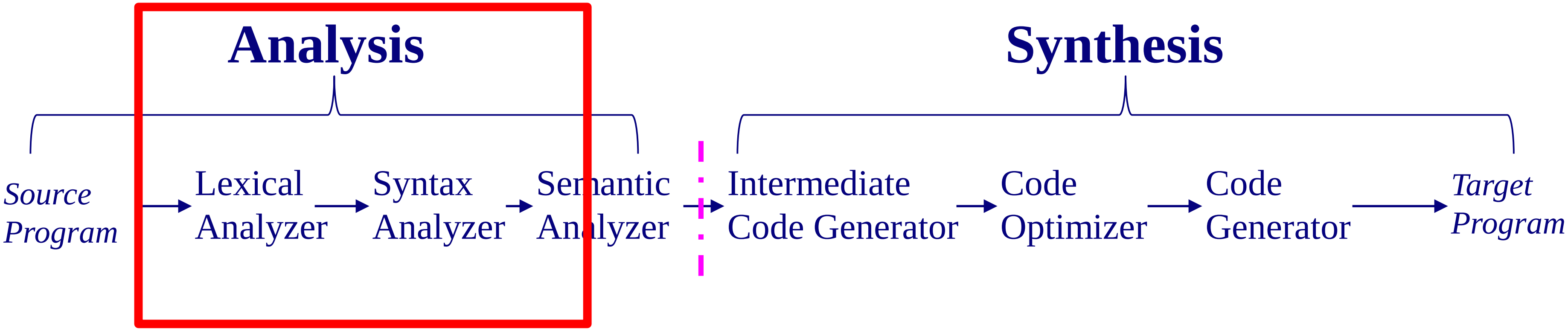
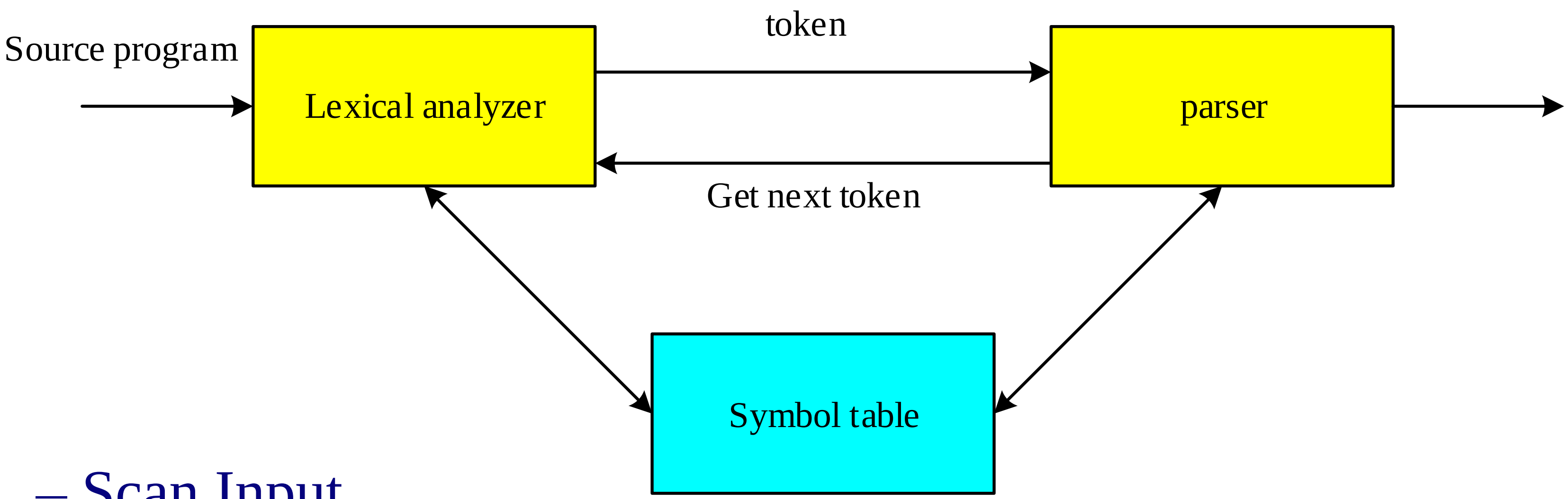
Midterm review

Compiling system

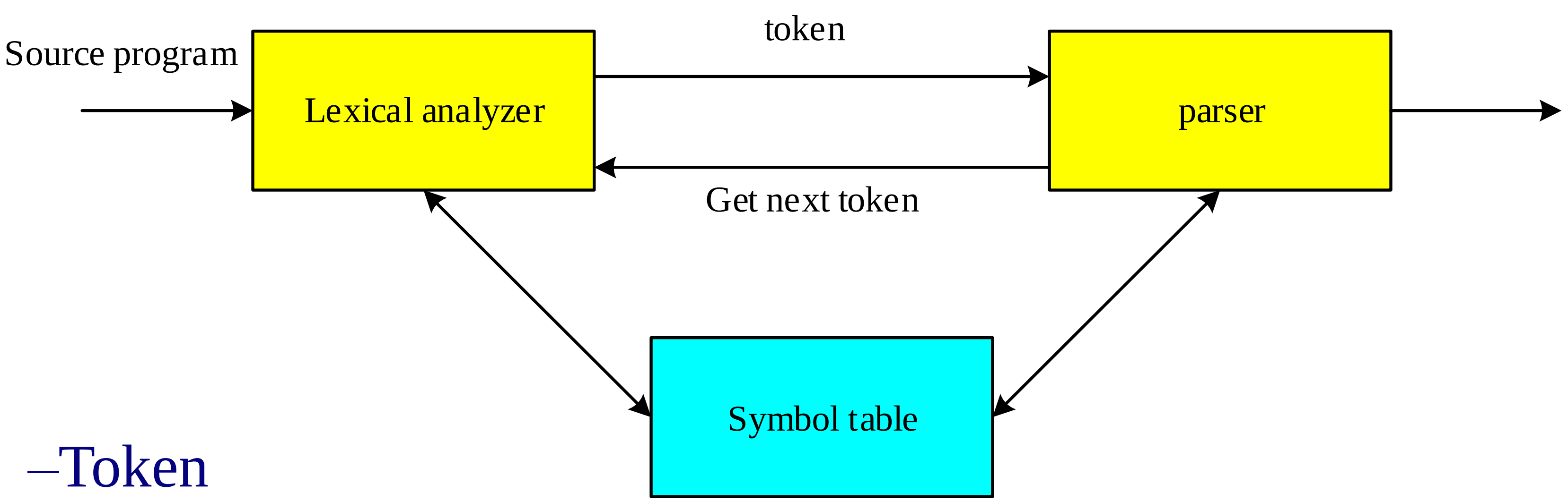


Lexical Analysis



– Scan Input
– Remove WS, NL, …
– Identify Tokens
– Create Symbol Table
– Insert Tokens into ST
– Generate Errors
– Send Tokens to Parser

Lexical Analysis



–Token

–Regular expression

–Finite state automata: NFA and DFA

–RE→ NFA→DFA

–RE→ DFA

–Minimization

–Handle error

Regular Definitions

Grammar:

stmt → |if *expr* then *stmt*
|if *expr* then *stmt* else *stmt*
| ε

expr → *term* relop *term* | *term*

term → id | num

if → if

then → then

else → else

relop → < + <= + > + >= + = + <>

id → letter (letter | digit)*

num → digit $^+$ (. digit $^+$) ? (E(+ | -) ? digit $^+$) ?

