COMPILER DESIGN.

better error identification than compiler

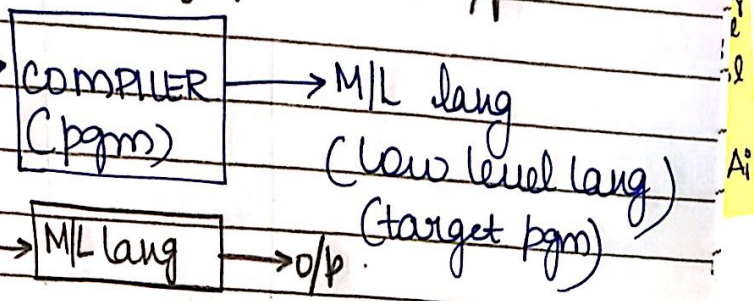
INTERPRETER: statement by statement

12.12

(first book in the syllabus)

Compiler executes the entire pgm

↳ faster at mapping i/p → o/p.



→ Translates one representation of program into another representation.

→ Fortran I → first compiler written

→ Source code - readable, expressive

optimize code for machine level → remove redundancy

Abstraction for compiler: (Low level lang)

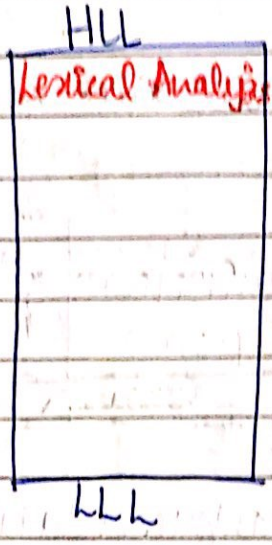
memory, register, stack, opcodes, addressing modes

Abstraction for HLL

variable, operator, expression

assembly lang → closer to HL lang than C++.

Phases of compiler (7): seq of steps to go from HLL → LLL



sequence
Logically connected
after each step there is a diff repr of lang.

error detection,
Compiler - optimises code
↳ removes the redundant code.
(uses diff order of execution but doesn't understand logic of pgm)

Editor → source pgm may be split into separate files. Task of collecting the source pgm is done by pre-processor.

Pre processing → (MACRO Defn are replaced expanded)

Compiler → produces an assembly lang as o/p

Assembler → relocatable m/c code as o/p

Linker/loader (Libraries are linked with pgm) - LINKER
used

Loader - puts all executable

08.12.17

UNIT I

WORLD STAR™ Date: Page:

Structure of Compiler

Analysis
Synthesis

Lexical Analyzer
↓ character stream (or) src code
↓ token stream
Syntax Analyzer
↓ syntax tree
Semantic Analyzer
↓ Disambiguated syntax tree
Intermediate code generator
↓ intermediate representation
M/C independent code optimized
↓ intermediate representation
Code generator
↓
M/C dependent code optimization
↓
Target M/C code.

changes

* set of valid characters in a lang
* develop words → valid or not

syntactically & semantically correct.

Analysis part (front end of compiler) (deals with src code)

makes the pgm to run faster, use less resource

perform transformations on intermediate representations so as to produce a better target program

Eg: → reads from left to right

position = initial + rate * 60

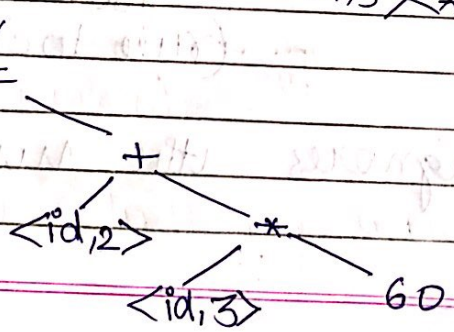
TOKENS → (Symbol table)

<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

<id, 1> → identifier

addof

SYNTAX TREE →



intermediate code

```
t1 = int to float (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

↓

