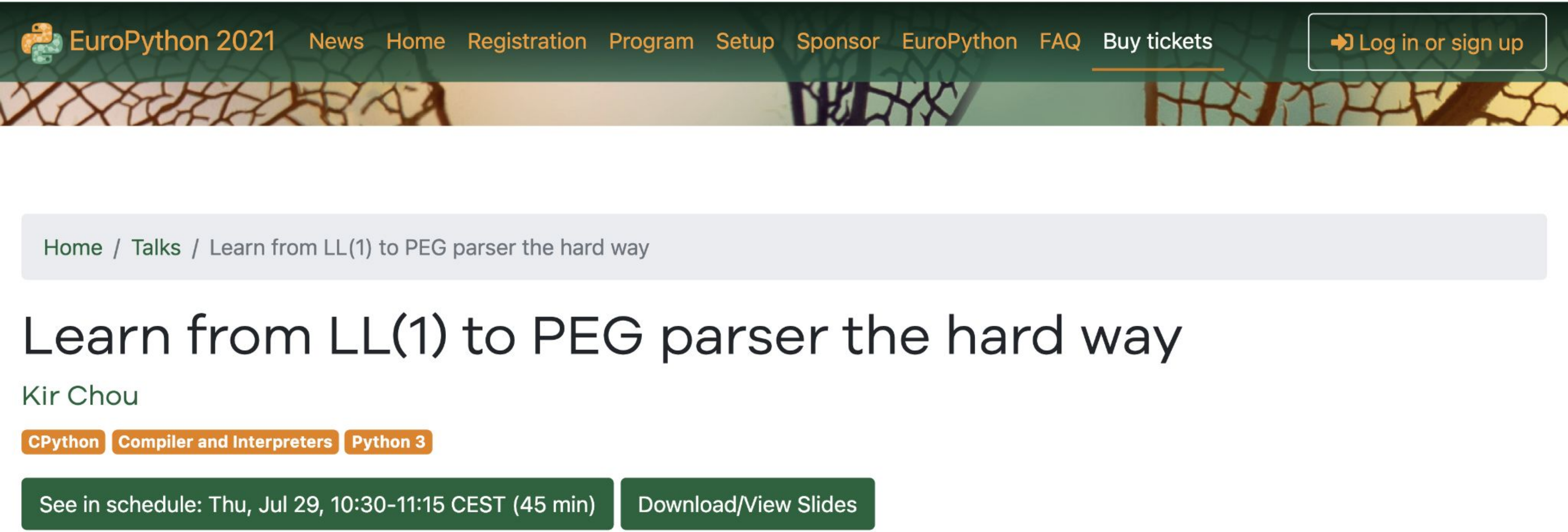
Standing on the shoulders of giants:

Learn from LL(1) to PEG parser the hard way

Kir Chou @ PyCon TW 2021



https://ep2021.europython.eu/talks/98ecgon-learn-from-ll1-to-peg-parser-the-hard-way

Standing on the shoulders of giants:

Learn from LL(1) to PEG parser the hard way

Kir Chou @ PyCon TW 2021

https://www.youtube.com/watch?v=DZTLgVBxET4

Standing on the shoulders of giants:

Learn from LL(1) to PEG parser the hard way

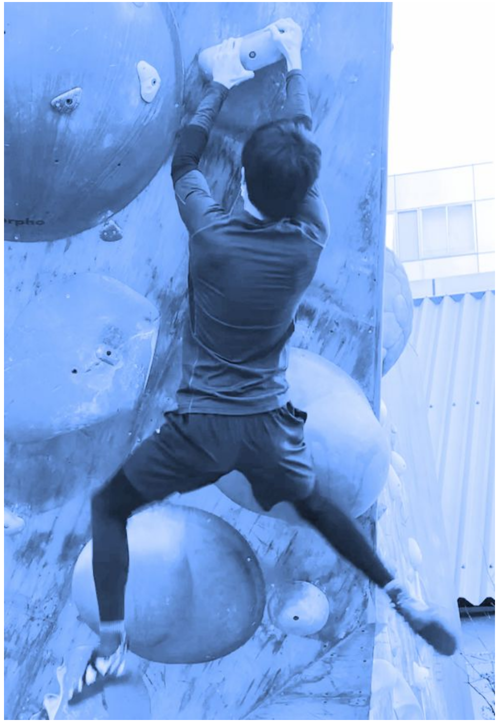
Kir Chou @ PyCon TW 2021

About me

Presented at PyCon since 2017

https://note35.github.io/about/

https://github.com/note35/Parser-Learning



Agenda

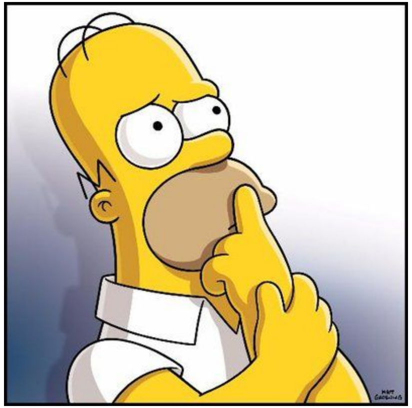
- Motivation
- What is parser in CPython?
- Parser 101 - CFG
- Parser 101 - Traditional parser (LL(1) / LR(0))
- Parser 102 - PEG and PEG parser
- Parser 102 - Packrat parser
- CPython’s PEG parser
- Take away

Motivation

Motivation

What's New In Python 3.9?

PEP 617, CPython now uses a new parser based on PEG;



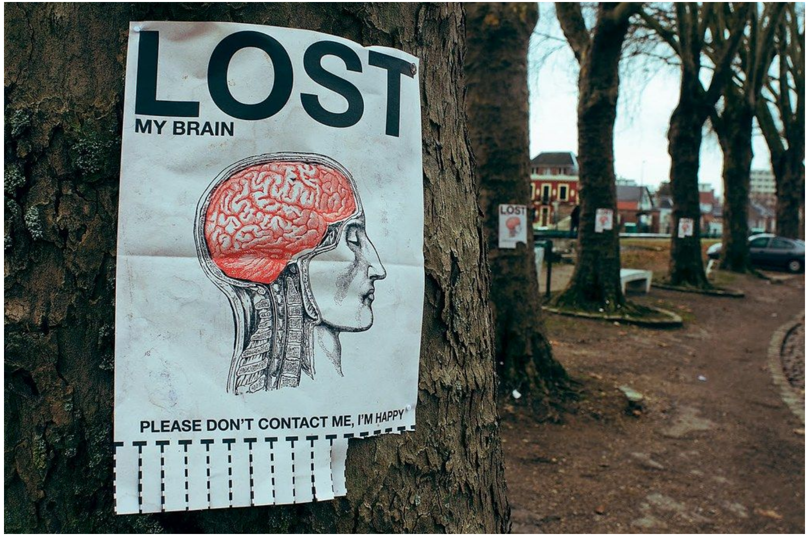
"IIRC, I took a Compiler class in school…"

Motivation (Cont.)

School taught us the brief concept of the Compiler’s frontend and backend.

School’s parser assignment used **Bison + YACC**.

And...



My motivation = Talk objectives

What is PEG parser?

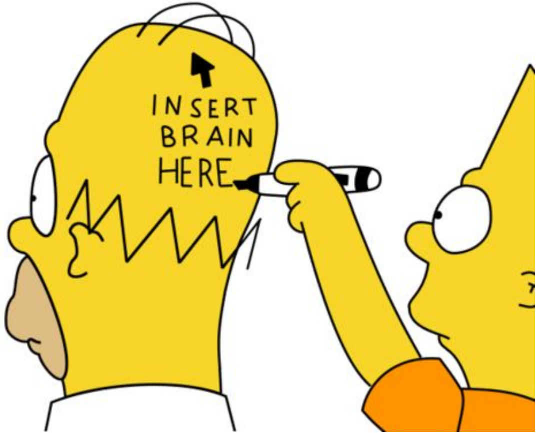
Why did python use LL(1) parser before?

