

Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root OR
- Reduce a string w of input to start symbol of grammar

Consider a grammar

$$S \rightarrow aABe$$

$$A \rightarrow Abc \mid b$$

$$B \rightarrow d$$

And reduction of a string

$$a \underline{b} b c d e$$

$$a \underline{A} b c d e$$

$$a A \underline{d} e$$

$$\underline{a A B e}$$

S

The sentential forms happen to be a *right most derivation in the reverse order*.

$$S \rightarrow a A \underline{B} e$$

$$\rightarrow a \underline{A} d e$$

$$\rightarrow a \underline{A} b c d e$$

$$\rightarrow a b b c d e$$

Shift reduce parsing

- Split string being parsed into two parts
 - Two parts are separated by a special character “.”
 - Left part is a string of terminals and non terminals
 - Right part is a string of terminals
- Initially the input is `.w`

Shift reduce parsing ...

- Bottom up parsing has two actions
- **Shift**: move terminal symbol from right string to left string

if string before shift is $\alpha.pqr$

then string after shift is $\alpha p.qr$

Shift reduce parsing ...

- **Reduce**: immediately on the left of “.” identify a string same as RHS of a production and replace it by LHS

if string before reduce action is $\alpha\beta.pqr$

and $A \rightarrow \beta$ is a production

then string after reduction is $\alpha A.pqr$

Example

Assume grammar is $E \rightarrow E+E \mid E * E \mid id$

Parse $id * id + id$

Assume an oracle tells you when to shift / when to reduce

| String | action (by oracle) |
|---------------|------------------------------|
| .id*id+id | shift |
| id.*id+id | reduce $E \rightarrow id$ |
| E.*id+id | shift |
| E*.id+id | shift |
| E*id.+id | reduce $E \rightarrow id$ |
| E * E . + id | reduce $E \rightarrow E * E$ |
| E . + id | shift |
| E + . id | shift |
| E + id . | Reduce $E \rightarrow id$ |
| E + E . | Reduce $E \rightarrow E + E$ |
| E . | ACCEPT |

Shift reduce parsing ...

- Symbols on the left of “.” are kept on a stack
 - Top of the stack is at “.”
 - Shift pushes a terminal on the stack
 - Reduce pops symbols (rhs of production) and pushes a non terminal (lhs of production) onto the stack
- The most important issue: when to shift and when to reduce
- Reduce action should be taken only if the result can be reduced to the start symbol

Issues in bottom up parsing

- How do we know which action to take
 - whether to shift or reduce
 - Which production to use for reduction?
- Sometimes parser can reduce but it should not:
 - $X \rightarrow \epsilon$ can always be used for reduction!

Issues in bottom up parsing

- Sometimes parser can reduce in different ways!
- Given stack δ and input symbol a , should the parser
 - Shift a onto stack (making it δa)
 - Reduce by some production $A \rightarrow \beta$ assuming that stack has form $\alpha\beta$ (making it αA)
 - Stack can have many combinations of $\alpha\beta$
 - How to keep track of length of β ?

Handles

- The basic steps of a bottom-up parser are
 - to identify a *substring* within a *rightmost sentential* form which matches the RHS of a rule.
 - when this substring is replaced by the LHS of the matching rule, it must produce the previous rightmost-sentential form.
- Such a substring is called a *handle*

Handle

- A *handle* of a right sentential form γ is
 - a production rule $A \rightarrow \beta$, and
 - an occurrence of a sub-string β in γ

such that

- when the occurrence of β is replaced by A in γ , we get the previous right sentential form in a rightmost derivation of γ .

Handle

Formally, if

$$S \xrightarrow{rm^*} \alpha A w \xrightarrow{rm} \alpha \beta w,$$

then

- β in the position following α ,
 - and the corresponding production $A \rightarrow \beta$ is a handle of $\alpha \beta w$.
-
- The string w consists of only terminal symbols

Handle

- We only want to reduce handle and not any RHS
- **Handle pruning**: If β is a handle and $A \rightarrow \beta$ is a production then replace β by A
- A right most derivation in reverse can be obtained by handle pruning.

Handle: Observation

- *Only terminal symbols can appear to the right of a handle in a rightmost sentential form.*
- Why?

Handle: Observation

Is this scenario possible:

- $\alpha\beta\gamma$ is the content of the stack
- $A \rightarrow \gamma$ is a handle
- The stack content reduces to $\alpha\beta A$
- Now $B \rightarrow \beta$ is the handle

In other words, handle is not on top, but buried *inside* stack

Not Possible! Why?

Handles ...

- Consider two cases of right most derivation to understand the fact that handle appears on the top of the stack

$$S \rightarrow \alpha Az \rightarrow \alpha\beta Byz \rightarrow \alpha\beta\gamma yz$$

$$S \rightarrow \alpha BxAz \rightarrow \alpha Bxyz \rightarrow \alpha\gamma xyz$$

Handle always appears on the top

Case I: $S \rightarrow \alpha Az \rightarrow \alpha\beta Byz \rightarrow \alpha\beta\gamma yz$

| stack | input | action |
|---------------------|-------|------------------------------------|
| $\alpha\beta\gamma$ | yz | reduce by $B \rightarrow \gamma$ |
| $\alpha\beta B$ | yz | shift y |
| $\alpha\beta By$ | z | reduce by $A \rightarrow \beta By$ |
| αA | z | |

Case II: $S \rightarrow \alpha Bx Az \rightarrow \alpha Bxyz \rightarrow \alpha\gamma xyz$

| stack | input | action |
|----------------|-------|----------------------------------|
| $\alpha\gamma$ | xyz | reduce by $B \rightarrow \gamma$ |
| αB | xyz | shift x |
| αBx | yz | shift y |
| αBxy | z | reduce $A \rightarrow \gamma$ |
| αBxA | z | |

Shift Reduce Parsers

- The general shift-reduce technique is:
 - if there is no handle on the stack then shift
 - If there is a handle then reduce
- Bottom up parsing is essentially the process of detecting handles and reducing them.
- Different bottom-up parsers differ in the way they detect handles.

