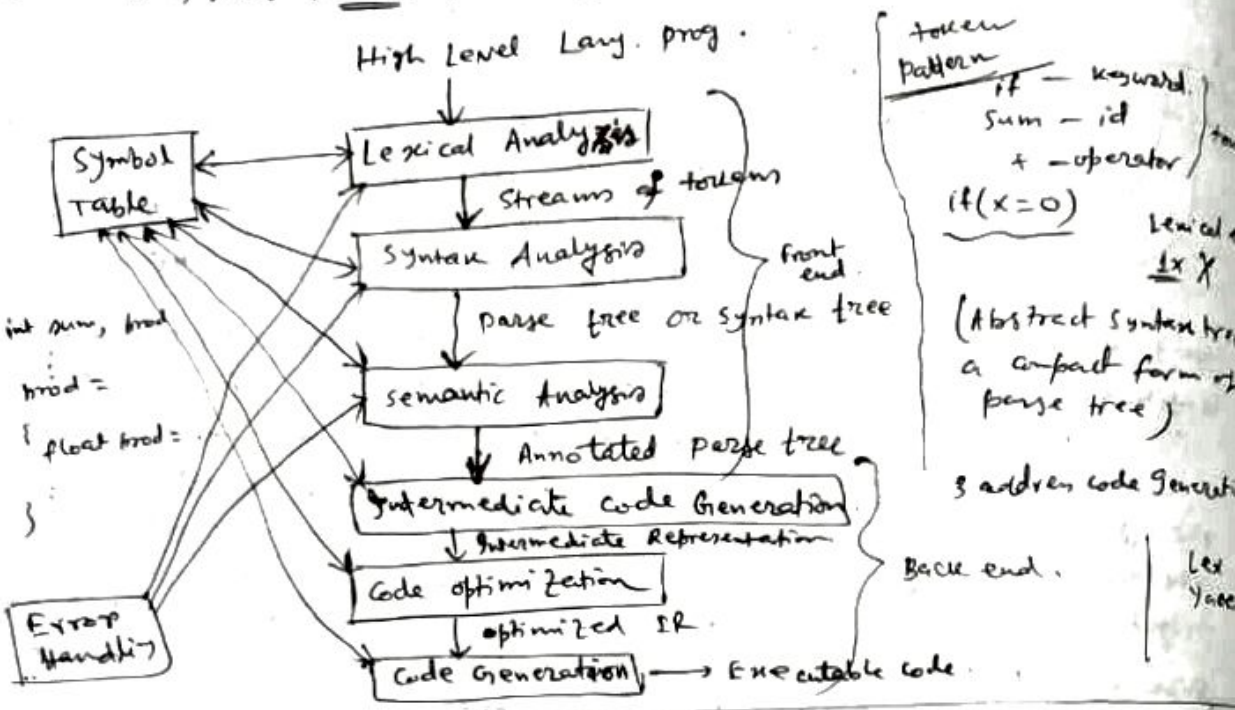
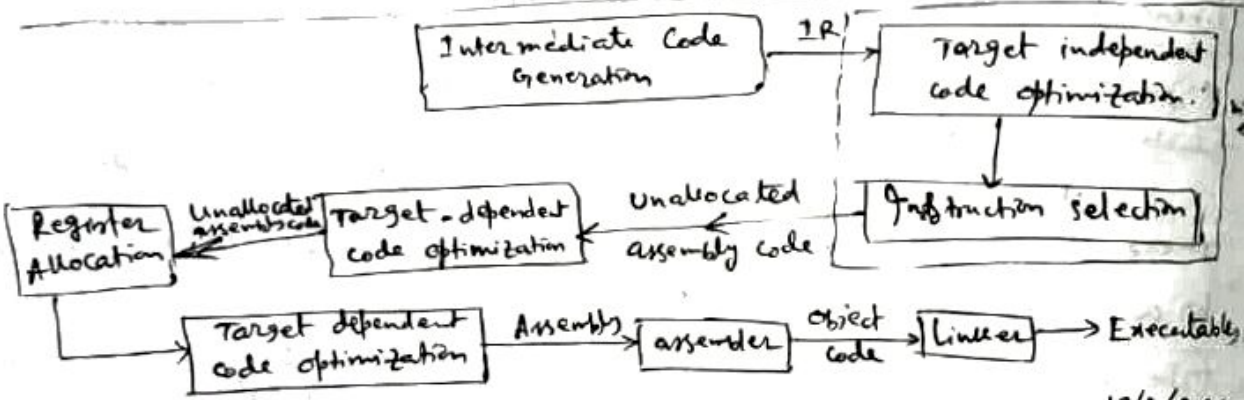
Compiler Design

Book → Aho, Sethi, Lam, Ullman (Principles of compiler design..)





13/1/2020

A Grammar can be regarded as a device that enumerates the sentences of a lang. And a lang. is a collection of sentences of finite length constructed from a finite alphabet of symbols.

Alphabets, Sentences } Grammar.
Generative, Analytic }

A generative grammar formalizes an algo. that generates valid strings in a lang. On the other hand, an analytic grammar is a set of rules to reduce an input string to a boolean result that indicates the validity of the string in the given lang.

Chomsky { Surface structure — syntax of the lang.
Deep structure — semantics "

Formally a grammar is defined as a quad-tuple $G = (N, \Sigma, P, S)$, where
N is a finite set of non-terminals.
Σ is a finite set of terminals and is disjoint from
P is a finite set of production rules of the form
$\alpha \in (N \cup$

Chomsky Hierarchy

a, b, c → terminal
A, B, C → non-terminal
α, β, γ, ω → strings

Unrestricted gram. or Type 0 grammars include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine.

