Compiler Design

DR.PARKAVI.A,
CSE DEPT,
MSRIT

Top down parsing

- Derives left most derivation of string
- Recursive descent parsing
  - Back tracking
- Class of grammars for which predictive parsers can be constructed are called as LL(k) parsers
  - LL(1) parser

Recursive-Descent Parsing

void $A()$ {
    Choose an A-production, $A \rightarrow X_1 X_2 \cdots X_k$;
    for ($i = 1$ to k) {
        if (X_i is a nonterminal)
            call procedure $X_i()$;
        else if (X_i equals the current input symbol a)
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

A typical procedure for a nonterminal in a top-down parser

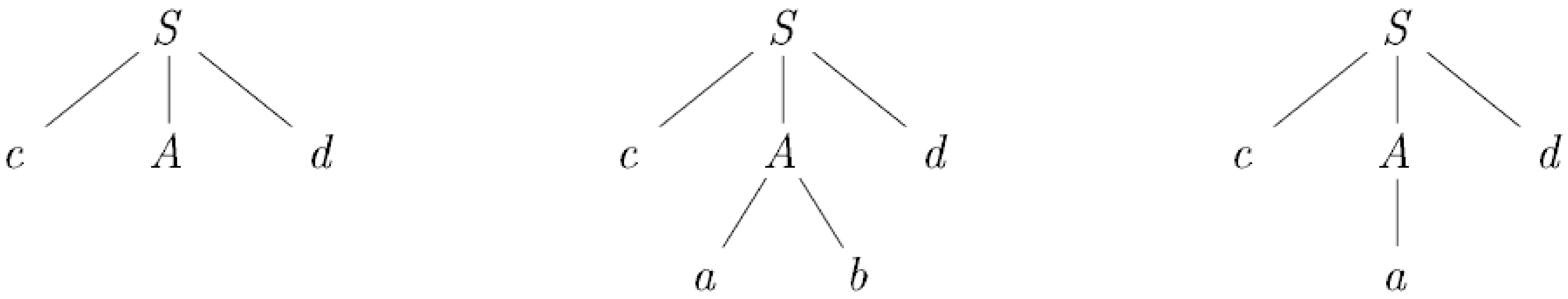
Construct top down parsing : Recursive descent parsing

Consider the grammar

$$
\begin{aligned}
S &\rightarrow c\ A\ d \\
A &\rightarrow a\ b \mid a
\end{aligned}
$$

- Input string: cad

Parsing



(a) (b) (c)

Tutorial Problem

- Construct recursive descent parser for the input string : id+id*id
- Using the given grammar:

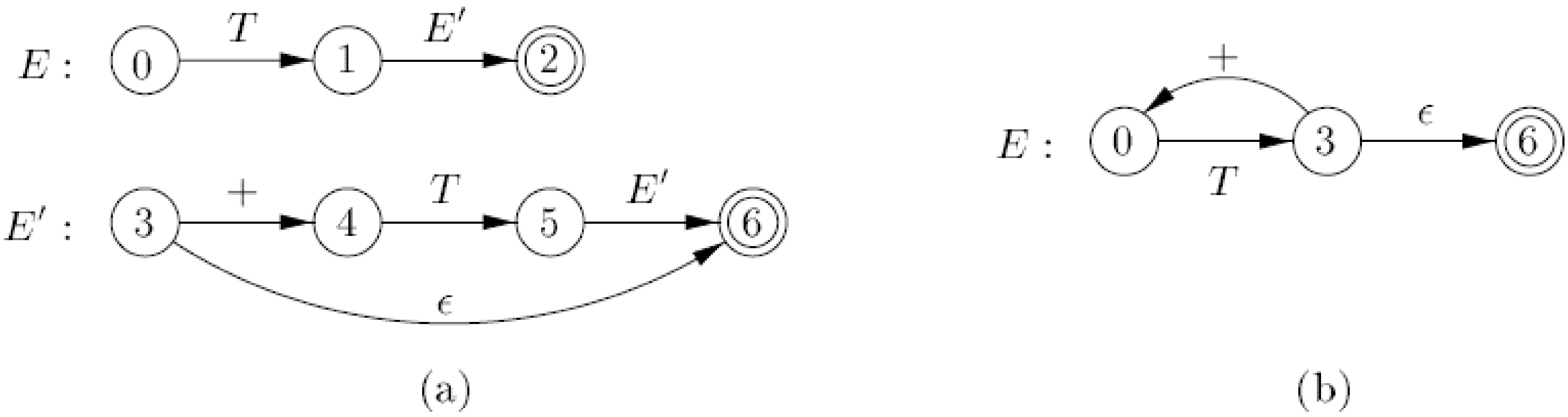
$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E'\ |\ \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T'\ |\ \epsilon \\
F & \rightarrow & (\ E\ )\ |\ \mathbf{id}
\end{array}
$$