Conflicts

- What happens when there is a choice
 - What action to take in case both shift and reduce are valid?
shift-reduce conflict
 - Which rule to use for reduction if reduction is possible by more than one rule?
reduce-reduce conflict

Conflicts

- Conflicts come either because of ambiguous grammars or parsing method is not powerful enough

Shift reduce conflict

Consider the grammar $E \rightarrow E+E \mid E^*E \mid id$

and the input $id+id^*id$

| stack | input | action |
|-------|-------|--------------------------------|
| E+E | *id | reduce by $E \rightarrow E+E$ |
| E | *id | shift |
| E* | id | shift |
| E*id | | reduce by $E \rightarrow id$ |
| E*E | | reduce by $E \rightarrow E^*E$ |
| E | | |

| stack | input | action |
|--------|-------|--------------------------------|
| E+E | *id | shift |
| E+E* | id | shift |
| E+E*id | | reduce by $E \rightarrow id$ |
| E+E*E | | reduce by $E \rightarrow E^*E$ |
| E+E | | reduce by $E \rightarrow E+E$ |
| E | | |

Reduce reduce conflict

Consider the grammar $M \rightarrow R+R \mid R+c \mid R$

$R \rightarrow c$

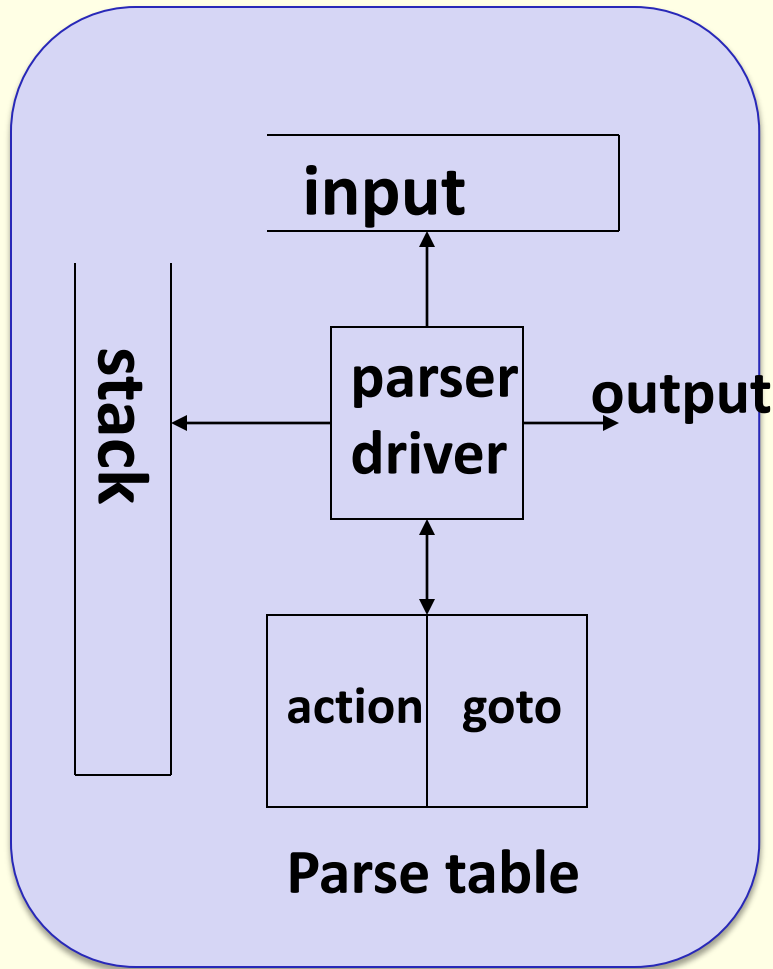
and the input

$c+c$

| Stack | input | action |
|-------|-------|-------------------------------|
| | c+c | shift |
| c | +c | reduce by $R \rightarrow c$ |
| R | +c | shift |
| R+ | c | shift |
| R+c | | reduce by $R \rightarrow c$ |
| R+R | | reduce by $M \rightarrow R+R$ |
| M | | |

| Stack | input | action |
|-------|-------|-------------------------------|
| | c+c | shift |
| c | +c | reduce by $R \rightarrow c$ |
| R | +c | shift |
| R+ | c | shift |
| R+c | | reduce by $M \rightarrow R+c$ |
| M | | |

LR parsing



- Input buffer contains the input string.
- Stack contains a string of the form $S_0X_1S_1X_2\dots X_nS_n$ where each X_i is a grammar symbol and each S_i is a state.
- Table contains action and goto parts.
- action table is indexed by state and terminal symbols.
- goto table is indexed by state and non terminal symbols.

Example

Consider a grammar
and its parse table

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

| 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 | | | |

action

goto

Actions in an LR (shift reduce) parser

- Assume S_i is top of stack and a_i is current input symbol
- Action $[S_i, a_i]$ can have four values
 1. sj : shift a_i to the stack, goto state S_j
 2. rk : reduce by rule number k
 3. acc : Accept
 4. err : Error (empty cells in the table)

Driving the LR parser

Stack: $S_0 X_1 S_1 X_2 \dots X_m S_m$ **Input:** $a_i a_{i+1} \dots a_n \$$

- If $\text{action}[S_m, a_i] = \text{shift } S$

Then the configuration becomes

Stack: $S_0 X_1 S_1 \dots X_m S_m a_i S$ **Input:** $a_{i+1} \dots a_n \$$

- If $\text{action}[S_m, a_i] = \text{reduce } A \rightarrow \beta$

Then the configuration becomes

Stack: $S_0 X_1 S_1 \dots X_{m-r} S_{m-r} A S$ **Input:** $a_i a_{i+1} \dots a_n \$$

Where $r = |\beta|$ and $S = \text{goto}[S_{m-r}, A]$

Driving the LR parser

Stack: $S_0 X_1 S_1 X_2 \dots X_m S_m$ **Input:** $a_i a_{i+1} \dots a_n \$$

- If $\text{action}[S_m, a_i] = \text{accept}$
Then parsing is completed. HALT
- If $\text{action}[S_m, a_i] = \text{error}$ (or empty cell)
Then invoke error recovery routine.

Parse $id + id * id$

| Stack | Input | Action |
|------------------------|--------------|---------------------------------|
| 0 | $id+id*id\$$ | shift 5 |
| 0 id 5 | $+id*id\$$ | reduce by $F \rightarrow id$ |
| 0 F 3 | $+id*id\$$ | reduce by $T \rightarrow F$ |
| 0 T 2 | $+id*id\$$ | reduce by $E \rightarrow T$ |
| 0 E 1 | $+id*id\$$ | shift 6 |
| 0 E 1 + 6 | $id*id\$$ | shift 5 |
| 0 E 1 + 6 id 5 | $*id\$$ | reduce by $F \rightarrow id$ |
| 0 E 1 + 6 F 3 | $*id\$$ | reduce by $T \rightarrow F$ |
| 0 E 1 + 6 T 9 | $*id\$$ | shift 7 |
| 0 E 1 + 6 T 9 * 7 | $id\$$ | shift 5 |
| 0 E 1 + 6 T 9 * 7 id 5 | $\$$ | reduce by $F \rightarrow id$ |
| 0 E 1 + 6 T 9 * 7 F 10 | $\$$ | reduce by $T \rightarrow T * F$ |
| 0 E 1 + 6 T 9 | $\$$ | reduce by $E \rightarrow E + T$ |
| 0 E 1 | $\$$ | ACCEPT |

