Syntax Analysis Or Parsing

Lecture 06

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.

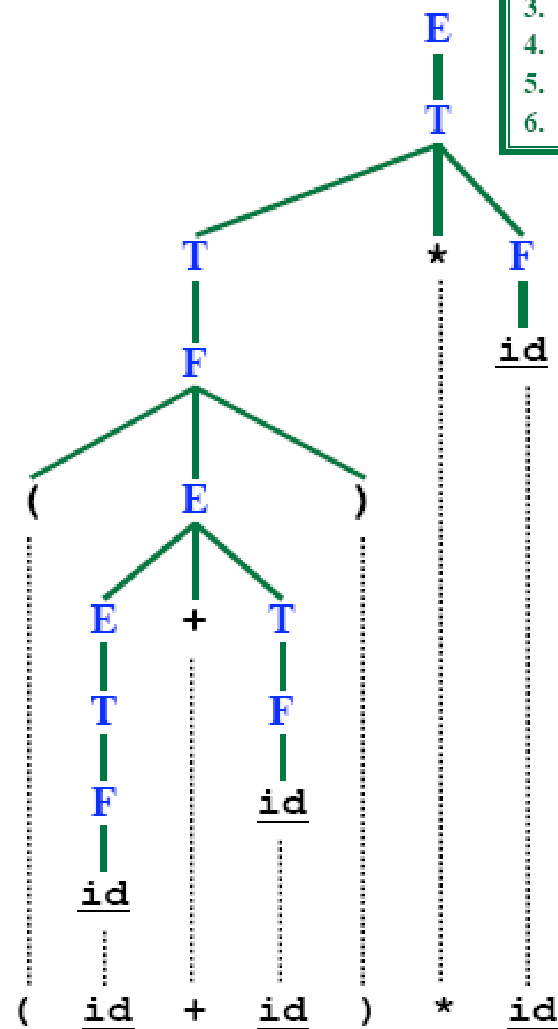
$S \Rightarrow \ldots \Rightarrow \omega$ (the right-most derivation of ω)

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

Rightmost Derivation

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

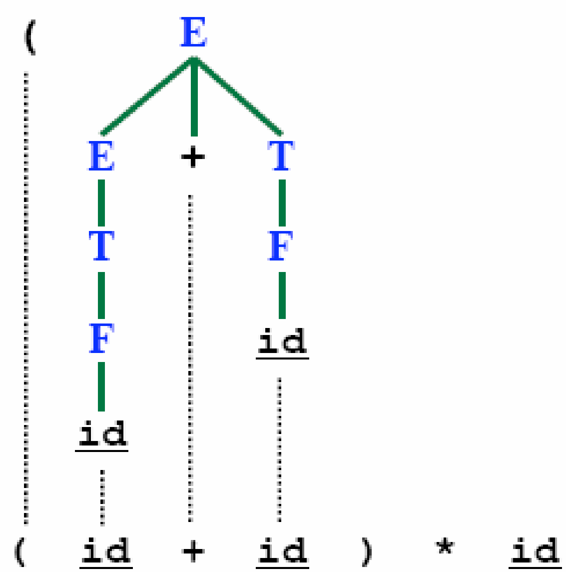
Rules Used:	*Right-Sentential Forms:*
	E
E → T	T
T → T * F	T * F
F → id	T * id
T → F	F * id
F → (E)	(E) * id
E → E + T	(E + T) * id
T → F	(E + F) * id
F → id	(E + id) * id
E → T	(T + id) * id
T → F	(F + id) * id
F → id	(id + id) * id



