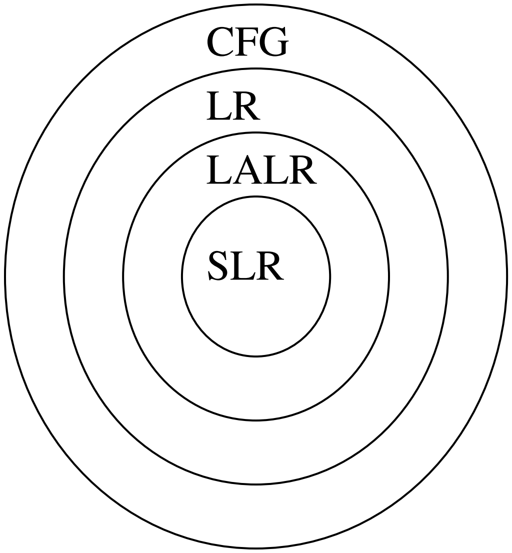
# Parsing

Part VI

# 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.

# LR Parsers

LR parsing is attractive because:

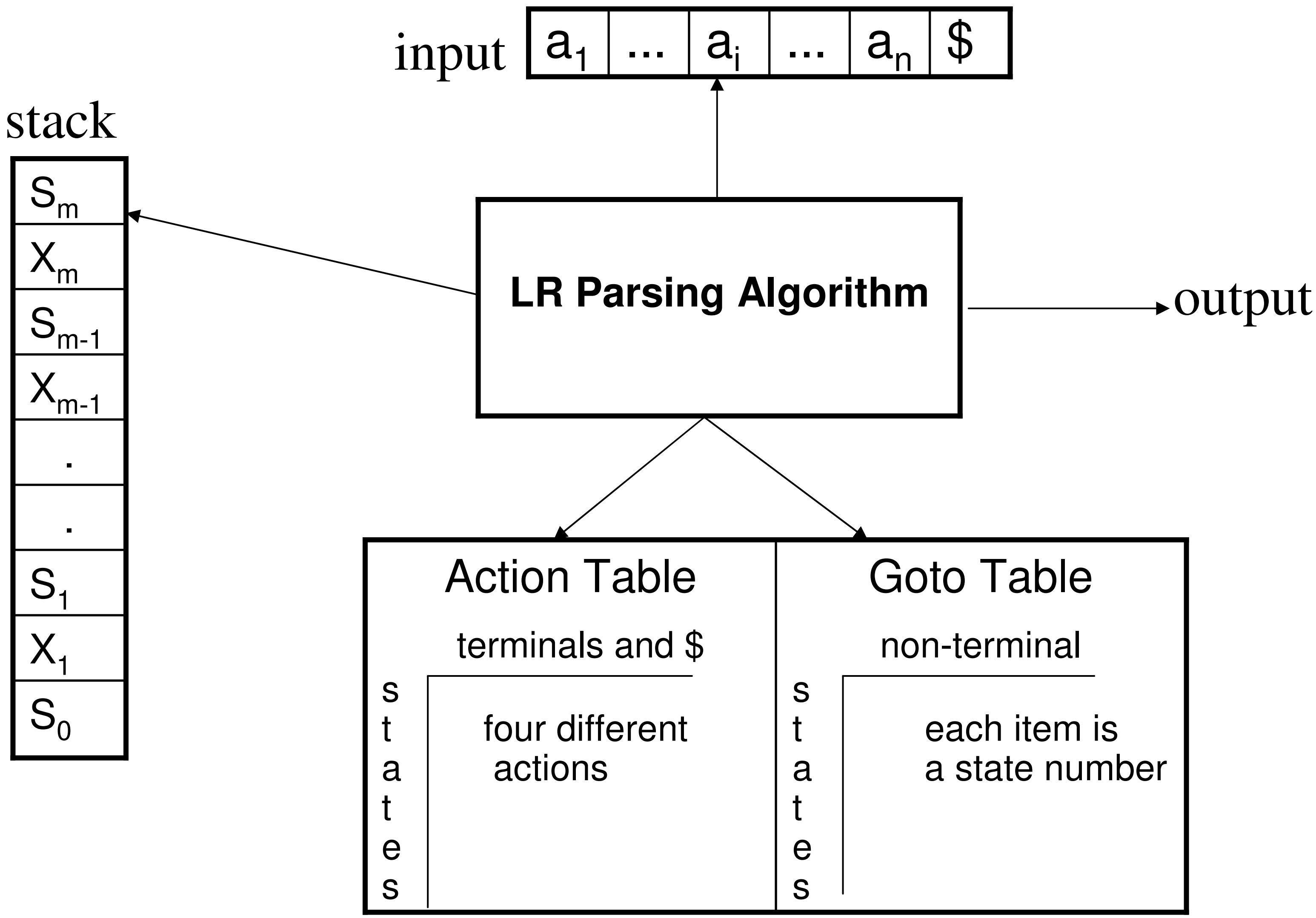
- LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