Configuration of a LR parser

- The tuple
 <Stack Contents, Remaining Input>
defines a *configuration* of a LR parser

- Initially the configuration is

$$\langle S_0, a_0a_1\dots a_n\$ \rangle$$

- Typical final configuration on a successful parse is

$$\langle S_0X_1S_i, \$ \rangle$$

LR parsing Algorithm

Initial state: **Stack:** S_0 **Input:** $w\$$

```
while (1) {  
  if (action[S,a] = shift S') {  
    push(a); push(S'); ip++  
  } else if (action[S,a] = reduce  $A \rightarrow \beta$ ) {  
    pop ( $2 * |\beta|$ ) symbols;  
    push(A); push (goto[S'',A])  
    ( $S''$  is the state at stack top after popping symbols)  
  } else if (action[S,a] = accept) {  
    exit  
  } else { error }  
}
```

Constructing parse table

Augment the grammar

- G is a grammar with start symbol S
- The augmented grammar G' for G has a new start symbol S' and an additional production $S' \rightarrow S$
- When the parser reduces by this rule it will stop with accept

Production to Use for Reduction

- How do we know which production to apply in a given configuration
- We can guess!
 - May require backtracking
- Keep track of “ALL” possible rules that can apply at a given point in the input string
 - But in general, there is no upper bound on the length of the input string
 - Is there a bound on number of applicable rules?

Some hands on!

- $E' \rightarrow E$
- $E \rightarrow E + T$
- $E \rightarrow T$
- $T \rightarrow T * F$
- $T \rightarrow F$
- $F \rightarrow (E)$
- $F \rightarrow id$

Strings to Parse

- $id + id + id + id$
- $id * id * id * id$
- $id * id + id * id$
- $id * (id + id) * id$

Parser states

- Goal is to know the valid reductions at any given point
- Summarize all possible stack prefixes α as a parser state
- Parser state is defined by a DFA state that reads in the stack α
- Accept states of DFA are unique reductions

Viable prefixes

- α is a viable prefix of the grammar if
 - $\exists w$ such that αw is a right sentential form
 - $\langle \alpha, w \rangle$ is a configuration of the parser
- As long as the parser has viable prefixes on the stack no parser error has been seen
- The set of viable prefixes is a regular language
- We can construct an automaton that accepts viable prefixes

LR(0) items

- An LR(0) item of a grammar G is a production of G with a special symbol “.” at some position of the right side
- Thus production $A \rightarrow XYZ$ gives four LR(0) items

$A \rightarrow .XYZ$

$A \rightarrow X.YZ$

$A \rightarrow XY.Z$

$A \rightarrow XYZ.$

LR(0) items

- An item indicates how much of a production has been seen at a point in the process of parsing
 - Symbols on the left of “.” are already on the stacks
 - Symbols on the right of “.” are expected in the input

Start state

- Start state of DFA is an empty stack corresponding to $S' \rightarrow .S$ item
- This means no input has been seen
- The parser expects to see a string derived from S

Closure of a state

- **Closure** of a state adds items for all productions whose LHS occurs in an item in the state, just after “.”
 - Set of possible productions to be reduced next
 - Added items have “.” located at the beginning
 - No symbol of these items is on the stack as yet

Closure operation

- Let I be a set of items for a grammar G
- $\text{closure}(I)$ is a set constructed as follows:
 - Every item in I is in $\text{closure}(I)$
 - If $A \rightarrow \alpha.B\beta$ is in $\text{closure}(I)$ and $B \rightarrow \gamma$ is a production then $B \rightarrow .\gamma$ is in $\text{closure}(I)$
- Intuitively $A \rightarrow \alpha.B\beta$ indicates that we expect a string derivable from $B\beta$ in input
- If $B \rightarrow \gamma$ is a production then we might see a string derivable from γ at this point

Example

For the grammar

$$E' \rightarrow E$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \text{id}$$

If I is $\{ E' \rightarrow .E \}$ then
 $\text{closure}(I)$ is

$$E' \rightarrow .E$$
$$E \rightarrow .E + T$$
$$E \rightarrow .T$$
$$T \rightarrow .T * F$$
$$T \rightarrow .F$$
$$F \rightarrow .\text{id}$$
$$F \rightarrow .(E)$$

Goto operation

- $\text{Goto}(I, X)$, where I is a set of items and X is a grammar symbol,
 - is closure of set of item $A \rightarrow \alpha X \beta$
 - such that $A \rightarrow \alpha X \beta$ is in I
- Intuitively if I is a set of items for some valid prefix α then $\text{goto}(I, X)$ is set of valid items for prefix αX

Goto operation

If I is $\{ E' \rightarrow E. , E \rightarrow E. + T \}$ then
 $\text{goto}(I, +)$ is

$E \rightarrow E + .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Sets of items

C : Collection of sets of LR(0) items for
grammar G'

$C = \{ \text{closure} (\{ S' \rightarrow .S \}) \}$

repeat

 for each set of items I in C

 for each grammar symbol X

 if $\text{goto}(I, X)$ is not empty and not in C

 ADD $\text{goto}(I, X)$ to C

until no more additions to C

Example

Grammar:

$E' \rightarrow E$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

$I_0: \text{closure}(E' \rightarrow .E)$

$E' \rightarrow .E$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_1: \text{goto}(I_0, E)$

$E' \rightarrow E.$

$E \rightarrow E. + T$

$I_2: \text{goto}(I_0, T)$

$E \rightarrow T.$

$T \rightarrow T. * F$

$I_3: \text{goto}(I_0, F)$

$T \rightarrow F.$

$I_4: \text{goto}(I_0, ($

$F \rightarrow (.E)$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

$I_5: \text{goto}(I_0, id)$

$F \rightarrow id.$

$I_6: \text{goto}(I_1, +)$
 $E \rightarrow E + \cdot T$
 $T \rightarrow \cdot T * F$
 $T \rightarrow \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot \text{id}$

$I_9: \text{goto}(I_6, T)$
 $E \rightarrow E + T \cdot$
 $T \rightarrow T \cdot * F$

$\text{goto}(I_6, F)$ is I_3
 $\text{goto}(I_6, ())$ is I_4
 $\text{goto}(I_6, \text{id})$ is I_5

$I_7: \text{goto}(I_2, *)$
 $T \rightarrow T * \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot \text{id}$

