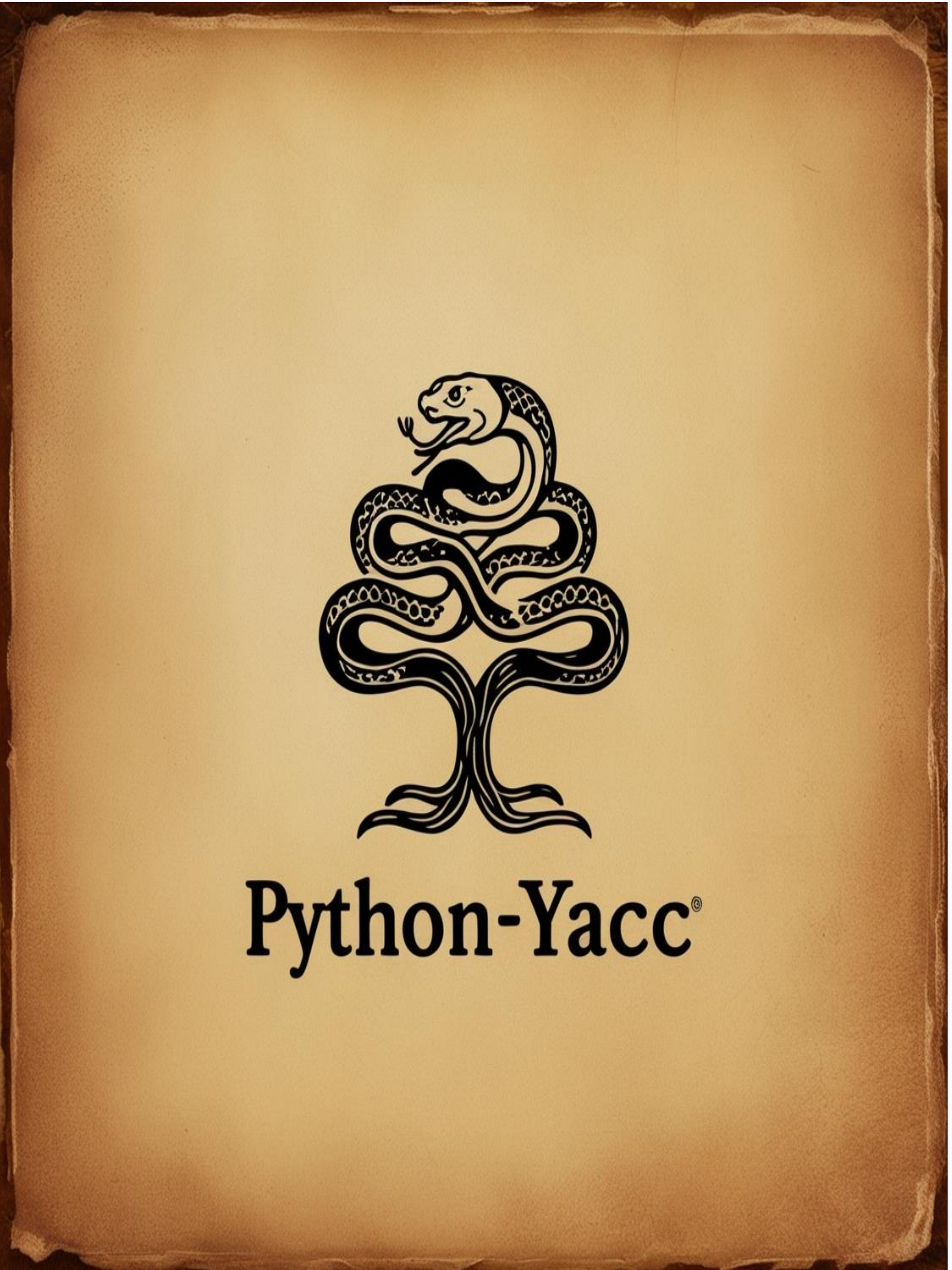


Python
Lex and Yacc

Prepared by

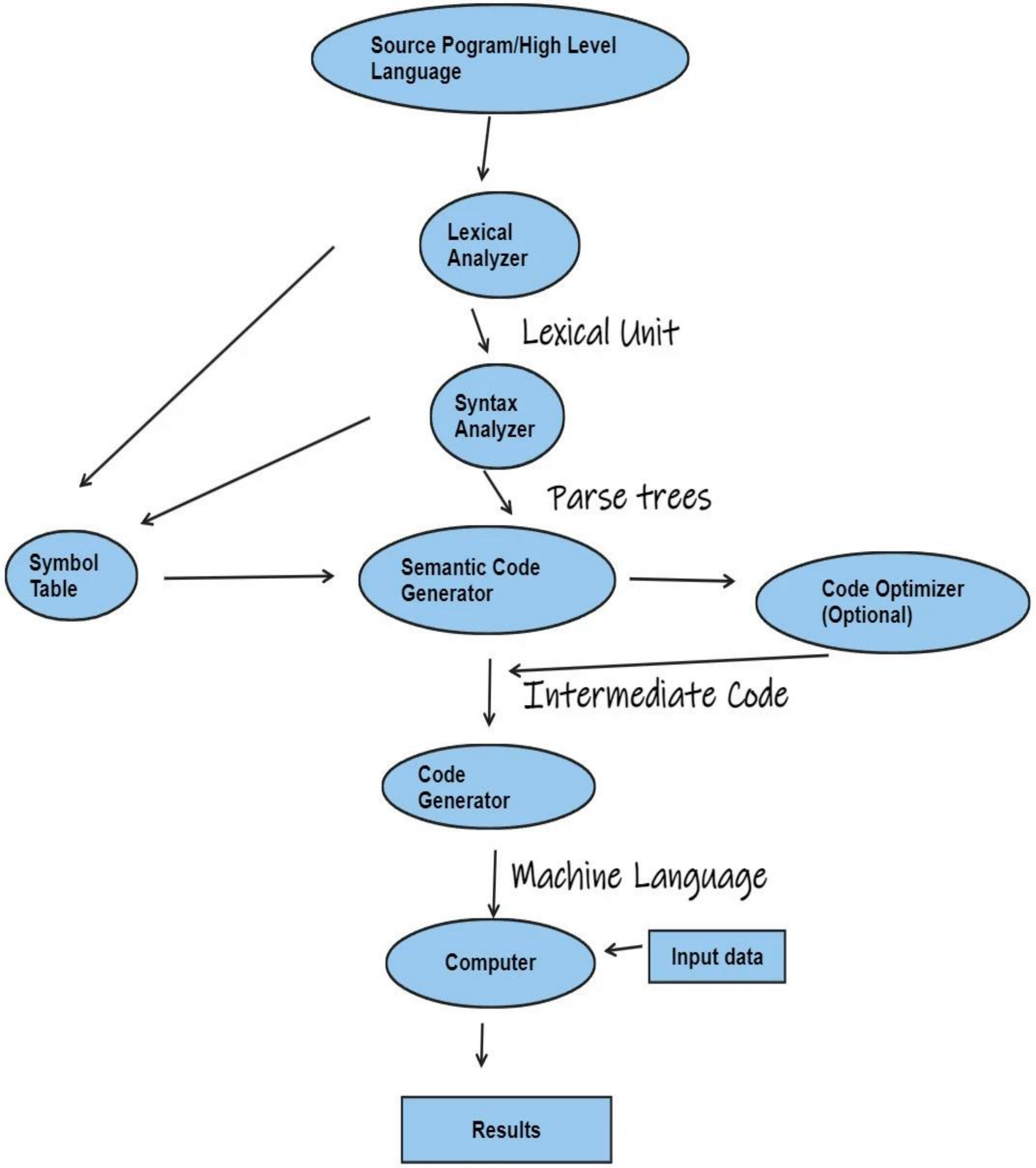
Zeyad moneer abo El-Zahab

Agenda

- ❑ Python Yacc
- ❑ General idea
- ❑ Rule function
- ❑ Combining Grammar Rule Functions
- ❑ Character Literals
- ❑ Empty productions
- ❑ Changing the Start Symbol
- ❑ Using an LR Parser
- ❑ Parser.out File
- ❑ Abstract Syntex Tree (AST)

Python YACC

- **Python YACC (Yet Another Compiler-Compiler)** is a powerful parser generator that allows you to create custom programming language parsers. It works in conjunction with the Python Lexical Analyzer (PLY) to provide a complete framework for building complex language parsers.



Python YACC (cont.)

