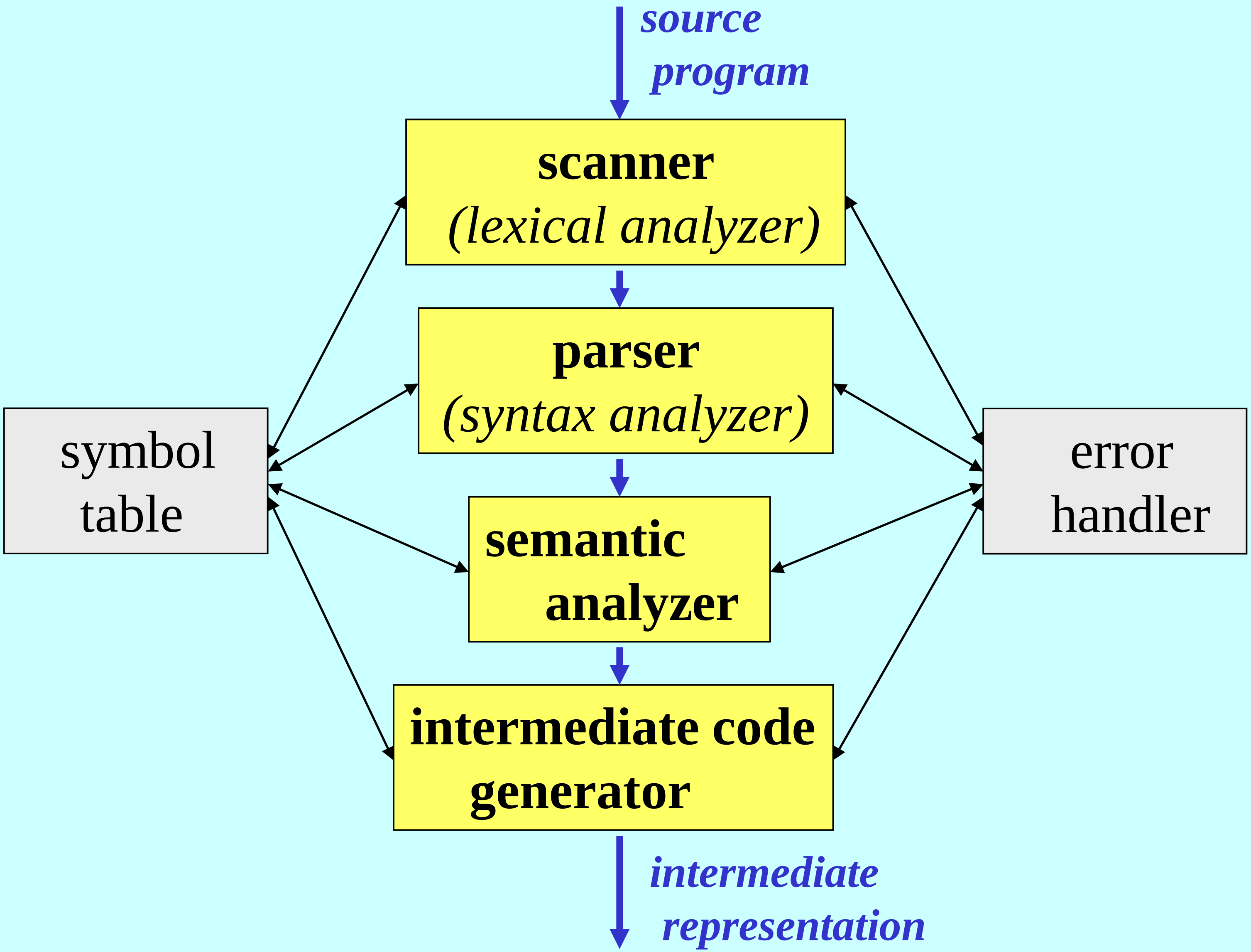
TM: decidability / computability

- the TM defines the most general model of computation
  - any computable function can be computed by a TM (*Church-Turing thesis*)
- the TM can be used to classify languages / problems / functions
  - non recursively enumerable
    - cannot 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*)

Compiler Structure

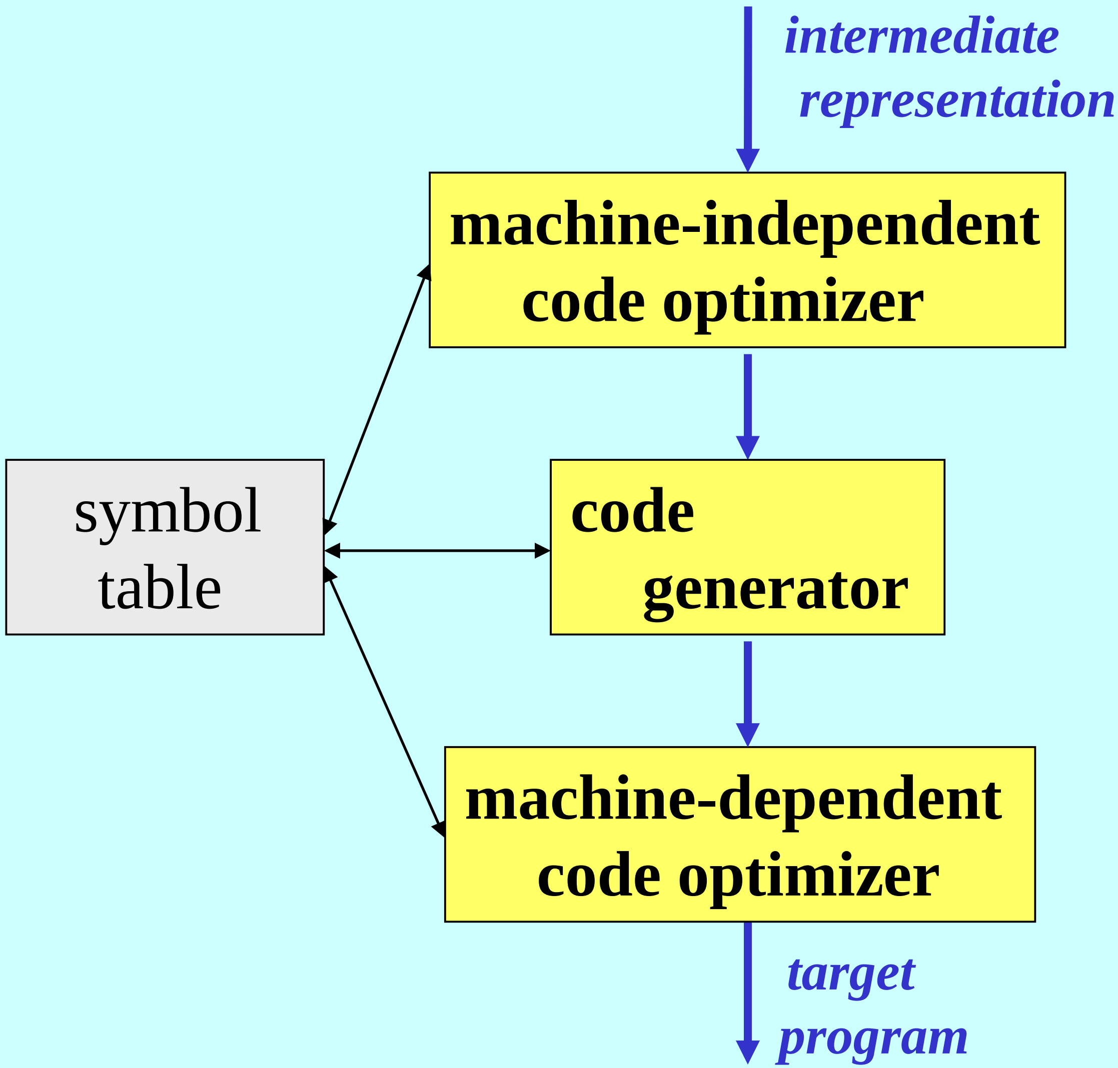


CS: phases of a front end

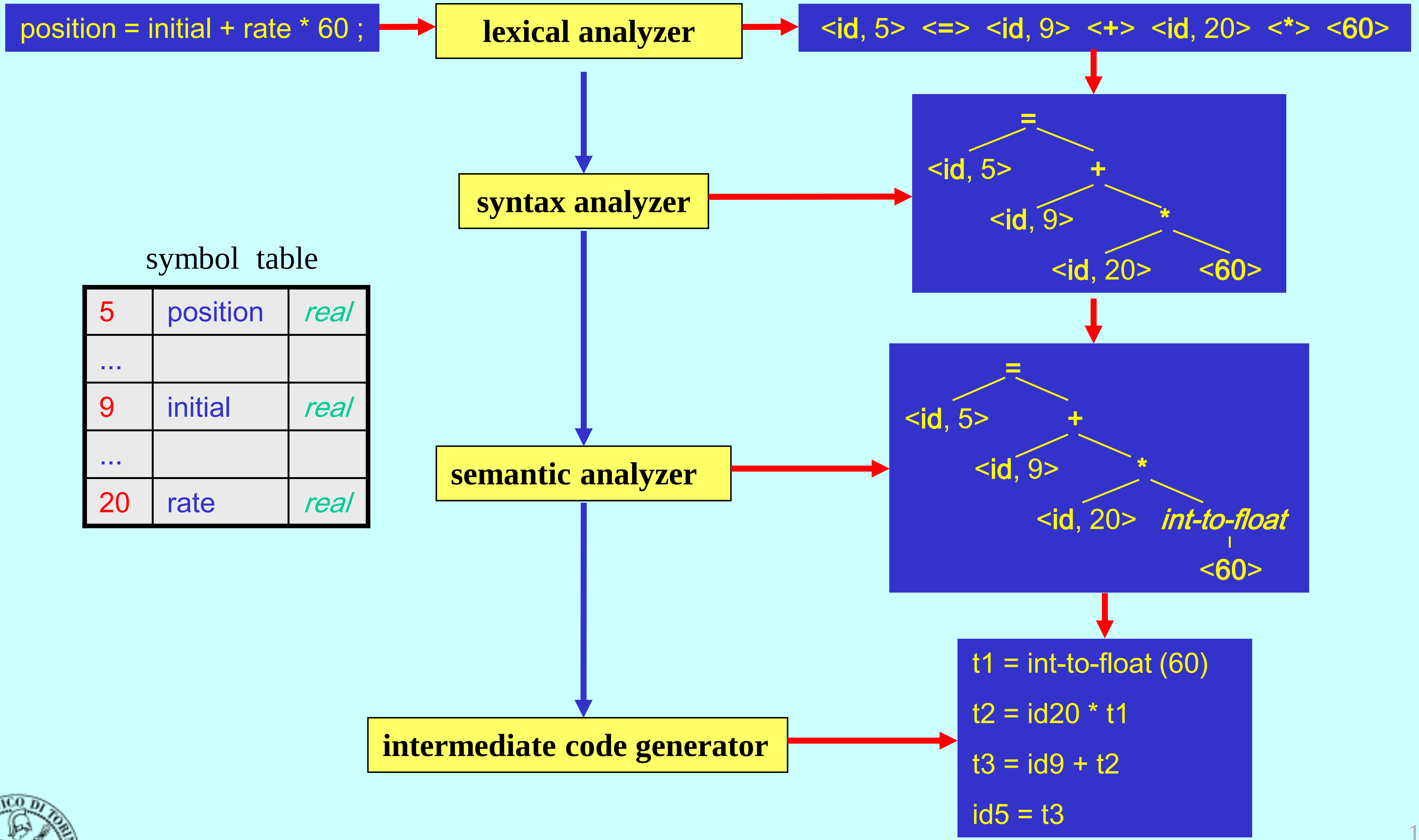


FL&C

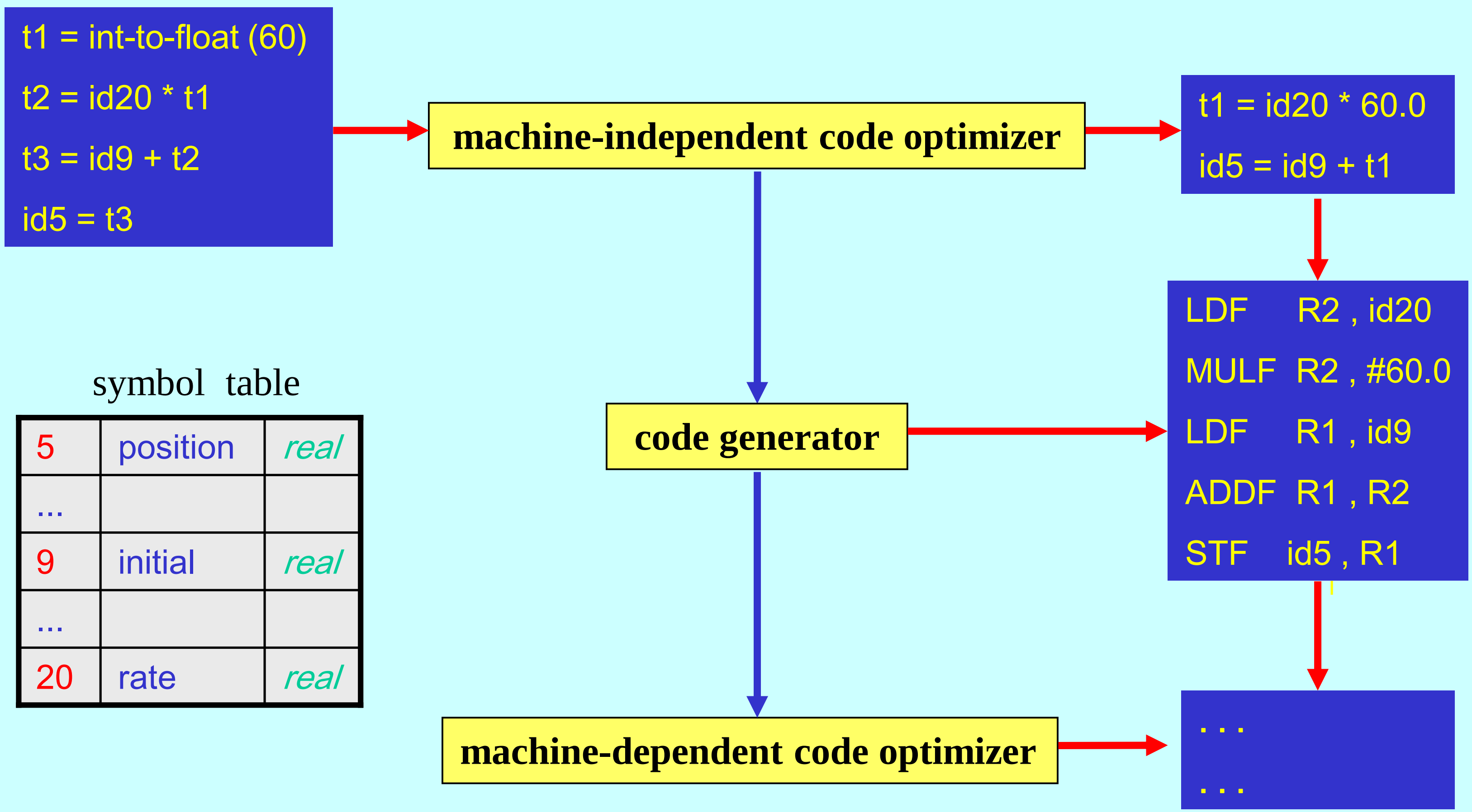
CS: phases of a back end



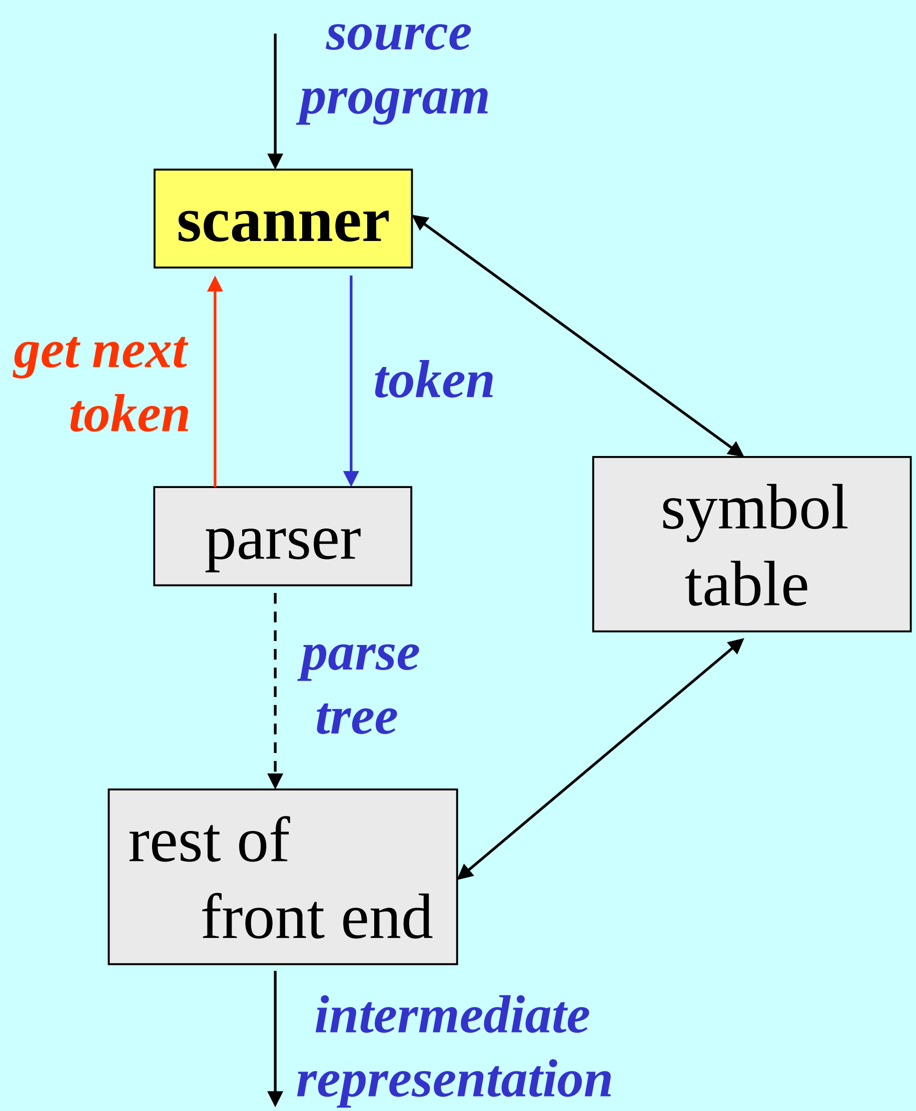
CS: front-end translation of an assignment statement



CS: back-end translation of an assignment statement



Lexical Analysis



LA: tokens, lexemes, patterns

- token
  - *terminal symbol* in the grammar for the source language
- lexeme
  - string of characters 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

LA: role of a scanner

- upon receiving a ***get next token*** command from the parser, the scanner reads input characters until it can identify a ***token***
- simplifies the job of the parser
  - discards as many irrelevant details as possible
    - *white space, tabs, newlines, comments, ...*
  - parser rules are only concerned with *tokens*, not with *lexemes*
    - parser does not care that an identifier is "*i*" or "*supercalifragilisticexpialidocious*"
- improves compiler efficiency
  - scanners are usually much faster than parsers

LA: regular definitions

- a *regular definition* is a sequence of definitions

$$\mathbf{d_1 \rightarrow r_1 \quad ; \quad d_2 \rightarrow r_2 \quad ; \quad \dots \quad ; \quad d_n \rightarrow r_n}$$

  - each $\mathbf{d_i}$ is a distinct name (*token*)
  - each $\mathbf{r_i}$ is a regular expression over $\Sigma \cup \{d_1, \dots, d_{i-1}\}$ representing a *pattern*

letter → A | B | ... | Z | a | b | ... | z

digit → 0 | 1 | ... | 9

id → letter (letter | digit)*

digits → digit digit*

optional_fraction → . **digits** | ε

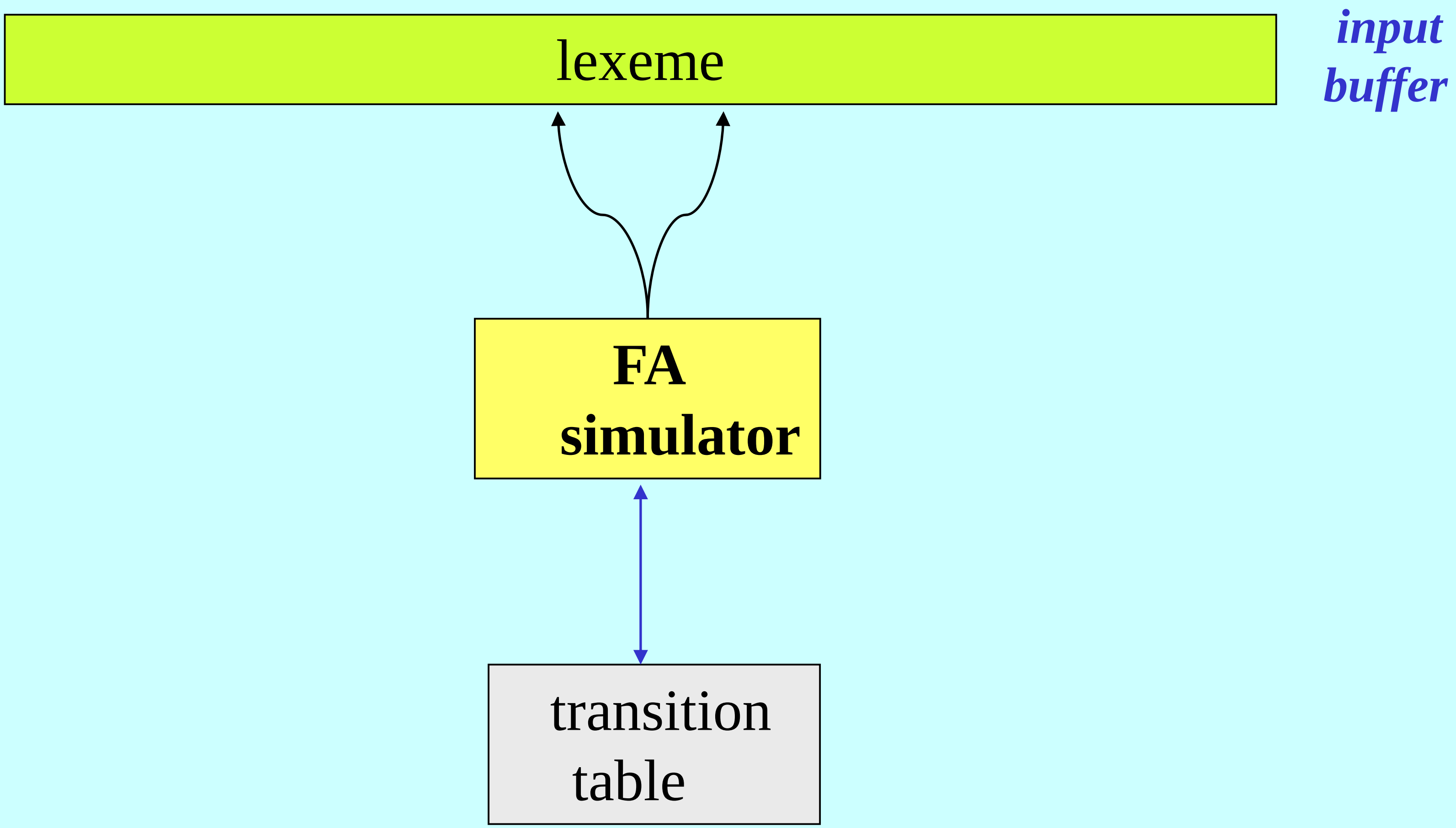
optional_exponent → (E **(** + **|** - **|** ε **) digits) |** ε

num → digits   optional_fraction   optional_exponent

LA: lexical-analyzer generators

- the task of constructing a lexical-analyzer is simple enough to be automated
- a *lexical-analyzer generator* transforms the *specification* of a scanner (*regular definitions*, *actions to be executed when a matching occurs, ...)* into a program implementing a *Finite Automaton* accepting the specified lexemes
- ***Lex*** *(UNIX)* and ***Flex*** *(GNU)* produce *C programs* implementing *FA*
- ***JFlex*** produces *Java programs* implementing *FA*

LA: schematic lexical analyzer



LA: simulating a DFA

- the function ***move (s, c)*** gives the state reached from state ***s*** on input symbol ***c***

```
s = s0 ;
c = nextchar ;
while ( c ≠ eof )
        { s = move (s, c) ;
          c = nextchar ; }
if (s ∈ F) return "accepted" ;
```

LA: simulating an NFA

- the function ***move (S, c)*** gives the set of states reached from the set of states ***S*** on input symbol ***c***

```
S = ε-closure (s0) ;
c = nextchar ;
while ( c ≠ eof )
    { S = ε-closure (move (S, c)) ;
      c = nextchar ; }
if ( S ∩ F ≠ ∅ ) return "accepted" ;
```

