Lesson 1: Week 10 - Notes 15 Context Free Grammars

Created: 2024-05-27T01:25:43.971614+10:00

Week 10 - Context free grammars

Useful links about context-free grammars

Context-free grammarhttps://en.wikipedia.org/wiki/Context-free_grammarContext-sensitive grammarhttps://en.wikipedia.org/wiki/Context-sensitive_grammar

Week 10 - Context free grammar

CFG Example

S → aSa | bSb | a | b | εis same asS → aSaS → bSbS → aS → bS → εS is the starting symbol and ε is the empty wordDoes this word 'abbabba' belong to the language of the grammar above?S → aSa → abSba → abbSbba → abbabba YESabab? NOelle, radar, level, and refer are examples of palindrome

context_free_grammar.py

Context free grammars

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming

[1]:
```
from itertools import product
```

Consider two kinds of symbols, in finite numbers: nonterminals, usually represented as uppercase letters, and terminals, usually represented as lowercase letters, one of which, denoted by ε, is special and represents the empty symbol. A production rule maps a nonterminal to a finite sequence of terminals and nonterminals, the former being the left hand side of the rule, the latter its right hand side. A context free grammar (CFG) is a finite number of production rules together with a distinguished nonterminal, referred to as the starting symbol.

Let $\mathcal{G}$ denote a context free grammar. The set of production rules of $\mathcal{G}$ is usually represented as follows. Let α be a nonterminal that is the left hand side of at least one of $\mathcal{G}$'s production rules. If only one production rule of $\mathcal{G}$ has α as left hand side, then one line of the represention is "$\alpha \to \Lambda$" with Λ the right hand side of that rule. If n production rules of $\mathcal{G}$ have α as left hand side with $n > 1$, then one line of the represention is "$\alpha \to \Lambda_1 | \ldots | \Lambda_n$" with Λ_1, ..., Λ_n the right hand sides of those n rules, in an arbitrary order.

A derivation of a $\mathcal{G}$ is a sequence of the form "$\Lambda_0 \to \cdots \to \Lambda_n$" for some $n \geq 0$ where: * Λ_0 is $\mathcal{G}$'s starting symbol; * for all $p < n$, Λ_{p+1} is obtained from Λ_p by replacing an occurrence in Λ_p of a nonterminal α by the right hand side of a production rule of $\mathcal{G}$ whose left hand side is α.

Λ_n is said to be generated by $\mathcal{G}$. If nothing but terminals occur in Λ_n, then the derivation cannot be extended to a longer derivation.

The language of $\mathcal{G}$ is the set of finite sequences of terminals generated by $\mathcal{G}$.

A derivation "$\Lambda_0 \to \cdots \to \Lambda_n$" is a leftmost derivation if for all $p < n$, the leftmost occurrence in Λ_p of a nonterminal is the one that is replaced by the right hand side of a production rule of $\mathcal{G}$ to yield Λ_{p+1}. It can be shown that each member of the language of $\mathcal{G}$ ends some leftmost derivation of $\mathcal{G}$: the language of $\mathcal{G}$ allows derivations to be restricted to leftmost ones.

Following are four examples of context free grammars.

1 Palindromes over $\{a, b\}$

Besides ε, the terminals are a and b. The starting symbol is the unique nonterminal, S. The grammar is represented as follows.

$S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$

It is easy to see that the language of this grammar is the set of finite sequences of a's and b's that read identically from left to right and from right to left (palindromes over $\{a, b\}$): the empty sequence, a, b, aa, bb, aaa, bbb, aba, bab...

Clearly, every palindrome Λ over $\{a, b\}$ ends a unique derivation, that involves nothing but sequences where S occurs once and only once, except for the final sequence, Λ. Hence all derivations are leftmost. For instance, the unique derivation of the palindrome *abbbaaaabbba* is:

$S \to aSa \to abSba \to abbSbba \to abbbSbbba \to abbbaSabbba \to abbbaaSaabbba \to abbbaaaabbba$

2 Members of the set $\{b^n a^m b^{2n} \mid n \geq 0, m \geq 0\}$

Besides ε, the terminals are a and b. The nonterminals are S and A, S being the starting symbol. The grammar is represented as follows.

$S \to bSbb \mid A$
$A \to aA \mid \varepsilon$

It is easy to see that the language of this grammar is the set of finite sequences of a's and b's of the form $b^n a^m b^{2n}$, $n \geq 0$, $m \geq 0$.

Clearly, every member Λ of $\{b^n a^m b^{2n} \mid n \geq 0, m \geq 0\}$ ends a unique derivation, that involves nothing but sequences where either S of A occurs once and only once, except for the final sequence, Λ. Hence all derivations are leftmost. For instance, the unique derivations of the empty sequence, *bbbbbb*, *aaa* and *bbbaabbbbbb* are:

$S \to A \to \varepsilon$

$S \to bSbb \to bbSbbbb \to bbAbbbb \to bbbbbb$

$S \to A \to aA \to aaA \to aaaA \to aaa$

$S \to bSbb \to bbSbbbb \to bbbSbbbbbb \to bbbAbbbbbb \to bbbaAbbbbbb \to bbbaaAbbbbbb \to bbbaabbbbbb$

3 Well-formed nested parentheses and square brackets

Here ε is not used. The terminals are (,), [and]. The starting symbol is the unique nonterminal, S. The grammar is represented as follows.

