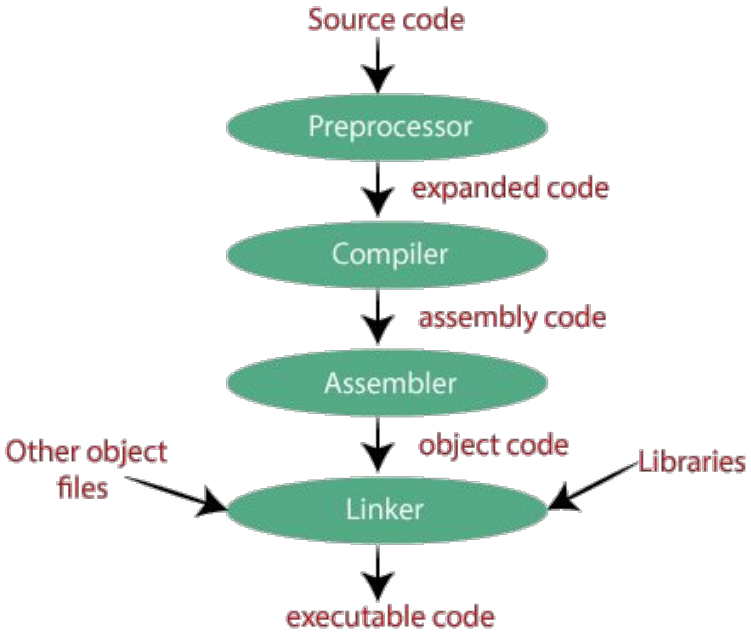
Parsing (Lex+Yacc)

Design Lab
CS 69202

Prof. Niloy Ganguly
Department of Computer Science and Engineering

IIT Kharagpur

Execution of a Compiled Program



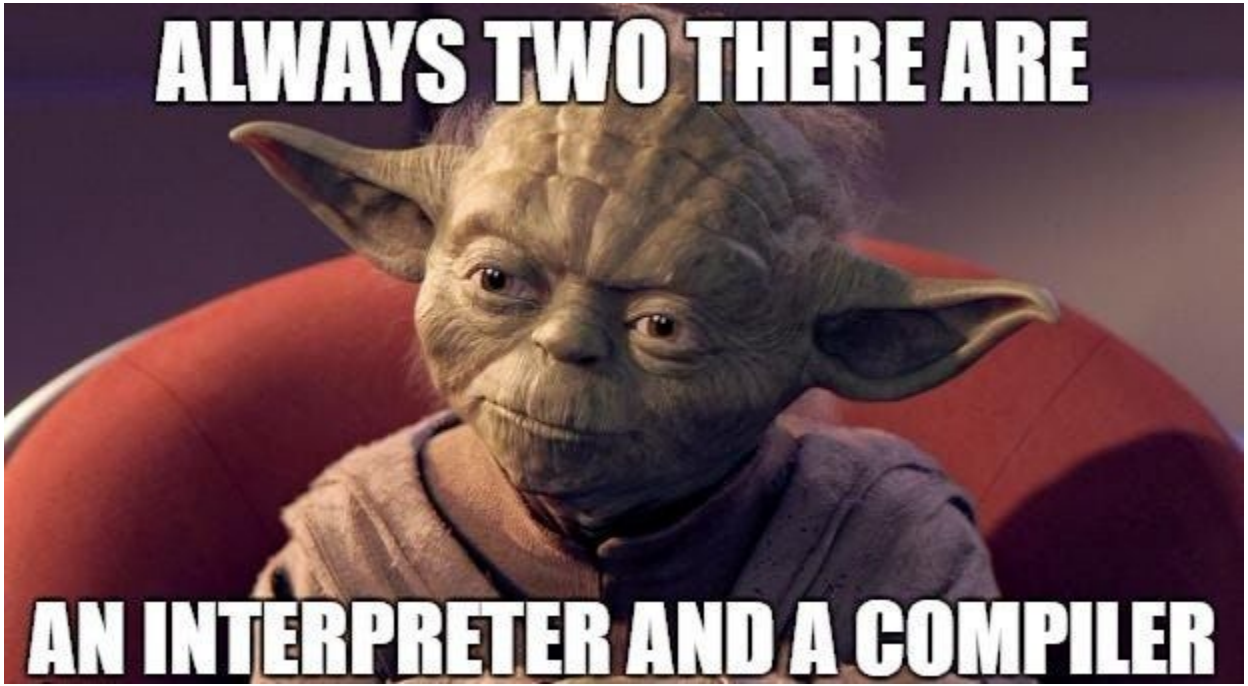
Compilers

- Definition : A computer program that translates computer code written in one programming language (the **source** language) into another language (the **target** language)
- Example (Language/Compiler):
  - C/GCC,Clang
  - C++/G++,Clang++
  - JAVA,Javac

Interpreters

- Definition : A computer program that **directly executes instructions** written in a programming or scripting language, **without requiring them previously** to have been compiled into a machine language program.
- Example (Language/Compiler):
  - Python,PyPy
  - Ruby, CRuby