$I_{10}: \text{goto}(I_7, F)$
 $T \rightarrow T * F \cdot$

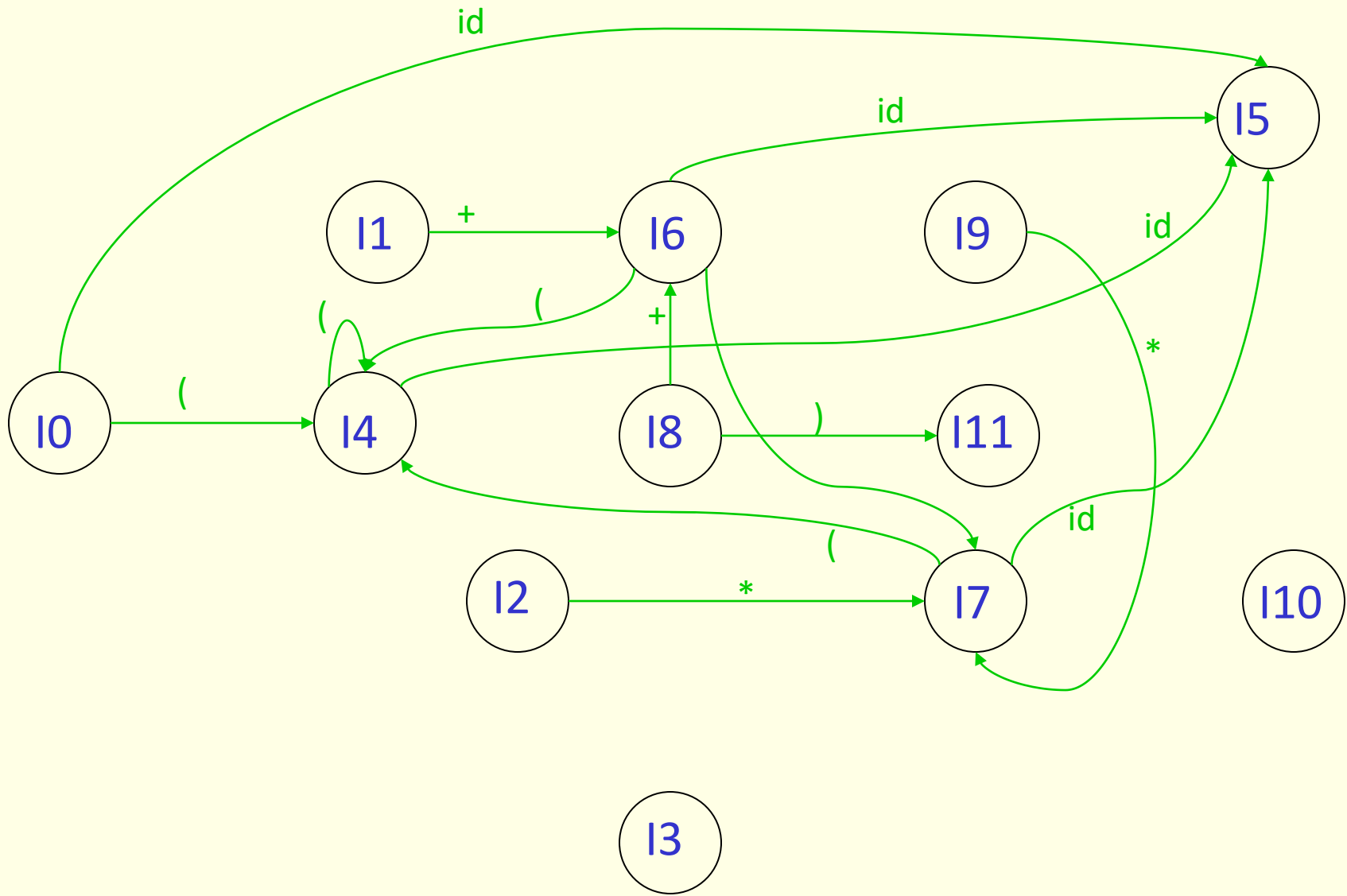
$\text{goto}(I_7, ())$ is I_4
 $\text{goto}(I_7, \text{id})$ is I_5

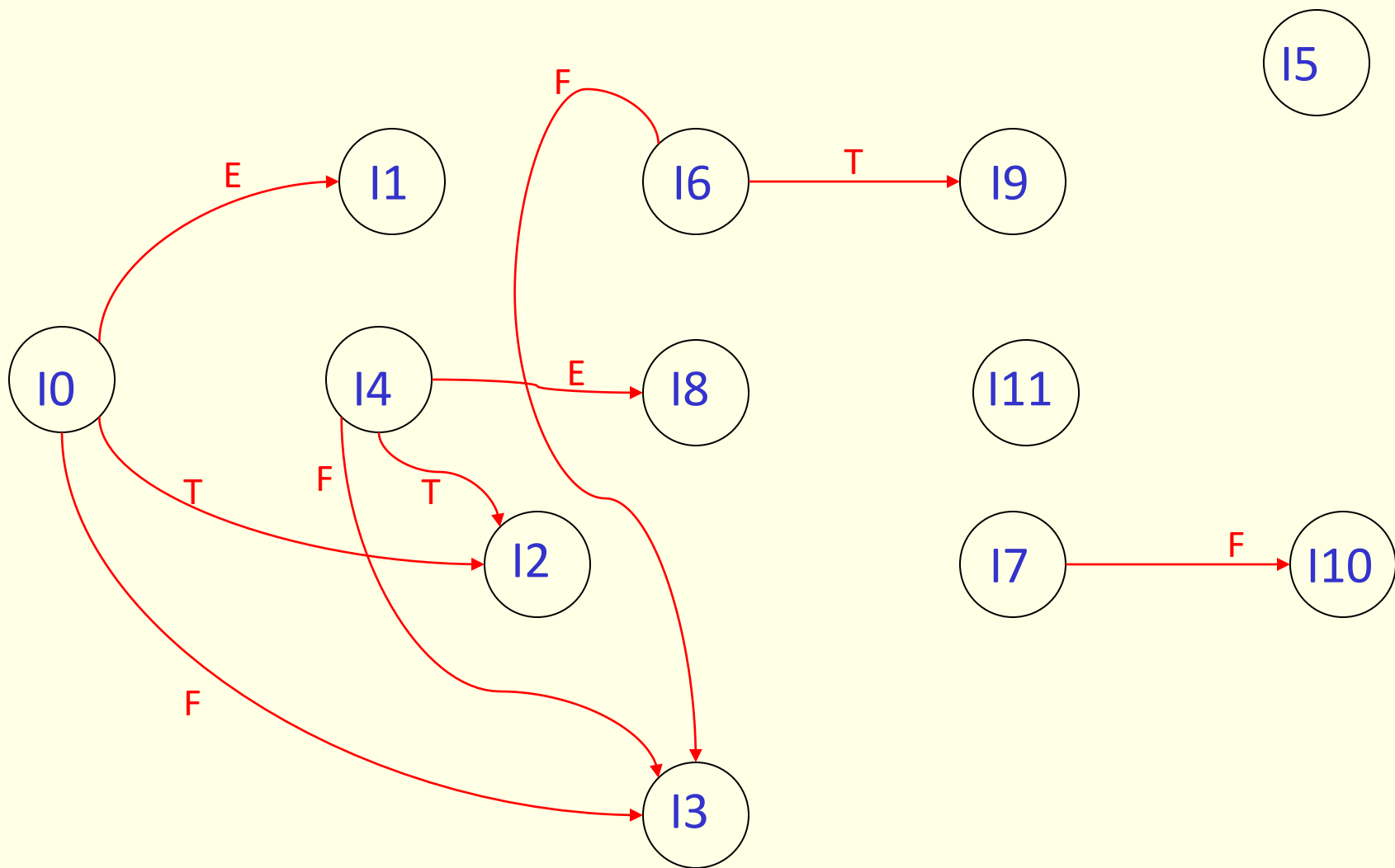
$I_8: \text{goto}(I_4, E)$
 $F \rightarrow (E \cdot)$
 $E \rightarrow E \cdot + T$

