COMPILER DESIGN

UNIT-2

Syntax Analysis

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

PARSER

- **Parsing:** determining if a string of tokens can be generated by a grammar or not

- **Parser / Syntax Analyser:**
  - input: grammar G, string w
  - output: parse tree for w if it belongs to the language, otherwise error

- Input: left to right

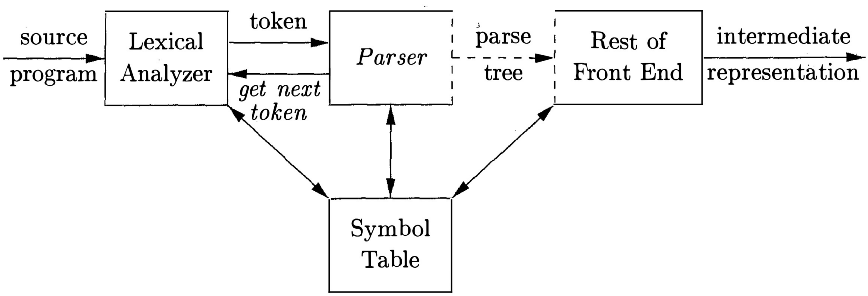


Figure 4.1: Position of parser in compiler model

Role of the Parser

- Validate syntax and report syntax errors
- construct parse tree and pass to semantic analyser
- Store info in symbol table about received tokens

Error Messages

- Report the presence of errors clearly and accurately. localise, pinpoint line, no duplicates/redundance
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.

Error Recovery Strategies

1. Panic mode recovery
2. Phrase level recovery
3. Error Productions
4. Global corrections

1. Panic Mode Recovery

- On discovering error, parser discards input symbols one at a time
- Until a synchronising token is found
- Usually delimiters: ; , }
- Based on source language
- Often skips amount of input and may miss additional errors
- Simplicity, guaranteed not to go in infinite loop

2. Phrase Level Recovery

- On discovery, parser performs local correction on the remaining input to make it syntactically correct
- Parser continues after that
- Typical corrections:
  - replace , with ;
  - delete extraneous ;
  - insert missing ;
- Weary of infinite loops
  - insert something on the input ahead of the current
- Used in error-repairing compilers
- Require intelligent behaviour
- Compiler must suggest

3. Error Productions

- Anticipate common errors and augment grammar with productions that generate errors
- Eg

$S \longrightarrow$ if (cond) {S} else {S} valid if-else

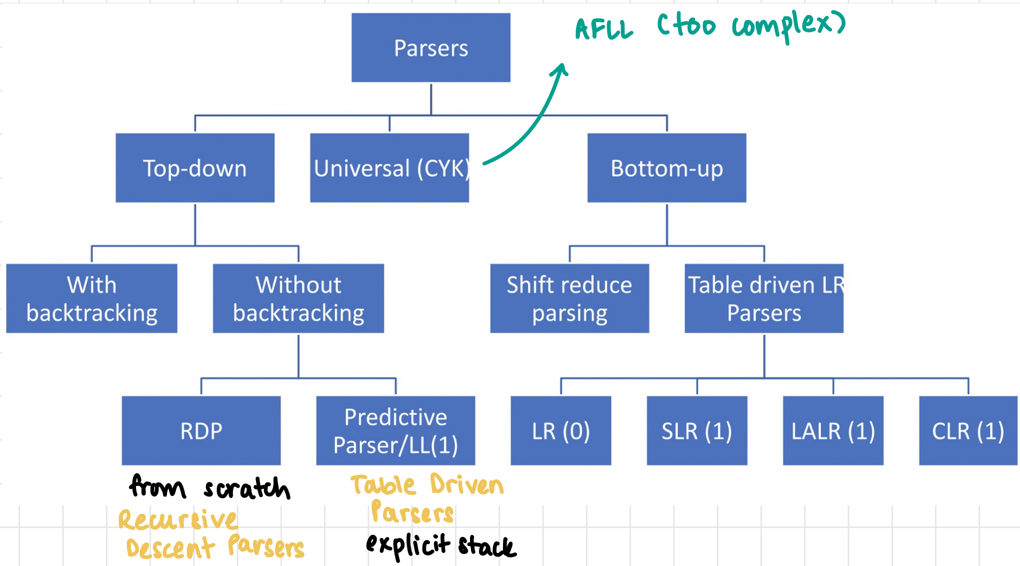
$S \longrightarrow$ fi (cond) {S} else {S} errrr production

- Complicates the grammar

4. Global Correction

- Theoretic concept (very hard to implement)
- Given an incorrect sequence x and a grammar G, find a parse tree for a string y such that x can be changed to y with minimal changes
- May not be what programmer intended

TYPES of PARERS



TOP DOWN PARSING

- Start from root and create nodes in preorder (find leftmost derivation)
- Alternatively, finding left most derivation for an input string
- Types
  1. With backtracking (read input multiple times)
  2. Without backtracking (no going back)
- Key problem: determine production to be applied for the non-terminal

Implementation of TDP

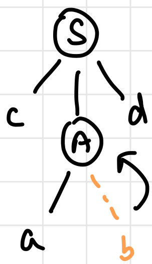
implementation
- Recursive (RDP) — use OS stack
- Non-recursive (table-driven) — explicit stack

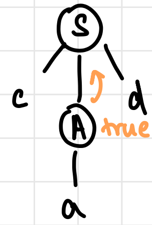
TDP with Backtracking

1. Try out all alternatives in the order in which they are listed
2. RDP implementation - Recursive descent parsing
   - Write a function for each non-terminal
3. No TDP works with left recursive grammars (direct or indirect / circular)

Q: Parse with TDP with backtracking

S → cAd   w= cad
A → ab | a





unread a

S → c A d
A → a b | a

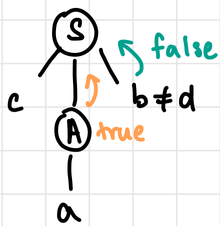
```
S() {
        if(inputSymbol++ == 'c')
        {
            if(A())
            {
                if(inputSymbol++ == 'd')
                {
                    return true;
                }
            }
        }
    return false;
}
```

```
A() {
        isave = inputPointer();
        if(inputSymbol++ == 'a')
        {
            if(inputSymbol++ == 'b') {
                return true;
            }
        }
        inputSymbol = isave;
        if(inputSymbol++ == 'a') {
            return true;
        }
    return false;
}
```

Problem

S → cAd   w = cab will give parsing unsuccessful
A → a | ab

order swapped



unread a

Q: $E \rightarrow E + T \mid T$

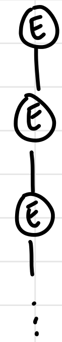
$T \rightarrow T * id \mid id$

Write pseudocode for E and parse id * id

```
E() {
    if (E()) {
        if (inputSymbol++ == '+') {
            if (T()) {
                return true;
            }
        }
    }

    else if (T()) {
        return true;
    }

    return false;
}
```



Infinite loop

Left-recursive grammars

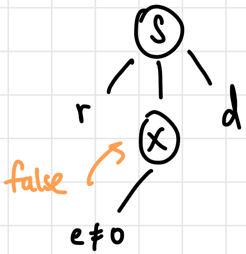
$A \rightarrow A\alpha$

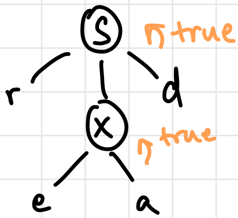
- Top-down parsers do not work with left recursive grammars

Q: Perform RDP with backtracking

$S \rightarrow rXd \mid rZd$
$X \rightarrow oa \mid ea$
$Z \rightarrow ai$

w = read





Q:

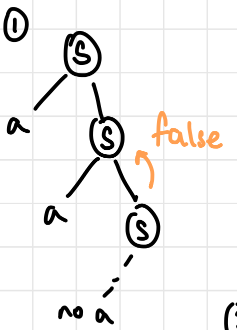
$S \rightarrow aSa \mid aa$ vs $S \rightarrow aa \mid aSa$