Why did Guido choose PEG parser?

What other parsers do we have?

What’s the difference between those parsers?

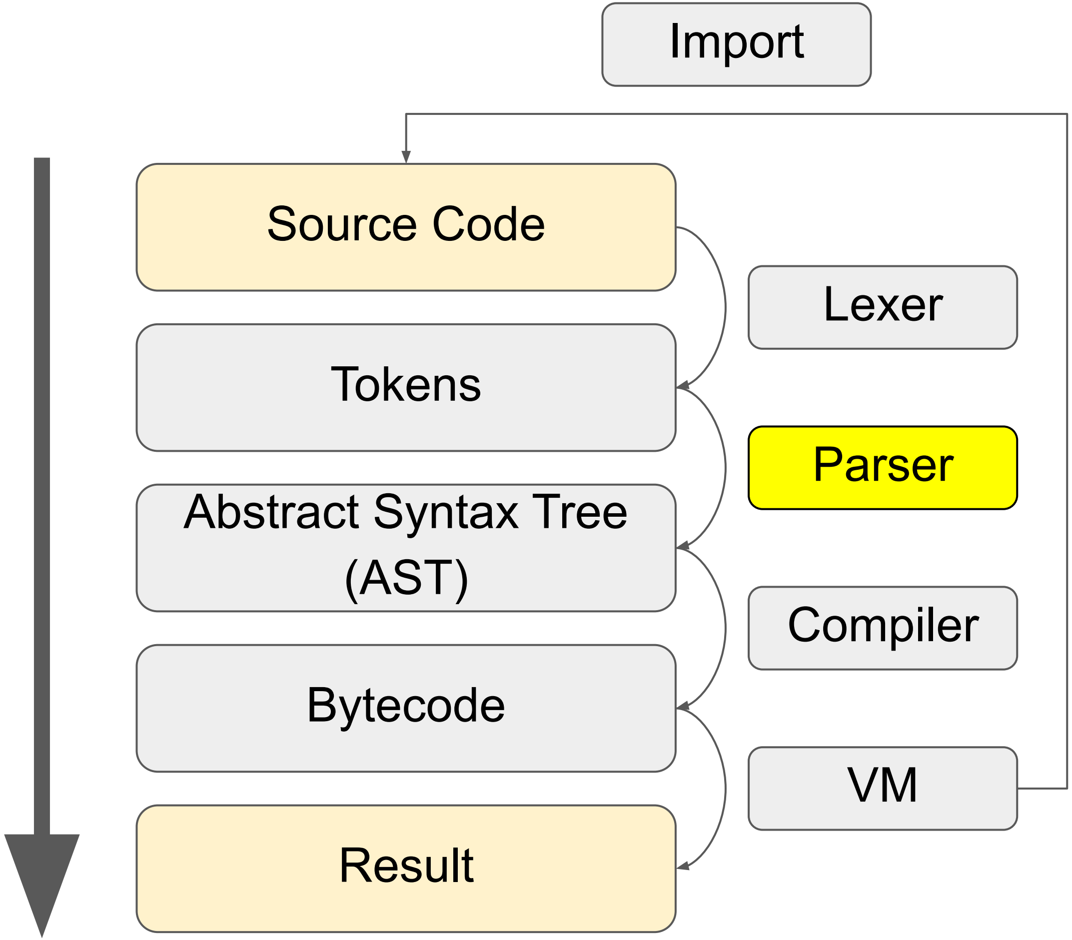
How to implement those parsers?



What is parser in CPython?

[CPython DevGuide - Design of CPython’s Compiler]

Compilation Steps



Lexer

https://docs.python.org/3/library/tokenize.html#examples

```
[1]: from io import BytesIO
     from tokenize import tokenize
```

```
[2]: example_tokens = tokenize(BytesIO('print(1 + 2 * 3)'.encode('utf-8')).readline
```

```
[3]: list(example_tokens)
```

```
[3]: [TokenInfo(type=57 (ENCODING), string='utf-8', start=(0, 0), end=(0, 0), line=''),
     TokenInfo(type=1 (NAME), string='print', start=(1, 0), end=(1, 5), line='print(1 + 2 * 3)'),
     TokenInfo(type=53 (OP), string='(', start=(1, 5), end=(1, 6), line='print(1 + 2 * 3)'),
     TokenInfo(type=2 (NUMBER), string='1', start=(1, 6), end=(1, 7), line='print(1 + 2 * 3)'),
     TokenInfo(type=53 (OP), string='+', start=(1, 8), end=(1, 9), line='print(1 + 2 * 3)'),
     TokenInfo(type=2 (NUMBER), string='2', start=(1, 10), end=(1, 11), line='print(1 + 2 * 3)'),
     TokenInfo(type=53 (OP), string='*', start=(1, 12), end=(1, 13), line='print(1 + 2 * 3)'),
     TokenInfo(type=2 (NUMBER), string='3', start=(1, 14), end=(1, 15), line='print(1 + 2 * 3)'),
     TokenInfo(type=53 (OP), string=')', start=(1, 15), end=(1, 16), line='print(1 + 2 * 3)'),
     TokenInfo(type=4 (NEWLINE), string='', start=(1, 16), end=(1, 17), line=''),
     TokenInfo(type=0 (ENDMARKER), string='', start=(2, 0), end=(2, 0), line='')]
```

Parser

https://docs.python.org/3/library/ast.html

```
[1]: import ast

[2]: example_ast = ast.parse('print(2 + 3 * 4)')

[3]: print(ast.dump(example_ast))

Module(body=[Expr(value=Call(func=Name(id='print',
ctx=Load()), args=[BinOp(left=Num(n=2), op=Add(),
right=BinOp(left=Num(n=3), op=Mult(), right=Num(n=
4)))], keywords=[]))])
```

Compiler

https://docs.python.org/3/library/dis.html#dis.disassemble

```
[4]: example_bytecode = compile(example_ast, '<string>', 'exec')
```

```
[5]: eval(example_bytecode)
```

