

Assembleur Python

PROJET D'INFORMATIQUE

ANNÉE UNIVERSITAIRE 2024–2025

PHELMA 2^{ÈME} ANNÉE SEI SICOM

Version 0.3

Signature : 48da70613c4d850a

AUTEUR: FRANÇOIS CAYRE

RELECTEURS: NICOLAS CASTAGNÉ, BRICE COLOMBIER, MICHEL DESVIGNES

Table des matières

I Outils pour l'analyse de code source	11
1 Langages et méta-langage EBNF	15
1.1 Notion(s) de grammaire	15
1.2 Forme de Backus-Naur étendue (EBNF)	16
1.3 Exemple : les expressions arithmétiques en EBNF	17
1.4 Analyse lexicale vs. analyse syntaxique	17
2 Analyse lexicale	19
3 Expressions régulières	21
3.1 Syntaxe des expressions régulières en EBNF	22
3.2 Exemples	23
3.3 Contre-exemples	24
3.4 Lecture (<i>parsing</i>) et représentation en mémoire d'une expression régulière	25
3.5 Mécanisme pour reconnaître une expression régulière	26
3.6 Extensions du mécanisme	30
4 Un analyseur lexical	37
4.1 Structure de lexème	38
4.2 Fichier de définitions de types de lexèmes	39
4.3 Analyseur lexical	41
5 Analyse syntaxique	43
6 Descente récursive	45
6.1 Idée générale	45
6.2 Un léger ajout à EBNF	45
6.3 Implantation d'une grammaire $LL(1)$ en C	46
6.4 Exemple : vérification d'expressions arithmétiques	49
7 Un analyseur syntaxique	53
7.1 Des objets de syntaxe abstraite	53
7.2 Exemple : syntaxe abstraite des expressions arithmétiques	54
7.3 Analyseur syntaxique	56

II	Assembleur Python	59
8	Sur le fonctionnement de Python	63
8.1	Boucle REPL de Python	63
8.2	Rappel : compilation, assemblage et édition des liens	65
8.3	Éléments d'assembleur (directives, étiquettes, instructions)	67
8.4	Notions de machine virtuelle et de bytecode	68
8.5	Génération et exécution de bytecode Python	69
8.6	Génération de code assembleur Python	70
9	Machine virtuelle (VM) Python	71
9.1	Évaluation de code avec une pile	71
9.2	État global de la VM	71
9.3	Objets Python en C	72
9.4	Nombre de références à un objet	74
9.5	Objet de type code	74
10	Spécification du format .pys (assembleur Python)	77
10.1	Unités lexicales	78
10.2	Grammaire formelle de l'assembleur Python	79
10.3	Directives	79
10.4	Étiquettes	82
10.5	Exemple de code assembleur Python	83
11	Spécification du format .pyc (bytecode Python)	87
11.1	Ordre des octets (boutisme)	87
11.2	En-têtes des fichiers .pyc	88
11.3	Valeurs de version_pyvm	89
12	Travail à réaliser	91
12.1	Description des livrables	91
12.2	Modalités de rendu et gestion de projet	99
12.3	Obtenir des conseils	102
12.4	Compétences	103
12.5	Notation	114
Annexes		115
A	Tests	117
A.1	Notion de jeu de test et d'oracle	117
A.2	Tests fonctionnels (ou "d'intégration") et tests unitaires	118
A.3	Notion de test positif et de test négatif	118
A.4	Quelques conseils	119
A.5	Le système de test fourni	120

B	Jeux d'instructions de la VM Python	123
B.1	Encodage des instructions Python	124
B.2	Spécification des instructions	124
C	Sérialisation des objets Python	151
C.1	Principe de la sérialisation en Python	151
C.2	Sérialisation des scalaires	151
C.3	Sérialisation des tuples et des listes	152
C.4	Sérialisation du code	153
C.5	Une spécificité de Python 3.x	155
D	Outil disponible	159
D.1	pyc-objdump	159
D.2	Installation	160

Résumé

Un langage, en informatique aussi, est un vocable qui recouvre une grande diversité. Vous avez commencé à utiliser le C, et avant vous utilisiez déjà Python.

Tous ces langages ont bien sûr des points communs, mais ils permettent chacun des choses radicalement différentes : là où Python vous exonère de la gestion ingrate de la mémoire, vous découvrirez que tout le C n'est qu'une forme d'ascèse visant la lecture et l'écriture à des *adresses* légales, et vous vous retrouvez à gérer explicitement vos blocs de mémoire. Là où Python ne vous a jamais demandé de rien compiler, vous avez découvert l'utilité de `make` (1) (encore un autre langage !) pour créer un exécutable à partir de code C. Là où Python vous donne immédiatement le résultat de votre calcul, le C nécessite d'être explicitement compilé avant de devoir être explicitement exécuté.