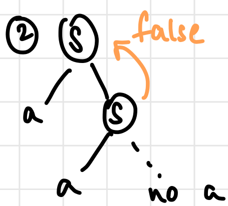
↑
have to specify as this

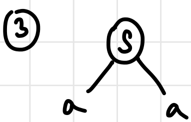
w = aaaa

input not empty
parse error

w = aa

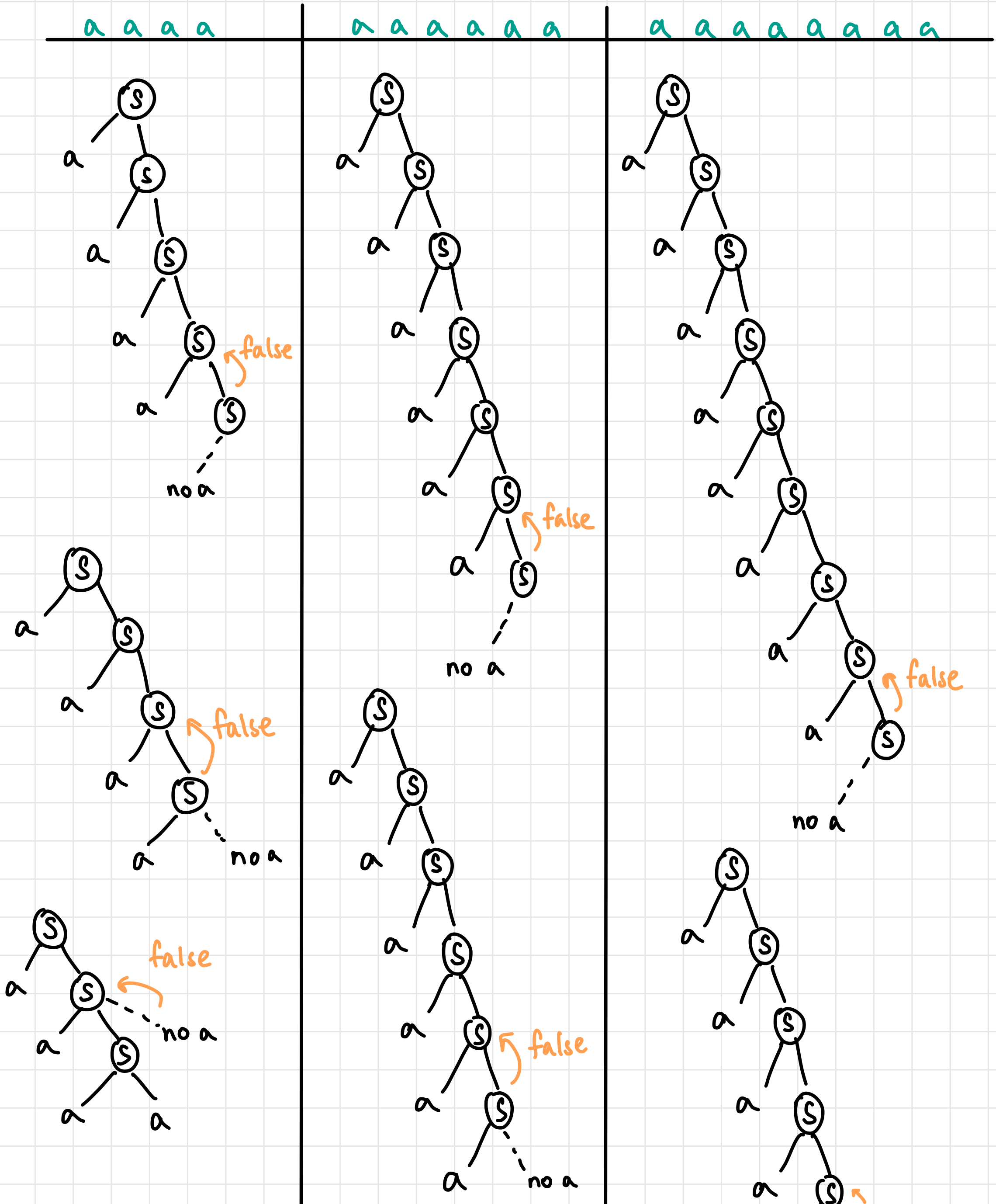


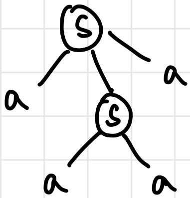




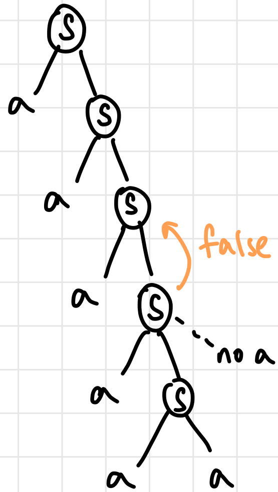
very long process

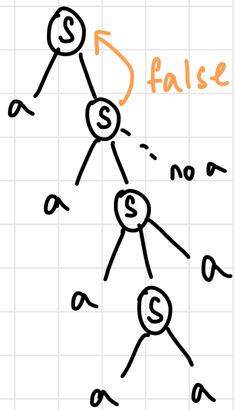
Q: G: $S \to aSa \mid aa$, accepts $L(G) = \{a^{2n}, n \geq 1\}$

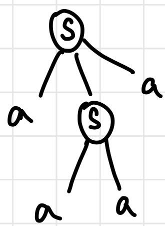




aaaa accepted

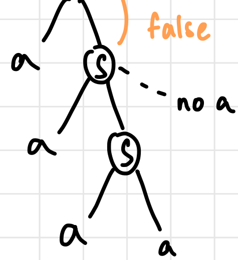


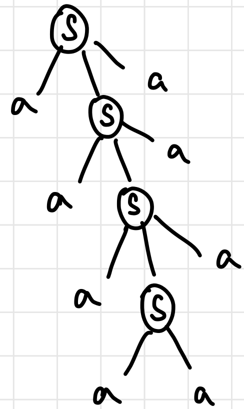




only aaaa parsed

aaaaaa not accepted





Drawbacks of RDP with Backtracking

1. Parser may not accept valid inputs because of the order of productions

2. No left recursive grammars (infinite loop)

3. Reversing semantic actions during parsing is an overhead

TDP Without Backtracking

1. Eliminate left recursion

2. Left factor grammar
   (common prefix)

$$S \to abA \mid acA \mid adA \mid by$$
$$A \to x \mid y$$

$$\downarrow$$

$$S \to aM \mid by$$
$$M \to bA \mid cA \mid dA$$
$$A \to x \mid y$$

Q: Left factor the grammar

$$S \to iCtS \mid iCtSeS \mid a$$
$$C \to b$$

$$S \to iCtSX \mid a$$
$$X \to \lambda \mid eS$$
$$C \to b$$

Eliminate Left Recursion

(1) Direct / Immediate

$A \to A\alpha$

$\alpha \in (V \cup T)^*$ → terminals

↓ non terminals

- Eg: $E \to E + T$
  $T \to T * F$

(2) Indirect

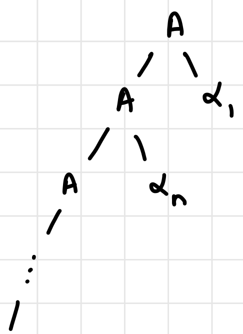
$S \to Aa$

$A \to Sd \mid c$

- Recursion in one or more steps

Eliminating Direct Left Recursion

$$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_s$$



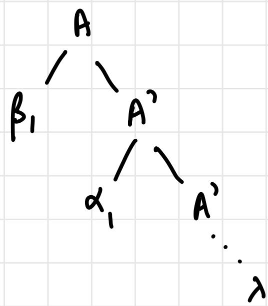
needs to end in a β eventually

string looks like $\beta_i \alpha_j \dots \alpha_k \alpha_l$

Same language as

$$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_s A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \lambda$$



Q: Eliminate left recursion and derive id * id with both grammars

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to id \mid num \mid (E)$$

① $\underbrace{E}_{A} \to \underbrace{E}_{A}\underbrace{+T}_{\alpha} \mid \underbrace{T}_{\beta}$

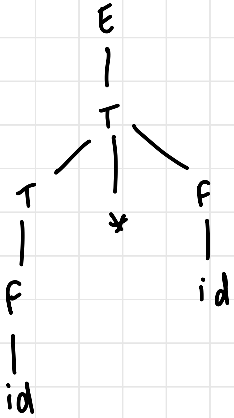
$$E \to T E'$$
$$E' \to + T E' \mid \lambda$$

② $\underbrace{T}_{A} \to \underbrace{T}_{A}\underbrace{*F}_{\alpha} \mid \underbrace{F}_{\beta}$

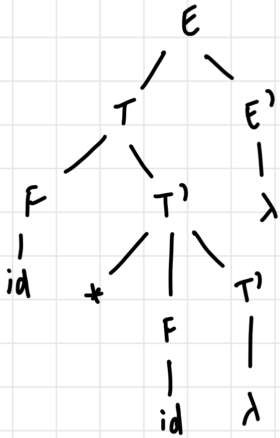
$$T \to F T'$$
$$T' \to * F T' \mid \lambda$$

③ $F \to id \mid num \mid (E)$

Grammar 1



Grammar 2



Q: Is the grammar suitable for TDP?

$S \rightarrow aAcB$
$A \rightarrow Ab \mid b \mid bc$
$B \rightarrow d$

No, ∵ rule #2 is left recursive ⇒ grammar is left recursive

We also want to avoid backtracking, no common prefix should exist (but here the grammar is not left factored)

No sequence of eliminating left recursion and left factoring

(a) Left factor

$S \rightarrow aAcB$
$A \rightarrow \underbrace{Ab}_{\alpha} \mid \underbrace{bX}_{\beta} \mid$
$X \rightarrow c \mid \lambda$
$B \rightarrow d$

(b) Eliminate left recursim

$S \to a A c B$
$A \to b X A'$
$A' \to b A' \mid \lambda$
$X \to c \mid \lambda$
$B \to d$

Eliminating Indirect Left Recursion

$S \to Aa \mid b$
$A \to Ac \mid Sd \mid e$

① $S \Rightarrow Aa \Rightarrow Sda$

② $A \Rightarrow Sd \Rightarrow Aad$

- Algorithm to eliminate left recursion requires that the non-terminals be ordered
- Use order in which they are listed
- If a production includes a non-terminal that has already been seen, replace it with its productions

① $S \to Aa \mid b$ ✓

② $A \to Ac \mid \underline{S}d \mid e$

seen before

$A \to A\underbrace{c}_{\alpha_1} \mid A\underbrace{ad}_{\alpha_2} \mid \underbrace{bd}_{\beta_1} \mid \underbrace{e}_{\beta_2}$

- Next step, eliminate left recursion for direct recursive grammars

① $S \to Aa \mid b$

② $A \to bdA' \mid eA'$

$A' \to cA' \mid adA' \mid \lambda$

Algorithm

1. Order NT's

   $A_1, A_2, \ldots, A_n$

2. for $i=1$ to n

   for $j=1$ to $i-1$

   Replace each prod of the form

   $A_i \to A_j \gamma$

   By

   $A_i \to \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$

   Where

   $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$

3. Eliminate immediate LR

   $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_s$

   Replace with

   $A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_s A'$
   $A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \lambda$

Q: Eliminate left recursion

$A \to Bxy \mid x$
$B \to CD$
$C \to A \mid c$
$D \to d$

1. Order NT's
   A, B, C, D

2. Replace indirect recursion

(1) $A \to Bxy \mid x$
(2) $B \to CD$
(3) $C \to Bxy \mid x \mid c$
$C \to CDxy \mid x \mid c$
(4) $D \to d$

3. Replace direct

(1) $A \to Bxy \mid x$
(2) $B \to CD$
(3) $C \to C\underbrace{Dxy}_{\alpha} \mid \underbrace{x}_{\beta_1} \mid \underbrace{c}_{\beta_2}$

$C \to xC' \mid cC'$
$C' \to DxyC' \mid \lambda$

(4) $D \to d$

$A \to Bxy \mid x$
$B \to CD$
$C \to xC' \mid cC'$
$C' \to DxyC' \mid \lambda$
$D \to d$

Implementing a Parser

1. RDP implementation — each NT is a function

2. Table-driven parser / predictive paring

1. RDP Implementation in C

consider the grammar

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \lambda$
$F \rightarrow id \mid num \mid (E)$

$ marks end of string

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

char input[10];
int i, error;

void E();
void Edash();
void T();
void Tdash();
void F();

void fail()
{
    printf("Error\n");
    exit(0);
}

int main()
{
    i = 0;
    error = 0;
    printf("Enter an arithmetic expression:  "); // Eg: a+a*a$
    scanf("%s", input);
    E();
    if (input[i] == '$' && error == 0)
        printf("\nAccepted..!!!\n");
    else
        printf("\nRejected..!!!\n");
}
```

```c
// E -> TE'
void E()
{
    T();
    Edash();
}

// E' -> +TE' | epsilon
void Edash()
{
    if (input[i] == '+')
    {
        i++;
        T();
        Edash();
    }
}

// T -> FT'
void T()
{
    F();
    Tdash();
}

// T' -> *FT' | epsilon
void Tdash()
{
    if (input[i] == '*')
    {
        i++;
        F();
        Tdash();
    }
}

// F -> id|num|(E)
void F()
{
    if (isalnum(input[i]))
        i++;
    else if (input[i] == '(')
    {
        i++;
        E();
        if (input[i] == ')')
            i++;
        else
            fail();
    }
    else
        fail();
}
```

- Go through the code