  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 can be constructed to recognize virtually all programming language constructs for which CFG grammars can be written

Drawback of LR method:

- Too much work to construct LR parser by hand
  - Fortunately tools (LR parsers generators) are available

# 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 $\beta = Y_1 Y_2 \ldots Y_r$
- 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.

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

# LR Parser Stack(s)

The knowledge of what we've parsed so far is in the stack.
Some knowledge is buried in the stack.
We need a "summary" of what we've learned so far.

**LR Parsing uses a second stack for this information.**

**Stack 1:** Stack of grammar symbols (terminals and nonterminals)
**Stack 2:** Stack of "states".

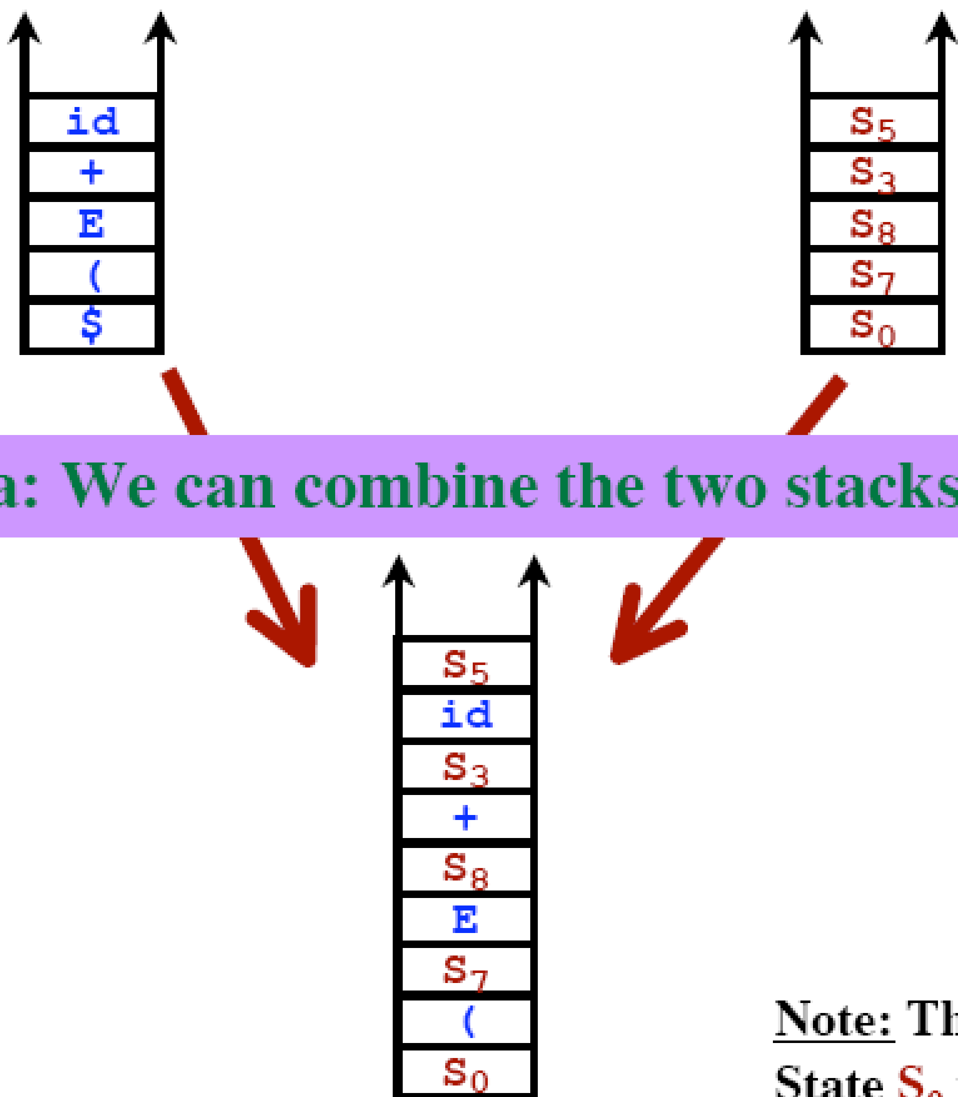
States = $\{ S_0, S_1, S_2, S_3, \ldots, S_N \}$
Implementation: Just use integers (0, 1, 2, 3, ...)
$\Rightarrow$ Just use a stack of integers