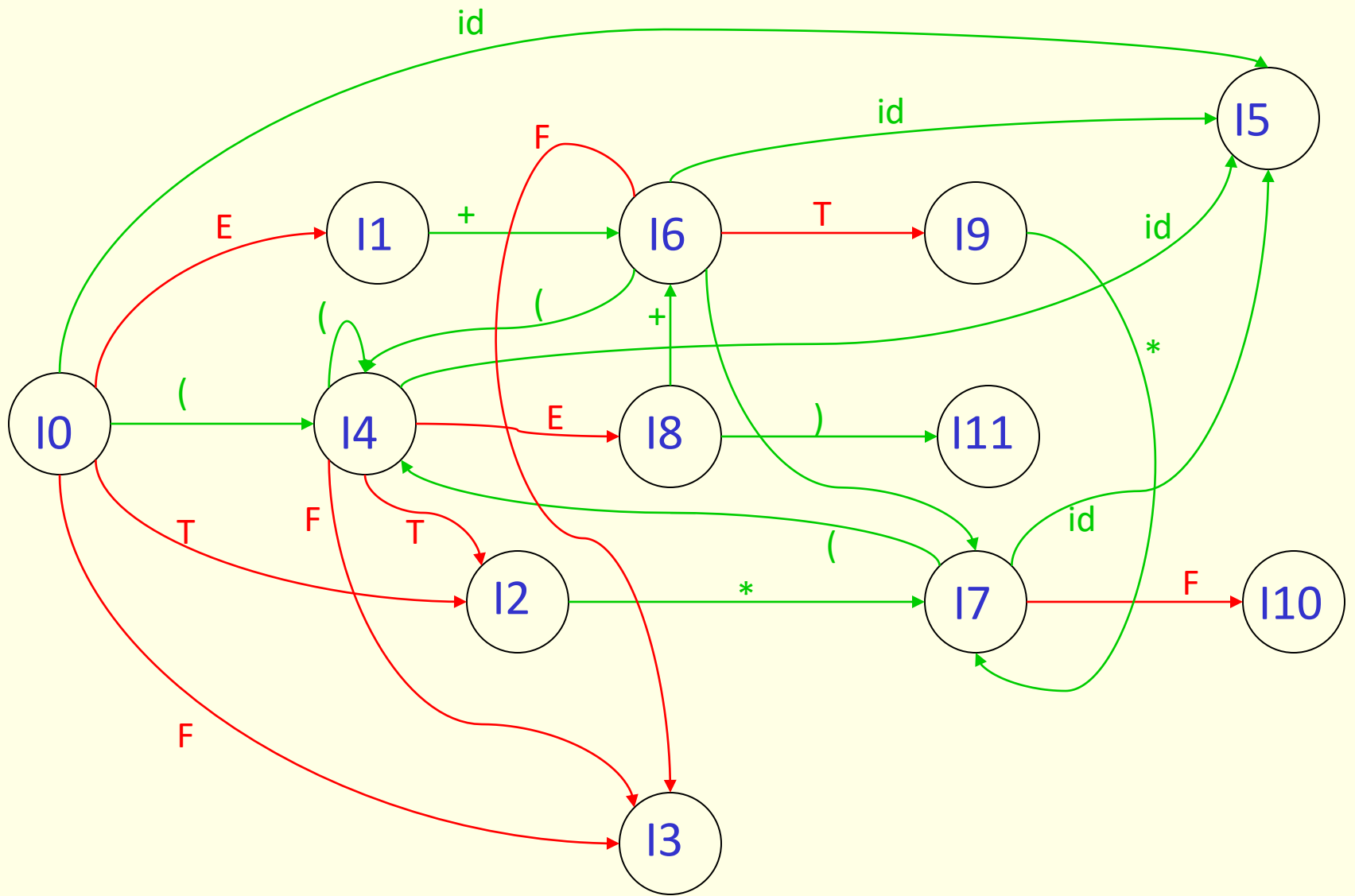
$I_{11}: \text{goto}(I_8,)$
 $F \rightarrow (E) \cdot$

$\text{goto}(I_4, T)$ is I_2
 $\text{goto}(I_4, F)$ is I_3
 $\text{goto}(I_4, ())$ is I_4
 $\text{goto}(I_4, \text{id})$ is I_5

$\text{goto}(I_8, +)$ is I_6
 $\text{goto}(I_9, *)$ is I_7







LR(0) (?) Parse Table

- The information is still not sufficient to help us resolve shift-reduce conflict.

For example the state:

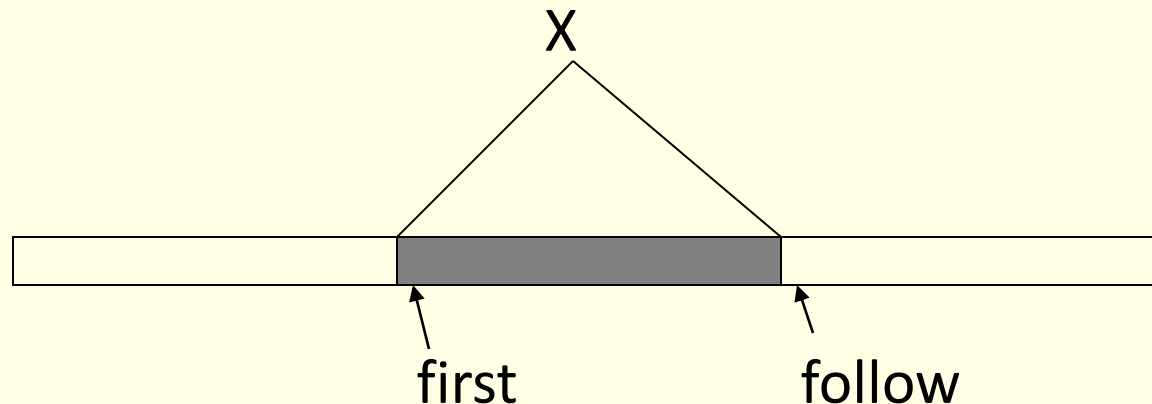
$$I_1: E' \rightarrow E.$$

$$E \rightarrow E. + T$$

- We need some more information to make decisions.

