01-10-19

# TDP

## Left Recursion

Grammar in which variables in RHS occurs ~~too~~ soon after LHS
i.e, $A \rightarrow A\alpha / \beta$.

- Left Recursion is used for left associativity

Some parsers do not accept left recursion.

$\therefore A \rightarrow A\alpha / \beta$

$\Downarrow$

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' / \epsilon$ - Elimination of left recursion

Ex:

exp → exp addop term / term.
A → A α β

$\Downarrow$

$\therefore$ exp → term exp'
exp' → addop term exp' | ϵ

term → term mulop factor | factor

$\Downarrow$

term → factor term'
term' → mulop ~~term~~ factor term' | ϵ

self ↑

Noted

- Left Recursion is present in grammar rules, of the form $A \rightarrow A\alpha / \beta$ where $\alpha$ and $\beta$ are strings of terminals and non-terminals and $\beta$ doesn't begin with A

To remove the left recursion, we rewrite the grammar rule into two rules, one that generates $\beta$ first and one that generates repetitions of $\alpha$, using right recursion instead of left recursion

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

In general, $A \to A\alpha_1 \mid A\alpha_2 \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \ldots \mid \beta_m$

$\Downarrow$

$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \epsilon$

Q Remove left recursion from following, also write the grammar

1) $G_1$

stmt_seq → stmt_seq ; stmt | stmt

stmt → s

$\Downarrow$

stmt_seq → stmt stmt_seq'

stmt_seq' → ; stmt stmt_seq' | $\epsilon$

stmt → s

$L(G_1) = \{ s, s;s, s;s;s \ldots \}$

$G_1 = (\{ stmt\_seq, stmt \}, \{ ;, s \}, stmt\_seq, P)$

$L(G_1) = \{ s(;s)^n : n \geq 0 \}$

G2 & G3 - lisp programming

$G_3$

2) lexp → atom | list'
atom → number | id
list → (lexp_seq)
lexp_seq → ~~lseq~~ lexp-seq , lexp | lexp.

$G_3 = (\{lexp\_seq, lexp, atom, list\}, \{;, number, id\}, lexp\_seq, P)$

$L(G_3) = \{4, 555, a, aa, (5) \ldots\}$

lexp → atom | list
atom → number | id
list → (lexp_seq)
lexp_seq → lexp_seq , lexp | lexp
⇓
lexp_seq → lexp lexp_seq'
lexp_seq' → , lexp lexp_seq' | ϵ

$L(G_3) = \{\ldots num), (50, 60), (300, A', Z) \ldots\}$

3) G2.
lexp → atom | list
atom → number | id
list → (lexp_seq)
lexp_seq → lexp_seq lexp | lexp
⇓
lexp_seq → lexp lexp_seq'
lexp_seq' → lexp lexp_seq' | ϵ

$L(G_2) = \{ 800, \text{eight} \ldots, (800\ A\ z), (400(44,8,J)) \ldots \}$

Left factoring (to remove common prefixes)

- Productions have common prefixes. Such productions are not accepted by some parser

Ex: $A \to \alpha\beta \mid \alpha\gamma$

$\downarrow$

$A \to \alpha A'$

$A' \to \beta \mid \gamma$

1) $G_1$

stmt_seq → stmt ; stmt_seq | stmt

stmt → s

$\Downarrow$

stmt_seq → stmt stmt_seq'

stmt_seq' → ; stmt_seq | ∈

2) $G_3$

if_stmt → if (exp) stmt | if (exp) stmt else stmt

exp → 0 | 1

stmt → s

$\Downarrow$

if_stmt → if (exp) stmt if_stmt'

if_stmt' → ∈ | else stmt

3) G2.

lexp → atom | list

atom → number | id

list → (lexp_seq)

lexp_seq → lexp lexp_seq | lexp

⇓

lexp_seq → lexp lexp_seq'

lexp_seq' → lexp_seq | ∈

Q. Validate strings if belong to $L(G)$ or not

1) A → (A) A | ∈

2) A → aSb | ∈

3) declaration → type var_list

type → char | int | float

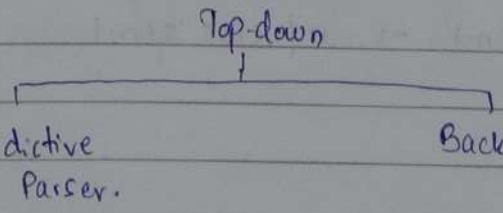
var_list → id var_list.

09-10-19 Parsing.

Top-down Bottom-up

1. Top-down

Parsing begins from root following a pre-order traversal



- Predictive is efficient in comparison with Backtracking but is less ~~more~~ powerful

Predictive
- Recursive descent
- LL(1)

2. Bottom-up (Shift-reduce parsers)

- Bottom-up are more efficient and powerful in comparison with top-down.

Bottom-up
- SLR(1)
- LR(1)
- LALR(1) (yacc tool uses LALR technique)

- LALR efficient among 3.

- LL(1)
  - scans input from left to right
  - Left most derivation is used for parsing
  - no. of look ahead symbols.

Ex: ~~S(S)~~ $S \to (S)S \mid \epsilon$

$G = (\{S\}, \{(,)\}, S, P)$

$L(G) = \{\epsilon, (), (()), ()() \cdots\}$

S → (S)S
S → ϵ

| | Parsing Stack | Input | Action |
|---|---|---|---|
| 1. | $S | ()$ | S → (S)S |
| 2. | $S)S( (reversed) | ()$ | Match, advance the i/p and pop matched string |
| | Top of stack is '(' and pointer in input is pointing to '(' : Match | | |
| 3. | $S)S | )$ | S → ϵ |
| 4. | $S). | )$ | Match |
| 5. | $S | $ | S → ϵ |
| 6. | $ | $ | accept. |

Info obtained from Parsing table

Steps involved to Build LL(1) Parser

1. Check whether the grammar is in the required form for LL(1) Parsing.
   (Grammar should not contain Left Recursion productions or productions with common prefixes or both)

2. If G is not in required form, reduce /bring it to required form by eliminating left recursion or common prefix. G' is obtained.