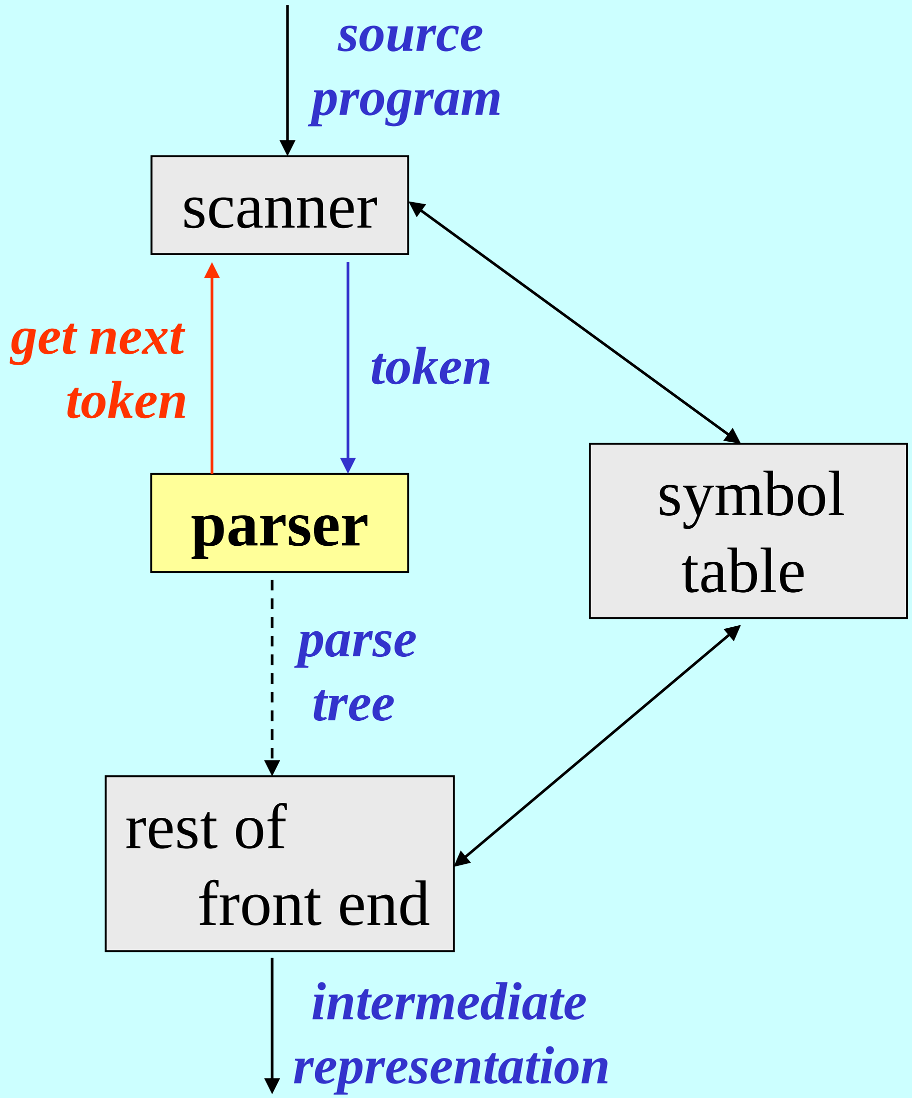
LA: space and time to recognize regular expressions

r : regular expression

x : input string

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

Syntax Analysis



LA: role of a parser

- the parser obtains a string of ***tokens*** from the scanner and
  - verifies that the string can be generated by the grammar for the source language , trying to build a parse tree
  - reports syntax errors and continues processing the input
- ***bottom-up*** parsers build parse trees from the bottom (leaves) to the top (root)
- ***top-down*** parsers build parse trees from the top (root) to the bottom (leaves)

SA: bottom-up parsing (1)

- bottom-up parsing attempts to construct a ***parse tree*** for an input string beginning at the *leaves* (the bottom) and working up towards the *root* in *postorder*
- this construction process ***reduces*** an *input string* to the *start symbol* of a grammar
- at each ***reduction*** step the *right side* of a production is *replaced* by its *left side symbol*, tracing out a *rightmost derivation* in *reverse*

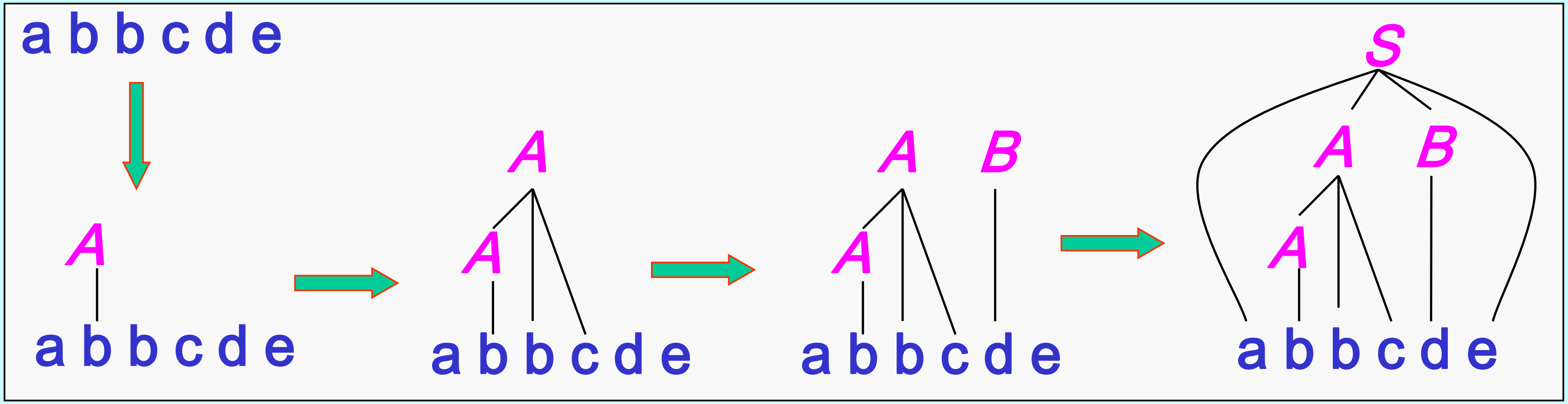
SA: bottom-up parsing (2)

$G = (\{S, A, B\}, \{a, b, c, d, e\}, P, S)$

$P = \{ S \rightarrow a\,A\,B\,e$

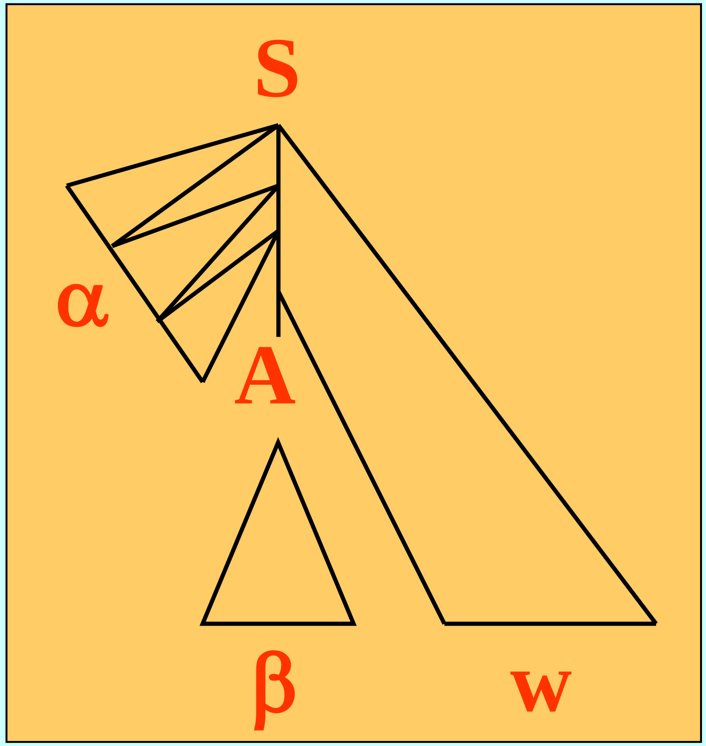
$A \rightarrow A\,b\,c \mid b$

$B \rightarrow d \}$



$S \Rightarrow_{rm} \underline{a\,A\,B\,e} \Rightarrow_{rm} a\,A\,\underline{d}\,e \Rightarrow_{rm} a\,\underline{A\,b\,c}\,d\,e \Rightarrow_{rm} a\,\underline{b}\,b\,c\,d\,e$

SA: handles



- 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 $A \rightarrow \beta$ is a ***handle*** of $\alpha\ \beta\ w$

  ($w \in T^*$ because $A \rightarrow \beta$ is the last applied rule)

SA: shift-reduce parsing (1)

- 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
- the parser
  - ***shifts*** input symbols onto the stack until a handle **β** is on top of the stack
  - then ***reduces*** **β** to the left side of the appropriate production

  until the input is *empty* and the stack contains the *start symbol*

SA: shift-reduce parsing (2)

$G = (\{S, A, B\}, \{a, b, c, d, e\}, P, S)$

$P = \{ S \rightarrow a\,A\,B\,e$

$A \rightarrow A\,b\,c \mid b$

$B \rightarrow d \}$

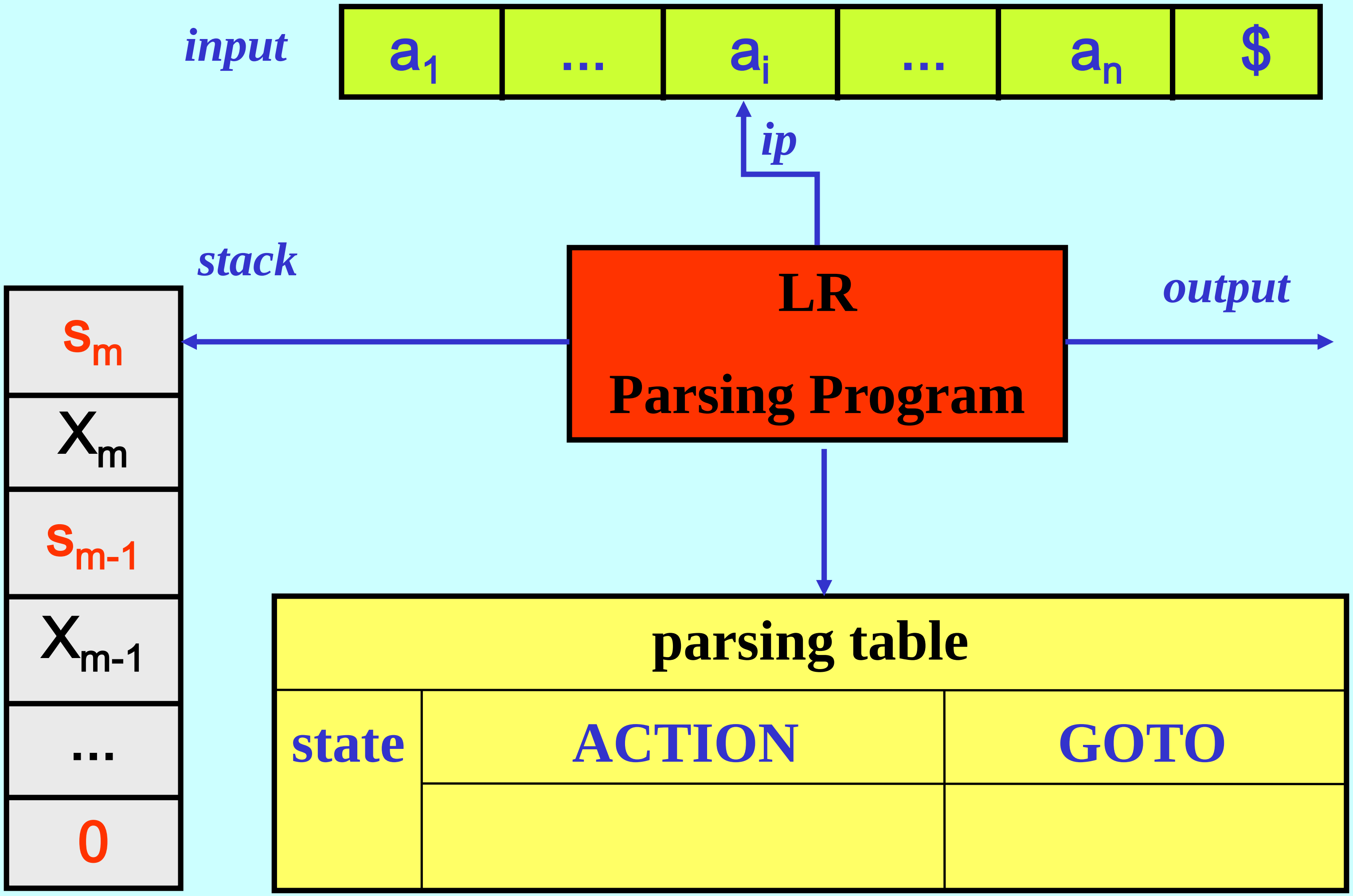
stack	input	action
$	a b b c d e $	shift
$ a	b b c d e $	shift
$ a b	b c d e $	reduce by $A \rightarrow b$
$ a A	b c d e $	shift
$ a A b	c d e $	shift
$ a A b c	d e $	reduce by $A \rightarrow A\,b\,c$
$ a A	d e $	shift
$ a A d	e $	reduce by $B \rightarrow d$
$ a A B	e $	shift
$ a A B e	$	reduce by $S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

SA: actions of a shift-reduce parser

- ***shift***
  - the next input symbol is shifted onto the top of the stack
- ***reduce***
  - the left end of the handle must be located within the stack
  - it must be decided with what non-terminal to replace the handle
- ***accept***
  - parsing is successfully completed
- ***error***
  - a syntax error has occurred

- a strategy for making ***parsing decisions*** is needed

SA: LR parsing



SA: LR parsing program

```
push 0 onto the stack ;
set ip to point to the first input symbol ;
repeat
  { let s be the state on top of the stack and a the symbol pointed by ip ;
    if ( ACTION[s , a] = shift t )
          { shift a onto the stack ;
            push state t onto the stack ;
            advance ip to the next input symbol }
    else if ( ACTION[s , a] = reduce A → β )
                { pop 2 * |β| symbols off the stack ;
                  let u be the state now on top of the stack ;
                  push A onto the stack ;
                  push GOTO[u , A] onto the stack ;
                  output the production A → β ; }
          else if ( ACTION[s , a] = accept )
                    return ;
                else error ;
  }
forever
```

SA: an LR parser for grammar G_0

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

$P = \{ \; E \rightarrow E + T \mid T \quad (1, 2)$

$T \rightarrow T * F \mid F \quad (3, 4)$

$F \rightarrow (E) \mid \text{id} \; \} \quad (5, 6)$

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

SA: a parsing table for grammar G_0

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			

SA: moves of an LR parser for grammar G_0

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

SA: LR(0) items (1)

- an ***LR(0) item*** of a CFG grammar G is a production of G with a ***dot*** at some position of the right side

$$A \rightarrow X\,Y\,Z$$

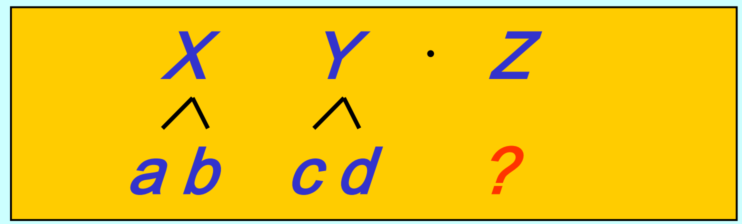
$$A \rightarrow \cdot X\,Y\,Z$$
$$A \rightarrow X \cdot Y\,Z$$
$$A \rightarrow X\,Y \cdot Z$$
$$A \rightarrow X\,Y\,Z \cdot$$

- an ***item*** indicates how much of a production we have seen at a given point in the parsing process

SA: LR(0) items (2)

- the ***dot*** indicates the current position of the parser

$$X \rightarrow a\,b$$
$$Y \rightarrow c\,d$$
$$A \rightarrow X\,Y \cdot Z$$



- an ***item*** with the dot at the end is called ***complete***
  - all the right side of the production has been recognized

SA: viable prefixes

- a ***viable prefix*** of a string γ is a prefix that can appear on the stack of a shift-reduce parser
  - it does not continue past the right end of the rightmost ***handle*** of γ
- we say that item $\mathbf{A} \rightarrow \beta_1 \cdot \beta_2$ is ***valid*** for a *viable prefix* $\alpha\,\beta_1$ if there is a derivation

$$\mathbf{S} \Rightarrow^*_{\mathbf{rm}} \alpha\,\mathbf{A}\,\mathbf{w} \Rightarrow_{\mathbf{rm}} \alpha\,\beta_1\,\beta_2\,\mathbf{w}$$

- if $\mathbf{A} \rightarrow \beta \cdot$ is a ***valid complete item*** for a *viable prefix* $\alpha\,\beta$, then $\mathbf{S} \Rightarrow^*_{\mathbf{rm}} \alpha\,\mathbf{A}\,\mathbf{w} \Rightarrow_{\mathbf{rm}} \alpha\,\beta\,\mathbf{w}$ and therefore $\mathbf{A} \rightarrow \beta$ is a ***handle*** of $\alpha\,\beta\,\mathbf{w}$

SA: recognizing viable prefixes (1)

- the sets of *viable prefixes* are regular languages
- the *FA* that represent them 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
- a *DFA* equivalent to such an NFA will have states corresponding to *sets of LR(0) items* and transitions labeled by *symbols in viable prefixes*

SA: recognizing viable prefixes (2)

- the function *closure(I)* finds the set of *LR(0) items* that recognize the same viable prefix
- the function *goto(I, X)* finds the set of *LR(0) items* that is reached from the set *I* with symbol *X*

```
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) ;
```

SA: recognizing viable prefixes (3)

- given a CFG grammar ***G = (N, T, P, S)*** , the function *items(G)* constructs the collection $C = \{I_0, I_1, \ldots, I_n\}$ of DFA states

```
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 ;
```

SA: construction of a DFA recognizing viable prefixes (1)

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

$P = \{ S \rightarrow (L) \mid x$

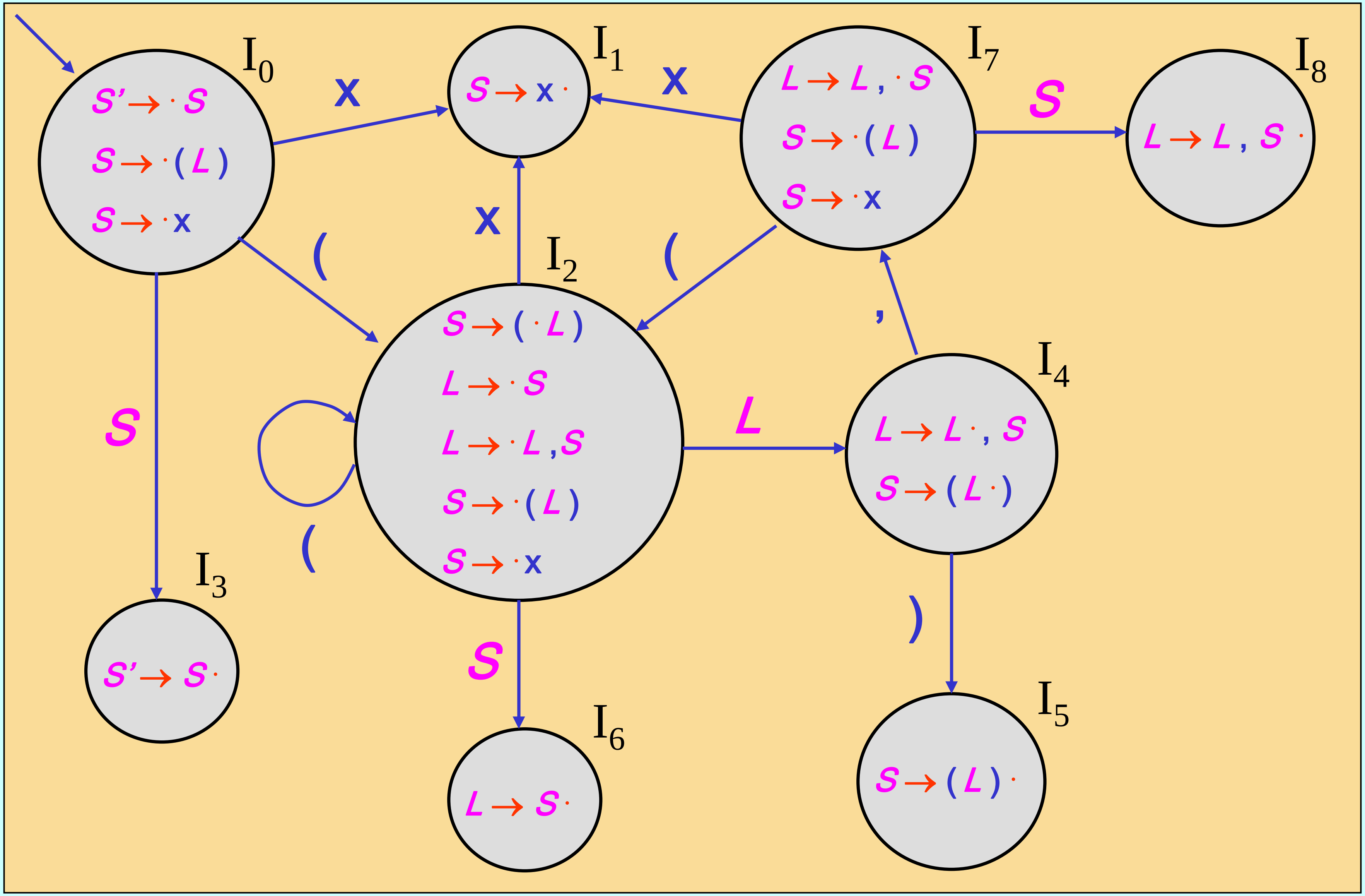
$L \rightarrow S \mid L , S \}$

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

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

FL&C

SA: construction of a DFA recognizing viable prefixes (2)



SA: LR(0) parsing tables

- the function *lr0Table(G)* constructs the *LR(0) parsing table* for the CFG *G*

void lr0Table (*CFG* G);

let $\{I_0, I_1, \ldots, I_n\}$ be the result of items (G) ;

for (i = 0 to n)

if ($A \rightarrow \alpha \cdot a\, \beta$ is in I_i and $a \in T$ and goto $(I_i, a) = I_j$)

set ACTION[**i** , ***a***] to ***shift j*** ;

if ($A \rightarrow \alpha \cdot$ is in I_i and $A \neq S'$)

set ACTION[**i** , ***a***] to ***reduce*** $A \rightarrow \alpha$ for all ***a*** in $T \cup \{\$\}$;

if ($S' \rightarrow S \cdot$ is in I_i)

set ACTION[**i** , **$**] to ***accept*** ;

if (goto $(I_i, X) = I_j$ and $X \in N$) set GOTO[**i** , **X**] to **j** ;

FL&C

SA: construction of an LR(0) parsing table for grammar G_1

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		

- the initial state of the parser is the one constructed from the set of items containing $S' \rightarrow \cdot S$

SA: moves of an LR(0) parser for grammar G_1

stack	input	action	
0	(x , (x) , x) $	s2	
0 (2	x , (x) , x) $	s1	
0 (2 x 1	, (x) , x) $	r2	*S* → x
0 (2 *S* 6	, (x) , x) $	r3	*L* → *S*
0 (2 *L* 4	, (x) , x) $	s7	
0 (2 *L* 4 , 7	(x) , x) $	s2	
0 (2 *L* 4 , 7 (2	x) , x) $	s1	
0 (2 *L* 4 , 7 (2 x 1	) , x) $	r2	*S* → x
0 (2 *L* 4 , 7 (2 *S* 6	) , x) $	r3	*L* → *S*
0 (2 *L* 4 , 7 (2 *L* 4	) , x) $	s5	
0 (2 *L* 4 , 7 (2 *L* 4) 5	, x) $	r1	*S* → (*L*)
0 (2 *L* 4 , 7 *S* 8	, x) $	r4	*L* → *L* , *S*
0 (2 *L* 4	, x) $	s7	
0 (2 *L* 4 , 7	x) $	s1	
0 (2 *L* 4 , 7 x 1	) $	r2	*S* → x
0 (2 *L* 4 , 7 *S* 8	) $	r4	*L* → *L* , *S*
0 (2 *L* 4	) $	s5	
0 (2 *L* 4) 5	$	r1	*S* → (*L*)
0 *S* 3	$	accept	

SA: conflicts in parsing tables

- parsing table entries defined in multiple ways determine parsing action conflicts
  - ***shift / reduce***
    - some entry in the ACTION table contains both a shift and a reduce action
  - ***reduce / reduce***
    - some entry in the ACTION table contains more reduce actions

FL&C

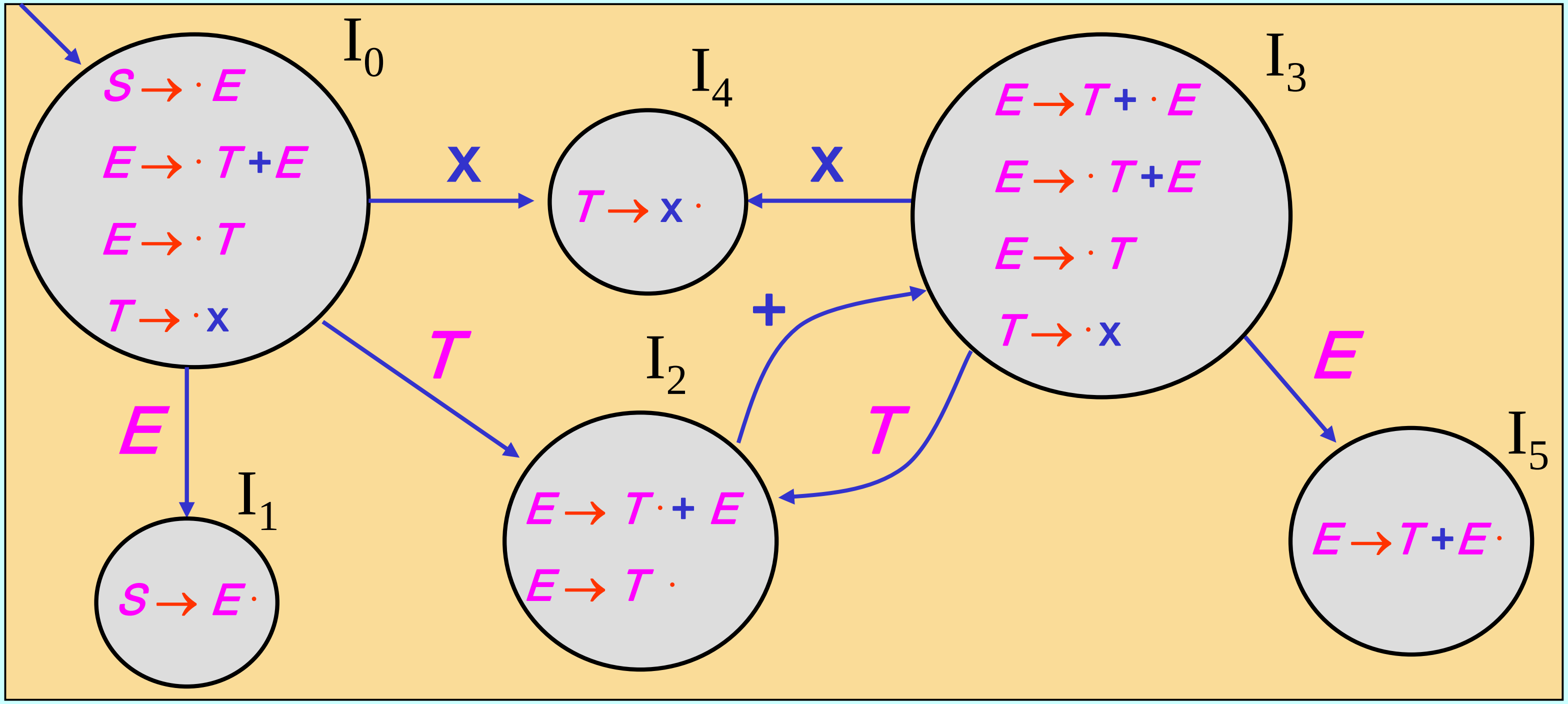
SA: construction of an LR(0) parsing table for grammar G_2 (1)