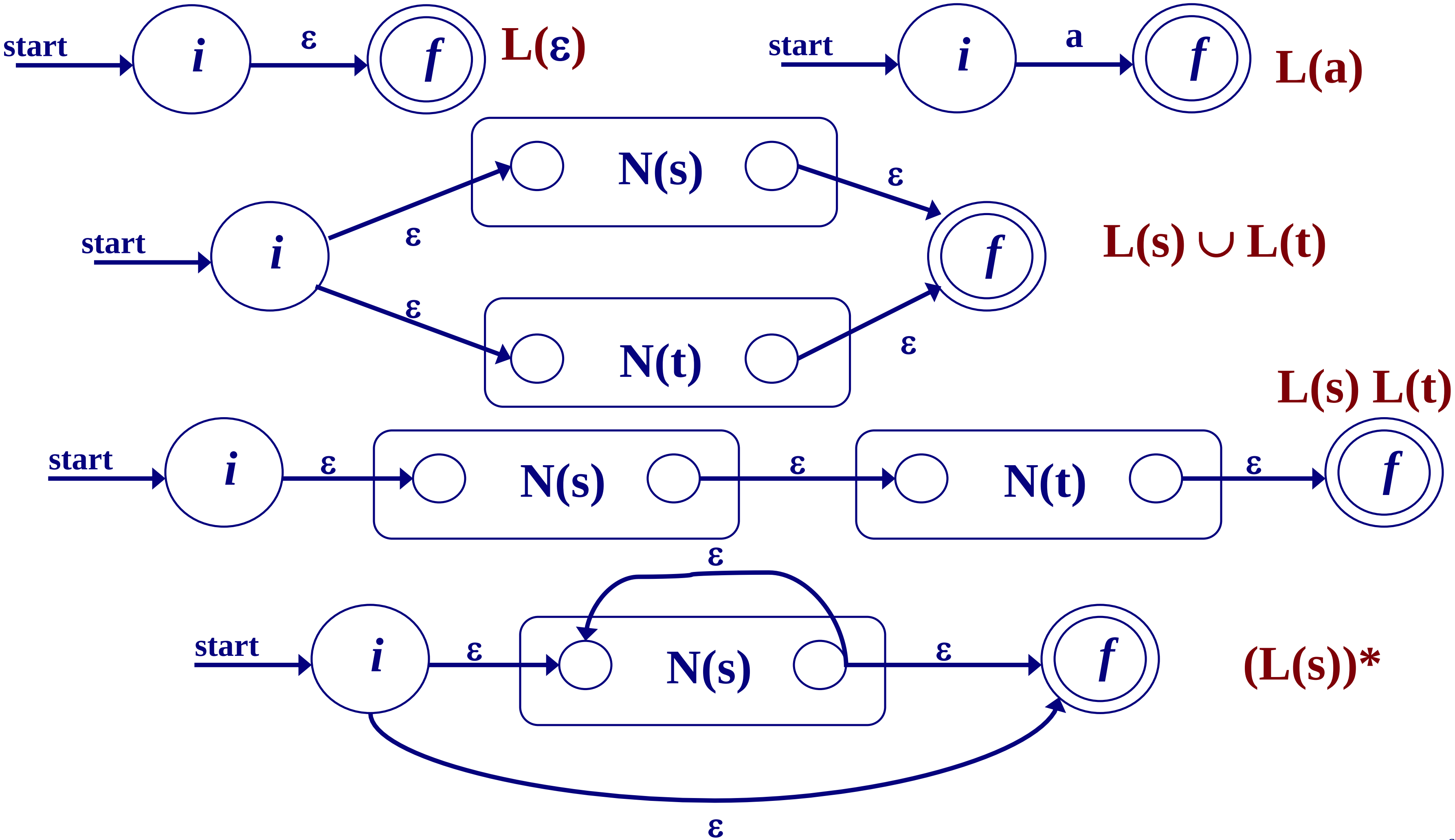
(a + b) * a

– RE→ NFA→DFA

– RE→ DFA

RE2NFA

Regular expression r::=ε | a | r + s | rs | r*   RE to NFA



NFA2DFA

put **ε-closure($\{s_0\}$)** as an unmarked state into the set of DFA (DStates)

while (there is one unmarked S_1 in DStates) do

mark S_1

for (each input symbol a)

$S_2 \leftarrow \varepsilon$-closure(move($S_1$,a))

if (S_2 is not in DStates) then

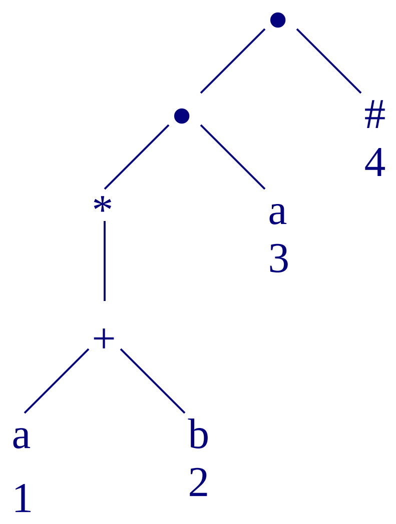
add S_2 into DStates as an unmarked state

transfunc[S_1,a] $\leftarrow S_2$

- the start state of DFA is ε-closure($\{s_0\}$)
- a state S in DStates is an accepting state of DFA if a state in S is an accepting state of NFA

RE2DFA

$(a+b)^*\, a \rightarrow (a+b)^*\, a\, \#$ augmented regular expression



Syntax tree of $(a+b)^*\, a\, \#$

- ✓ each symbol is numbered (positions)
- ✓ each symbol is at a leave
- ✓ inner nodes are operators

Then each alphabet symbol (plus #, exclude **ε**) will be numbered (position numbers).

How to evaluate firstpos, lastpos, nullable

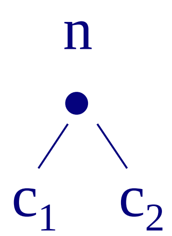
n	nullable(n)	firstpos(n)	lastpos(n)
leaf labeled ε	**true**	**Φ**	**Φ**
leaf labeled with position i	**false**	**{i}**	**{i}**
$+$ c_1 c_2	**nullable(c_1) or nullable(c_2)**	**firstpos(c_1) ∪ firstpos(c_2)**	**lastpos(c_1) ∪ lastpos(c_2)**
$\bullet$ c_1 c_2	**nullable(c_1) and nullable(c_2)**	**if (nullable(c_1))** **firstpos(c_1) ∪ firstpos(c_2)** **else firstpos(c_1)**	**if (nullable(c_2))** **lastpos(c_1) ∪ lastpos(c_2)** **else lastpos(c_2)**
$*$ c_1	**true**	**firstpos(c_1)**	**lastpos(c_1)**

Rules for computing nullable and firstpos.

How to evaluate followpos

> Two-rules define the function followpos:

1. If **n** is **concatenation-node** with left child c_1 and right child c_2, and **i** is a position in **lastpos(c_1)**, then all positions in **firstpos(c_2)** are in **followpos(i)**.
2. If **n** is a **star-node**, and **i** is a position in **lastpos(n)/lastpos(c),** then all positions in **firstpos(n)/firstpos(c)** are in **followpos(i)**.



> If firstpos and lastpos have been computed for each node, followpos of each position can be computed by making one depth-first traversal of the syntax tree.