$S \to SS$
$S \to ()$
$S \to (S)$
$S \to []$
$S \to [S]$

It is easy to see that the language of this grammar is the set of well-formed nested parentheses and square brackets. For instance, ()[] is a member of this set; it has two derivations: * $S \to SS \to ()S \to ()[]$, which is leftmost; * $S \to SS \to S[] \to ()[]$, which is not leftmost.

Some well-formed nested parentheses and square brackets have many leftmost derivations. For instance, ()()() has two: * $S \to SS \to ()S \to ()SS \to ()()S \to ()()()$ * $S \to SS \to SSS \to ()SS \to ()()S \to ()()()$

For another example, the leftmost derivation of ([[[()()[][]]]([])]) is:

$S \to (S) \to ([S]) \to ([SS]) \to ([[S]S]) \to ([[[S]]S]) \to ([[[SS]]S]) \to ([[[SSS]]S]) \to ([[[SSSS]]S]) \to ([[[()SSS]]S]) \to ([[[()()SS]]S]) \to ([[[()()[]S]]S]) \to ([[[()()[][]]]S]) \to$

$([[[()()[][]]](S)]) \to ([[[()()[][]]]([])])$

4 Strings over $\{a, b\}$ with an unequal number of a's and b's

Besides ε, the terminals are a and b. The nonterminals are S, T and U, S being the starting symbol. The grammar is represented as follows.

$S \to U \mid V$
$U \to TaU \mid TaT$
$V \to TbV \mid TbT$
$T \to aTbT \mid bTaT \mid \varepsilon$

To see that the language of this grammar is the set of finite sequences of a's and b's with an unequal number of a's and b's, read the four lines in the grammar's representation as follows:

- Line 1: a string with an unequal number of a's and b's is a string with more a's or a string with more b's.
- Line 2: a string with more a's than b's has a smallest initial segment with more a's; this initial segment ends in a, has an equal number of a's and b's before that last occurrence of a, and is followed by a string with a number of occurrences of a's at least equal to the number of occurrences of b's.
- Line 3: a string with more b's than a's has a smallest initial segment with more b's; this initial segment ends in b, has an equal number of a's and b's before that last occurrence of b, and is followed by a string with a number of occurrences of b's at least equal to the number of occurrences of a's.
- Line 4: a nonempty string with an equal number of a's and b's either starts with a or with b. If it starts with a, that initial a is followed by a smallest substring with a number of occurrences of b equal to 1 plus the number of occurrences of a; such a substring ends in b, has an equal number of a's and b's before that last occurrence of b, and is followed by a string with a number of occurrences of a's equal to the number of occurrences of b's. If it starts with b, that initial b is followed by a smallest substring with a number of occurrences of a equal to 1 plus the number of occurrences of b; such a substring ends in a, has an equal number of a's and b's before that last occurrence of a, and is followed by a string with a number of occurrences of a's equal to the number of occurrences of b's.

The reading shows that every string over $\{a, b\}$ with an unequal number of a's and b's is generated by the grammar. It is easy to verify that conversely, any string over $\{a, b\}$ that is generated by the grammar has an unequal number of a's and b's.

Some strings over $\{a, b\}$ with an unequal number of a's and b's have many leftmost derivations. For instance, aba has two: * $S \to U \to TaT \to aT \to abTaT \to abaT \to aba$ * $S \to U \to TaT \to aTbTaT \to abTaT \to abaT \to aba$

For other examples, here are some leftmost derivations of $abbabaaab$ and $bbbbaa$, preceded with visualisations that illustrate how to obtain them:

	a		b		b		a		b		a		a		b
S															
U															
T											a	T			
	a	T	b	T							a	T			
	a		b	T							a	T			
	a		b		b	T	a	T			a	T			
	a		b		b		a	T			a	T			
	a		b		b		a		b	T	a	T	a	T	
	a		b		b		a		b		a	T	a	T	
	a		b		b		a		b		a	T			
	a		b		b		a		b		a		a	T	b T
	a		b		b		a		b		a		a		b T
	a		b		b		a		b		a		a		b

$S \to U \to TaT \to aTbTaT \to abTaT \to abbTaTaT \to abbaTaT \to abbabTaTaT \to abbabaTaT \to abbabaaT \to abbabaaaTbT \to abbabaaabT \to abbabaaab$

	b		b		b		b		a		a
S											
V											
T	b	V									
	b	V									
	b	T	b	T							
	b		b	T							
	b		b		b	T				a	T
	b		b		b		b	T	a T	a	T
	b		b		b		b		a T	a	T
	b		b		b		b		a		a T
	b		b		b		b		a		a

$S \to V \to TbV \to bV \to bTbT \to bbT \to bbbTaT \to bbbbTaaT \to bbbbaaT \to bbbbaa$