3. Compute first and follow sets.

4. Construct the LL(1) Parsing table using algorithm.

5. Verify parsing table by parsing valid strings using the parsing stack.

- Computation of first and follow sets

1. first(x) = x  if x is a terminal

2. $X \to y_1 y_2 \cdots y_k$  if x is a non-terminal
   then,
   first(x) = first($y_1$)

3. if $X \to \epsilon$ , first(X) = {ε}

Ex: $e \to e$ addop $t \mid t$
addop → + | -
$t \to t$ mulop $f \mid f$
mulop → *
$f \to$ id | num | (e).

$e \to e$ addop $t \mid t$
⇓
$e \to t e'$
$e' \to$ addop $t\ e' \mid \epsilon$

$t \to t$ mulop $f \mid f$
⇓
$t \to f t'$
$t' \to$ mulop $f\ t' \mid \epsilon$

∴ G' is
$e \to t e'$
$e' \to$ addop $t\ e' \mid \epsilon$
addop → + | -
$t \to f t'$
$t' \to$ mulop $f t' \mid \epsilon$

mulop → *

f → num | (e)

| | Pass-1 | Pass-2 |
|---|---|---|
| e → te' | first(e) = first(t) | first(e') = {+, -, ε} |
| e' → addop t e' | first(e') = first(addop) | first(t) = {num, (} |
| t → f t' | first(t) = first(f) | first(t') = {*, ε} |
| t' → mulop f t' | first(t') = first(mulop) | |
| e' → ε | first(e') = {ε, first(addop)} | |
| t' → ε | first(t') = {ε, first(mulop)} | |
| addop → + \| - | first(addop) = {+, -} | |
| mulop → * | first(mulop) = {*} | |
| f → num | first(f) = {num} | |
| f → (e) | first(f) = {num, (} | |

Pass-3

first(e) = {num, (}

Ex: stmt_seq → stmt_seq ; stmt | stmt

stmt → s

⇓

stmt_seq → stmt stmt_seq'

stmt_seq' → ; stmt stmt_seq' | ε

stmt → s

Pass -1

stmt_seq → stmt stmt_seq

first (stmt_seq) = first(stmt)

stmt_seq' → ; stmt stmt_seq'

stmt_seq' → ϵ

stmt → s

first (stmt-seq') = {;}

first (stmt-seq') = {ϵ, ;}

first (stmt) = {s}.

Pass - 2

first(stmt_seq) = {s}

A → A a A | a

A → a A'

A' → a A A' | ϵ

S → X A B

A → a a A | ϵ

X → a

B → b b B | ϵ

14-10-19

- Computation of follow sets

Given a non-terminal A, the set follow of A consisting of terminals and $ is defined as

R1 i) if A is a start symbol / start variable then $ is in follow(A)

R2 ii) if there is a production $A \to \alpha B \beta$ then first($\beta$) - {$\epsilon$} is in follow(B)

R3 iii) if there is a production $A \to \alpha B$ or $A \to \alpha B \beta$ where first($\beta$) contains $\epsilon$ then whatever is in follow(A) will be in follow(B) i.e. follow(A) = follow(B)

Ex:
exp → term exp'
exp' → addop term exp'
term → factor term'
term' → mulop factor term'
exp' → $\epsilon$
term' → $\epsilon$
addop → + | -
mulop → *
factor → (exp) | number

| | first | follow |
|---|---|---|
| exp | {(, number} | {$, )} |
| exp' | {+, -, $\epsilon$} | {$, )} |
| term | {(, number} | {+, -, $, )} |
| term' | {*, $\epsilon$} | {+, -, $, )} |
| addop | {+, -} | {(, number} |
| mulop | {*} | {(, number} |
| factor | {(, number} | {*, +, -, $, )} |

| | Pass-1 | Pass-2 |
|---|---|---|
| exp → term exp' | follow(exp) = {$} R1 | |
| | follow(exp') = {$} R3 | follow(exp') = follow(exp) = {$, )} |
| exp' → addop term exp' | follow(term) = first(exp') - {ε} = {+, -} R2 | |
| | follow(term) = {+, -, first(exp)} = {+, -, $} R3 | follow(term) = {+, -, first(exp)} = {+, -, $, )} |
| term → factor term' | follow(term') = follow(term) = {+, -, $} R3 | follow(term') = follow(term) = {+, -, $, )} |
| term' → mulop factor term' | follow(factor) = first(term') - {ε} = {*} R2 | |
| | follow(factor) = {*, follow(term')} = {*, +, -, $) R3 | follow(factor) = {*, follow(term')} = {*, +, -, $, )} |
| factor → (exp) | follow(exp) = {$, first())} = {$, )} R2 | |

- To compute follow(addop) and follow(mulop)

  Take exp' → addop term exp'

  A → B β where α = ε

  By R2, follow(addop) = first(factor) - {ε}
  = {(, number}

  III 4, follow(mulop) = first(factor) - {ε}
  = {(, number}

- Parsing table is a 2D Array

* LL(1) Parsing Table.

| M(N,T) ↓ non-terminals → terminals | + | - | * | ( | ) | number | $ |
|---|---|---|---|---|---|---|---|
| exp | | | | exp → term exp' | | exp → term exp' | |
| exp' | exp' → addop term exp' | exp' → addop term exp' | | | exp' → ϵ | | exp' → ϵ |
| term | | | | term → factor term' | | term → factor term' | |
| term' | term' → ϵ | term' → ϵ | term' → mulop factor term' | | term' → ϵ | | term' → ϵ |
| addop | addop → + | addop → - | | | | | |
| mulop | | | mulop → * | | | | |
| factor | | | | factor → (exp) | | factor → number. | |

- Empty cells represent Errors.