Algorithm (RE ➔ DFA)

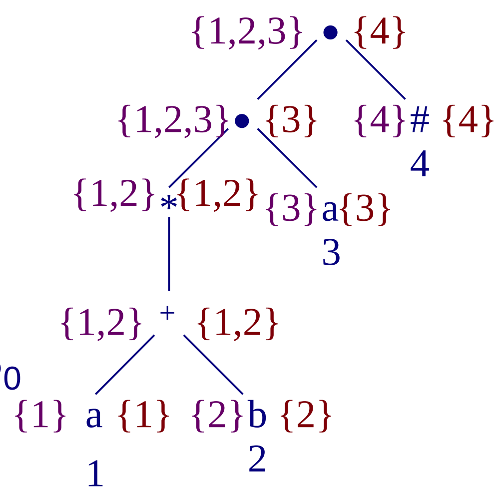
- Create the syntax tree of (r) #
- Calculate the functions: followpos, firstpos, lastpos, nullable
- Put firstpos(root) into the states of DFA as an unmarked state.
- *while (there is an unmarked state S in the states of DFA) do*
  - *mark **S***
  - ***for** each input symbol **a do***
    - *let $s_1,\ldots,s_n$ are positions in **S** and symbols in those positions are **a***
    - $S' \leftarrow followpos(s_1) \cup \ldots \cup followpos(s_n)$
    - ***move(S,a) ← S'***
    - ***if** (**S'** is not empty and not in the states of DFA)*
      - *put **S'** into the states of DFA as an unmarked state.*

- *the start state of DFA is firstpos(root)*
- *the accepting states of DFA are all states containing the position of #*

Example -- (a + b) * a

1 2 3 4

followpos(1)={1,2,3}   followpos(2)={1,2,3}   followpos(3)={4}   followpos(4)={}



S_0=firstpos(root)={1,2,3}

$\Downarrow$ mark S_0

a: followpos(1) $\cup$ followpos(3)={1,2,3,4}=S_1 move(S_0,a)=S_1

b: followpos(2)={1,2,3}=S_0 move(S_0,b)=S_0

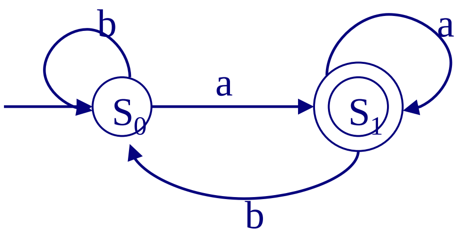
$\Downarrow$ mark S_1

a: followpos(1) $\cup$ followpos(3)={1,2,3,4}=S_1 move(S_1,a)=S_1

b: followpos(2)={1,2,3}=S_0 move(S_1,b)=S_0

start state: S_0

accepting states: {S_1}



	a	b
S_0	S_1	S_0
S_1	S_1	S_0

Minimizing the Number of States of a DFA

- partition the set of states into two groups:
  - G_1 : set of accepting states
  - G_2 : set of non-accepting states

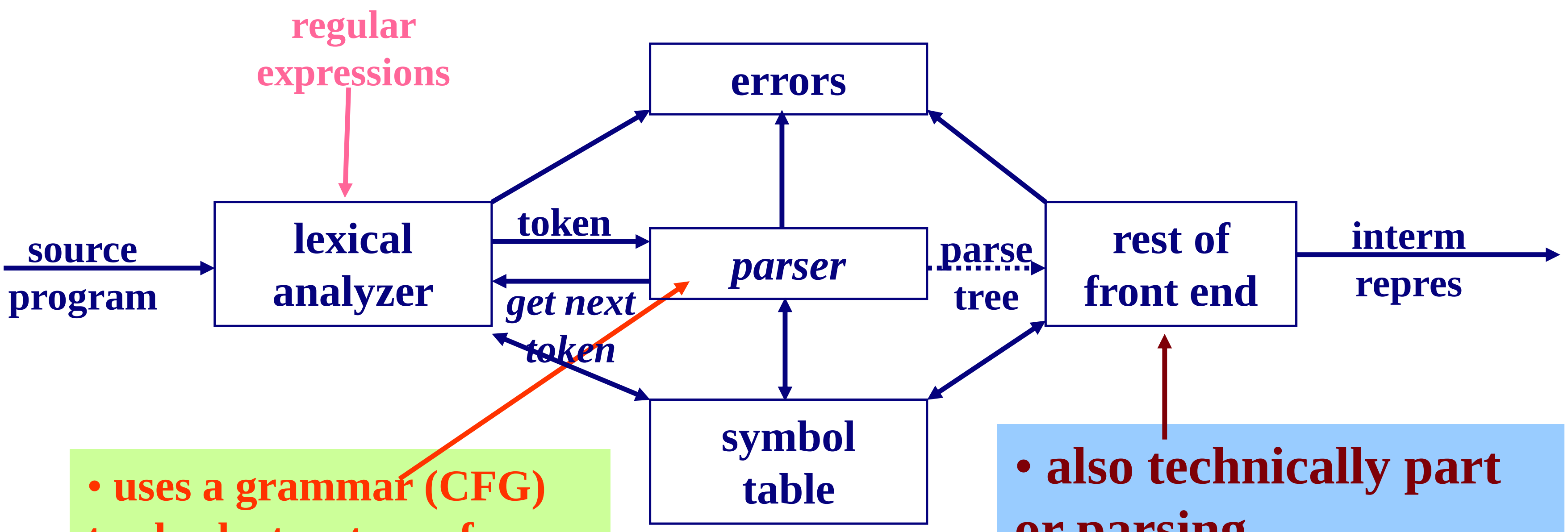
- For each new group G
  - partition G into subgroups such that states s_1 and s_2 are in the same group if

    for all input symbols a, states s_1 and s_2 have transitions to states in the same group.

- Start state of the minimized DFA is the group containing the start state of the original DFA.
- Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

Parsing During Compilation

- Parser works on a stream of tokens.
- The smallest item is a token.



- uses a grammar (CFG) to check structure of tokens
- produces a parse tree
- syntactic errors and recovery
- recognize correct syntax
- report errors

- also technically part or parsing
- includes augmenting info on tokens in source, type checking, semantic analysis

Concepts & Terminology

A Context Free Grammar (CFG), is described by (T, NT, S, PR), where:

T: Terminals / tokens of the language

NT: Non-terminals, S: Start symbol, $S \in NT$

PR: Production rules to indicate how T and NT are combined to generate valid strings of the language.

PR: NT → (T | NT)*

EXPR → if EXPR then EXPR else EXPR fi

| while EXPR loop EXPR pool

| id

Derivation Example

Leftmost: Replace the leftmost non-terminal symbol

$$E \underset{lm}{\Rightarrow} E \text{ op } E \underset{lm}{\Rightarrow} \text{id op } E \underset{lm}{\Rightarrow} \text{id} * E \underset{lm}{\Rightarrow} \text{id} * \text{id}$$

Rightmost: Replace the leftmost non-terminal symbol

$$E \underset{rm}{\Rightarrow} E \text{ op } E \underset{rm}{\Rightarrow} E \text{ op id} \underset{rm}{\Rightarrow} E * \text{id} \underset{rm}{\Rightarrow} \text{id} * \text{id}$$

Parse tree

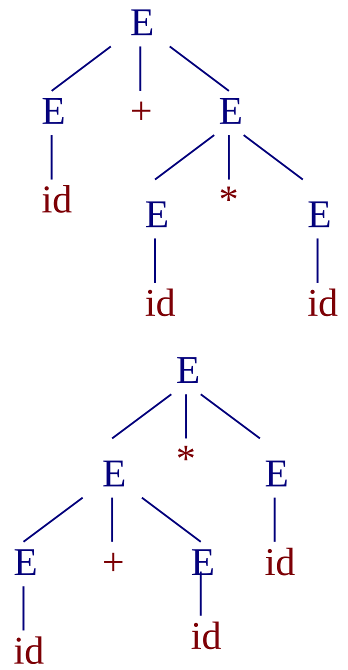
- ***top-down parsers*** vs. ***left-most derivation***
- ***bottom-up parsers*** vs. ***right-most derivation (reverse order)***

Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

$$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$$
$$\Rightarrow id+id*E \Rightarrow id+id*id$$

$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E$$
$$\Rightarrow id+id*E \Rightarrow id+id*id$$



Two parse trees for id+id*id.

1. **Grammar rewritten** to eliminate the ambiguity
2. Enforce **precedence** and **associativity**
3. **{ $a^i b^i$ | i>=0} is context-free**
4. **{ $a^i b^i c^i$ | i>=0} is not context-free**

Top-Down Parsing

- The parse tree is created top to bottom (from root to leaves).
- **By always replacing the leftmost non-terminal symbol via a production rule, we are guaranteed of developing a parse tree in a left-to-right fashion that is consistent with scanning the input.**
- Top-down parser
  - **Recursive-Descent Parsing**
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient
    - Left-recursion

$A \Rightarrow aBc \Rightarrow adDc \Rightarrow adec$

(scan a, scan d, scan e, scan c - accept!)