- **Assumes you have defined a BNF grammar**
  - BNF : which stands for Backus-Naur Form, is a formal method that allows you to describe the structure and composition of sentences in a formal way, and is commonly used to describe the rules of syntax of programming languages.

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

term       : term * factor
           | term / factor
           | factor

factor     : NUMBER
           | ( expression )
```

```
# <expression> can be either a <term> or an expression with an addition or subtraction operation.
# <term> can be a multiplier or a divider by <factor>.
# <factor> can be an expression in parentheses or a number.
# <number> can be a string of numbers.
```

```
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!
```

Python YACC example

```python
import ply.yacc as yacc
import mylexer              # Import lexer information
tokens = mylexer.tokens     # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
    | expr MINUS term
    | term'''

def p_term(p):
    '''term : term TIMES factor
    | term DIVIDE factor
    | factor'''

def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()          # Build the parser
```

token information imported from lexer

grammar rules encoded as functions with names p_*rulename*

Note: Name doesn't matter as long as it starts with p_

Python YACC example

```python
import ply.yacc as yacc
import mylexer                # Import lexer information
tokens = mylexer.tokens       # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
    | expr MINUS term
    | term'''

def p_term(p):
    '''term : term TIMES factor
    | term DIVIDE factor
    | factor'''

def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()         # Build the parser
```

docstrings contain grammar rules from BNF

Builds the parser using introspection

General idea

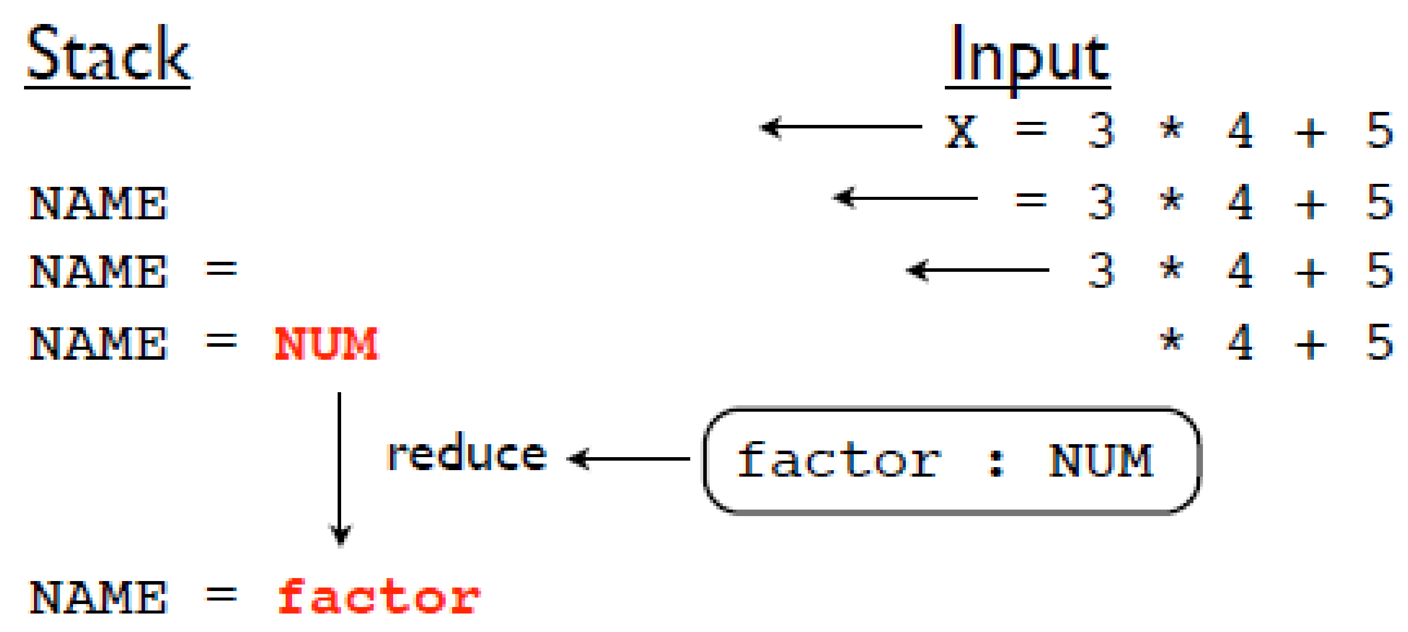
- Input tokens are shifted onto a parsing stack

Stack	Input
	← X = 3 * 4 + 5
NAME	← = 3 * 4 + 5
NAME =	← 3 * 4 + 5
NAME = NUM	* 4 + 5

- This continues until a complete grammar rule appears on the top of the stack

General idea

- If rules are found, a "reduction" occurs



- RHS of grammar rule replaced with LHS

Rule Functions

- During reduction, rule functions are invoked

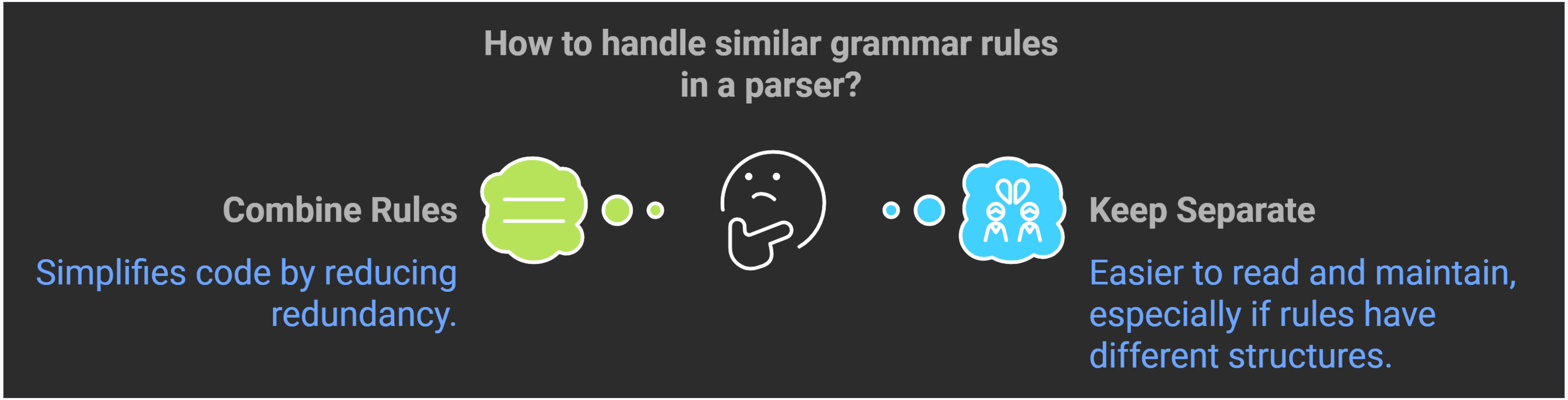
```
def p_factor(p):
    'factor : NUMBER'
```

- Parameter p contains grammar symbol values