Uppercase and lowercase alphabetic characters will play the role of nonterminals and terminals, respectively. For ε, it is convenient to use , which is an alphabetic character, hence can be part of a Python identifier or be an identifier by itself. Since ε represents the empty symbol, we initialise to the empty string:

[2]:
```
' '.isalpha()

 = ''
```

[2]:
```
True
```

It is natural to capture the representation of $\mathcal{G}$'s set of production rules as a dictionary, whose keys are the nonterminals, and whose values are the sets of strings that capture the right hand sides of the production rules with a given left hand side. The starting symbol is a one of the keys. Let us for some time fix $\mathcal{G}$'s production rules and starting symbol to the following:

[3]:
```
rules = {'J': {'a', 'b', 'c'}, 'D': {'H', 'ab'}, 'B': {'EH'}, 'I': {'IC'},
         'H': {}, 'S': {'BG', 'b'}, 'G': {'a', '', 'Sbc'}, 'F': {'Cd'},
         'E': {'GH', 'GFHGa'}, 'C': {'a', 'bc'}, 'A': {'FI', 'J'}
        }

starting_symbol = 'S'
```

Let us write code to display $\mathcal{G}$'s representation. We intend to display first the line for the starting symbol, then the lines for all other nonterminals in alphabetical order. For a given nonterminal, we intend to display the right hand sides of the production rules having that nonterminal as left hand side in lexicographic order. The following generator function yields the nonterminals as desired:

[4]:
```
def ordered_nonterminals():
    yield starting_symbol
    yield from sorted(rules.keys() - {starting_symbol})

list(ordered_nonterminals())
```

[4]:
```
['S', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

, which evaluates to the empty string, needs to be treated specially to be properly displayed:

[5]:
```
[production or ' ' for production in sorted(rules['G'])]
```

[5]:
```
[' ', 'Sbc', 'a']
```

The following code fragment first displays what sits to the right of $\rightarrow$ on the line for the nonterminal G, then the whole line for the nonterminal G, then the whole representation:

[6]:
```
print(' | '.join(production or ' ' for production in sorted(rules['G'])))
print()

print(' -> '.join(('G', ' | '.join(production or ' '
                                   for production in sorted(rules['G'])
                                  )
                  )
                 )
     )
print()

print('\n'.join(' -> '.join((f'{nonterminal}',
                              ' | '.join(production or ' ' for production in
                                         sorted(rules[nonterminal])
                                        )
                             )
                            ) for nonterminal in ordered_nonterminals()
                )
     )
```

```
  | Sbc | a

G ->   | Sbc | a

S -> BG | b
A -> FI | J
B -> EH
C -> a | bc
D -> H | ab
E -> GFHGa | GH
F -> Cd
G ->   | Sbc | a
H ->
I -> IC
J -> a | b | c
```

Let us for a moment get back to $\mathcal{G}$ being arbitrary. Clearly, the empty sequence belongs to the language of $\mathcal{G}$ only if ε is involved in $\mathcal{G}$'s set of production rules. It turns out that any other member of the language of $\mathcal{G}$ does not need ε to be generated: in case it appears in $\mathcal{G}$'s set of production rules, ε can be eliminated, resulting in a new set of production rules for a grammar whose language does not contain the empty sequence where the language of $\mathcal{G}$ might, but otherwise is the language of $\mathcal{G}$.

The elimination of ε proceeds in two stages, and is easily verified to correctly produce a set of production rules not involving ε for a CFG whose language is the language of $\mathcal{G}$, with the possible exception of the empty sequence. Let $\mathcal{G}'$ denote that CFG derived from $\mathcal{G}$.

- In the first stage, one identifies the set $\mathfrak{S}_1$ of nonterminals that produce ε directly (so the nonterminals α such that one of the production rules of $\mathcal{G}$ maps α to ε), then the set $\mathfrak{S}_2$ of nonterminals that produce a sequence of symbols that all belong to $\mathfrak{S}_1$, then the set $\mathfrak{S}_3$ of nonterminals that produce a sequence of symbols that all belong to $\mathfrak{S}_2$… until no new nonterminal is discovered. Denote by $\mathfrak{S}$ the resulting set of nonterminals.
- In the second stage, for all production rules $\mathcal{R}$ of $\mathcal{G}$ that map a nonterminal α to a sequence of symbols $\beta_0 \dots \beta_n$, any occurrence of β_i, $i \leq n$, that belongs to $\mathfrak{S}$, is either kept or eliminated in the production rules of $\mathcal{G}'$ that correspond to $\mathcal{R}$. Hence if a production rule $\mathcal{R}$ of $\mathcal{G}$ has k occurrences of a member of $\mathfrak{S}$ on its right hand side, then $\mathcal{G}'$ has 2^k production rules that correspond to $\mathcal{R}$.

Let us again fix $\mathcal{G}$'s production rules and starting symbol as we did above. It is then easy to verify that $\mathfrak{S} = \{G, H, E, D, B, S\}$, and that since both G and H belong to $\mathfrak{S}$, the production rule that maps E to $GFHGa$ correspondingly has:

- one production rule that maps E to $GFHGa$;
- one production rule that maps E to $GFHa$;
- one production rule that maps E to $GFGa$;
- one production rule that maps E to GFa;
- one production rule that maps E to $FHGa$;
- one production rule that maps E to FHa;
- one production rule that maps E to FGa;
- one production rule that maps E to Fa.

The following code fragment implements the first stage, tracing execution:

[7]:
```
generating_  = set()
left_to_examine = rules.keys()
while True:
    print('Nonterminals now known to generate  :', generating_ )
    print('Nonterminals left to (re)examine:', left_to_examine)
    new_nonterminals_generating_  = set()
    for nonterminal in left_to_examine:
        print(f'  Considering {nonterminal}',
              ' | '.join(production or ' '
                         for production in sorted(rules[nonterminal])
                         ), sep=' -> ', end=' ... '
             )
        if any(production == '' or set(production) <= generating_
                    for production in rules[nonterminal]
               ):
            print('found out to generate  ')
            generating_ .add(nonterminal)
            new_nonterminals_generating_ .add(nonterminal)
        else:
            print()
    if new_nonterminals_generating_ :
        left_to_examine -= generating_
        print()
    else:
        break
```