  - **Predictive Parsing**
    - no backtracking, at each step, only one choices of production to use
    - efficient
    - needs a special form of grammars (**LL(k) grammars, k=1 in practice**).
    - **Recursive Predictive Parsing** is a special form of Recursive Descent parsing without backtracking.
    - **Non-Recursive (Table Driven) Predictive Parser** is also known as LL(k) parser.

Implementation of Recursive-Descent

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }

bool E2() { return T() && term(PLUS) && E(); }

bool E() {TOKEN *save = next;
        return (next = save, E1())||(next = save, E2()); }

bool T1() { return term(INT); }

bool T2() { return term(INT) && term(TIMES) && T(); }

bool T3() { return term(OPEN) && E() && term(CLOSE); }
bool T() { TOKEN *save = next; return (next = save, T1())
                                    ||(next = save, T2())
                                    ||(next = save, T3());}
```

Immediate Left-Recursion

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\, A'$

$A' \rightarrow \alpha\, A' \mid \varepsilon$ an equivalent grammar: replaced by right-recursion

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\, A' \mid \ldots \mid \beta_n\, A'$

$A' \rightarrow \alpha_1\, A' \mid \ldots \mid \alpha_m\, A' \mid \varepsilon$ an equivalent grammar

Elimination of Left-Recursion

Algorithm eliminating left recursion.

- Arrange non-terminals in some order: $A_1 \ldots A_n$
- **for** i **from** 1 **to** n **do** {
    - **for** j **from** 1 **to** i-1 **do** {

      replace each production

      $$A_i \rightarrow A_j \, \gamma$$

      by

      $$A_i \rightarrow \alpha_1 \, \gamma \mid \ldots \mid \alpha_k \, \gamma$$

      where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$ are all the current A_j-productions.

      }
    - **eliminate immediate left-recursions among A_i-productions**

}

Algorithm to eliminate left recursion from a grammar.

Recursive Predictive Parsing (Example)

$A \rightarrow aBe \mid cBd \mid C$

$B \rightarrow bB \mid \varepsilon$

$C \rightarrow f$

```
proc C {   match the current token with f,
           and move to the next token; }

proc A {
     case of the current token {
          a:   - match the current token with a,
                 and move to the next token;
               - call B;
               - match the current token with e,
                 and move to the next token;
          c:   - match the current token with c,
                 and move to the next token;
               - call B;
               - match the current token with d,
                 and move to the next token;
          f:   - call C
     }
}