```
Enter an arithmetic expression:  (a)+(a*(a))$
Accepted..!!!
```

PREDICTIVE PARSING

- Parsers that choose between productions by looking k symbols ahead in the input - LL(k) class of grammars

L: left to right reading of input
L: leftmost derivation of string
k: lookahead symbols

- If k not specified, LL(1) assumed
- Lookahead pointer points to next input symbols
- Uses explicit stack and end symbol \$ for bottom of stack and end of input

FIRST and FOLLOW

- Functions associated with a grammar G that allow parser to choose what production to apply

FIRST(α)

- FIRST(α) where $\alpha \in$ set of non-terminals : set of terminals that begin strings derived from non-terminal α
- FIRST(t) where t is a terminal : $\{t\}$
- Rules for computing FIRST(α)

1. If α is a terminal, FIRST(α) = $\{\alpha\}$
2. If $\alpha \rightarrow \lambda$, then add λ to FIRST(α)

3. If $\alpha \rightarrow Y_1 Y_2 Y_3$
   - $\text{FIRST}(\alpha) = \text{FIRST}(Y_1)$
   - If $\lambda \in \text{FIRST}(Y_1)$, then $\text{FIRST}(\alpha) = \{\text{FIRST}(Y_1) - \lambda\} \cup \{\text{FIRST}(Y_2)\}$
   - If $\lambda \in \text{FIRST}(Y_i)$ for all $1 \leq i \leq n$, then add λ to $\text{FIRST}(\alpha)$

Q: For the grammar, compute FIRST of all NTs

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow * FT' \mid \lambda$
$F \rightarrow id \mid num \mid (E)$

- FIRST(E) = {id, num, (}
- FIRST(E') = {+, λ}
- FIRST(T) = {id, num, (}
- FIRST(T') = {*, λ}
- FIRST(F) = {id, num, (}

FOLLOW(A)

- FOLLOW(A) where A is a non-terminal: set of terminals a that appear immediately to the right of A in some sentential form

- Set of terminals a such that there exists a derivation of the form

$$S \rightarrow \alpha A a \beta$$

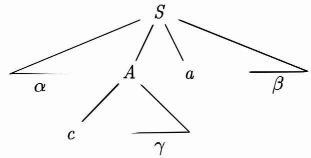


Figure 4.15: Terminal c is in FIRST(A) and a is in FOLLOW(A)

- Useful when there are nullable non-terminals

- Rules to compute FOLLOW(A)

1. FOLLOW(S) = {\$} where S = start symbol

2. If $A \to \alpha B \beta$ where $\lambda \in$ FIRST(β), then FOLLOW(B) = {FIRST(β) − λ} ∪ {FOLLOW(A)}

3. If $A \to \alpha B$, FOLLOW(B) = FOLLOW(A)

Q: Compute FOLLOW(E), FOLLOW(E'), FOLLOW(T), FOLLOW(T'), FOLLOW(F) for each production

$E \to TE'$
$E' \to +TE' \mid \lambda$
$T \to FT'$
$T' \to *FT' \mid \lambda$
$F \to id \mid num \mid (E)$

$E \to TE'$	FOLLOW(E) = {\$} FOLLOW(T) = {+} ∪ FOLLOW(E) FOLLOW(E') = FOLLOW(E)	{\$} {+, \$} {\$}
$E' \to +TE' \mid \lambda$	FOLLOW(T) = {+} ∪ FOLLOW(E') FOLLOW(E') = FOLLOW(E')	{+, \$} {\$}
$T \to FT'$	FOLLOW(F) = {*} FOLLOW(T') = FOLLOW(T)	{*} {+, \$}
$T' \to *FT'$	FOLLOW(F) = {*} FOLLOW(T') = FOLLOW(T')	{*} {+, \$}
$F \to (E)$	FOLLOW(E) = {)}	{)}

Steps to Construct LL(1) Parsing Table

- Eliminate left recursion and left factor the grammar
- Calculate FIRST and FOLLOW sets of all non-terminals
- Draw a table where first row contains the terminals and the first column contains the non-terminals
- All the null productions will go under the elements of FOLLOW(LHS)
- For each production $A \to \alpha$

  1. For each terminal in FIRST(α), make the entry $A \to \alpha$ in the table
  2. If FIRST(α) contains λ, then for each terminal in FOLLOW(A), make an entry $A \to \alpha$ in the table
  3. If the FIRST(α) contains λ and FOLLOW(A) contains \$, then make an entry $A \to \alpha$ in the table under \$

Q: Construct LL(1) parsing table (use FIRST & FOLLOW)

$E \to TE'$ ⟶ add under all elements in FIRST(T)
$E' \to +TE' \mid \lambda$ ⟶ look at rule 2 for λ
$T \to FT'$
$T' \to *FT' \mid \lambda$
$F \to id \mid num \mid (E)$

FIRST(E) = {id, num, (}
FIRST(E') = {+, λ}
FIRST(T) = {id, num, (}
FIRST(T') = {*, λ}
FIRST(F) = {id, num, (}

FOLLOW(E) = {$}
FOLLOW(E') = {$}
FOLLOW(T) = {+, $}
FOLLOW(T') = {+, $}
FOLLOW(F) = {*}

	id	+	*	num	(	)	$
E	E -> TE'			E -> TE'	E -> TE'		
E'		E'->+TE'				E' ->λ	E' ->λ
T	T -> FT'			T -> FT'	T -> FT'		
T'		T'->λ	T'->*FT'			T'->λ	T'->λ
F	F->id			F->num	F->(E)		

- To check if grammar belongs to LL(1), each box in LL(1) parser table should have at most one production

Q: Show LL(1) parsing table for the grammar

$$
\begin{array}{lcl}
S & \rightarrow & iEtSS' \mid a \\
S' & \rightarrow & eS \mid \epsilon \\
E & \rightarrow & b
\end{array}
$$

Non - terminal	Input Symbol					
	a	b	e	i	t	$\$$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Parsing Using an LL(1) Parsing Table

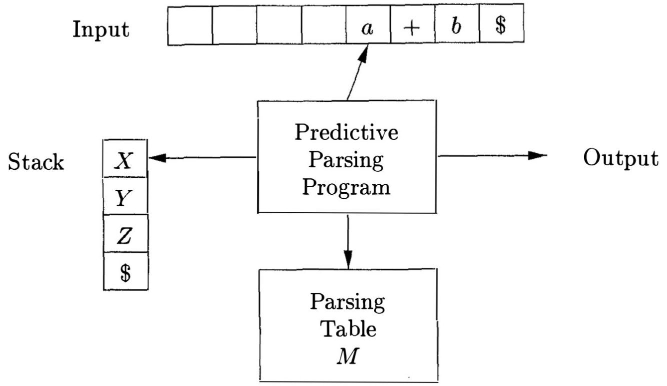


Figure 4.19: Model of a table-driven predictive parser

- Given: parsing table ; need to parse input string
- Stack contains start symbol S
  Input buffer contains input string w

Stack	Input Buffer	Action
S $	w $	

- If string accepted, stack and input buffer contain $

Stack	Input Buffer	Action
$	$	accept

Parsing Algorithm

- Let a = current input symbol
- Let M = parsing table
- Let w = input string
- Let ip = pointer to first symbol of w
- Let X = top of stack

set ip to point to the first symbol of w;
set X to the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
    if (X is a) pop the stack and advance ip; → top of stack = input symbol
    else if (X is a terminal) $error()$; → top of stack input symbol does not match
specified error entry ← **else if** ($M[X, a]$ is an error entry) $error()$;
    else if ($M[X, a] = X \to Y_1 Y_2 \cdots Y_k$) { → if table entry is production
        output the production $X \to Y_1 Y_2 \cdots Y_k$;
        pop the stack;
        push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
    }
    set X to the top stack symbol;
}

Figure 4.20: Predictive parsing algorithm

Q: Given the following parsing table, parse the input string
w = id + id * id

table M

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	$F \to$ **id**			$F \to (E)$		

	Stack	Input Buffer	Action
1.	top → E $	id + id * id $	

Find M[E, id] : action for current input symbol

	Stack	Input Buffer	Action
	E$	id + id * id $	$E \rightarrow TE'$ pop E push TE'
2.	TE' $	id + id * id $	$T \rightarrow FT'$ pop T push FT'
3.	FT'E'$	id + id * id $	$F \rightarrow id$ pop F push id
4.	id T' E' $	id + id * id $	id == id pop id advance ip
5.	T'E' $	+ id * id $	$T' \rightarrow \lambda$ pop T'
6.	E' $	+ id * id $	$E' \rightarrow +TE'$ pop E' push + TE'
7.	+TE' $	+ id * id $	+ == + pop + advance ip

8.	$TE'\$$	$id * id\ \$$	$T \to FT'$ pop T push FT'
9.	$FT'E'\$$	$id * id\ \$$	$F \to id$ pop F push id
10.	$id\ T'E'\$$	$id * id\ \$$	id == id pop id advance ip
11.	$T'E'\$$	$* id\ \$$	$T' \to * FT'$
12.	$* FT'E'\$$	$* id\ \$$	$* == *$
13.	$FT'E'\$$	$id\ \$$	$F \to id$
14.	$id\ T'E'\$$	$id\ \$$	id == id
15.	$T'E'\$$	$\$$	$T' \to \lambda$
16.	$E'\$$	$\$$	$E' \to \lambda$
17.	$\$$	$\$$	Accept

Q: Construct LL(1) parsing table for the grammar

$S \to a \mid (L)$
$L \to L, S \mid S$

Parse $w = (a, a)$

Left recursion

① $S \to a \mid (L)$

② $L \to L, S \mid a \mid (L)$

$$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

$$\downarrow$$

$$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \lambda$$

① $S \to a \mid (L)$

② $L \to aL' \mid (L)L'$

③ $L' \to , S L' \mid \lambda$

FIRST(S) = $\{a, (\}$
FIRST(L) = $\{a, (\}$
FIRST(L') = $\{,, \lambda\}$

FOLLOW(S) = $\{\$, ,,)\}$
FOLLOW(L) = $\{)\}$
FOLLOW(L') = $\{)\}$

	a	(	)	,	\$
S	$S \to a$	$S \to (L)$			
L	$L \to aL'$	$L \to (L)L'$			
L'			$L' \to \lambda$	$L' \to , SL'$	

Stack	Input Buffer	Action
S$	(a,a)$	S→(L)
(L)$	(a,a)$	(== (
L)$	a,a)$	L→aL'
aL')$	a,a)$	a == a
L')$	,a)$	L'→,SL'
,SL')$	,a)$	, == ,
SL')$	a)$	S→a
aL')$	a)$	a == a
L')$	)$	L'→λ
)$	)$	) ==)
$	$	Accept

Identify whether a Grammar Belongs to LL(1)

- If not left factored or is left recursive, not LL(1)
- If LL(1) table has >1 entry per cell, not LL(1) grammar

- Check without table: a grammar is NOT LL(1) if:

1. For $A \to a_1 | a_2 | \dots | a_n$, $n \geq 2$ alternatives

   If $FIRST(a_1) \cap FIRST(a_2) \neq \phi$

   (something in common)

2. $A \to a | \lambda$

   If $FIRST(a) \cap FOLLOW(A) \neq \phi$

Q: Is the given grammar LL(1)? Compute FIRST & FOLLOW sets also

$S \to AaAb \mid BbBa$
$A \to \lambda$
$B \to \lambda$

$FIRST(S) = \{a, b\}$
$FIRST(A) = \{\lambda\}$
$FIRST(B) = \{\lambda\}$

$FOLLOW(S) = \{\$\}$
$FOLLOW(A) = \{a, b\}$
$FOLLOW(B) = \{a, b\}$

$FIRST(AaAb) \cap FIRST(BbBa) = \{a\} \cap \{b\} = \phi$

$\therefore$ it is LL(1)

Q: Is the grammar in LL(1)? If not, modify and re-check

$S \to iCtS \mid iCtSeS \mid a$
$C \to b$

$FIRST(iCtS) \cap FIRST(iCtSeS) \cap FIRST(a)$

$\{i\} \cap \{i\} = \{i\} \neq \phi$

$\therefore$ not LL(1)

Modification: left factor

$S \to iCtSX \mid a$
$X \to eS \mid \lambda$
$C \to b$

$FIRST(iCtSX) \cap FIRST(a) = \{i\} \cap \{a\} = \phi$

$FIRST(eS) \cap FOLLOW(X) = \{e\} \cap \{\$, e\} = \{e\} \neq \phi$

$\therefore$ not LL(1)

LL(k) Grammars

- $k > 1$. read k input characters to choose production
- LL(2):

  $S \to ab \mid ac \mid ad$
- If LL(k) table has $> k$ entries in a cell, grammar not LL(k)

Q: Is the grammar LL(k)? What is k?

$S \to iCtS \mid iCtSeS \mid a$
$C \to b$

No $\because$ not left factored

Q: Is the grammar LL(k)? What is k?

$S \rightarrow aaB \mid aaC$
$B \rightarrow b$
$C \rightarrow c$

FIRST(S) = a
FIRST(B) = b
FIRST(C) = c

FOLLOW(S) = \$
FOLLOW(B) = \$
FOLLOW(C) = \$

	a	b	c	\$
S	$S \rightarrow aaB$ $S \rightarrow aaC$			
B		$B \rightarrow b$		
C			$C \rightarrow c$	

- $k \gtrsim 2$
- $k = 3$ ∵ 3 input symbols to be read

Q: Construct LL(1) parsing table, parse $w = \lambda$

$S \rightarrow AB$
$A \rightarrow a \mid \lambda$
$B \rightarrow b \mid \lambda$

FIRST(S) = $\{a, b, \lambda\}$
FIRST(A) = $\{a, \lambda\}$
FIRST(B) = $\{b, \lambda\}$

FOLLOW(S) = {\$}
FOLLOW(A) = {b, \$}
FOLLOW(B) = {\$}

	a	b	$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to a$	$A \to \lambda$	$A \to \lambda$
B		$B \to b$	$B \to \lambda$

Stack	Input Buffer	Action
S$	$	$S \to AB$ pop S push AB
AB$	$	$A \to \lambda$
B$	$	$B \to \lambda$
$	$	Accept

Q: Is the grammar LL(1)? Compute FIRST and FOLLOW. Parse w= cde

$S \to ABCDE$
$A \to a \mid \lambda$
$B \to b \mid \lambda$
$C \to c$
$D \to d \mid \lambda$
$E \to e \mid \lambda$

FIRST(S) = {a, b, c}
FIRST(A) = {a, λ}
FIRST(B) = {b, λ}
FIRST(C) = {c}
FIRST(D) = {d, λ}
FIRST(E) = {e}

FOLLOW(S) = {$}
FOLLOW(A) = {b, c}
FOLLOW(B) = {c}
FOLLOW(C) = {d, e}
FOLLOW(D) = {e, $}
FOLLOW(E) = {$}

	a	b	c	d	e	$
S	S→ABCDE	S→ABCDE	S→ABCDE			
A	A→a	A→λ	A→λ			
B		B→b	B→λ			
C			C→c			
D				D→d	D→λ	D→λ
E					E→e	E→λ

Stack	Input Buffer	Action
S$	cde$	S→ABCDE
ABCDE$	cde$	A→λ
BCDE$	cde$	B→λ
CDE$	cde$	C→c
cDE$	cde$	c==c
DE$	de$	D→d
dE$	de$	d==d
E$	e$	E→e
e$	e$	e==e
$	$	Accept

Q: Are the grammars in LL(k)? Find k

1. $S \to Abbx \mid Bbby$
   $A \to x$
   $B \to x$

2. $S \to Z$
   $Z \to aMa \mid bMb \mid aRb \mid bRa$
   $M \to c$
   $R \to c$

1. LL(4)

2. LL(3)

Error Recovery in LL(1) Parser

- If an entry M[X,Y] is blank in the parsing table, if it is encountered while parsing → syntax error

- Panic Mode Recovery: discard all following input symbols until delimiter found, then restart parsing
  - Delimiters: synchronizing tokens

- For non-terminal X in the parsing table, add synch token under the elements of FOLLOW(X) (if blank)

- Parsing procedure