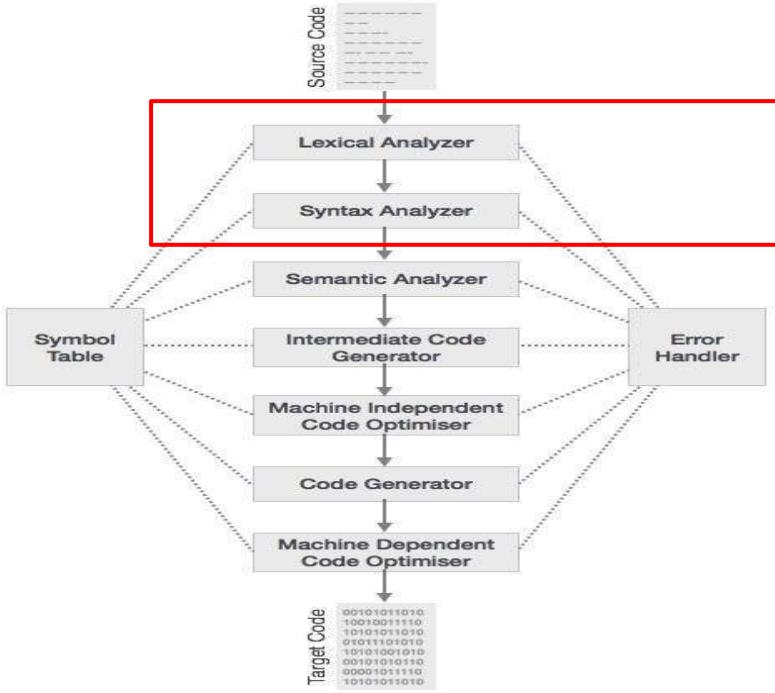
Compiler vs Interpreter



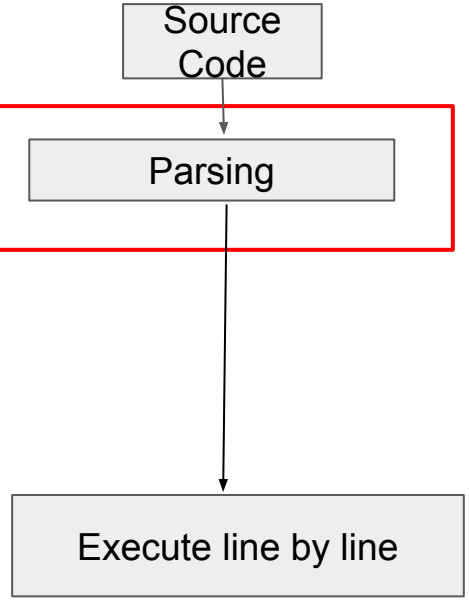
Compiler vs Interpreter

Compiler	Interpreter
• A compiler takes the entire program in one go.	• An interpreter takes a single line of code at a time.
• The compiler generates an intermediate machine code.	• The interpreter never produces any intermediate machine code.
• The compiler is best suited for the production environment.	• An interpreter is best suited for a software development environment.
• The compiler is used by programming languages such as C, C ++, C #, Scala, Java, etc.	• An interpreter is used by programming languages such as Python, PHP, Perl, Ruby, etc.

