R4.A.12 Automates et Langages

Thibault Godin ; Lucie Naert

IUT Vannes, Département informatique

Motivations

- Comment l'ordinateur comprend-il un programme ?
- Que puis-je faire en `JAVA` ?
- `JAVA` est-il plus puissant que `Python` ?

Théorie de la compilation

- Comment décrire un langage ?
- Comment comprendre/faire comprendre un programme ?
- Comment vérifier qu'un programme est correct ?

Langage ?

- C/JAVA/Python/SQL
- HTML–CSS/XML/JSON
- $\TeX$/docx/ods/JPG

- Français/Anglais/Finnois
- Lojban/Esperanto/Quenya
- math/logique
- ADN

Quel(s) point(s) commun(s)/ différence $\rightsquigarrow$ Quelle définition générale ?

Quelques langages qu'on aimerait décrire

- les mots français
- les mots allemands
- les palindromes (en binaire)
- les mots n'ayant que des a
- les mots ayant un nombre pair de lettres
- les mots n'ayant que des a et un nombre pair de lettres
- les mots n'ayant pas deux b de suite
- les mots n'ayant pas deux b de suite ni deux a
- les nombres pairs (en binaire)
- les multiples de 3 (en binaire)
- les nombres premiers
- les expressions arithmétiques
- les programmes Python

Comment étudier un langage ?

Décrire :

ensembles,

expression régulières

Calculer :

programmes,

ordinateur,

automates

Produire :

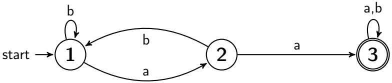
compilateur de compilateur,

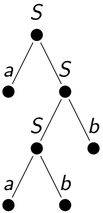
Arbre

Grammaire

$$\{\boldsymbol{w} \in \Sigma^* | \exists k \in \mathbb{N} |\boldsymbol{w}| = 2k\}$$

$$\varepsilon + ((\mathtt{a}\Sigma + (\mathtt{b}\Sigma))^*$$





$$S \rightarrow SS|(S)|\varepsilon$$

Langages informatiques

Sujet précédent

9. Composants de plus haut niveau

Sujet suivant

La bibliothèque standard

Cette page

Signalement de bogue
Montrer le code source

10. Spécification complète de la grammaire

Ceci est la grammaire complète de Python, issue directement de la grammaire utilisée pour générer l'analyseur syntaxique CPython (voir Grammar/python.gram). La version ci-dessous ne comprend pas les détails relatifs à la génération de code et la reprise sur erreur.

The notation is a mixture of EBNF and PEG. In particular, & followed by a symbol, token or parenthesized group indicates a positive lookahead (i.e., is required to match but not consumed), while ! indicates a negative lookahead (i.e., is required _not_ to match). We use the | separator to mean PEG's "ordered choice" (written as / in traditional PEG grammars). See PEP 617 for more details on the grammar's syntax.

```
# PEG grammar for Python


file: [statements] ENDMARKER
interactive: statement_newline
eval: expressions NEWLINE* ENDMARKER
func_type: '(' [type_expressions] ')' '->' expression NEWLINE* ENDMARKER
fstring: star_expressions

# type_expressions allow */** but ignore them
type_expressions:
    | ','.expression+ ',' '*' expression ',' '**' expression
    | ','.expression+ ',' '*' expression
    | ','.expression+ ',' '**' expression
    | '*' expression ',' '**' expression
    | '*' expression
    | '**' expression
    | ','.expression+

statements: statement+
statement: compound_stmt  | simple_stmts
statement_newline:
    | compound_stmt NEWLINE
    | simple_stmts
    | NEWLINE
    | ENDMARKER
simple_stmts:
    | simple_stmt !';' NEWLINE  # Not needed, there for speedup
    | ';'.simple_stmt+ [';'] NEWLINE
# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
simple_stmt:
    | assignment
    | star_expressions
    | return_stmt
    | import_stmt
    | raise_stmt
    | 'pass'
    | del_stmt
    | yield_stmt
    | assert_stmt
    | 'break'
```

https://docs.python.org/fr/3/reference/grammar.html

Langages informatiques

" This PEP proposes replacing the current LL(1)-based parser of CPython with a new PEG-based parser. This new parser would allow the elimination of multiple "hacks" that exist in the current grammar to circumvent the LL(1)-limitation. It would substantially reduce the maintenance costs in some areas related to the compiling pipeline such as the grammar, the parser and the AST generation. The new PEG parser will also lift the LL(1) restriction on the current Python grammar."
`https://www.python.org/dev/peps/pep-0617/`

citation sur la manière de décrire le parseur "We did not seriously consider alternative ways to implement the new parser [...]"