```
if M[X,a] == blank :  // syntax error
    ignore input symbol a

if M[X,a] == synch :
    if X is the only symbol in the stack:
        ignore input symbol a
    else:
        pop X from stack
```

Q: Implement parsing table with panic mode recovery. Parse w =)id* +id

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \lambda$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \lambda$

$F \rightarrow num \mid id \mid (E)$

	FIRST	FOLLOW
E	num, id, (	), $
E'	+, λ	), $
T	num, id, (	+, $,)
T'	*, λ	+, $,)
F	num, id, (	*, +, $,)

	+	*	num	id	(	)	$
E			E→TE'	E→TE'	E→TE'	synch	synch
E'	E'→+TE'					E'→λ	E'→λ
T	synch		T→FT'	T→FT'	T→FT'	synch	synch
T'	T'→λ	T'→*FT'				T'→λ	T'→λ
F	synch	synch	F→num	F→id	F→(E)	synch	synch

Stack	Input Buffer	Action
E$	)id*+id$	synch → only el in stack
E$	id*+id$	E→TE'
TE'$	id*+id$	T→FT'
FT'E'$	id*+id$	F→id
idT'E'$	id*+id$	id==id
T'E'$	*+id$	T'→*FT'
*FT'E'$	*+id$	* == *
FT'E'$	+id$	synch → pop top of stack
T'E'$	+id$	T'→λ
E'$	+id$	E'→+TE'
+TE'$	+id$	+ == +

TE'$	id $	T→FT'
FT'E'$	id $	F→id
idT'E'$	id $	id==id
T'E'$	$	T'→λ
E'$	$	E'→λ
$	$	Accept

Q: Parse id (+) id using above tables and grammar

Stack	Input Buffer	Action
E$	id (+) id$	E→TE'
TE'$	id (+) id $	T→FT'
FT'E'$	id (+) id $	F→id
idT'E'$	id (+) id $	id==id
T'E'$	(+) id $	ignore (
T'E'$	+) id $	T'→λ
E'$	+) id $	E'→+TE'
+TE'$	+) id $	+==+
TE'$	) id $	synch
E'$	) id $	E'→λ
$	) id $	Restart (stop)

Restart

E$	)id$	synch
E$	id$	E→TE'
TE'$	id$	T→FT'
FT'E'$	id$	F→id
idT'E'$	id$	id==id
T'E'$	$	T'→λ
E'$	$	E'→λ
$	$	Accept

RDP without Backtracking vs Predictive Parsers

RDP without Backtracking	Predictive Parsers
It uses mutually recursive procedures for every non-terminal entity to parse strings.	It uses a lookahead pointer which points to next k input symbols. This places a constraint on the grammar.
It accepts all grammars.	It accepts only a LL(k) grammars.

- RDP without BT is more powerful

Q: Implement PT for grammar with PMR. Parse w= a(a)

$S \rightarrow a \mid (L)$
$L \rightarrow aL' \mid (L)L'$
$L' \rightarrow ,SL' \mid \lambda$

	FIRST		FOLLOW		
S	a	(	\$	,	)
L	a	(	)		
L'	,	λ	)		

	a	(	)	,	\$
S	$S\rightarrow a$	$S\rightarrow (L)$	synch	synch	synch
L	$L\rightarrow aL'$	$L\rightarrow (L)L'$	synch		
L'			$L'\rightarrow \lambda$	$L'\rightarrow ,SL'$	

Stack	Input Buffer	Action
S$	a(a)$	S→a
a$	a(a)$	a==a
$	(a)$	Restart (stop)

Stack	Input Buffer	Action
S$	(a)$	S→(L)
(L)$	(a)$	(==(
L)$	a)$	L→aL'
aL')$	a)$	a==a
L')$	)$	L'→λ
)$	)$	)==)
$	$	Accept

Q: Parse (,a,a)

Stack	Input Buffer	Action
S$	(,a,a)$	S→(L)
(L)$	(,a,a)$	(==(
L)$	,a,a)$	blank
L)$	a,a)$	L→aL'
aL')$	a,a)$	a==a
L')$	,a)$	L'→,SL'
,SL')$	,a)$	,==,
SL')$	a)$	S→a
aL')$	a)$	a==a
L')$	)$	L'→λ
)$	)$	)==)
$	$	Accept

BOTTOM UP PARSER

- Start with input string (leaves) and derive start symbol (root)
- Reads L to R and provides rightmost derivation
- No requirement of left factoring or free of left recursion

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

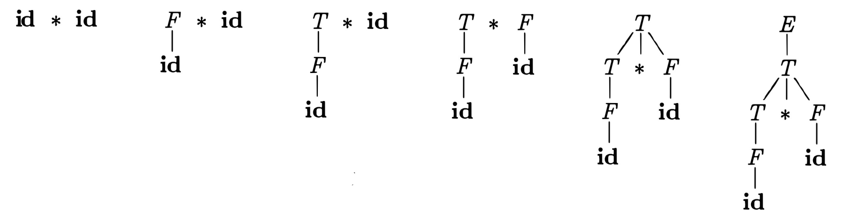


Figure 4.25: A bottom-up parse for **id** $*$ **id**

1. Shift - Reduce Parser

- Process of reducing a string w to the start symbol of the grammar
- Reduction step: specific substring matching the body of a production is replaced by the non-terminal at the head of that production
- Key decisions: when to reduce and what to reduce
- Constructs rightmost derivation in reverse

HANDLE PRUNING

- Handle: substring on stack that matches the body of a production whose reduction represents one step along the reverse of a rightmost derivation

- The leftmost substring that matches the body of some production need not be a handle

Right Sentential Form	Handle	Reducing Production
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

is not a handle as $E \to E * id$ is not valid ← T

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

- Formally, if $S \underset{rm}{\Rightarrow} \alpha A w \underset{rm}{\Rightarrow} \alpha \beta w$
  - Production $A \to \beta$ is a handle of $\alpha \beta w$
  - For convenience, the body β of $A \to \beta$ is referred to as a handle
  - Grammar could be ambiguous with more than one RMD of $\alpha \beta w$ and each right-sentential form may have multiple possible handles

- RMD in reverse can be obtained by handle pruning

- Suppose w is a string of terminals that needs to be reverse-derived such that $w = \gamma_n$ where γ_n is the n^{th} right sentential form

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

- To reconstruct, we locate a handle β_n in γ_n and replace β_n with the head of the production $A_n \rightarrow \beta_n$ to get the previous right sentential form γ_{n-1}

- Repeat until start symbol is reached

SHIFT-REDUCE

- Stack holds grammar symbols and input buffer holds rest of string to be parsed

- Handle appears at top of the stack before it is identified as handle

- \$: bottom of stack, right end of input

- Convention for BUP: top of stack shown on the right (unlike TDP)

STACK	INPUT
\$	w \$

- Left-to-right scan of input string; parser shifts 0 or more input symbols onto stack (called shift)

- Once ready to, a string β of grammar symbols at the top of the stack is reduced to the head of the appropriate production

- Repeat until stack contains start symbol or error

STACK	INPUT
\$ S	\$

- Steps of Shift-Reduce parser

Stack	Input	Action
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \to T * F$
$\$ T$	$\$$	reduce by $E \to T$
$\$ E$	$\$$	accept

Figure 4.28: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

Possible Actions of Shift-Reduce Parser

1. *Shift.* Shift the next input symbol onto the top of the stack.
2. *Reduce.* The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. *Accept.* Announce successful completion of parsing.
4. *Error.* Discover a syntax error and call an error recovery routine.

Conflicts in Shift-Reduce Parsing

- For some grammers, shift-reduce parsers fail
  - Cannot decide whether to shift or reduce (shift/reduce conflict)
  - Cannot decide which reduction to make (reduce/reduce conflict)

- Syntactic constructs that give rise to such grammars
  - such grammars not in the LR(k) class of grammers (non-LR grammars)

Example 4.38: An ambiguous grammar can never be LR. For example, consider the dangling-else grammar (4.14) of Section 4.3:

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{other} \end{array}$$

If we have a shift-reduce parser in configuration

STACK	INPUT
$\cdots$ **if** *expr* **then** *stmt*	**else** $\cdots$ \$

we cannot tell whether **if** *expr* **then** *stmt* is the handle, no matter what appears below it on the stack. Here there is a shift/reduce conflict. Depending on what follows the **else** on the input, it might be correct to reduce **if** *expr* **then** *stmt* to *stmt*, or it might be correct to shift **else** and then to look for another *stmt* to complete the alternative **if** *expr* **then** *stmt* **else** *stmt*.

- Can resolve on else in favour of shifting, as there is an else associated with the previously unmated then

C Pseudocode with S|R Conflict