```
def p_factor(p):
    'factor : NUMBER'

      p[0]       p[1]
```

Combining Grammar Rule Functions

- When grammar rules are similar, they can be combined into a single function.
- When combining grammar rules into a single function, it is usually a good idea for all of the rules to have a similar structure (e.g., the same number of terms).



Combining Grammar Rule Functions

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


def p_expression_minus(t):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]
```

Separate rules

```python
def p_binary_operators(p):
    '''expression : expression PLUS term
                  | expression MINUS term
       term       : term TIMES factor
                  | term DIVIDE factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

Combine rules

Character Literals

- If desired, a grammar may contain tokens defined as single character literals.
- A character literal must be enclosed in quotes such as '+'. In addition, if literals are used, they must be declared in the corresponding lex file through the use of a special literals declaration:
- Character literals are limited to a single character. Thus, it is not legal to specify literals such as '<=' or '=='. For this, use the normal lexing rules (e.g., define a rule such as t_EQ = r'==').

Character Literals

```python
def p_binary_operators(p):
    '''expression : expression PLUS term
                  | expression MINUS term
       term       : term TIMES factor
                  | term DIVIDE factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

```python
# Literals.  Should be placed in module given to lex()
literals = ['+','-','*','/' ]
```

In LEX file

Empty Productions

- An empty production allows a rule to produce 'nothing'.
- Useful for making parts of the grammar optional.
- Makes rules clearer and easier to understand.
- Improves readability by clearly expressing intentions.