```
t1 = id3 * 60.0
id1 = id2 + t1
```

} optimised code

↓

```
LDF   R2, id3
MULF  R2, R2, #60
ADF   R1, R1, R2
STF   id1, R1
```

} assembly level inst

~~LDF R1, id2~~

lexical analyser : separates tokens → basic unit of pgming lang

dividing up the char as

variables, identifiers, operators

{ ~~syntax analyser~~

* how you validate the structure of pgm
* (use finite automata)
  ↓
  use an RE (rule to define)
* ignores the non read characters (white space)

modeled using RE + validated using automa

SYMBOL TABLE : look up & insert

it needs a look ahead char.

eg: a = b+c

↓

sees '=' & decides that 'a' is an identifier

"lookahead operation" → then knows its valid / not then returns token to input buffer.

Syntax analyzer

Checks the struct of individual words in source pgm then it generates a parse tree. Otherwise it throws an error.

↳ then related to context of previous word.

eg: We is in class.

↓

wrong usage w.r.t We.

a=b+c; → relate to type of variable & then classify them as valid / not

Semantic analyzer → how the expr are formed.

Checks the semantics based on rule. w.r.t langspec.

⁂

Dis ambiguates the operation

a+b
int + int
S1 + S2
float + float

} finds the context in which '+' is used.

Intermediate code generator

3 addr code op.

a = b + c ↑

→ perform the optimisation.

(eliminate common sub expr - use previously calc)

M/C indep optimised code

code generator

→ knows the registers available
→ resources reqd
→ seq of op / m/c cycles

An inst can be replaced by a seq of inst.

WORLD STAR

Role of lexical Analyzer:

Functions of LA: Error repeating
RE → implem using FA.

reads sequence of characters (char by char)

i	f	(	x	1	*	x	2	)	<	1	.	0

create/generate token streams

if	(	x1	*	x2	)	<	1.0

Terms:

Token → returned by LA
Lexeme - seq of char that matches pattern specified by lang
pattern - Rule for constructing tokens

forms

1. scanning → removes unwanted characters from i/p seq.
2. lexical analysis

Lexical Analysis V/s Parsing

simplicity of design
compiler efficiency is improved
compiler portability

(lookahead operation)*

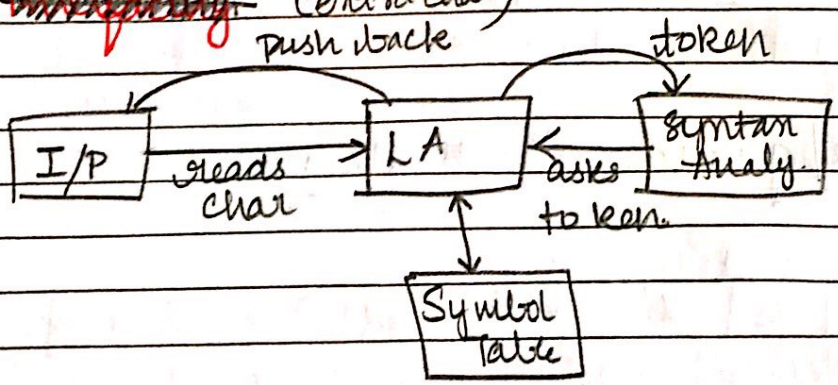
lexeme: identifier rule
letter (letter | digit)*
this rule is matched

a = b + c → lexeme

getsaved id is called token

id assign id + id → token

~~interfacing~~ (extra char) push back



3 ways to impl:

→ lex tool

low level lang → more efficient

high level easier

use

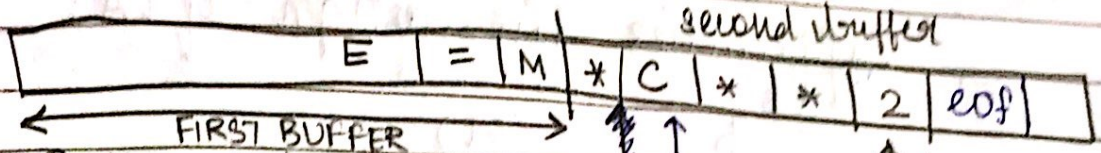
↓

structure

Problems

- ✓ reads char by char → can't tell what word it is.
- ✓ req lookahead op
- ✓ buffer pairs (4096 bytes)

INPUT BUFFERS



Buffer pairs
4096 bytes

lexemeBegin (begin of token)