Compiler vs Interpreter



Compiler Phases



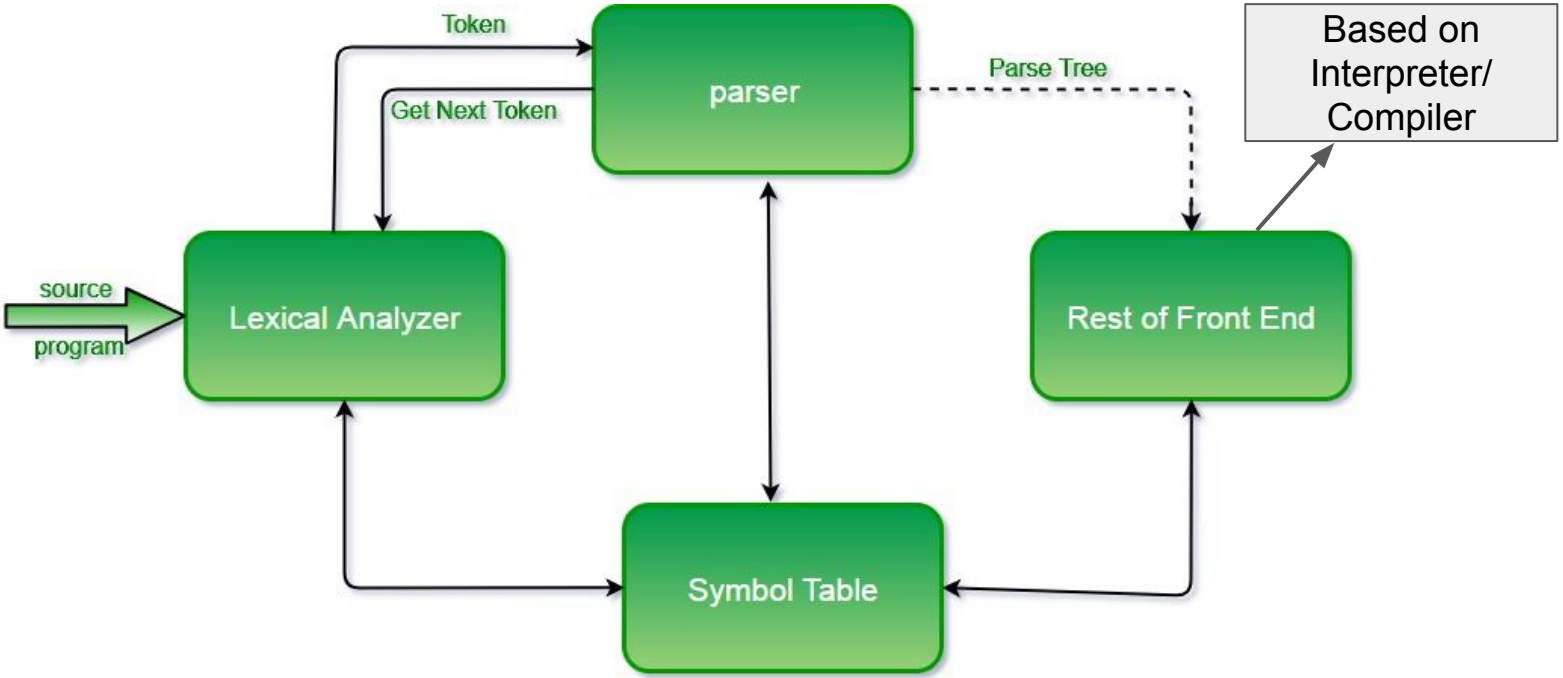
Interpreter Phases

Common Phase of Interpreter and Compiler

Parsing

- Definition : The process of **analyzing** a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar.
- Comes from Latin pars (orationis), meaning part (of speech)
- This consists of two parts : **Lexical Analysis (Lexer)** and Syntax Analysis (Parser).
- Lex and YACC are tools for Lexer and Parser respectively.
- For this course, we use **Python Lex YACC (PLY)** package.

Parsing Stages



Lexical Analyzer (Lexer)

- Lexical analysis is the process of breaking down the source code of the program into smaller parts, called tokens, such that a computer can easily understand.
- These tokens can be individual words or symbols in a sentence, such as keywords, variable names, numbers, and punctuation.
- Also known as a scanner
- Can be enumerated with the Deterministic Finite Automata.
- The output generated from Lexical Analysis are a sequence of tokens sent to the parser for syntax analysis.

Token

- A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages/documents.
- Example
  - Type token (id, number, real, . . .)
  - Punctuation tokens (IF, void, return, . . .)
  - Alphabetic tokens (keywords)

Lex

- Lex is a tool or a computer program that generates Lexical Analyzers (converts the stream of characters into tokens)
- In context of PLY, lex.py is used to tokenize an input string.
- For example, suppose you're writing a programming language and a user supplied the following input string: x = 3 + 42 * (s - t)
- A tokenizer splits the string into individual tokens : 'x','=', '3', '+', '42', '*', '(', 's', '-', 't', ')'
- Tokens are usually given names to indicate what they are. They are written in Uppercase Letters.
- For example: 'ID','EQUALS','NUMBER','PLUS','NUMBER','TIMES', 'LPAREN','ID','MINUS','ID','RPAREN' (in order as above)

Lex

- The input is broken into pairs of token types and values.
- The example x = 3 + 42 * (s - t) can be converted to :

```
('ID','x'), ('EQUALS','='), ('NUMBER','3'),
('PLUS','+'), ('NUMBER','42), ('TIMES','*'),
('LPAREN','('), ('ID','s'), ('MINUS','-'),
('ID','t'), ('RPAREN',')'
```

Steps to make a Lexer

- Initialise a tuple (or list) of tokens in a variable **‘tokens’** to be used for Lexer.

```
1  tokens = (
2      'NUMBER',
3      'SUM',
4      'DIVIDE',
5      ...
6  )
```

Steps to make a Lexer

- Declare tokens with the names in the token tuple as t_<TokenName>. Declare it either in variable (for matching expressions and no action):

```
t_NUMBER=r'[0-9]+'
```

- or in function (to get value of token for your objective or to perform some actions in token :

```
def t_NUMBER(t):
    r'[0-9]+' #First statement of function will be
              #a regular expression in r-strings
    t.value = int(t.value) #This feature is available
                           #in function format. You can
                           #edit these properties like value here.
                           #See documentation for More.
    return t #return the value of token
```

Steps to make a Lexer

- There will be tokens like **t_newline, t_ignore** and **t_error** (all in **lowercase** - **default token functions of PLY**) which has certain functions like handling new lines, ignores a character and handling errors respectively.
- The r-strings (like r'[0-9]+') are called Regexes (discussed later).
- For more, see documentation : https://www.dabeaz.com/ply/
- An example code snippet is shown next.