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

$P = \{ \quad S \rightarrow E$ (0)

$E \rightarrow T + E \mid T$ (1, 2)

$T \rightarrow x \}$ (3)



SA: construction of an LR(0) parsing table for grammar G_2 (2)

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

SA: LR(0) grammars

- a ***grammar G*** is ***LR(0)*** if the ACTION table generated by function *lr0Table(G)* does not comprise conflicts
  - 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 (avoiding *reduce/reduce* conflicts)
    - no other item in the set has a terminal symbol immediately at the right of the dot (avoiding *shift/reduce* conflicts)
- *LR(0)* grammars are non-ambiguous

SA: LR(0) parsers

- an ***LR(0)*** parser
  - scans the input from left to right (***L***)
  - constructs a rightmost derivation in reverse (***R***)
  - uses ***0*** lookahead input symbols in making parsing decisions
- the class of languages that can be parsed using ***LR(0)*** parsers is a *proper subset* of the *deterministic* CFL's

SA: LR(k) parsers (1)

- more powerful parsers can be constructed when more than ***0*** lookahead input symbols are used in making parsing decisions
- function *lr0Table(G)* sets ACTION[**i** , ***a***] to ***reduce*** **A → α** for all ***a*** in ***T*** ∪ {$} , when **A → α ·** is in I_i
- if the function would be informed about which input symbols *after the dot* (that is after symbol **A**) are valid, it could set the ***reduce*** **A → α** action for them only, thus avoiding several potential conflicts

SA: FIRST and FOLLOW sets

- with respect to 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
  - ***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)***

SA: algorithm to compute FIRST, FOLLOW and nullable

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

SA: computation of FIRST, FOLLOW and nullable for grammar G_2

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

$P = \{ \quad S \rightarrow E$ (0)

$E \rightarrow T + E \mid T$ (1,2)

$T \rightarrow x \; \}$ (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	+ $

SA: Simple LR (SLR) parsing tables

- the function *slrTable(G)* constructs the *SLR parsing table* for the CFG *G*

```
void slrTable (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 FOLLOW(A);
    if ( S' → S ·  is in Ii )
      set  ACTION[i , $]  to  accept ;
    if ( goto (Ii , A) = Ij  and  A∈N ) set  GOTO[i , A]  to  j ;
```

SA: construction of an SLR parsing table for grammar G_2

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

FL&C

SA: construction of an SLR parsing table for grammar G_0 (1)

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

$P = \{ \quad E \rightarrow E + T \mid T \qquad (1, 2)$

$T \rightarrow T * F \mid F \qquad (3, 4)$

$F \rightarrow (E) \mid id \ \} \qquad (5, 6)$

$G_0' = (\{ E', E, T, F \}, \{ id, +, *, (,) \}, P', E')$

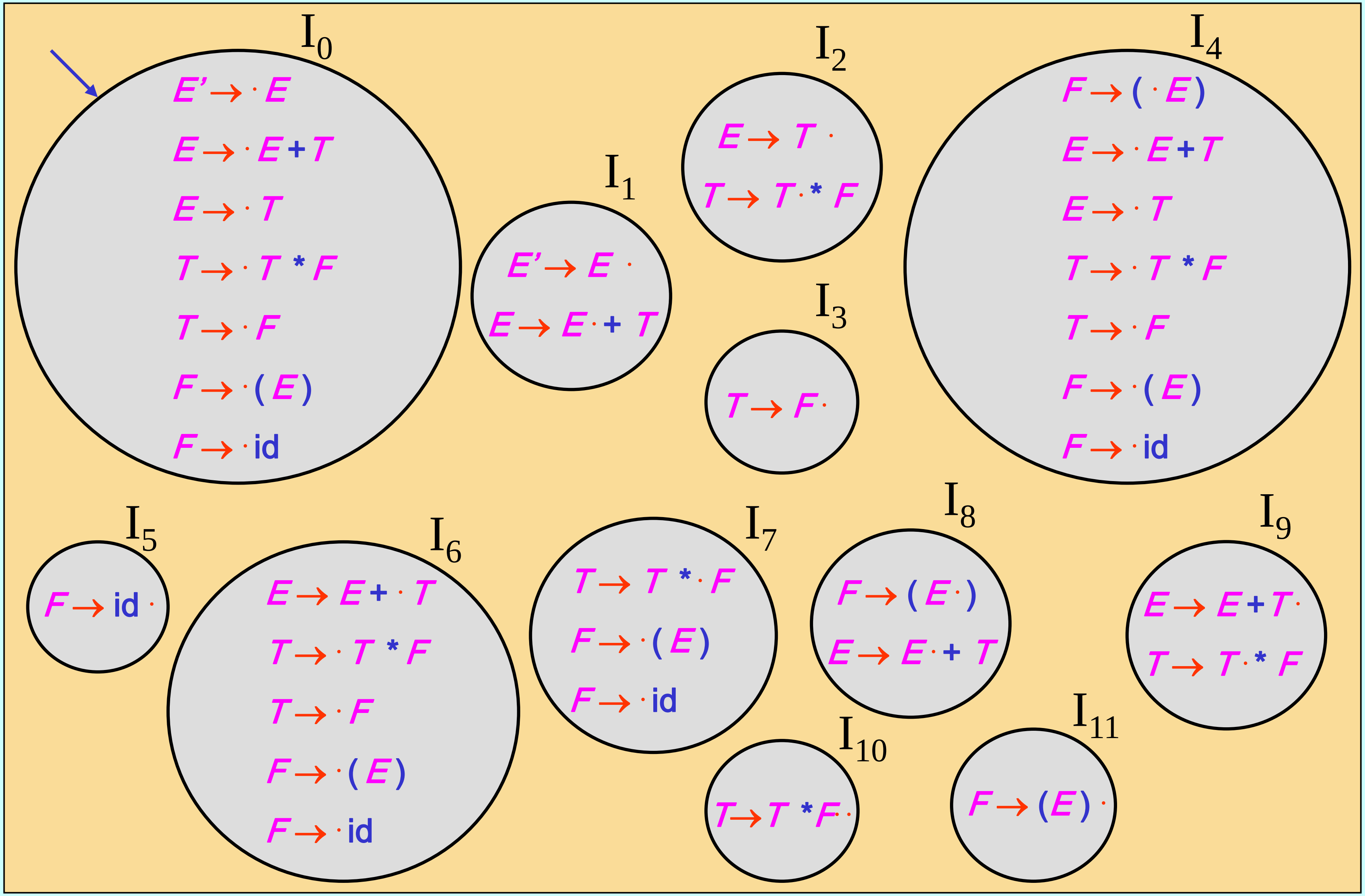
$P' = \{ \quad E' \rightarrow E \qquad (0)$

$E \rightarrow E + T \mid T \qquad (1, 2)$

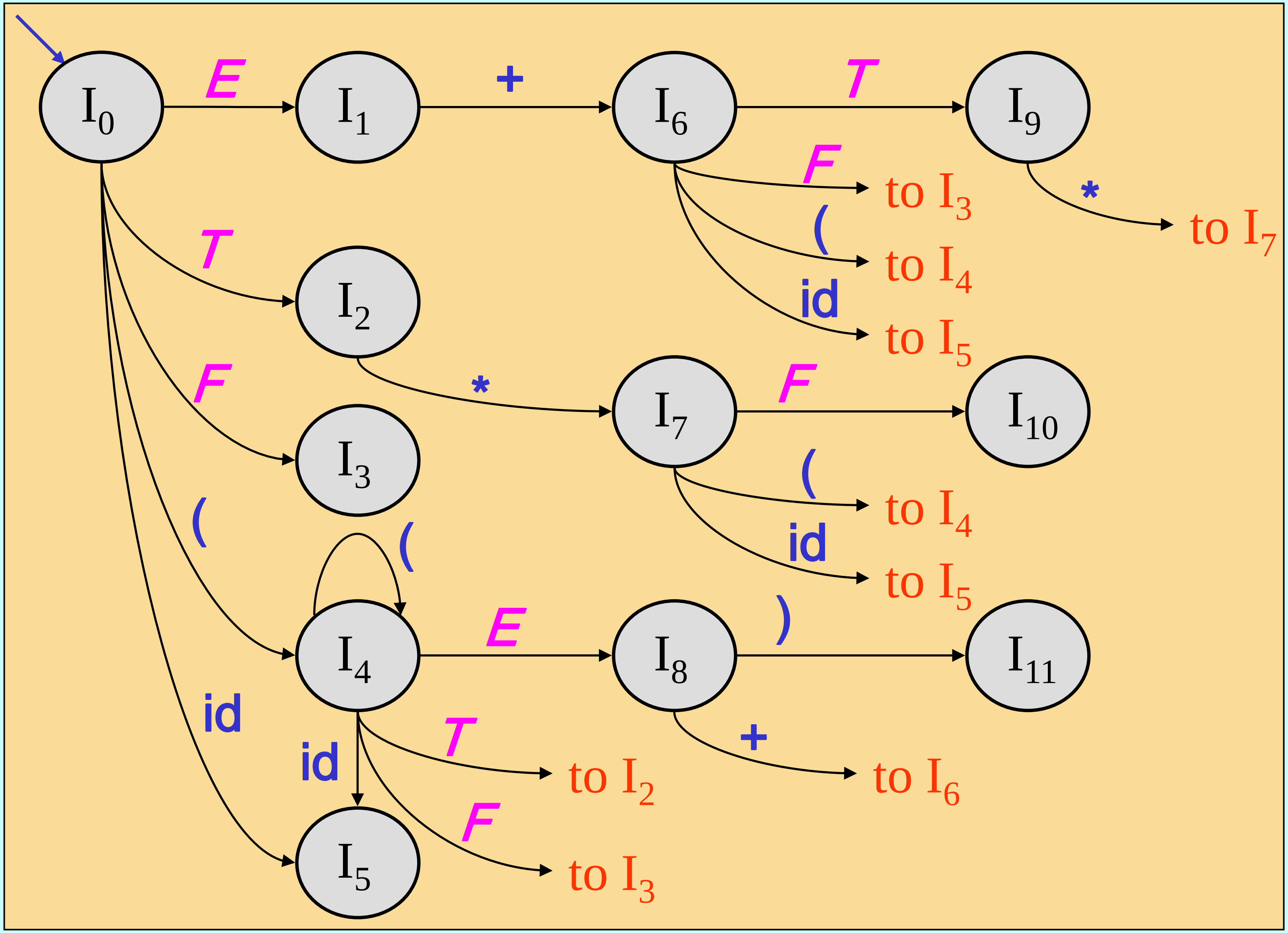
$T \rightarrow T * F \mid F \qquad (3, 4)$

$F \rightarrow (E) \mid id \ \} \qquad (5, 6)$

SA: construction of an SLR parsing table for grammar G_0 (2)



SA: construction of an SLR parsing table for grammar G_0 (3)



SA: construction of an SLR parsing table for grammar G_0 (4)

G_0 = ({ *E* , *T* , *F* } , { id , + , ∗ , (,) } , *P* , *E*)

P = { *E* → *E* + *T* | *T* (1, 2)

T → *T* ∗ *F* | *F* (3, 4)

F → (*E*) | 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	$ +) *

SA: construction of an SLR parsing table for grammar G_0 (5)

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			

SA: LR(k) parsers (2)

- ***FOLLOW(A)*** is the set of terminals that can immediately follow ***A*** in any string generated by a given grammar ***G***
- it takes into account all the contexts where ***A*** can appear
- by taking into account 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)***, thus avoiding further potential conflicts

SA: LR(1) items

- an ***LR(1) item*** of a CFG grammar G is a production of G with a ***dot*** at some position of the right side, and a ***lookahead*** (*terminal* or *$*) symbol
- an *LR(1) item* $[A \rightarrow \alpha \cdot, a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is ***a***
- we say item $[A \rightarrow \beta_1 \cdot \beta_2 , a]$ is ***valid*** for a viable prefix $\alpha \beta_1$ if :
  - there is a derivation $S \Rightarrow^*_{rm} \alpha A w \Rightarrow^*_{rm} \alpha \beta_1 \beta_2 w$
  - either ***a*** is the first symbol of **w**, or **w** is **ε** and ***a*** is ***$***

SA: recognizing viable prefixes (1)

- the ***valid LR(1) items*** of a CFG grammar are the *states* of a NFA recognizing viable prefixes
- a *DFA* equivalent to such a NFA will have states corresponding to *sets of LR(1) items* and transitions labeled by the *symbols of the viable prefixes*

FL&C

SA: recognizing viable prefixes (2)

- the function *closure1(I)* finds the set of *LR(1) items* that recognize the same viable prefix
- the function *goto1(I, X)* finds the set of *LR(1) items* that is reached from the set *I* with symbol *X*

```
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) ;
```

SA: recognizing viable prefixes (3)

- given a CFG grammar ***G = (N, T, P, S)*** , the function *items1(G)* constructs the collection $C = \{I_0 , I_1 , \ldots , I_n\}$ of DFA states

```
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 ;
```

SA: construction of a DFA that recognizes viable prefixes (1)

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

$P = \{ S \rightarrow V = E \mid E$

$E \rightarrow V$

$V \rightarrow x \mid *E \}$

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

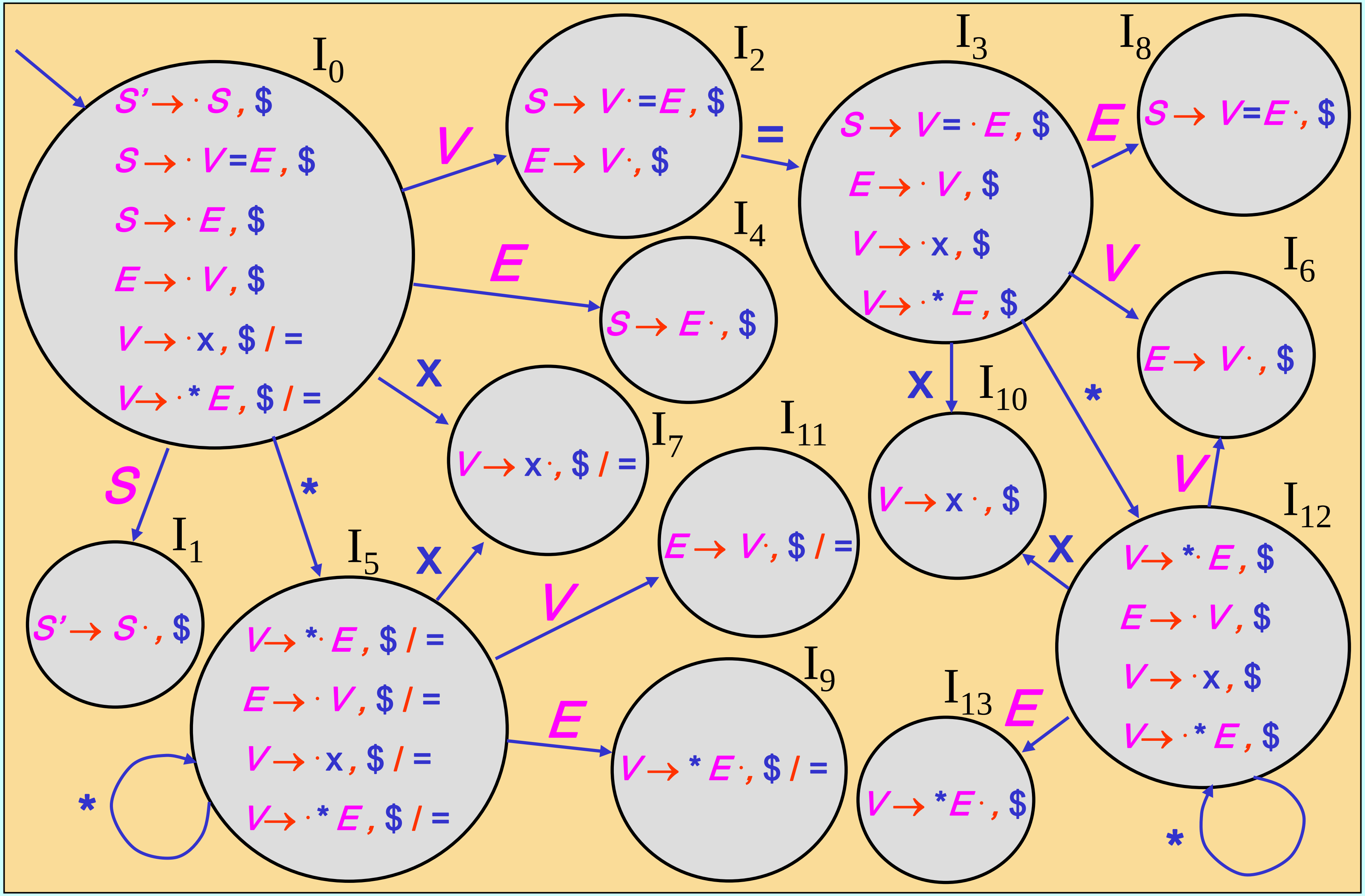
$P' = \{ S' \rightarrow S$ (0)

$S \rightarrow V = E \mid E$ (1, 2)

$E \rightarrow V$ (3)

$V \rightarrow x \mid *E \}$ (4, 5)

SA: construction of a DFA that recognizes viable prefixes (2)



SA: LR(1) parsing tables

- the function *lr1Table(G)* constructs the *LR(1) parsing table* for the CFG *G*

```
void lr1Table (CFG G);
  let {I0 , I1 , … , In} be the result of items1 (G) ;
  for ( i = 0 to n )
    if ( [A → α · a β , b] is in Ii and a∈T and goto1 (Ii , a) = Ij )
       set ACTION[i , a] to shift j ;
    if ( [A → α · , a] is in Ii and A ≠ S' )
       set ACTION[i , a] to reduce A → α ;
    if ( [S' → S · , $] is in Ii )
       set ACTION[i , $] to accept ;
    if ( goto1 (Ii , A) = Ij and A∈N ) set GOTO[i , A] to j ;
```

SA: construction of an LR(1) parsing table for grammar G_3

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			

SA: LR(1) grammars

- a ***grammar G*** is ***LR(1)*** if the ACTION table generated by function *lr1Table(G)* does not comprise conflicts
  - if any *set of LR(1) items* generated by function *items1(G)* contains a ***complete item*** $[A \rightarrow \alpha \cdot, a]$, (originating a *reduce* action) then
    - no other complete item in the set has ***a*** as lookahead symbol (avoiding *reduce/reduce* conflicts)
    - no other item in the set has ***a*** immediately at the right of the dot (avoiding *shift/reduce* conflicts)
- *LR(1)* grammars are non-ambiguous

SA: LR(1) parsers

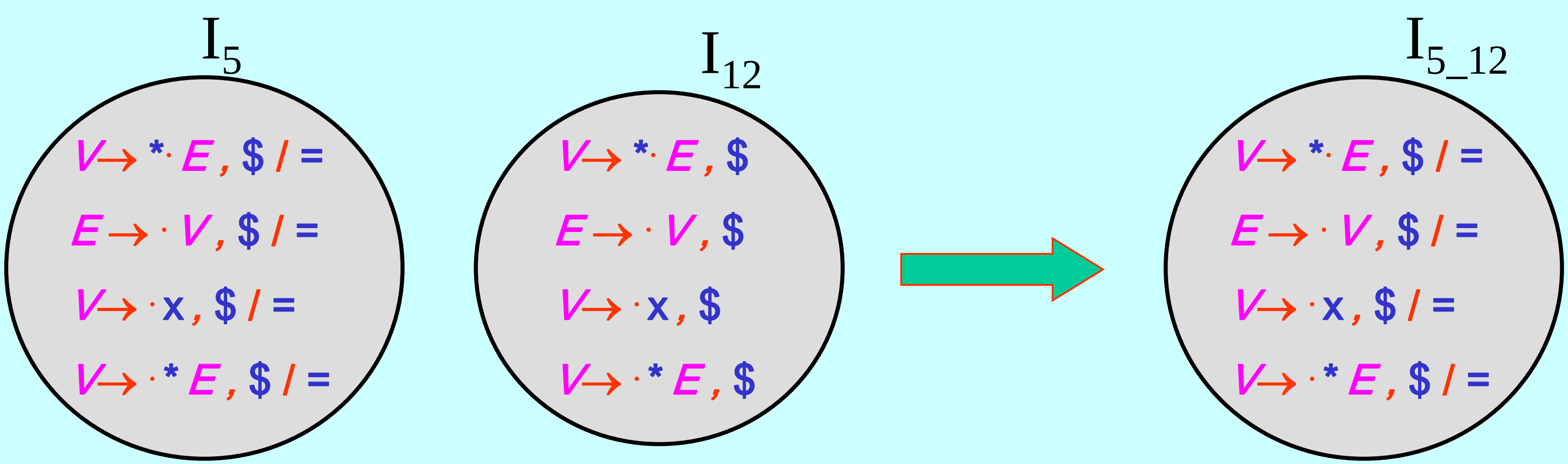
- an ***LR(1)*** parser
  - scans the input from left to right (***L***)
  - constructs a rightmost derivation in reverse (***R***)
  - uses ***1*** lookahead input symbols in making parsing decisions
- the class of languages that can be parsed using ***LR(1)*** parsers is exactly the class of the *deterministic* CFL's

SA: Lookahead LR(1) parsers (LALR)

- *LR(1) parsing tables* can be *very large* (several thousand states) for grammars generating common programming languages
- *SLR parsing tables* for the same languages are *much smaller* (several hundred states) but can contain *conflicts*
- *LALR(1) parsing tables* have the same states of *SLR tables* and can conveniently express most programming languages

SA: LALR(1) parsing tables

- two *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 *sets of LR(1) items* having the same ***core***



FL&C

SA: construction of an LALR(1) parsing table for grammar G_3

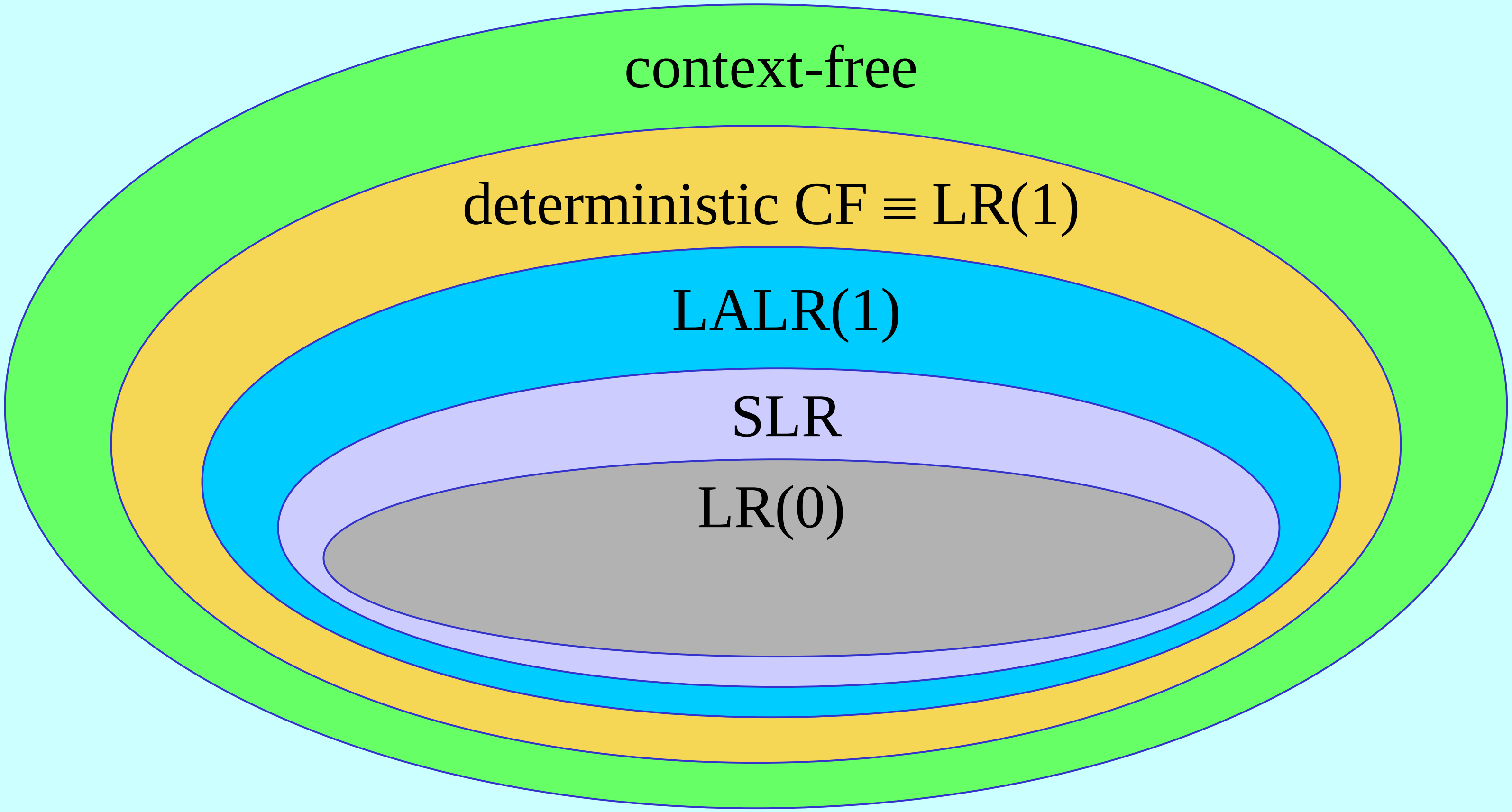
state	ACTION				GOTO		
	x	*	=	$	*S*	*E*	*V*
0	s7_10	s5_12			1	4	2
1				acc			
2			s3	r3			
3	s7_10	s5_12				8	6_11
4				r2			
5_12	s7_10	s5_12				9_13	6_11
6_11			r3	r3			
7_10			r4	r4			
8				r1			
9_13			r5	r5			

- grammar G_3 is *LALR(1)* but it is not *SLR*
  - FOLLOW(*E*) = { = , $ }
  - in the SLR table: ACTION[2,=] = **s3 , r3**

SA: conflicts in LALR(1) parsing tables

- the merging of states with common cores can never produce a *shift/reduce* conflict which was not present in one of the original states
  - shift actions depend only on the core, not the lookahead
- it is possible that merging will produce a *reduce/reduce* conflict
- the class of languages that can be parsed using ***LALR(1)*** parsers is a *proper subset* of the *deterministic* CFL’s