Langages informatiques

Chapter 18. Syntax

This chapter presents a grammar for the Java programming language.

The grammar presented piecemeal in the preceding chapters (§2.3) is much better for exposition, but it is not well suited as a basis for a parser. The grammar presented in this chapter is the basis for the reference implementation. Note that it is not an LL(1) grammar, though in many cases it minimizes the necessary look ahead.

The grammar below uses the following BNF-style conventions:

- *[x]* denotes zero or one occurrences of *x*.
- *{x}* denotes zero or more occurrences of *x*.
- *(x | y)* means one of either *x* or *y*.

```
Identifier:
    IDENTIFIER

QualifiedIdentifier:
    Identifier { . Identifier }

QualifiedIdentifierList:
    QualifiedIdentifier { , QualifiedIdentifier }
```

```
CompilationUnit:
    [[Annotations] package QualifiedIdentifier ;]
                                {ImportDeclaration} {TypeDeclaration}

ImportDeclaration:
    import [static] Identifier { . Identifier } [. *] ;

TypeDeclaration:
    ClassOrInterfaceDeclaration
    ;

ClassOrInterfaceDeclaration:
    {Modifier} (ClassDeclaration | InterfaceDeclaration)

ClassDeclaration:
    NormalClassDeclaration
    EnumDeclaration

InterfaceDeclaration:
    NormalInterfaceDeclaration
    AnnotationTypeDeclaration



NormalClassDeclaration:
    class Identifier [TypeParameters]
                                [extends Type] [implements TypeList] ClassBody

EnumDeclaration:
    enum Identifier [implements TypeList] EnumBody

NormalInterfaceDeclaration:
    interface Identifier [TypeParameters] [extends TypeList] InterfaceBody

AnnotationTypeDeclaration:
    @ interface Identifier AnnotationTypeBody
```

```
Type:
    BasicType {[]}
```

https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html

Langage formel

Σ : alphabet $\rightsquigarrow$ ensemble *fini*

$\boldsymbol{u}, \boldsymbol{v}$: mots sur l'alphabet Σ $\rightarrow$ suites *finies* d'éléments de Σ

$$\boldsymbol{u} : \begin{array}{rcl} \{1, \ldots, \ell\} & \longrightarrow & \Sigma \\ k & \longmapsto & \boldsymbol{u}[k] \end{array}$$

- $\ell = |\boldsymbol{u}|$: longueur du mot
- un *unique* mot de longueur 0 : ε

- ensemble de tous les mots : Σ^*
- deux mots $\boldsymbol{u}$ et $\boldsymbol{v}$ sont égaux si
  - même longueur ($|\boldsymbol{u}| = |\boldsymbol{v}|$)
  - $\forall i \in [\![1 : \ell]\!], \boldsymbol{u}[i] = \boldsymbol{v}[i]$

Définition

un *langage* (sur Σ) est un ensemble de mots de Σ.

L langage $\iff L \subset \Sigma^* \iff L \in \mathcal{P}(\Sigma^*)$

✎ Donner quelques exemples de mots et de langages sur les alphabets $\{0, 1\}, \{a, b, c\}$ et ASCII.

Construire de nouveaux mots

plus longs :

la *concaténation* est l'opération sur les mots :

$$\begin{array}{rccl} . : & \Sigma^* \times \Sigma^* & \longrightarrow & \Sigma^* \\ & (\boldsymbol{u}, \boldsymbol{v}) & \longmapsto & \boldsymbol{u}.\boldsymbol{v} = \boldsymbol{w} \end{array}$$

$$\text{t.q. } |\boldsymbol{w}| = |\boldsymbol{u}| + |\boldsymbol{v}| \text{ et } \forall i \in [\![1 : |\boldsymbol{w}|]\!], \boldsymbol{w}[i] = \begin{cases} \boldsymbol{u}[i] & \text{si } i \leq |\boldsymbol{u}| \\ \boldsymbol{v}[i - |\boldsymbol{u}|] & \text{sinon} \end{cases}$$

Donner la concaténation des mots *aab* et *bb*.