Lex - Code Snippet

```
import ply.lex as lex

# List of token names.   This is always required
tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
)

# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

```
# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()
```

r strings denote regular expression patterns

t.value denotes the type of value collected by the regular expression. Default string type.

Lex - Code Snippet

```
# Test it out
data = '''
3 + 4 * 10
  + -20 *2
'''

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break      # No more input
    print(tok)
```

```
$ python example.py
LexToken(NUMBER,3,2,1)
LexToken(PLUS,'+',2,3)
LexToken(NUMBER,4,2,5)
LexToken(TIMES,'*',2,7)
LexToken(NUMBER,10,2,10)
LexToken(PLUS,'+',3,14)
LexToken(MINUS,'-',3,16)
LexToken(NUMBER,20,3,18)
LexToken(TIMES,'*',3,20)
LexToken(NUMBER,2,3,21)
```

Output of form :
LexToken(tok.type, tok.value, tok.lineno, tok.lexpos)

Steps to make a Lexer

- Disclaimer : Note that if there are tokens t_A and t_B where

```
t_A = r'(' #accepts left brace
t_B = r'[()]+' #accepts left/right brace
```

If we write t_B over t_A, t_A is overshadowed. Kindly make note while declaring functions.

Regex

- A Regular Expression or RegEx is a special sequence of characters that uses a search pattern to find a string or set of strings.
- It can detect the presence or absence of a text by matching it with a particular pattern and also can split a pattern into one or more sub-patterns.
- These expressions are satisfied by Deterministic Finite Automata.
- It is declared in quotes (“” or ‘’) or in raw string format(r’’ or r””)
- Python has a built-in module named “re” that is used for regular expressions in Python.

Regex

- Regex consists of Metacharacters which have special meanings as shown

Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"planet$"
*	Zero or more occurrences	"he.*o"
+	One or more occurrences	"he.+o"
?	Zero or one occurrences	"he.?o"
{}	Exactly the specified number of occurrences	"he.{2}o"
|	Either or	"falls|stays"
()	Capture and group	

Regex

- Regex also consists of special sequences which consists of backslash (\) followed by one of the characters in the list below, and has a special meaning :

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"

Regex

- Regex also consists of sets to aggregate characters to be searched.

Set	Description
[arn]	Returns a match where one of the specified characters (`a` , `r` , or `n`) is present
[a-n]	Returns a match for any lower case character, alphabetically between `a` and `n`
[^arn]	Returns a match for any character EXCEPT `a` , `r` , and `n`
[0123]	Returns a match where any of the specified digits (`0` , `1` , `2` , or `3`) are present
[0-9]	Returns a match for any digit between `0` and `9`
[0-5][0-9]	Returns a match for any two-digit numbers from `00` and `59`
[a-zA-Z]	Returns a match for any character alphabetically between `a` and `z` , lower case OR upper case
[+]	In sets, `+` , `*` , `.` , `|` , `()` , `$` , `{}` has no special meaning, so `[+]` means: return a match for any `+` character in the string

Regex

- Using the metacharacters, special sequences and sets, we match patterns corresponding to our needs.
- An example of validating a sentence that starts with ‘The’ and ends with ‘Spain’. is as shown:

```
import re

#Check if the string starts with "The" and ends with "Spain":

txt = "The rain in South of Spain"
x = re.search(r"^The.*Spain$", txt)