Rightmost Derivation In Reverse

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

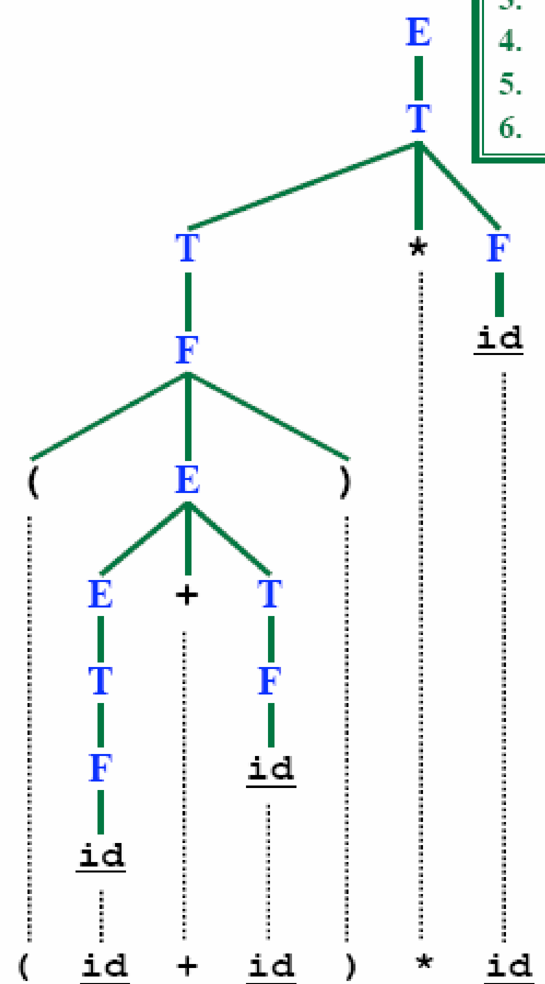
Rules Used:	***Right-Sentential Forms:***
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id



Rightmost Derivation In reverse

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

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id
F → (E)	F * id
T → F	T * id
F → id	T * F
T → T * F	T
E → T	E



LR parsing corresponds to rightmost derivation in reverse

Reduction

- A reduction step replaces a specific substring (matching the body of a production)

```
(id + id) * id
(F + id) * id
(T + id) * id
(E + id) * id
(E + F) * id
(E + T) * id
(E) * id
F * id
T * id
T * F
T
E
```

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

- Reduction is the opposite of derivation
- Bottom up parsing is a process of reducing a string ω to the start symbol S of the grammar

HANDLE

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

Handle Pruning

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

n-th right-sentential form

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

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.

Shift-Reduce Parsing

- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.
- data structures: input-string and stack
- Operations
  - 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.
  - Accept: Announce successful completion of parsing
  - Error: Discover a syntax error and call error recovery

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: abbcde

Rightmost derivation:

S → a T R e
→ a T **d** e
→ a **T b c** d e
→ a b b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: bcde

➔ Shift a, Shift b

a b

Rightmost derivation:

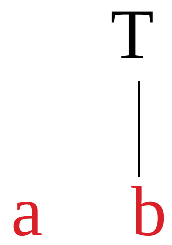
S ➔ a T R e
➔ a T **d** e
➔ a **T b c** d e
➔ a b b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: bcde

→ Shift a, Shift b
→ Reduce T → b



Rightmost derivation:

S → a T R e
→ a T **d** e
→ <u>a **T**</u> **b c** d e
→ **a b** b c d e

SHIFT REDUCE PARSING

S → a T R e
T → T b c | b
R → d

Remaining input: de

→ Shift a, Shift b
→ Reduce T → b
→ Shift b, Shift c

T

a b b c

Rightmost derivation:

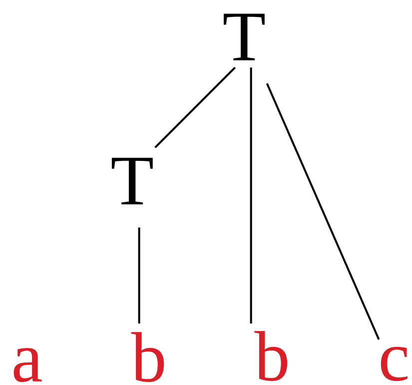
S → a T R e
→ a T **d** e
→ **a T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: de

- → Shift a, Shift b
- → Reduce T → b
- → Shift b, Shift c
- → Reduce T → T b c



Rightmost derivation:

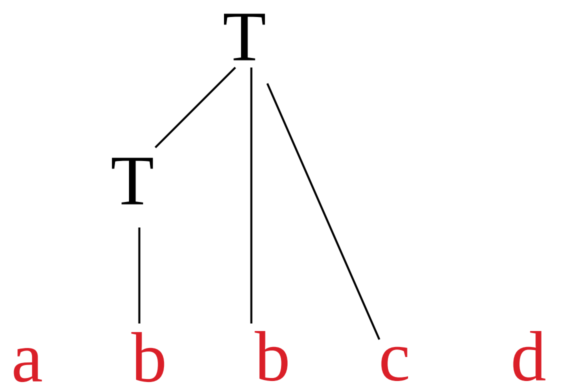
S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: e

- ➔ Shift a, Shift b
- ➔ Reduce T → b
- ➔ Shift b, Shift c
- ➔ Reduce T → T b c
- ➔ Shift d