SA: hierarchy of context-free languages



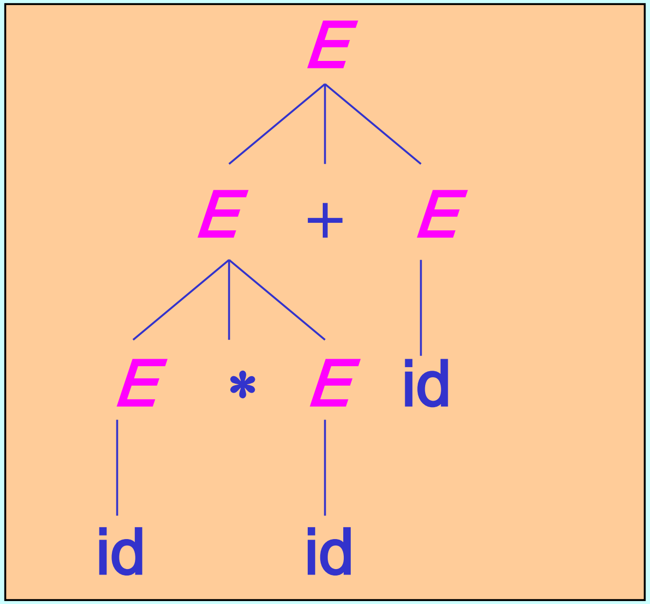
SA: using ambiguous grammars

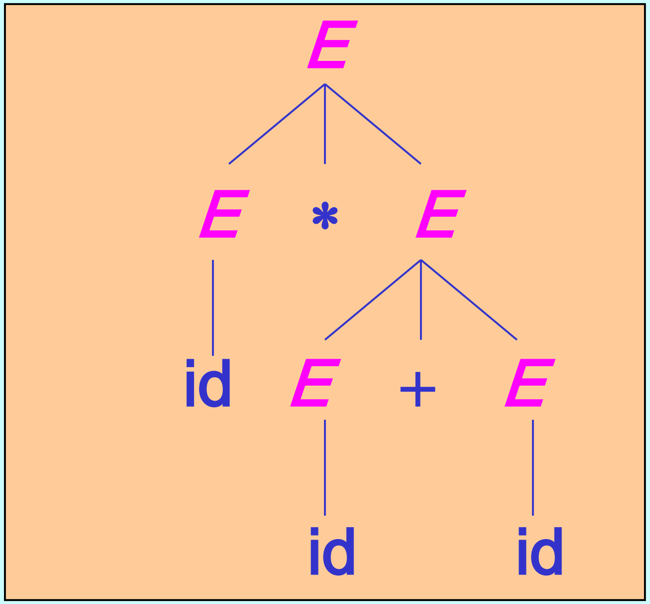
- ambiguous grammars are not *LR(k)*
- some ambiguous grammars provide *shorter, more natural specifications* than any equivalent unambiguous grammar
- in some cases disambiguating rules, such as *precedence* and *associativity*, can be specified
- the resulting parser can be more *efficient*
- ambiguous constructs should be used *sparingly* and in a strictly controlled fashion

SA: LR parsing of ambiguous grammar G_4 (1)

$G_4 = (\{ E \}, \{ id, +, *, (,) \}, P, E)$

$P = \{ E \rightarrow E + E \mid E * E \mid (E) \mid id \}$ (1, 2, 3, 4)





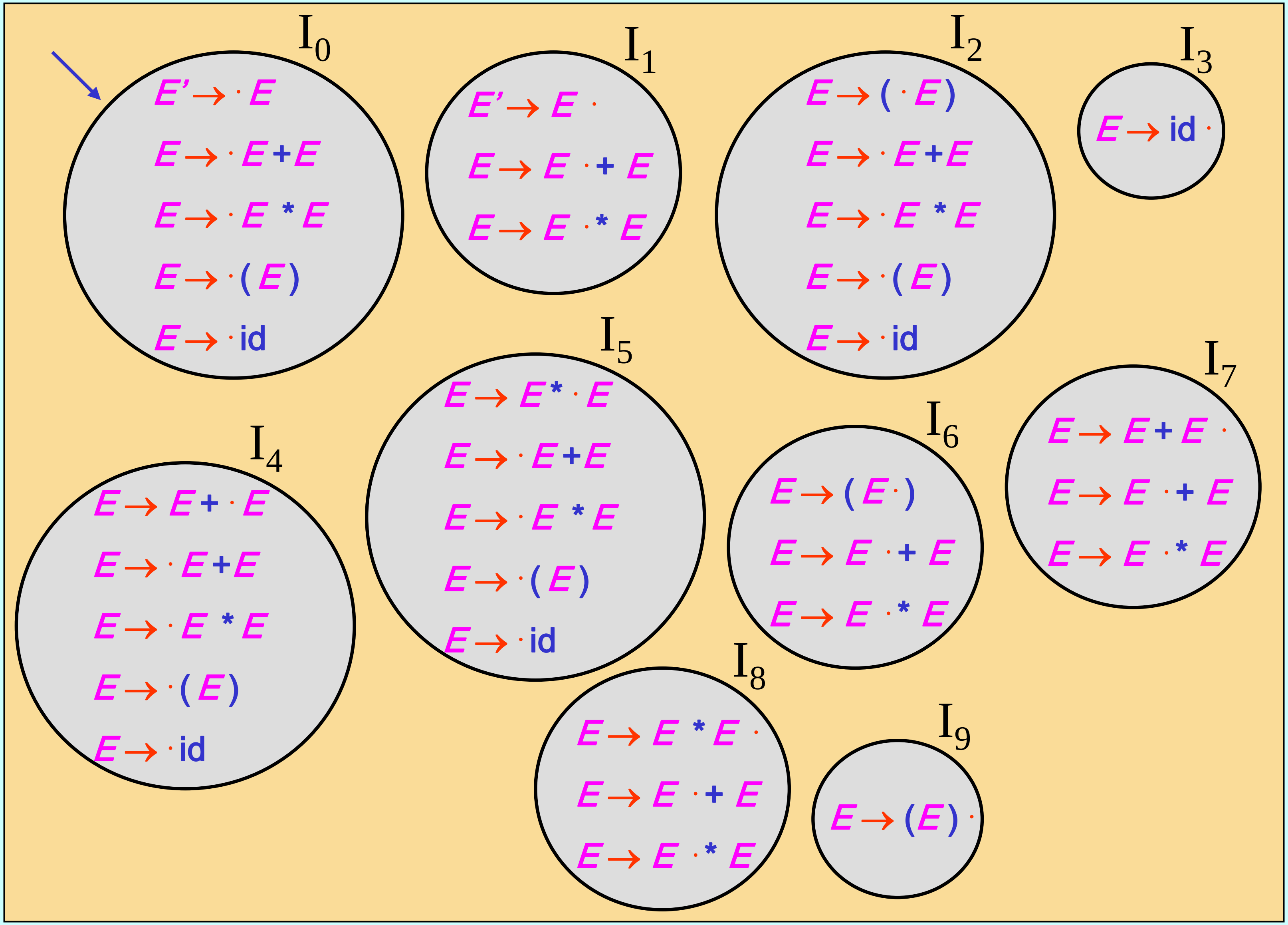
$G_4' = (\{ E', E \}, \{ id, +, *, (,) \}, P', E')$

$P' = \{ E' \rightarrow E$ (0)

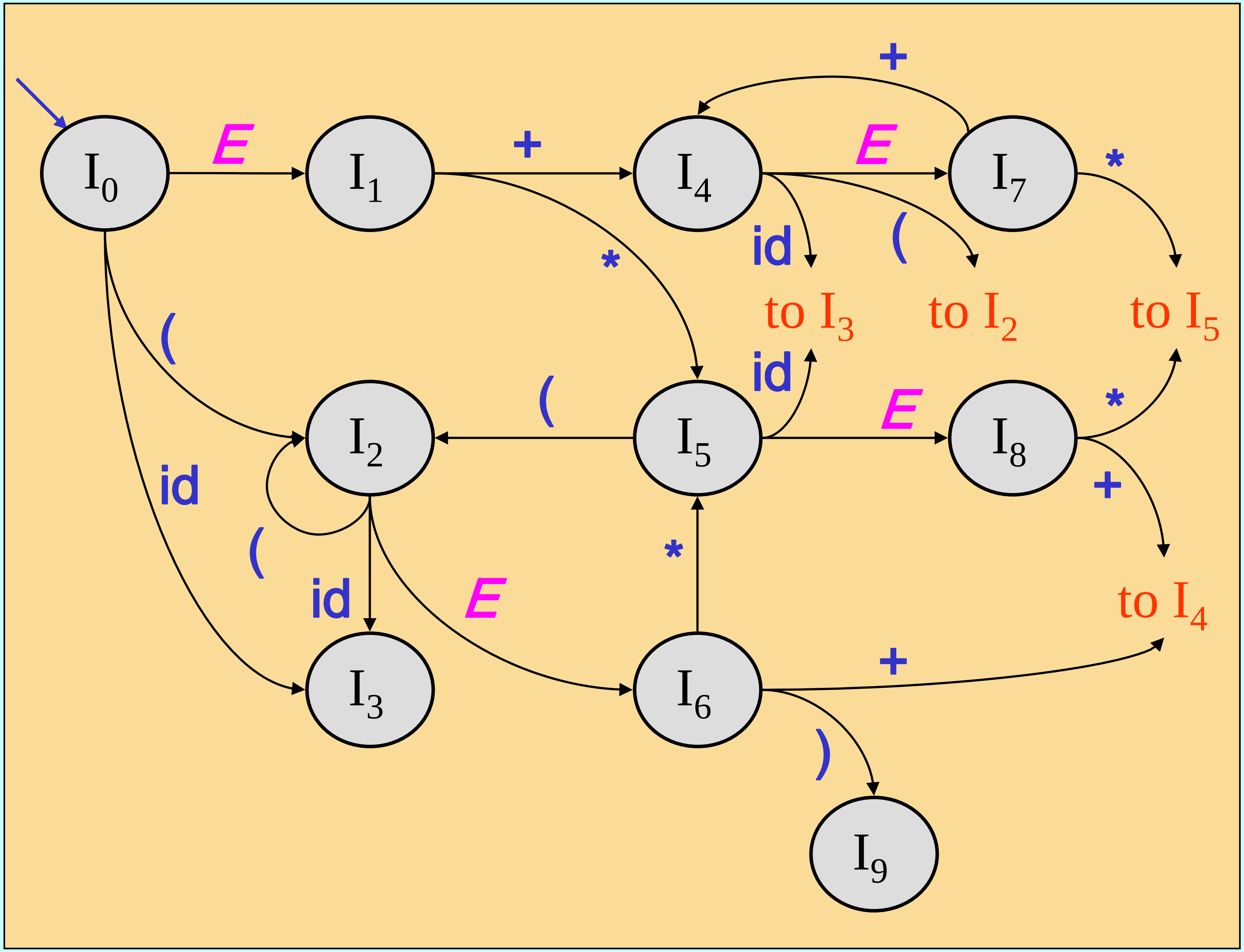
$E \rightarrow E + E \mid E * E \mid (E) \mid id \}$ (1, 2, 3, 4)

FL&C

SA: LR parsing of ambiguous grammar G_4 (2)



SA: LR parsing of ambiguous grammar G_4 (3)



FL&C

SA: LR parsing of ambiguous grammar G_4 (4)

FOLLOW(*E*) = { + , * ,) , $ }

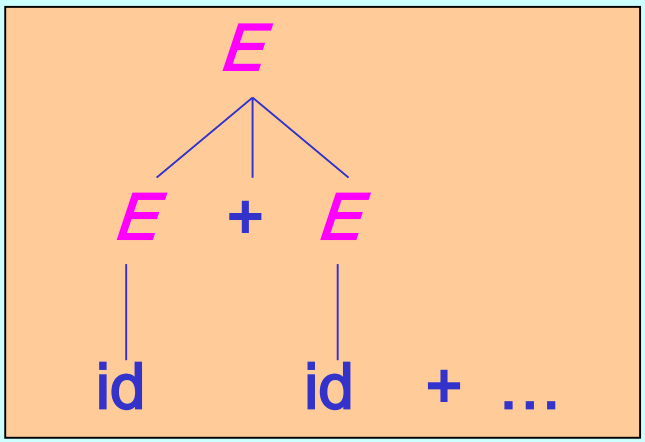
state	ACTION						GOTO
	id	+	*	(	)	$	*E*
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		s4 , r1	s5 , r1		r1	r1	
8		s4 , r2	s5 , r2		r2	r2	
9		r3	r3		r3	r3	

shift / reduce conflicts

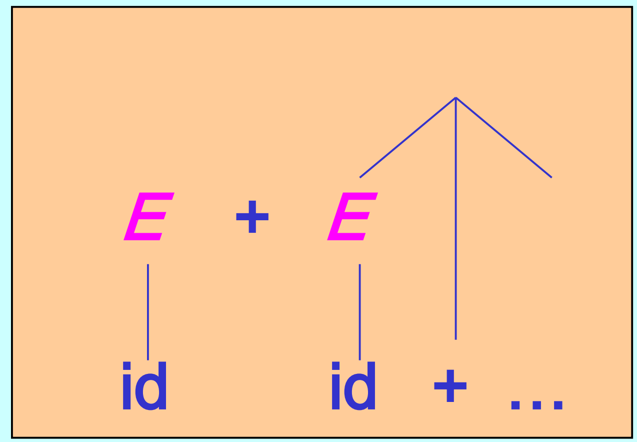
FL&C

SA: resolving conflicts by associativity directives (1)

- conflict in ACTION[7 , +] = **s4 , r1** is due to the items $E \rightarrow E + E \cdot$ and $E \rightarrow E \cdot + E$
- the top of the stack is $E + E$ and the next input symbol is **+**



parse tree produced by *reducing* (**+** is assumed to be ***left-associative***)

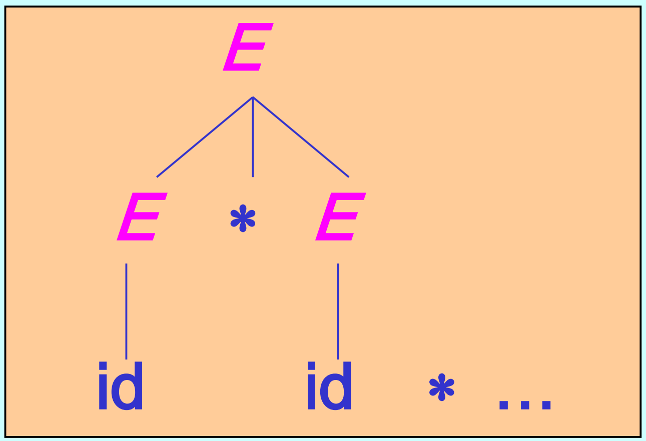


parse tree produced by *shifting* (**+** is assumed to be ***right-associative***)

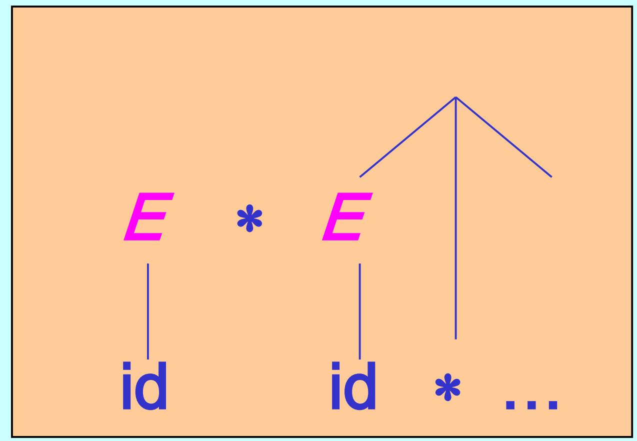
FL&C

SA: resolving conflicts by associativity directives (2)

- conflict in ACTION[**8** , *] = **s5 , r2** is due to the items $E \rightarrow E * E \cdot$ and $E \rightarrow E \cdot * E$
- the top of the stack is $E * E$ and the next input symbol is *



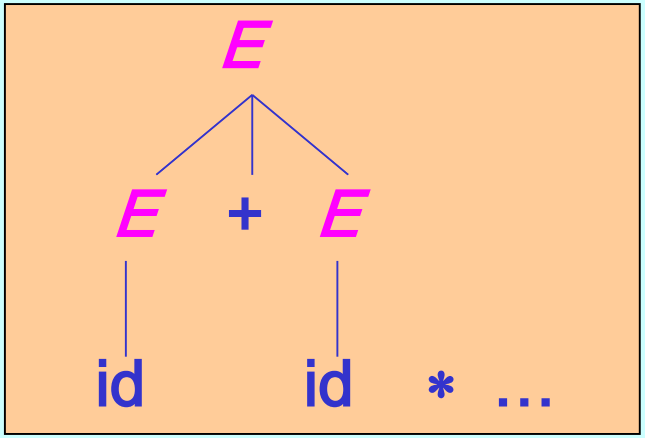
parse tree produced by *reducing* (* is assumed to be ***left-associative***)



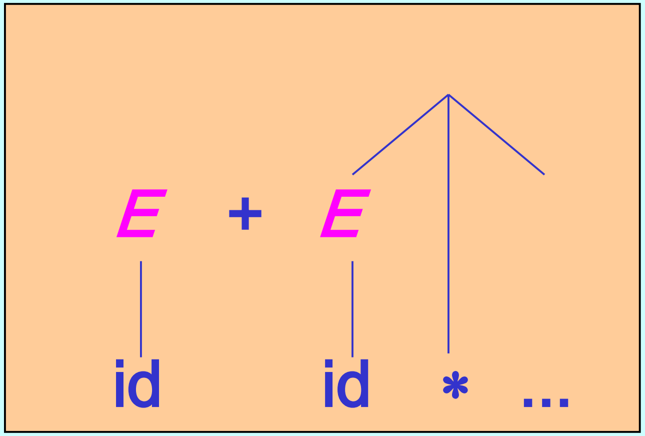
parse tree produced by *shifting* (* is assumed to be ***right-associative***)

SA: resolving conflicts by precedence directives (1)

- conflict in ACTION[7 , *] = **s5 , r1** is due to the 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 *



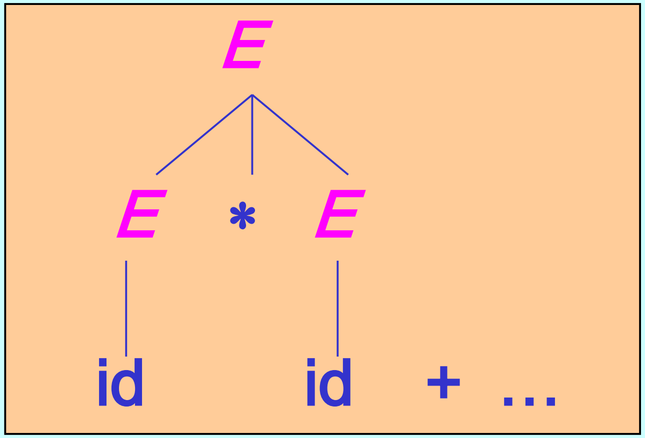
parse tree produced by *reducing* (+ takes ***precedence*** over *)



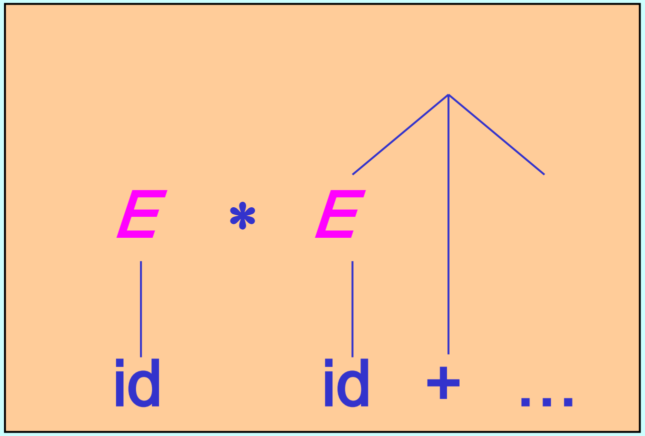
parse tree produced by *shifting* (* takes ***precedence*** over +)

SA: resolving conflicts by precedence directives (2)

- conflict in ACTION[**8** , +] = **s4 , r2** is due to the items $E \rightarrow E * E \cdot$ and $E \rightarrow E \cdot + E$
- the top of the stack is $E * E$ and the next input symbol is **+**



parse tree produced by *reducing* (***** takes ***precedence*** over **+**)



parse tree produced by *shifting* (**+** takes ***precedence*** over *****)

FL&C

SA: resolving conflicts by associativity and precedence directives

* and + are ***left-associative***

* takes ***precedence*** over +

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	

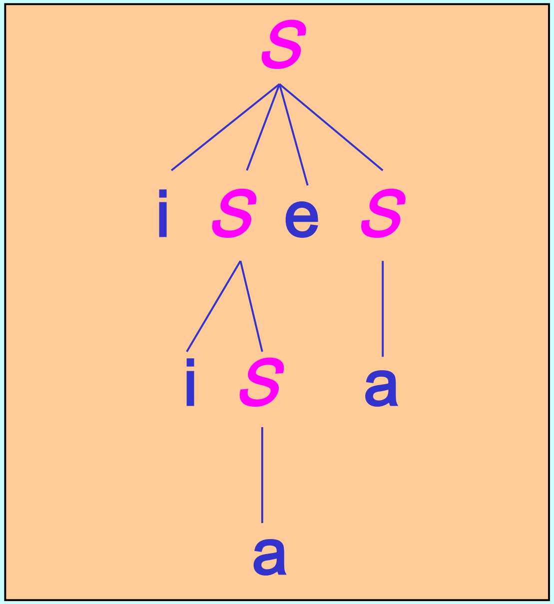
SA: LR parsing of ambiguous grammar G_5 (1)

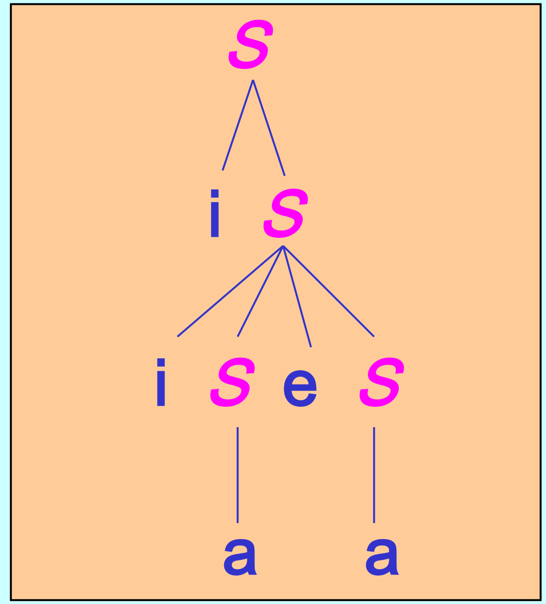
$G_5 = (\{ S \}, \{ i, e, a \}, P, S)$

$P = \{ S \rightarrow i\ S\ e\ S \mid i\ S \mid a \}$ (1, 2, 3)

i : if exp then

e : else



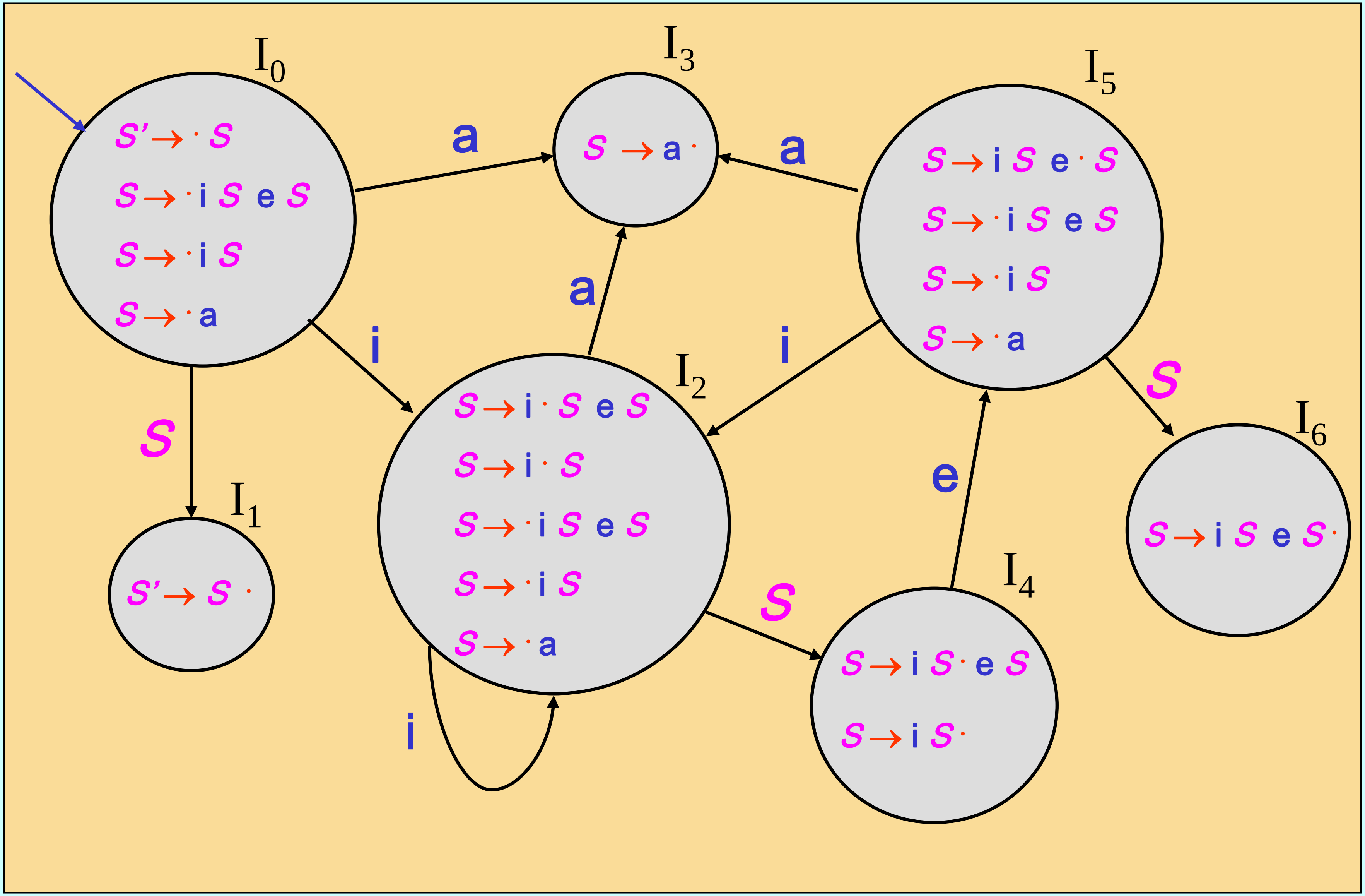


$G_5' = (\{ S', S \}, \{ i, e, a \}, P', S')$

$P' = \{ S' \rightarrow S$ (0)