Que vaut la concaténation de $\boldsymbol{u}$ avec le mot vide ε ?

rmq : on omet souvent le "." dans l'écriture $\boldsymbol{u}.\boldsymbol{v}$

On note $\boldsymbol{u}^p = \boldsymbol{u}.\boldsymbol{u} \ldots \boldsymbol{u}$. (la concaténation de p mots $\boldsymbol{u}$)

On a $\boldsymbol{u}^{p+q} = \boldsymbol{u}^p.\boldsymbol{u}^q$ (sous réserve de poser $\boldsymbol{u}^0 = \varepsilon$).

Construire de nouveaux mots

plus courts :

Soient $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{x}, \boldsymbol{y}$ des mots de Σ^*

- $\boldsymbol{v}$ est un *facteur* de $\boldsymbol{u}$ si : $\boldsymbol{u} = \boldsymbol{xvy}$
- $\boldsymbol{v}$ est un *préfixe* de $\boldsymbol{u}$ si : $\boldsymbol{u} = \boldsymbol{vy}$
- $\boldsymbol{v}$ est un *suffixe* de $\boldsymbol{u}$ si : $\boldsymbol{u} = \boldsymbol{xv}$

On note $\text{pref}_{\boldsymbol{u}}$ l'ensemble des préfixes du mot $\boldsymbol{u}$ et $\text{suff}_{\boldsymbol{u}}$ l'ensemble des suffixes.

Le miroir ou transposé $\boldsymbol{u}^R$ du mot $\boldsymbol{u} = u[1].u[2] \cdots u[\ell]$ est défini par : $\boldsymbol{u}^R = u[\ell].u[\ell - 1] \cdots u[1]$.

Un palindrome est un mot égal à son miroir

✍ Donner les préfixes et facteurs du mots 001011. Que sont les préfixes du miroir de $\boldsymbol{u}$?

Construire de nouveaux langages

Opérations ensemblistes :

Si L et L' sont des langages, se sont en particulier des *ensembles* $\rightsquigarrow$ opérations usuelles :

- $L \cup L' = L + L'$
- $L \cap L'$
- ${}^{c}L = \overline{L}$
- $L \setminus L' = L \cap \overline{L'}$
- $L \Delta L' = (L \cup L') \setminus (L \cap L')$

Construire de nouveaux langages

Concaténation :

$$L.L' = \{\boldsymbol{w} \in \Sigma^* | \exists \boldsymbol{u} \in L, \exists \boldsymbol{v} \in L', \boldsymbol{w} = \boldsymbol{u}.\boldsymbol{v}\} = \{\boldsymbol{u}.\boldsymbol{v} | \boldsymbol{u} \in L \boldsymbol{v} \in L'\}$$

Préfixe/Suffixe/Facteur :

$$\mathrm{Pref}(L) = \{\boldsymbol{u} \in \Sigma^* | \exists \boldsymbol{w} \in L, \exists \boldsymbol{v} \in \Sigma^*, \boldsymbol{w} = \boldsymbol{u}.\boldsymbol{v}\}$$

Récapitulons :

alphabet Σ, mots $\boldsymbol{w} \in \Sigma^*$

$\cup, \cap, \overline{}, .$

Que peut-on faire comme langages ?

$\rightsquigarrow$ Langages ayant un nombre fini de mot $\mathcal{L}_f$ est la classe des langages *finis*

✎ Donner un exemple de langage fini sur $\{a, b\}$

$L \in \mathcal{L}_f \rightsquigarrow \overline{L} \rightsquigarrow$ Langages contenant tous les mots sauf un nombre fini $\overline{\mathcal{L}_f}$ est la classe des langages *co-finis*

✎ Donner un exemple de langage co-fini sur $\{a, b\}$

✎ Et pour les autres opérations ?