proc B {
     case of the current token {
b:- match the current token with b,
                 and move to the next token;
  - call B
          e,d:  do nothing
     }
}
```

first set of C/A

follow set of B

Left-Factoring -- Algorithm

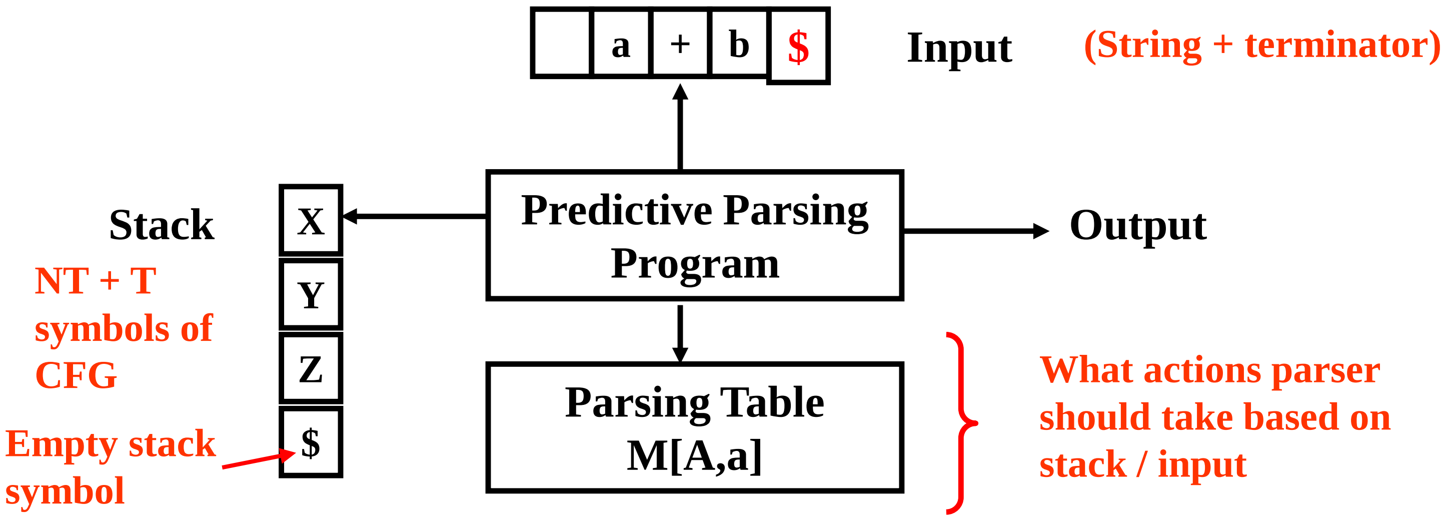
- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha\beta_1 \mid ... \mid \alpha\beta_n \mid \gamma_1 \mid ... \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid ... \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid ... \mid \beta_n$$

Non-Recursive / Table Driven



General parser behavior: X : top of stack a : current input

1. When X=a = $ halt, accept, success
2. When X=a ≠ $, POP X off stack, advance input, go to 1.
3. When X is a non-terminal, examine M[X,a]
   if it is an error → call recovery routine
   if M[X,a] = {X → UVW}, POP X, PUSH W,V,U
   DO NOT expend any input

Notice the pushing order

FIRST & FOLLOW

FIRST: **Is used to help find the appropriate reduction to follow given the top-of-the-stack non-terminal and the current input symbol.**

<u>Example:</u> If $A \rightarrow \alpha$, and a is in FIRST(α), then when a=input, replace A with α (in the stack).

(a is one of first symbols of α, so when A is on the stack and a is input, POP A and PUSH α.)

FOLLOW: **Is used when FIRST has a conflict, to resolve choices, or when FIRST gives no suggestion. When $\alpha \rightarrow \varepsilon$ or $\alpha \overset{*}{\Rightarrow} \varepsilon$, then what follows A dictates the next choice to be made.**

<u>Example:</u> If $A \rightarrow \alpha$, and b is in FOLLOW(A), then when $\alpha \overset{*}{\Rightarrow} \varepsilon$ <u>and</u> b is an input character, then we expand A with α, which will eventually expand to ε, of which b follows!

($\alpha \overset{*}{\Rightarrow} \varepsilon$: i.e., FIRST(α) contains ε.)

Compute FIRST for Any String X

1. If X is a terminal, FIRST(X) = {X}
2. If $X \rightarrow \varepsilon$ is a production rule, add ε to FIRST (X)
3. If X is a non-terminal, and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production rule

Place FIRST(Y_1) in FIRST (X)

if $Y_1 \Rightarrow \varepsilon$, Place FIRST ($Y_2$) in FIRST(X)

if $Y_2 \Rightarrow \varepsilon$, Place FIRST($Y_3$) in FIRST(X)

…

if $Y_{k-1} \Rightarrow \varepsilon$, Place FIRST($Y_k$) in FIRST(X)

NOTE: As soon as $Y_i \not\overset{*}{\Rightarrow} \varepsilon$, Stop.

Repeat above steps until no more elements are added to any FIRST( ) set.

Checking "$Y_i \Rightarrow \varepsilon$?" essentially amounts to checking whether ε belongs to FIRST(Y_i)

First(X) contains ε iff all Y_i contains ε

Compute FOLLOW (for non-terminals)

1. If S is the start symbol ➔ **$** is in FOLLOW(S) **Initially S$**

2. If $A \rightarrow \alpha B \beta$ is a production rule