➢ Example syntax:

```python
def p_empty(p):
    'empty :'
    pass
```

```python
def p_item_list(p):
    '''item_list : item_list item
                 | empty'''
```

Changing the Start Symbol

- By default, Yacc uses the first defined grammar rule as the start symbol.
- The start symbol is the entry point for parsing the input text.
- How to Change the Start Symbol
    - Use the 'start' specifier in your file to set a different start symbol
    - Example : start = 'foo'
    - This makes 'foo' the starting grammar rule, even if it is not the first rule defined.
- The use of a start specifier may be useful during debugging since you can use it to test specific parts of the grammar.

Changing the Start Symbol

```python
#----- start symbol ----------------------

start = 'foo'

def p_bar(p):
    'bar : A B'

# This is the starting rule due to the start specifier above
def p_foo(p):
    'foo : bar X'
...

parser = yacc.yacc(start='foo')
```

Using an LR Parser

- LR (Left-to-right, Rightmost derivation) parsers use tables to decide how to process input tokens.
- Rule functions generally process values on right hand side of grammar rule, Result is then stored in left hand side
- Results propagate up through the grammar
- Bottom-up parsing

Parser.out File

- A debugging file generated by yacc.py to assist in resolving conflicts in LR parsing.
- Command to Generate: yacc.yacc (debug=1)
- Purpose: Helps track down shift/reduce and reduce/reduce conflicts.
- Structure: Information about unused terminals, grammar rules, states, and possible actions.

Parser.out File

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

# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
```

LEX file

```python
yacc.yacc(debug=1)
```

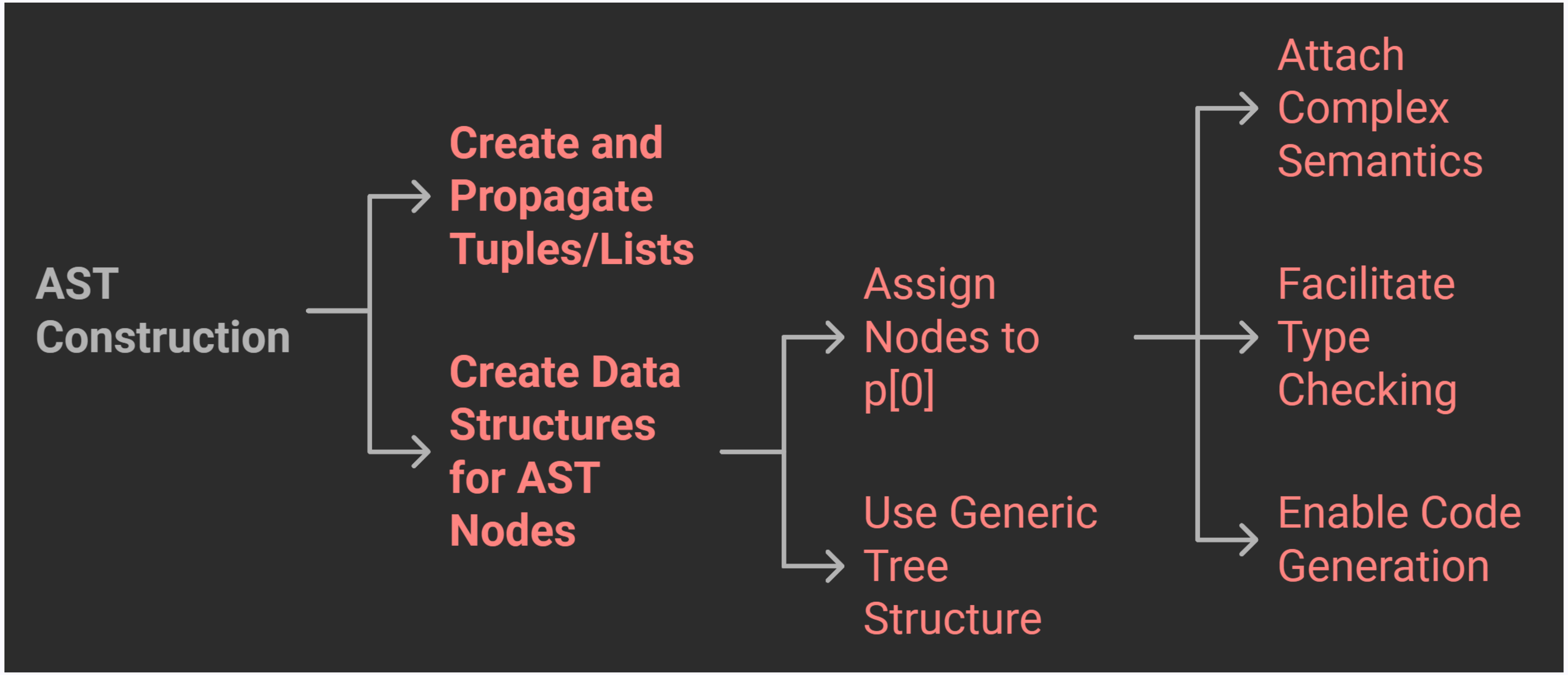
```
Grammar