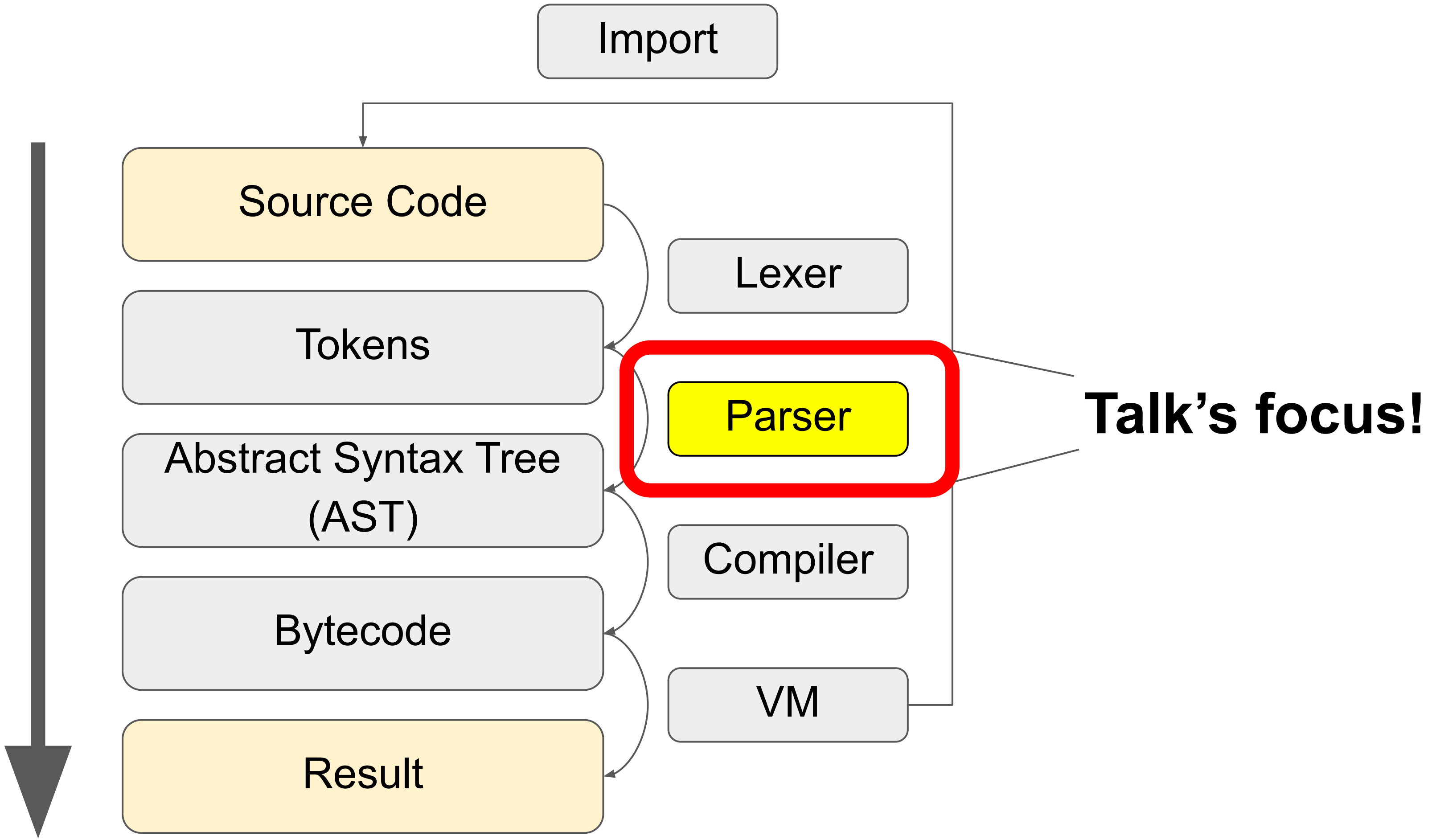
```
14
```

= print(2*3+4)

```
[6]: import dis
```

```
[7]: dis.disassemble(example_bytecode)
```

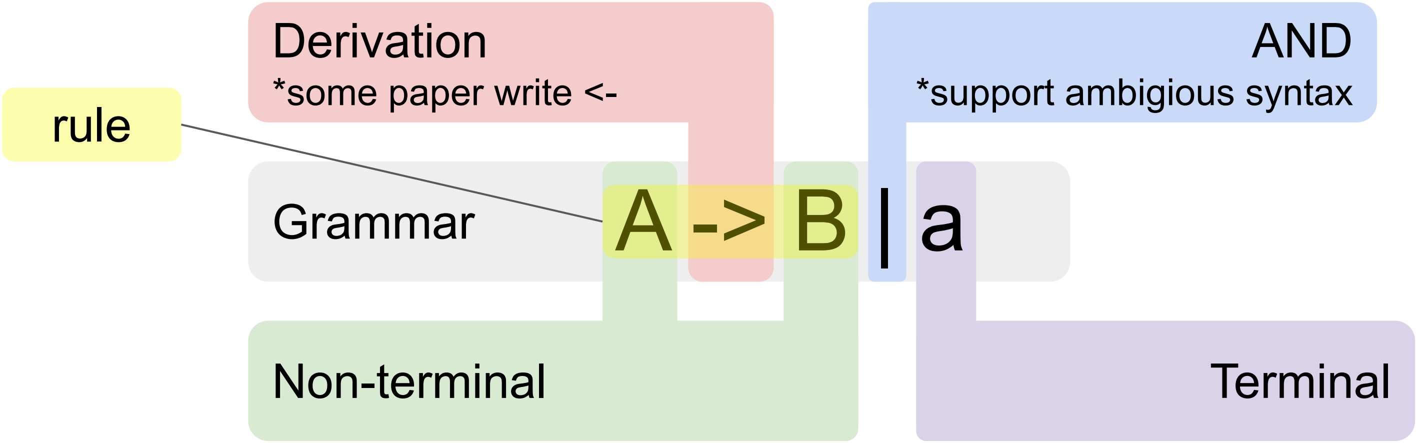
```
  1           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 (14)
              4 CALL_FUNCTION            1
              6 POP_TOP
              8 LOAD_CONST               1 (None)
             10 RETURN_VALUE
```


Import
Source Code
Lexer
Tokens
Parser
Abstract Syntax Tree (AST)
Compiler
Bytecode
VM
Result
Talk's focus!

Parser 101 - CFG

[Uncode - GATE Computer Science - Compiler Design Lecture]

Context Free Grammar (CFG)



Interpretation of this Grammar

“Both B and a can be derived from A”

What is “Context Free”?

Left-hand side in all the rules only contains 1 **non-terminal**.

Valid CFG Example:

```
S -> aSb
```

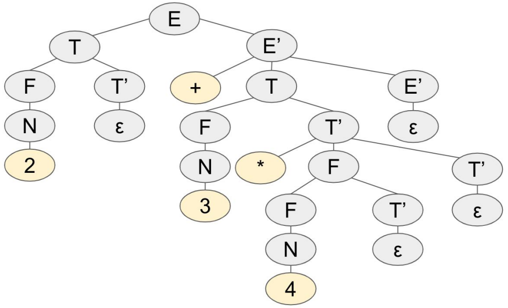
Invalid CFG Example:

```
xSy -> axSyb
```

Semantic Analysis: Parse Tree

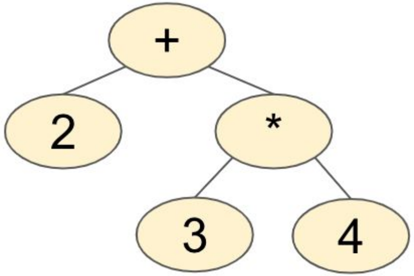
Concret Syntax Tree (CST)

An ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.



Abstract Syntax Tree (AST)

A tree representation of the abstract syntactic structure of source code written in a programming language.



CFG Simplification

1. Ambiguous -> Unambiguous

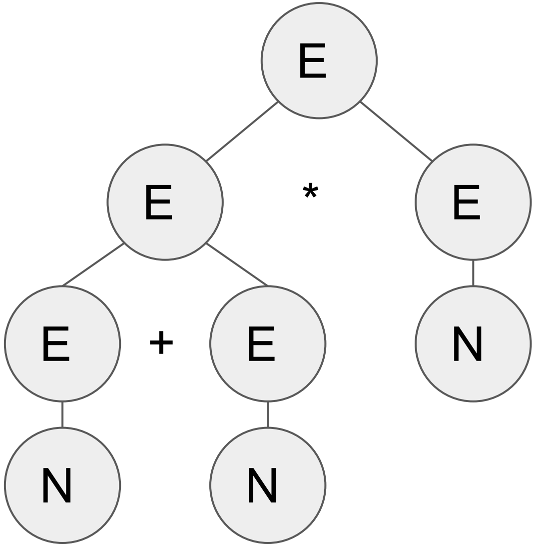
2. Nondeterministic -> Deterministic

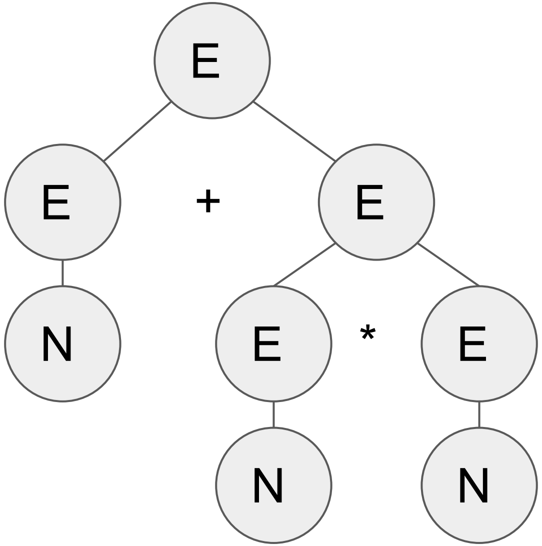
3. Left recursion -> No left recursion

Ambiguious Definition

A grammar contains rules that can generate more than one tree.