LL(1) Parsers

- Top down parser
- No back tracking
- LL(1) grammars
- Left to right
- Look ahead 1
- Left most derivation

Example for transition diagram for the given grammar

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E'\ \mid\ \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T'\ \mid\ \epsilon \\
F & \rightarrow & (\ E\ )\ \mid\ \mathbf{id}
\end{array}
$$



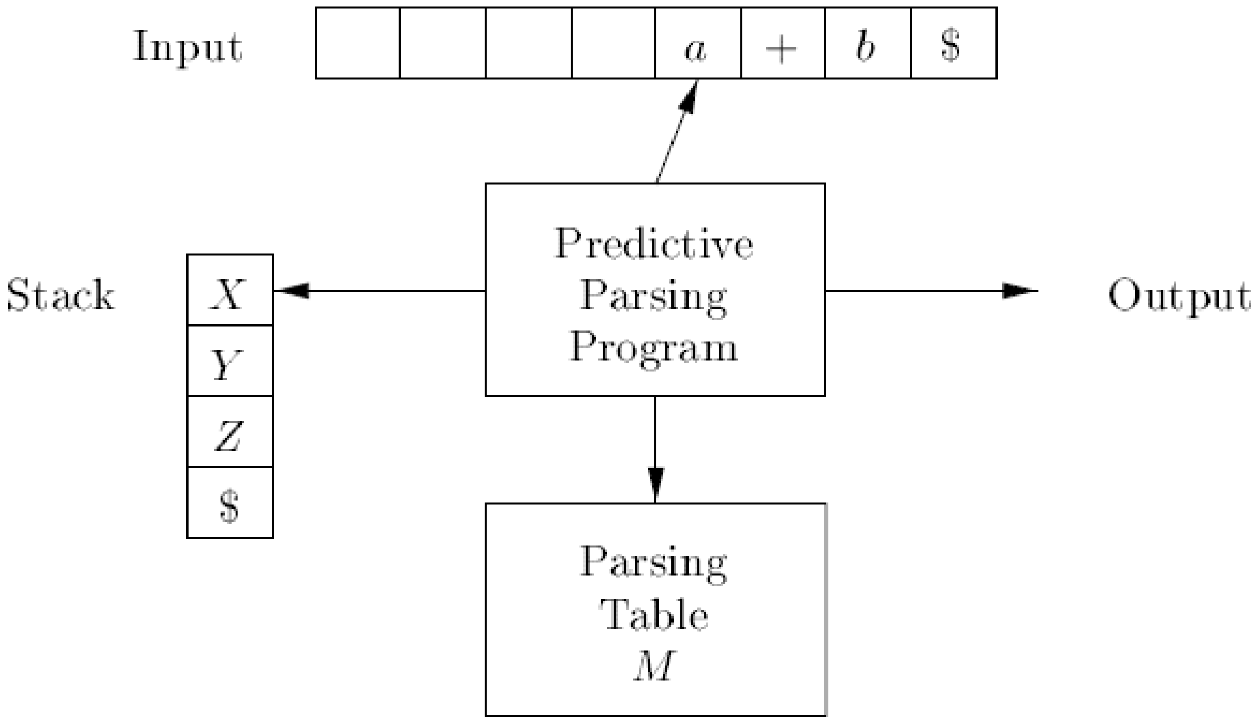
Transition diagrams for nonterminals E and E' of grammar

Predictive parsing algorithm

Algorithm Table-driven predictive parsing.