forward (end of char in a lexeme)

End of first buff → reloads second buffer

end of buffer | char.

→ sentinals — 2 tests

combines end of lexeme | buffer

E | eof | = | eof | m

includes ~~terminator~~ for every lexeme eof

↓

sentinal char.

Lookup & insert

initialise Symbol table with keywords

↓

help in finding identifier

14/12/17

WORLD STAR™ Date Page

Difficulties in design of LA:

Eg: a = b + c (or) a = b + c

Handling Blanks.

count = count + 1

co unt = co unt + 1

} depends on specification of language

DO10 I = 1.25

DO10 I = 1,25

PL/I Programming — no reserved words

if then then = else else else = then

C++ template

F<B1>

Nested template F<B1<B2>>

Cin >> val

defn

How to break a text into tokens:

if (x == 0) a = x <<1

iff (x == 0) a = x < 1

→ retrensiff was an identifier

→ multiple interpretation

stored + passed to syntax analyser

Maximal munch principle: complexity increased

To obtain an appropriate single interpretation

Go with the longest match + pass that

SPECIFICATION OF TOKENS

Name of RE (distinct) (lang over Σ) → character set.

$d_1 - r_1$

$d_2 - r_2 \rightarrow \Sigma \cup d_1$

$d_3 - r_3 \rightarrow \Sigma \cup d_1 \cup d_2$

left of r1 also used

eg: Specification for fax number

91 - (044) - 2235 - 8800

digit - [0-9]

country - digit +

city - '(' digit + ')'

exchange - digit +

number - digit +

Phone number: country - (city) - exchange - number

email ID abc@gmail.com

alphabets - [a-z]

letter - [a-z A-Z 0-9]

name - letter+

atsym - @

servername - alphabets+

dot - .

tld - alphabets+

email ID: name atsym servername dot tld.

name @ servername . tld.

→ limitation of no. of specifications is handled in impl. ⇒ lex tool.

eg: Limit the country code to 3 digits

→ also specify boundaries.

15/12/17

① Specification for identification:

letter [a-z A-Z]

digit [0-9]

(letter) (letter | digit)*

② Unsigned number

digits = digit+

fraction : "." digits | ε

exponent : (E('+'|'-'|ε) digits) | ε

Number :

digits fraction exponent

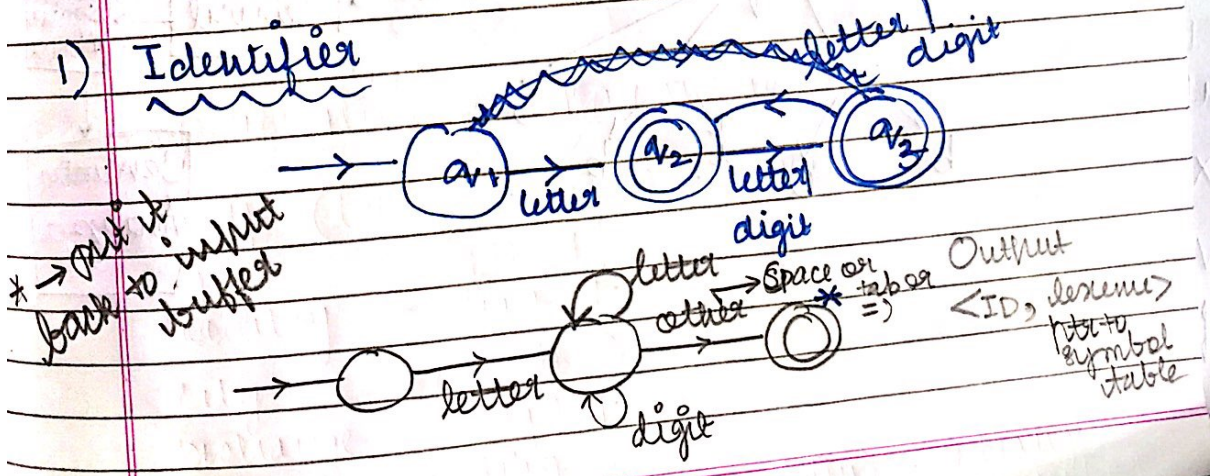
TOKENIZATION:

1. Write RE for lexemes of token
2. Construct R matching all lexemes of all token $R = R1 + R2 + R3 + \cdots$ (prioritizing / ordering)
3. Let input be $x_1, x_2, \ldots, x_n$ for $1 \le i \le n$ check $x_1 \cdots x_i \in L(R)$
4. $x_1 \cdots x_i \in L(R) \Rightarrow x_1 \cdots x_i \in L R_j$ for some j
5. Remove $x_1 \cdots x_i$ from input goto (3)

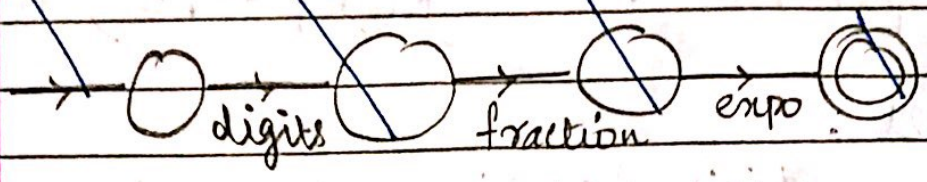
Recognition of tokens:

State Transition Diag ⇒ tokens.

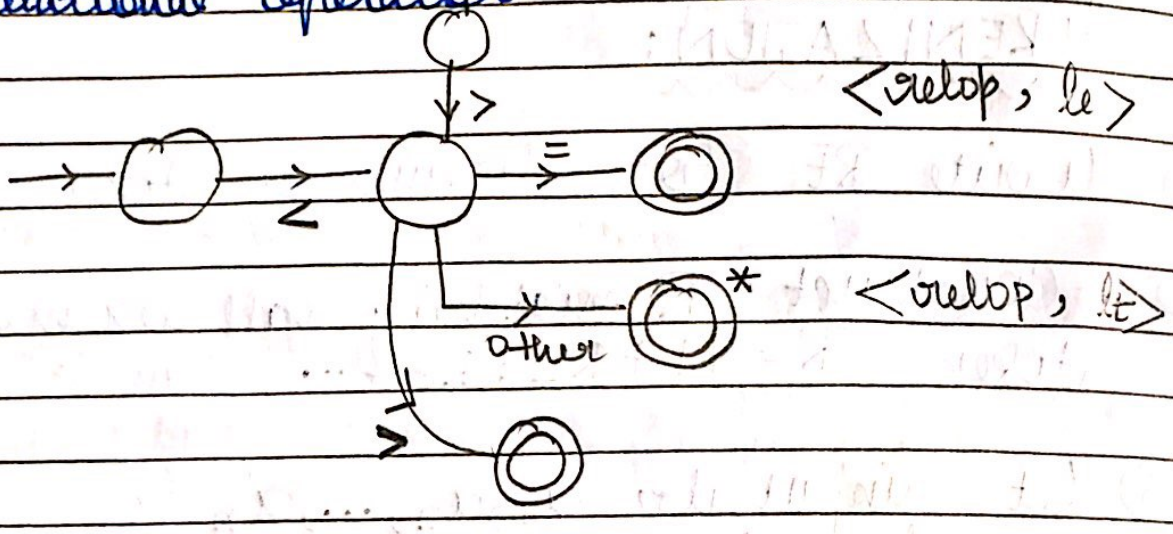
1) Identifier



② Unsigned number:



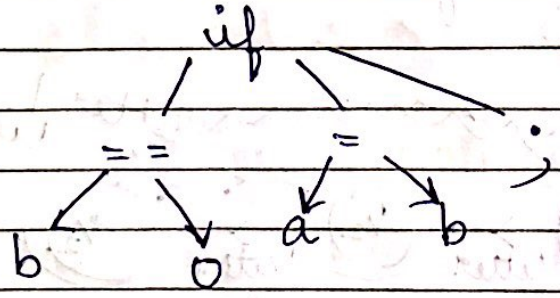
② Relational operator.