➔ everything in **FIRST(β)** is FOLLOW(B) except ε

3. If ($A \rightarrow \alpha B$ is a production rule) or
($A \rightarrow \alpha B \beta$ is a production rule and ε is in FIRST(β))
➔ everything in **FOLLOW(A)** is in FOLLOW(B).
(Whatever followed A must follow B, since nothing follows B from the production rule)

We apply these rules until nothing more can be added to any FOLLOW set.

Constructing LL(1) Parsing Table

Algorithm:

1. Repeat Steps 2 & 3 for each rule $A \rightarrow \alpha$
2. Terminal a in FIRST(α)? Add $A \rightarrow \alpha$ to M[A, a]

3.1 ε in FIRST(α)? Add $A \rightarrow \alpha$ to M[A, b] for all terminals b in FOLLOW(A).

3.2 ε in FIRST(α) and \$ in FOLLOW(A)? Add $A \rightarrow \alpha$ to M[A, \$]

4. All undefined entries are errors.

Bottom-Up Parsing

- Goal: creates the parse tree of the given input starting from leaves towards the root.
- How: construct the right-most derivation of the given input in the reverse order.

$$S \Rightarrow r_1 \Rightarrow \ldots \Rightarrow r_n \Rightarrow \omega \quad \text{(the right-most derivation of } \omega\text{)}$$

$$\leftarrow \quad \text{(finds the right-most derivation in the reverse order)}$$

- Techniques:
  - General technique: **shift-reduce parsing**
    - ✓Shift: pushes the current symbol in the input to a stack.
    - ✓Reduction: replaces the symbols $X_1X_2\ldots X_n$ at the top of the stack by A if $A \rightarrow X_1X_2\ldots X_n$.
  - LR parsers (SLR, LR, LALR)

Bottom-up Parsing

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$$S=\gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = \omega$$

input string

handle-pruning

Let ω = input string

repeat

pick a non-empty substring β of ω where $A \rightarrow \beta$ is a production

if (no such β)

backtrack

else

replace one β by A in ω

until ω = "S" (the start symbol) or all possibilities are exhausted

handle ($A \rightarrow \beta$, pos)

Shift-Reduce Parser

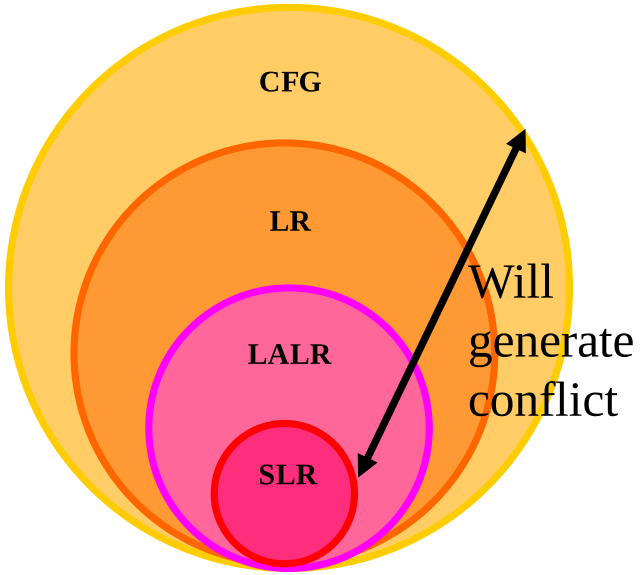
- Shift-reduce parsers require a stack and an input buffer
  - Initial stack just contains only the end-marker **$**
  - The end of the input string is marked by the end-marker **$**.

- There are four possible actions of a shift-parser action:

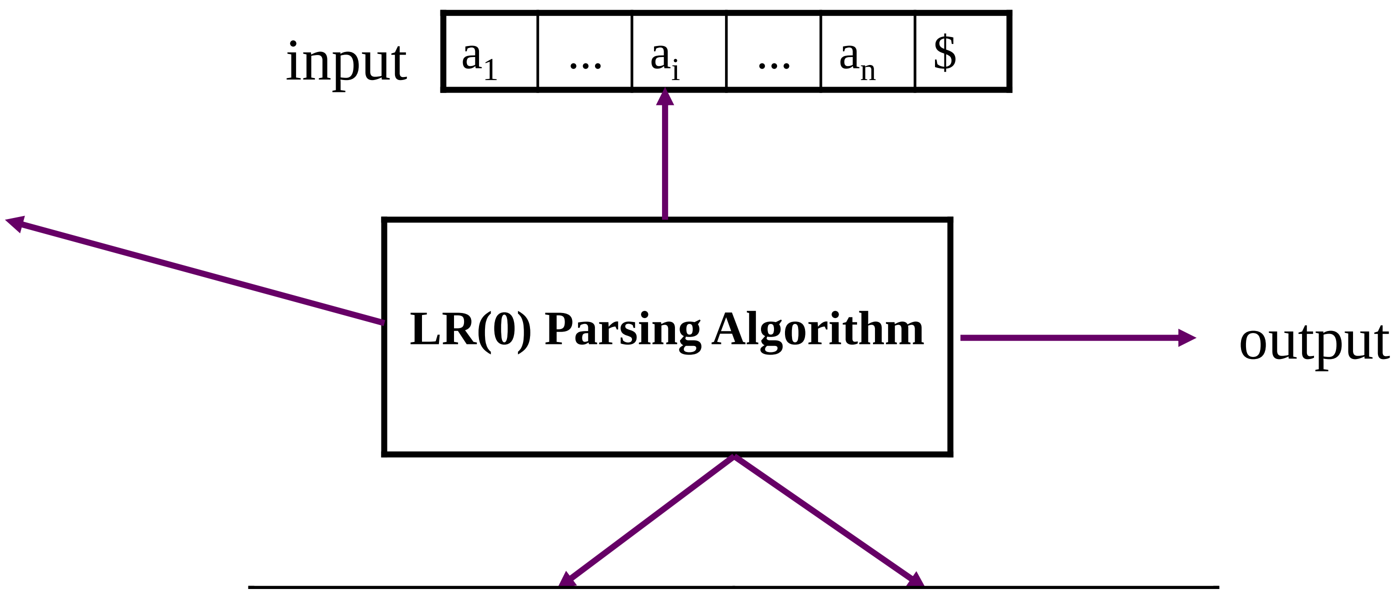
1. **Shift** : The next input symbol is shifted onto the top of the stack.
2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
3. **Accept**: Successful completion of parsing.
4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.

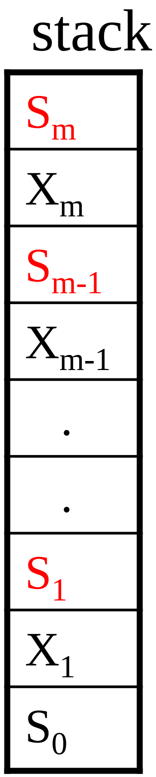
Shift-Reduce Parsers

1. **LR-Parsers**
   - covers wide range of grammars.
     - **SLR** – simple LR parser
     - **LR** – most general LR parser
     - **LALR** – intermediate LR parser (lookahead LR parser)