INPUT: A string w and a parsing table M for grammar G.

OUTPUT: If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.



Model of a table-driven predictive parser

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above $\$$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

```
let a be the first symbol of w;
let X be the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X = a ) pop the stack and let a be the next symbol of w;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1Y2 ··· Yk ) {
                output the production X → Y1Y2 ··· Yk;
                pop the stack;
                push Yk, Yk-1, ... , Y1 onto the stack, with Y1 on top;
        }
        let X be the top stack symbol;
}
```

Parse table

NON - TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

: Parsing table M

Parse the input strings

- Input strings
  - id+id
  - id+id*id

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to +\ TE'$
$\mathbf{id}\ +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id}\ +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id}\ +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$*\ FT'E'\$$	$* \mathbf{id}\$$	output $T' \to *\ FT'$
$\mathbf{id} + \mathbf{id}\ *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id}\ *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Parse table construction

Algorithm Construction of a predictive parsing table.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

First and Follow

NT	FIRST	FOLLOW
E	(,id	),$
E’	+, epcilon	),$
F	(,id	+,*,),$
T’	*,epcilon	+,),$
T	(,id	+,),$

First Computation

To compute $\text{FIRST}(X)$ for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.

2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \ldots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ, then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \overset{*}{\Rightarrow} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.

3. If $X \rightarrow \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

Follow Algorithm

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.

2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).

3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Problem1 : Check whether the given grammar is LL(1) or not

A -> a | B | C

B -> b | ε

C -> c e | d e

Problem 2: Construct Predictive parser and validate string for the given grammar

S−>A

A−>aB|Ad

B−>bBC|f

C→g

Construct Predictive parser for the following grammar . Show example parsing for error string and correct string

$E \rightarrow T\ X$

$X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow (\ E\ ) \mid int\ Y$

$Y \rightarrow *\ T \mid \varepsilon$

Synchronizing tokens

Non - terminal	Input Symbol					
	id	$+$	$*$	(	)	\$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$	synch	synch	$F \to (E)$	synch	synch

Synchronizing tokens added to the parsing table

Error recovery in Predictive parsing

- Panic mode
  - Synchronizing tokens
    - Follow(A)
    - FIRST(A)
    - A→ epsilon
    - If Terminal on top of stack , Pop it
- Phrase level
  - For blank entries in table, call error recovery routines for insert, delete , change characters

STACK	INPUT	REMARK
$E\,\$$	$)\,\mathbf{id} * + \mathbf{id}\,\$$	error, skip)
$E\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	**id** is in $\text{FIRST}(E)$
$TE'\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$FT'E'\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$\mathbf{id}\ T'E'\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$T'E'\,\$$	$* + \mathbf{id}\,\$$	
$* FT'E'\,\$$	$* + \mathbf{id}\,\$$	
$FT'E'\,\$$	$+ \mathbf{id}\,\$$	error, $M[F, +] = \text{synch}$
$T'E'\,\$$	$+ \mathbf{id}\,\$$	F has been popped
$E'\,\$$	$+ \mathbf{id}\,\$$	
$+ TE'\,\$$	$+ \mathbf{id}\,\$$	
$TE'\,\$$	$\mathbf{id}\,\$$	
$FT'E'\,\$$	$\mathbf{id}\,\$$	
$\mathbf{id}\,T'E'\,\$$	$\mathbf{id}\,\$$	
$T'E'\,\$$	$\$$	
$E'\,\$$	$\$$	
$\$$	$\$$	

Bottom up Parsing

- SR parsers
- Reduction
- Handle pruning

Example Grammar

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T \,/\, F \mid F \\
F & \rightarrow & (\, E \,) \mid \mathbf{id}
\end{array}
$$