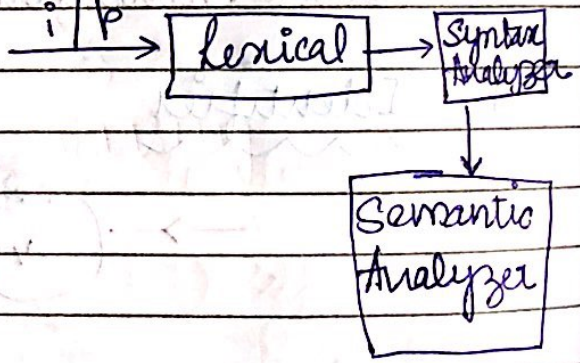


29/12/17

Syntax Analysis

if	(	b	==	0	)	a	=	b	;





error reporting + recovery Model using context free grammar recognized using pushdown automata.

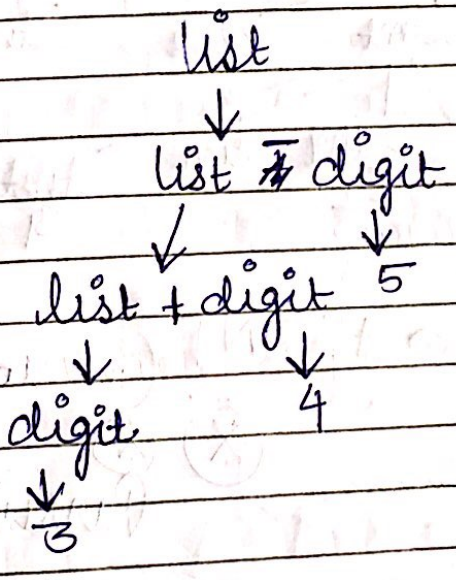
Eg - grammar

1. $S \rightarrow (S)\ S \mid \epsilon$

2. list → list + digit
| list - digit
| digit

digit → 0 | 1 | 2 ... | 9

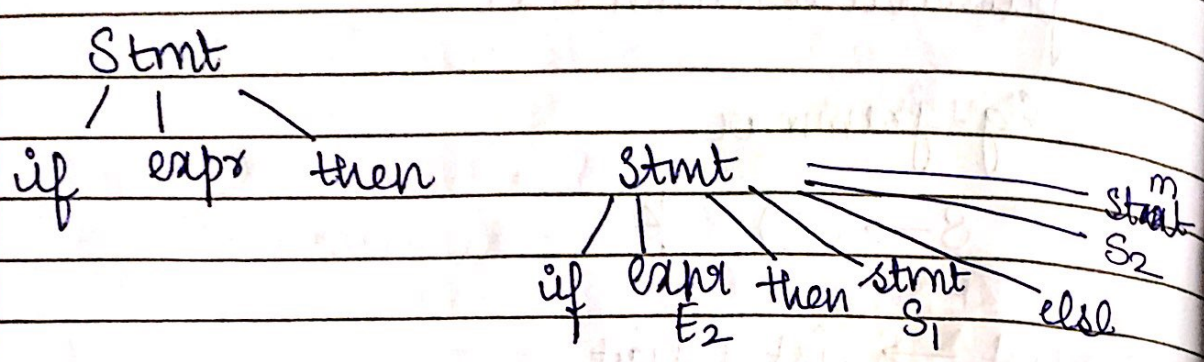
string = 3 + 4 - 5



dangling else grammar

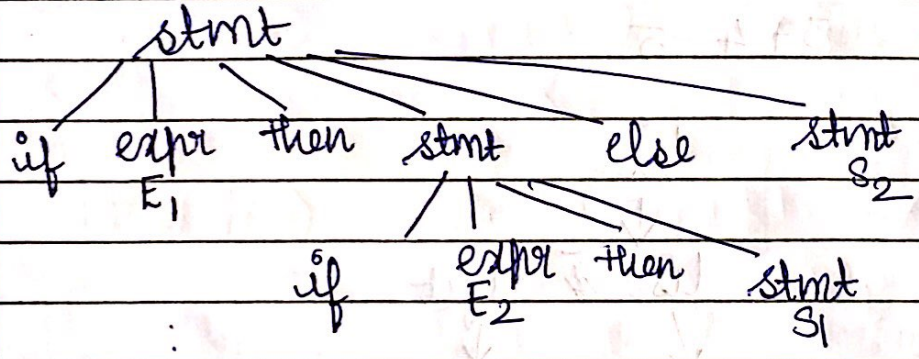
if E_1 then if E_2 then S_1 else.

(1)



stmt → if expr then stmt
| if expr then stmt else stmt
| other

(2)



26.12.17 → ABSENT (get notes)

Parsing

28.12.17

Follow set

E E' T T' F

1. $

2. $, +)

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

apply II + III rule contd.

(2) αBβ

E ⇒ TE' ⇒
Follow(T) = FIRST(E') = {+}

(3) αBβ ; β ⇒ ε
αB

Follow(T) = Follow(E) = {$} ⇒ {$, +,)}

Follow(E') contains Follow(E) = {$,)}

I rule

αBβ
β → ε

Compute the follow sets → A M

II prod

$E' \Rightarrow +TE'$

② $A \longrightarrow \alpha B\beta$

Follow(T) = FIRST(E') = {+}

③ αB $\not\beta$

Follow(T) = {+, $}

III:

$T \rightarrow FT'$

②$A \rightarrow \alpha B|\beta$

Follow(F) = FIRST(T') = {*}

③ αB

Follow(F) = Follow(T) = {+, $} ~~*~~

$\Rightarrow$ {*, +, $}

T':

② $A \rightarrow \alpha B|\beta$

Follow(T') = Follow(T) = {~~*~~, +, $}

IV:

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

follow(F) = FIRST(T') = {*}

follow(F) = follow(T) = {+, $}

V:

$F \rightarrow (E) \mid id$

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

Ans:
Follow(E) = { $,), + }
Follow(E') = { $, +,) }
Follow(F) = { *, +, $ }
Follow(T) = { +, $ }
Follow(T') = { *, +,), $ }

refer notes

Parse Table

	id	+	*	(	)	$
E	E→TE'	Sync		E→TE'	Sync ∩ Follow set	sync
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id	sync	sync	F→(E)		sync

$A \rightarrow \alpha$: $M[A, FIRST(\alpha)]$

New Year 2018

02/01/18

Input validation - id + id * id $

Stack	Input	Action	Matched
E$	id+id*id$		
T E'$ (replace with production)	id+id*id$	E→TE' (parse table)	
F T' E'	id+id*id$	T→FT'	
id T' E'	+id*id$	F→id	id
E' (matched)	+id*id$	T'→ε	
+ T E' (matched)	+id*id$	E'→+TE'	id+
F T' E'	id*id$	T→FT'	
id T' E'	id*id$	F→id	id+id
* F T' E'	*id$	T'→*FT	id+id*
id T' E'	id$	F→id	id+id*
E'$	$	T'→ε	id+id* id
$	$	E'→ε	

given string can be derived ∴ valid

Error recovery:

Panic mode error recovery

(→ detects input till it finds valid i/p & discards it.)

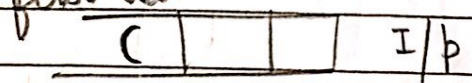
→ discard symbols until it finds terminals in first set of symbols.

(i.e. discard upto what I have invalid input)

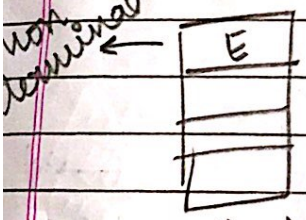