```
1.   int a = 10, b = 20, c = 30;
2.   if (a>b)
3.       if (a>c)
4.           printf("%d", a);
5.       else if (c>=a)
6.           printf("%d", c);
7.   else if (b>c)
8.       printf("%d", b);
9.   else
10.      printf("%d", c);
```

The else if on line 7 will be considered as a part of the inner if statement because the compiler shifts it rather than reducing lines 2 to 6 together.

Q: Parse id*id using SR parser and the given grammar

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$

$w = id*id$

Stack	Input Buffer	Action
$	id*id$	shift id
$id	*id$	reduce F→id
$F	*id$	reduce T→F
$T	*id$	reduce E→T or shift * (conflict) choose: shift *
$T*	id$	shift id
$T*id	$	reduce F→id
$T*F	$	reduce T→T*F
$T	$	reduce E→T
$E	$	Accept

Q: For $S \to 0S1 \mid 01$, indicate handle for the following right sentential forms and parse

(a) 000111

Stack	Input Buffer	Action
$	000111$	shift
$0	00111$	shift
$00	0111$	shift
$000	111$	shift

$\$00\underbrace{01}_{\text{handle}}$	11\$	reduce
$\$00S$	11\$	shift
$\$0\underbrace{0S1}_{\text{handle}}$	1\$	reduce
$\$0S$	1\$	shift
$\$\underbrace{0S1}_{\text{handle}}$	\$	reduce
\$S	\$	Accept

(b) 00S11

Stack	Input Buffer	Action
\$	00S11\$	shift
\$0	0S11\$	shift
\$00	S11\$	shift
\$00S	11\$	shift
$\$0\underbrace{0S1}_{\text{handle}}$	1\$	reduce
$\$0S$	1\$	shift
$\$\underbrace{0S1}_{\text{handle}}$	\$	reduce
\$S	\$	Accept

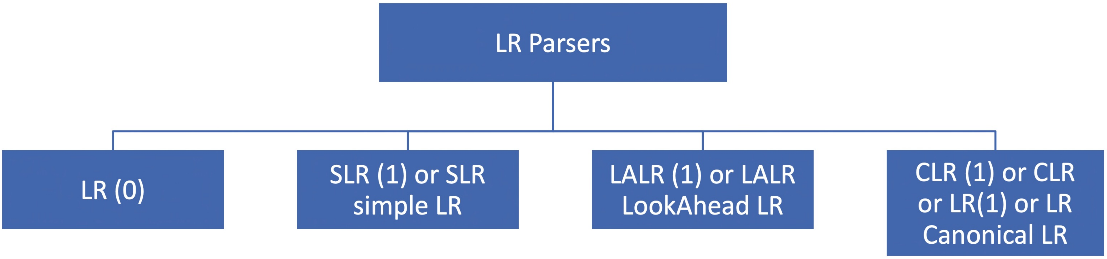
2. Table-Driven LR Parsers

- Most prevalent bottom-up parser: LR(k) parser

- L: left-to-right scan of input
  R: rightmost derivation in reverse
  k: no. of input tokens of lookahead used in making parsing decisions

- If k is omitted, LR(1) is assumed (practically, only LR(k) where $k \le 1$ are considered)

TYPES of LR PARSERS



- Power of a parser: how quickly it can catch errors
  - power increases from LR(0) < SLR(1) < LALR(1) < CLR(1)

- Number of states required: $n_{LR(0)} = n_{SLR} = n_{LALR} \le n_{CLR}$

- All 4 use DFAs
  - LR(0) and SLR(1) — LR(0) automata
  - LALR(1) and CLR(1) — LR(1) automata

Items

- How to decide between shifting and reducing? How to decide if handle exists?

- Item: production with a dot in the RHS

Thus, production $A \to XYZ$ yields the four items

$$\begin{aligned} A &\to \cdot XYZ \\ A &\to X{\cdot}YZ \\ A &\to XY{\cdot}Z \\ A &\to XYZ\cdot \end{aligned}$$

- Dot indicates where we are in the production

- If . at end of prod, it is called final item or kernel item*; indicates that handle is on TOS and can perform reduce

- $A \to X \cdot YZ$ means X is seen and YZ is expected to be seen next

1. LR(0) Automaton

- Canonical LR(0) collection : collection of set of items

- Augmented grammar: if G is a grammar with start symbol S, G' is an augmented grammar with new start symbol S' and production $S' \to S$
  - Accept only when reducing S to S'

- Two functions: CLOSURE and GOTO

* see page 57

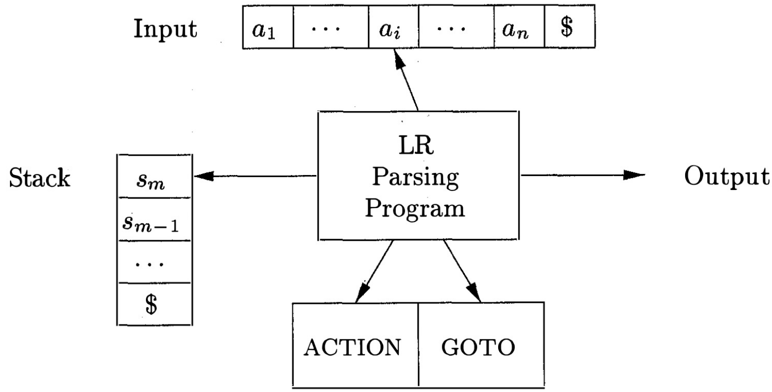


Figure 4.35: Model of an LR parser

CLOSURE(I)

- I: set of items for grammar G
- CLOSURE(I): set of items constructed from I by the two rules:

1. Add every item in I to CLOSURE(I)
2. If $A \to \alpha \cdot B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a prod, then add $B \to \cdot \gamma$ to CLOSURE(I)

In simple words, if there is a non-terminal after the . in any of the items in CLOSURE(I), add all prod of that non-terminal

GOTO(I,X)

- Closure of all set of items $[A \to \alpha X \cdot \beta]$ such that $[A \to \alpha . X B]$ is in I
- Used to define transition for each symbol after the dot

Q: Examine whether grammar is in LR(0) or not

$S \to AA$
$A \to aA \mid b$

Steps

1. Augment grammar