Il se trouve que l'implantation de référence du langage Python a été écrite en C. Nous nous proposons dans ce projet de plonger dans quelques arcanes de Python, pour comprendre une petite partie de ce que cela implique que d'écrire un langage nouveau et radicalement différent en utilisant un autre langage déjà disponible.

Ce projet est aussi l'occasion de se confronter à plusieurs questions importantes en informatique : comment spécifier un langage ? comment analyser automatiquement un code source ? comment formater du code exécutable pour le stocker sur disque ? quels sont les liens entre un code machine et son environnement d'exécution ? Plus généralement, ce projet donne à voir un peu de l'*ingénierie* des langages informatiques.

Vos enseignants d'Informatique.

Introduction générale

Guide de lecture de ce document

Ce document est structuré en deux parties :

1. une description des outils théoriques et pratiques que vous aurez à mettre en œuvre, et qui est rédigée plutôt à la manière de notes de cours ;
2. la description de ce que vous aurez à faire.

Le travail que vous aurez à rendre se décompose en quatre livrables qui marquent chacun une étape importante dans le projet. Les deux premiers livrables correspondent à la mise en œuvre successive des deux outils décrits dans la première partie (analyses lexicale et syntaxique).

Il vous faudra donc comprendre quoi lire et quoi *ne pas* lire au fil du projet ! Une première lecture intégrale et relativement superficielle du document est conseillée pour avoir une vue d'ensemble. Chaque livrable sera l'occasion d'approfondir tel ou tel aspect du projet.

Les livrables ne sont pas indépendants : le deuxième se base sur le premier, *etc.*

Pré-requis

Ce projet nécessitera l'utilisation d'une forme de généricité dans votre manière de programmer en C. En effet, nous aurons à manipuler dans le même programme des listes et des files d'objets de *natures* différentes.

À cet égard, si notre implantation utilise des structures de données génériques et des fonctions de rappel, il aurait été parfaitement possible d'utiliser à la place une `union` étiquetée (mais le code final aurait très certainement été *beaucoup* plus indigeste).

Première partie

**Outils pour l'analyse de code
source**

Dans cette partie, nous présentons quelques idées et mécanismes utiles pour l'analyse d'un code source. Un code source est stocké sous forme de texte, et donc constitué de caractères pris les uns après les autres, qu'il va nous falloir analyser.

Cette analyse du code source vise à déterminer s'il respecte les conventions établies par le langage dont il se réclame (sinon, nous aurons détecté une erreur de syntaxe).

La première étape de cette analyse consiste à vérifier que le code source peut se découper en *mots* acceptés par le langage : nous parlerons alors d'analyse *lexicale*.

La seconde étape consiste à vérifier que les mots du code source forment des *phrases correctes* du point de vue de la *grammaire* du langage : on parlera alors d'analyse *syntaxique*.

Dans cette partie, nous illustrons notre propos à l'aide du langage des expressions arithmétiques.

Chapitre 1

Langages et méta-langage EBNF

Dans ce projet, nous allons devoir implanter deux langages informatiques : le langage des expressions dites régulières (*cf. infra*) et le langage assembleur Python. Mais pour pouvoir implanter un langage, il faut commencer par pouvoir en parler. Et donc utiliser un langage particulier pour cela. Un langage qui permet de parler des langages s'appelle un méta-langage.

Notons tout de suite que nous restreignons notre propos aux langages informatiques, les langues naturelles intégrant des ambiguïtés assez inaccessibles aux machines actuelles (pour la même raison qu'elles ne comprennent pas les blagues).

1.1 Notion(s) de grammaire

Un langage informatique est régi par une grammaire formelle, qui est spécifiée par :

1. un ensemble fini de symboles appelés *terminaux*, qui sont les unités lexicales, les “mots” du langage – dans le cas du langage des expressions régulières, ces mots seront pourtant essentiellement des caractères ;
2. un ensemble fini de symboles appelés *non-terminaux*, qui nomment les structures du langage ;
3. un ensemble de *règles de production*, qui expliquent chacune comment produire un non-terminal (plus “abstrait”) à partir de terminaux ou d'autres non-terminaux (moins “abstraits”) – de bonnes vieilles “règles de grammaire” donc ;
4. un élément parmi les non-terminaux appelé l'*axiome* du langage, qui est le non-terminal le plus “abstrait” et qui permet de produire le langage.