Then it continues parsing

follow set → pop from stack

first set



discards in case of error



Symbols in follow set
→ synchronisation char.
("use "sync" for these entries in parse table)

Pg 230

follow: strings set can end with valid symb.

first set: valid start symb.

discard i/p until find a first set of corr. non terminal (D.L)

discard till you find follow set
↓
then pop the matching elts & continue parsing

id + id

I/P

id

E

(2) id * id

↓

sync in parse table.

so discards in. + continues parsing

LL1 grammar → predective parsing method.

(1) $S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \not\varepsilon$.

$E \rightarrow b$

} PROBLEM.

FIRST SET AND FOLLOW SET:

First(S) = {i, a}

First (S') = {e}

First (E) = {b}

FOLLOW(S) = {$}.

(1) $S \rightarrow iEtSS' \mid a$.

Follow(S') = FOLLOW(S) = {$}

FOLLOW(S) = FIRST(S') = {e}

(2) $S' \rightarrow eS \mid \not\varepsilon$

FOLLOW(S) = FOLLOW(S') = {$}

FOLLOW (S') = {e, $}

FOLLOW(S) = {e, $}

FOLLOW

	i	t	a	b	e	$
S	S→iEtSS'		S→a			
S'					S'→eS.	S'→ε
E				E→b		

04.01.2018

BUTTOM UP PARSING: start from leaf and reach start symbol.

(CLR parsers) L→R parsing, right sentential parser

Reduce the string 'w' of i/p symbol to start symbol

S ⇐ w

Grammar	i/p string
S → aABe	abbcde
A → Abc | b	finds a substring that matches right side of Prod. Replaces that with left side symbol.
B → d	

Shift reduce parsing :

→ Split the string into two parts
→ Separated by special character '.'
→ left part is string of terminals & Non terminals
→ Right part is string of terminals

$\alpha\beta.\beta r$

Shift → $\alpha\beta\beta.r$

Reduce → $\alpha A\beta.r$ if $A \to \beta$ ← reads substring before the "."

• abbcde
a • bbcde
aA • bcde
aAb • cde
aAbc • de
aA • de ⇒ aAd • e ⇒ aABe •
S •

• abbcde

Shift or reduce conflict.

aAbc • de

replace 'b'

or → (take a substring)

aAAcde
aAAcBe
(cant reach start symbol)

aAde
aABe ⇒ aABe •
S •

perform rightmost derivation in reverse order

called as handle
(handle pruning)

GRAMMAR

E → E+E | E*E | id

i/p → id * id + id

• id * id + id
replace (id • * id + id → shift
E • * id + id → reduce
E* • id + id → shift
E * id • + id → shift
E * E • + id → reduce
E • + id → reduce
E + • id → shift
E + id • → shift
E + E • → reduce
E • → reduce
↳ Accepted

correct precedence coz it puts '*' then add

Shift reduce conflicts.

shift → E*E+ • id
E*E+id • (S)
E*E+E • (R)
E*E (R)
E (R)
✗
wrong precedence
Computationally wrong

Scanned by CamScanner

Reduce-reduce conflict:

a Abcde

↓

reduce b or Abc.

DEFN (X)

Handle – Handle is a string that matches right hand side of a production whose reduction represents one step along the reverse of right most derivation