   - SLR, LR and LALR work same, only their parsing tables are different.



LR(0) Parsing Algorithm Implmentation





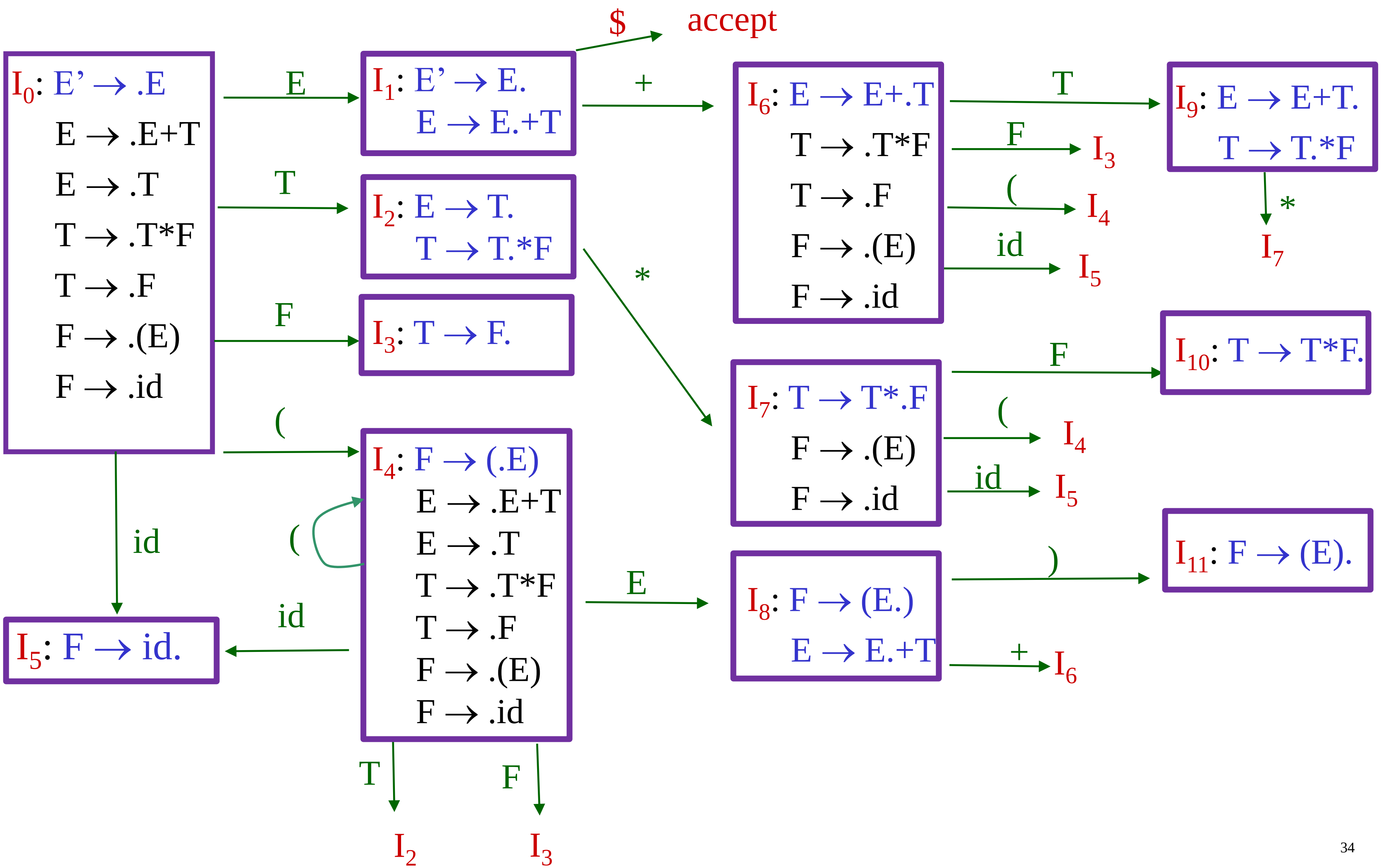
Idea: Assume: stack contains β and next input is a

DFA on input β terminates in state s

- Reduce by $X \to \beta$
  - if s contains item $X \to \beta\bullet$
- Shift if s contains item $X \to \beta\bullet a\omega$, i.e. DFA has (s, a, s')

How to update state?

The Canonical LR(0) Collection -- Example



Actions of A LR(0)-Parser

1. **shift s** -- shifts the next input symbol a_i and the state **s** onto the stack (s_n where n is a state number)

   $$(S_o\ X_1\ S_1 \ldots X_m\ S_m,\ a_i\ a_{i+1} \ldots a_n\ \$) \Rightarrow (S_o\ X_1\ S_1 \ldots X_m\ S_m\ a_i\ s,\ a_{i+1} \ldots a_n\ \$)$$

2. **reduce $A \rightarrow \beta$**
   - pop $2|\beta|$ (=r) items from the stack;
   - then push **A** and **s** where **s=goto[s_{m-r},A]**

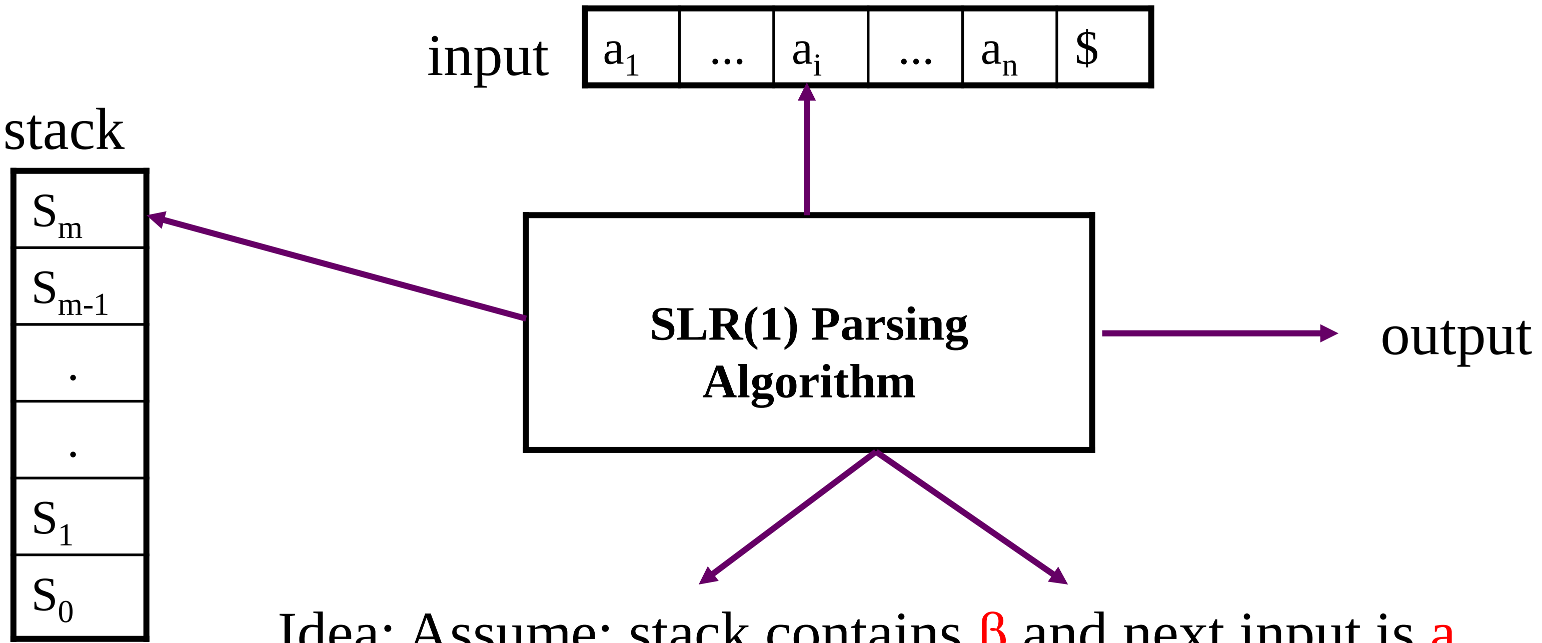
   $$(S_o\ X_1\ S_1 \ldots X_m\ S_m,\ a_i\ a_{i+1} \ldots a_n\ \$) \Rightarrow (S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ A\ s,\ a_i \ldots a_n\ \$)$$

   - Output is the reducing production reduce $A \rightarrow \beta$, (and others)

3. **Accept** – Parsing successfully completed

4. **Error** -- Parser detected an error (an empty entry in the action table)

SLR(1) Parsing Algorithm



Idea: Assume: stack contains β and next input is a
DFA on input β terminates in state s

- Reduce by $X \to \beta$
  - if **a in FOLLOW(X)** (details refer to next slide)
- Shift if s contains item $X \to \beta.a\omega$,
  i.e. DFA has (s, a, s')

Constructing SLR(1) Parsing Table)

(of an augumented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'.
   $C \leftarrow \{I_0,...,I_n\}$

2. Create the parsing action table as follows:
   - If a is a terminal, $A \rightarrow \alpha . \mathbf{a} \beta$ in I_i and goto(I_i,**a**)=I_j then action[i,a] is ***shift j*.**
   - If $A \rightarrow \alpha.$ is in I_i , then action[i,a] is ***reduce*** $\boldsymbol{A \rightarrow \alpha}$ **for all a in FOLLOW(A) where A≠S'.**
   - If $S' \rightarrow S.$ is in I_i , then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).

3. Create the parsing goto table :
   - for all non-terminals A, if goto(I_i,A)=I_j then goto[i,A]=j

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow .S$

Constructing Canonical LR(1) Parsing Tables

- In SLR method, the state i makes a reduction by $A \rightarrow \alpha$ when the current token is a:
  - if $A \rightarrow \alpha.$ in the I_i and a is FOLLOW(A)
- In some situations, βA cannot be followed by the terminal a in a right-sentential form when $\beta\alpha$ and the state i are on the top stack. This means that making reduction in this case is not correct.

