23/07/2024

BOOK
Compiler Design & Techniques - Ullman

Compilers principles Techniques & Tools. - Ullman.

Two phases of compilation

Analysis Phase includes -
- Lexical Analyser
- Syntax Analyser
- Semantic Analyser.

Synthesis Phase includes:-
- Intermediate Code Generator
- Code Optimisation.
- Code Generator.

Symbol table → datastructure to hold new variables (name, type, line number, memory location)

During lexical analysis phase, some lexical error may be detected (say, let there be a pattern for variable names) → It is detected in the first phase. ~~It~~
lexical analyser generates a set of tokens, which is fed to syntax analyser, which in turn, checks the syntax of a sentence.

for grammar (ToC)
- V → variables
- Σ → Alphabet
- S → Start symbol
- P → Production Rules

(compiler)

a = b + c → Statement (Assignment statement)
Tokens = a, =, b, +, c (5)
Parse tree is generated.

1) a=b*5.5 ← checking in semantic analyser (e.g. Type mismatch)
   at * float

2) arr[5.5] ← array size mismatch, has to be int

Both the above cases will be passed through syntax analyser but fails at semantic analyser.

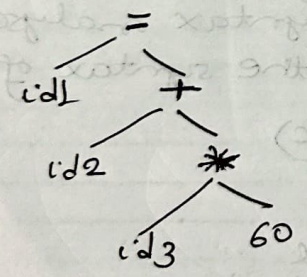
Eg) position = initial + rate × 60 ← 7 tokens.
↓
lexical analyser → determines the class of tokens
↓ type of operators (regular, conditional)

id1 = id2 + id3 * 60
Identifiers

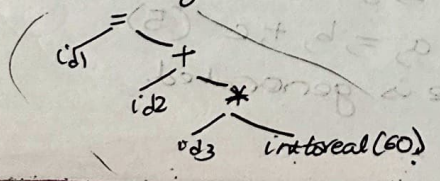
New identifiers are stored in Symbol Table.

Errors are stored in Error Handler
↓
error corrector may be
- automatic
- manual & re-compile.

↓
Syntax Analyser ← Set of production rules (grammar) to check if they are syntatically correct or not.
↓
Starts from start symbol & generates parse tree.



↓
Semantic Analyser ← produces another tree.



checks correct definition.

Each is dependent on the above tools.

↓
Intermediate Code Generator.

one form of 3 address code

temp1 = inttoreal(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3

addr
2
3
3
2

maximum of 3 address-form is possible.

Limitation of resources forces us to break the code into small-sized expressions, one such being 3-address form.

↓
code optimisation (OPTIONAL) // optimisation through no. of statements & temporary variables

temp1 = id3 * 60
id1 = id2 + temp1

↓
Code Generator. ← produces assembly level program.
// Can be derived through Instruction Set

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Compiler s/w tools

- Scanner Generators (lex) :- generates a set of tokens acc. to regular expression,
- Parser generator → It generates a C-code.
- Data-flow analysis → used in code optimisation.
- Code Generator.

Regular expression for variables

$l(d+l)^*$ l → set of letters, d → set of digits

Token → Lexical unit (category of pattern)

Pattern → Set of strings associated with a token. Eg:- Regular expression.

Lexeme → original name of the token

Initially, both function names & variable names are considered as Identifiers.

Regular expressions for keywords are letters of the keyword itself.

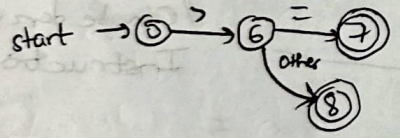
Eg:- int → i, n, t

if → i, f

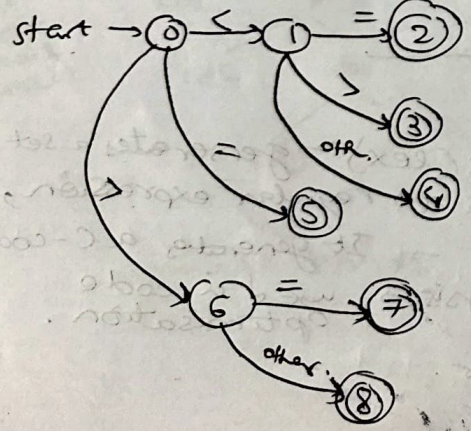
Relational operator (relop) → <, <=, <>, etc.

(relop: category of token)

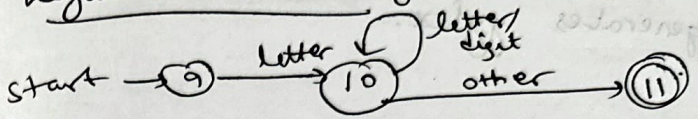
Transition Diagram is used to match the patterns.



