Travaux dirigés

Introduction à la Compilation

1-Analyse lexicale - Solutions

Jean-Christophe Le Lann, Quentin Ducasse

Janvier 2023

Exercice 1 : Identification de lexèmes

Question a : Etablir la liste des lexèmes du code Python fourni sur Moodle.

```python
import numpy as np
from logging import Logger

class NumberLogger(Logger):
    def __init__(self, file):
        self.file = file


    def log_nb(self, nb):
        self.log("INFO", np.format_float_scientific(nb))

if __name__ == "__main__":
    lg = NumberLogger()
    lg.log_nb(1.245)
```

Solution - Différents types de lexèmes sont présents :

- Mot-clés : `import`, `as`, `from`, `class`, `def`, `self`, `__init__`, `__name__`
- Ponctuation : `( ) : , . = ==`
  `t`
- Identifiants : `numpy`, `np`, `logging`, `Logger`, `NumberLogger`, `file`, `log_nb`, `nb`, `format_float_scientific`, `lg`
- Valeurs litérales : `INFO`, `__main__`, `1.245`

Question b : Etablir la liste des lexèmes du code C fourni sur Moodle.

```c
#include <stdio.h>

typedef struct complex {
    float real;
    float imag;
} complex;

complex add(complex n1, complex n2) {
    complex temp;
    temp.real = n1.real + n2.real;
    temp.imag = n1.imag + n2.imag;
    return temp;
}


int main() {
    complex n1 = {.float = 1, .imag = 3};
    complex n2 = {.float = 2, .imag = 4};
    complex result;

    result = add(n1, n2);

    printf("Sum = %.1f + %.1fi", result.real, result.imag);
    return 0;
}
```

Différents types de lexèmes sont présents :

- **Mot-clés :** `include`, `typedef`, `struct`, `return`, `int`, `main`
- **Ponctuation :** `< > { } ( ) ; = +, . , %`
- **Identifiants :** `stdio`, `h`, `complex`, `real`, `imag`, `add`, `n1`, `n2`, `temp`, `result`, `printf`
- **Valeurs littérales :** `1`, `2`, `3`, `4`, `0`, `"Sum = %.1f + %.1fi"`

Question c : Etablir la liste des lexèmes d'un code personnel, d'une langage de votre choix, que vous avez eu l'occasion d'écrire dans un autre cours.

Exercice 2 : Expressions régulières

Question a : A l'aide du site web "Pythex" (ou "Rubular"), établir l'expression régulière unique décrivant les nombres en virgule flottante. On pourra bien entendu y inclure la notation scientifique, comme dans `-123.12345e-42`

Solution : L'expression régulière correspondante est :

- `[+-]?[0-9]+(\.[0-9]*)?(e[+-]?[0-9]+)?`
- ou `[+-]?\d+(\.\d*)?(e[+-]?\d+)?`

Question b : Un fichier contenant des mots français est fourni sous Moodle. Ecrire un programme Python permettant de lister l'ensemble des mots vérifiant les motifs suivants :

- Mots se terminant par `ion`.
- Mots possédant le motif `at`, non-suivi de 't', ni de 'r', ni de 'i'.

Solution : Les expressions régulières correspondantes sont :

- `ion$`
- `at[^irt]`

Solution [Ruby] : On pourrait écrire :

```ruby
filename=ARGV.first
lines=IO.readlines(filename)
p lines.count{|line| line.match(/ion$/)!=nil}
```

ou un "one-liner" :

```ruby
IO.readlines(ARGV.first).count{|line| line.match(/ion$/)!=nil}
```

Pour le second motif :

```ruby
IO.readlines(ARGV.first).count{|line| line.match(/at[^tri]/)!=nil}
```

Solution [Python] :

```python
import re

def test_regex_on(pattern, file):
    regex = re.compile(pattern)
    matching_words = []
    with open(file, "r") as dic_fr:
        for line in dic_fr:
            if regex.match(line):
                matching_words.append(line.strip())
    print(f"Found {len(matching_words)} matches for pattern '{str(pattern)}'")
    return matching_words

if __name__ == "__main__":
    pattern1 = r'.*ion$'
```

```
    pattern2 = r'.*at[^irt].*$'
    # Pattern 1 : 2096 matches
    test_regex_on(pattern1, "dictionary.txt")
    # Pattern 2 : 7206 matches
    test_regex_on(pattern2, "dictionary.txt")
```

Exercice 3 : Lexer simple

Question : Compléter la liste des expressions régulières permettant d'automatiser l'analyse lexicale réalisée (à la main) dans l'exercice 1.

