Décrire les langages de programmation
Retour sur la hiérarchie des langages
Grammaires : Séance 3

Marie-Laure Potet

Grenoble INP-Ensimag

March 3, 2022

Summary

Grammaire hors-contexte

Grammaires (Langages) hors-contexte :

⇒ Une classe qui fournit un bon compromis Expressivité/Décidabilité :

- Permet de décrire la syntaxe des langages de programmation
- Des algorithmes efficaces de reconnaissance (TL2)

Summary

Langages de programmation

$t[i] = 3/y$;

- Lexicographie : les mots du langage
  - par expression régulière ou grammaire. Ex : $(0 + \ldots + 9)^+$
- Syntaxe : les programmes et constructions intermédiaires (fonction, expression, instruction . . .)
  - par grammaire hors-contexte. Ex : $Affect \rightarrow var = exp$;
- Sémantique statique : les règles à vérifier non prises en compte par la syntaxe (typage, déclaration des identificateurs . . .)
  - Ex : *exp* doit être un sous-type de *var* (non hors-contexte)

⇒ ce que vérifie un compilateur

Langages de programmation (2)

- Sémantique à l'exécution : ce qui dépend des valeurs (comportements nominaux et erronés . . .). Ex :
  - cas nominal : la valeur de *exp* est rangée en mémoire à l'adresse dénotée par *var*
  - erreur si erreur dans l'évaluation de *exp* ou *var* ou si sous-typage non respecté.

⇒ ce qu'implémente un compilateur
⇒ Interpréteur : analyse + évaluation dans la foulée

Python :

- Une grammaire : https://docs.python.org/3/reference
- des analyseurs statiques : par exemple mypy, Pyre (Facebook) compatible PEP 484

Mais lorsqu'on travaille sur des millions de lignes de code, que ce soit avec Python ou tout autre langage, des difficultés se présentent. Notamment, pour ce langage dynamiquement typé, l'absence de typage statique rend ardue la modification d'une large base de code existante, avec le risque d'introduire des erreurs. De plus l'absence de typage statique rend difficile la création d'outils de recherche de code, de saisie semi-automatique, de navigation et de refactoring de qualité.

Exemple - Norme du langage C

6.8.4.1 The if statement
Syntax

```
selection-statement ::=
          if ( expression ) statement
       |  if ( expression ) statement else statement
```

Constraints

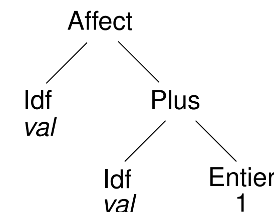
```
The controlling expression of an if statement shall have
scalar type.
```

Semantics

```
In both forms, the first substatement is executed if the
expression compares unequal to 0.
In the else form, the second substatement is executed if
the expression compares equal to 0.
```


Définition guidée par la syntaxe

- **Typage** : calcul des types des feuilles vers la racine
- **Génération de code** : produire un programme dans un langage cible donné, à partir de l'arbre abstrait décoré et optimisé.




```
LOAD #1, R0
ADD @val, R0
STORE R0, @val
```

Arbre abstrait du programme

`@val` : adresse de l'objet val
`LOAD` : chargement dans un registre
`STORE` : chargement à une adresse

⇒ Non ambiguïté = un arbre = une interpétation (-:

Summary

Ambiguïté : rappel

Définition: (Grammaire ambiguë)

Une grammaire G est dite ambiguë ssi il existe au moins un mot de $L(G)$ pour lequel il existe plusieurs dérivations canoniques.

$\Rightarrow$ plusieurs dérivations canoniques = plusieurs arbres de dérivation

- Il existe des langages intrinsèquement ambigus (qu'on évite lorsqu'on construit un langage)
- il existe des conditions suffisantes pour montrer la non-ambiguïté d'une grammaire (un arbre unique pour chaque mot du langage)

Condition suffisante pour qu'une grammaire ne soit pas ambiguë

Rappel :

Théorème:

Une condition suffisante pour qu'une grammaire G ne soit pas ambiguë est que les deux propositions ci-dessous soient vérifiées :

1. *pour tout couple de règles* $(A \rightarrow \alpha, A \rightarrow \beta)$ *de G tel que* $\alpha \neq \beta$, $L(\alpha) \cap L(\beta) = \emptyset$;
2. *pour toute règle de la forme* $A \rightarrow X_1X_2...X_n$, *où* $X_i \in V_T \cup V_N$, $\forall w \in V_T^*$ *tel que* $X_1X_2...X_n \Longrightarrow^* w$, $\exists!(w_1, w_2, ...w_n)$ *tel que* $w_i \in V_T^*$, $w = w_1w_2...w_n$ *et* $\forall i$, $X_i \Longrightarrow^* w_i$.