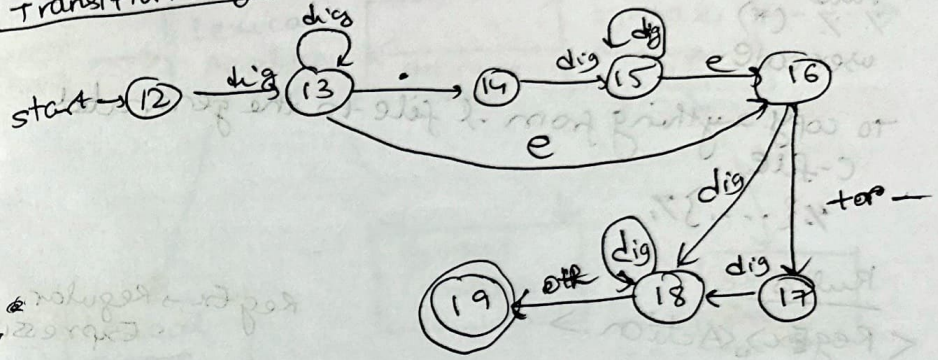
Relops Transition Diagram

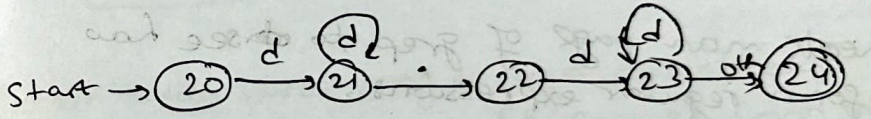


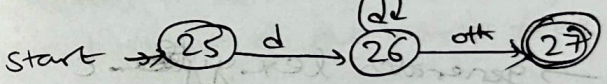
Keywords Transition Diag



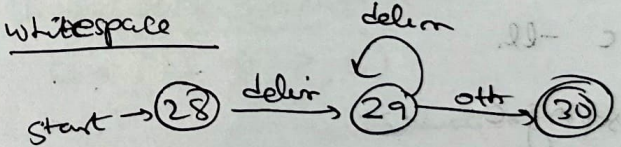
Transition Diag for nos.







Whitespace



FILE *yyin;
char *yytext
int yylex();

IDs of tokens & Integer values to be returned.

WHILE_TOK.
ID_TOK
GT_TOK, - etc.

```
#define LPAREN_TOK '('
# define GT_TOK '>'
# define ID_TOK 350
# define INTCONST 351
```

Header file to be made & included.

for single char, use ASCII

30.07.2024

lex.l ← generates yylex.

Defn
%%
rules
%%
user code.

To copy anything from .l file to the generated C-file
%{ ... }%

Rules
<RegEx> <Action>

RegEx → Regular Expression.

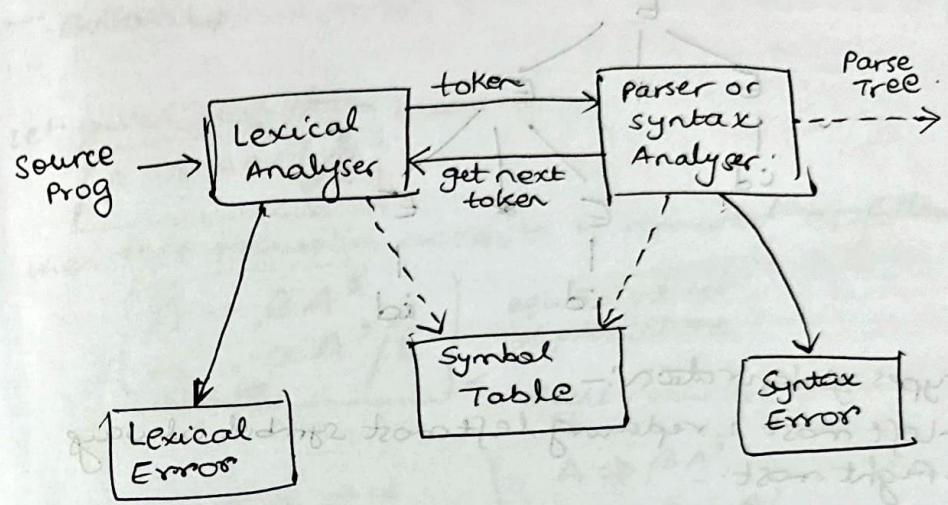
* check man page of grep to see how to form regular expressions.

- Install flex
  lex prog.l → generates lex.yy.c
  cc lex.yy.c -ll
  ./a.out testprog.c

yyler, yylineno ← other variables can be used. Available in manual

Syntax Analyser

05/08/2024



Parser takes source prog as input and checks whether a sentence can be formed from the start symbol of the parse tree generated.