Rightmost derivation:

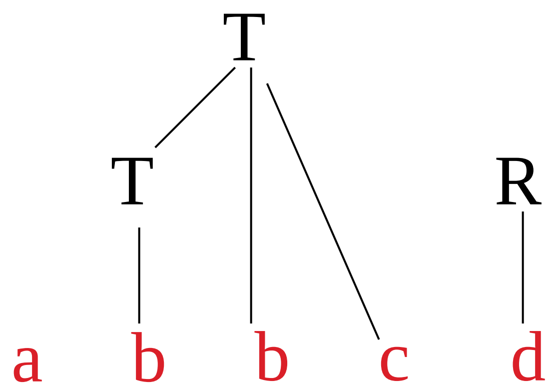
S ➔ a T R e
➔ a T **d** e
➔ a **T b c** d e
➔ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input: e

→ Shift a, Shift b
→ Reduce T → b
→ Shift b, Shift c
→ Reduce T → T b c
→ Shift d
→ Reduce R → d



Rightmost derivation:

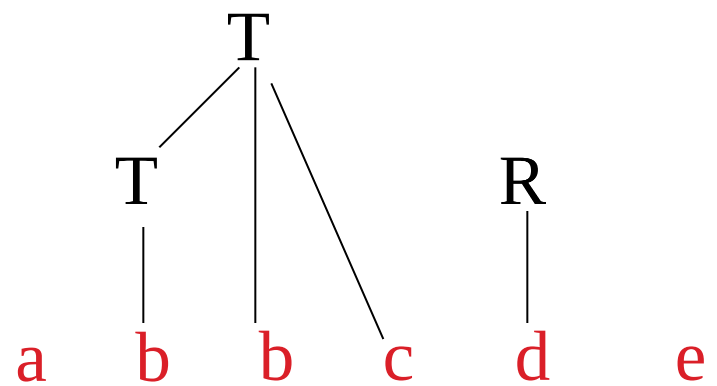
S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e
T → T b c | b
R → d

Remaining input:

- → Shift a, Shift b
- → Reduce T → b
- → Shift b, Shift c
- → Reduce T → T b c
- → Shift d
- → Reduce R → d
- → Shift e



Rightmost derivation:

S → a T R e
→ a T **d** e
→ a **T b c** d e
→ **a b** b c d e

Shift Reduce Parsing

S → a T R e

T → T b c | b

R → d

→ Shift a, Shift b

→ Reduce T → b

→ Shift b, Shift c

→ Reduce T → T b c

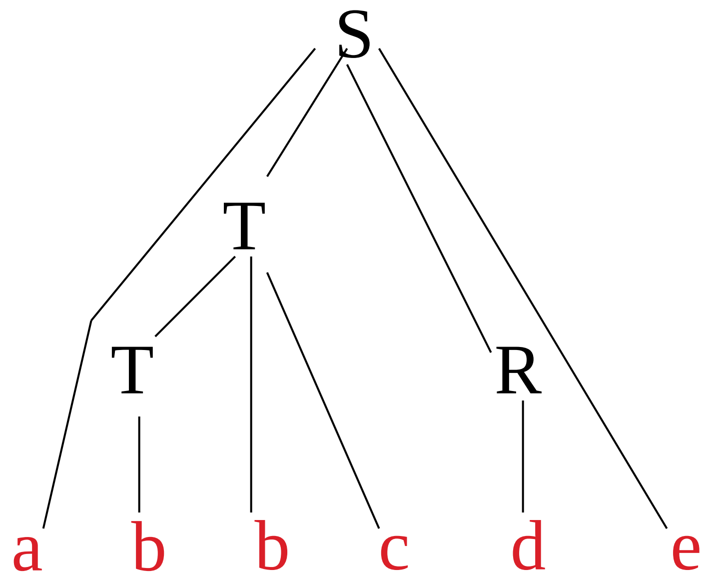
→ Shift d

→ Reduce R → d

→ Shift e

→ Reduce S → a T R e

Remaining input:



Rightmost derivation:

S → a T R e

→ a T **d** e

→ a **T b c** d e

→ **a b** b c d e

Example Shift-Reduce Parsing

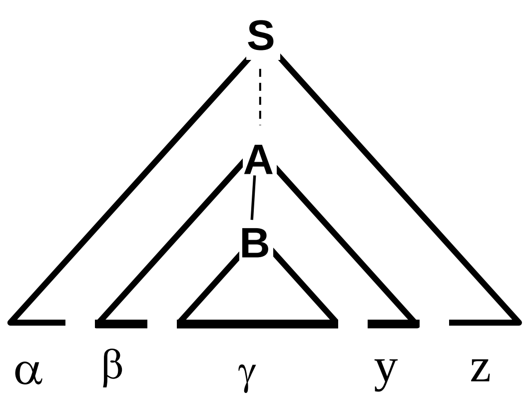
Consider the grammar:

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

Stack	Input	Action
\$	$id_1 + id_2$\$	shift
\$$id_1$	$+ id_2$\$	reduce 6
\$F	$+ id_2$\$	reduce 4
\$T	$+ id_2$\$	reduce 2
\$E	$+ id_2$\$	shift
\$E +	id_2\$	shift
\$E + id_2		reduce 6
\$E + F		reduce 4
\$E + T		reduce 1
\$E		accept

Shift-Reduce Parsing

- Handle will always appear on Top of stack, never inside
- Possible forms of two successive steps in any rightmost derivation
- CASE 1:

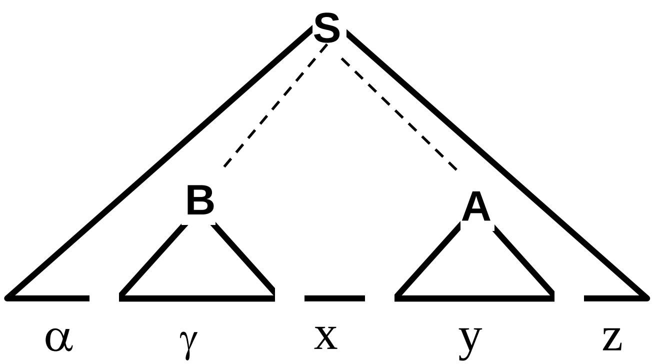


$$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha Az \underset{rm}{\Rightarrow} \alpha\beta Byz \underset{rm}{\Rightarrow} \alpha\beta\gamma yz$$

STACK	**INPUT**
$\$\alpha\beta\gamma$	yz$
After Reducing the handle	
$\$\alpha\beta B$	yz$
Shifting from Input	
$\$\alpha\beta By$	z$
Reduce the handle	
$\$\alpha A$	z$

Shift-Reduce Parsing

- Case 2:



$$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha BxAz \underset{rm}{\Rightarrow} \alpha Bxyz \underset{rm}{\Rightarrow} \alpha\gamma xyz$$

STACK	**INPUT**
$\$\alpha\gamma$	xyz$
After Reducing the handle	
$\$\alpha B$	xyz$
Shifting from Input	
$\$\alpha Bxy$	z$
Reducing the handle	
$\$\alpha BxA$	z$

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 Conflict in Ambiguous Grammar

$$
\begin{aligned}
stmt \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ stmt \\
\mid\ & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\
\mid\ & \textbf{other}
\end{aligned}
$$

STACK

....**if** *expr* **then** *stmt*

- We can't decide whether to shift or reduce?

Reduce-Reduce Conflict in Ambiguous Grammar

1. $stmt \rightarrow \textbf{id}(parameter\_list)$
2. $stmt \rightarrow expr := expr$
3. $parameter\_list \rightarrow parameter\_list, parameter$
4. $parameter\_list \rightarrow parameter$
5. $parameter\_list \rightarrow \textbf{id}$
6. $expr \rightarrow \textbf{id}(expr\_list)$
7. $expr \rightarrow \textbf{id}$
8. $expr\_list \rightarrow expr\_list, expr$
9. $expr\_list \rightarrow expr$

STACK

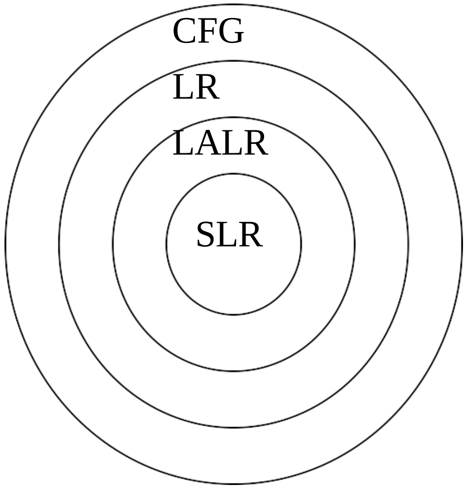
….id (id

- We can't decide which production will be used to reduce **id**?