$S \rightarrow i\ S\ e\ S \mid i\ S \mid a \}$ (1, 2, 3)

SA: LR parsing of ambiguous grammar G_5 (2)



FL&C

SA: LR parsing of ambiguous grammar G_5 (3)

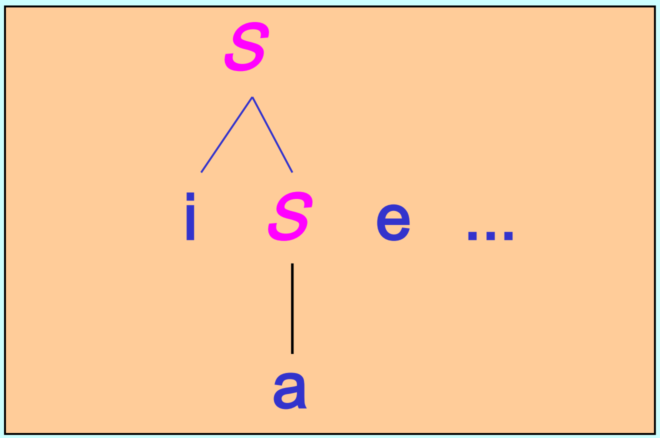
FOLLOW(*S*) = { e , $ }

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	

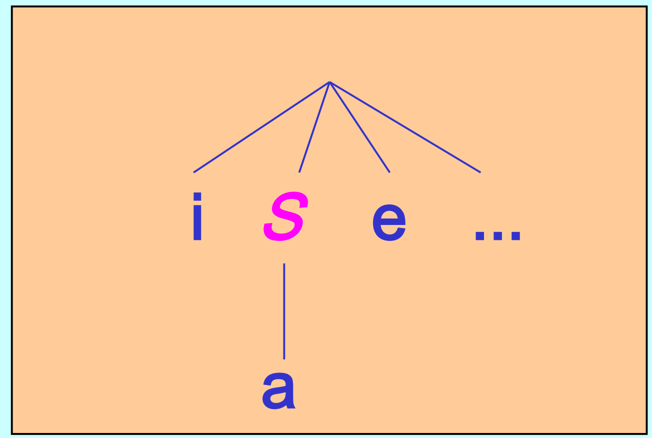
shift / reduce conflict

SA: resolving shift/reduce conflicts in favor of shift (1)

- conflict in ACTION[**4** , **e**] = **s5 , r2** is due to the items $S \rightarrow$ **i** $S \cdot$ **e** S and $S \rightarrow$ **i** $S \cdot$
- the top of the stack is **i** S and the next input symbol is **e**



parse tree produced by *reducing* (**e** is not associated with the previous **i**)



parse tree produced by *shifting* (**e** is associated with the previous **i**)

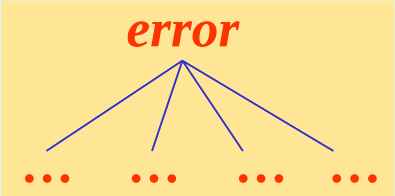
FL&C

SA: resolving shift/reduce conflicts in favor of shift (2)

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	

SA: error recovery in LR parsing

- *blanks* in LR parsing tables mean ***error actions*** and cause the parser to *stop*
- this behavior would be unkind to the user, who would like to have *all the errors reported* , not just the first one
- local error recovery mechanisms use a special ***error*** symbol to allow *parsing to resume*
- whenever the ***error*** symbol appears in a grammar rule, it can *match* a sequence of *erroneous input symbols*



SA: recovery using the error symbol (1)

$G_6 = (\{ S, E \}, \{ \text{id}, +, *, (,), ; \}, P, S)$

$P = \{ S \rightarrow S ; E \mid E$ (1, 2)

$\mid \textit{error} ; E$ (3)

$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$ (4, 5, 6, 7)

$\mid (\textit{error}) \}$ (8)

- production (3): $S \rightarrow \textit{error} ; E$ specifies that the parser, encountering a syntax error, can skip to the next ; (semicolon)
- production (8): $E \rightarrow (\textit{error})$ specifies that the parser, encountering a syntax error after a ((left parenthesis), can skip to the next) (right parenthesis)

SA: recovery using the error symbol (2)

- let $A \rightarrow$ ***error*** α be a grammar production
- in the construction of the parsing table:
  - ***error*** is considered a terminal symbol
  - error productions are treated as ordinary productions
- on encountering an *error action* (a blank in the table), the parser:
  - *pops* the stack until a state is reached where the action for ***error*** is *shift* (a state including an item $A \rightarrow \cdot$ ***error*** α)
  - *shifts* a fictitious ***error*** token onto the stack, as though ***error*** was found on input
  - *skips* ahead on the input discarding symbols until a substring is found that can be reduced to α
  - *reduces* the handle ***error*** α (at this point on top of the stack) to A
  - *emits* a diagnostic message
  - *resumes* normal parsing

SA: recovery using the error symbol (3)

- ***error rules*** may introduce both *shift/reduce* and *reduce/reduce* conflicts
- they cannot be inserted anywhere into an LALR grammar
- this error recovery mechanism is not powerful enough to correctly report all syntactic errors

SA: LR parser generators

- the task of constructing a parser is simple enough to be automated
- an *LR parser generator* transforms the *specification* of a parser (*grammar, conflict resolution directives, ...*) into a program implementing an LR parser
- ***Yacc*** *(UNIX)* and ***Bison*** *(GNU)* produce *C programs* implementing *LALR(1) parsers*
- ***CUP*** and ***SableCC*** produces *Java programs* implementing *LALR(1) parsers*

SA: top-down parsing

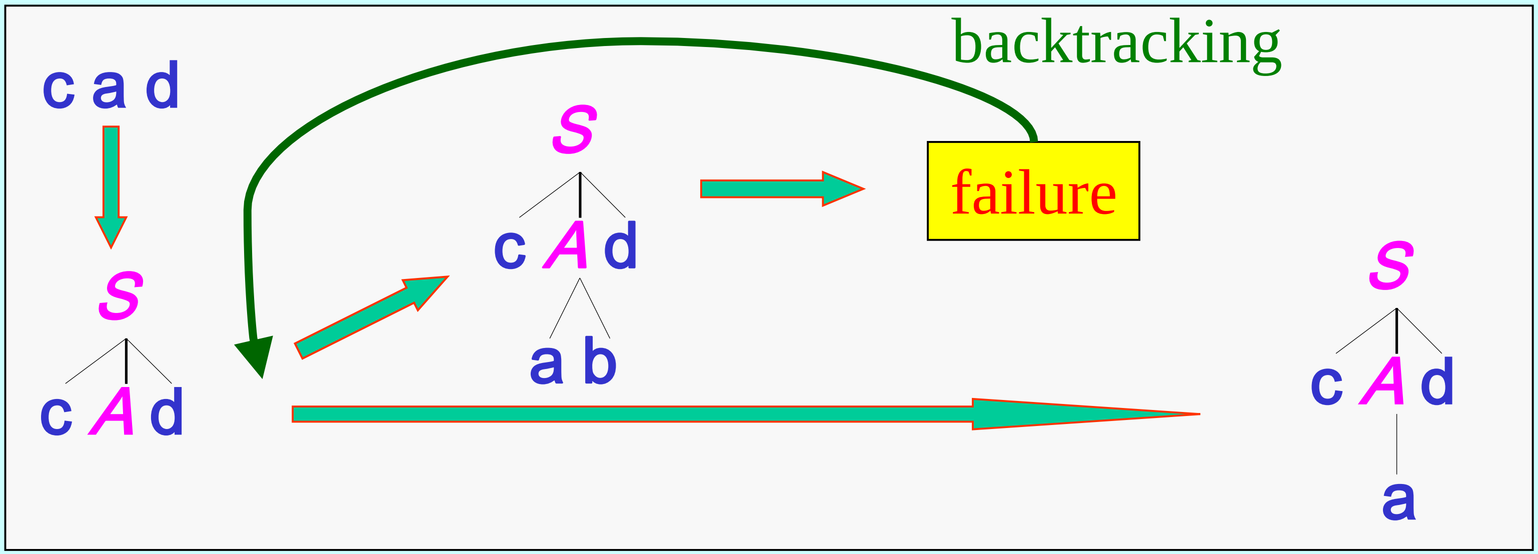
- top-down parsing attempts to construct a ***parse tree*** for an input string beginning at the *root* (the top) and working down towards the *leaves*
- this construction process ***creates*** the nodes of the tree in *preorder* until it obtains the *input string*
- at each ***creation*** step the *left side symbol* of a production is *replaced* by its *right side*, tracing out a *leftmost derivation*

SA: recursive-descent parsing

$G = (\{S, A\}, \{a, b, c, d\}, P, S)$

$P = \{ S \rightarrow c\,A\,d$

$\quad A \rightarrow a\,b \mid a \}$



$S \Rightarrow_{lm} c\,A\,d \Rightarrow_{lm} c\,a\,d$

SA: left-recursive grammars

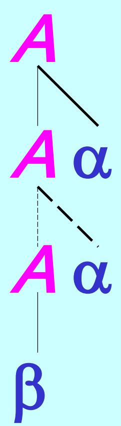
- a production like $A \rightarrow A\,\alpha$ is called a ***left-recursive production***
- a ***grammar*** is ***left-recursive*** if it can generate a derivation $A \Rightarrow^* A\,\alpha$
- a ***left-recursive grammar*** can cause a *top-down parser* to go into an *infinite loop*
  - $A \Rightarrow_{lm}^* A\,\alpha \Rightarrow_{lm}^* A\,\alpha\,\alpha \Rightarrow_{lm}^* A\,\alpha\,\alpha \ldots \alpha$

SA: eliminating left-recursive productions (1)

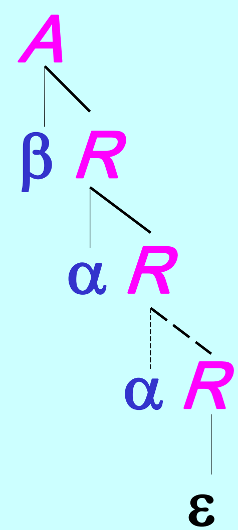
- *left-recursive* productions can be replaced by *right-recursive* productions

$$A \rightarrow A\,\alpha \mid \beta \quad \equiv \quad \{\ A \rightarrow \beta\, R \qquad R \rightarrow \alpha\, R \mid \varepsilon\ \}$$

(β does not start with A)



$$A \Rightarrow^* \beta\, \alpha^*$$



SA: eliminating left-recursive productions (2)

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

$P = \{ \; E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{id} \; \}$

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

$P = \{ \; E \rightarrow T E'$

$E' \rightarrow + T E' \mid \varepsilon$

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid \varepsilon$

$F \rightarrow (E) \mid \text{id} \; \}$

SA: eliminating left-recursion (1)

let $G = (\{ A_1 , A_2 , \dots , A_n \}, T, P, A_1)$ be a CFG grammar with no *ε-production* ;

for (i = 1 to n)

for (j = 1 to i – 1)

replace each production of the form $A_i \rightarrow A_j \gamma$ by the productions $A_i \rightarrow \delta \gamma$ where $A_j \rightarrow \delta$ are all the A_j -productions ;

eliminate left-recursive productions among A_i -productions ;

SA: eliminating left-recursion (2)

P_1 = { $S \rightarrow A$ a | b
$A \rightarrow A$ c | S d | c }

↓

P_2 = { $S \rightarrow A$ a | b
$A \rightarrow A$ c | A a d | b d | c }

↓

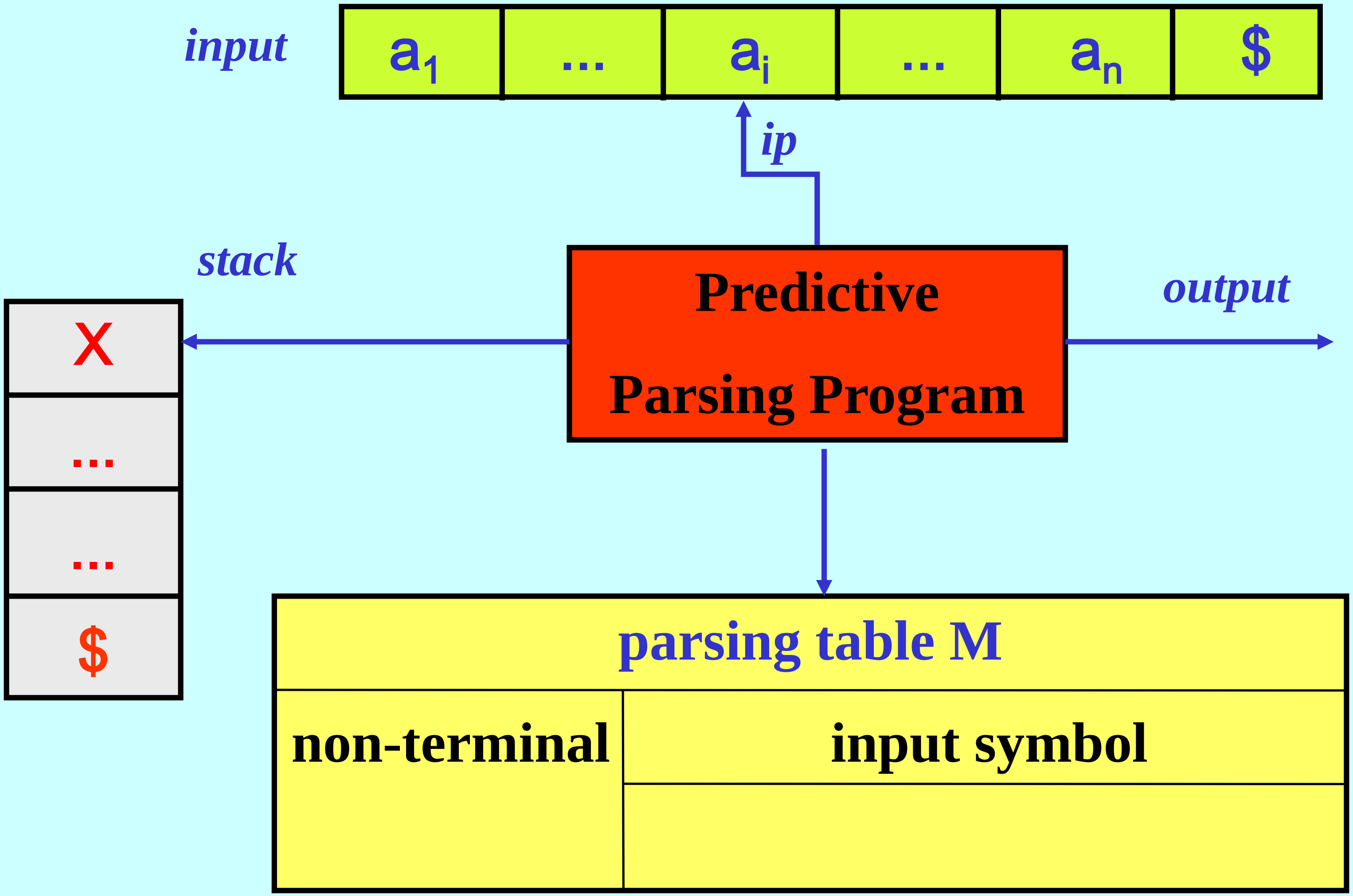
P_3 = { $S \rightarrow A$ a | b
$A \rightarrow$ b d A' | c A'
$A' \rightarrow$ c A' | a d A' | ε }

SA: predictive parsing

- *backtracking* can be avoided if it is possible to detect which alternative rule among $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ has to be applied, by considering the current *input symbol*

S	$\rightarrow$	**if** (E) S **else** S
		| **while** (E) S
		| { S ; S }
		| **id** = E

SA: non-recursive predictive parsing



SA: predictive parsing program

```
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 */
```

FL&C

SA: a predictive parser for grammar G_1

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

$P = \{ \quad E \rightarrow T E'$

$\quad E' \rightarrow + T E' \mid \varepsilon$

$\quad T \rightarrow F T'$

$\quad T' \rightarrow * F T' \mid \varepsilon$

$\quad F \rightarrow (E) \mid \text{id} \quad \}$

non terminal	input symbol					
	id	+	*	(	)	$
E	$E \rightarrow T E'$			$E \rightarrow T E'$		
E'		$E' \rightarrow + T E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F T'$			$T \rightarrow F T'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow * F T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

SA: moves of a predictive parser for grammar G_1

stack	input	output
$ *E*	id + id * id $	
$ *E' T*	id + id * id $	$E \rightarrow T E'$
$ *E' T' F*	id + id * id $	$T \rightarrow F T'$
$ *E' T'* id	id + id * id $	$F \rightarrow$ id
$ *E' T'*	+ id * id $	
$ *E'*	+ id * id $	$T' \rightarrow \varepsilon$
$ *E' T* +	+ id * id $	$E' \rightarrow + T E'$
$ *E' T*	id * id $	
$ *E' T' F*	id * id $	$T \rightarrow F T'$
$ *E' T'* id	id * id $	$F \rightarrow$ id
$ *E' T'*	* id $	
$ *E' T' F* *	* id $	$T' \rightarrow * F T'$
$ *E' T' F*	id $	
$ *E' T'* id	id $	$F \rightarrow$ id
$ *E' T'*	$	
$ *E'*	$	$T' \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$

SA: construction of predictive parsing tables

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** ;

SA: computation of FIRST and FOLLOW for grammar G_1

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

$P = \{ \; E \rightarrow T E'$

$E' \rightarrow + T E' \mid \varepsilon$

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid \varepsilon$

$F \rightarrow (E) \mid \text{id} \; \}$

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

SA: construction of a predictive parsing table for grammar G_1

non terminal	input symbol					
	id	+	*	(	)	$
E	$E \rightarrow T\, E'$			$E \rightarrow T\, E'$		
E'		$E' \rightarrow +\, T\, E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\, T'$			$T \rightarrow F\, T'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *\, F\, T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ id			$F \rightarrow (\, E\,)$		

SA: LL(1) grammars

- a ***grammar G*** is ***LL(1)*** if its predictive parsing table has no multiply-defined entries
  - whenever $A \rightarrow \alpha \mid \beta$ then
    - $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$
    - at most one of α and β is ***nullable***
    - if α is ***nullable*** then $FIRST(\beta) \cap FOLLOW(A) = \varnothing$
- no ambiguous or left-recursive grammar can be *LL(1)*
- an ***LL(1)*** parser
  - scans the input from left to right (***L***)
  - constructs a leftmost derivation (***L***)
  - uses ***1*** lookahead input symbols in making parsing decisions
- the class of languages that can be parsed using ***LL(1)*** parsers is a *proper subset* of the *deterministic* CFL's

SA: LL parser generators

- an *LL parser generator* transforms the *specification* of a parser into a program implementing an LL parser
- ***JavaCC*** produces *Java programs* implementing *LL(k) parsers*
- ***ANTLR*** produces *Java*, *C++* and *Python programs* implementing *recursive descent LL(k) parsers*
- ***Coco/R*** produces *Java*, *C++*, *C#, … programs* implementing *recursive descent LL(k) parsers*

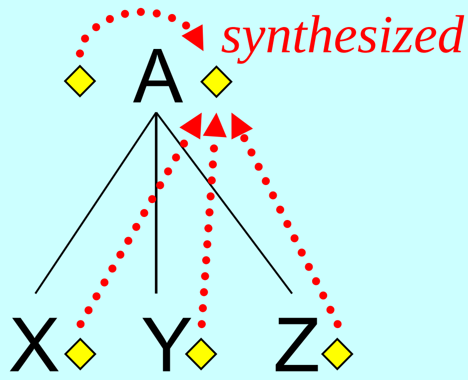
Syntax-Directed Translation: Syntax-Directed Definitions

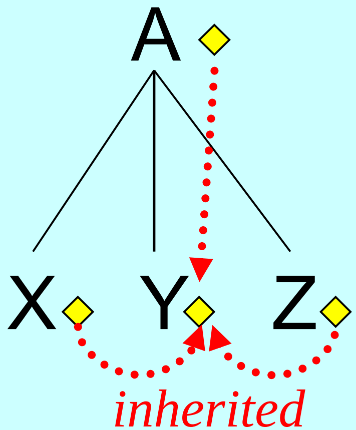
- a ***Syntax-Directed Definition (SDD)*** is a context-free grammar in which
  - each *symbol* can have an associated set of ***attributes***
    - numbers, types, table references, strings, memory locations, ...
  - each *production* can have an associated set of ***semantic rules***
    - evaluating attributes, interacting with the symbol table, writing lines of intermediate code to a buffer, printing messages, ...

FL&C

SDT: inherited and synthesized attributes

- a semantic rule associated with a production **A → X Y Z** can refer only attributes associated with symbols in that production
  - *synthesized attributes*
    - are *evaluated* in rules where the *associated* symbol is on the left side of the production
  - *inherited attributes*
    - are *evaluated* in rules where the *associated* symbol is on the right side of the production



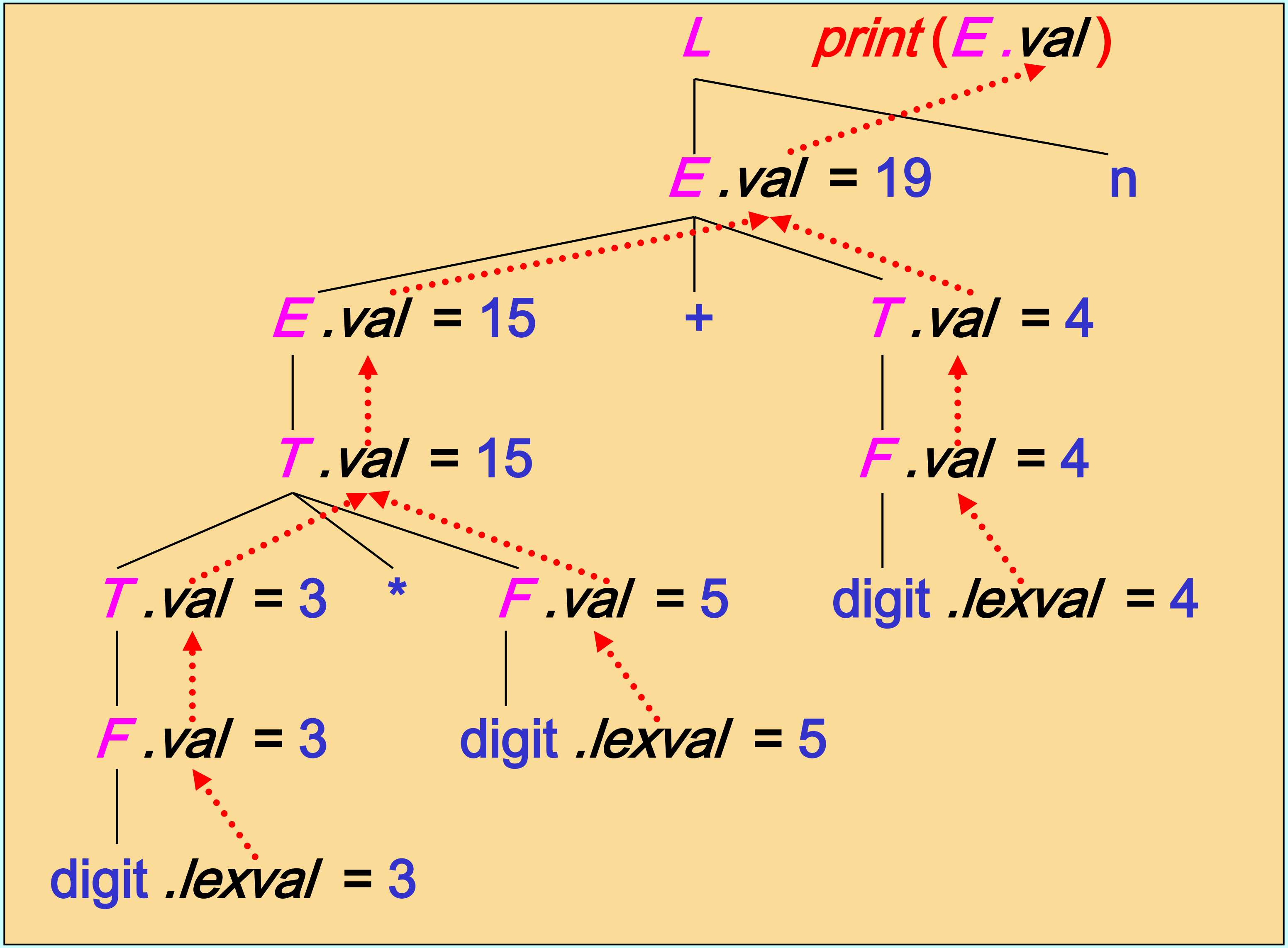


SDT: SDD for a desk calculator

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*

- each of the non-terminals E, T and F has a single ***synthesized*** attribute, named *val*
- the terminal **digit** has an attribute *lexval* which is the integer value returned by the scanner