Type 1 grammars or context sensitive grammars generate the context sensitive languages. These grammars have rules of the form

$\alpha A \beta \to \alpha \gamma \beta$ with A a nonterminal and α, β, γ strings of terminals & non-terminals.

The strings $\alpha \& \beta$ may be empty but γ must be non-empty. The rule $S \to \epsilon$ is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton. (a TM whose tape is finite by multiple (non-deterministic of i/p size)

Type 2 grammars or ... generates the context free languages which are defined by rules of the form

$A \to \gamma$ with A a non-terminal & γ a string of terminals & nonterminals

These languages are exactly all languages that can be recognized by a Non-deterministic PDA.

Type 3 grammar or Regular gram generates R.L. Such a grammar restricts its rule to a single nonterminal in it's L.H.S and right hand side, a single nonterminal possibly followed by terminal or preceded by terminal but not both by a single nonterminal.

$A \to aB$ or $A \to Ba$

The rule $S \to \epsilon$ is allowed if S does not appear on the right hand side.

These languages are exactly all language that form - finite state automaton

Lexical analysis

p + q + ...

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

$E \Rightarrow T + id \Rightarrow id + id$
(Sentential form)

id + id

Lexical Analysis

Token: Token is a sequence of characters to be treated as a single unit.

Pattern: A rule which describes which strings are assigned to a token

Lexeme → the exact sign of i/p chars matched by a pattern

Patterns: a^*, ba^*c — Lexeme → sum, +, < — token → id, arithoperator

<id, pointer to symbol table entry for sum> Sum = a+b

Sum = a+b <assign-op> <id, pointer to a> <arith op> <id, pointer to b>

id = id + id.

14/1/2020

Regular Expressions

① ϵ is a regular expression and $L\{\epsilon\}$ is ϵ

② if 'a' is a symbol in Σ, then 'a' is also a RE and $L(a) = \{$

if r and s are two REs

③ $(r)|(s)$ is a RE denoting the language $L(r) \cup L(s)$

④ $(r)(s)$ is a RE denoting $L(r)\,L(s)$

⑤ $(r)^*$ is a RE denoting $(L(r))^*$

⑥ (r) is a RE denoting $L(r)$

precedence: * > concatination > |

Axioms

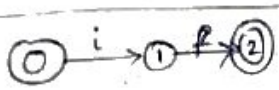
$r|s = s|r$ $\quad r|(s|t) = (r|s)|t$

$r(st) = (rs)t$ $\quad r(s|t) = rs|rt$

$\epsilon r = r\epsilon = r$ $\quad r^* = (r|\epsilon)^*$ $\quad r^{**} = r^*$ → idempotent

$r^+ = rr^* = r^*r$ (positive closure)

$r? = r|\epsilon$ $\quad [abc] = a|b|c$ $\quad [a\text{-}z] = a|b|\ldots|z$



NFA
DFA

NFA: ① A finite set of states S

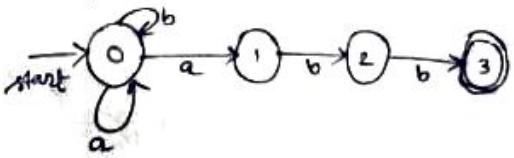
② A set of i/p symbols Σ (ϵ is not included)

③ A transition function that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$, a set of next states.

④ A ~~starting~~ state $s_0 \in S$ → the start state.

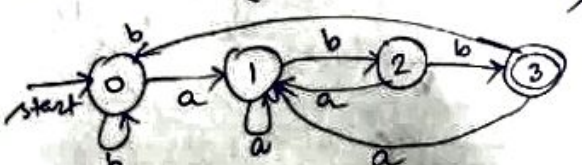
⑤ $F \subseteq S$ → a set of final and accepting states.

$(a|b)^*abb$



3a There is no move on i/p ϵ (for DFA)

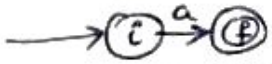
for each state s and input symbol a, there is exactly one edge out of s labeled a. (for DFA point 3)



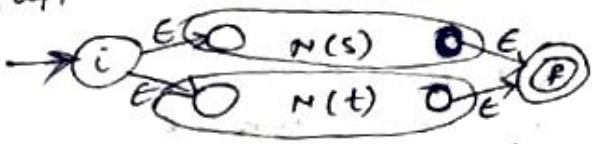
Mc Naughton - Yamada - Thompson algorithm

(1) For expression ϵ construct the NFA

For any subexpression a in Σ, construct the NFA

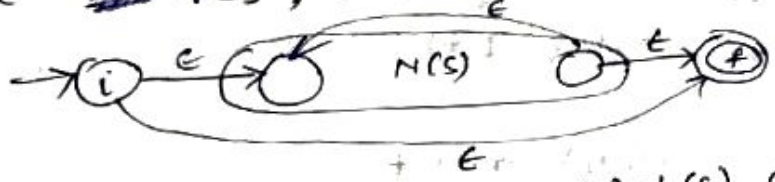


(2) Suppose N(s) and N(t) are NFA's for RE's s & t
Suppose $r = s|t$. Then N(r) is constructed as follows:



(3) Suppose $r = st$, then N(r) is constructed as follows:

(4) Suppose $r = s^*$, then N(r) is constructed as follows:



(5) Finally, suppose $r = (s)$, then $L(r) = L(s)$ and N(r) uses N(s)

Properties of the Constructed NFA N(r)

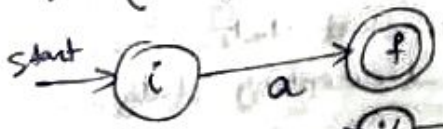
(1) N(r) has at most twice as many states as there are operators and operands in r.
(2) N(r) has one start state and one accepting state.
(3) Each state of N(r) other than the accepting state has either one outgoing transition on a symbol in the alphabet or (Σ)
two outgoing transitions, both on ϵ.

