Example (Contd.)

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

Firstop+(E) = {+, *, (, id}
Firstop+(T) = {*, (, id}
Firstop+(F) = {(, id}

Lastop+(E) = {+,*,), id}
Lastop+(T) = {*,), id}
Lastop+(F) = {), id}

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

LR Parsing

- The most prevalent type of bottom-up parsers
- LR(k), mostly interested on parsers with $k \leq 1$
- 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

LR Parsing Methods

- SLR – Simple LR. Easy to implement, less powerful
- Canonical LR – most general and powerful. Tedious and costly to implement, contains much more number of states compared to SLR
- LALR – Look Ahead LR. Mix of SLR and Canonical LR. Can be implemented efficiently, contains same number of states as simple LR for a grammar

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

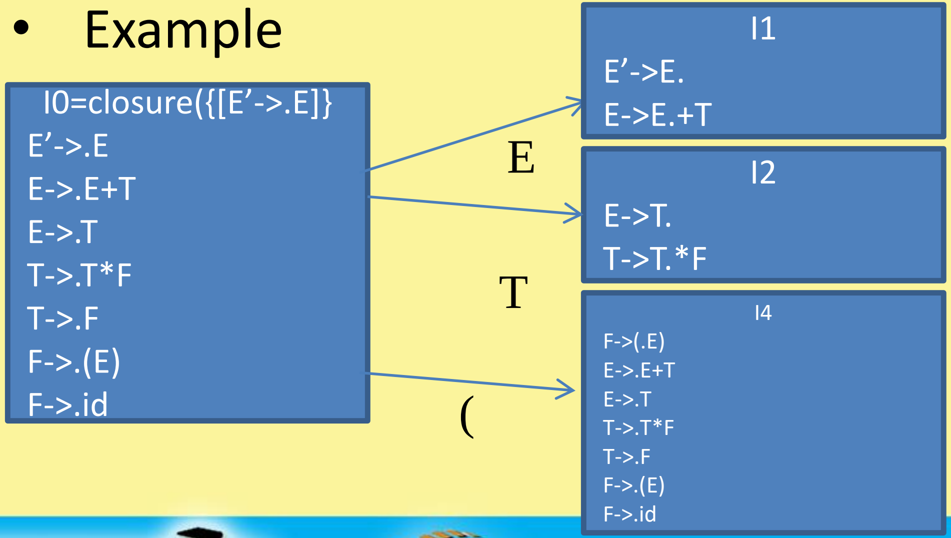
- Augmented grammar:
  - G with addition of a production: S'->S
- Closure of item sets:
  - If I is a set of items, closure(I) is a set of items constructed from I by the following rules:
    - Add every item in I to closure(I)
    - If A->α.Bβ is in closure(I) and B->γ is a production then add the item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