Context Free Grammar

$G = (V, \Sigma, P, S)$ ← production rules

Consider a grammar :- (5 production rules)

$E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$ ← to generate arithmetic expressions like id+id*id

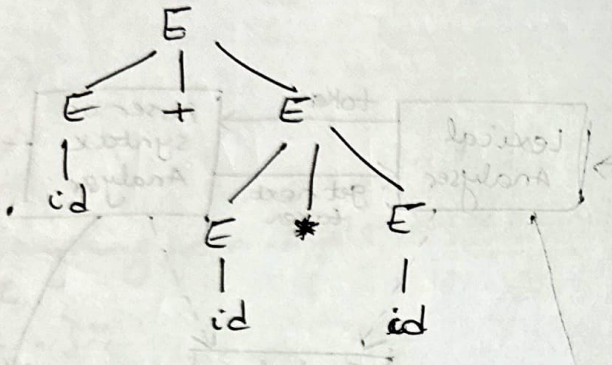
Start symbol is E.

To derive id+id*id from above grammar :-

$E \Rightarrow E+E$
$\Rightarrow E+E*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

can also start from
$E \Rightarrow E*E$
$\Rightarrow E+E*E$

The corresponding parse tree would be



Types of derivation:-
- Left most : replacing left most symbol always
- Right most:

Ambiguous grammar

If a sentence has 2 or more derivations from the start symbol of that grammar, then the grammar is considered to be ambiguous (i.e. derivation is not unique)

Hence the above mentioned grammar, i.e.
$E \to E+E \mid E*E \mid (E) \mid -E \mid id$ is ambiguous as there are 2 derivations (maybe more) for id + id * id.

Elimination of left recursion

A grammar is left recursive if it has a non-terminal 'A' such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.

Two methods of parsing
- Top down (cannot handle left recursive)
- Bottom up

let given production rule is

$$A \to A\alpha / \beta$$

then left recursion needs to be removed as follows

$$A \to \beta A'$$
$$A' \to \alpha A' / \epsilon$$

equivalent set of productions

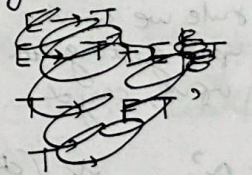
← Right recursive grammar.

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

$A \Rightarrow \beta A'$
$\Rightarrow \beta\alpha A'$
$\Rightarrow \beta\alpha\alpha A'$
$\Rightarrow \beta\alpha\alpha\alpha A'$
$\Rightarrow \beta\alpha\alpha\alpha(\epsilon)$
$\Rightarrow \beta\alpha\alpha\alpha$

Consider,
$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$ ← (id)

Start symbol is E



convert to

$E \to E+T \mid T$ (A, A α, β)

$T \to T*F \mid F$ (A, A α, β)

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

other rules remain same.

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

If grammar is in this form,

$$A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$$

After removal of left recursion, it becomes,

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$$

Left factoring

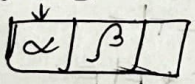
- ~~used in predictive parser~~
- It is a grammar transformation that is useful for producing a grammar suitable for predictive parser.

If the grammar is in the form

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

to choose which production rule we consider upon getting an input, say α

let inputs are α, β then yylex is called by parser & tokens are as follows



equivalent

after left factoring, it changes to

$A \to \alpha A'$ ✓ (chosen if i/p is α)
↳ common portion to all productions

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

it is tried for 2 times as there is β_1 & β_2 (2 such instances of β)

Consider

iEts ← if Else then statement

$S \to iEts \mid iEtseS \mid a$
$E \to b$

↓ After left factoring

$S \to iEtsS' \mid a$
$S' \to eS \mid \epsilon$
$E \to b$

Removes ambiguity of grammar

Different types of parsers

① Top-down parsing (starts from start symbol)
② - Recursive descent parsing
  - Predictive parser
  - Non Recursive predictive parser

② Bottom up parsing (starts from leaf)
- Shift reduce parser
- Operator precedence parser
- LR parser ← for look ahead
  - SLR
  - LALR
  - CLR (canonical)

fn yyparser() calls yylex() for token

06/08/2024

Top down parsing

- Recursive descent parsing

Consider the grammar

$S \to cAd$

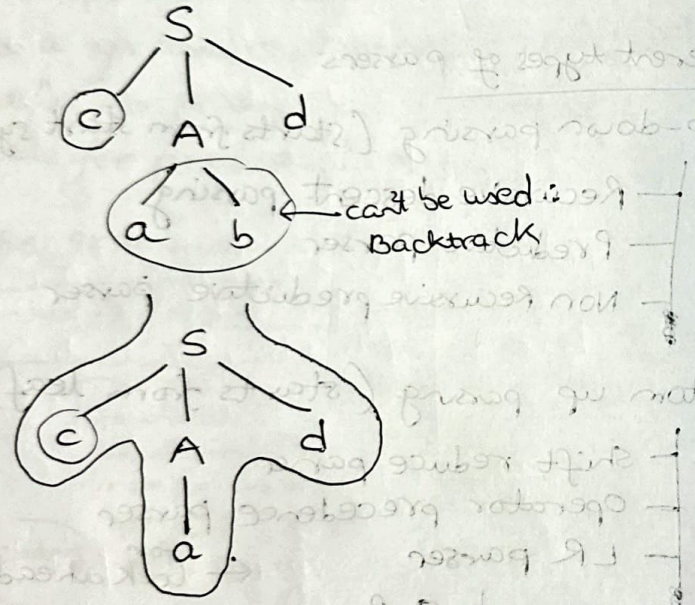
$A \to ab \mid a$

word I/P

$w = cad$

leftmost symbol (read →)