2. Number prod in original grammar
3. Follow LR(0) parsing algorithm
4. Create parsing table

1. Augment grammar

$S' \to S$
$S \to AA$
$A \to aA$
$A \to b$

2. Number prods from original grammar

$S' \to S$
1. $S \to AA$
2. $A \to aA$
3. $A \to b$

Initial item: $S' \to .S$

3. Follow LR(0) parsing algorithm

Step 1: start with initial item

$S' \to .S$

Step 2: Adding prods of NTs after dot

$S' \to .S$
$S \to .AA$
$A \to .aA$
$A \to .b$

Step 3: make the prods a closure (put it in a box and name it)

State I_0

$S' \to .S$
$S \to .AA$
$A \to .aA$
$A \to .b$

Step 4: Transitions using goto

GOTO(I, X): define trans from state I_j on every symbol X after the dot

State I_0

$S' \to .S$
$S \to .AA$
$A \to .aA$
$A \to .b$

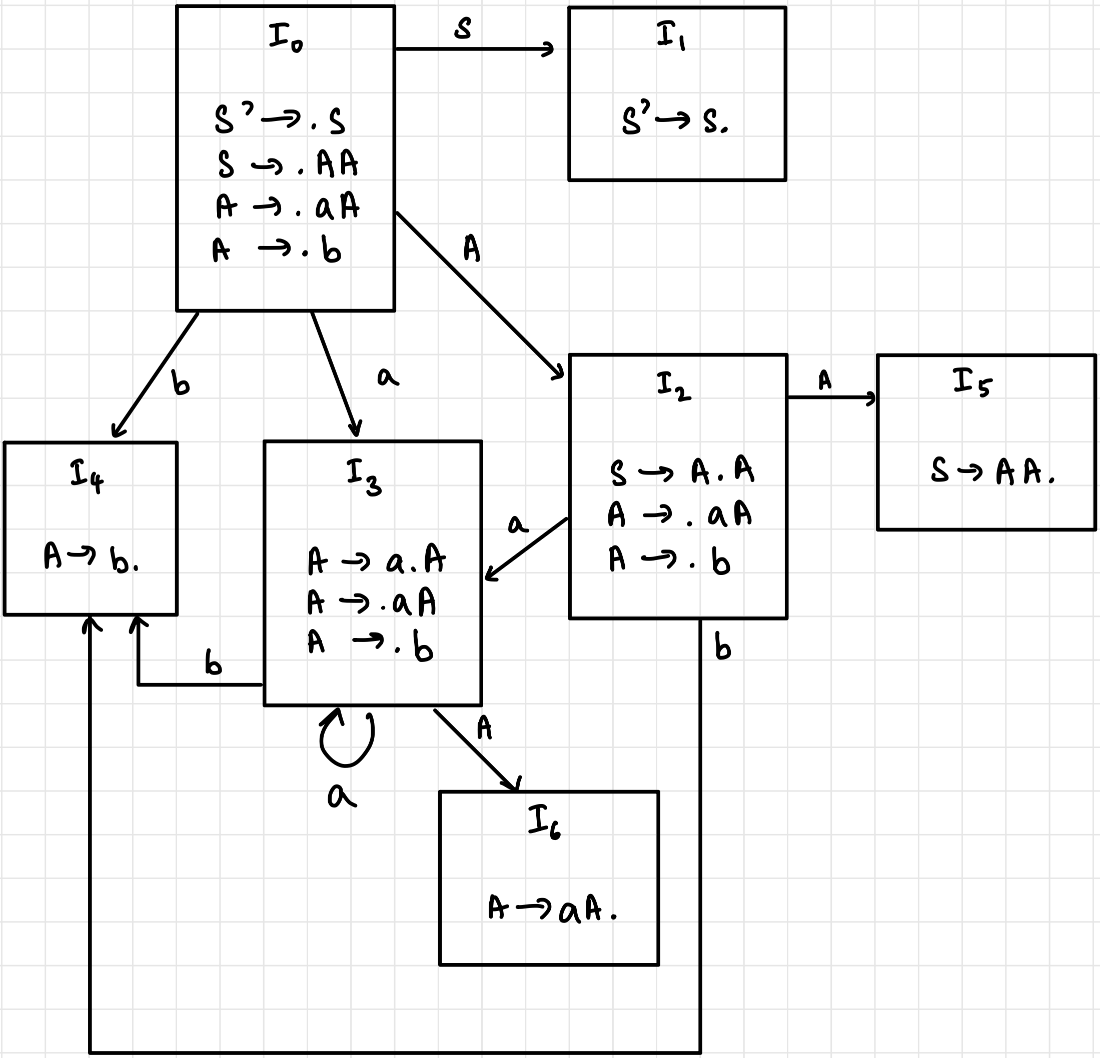
$I_0 \xrightarrow{S} S' \to S.$

$I_0 \xrightarrow{A} S \to A.A$

$I_0 \xrightarrow{a} A \to a.A$

$I_0 \xrightarrow{b} A \to b.$

Step 5: Repeat steps 2-4 until no more states can be added



Canonical collection of LR(0) items: $C = \{I_0, I_1, I_2, I_3, I_4, I_5, I_6\}$

$S' \to S.$: special final item

4. Create LR(0) parsing table

- Three parts:
  (a) state: state numbers from automaton
  (b) Action: sub-columns for each terminal and \$
  (c) Goto: sub-columns for each non-terminal

- Structure

State	Action			Goto	
	a	b	\$	S	A
0					

- $I_0 \xrightarrow{S} I_1$ ⇒ goto of S for 0 is 1
  $I_0 \xrightarrow{A} I_2$ ⇒ goto of A for 0 is 2
  so on

- State I_1 contains special final item ⇒ Accept should go under \$

- Trans on a symbol (terminal) is shift, denoted by s_x where x is the next state

- When state has a final item, reduce; denoted by r_x where x is the rule no. that the prod corresponds to. r_x placed in entire row

State	Action			Goto	
	a	b	\$	S	A
0	s_3	s_4		1	2
1			accept		
2	s_3	s_4			5
3	s_3	s_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

I_4: $A \to b.$ prod 3
I_5: $S \to AA.$ prod 1
I_6: $A \to aA.$ prod 2

- If more than one entry per cell, not LR(0)
- ∴ grammar is LR(0)

Q: Use the above PT to parse w=abb - LR(0)

State	Action			Goto	
	a	b	$	S	A
0	s_3	s_4		1	2
1			accept		
2	s_3	s_4			5
3	s_3	s_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

$S' \to S$
1. $S \to AA$
2. $A \to aA$
3. $A \to b$

Stack	Input Buffer	Action
$0	abb$	s_3 push a push 3 read a
$0a3	bb$	s_4 push b push 4 read b
$0a3b4	b$	r_3 : reduce prev sym using 3rd prod pop 2x sym off stack; x= size of RHS of prod 3 x=1 2x=2

pop 4
pop b
push A

TOS is state 3
goto (3, A) = 6
push 6

$0a3A6	b$	r_2

reduce prev sym using 2^{nd} prod $A \to aA$
pop 6
pop A
pop 3
pop a
push A

TOS is state 0
goto (0, A) = 2
push 2

$0A2	b$	s_4
$0A2b4	$	r_3 : reduce prev sym to prod 3 $A \to b$

pop 4
pop b
push A
goto (2, A) = 5
push 5

$0A2A5	$	r_1

$S \to AA$
pop 4 items
push S
push goto (0, S) = 1

\$ 0S1	\$	accept

Classes of Items

1. *Kernel items*: the initial item, $S' \to \cdot S$, and all items whose dots are not at the left end.

2. *Nonkernel items*: all items with their dots at the left end, except for $S' \to \cdot S$.

LR(0) - No Lookahead

Whenever there is a final item, we put the reduce move in the entire row corresponding to the state that contains the final item.

Considering the string,

a b b

Being at this position, we decide to reduce the previous 'b'.

Hence, we replace 'b' by 'A' using A → b

reduced to

a A b

Irrespective of the symbol we have here, we always reduce previous 'b' to 'A'.

This behaviour can be erroneous

Q: Is the grammar LR(0)?

$S \to A \mid a$

$A \to a$

- Is the grammar ambiguous? can a string be derived in multiple ways?

$w = a$

① $S \overset{rm}{\Rightarrow} A$ $\qquad$ $S \rightarrow A$
