Introduction to Bottom Up Parser

Bottom-Up Parsing

- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root.
- A bottom-up parser tries to find the right-most derivation of the given input in the reverse order.

  $S \Rightarrow ... \Rightarrow \omega$ (the right-most derivation of ω)

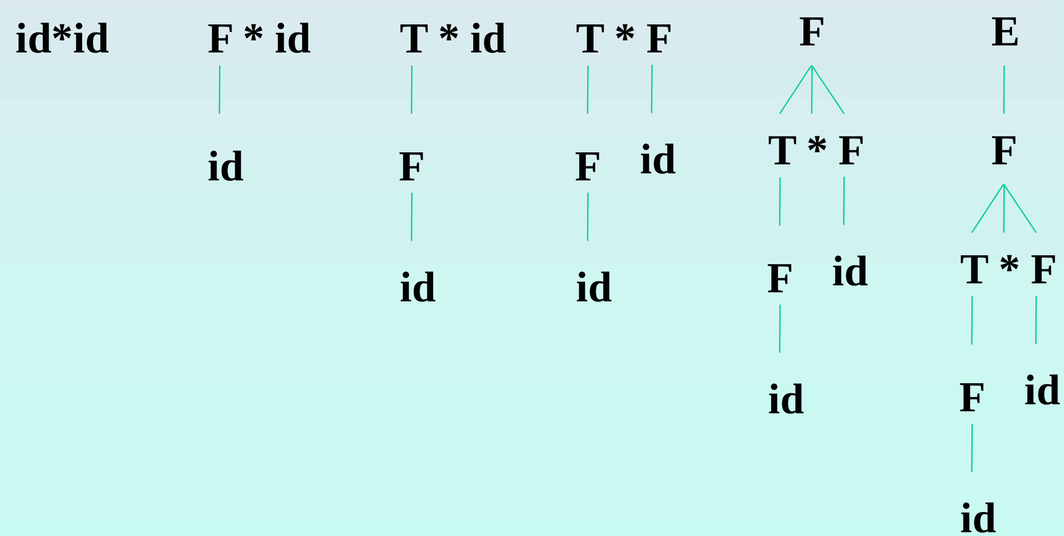
  $\leftarrow$ (the bottom-up parser finds the right-most derivation in the reverse order)

- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.
  - At each shift action, the current symbol in the input string is pushed to a stack.
  - At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.
  - There are also two more actions: accept and error.

Introduction

- Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)
- Example: id*id

E -> E + T | T
T -> T * F | F
F -> (E) | **id**



Shift-reduce parser

- The general idea is to shift some symbols of input to the stack until a reduction can be applied
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply
- A reduction is a reverse of a step in a derivation
- The goal of a bottom-up parser is to construct a derivation in reverse:
  - E=>T=>T*F=>T*id=>F*id=>id*id

• Bottom-Up Parsing

•Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.

•Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
•At the shift action, the current symbol in the input string is pushed to a stack.
•At each reduction, the symbols will replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

•Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.

A String $\xrightarrow[\text{reduce to}]{}$ the starting symbol

Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

  a string $\xrightarrow[\text{reduced to}]{}$ the starting symbol

- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

Rightmost Derivation: $S \overset{*}{\underset{rm}{\Rightarrow}} \omega$

Shift-Reduce Parser finds: $\omega \underset{rm}{\Leftarrow} \ldots \underset{rm}{\Leftarrow} S$

Shift-Reduce Parsing -- Example

$S \rightarrow aABb$ input string: aaabb

$A \rightarrow aA \mid a$ aaAbb

$B \rightarrow bB \mid b$ aAbb $\Downarrow$ reduction

aABb

S

$$S \underset{rm}{\Rightarrow} aABb \underset{rm}{\Rightarrow} aAbb \underset{rm}{\Rightarrow} aaAbb \underset{rm}{\Rightarrow} aaabb$$

Right Sentential Forms

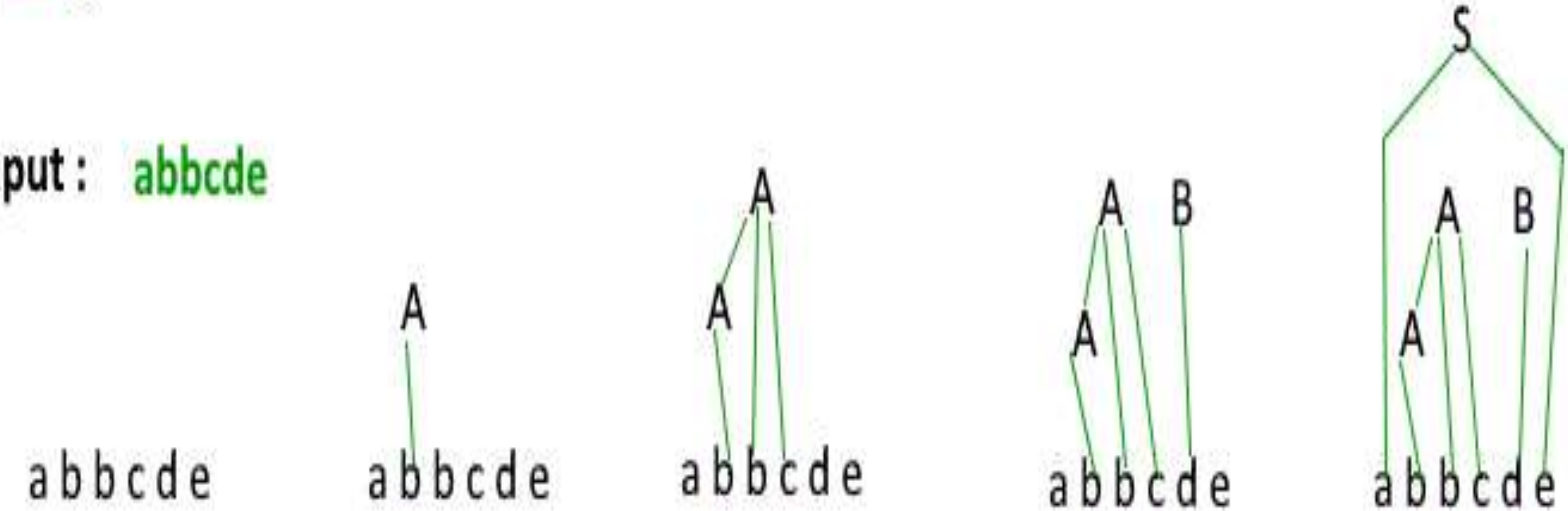
- How do we know which substring to be replaced at each reduction step?

S → aABe

A → Abc/b

B → d

Input : abbcde



abbcde ← aAbcde ← aAde ← aABe ← S

Handle

- Informally, a **handle** of a string is a substring that matches the right side of a production rule.
  - But not every substring matches the right side of a production rule is handle

- A **handle** of a right sentential form γ ($\equiv \alpha\beta\omega$) is

  a production rule $A \rightarrow \beta$ and a position of γ

  where the string β may be found and replaced by A to produce

  the previous right-sentential form in a rightmost derivation of γ.

$$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A \omega \underset{rm}{\Rightarrow} \alpha\beta\omega$$

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- We will see that ω is a string of terminals.

Handle pruning

- A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation

Right sentential form	Handle	Reducing production
id*id	id	F->id
F*id	F	T->F
T*id	id	F->id
T*F	T*F	E->T*F

Shift reduce parsing

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

Stack	Input
$	w$

- Acceptance configuration

Stack	Input
$S	$

Handle Pruning

- 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

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.

A Shift-Reduce Parser

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Right-Most Derivation of `id+id*id`

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$

$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$

<u>Right-Most Sentential Form</u>	<u>Reducing Production</u>
<u>id</u>+id*id	$F \rightarrow id$
<u>F</u>+id*id	$T \rightarrow F$
<u>T</u>+id*id	$E \rightarrow T$
E+<u>id</u>*id	$F \rightarrow id$
E+<u>F</u>*id	$T \rightarrow F$
E+T*<u>id</u>	$F \rightarrow id$
E+<u>T*F</u>	$T \rightarrow T*F$
<u>E+T</u>	$E \rightarrow E+T$
E	

<u>Handles</u> are red and underlined in the right-sentential forms.

A Stack Implementation of A Shift-Reduce Parser

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

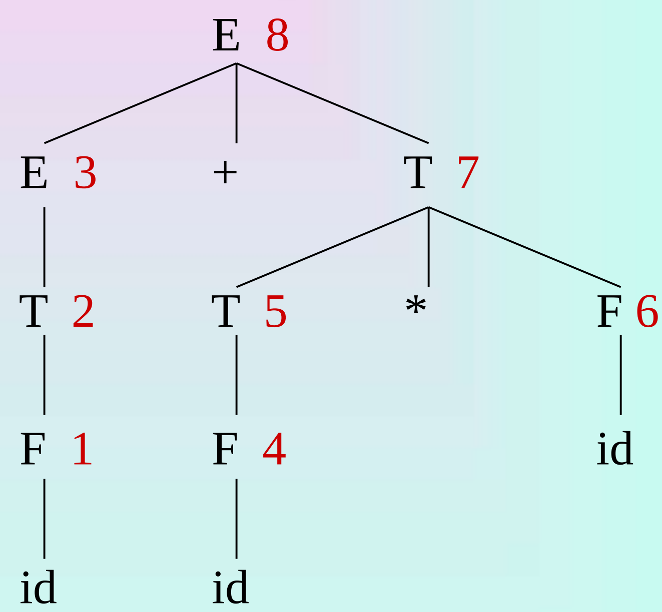
1. **Shift** : The next input symbol is shifted onto the top of the stack.
2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
3. **Accept**: Successful completion of parsing.
4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.

- Initial stack just contains only the end-marker $.
- The end of the input string is marked by the end-marker $.

A Stack Implementation of A Shift-Reduce Parser

Stack	Input	Action
\$	id+id*id\$	shift
\$id	+id*id\$	reduce by F → id
\$F	+id*id\$	reduce by T → F
\$T	+id*id\$	reduce by E → T
\$E	+id*id\$	shift
\$E+	id*id\$	shift
\$E+id	*id\$	reduce by F → id
\$E+F	*id\$	reduce by T → F
\$E+T	*id\$	shift
\$E+T*	id\$	shift
\$E+T*id	\$	reduce by F → id
\$E+T*F	\$	reduce by T → T*F
\$E+T	\$	reduce by E → E+T
\$E	\$	accept

Parse Tree



Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction.
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.

left to right scanning

right-most derivation

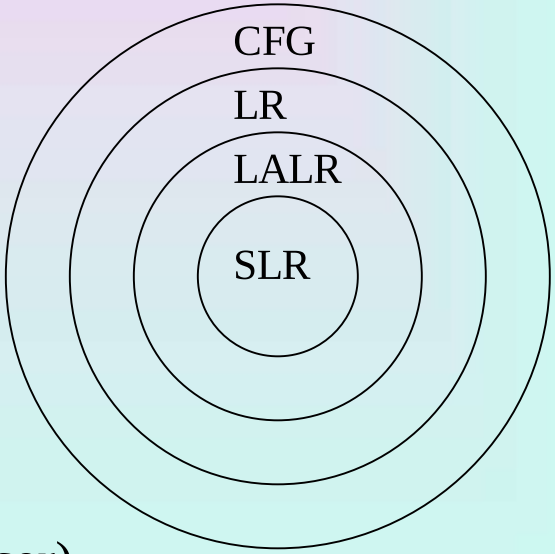
k lookhead

- An ambiguous grammar can never be a LR grammar.