E -> E + E | E * E | Num





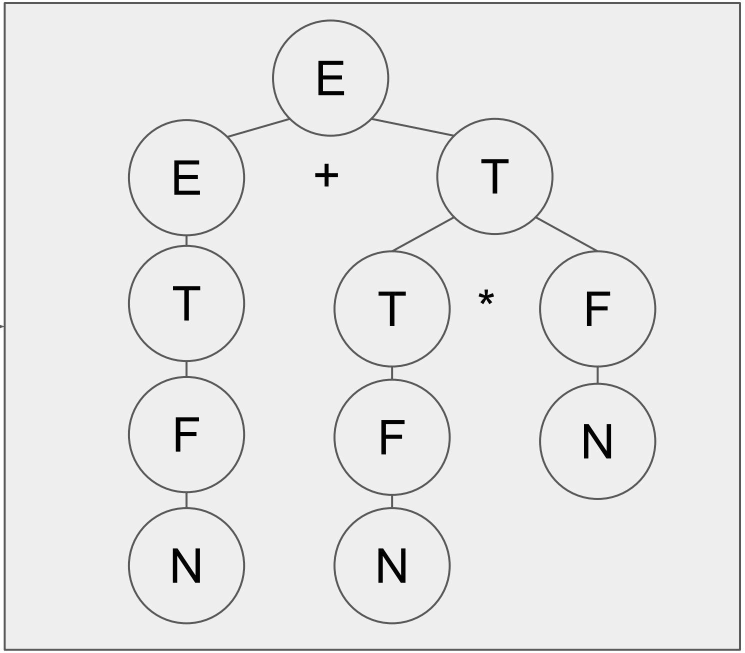
Ambiguious -> Unambiguous

E -> E + E | E * E | Num

Step1
Rewrite Grammar

E -> E + T | T
T -> T * F | F
F -> Num

Step2
Make sure the grammar only generate one tree



Non-deterministic -> Deterministic

A grammar contains rules that have common prefix.

A -> ab | ac

Rewrite Grammar

A -> aA'
A' -> b | c

**A non-deterministic grammar can be rewritten into more than one deterministic grammar.*

Left recursion -> No left recursion

A grammar contains direct or indirect left recursion.

E in first E + T will recursively derives to second E + T,
E in second E + T will repeat it to third E + T,
and so on recursively.

E -> E + T | T
T -> T * F | F
F -> Num

Rewrite Grammar

E -> TE’
E’ -> +TE’ | None
T -> FT’
T’ -> *FT’ | None
F -> Num

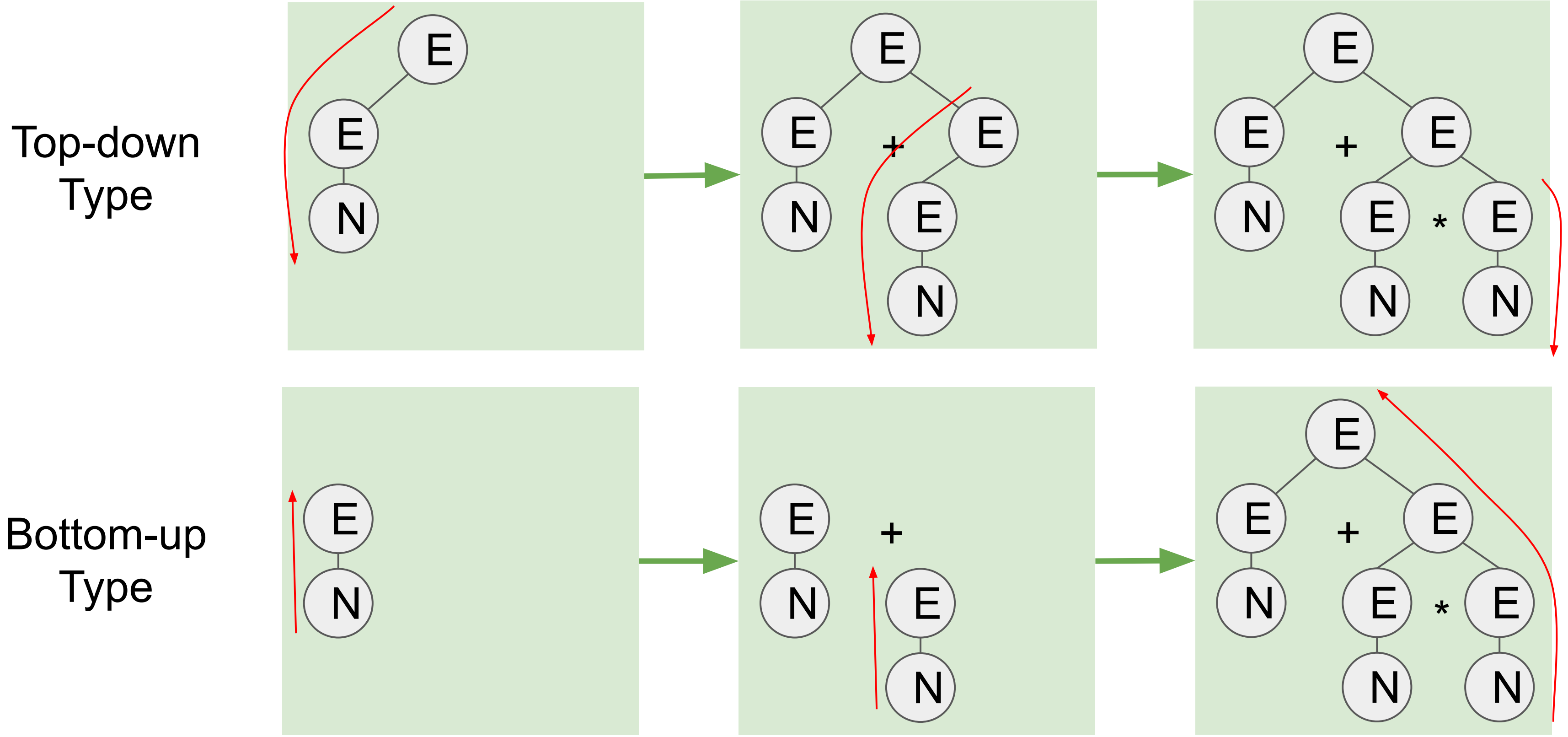
Recap: CFG Simplification

	Before	After
Ambiguous	E -> E + E | E * E | Num	E -> E + T | T T -> T * F | F F -> Num
Non-deterministic	A -> ab | ac	A -> aA' A' -> b | c
Left Recursion	E -> E + T | T T -> T * F | F F -> Num	E -> TE' E' -> +TE' | None T -> FT' T' -> *FT' | None F -> Num

Parser 101 - Traditional parser

[Uncode - GATE Computer Science - Compiler Design Lecture]