Eg : $S \Rightarrow \alpha A w \Rightarrow \alpha \beta w$

$S \xrightarrow[rm]{*} \alpha A w \xrightarrow[rm]{} \alpha \beta w$

$A \to \beta$ is a handle.

β - handle.

Handle pruning: If β is the handle of $A \to \beta$ is a production of replace β by A is called h.p.

Right most derivation in reverse can be obtained by handle pruning.

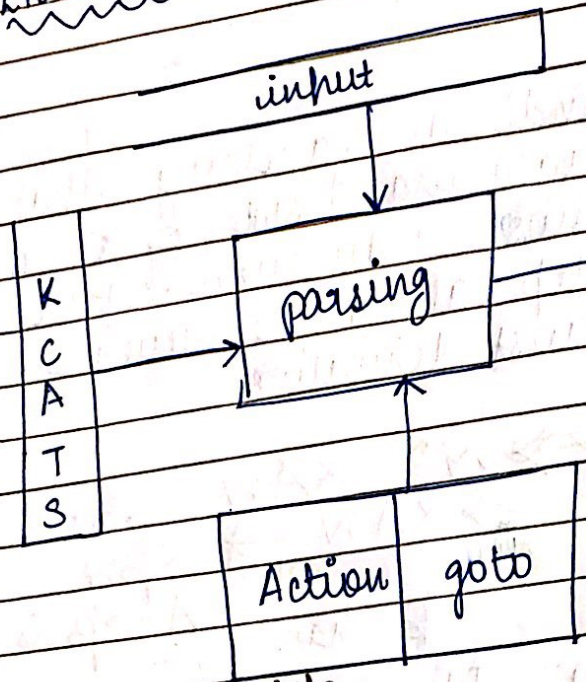
→ handles left recursion
→ better error recovery

Input : id + id * .

ACV

05/01/2018

LR Parser (SLR) - LR(0) Items



stack — state info $S_0 X_1, S_1 X_1 \dots X_n S_n$

Action table — Terminals

Goto — Non terminals

States

I0 → state name

S. Shift
Reduce
accept
error
} possible states.

Parser state symbol represents what is on the stack + the goal is to know valid reductions at any given time. Summarizes all possible state info for the grammar.
(Stack prefixes as a parser state)
↓
defined by DFA.
Accept states of DFA are unique reductions.

AUGMENT THE GRAMMAR:

$E' \rightarrow E$ → add this production ?
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

simply adding that production ∴ more than 1 prod for start symbol.

Construct LR(0) Items:

I_0 ↓ state name

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

Find all possible valid reduction.

(I_0, E) → apply i/p then what is my state

$(I_0, E) = I_1$

$E' \rightarrow E.$
$E \rightarrow E.+T$ } perform shift for non term 'E' on right side of prod.

$(I_0, T) = I_2$

$E \rightarrow T.$
$T \rightarrow T.*F$

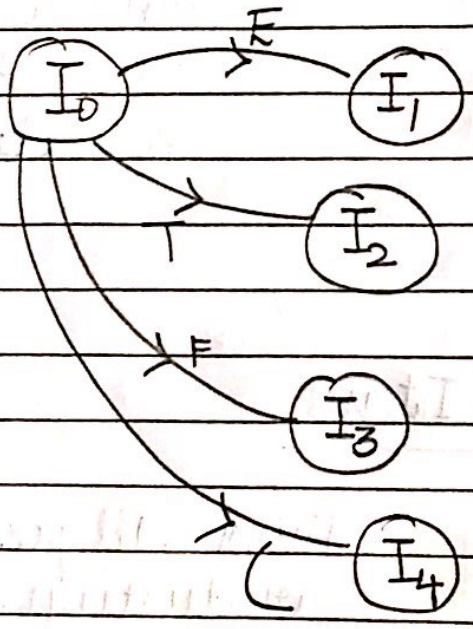
$(I_0, F) = I_3$

$T \rightarrow F.$

Shift for only non terminal

I_0 $(I_0, ( ) = I_4$

$F \rightarrow (.E)$
$E \rightarrow .E+T$ → non terminal so add E prod also
$E \rightarrow .T$ Call derivation related to E)
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$



$(I_0, id) = I_5$
$F \rightarrow id.$

11 states

$(I_1, +) = I_6$

$E \longrightarrow E + .T$
$T \longrightarrow .T * F$
$T \longrightarrow .F$
$F \longrightarrow .(E)$
$F \longrightarrow .id$

• Nonterminal add prood of N.T.

$(I_2, *) = I_7$

$T \longrightarrow T * .F$
$F \longrightarrow .(E)$
$F \longrightarrow .id$

$(I_4, E) = I_8$

$F \longrightarrow (E.)$
$E \longrightarrow E. + T$

$(I_4, T) = I_9$

$E \longrightarrow T.$
$T \longrightarrow T. * F$

$(I_4, F) = I_{10}$

$T \longrightarrow F.$

$(I_4, () = I_{11}$

$F \longrightarrow (.E)$
$E \longrightarrow .E + T$
$E \longrightarrow .T$
$T \longrightarrow .T * F$
$T \longrightarrow .F$
$F \longrightarrow .(E)$
$F \longrightarrow .id$

I_0 :

(2) $S' \to .S$
$S \to .iEtS$
$S \to .iEtSeS$
$S \to .a$
~~$E \to .b$~~

$(I_0, S) : I_1$

$S' \to S.$

$(I_0, i) : I_2$