```
Nonterminals now known to generate  : set()
Nonterminals left to (re)examine: dict_keys(['J', 'D', 'B', 'I', 'H', 'S', 'G',
'F', 'E', 'C', 'A'])
  Considering J -> a | b | c …
  Considering D -> H | ab …
  Considering B -> EH …
  Considering I -> IC …
  Considering H ->  … found out to generate
  Considering S -> BG | b …
  Considering G ->  | Sbc | a … found out to generate
  Considering F -> Cd …
  Considering E -> GFHGa | GH … found out to generate
  Considering C -> a | bc …
  Considering A -> FI | J …

Nonterminals now known to generate  : {'H', 'G', 'E'}
Nonterminals left to (re)examine: {'B', 'J', 'F', 'I', 'A', 'D', 'S', 'C'}
  Considering B -> EH … found out to generate
  Considering J -> a | b | c …
  Considering F -> Cd …
```

```
  Considering I -> IC …
  Considering A -> FI | J …
  Considering D -> H | ab … found out to generate
  Considering S -> BG | b … found out to generate
  Considering C -> a | bc …

Nonterminals now known to generate  : {'B', 'G', 'E', 'D', 'S', 'H'}
Nonterminals left to (re)examine: {'J', 'F', 'I', 'A', 'C'}
  Considering J -> a | b | c …
  Considering F -> Cd …
  Considering I -> IC …
  Considering A -> FI | J …
  Considering C -> a | bc …
```

For the second stage, rules whose right hand side does not contain any nonterminal in $\mathfrak{S}$ are taken as such. For all other production rules $\mathcal{R}$ of $\mathcal{G}$, it is convenient to use the `product` class from the `itertools` module to generate all possible right hand sides of the rules that correspond to $\mathcal{R}$ in $\mathcal{G}'$. For the production rule that maps E to $GFGHa$, with both G and H but not F belonging to $\mathfrak{S}$, the right hand sides of the corresponding rules of $\mathcal{G}'$ can be obtained as follows:

[8]:
```
list(product(*((symbol, '') if symbol in generating_  else (symbol,)
                   for symbol in ('G', 'F', 'H', 'G', 'a')
               )
            )
    )

[''.join(symbols) for symbols in product(
        *((symbol, '') if symbol in generating_  else (symbol,)
             for symbol in ('G', 'F', 'H', 'G', 'a')
         )
                                        )
]
```

[8]:
```
[('G', 'F', 'H', 'G', 'a'),
 ('G', 'F', 'H', '', 'a'),
 ('G', 'F', '', 'G', 'a'),
 ('G', 'F', '', '', 'a'),
 ('', 'F', 'H', 'G', 'a'),
 ('', 'F', 'H', '', 'a'),
 ('', 'F', '', 'G', 'a'),
 ('', 'F', '', '', 'a')]
```

[8]:
```
['GFHGa', 'GFHa', 'GFGa', 'GFa', 'FHGa', 'FHa', 'FGa', 'Fa']
```

The following code fragment uses this technique and implements the second stage, tracing execution. When a rule $\mathcal{R}$ is produced that has an empty right hand side (because it comes from a rule of $\mathcal{G}$ whose right hand side consists of nothing but nonterminals in $\mathfrak{S}$ and none of those terminals is kept, then $\mathcal{R}$ is eventually deleted. When for a given nonterminal α, all produced rules with α as

left hand side have an empty right hand side and so are deleted, then the representation of $\mathcal{G}'$ loses the line for α:

[9]:
```
rules_without_  = {nonterminal: rules[nonterminal]
                   for nonterminal in rules if nonterminal not in generating_
                  }
for nonterminal in generating_ :
    print('Creating rules to replace', nonterminal,
          ' | '.join(production or ''
                     for production in sorted(rules[nonterminal])
                    ), sep=' -> '
         )
    new_productions =\
        {''.join(symbols) for production in rules[nonterminal] if production
                          for symbols in product(
                        *((symbol, '') if symbol in generating_  else (symbol,)
                          for symbol in production
                         )
                                                 )
        }
    print('    New rules:', nonterminal, '-> ', end='')
    if new_productions:
        print(' | '.join(production or "''"
                         for production in sorted(new_productions)
                        )
             )
    else:
        print("''")
    new_productions -= {''}
    if new_productions:
        rules_without_ [nonterminal] = new_productions