Parser classification



LL / LR Parser

LL(k) = Left-to-right,

LR(k) = Left-to-right,

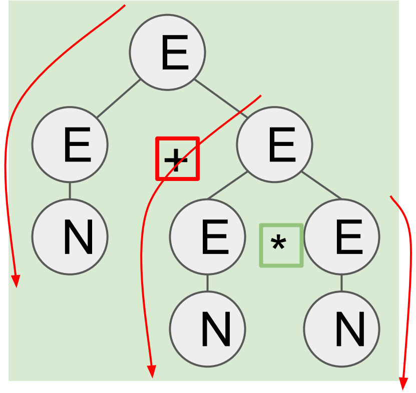
Input String: 2 + 3 * 4

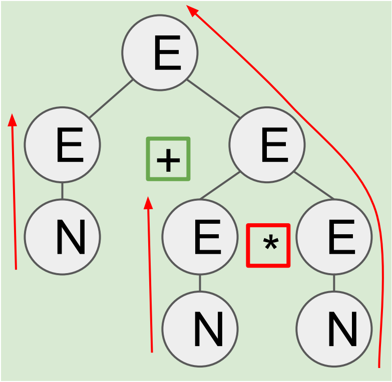
**Both LL/LR parser scan input string from left to right*

LL / LR Parser

LL(k) = Left-to-right, Leftmost derivation,

LR(k) = Left-to-right, Rightmost derivation,





$+ \longrightarrow *$

**The derivation time of LL/LR parser is different.*

$* \longrightarrow +$

LL / LR Parser

LL(k) = Left-to-right, Leftmost derivation, k-token lookahead (k>0)

LR(k) = Left-to-right, Rightmost derivation, k-token lookahead (k>=0)

Input String: 2 + 3 * 4

I am "a token of number".
*If I perform **1-token** lookahead and*
meet "a token of +",
what to do next?

Top Down - Recursive descent parser

LL(k) - Implementation

Step2
**parse from left to right*

E -> TE’
E’ -> +TE’ | None
T -> FT’
T’ -> *FT’ | None
F -> Num

Step1
write function for each non-terminal

2 + 3 * 4

**perform k-lookahead*

parse_Tp(parse_F())

parse_Ep(parse_T())

parse_E()

Step3
**recursively parse the input string started from first rule parse_E()*

LL(1) - Example code

Grammar

E -> TE’

E’ -> +TE’ | None

T -> FT’

T’ -> *FT’ | None

F -> Num

```python
def parse_Ep(self, lval: int) -> int:
    cur_symb = self.E[self.idx]
    if cur_symb == '+':
        self.idx += 1
        return self.parse_Ep(lval + self.parse_T())
    else:
        return lval

def parse_T(self) -> int:
    return self.parse_Tp(self.parse_F())
```

**perform 1-lookahead*

Derivation

Top Down - Non recursive descent parser

LL(1) - Parsing table

Grammar	First	Follow
E -> TE'	NUM	$
E'-> None / +TE'	+, None	$
T -> FT'	NUM	+, $
T'-> None / *FT'	*, None	+, $
F -> NUM	NUM	+, *, $

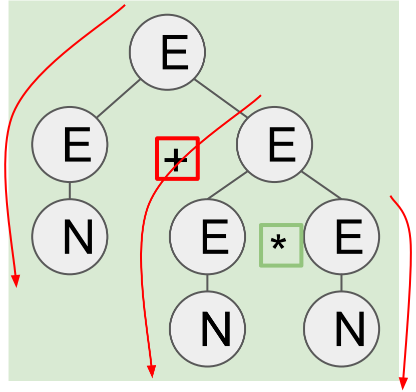
∅	NUM	+	*	$
E	TE'			
E'		+TE'		None
T	FT'			
T'		None	*FT'	None
F	NUM			

Step1
*Build **first/follow table** for each non-terminal*
Note: $ means endmark

Step2
*Build **parsing table** based on **first/follow table***

LL(1) - Implementation

```
STACK                | INPUT          | ACTION
------------------------------------------------------------
[$, E]               | 2 + 3 * 4 $ | Reduce: E -> TE'
[$, E', T]           | 2 + 3 * 4 $ | Reduce: T -> FT'
[$, E', T', F]       | 2 + 3 * 4 $ | Reduce: F -> N
[$, E', T', N]       | 2 + 3 * 4 $ | Shift
[$, E', T']          |   + 3 * 4 $ | Reduce: T' -> None
[$, E']              |   + 3 * 4 $ | Reduce: E' -> +TE'
[$, E', T, +]        |   + 3 * 4 $ | Shift
[$, E', T]           |     3 * 4 $ | Reduce: T -> FT'
[$, E', T', F]       |     3 * 4 $ | Reduce: F -> N
[$, E', T', N]       |     3 * 4 $ | Shift
[$, E', T']          |       * 4 $ | Reduce: T' -> *FT'
[$, E', T', F, *]    |       * 4 $ | Shift
[$, E', T', F]       |         4 $ | Reduce: F -> N
[$, E', T', N]       |         4 $ | Shift
[$, E', T']          |           $ | Reduce: T' -> None
[$, E']              |           $ | Reduce: E' -> None
[$]                  |           $ | Acc
```



Step3
Implement with stack (take shift/reduce action based on ***parsing table****)*

LL(1) - Example code

Grammar

E -> TE’

E’ -> +TE’ | None

T -> FT’

T’ -> *FT’ | None

F -> Num

```
parsing_table = {
  'E': {'N': ['T', 'Ep']},
  'Ep': {'+': ['+', 'T', 'Ep'],
         '-': ['-', 'T', 'Ep'], '$': []},
  'T': {'N': ['F', 'Tp']},
  'Tp': {'+': [], '-': [],
         '*': ['*', 'F', 'Tp'],
         '/': ['/', 'F', 'Tp'], '$': []},
  'F': {'N': ['N']}
}
```

```
while self.stack[-1] != '$':
    cur_node = self.stack.pop()
    cur_non_term = cur_node.key
    cur_token = self.E[self.idx]
    if (cur_non_term == 'N' and type(cur_token) == int) or cur_non_term == cur_token:
        # terminal: int or +-*/
        self.idx += 1
        cur_node.val = cur_token
    else:
        # non-terminal: E, Ep, T, Tp, F
        new_symbs = parsing_table[cur_non_term][to_key(cur_token)].copy()
        if len(new_symbs) == 0:
            cur_node.children.append(Node('None', None))
        else:
            while new_symbs:
                new_symb = new_symbs.pop()
                new_node = Node(new_symb, parent=cur_node)
                cur_node.children.append(new_node)
                self.stack.append(new_node)
return self.tree_root
```