***When deciding on an action...***

- Consult the Parsing Tables (ACTION, and GOTO)
- Consult the top of the stack of states

# LR Parser Stack(s)

Stack of Grammar Symbols: Stack of States:




**Idea: We can combine the two stacks into one!**

**Note:** The $ will not be needed.
State $S_0$ will signal the stack bottom.

# (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. $F \rightarrow \underline{id}$

***Key to Notation***

**S4**=“Shift input symbol and push state 4”
**R5**= “Reduce by rule 5”
**Acc**=Accept
**(blank)**=Syntax Error

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 | | | | |

# Example LR Parse: (id+id)*id

| STACK | INPUT | ACTION |
|---|---|---|
| 0 | (id+id)*id$ | |

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → ( E )
6. F → id

# Example LR Parse: (id+id)*id

| STACK | INPUT | ACTION |
|---|---|---|
| 0 | (id+id)*id$ | |
| 0(4 | id+id)*id$ | Shift 4 |

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → ( E )
6. F → id

# Example LR Parse: (id+id)*id

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → ( E )
6. F → id

| STACK | INPUT | ACTION |
|---|---|---|
| 0 | (id+id)*id$ | |
| 0(4 | id+id)*id$ | Shift 4 |
| 0(4id5 | +id)*id$ | Shift 5 |
| 0(4F3 | +id)*id$ | Reduce by F → id |
| 0(4T2 | +id)*id$ | Reduce by T → F |
| 0(4E8 | +id)*id$ | Reduce by E → T |
| 0(4E8+6 | )*id$ | Shift 6 |
| 0(4E8+6id5 | )*id$ | Shift 5 |
| 0(4E8+6F3 | )*id$ | Reduce by F → id |
| 0(4E8+6T9 | )*id$ | Reduce by T → F |
| 0(4E8 | )*id$ | Reduce by E → E + T |
| 0(4E4)11 | *id$ | Shift |
| 0F3 | *id$ | Reduce by F → ( E ) |
| 0T2 | *id$ | Reduce by T → F |
| 0T2*7 | id$ | Shift 7 |
| 0T2*7id5 | $ | Shift 5 |
| 0T2*7F10 | $ | Reduce by F → id |
| 0T2 | $ | Reduce by T → T * F |
| 0E1 | $ | Reduce by E → T |
| | | Accept |

# 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 | |

# LR Parsing Algorithm

***Input:***

- String to parse, w
- Precomputed ACTION and GOTO tables for grammar G

***Output:***

- Success, if w ∈ L(G) plus a trace of rules used
- Failure, if syntax error

```
push state 0 onto the stack
loop
   s = state on top of stack
   c = next input symbol
   if ACTION[s,c] = "Shift N" then
     push c onto the stack
     advance input
     push state N onto stack
   elseif ACTION[s,c] = "Reduce R"
    then
     let rule R be A → β
     pop 2*|β| items off the stack
     s' = state now on stack top
     push A onto stack
     push GOTO[s',A] onto stack
     print "A → β"
   elseif ACTION[s,c] = "Accept"
    then
     return success
   else
     print "Syntax error"
     return
   endIf
endLoop
```

# LR Parsing Algorithm

- The symbol $a_i$ are not to be held on the stack
  - It can be recovered from the state s if needed (never needed in practice)
- A configuration of an LR parser

$$(\underbrace{S_0\, S_1 \ldots S_m}_{\text{Stack contents}},\ \underbrace{a_i\, a_{i+1} \ldots .\, a_n\$}_{\text{remaining input}})$$

represents the corresponding right sentential form

$$X_1\ X_2 \ldots X_m\ a_i\ a_{i+1} \ldots a_n$$

- Essentially similar to shift-reduce parsers
  - Instead of grammar symbol the stack holds states from which grammar symbols can be recovered
  - S0 does not represents a grammar symbol rather bottom-of stack marker