Solution : Pour les mêmes bouts de code présentés au-dessus, les expressions régulières du lexer correspondent au lexème lui-même avec comme cas particuliers :

- Identifiant : `(r'[a-z]\w*', 'IDENTIFIER')`
- Nombre : `(r'-?\d+', 'NUMBER')`
- Commentaire C : `(r'\/\/[\s\S]*\n', 'COMMENT')`
- Commentaire Python : `(r'\#[\s\S]*\n', 'COMMENT')`

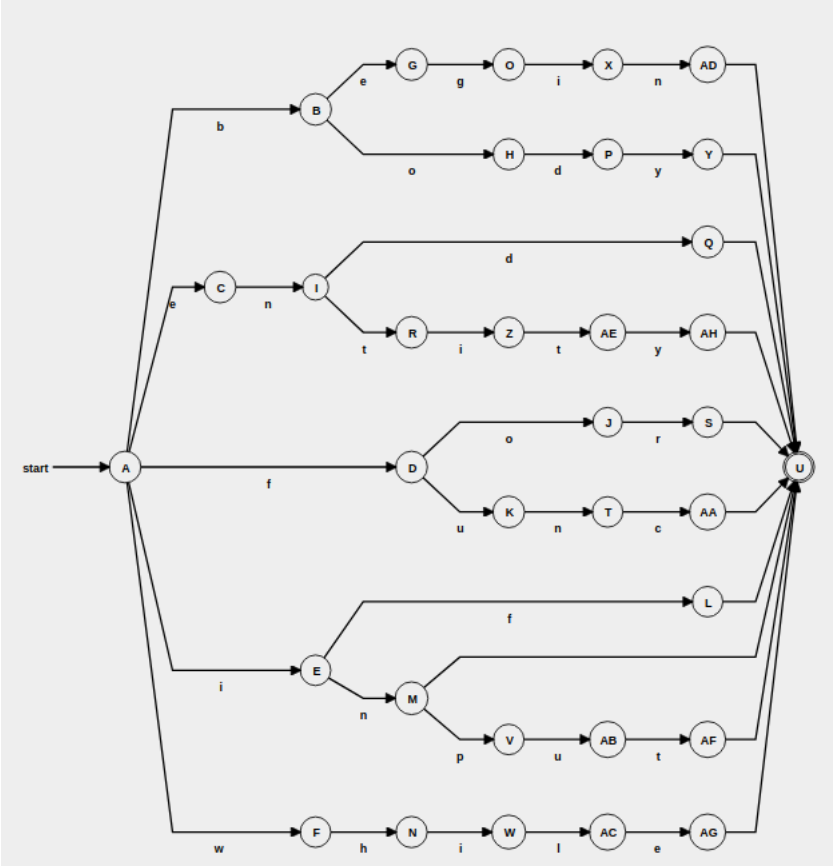
Le code donné est :

```
import re
import sys

regexExpressions = [
    # Comments and whitespaces
    (r'\/\/[\s\S]*\n', 'COMMENT'),
    (r'[ \n\t]+', None),

    # Special characters
    (r'\;', 'TERMINATOR'),
    (r'\=', 'ASSIGN'),
    (r'\+', 'ADDITION'),
    (r'\-', 'SUBTRACTION'),
    (r'\*', 'MULTIPLICATION'),
    (r'\/', 'DIVISION'),

    # Identifiers & Integers
    (r'[a-z]\w*', 'IDENTIFIER'), # nom_planete1
    (r'-?\d+',       'NUMBER'),
]

...
```

Exercice 4 : Lexer optimisé

Question 1 A l'aide de l'outil `https://cyberzhg.github.io/toolbox/nfa2dfa`, entrez ces mots clés combinés par un "ou" (barre verticale) et visualisez l'automate de reconnaissance.