Lorsque les règles de production ne dépendent pas du contexte, la grammaire est qualifiée de *non-contextuelle*, ce qui est le cas de tous les langages informatiques que vous utiliserez dans ce projet (C compris !) Un exemple de langage informatique réel utilisant une grammaire sensible au contexte est Perl, ce qui en fait un des langages les plus notoirement difficiles à implanter. La bonne nouvelle est donc que nos règles de production seront toujours les mêmes !

À ce stade de l'exposition, il eût pu convenir de vous infliger un exemple édifiant d'une grande beauté formelle, mais nous préférons attendre, pour illustrer ces notions, d'avoir présenté un outil plus visuel qui permette de les saisir tout aussi efficacement et de manière moins abstraite.

1.2 Forme de Backus-Naur étendue (EBNF)

La forme de Backus-Naur étendue (EBNF) est un méta-langage permettant de représenter une grammaire formelle non-contextuelle. Pourtant normalisée par l'ISO (qui ne l'utilise d'ailleurs même pas pour ses autres publications...), elle est à l'origine d'une foule de dialectes – par exemple, l'Internet Engineering Task Force utilise sa forme de Backus-Naur *augmentée* (ABNF) et Python utilise [une mixture d'EBNF et de PEG](#) (un outil offrant de meilleures capacités de représentation). On comprend dans tous les cas qu'il a fallu améliorer la première méta-syntaxe : la forme de Backus-Naur (développée par John Backus et Peter Naur au tournant des années 1960 lors de la création du langage ALGOL60).

Puisque, manifestement, l'appellation EBNF évoque toujours la même idée avec quelques variations plus ou moins importantes, le lecteur ne nous en voudra pas de lui proposer *notre* EBNF, certes un peu édulcorée, mais très utilisable.

EBNF est un méta-langage : ses symboles sont logiquement appelés des méta-symboles. EBNF utilise les méta-symboles suivants :

1. `:=`, pour introduire une règle de production ;
2. `|`, pour introduire une alternative (utilisé horizontalement et verticalement) ;
3. `...`, pour introduire un intervalle évident entre deux terminaux ;
4. `*`, pour introduire une répétition un nombre positif de fois ;
5. `+`, pour introduire une répétition un nombre strictement positif de fois ;
6. `[...]`, pour introduire un élément optionnel ;
7. `(...)`, pour grouper des éléments.

En EBNF (comme on dirait “en français”), on notera :

1. les terminaux entre simples guillemets (`'terminal'`) ;
2. les non-terminaux entre chevrons (`<non-terminal>`).

Par convention, l'axiome est le non-terminal de la première règle de production, et son nom est celui du langage. L'ensemble des non-terminaux est l'ensemble des parties gauches des règles de production. L'ensemble des terminaux apparaît dans l'énoncé des règles de production. L'opérateur | court-circuite : lorsqu'une règle contient plusieurs alternatives, la première validée est sélectionnée et les suivantes ne sont pas testées. Donc EBNF permet bien de décrire une grammaire formelle non-contextuelle.

1.3 Exemple : les expressions arithmétiques en EBNF

Nous donnons à la Fig. 1.1 la grammaire des expressions arithmétiques en EBNF, autorisant les nombres entiers ou avec une partie décimale, les expressions parenthésées et les noms de variables (vous voyez au passage qu'on peut donc décrire des grammaires avec plus ou moins de puissance expressive).

1.4 Analyse lexicale vs. analyse syntaxique

Nous avons écrit notre grammaire de la Fig. 1.1 de manière à ce que deux parties apparaissent clairement :

1. *partie syntaxique* : jusqu'au non-terminal <factor> inclus, on ne met en jeu que des non-terminaux possédant une signification au regard de l'écriture de l'arithmétique ;
2. *partie lexicale* : à partir du non-terminal <number>, nous ne faisons qu'expliquer comment produire des nombres et des noms de variables bien écrits, qui sont le *vocabulaire de l'arithmétique*.