Shift-Reduce Parsers

- There are two main categories of shift-reduce parsers

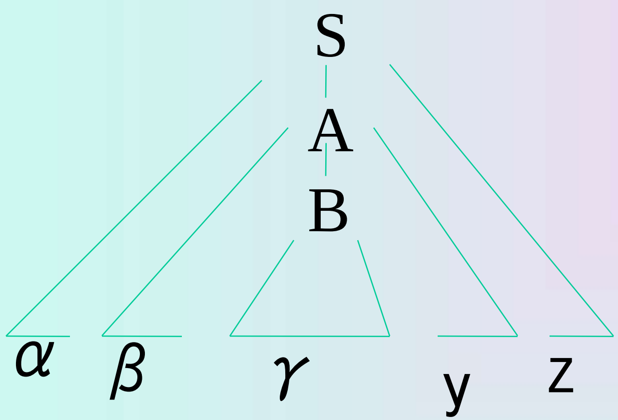
1. **Operator-Precedence Parser**
   – simple, but only a small class of grammars.

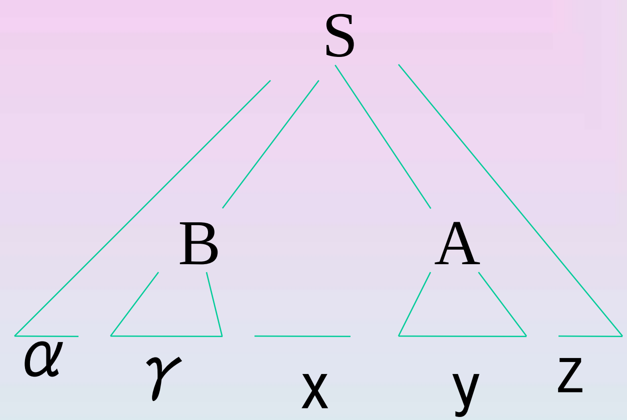


2. **LR-Parsers**
   – covers wide range of grammars.
      - SLR – simple LR parser
      - LR – most general LR parser
      - LALR – intermediate LR parser (lookhead LR parser)

   – SLR, LR and LALR work same, only their parsing tables are different.

Handle will appear on top of the stack





Stack	Input
$\$\alpha\ \beta\ \gamma$	yz$
$\$\alpha\ \beta\,B$	yz$
$\$\alpha\ \beta\,By$	z$

Stack	Input
$\$\alpha\ \gamma$	xyz$
$\$\alpha\,Bxy$	z$

Conflicts during shit reduce parsing

- Two kind of conflicts
  - Shift/reduce conflict
  - Reduce/reduce conflict
- Example:

$$
\begin{aligned}
stmt \longrightarrow\ & \textbf{If}\ expr\ \textbf{then}\ stmt \\
\mid\ & \textbf{If}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\
\mid\ & \textbf{other}
\end{aligned}
$$

Stack	Input
… if expr then stmt	else …$

Reduce/reduce conflict

stmt -> id(parameter_list)
stmt -> expr:=expr
parameter_list->parameter_list, parameter
parameter_list->parameter
parameter->id
expr->id(expr_list)
expr->id
expr_list->expr_list, expr
expr_list->expr

Stack	Input
… id(id	,id) …$

Two data structures are required to implement a shift-reduce parser-

- A **Stack** is required to hold the grammar symbols.
- An **Input buffer** is required to hold the string to be parsed.

Working-

Initially, shift-reduce parser is present in the following configuration where-

- Stack contains only the $ symbol.
- Input buffer contains the input string with $ at its end.

The parser works by-

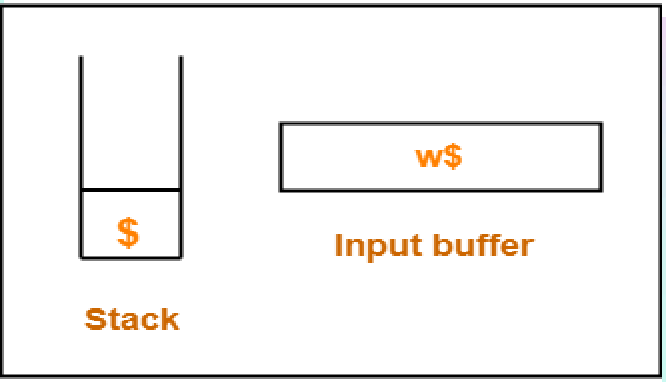
- Moving the input symbols on the top of the stack.
- Until a handle β appears on the top of the stack.

The parser keeps on repeating this cycle until-

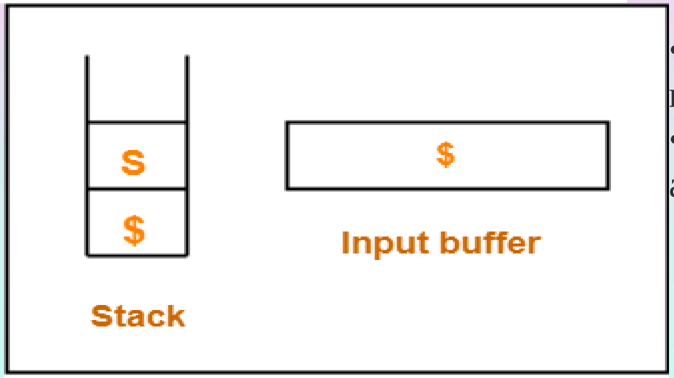
- An error is detected.
- Or stack is left with only the start symbol and the input buffer becomes empty.

After achieving this configuration,

- The parser stops / halts.
- It reports the successful completion of parsing.



Initial Configuration



Final Configuration

Possible Actions-

A shift-reduce parser can possibly make the following four actions-

1. Shift-

In a shift action,

- The next symbol is shifted onto the top of the stack.

2. Reduce-

In a reduce action,

- The handle appearing on the stack top is replaced with the appropriate non-terminal symbol.

3. Accept-

In an accept action,

- The parser reports the successful completion of parsing.

4. Error-

In this state,

- The parser becomes confused and is not able to make any decision.
- It can neither perform shift action nor reduce action nor accept action.

Problem-01:

Consider the following grammar-

E → E – E

E → E x E

E → id

.

Parse the input string

id – id x id

using a shift-reduce parser

Solution-

The priority order is:

id > x > –

Stack	Input Buffer	Parsing Action
$	id – id x id $	Shift
$ id	– id x id $	Reduce E → id
$ E	– id x id $	Shift
$ E –	id x id $	Shift
$ E – id	x id $	Reduce E → id
$ E – E	x id $	Shift
$ E – E x	id $	Shift
$ E – E x id	$	Reduce E → id
$ E – E x E	$	Reduce E → E x E
$ E – E	$	Reduce E → E – E
$ E	$	Accept

- **Problem-02:**
- Consider the following grammar-
  - S → (L) | a
  - L → L , S | S
- Parse the input string
- (a , (a , a))
- using a shift-reduce parser.

Stack	Input Buffer	Parsing Action
$	(a , (a , a)) $	Shift
$ (	a , (a , a)) $	Shift
$ (a	, (a , a)) $	Reduce S → a
$ (S	, (a , a)) $	Reduce L → S
$ (L	, (a , a)) $	Shift
$ (L ,	(a , a)) $	Shift
$ (L , (	a , a)) $	Shift
$ (L , (a	, a)) $	Reduce S → a
$ (L , (S	, a)) $	Reduce L → S
$ (L , (L	, a)) $	Shift
$ (L , (L ,	a)) $	Shift
$ (L , (L , a	)) $	Reduce S → a
$ (L , (L , S)	)) $	Reduce L → L , S
$ (L , (L	)) $	Shift
$ (L , (L)	) $	Reduce S → (L)
$ (L , S	) $	Reduce L → L , S
$ (L	) $	Shift
$ (L)	$	Reduce S → (L)
$ S	$	Accept

- **Problem-03:**
-
- Consider the following grammar-
    - $S \rightarrow T\ L$
  - $T \rightarrow int \mid float$
  - $L \rightarrow L\ ,\ id \mid id$

- Parse the input string
- int id , id ;
- using a shift-reduce parser.

Stack	Input Buffer	Parsing Action
$	int id , id ; $	Shift
$ int	id , id ; $	Reduce T → int
$ T	id , id ; $	Shift
$ T id	, id ; $	Reduce L → id
$ T L	, id ; $	Shift
$ T L ,	id ; $	Shift
$ T L , id	; $	Reduce L → L , id
$ T L	; $	Shift
$ T L ;	$	Reduce S → T L
$ S	$	Accept

Problem-04:

Considering the string

“10201”,

design a shift-reduce parser for the following grammar-

$S \rightarrow 0S0 \mid 1S1 \mid 2$

Stack	Input Buffer	Parsing Action
$	1 0 2 0 1 $	Shift
$ 1	0 2 0 1 $	Shift
$ 1 0	2 0 1 $	Shift
$ 1 0 2	0 1 $	Reduce S → 2
$ 1 0 S	0 1 $	Shift
$ 1 0 S 0	1 $	Reduce S → 0 S 0
$ 1 S	1 $	Shift
$ 1 S 1	$	Reduce S → 1 S 1
$ S	$	Accept

Operator-Precedence Parser

- **Operator grammar**
  - small, but an important class of grammars
  - we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.
- In an *operator grammar*, no production rule can have:
  - ε at the right side
  - two adjacent non-terminals at the right side.

- Ex:

$E \rightarrow AB$	$E \rightarrow EOE$	$E \rightarrow E+E \mid$
$A \rightarrow a$	$E \rightarrow id$	$E*E \mid$
$B \rightarrow b$	$O \rightarrow + \mid * \mid /$	$E/E \mid id$
not operator grammar	not operator grammar	operator grammar

Precedence Relations

- In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

a <· b	b has higher precedence than a
a =· b	b has same precedence as a
a ·> b	b has lower precedence than a

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators. (Unary minus causes a problem).

Using Operator-Precedence Relations

- The intention of the precedence relations is to find the handle of a right-sentential form,

  <· with marking the left end,

  =· appearing in the interior of the handle, and

  ·> marking the right hand.

- In our input string $\$a_1a_2...a_n\$$, we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).

Using Operator -Precedence Relations

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\text{^}E \mid (E) \mid -E \mid id$

The partial operator-precedence table for this grammar

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

- Then the input string id+id*id with the precedence relations inserted will be:

$ <· id ·> + <· id ·> * <· id ·> $

To Find The Handles

1. Scan the string from left end until the first ·> is encountered.
2. Then scan backwards (to the left) over any =· until a <· is encountered.
3. The handle contains everything to left of the first ·> and to the right of the <· is encountered.

$ <· id ·> + <· id ·> * <· id ·> $	E → id	$ id + id * id $
$ <· + <· id ·> * <· id ·> $	E → id	$ E + id * id $
$ <· + <· * <· id ·> $	E → id	$ E + E * id $
$ <· + <· * ·> $	E → E*E	$ E + E * ·E $
$ <· + ·> $	E → E+E	$ E + E $
$ $		$ E $

Operator-Precedence Parsing Algorithm

- The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals

Algorithm:

```
set p to point to the first symbol of w$ ;
repeat forever
  if ( $ is on top of the stack and p points to $ ) then return
  else {
    let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by p;
    if ( a <· b or a =· b ) then {        /* SHIFT */
      push b onto the stack;
      advance p to the next input symbol;
    }
    else if ( a ·> b ) then               /* REDUCE */
      repeat pop stack
      until ( the top of stack terminal is related by <· to the terminal most recently popped );
    else error();
  }
```

Operator-Precedence Parsing Algorithm -- Example

stack	input	action		
$	id+id*id$	$ <· id	shift	
$id	+id*id$	id ·> +	reduce	E → id
$	+id*id$	shift		
$+	id*id$	shift		
$+id	*id$	id ·> *	reduce	E → id
$+	*id$	shift		
$+*	id$	shift		
$+*id	$	id ·> $	reduce	E → id
$+*	$	* ·> $	reduce	E → E*E
$+	$	+ ·> $	reduce	E → E+E
$	$	accept		

How to Create Operator-Precedence Relations

- We use associativity and precedence relations among operators.