21/1/2020.

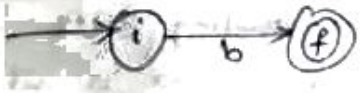
NFA

$r = (a|b)^* abb$

1



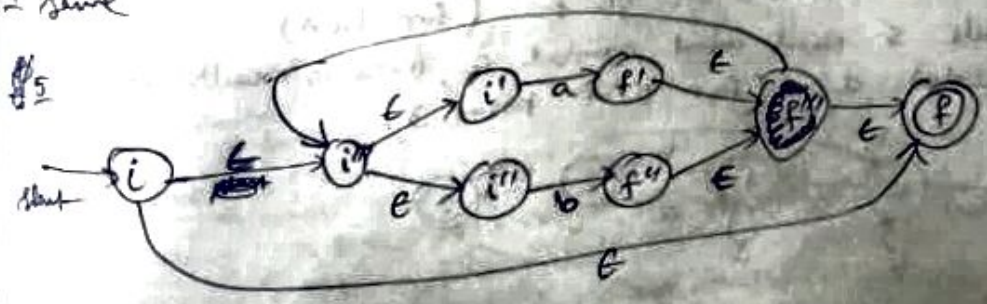
2



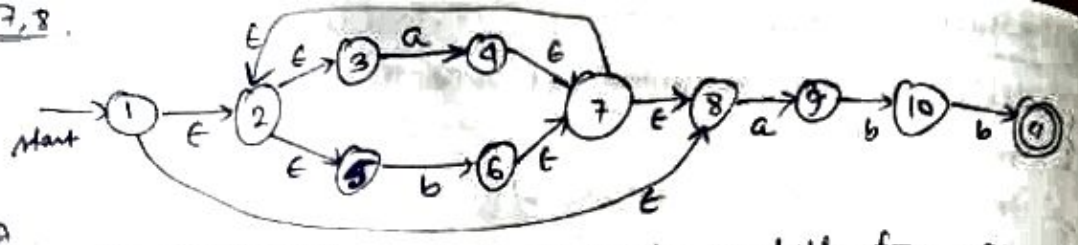
3

4 same

5



1,7,8.



DFA

ε-closure (s) → set of NFA states reachable from NFA state on ε-transition alone.

ε-closure (T) – set of NFA states reachable from some NFA state in set T on ε-transition alone.

$= \bigcup_{s \text{ in } T}$ ε-closure (s)

move (T, a) → set of NFA states to which there is a transition on input symbol 'a' from some state 's' in 'T'.

ε-closure (1) = {1, 2, 3, 5, 8} = A.

ε-closure (move (A, a)) = ε-closure (4, 9) = {2, 3, 4, 5, 7, 8, 9} = B

ε-closure (move (A, b)) = ε-closure (6) = {2, 3, 5, 6, 7, 8}. = C

NFA state	DFA state	a	b
{1, 2, 3, 5, 8}	A	B	C
{2, 3, 4, 5, 7, 8, 9}	B	B	D
{2, 3, 5, 6, 7, 8}	C	B	C
{2, 3, 5, 6, 7, 8, 10}	D	B	E
{2, 3, 5, 6, 7, 8, 11}	E	B	C

ε-closure (move (B, a)) = ε-closure (4, 9) = B

ε-closure (move (B, b)) = ε-closure (6, 10) = {2, 3, 5, 6, 7, 8, 10} = D

ε-closure (move (C, a)) = ε-closure (4, 9) = B

ε-closure (move (C, b)) = ε-closure (6) = C

ε-closure (move (D, a)) = B

ε-closure (move (D, b)) = ε-closure (6, 11) = {2, 3, 5, 6, 7, 8, 11} = E

ε-closure (move (E, a)) = ε-closure (4, 9) = B.

Computing ε-closure

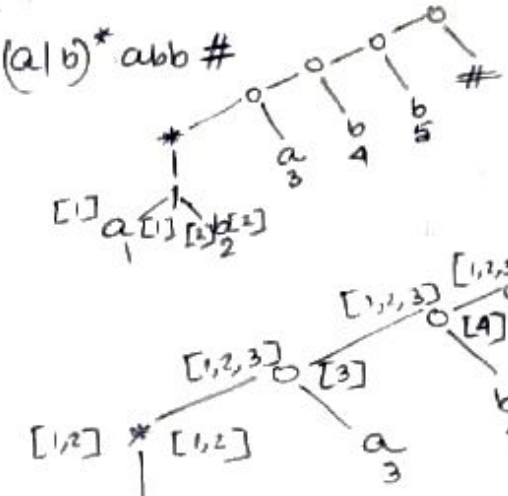
```
push all states of T into stack;
initialize ε-closure (T) to T;
while (stack is not empty) {
        pop t, the top element of the stack
        for (each state u with an edge from t to u labeled ε)
            if (u is not in ε-closure (T))
            {   add u to ε-closure (T)
                push u onto stack;
            }
}
```

Subset construction

```
while (there is an unmarked state T in D states)
{ mark T;
  for (each input symbol a)
  { μ = ε-closure (move (T,a))
    if (μ is not in D states)
        add μ as an unmarked state to D states
    D tran [T,a] = μ
  }
}
```

22/1/2020

$(a|b)^*abb\#$



AST
(Abstract Syntax Tree)

nullable
first pos
last pos
} all nodes

follow pos ⇒ only for operator

nullable (n) is true for a syntax tree node n if and only if the subexpression presented by n has ε in its language.

firstpos (n) is the set of positions in the subtree rooted at n that correspond to the first symbol of at least one string in the language of the subexpression rooted at n.