$S \overset{rm}{\Rightarrow} a$ $\qquad$ $A \rightarrow a$

② $S \overset{rm}{\Rightarrow} a$ $\qquad$ $S \rightarrow a$

Yes, it is ambiguous

$\therefore$ not LR(0)

- Full solution:

1. Augment grammar

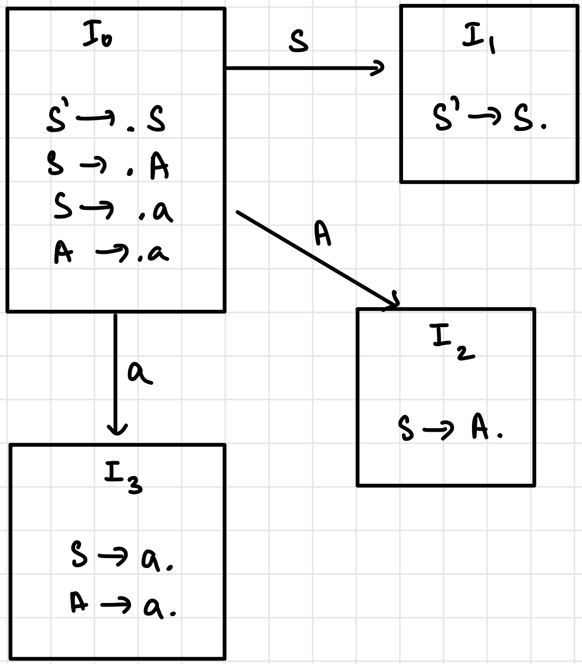
$S' \rightarrow S$
$S \rightarrow A$
$S \rightarrow a$
$A \rightarrow a$

2. Number prods

$S' \rightarrow S$
1. $S \rightarrow A$
2. $S \rightarrow a$
3. $A \rightarrow a$

3. Apply parsing algorithm

initial item: $S' \rightarrow .S$



special final prod

4. Table

State	Action		Goto	
	a	$	S	A
0	s_3		1	2
1		accept		
2	r_1	r_1		
3	r_2 / r_3	r_2 / r_3		

2 r/r conflicts

Q: Is this LR(0)?

$S \to AaAb \mid BbBa$
$A \to \lambda$
$B \to \lambda$

1. Aug & num

$S' \to S$
1. $S \to AaAb$
2. $S \to BbBa$
3. $A \to \lambda$
4. $B \to \lambda$

2. Algo

I_0

$S' \to .S$
$S \to .AaAb$
$S \to .BbBa$
$A \to .$
$B \to .$

conflict: r_3/r_4 in state 0

$\therefore$ not LR(0)

even though not ambiguous
$\because$ no lookahead

Q: Is the grammar LR(0)?

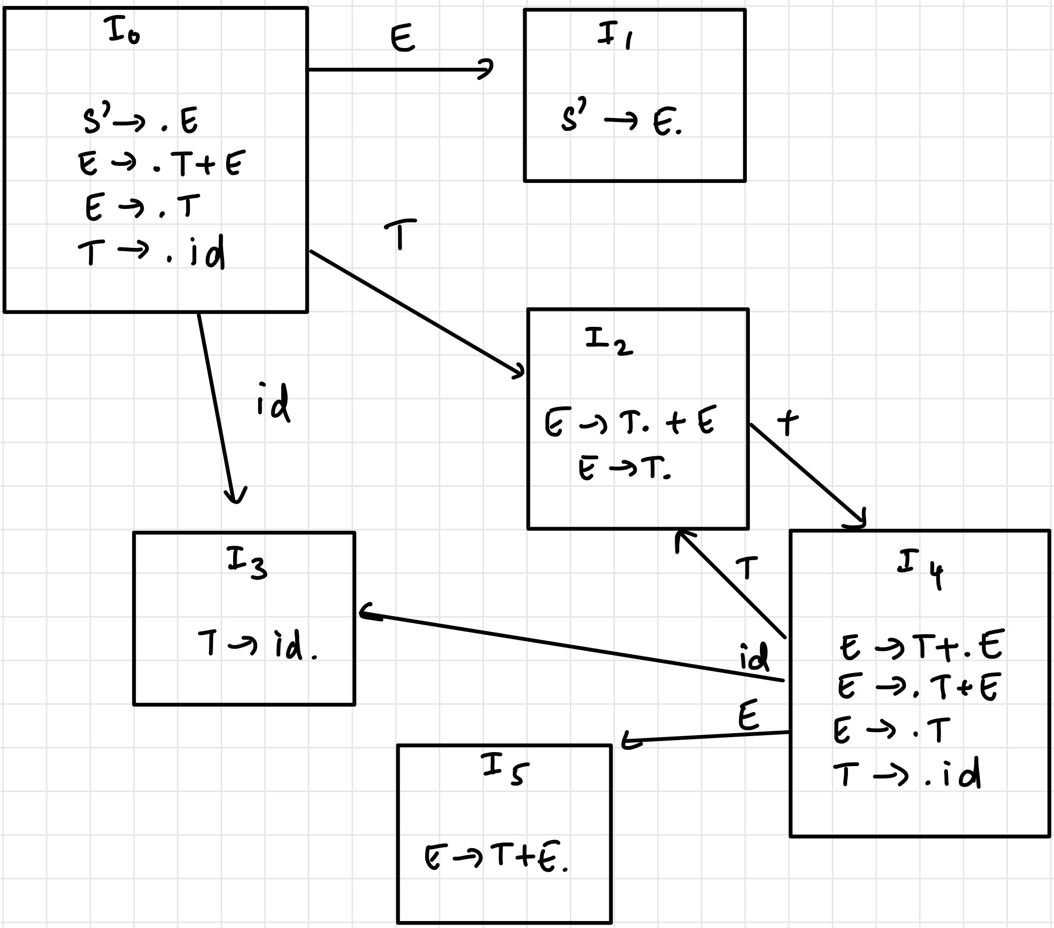
$E \to T+E \mid T$
$T \to id$

1. Aug & num

$S' \to E$

1. $E \to T + E$
2. $E \to T$
3. $T \to id$

2. Algo



3. Table

state	Action			Goto	
	+	id	$	E	T
0		s_3		1	2
1			accept		
2	s_4/r_2	r_2	r_2		
3	r_3	r_3	r_3		
4		s_3		5	2
5	r_1	r_1	r_1		

s/r conflict – can be resolved with lookahead

2. SLR(1)

- Difference: one lookahead for actions
- Places reduce move in FOLLOW (LHS) only, not entire row

0: $E' \to E$
1. $E \to T+E$
2. $E \to T$
3. $T \to id$

FOLLOW(E) = {$}
FOLLOW(T) = {+, $}

state	Action			Goto	
	+	id	$	E	T
0		s_3		1	2
1			accept		
2	s_4		r_2		
3	r_3		r_3		
4		s_3		5	2
5			r_1		

blank = error

SLR detects errors faster

∴ SLR(1) grammar

Q: Parse w = id+id using SLR(1) table above

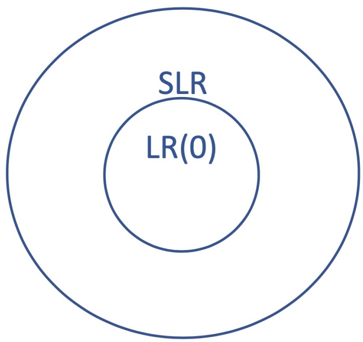
Stack	Input Buffer	Action
\$0	id+id \$	s_3 push id push 3
\$0 id 3	+id \$	r_3 $T \rightarrow id$ pop 3 pop id push T push goto(0,T) = 2
\$ 0T2	+ id \$	s_4 push + push 4
\$ 0T2+4	id \$	s_3 push id push 3
\$ 0 T2+4 id 3	\$	r_3 $T \rightarrow id$ pop 3 pop id push T goto (4, T) = 2
\$ 0T2 + 4T2	\$	r_2 $E \rightarrow T$ pop 2 items

$ 0 T 2 + 4 E 5 $

push E
goto(4, E) = 5
r_1
E → T + E
pop 6 items
push E
goto(0, E) = 1
accept

$ 0 E 1 $

SLR(1) vs LR(0)



Viable Prefixes

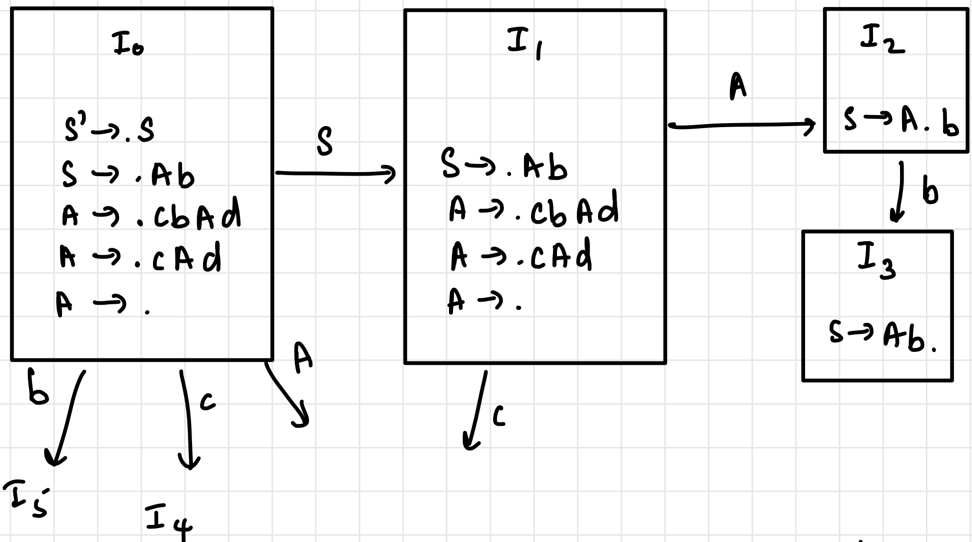
prefix of the handle

Rightmost derivation of id*id	Set of prefixes of a right sentential form	Viable Prefixes
E → T	T	T
→ T * F	T, T *, T * F	T, T *, T * F
→ T * id	T, T *, T * id	T, T *, T * id
→ F * id	F, F *, F * id	F
→ id * id	id, id *, id * id	id

Q: Provide a production with shortest RHS that will intro slr conflict in SLR(1) parser for the grammar

1. $S \to Ab$
2. $A \to cbAd$
3. $A \to cAd$
4. $A \to \lambda$

$A \to b$



$A \to c.bAd$
$A \to c.Ad$
$A \to .cbAd$
$A \to .cAd$
$A \to .$

FOLLOW(S) = \$
FOLLOW(A) = b, d

∴ a prod from $A \to b$ or d

state	Action				Goto	
	a	b	c	d	S	A
0		s_5 / r_0	s_4	r_0		

∴ intro prod $A \rightarrow b$

Q: Identify if LL(1), LR(0) or SLR(1)?

$X \rightarrow Yz \mid a$
$Y \rightarrow bZ \mid \lambda$
$Z \rightarrow \lambda$

FIRST(X) = b, z, a
FIRST(Y) = b, λ
FIRST(Z) = λ

FOLLOW(X) = \$
FOLLOW(Y) = z
FOLLOW(Z) = z

LL(1) table

	a	b	z	\$
X	$X \rightarrow a$	$X \rightarrow Yz$	$X \rightarrow Yz$	
Y		$Y \rightarrow bZ$	$Y \rightarrow \lambda$	
Z			$Z \rightarrow \lambda$	

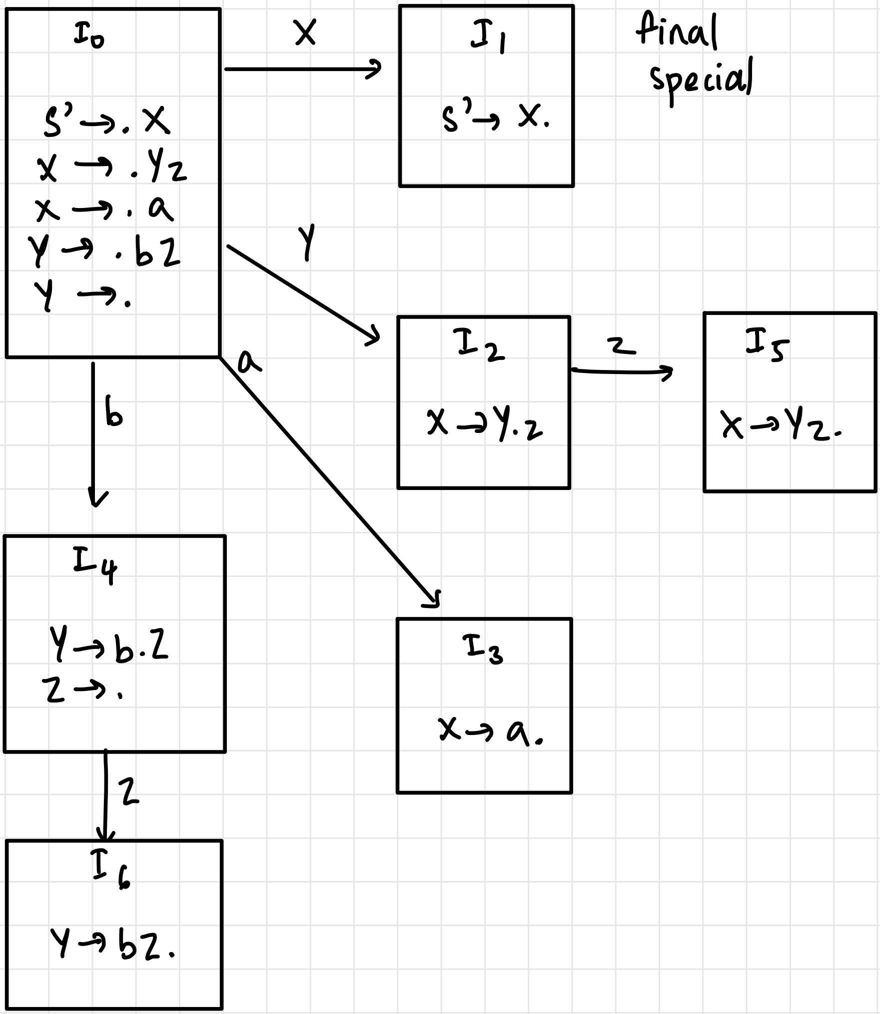
∴ grammar is LL(1)

LR(0)

1. Augment & number

$S' \rightarrow X$
1. $X \rightarrow Yz$
2. $X \rightarrow a$
3. $Y \rightarrow bZ$
4. $Y \rightarrow \lambda$
5. $Z \rightarrow \lambda$

2. Algorithm



I0
S' → .X
X → .Yz
X → .a
Y → .bZ