1. If operator O_1 has higher precedence than operator O_2,
   ➔ $O_1 \cdot\!> O_2$ and $O_2 <\!\cdot\, O_1$

2. If operator O_1 and operator O_2 have equal precedence,
   they are left-associative ➔ $O_1 \cdot\!> O_2$ and $O_2 \cdot\!> O_1$
   they are right-associative ➔ $O_1 <\!\cdot\, O_2$ and $O_2 <\!\cdot\, O_1$

3. For all operators O,
   $O <\!\cdot\, id$, $id \cdot\!> O$, $O <\!\cdot\, ($, $(\, <\!\cdot\, O$, $O \cdot\!>)$, $) \cdot\!> O$, $O \cdot\!> \$$, and $\$ <\!\cdot\, O$

4. Also, let

$(\doteq)$	$\$ <\!\cdot\, ($	$id \cdot\!>)$	$) \cdot\!> \$$
$(<\!\cdot\, ($	$\$ <\!\cdot\, id$	$id \cdot\!> \$$	$) \cdot\!>)$
$(<\!\cdot\, id$			

Operator-Precedence Relations

	+	-	*	/	^	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
^	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=·	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Handling Unary Minus

- Operator-Precedence parsing cannot handle the unary minus when we also the binary minus in our grammar.
- The best approach to solve this problem, let the lexical analyzer handle this problem.
  - The lexical analyzer will return two different operators for the unary minus and the binary minus.
  - The lexical analyzer will need a lookhead to distinguish the binary minus from the unary minus.
- Then, we make

O $<\cdot$ unary-minus	for any operator
unary-minus $\cdot>$ O	if unary-minus has higher precedence than O
unary-minus $<\cdot$ O	if unary-minus has lower (or equal) precedence than O

Operator-Precedance Grammars

Let G be an ∈-free operator grammar(No ∈-Production).For each terminal symbols *a* and *b*, the following conditions are satisfies.

1. $a \doteq b$, if ∃ a production in RHS of the form $\alpha a \beta b \gamma$, where β is either ∈ or a single non Terminal. <u>*Ex*</u> *S→iCtSeS implies* $i \doteq t$ *and* $t \doteq e$.
2. $a <\cdot b$ if for some non-terminal A ∃ a production in RHS of the form $A \rightarrow \alpha a A \beta$, and $A \Rightarrow^{+} \gamma b \delta$ where γ is either ∈ or a single non-terminal. <u>*Ex*</u> *S→iCtS and* $C \Rightarrow^{+} b$ *implies* $i <\cdot b$.
3. $a \cdot> b$ if for some non-terminal A ∃ a production in RHS of the form $A \rightarrow \alpha A b \beta$, and $A \Rightarrow^{+} \gamma a \delta$ where δ is either ∈ or a single non-terminal. <u>*Ex*</u> *S→iCtS and* $C \Rightarrow^{+} b$ *implies* $b \cdot> t$.

<u>Example</u>:E→E+E | E*E | (E) | id is not a Operator precedence Grammar

By Rule no. 3 we have $+ <\cdot +$ & $+ \cdot> +$. Where as we can modify the Grammar is as follow

E→E+T | T, T→T*F | F, F→(E) | id

Operator Precedence Relations.

To find the Table we have to find the last & first terminal for each non-terminal as follows:

Non terminal	First terminal	Last terminal
E	*, +, (, **id**	*, +,), **id**
T	*, (, **id**	*,), **id**
F	(, **id**	), **id**

By Applying the Rule of Operator Precedence Grammar

	+	*****	**(**	**)**	**id**	**$**
+	·>	<·	<·	·>	<·	·>
*****	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≐	<·	
)	·>	·>		·>		·>
id	·>	·>		·>		·>
$	<·	<·	<·		<·	

Operator Precedence Relations, Continue….

To produce the Table we have to follow the procedure as:

LEADING(A) = $\{ a \mid A \Rightarrow^{+} \gamma a \delta$, where γ is ϵ or a single non-terminal.$\}$

TRAILING(A) = $\{ a \mid A \Rightarrow^{+} \gamma a \delta$, where δ is ϵ or a single non-terminal.$\}$

Precedence Functions

- Compilers using operator precedence parsers do not need to store the table of precedence relations.
- The table can be encoded by two precedence functions f and g that map terminal symbols to integers.
- For symbols a and b.

$f(a) < g(b)$ whenever $a \lessdot b$

$f(a) = g(b)$ whenever $a \doteq b$

$f(a) > g(b)$ whenever $a \gtrdot b$

Disadvantages of Operator Precedence Parsing

- **Disadvantages**:
  - It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.

- **Advantages**:
  - simple
  - powerful enough for expressions in programming languages

Error Recovery in Operator-Precedence Parsing

Error Cases:

1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found (reduction step), but there is no production with this handle as a right side

Error Recovery:

1. Each empty entry is filled with a pointer to an error routine.
2. Decides the popped handle “looks like” which right hand side. And tries to recover from that situation.

LR Parsers

- The most powerful shift-reduce parsing (yet efficient) is:

LR(k) parsing.

left to right scanning

right-most derivation

k lookhead (k is omitted ➔ it is 1)

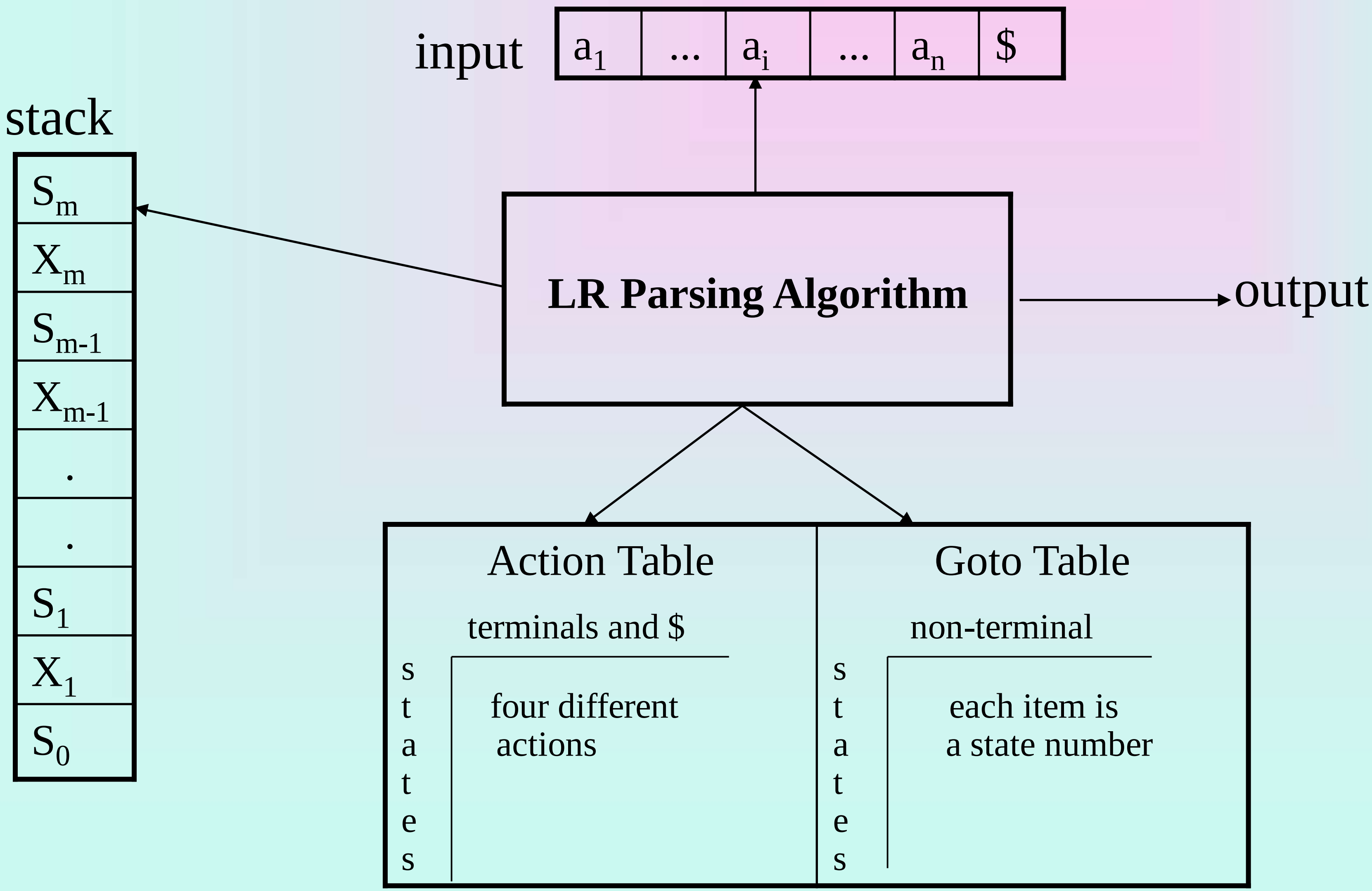
- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

    LL(1)-Grammars $\subset$ LR(1)-Grammars
  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

LR Parsers

- **LR-Parsers**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (look-head LR parser)
  - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

LR Parsing Algorithm



A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$(\underbrace{S_o\ X_1\ S_1 \ldots X_m\ S_m}_{\text{Stack}},\ \underbrace{a_i\ a_{i+1} \ldots a_n\ \$}_{\text{Rest of Input}})$$

- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)

- A configuration of a LR parsing represents the right sentential form:

$$X_1 \ldots X_m\ a_i\ a_{i+1} \ldots a_n\ \$$$

Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack

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

2. **reduce A→β** (or **`rn`** where n is a production number)
   - pop 2|**β**| (=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→β

3. **Accept** – Parsing successfully completed

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

Reduce Action

- pop 2|**β**| (=r) items from the stack; let us assume that **β** = $Y_1 Y_2 \ldots Y_r$
- then push **A** and **s** where **s=goto[s_{m-r},A]**

$$(S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ Y_1\ S_{m-r} \ldots Y_r\ S_m,\ a_i\ a_{i+1} \ldots a_n\ \$\ )$$

$$\rightarrow (S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ A\ s,\ a_i \ldots a_n\ \$\ )$$

- In fact, $Y_1 Y_2 \ldots Y_r$ is a handle.

$$X_1 \ldots X_{m-r}\ A\ a_i \ldots a_n\ \$ \Rightarrow X_1 \ldots X_m\ Y_1 \ldots Y_r\ a_i\ a_{i+1} \ldots a_n\ \$$$

(SLR) Parsing Tables for Expression Grammar

1) $E \rightarrow E+T$
2) $E \rightarrow T$
3) $T \rightarrow T*F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

Action Table | Goto Table

state	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				

Actions of A (S)LR-Parser -- Example

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Constructing SLR Parsing Tables – LR(0) Item

- An **LR(0) item** of a grammar G is a production of G a dot at the some position of the right side.
- Ex: $A \rightarrow aBb$ Possible LR(0) Items: $A \rightarrow \bullet aBb$
  (four different possibility) $A \rightarrow a \bullet Bb$
  $A \rightarrow aB \bullet b$
  $A \rightarrow aBb \bullet$
- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- *Augmented Grammar*:

  G' is G with a new production rule $S' \rightarrow S$ where S' is the new starting symbol.