if x:
  print("YES! We have a match!")
else:
  print("No match")
```

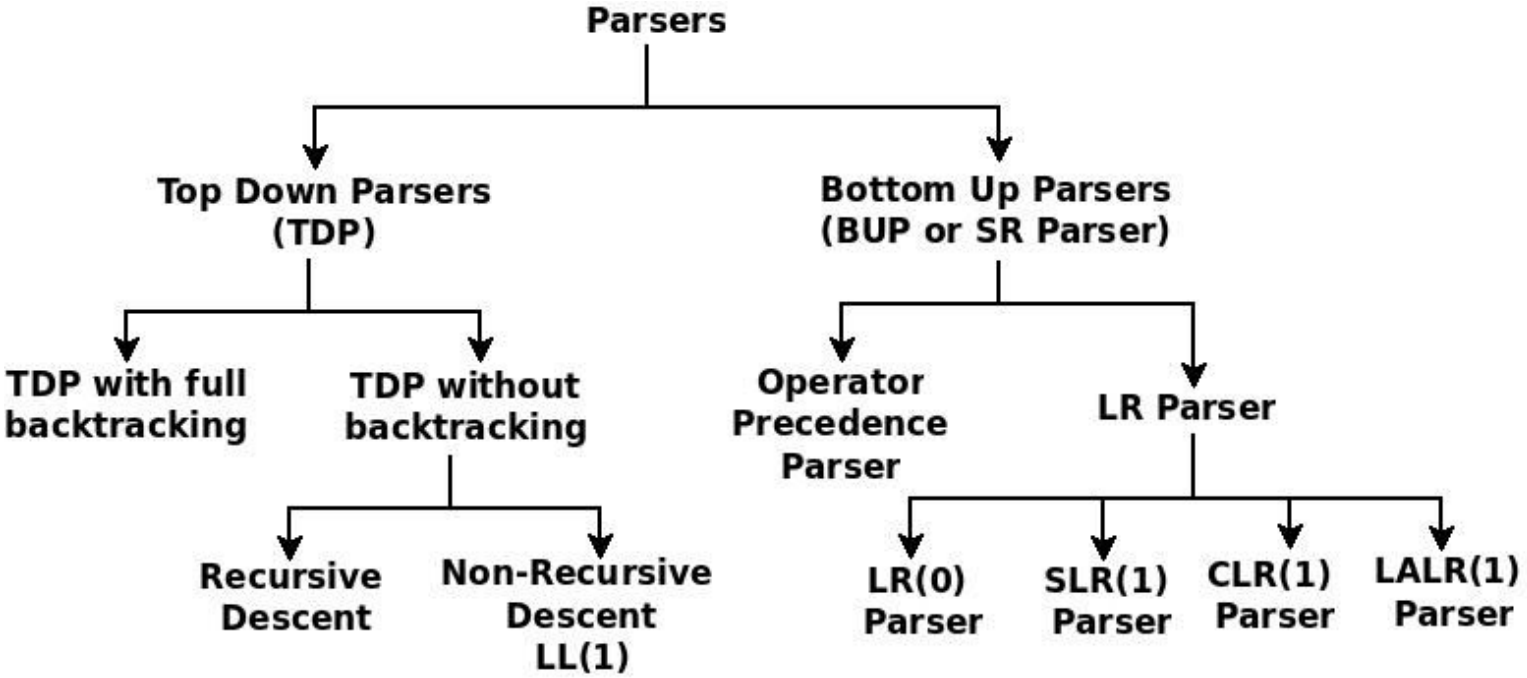
Regex

- For more info, refer to documentation.
- Since it is difficult to know all, a reference is provided.
  - Reference Link

Syntax Analyzer (Parser)

- Second phase of the compilation process, following lexical analysis.
- Its primary goal is to verify the syntactical correctness of the source code/document.
- It takes the tokens generated by the lexical analyzer and attempts to build a Parse Tree or Abstract Syntax Tree (AST), representing the program's structure.
- During this phase, the syntax analyzer checks whether the input string adheres to the grammatical rules of the language using context-free grammar. If the syntax is correct, the analyzer moves forward; otherwise, it reports an error.
- The parser accepts Deterministic Context Free Language (DCFLs).

Syntax Analyzer (Parser)



YACC

- Short for Yet Another Compiler Compiler created in 1970 for Unix Systems.
- Generates the LALR(1) parsers from formal grammar specifications.
- Plays an important role in compiler and interpreter development since it provides a means to specify the grammar of a language and to produce parsers that either interpret or compile code written in that language.
- In terms of PLY, yacc.py is used to parse language syntax.
- Before showing an example, there are a few important bits of background that must be mentioned.

LALR(1)

- Look Ahead Left to Right (LALR) parser.
- Most powerful parser after CLR parser which can handle large classes of grammar.
- Contains set of productions in form of states.
- In addition, consists of lookahead symbols.
- Input when fed to LALR parser, uses operations like shift (transfer a symbol from input to stack) or reduce (the symbols collected in stack is reduced to a non-terminal symbol) to process the string.