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 canbe written

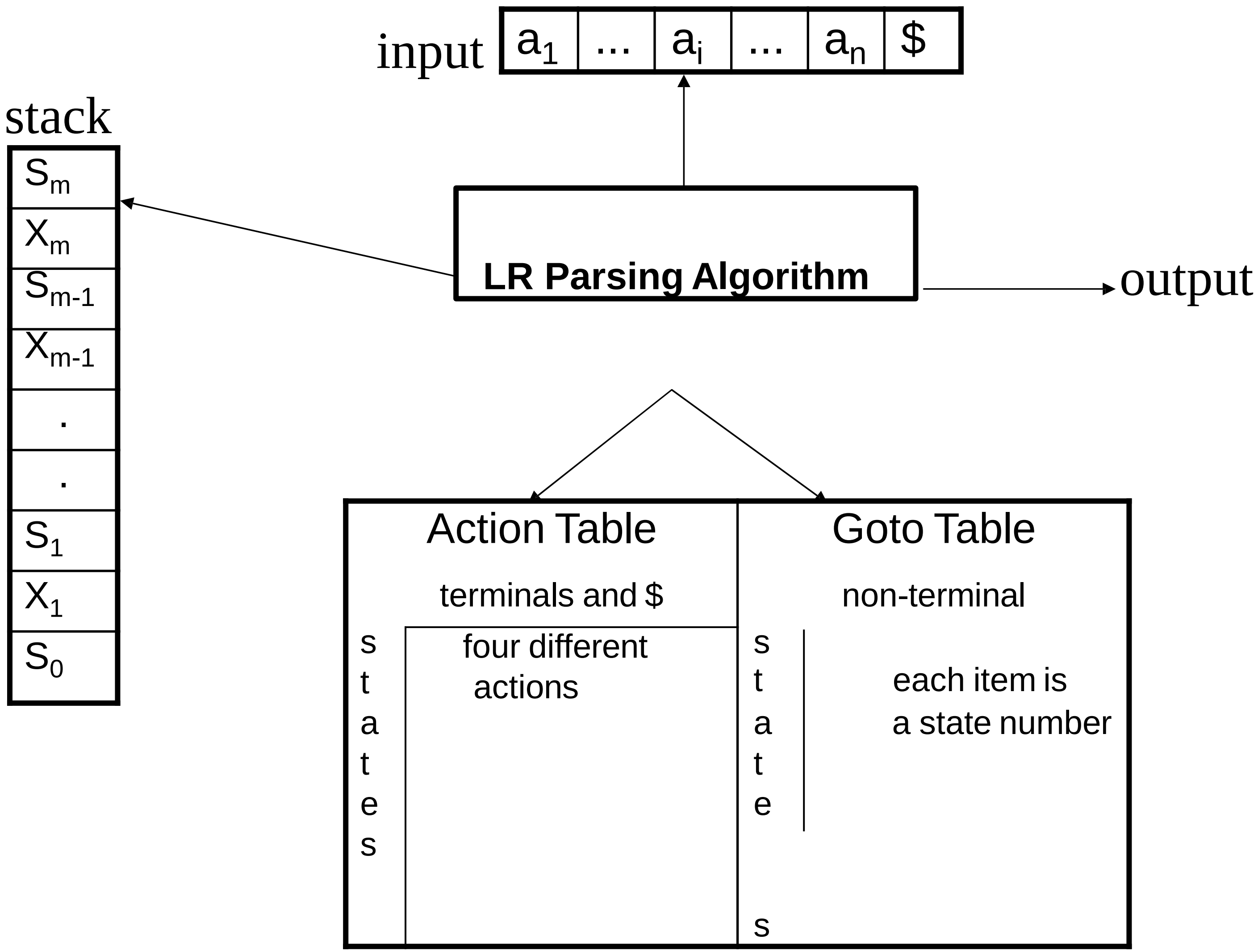
Drawback of LR method:

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

LL vs. LR

- LR (shift reduce) is more powerful than LL (predictive parsing)
- Can detect a syntactic error as soon as possible.
- LR is difficult to do by hand (unlike LL)

LR Parsing Algorithm



A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

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

- 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

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

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]**

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

   - Output is the reducing production reduce A→β

2. **Accept** – Parsing successfully completed

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

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.