Start from start symbol S



can't be used : Backtrack

use left factoring

- Predictive Parser

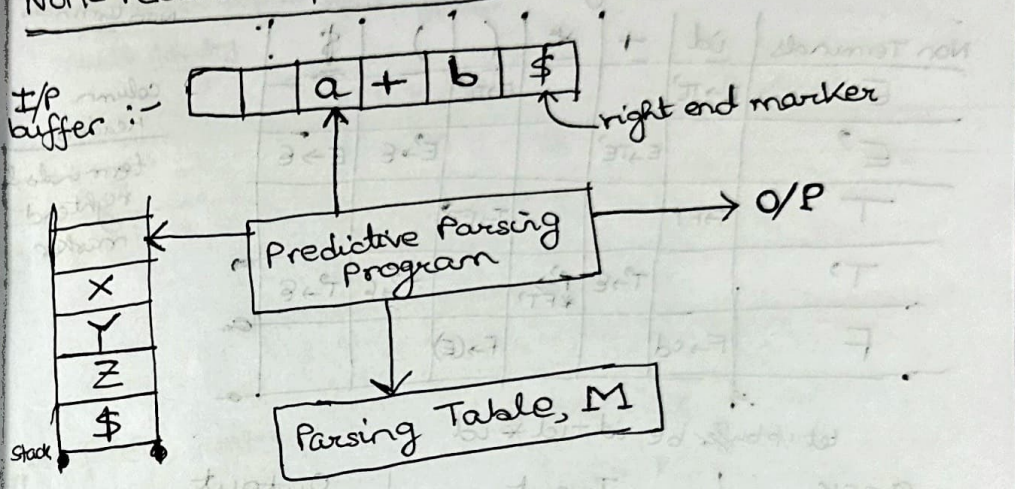
$A \to aA'$

$A' \to b \mid \epsilon$

} Rewriting $S \to ab \mid a$

No backtracking reqd.

Non-recursive predictive parser



The action includes: Inserting / Removing something from the stack

Parsing actions

1. if $X = a = \$$, then parser halts and announces the successful completion of the parsing
2. if $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. if X is a non-terminal, the parsing program consults the entry of parsing table, $M[X, a]$

---

consider a grammar:

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

Start symbol
E

Parsing Table

Rowheads
Non terminals

Non Terminals	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)		

column headers: terminals + rightend marker

let i/p buffer be id+id*id

Stack	Input	Output
① $E	id+id*id$	← Initial config.
② $E'T	id+id*id$	E→TE'
③ $E'T'F	id+id*id$	T→FT'
④ $E'T'id	id+id*id$	F→id

[1] Initialisation

[2] Find E & id entry in parse table.

E → T E'

← read in this fashion, into the stack

[3] then find (row: T (stack head), col: id.)

T → F T'

then find (row: F (stack head), col: id.) ; F→id.

[5] Now stack top is id, a terminal, so pop from both stack top & input symbol.

Table contd

	Stack	Input	Output
⑤	$E'T'	+id*id$	
⑥	$E'	+id*id$	

⑥ if ε, nothing is pushed to stack

Table contd

Stack	Input	Output

Upon passing a blank cell, the sentence is adjudged to be invalid [i.e. blank cell entry in the table.

Display error or call recovery routine from blank cell.

13/08/2024

Construction of Parsing Table

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

$FIRST(E) = FIRST(T) = FIRST(F) = \{(, id\}$

$FIRST(E') = \{+, \epsilon\}$

$FIRST(T') = \{*, \epsilon\}$

$FOLLOW(E) = FOLLOW(E') = \{), \$\}$

$FOLLOW(T) = FOLLOW(T') = \{+,), \$\}$

$FOLLOW(F) = \{*, +,), \$\}$

$E \to TE'$

$FIRST(TE') = FIRST(T) = \{(, id\}$

Let parsing table be M.

$M[E, (]$ & $M[E, id]$ ← create entry in these two cells, contents of cell is $E \to TE'$

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

$FIRST(+TE') = \{+\}$

$M[E', +]$ ← entry will be $E' \to +TE'$

$E' \to \epsilon$

$FIRST(\epsilon)$ ← if empty symbol, then find follow of E.

$= FOLLOW(E') = \{), \$\}$

$M[E',)]$ & $M[E', \$]$. ← entry will be $E' \to \epsilon$.