# Modified LR Parsing Algorithm

***Input:***

- String to parse, w
- Precomputed ACTION and GOTO tables for grammar G

***Output:***

- Success, if w ∈ L(G) plus a trace of rules used
- Failure, if syntax error

```
push state 0 onto the stack
loop
  s = state on top of stack
  c = next input symbol
  if ACTION[s,c] = "Shift N" then
    push c onto the stack
    advance input
    push state N onto stack
  elseif ACTION[s,c] = "Reduce R"
   then
    let rule R be A → β
    pop 2*|β| items off the stack
    s' = state now on stack top
    push A onto stack
    push GOTO[s',A] onto stack
    print "A → β"
  elseif ACTION[s,c] = "Accept"
   then
    return success
  else
    print "Syntax error"
    return
  endIf
endLoop
```

# Actions of A (S)LR-Parser – New Version

| stack | input | action | output |
|---|---|---|---|
| 0 | id*id+id$ | shift 5 | |
| 0 5 | *id+id$ | reduce by $F \rightarrow id$ | $F \rightarrow id$ |
| 0 3 | *id+id$ | reduce by $T \rightarrow F$ | $T \rightarrow F$ |
| 0 2 | *id+id$ | shift 7 | |
| 0 2 7 | id+id$ | shift 5 | |
| 0 2 7 5 | +id$ | reduce by $F \rightarrow id$ | $F \rightarrow id$ |
| 0 2 7 10 | +id$ | reduce by $T \rightarrow T*F$ | $T \rightarrow T*F$ |
| 0 2 | +id$ | reduce by $E \rightarrow T$ | $E \rightarrow T$ |
| 0 1 | +id$ | shift 6 | |
| 0 1 6 | id$ | shift 5 | |
| 0 1 6 5 | $ | reduce by $F \rightarrow id$ | $F \rightarrow id$ |
| 0 1 6 3 | $ | reduce by $T \rightarrow F$ | $T \rightarrow F$ |
| 0 1 6 9 | $ | reduce by $E \rightarrow E+T$ | $E \rightarrow E+T$ |
| 0 1 | $ | 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.
  - States represent sets of "items"
- LR parser makes shift-reduce decision by maintaining states to keep track of where we are in a parsing process

# Constructing SLR Parsing Tables – LR(0) Item

- An item indicates how much of a production we have seen at a given point in the parsing process
- For Example the item $A \rightarrow X \bullet YZ$
  - We have already seen on the input a string derivable from X
  - We hope to see a string derivable from YZ
- For Example the item $A \rightarrow \bullet XYZ$
  - We hope to see a string derivable from XYZ
- For Example the item $A \rightarrow XYZ \bullet$
  - We have already seen on the input a string derivable from XYZ
  - It is possibly time to reduce XYZ to A

- Special Case:

  Rule: $A \rightarrow \varepsilon$ yields only one item

  $A \rightarrow \bullet$

# Constructing SLR Parsing Tables

- A collection of sets of LR(0) items (**the canonical LR(0) collection**) is the basis for constructing SLR parsers.
- Canonical LR(0) collection provides the basis of constructing a DFA called **LR(0) automaton**
  - This DFA is used to make parsing decisions
- Each state of LR(0) automaton represents a set of items in the canonical LR(0) collection
- To construct the canonical LR(0) collection for a grammar
  - Augmented Grammar
  - CLOSURE function
  - GOTO function

# Grammar Augmentation

Augment the grammar by adding...

- A new start symbol, S'
- A new rule S' → S

"Goal"

| | |
|---|---|
| 1. E → E + T | 0. S' → E |
| 2. E → T | 1. E → E + T |
| 3. T → T * F | 2. E → T |
| 4. T → F | 3. T → T * F |
| 5. F → ( E ) | 4. T → F |
| 6. F → id | 5. F → ( E ) |
| | 6. F → id |

Our goal is to find an S', followed by $.

S' → • E , $

Whenever we are about to reduce using rule 0...

Accept! Parse is finished!

## 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 $A \rightarrow \alpha.B\beta$ is in ***closure(I)*** and $B \rightarrow \gamma$ is a production rule of G;
   then $B \rightarrow .\gamma$ 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

$E' \to E$

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to id$

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

$\{\ E' \to \bullet E$ ← 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 (CC)

- 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({$S' \rightarrow \bullet S$}) }