Solution:



Question 2 Coder un automate unique de reconnaissance des lexèmes précédents.

Solution: **[Ruby]** Note : le dernier lexème pose problème (attend un espace).

```ruby
class Token
  attr_accessor :kind,:value
  def initialize kind,value
    @kind,@value=kind,value
  end

  def to_s
```

```
    "(#{kind},#{value})"
  end
end

 class LexerFSM
  SPACE=' '
  def initialize txt
    @txt=txt.chars
    @value=[]
    @ptr=0
    @tokens=[]
    @state=:A
  end

  def tokenize
    while @txt.any?
      #puts "state #{@state}"
      case @state
      when :A
        case get_char
        when 'b'
          @state=:B
        when 'e'
          @state=:C
        when 'f'
          @state=:D
        when 'i'
          @state=:E
        when 'w'
          @state=:F
        when ' '
          @state=:A
        else
          @state=:parse_ident
        end
      when :B
        case get_char
        when 'e'
          @state=:G
        when 'o'
          @state=:H
        else
          @state=:parse_ident
        end
      when :C
        case get_char
        when 'n'
          @state=:I
        else
          @state=:parse_ident
        end
      when :D
```

```
  case get_char
  when 'o'
    @state=:J
  when 'u'
    @state=:K
  else
    @state=:parse_ident
  end
when :E
  case get_char
  when 'f'
    @state=:L
  when 'n'
    @state=:M
  else
    @state=:parse_ident
  end
when :F
  case get_char
  when 'h'
    @state=:N
  else
    @state=:parse_ident
  end
when :G
  case get_char
  when SPACE
    @state=:U
  else
    @state=:parse_ident
  end
when :H
  case get_char
  when 'd'
    @state=:P
  else
    @state=:parse_ident
  end
when :I
  case get_char
  when 'd'
    @state=:Q
  when 't'
    @state=:R
  else
    @state=:parse_ident
  end
when :J
  case get_char
  when 'r'
    @state=:S
  else
```

```
    @state=:parse_ident
  end
when :K
  case get_char
  when 'n'
    @state=:T
  else
    @state=:parse_ident
  end
when :L
  case get_char
  when SPACE
    @state=:U
  else
    @state=:parse_ident
  end
when :M
  case get_char
  when SPACE
    @state=:U
  when 'p'
    @state=:V
  else
    @state=:parse_ident
  end
when :N
  case get_char
  when 'i'
    @state=:W
  else
    @state=:parse_ident
  end
when :O
  case get_char
  when 'i'
    @state=:X
  else
    @state=:parse_ident
  end
when :P
  case get_char
  when 'y'
    @state=:Y
  else
    @state=:parse_ident
  end
when :Q
  case get_char
  when SPACE
    @state=:U
  else
    @state=:parse_ident
```

```
  end
when :R
  case get_char
  when 'i'
    @state=:Z
  else
    @state=:parse_ident
  end
when :S
  case get_char
  when SPACE
    @state=:U
  else
    @state=:parse_ident
  end
when :T
  case get_char
  when 'c'
    @state=:AA
  else
    @state=:parse_ident
  end
when :U #terminal for all keywords
  emit_token()
  @state=:A
when :V
  case get_char
  when 'u'
    @state=:AB
  else
    @state=:parse_ident
  end
when :W
  case get_char
  when 'l'
    @state=:AC
  else
    @state=:parse_ident
  end
when :X
  case get_char
  when 'n'
    @state=:AD
  else
    @state=:parse_ident
  end
when :Y
  case get_char
  when SPACE
    @state=:U
  else
    @state=:parse_ident
```

```
  end
when :Z
  case get_char
  when 't'
    @state=:AE
  else
    @state=:parse_ident
  end
when :AA
  case get_char
  when SPACE
    @state=:U
  else
    @state=:parse_ident
  end
when :AB
  case get_char
  when 't'
    @state=:AF
  else
    @state=:parse_ident
  end
when :AC
  case get_char
  when 'e'
    @state=:AG
  else
    @state=:parse_ident
  end
when :AD
  case get_char
  when SPACE
    @state=:U
  else
    @state=:parse_ident
  end
when :AE
  case get_char
  when 'y'
    @state=:AH
  else
    @state=:parse_ident
  end
when :AF
  case get_char
  when SPACE
    @state=:U
  else
    @state=:parse_ident
  end
when :AG
  case get_char
```

```
        when SPACE
          @state=:U
        else
          @state=:parse_ident
        end
      when :AH
        case get_char
        when SPACE
          @state=:U
        else
          @state=:parse_ident
        end
      when :parse_ident
        case get_char
        when SPACE
          emit_token :ident
          @state=:A
        when /[a-z]/
          @state=:parse_ident
        when nil
          break
        end
      end
    end
    return @tokens
  end

  def get_char
    char=@txt[@ptr]
    return if char.nil?
    @ptr+=1
    @value << char
    return char
  end

  def emit_token kind=nil
    @value=@value.delete ' '
    kind="kw_#{@value}" unless kind
    @tokens << token=Token.new(kind,@value)
    #puts "emit token : #{token}"
    @value=""
  end
end

code=IO.read(ARGV.first)
lexer=LexerFSM.new(code)
lexer.tokenize
```