rules_without_
```

```
Creating rules to replace -> B -> EH
    New rules: B -> '' | E | EH | H
Creating rules to replace -> G ->   | Sbc | a
    New rules: G -> Sbc | a | bc
Creating rules to replace -> E -> GFHGa | GH
    New rules: E -> '' | FGa | FHGa | FHa | Fa | G | GFGa | GFHGa | GFHa | GFa |
GH | H
Creating rules to replace -> D -> H | ab
    New rules: D -> '' | H | ab
Creating rules to replace -> S -> BG | b
    New rules: S -> '' | B | BG | G | b
Creating rules to replace -> H ->
    New rules: H -> ''
```

```
[9]: {'J': {'a', 'b', 'c'},
      'I': {'IC'},
      'F': {'Cd'},
      'C': {'a', 'bc'},
      'A': {'FI', 'J'},
      'B': {'E', 'EH', 'H'},
      'G': {'Sbc', 'a', 'bc'},
      'E': {'FGa',
       'FHGa',
       'FHa',
       'Fa',
       'G',
       'GFGa',
       'GFHGa',
       'GFHa',
       'GFa',
       'GH',
       'H'},
      'D': {'H', 'ab'},
      'S': {'B', 'BG', 'G', 'b'}}
```

Let us for a moment get back to $\mathcal{G}$ being arbitrary. A natural question is whether, given a finite sequence Λ of terminals, $\mathcal{G}$ generates Λ. It Λ is empty, then it suffices to check whether $\mathcal{G}$'s starting symbol belongs to $\mathfrak{S}$. If Λ is not empty, then it is equivalent to ask whether Λ is generated by $\mathcal{G}'$. This makes the question easier to answer: if ε is present in some production rules of $\mathcal{G}$, then a derivation of Λ by $\mathcal{G}$ might involve arbitrarily long sequences of symbols, because any symbol in $\mathfrak{S}$ can eventually be erased. On the other hand, in any derivation of $\mathcal{G}'$, sequences of symbols can only increase in size as the derivation progresses. Since the number of sequences of terminals and nonterminals of a CFG of a given size is finite, the search for a derivation of Λ can be exhaustive and yield a definite outcome in finite time. The search can be reduced to leftmost derivations.

Still, it is of interest to ask the more general question: given a nonempty finite sequence Λ of terminals, does $\mathcal{G}$ generate Λ without making use of ε? Then one can ask whether $\mathcal{G}'$ generates Λ without making use of ε, which is the same as asking whether $\mathcal{G}'$ generates Λ, which is equivalent to asking whether $\mathcal{G}$ generates Λ.

The method is the following. We work with pairs (w_1, w_2) with w_1 a sequence of terminals and w_2 a sequence of nonterminals and terminals. Only pairs where w_1 is an initial segment of Λ are "promising". The aim is to eventually generate (Λ, ε). A set $\mathfrak{A}$ keeps track of all promising pairs that have been generated, and a subset $\mathfrak{B}$ of $\mathfrak{A}$ keeps track of the pairs on which we can further operate. We start with $\mathfrak{A}$ and $\mathfrak{B}$ containing (S, ϵ) only with S denoting $\mathcal{G}$'s starting symbol, and we proceed in stages. At any given stage, if $\mathfrak{B}$ is empty, then one declares that Λ cannot be generated; otherwise, one gets a pair (w_1, w_2) out of $\mathfrak{B}$ and move from the left of w_2 to the right of w_1 the longest initial segment of w_2 consisting of terminals, resulting in a pair (w_1', w_2') with w_2' empty or starting with a nonterminal α.

- If w_1' is not an initial segment of Λ, then (w_1', w_2') has a bad start.
- Otherwise, if w_2' is empty, then either w_1' is Λ and known to be generated by $\mathcal{G}$ without using ε and we are done, or w_1' is not Λ (which is equivalent to w_1' being shorted than Λ), so not

what we want.

- Otherwise, we consider all production rules of $\mathcal{G}$ with α as left hand side and not ε as right hand side. For each such rule, we replace in w_2' the occurrence of α on the left with the rule's right hand side, resulting in a new pair (w_1', w_2'').
  - If the combined length of w_1' and w_2'' is longer than the length of Λ, then (w_1', w_2'') is too long.
  - If (w_1', w_2'') belongs to $\mathfrak{A}$, then it has been seen already.
  - Otherwise, (w_1', w_2'') is "promising", recorded in $\mathfrak{A}$ as seen, and in $\mathfrak{B}$ for potential further consideration.

The function that follows implements this method. Rather than a set, it uses a list for $\mathfrak{A}$, and at every stage, pops its rightmost element, making the method a form of depth-first search. We trace execution with a CFG that generates the palindromes over $\{a, b\}$, defined with a little complication to easily witness the case where we process a pair that has been seen already:

[10]:
```
def can_generate_with_no_ (word):
    generated_bigrams = [('', starting_symbol)]
    seen_bigrams = set(generated_bigrams)
    while generated_bigrams:
        w_1, w_2 = generated_bigrams.pop()
        print('Considering', (w_1, w_2), end=' ... ')
        while w_2 and not w_2[0].isupper():
            w_1, w_2 = w_1 + w_2[0], w_2[1: ]
        print('transformed to', (w_1, w_2), end=' ... ')
        if not word.startswith(w_1):
            print('bad start')
            continue
        if not w_2:
            if len(w_1) == len(word):
                print('what I want!')
                return
            print('not what I want')
            continue
        print('ok')
        for pattern in rules[w_2[0]]:
            if not pattern:
                continue
            print('  Looking at rule', w_2[0], '->', pattern, end=' ... ')
            new_bigram = w_1, pattern + w_2[1: ]
            if len(new_bigram[0]) + len(new_bigram[1]) > len(word):
                print(new_bigram, 'too long')
                continue
            if new_bigram in seen_bigrams:
                print(new_bigram, 'already seen')
                continue
            generated_bigrams.append(new_bigram)
            seen_bigrams.add(new_bigram)
            print(new_bigram, 'to consider')
```