Example of shift reduce parsing

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \,\$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \,\$$	reduce by $F \rightarrow \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \,\$$	reduce by $T \rightarrow F$
$\$\,T$	$* \mathbf{id}_2 \,\$$	shift
$\$\,T *$	$\mathbf{id}_2 \,\$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$\,T$	$\$$	reduce by $E \rightarrow T$
$\$\,E$	$\$$	accept

Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

Conflicts in shift reduce parser

- Shift shift conflict
- Reduce shift conflict

LR Parsing

DR PARKAVI A

Viable prefixes

- Shift reduce parser
- Stack contents mu

$$E \underset{rm}{\overset{*}{\Rightarrow}} F * \mathbf{id} \underset{rm}{\Rightarrow} (E) * \mathbf{id}$$

Then, at various times during the parse, the stack will hold (, $(E$, and (E), but it must not hold $(E)*$, since (E) is a handle, which the parser must reduce to F before shifting $*$.

Continued…

The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called *viable prefixes*. They are defined as follows: a viable prefix is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form.

SLR parsing is based on the fact that LR(0) automata recognize viable prefixes. We say item $A \to \beta_1 \cdot \beta_2$ is *valid* for a viable prefix $\alpha\beta_1$ if there is a derivation $S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$. In general, an item will be valid for many viable prefixes.

Example Let us consider the augmented expression grammar again, whose sets of items and GOTO function are exhibited Clearly, the string $E + T*$ is a viable prefix of the grammar. The automaton will be in state 7 after having read $E + T*$. State 7 contains the items

$$
\begin{aligned}
T &\to T * \cdot F \\
F &\to \cdot(E) \\
F &\to \cdot\mathbf{id}
\end{aligned}
$$

which are precisely the items valid for $E+T*$. To see why, consider the following three rightmost derivations

$$
\begin{aligned}
E' &\underset{rm}{\Rightarrow} E \\
&\underset{rm}{\Rightarrow} E + T \\
&\underset{rm}{\Rightarrow} E + T * F
\end{aligned}
\qquad
\begin{aligned}
E' &\underset{rm}{\Rightarrow} E \\
&\underset{rm}{\Rightarrow} E + T \\
&\underset{rm}{\Rightarrow} E + T * F \\
&\underset{rm}{\Rightarrow} E + T * (E)
\end{aligned}
\qquad
\begin{aligned}
E' &\underset{rm}{\Rightarrow} E \\
&\underset{rm}{\Rightarrow} E + T \\
&\underset{rm}{\Rightarrow} E + T * F \\
&\underset{rm}{\Rightarrow} E + T * \mathbf{id}
\end{aligned}
$$

More powerful LR parsers

- Uses look aheads
  - LR(1) items
- Lookahead LR (LALR)
  - Lookaheads are introduced

Canonical LR(1) Items

The general form of an item becomes $[A \to \alpha \cdot \beta, a]$, where $A \to \alpha\beta$ is a production and a is a terminal or the right endmarker \$. We call such an object an *LR(1) item*.

Viable prefix

Formally, we say LR(1) item $[A \to \alpha{\cdot}\beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \underset{rm}{\overset{*}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$, where

1. $\gamma = \delta\alpha$, and

2. Either a is the first symbol of w, or w is ϵ and a is \$.

Viable prefix

$$S \rightarrow B\ B$$
$$B \rightarrow a\ B \ \mid\ b$$

There is a rightmost derivation $S \underset{rm}{\overset{*}{\Rightarrow}} aaBab \underset{rm}{\Rightarrow} aaaBab$. We see that item $[B \rightarrow a{\cdot}B, a]$ is valid for a viable prefix $\gamma = aaa$ by letting $\delta = aa$, $A = B$, $w = ab$, $\alpha = a$, and $\beta = B$ in the above definition. There is also a rightmost derivation $S \underset{rm}{\overset{*}{\Rightarrow}} BaB \underset{rm}{\Rightarrow} BaaB$. From this derivation we see that item $[B \rightarrow a{\cdot}B, \$]$ is valid for viable prefix Baa. $\square$