Non-terminal stack

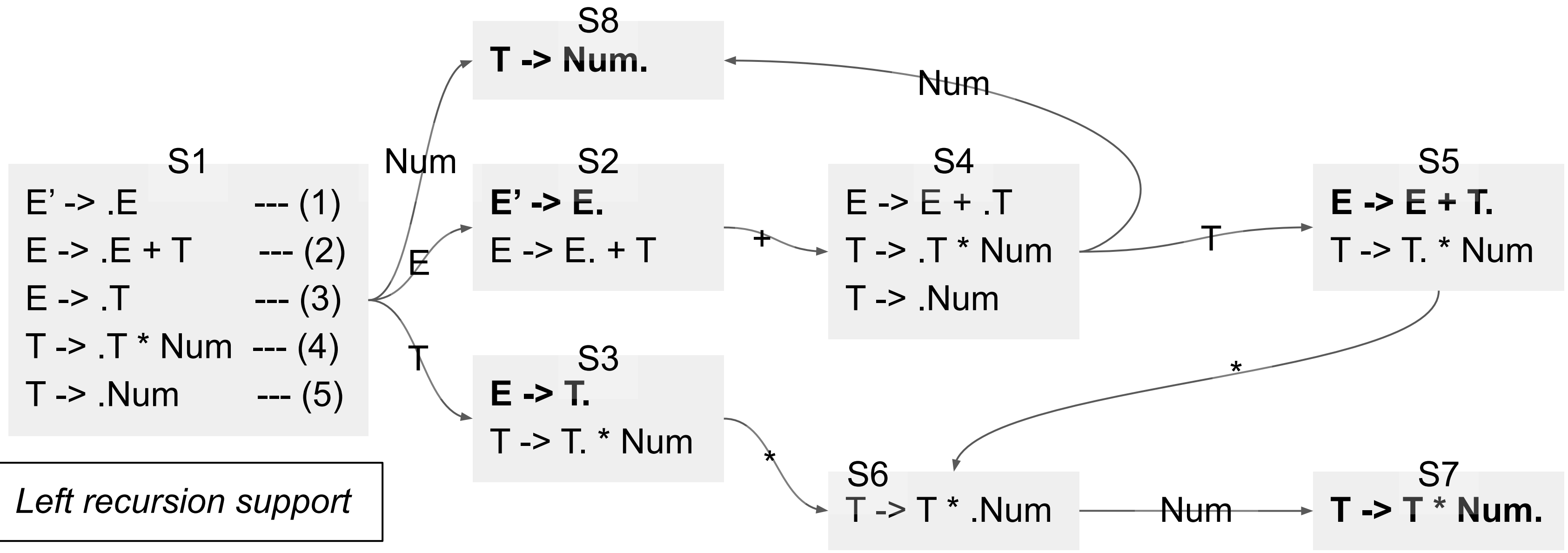
Shift

Reduce (Derivation)

Reduce (Derivation)

Bottom Up - LR(0) parser

LR(0) - Deterministic finite automaton



Left recursion support

Step1
Build Deterministic Finite Automaton(DFA)

LR(0) - Parsing table

Step2
*Build **parsing table***
(For parser like SLR(1), it requires first/follow table)

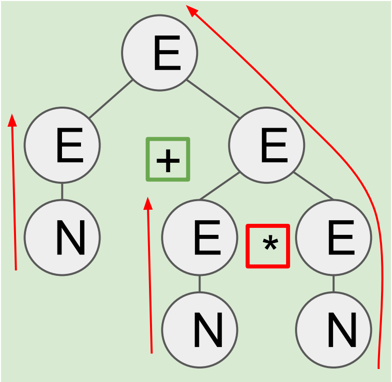
STATE	ACTION				GOTO	
	N	+	*	$	E	T
1	s8				2	3
2	r1	s4	r1	acc		
3	r3	r3	s6	r3		
4	s8					5
5	r2	r2	s6	r2		
6	s7					
7	r4	r4	r4	r4		
8	r5	r5	r5	r5		

Shift

Reduce (Derivation)

LR(0) - Implementation

STACK	SYMBOLS	INPUT	ACTION
[1]		2 + 3 * 4 $	shift
[1, 8]	Num	+ 3 * 4 $	reduce by T -> Num
[1, 3]	T	+ 3 * 4 $	reduce by E -> T
[1, 2]	E	+ 3 * 4 $	shift
[1, 2, 4]	E +	3 * 4 $	shift
[1, 2, 4, 5]	E + Num	* 4 $	reduce by T -> Num
[1, 2, 4, 5]	E + T	* 4 $	shift
[1, 2, 4, 5, 6]	E + T *	4 $	shift
[1, 2, 4, 5, 6, 7]	E + T * Num	$	reduce by T -> T * Num
[1, 2, 4, 5]	E + T	$	reduce by E -> E + T
[1, 2]	E	$	acc



Step3
Implement with stack
(take shift/reduce action based on ***parsing table****)*

LR(0) - Example code

Grammar

E -> E + T | T

T -> T * F | F

F -> Num

```
# negative => reduce / positive => shift
parsing_table = {
1: {'N': 12, 'E': 2, 'T': 3},
2: {'+': 4, '-': 5, '$': -1},
3: {'+': -4, '-': -4, '$': -4, '*': 8, '/': 9, 'N': -4},
4: {'N': 12, 'T': 6},
5: {'N': 12, 'T': 7},
6: {'+': -2, '-': -2, '$': -2, '*': 8, '/': 9, 'N': -2},
7: {'+': -3, '-': -3, '$': -3, '*': 8, '/': 9, 'N': -3},
8: {'N': 10},
9: {'N': 11},
10:{'+': -5, '-': -5, '$': -5, '*': -5, '/': -5, 'N': -5},
11:{'+': -6, '-': -6, '$': -6, '*': -6, '/': -6, 'N': -6},
12:{'+': -7, '-': -7, '$': -7, '*': -7, '/': -7, 'N': -7},
}
```

```
while True:
    symb = self.E[self.pos]
    nxt_state = parsing_table[self.states[-1]][self.symb2key(symb)]
    if nxt_state > 0:
        # shift
        ...
        self.pos += 1
    else:
        # reduce
        pop_n, derived_symb = reduce_rules[-nxt_state]
        if pop_n == 0:
            # assume all input is valid, symbs will leave the answer
            return self.symbs[0]
        elif: pop_n == 1:  # simple reduction
            ...
        elif pop_n == 3:  # operator reduction
            ...
```

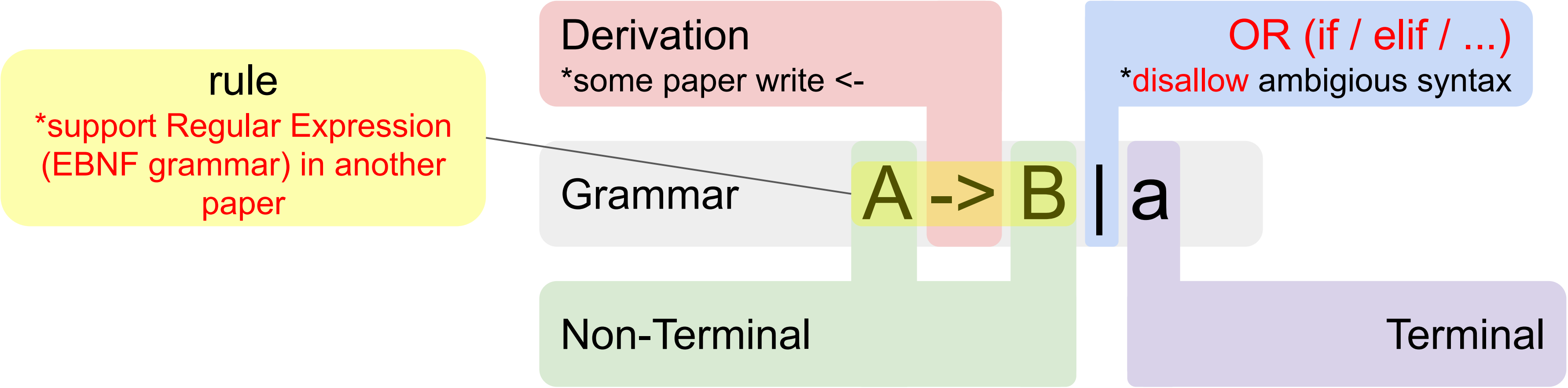
Shift

Reduce (Derivation)

Parser 102 - PEG and PEG parser

Introduced in 2002 ([Packrat Parsing: Simple, Powerful, Lazy, Linear Time])

Parsing Expression Grammar (PEG)



*Difference from traditional CFG

A will try **A -> B** first.

Only after it fails at **A -> B**, **A** will only try **A -> a**.