- Repeat the follow 2 steps for each non-terminal A and the production choice $A \to \alpha$

1. For each token a in first($\alpha$), add $A \to \alpha$ to the entry M[A, a]
2. If $\epsilon$ is in first($\alpha$), for each element a of follow(A) where a can be a token or $, add $A \to \alpha$ to the entry M[A, a]

| Parsing Stack | Input | Action |
|---|---|---|
| $exp (Start) | 2+3$ | exp → term exp' |
| $exp' term (reversed) | 2+3$ | term → factor term' |
| $exp' term' factor | 2+3$ | factor → number. |
| $exp' term' number | 2+3$ | Match |
| $exp' term' | +3$ | term' → ϵ |
| $exp' | +3$ | exp' → addop term exp' |
| $exp' term addop | +3$ | addop → + |
| $exp' term + | +3$ | Match |

| | | |
|---|---|---|
| $ exp term | 3 $ | term → factor term' |
| $ exp' term' factor | 3 $ | factor → number. |
| $ exp' term' number | 3 $ | Match. |
| $ exp' term' | $ | term → ∈ |
| $ exp' | $ | exp' → ∈ |
| $ | $ | ACCEPT. |

HW

| Parsing Stack | Input | Action. |
|---|---|---|
| | 12 * 3 + 6 | |

16-10-19

$A \to \beta A'$
$A' \to \alpha A' | \epsilon$.

$A \to \alpha A'$
$A' \to \beta | \gamma$

Q1) Construct LL(1) Parsing Table

1. stmt_seq → stmt ; stmt_seq | stmt
   stmt → s.

⇓

stmt_seq → stmt stmt_seq'
stmt_seq' → ; stmt_seq

stmt_seq → stmt stmtseq'
stmt_seq' → ; stmt_seq | ϵ
stmt → s.

Computation of First sets

Pass-1

stmt_seq → stmt stmt_seq'
stmt_seq' → ; stmt_seq
stmt_seq' → ϵ
stmt → s

first (stmt_seq) = first (stmt)
first (stmt_seq') = first (;) = {;}
first (stmt_seq') = {;, ϵ}
first (stmt) = {s}.

Pass - 2

first (stmt_seq) = {s}

Computation of follow sets.

Pass-1

with

stmt_seq → stmt stmt_seq'

follow (stmt_seq) = {$} R1

follow (stmt_seq') = follow (stmt_seq)
= {$} R3

α - stmt B - stmt_seq

~~stmt_seq' → ; stmt_seq~~

follow (stmt) = first (stmt_seq') - {ϵ}
= {;} R2

B - stmt . β - stmt_seq'

follow (stmt) = {follow (stmt_seq), ;}
= {;, $} R3

B - stmt , β - stmt_seq'

∴

| | first | follow |
|---|---|---|
| stmt_seq | {s} | {$} |
| stmt_seq' | {;, ε} | {$} |
| stmt | {s} | {;, $} |

| M[N,T] | ; | s | $ |
|---|---|---|---|
| stmt_seq | | stmt_seq → stmt stmt_seq' | |
| stmt_seq' | stmt_seq' → ; stmt_seq | | stmt_seq' → ε |
| stmt | | stmt → s | |

L(G) = { s, s;s , s;s;s ... }

2. ~~stmt → if_stmt | other~~

~~if_stmt →~~

| Parsing Stack | Input | Action |
|---|---|---|
| $ stmt_seq | s;s $ | stmt_seq → stmt stmt_seq' |
| $ stmt_seq' stmt | s;s $ | ~~Match~~, stmt → s. |
| $ stmt_seq' s | s;s $ | Match |
| $ stmt_seq' | ;s $ | stmt_seq' → ; stmt_seq |
| $ stmt_seq ; | ;s $ | Match |
| $ stmt_seq | s $ | stmt_seq → stmt stmt_seq' |

| | | |
|---|---|---|
| $ stmt_seq' stmt | s $ | stmt → s |
| $ stmt_seq' s | s $ | Match |
| $ stmt_seq' | $ | stmt_seq' → ϵ |
| $ | $ | Accept. |

A → G

2) stmt → if_stmt | ~~other~~ s

if_stmt → if (exp) stmt | if (exp) else stmt

exp → 0 | 1

~~stmt → s~~

⇓

stmt → if_stmt | ~~other~~ s

if_stmt → if(exp) stmt if_stmt'

if_stmt' → ϵ

if_stmt' → else stmt

exp → 0 | 1

~~stmt → s~~

stmt
if_stmt
if_stmt'
exp

first set

stmt → if_stmt

stmt → ~~other~~ s

if_stmt → if (exp) stmt if_stmt'

if_stmt' → ϵ

if_stmt' → else stmt

exp → 0

exp → 1

~~stmt → s~~

Pass-1

first(stmt) = first (if_stmt)

first (stmt) = { first (if_stmt), ~~other~~ s }

first(if_stmt) = { if }

first (if_stmt') = { ϵ }

first (if_stmt') = { ϵ, else }

first(exp) = {0}

first (exp) = {0, 1}

~~first (stmt) = {s}~~

Pass-2

first(stmt) = {if}

first(stmt) = {if, ~~other~~ s}

Computation of follow sets.