Les expressions arithmétiques

Une grammaire d'expressions arithmétiques ambiguë:

$V_T = \{0, \ldots, 9, +, -, *, (,)\}$

$$Exp \rightarrow Num \mid Exp + Exp \mid Exp - Exp \mid Exp * Exp \mid (Exp)$$
$$Num \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

2 arbres de dérivation pour $2 + 3 * 5$ et $2 - 3 - 5$ (et donc potentiellement 2 interprétations différentes)

Grammaires pour les expressions

⇒ Une technique de description des expressions arithmétiques prenant en compte les priorités et les propriétés d'associativité :

1. Stratification des expressions en fonction de la priorité des opérateurs
   - $X_0, X_1, \ldots, X_n$: du - au + prioritaire
2. $X_i \rightarrow X_{i+1}$ pour $i \in 0..n-1$
3. Prise en compte des propriétés d'associativité
   1. $X_i \rightarrow X_{i+1}\ op_i\ X_{i+1}$ si op_i non associatif (ex = ou <)
   2. $X_i \rightarrow X_i\ op_i\ X_{i+1}$ si op_i associatif à gauche (ex -)
   3. $X_i \rightarrow X_{i+1}\ op_i\ X_i$ si op_i associatif à droite (ex +)
4. les parenthèses : toujours opérateur unaire de plus forte priorité

Grammaire non ambiguë des expressions arithmétiques

Généralement $X_0 = Exp$, $X_1 = Terme$, $\ldots X_n = Facteur$

$V_T = \{0, \ldots, 9, +, -, *, (,)\}$

$$
\begin{array}{lcl}
Exp & \rightarrow & Exp + Terme \mid Exp - Terme \mid Terme \\
Terme & \rightarrow & Terme * Facteur \mid Facteur \\
Facteur & \rightarrow & Num \mid (Exp) \\
Num & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

⇒ arbre de dérivation pour $2 + 3 * 5$ et $2 - 3 - 5$
⇒ preuve de non ambiguïté

Summary

BNF étendue

Dans la pratique notation étendue (BNF étendue pour Backus Normal Form). Partie droite de règle de la forme :

- x avec $x \in V_T$
- A avec $A \in V_N$
- ϵ
- $[e]$ avec e une partie droite de règle (partie optionnelle)
- $\{e\}$ ou $(e)^*$ avec e une partie droite de règle (partie itérative)
- $e_1 e_2$ avec e_1 et e_2 des parties droites de règle

Exemple :

```
if_stmt    ::=  "if" expression ":" suite
                ("elif" expression ":" suite)*
                ["else" ":" suite]

suite     ::=   stmt_list NEWLINE
              | NEWLINE INDENT statement+ DEDENT
```

Exemple de norme

Syntaxe du langage python :
`https://docs.python.org/3/reference`

Priorité des opérateurs :
`https://docs.python.org/3/reference/expressions.html#operator-precedence`

⇒ Quelques morceaux choisis de la norme Python

Lexicographie (1)

2.4.5. Integer literals

Integer literals are described by the following lexical definitions:

```
integer      ::=  decinteger | bininteger | octinteger | hexinteger
decinteger   ::=  nonzerodigit (["_"] digit)* | "0"+ (["_"] "0")*
bininteger   ::=  "0" ("b" | "B") (["_"] bindigit)+
octinteger   ::=  "0" ("o" | "O") (["_"] octdigit)+
hexinteger   ::=  "0" ("x" | "X") (["_"] hexdigit)+
nonzerodigit ::=  "1"..."9"
digit        ::=  "0"..."9"
bindigit     ::=  "0" | "1"
octdigit     ::=  "0"..."7"
hexdigit     ::=  digit | "a"..."f" | "A"..."F"
```

There is no limit for the length of integer literals apart from what can be stored in available memory.

Underscores are ignored for determining the numeric value of the literal. They can be used to group digits for enhanced readability. One underscore can occur between digits, and after base specifiers like `0x`.