lastpos (n) is the set of positions in the subtree rooted at n that correspond to the last symbol of at least one string in the language of the subexpression rooted at n.

node n	nullable(n)	firstpos(n)	lastpos(n)
① leaf node labelled ε	true	ϕ	ϕ
② leaf node labelled with position i	false	$\{i\}$	$\{i\}$
③ OR node $n = c_1 \mid c_2$	nullable (c_1) or nullable (c_2)	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
④ cat node $n = c_1 c_2$	nullable (c_1) and nullable (c_2)	if (nullable(c_1)) firstpos(c_1) ∪ firstpos(c_2) else firstpos(c_1)	if (nullable(c_2)) lastpos(c_1) ∪ lastpos(c_2) else lastpos(c_2)

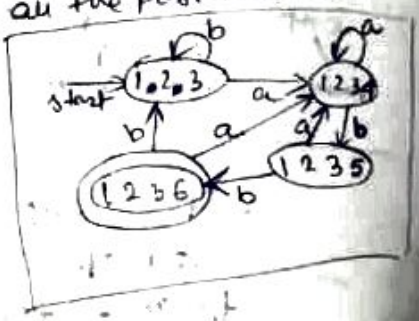
node n	nullable(n)	firstpos(n)	lastpos(n)
(5) star node $n = c_1^*$	true	firstpos(c_1)	lastpos(c_1)

followpos(p) for a position p, is the set of positions q in the entire syntax tree such that there is some string $x = a_1 a_2 \dots a_n$ in $(L(r)\#)$ such that for some i, there is a way to explain the membership of x in $(L(r)\#)$ by matching a_i to position p of the syntax tree and a_{i+1} to position q.

Computing followpos.

1 If n is a cat node with left child c_1 and right child c_2, then for every position i in lastpos(c_1), all the positions in firstpos(c_2) are in followpos(i)

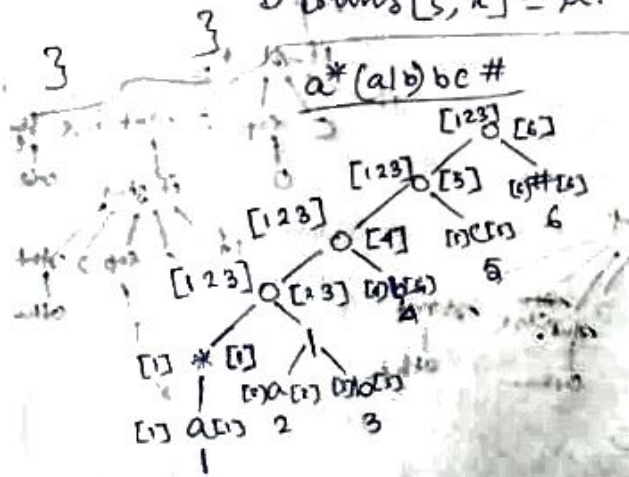
node(n)	followpos
1	3, 1, 2
2	3, 1, 2
3	4
4	5
5	6
6	ϕ



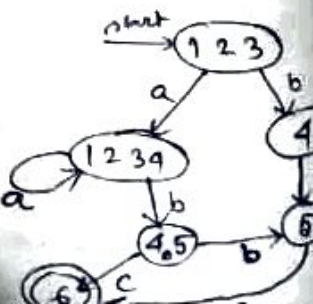
2 If n is a star node and i is a position in lastpos(n), then all positions in firstpos(n) are in followpos(i)

Initialize Dstates to contain only the unmarked state firstpos(n_0) where n_0 is the root of syntax tree T for $(r)\#$

while (there is an unmarked state S in Dstates)
{
  mark S
  for (each input symbol a)
  {
    let U be the union of followpos(p) for all p in S that correspond to a
    if (U is not in Dstates)
      add U as an unmarked state to Dstates
    Dtrans[S, a] = U.
  }
}



node(n)	followpos
1	1 2 3
2	4
3	4
4	5
5	6
6	ϕ



Parsing

5/2/2020

$A \to \gamma$

exp → exp op exp | (exp) | num

op → + | - | *

35 + 2 * 3

(num + num * num)

exp ⇒ exp op exp [exp → exp op exp]

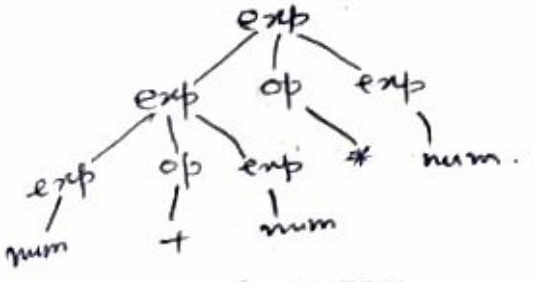
⇒ exp op exp op exp

⇒ num op exp op exp ⇒ num + exp op exp

⇒ num + num op exp

⇒ num + num * exp

⇒ num + num * num.



exp ⇒ exp op exp

⇒ ....

exp → exp addop term | term

addop → + | -

term → term mulop factor | factor

mulop → *

factor → (exp) | number

35+2+3-5-7

exp ⇒ exp addop term

⇒ exp addop term addop term

⇒ exp addop term addop term addop term

⇒ exp addop term addop term addop term addop term

⇒ term addop

⇒ factor addop ... ⇒ num addop ... ⇒ num + ...

3*2+35

exp ⇒ exp addop term ⇒ term addop term ⇒ term addop factor ...