De fait, comme c'est la règle dans le domaine, nous n'emploierons pas les mêmes techniques de programmation pour former le *lexique* que pour vérifier la *syntaxe*. Nous aurons donc un analyseur lexical (🇬🇧 *lexer*) qui alimentera un analyseur syntaxique (🇬🇧 *parser*) en unités lexicales de base appelées *lexèmes*, et chacun sera régi par une grammaire EBNF en rapport avec son attribution. La grammaire EBNF de l'analyseur lexical aurait <lexem> comme axiome et ses terminaux seraient des caractères, alors que la grammaire EBNF de l'analyseur syntaxique aurait l'axiome du langage et ses terminaux seraient des <lexems>.¹

Une intuition technique justifiant ces deux approches séparées repose sur l'observation que la partie syntaxique peut contenir des définitions mutuellement récursives (cf. la dernière alternative du non-terminal <factor> qui rappelle l'axiome <arith-expr> dans la Fig. 1.1), alors que la partie lexicale n'en contient pas. On doit donc pouvoir en tirer parti au moment de l'implantation.

1. Les grammaires des langages informatiques usuels (C, Python) contiennent jusqu'à quelques centaines de règles de production *syntaxiques*.

$\langle \text{arith-expr} \rangle ::= \langle \text{term} \rangle ('+' | '-' \langle \text{term} \rangle)^*$
 $\langle \text{term} \rangle ::= \langle \text{signed-factor} \rangle ('*' | '/' \langle \text{signed-factor} \rangle)^*$
 $\langle \text{signed-factor} \rangle ::= [\langle \text{sign} \rangle] \langle \text{factor} \rangle$
 $\langle \text{factor} \rangle ::= \langle \text{number} \rangle$
 $\quad | \langle \text{identifieur} \rangle$
 $\quad | '(' \langle \text{arith-expr} \rangle ')'$
 $\langle \text{number} \rangle ::= \langle \text{number-dec} \rangle$
 $\quad | \langle \text{number-float} \rangle$
 $\langle \text{number-dec} \rangle ::= \langle \text{digit} \rangle^+$
 $\langle \text{number-float} \rangle ::= \langle \text{digit} \rangle^* '.' \langle \text{digit} \rangle^* [('e' | 'E') [\langle \text{sign} \rangle] \langle \text{digit} \rangle^+]$
 $\langle \text{identifieur} \rangle ::= ('_' | \langle \text{letter} \rangle) ('_' | \langle \text{letter} \rangle | \langle \text{digit} \rangle)^*$
 $\langle \text{digit} \rangle ::= '0' .. '9'$
 $\langle \text{sign} \rangle ::= '+' | '-'$
 $\langle \text{letter} \rangle ::= \langle \text{letter-min} \rangle | \langle \text{letter-maj} \rangle$
 $\langle \text{letter-min} \rangle ::= 'a' .. 'z'$
 $\langle \text{letter-maj} \rangle ::= 'A' .. 'Z'$

FIGURE 1.1 – Grammaire en EBNF des expressions arithmétiques avec expressions parenthésées (cf. la dernière alternative du non-terminal $\langle \text{factor} \rangle$) et variables (dont le nom ne peut pas démarrer par un chiffre, cf. le non-terminal $\langle \text{identifieur} \rangle$).

Chapitre 2

Analyse lexicale

Notion de lexème

Le texte d'un code source est constitué de "mots" (des valeurs, des noms, des symboles) que l'on appelle collectivement des lexèmes. Il s'agit des unités lexicales de base que l'on écrit pour former un programme.

Pour nous, un lexème sera entièrement décrit par la donnée des attributs suivants :

1. une *valeur* : la chaîne de caractère non vide qu'il représente (`print`, `[`, `3.14`, *etc.*);
2. un *type* : s'agit-il d'une parenthèse ? d'un nom de symbole ? d'un nombre ? *etc.*
3. des *coordonnées* : à quelle colonne de quelle ligne du code source démarre ce lexème ?

La première étape dans l'implantation d'un langage consiste à produire des lexèmes à partir de la bête suite de caractères qu'est le code source. On parle alors d'*analyse lexicale* pour dire que l'on va analyser le texte du code source en unités lexicales élémentaires : les lexèmes.

La seconde étape consistera à s'assurer que les lexèmes entre eux suivent un enchaînement possible, autrement dit qu'ils forment des phrases correctes du point de vue d'une grammaire. On parlera alors d'*analyse syntaxique*.

Mais commençons donc par comprendre en quoi un outil simple et puissant, les expressions régulières, nous permettront de générer de magnifiques lexèmes.¹

1. Dans la grande majorité des expositions sur le sujet de l'analyse syntaxique, les lexèmes sont réputés être donnés, et provenir d'un endroit étrange par une opération inconnue du programmeur. Nous nous faisons donc un devoir de vous proposer une manière, assez élégante au demeurant, de générer des lexèmes.