Pass-1

stmt → if_stmt  follow(stmt) = {$}  R1

α A β  follow(if_stmt) = follow(stmt) = {$}  R3

if_stmt → if(exp) stmt if_stmt'  follow(stmt) = first(if_stmt') - {ε}
= {else, $}  R2

follow(exp) = {)}

α if(exp) stmt β if_stmt'  follow(if_stmt') = follow(if_stmt)
= {$}  R3

| | first | follow |
|---|---|---|
| stmt | {s, if} | {$, else} |
| if_stmt | {if} | {$, else} |
| if_stmt' | {else, ε} | {$, else} |
| exp | {0, 1} | {)} |

Pass-2

follow(if_stmt) = follow(stmt) = {$, else}

follow(if_stmt') = follow(if_stmt) = {$, else}

| M(N,T) | if | else | ( | ) | s | 0 | 1 | $ |
|---|---|---|---|---|---|---|---|---|
| stmt | stmt → if_stmt | | | | stmt → s | | | |
| if_stmt | if_stmt → if(exp) stmt if_stmt' | | | | | | | |
| if_stmt' | if_stmt' → ε | if_stmt' → else stmt | | | | | | if_stmt' → ε |
| exp | | | | | | exp → 0 | exp → 1 | |

- There is a conflict in M[if_stmt', else]. ~~Hence~~ Hence not LL(1) Parsing.

21|10|19. LL(1) Parsing table with error recovery entries.

| M[N,T] | ( | number | ) | + | - | * | $ |
|---|---|---|---|---|---|---|---|
| exp | exp → term exp' | exp → term exp' | POP | Scan | Scan | Scan | POP |
| exp' | Scan | Scan | exp' → ε | exp' → addop term exp' | exp' → addop term exp' | Scan | exp' → ε |
| addop | POP | POP | Scan | addop → + | addop → - | Scan | POP |
| term | term → factor term' | term → factor term' | POP | POP | POP | Scan | POP |
| term' | Scan | Scan | term' → ε | term' → ε | term' → ε | term' → mulop factor term' | term' → ε |
| mulop | POP | POP | Scan | Scan | Scan | mulop → * | POP |
| factor | factor → (exp) | factor → number | POP | POP | POP | POP | POP |

Given a non-terminal A at top of stack and an input token ~~ta~~ that is not in first(A) or follow(A), if $\epsilon$ is in first(A) there are 3 possible alternatives.

1. Pop non-terminal A from stack
2. Successively pop tokens from the input until a token is seen for which we can re-start the parse
3. Push a new non-terminal onto stack.

We choose alternative-1 if the current input token is $\$$. or is in follow(A). We choose alternative-2 if the current input token is not $\$$ and is not in first(A) $\cup$ follow(A). Alternative-3 is occasionally useful in special situations, but is rarely appropriate.

We indicate 1st action in parsing table by notation POP and the second by notation SCAN.

(Note that a pop action is equivalent to a reduction by $\epsilon$ production).

| Parsing stack | Input | Action |
|---|---|---|
| $\$$ exp | (2+*) $\$$ | exp → term exp' |
| $\$$ exp' term | (2+*) $\$$ | term → factor term' |
| $\$$ exp' term' factor | (2+*) $\$$ | factor → (exp) |
| $\$$ exp' term' ) exp ( | (2+*) $\$$ | Match. |
| $\$$ exp' term' ) exp | 2+*) $\$$ | exp → term exp' |
| $\$$ exp' term' ) exp' term | 2+*) $\$$ | term → factor term' |
| $\$$ exp' term' ) exp' term' factor | 2+*) $\$$ | factor → number |
| $\$$ exp' term' ) exp' term' number | 2+*) $\$$ | Match |
| $\$$ exp' term' ) exp' term' | +*) $\$$ | term' → $\epsilon$ |
| $\$$ exp' term' ) exp' | +*) $\$$ | exp' → addop term exp' |
| $\$$ exp' term' ) exp' term addop | +*) $\$$ | addop → + |
| $\$$ exp' term' ) exp' term + | +*) $\$$ | Match. |
| $\$$ exp' term' ) exp' term | *) $\$$ | Scan. (Error). |

| | | |
|---|---|---|
| $exp'term')exp'term | )$ | Pop (error) |
| $exp'term')exp' | ↑)$ | exp'→ϵ |
| $exp'term') | ↑)$ | Match. |
| $exp'term' | ↑$ | ~~pop (error)~~ term'→ϵ |
| $exp' | $ | ~~pop~~ exp'→ϵ |
| $ | $ | ~~Accept~~. |

| Parsing stack | Input | Action |
|---|---|---|
| $exp | 2+3(↑5*2)$ | exp→termexp' |
| $exp'term | 2+3(↑5*2)$ | term→factorterm' |
| $exp'term'factor | 2+3(5*2)$ | factor→number. |
| $exp'term'number | 2+3(5*2)$ | Match |
| $exp'term' | +3(5*2)$ | term'→ϵ |
| $exp' | +3(5*2)$ | exp'→addoptermexp' |
| $exp'termaddop | +3(5*0)$ | addop→+ |
| $exp'term+ | +3(5*2)$ | Match |
| $exp'term | 3(5*2)$ | term→factorterm' |
| $exp'term'factor | 3(5*2)$ | factor→number |
| $exp'term'number | 3(5*2)$ | Match |
| $exp'term' | (5*2)$ | Scan (Error) |
| $exp'term' | 5*2)$ | Scan |
| $exp'term' | *2)$ | term'→mulopfactorterm' |
| $exp'term'factor mulop | *2)$ | mulop→* |
| $exp'term'factor * | *2)$ | Match |
| $exp'term'factor | 2)$ | factor→number |
| $exp'term'number | 2)$ | Match |
| $exp'term' | )$ | term'→ϵ |
| $exp' | )$ | ~~Pop~~ exp'→ϵ |
| $ | )$ | ')' is simply discarded |
| $ | $ | ~~Accept~~ |

10-19.