```
    print('Cannot be generated')

rules = {'T': {'U'}, 'U': {'T', 'S'}, 'S': {'aSa', 'bSb', 'a', 'b',  }}
starting_symbol = 'T'

can_generate_with_no_ ('aabaa')
print()

can_generate_with_no_ ('aabbaa')
```

```
Considering ('', 'T') … transformed to ('', 'T') … ok
  Looking at rule T -> U … ('', 'U') to consider
Considering ('', 'U') … transformed to ('', 'U') … ok
  Looking at rule U -> T … ('', 'T') already seen
  Looking at rule U -> S … ('', 'S') to consider
Considering ('', 'S') … transformed to ('', 'S') … ok
  Looking at rule S -> aSa … ('', 'aSa') to consider
  Looking at rule S -> bSb … ('', 'bSb') to consider
  Looking at rule S -> a … ('', 'a') to consider
  Looking at rule S -> b … ('', 'b') to consider
Considering ('', 'b') … transformed to ('b', '') … bad start
Considering ('', 'a') … transformed to ('a', '') … not what I want
Considering ('', 'bSb') … transformed to ('b', 'Sb') … bad start
Considering ('', 'aSa') … transformed to ('a', 'Sa') … ok
  Looking at rule S -> aSa … ('a', 'aSaa') to consider
  Looking at rule S -> bSb … ('a', 'bSba') to consider
  Looking at rule S -> a … ('a', 'aa') to consider
  Looking at rule S -> b … ('a', 'ba') to consider
Considering ('a', 'ba') … transformed to ('aba', '') … bad start
Considering ('a', 'aa') … transformed to ('aaa', '') … bad start
Considering ('a', 'bSba') … transformed to ('ab', 'Sba') … bad start
Considering ('a', 'aSaa') … transformed to ('aa', 'Saa') … ok
  Looking at rule S -> aSa … ('aa', 'aSaaa') too long
  Looking at rule S -> bSb … ('aa', 'bSbaa') too long
  Looking at rule S -> a … ('aa', 'aaa') to consider
  Looking at rule S -> b … ('aa', 'baa') to consider
Considering ('aa', 'baa') … transformed to ('aabaa', '') … what I want!

Considering ('', 'T') … transformed to ('', 'T') … ok
  Looking at rule T -> U … ('', 'U') to consider
Considering ('', 'U') … transformed to ('', 'U') … ok
  Looking at rule U -> T … ('', 'T') already seen
  Looking at rule U -> S … ('', 'S') to consider
Considering ('', 'S') … transformed to ('', 'S') … ok
  Looking at rule S -> aSa … ('', 'aSa') to consider
  Looking at rule S -> bSb … ('', 'bSb') to consider
  Looking at rule S -> a … ('', 'a') to consider
```