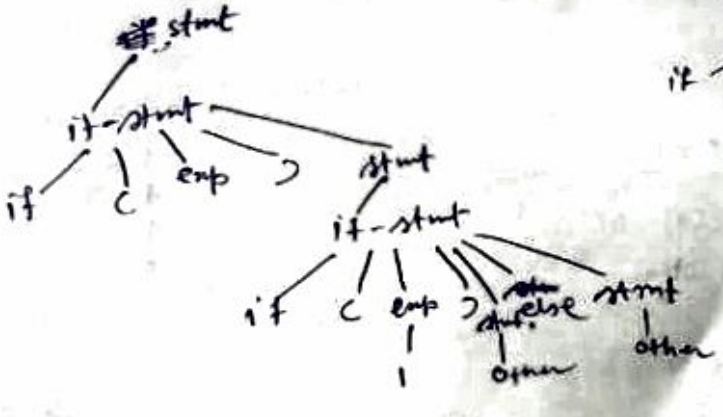
```
if ( )
{

}
if ( )
{

}
else ()
{

}
```

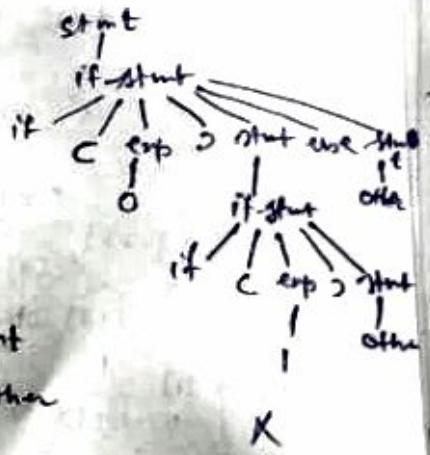
stmt → if-stmt | other

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

exp → 0 | 1

if(0) if(1) other else other





matched / unmatched

Stmt → m stmt | um stmt

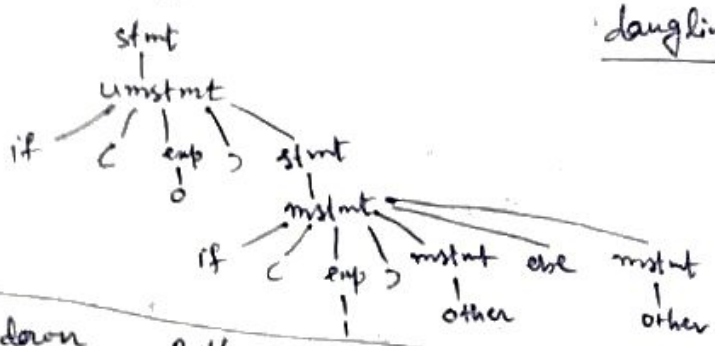
mstmt → if (exp) mstmt else mstmt | other

umstmt → if (exp) stmt | if (exp) mstmt else umstmt

exp → 0 | 1

if (0) if (1) other else other

dangling else



Top-down parsing	Bottom-up parsing
S → cAd	cad
A → ab | a	S ⇒ cAd ⇒ cabd
	cad

backtracking } top down

Recursive-Descent (procedures) } top down
Predictive (non backtracking) }

LL(1) — 1 token look ahead.

Left to right scan — Leftmost derivation

A → Aa | b

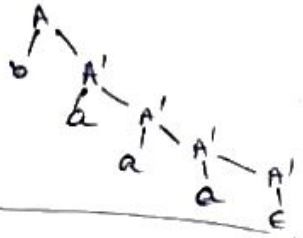
ba
baa
baaa...

left recursion

A → bA'

A' → aA' | ∈

right recursion.



① A → Ba | Aa | c

② B → Bb | Ab | d

A ⇒ Ba ⇒ Aba

for i = 1 to n do
  for j = 1 to i-1 do
    replace each grammar rule choice of the form $A_i \to A_j\beta$ by the rule
    $A_i \to \alpha_1\beta \mid \alpha_2\beta \mid \dots \mid \alpha_k\beta$, where
    $A_j \to \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$ is the current rule for A_j
  Eliminate all immediate left recursion

① for i=1, A → ~~Ba | B~~ BaA' | cA'
A' → aA' | ∈.

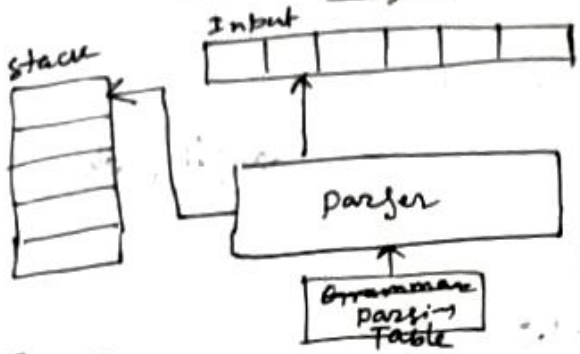
② for i=2, j=1,
A → BaA' | cA'
A' → aA' | ∈
B → Bb | BaA'b | cA'b | d

③ A → BaA' | cA'
A' → aA' | ∈
B → cA'bB' | dB'
B' → bB' | aA'bB' | ∈

$A \to \alpha\beta_1 \mid \alpha\beta_2 \Rightarrow A \to \alpha A'$, $A' \to \beta_1 \mid \beta_2$ } left factoring

11/2/2020

LL(1)



exp → exp + term | term

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid id$

a+b

35*4*3

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

Derivation for $E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow (E)T'E' \Rightarrow \dots$

FIRST

FOLLOW

$E \Rightarrow TE' \overset{*}{\Rightarrow} E'$ (say $T \to \epsilon$)

① $A \to x$

$A \to Y_1 Y_2 Y_3 \dots Y_j \dots Y_n$

Computing FIRST

① if x is a terminal, then FIRST(x) is $\{x\}$.