- Input is accepted if it reaches the final state/ accept state.
- Error hits if there is a conflict in operations while input is processed.

LALR(1)

- For the grammar with productions :

  S -> AA
  A -> aA
  A -> b

  Add a new production S’ -> S

  Now create the state table with productions as follows :

LALR(1)

$S' \rightarrow \cdot S, \$$ [initial Production]

$S \rightarrow \cdot AA, \$$ [from S']

$A \rightarrow \cdot aA, a|b$ [from S]

$A \rightarrow \cdot b, a|b$ [from S]

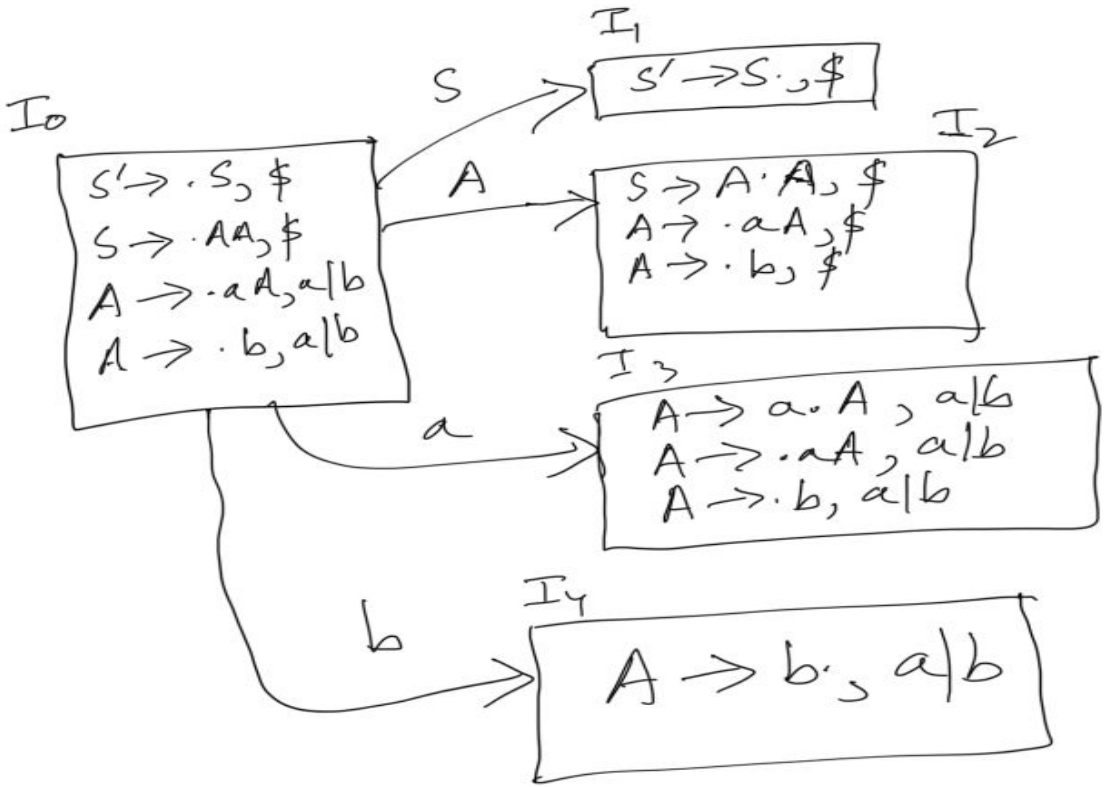
LALR(1)

- Initial production is S’ -> .S
- The ‘.’ is put to show at which state the parser has parsed.
- Initially no states are parsed.
- For the initial state, lookahead is always set to $.
- Hence the first production becomes S’ -> .S, $
- Then we see, the immediate rightmost symbol after ‘.’ is S, so include its productions also.
  - S -> .AA, $
- The lookahead of S is the symbol after S. Here it is nothing. Since there are no symbols, the lookahead of S’ is used as lookahead of S. Lookahead of symbol X is FOLLOW (X) where X != S’ and the terminal symbol which appears after X in a given production.

LALR(1)

- Now A is the next symbol after '.' in S->.AA,$
- Thus include A in initial state.
  - A -> .aA, a/b
  - A -> .b, a/b
- The lookahead is {a,b} since FOLLOW(A) is {a,b} (obtained if replacing 2nd A in S->.AA i.e. S -> .AaA and S -> .Ab)
- Now, we move onto next state with 1 symbol shift.

LALR(1)

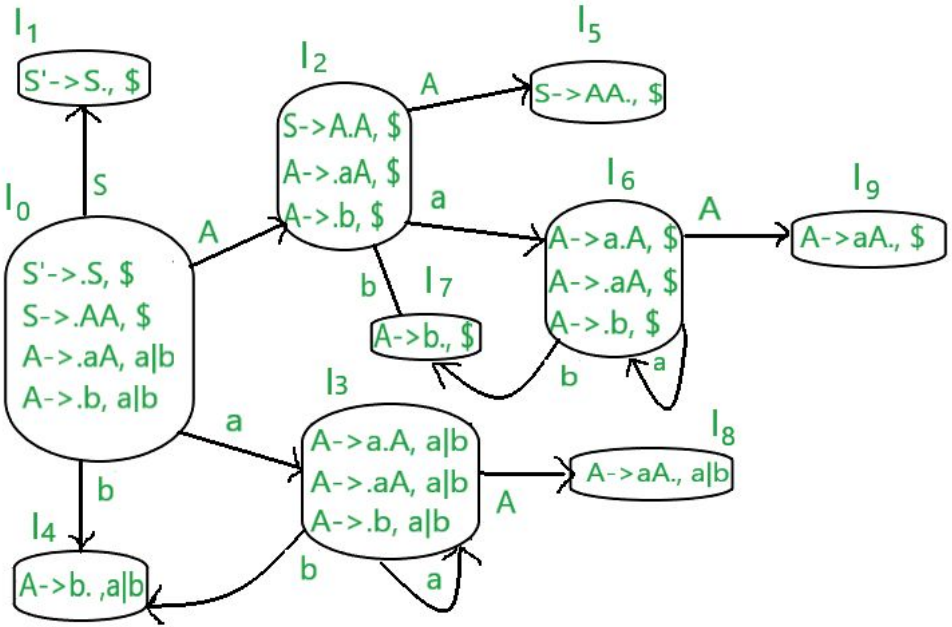

I0
S'→ · S, $
S → · AA, $
A → · aA, a|b
A → · b, a|b
S
I1
S' → S · , $
A
I2
S → A · A, $
A → · aA, $
A → · b, $
a
I3
A → a · A, a|b
A → · aA, a|b
A → · b, a|b
b
I4
A → b · , a|b

LALR(1)

- On shifting by a symbol, the '.' shifts right by 1 symbol and the lookahead symbols are the same lookahead symbols corresponding to the production generated from initial state.