$S \to i.EtS$
$S \to i.EtSeS$
$E \to .b$

$(I_0, a) : I_3$

$S \to a.$

~~$(I_0, b) : I_4$~~ $(I_2, b) : I_4$

~~$E \to b.$~~ $E \to b.$

$(I_2, E) : I_5$

$S \to iE.tS$
$S \to iE.tSeS$

~~$(I_2, b) : I_6$~~

~~$E \to b.$~~

$(I_5, t) : I_6$

$S \to iEt.S$
$S \to iEt.SeS$
$S \to .iEtS$
$S \to .iEtSeS$
$S \to .a$

$(I_6, S) : I_7$

$S \to iEtS.$
$S \to iEtS.eS$
$S \to .iEtS$
$S \to .iEtSeS$
$S \to .a$

$(I_7, e) : I_8$

$S \to iEtSe.S$
$S \to .iEtS$
$S \to .iEtSeS$
$S \to .a$

$(I_8, S) : I_9$

$S \to iEtSeS.$
$S \to .iEtS$
$S \to .iEtSeS$
$S \to .a$

11.01.18

stack : $S_0 X_1, S_1, X_2, \ldots X_m S_m$

Input $a_i, a_{i+1}, \ldots a_k \$$

if action is $[S_m, a_i]$ = shift S then stack configuration becomes

$S_0 X_1, S_1 X_2, \ldots, X_m S_m a_i S$, Input $a_{i+1}, \ldots a_n \$$

if action $[S_m, a_i]$ = reduce $A \to \beta$ then configuration becomes stack $S_0 X_1 S_1 X_2 \ldots X_{m-r} S_{m-r} A S$

Input : $a_i, a_{i+1} \ldots a_n \$$

FIRST(S) = { (, a }

FIRST(L) = { (, a }

① $S \to (L) \mid a$

$L \to L,S \mid S$

Follow(S): { $,) }

Follow(L): {) }

$\to \alpha B \beta$

$S' \to S$

① $S \to (L)$

② $S \to a$

③ $L \to L,S$

④ $L \to S$

$S \to (\quad L$

$L \to L \quad S$

$A \to B \quad \beta$

I_0: $S' \to \cdot S$
$S \to \cdot(L)$
$S \to \cdot a$

$(I_0, S) : I_1$
$S' \to S \cdot$

$(I_0, () : I_2$
$S \to (\cdot L)$
$L \to \cdot L,S$
$L \to \cdot S$
$S \to \cdot(L)$
$S \to \cdot a$

$(I_0, a) = I_3$
$S \to a \cdot$

$(I_2, S) = I_5$
$L \to S \cdot$

$(I_2, L) = I_4$
$S \to (L \cdot)$
$L \to L \cdot, S$

$(I_4,)) = I_6$
$S \to (L) \cdot$

$(I_4, ,) = I_7$
$L \to L, \cdot S$
$S \to \cdot(L)$
$S \to \cdot a$

$(I_7, S) = I_8$
$L \to L, S \cdot$

LR parsing algorithm

Initial stack state S_0 Input : w$

If action [Si, a] shift S'

then push(a), push(S'), ip++

else if action [Si, a] = reduce A→β

then pop 2*|β| symbols

push(A), pop goto[S'', A]

else if action [Si, a] = accept

then exit

else

error.

States	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		r2	S7		r2	r2 → follow(T)			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Reduce when • in right end.

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

Rules:

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

① E → E + T
② E → T
③ T → T * F
④ T → F
⑤ F → (E)
⑥ F → id

0T2 +id*id

0E

Stack	Input	Action
0	id*id+id.	Shift + go to
0id5	*id+id	Reduce E→id
0F3	*id+id	Reduce T→F
0T2	*id+id	Shift + go to
0T2*7	id+id	
0T2*7id5	+id	reduce by 6
0T2*7F10	+id	reduce by 3
0T2*7F	+id	
0T2	+id	reduce by 2
0E1	+id	
0E1+6	id$	shift + go to 6

0E1+6id5	$	reduce by (6)
0E1+6F3	$	reduce by (4)
0E1+6T9	$	reduce by (1)
0E1	$	accept.

23.01.18