The Closure Operation

- If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from I by the two rules:

1. Initially, every LR(0) item in I is added to closure(I).
2. If $A \rightarrow \alpha \bullet B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow \bullet \gamma$ will be in the closure(I).
   We will apply this rule until no more new LR(0) items can be added to closure(I).

The Closure Operation -- Example

$E' \to E$

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to id$

$closure(\{E' \to \bullet E\}) =$

$\{ E' \to \bullet E$ ← kernel items

$E \to \bullet E+T$

$E \to \bullet T$

$T \to \bullet T*F$

$T \to \bullet F$

$F \to \bullet (E)$

$F \to \bullet id \}$

Goto Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha \bullet X\beta$ in I
    then every item in **closure({$A \rightarrow \alpha X \bullet \beta$})** will be in goto(I,X).

Example:

I ={ $E' \rightarrow \bullet E$, $E \rightarrow \bullet E+T$, $E \rightarrow \bullet T$,
$T \rightarrow \bullet T*F$, $T \rightarrow \bullet F$,
$F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$ }

goto(I,E) = { $E' \rightarrow E \bullet$, $E \rightarrow E \bullet +T$ }

goto(I,T) = { $E \rightarrow T \bullet$, $T \rightarrow T \bullet *F$ }

goto(I,F) = { $T \rightarrow F \bullet$ }

goto(I,() = { $F \rightarrow (\bullet E)$, $E \rightarrow \bullet E+T$, $E \rightarrow \bullet T$, $T \rightarrow \bullet T*F$, $T \rightarrow \bullet F$,
$F \rightarrow \bullet (E)$, $F \rightarrow \bullet id$ }

goto(I,id) = { $F \rightarrow id \bullet$ }

Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.

- ***Algorithm***:

  C is { closure({S'→•S}) }

  repeat the followings until no more set of LR(0) items can be added to ***C***.

      for each I in ***C*** and each grammar symbol X

          if goto(I,X) is not empty and not in ***C***

              add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

The Canonical LR(0) Collection -- Example

I_0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_1: E' → E.

I_2: E → T.
T → T.*F

I_3: T → F.

I_4: F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_5: F → id.

I_6: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I_7: T → T*.F
F → .(E)
F → .id

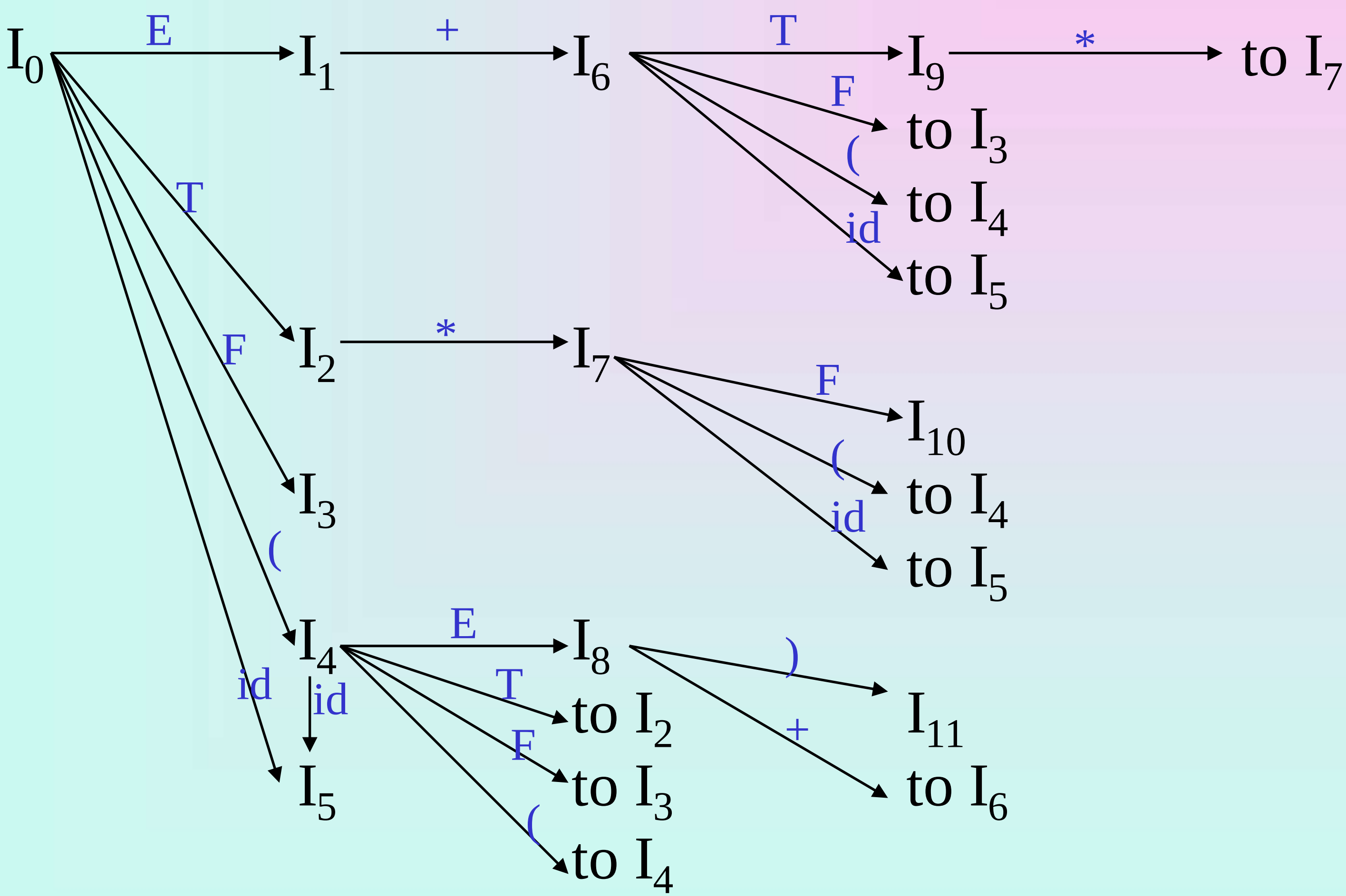
I_8: F → (E.)
E → E.+T

I_9: E → E+T.
T → T.*F

I_{10}: T → T*F.

I_{11}: F → (E).

Transition Diagram (DFA) of Goto Function



LR Parsing

- The most prevalent type of bottom-up parsers
- LR(k), mostly interested on parsers with k<=1
- Why LR parsers?
  - Table driven
  - Can be constructed to recognize all programming language constructs
  - Most general non-backtracking shift-reduce parsing method
  - Can detect a syntactic error as soon as it is possible to do so
  - Class of grammars for which we can construct LR parsers are superset of those which we can construct LL parsers

States of an LR parser

- States represent set of items
- An LR(0) item of G is a production of G with the dot at some position of the body:
  - For A->XYZ we have following items
    - A->.XYZ
    - A->X.YZ
    - A->XY.Z
    - A->XYZ.
  - In a state having A->.XYZ we hope to see a string derivable from XYZ next on the input.
  - What about A->X.YZ?

Constructing canonical LR(0) item sets

- Augmented grammar:
  - G with addition of a production: S'->S
- Closure of item sets:
  - If I is a set of items, closure(I) is a set of items constructed from I by the following rules:
    - Add every item in I to closure(I)
    - If A->α.Bβ is in closure(I) and B->γ is a production then add the item B->.γ to clsoure(I).
- Example:

E'->E
E -> E + T | T
T -> T * F | F
F -> (E) | **id**

I0=closure({[E'->.E]}
E'->.E
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.id

Constructing canonical LR(0) item sets (cont.)

- Goto (I,X) where I is an item set and X is a grammar symbol is closure of set of all items [A-> αX. β] where [A-> α.X β] is in I
- Example

I0=closure({[E’->.E]}
E’->.E
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.id

E

I1
E’->E.
E->E.+T

T

I2
E’->T.
T->T.*F

(

I4
F->(.E)
E->.E+T
E->.T
T->.T*F
T->.F
F->.(E)
F->.id

Closure algorithm

```
SetOfItems CLOSURE(I) {
    J=I;
    repeat
        for (each item A-> α.Bβ in J)
            for (each prodcution B->γ of G)
                if (B->.γ is not in J)
                    add B->.γ to J;
    until no more items are added to J on one round;
    return J;
```

GOTO algorithm

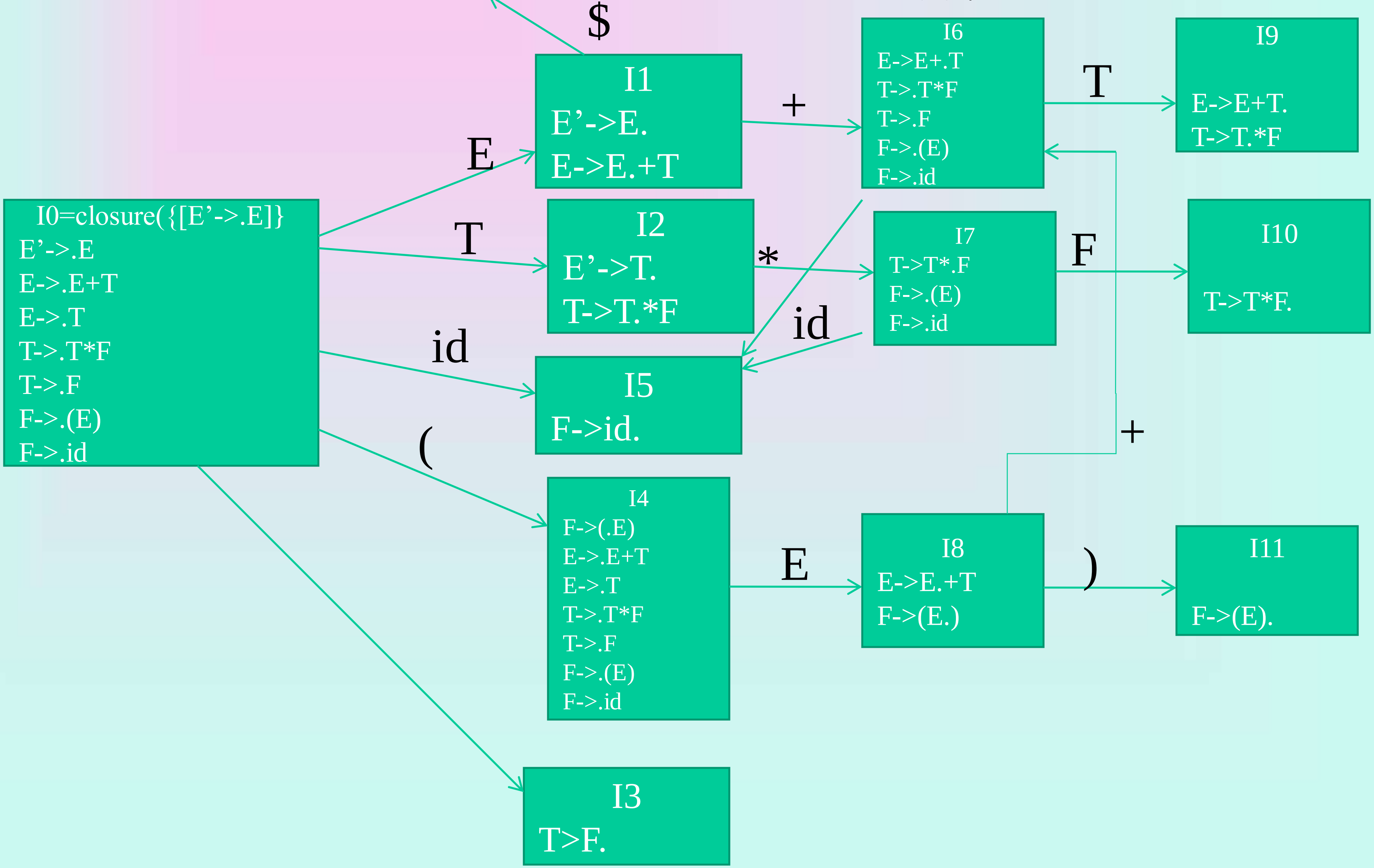
```
SetOfItems  GOTO(I,X) {
  J=empty;
   if (A-> α.X β is in I)
          add CLOSURE(A-> αX. β ) to J;
   return J;
}
```