LL(1) grammar

A grammar is said LL(1) if any entry in its parsing table contains one symbol, i.e. the parsing table does not contain multiple entries.

(Not LL(1) grammar) (*) Qs

Left grammar

1) $S \to iEtSS' \mid a$
   $S' \to eS \mid \epsilon$
   $E \to b$

   start symbol: S

   } Ambiguous grammar has multiple entries.

2) $E \to E+T \mid T$
   $T \to TF \mid F$
   $F \to F* \mid a \mid b$

   start symbol: E

   a) Eliminate left recursion
   b) Compute fast follow
   c) Construct parsing table
   d) Check if grammar is LL(1) or not.
   e) Input string: a+a+a.

Bottom-Up Parsing

Consider grammar

$S \to aABe$

$A \to Abc \mid b$

$B \to d$

word: abbcde

Choose a substring.

Say, b. then it matches with $A \to b$.

So, aAbcde

Choosing Substrings

1) a b b c d e — A → B
2) a A b c d e — A → Abc is the handle.
3) a A d e — B → d
4) a A B e — S → aABe is the handle.
5) S

Handle Pruning

Rightmost derivation in Reverse.

Consider grammar

E → E+E
E → E*E
E → (E)
E → id

I/P string

$id_1 + id_2 * id_3$

Right Sentential Form	Handle	Reducing Production
$id_1 + id_2 * id_3$	id_1	$E \to id$
$E + id_2 * id_3$	id_2	$E \to id$
$E + E * id_3$	id_3	$E \to id$
$E + E * E$	$E*E$	$E \to E*E$
$E + E$	$E+E$	$E \to E+E$
E		

Handle Pruning ↓

Can be implemented through shift reduced parser.

Stack implementation of shift reduce parser

	Stack	Input Buffer	Action
1)	$	$id_1 + id_2 * id_3$ $	shift
2)	$id_1	$+ id_2 * id_3$ $	Reduce with E→id
3)	$E	$+ id_2 * id_3$ $	accept
4)	$E+	$id_2 * id_3$ $	shift
5)	$E+id_2	$* id_3$ $	shift
6)	$E+E	$* id_3$ $	Reduce with E→id
7)	$E+E*	id_3 $	shift
8)	$E+E*id_3	$	Reduce with E→id
9)	$E+E*E	$	Reduce with E→E*E
10)	$E+E	$	Reduce with E→E+E
11)	$E	$	accept

4 types of actions are possible:-
1. Shift
2. Reduce
3. Accept
4. Error

} parsing table guides the action.

Operator precedence & LR parsing

1) Operator Precedence Parsing

OPERATOR GRAMMAR

(I) No production right side is ε

(2) Has no production right side with two adjacent non-terminals

$E \to EAE \mid (E) \mid -E \mid id$ — Not operator grammar (3 consecutive non-terminals)

$A \to + \mid - \mid * \mid / \mid \uparrow$

Conversion to operator grammar

$E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid id$ (UE)

Relation	Meaning
$a \lessdot b$	a yields precedence to b
$a \doteq b$	a has the same precedence as b
$a \gtrdot b$	a takes precedence over b

Parsing Table

Row & Column headers → both terminal symbols.

	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

(Left / Right)

Rule is self defined

empty cells are errors

i/p string : $ id + id + id $ (add if not present)

$ \lessdot id \gtrdot + \lessdot id \gtrdot * \lessdot id \gtrdot $

to be chosen as substring

After replacement,

$ E + E * E $ → $ E + E $

Remove all non terminals

$ + * $

$ \lessdot + \lessdot * \gtrdot $ — substring.