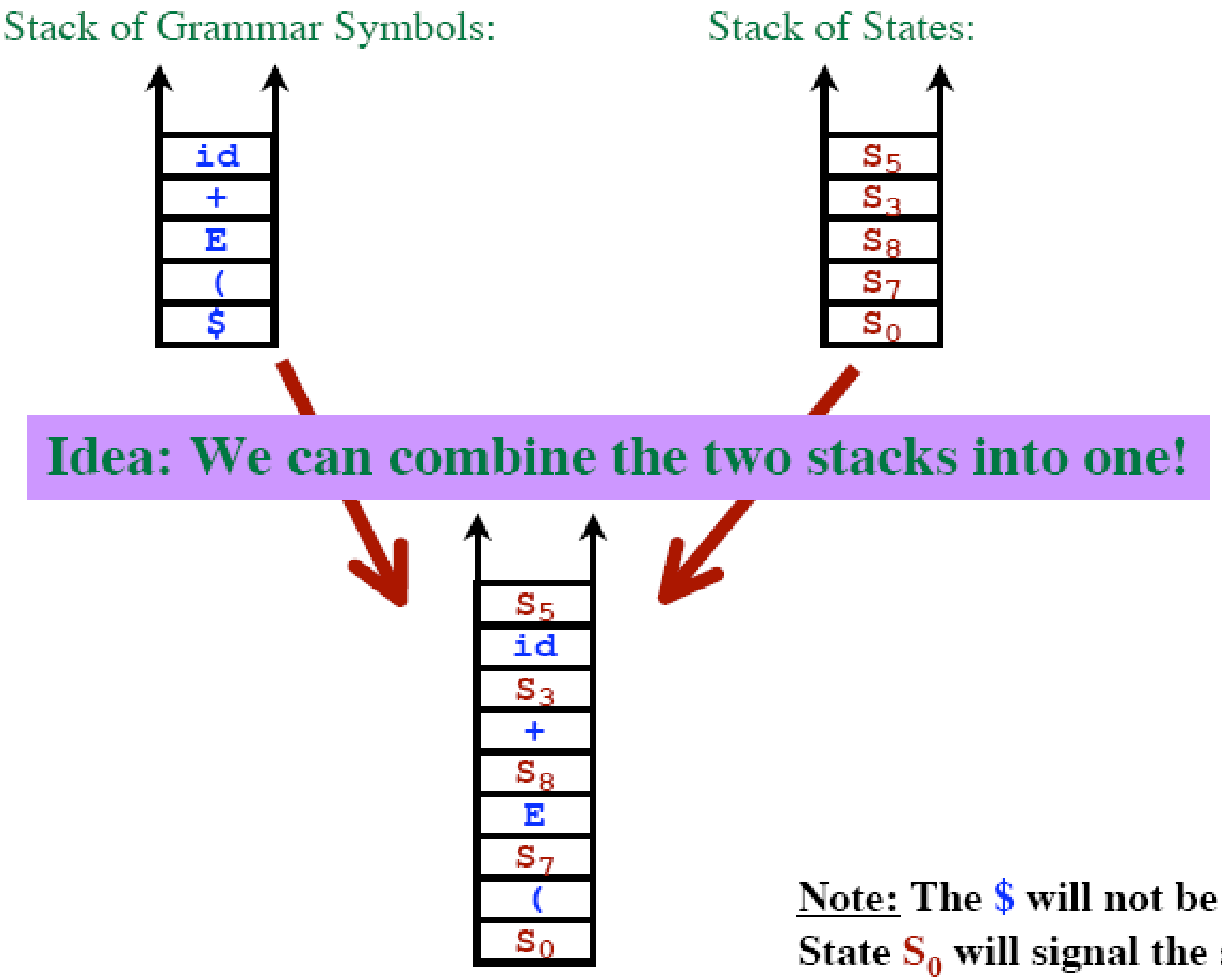
Stack 1: Stack of grammar symbols (terminals and nonterminals)
Stack 2: Stack of "states".

States = { S_0, S_1, S_2, S_3, ... , 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)



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

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

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: (four different possibility)

  $A \rightarrow \bullet aBb$

  $A \rightarrow a \bullet Bb$

  $A \rightarrow aB \bullet b$

  $A \rightarrow aBb \bullet$
- 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

Grammar Augmentation

Augment the grammar by adding...