Canonical LR(0) items

```
Void items(G') {
    C= CLOSURE({[S'->.S]});
    repeat
        for (each set of items I in C)
            for (each grammar symbol X)
                if (GOTO(I,X) is not empty and not in C)
                    add GOTO(I,X) to C;
    until no new set of items are added to C on a round;
}
```

Example

E'->E
E -> E + T | T
T -> T * F | F
F -> (E) | **id**

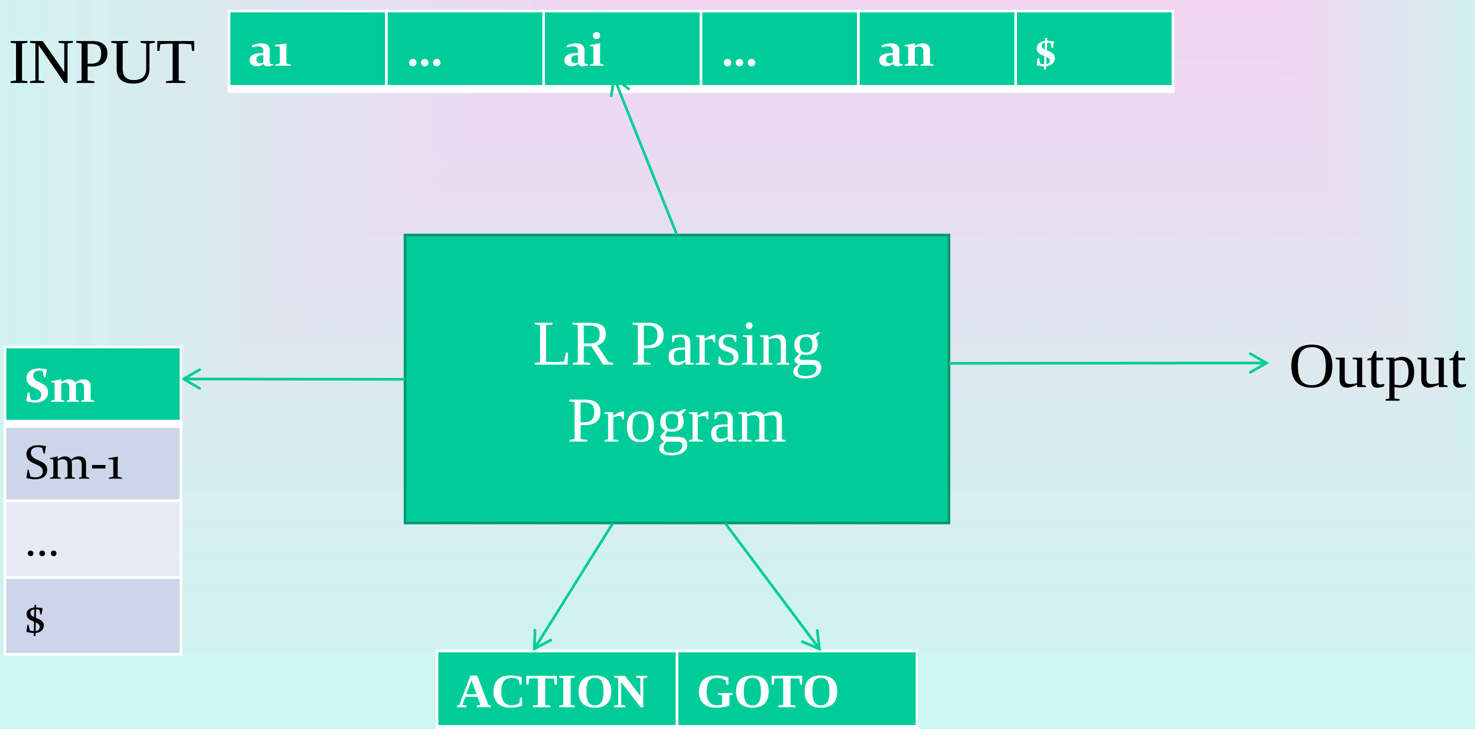


Use of LR(0) automaton

- Example: id*id

Line	Stack	Symbols	Input	Action
(1)	0	$	id*id$	Shift to 5
(2)	05	$id	*id$	Reduce by F->id
(3)	03	$F	*id$	Reduce by T->F
(4)	02	$T	*id$	Shift to 7
(5)	027	$T*	id$	Shift to 5
(6)	0275	$T*id	$	Reduce by F->id
(7)	02710	$T*F	$	Reduce by T->T*F
(8)	02	$T	$	Reduce by E->T
(9)	01	$E	$	accept

LR-Parsing model



LR parsing algorithm

```
let a be the first symbol of w$;
while(1) { /*repeat forever */
    let s be the state on top of the stack;
    if (ACTION[s,a] = shift t) {
            push t onto the stack;
            let a be the next input symbol;
    } else if (ACTION[s,a] = reduce A->β) {
            pop |β| symbols of the stack;
            let state t now be on top of the stack;
            push GOTO[t,A] onto the stack;
            output the production A->β;
    } else if (ACTION[s,a]=accept) break; /* parsing is done */
    else call error-recovery routine;
}
```