SDT: annotated parse tree for 3*5+4n

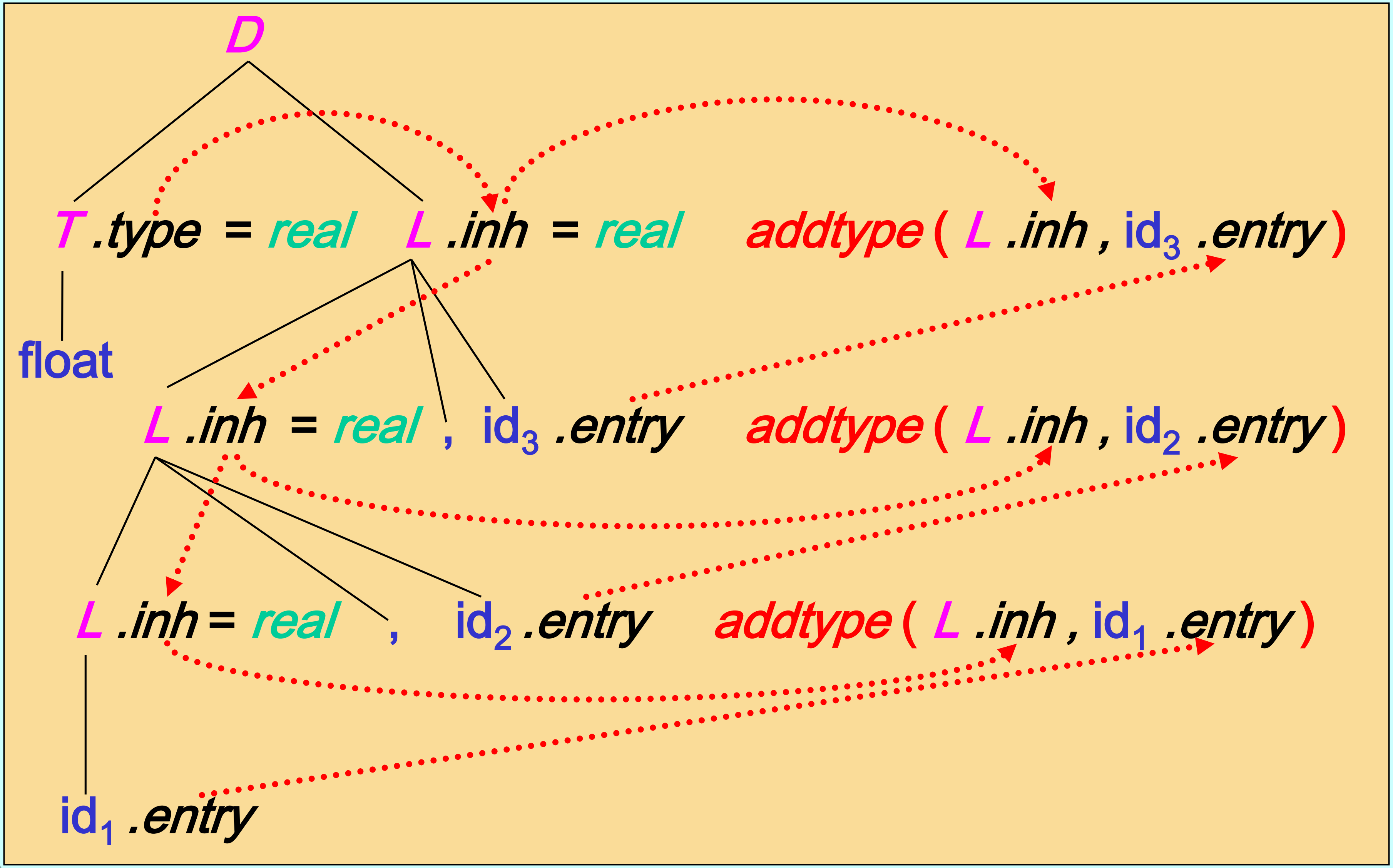


SDT: SDD for simple declarations

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*)

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

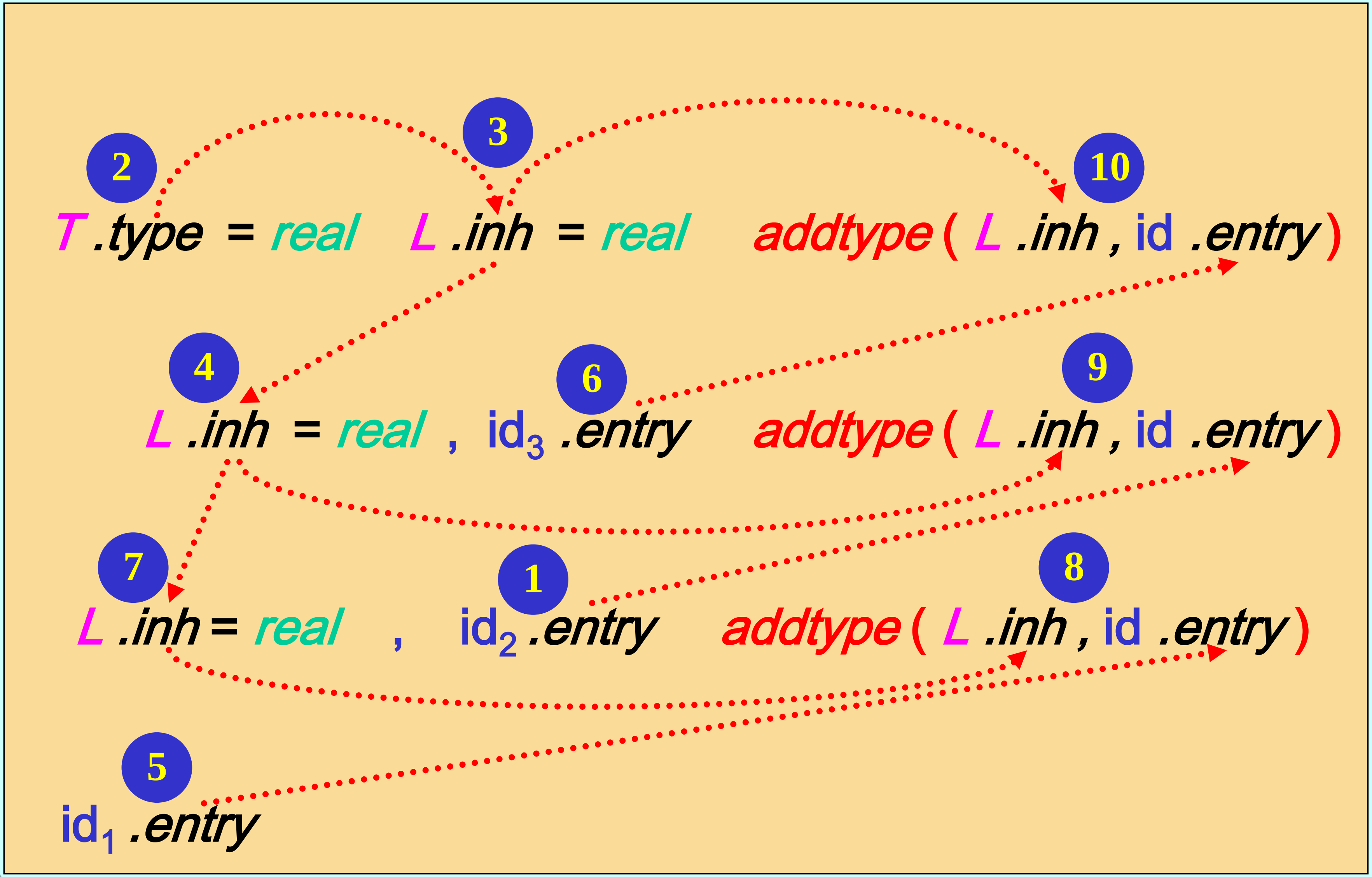
SDT: annotated parse tree for float id_1 , id_2 , id_3



SDT: evaluation orders for SDD's

- an attribute at a node in an annotated parse tree cannot be evaluated before the evaluation of all attributes upon which its value *depends*
- the *dependency relations* in a parse tree define a ***dependency graph*** representing the flow of information among attributes and semantic rules
- any ***topological sort*** of the dependency graph is an allowable *order of evaluation* for an *SDD*
- any *directed acyclic graph* has at least one topological sort

SDT: topological sorts of a dependency graph



SDT: ordering the evaluation of SDD's

- *syntax-directed translation* can be performed by:
  - creating a *parse tree*
  - visiting the *parse tree* and evaluating an *SDD* according to a *topological sort* of the *dependency graph*

- checking if the *dependency graph* of any *parse tree* from a given *SDD* contains *cycles*, is a problem of *extreme time-complexity*

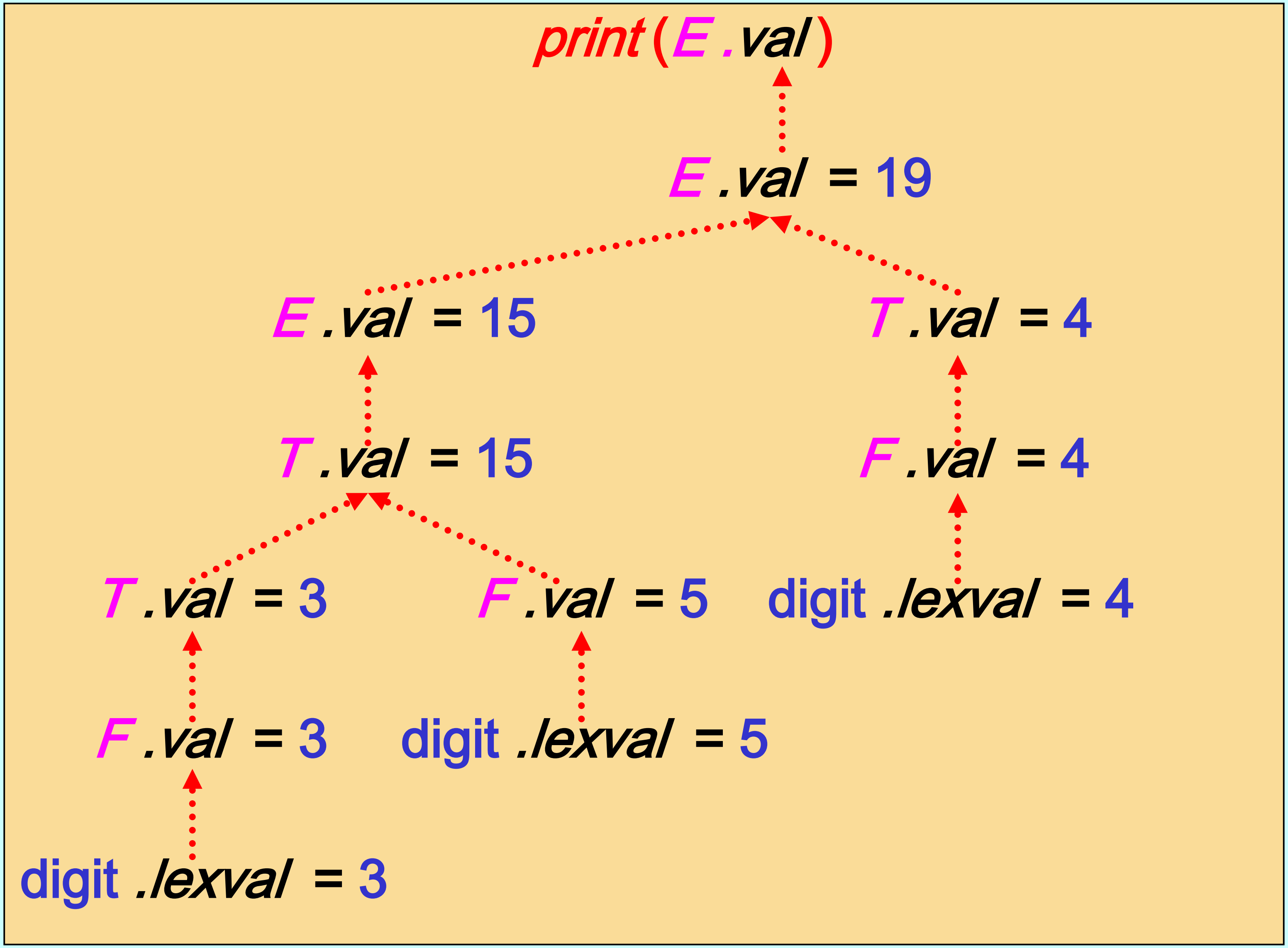
- it is possible to define *classes of SDD's* (*S-attributed* and *L-attributed*) in ways that:
  - *cycles* are not allowed
  - translation is performed in connection with *top-down* or *bottom-up* parsing, without explicitly creating the *tree nodes*

SDT: S-attributed definitions

- an *SDD* is ***S-attributed*** if every attribute is ***synthesized***
  - all 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*

SDT: dependency trees of S-attributed definitions

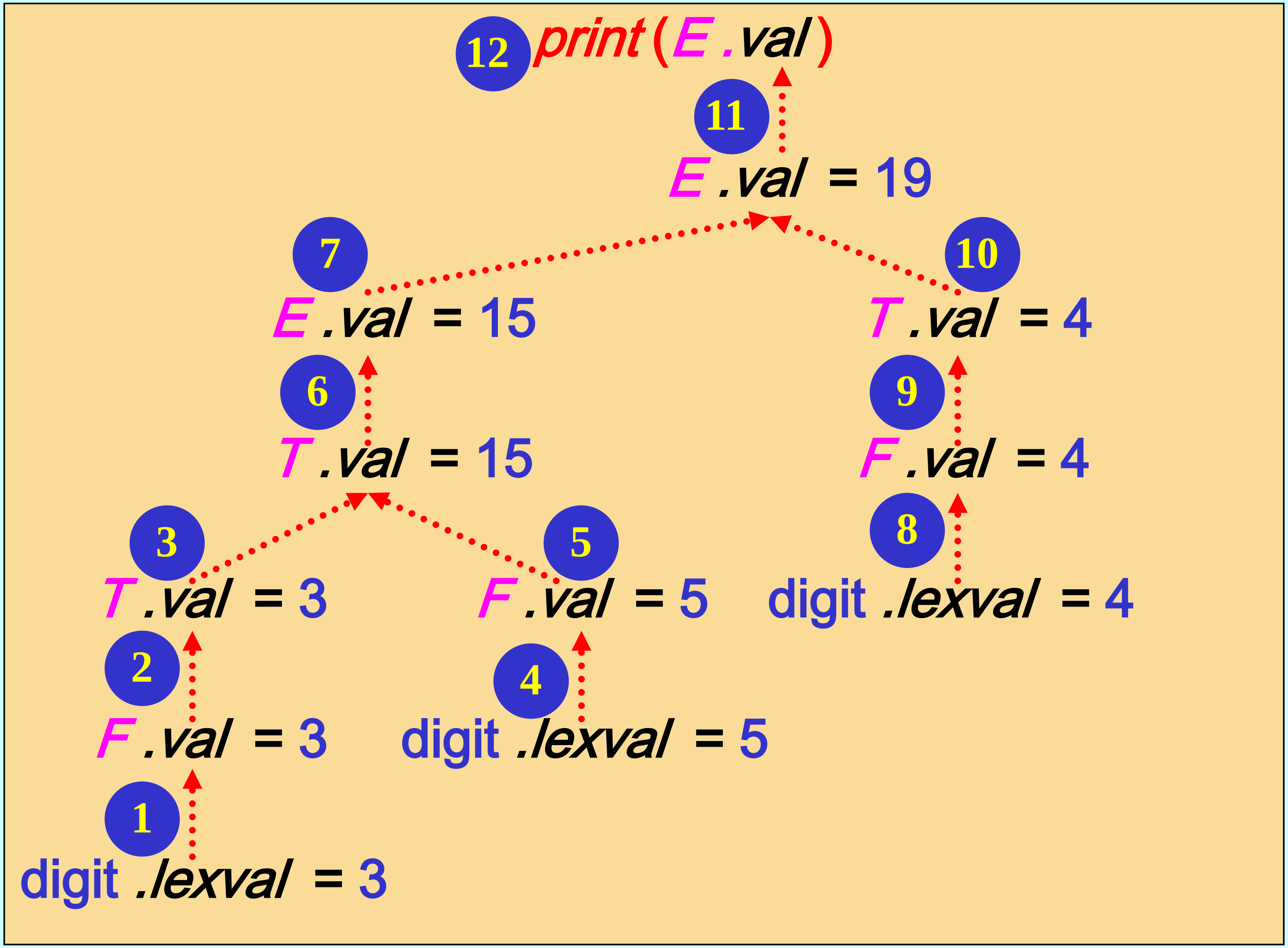


SDT: evaluation orders for S-attributed definitions

- *S-attributed definitions* can be evaluated in any *bottom-up* order
- the evaluation order of function *postorder(rootNode)* corresponds to the order in which a *bottom-up parser* creates nodes in a *parse tree*

```
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 ;
```

SDT: postorder evaluation of S-attributed definitions



SDT: Syntax-Directed Translation Schemes (1)

- a ***Syntax-Directed Translation Scheme (SDT)*** is an *SDD* with the actions of each semantic rule *embedded* at some positions in the right side of the associated production
  - an *SDT* implementation executes each action as soon as all the grammar symbols to the left of the action are processed
  - an *SDT* having all actions at the right ends of the productions is called ***postfix SDT***

SDT: Syntax-Directed Translation Schemes (2)

- the action ***a*** in the rule **A → X { *a* } Y** should be performed:
  - in *bottom-up parsing*
    - as soon as this occurrence of **X** appears on the top of the parsing stack
  - in *top-down parsing*
    - 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

SDT: bottom-up evaluation of S-attributed definitions

- *S-attributed SDD's* can be converted to ***postfix SDT's*** simply by placing each *action* at the right end of the associated production
- *actions* in a ***postfix SDT*** can be executed by a *bottom-up parser* along with *reductions*

$$
\begin{array}{lll}
L \rightarrow E\ \mathbf{n} & \{\ print(E.val)\ \} \\
E \rightarrow E_1 + T & \{\ E.val = E_1.val + T.val\ \} \\
E \rightarrow T & \{\ E.val = T.val\ \} \\
T \rightarrow T_1 * F & \{\ T.val = T_1.val * F.val\ \} \\
T \rightarrow F & \{\ T.val = F.val\ \} \\
F \rightarrow (\ E\ ) & \{\ F.val = E.val\ \} \\
F \rightarrow \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval\ \}
\end{array}
$$

SDT: stack implementation of postfix SDT's (1)

- *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 in β have been evaluated
  - when the ***reduction*** of β occurs, the associated actions can be executed

stack

state	*symbol*	*attributes*	
s_m	X_m	X_m .*val*	← ***top***
s_{m-1}	X_{m-1}	X_{m-1}.*val*	
...	...	...	
s_1	X_1	X_1 .*val*	

SDT: stack implementation of postfix SDT's (2)

$L \rightarrow E$ n { *print*(*stack*[*top* - 1].*val*) }

$E \rightarrow E + T$ { *n_top* = *top* – 3 + 1 ;
stack[*n_top*].*val* = *stack*[*top* - 2].*val* + *stack*[*top*].*val* ;
top = *n_top* }

$E \rightarrow T$

$T \rightarrow T * F$ { *n_top* = *top* - 3 + 1 ;
stack[*n_top*].*val* = *stack*[*top* - 2].*val* * *stack*[*top*].*val* ;
top = *n_top* }

$T \rightarrow F$

$F \rightarrow (E)$ { *n_top* = *top* - 3 + 1 ;
stack[*n_top*].*val* = *stack*[*top* - 1].*val* ;
top = *n_top* }

$F \rightarrow$ digit

SDT: L-attributed definitions

- an *SDD* is ***L-attributed*** if any production $A \rightarrow X_1\ X_2\ \ldots\ X_n$ has:
  - ***synthesized*** attributes
  - ***inherited*** attributes $X_i.a$ computed in terms of:
    - inherited attributes associated with symbol A
    - inherited or synthesized attributes associated with symbols $X_1\ X_2\ \ldots\ X_{i-1}$ located at the left (***L***) of X_i

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	$addtype(L.inh, id.entry)$

SDT: SDT's for L-attributed definitions

- to convert an *L-attributed SDD* to an *SDT*:
  - place the actions that compute an *inherited attribute* for a symbol ***X*** immediately *before* that occurrence of ***X***
  - place the actions that compute a *synthesized attribute* at the *end* of the production

$D \rightarrow T \; \{ L.inh = T.type \} \; L$

$T \rightarrow \text{int} \; \{ T.type = integer \}$

$T \rightarrow \text{float} \; \{ T.type = real \}$

$L \rightarrow \{ L_1.inh = L.inh \} \; L_1 \; , \; \text{id} \; \{ addtype(L.inh, \text{id}.entry) \}$

$L \rightarrow \text{id} \; \{ addtype(L.inh, \text{id}.entry) \}$

SDT: bottom-up evaluation of L-attributed definitions

- a *bottom-up parser* is aware of the production it is using only when it performs a *reduction*
- it can therefore *execute actions* associated with a production only when they are placed at the *end* of the production
- *actions* that compute *inherited attributes* are *not* placed at the *end* of productions
- it is possible to *transform* an *L-attributed* definition into an equivalent definition where all *actions* are placed at the *end* of productions

SDT: inheriting attributes on the parser stack (1)

- in an *L-attributed* translation scheme with a rule
  $A \rightarrow X \{Y.i = X.s\} Y$ where:
  - $X.s$ is a *synthesized attribute*
  - $Y.i$ is an *inherited attribute* defined by a ***copy rule***
- the value of $X.s$ is already on the parser stack before any reduction to Y is performed
- it can then be retrieved on the stack *one position before* Y and used anywhere $Y.i$ is called for
- the copy rule $\{Y.i = X.s\}$ can be eliminated

SDT: inheriting attributes on the parser stack (2)

$D \rightarrow T$ { $L.inh = T.type$ } 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 { $addtype(stack[top-1].val, stack[top].val)$ }

SDT: inheriting attributes on the parser stack (3)

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	

SDT: inheriting attributes on the parser stack (4)

- reaching into the parser stack for an attribute value works only if the grammar allows the position of the attribute value to be predicted
- in the *SDT*:

$S \rightarrow$ 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)

the value of $A.s$ can be either one or two positions in the stack before C

- in order to place the value of $A.s$ always one position before C, it is possible to insert just before C in rule (2) a new *marker non-terminal* M with a synthesized attribute $M.s$ having the same value of $A.s$

SDT: inheriting attributes on the parser stack (5)

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

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

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

FL&C

SDT: simulating the evaluation of inherited attributes (1)

- in an *L-attributed* translation scheme with a rule $A \rightarrow X \{ Y.i = f(X.s) \} Y$ where:
  - $X.s$ is a *synthesized attribute*
  - $Y.i$ is an *inherited attribute* ***not*** defined by a ***copy rule***
- the value of $Y.i$ is not just a copy of $X.s$ and therefore it is not already on the parser stack before any reduction to Y is performed
- it is possible to insert just before Y a new *marker non-terminal* M with:
  - 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 $M \rightarrow \varepsilon \{ M.s = f(M.i) \}$

SDT: simulating the evaluation of inherited attributes (2)

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

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

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

SDT: bottom-up evaluation of L-attributed definitions

- systematic introduction of *markers* makes it possible to evaluate *L-attributed* translation schemes during *bottom-up parsing*
- unfortunately, an *LR(1)* grammar *may not remain LR(1)* after *markers* introduction
- *LL(1)* grammars *remain LL(1)* even when *markers* are introduced
- since *LL(1)* grammars are a proper subset of the *LR(1)* grammars, every *L-attributed* translation scheme based on an *LL(1)* grammar can be parsed *bottom-up*

Semantic Analysis

- the semantic analysis phase checks the source programs for *semantic errors* and gathers *type information* for the subsequent code-generation phase
  - type checks
    - the type of a construct must match that expected by its context
  - name-related and uniqueness checks
    - objects must be declared exactly once
  - flow-of-control checks
    - statements (such as *break* and *continue*) that cause flow of control to leave a construct must have a place where to go

SA: type expressions (1)

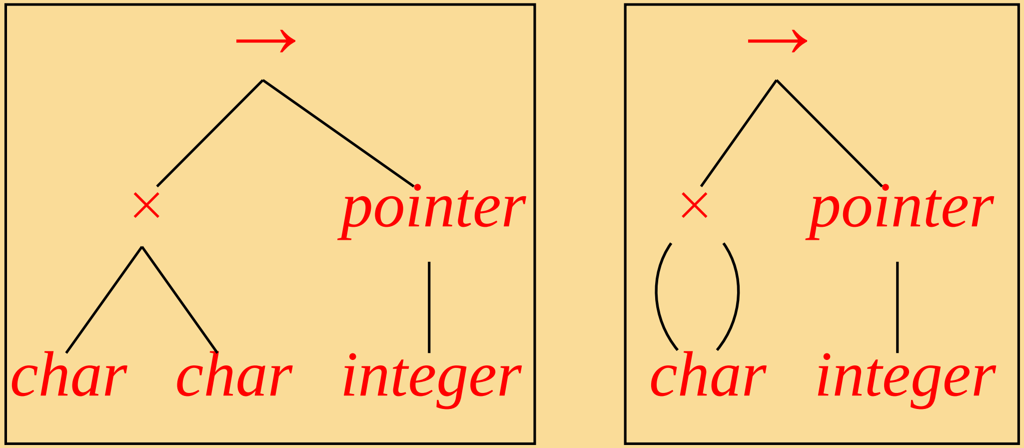
- a ***type expression T*** denotes the type of a language construct, that can be:
  - a *basic type*
    - *integer, real, char, boolean, void, ... , type_error*
  - a *type constructor* applied to *type expressions*
    - *array*
      - *array (index-set ,T)*
    - *Cartesian product*
      - $T_1 \times T_2 \times \dots T_n$
    - *record*
      - *record* $((\,name_1 \times T_1\,) \times (\,name_2 \times T_2\,) \times \dots (\,name_n \times T_n\,))$
    - *pointer*
      - *pointer (T)*
    - *function*
      - $T_1 \times T_2 \times \dots T_n \;\rightarrow\; T$

SA: type expressions (2)

- *type expressions* can be conveniently represented by *trees* or *DAG's* with
  - *type constructors* as *interior nodes*
  - *basic types* or *type names* as *leaves*

int* **f_name (*char* a, *char* b) {...}**

char × char → pointer (integer)



SA: equivalence of type expressions

```
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
```


SA: a simple type checker

$P \rightarrow D \;;\; S$

$D \rightarrow D \;;\; D \;\mid\; \text{id} : T$

$T \rightarrow \text{boolean} \mid \text{integer} \mid \text{array [num] of } T \mid T\,^*$

$S \rightarrow \text{id} = E \mid S \;;\; S \mid \text{if} \;(\; E \;)\; S \mid \text{while} \;(\; E \;)\; S$

$E \rightarrow \text{bool} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E\,[\,E\,] \mid {}^*E$

$P \rightarrow D \;;\; S$	
$D \rightarrow D \;;\; D$	
$D \rightarrow \text{id} : T$	{ *addtype* (T *.type* , id *.entry*) }
$T \rightarrow \text{boolean}$	{ T *.type* = *boolean* }
$T \rightarrow \text{integer}$	{ T *.type* = *integer* }
$T \rightarrow \text{array [num] of } T_1$	{ T *.type* = *array* (num *.val* , T_1 *.type*) }
$T \rightarrow T_1\,^*$	{ T *.type* = *pointer* (T_1 *.type*) }