$S \rightarrow AaAb$ $\qquad S \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab$ $\qquad S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$ $\qquad AaAb \Rightarrow Aa\,\varepsilon\,b$ $\qquad BbBa \Rightarrow Bb\,\varepsilon\,a$

$B \rightarrow \varepsilon$ $\qquad Aab \Rightarrow \varepsilon\,ab$ $\qquad Bba \Rightarrow \varepsilon\,ba$

{ I_0: $S \rightarrow AaAb$, $S \rightarrow BbBa$, $A \rightarrow \varepsilon$, $B \rightarrow \varepsilon$ }: reduce/reduce conflict

FOLLOW(A)=FOLLOW(B)={a,b}

LR(1) Item

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.

- A LR(0) item is:

$$A \rightarrow \alpha \bullet \beta$$

- A LR(1) item is:

$A \rightarrow \alpha \bullet \beta, a$ where **a** is the look-ahead of the LR(1) item

(**a** is a terminal or end-marker.)

LR(1) Automaton

- The states of LR(1) automaton are similar to the construction of the one for LR(0) automaton, except that *closure* and *goto* operations work a little bit different.

closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if $A \rightarrow \alpha \bullet B\beta, a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma,$ **b** will be in the closure(I) for each terminal b in **FIRST(βa)** .

LR(0) automaton

- if $A \rightarrow \alpha \bullet B\beta$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma,$ will be in the closure(I) .

Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for G'.
   $C \leftarrow \{I_0,...,I_n\}$

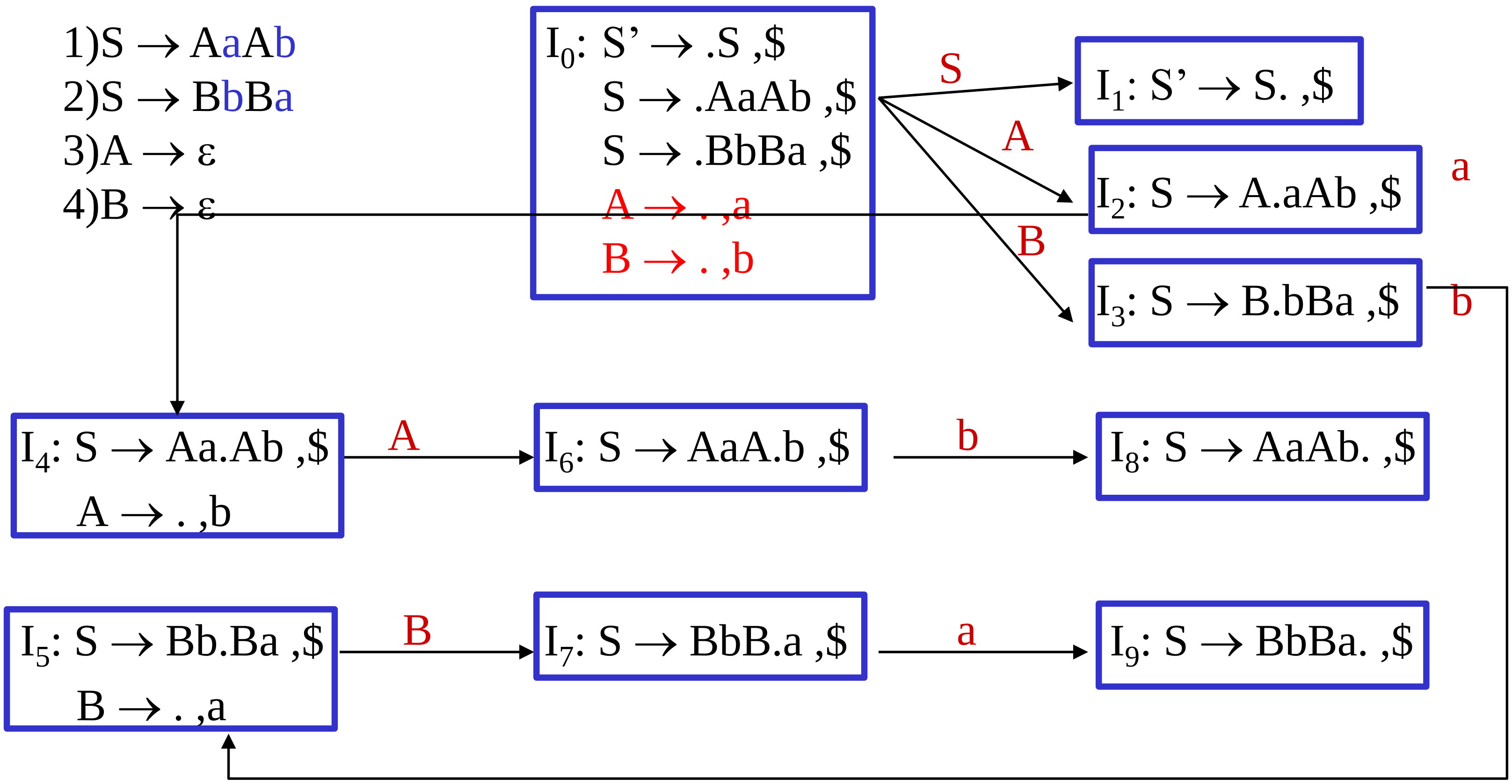
2. Create the parsing action table as follows
   - If a is a terminal, **$A \rightarrow \alpha \bullet a\beta$, b in I_i** and **goto(I_i,a)=I_j** then action[i,a] is ***shift j*.**
   - If **$A \rightarrow \alpha \bullet$, a** is in I_i , then **action[i,a]** is ***reduce $A \rightarrow \alpha$*** where $A \neq S'$.
   - If $S' \rightarrow S \bullet$, \$ is in I_i , then action[i,\$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(1).

3. Create the parsing goto table
   - for all non-terminals A, if goto(I_i,A)=I_j then goto[i,A]=j

4. All entries not defined by (2) and (3) are errors.

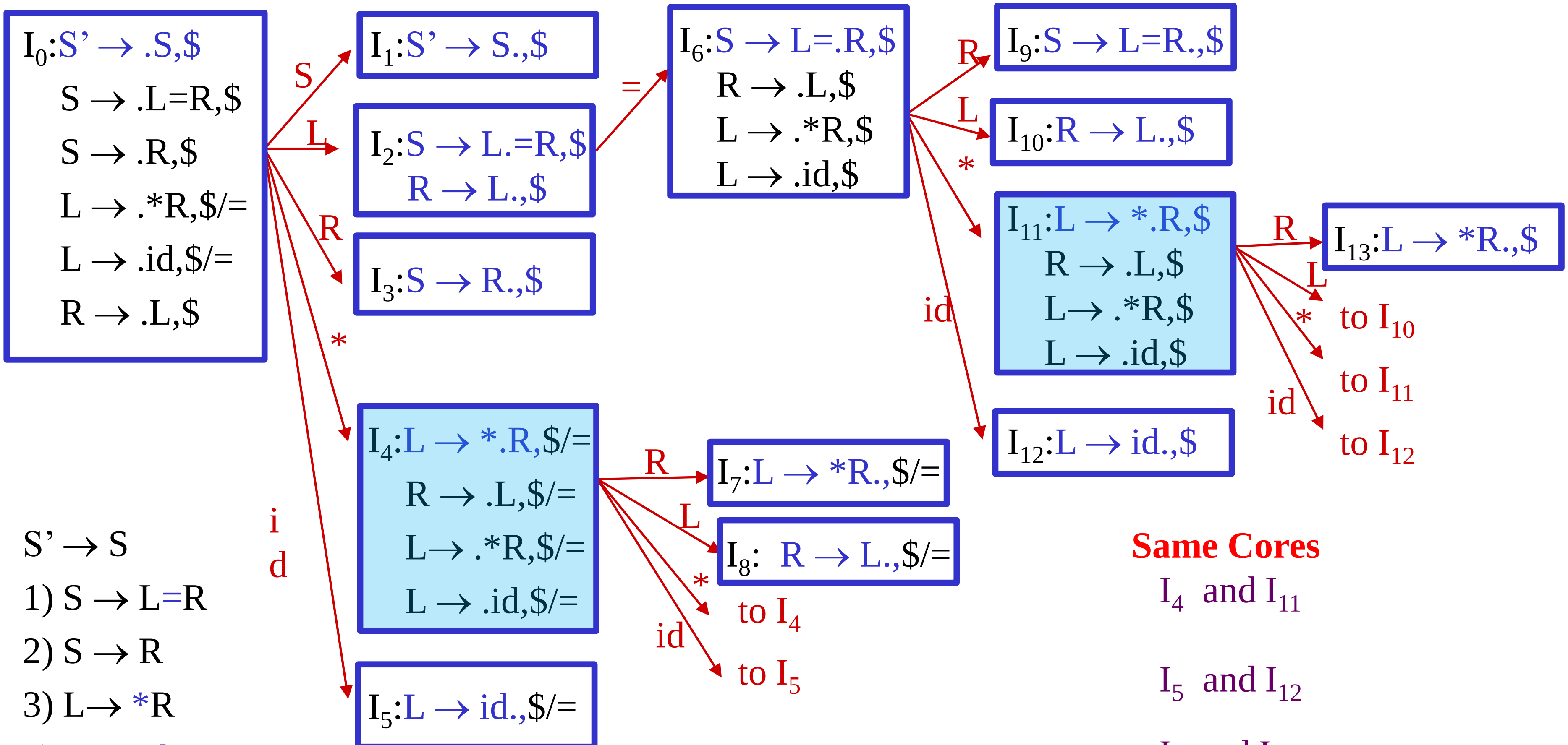
5. Initial state of the parser contains $S' \rightarrow .S$, \$

LR(1) Automaton example



if $S \rightarrow .AaAb, \$$ in closure(I) and $A \rightarrow \varepsilon$ is a production rule of G; then $A \rightarrow .,$ **a** will be in the closure(I) for each terminal a in **FIRST(AaAb$)** ={a}.

Canonical LR(1) Collection – Example2



S' → S
1) S → L=R
2) S → R
3) L→ *R
4) L → id
5) R → L

Same Cores

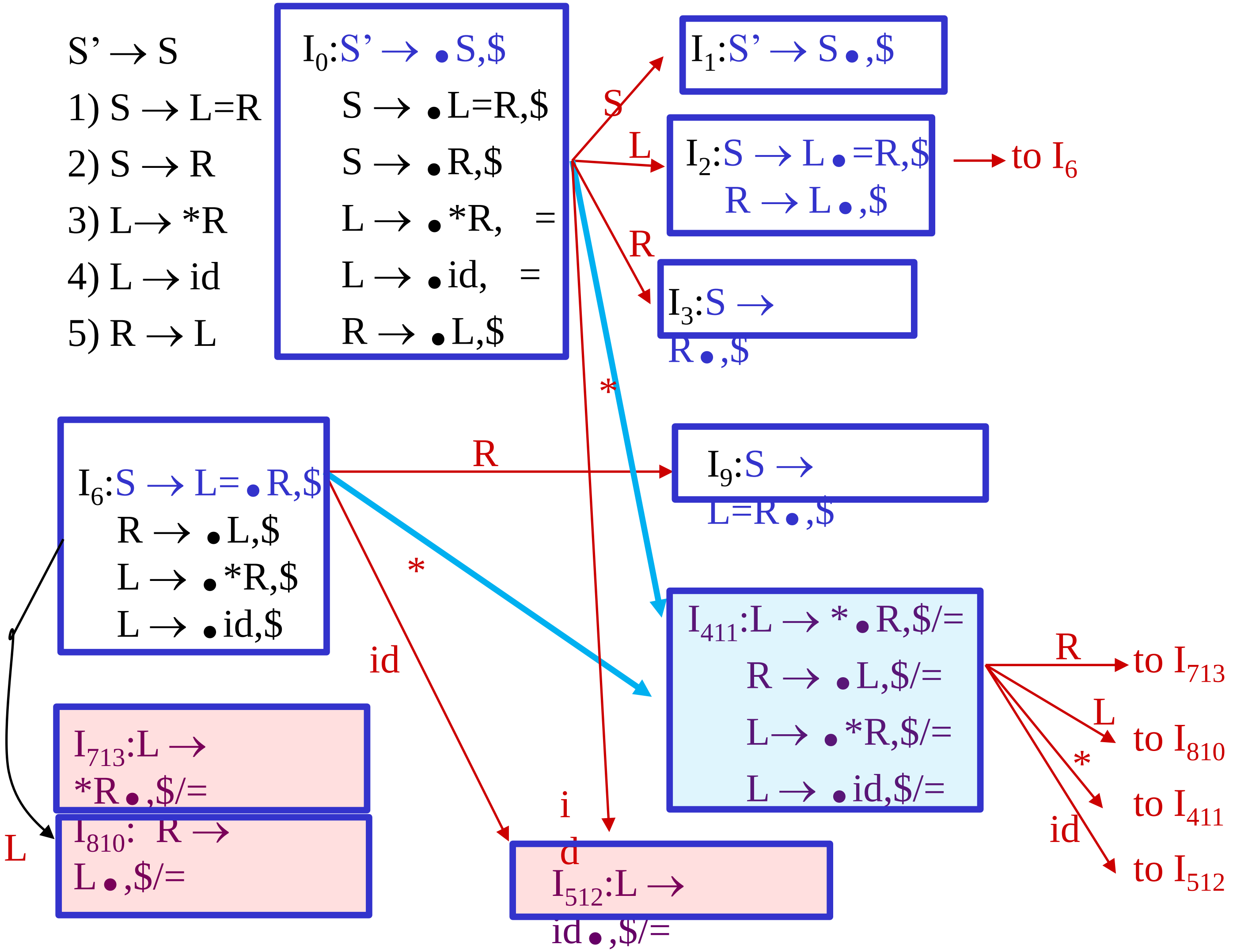
I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

Canonical LALR(1) Collection – Example2



Same Cores

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}