CLR Parser

Constructing LR(1) sets of items

SetOfItems $\text{CLOSURE}(I)$ {
repeat
for (each item $[A \rightarrow \alpha \cdot B\beta, a]$ in I)
for (each production $B \rightarrow \gamma$ in G')
for (each terminal b in $\text{FIRST}(\beta a)$)
add $[B \rightarrow \cdot\gamma, b]$ to set I;
until no more items are added to I;
return I;
}

SetOfItems $\text{GOTO}(I, X)$ {
initialize J to be the empty set;
for (each item $[A \rightarrow \alpha \cdot X\beta, a]$ in I)
add item $[A \rightarrow \alpha X \cdot \beta, a]$ to set J;
return $\text{CLOSURE}(J)$;
}

Continued…

```
void items(G') {
    initialize C to {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 sets of items are added to C;
}
```

Algorithm Construction of the sets of LR(1) items.

INPUT: An augmented grammar G'.

OUTPUT: The sets of LR(1) items that are the set of items valid for one or more viable prefixes of G'.

METHOD: The procedures CLOSURE and GOTO and the main routine *items* for constructing the sets of items were shown

Example G:

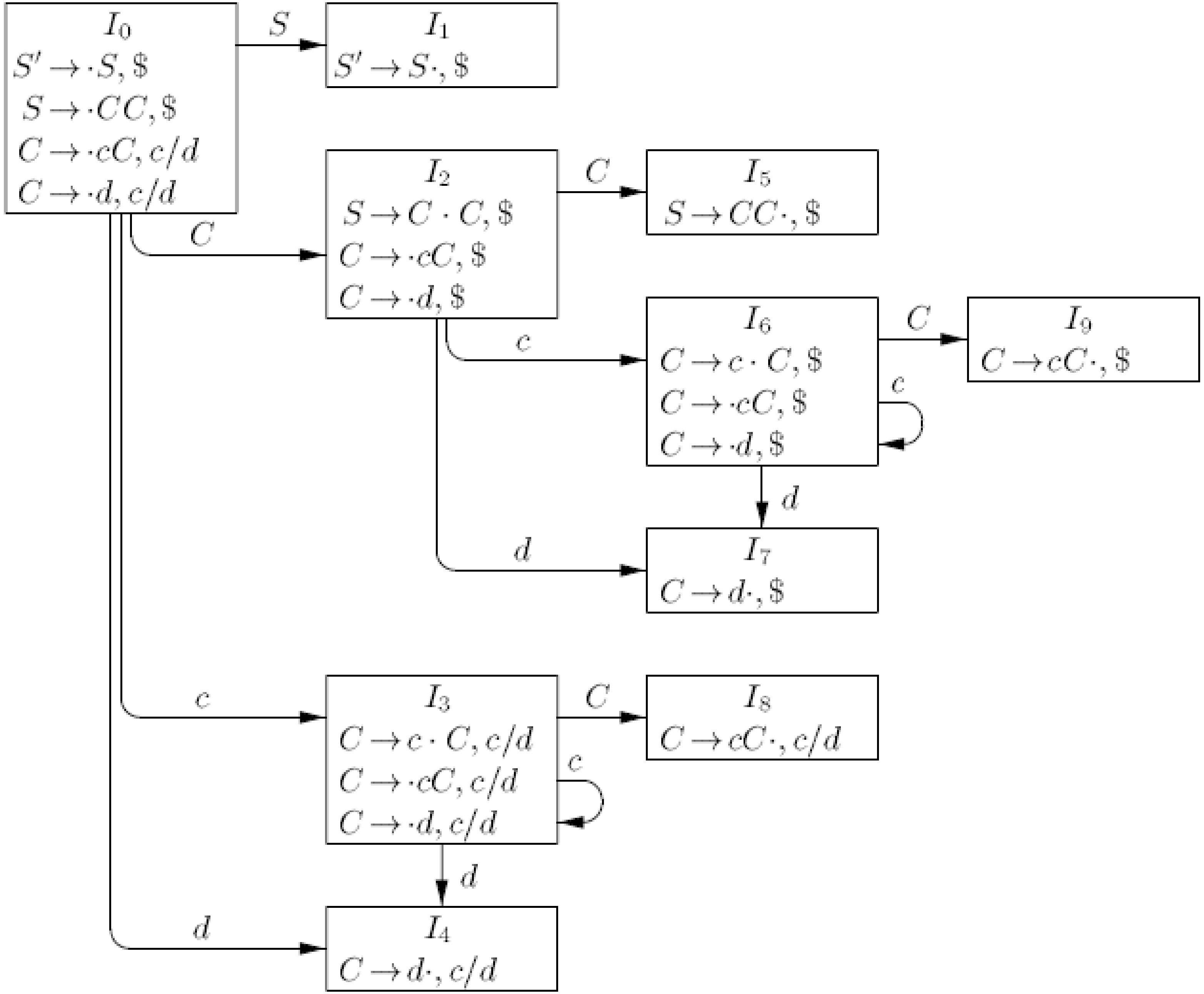
$$\begin{array}{lcl} S & \rightarrow & C\ C \\ C & \rightarrow & c\ C \ \mid \ d \end{array}$$