SA: type checking of expressions

$E \rightarrow$	bool	{ E.*type* = *boolean* }
$E \rightarrow$	num	{ E.*type* = *integer* }
$E \rightarrow$	id	{ E.*type* = *lookup* (id .*entry*) }
$E \rightarrow$	E_1 mod E_2	{ E.*type* = if (E_1.*type* = *integer* and E_2.*type* = *integer*) then *integer* else *type_error* }
$E \rightarrow$	E_1 [E_2]	{ E.*type* = if (E_2.*type* = *integer* and E_1.*type* = *array* (*s*, *t*)) then *t* else *type_error* }
$E \rightarrow$	* E_1	{ E.*type* = if (E_1.*type* = *pointer* (*t*)) then *t* else *type_error* }

SA: type checking of statements

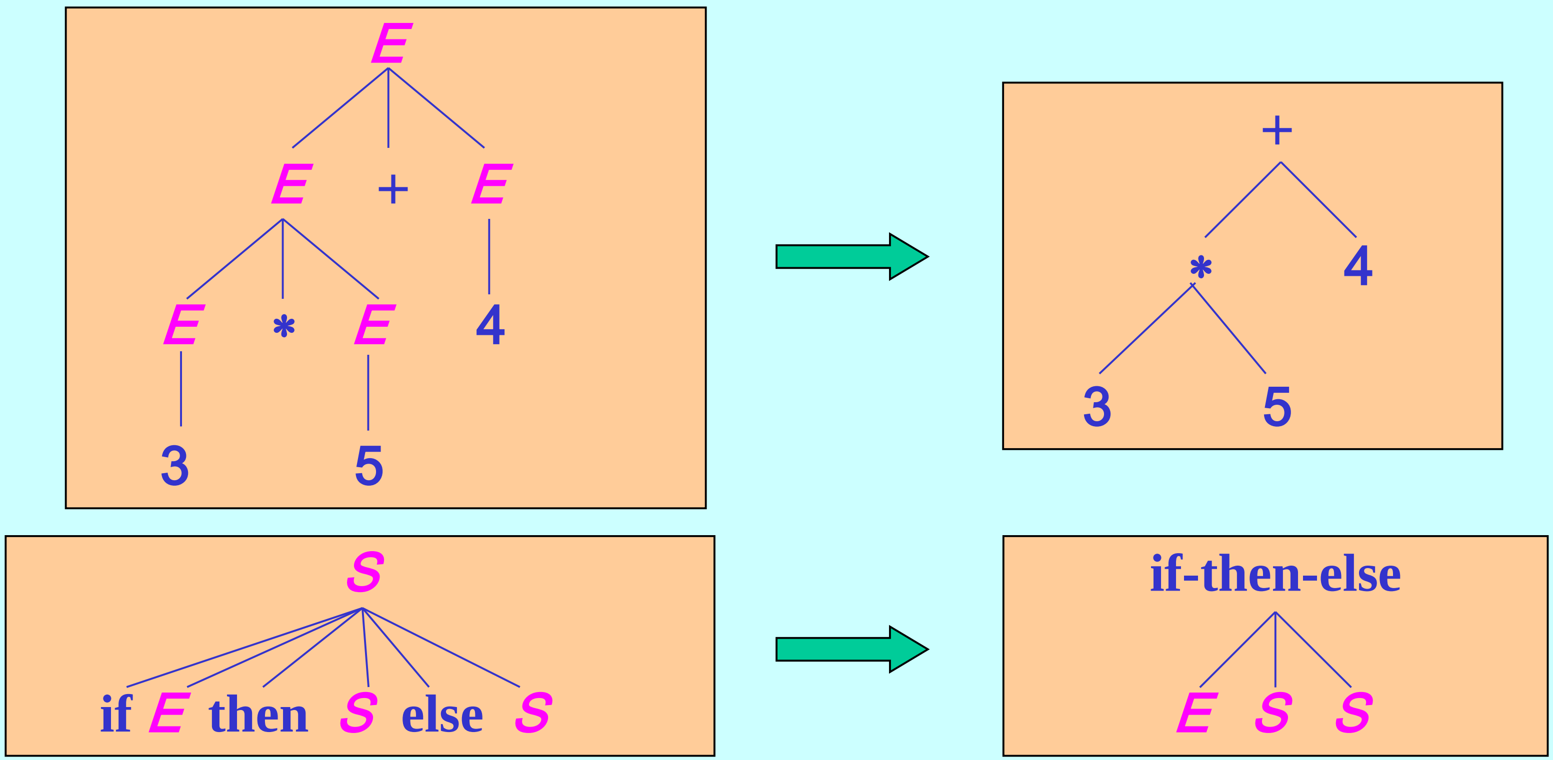
$S \rightarrow$ id = E	{ S .*type* = *if* (*equivalent* (id .*type* , E .*type*)) *then* *void* *else* *type_error* }
$S \rightarrow S_1$; S_2	{ S .*type* = *if* (S_1 .*type* = *void* *and* S_2 .*type* = *void*) *then* *void* *else* *type_error* }
$S \rightarrow$ if (E) S_1	{ S .*type* = *if* (E .*type* = *boolean*) *then* S_1 .*type* *else* *type_error* }
$S \rightarrow$ while (E) S_1	{ S .*type* = *if* (E .*type* = *boolean*) *then* S_1 .*type* *else* *type_error* }

SA: type checking of functions

$T \rightarrow T_1 \rightarrow T_2$	$\{ T.type = T_1.type \rightarrow T_2.type \}$
$E \rightarrow E_1 (E_2)$	$\{ E.type =$ *if* $(E_2.type = s$ *and* $E_1.type = s \rightarrow t)$ *then* t *else* $type\_error \}$

Intermediate-Code Generation: intermediate representations

- ***syntax tree***
  - condensed form of a *parse tree* where operators and keywords replace their non-terminal parent nodes

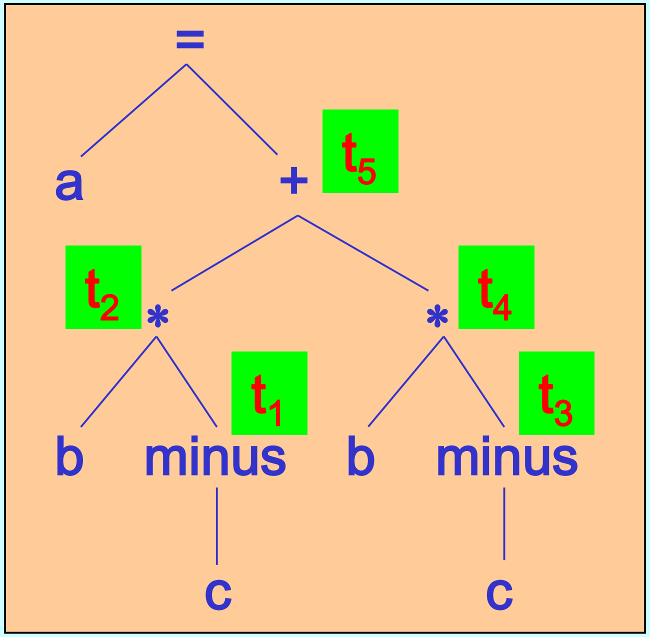


ICG: intermediate representations (2)

➢ *three-address code*

- linearized representation of a *syntax tree* in which explicit names correspond to interior nodes

a = b ∗ - c + b ∗ - c

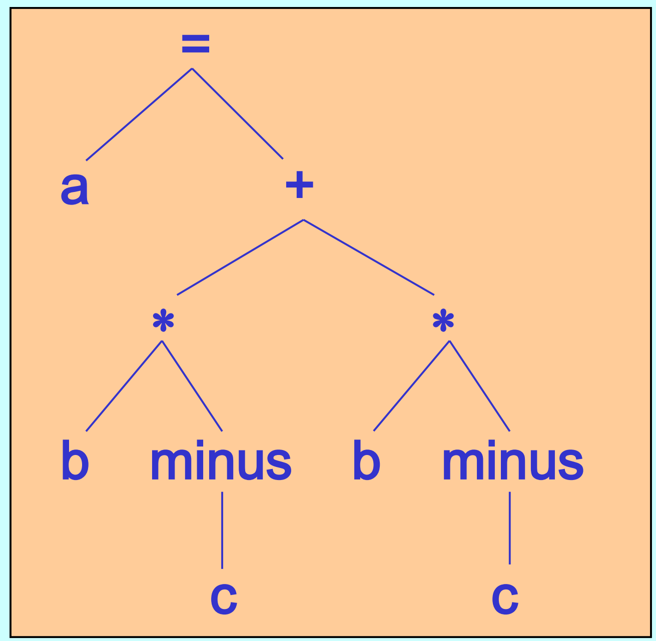


t_1 = minus c
t_2 = b * t_1
t_3 = minus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

ICG: construction of syntax trees

S	$\rightarrow$	id = E	{ $S.n$ = *new* *Assign* (*get* (id .*lexeme*), $E.n$) }
E	$\rightarrow$	$E_1 + E_2$	{ $E.n$ = *new* *Op* (+ , $E_1.n$, $E_2.n$) }
		| $E_1 * E_2$	{ $E.n$ = *new* *Op* (* , $E_1.n$, $E_2.n$) }
		| - E_1	{ $E.n$ = *new* *Minus* ($E_1.n$) }
		| (E_1)	{ $E.n$ = $E_1.n$ }
		| id	{ $E.n$ = *get* (id .*lexeme*) }

a = b * - c + b * - c



ICG: three-address code

- *three-address code* is built from two concepts:
  - ***address***
    - source-program name
    - constant
    - compiler-generated temporary name
  - ***instruction***
    - *assignment*
      - **x** = **y**
      - **x** = $\boldsymbol{op_1}$ **y**
      - **x** = **y** $\boldsymbol{op_2}$ **z**
        - » **x**, **y**, **z** are addresses
        - » $\boldsymbol{op_1}$ is a unary operator (minus, negation, shift, conversion , ...)
        - » $\boldsymbol{op_2}$ is a binary operator (arithmetic, logical, ...)

ICG: three-address code instructions

- *indexed assignment*
  - **x = y [i]**
  - **x [i] = y**
- *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:* $p\ (x_1, x_2, \ldots, x_n)$
  - **param x**
  - **call p , n**
- *procedure return*
  - **return y**

ICG: three-address code implementations

- ***quadruples***
  - objects with *4 fields*
    - **op , arg$_1$, arg$_2$, result**
- ***triples***
  - objects with *3 fields*
    - **op , arg$_1$, arg$_2$**
    - the result of an operation is referred by its position

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

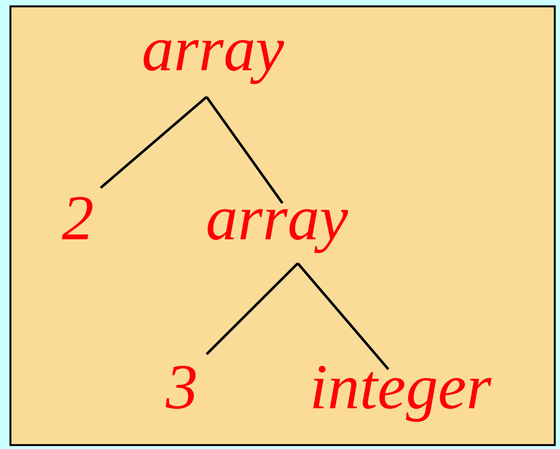
	op	arg$_1$	arg$_2$
(0)	minus	c	
(1)	*	b	(0)
(2)	minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

ICG: type width (1)

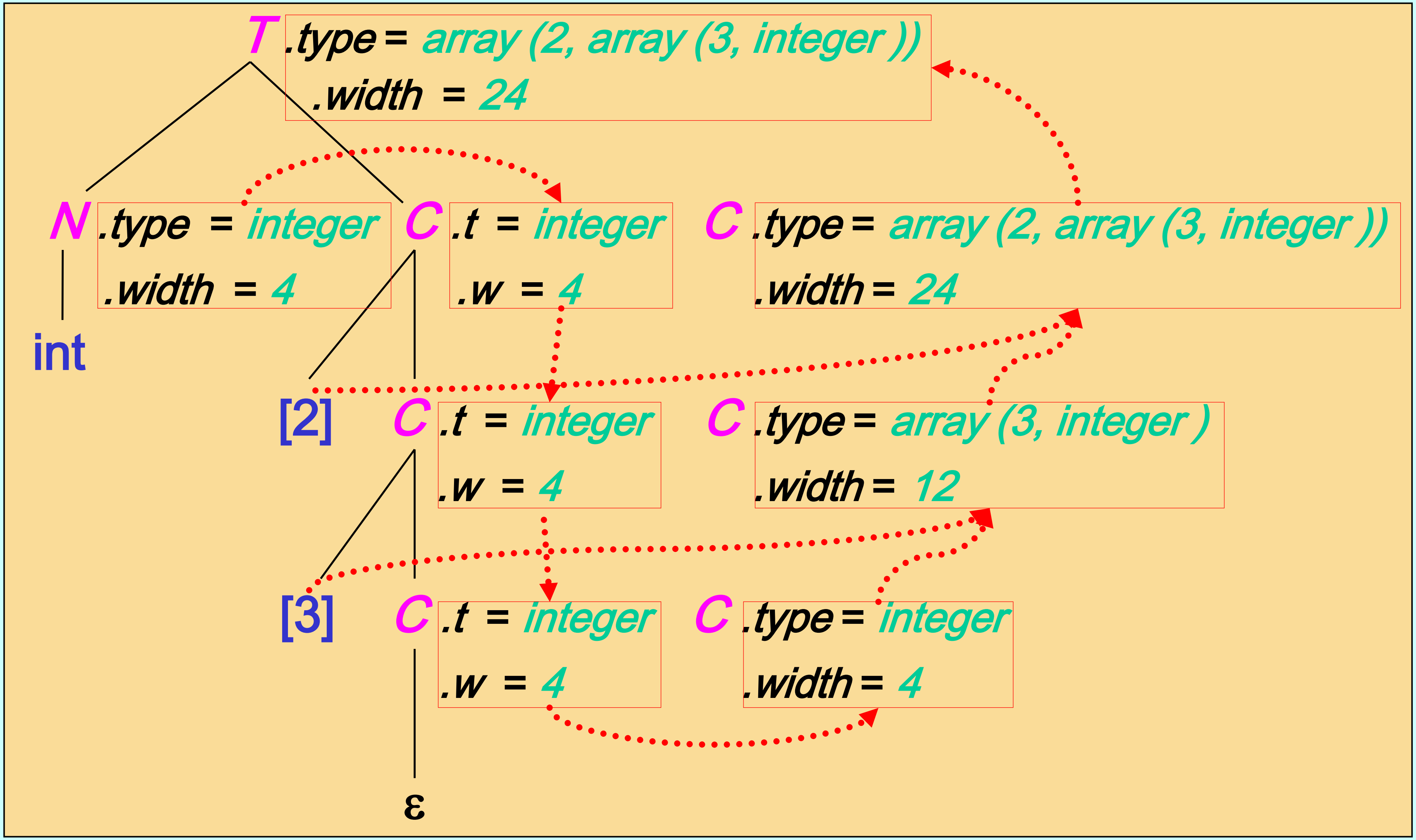
$T \rightarrow$	N	$\{C.t = N.type;\ C.w = N.width\}$
	C	$\{T.type = C.type;\ T.width = C.width\}$
$N \rightarrow$	int	$\{N.type = integer;\ N.width = 4\}$
$N \rightarrow$	real	$\{N.type = real;\ N.width = 8\}$
$C \rightarrow$	ε	$\{C.type = C.t;\ C.width = C.w\}$
$C \rightarrow$	[num]	$\{C_1.t = C.t;\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

int [2] [3]

array (2 , array (3, integer))



ICG: type width (2)



ICG: scope of declarations

- *scope* of a declaration of an identifier ***x***
  - the *region of program* in which uses of ***x*** refer to this declaration
- *static (lexical) scope*
  - the scope of a declaration is determined by *where* the declaration appears in the program and by *keywords* like *public*, *private* and *protected*
- *multiple* declarations
  - *nested environments* are allowed, where identifiers can be *redeclared*
- *most-closely nested* rule
  - an identifier ***x*** is in the scope of the *most-closely nested* declaration of ***x***

ICG: multiple declarations

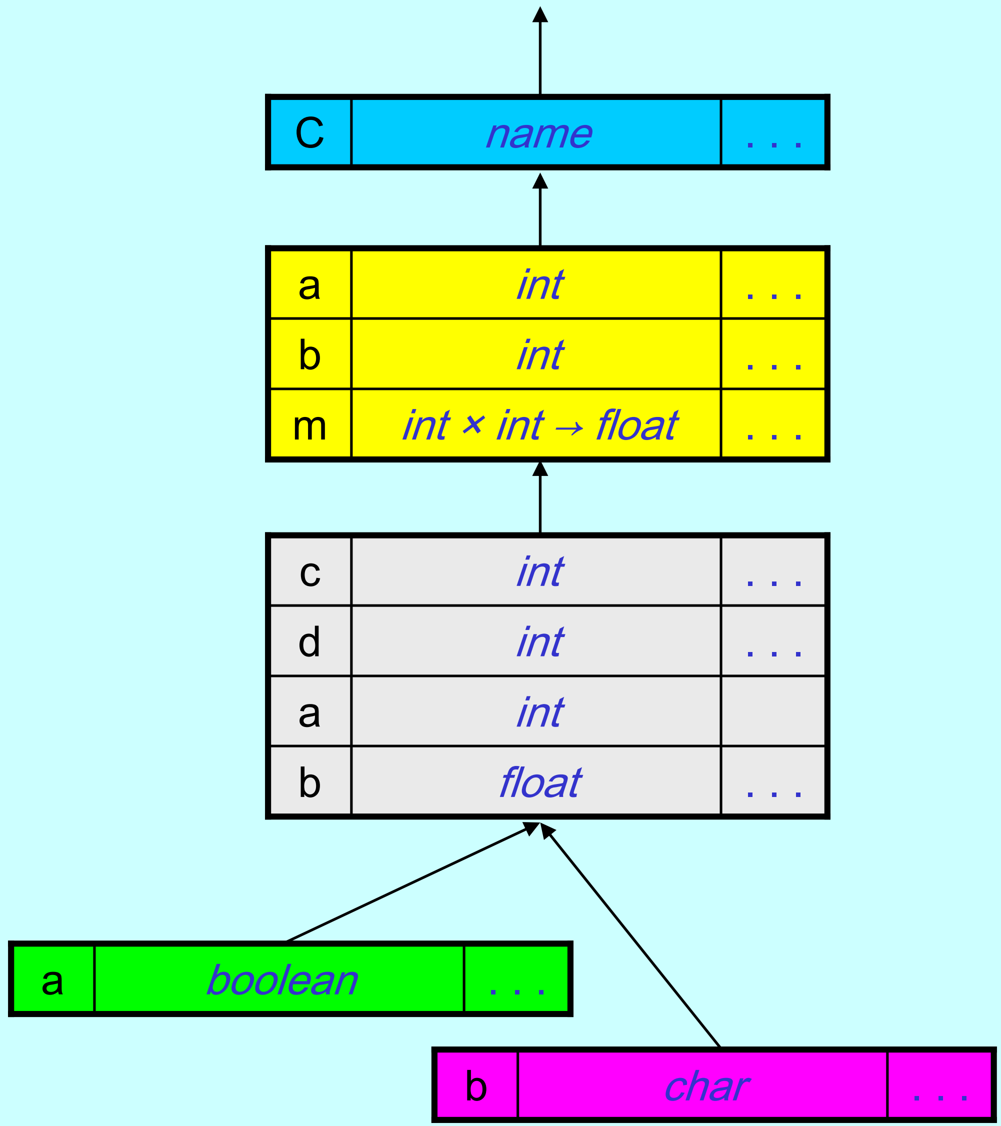
```
class C {
    int a;
    int b;
    float m ( int c, int d ) {
        int a;
        float b;
        . . .
        { boolean a;
          . . .
        }
        . . .
        { char b;
          . . .
        }
        . . .
    }
}
```

ICG: symbol tables

- *data structures* used to *hold information* about source-program constructs
- *information* is
  - collected incrementally in the *analysis phase*
  - used in the *synthesis phase* to generate the code
- *entries* in the symbol table contain information about an ***identifier***
  - *character string* (lexeme)
  - *type*
  - *position* (in storage)
  - . . .
- *multiple declarations* of the same identifier can be supported by setting up a *separate* symbol table for each *scope*
- the *most-closely nested* rule can be implemented by *chaining* the symbol tables
  - the table for a *nested* scope points to the table for its *enclosing* scope

ICG: chained symbol tables

```
class C {
    int a;
    int b;
    float m ( int c, int d ) {
        int a;
        float b;
        . . .
        { boolean a;
          . . .
        }
        . . .
        { char b;
          . . .
        }
        . . .
    }
}
```



ICG: implementation of chained 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 ;
    }
}
```


ICG: storage layout for sequences of declarations

$P \rightarrow$ { offset = 0 } D
$D \rightarrow D\ D$
$D \rightarrow T$ id ; { *top.put* (id *.lexeme* , *T* *.type* , offset) ;
offset = offset + *T* *.width* }

- variable **offset** keeps track of the next available *relative address*
- function *top.put* (id *.lexeme* , *T* *.type* , offset) creates a symbol-table entry for id *.lexeme* , with type *T* *.type* and relative address **offset** in the data area of the current (*top*) symbol table

FL&C

ICG: storage for records (structures, classes, blocks, ...)

- the production $T \rightarrow$ **record { D }** adds ***record types***
  - since a field name **x** in a record type does not conflict with other uses of **x** , each *record type* will get *its own symbol table*
  - the **offset** for a field name is relative to the data area of its symbol table
  - a *record type* can be represented by the type expression *record (t)*, where *t* is a symbol table that holds information about the fields of the record