# Bottom-Up Parser

→ Even if CFG is used, grammar is restricted as to not have left recursion & common prefixes. Hence LL(1) Parsing is less efficient & less powerful.

→ Bottom-up Parsers also called as shift/Reduce parsers because shift/Reduce actions are performed

| Parsing Stack | Input string | Action |
|---|---|---|
| $ Initial state | i/p sym $ | shift/Reduce |
| $ start variable | $ | Accept. |

→ LR(1) — SLR(1) (Simple LR (1))
LR(1) (Cannonical LR(1))
LALR(1) (Look ahead LR (1))

- read from left to right
- no. of look ahead symbols
- trace of rightmost derivation

## SLR(1). Parsing.

Steps involved in building SLR(1) Parser.

1. Augument a new production $S' \to S$ with $S'$ as a new start variable
2. Compute collection of LR(0) item sets.
3. Construct DFA of LR(0) items.

4. Construct SLR(1) Parsing table using SLR(1) algorithm.
5. Verify the parsing table by taking a valid input string and parse it using a parsing stack.

- LR(0) item : 0 indicates no look ahead info used
  → is a production itself say $A \to \alpha X \beta$
  $X \in T$
  $\alpha, \beta \in (V \cup T)^*$

$A \to .\alpha X \beta$ is called LR(0) item, initial item
(when $\alpha$ is to be processed next
and when $\alpha$ is processed $A \to \alpha . X \beta$.
X is expected to the processed next
$A \to \alpha X . \beta$ when X is processed
↓ : so on
$A \to \alpha X \beta .$ is called complete item
which is a processed string called handle i.e, the handle $\alpha X \beta$ can be replaced by A.
i.e, Action is reduced

when x is being processed i.e, when a terminal is being processed Shift action is done and x is pushed on stack

Ex: G: $E \to E+n \mid n$
$\therefore G = (E, \{+, n\}, E, P)$
$L(G) = \{ n, n+n, 23, 46, 26+50, 189+278+999 \ldots \}$

Step 1.
1. $E' \to E$
2. $E \to E+n$ : $G'$
3. $E \to n$

Step 2:

$E' \to .E$

$E \to .E+n$

$E \to .n$ : $(I_0)$.

(If a non-terminal occurs after ., find closure of it i.e, write down all productions of E and make them as initial items)

$Goto(I_0, E) = I_1$ (In $I_0$, if E occurs after ., then list down the productions by advancing the .)

$I_1$ : $E' \to E.$

$E \to E.+n$

$Goto(I_0, n) = I_2$

$I_2$ : $E \to n.$

$Goto(I_1, +) = I_3$

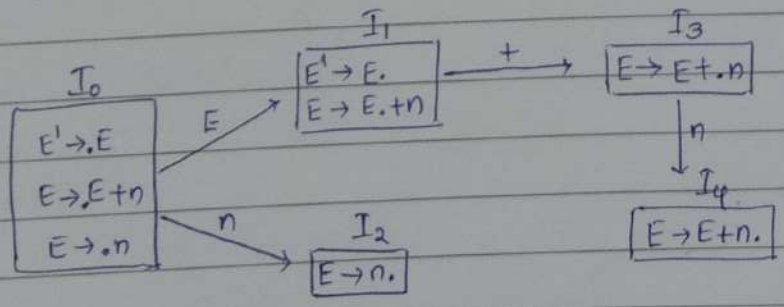
$I_3$ : $E \to E+.n$

$Goto(I_3, n) = I_4$

$I_4$ : $E \to E+n.$

$I_0, I_1 \ldots I_4$ are called collection of LR(0) item sets.

Step 3:




(Initial State)

∴ DFA of LR(0) items is obtained

Step 4: ~~DLR~~ SLR(1) Parsing Table

| States | Input Symbols | | | Goto( ) |
|---|---|---|---|---|
| | + | n | $ | E |
| 0 | | S2 | | 1 |
| 1 | S3 | | accept | |
| 2 | r3 | | r3 | |
| 3 | | S4 | | |
| 4 | r2 | | r2 | |

→ when item is $S' \to S.$ , Accept is written under $ as it is a complete item with start variable

→ when $A \to \alpha.$ where $\alpha$ is a ~~n~~ terminal is obtained then reduce action is performed and for every a in follow(A), we write reduce by production no. Productions are numbered in step 1.