② if $x \to \epsilon$ is a production, then add ϵ to FIRST(x)

③ if X is a nonterminal and $X \to Y_1 Y_2 \dots Y_k$ is a production then place a in FIRST(X) if for some i, a is in FIRST(Y_i) and ϵ is in all of FIRST(Y_1), ..., FIRST(Y_{i-1})

$A \to \alpha$ FIRST of α to be included.

~~FIRST(+) = {+}~~

FIRST(E) = [illegible] FIRST(T) = FIRST(F) = {(, id}

FIRST(E') = {+, ε}

FIRST(T') = {*, ε}

$E \overset{*}{\Rightarrow} +TE'$ (if E' → ε)

E'
T'

ε	*	...

Computing FOLLOW

① Place $ in FOLLOW(S)

where S is the start symbol & $ is i/p end marker.

$S \overset{*}{\Rightarrow} \dots A \to \alpha B\beta$ (if $\beta \to \epsilon$, $A \to \alpha B$)

$\gamma A \delta \Rightarrow \gamma \alpha B \beta \delta \Rightarrow \gamma \alpha B \delta$

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

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

FOLLOW(E) = {+,)}
FOLLOW(T) = {*, +,)}
FOLLOW(F) = {*, +,)}
Grammar with left recursion

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

Grammar with right recursion

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

A→α
FIRST(α) = {a, ...}
M[A, a]

E→TE' E'→+TE' T→FT' E'→ϵ T'→ϵ

For each production A→α of the grammar do the following steps:
① For each terminal a in FIRST(α), add A→α to M[A,a]
② If ϵ is in FIRST(α), add A→α to M[A, b] for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and $ is in FOLLOW(A) then add A→α to M[A, $].

id + id * id $

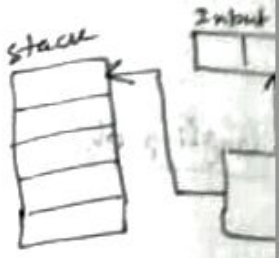
Stack	Input	Action
$ E	id + id * id $	
$ E'T	id + id * id $	E→TE'
$ E'T'F	id + id * id $	T→FT'
$ E'T'id	id + id * id $	F→id
$ E'T'	+ id * id $	match
$ E'	+ id * id $	T'→ϵ
$ E'T+	+ id * id $	E'→+TE'
$ E'T	id * id $	match
$ E'T'F	id * id $	T→FT'
$ E'T'id	id * id $	F→id
$ E'T'	* id $	match
$ E'T'F*	* id $	T'→*FT'
$ E'T'F	id $	match
$ E'T'id	id $	F→id
$ E'T'	$	match
$ E'	$	T'→ϵ
$	$	E'→ϵ
$	$	match

12/2/2020

X = a = $
M(E, id)
E ⇒ TE'
⇒ FT'E'
⇒ idT'E'

Error
+id+id*id$
M[X,a]

A→aβ1 | aβ2



FIRST
FOLLOW

① A→X
A→Y1Y2Y3...Yn
Computing FIRST
① if X is
② if X→ϵ is
③ if X is a n
then place a in
and ϵ in in

~~FIRST(+) = {+}~~
FIRST(E) =
FIRST(E') = {+,
FIRST(T') = {*,

Computing FOLLOW
① Place $ in
FOLLOW(S)
where S is the
② if there is a
A→α
then everyt
③ if there is a
where FIRS
everything

Policies for error recovery

1) A parser should try to determine that an error has occured as soon as possible.
2) After an error has occured, the parser should pick a likely place to resume parsing.
   A parser should try to parse as much as possible.
3) A parser should try to avoid the error cascade problem.
4) A parser must avoid infinite loops on errors.

Panic mode

The error handler will consume a possibly large no. of tokens in an attempt to find a place to resume parsing. A set of synchronizing tokens (synchronizing tokens) are used for this purpose. When an error is encountered, the parser scans ahead, until one of these synchronizing tokens is seen in input.

Error recovery in LL(1) parser.

1) Pop A from the stack if current input token is $ or in FOLLOW(A). [POP] $BCA | abcd --- $ |
2) Successively pop (remove) tokens from the input until a token is seen for which we can restart the parse.
   [If the current token is not $ and is not in FIRST(A) ∪ FOLLOW(A).
   [SCAN]
3) Push a new non-terminal onto the stack an scan forward until a symbol in the FIRST set of it is found.

	id	+	*	(	)	$
E	E→TE'	Scan	Scan	E→TE'	Pop	Pop
E'	Scan	E'→+TE'	Scan	Scan	E'→ϵ	E'→ϵ
T	T→FT'	Pop	Scan	T→FT'	Pop	Pop
T'	Scan	T'→ϵ	T'→FT'	Scan	T'→ϵ	T'→ϵ
F	F→id	Pop	Pop	F→(E)	Pop	Pop

Z→d | XYZ
Y→c | ϵ
X→Y | a.

FIRST(Z) = {d, a, c}
FIRST(X) = {a, c, ϵ}
FIRST(Y) = {c, ϵ}

FOLLOW(Z) = {$}
FOLLOW(X) = {c, d, a}
FOLLOW(Y) = {d, a, c}

	a	c	d	$
Z	Z→XYZ	Z→XYZ	Z→d Z→XYZ	
Y	Y→ϵ	Y→ϵ Y→c	Y→ϵ	
X	X→Y X→a	X→Y	X→Y	

A grammar is an LL(1) grammar if all productions confirm to the following LL(1) conditions:

1) For each production $A \to \sigma_1 | \sigma_2 | \cdots | \sigma_n$

$FIRST(\sigma_i) \cap FIRST(\sigma_j) = \phi$, for $\forall\ i \neq j$.

2) If non terminal X can derive an empty string, then

$FIRST(X) \cap FOLLOW(X) = \phi$.