Rule 0     S' -> assign
Rule 1     assign -> NAME EQUALS expr
Rule 2     expr -> expr PLUS term
Rule 3     expr -> expr MINUS term
Rule 4     expr -> term
Rule 5     term -> term TIMES factor
Rule 6     term -> term DIVIDE factor
Rule 7     term -> factor
Rule 8     factor -> NUMBER

Terminals, with rules where they appear

DIVIDE               : 6
MINUS                : 3
NUMBER               : 8
PLUS                 : 2
TIMES                : 5
error                :

Nonterminals, with rules where they appear

EQUALS               : 1
NAME                 : 1
assign               : 0
expr                 : 1 2 3
factor               : 5 6 7
term                 : 2 3 4 5 6
```

Parser.out file

Abstract Syntax Tree (AST) Construction

- An Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code.
- Each node in the tree represents a construct occurring in the code.
- The AST can be used for further analysis, optimization, or code generation, making it a powerful tool for compiler and interpreter development

Abstract Syntax Tree (AST) Construction

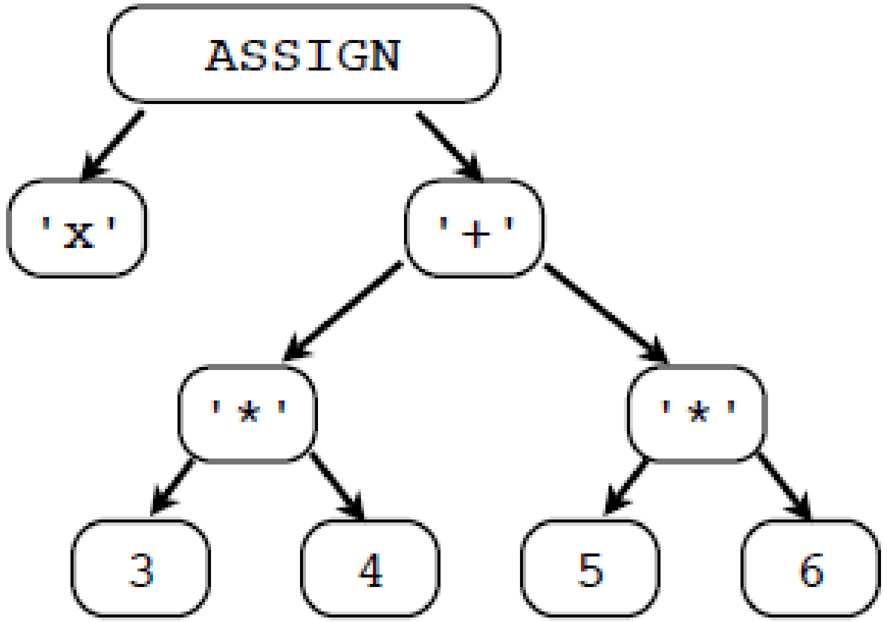


Abstract Syntax Tree (AST) Construction Example

```
def p_assign(p):
    '''assign : NAME EQUALS expr'''
    p[0] = ('ASSIGN',p[1],p[3])

def p_expr_plus(p):
    '''expr : expr PLUS term'''
    p[0] = ('+',p[1],p[3])

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

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

def p_factor(p):
    '''factor : NUMBER'''
    p[0] = ('NUM',p[1])
```

Abstract Syntax Tree (AST) Construction Example

```
>>> t = yacc.parse("x = 3*4 + 5*6")
>>> t
('ASSIGN','x',('+',
                  ('*',('NUM',3),('NUM',4)),
                  ('*',('NUM',5),('NUM',6))
               )
)
>>>
```



Thank you

Any questions?