  **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

$I_0$: $E' \rightarrow .E$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

$I_1$: $E' \rightarrow E.$
$E \rightarrow E.+T$

$I_2$: $E \rightarrow T.$
$T \rightarrow T.*F$

$I_3$: $T \rightarrow F.$

$I_4$: $F \rightarrow (.E)$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

$I_5$: $F \rightarrow id.$

$I_6$: $E \rightarrow E+.T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

$I_7$: $T \rightarrow T*.F$
$F \rightarrow .(E)$
$F \rightarrow .id$

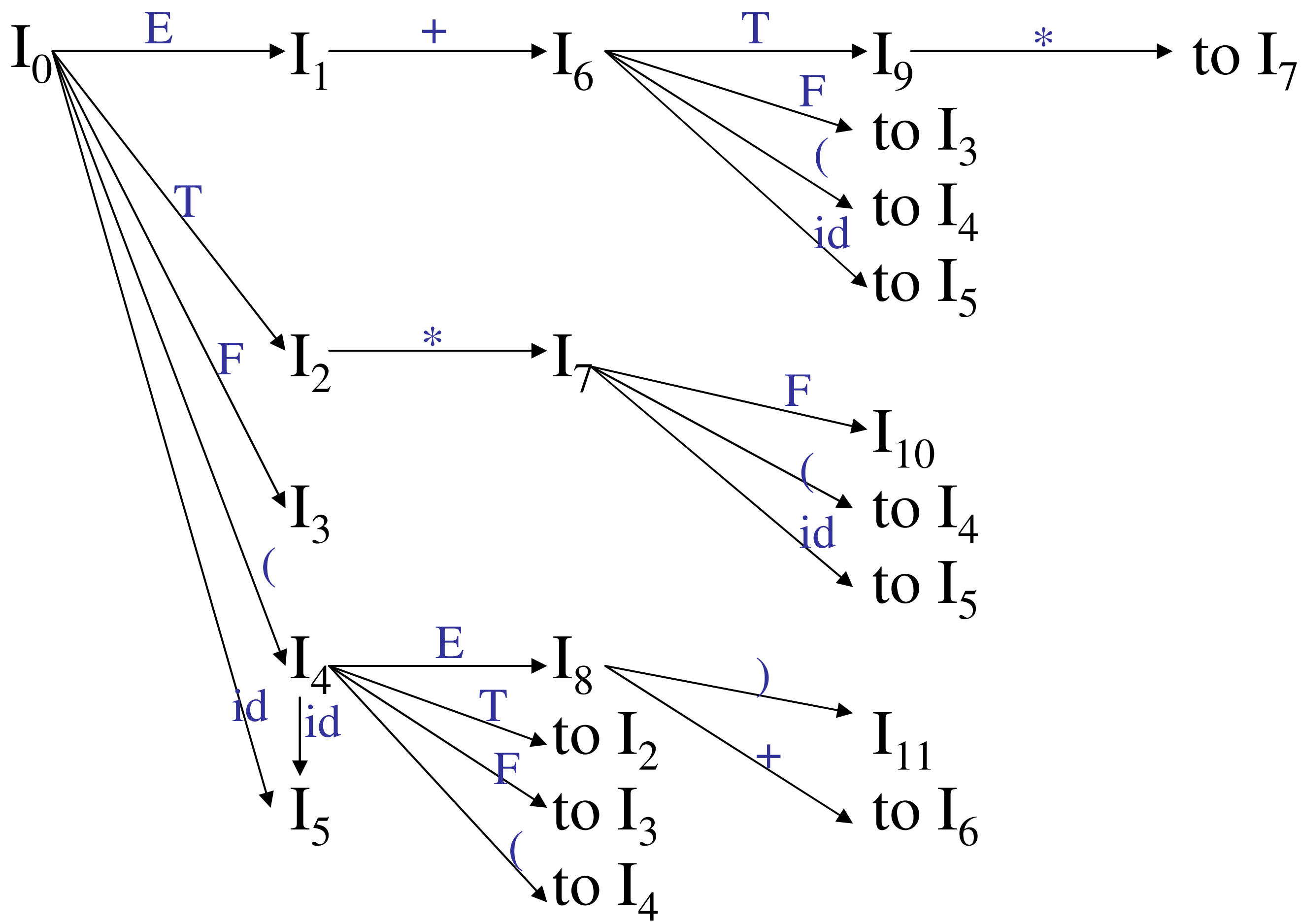
$I_8$: $F \rightarrow (E.)$
$E \rightarrow E.+T$

$I_9$: $E \rightarrow E+T.$
$T \rightarrow T.*F$

$I_{10}$: $T \rightarrow T*F.$

$I_{11}$: $F \rightarrow (E).$

# Transition Diagram (DFA) of Goto Function



# Constructing SLR Parsing Table

(of an augumented grammar G')

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