18/2/2020

Bottom-up parsing

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

id + id + id

$E \Rightarrow E+T$
$\Rightarrow E+F$
$\Rightarrow E+id$
$\Rightarrow E+T+id$
$\Rightarrow E+F+id$
$\Rightarrow E+id+id$
$\Rightarrow T+id+id$
$\Rightarrow F+id+id$
$\Rightarrow id+id+id$

rightmost derivation

Handle

(matches with RHS of a rule in the that matches with the

A handle is a string together with the pair in the right sentential form where it occurs and whose reduction to the nonterminal on the left side of the production represents one step along the reverse dirn. of the rightside of a prod.

Handle pruning

	Stack	Input	shift-reduce parsing
initial	$	w$	
final	$ S	$	
	$ id	id + id + id $	shift
	$ F	+ id + id $	reduction
	$ T	+ id + id $	
	$ E	+ id + id $	
	$ E+	id + id $	shift
	$ E+id	+ id $	
	$ E+F	+ id $	
	$ E+T	+ id $	
	$ E	+ id $	
	$ E+	id $	
	$ E+id	$	
	$ E+F	$	
	$ E+T	$	
	$ E	$	

viable prefix

LR(1) → 0 to

Left to right scan

Right most deri

SLR

Simple LR parser

closure

goto

closure (i)

is computed for a set of Items I

① Every item in I is added to closure (I)

② If $A \to \alpha.B\beta$ is in closure(I) and $B \to \gamma$ is a production, then $B \to .\gamma$ is added to closure (I).

$E' \to E$
$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

augmented grammar

I_0 : $E' \to .E$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

Goto operation

goto (I, X) is the closure of the set of all items $[A \to \alpha X . \beta]$ such that

I_1 : goto (I_0, E)
$E' \to E.$
$E \to E. + T$

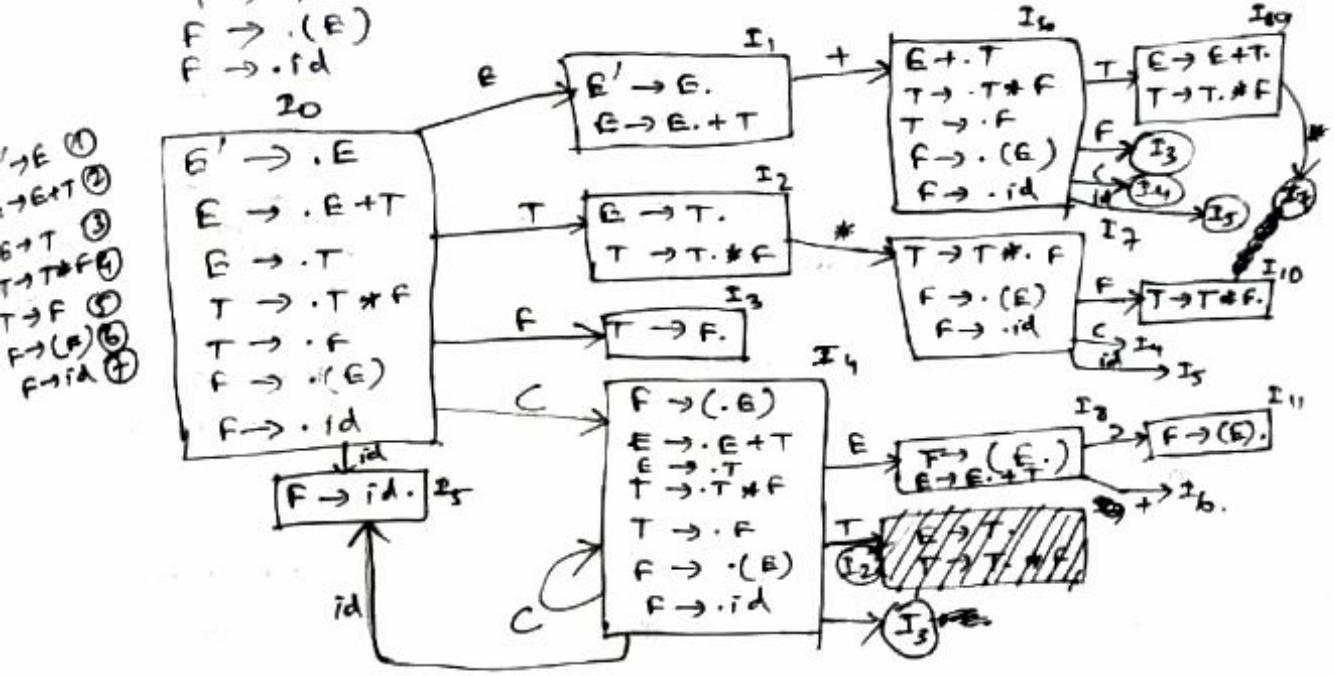
I_2 : goto (I_0, T)
$E \to T.$
$T \to T. * F$

I_3 : goto (I_0, F)
$T \to F.$

19/2/2020

I_4 : goto $(I_0, ()$
$F \to (.E)$
$E \to .E + T$
$E \to .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_5 : goto (I_0, id)
$F \to id.$



	Action						Go to		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r3	s7		r3	r3			
3		r5	r5		r5	r5			
4	s5			s4			8	2	3
5		r7	r7		r7	r7			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r2	s7		r2	r2			
10		r4	r4		r4	r4			
11		r6	r6		r6	r6			

$A \to \alpha.a\beta$
I_i
I_j s_j

If $A \to \alpha.$ in I_i
action $[i,a]$
reduce $A \to \alpha$
follow(A)

$S_0 X_1 S_1 X_2 S_2 \cdots X_m S_m$
action $[S_m, a_i]$ = reduce $A \to \beta$
$S_0 X_1 S_1 X_2 S_2 \cdots X_{m-r} S_{m-r} A S$
S = goto $[S_{m-r}, A]$

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

Stack	Input	Action
$0	id+id$	shift
$0id5	+id$	reduce
$0F3	+id$	reduce
$0T2	+id$	reduce
$0E1	+id$	shift
$0E1+6	id$	shift.
$0E1+6id5	$	reduce
$0E1+6F3	$	reduce
$0E1+6T9	$	reduce
$0E1	$	
$		

1) Construct $C = \{I_0, I_1, \dots, I_n\}$, the ~~LR(0)~~ collect of sets of LR(0) items for G'.