$ \lessdot + \gtrdot $

$ E $

start symbol reached → accepted ✓ syntax correct

20/08/2024

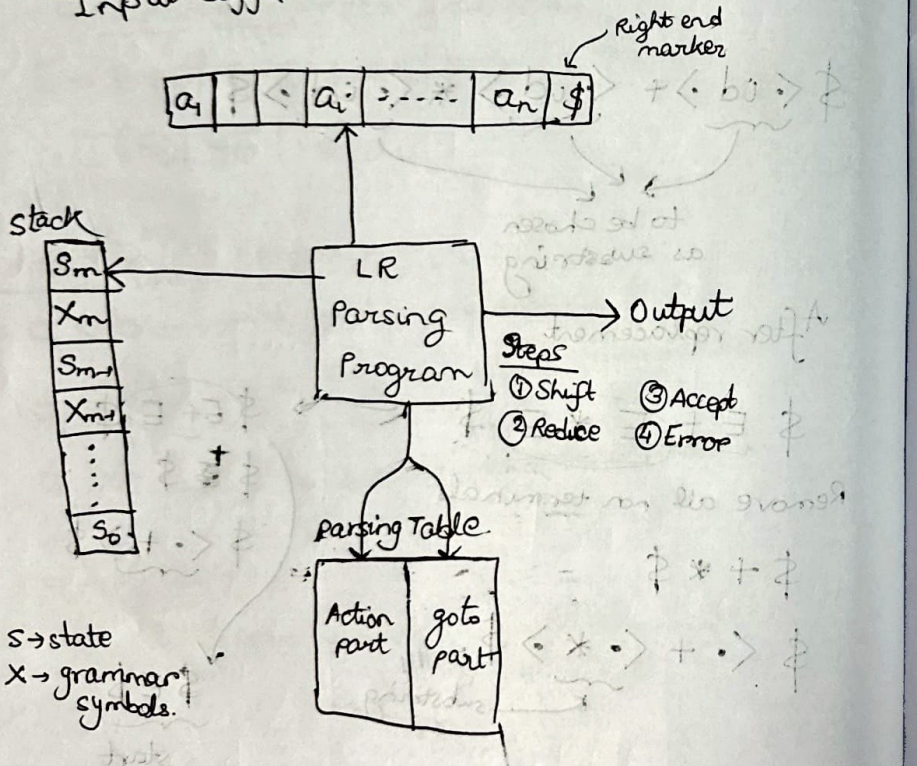
LR Parser

Technique : LR(K) Passing

L → Scan the I/P from Left to Right
R → Right-most derivation is produced in reverse
K → in every step of derivation, no. of symbols needs to be consumed.
[no. of I/P symbols]

Default value of K = 1

Input buffer



construction of passing table methods :-
(a) SLR : Simple LR
(b) CLR : Canonical LR
(c) LALR : LookAhead LR

Consider the foll. production rules

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

Parsing Table

State	Action (terminals)						Goto (Non-terminals)		
	id	+	*	(	)	$	E	T	F
0	S_5	.	.	S_4	.	.	1	2	3
1	.	S_6	.	.	.	Accept			
2	.	r_2	S_7	.	r_2	r_2			
3	.	r_4	r_4	.	r_4	r_4			
4	S_5	.	.	S_4	.	.	8	2	3
5	.	r_6	r_6	.	r_6	r_6			
6	S_5	.	S_4	.	.	.		9	3
7	S_5	.	S_4	.	.	.			10
8	.	S_6	.	.	S_{11}	.			
9	.	r_1	S_7	.	r_1	r_1			
10	.	r_3	r_3	.	r_3	r_3			
11	.	r_5	r_5	.	r_5	r_5			

S_i → shift & stack state i
r_j → reduce by prod^n no j
& pop out 2 × no. of symbols in RHS of prod^n Rule j

Initial stack symbol = 0

	Stack	Input	Action	from parse table
①	0	id*id+id$	Shift	check [0][id]
②	0 id 5 (5 ← state no.) (id popped)	*id+id$	Reduce by r6 production F→id.	[5][*]
③	0 F 3 (F as per r6)	*id+id$	Reduce by r4 production T→F	[0][F]
④	0 T 2 (T as per r4)	*id+id$	Shift	[2][*]
⑤	0 T 2 * 7 (7 shifted ← SN)	id+id$	Shift	
⑥	0 T 2 * 7 id 5 (id popped)	+id$	Reduce by r6 production F→id.	
⑦	0 T 2 * 7 F 10 (popped)	+id$	Reduce by r3 T→T*F	[10][+]
⑧	0 T 2			

... and so on

YACC (.y extension)

Takes specification of CFG, i.e. production rules of context free grammars as input.

Produces a parser yyparse() as o/p in C code.

default file: y.tab.c

yacc -d ← to generate y.tab.h

Format of YACC file:

```
defn
%%
rules
%%
code
```

%{ ... %} copied directly to C-file

Token name: %token

without % ← non-terminals

$A \rightarrow B_1 B_2 \dots B_m \mid C_1 C_2 \dots C_n \mid D_1 D_2 \dots D_k$ } format in yacc.

LAB

→ main fn,
→ Statements (like local/global decl.)
- Assignment
- conditional
- Iterative.
- Fn call.

→ user defined fns.

} Write yacc specs to design a parser for a subset of the C language. Consider the foll. prog. lang. constructs

Same language prev. considered.

Variables of data types allowed in the subset can be defined at the very beginning of any fn/block. A variable must be defined before it is used.

lex pTest.l
yacc pTest.y -d ← to generate header file with token IDs automatically generated.

cc y.tab.c -ll
./a.out i/p file

whole C program starting from start symbol (main) is like a sentence, to be parsed using prodn rules, part-by-part.

LR Parsing

- SLR
- CLR
- LALR

Constructing SLR parsing table:

The grammar for which an SLR table can be constructed, is called SLR grammar.

Default value of $k = 0$.

LR(0) items

Consider a grammar production rule.