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

1. E → E + T 2. E → T 3. T → T * F 4. T → F 5. F → (E) 6. F → <u>id</u>	0. S' → E 1. E → E + T 2. E → T 3. T → T * F 4. T → F 5. F → (E) 6. F → <u>id</u>

"Goal"

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

$E' \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

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

{ $E' \rightarrow \bullet E$ kernel items

$E \rightarrow \bullet E+T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T*F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet id$ }

GOTO 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({$S' \to \bullet S$}) }

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

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

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

I_3: $T \to F.$

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

I_5: $F \to id.$

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

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

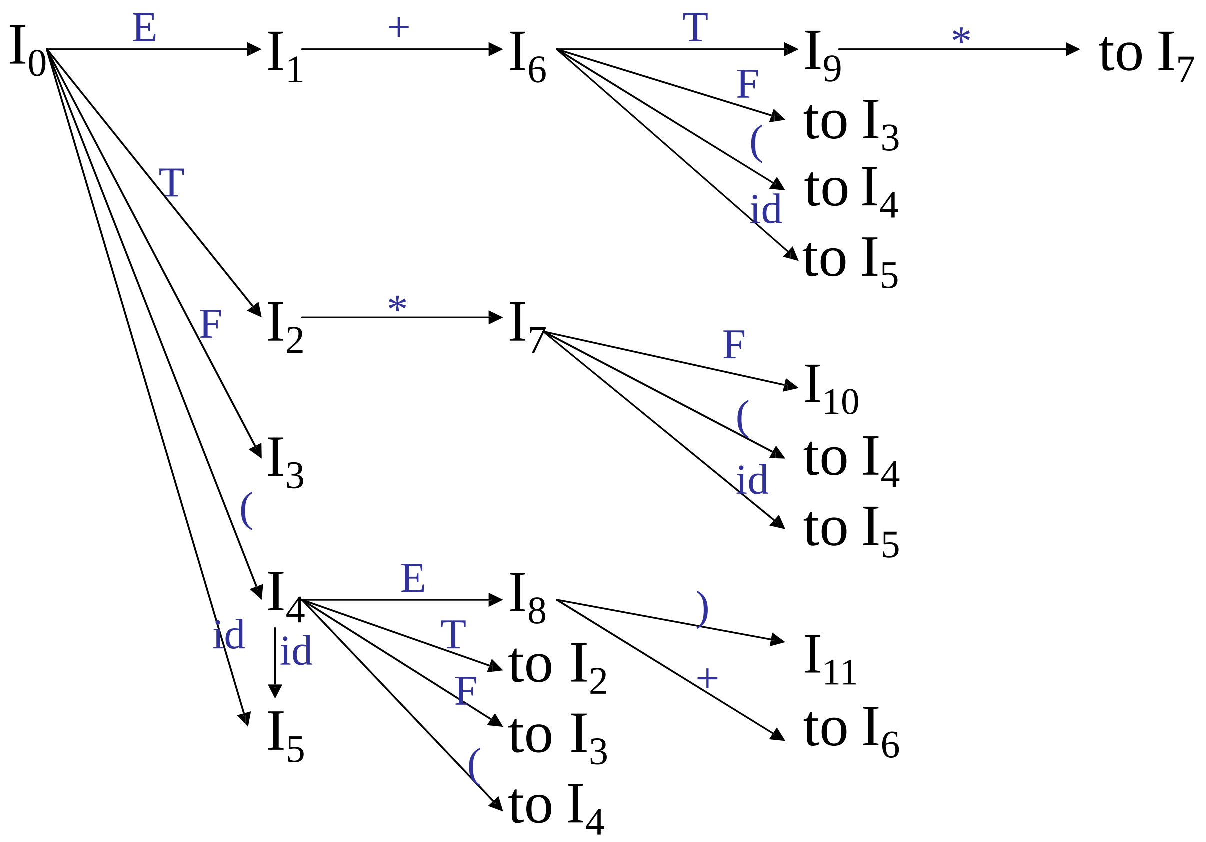
I_8: $F \to (E.)$
$E \to E.+T$

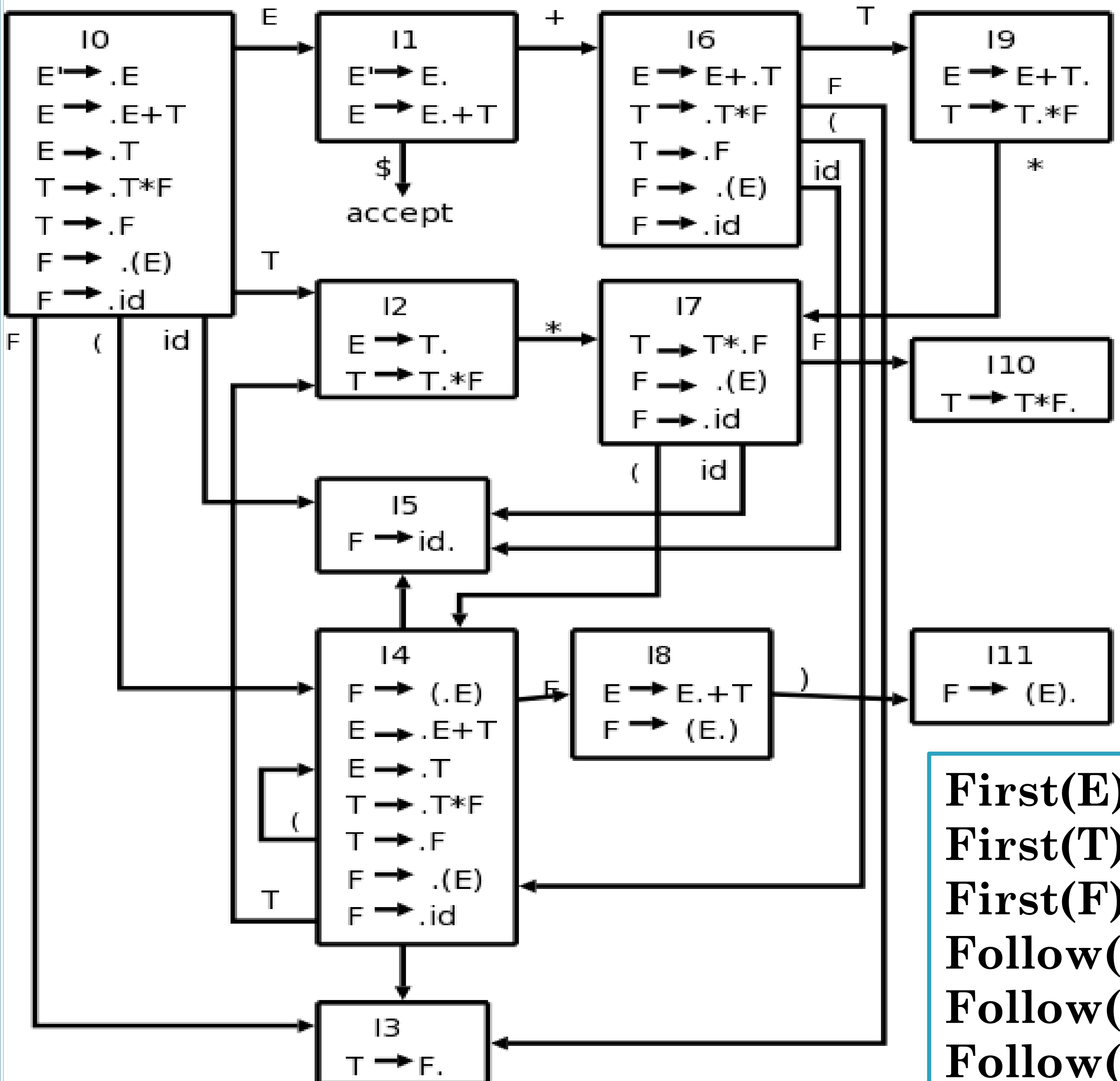
I_9: $E \to E+T.$
$T \to T.*F$

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

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

Transition Diagram (DFA) of Goto Function





First(E)={ (, id }
First(T)={ (, id }
First(F)={ (, id }
Follow(E)={ $,), + }
Follow(T)={ $,), +, * }
Follow(F)={ $,), + , *}

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* $A \rightarrow \alpha$ 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) Parsing Tables for Expression Grammar

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

SLR Grammar: Review

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

1. First item indicates
action[2,=] = shift 6
2. Second item indicates
FOLLOW(R)={=,\$}
action[2,=] = 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 $A \rightarrow \varepsilon$
→ reduce by $B \rightarrow \varepsilon$
reduce/reduce conflict

b → reduce by $A \rightarrow \varepsilon$
→ reduce by $B \rightarrow \varepsilon$
reduce/reduce conflict

Conflict

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

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

If we reduce by Rule 6
then there is no right-sentential
form of the grammar that begins
with R=….

Any Questions ?