```
T → 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 ( ) }
```

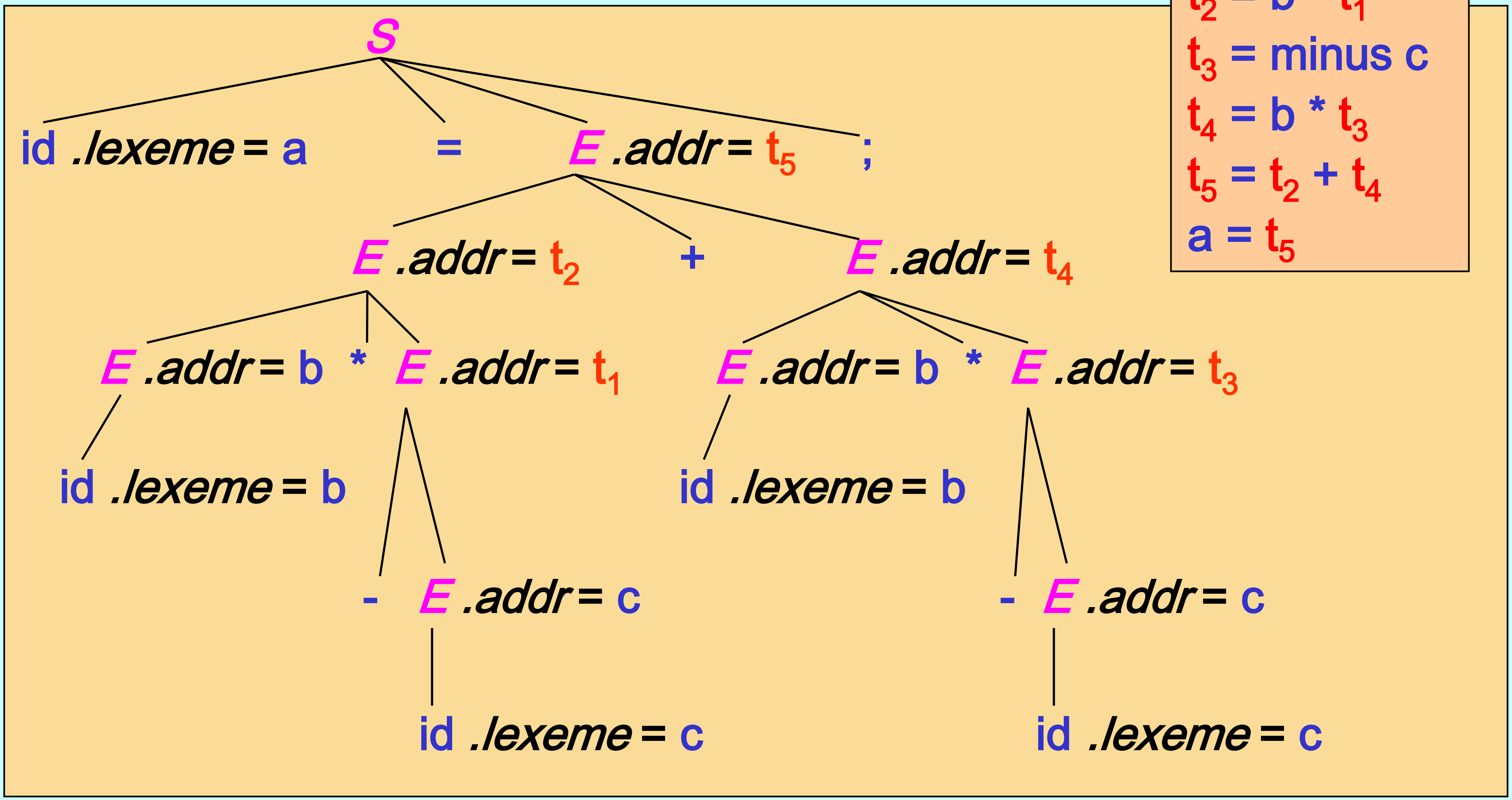
- functions *Env.push* (top) and *Storage.push* (offset) save the current symbol table and offset onto stacks
- functions *Env.pop* ( ) and *Storage.pop* ( ) retrieve the saved symbol table and offset

ICG: translation of assignment statements

$S \rightarrow$	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*) }

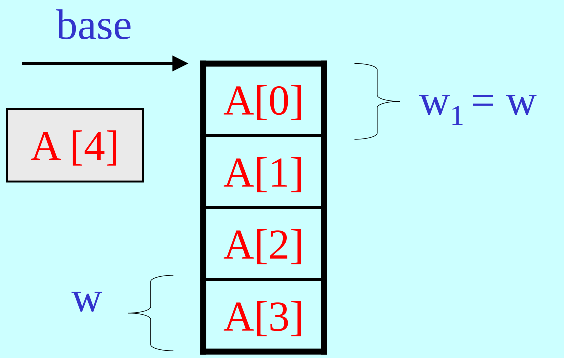
- function *gen* (*three-address instruction*) constructs a three-address instruction and appends it to the sequence generated so far
- function *top.get* (id *.lexeme*) retrieves the entry for id *.lexeme* in the data area of the current (*top*) symbol table

ICG: translation of a = b * - c + b * - c ;

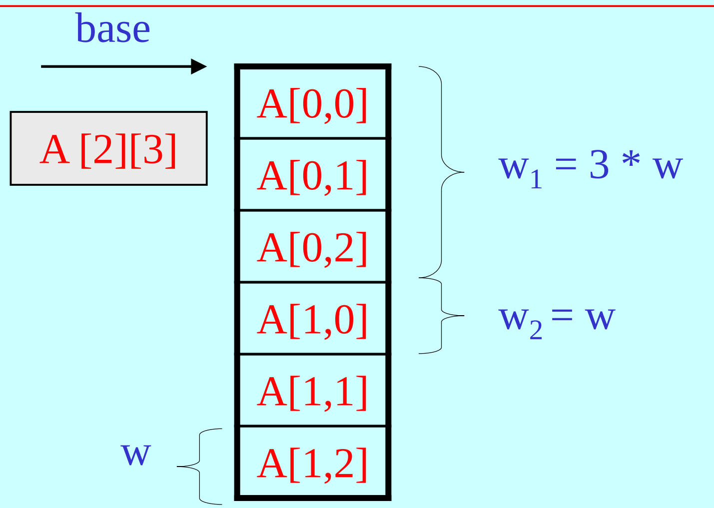


t_1 = minus c
t_2 = b * t_1
t_3 = minus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

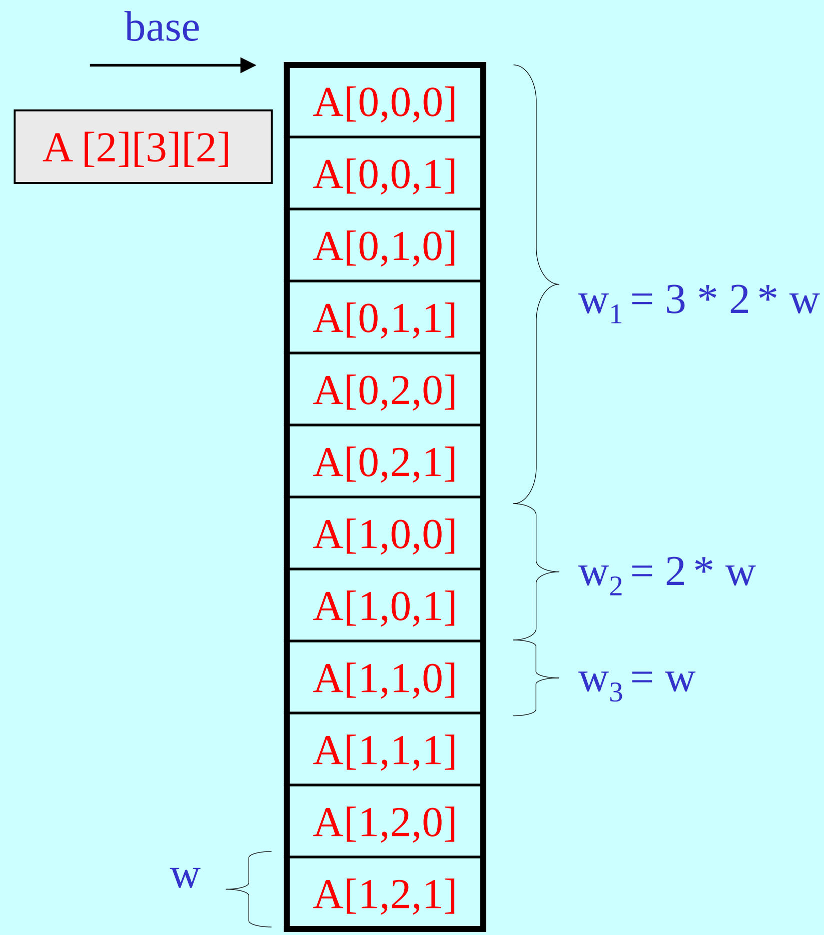
ICG: addressing array elements (1)



address (A[i_1]) = base + $i_1 * w_1$



address (A[i_1][i_2]) = base + $i_1 * w_1 + i_2 * w_2$



address (A[i_1][i_2][i_3]) = base + $i_1 * w_1$ + $i_2 * w_2 + i_3 * w_3$

ICG: addressing array elements (2)

$$A\ [n_1][n_2]...[n_k]$$

$$\mathit{address}\ (\ A[i_1][i_2]...[i_k]\ )\ =\ base\ +\ i_1 * w_1\ +\ i_2 * w_2\ +\ ...\ +\ i_k * w_k$$

for $1 \leq j \leq k\text{-}1$: $w_j = n_{j+1} * n_{j+2} * ... * n_k * w$

for $j = k$: $w_k = w$

ICG: translation of array references (1)

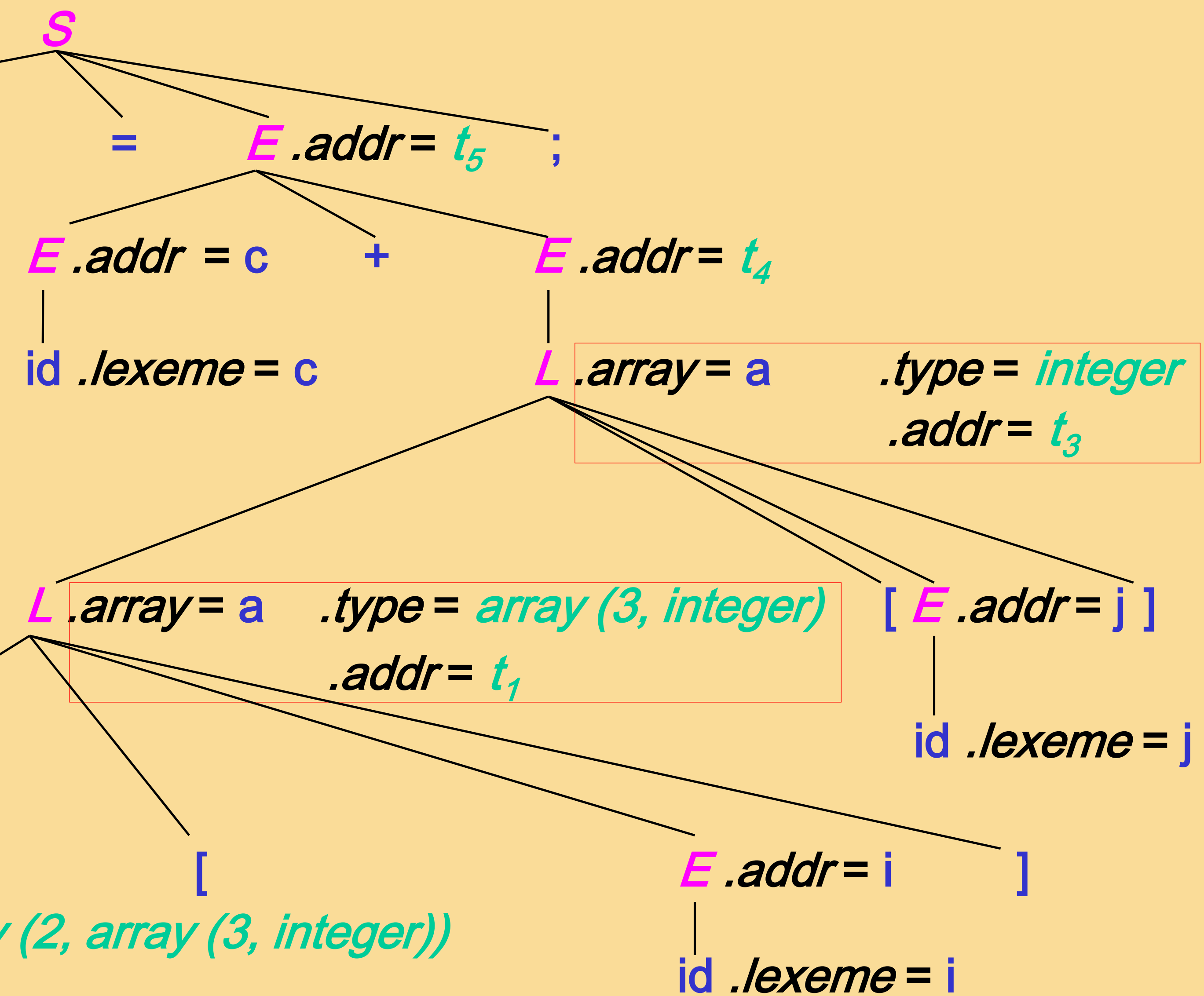
$L \rightarrow L$ [E] | id [E]

- ***L .addr***
  - sum of the terms $i_j * w_j$
- ***L .array***
  - pointer to the symbol-table entry for the array name
  - ***L .array.base***
    - base address of the array
  - ***L .array.type***
    - type of the array
  - ***L .array.type.elem***
    - type of the array elements
- ***L .type***
  - type of the sub-array generated by ***L***
  - ***L .type.width***
    - width of the sub-array generated by ***L***
  - ***L .type.elem***
    - type of the elements of the sub-array generated by ***L***

ICG: translation of array references (2)

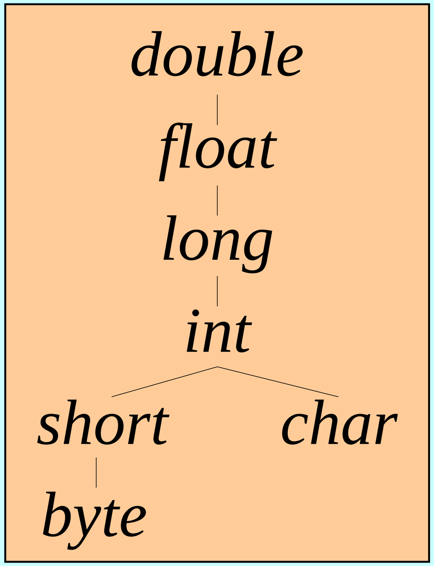
```
S → id = E ;      { gen ( top.get ( id .lexeme ) "=" E .addr ) }
   | L = E ;      { gen ( L .array.base "[" L .addr "]" "=" E .addr ) }
E → E1 + E2       { E .addr = new Temp ( ) ;
                    gen ( E .addr "=" E1 .addr "+" E2 .addr ) }
   | id           { E .addr = top.get ( id .lexeme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

ICG: translation of x = c + a [i] [j] ;



$t_1 = i * 12$
$t_2 = j * 4$
$t_3 = t_1 + t_2$
$t_4 = a [t_3]$
$t_5 = c + t_4$
$x = t_5$

SA: type conversions (1)




```
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
```

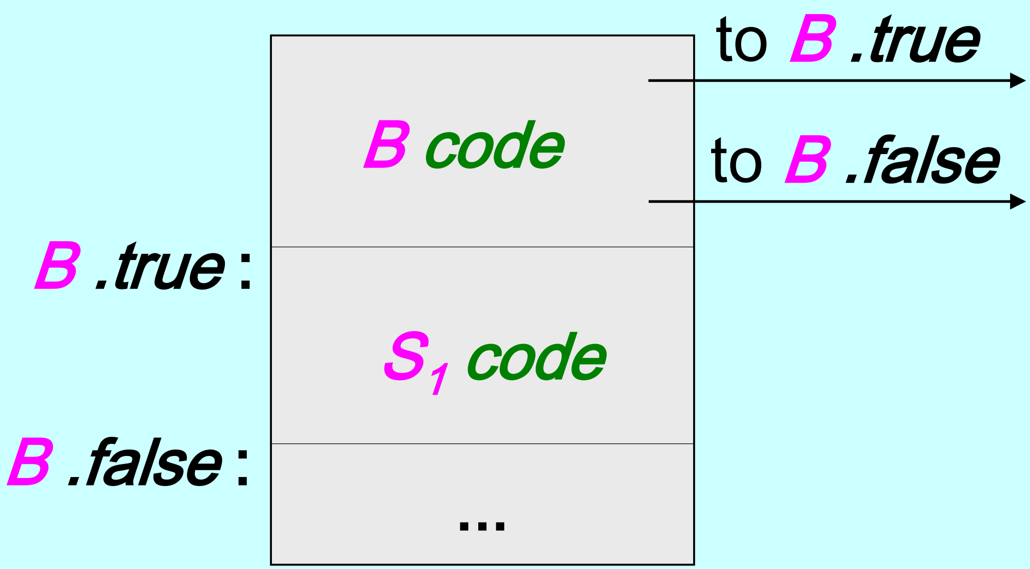
- function ***widen*(*a* , *t* , *w*)** generates type conversions if needed to widen an address ***a*** of type ***t*** into an address of type ***w***
- function ***max*(*t*$_1$, *t*$_2$)** returns the maximum of two types in the widening hierarchy

SA: type conversions (2)

$$
\begin{array}{l}
E \rightarrow E_1 + E_2 \quad \{\, E.type = max(E_1.type\ ,\ E_2.type)\ ; \\
\qquad a_1 = widen(E_1.addr, E_1.type\ ,\ E.type)\ ; \\
\qquad a_2 = widen(E_2.addr, E_2.type\ ,\ E.type)\ ; \\
\qquad E.addr = new\ Temp(\,)\ ; \\
\qquad gen(E.addr\ \text{“=”}\ a_1\ \text{“+”}\ a_2)\,\}
\end{array}
$$

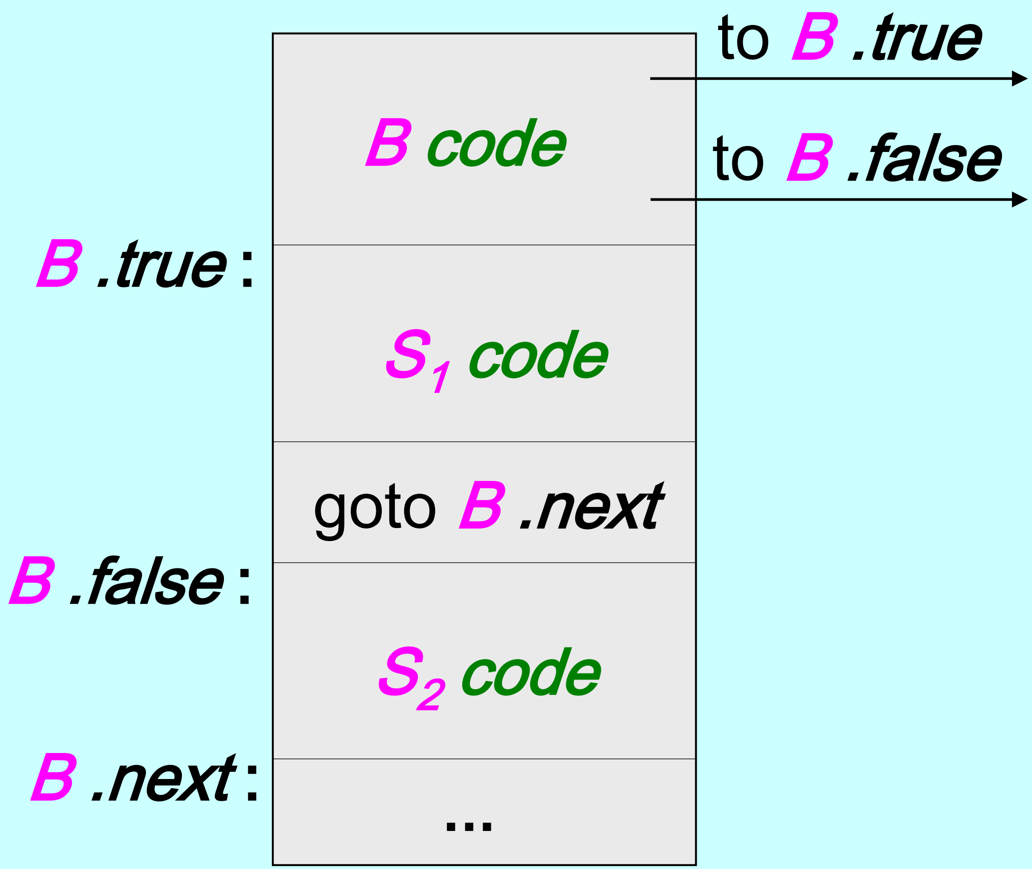
ICG: translation of flow-of-control statements (1)

$S \rightarrow$ if (B) S_1



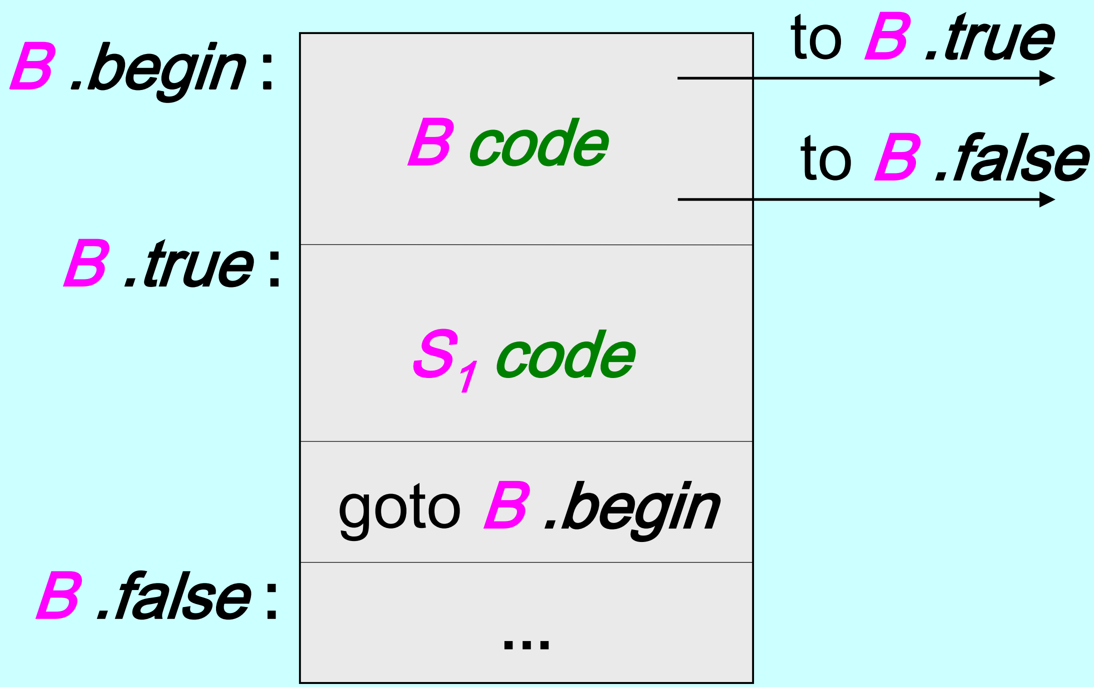
ICG: translation of flow-of-control statements (2)

$S \rightarrow$ if (B) S_1 else S_2



ICG: translation of flow-of-control statements (3)

$S \rightarrow$ while (B) S_1



ICG: translation of flow-of-control statements (4)

```
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 ) }
```

ICG: translation of Boolean expressions

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

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

- ***AND*** (**&&**) and ***OR*** (**||**) operators are *left associative*
- ***NOT*** (**!**) takes *precedence* over ***AND***, which takes *precedence* over ***OR***
- the semantic definition of the programming language determines whether all parts of an expression must be evaluated

ICG: evaluation of Boolean expressions

```
if ( x < 100 || x > 200 && x != y ) x = 0 ;
```

↓

if x < 100 goto L_1	t_3 = false
t_1 = false	goto L_6
goto L_2	L_5 : t_3 = true
L_1 : t_1 = true	L_6 : $t_4 = t_2$ && t_3
L_2 : if x > 200 goto L_3	$t_5 = t_1$ || t_4
t_2 = false	if t_5 goto L_7
goto L_4	goto L_8
L_3 : t_2 = true	L_7 : x = 0
L_4 : if x != y goto L_5	L_8 :

ICG: control-flow translation of Boolean expressions (1)

```
if ( x < 100 || x > 200 && x != y ) x = 0 ;
```

↓

```
      if  x < 100 goto L2
      goto L3
L3 :  if  x > 200 goto L4
      goto L1
L4 :  if  x != y goto L2
      goto L1
L2 :  x = 0
L1 :
```

FL&C

ICG: control-flow translation of Boolean expressions (2)

$B \rightarrow$	{ B_1 .true = B .true ; B_1 .false = new Label ( ) }
B_1 ||	{ B_2 .true = B .true ; B_2 .false = B .false ; gen (B_1 .false) }
B_2	
$B \rightarrow$	{ B_1 .true = new Label ( ) ; B_1 .false = B .false }
B_1 &&	{ B_2 .true = B .true ; B_2 .false = B .false ; gen (B_1 .true) }
B_2	
$B \rightarrow$!	{ B_1 .true = B .false ; B_1 .false = B .true }
B_1	
$B \rightarrow E_1$ rel E_2	{ gen (“if ” E_1 .addr rel.op E_2 .addr “goto” B .true) ; gen (“goto” B .false) }
$B \rightarrow$ true	{ gen (“goto” B .true) }
$B \rightarrow$ false	{ gen (“goto” B .false) }

ICG: translation of flow-of-control statements (5)

```
while ( a < x )
    if ( c > d )
        x = y + z ;
    else
        x = y - z ;
```

→

L_1: if a < x goto L_2
goto L_{next}
L_2: if c > d goto L_3
goto L_4
L_3: $t_1 = y + z$
$x = t_1$
goto L_1
L_4: $t_2 = y - z$
$x = t_2$
goto L_1

L_{next} :

ICG: back-patching (1)

- in the code for flow-of-control statements, *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 separate pass of translation is needed to *bind labels* to *instruction addresses*
- a complementary approach, called ***back-patching***, 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*
- each such jump is put on a *list of jumps* having the *same target*
- jump instructions in a list are then *completed* when the *proper target* can be *determined*

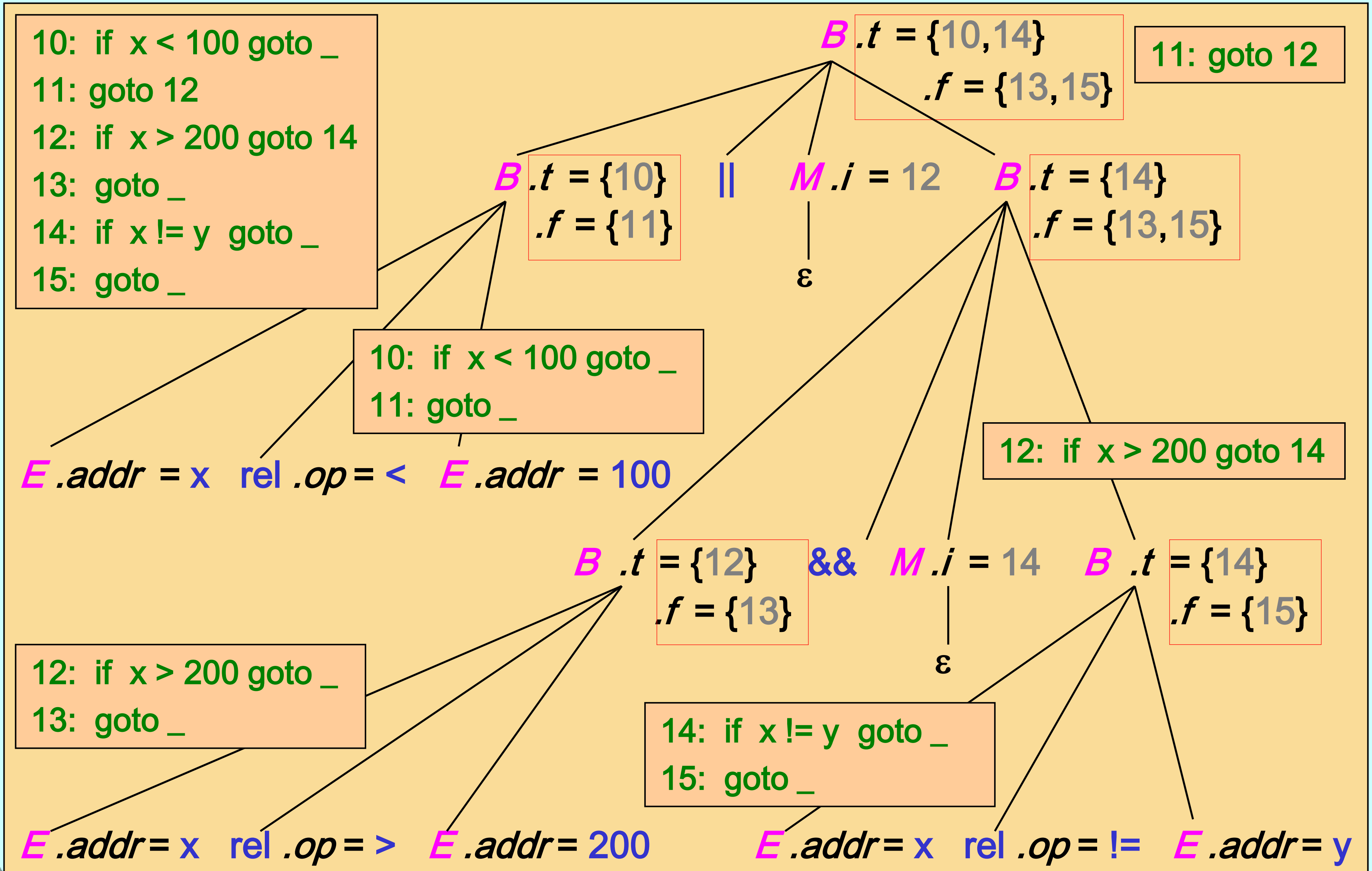
ICG: back-patching (2)

- function ***makelist*(*i*)** creates a new list of jumps containing only the index ***i*** into the sequence of instructions
  - returns a pointer to the newly created list
- function ***merge*(p_1 , p_2)** concatenates the lists pointed to by p_1 and p_2
  - returns a pointer to the concatenated list
- function ***backpatch*(*p* , *i*)** inserts ***i*** as the target label for each of the instructions on the list pointed to by ***p***

ICG: back-patching 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 }

ICG: translation of x < 100 || x > 200 && x != y



ICG: back-patching for flow-of-control statements (1)

$S \rightarrow$ 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 _") }

ICG: back-patching for flow-of-control statements (2)

```
while ( a < x )
    if ( c > d )
        x = y + z ;
    else
        x = y - z ;
```

→

10: if a < x goto 12
11: goto _
12: if c > d goto 14
13: goto 17
14: t_1 = y + z
15: x = t_1
16: goto 10
17: t_2 = y - z
18: x = t_2
19: goto 10

S .nextlist = {11}

ICG: back-patching for flow-of-control statements (3)