Chapitre 3

Expressions régulières

Les deux grandes difficultés de l'analyse lexicale consistent :


1. à déterminer quel mot est un lexème et lequel n'en est pas un (et ainsi lever les premières erreurs de syntaxe) – par exemple, 123a n'est pas un lexème du langage C ;
2. à reconnaître automatiquement le *type* des lexèmes (le non-terminal qui lui correspond).

Comment distinguer un nombre d'un nom de variable ou d'un mot-clef du langage ? Nous vous proposons pour cela d'utiliser un outil que vous croiserez très vraisemblablement encore sur votre route : les expressions régulières.

Les expressions régulières ont été inventées par le mathématicien Stephen C. Kleene dans les années 1950 pour décrire les automates finis. Pour nous, il s'agira d'une manière de spécifier des règles d'enchaînements autorisés de caractères pour chaque type de lexèmes. Pour un type de lexème donné, une expression régulière est donc une expression (formée de caractères) qui décrira l'ensemble des lexèmes s'écrivant correctement suivant différentes règles d'enchaînements possibles de caractères.

On aura donc (au moins) une expression régulière pour décrire un *type* de lexème. L'écriture de ces fameuses expressions régulières obéit donc à des règles. Et pour exprimer ces règles, nous avons besoin d'un langage. Nous introduisons donc tout naturellement la grammaire EBNF du langage des expressions régulières utile à ce projet à la Fig. 3.1.

Vous remarquerez la grande similitude entre le langage des expressions régulières et le méta-langage EBNF : ils permettent tous les deux de spécifier les mêmes types d'apparition d'un élément, mais pas toujours de la même manière (pour parler de quelque-chose d'optionnel, on utilise les crochets en EBNF, alors qu'on utilisera l'opérateur ? dans une expression régulière, mais * et + conservent la même signification dans les deux langages).

Il se trouve que nous vous proposerons dans la foulée une amorce de code, que nous commenterons, qui permettra de commencer à implanter efficacement cette grammaire pour produire un analyseur lexical ( *lexer*) : un logiciel

```

<regexp> ::= <char-group>*
<char-group> ::= <char-set> [ '*' | '+' | '?' ]
<char-set> ::= ' ' | ( [ '^' ] ( <char> | ( '[' <group-interval> ']' ) ) )
<group-interval> ::= ( <char:1> [ '-' <char:2> ] )+
<char> ::= ( '\' ( <meta> | <C-escaped> ) ) | ^<meta>
<meta> ::= '*' | '+' | '?' | '.' | '^' | '[' | ']' | '-' | '\'
<C-escaped> ::= 'n' | 't'

```

FIGURE 3.1 – Grammaire en EBNF des expressions régulières du projet.

qui lit un code source et des définitions de types de lexèmes, et qui retourne la liste des lexèmes trouvés, ou génère une erreur si une suite de caractères ne correspond à aucun type de lexème.

3.1 Syntaxe des expressions régulières en EBNF

En d'autres termes, voici les quelques opérateurs de syntaxe dont nous aurons besoin pour décrire nos expressions régulières :

- . (un point), pour désigner n'importe quel caractère ASCII ;
- ? (un point d'interrogation), pour signifier que ce qui précède peut apparaître au plus une fois (et peut, donc, ne *pas* apparaître) ;
- + (le symbole de l'addition), pour signifier que ce qui précède doit apparaître au moins une fois ;
- * (une étoile), pour signifier que ce qui précède peut apparaître un nombre arbitraire de fois (y compris, donc, ne *pas* apparaître) ;
- ^ (un accent circonflexe), pour signifier que ce qui suit ne peut pas apparaître ;
- les caractères crochet [et]. Ce qui est placé entre crochets permet de désigner plusieurs caractères. Entre crochets, on trouvera soit des caractères isolés (ainsi, [at9] signifie a, t ou 9), soit des intervalles de caractères, entre deux caractères séparés par un tiret - (ainsi, [a-z] signifie n'importe quel caractère entre a et z). Plusieurs intervalles peuvent se succéder (ainsi [ac-e0-9z] signifie a, z ou n'importe quel caractère entre c et e ou entre 0 et 9).
- Il nous faut un moyen de désigner chacun des caractères spéciaux <meta> du langage des expressions régulières en tant que caractère, par exemple pour pouvoir différencier le *caractère* * de l'*opérateur* *. Dans notre langage, le caractère \ (contre-oblique) produira ainsi un *échappement*, afin de désigner le caractère qui suit. Par exemple, \. dé-

signe le caractère `.` (un point), `*` désigne le caractère `*` (une étoile), `\\` désigne le caractère `\` (contre-oblique), *etc.* Nous décidons pour simplifier que les caractères spéciaux `<meta>` doivent toujours être échappés.