Closure algorithm

```
SetOfItems CLOSURE(I) {
    J=I;
    repeat
        for (each item A-> α.Bβ in J)
            for (each production 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

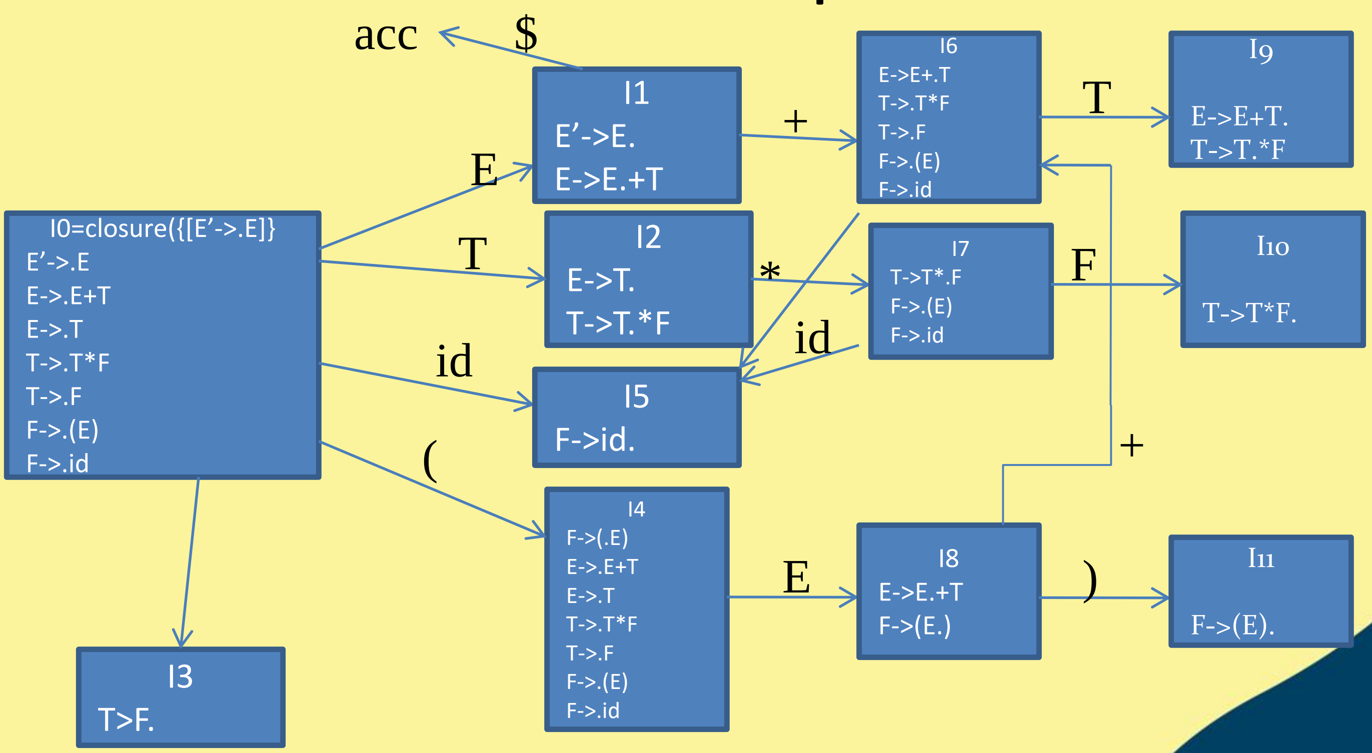
```
SetOfItems  GOTO(I,X) {
  J=empty;
   if (A-> α.X β is in I)
        add CLOSURE(A-> αX. β ) to J;
   return J;
}
```

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**

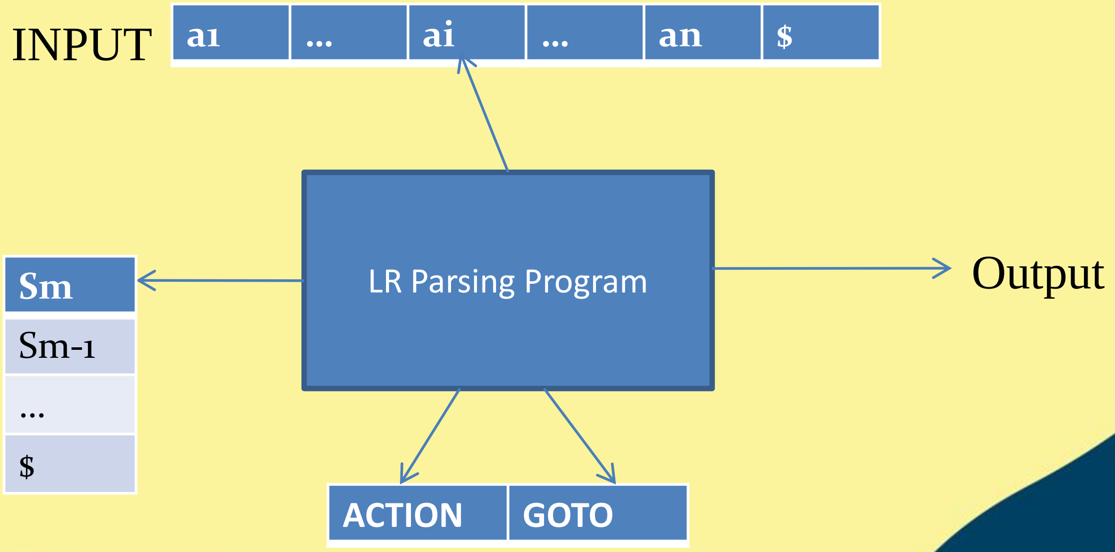


Use of LR(0) automaton

- 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

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

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

(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$

Constructing SLR parsing table

- Method
  - Construct C={I0,I1, … , In}, the collection of LR(0) items for G'
  - State i is constructed from state Ii:
    - If [A->α.aβ] is in Ii and Goto(Ii,a)=Ij, then set ACTION[i,a] to "shift j"
    - If [A->α.] is in Ii, then set ACTION[i,a] to "reduce A->α" for all a in follow(A)
    - If [S'->S.] is in Ii, then set ACTION[i,$] to "Accept"
  - If any conflicts appears then we say that the grammar is not SLR(1).
  - If GOTO(Ii,A) = Ij then GOTO[i,A]=j
  - All entries not defined by above rules are made "error"
  - The initial state of the parser is the one constructed from the set of items containing [S'->.S]

Example grammar which is not SLR