Y → .
X
I1
S' → X.
final
special
Y
I2
X → Y.z
z
I5
X → Yz.
a
I3
X → a.
b
I4
Y → b.Z
Z → .
Z
I6
Y → bZ.

LR(0) table

state	Actions				Goto		
	a	b	z	$	x	y	z
0	s_3/r_4	s_4/r_4	r_4	r_4			

s/r conflicts => not LR(0)

NO GRAMMAR WITH λ PROD IS LR(0)

SLR(1) table

state	Actions				Goto		
	a	b	z	$	x	y	z
0	s_3	s_4	r_4		1	2	
1				accept			
2			s_5				
3				r_2			
4			r_5				6
5				r_1			
6			r_3				

$\therefore$ SLR(1)

3. CLR(1)

- Modify SLR(1) to use LR(1) items
  - eg: $A \to X.YZ, \{a, b\}$
    lookahead symbols
- Place reduce moves in LA symbols instead of entire row – LR(0) – or FOLLOW(LHS) – SLR
- Using LR(0): start with

$$\boxed{S' \to .S} \xrightarrow{S} \boxed{S' \to S.}$$

- Using LR(1) or CLR: start with

$$\boxed{S' \to .S, \$} \xrightarrow{S} \boxed{S' \to S., \$}$$

- LR(1) is powerful; LR(0) ⊂ LR(1), LL(1) ⊂ LR(1)
- Huge automata

Calculating the lookahead (the set after the comma)

- Let an LR(1) item be

$$A \to \alpha . B\beta, a$$

- NT B after . ⇒ add B prods

- Let $B \to .\gamma$, something

- How to find the lookahead for B?

  - Lookahead = FIRST(βa)

Q: Consider grammar

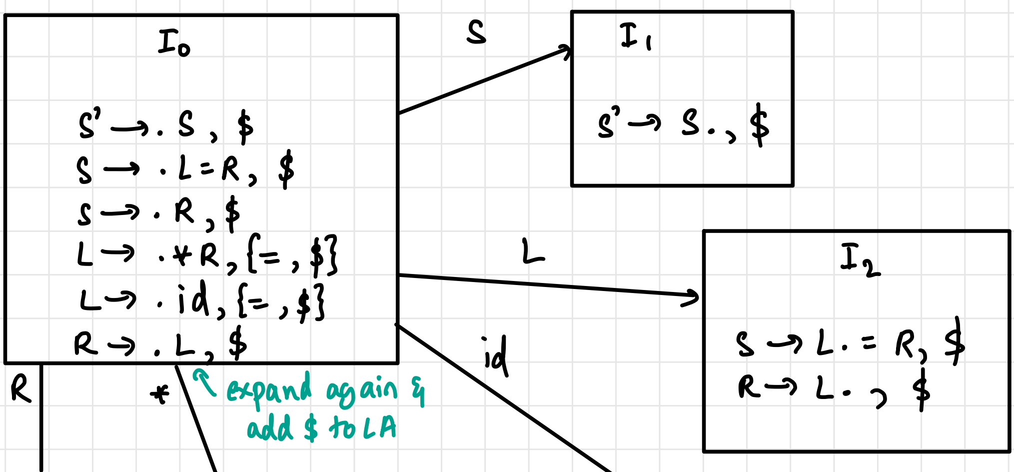
$S \to L = R \mid R$
$L \to *R \mid id$
$R \to L \Rightarrow R \to *R \mid id$

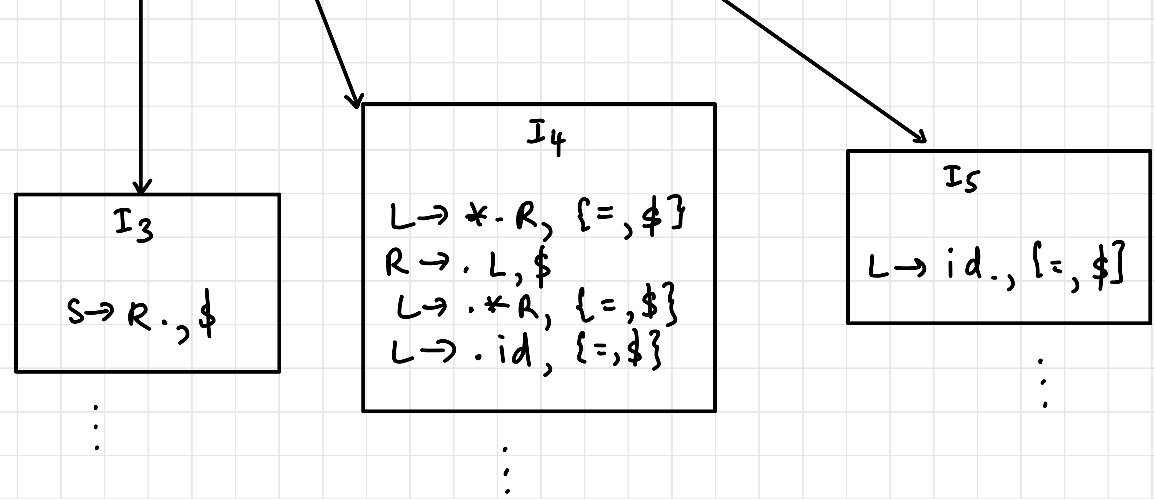
1. Augment & number

$S' \to S$
1. $S \to L = R$
2. $S \to R$
3. $L \to *R$
4. $L \to id$
5. $R \to L$

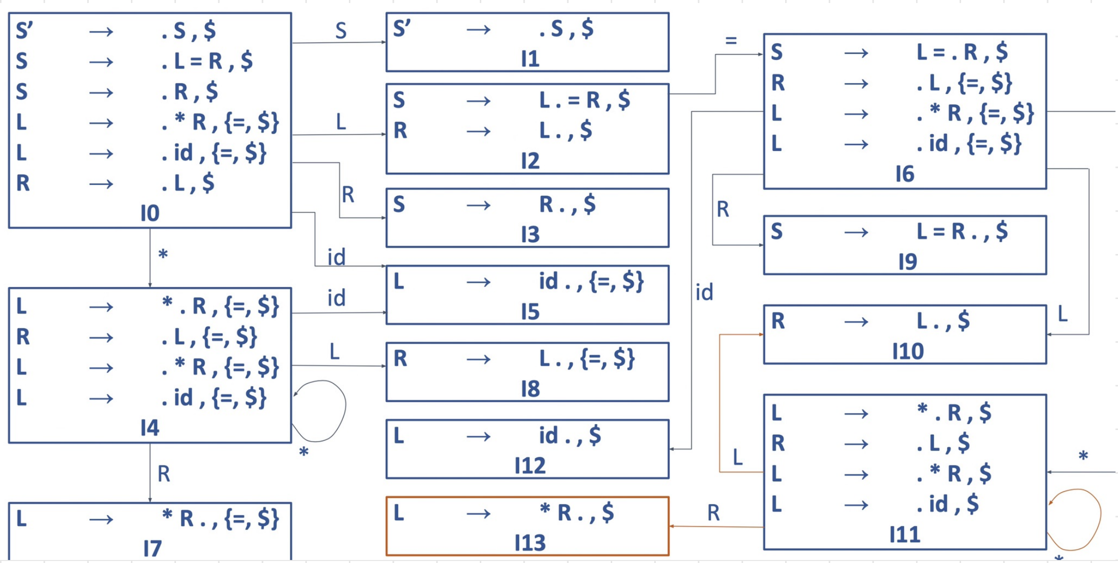
FIRST(S) = $*$, id
FIRST(L) = $*$, id
FIRST(R) = $*$, id

2. Algorithm





Complete DFA



- Parsing & table: same algorithm

4. LALR(1)

- In CLR automata, there will be some states with same LR(0) itemsets but diff lookaheads
- Merge the lookaheads (union)
- Watch out for r/r conflicts (weaker than CLR parsers)
  - upon merge, only r/r conflicts possible
  - CLR states may have s/r or r/r conflicts
- Reduce no. of states
- Yacc is LALR parser

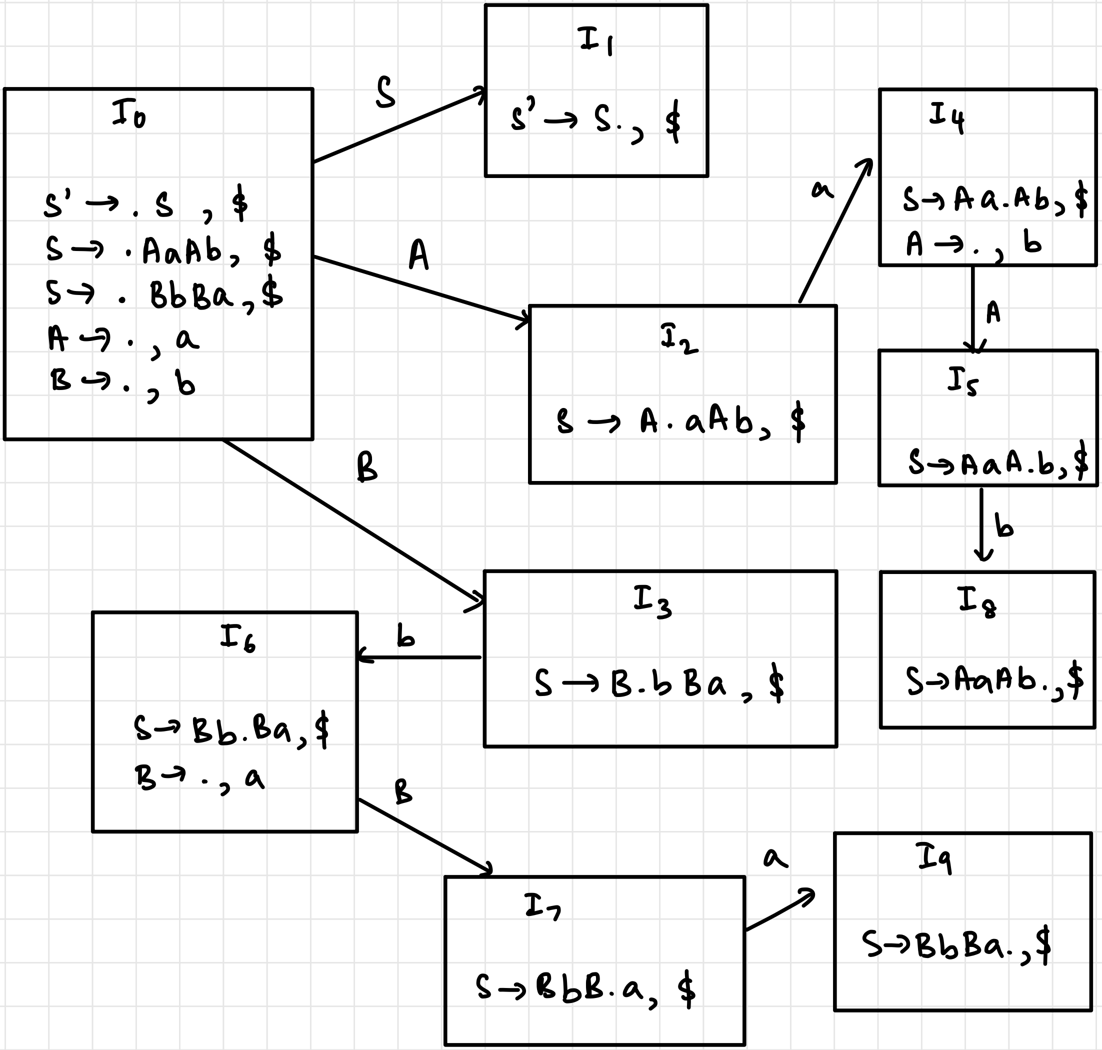
Q: Create automaton and table for

$S \to AaAb \mid BbBa$
$A \to \lambda$
$B \to \lambda$

1. Augment & no.

$S' \to S$
1. $S \to AaAb$
2. $S \to BbBa$
3. $A \to \lambda$
4. $B \to \lambda$

2. Algorithm



I0
S' → . S , $
S → . AaAb, $
S → . BbBa, $
A → . , a
B → . , b
S
I1
S' → S. , $
A
I2
S → A . aAb, $
a
I4
S → Aa.Ab, $
A → . , b
A
I5
S → AaA.b, $
b
I8
S → AaAb. , $
B
I3
S → B.bBa , $
b
I6
S → Bb.Ba, $
B → . , a
B
I7
S → BbB.a, $
a
I9
S → BbBa. , $

3. Parsing table (CLR)

state	Actions			Goto		
	a	b	$	S	A	B
0	r_3	r_4		1	2	3
1			accept			
2	S_4					
3		S_6				
4		r_3			5	
5		S_8				
6	r_4					7
7	S_9					
8			r_1			
9			r_2			

∴ CLR grammar

Q: Is the grammar CLR and LALR?

$S \to Aa \mid bAc \mid Bc \mid bBa$
$A \to d$
$B \to d$

1. Aug & num

$S' \to S$
1. $S \to Aa$
2. $S \to bAc$
3. $S \to Bc$
4. $S \to bBa$
5. $A \to d$
6. $B \to d$

2. Algorithm

I_0 $\xrightarrow{S}$

$S' \rightarrow .S, \$$
$S \rightarrow .Aa, \$$
$S \rightarrow .bAC, \$$
$S \rightarrow .Bc, \$$
$S \rightarrow .bBa, \$$
$A \rightarrow .d, a$
$B \rightarrow .d, c$

⋮

SUMMARY OF TD-BUP

Parser	LR(0)	SLR(1) or SLR (Simple LR)	LALR(1) or LALR (LookAhead LR)	CLR(1) or CLR or LR(1) or LR (Canonical LR)
DFA	LR(0) automata LR(0) item		LR(1) automata LR(1) item = LR(0) item + lookahead	
Reduce Move Placement in Action part of the Parsing table	Place the reduce move in the entire row for the state that contains a final item	Place the reduce move only in the Follow(LHS).	Place the reduce move in the lookahead	Place the reduce move in the lookahead
Power	Least Powerful	More powerful than LR(0)	More powerful than SLR	Most Powerful
Number of States	n : Number of States in a parser n(LR(0)) = n(SLR) = n(LALR) <= n(CLR)			