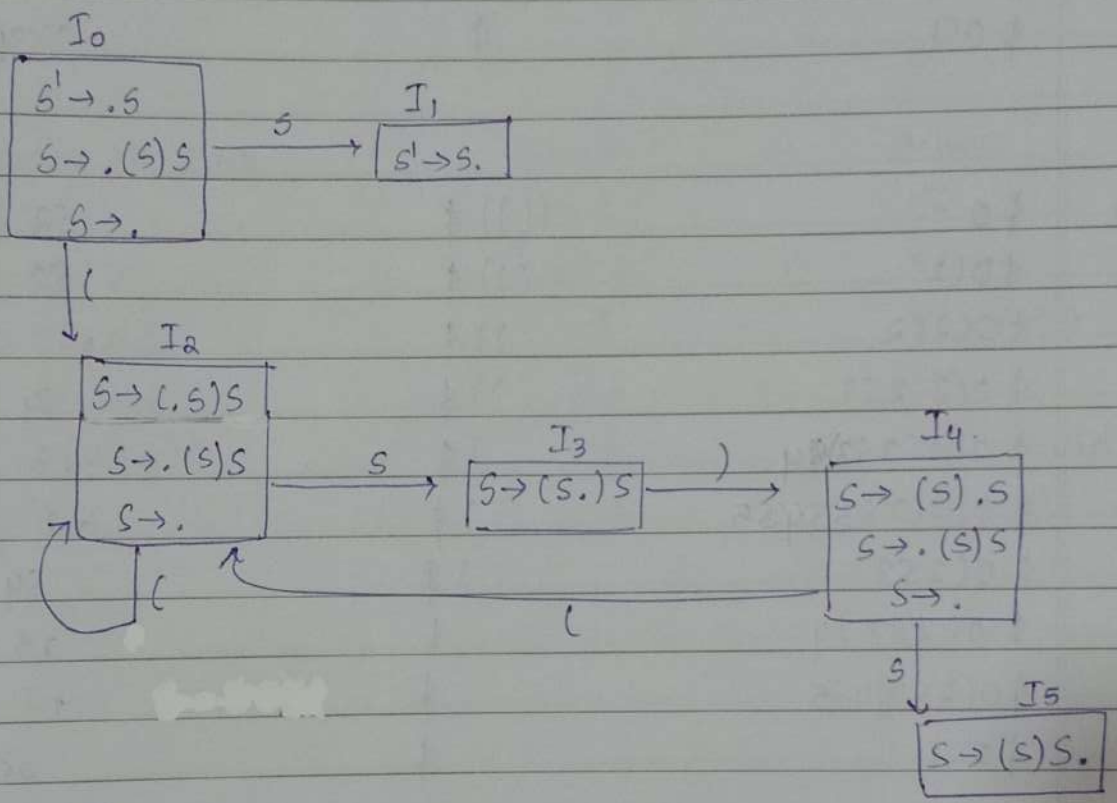
follow(E) = {$, +}

30-10-19

Q Construct SLR(1), PT and verify the input string ()

$S \to (S)S \mid \epsilon$

→ G' : ① $S' \to S$
② $S \to (S)S$
③ $S \to \epsilon$




Parsing Table

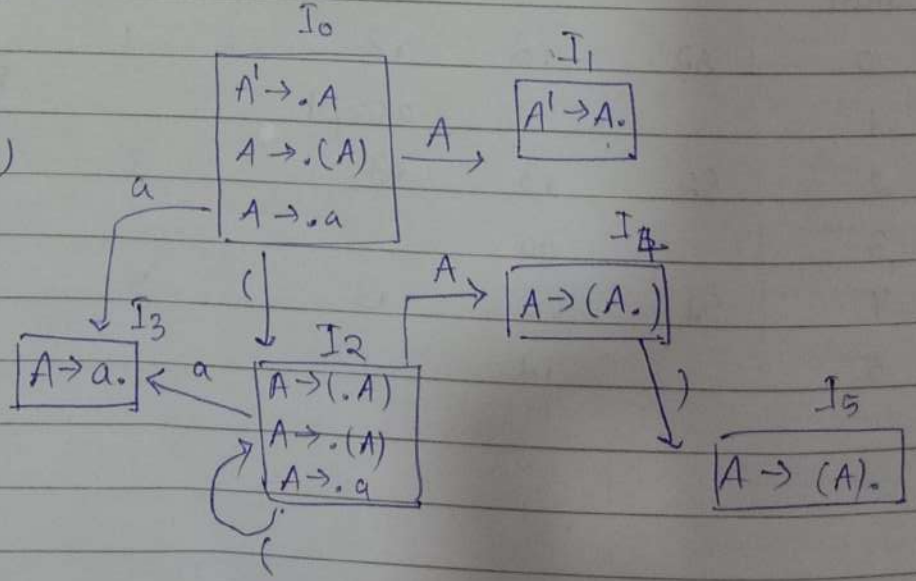
| State | Input Symbols ( | ) | $ | Goto S | follow(S) = {), $} |
|---|---|---|---|---|---|
| 0 | S2 | r3 | r3 | 1 | |
| 1 | | | accept | | when A'→A. |
| 2 | S2 | r3 | r3 | 3 | is reached, |
| 3 | | S4 | | | action is |
| 4 | S2 | r3 | r3 | 5 | accept under |
| 5 | | r2 | r2 | | $ |

| | Parsing stalk | Input | Action |
|---|---|---|---|
| ① | $0 | ()$ | S2 |
| | $0(2 | )$ | r3 S→ϵ |
| | $0(253 | )$ | S4 |
| | $0(253)4 | $ | r3 S→ϵ |
| | $0(253)455 | $ | r2 S→(S)S |
| | $051 | $ | accept |

| | | | |
|---|---|---|---|
| ② | $0 | (())$ | S2 |
| | $0(2 | ())$ | S2 |
| | $0(2(2 | ))$ | r3 S→ϵ |
| | $0(2(253 | ))$ | S4 |
| | $0(2(253)4 | )$ | r3 S→ϵ |
| | $0(2(253)455 | )$ | r2 S→(S)S |
| | $0(253 | )$ | S4 |
| | $0(253)4 | $ | r3 S→ϵ |
| | $0(253)455 | $ | r2 S→(S)S |
| | $051 | $ | accept |

Q. G: A → (A) | a

G': ① A' → A
② A → (A)
③ A → a



| States | ( | ) | a | $ | A | follow(A) = |
|---|---|---|---|---|---|---|
| 0 | S2 | | S3 | | 1 | {$, )} |
| 1 | | | | accept | | |
| 2 | S2 | | S3 | | 4 | |
| 3 | | r3 | | r3 | | |
| 4 | | S5 | | | | |
| 5 | | r2 | | r2 | | |

| Parsing stack | Input | Action |
|---|---|---|
| $0 | (a) $ | S2 |
| $ 0(2 | a) $ | S3 |
| $ 0(2a3 | ) $ | r3 A → a |
| $ 0(2A4 | ) $ | S5 |
| $ 0(2A4)5 | $ | r2 A → (A) |
| $ 0A1 | $ | accept. |

:= assignment op in Pascal
~
one token.