S -> L=R | R
L -> *R | id
R -> L

I0
S'->.S
S -> .L=R
S->.R
L -> .*R |
L->.id
R ->. L

I1
S'->S.

I2
S ->L.=R
R ->L.

I3
S ->R.

I4
L->*.R
R->.L
L->.*R
L->.id

I5
L -> id.

I6
S->L=.R
R->.L
L->.*R
L->.id

I7
L -> *R.

I8
R -> L.

I9
S -> L=R.

Action = Shift 6
Reduce R->L

2

More powerful LR parsers

- Canonical-LR or just LR method
  - Use lookahead symbols for items: LR(1) items
  - Results in a large collection of items
- LALR: lookaheads are introduced in LR(0) items

LR(1) Grammar

- A grammar is said to be LR(1) if in a single left-to-right scan, we can construct a reverse rightmost derivation, while using atmost a single token lookahead to resolve ambiguities.
- LR(k) parsers use k token lookahead

LR(k) item

- A pair [α;β], where
  - α is a production from G with a . at some position in the right hand side
  - β is a lookahead string contains k symbols (terminals or $)
- Several LR(1) items may have same core. [A->X.YZ;a] and [A->X.YZ;b] are represented together as [A->X.YZ;{a,b}]

Usage of LR(1) Lookahead

- Carry them along to allow choosing correct reduction when there is any choice
- Lookaheads are bookkeeping, unless item has a . at right end
  - In [A->X.YZ;a], a has no direct use
  - In [A->XYZ.;a], a is useful
  - If there are two items [A->XYZ.;a] and [B->XYZ.;b], we can decide between reducing to A or B by looking at limited right context

Closure algorithm