2) State i is constructed as follows:

a) If $[A \to \alpha . a\beta]$ is in I_i and goto $(I_i, a) = I_j$ then set action$[i, a]$ to "shift j"

b) If $[A \to \alpha.]$ is in I_i, then set action $[i, a]$ to "reduce for all 'a' in FOLLOW(A). A may not be S'.

c) If $[S' \to S.]$ is in I_i, then set action $[i, \$]$ to "accept".

3) Goto transition for state i are constructed for all non terminals A. If goto$(I_i, A) = I_j$, then goto$[i, A] = j$.

4) All other entries are made error.

24/2/2020

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

$S \to L.=R$
$R \to L.$

=

($\$ =$)

shift reduce conflict

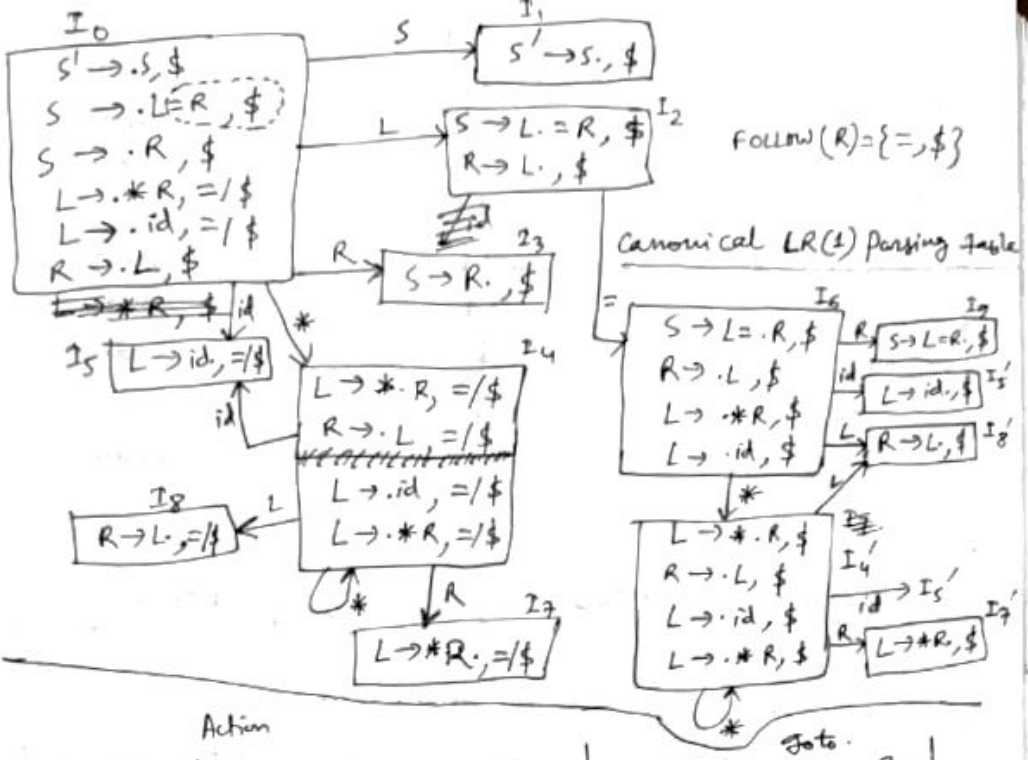
$A\gamma \Rightarrow \alpha\beta$
$A\alpha$

① $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L$
$\Rightarrow L=id \Rightarrow id=id.$

② $S' \Rightarrow S \Rightarrow L=R \Rightarrow L=L \Rightarrow L=id \Rightarrow *R=id. \Rightarrow *L=id \Rightarrow *id=id$

$\alpha A\beta \Rightarrow \alpha B\beta$ $\quad \frac{A \to B}{A \to C}$ $\quad A\gamma \overset{*}{\Rightarrow} C\gamma$ $\quad$ FIRST (β) in FOLLOW(A).
FIRST (β) U FIRST $(\gamma) \to$ FOLLOW(A)

Closure function for LR(1) items –
For each item $A \to \alpha . B\beta$, X in state I, each production $B \to \gamma$ in the grammar, and each terminal b in FIRST(βX), add $B \to .\gamma, b$ to I.



Canonical LR(1) Parsing table

	Action					Goto		
	id	*	*	=	$	S	L	R
						1	2	3
0	S5		S4					
1					accept			
2				S6	reduce R→L			
					reduce S→R			
3							8	7
4	S5		S4					
	~~reduce L→id~~			reduce L→id	reduce L→id		8'	9
5	~~reduce L→id~~							
6	S5'		S4'					
				reduce L→*R	reduce L→*R			
7				reduce R→L	reduce R→L			
8					reduce S→L=R		8'	7'
9								
4'	S5'		S4'					
5'					reduce L→id			
7'					reduce L→*R			
8'					reduce R→L			

LALR (Look ahead LR) (yacc)

merge I4, I4' I5, I5' (core items same)

(space, time compromise) I8, I8' (union of terms (follows))

I7, I7'

reduce-shift conflicts

reduce-reduce conflict X