$A \rightarrow XYZ$

Possible items of this prod^n rule. (i.e. somewhere on RHS, a dot (·) will be there):

$A \rightarrow \cdot XYZ$
$A \rightarrow X \cdot YZ$
$A \rightarrow XY \cdot Z$
$A \rightarrow XYZ \cdot$

} LR(0) items

SLR table is based on canonical LR(0) items.

Canonical LR(0) items

- First the grammar needs to be augmented. (#)
- Use closure & goto operator.

(#) Augment the grammar by introducing a new start symbol. [In the derivation, if the augmented rule appears, then parsing needs to be stopped].

let) Existing start variable: S
Introducing a new start symbol (S')
Add prod^n rule: $S' \rightarrow S$.

consider a grammar (already augmented)

New start symbol: E'

$E' \rightarrow E$
① $E \rightarrow E+T \mid T$ ②
③ $T \rightarrow T*F \mid F$ ④
⑤ $F \rightarrow (E) \mid id$ ⑥

① - ⑥ : Rule Nos.

Initial item (Augmented prod^n rule)

$I = \{[E' \rightarrow \cdot E]\}$; Initial set of items. (put (·) on RHS start)

Find the closure of I

I_0 = closure (I) =:

$E' \rightarrow \cdot E$ // Items themselves are included.

{If after (·) there is a non-terminal, then we need to consider closure of that non terminal

$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$

T is a non terminal,
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$.

closure means all rules starting with that non-terminal symbol.

Next, find goto of I_0

whatever symbol 's' is after dot(·), we need to find goto(I_0, S)

shift dot (·) one place right

After dot if there is a non terminal (same) in any rule, then that closure is included.

I_1 = goto(I_0, E):

$E' \to E\cdot$

$E \to E\cdot + T$

I_2 = goto(I_0, T):

$E \to T\cdot$

$T \to T\cdot * F$

I_3 = goto(I_0, F):

$T \to F\cdot$

~~I_4 = goto$(I_0, ($~~

I_4 = goto$(I_0, ($):

$F \to (\cdot E)$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

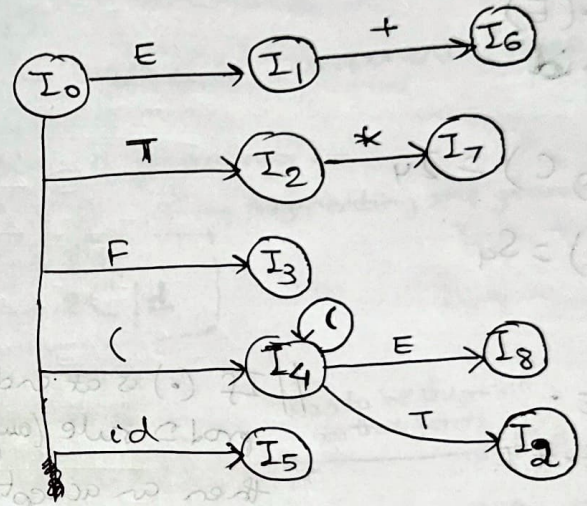
I_5 = goto(I_0, id):

$F \to id\cdot$

Then recursively find goto of each I_n set formed.

incomplete

Transition Diagram (constructed from goto's)



Next, find the parsing table:

Col headers: Terminals, Right end marker, goto symbols (non terminals)

Row headers: No. of sets I_n.

Populate table based on Transition diagram.

I_0

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

// parsing table (M) drawn earlier.

$goto(I_0, C) = I_4$

$M(0, C) = S_4$

I_1

$E' \to E \cdot$
$E \to E \cdot + T$

// If (.) is at end of prod^n rule (augmented), then an accept needs to be created, i.e.

$M(1, \$) = Accept$

$M(1, +) = S_6$

I_2

$E \to T \cdot$
$T \to T \cdot * F$

// If (.) is at end of prod^n rule (Non-augmented), then we need to find follow of LHS Non-terminal symbol of that prod^n rule.

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

Since $E \to T$ is r_2, hence

$M(I_2, \$) = M(I_2, +) = M(I_2,)) = r_2$

03/09/2024

Construct Canonical LR parsing table

$[A \to \alpha \cdot \beta, a]$ ← Extra info is incorporated in a

LR(1) items

where
$A \to \alpha\beta$: grammar production rules
a : terminal / Right end marker.