Note that leading zeros in a non-zero decimal number are not allowed. This is for disambiguation with C-style octal literals, which Python used before version 3.0.

Lexicographie (2)

2.4.6. Floating point literals

Floating point literals are described by the following lexical definitions:

```
floatnumber   ::=  pointfloat | exponentfloat
pointfloat    ::=  [digitpart] fraction | digitpart "."
exponentfloat ::=  (digitpart | pointfloat) exponent
digitpart     ::=  digit (["_"] digit)*
fraction      ::=  "." digitpart
exponent      ::=  ("e" | "E") ["+" | "-"] digitpart
```

Note that the integer and exponent parts are always interpreted using radix 10. For example, `077e010` is legal, and denotes the same number as `77e10`. The allowed range of floating point literals is implementation-dependent. As in integer literals, underscores are supported for digit grouping.

Quelques instructions (1)

8.1. The `if` statement¶

The `if` statement is used for conditional execution:

```
if_stmt ::=  "if" assignment_expression ":" suite
             ("elif" assignment_expression ":" suite)*
             ["else" ":" suite]
```

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section Boolean operations for the definition of true and false); then that suite is executed (and no other part of the `if` statement is executed or evaluated). If all expressions are false, the suite of the `else` clause, if present, is executed.

Quelques instructions (2)

8.3. The `for` statement

The `for` statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```
for_stmt ::=  "for" target_list "in" expression_list ":" suite
              ["else" ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The suite is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments (see Assignment statements), and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a `StopIteration` exception), the suite in the `else` clause, if present, is executed, and the loop terminates.

Quelques expressions

6.11. Boolean operations

```
or_test  ::=  and_test | or_test "or" and_test
and_test ::=  not_test | and_test "and" not_test
not_test ::=  comparison | "not" not_test
```

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, `None`, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true. User-defined objects can customize their truth value by providing a `__bool__()` method.

The operator `not` yields `True` if its argument is false, `False` otherwise.

The expression `x and y` first evaluates *x*; if *x* is false, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

The expression `x or y` first evaluates *x*; if *x* is true, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

Note that neither `and` nor `or` restrict the value and type they return to `False` and `True`, but rather return the last evaluated argument. This is sometimes useful, e.g., if `s` is a string that should be replaced by a default value if it is empty, the expression `s or 'foo'` yields the desired value. Because `not` has to create a new value, it returns a boolean value regardless of the type of its argument (for example, `not 'foo'` produces `False` rather than `''`.)

Table des priorités

Operator	Description
`:=`	Assignment expression
`lambda`	Lambda expression
`if` – `else`	Conditional expression
`or`	Boolean OR
`and`	Boolean AND
`not x`	Boolean NOT
`in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `!=`, `==`	Comparisons, including membership tests and identity tests
`|`	Bitwise OR
`^`	Bitwise XOR
`&`	Bitwise AND
`<<`, `>>`	Shifts
`+`, `-`	Addition and subtraction
`*`, `@`, `/`, `//`, `%`	Multiplication, matrix multiplication, division, floor division, remainder [5]
`+x`, `-x`, `~x`	Positive, negative, bitwise NOT
`**`	Exponentiation [6]
`await x`	Await expression

Summary

Rappel

Classification des grammaires (Hiérarchie de Chomsky) :

- Grammaire générale : règles de la forme $u \rightarrow v$ avec $u \neq \epsilon$.
- Grammaire sous-contexte : règles de la forme $u \rightarrow v$ avec $|u| \leq |v|$ et $u \neq \epsilon$.
- Grammaire hors-contexte : règles de la forme $A \rightarrow w$
- Grammaire règulière : règles de la forme $A \rightarrow \epsilon$ et $A \rightarrow aB$

avec $a \in V_T$, $A, B \in V_N$ et $u, v, w \in (V_T \cup V_N)^*$

$\Rightarrow$ on va voir : grammaire régulière et automate, langage hors-contexte comme une sous-classe des sous-contexte, propriétés et lemmes de fermeture.

Grammaire régulière et automate

1) De la grammaire à l'automate :

Soit $G = (V_T, V_N, S, R)$ une grammaire régulière. On construit l'automate A tel que $L(A) = L(G)$ de la manière suivante :

- $A = (V_T, V_N, S, F, \delta)$ et
- $X \in F$ si et seulement si $X \rightarrow \epsilon \in R$
- $(X, a, Y) \in \delta$ si et seulement si $X \rightarrow aY \in R$

Ex : les chaines sur $\{a, b\}$ ayant un nombre pair de a.