```
  Looking at rule S -> b … ('', 'b') to consider
Considering ('', 'b') … transformed to ('b', '') … bad start
Considering ('', 'a') … transformed to ('a', '') … not what I want
Considering ('', 'bSb') … transformed to ('b', 'Sb') … bad start
Considering ('', 'aSa') … transformed to ('a', 'Sa') … ok
  Looking at rule S -> aSa … ('a', 'aSaa') to consider
  Looking at rule S -> bSb … ('a', 'bSba') to consider
  Looking at rule S -> a … ('a', 'aa') to consider
  Looking at rule S -> b … ('a', 'ba') to consider
Considering ('a', 'ba') … transformed to ('aba', '') … bad start
Considering ('a', 'aa') … transformed to ('aaa', '') … bad start
Considering ('a', 'bSba') … transformed to ('ab', 'Sba') … bad start
Considering ('a', 'aSaa') … transformed to ('aa', 'Saa') … ok
  Looking at rule S -> aSa … ('aa', 'aSaaa') too long
  Looking at rule S -> bSb … ('aa', 'bSbaa') too long
  Looking at rule S -> a … ('aa', 'aaa') to consider
  Looking at rule S -> b … ('aa', 'baa') to consider
Considering ('aa', 'baa') … transformed to ('aabaa', '') … bad start
Considering ('aa', 'aaa') … transformed to ('aaaaa', '') … bad start
Cannot be generated
```

Let us organise the whole code in a class `ContextFreeGrammar`, whose `__init()__` method receives as arguments the dictionary capturing the production rules and the starting symbol of $\mathcal{G}$. It is natural to let `__init()__` compute once and for all whether $\mathcal{G}$ generates the empty sequence, and also compute $\mathcal{G}'$. We want `ContextFreeGrammar` to define a method `can_generate_with_no_ ()`, meant to be passed as argument a string of terminal symbols to determine whether $\mathcal{G}$ can generate this sequence without making any use of ε, which is interesting in its own right. We also want `ContextFreeGrammar` to define a method `can_generate()`, meant to be passed as argument a string of terminal symbols to determine whether $\mathcal{G}$ generates this sequence, which we know is work for $\mathcal{G}'$. This suggests defining another class, `ContextFreeGrammarWithout` , able to complete the work required by `ContextFreeGrammar`'s `can_generate()` method, which is precisely what `ContextFreeGrammar`'s `can_generate_with_no_ ()` method does when no production rule involves ε. Though `ContextFreeGrammarWithout` seems to be a more specific type than `ContextFreeGrammar`, it would not be appropriate to let the `__init()__` method of `ContextFreeGrammarWithout` compute whether the empty sequence can be generated (the answer is No), nor compute a grammar not involving ε and generating the same language (empty sequence included), since `self` could just be returned. So we do not want `ContextFreeGrammarWithout` 's `__init()__` method to call `ContextFreeGrammar`'s `__init()__` method. We only want `ContextFreeGrammarWithout` to inherit some of `ContextFreeGrammar`'s methods: `__str()__` if that method has been defined to nicely output the grammar's representation, and `can_generate()`, whose implementation should be straightforward and just call `ContextFreeGrammar`'s `can_generate_with_no_ ()` method, so more precisely, overwrite `ContextFreeGrammar`'s implementation of `can_generate()`, which itself essentially calls `ContextFreeGrammarWithout` 's `can_generate()` method on the object produced by `__init()__`'s computation of $\mathcal{G}'$. This design makes `ContextFreeGrammar` a mixin of `ContextFreeGrammarWithout` . The syntax is `class ContextFreeGrammarWithout (ContextFreeGrammar)`, but it does not intend to make a `ContextFreeGrammarWithout` object a kind of `ContextFreeGrammar` object: rather, it just intends a `ContextFreeGrammarWithout` object to make use by inheritance of the `ContextFreeGrammar`

methods that make sense to a `ContextFreeGrammarWithout` object, such as `can_generate()`, but not of those that are irrelevant, such as `can_generate_with_no_ ()`.

To summarise, the key design of `ContextFreeGrammar` and `ContextFreeGrammarWithout` is outlined below. The syntax `class ContextFreeGrammarWithout (ContextFreeGrammar)` is meant to be read as: `ContextFreeGrammarWithout` is a subclass of `object` that can use methods of `ContextFreeGrammar` when appropriate. The fact that `ContextFreeGrammarWithout` 's `__init()__` method does not call `ContextFreeGrammar`'s `__init()__` method might be the best formal indicator that `ContextFreeGrammar` is a mixin of `ContextFreeGrammarWithout` , not a genuine parent class:

[11]:
```
class ContextFreeGrammar:
    def __init__(self, rules, starting_symbol):
        self.rules = rules
        self.starting_symbol = starting_symbol
        # Compute self._starting_symbol_generates_  (True or False).
        # Compute self.with_ _eliminated (a CFG making no use of   and
        # generating the same language, with the possible exception of
        # the empty sequence).

    def __str__(self):
        pass

    def can_generate_with_no_ (self, word):
        pass

    def can_generate(self, word):
        if word == '':
            return self._starting_symbol_generates_
        return self.with_ _eliminated.can_generate(word)


class ContextFreeGrammarWithout (ContextFreeGrammar):
    def __init__(self, rules, starting_symbol):
        self.rules = rules
        self.starting_symbol = starting_symbol

    def can_generate(self, word):
        return self.can_generate_with_no_ (word)
```

Putting things together:

[12]:
```
class ContextFreeGrammar:
    def __init__(self, rules, starting_symbol):
        self.rules = rules
        self.starting_symbol = starting_symbol
        generating_  = self._generates_ ()
        self._starting_symbol_generates_  = starting_symbol in generating_
```