Question 3 On cherche à mesurer le gain en performance du lexer optimisé par rapport au lexer naïf. Ecrire un script Python (ou autre) qui génère aléatoirement un fichier contenant un grand nombre des lexèmes précédent. Mesurer la performance respective des deux lexers

en lexèmes / seconde.

Solution [Ruby] On présente d'abord un code de lexer, qui n'utilise que des expressions régulières de manière apparemment "naïve".

```ruby
class Token
  attr_accessor :kind,:value
  def initialize kind,value
    @kind,@value=kind,value
  end

  def to_s
    "(#{kind},#{value})"
  end
end

class Lexer
  def initialize txt
    @txt=txt
    @tokens=[]
  end

  def tokenize
    while @txt.size>0
      case @txt
      when /\Awhile\b/
        @tokens << Token.new(:while,$&)
      when /\Aif\b/
        @tokens << Token.new(:if,$&)
      when /\Aelse\b/
        @tokens << Token.new(:else,$&)
      when /\Aend\b/
        @tokens << Token.new(:end,$&)
      when /\Aif\b/
        @tokens << Token.new(:if,$&)
      when /\Aend\b/
        @tokens << Token.new(:end,$&)
      when /\Aentity\b/
        @tokens << Token.new(:entity,$&)
      when /\Afunc\b/
        @tokens << Token.new(:func,$&)
      when /\Afor\b/
        @tokens << Token.new(:for,$&)
      when /\A[a-z]+/
        @tokens << Token.new(:ident,$&)
      when /\A\s+/
      else
        raise "parsing ERROR : #{@txt}"
      end
      @txt.delete_prefix!($&)
    end
```

```
    return @tokens
  end
end

code=IO.read(ARGV.first)
lexer=Lexer.new(code)
lexer.tokenize
```

Voici un script qui permet de générer un nombre de lexèmes voulus dans un fichier.

```
KEYWORDS=['if','while','input','func','for','in','body','begin','end','entity']

n=ARGV.first.to_i
tokens=[]
n.times do |i|
  tokens << KEYWORDS.sample
  if rand(2)==0 # add ~50% identifiers
    ident=""
    rand(10).times do
      ident+=(('a'..'z').to_a).sample
    end
    tokens << ident
  end
end

File.open("test.txt",'w'){|f| f.puts tokens}
```

En lançant les deux lexers en ligne de commande, précédé de "time", on peut mesurer les performances des deux programmes.

	lexer execution time (s)	
#tokens	optimized	naïve
10000	0.108	0.310
100000	0.518	18.881
200000	0.953	86.414

Notre lexer optimisé, basé sur l'explicitation d'un automate, est explonentiellement plus rapide que le lexer naïf !