- See states I_k (k=1,2,3,4) to understand.
- I_2 comes after shifting from symbol A.
- Only S-> .AA in the initial production possess the transition. Hence it appears as S->A.A,$.
- Since A is the immediate right symbol after '.', productions of A are also introduced.
- Since it comes from production with lookahead $, the productions of A here will have lookahead $ instead of {a,b}.

LALR(1)

- The final transition diagram looks like this :



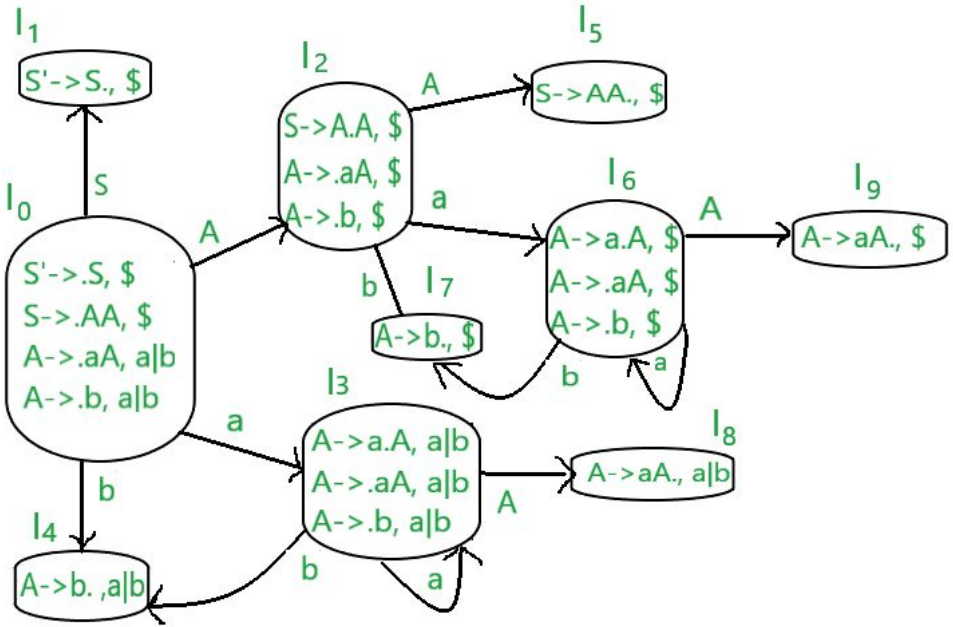
LALR(1)

- The tables are then formed from the state diagram which looks like the following (note ACTION is for terminal symbols, GOTO is for non - terminals) :

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
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		

LALR(1)

- If you observe the diagram, you will see $\{I_3, I_6\}$, $\{I_4, I_7\}$ and $\{I_8, I_9\}$ are pairs whose productions are same although the lookaheads may be different.
- Thus, merge such states.



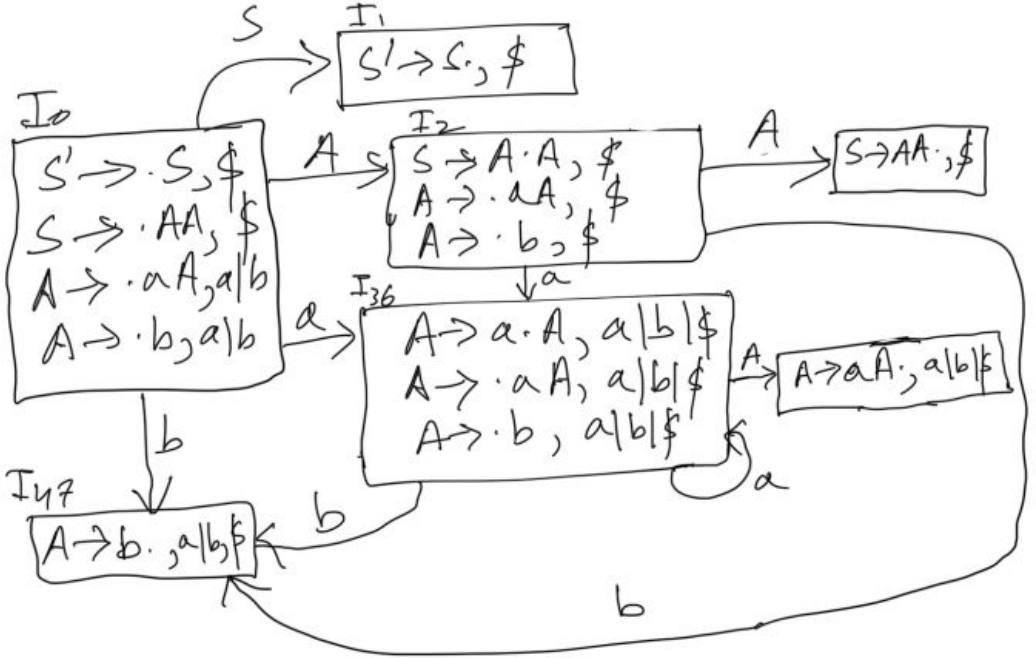
LALR(1)

- The merged tables are then formed from the state diagram which looks like :

	ACTION			GOTO	
	a	b	$	A	S
0	S36	S47		2	1
1			accept		
2	S36	S47		5	
36	S36	S47		89	
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

LALR(1)

- The following is the merged LALR state diagram:



LALR(1)

- To parse string ‘abab’, following is the steps to do

State	Input Buffer	Action
$	abab$	Shift a
$a	bab$	Shift b
$ab	ab$	Reduce A ->b
$aA	ab$	Reduce A -> aA
$A	ab$	Shift a
$Aa	b$	Shift b
$Aab	$	Reduce A -> b
$AaA	$	Reduce A -> aA
$AA	$	Reduce S -> AA
$S	$	Reduce S’ -> S
$	$	Accept

Parsing in YACC

- First, syntax is usually specified in terms of a grammar. For example, if you wanted to parse simple arithmetic expressions, you might first write an unambiguous grammar specification like this:

```
expression : expression + term
           | expression - term
           | term

term       : term * factor
           | term / factor
           | factor

factor     : NUMBER
           | ( expression )
```

YACC

- In the grammar, symbols such as NUMBER, +, -, *, and / are known as terminals and correspond to raw input tokens.
- Identifiers such as term and factor refer to grammar rules comprising a collection of terminals and other rules. These identifiers are known as non-terminals.
- The following shows the grammar and the actions which is intended to be performed.

YACC

Grammar	Action
expression0 : expression1 + term	expression0.val = expression1.val + term.val
| expression1 - term	expression0.val = expression1.val - term.val
| term	expression0.val = term.val
term0 : term1 * factor	term0.val = term1.val * factor.val
| term1 / factor	term0.val = term1.val / factor.val
| factor	term0.val = factor.val
factor : NUMBER	factor.val = int(NUMBER.lexval)
| (expression)	factor.val = expression.val

YACC

- The following example illustrates the LALR(1) steps that are performed if you wanted to parse the expression 3 + 5 * (10 - 20) using the grammar defined above. In the example, the special symbol $ represents the end of input.

YACC

Step	Symbol Stack	Input Tokens	Action
1		3 + 5 * (10 - 20)$	Shift 3
2	3	+ 5 * (10 - 20)$	Reduce factor : NUMBER
3	factor	+ 5 * (10 - 20)$	Reduce term : factor
4	term	+ 5 * (10 - 20)$	Reduce expr : term
5	expr	+ 5 * (10 - 20)$	Shift +
6	expr +	5 * (10 - 20)$	Shift 5
7	expr + 5	* (10 - 20)$	Reduce factor : NUMBER
8	expr + factor	* (10 - 20)$	Reduce term : factor
9	expr + term	* (10 - 20)$	Shift *
10	expr + term *	(10 - 20)$	Shift (
11	expr + term * (	10 - 20)$	Shift 10
12	expr + term * (10	- 20)$	Reduce factor : NUMBER
13	expr + term * (factor	- 20)$	Reduce term : factor
14	expr + term * (term	- 20)$	Reduce expr : term
15	expr + term * (expr	- 20)$	Shift -
16	expr + term * (expr -	20)$	Shift 20
17	expr + term * (expr - 20	)$	Reduce factor : NUMBER
18	expr + term * (expr - factor	)$	Reduce term : factor
19	expr + term * (expr - term	)$	Reduce expr : expr - term
20	expr + term * (expr	)$	Shift)
21	expr + term * (expr)	$	Reduce factor : (expr)
22	expr + term * factor	$	Reduce term : term * factor
23	expr + term	$	Reduce expr : expr + term
24	expr	$	Reduce expr
25		$	Success!

YACC

- To implement this part of parsing i.e. syntax analysis, one should write the functions accordingly as shown in the following snippet for example :

YACC Code Snippet

```
# Yacc example

import ply.yacc as yacc

# Get the token map from the lexer.  This is required.
from calclex import tokens

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]
```

CFG Rules in string

Store outputs of child terminals/non-terminals in a terminal

```
def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# Error rule for syntax errors
def p_error(p):
    print("Syntax error in input!")

# Build the parser
parser = yacc.yacc()

while True:
   try:
       s = raw_input('calc > ')
   except EOFError:
       break
   if not s: continue
   result = parser.parse(s)
   print(result)
```

Input

Steps to write for YACC

- The parsing rules are made in a function as shown (with p as parameter). Each function name should be of type p_<function name>. (<function name> in **lowercase**)
- The function doesn't return anything.
- The first line is the string (not raw string) or documented string (more than one production), second line is the accumulation of values in terminals and non terminals.

```
def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]
```

Steps to write for YACC

- The elements of p have the following basis :
  - p[0] is the value associated with the left side of production
  - p[1], p[2] … are the values associated with right side of production.

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

Steps to write for YACC

- Inbuilt function like p_error is present which is invoked in case of Syntax errors.

```
# Error rule for syntax errors
def p_error(p):
    print("Syntax error in input!")
```

Steps to write for YACC

- Note p[0] **must** be a **Non Terminal.**

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

How to view the Implemented Rules?

- There are files like parsetab.py and parser.out which are generated on executing the code.
- The rules made in PLY are processed and are written in the files. The LALR tables are created in parser.out and the execution code of PLY module is created in parsetab.py

Objective of Parsing

- Pattern matching and Syntax preservation.
- Calculator based evaluation.
- Web Scraping.
- Semantic Check of Natural Languages.

…and many more


INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR
भारतीय प्रौद्योगिकी संस्थान खड़गपुर
1951
योगः कर्मसु कौशलम्

Thanks!

State	Input Buffer	Action
$	abab$	Shift a
$a	bab$	Shift b
$ab	ab$	Reduce A ->b
$aA	ab$	Reduce A -> aA
$A	ab$	Shift a
$Aa	b$	Shift b
$Aab	$	Reduce A -> b
$AaA	$	Reduce A -> aA
$AA	$	Reduce S -> AA
$S	$	Reduce S’ -> S
$	$	Accept