02-11-19

SLR(1) grammar identification.

A grammar is CLR(1) iff for any state S, the following 2 conditions are satisfied.

i) for any item $A \to \alpha . x \beta$ in state S with x as terminal there is no complete item where $B \to \gamma .$ in S with x in follow(B)

ii) For any two complete items $A \to \alpha .$ and $B \to \beta .$ in S, follow(A) $\cap$ follow(B) is empty.

A violation of i condition represents a shift / reduce conflict. A violation of ii) condition represents a reduce-reduce conflict. If either of condition is not satisfied then grammar is not an SLR(1) grammar.

Q. G: stmt → call_stmt | assign_stmt
call_stmt → id
assign_stmt → var := exp
var → var [exp] | id
exp → var | num

⇓

G': ① $S' \to S$ ② $S \to C \mid A$
③ $C \to id$
④ $A \to V := E$
⑤ $V \to V[E] \mid id$
⑥ $E \to V \mid n$

$I_0$.

$S' \to .S$

$S \to .C$

$S \to .A$

$C \to .id$

$A \to .V := E$

$V \to .V[E]$

$V \to .id$

$\xrightarrow{id}$ $I_1$

$C \to id.$

$V \to id.$

follow(C) = {$}

follow(V) = {$, [, :=}

follow(C) ∩ follow(V) = {$}

∴ M[1, $] will have entries ~~reduce~~ r(C → id) and r(V → id) (reduce-reduce conflict)

∴ Given G is not SLR(1).

- LR(1) Parser, Cannonical LR(1) Parser (CLR(1)), General LR(1) Parser

→ Early decision making and error detection due to one look ahead symbol used.

LR(1) useds LR(1) item sets for constuction of DFA.

unlike SLR(1) which uses LR(0) item setr.

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

LR(0) item  lookahead sym

LR(1) item

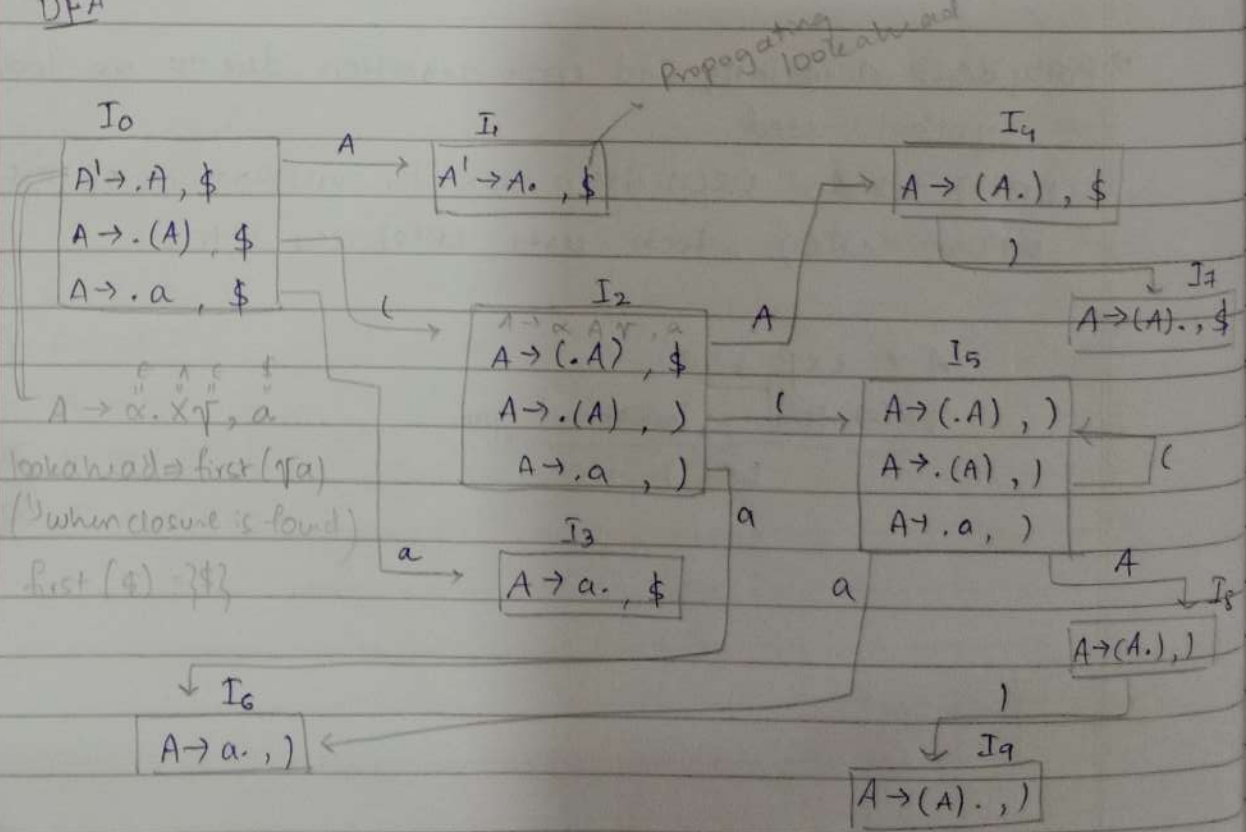
Steps

1. Add a new start & variable $S' \to S$
2. Compute collection of LR(1) item sets
3. DFA of LR(1) item sets
4. Construct LR(1) PT using alg.
5. Verify PT & taking a valid input string using parsing stack