6

Unambiguous ↳SLR

Construct SLR parse table

$S \rightarrow L=R \mid R$
$L \rightarrow *R \mid id$
$R \rightarrow L$

① $S \rightarrow L=R$
② $S \rightarrow R$
③ $L \rightarrow *R$
④ $L \rightarrow id$
⑤ $R \rightarrow L$

Augmented form

$S' \rightarrow .S$ $
$S \rightarrow .L=R$ $
$S \rightarrow .R$ $
$L \rightarrow .*R$ =
$L \rightarrow .id$ =
$R \rightarrow .L$ $

goto $(I_0, S) = I_1$

$S' \rightarrow S.$ $

goto $(I_0, L) = I_2$

$S \rightarrow L.=R$ $
$R \rightarrow L.$ $

goto $(I_0, R) = I_3$

$S \rightarrow R.$ $

goto $(I_0, *) = I_4$

$L \rightarrow *.R$ =
$R \rightarrow .L$ =
$L \rightarrow .id$ =
$L \rightarrow .*R$ =

goto $(I_0, id) = I_5$

$L \rightarrow id.$ =

goto $(I_2, =) = I_6$

$S \rightarrow L=.R$ $
$R \rightarrow .L$ $
$L \rightarrow .id$ $
$L \rightarrow .*R$ $

goto $(I_4, R) : I_7$

$L \rightarrow *R.$ =

goto $(I_4, L) : I_8$

$R \rightarrow L.$ =

goto $(I_6, R) : I_9$

$S \rightarrow L=R.$ $

goto $(I_6, L) : I_{10}$ $
goto $(I_4, id) : I_5$ =
goto $(I_4, *) : I_4$ =
goto $(I_6, id) : I_{12}$ $
goto $(I_6, *) = I_{11}$ $

given new state from new location

	Action				Goto		
	*	=	id	$	S	L	R
0	S4		S5		1	2	3
1				accept			
2		S6/r5		r5			
3		r2		r2			
4	S4		S5			8	7
5		r4		r4			
6	S4		S5			8	9
7		r3		r3			
8		r5		r5			
9		r1		r1			
10							
11							
12							
13							

	FIRST	FOLLOW
S'		{$}
S		{$}
L	{*, id}	{=, $}
R	*, id	{*, id, =, $}

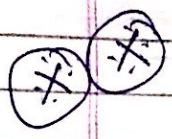
* - Shift reduce conflict. - unambiguity Parsing is not powerful enough to avoid conflict.

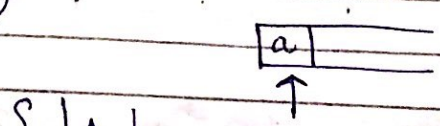
I_{13}: Go to (I_{11}, R)
$L \rightarrow *R \cdot$ $

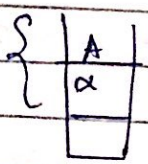
Goto $(I_{11}, L) = I_{10}$
$(I_{11}, *) = I_{11}$
$(I_{11}, id) = I_{12}$

reduce if lookahead is $ or =

CLR → Canonical LR method







'a' follow of subset of αA then only valid. (multiple stack info)

CLR considers f(subset of αA.)
SLR " only f(A.)

LR(1)
↳ 1 lookahead

Closure set computation for LR(1):

$[A \rightarrow \alpha \cdot B\beta \ , \ a]$

$[B \rightarrow \cdot\gamma \ , \ first(\beta a)]$

↓
Follow of B
∴ reduces to B

I_1 : Goto (I_0, S)

$S' \rightarrow S. \ \$$

I_2 Goto (I_0, L) $\$$ deriving from S then lookahead $

$S \rightarrow L. = R\$$
$R \rightarrow L. \ \$$ follow(R)

I_3 : Goto(I_0, R)

$S \rightarrow R.\$$

1 problem
1 theory

I_4 : Goto(I_0, *)

$L \rightarrow *.R$

25.01.18

CLR: table → no. of states ↑

States	=	*	id	$	S	L	R
0		S4	S5		1	2	3
1				acc			
2	S6			r5			
3				r3			
4		S4	S5			8	7
5	r4			r4			
6		S4	S5				
7	r3			r3		8	9
8	r5			r5			
9				r1			
10	r5			r5			
11		S11	S12				
12	r4			r4			
13				r3			

LALR → reduce - reduce conflict