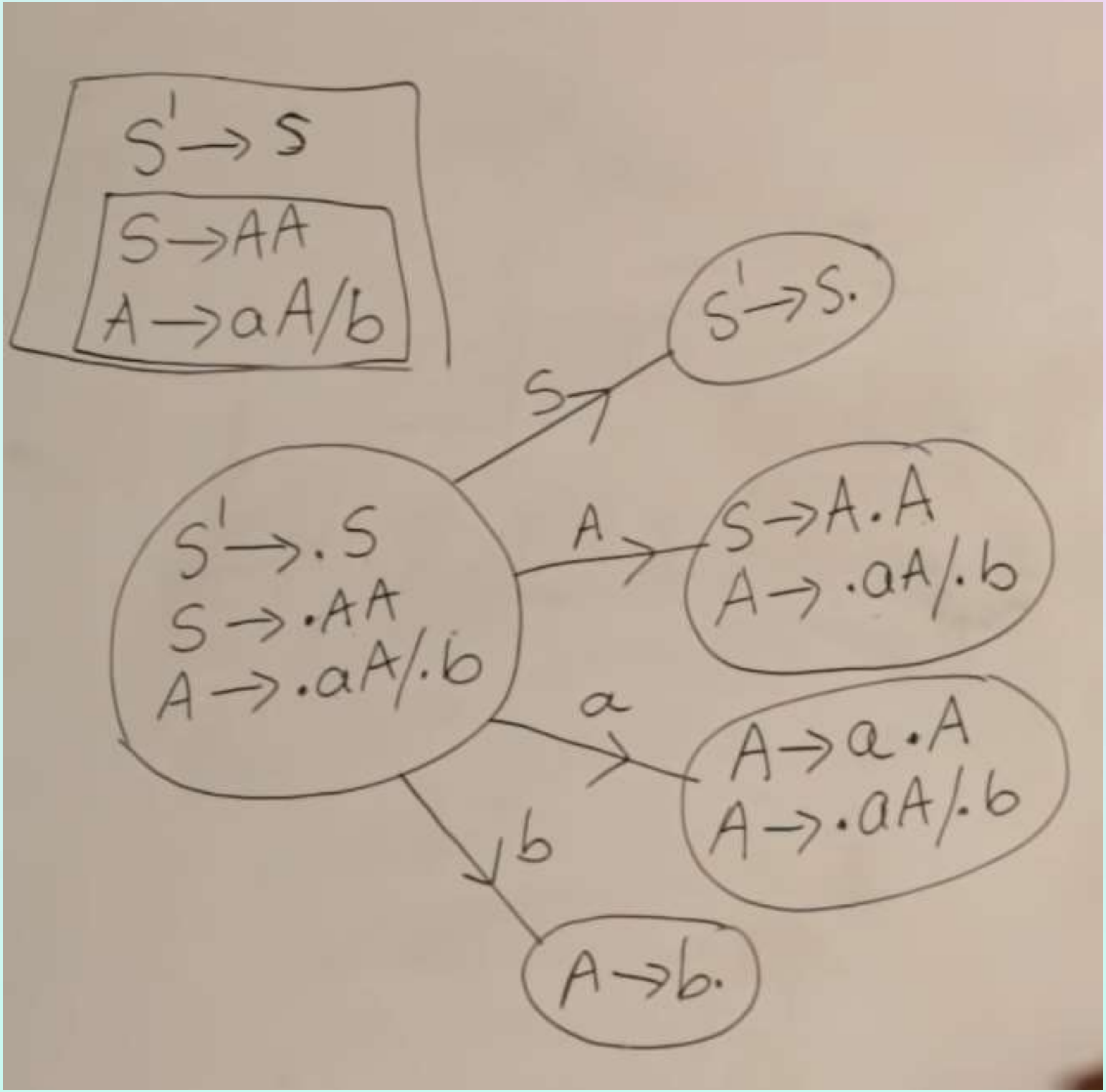
Example

(0) E'->E
(1) E -> E + T
(2) E-> T
(3) T -> T * F
(4) T-> F
(5) F -> (E)
(6) F->**id**

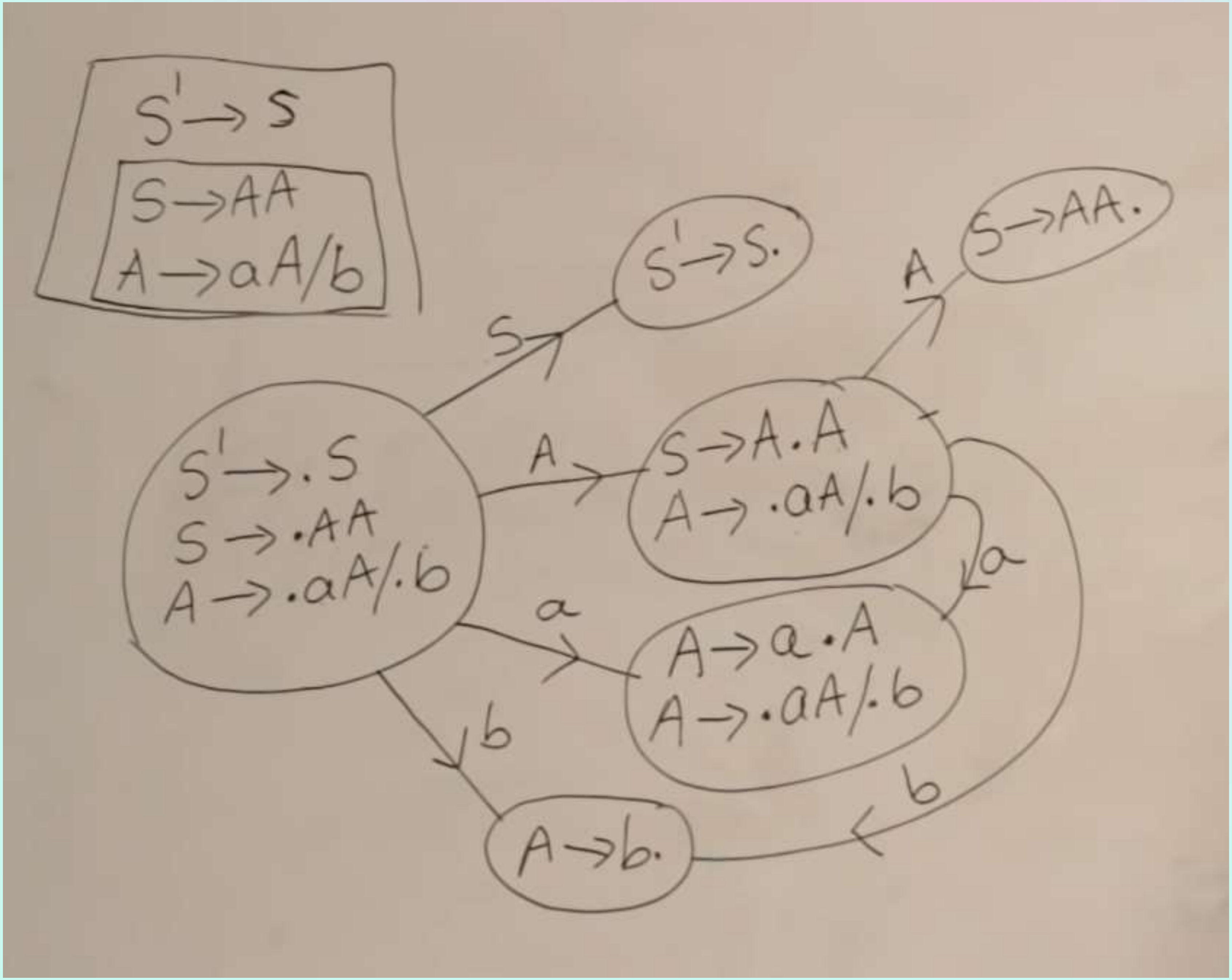
id*id+id?

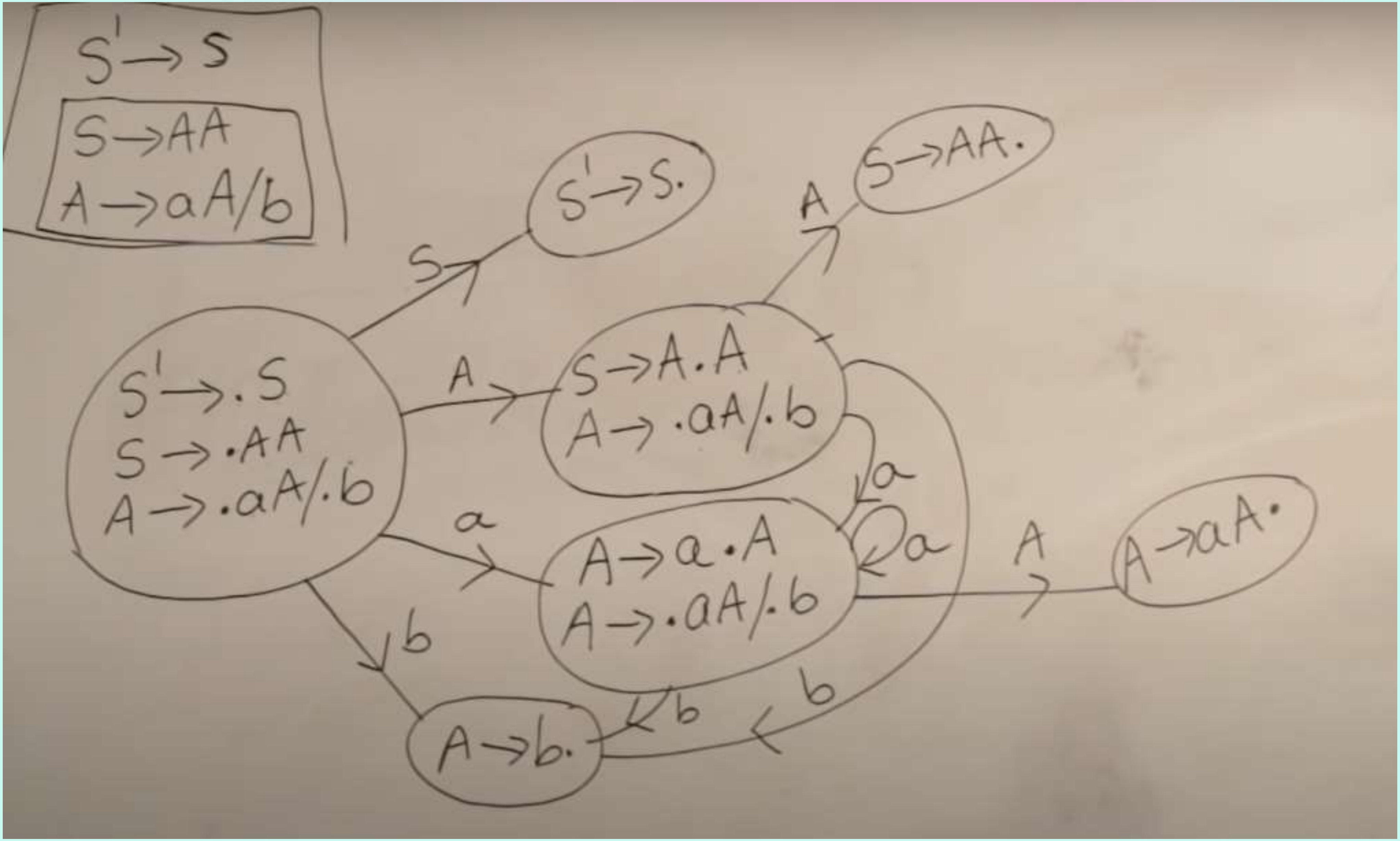
STATE	ACTON						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		R2	S7		R2	R2			
3		R 4	R7		R4	R4			
4	S5			S4			8	2	3
5		R 6	R 6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

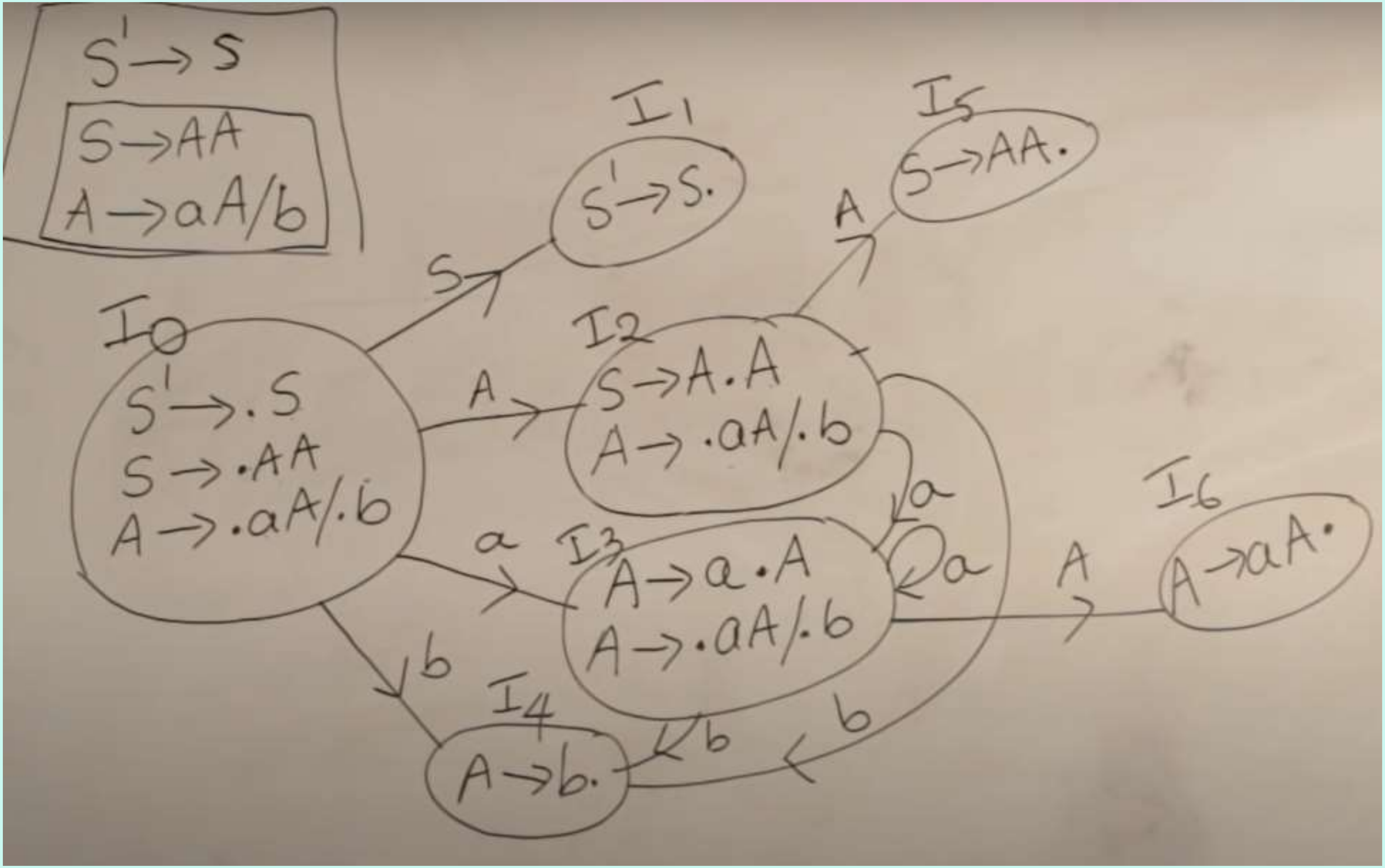
Line	Stack	Symbols	Input	Action
(1)	0		id*id+id$	Shift to 5
(2)	05	id	*id+id$	Reduce by F->id
(3)	03	F	*id+id$	Reduce by T->F
(4)	02	T	*id+id$	Shift to 7
(5)	027	T*	id+id$	Shift to 5
(6)	0275	T*id	+id$	Reduce by F->id
(7)	02710	T*F	+id$	Reduce by T->T*F
(8)	02	T	+id$	Reduce by E->T
(9)	01	E	+id$	Shift
(10)	016	E+	id$	Shift
(11)	0165	E+id	$	Reduce by F->id
(12)	0163	E+F	$	Reduce by T->F
(13)	0169	E+T`	$	Reduce by E->E+T
(14)	01	E	$	accept



S'→S
S→AA
A→aA/b
S'→S.
S
S'→.S
S→.AA
A→.aA/.b
A
S→A.A
A→.aA/.b
a
A→a.A
A→.aA/.b
b
A→b.