Grammaire régulière et automate (2)

2) De l'automate à la grammaire :

Soit $A = (V, Q, q_0, F, \delta)$. On construit G telle que $L(A) = L(G)$ de la manière suivante :

- $G = (V, Q, q_0, R)$ et
- $q \rightarrow \epsilon \in R$ si et seulement si $q \in F$
- $p \rightarrow aq \in R$ si et seulement si $(p, a, q) \in \delta$

Ex : les chaines sur $\{a, b\}$ ayant un nombre pair de a.

Preuve- Sens 1

Montrons pour tout w : $S \Longrightarrow^* w$ ssi $\delta^*(S, w) \cap F \neq \emptyset$ et plus généralement :

$$N \Longrightarrow^* w \text{ ssi } \delta^*(N, w) \cap F \neq \emptyset$$

Par $\delta^*(N, w)$ on désigne l'ensemble des états accessibles à partir de N par un chemin d'étiquette w.

On fait une preuve par récurrence sur $\Longrightarrow^n$:
1) Soit $n = 1$. Par la forme des règles on a nécessairement $N \Longrightarrow^1 \epsilon$ (application d'une règle de la forme $N \rightarrow \epsilon$) et donc $N \in F$ (par définition de F) et donc $\delta^*(N, \epsilon) \cap F \neq \emptyset$.
2) Soit n avec n > 2. $N \Longrightarrow^n w$ ssi il existe une règle $N \rightarrow aB$ et $w = av$ et $B \Longrightarrow^{n-1} v$, i.e. :

$$N \Longrightarrow^1 aB \Longrightarrow^{n-1} av = w$$

On a alors par hypothèse de récurrence : $\delta^*(B, v) \cap F \neq \emptyset$. Par construction de la relation de transition δ de l'automate A on a $B \in \delta(N, a)$. Donc $\delta^*(N, av)$ mène à un état final.

Preuve- Sens 2

On peut partir d'un automate déterministe. On raisonne sur la longueur du chemin.

On montre donc pour tout état q et toute chaîne w :

$$\delta^n(q, w) \in F \text{ ssi } q \Longrightarrow^* w$$

Preuve par récurrence sur n avec cas de base $n = 0$.

A FAIRE

Langage hors-contexte et sous-contexte

Rappels :

- une grammaire hors-contexte sans ϵ-règle (i.e. pas de règles de la forme $A \rightarrow \epsilon$) est une grammaire sous-contexte.
- Grammaire sous-contexte : on peut admettre $Z \rightarrow \epsilon$ avec Z l'axiome et aucune occurrence de Z en partie droite de règle.

Définition:

Soit $G = (V_T, V_N, S, R)$ une grammaire hors-contexte. On peut construire une grammaire G' sans ϵ-règle telle que :

$$L(G') = L(G) - \{\epsilon\}$$

$\Rightarrow$ donc tout langage hors-contexte peut être décrit par une grammaire sous-contexte.

Elimination des ϵ-règles

$\Rightarrow$ algorithme en deux étapes :

Etape 1 : Calcul de l'ensemble E des non-terminaux produisant ϵ :

$$E = \{X \in V_N \mid X \Longrightarrow^* \epsilon\}$$

Calcul par itération :

- $E_0 = \{X \mid X \rightarrow \epsilon \in R\}$
- $E_{k+1} = E_k \cup \{X \mid X \rightarrow X_1 ... X_N \in R \text{ et } X_i \in E_k\}$
- Arrêt lorsque $E = E_n = E_{n+1}$.

Exemple : $S \rightarrow aSb \mid A \mid B \quad A \rightarrow aA \mid \epsilon \quad B \rightarrow bB \mid b$

Résultat: $E = \{A, S\}$

Elimination des ϵ-règles (2)

Etape 2 : recopie des règles en éliminant les combinaisons possibles des symboles de E dans les règles et en enlevant les ϵ-règles.
Exemple avec $E = \{A, S\}$:

$S \rightarrow aSb$	produit	$S \rightarrow aSb$	✓
		$S \rightarrow ab$	✓
$S \rightarrow A$	produit	$S \rightarrow A$	✓
		$S \rightarrow \epsilon$	✗

Grammaire G : $S \rightarrow aSb \mid A \mid B \quad A \rightarrow aA \mid \epsilon \quad B \rightarrow bB \mid b$ et $E = \{A, S\}$:
Grammaire G' : $S \rightarrow aSb \mid ab \mid A \mid B \quad A \rightarrow aA \mid a \quad B \rightarrow bB \mid b$
$\Rightarrow$ On a $L(G') = L(G) - \{\epsilon\}$.