Example of difference

Grammar1: A -> a b | a

Grammar2: A -> a | a b

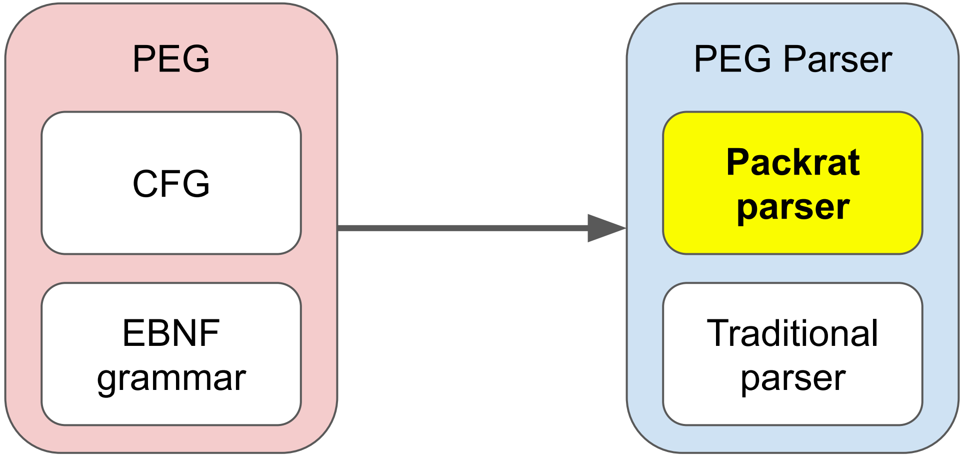
- *LL/LR parser will fail to complete when the input grammar is ambiguous.*
- *PEG parser only tries the first PEG rule. The latter rule will never succeed.*

> "A PEG parser generator will resolve unintended ambiguities earliest-match-first, which may be arbitrary and lead to surprising parses." (source)

PEG Parser

PEG parser means “parser generated based on **PEG**”.

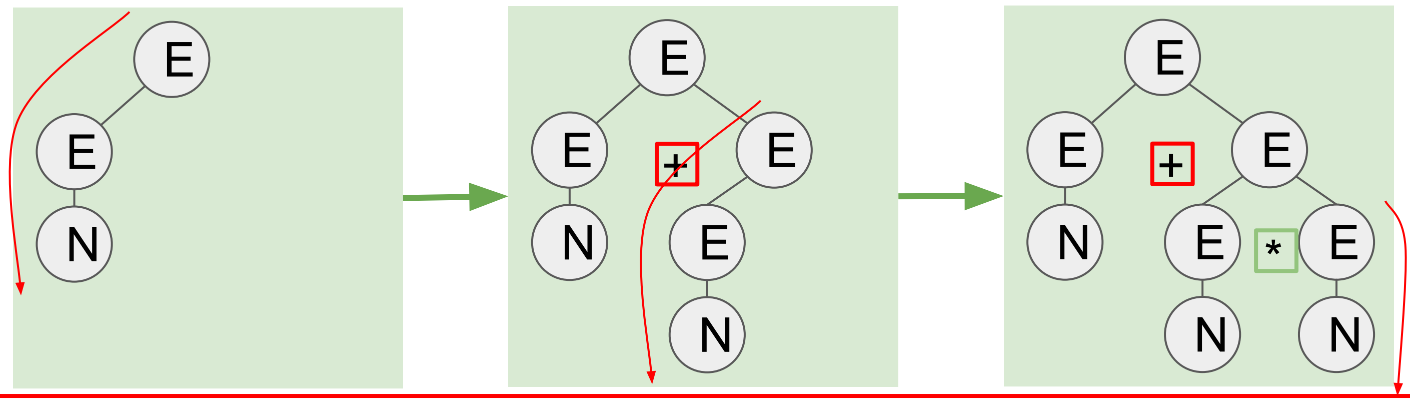
PEG parser can be a Packrat parser, or other traditional parser with k-lookahead limitation. **Mostly, PEG parser means Packrat parser**.



Parser 102 - Packrat parser

Type of Packrat parser

Top-down
Type



Packrat parser is top-down type.

**Idea of memoization was Introduced in 1970*

Packrat Parsing - Implementation

Step2
**parse from left to right*

Left recursion support

E -> E + T | T
T -> T * F | F
F -> Num

Step1
**write function for each non-terminal (PEG rule)*

2 + 3 * 4

**perform infinite lookahead + memoization*

parse_T() and parse_F()

parse_E() and parse_T()

parse_E()

Step3
**recursively parse the input string started from first rule parse_E()*

Packrat Parsing - Example code

Memoization

Grammar

E -> E + T | T
T -> T * F | F
F -> Num

```
@memo_left_rec
def pAdditive(self, idx) -> Tuple[int, int]:
    try:
        vleft, nidx = self.pAdditive(idx)
        symb, nnidx = self.pChar(nidx)
        vright, nnnidx = self.pMultitive(nnidx)
        if symb == '+':  # Additive -> Additive + Multitive
            return vleft + vright, nnnidx
        raise Exception('failed to run above derivation')
    except:
        return self.pMultitive(idx)  # Additive -> Multitive
```

Derivation

[509. Fibonacci Number]

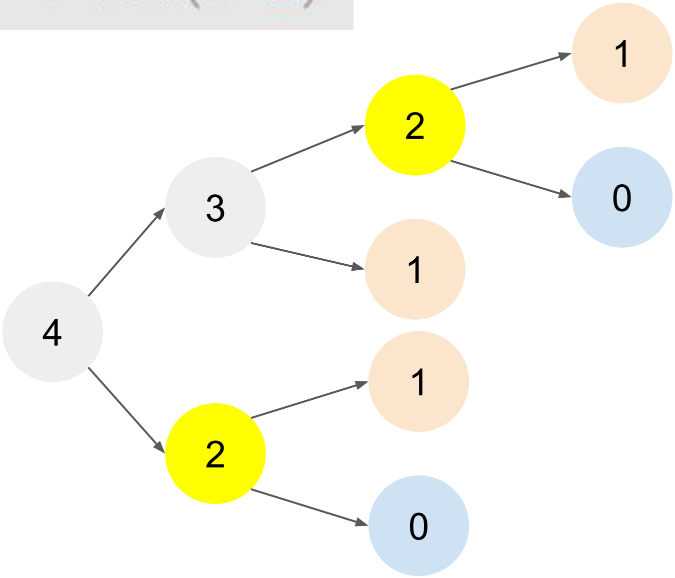
Packrat - what is memoization?

```python
def fib(n: int) -> int:
    if n < 2: return n
    return fib(n-1) + fib(n-2)
```

fib(0) = 0
fib(1) = 1
fib(2) = fib(1) + fib(0) = 1
fib(3) = fib(2) + fib(1) = fib(1) + fib(0) + fib(1) = 2
...

if n = 4, we calculate
fib(2), fib(0) twice, fib(1) thrice, fib(4), fib(3) once

TIme Complexity: O(2^n)



Packrat - what is memoization? (Cont.)

```
from functools import lru_cache
@lru_cache(maxsize=None)
def fib(n: int) -> int:
    if n < 2: return n
    return fib(n-1) + fib(n-2)
```

if n = 4, we…
calculate fib(4), fib(3), fib(2), fib(1), fib(0) once

Time Complexity: O(2^n) => O(n)
Space Complexity: O(1) => O(n)

Source: Guido's Medium (Left-recursive PEG Grammars)

Left recursion in Packrat parser

```
def parse_E():
    if parse_E() and parse_Char() == '+' and parse_T():
        ...
    if parse_T():
        ...
    raise Exception('Invalid Grammar')
```

Approach 1
if (count of operator) < (count function call):
  return False
Approach 2
reverse the call stack (adopted in CPython!)

```
def parse_E():
    if oracle_parse_E() and parse_Char() == '+' and parse_T():
        ...
    if parse_T():
        ...
    raise Exception('Invalid Grammar')
```

Normal Memoization

```
if key in memo:
    # Sub-problem is solved before => get the answer from cache
    res, endpos = memo[key]
    self.reset(endpos)
else:
    # Calculate the answer and set the answer to cache
    res = func(self, *args)
    endpos = self.mark()
    memo[key] = res, endpos
```

Left-recursion Memoization

```python
if key in memo: ...
else:
    # Prime the cache with a failure.
    memo[key] = lastres, lastpos = None, pos
    # Loop until no longer parse is obtained.
    while True:
        self.reset(pos)
        res = func(self, *args)
        endpos = self.mark()
        if endpos <= lastpos:
            break
        memo[key] = lastres, lastpos = res, endpos
    res = lastres
    self.reset(lastpos)
```

**perform infinite-lookahead*

Traditional parser V.S Packrat parser

Red text: 3 highlighted characteristics of Packrat parser.