Constructing parse table

- **First(α)** for a string of terminals and non terminals α is
 - Set of symbols that might begin the fully expanded (made of only tokens) version of α
- **Follow(X)** for a non terminal X is
 - set of symbols that might follow the derivation of X in the input stream



Compute first sets

- If X is a terminal symbol then $\text{first}(X) = \{X\}$
- If $X \rightarrow \epsilon$ is a production then ϵ is in $\text{first}(X)$
- If X is a non terminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then
 - if for some i , a is in $\text{first}(Y_i)$
 - and ϵ is in all of $\text{first}(Y_j)$ (such that $j < i$)
 - then a is in $\text{first}(X)$
- If ϵ is in $\text{first}(Y_1) \dots \text{first}(Y_k)$ then ϵ is in $\text{first}(X)$
- Now generalize to a string α of terminals and non-terminals

Example

- For the expression grammar

$$E \rightarrow T E' \qquad E' \rightarrow +T E' \mid \epsilon$$

$$T \rightarrow F T' \qquad T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

$$\begin{aligned} \text{First}(E) &= \text{First}(T) = \text{First}(F) \\ &= \{ (, \text{id} \} \end{aligned}$$

$$\begin{aligned} \text{First}(E') \\ &= \{ +, \epsilon \} \end{aligned}$$

$$\begin{aligned} \text{First}(T') \\ &= \{ *, \epsilon \} \end{aligned}$$

Compute follow sets

1. Place $\$$ in $\text{follow}(S)$ // S is the start symbol
 2. If there is a production $A \rightarrow \alpha B \beta$
then everything in $\text{first}(\beta)$ (except ϵ) is in $\text{follow}(B)$
 3. If there is a production $A \rightarrow \alpha B \beta$ and $\text{first}(\beta)$ contains ϵ
then everything in $\text{follow}(A)$ is in $\text{follow}(B)$
 4. If there is a production $A \rightarrow \alpha B$
then everything in $\text{follow}(A)$ is in $\text{follow}(B)$
- Last two steps have to be repeated until the follow sets converge.

Example

- For the expression grammar

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

$$\text{follow}(E) = \text{follow}(E') = \{ \$,) \}$$

$$\text{follow}(T) = \text{follow}(T') = \{ \$,), + \}$$

$$\text{follow}(F) = \{ \$,), +, * \}$$

Construct SLR parse table

- Construct $C = \{I_0, \dots, I_n\}$ the collection of sets of LR(0) items
- If $A \rightarrow \alpha.a\beta$ is in I_i and $\text{goto}(I_i, a) = I_j$ then $\text{action}[i, a] = \text{shift } j$
- If $A \rightarrow \alpha.$ is in I_i then $\text{action}[i, a] = \text{reduce } A \rightarrow \alpha$ for all a in $\text{follow}(A)$
- If $S' \rightarrow S.$ is in I_i then $\text{action}[i, \$] = \text{accept}$
- If $\text{goto}(I_i, A) = I_j$ then $\text{goto}[i, A] = j$ for all non terminals A
- All entries not defined are errors

Notes

- This method of parsing is called SLR (Simple LR)
- LR parsers accept LR(k) languages
 - L stands for left to right scan of input
 - R stands for rightmost derivation
 - k stands for number of lookahead token
- SLR is the simplest of the LR parsing methods. SLR is too weak to handle most languages!
- If an SLR parse table for a grammar does not have multiple entries in any cell then the grammar is unambiguous
- All SLR grammars are unambiguous
- Are all unambiguous grammars in SLR?

Practice Assignment

Construct SLR parse table for following grammar

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

Show steps in parsing of string

$$9*5+(2+3*7)$$

- Steps to be followed
 - Augment the grammar
 - Construct set of LR(0) items
 - Construct the parse table
 - Show states of parser as the given string is parsed

Example

- Consider following grammar and its SLR parse table:

$S' \rightarrow S$

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

$I_1: \text{goto}(I_0, S)$

$S' \rightarrow S.$

$I_2: \text{goto}(I_0, L)$

$S \rightarrow L.=R$

$R \rightarrow L.$

$I_0: S' \rightarrow .S$

$S \rightarrow .L=R$

$S \rightarrow .R$

$L \rightarrow .*R$

$L \rightarrow .id$

$R \rightarrow .L$

Assignment (not to be submitted):
Construct rest of the items and the parse table.

SLR parse table for the grammar

| | = | * | id | \$ | S | L | R |
|---|-------|----|----|-----|---|---|---|
| 0 | | s4 | s5 | | 1 | 2 | 3 |
| 1 | | | | acc | | | |
| 2 | s6,r6 | | | r6 | | | |
| 3 | | | | r3 | | | |
| 4 | | s4 | s5 | | | 8 | 7 |
| 5 | r5 | | | r5 | | | |
| 6 | | s4 | s5 | | | 8 | 9 |
| 7 | r4 | | | r4 | | | |
| 8 | r6 | | | r6 | | | |
| 9 | | | | r2 | | | |

The table has multiple entries in action[2,=]

- There is both a shift and a reduce entry in action[2,=]. Therefore state 2 has a shift-reduce conflict on symbol "=", However, the grammar is not ambiguous.
- Parse id=id assuming reduce action is taken in [2,=]

| Stack | input | action |
|--------------|--------------|------------------------------|
| 0 | id=id | shift 5 |
| 0 id 5 | =id | reduce by $L \rightarrow id$ |
| 0 L 2 | =id | reduce by $R \rightarrow L$ |
| 0 R 3 | =id | error |

- if shift action is taken in [2,=]

| Stack | input | action |
|----------------|--------------|-----------------|
| 0 | id=id\$ | shift 5 |
| 0 id 5 | =id\$ | reduce by L→id |
| 0 L 2 | =id\$ | shift 6 |
| 0 L 2 = 6 | id\$ | shift 5 |
| 0 L 2 = 6 id 5 | \$ | reduce by L→id |
| 0 L 2 = 6 L 8 | \$ | reduce by R→L |
| 0 L 2 = 6 R 9 | \$ | reduce by S→L=R |
| 0 S 1 | \$ | ACCEPT |

Problems in SLR parsing

- No sentential form of this grammar can start with $R=...$
- However, the reduce action in action[2,=] generates a sentential form starting with $R=$
- Therefore, the reduce action is incorrect
- In SLR parsing method state i calls for reduction on symbol “a”, by rule $A \rightarrow \alpha$ if I_i contains $[A \rightarrow \alpha.]$ and “a” is in $\text{follow}(A)$
- However, when state I appears on the top of the stack, the viable prefix $\beta\alpha$ on the stack may be such that βA can not be followed by symbol “a” in any right sentential form
- Thus, the reduction by the rule $A \rightarrow \alpha$ on symbol “a” is invalid
- **SLR parsers cannot remember the left context**

Canonical LR Parsing

- Carry extra information in the state so that wrong reductions by $A \rightarrow \alpha$ will be ruled out
- Redefine LR items to include a terminal symbol as a second component (look ahead symbol)
- The general form of the item becomes $[A \rightarrow \alpha.\beta, a]$ which is called LR(1) item.
- Item $[A \rightarrow \alpha., a]$ calls for reduction only if next input is a . The set of symbols “ a ”s will be a subset of $\text{Follow}(A)$.