```
SetOfItems CLOSURE(I) {
    J=I;
    repeat
        for (each item [A-> α.Bβ;a] in J)
            for (each prodcution B->γ of G and each terminal b in First(βa)
                if ([B->.γ;b] is not in J)
                    add [B->.γ;b] 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.Xβ;a] is in I)
         add CLOSURE([A-> αX.β;a] ) to J;
   return J;
}
```

Constructing LR(1) Parsing Table

- Method
  - Construct C={I0,I1, … , In}, the collection of LR(1) items for G’
  - State i is constructed from state Ii:
    - If [A->α.aβ;b] is in Ii and Goto(Ii,a)=Ij, then set ACTION[i,a] to “shift j”
    - If [A->α.;a] is in Ii, then set ACTION[i,a] to “reduce A->α”
    - If [S’->S.;$] is in Ii, then set ACTION[i,$] to “Accept”
  - If any conflicts appears then we say that the grammar is not SLR(1).
  - If GOTO(Ii,A) = Ij then GOTO[i,A]=j
  - All entries not defined by above rules are made “error”
  - The initial state of the parser is the one constructed from the set of items containing [S’->.S;$]

Example LR(1) Parser

goal -> expr
expr -> term + expr
expr -> term
term -> factor * term
term -> factor
factor -> id

I_0 : $[goal \to \cdot expr, \$], [expr \to \cdot term + expr, \$], [expr \to \cdot term, \$], [term \to \cdot factor * term, \{+, \$\}],$
$[term \to \cdot factor, \{+, \$\}], [factor \to \cdot \mathbf{id}, \{+, *, \$\}]$

I_1 : $[goal \to expr \cdot, \$]$

I_2 : $[expr \to term \cdot, \$], [expr \to term \cdot +expr, \$]$

I_3 : $[term \to factor \cdot, \{+, \$\}], [term \to factor \cdot *term, \{+, \$\}]$

I_4 : $[factor \to \mathbf{id} \cdot, \{+, *, \$\}]$

I_5 : $[expr \to term + \cdot expr, \$], [expr \to \cdot term + expr, \$], [expr \to \cdot term, \$],$
$[term \to factor * term, \{+, \$\}], [term \to \cdot factor, \{+, \$\}], [factor \to \cdot \mathbf{id}, \{+, *, \$\}]$

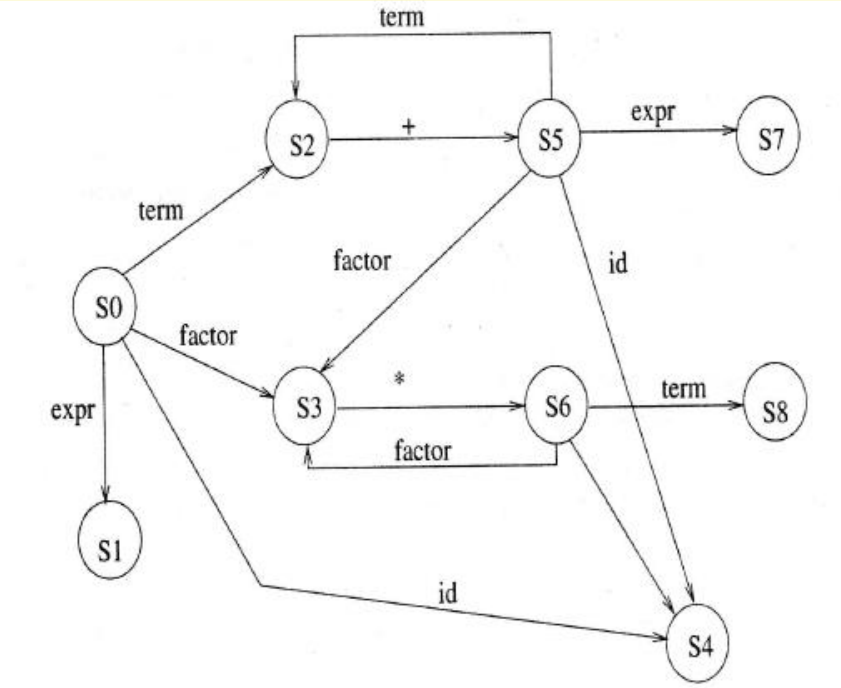
I_6 : $[term \to factor * \cdot term, \{+, \$\}], [term \to \cdot factor * term, \{+, \$\}], [term \to \cdot factor, \{+, \$\}],$
$[factor \to \cdot \mathbf{id}, \{+, *, \$\}]$

I_7 : $[expr \to term + expr \cdot, \$]$

I_8 : $[term \to factor * term \cdot, \{+, \$\}]$

LR(1) items

Example LR(1) Parser (Contd.)



	id	+	*	$	Expr	Term	factor
0	S4				1	2	3
1				Acc			
2		S5		R3			
3		S5	R6	R5			
4		R6	R6	R6			
5	S4				7	2	3
6	S4					8	3
7				R2			
8		R4		R4			

LALR(1) Parsing

- Reduces number of states in an LR(1) parser
- Merges states differing only in lookahead sets
- SLR and LALR tables have same number of states
- For a C-like language, several hundred states in SLR and LALR parsers, several thousands for LR(1)

Example

S' -> S
S -> CC
C -> cC | d

LR(1) items

I_0	:	$[S' \rightarrow \cdot S, \$], [S \rightarrow \cdot CC, \$], [C \rightarrow \cdot cC, \{c,d\}], [C \rightarrow \cdot d, \{c,d\}]$