G` Augmented Grammar

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & C\ C \\ C & \rightarrow & c\ C \mid d \end{array}$$

Find closure { S’→S,$}

LR(1) Automaton


I_0
$S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$
S
I_1
$S' \to S\cdot, \$$
C
I_2
$S \to C \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$
C
I_5
$S \to CC\cdot, \$$
c
I_6
$C \to c \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$
C
I_9
$C \to cC\cdot, \$$
c
d
d
I_7
$C \to d\cdot, \$$
c
I_3
$C \to c \cdot C, c/d$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$
C
I_8
$C \to cC\cdot, c/d$
c
d
d
I_4
$C \to d\cdot, c/d$

CLR Parsing Table Construction

Algorithm : Construction of canonical-LR parsing tables.

INPUT: An augmented grammar G'.

OUTPUT: The canonical-LR parsing table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C' = \{I_0, I_1, \cdots, I_n\}$, the collection of sets of LR(1) items for G'.
2. State i of the parser is constructed from I_i. The parsing action for state i is determined as follows.
   (a) If $[A \rightarrow \alpha{\cdot}a\beta, b]$ is in I_i and GOTO$(I_i, a) = I_j$, then set ACTION$[i, a]$ to "shift j." Here a must be a terminal.
   (b) If $[A \rightarrow \alpha\cdot, a]$ is in I_i, $A \neq S'$, then set ACTION$[i, a]$ to "reduce $A \rightarrow \alpha$."
   (c) If $[S' \rightarrow S\cdot, \$]$ is in I_i, then set ACTION$[i, \$]$ to "accept."

If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.

Continued

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error."

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S, \$]$.

Continued…

- If multiple entries in a cell, in the parse table, then the given grammar is not CLR grammar or Not a LR(1) grammar.

State	ACTION			GOTO	
	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		

CLR Parsing Home work Tutorial

- It is same as SLR Parsing
- Same algorithm of parsing , has to be followed for all LR Parsers

Eg:

- Parse input : “cd”

using the given grammar of CLR parser. Use CLR parse table.

Algorithm : An easy, but space-consuming LALR table construction.

INPUT: An augmented grammar G'.

OUTPUT: The LALR parsing-table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.

2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

3. Let $C' = \{J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in Algorithm of CLR. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cup I_2 \cup \cdots \cup I_k$, then the cores of GOTO$(I_1, X)$, GOTO$(I_2, X), \ldots,$ GOTO(I_k, X) are the same, since $I_1, I_2, \ldots, I_k$ all have the same core. Let K be the union of all sets of items having the same core as GOTO(I_1, X). Then GOTO$(J, X) = K$.