Summary

Propriétés des langages réguliers

Langages réguliers :

- fermés par opérations ensemblistes (union, intersection, complémentaire) et substitution régulière.
- lemme de l'étoile (ou de la pompe) : si L est régulier et non fini alors il existe une boucle productive (non étiquetée par ϵ) dans l'automate reconnaissant L. Donc il existe un n tel que pour tout mot r de L de taille supérieure ou égale à n
  $\exists\ u, v, w$ tels que :
  1. $r = uvw$
  2. $|uv| \leq n$
  3. $v \neq \epsilon$
  4. $uv^k w \in L, \forall k \in N$.

Propriétés des langages hors-contexte

Langages hors-contexte :

- fermés par union, concaténation et itération (mais pas par complémentaire (exercice 4 feuille 1 Partie 2) ni par intersection)
- fermés par substitution hors-contexte
- lemme de la pompe pour les hors-contexte : si L est hors-contexte et non fini alors il existe un parenthésage (ne produisant pas ϵ) dans la grammaire engendrant L. Donc il existe un n tel que pour tout mot r de L de taille supérieure ou égale à n
  $\exists\ x, u, v, w, y$ tels que :
  1. $r = xuvwy$
  2. $|uvw| \leq n$
  3. $uw \neq \epsilon$
  4. $xu^k vw^k y \in L, \forall k \in N$

Principe de la preuve (1)

Soit L un langage hors-contexte infini. Alors il existe une grammaire hors-contexte vérifiant les propriétés suivantes :

1. pas de ϵ-règle
2. pas de règle de la forme $A \rightarrow\ B$ (sinon on remplace)
3. tous les éléments de V_N sont productifs (i.e. $A \Longrightarrow^* w$ avec $w \in V_T^*$)
4. tous les éléments de V_N sont accessibles (i.e. $\exists x, y$ tels que $S \Longrightarrow^* xAy$)
5. soit m la taille maximale d'une partie droite de règle

$\Rightarrow$ les mots de taille strictement supérieure à $m^{|V_N|}$ ont un arbre de dérivation de hauteur strictement supérieure à $|V_N|$ (symbole auto-imbriqué).

Principe de la preuve (2)

Si L est infini alors :

1. il existe un symbole auto-imbriqué $A \Longrightarrow^* uAw$ avec $uw \neq \epsilon$ (pas de 1-règle ni de ϵ-règle)
2. accessibles + productifs garantit :
   1. $S \Longrightarrow^* xAy$ avec $x, y \in V_T^*$
   2. $A \Longrightarrow^* v$ avec $v \in V_T^*$

On déduit :

$$S \Longrightarrow^* xAy \Longrightarrow^* xuAwy \Longrightarrow^* xuvwy$$

Mais aussi :

$$\begin{aligned} S \Longrightarrow^* xAy &\Longrightarrow^* xuAwy \\ &\Longrightarrow^* xu^2Aw^2y \Longrightarrow^* \dots \Longrightarrow^* xu^nAw^ny \\ &\Longrightarrow^* xu^nvw^ny \end{aligned}$$

Exemple

Montrons que le langage L des mots de la forme $a^ib^ic^i$ n'est pas hors-contexte.
On suppose qu'il existe l tel que $a^lb^lc^l = xuvwy$ et donc xu^kvw^ky dans L quelque soit k.

1. x de la forme a^p avec $p < l$ (sinon on répéterait des b et des c sans des a)
2. y de la forme c^r avec $r < l$ (sinon on répéterait des a et des b sans des c)
3. u de la forme $a^{p'}$ (sinon on répéterait une séquence de a suivie de b (u^n) et donc on aurait des a après des b)
4. idem pour w (w de la forme $c^{r'}$)
5. donc ni u ni w ne contiennent de b. On sort du langage : on répéterait des a et des c sans b

CQFD