Closure(I)

repeat

 for each item $[A \rightarrow \alpha.B\beta, a]$ in I

 for each production $B \rightarrow \gamma$ in G'

 and for each terminal b in $\text{First}(\beta a)$

 add item $[B \rightarrow .\gamma, b]$ to I

until no more additions to I

Example

Consider the following grammar

$$S' \rightarrow S$$

$$S \rightarrow CC$$

$$C \rightarrow cC \mid d$$

Compute $\text{closure}(I)$ where $I = \{[S' \rightarrow .S, \$]\}$

| | |
|----------------------|------|
| $S' \rightarrow .S,$ | $\$$ |
| $S \rightarrow .CC,$ | $\$$ |
| $C \rightarrow .cC,$ | c |
| $C \rightarrow .cC,$ | d |
| $C \rightarrow .d,$ | c |
| $C \rightarrow .d,$ | d |

Example

Construct sets of LR(1) items for the grammar on previous slide

| | | | |
|----------------------------|-----|----------------------------|-----|
| $l_0: S' \rightarrow .S,$ | \$ | $l_4: \text{goto}(l_0, d)$ | |
| $S \rightarrow .CC,$ | \$ | $C \rightarrow d.,$ | c/d |
| $C \rightarrow .cC,$ | c/d | | |
| $C \rightarrow .d,$ | c/d | $l_5: \text{goto}(l_2, C)$ | |
| | | $S \rightarrow \bar{C}C.,$ | \$ |
| $l_1: \text{goto}(l_0, S)$ | | $l_6: \text{goto}(l_2, c)$ | |
| $S' \rightarrow S.,$ | \$ | $C \rightarrow c.C,$ | \$ |
| | | $C \rightarrow .cC,$ | \$ |
| $l_2: \text{goto}(l_0, C)$ | | $C \rightarrow .d,$ | \$ |
| $S \rightarrow C.C,$ | \$ | | |
| $C \rightarrow .cC,$ | \$ | $l_7: \text{goto}(l_2, d)$ | |
| $C \rightarrow .d,$ | \$ | $C \rightarrow d.,$ | \$ |
| | | | |
| $l_3: \text{goto}(l_0, c)$ | | $l_8: \text{goto}(l_3, C)$ | |
| $C \rightarrow c.C,$ | c/d | $C \rightarrow cC.,$ | c/d |
| $C \rightarrow .cC,$ | c/d | | |
| $C \rightarrow .d,$ | c/d | $l_9: \text{goto}(l_6, C)$ | |
| | | $C \rightarrow cC.,$ | \$ |

Construction of Canonical LR parse table

- Construct $C = \{I_0, \dots, I_n\}$ the sets of LR(1) items.
- If $[A \rightarrow \alpha.a\beta, b]$ is in I_i and $\text{goto}(I_i, a) = I_j$
then $\text{action}[i, a] = \text{shift } j$
- If $[A \rightarrow \alpha., a]$ is in I_i
then $\text{action}[i, a] = \text{reduce } A \rightarrow \alpha$
- If $[S' \rightarrow S., \$]$ is in I_i
then $\text{action}[i, \$] = \text{accept}$
- If $\text{goto}(I_i, A) = I_j$ then $\text{goto}[i, A] = j$ for all non terminals A

Parse table

| State | c | d | \$ | S | C |
|-------|----|----|-----|---|---|
| 0 | s3 | s4 | | 1 | 2 |
| 1 | | | acc | | |
| 2 | s6 | s7 | | | 5 |
| 3 | s3 | s4 | | | 8 |
| 4 | r3 | r3 | | | |
| 5 | | | r1 | | |
| 6 | s6 | s7 | | | 9 |
| 7 | | | r3 | | |
| 8 | r2 | r2 | | | |
| 9 | | | r2 | | |

Notes on Canonical LR Parser

- Consider the grammar discussed in the previous two slides. The language specified by the grammar is c^*dc^*d .
- When reading input $cc\dots dcc\dots d$ the parser shifts cs into stack and then goes into state 4 after reading d . It then calls for reduction by $C \rightarrow d$ if following symbol is c or d .
- IF $\$$ follows the first d then input string is c^*d which is not in the language; parser declares an error
- On an error canonical LR parser never makes a wrong shift/reduce move. It immediately declares an error
- **Problem:** Canonical LR parse table has a large number of states

LALR Parse table

- Look Ahead LR parsers
- Consider a pair of similar looking states (same kernel and different lookaheads) in the set of LR(1) items
 $I_4: C \rightarrow d. , c/d$ $I_7: C \rightarrow d., \$$
- Replace I_4 and I_7 by a new state I_{47} consisting of $(C \rightarrow d., c/d/\$)$
- Similarly I_3 & I_6 and I_8 & I_9 form pairs
- Merge LR(1) items having the same core

Construct LALR parse table

- Construct $C = \{I_0, \dots, I_n\}$ set of LR(1) items
- For each core present in LR(1) items find all sets having the same core and replace these sets by their union
- Let $C' = \{J_0, \dots, J_m\}$ be the resulting set of items
- Construct action table as was done earlier
- Let $J = I_1 \cup I_2 \dots \cup I_k$

since I_1, I_2, \dots, I_k have same core, $\text{goto}(J, X)$ will have the same core

Let $K = \text{goto}(I_1, X) \cup \text{goto}(I_2, X) \dots \cup \text{goto}(I_k, X)$ then $\text{goto}(J, X) = K$

LALR parse table ...

| State | c | d | \$ | S | C |
|-------|-----|-----|-----|---|----|
| 0 | s36 | s47 | | 1 | 2 |
| 1 | | | acc | | |
| 2 | s36 | s47 | | | 5 |
| 36 | s36 | s47 | | | 89 |
| 47 | r3 | r3 | r3 | | |
| 5 | | | r1 | | |
| 89 | r2 | r2 | r2 | | |

Notes on LALR parse table

- Modified parser behaves as original except that it will reduce $C \rightarrow d$ on inputs like ccd. The error will eventually be caught before any more symbols are shifted.
- In general core is a set of LR(0) items and LR(1) grammar may produce more than one set of items with the same core.
- Merging items never produces shift/reduce conflicts but may produce reduce/reduce conflicts.
- SLR and LALR parse tables have same number of states.