S' → S
S → AA
A → aA/b
S' → S.
S → AA.
S' → .S
S → .AA
A → .aA/.b
S → A.A
A → .aA/.b
A → a.A
A → .aA/.b
A → b.
S
A
A
a
a
b
b

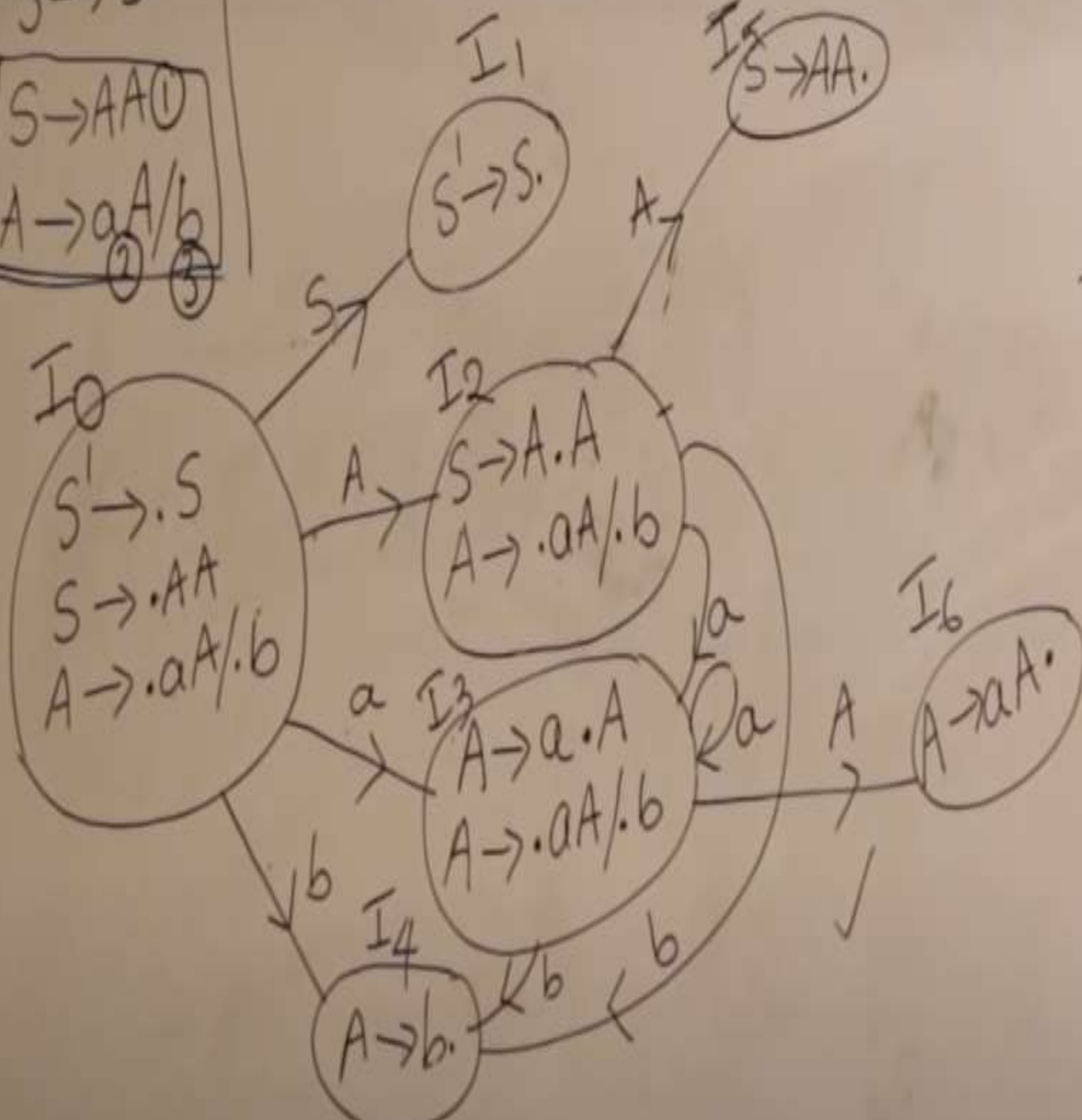

S' → S
S → AA
A → aA/b
S' → S.
S → AA.
S' → .S
S → .AA
A → .aA/.b
S → A.A
A → .aA/.b
A → a.A
A → .aA/.b
A → aA.
A → b.
S
A
a
b


S' → S
S → AA
A → aA/b
I1
S' → S.
I5
S → AA.
I0
S' → .S
S → .AA
A → .aA/.b
S
A
I2
S → A.A
A → .aA/.b
a
I3
A → a.A
A → .aA/.b
A
I6
A → aA.
b
I4
A → b.

$S' \to S$

$S \to AA$ ①

$A \to aA \,/\, b$ ② ③



	action			Goto	
	a	b	\$	A	S
0	s_3	s_4		2	1
1			accept		
2	s_3	s_4		5	
3	s_3	s_4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

Constructing SLR 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, $A \rightarrow \alpha.a\beta$ 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 $A \neq S'$.
   - 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 $S' \rightarrow .S$

Parsing Tables of Expression Grammar

Action Table | Goto Table

state	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				

SLR(1) Grammar

- An LR parser using SLR(1) parsing tables for a grammar G is called as the SLR(1) parser for G.
- If a grammar G has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).
- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

shift/reduce and reduce/reduce conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.

- If a state does not know whether it will make a reduction operation using the production rule `i` or `j` for a terminal, we say that there is a **reduce/reduce conflict**.

- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

Conflict Example

$S \rightarrow L{=}R$
$S \rightarrow R$
$L \rightarrow {*}R$
$L \rightarrow id$
$R \rightarrow L$

I_0: $S' \rightarrow .S$
$S \rightarrow .L{=}R$
$S \rightarrow .R$
$L \rightarrow .{*}R$
$L \rightarrow .id$
$R \rightarrow .L$

I_1: $S' \rightarrow S.$

I_2: $S \rightarrow L.{=}R$
$R \rightarrow L.$

I_3: $S \rightarrow R.$

I_4: $L \rightarrow {*}.R$
$R \rightarrow .L$
$L \rightarrow .{*}R$
$L \rightarrow .id$

I_5: $L \rightarrow id.$

I_6: $S \rightarrow L{=}.R$
$R \rightarrow .L$
$L \rightarrow .{*}R$
$L \rightarrow .id$

I_7: $L \rightarrow {*}R.$

I_8: $R \rightarrow L.$

I_9: $S \rightarrow L{=}R.$

Problem

FOLLOW(R)={=,$}

= → shift 6
  → reduce by $R \rightarrow L$

shift/reduce conflict

Conflict Example2

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

I_0: $S' \rightarrow .S$

$S \rightarrow .AaAb$

$S \rightarrow .BbBa$

$A \rightarrow .$

$B \rightarrow .$

Problem

FOLLOW(A)={a,b}

FOLLOW(B)={a,b}

a → reduce by $A \rightarrow \varepsilon$

a → reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

b → reduce by $A \rightarrow \varepsilon$

b → reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

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 the $A \rightarrow \alpha .$ 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.

$S \rightarrow AaAb$	$S \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab$	$S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba$
$S \rightarrow BbBa$		
$A \rightarrow \varepsilon$	$Aab \Rightarrow \varepsilon\, ab$	$Bba \Rightarrow \varepsilon\, ba$
$B \rightarrow \varepsilon$	$AaAb \Rightarrow Aa\, \varepsilon\, b$	$BbBa \Rightarrow Bb\, \varepsilon\, a$

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(1) item is:

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

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

LR(1) Item (cont.)

- When β (in the LR(1) item $A \rightarrow \alpha \bullet \beta, a$) is not empty, the look-head does not have any affect.
- When β is empty ($A \rightarrow \alpha \bullet, a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is **a** (not for any terminal in FOLLOW(A)).
- A state will contain $A \rightarrow \alpha \bullet, a_1$ where $\{a_1,...,a_n\} \subseteq$ FOLLOW(A)

$$\begin{array}{l} A \rightarrow \alpha \bullet, a_1 \\ \quad ... \\ A \rightarrow \alpha \bullet, a_n \end{array}$$

Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, 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) .

goto operation

- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta,a$ in I
    then every item in **closure({$A \rightarrow \alpha X.\beta,a$})** will be in goto(I,X).

Construction of The Canonical LR(1) Collection

- ***Algorithm***:

  C is { closure({S'→.S,$}) }

  repeat the followings until no more set of LR(1) items can be added to ***C***.

      for each I in ***C*** and each grammar symbol X

          if goto(I,X) is not empty and not in ***C***

              add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

A Short Notation for The Sets of LR(1) Items

- A set of LR(1) items containing the following items

$$A \rightarrow \alpha \bullet \beta, a_1$$

$$\ldots$$

$$A \rightarrow \alpha \bullet \beta, a_n$$

can be written as

$$A \rightarrow \alpha \bullet \beta, a_1/a_2/\ldots/a_n$$

Canonical LR(1) Collection -- Example

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

$$\begin{array}{lll}
I_0: S' \rightarrow .S\ ,\$ & \xrightarrow{S} & I_1: S' \rightarrow S.\ ,\$ \\
\quad S \rightarrow .AaAb\ ,\$ & \xrightarrow{A} & I_2: S \rightarrow A.aAb\ ,\$ \xrightarrow{a} \text{to } I_4 \\
\quad S \rightarrow .BbBa\ ,\$ & \xrightarrow{B} & I_3: S \rightarrow B.bBa\ ,\$ \xrightarrow{b} \text{to } I_5 \\
\quad A \rightarrow .\ ,a & & \\
\quad B \rightarrow .\ ,b & &
\end{array}$$

$$\begin{array}{l}
I_4: S \rightarrow Aa.Ab\ ,\$ \xrightarrow{A} I_6: S \rightarrow AaA.b\ ,\$ \xrightarrow{a} I_8: S \rightarrow AaAb.\ ,\$ \\
\quad A \rightarrow .\ ,b
\end{array}$$

$$\begin{array}{l}
I_5: S \rightarrow Bb.Ba\ ,\$ \xrightarrow{B} I_7: S \rightarrow BbB.a\ ,\$ \xrightarrow{b} I_9: S \rightarrow BbBa.\ ,\$ \\
\quad B \rightarrow .\ ,a
\end{array}$$

Canonical LR(1) Collection – Example2

$S' \rightarrow S$
1) $S \rightarrow L=R$
2) $S \rightarrow R$
3) $L \rightarrow *R$
4) $L \rightarrow id$
5) $R \rightarrow L$

I_0: $S' \rightarrow .S,\$$
$S \rightarrow .L=R,\$$
$S \rightarrow .R,\$$
$L \rightarrow .*R,\$/=$
$L \rightarrow .id,\$/=$
$R \rightarrow .L,\$$

$\xrightarrow{S}$ I_1: $S' \rightarrow S.,\$$

$\xrightarrow{L}$ I_2: $S \rightarrow L.=R,\$$ $\longrightarrow$ to I_6
$R \rightarrow L.,\$$

$\xrightarrow{R}$ I_3: $S \rightarrow R.,\$$

$\xrightarrow{*}$ I_4: $L \rightarrow *.R,\$/=$
$R \rightarrow .L,\$/=$
$L \rightarrow .*R,\$/=$
$L \rightarrow .id,\$/=$

$\xrightarrow{R}$ to I_7
$\xrightarrow{L}$ to I_8
$\xrightarrow{*}$ to I_4
$\xrightarrow{id}$ to I_5

$\xrightarrow{id}$ I_5: $L \rightarrow id.,\$/=$

I_6: $S \rightarrow L=.R,\$$
$R \rightarrow .L,\$$
$L \rightarrow .*R,\$$
$L \rightarrow .id,\$$

$\xrightarrow{R}$ to I_9
$\xrightarrow{L}$ to I_{10}
$\xrightarrow{*}$ to I_{11}
$\xrightarrow{id}$ to I_{12}

I_7: $L \rightarrow *R.,\$/=$

I_8: $R \rightarrow L.,\$/=$

I_9: $S \rightarrow L=R.,\$$

I_{10}: $R \rightarrow L.,\$$

I_{11}: $L \rightarrow *.R,\$$
$R \rightarrow .L,\$$
$L \rightarrow .*R,\$$
$L \rightarrow .id,\$$

$\xrightarrow{R}$ to I_{13}
$\xrightarrow{L}$ to I_{10}
$\xrightarrow{*}$ to I_{11}
$\xrightarrow{id}$ to I_{12}

I_{12}: $L \rightarrow id.,\$$

I_{13}: $L \rightarrow *R.,\$$

I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

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*** $\boldsymbol{A\rightarrow\alpha}$ 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) Parsing Tables – (for Example2)

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			

no shift/reduce or
no reduce/reduce conflict

$\Downarrow$

so, it is a LR(1) grammar

LALR Parsing Tables

- **LALR** stands for **LookAhead LR.**

- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- ***yacc*** creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

Creating LALR Parsing Tables

Canonical LR(1) Parser ➔ LALR Parser

shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrink process does not produce a **shift/reduce** conflict.