Q G: $A \to (A) \mid a$

G': 1. $A' \to A$
2. $A \to (A)$
3. $A \to a$

DFA



| State | I/P symbols (T) | | | | Goto (NT) |
|---|---|---|---|---|---|
| | ( | ) | a | $ | A |
| 0 | s2 | | s3 | | 1 |
| 1 | | | | accept | |
| 2 | s5 | | s6 | | 4 |
| 3 | | r3 | | r3 | |
| 4 | | s7 | | | |
| 5 | s5 | | s6 | | 8 |
| 6 | | r3 | | | |
| 7 | | | | r2 | |
| 8 | | s9 | | | |
| 9 | | r2 | | | |

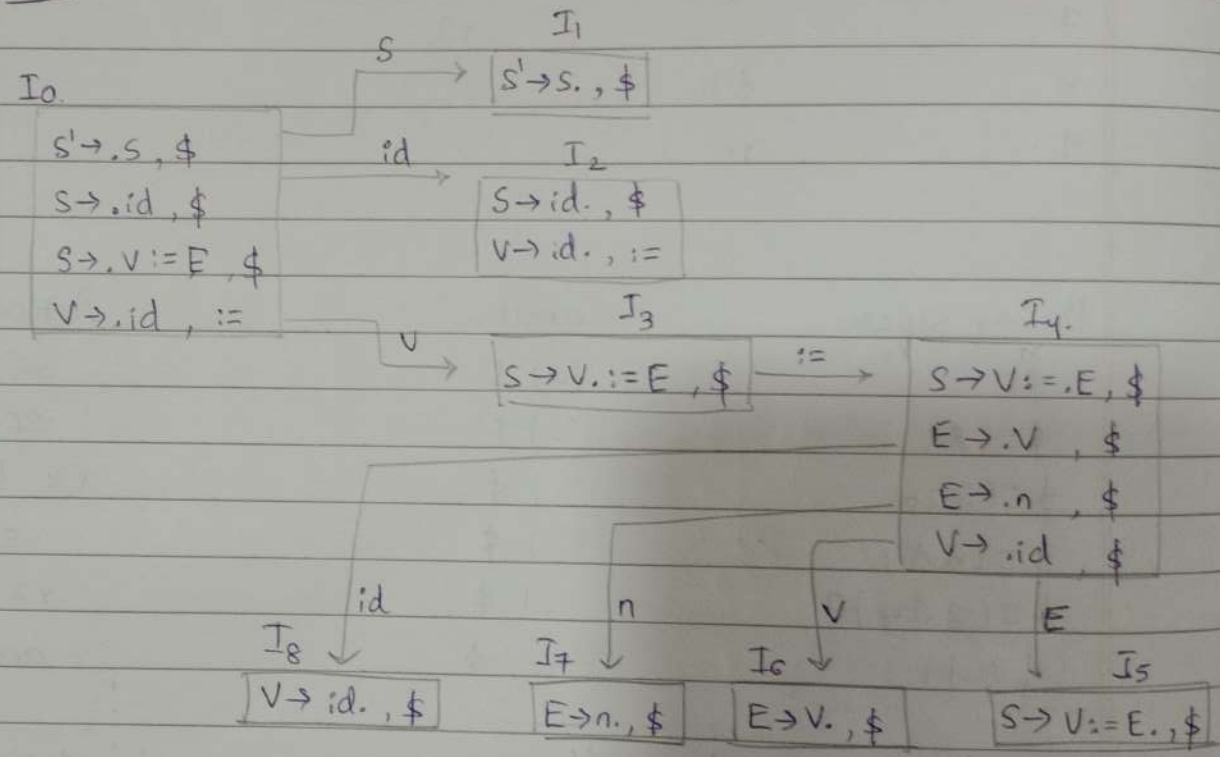
| Parsing stack | input | Action |
|---|---|---|
| $0 | (a) $ | s2 |
| $0(2 | a) $ | s6 |
| $0(2a6 | ) $ | r3 (A → a) |
| $0(2A4 | ) $ | s7 |
| $0(2A4)7 | $ | r2 (A → (A)) |
| $0A1 | $ | accept |

| Parsing stack | input | Action |
|---|---|---|
| $0 | ((a)) $ | s2 |
| $0(2 | (a)) $ | s5 |
| $0(2(5 | a)) $ | s6 |
| $0(2(5a6 | )) $ | r3(A→a) |
| $0(2(5A8 | )) $ | s9 |
| $0(2(5A8)9 | ) $ | r2(A→(A)) |
| $0(2A4 | ) $ | s7 |
| $0(2A4)7 | $ | r2(A→(A)) |
| $0A1 | $ | accept |

Q. $S \to id$
$S \to V := E$
$V \to id$
$E \to V \mid n$

$G'$: $S' \to S$ ①
$S \to id$ ②
$S \to V := E$ ③
$V \to id$ ④
$E \to V$ ⑤
$E \to n$ ⑥

DFA




| States | id | := | n | $ | S | V | E |
|---|---|---|---|---|---|---|---|
| 0 | S2 | | | | 1 | 3 | |
| 1 | | | | accept | | | |
| 2 | | r4 | | r2 | | | |
| 3 | | S4 | | | | | |
| 4 | S8 | | S7 | | | 6 | 5 |
| 5 | | | | r3 | | | |
| 6 | | | | r5 | | | |
| 7 | | | | r6 | | | |
| 8 | | | | r4 | | | |

| Parsing Stack | Input | Action |
|---|---|---|
| $0 | 2+8$ | S2 |
| $0n2 | +8$ | r3 (E→n) |
| $0E1 | +8$ | S3 |
| $0E1+3 | 8$ | S4 |
| $0E1+3n4 | $ | r2 (E→E+n) |
| $0E1 | $ | accept |

Q. S' → S
S → (S)S
S → ∈