```
        rules_without_  = {nonterminal: rules[nonterminal]
                                for nonterminal in rules
                                    if nonterminal not in generating_
                          }
        for nonterminal in generating_ :
            new_productions =\
          {''.join(symbols) for production in rules[nonterminal] if production
                                 for symbols in product(
                        *((symbol, '') if symbol in generating_  else (symbol,)
                              for symbol in production
                         )
                                                      )
          }
            new_productions -= {''}
            if new_productions:
                rules_without_ [nonterminal] = new_productions
        self.without_  = ContextFreeGrammarWithout (rules_without_ ,
                                                     self.starting_symbol
                                                    )

    def __str__(self):
        return '\n'.join(' -> '.join((f'{nonterminal}',
                                      ' | '.join(production or ' '
                                                  for production in
                                                sorted(self.rules[nonterminal])
                                                )
                                     )
                                    ) for nonterminal in
                                                 self._ordered_nonterminals()
                         )

    def _ordered_nonterminals(self):
        yield self.starting_symbol
        yield from sorted(self.rules.keys() - {self.starting_symbol})

    def _generates_ (self):
        generating_  = set()
        left_to_examine = self.rules.keys()
        while True:
            new_nonterminals_generating_  = set()
            for nonterminal in left_to_examine:
                if any(production == '' or set(production) <= generating_
                            for production in self.rules[nonterminal]
                       ):
                    generating_ .add(nonterminal)
                    new_nonterminals_generating_ .add(nonterminal)
            if new_nonterminals_generating_ :
```

```
                left_to_examine -= new_nonterminals_generating_
            else:
                break
        return generating_

    def can_generate_with_no_ (self, word):
        if word == '':
            return False
        generated_bigrams = [('', self.starting_symbol)]
        seen_bigrams = set(generated_bigrams)
        while generated_bigrams:
            w_1, w_2 = generated_bigrams.pop()
            while w_2 and not w_2[0].isupper():
                w_1, w_2 = w_1 + w_2[0], w_2[1 :]
            if not word.startswith(w_1):
                continue
            if not w_2:
                if len(w_1) == len(word):
                    return True
                continue
            for pattern in self.rules[w_2[0]]:
                if not pattern:
                    continue
                new_bigram = w_1, pattern + w_2[1 :]
                if len(new_bigram[0]) + len(new_bigram[1]) <= len(word)\
                   and new_bigram not in seen_bigrams:
                    generated_bigrams.append(new_bigram)
                    seen_bigrams.add(new_bigram)
        return False

    def can_generate(self, word):
        if word == '':
            return self._starting_symbol_generates_
        return self.without_ .can_generate(word)


class ContextFreeGrammarWithout (ContextFreeGrammar):
    def __init__(self, rules, starting_symbol):
        self.rules = rules
        self.starting_symbol = starting_symbol

    def can_generate(self, word):
        return self.can_generate_with_no_ (word)
```

[13]:
```
rules = {'S': {'aSa', 'bSb', 'a', 'b',  }}
starting_symbol = 'S'
```

```
CFG = ContextFreeGrammar(rules, starting_symbol)
print(CFG)
CFG.can_generate_with_no_ ('ababa')
CFG.can_generate_with_no_ ('abaaba')
CFG.can_generate('')
CFG.can_generate('abaaba')
CFG.can_generate('abaabba')
```

```
S ->  | a | aSa | b | bSb
```

[13]: True

[13]: False

[13]: True

[13]: True

[13]: False

[14]:
```
rules = {'S': {'bSbb', 'A'}, 'A': {'aA',  }}
starting_symbol = 'S'

CFG = ContextFreeGrammar(rules, starting_symbol)
print(CFG)
CFG.can_generate_with_no_ ('bbaabbbb')
CFG.can_generate('')
CFG.can_generate('bbaabbbb')
CFG.can_generate('bbbaabbbb')
```

```
S -> A | bSbb
A ->  | aA
```

[14]: False

[14]: True

[14]: True

[14]: False

[15]:
```
rules = {'S': {'SS', '()', '(S)', '[]', '[S]'}}
starting_symbol = 'S'

CFG = ContextFreeGrammar(rules, starting_symbol)
print(CFG)
CFG.can_generate('')
CFG.can_generate('([][()])[()]()')
```

```
CFG.can_generate('([][(])[()]()')
```

```
S -> () | (S) | SS | [S] | []
```

[15]: False

[15]: True

[15]: False

[16]:
```
rules = {'S': {'U', 'V'}, 'U': {'TaU', 'TaT'}, 'V': {'TbV', 'TbT'},
         'T': {'aTbT', 'bTaT', }
        }
starting_symbol = 'S'

CFG = ContextFreeGrammar(rules, starting_symbol)
print(CFG)
CFG.can_generate('')
CFG.can_generate('abbbbabaa')
CFG.can_generate('abbbabaa')
```

```
S -> U | V
T ->   | aTbT | bTaT
U -> TaT | TaU
V -> TbT | TbV
```

[16]: False

[16]: True

[16]: False