I_1	:	$[S' \rightarrow S\cdot, \$]$
I_2	:	$[S \rightarrow C \cdot C, \$], [C \rightarrow .cC, \$], [C \rightarrow \cdot d, \$]$
I_3	:	$[C \rightarrow c \cdot C, \{c,d\}], [C \rightarrow \cdot cC, \{c,d\}], [C \rightarrow \cdot d, \{c,d\}]$
I_4	:	$[C \rightarrow d\cdot, \{c,d\}]$
I_5	:	$[C \rightarrow CC\cdot, \$]$
I_6	:	$[C \rightarrow c \cdot C, \$], [C \rightarrow .cC, \$], [C \rightarrow .d, \$]$
I_7	:	$[C \rightarrow d\cdot, \$]$
I_8	:	$[C \rightarrow cC\cdot, \{c,d\}]$
I_9	:	$[C \rightarrow cC\cdot, \$]$

States I_4 and I_7, I_3 and I_6, I_8 and I_9 can be merged

I_{47}: [C ->d.;{c,d,$}]
I_{36}: [C->c.C;{c,d,$], [C->.c;{c,d,$}],
[C->.d;{c,d,$}]
I_{89}: [C->cC.;{c,d,$}]

LALR Construction – Step-by-Step Approach

- Sets of states constructed as in LR(1) method
- At each point where a new set is spawned, it may be merged with an existing set
- When a new state S is created, all other states are checked to see if one with the same core exists
- If not, S is kept; otherwise it is merged with the existing set T with the same core to form state ST

Using Ambiguous Grammars

E->E+E
E->E*E
E->(E)
E->id

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

I0: E'->.E
E->.E+E
E->.E*E
E->.(E)
E->.id

I1: E'->E.
E->E.+E
E->E.*E

I2: E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id

I3: E->id.

I4: E->E+.E
E->.E+E
E->.E*E
E->.(E)
E->.id

I5: E->E*.E
E->(.E)
E->.E+E
E->.E*E
E->.(E)
E->.id

I6: E->(E.)
E->E.+E
E->E.*E

I7: E->E+E.
E->E.+E
E->E.*E

I8: E->E*E.
E->E.+E
E->E.*E

I9: E->(E).

STATE	ACTON						GO TO
	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				
7		**R1**/ S4	**S5**/ R1		R1	R1	
8		**R2**/ S4	**R2**/ S5		R2	R2	
9		R3	R3		R3	R3	

Error Recovery in LR Parsing

- Undefined entries in LR parsing table means error
- Proper error messages can be flashed to the user
- Error handling routines can be made to modify the parser stack by
  - popping out some entries
  - pushing some desirable entries into the stack
- Brings parser at a descent stage from which it can proceed further
- Enables detection of multiple errors and flashing them to the user for correction

Error Recovery – Example

E’ -> E

E -> E + E | E * E| id

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

I0: {[E’->.E],[E->.E+E],[E->.E*E],[E-.id]}
I1: {[E’->E.],[E->E.+E],[E->E.*E]}
I2: {[E->id.]}
I3: {[E->E+.E],[E->.E+E],[E->.E*E],[E->.id]}
I4: {[E->E*.E],[E->.E+E],[E->.E*E],[E->.id]}
I5: {[E->E+E.],[E->E.+E],[E->E.*E]}
I6: {E->E+E.],[E->E.+E],[E->E.*E]}
I7: {[E->E*E.],[E->E.+E],[E->E.*E]}

State	ACTION				GOTO
	id	+	*	$	E
0	S2	e1	e1	e1	1
1	e2	S3	S4	Acc	
2	e2	R3	R3	R3	
3	S2	e1	e1	e1	5
4	S2	e1	e1	e1	6
5	e2	R1	S4	R1	
6	e2	R2	R2	R2	

e1: Seen operator or end of string while expecting id

e2: Seen id while expecting operator