Notes on LALR parse table...

- Merging items may result into conflicts in LALR parsers which did not exist in LR parsers
- New conflicts can not be of shift reduce kind:
 - Assume there is a shift reduce conflict in some state of LALR parser with items $\{[X \rightarrow \alpha., a], [Y \rightarrow \gamma.a\beta, b]\}$
 - Then there must have been a state in the LR parser with the same core
 - Contradiction; because LR parser did not have conflicts
- LALR parser can have new reduce-reduce conflicts
 - Assume states $\{[X \rightarrow \alpha., a], [Y \rightarrow \beta., b]\}$ and $\{[X \rightarrow \alpha., b], [Y \rightarrow \beta., a]\}$
 - Merging the two states produces $\{[X \rightarrow \alpha., a/b], [Y \rightarrow \beta., a/b]\}$

Notes on LALR parse table...

- LALR parsers are not built by first making canonical LR parse tables
- There are direct, complicated but efficient algorithms to develop LALR parsers
- Relative power of various classes
 - $SLR(1) \leq LALR(1) \leq LR(1)$
 - $SLR(k) \leq LALR(k) \leq LR(k)$
 - $LL(k) \leq LR(k)$

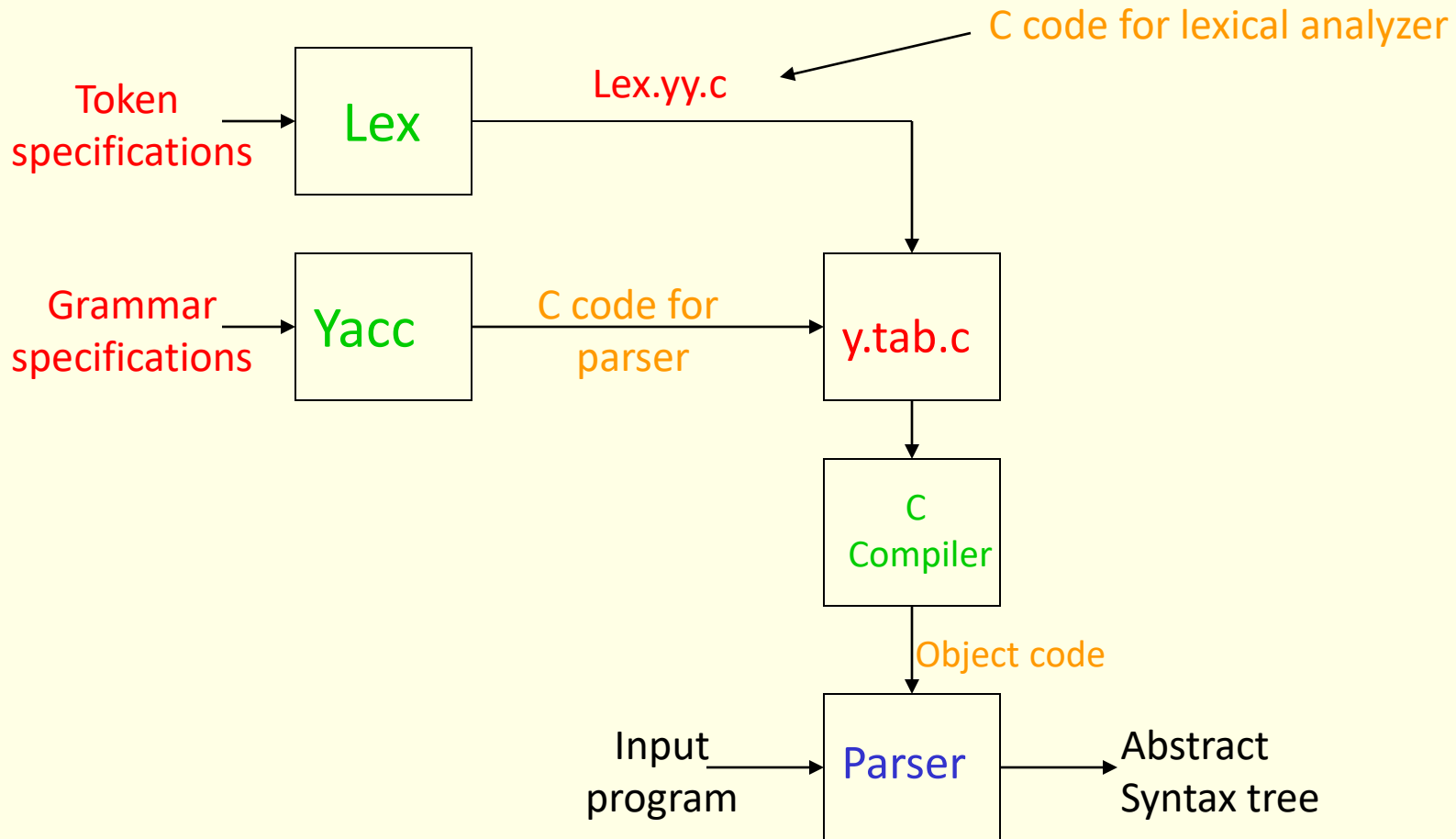
Error Recovery

- An error is detected when an entry in the action table is found to be empty.
- Panic mode error recovery can be implemented as follows:
 - scan down the stack until a state S with a goto on a particular nonterminal A is found.
 - discard zero or more input symbols until a symbol a is found that can legitimately follow A .
 - stack the state $\text{goto}[S,A]$ and resume parsing.
- **Choice of A:** Normally these are non terminals representing major program pieces such as an expression, statement or a block. For example if A is the nonterminal stmt , a might be semicolon or end.

Parser Generator

- Some common parser generators
 - YACC: **Y**et **A**nother **C**ompiler **C**ompiler
 - Bison: GNU Software
 - ANTLR: **A**nother **T**ool for **L**anguage **R**ecognition
- Yacc/Bison source program specification (accept LALR grammars)
 - declaration
 - %%
 - translation rules
 - %%
 - supporting C routines

Yacc and Lex schema



Refer to YACC Manual

Bottom up parsing ...

- A more powerful parsing technique
- LR grammars – more expensive than LL
- Can handle left recursive grammars
- Can handle virtually all the programming languages
- Natural expression of programming language syntax
- Automatic generation of parsers (Yacc, Bison etc.)
- Detects errors as soon as possible
- Allows better error recovery