The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L \bullet = R, \$$ ➔ $S \rightarrow L \bullet = R$ ← Core

$R \rightarrow L \bullet , \$$ $R \rightarrow L \bullet$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

$I_1: L \rightarrow id \bullet , =$ A new state: $I_{12}: L \rightarrow id \bullet , =$

➔ $L \rightarrow id \bullet , \$$

$I_2: L \rightarrow id \bullet , \$$ have same core, merge them

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

Creation of LALR Parsing Tables

- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.

  $C=\{I_0,...,I_n\} \Rightarrow C'=\{J_1,...,J_m\}$ where $m \leq n$

- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  - Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores

    $\Rightarrow$ cores of $goto(I_1,X),...,goto(I_2,X)$ must be same.
  - So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.

- If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.
- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

$$A \to \alpha \bullet ,a \qquad \text{and} \qquad B \to \beta \bullet a\gamma ,b$$

- This means that a state of the canonical LR(1) parser must have:

$$A \to \alpha \bullet ,a \qquad \text{and} \qquad B \to \beta \bullet a\gamma ,c$$

But, this state has also a shift/reduce conflict. i.e. The original canonical LR(1) parser has a conflict.

(Reason for this, the shift operation does not depend on lookaheads)

Reduce/Reduce Conflict

- But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

$$I_1 : A \to \alpha \bullet ,a \qquad\qquad I_2 : A \to \alpha \bullet ,b$$

$$B \to \beta \bullet ,b \qquad\qquad B \to \beta \bullet ,c$$

$$\Downarrow$$

$$I_{12} : A \to \alpha \bullet ,a/b$$

$$B \to \beta \bullet ,b/c$$

➔ reduce/reduce conflict

Canonical LALR(1) Collection – Example2

$S' \to S$
1) $S \to L=R$
2) $S \to R$
3) $L \to *R$
4) $L \to id$
5) $R \to L$

$I_0: S' \to \bullet S, \$$
$S \to \bullet L=R, \$$
$S \to \bullet R, \$$
$L \to \bullet *R, \$/=$
$L \to \bullet id, \$/=$
$R \to \bullet L, \$$

$I_0 \xrightarrow{S} I_1$, $I_0 \xrightarrow{L} I_2$, $I_0 \xrightarrow{R} I_3$, $I_0 \xrightarrow{*} I_{411}$, $I_0 \xrightarrow{id} I_{512}$

$I_1: S' \to S \bullet, \$$

$I_2: S \to L \bullet =R, \$$ $\longrightarrow$ to I_6
$R \to L \bullet, \$$

$I_3: S \to R \bullet, \$$

$I_{411}: L \to * \bullet R, \$/=$
$R \to \bullet L, \$/=$
$L \to \bullet *R, \$/=$
$L \to \bullet id, \$/=$

$\xrightarrow{R}$ to I_{713}
$\xrightarrow{L}$ to I_{810}
$\xrightarrow{*}$ to I_{411}
$\xrightarrow{id}$ to I_{512}

$I_{512}: L \to id \bullet, \$/=$

$I_6: S \to L= \bullet R, \$$
$R \to \bullet L, \$$
$L \to \bullet *R, \$$
$L \to \bullet id, \$$

$\xrightarrow{R}$ to I_9
$\xrightarrow{L}$ to I_{810}
$\xrightarrow{*}$ to I_{411}
$\xrightarrow{id}$ to I_{512}

$I_9: S \to L=R \bullet, \$$

$I_{713}: L \to *R \bullet, \$/=$

$I_{810}: R \to L \bullet, \$/=$

Same Cores
I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

LALR(1) Parsing Tables – (for Example2)

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			

no shift/reduce or
no reduce/reduce conflict

$\Downarrow$

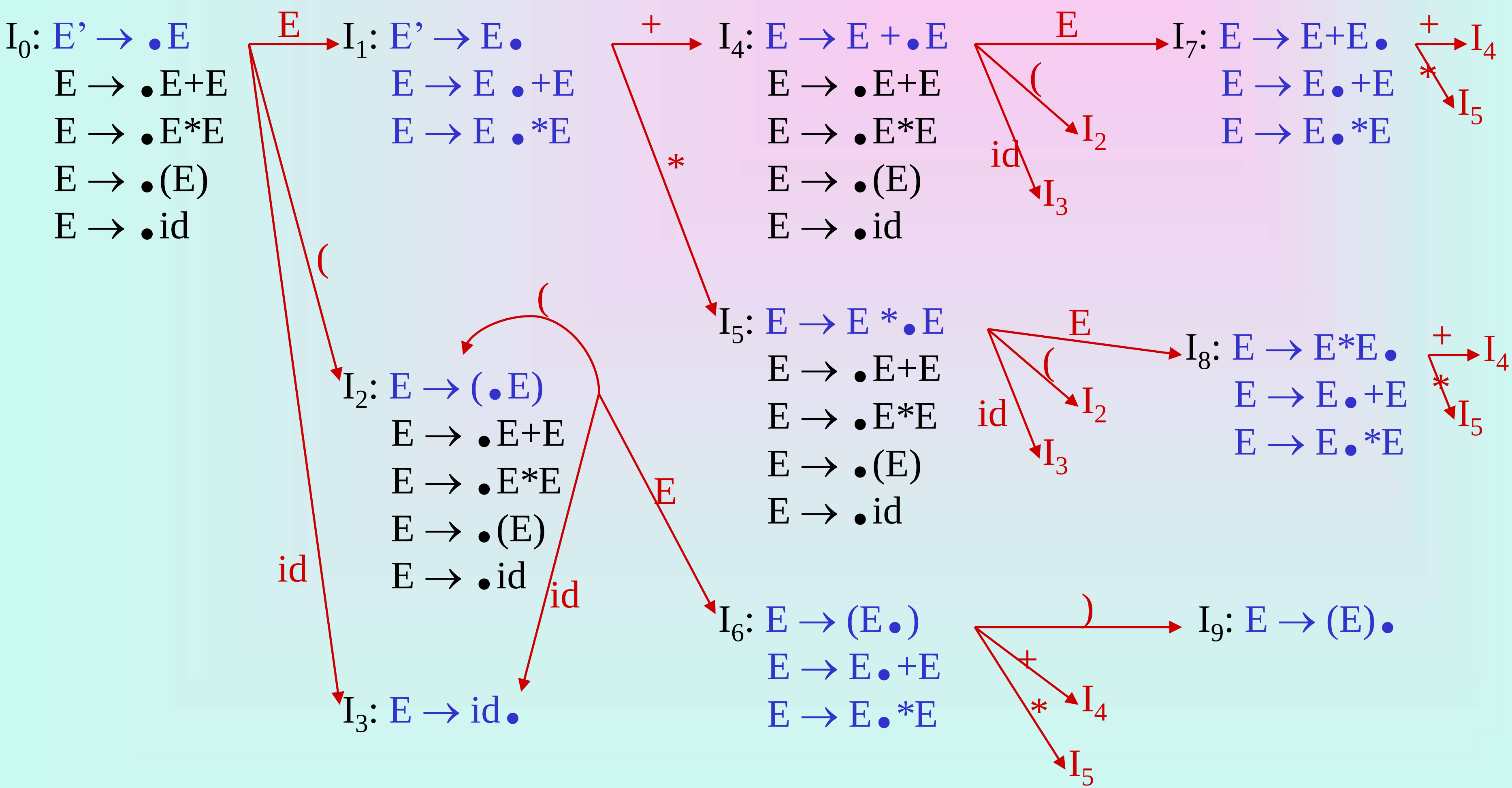
so, it is a LALR(1) grammar

Using Ambiguous Grammars

- All grammars used in the construction of LR-parsing tables must be un-ambiguous.
- Can we create LR-parsing tables for ambiguous grammars ?
  - Yes, but they will have conflicts.
  - We can resolve these conflicts in favor of one of them to disambiguate the grammar.
  - At the end, we will have again an unambiguous grammar.
- Why we want to use an ambiguous grammar?
  - Some of the ambiguous grammars are **much natural**, and a corresponding unambiguous grammar can be very complex.
  - Usage of an ambiguous grammar may **eliminate unnecessary reductions**.
- Ex.

$$E \rightarrow E+E \mid E*E \mid (E) \mid id \quad \Rightarrow \quad \begin{array}{l} E \rightarrow E+T \mid T \\ T \rightarrow T*F \mid F \\ F \rightarrow (E) \mid id \end{array}$$

Sets of LR(0) Items for Ambiguous Grammar



SLR-Parsing Tables for Ambiguous Grammar

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

State I_7 has shift/reduce conflicts for symbols + and *.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{+} I_4 \xrightarrow{E} I_7$$

when current token is +
- shift ➔ + is right-associative
- reduce ➔ + is left-associative

when current token is *
- shift ➔ * has higher precedence than +
- reduce ➔ + has higher precedence than *

SLR-Parsing Tables for Ambiguous Grammar

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

State I_8 has shift/reduce conflicts for symbols + and *.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{*} I_5 \xrightarrow{E} I_7$$

when current token is *

shift ➔ * is right-associative

reduce ➔ * is left-associative

when current token is +

shift ➔ + has higher precedence than *

reduce ➔ * has higher precedence than +

SLR-Parsing Tables for Ambiguous Grammar

	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		

Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
- Errors are never detected by consulting the goto table.
- An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
- A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
- The SLR and LALR parsers may make several reductions before announcing an error.
- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.

Panic Mode Error Recovery in LR Parsing

- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state s).
- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow A.
  - The symbol a is simply in FOLLOW(A), but this may not work for all situations.
- The parser stacks the nonterminal **A** and the state **goto[s,A]**, and it resumes the normal parsing.
- This nonterminal A is normally is a basic programming block (there can be more than one choice for A).
  - stmt, expr, block, ...

Phrase-Level Error Recovery in LR Parsing

- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.
- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
  - missing operand
  - unbalanced right parenthesis