LR(1) ← length of 2nd component (Here, a)

Consider a grammar :-

$S' \to S$ ← Augmenting the grammar
$S \to CC$
$C \to cC \mid d$

Initial Item

$I = \{[S' \to \cdot S, \$]\}$ → has to be shifted to this form → $[A \to \alpha \cdot B\beta, a]$

$A = S'$
$\alpha = \epsilon$
$B = S$
$\beta = \epsilon$
$a = \$$

closure of I

I_0 = closure(I):

$\{[S' \to \cdot S, \$]$

// $FIRST(\beta a) = FIRST(\$) = \{\$\}$

FIRST(βa) ← second component of closure

2. $S \to \cdot CC, \$$
3. $C \to \cdot cC, \$/c/d$
   $C \to \cdot d, \$/c/d$

$A = S$	$A = S'$
$\alpha = \epsilon$	$\alpha = \epsilon$
$B = C$	$B = S$
$\beta = C$	$\beta = \epsilon$
$a = \$$	$a = \$$

$FIRST(Ba) = First(C) = \{c, d\}$

↳ after : of I_0

I_1 = goto (I_0, S) :

$S' \to S \cdot, \$$

I_2 = goto (I_0, C)

$S \to C \cdot C, \$$ — after · there is a non-terminal

$C \to \cdot\, cC, \$$

$C \to \cdot\, d, \$$ } same symbol as that being referred to above.

I_3 = goto (I_0, c)

$C \to c \cdot C, c/d$

$C \to \cdot\, cC, c/d$

$C \to \cdot\, d, c/d$

I_4 = goto (I_0, d)

$C \to d \cdot, c/d$

$\Rightarrow$ goto $(I_1) \to$ (X) After (·) nothing is there

I_5 = goto (I_2, C)

$S \to CC \cdot, \$$

I_6 = goto (I_2, c)

$C \to c \cdot C, \$$

$C \to \cdot\, cC, \$$

$C \to \cdot\, d, \$$

I_7 = goto (I_2, d)

$C \to d \cdot, \$$

I_8 = goto (I_3, C)

$C \to cC \cdot, \$/c/d$

$\Rightarrow$ goto $(I_3, c) = I_3$

$\Rightarrow$ goto $(I_3, d) = I_4$

$\Rightarrow$ I_4 & I_5 → NO gotos (dot at end)

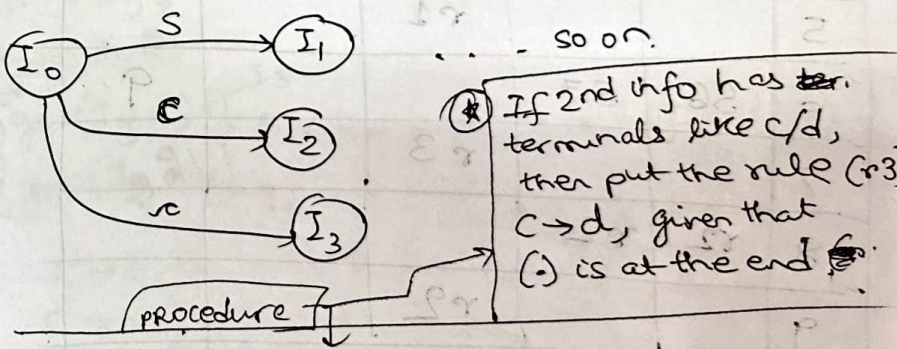
$\Rightarrow$ goto $(I_6, c) = I_6$

$\Rightarrow$ goto $(I_6, d) = I_7$

I_9 $\Rightarrow$ goto (I_6, C)

$C \to cC \cdot, \$$

Transition Diagram



(*) If any item I_n has

$S' \to S \cdot$ (augmented prod^n rule with dot at end & $ as 2nd info, then put 'Accept' in that cell.

Writing the Rules again

$S' \to S$ ← augmenting given

① $S \to CC$

② $C \to cC \mid d$

S3 → shift, stack top = 3

No. of states

Parsing Table

State	Action			goto	
	c	d	$	S	C
0	S_3	S_4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Construct LALR parsing table

① Construct the LR(1) items

② States that have same types of productions : Take union of that

Consider the (same as CLR) following grammar :-

$S' \to S$

$S \to CC$

$C \to cC \mid d$

I_0 :

$S' \to \cdot S, \$$

$S \to \cdot CC, \$$

$C \to \cdot cC, c/d$

$C \to \cdot d, c/d$

$I_1 = goto(I_0, S) \to$ ~~$I_0 \odot S, \$$~~

$S' \to S \cdot, \$$

$I_2 = goto(I_0, C)$

I_{36} ← new name after union

$goto(I_0, c) \leftarrow I_3$

$goto(I_2, c) \leftarrow I_6$

Similar prod^n rule, diff 2nd symbol.

$C \to c \cdot C, c/d, \$$

$C \to \cdot cC, c/d, \$$

$C \to \cdot d, c/d, \$$

$I_{47} = \{ goto(I_0, d) \leftarrow I_4 ,\ goto(I_2, d) \leftarrow I_7 \}$ merged

$C \to d \cdot, c, d, \$$

I_{89} { goto$(I_3, C) \to I_8$; goto$(I_6, C) \to I_9$ } merged

$C \to cC,\ c/d/\$$

parsing table

state	Action c	Action d	Action $	goto S	goto C
0					
1					
2					
36					
47					
5					
89					

fill as like CLR