Traditional parser vs Packrat parser

	Packrat	**Traditional**
Scan	Left-to-right (*Right-to-left memo)	Left-to-right
Left Recursion	Support (*Not support in first paper)	LL needs to rewrite the grammar
Ambigious	Disallowed (**determinism**)	Allowed
Space Complexity	O(Code Size) (**space consumption**)	O(Depth of Parse Tree)
Worst Time Complexity	Super linear time (**statelessness**) *Because of feature like **typedef** in C	Expotenial time
Capability	Basically covers all traditional cases (infinite lookahead)	No left-recursion/ambigious for LL Has k lookup limitations for both (e.g. [dangling else])

New rule in Python 3.10 based on PEG

Parenthesized context managers

PEP 622/634/635/636 - Structural Pattern Matching

CPython’s PEG parser

CPython Parser - Before/After

CPython3.8 and before use LL(1) parser written by Guido **30 years ago**

The parser requires steps to generate CST and convert CST to AST.

CPython3.9 uses PEG (Packrat) parser (Infinite lookahead)

PEG rule supports left-recursion

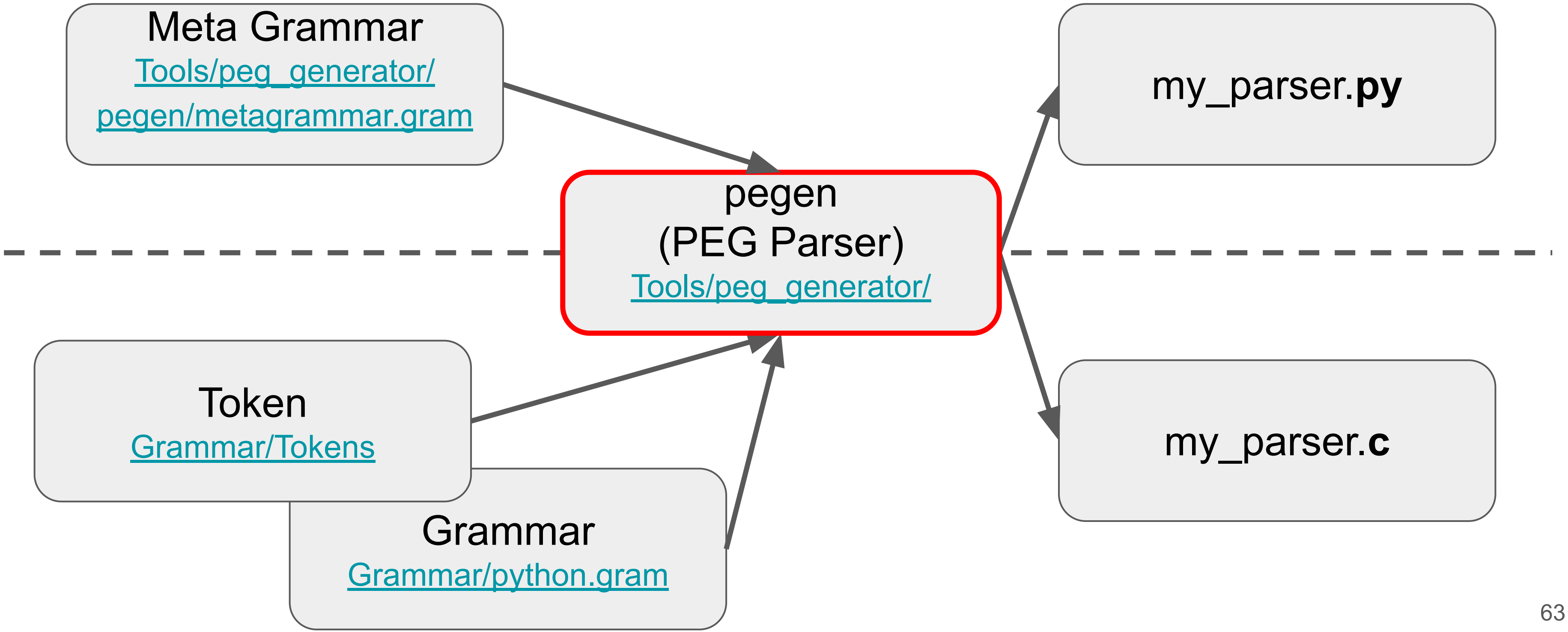
No more CST to AST step - source

CPython3.10 drops LL(1) parser support

This answers "Why PEG?"

*CPython contains a peg parser generator written in python3.8+ (because of warlus operator)

CPython Parser - Workflow



Input: Meta Grammar Example

Syntax Directed Translation (SDT)

Parser header (python code)

rule

non-Terminal

return type

PEG rule divider

PEG rule

action (python code)

```
@subheader """\

"""

start[int]: ret=additive { ret }
additive[int]:
      | lval=additive '+' rval=multitive { lval + rval }
      | lval=additive '-' rval=multitive { lval - rval }
      | multitive { multitive }
multitive[int]:
      | lval=multitive '*' rval=decimal { lval * rval }
      | lval=multitive '/' rval=decimal { lval // rval }
      | decimal { decimal }
decimal[int]:
      | NUMBER { int(number.string) }
```

Output: Generated PEG Parser

(Partial code)

```python
class GeneratedParser(Parser):

    @memoize
    def start(self) -> Optional[int]:
        # start: additive
        mark = self.mark()
        cut = False
        if (
            (ret := self.additive())
        ):
            return ret
        self.reset(mark)
        if cut: return None
        return None
```

```python
@memoize
def decimal(self) -> Optional[int]:
    # decimal: NUMBER
    mark = self.mark()
    cut = False
    if (
        (number := self.number())
    ):
        return int ( number . string )
    self.reset(mark)
    if cut: return None
    return None
```

Recap: Benefit / Performance

Benefit

- Grammar is more flexible: from LL(1) to LL(∞) (infinite lookahead)
- Hardware supports Packrat's memory consumption now
- Skip intermediate parse tree (CST) construction

Performance

- Within 10% of LL(1) parser both in speed and memory consumption (PEP 617)

Take away

Recap

- Parser 101 (Compiler class in school)
  - CFG
  - Traditional Parser
    - Top-down: LL(1)
    - Bottom-up: LR(0)
- Parser 102
  - PEG
  - Packrat Parser
- CPython
  - Parser in CPython
  - CPython’s PEG parser

Q. How to verify my understanding?

A. Get your hands dirty!

Leetcode: 227. Basic Calculator II

You can implement traditional parser like LL(1) and LR(0) parser, and Packrat parser from scratch!

Need Answer? note35/Parser-Learning

Q & A

Appendix

Related Articles

Guido van Rossum

[PEG Parsing Series Overview]

Bryan Ford

[Packrat Parsing: Simple, Powerful, Lazy, Linear Time]

[Parsing Expression Grammars: A Recognition-Based Syntactic Foundation]

Related Talks

Guido van Rossum @ North Bay Python 2019
Writing a PEG parser for fun and profit

Pablo Galindo and Lysandros Nikolaou @ Podcast.__init__
The Journey To Replace Python's Parser And What It Means For The Future

Emily Morehouse-Valcarcel @ PyCon 2018
The AST and Me

Alex Gaynor @ PyCon 2013
So you want to write an interpreter?

Thanks for your listening!