Trois premières classes de langages

	$\mathcal{L}_f$	$\overline{\mathcal{L}_f}$	$\mathcal{L}_f \cup \overline{\mathcal{L}_f}$
Union	clos	clos [1]	clos
Intersection	clos [2]	clos	clos
Concaténation	clos	clos	non-clos
Complément	non-clos	non-clos	clos
Préfixe	clos	clos [3]	clos
Suffixe	clos	clos	clos
Miroir	clos	clos	clos

$\rightsquigarrow$ pas forcément les langages très intéressants ...

Montrer que $\mathcal{L}_f \cup \overline{\mathcal{L}_f}$ n'est pas clos pour la concaténation

$\{a\}.\Sigma^*$

1. en fait, l'union de tout langage avec un langage cofini est cofinie
2. en fait, l'intersection de tout langage avec un langage fini est finie
3. en fait, pour tout L cofini, $\text{Pref}_L = \Sigma^*$

Les langages que l'on décrit

Avec $\mathcal{L}_f \cup \overline{\mathcal{L}_f}$:

- les mots français
- ~~les mots allemands~~
- ~~les palindromes (en binaire)~~
- ~~les mots n'ayant que des a~~
- ~~les mots ayant un nombre pair de lettres~~
- ~~les mots n'ayant que des a et un nombre pair de lettres~~
- ~~les mots n'ayant pas deux b de suite~~
- ~~les mots n'ayant pas deux b de suite ni deux a~~
- ~~les nombres pairs (en binaire)~~
- ~~les multiples de 3 (en binaire)~~
- ~~les nombres premiers~~
- ~~les expressions arithmétiques~~
- ~~les programmes Python~~

Une (presque) nouvelle opération

rmq : une concaténation de programmes Python reste un programme Python

$\rightsquigarrow$ "autant de concaténations que l'on souhaite" :
fermeture de Kleene (ou étoile) d'un langage L :
$L^* = \bigcup_{i=0}^{\infty} L^i \quad ; \quad L^+ = L.L^*$

rmq : $L^0 = \{\varepsilon\}$

L^* contient tous les mots constructible en concaténant un nombre fini (éventuellement réduit à zéro) de mots de L

permet d'exprimer de manière formelle (et compacte) des langages complexes.

Décrire le langage suites de 0 et de 1 contenant la séquence 111
$\{0,1\}^*.\{111\}.\{0,1\}^*$.

Fermeture pour l'étoile

	$\mathcal{L}_f$	$\overline{\mathcal{L}_f}$	$\mathcal{L}_f \cup \overline{\mathcal{L}_f}$
Étoile	non-clos	clos [4]	non-clos

$\rightsquigarrow$ classe de langages close pour l'étoile : les langages *rationnels*, *i.e.* la plus petite classe $\mathcal{L}_{rat}$ telle que :

- $\emptyset \in \mathcal{L}_{rat}$
- $\forall a \in \Sigma, \{a\} \in \mathcal{L}_{rat}$
- $\mathcal{L}_{rat}$ est clos pour l'union, la concaténation et l'étoile

	$\mathcal{L}_{rat}$
Union	clos
Intersection	?
Concaténation	clos
Complément	?
Préfixe	clos
Suffixe	?
Miroir	?
Étoile	clos

4. On utilise $(\Sigma^* \setminus F) \cup (\Sigma^* \setminus F') = \Sigma^* \setminus (F \cap F')$

$Rat = Reg$

représentation adaptée plus compacte : les *expressions régulières* (`regex`)

- $\emptyset \leftrightarrow \varnothing$
- $\{a\} \leftrightarrow \mathtt{a}$
- $L \cup L' \leftrightarrow \mathtt{e} + \mathtt{e}' \leftrightarrow \mathtt{e}|\mathtt{e}'$
- $L.L' \leftrightarrow \mathtt{ee}'$
- $L^* \leftrightarrow \mathtt{e}^*$

Décrire les langages suivants :

- $a|b^*$
- $(a|b)^*$
- $ab^*(c|\varepsilon)$
- $(0|(1(01^*0)^*1))^*$

Définition-Théorème

Les langages décrits par les expressions régulières sont exactement les langages rationnels

En pratique : `grep`