- Enfin, notons que *seuls* les caractères spéciaux peuvent être échappés. Ainsi `\z` ne signifierait rien pour nous ! À cause du langage C que nous utilisons pour implanter les expressions régulières, les deux exceptions à cette dernière observation sont les deux combinaisons suivantes :
- `\n`, pour désigner la fin d'une ligne ;
- `\t`, pour désigner une tabulation.

L'on pourrait encore enrichir le langage des expressions régulières et de fait, il en existe plusieurs dialectes, chacun avec sa solide base d'utilisateurs. Par exemple, la commande `grep(1)` accepte plusieurs dialectes d'expressions régulières et une option permet de choisir lequel on veut utiliser.

3.2 Exemples

Donnons tout de suite quelques exemples d'expressions régulières :

1. `abba` : la chaîne de caractères `abba` ;
2. `a+b*` : au moins un `a`, peut-être suivi(s) d'un nombre quelconque de caractères `b` (`a`, `abbbbb`, `aabb`, *etc.*)
3. `.*` : ensemble de toutes les chaînes de caractères de longueur finie ;
4. `.+` : ensemble de toutes les chaînes de caractères non vides de longueur finie.

Continuons en illustrant le principe des groupes de caractères :

1. `[abc]` : les caractères `a`, `b` ou `c` ;
2. `[a-f]` : les caractères `a`, `b`, `c`, `d`, `e` ou `f` ;
3. `[a\ -f]` : les caractères `a`, `-` ou `f` ;
4. `^[a-f]` : n'importe quel caractère, excepté `a`, `b`, `c`, `d`, `e` et `f`.

Au passage, observez que l'on pourrait fort bien considérer l'opérateur `.` (le point, qui désigne n'importe quel caractère) comme un raccourci syntaxique pour `^[]` (le complément de l'ensemble vide des caractères est l'ensemble de tous les caractères).

Formons maintenant quelques groupes déjà un peu plus utiles :

1. `[a-zA-Z]` : soit une lettre minuscule, soit une lettre majuscule ;
2. `[0-9]` : un chiffre décimal ;
3. `[0-9a-fA-F]` : un chiffre hexadécimal.

Donnons maintenant quelques expressions régulières en rapport avec des objets familiers :

1. `[+\-]?[0-9]+` : un entier relatif (observez comment on permet de n'avoir pas de signe dans l'écriture de l'entier relatif à l'aide de `?`, et comment `-` a dû être échappé pour ne pas décrire le début d'un intervalle de caractères) ;
2. `[1-9][0-9]*` : un entier non signé en décimal en C (qui interdit que le premier caractère soit un zéro) ;
3. `0x[0-9a-fA-F]+` : un entier non signé en hexadécimal en C (qui doit commencer par `0x`, par exemple `0x1ab9c`) ;
4. `[_a-zA-Z][_a-zA-Z0-9]*` : un nom de symbole en C, ne pouvant pas avoir un chiffre décimal pour caractère initial (exemples : `toto`, `h4X0r`, `big_fat_warning`).

Et poursuivons avec quelques expressions régulières utiles pour structurer un code source :

1. `^\n*` : tous les caractères jusqu'à la fin de la ligne courante du code source (*i.e.* un nombre indéfini de fois un caractère différent de `\n`) ;
2. `[\t]+` : un blanc (au moins une espace ou une tabulation horizontale) ;
3. `\n+` : au moins un retour à la ligne.

Observez que les deux dernières expressions régulières nous permettent de découper des mots entendus au sens ordinaire, séparés par des blancs. D'ores et déjà, il est donc absolument certain que chercher à utiliser `strtok(3)` pour découper un code source dans ce projet constitue une erreur stratégique majeure puisque nos expressions régulières constituent *un seul* mécanisme *strictement plus puissant*.

La puissance de ce formalisme mathématique réside dans le fait qu'une expression régulière, souvent pas très longue, permet de décrire de manière intensionnelle (donc utilisable par un humain pour *définir*) des ensembles dénombrables de chaînes de caractères (de taille potentiellement infinie).

À la vue des exemples ci-dessus, il est clair que nous pouvons utiliser une expression régulière pour définir un type de lexème utile pour décrire un code source (nombre, nom de symbole, parenthèses, *etc.*) Nous aurons bien mérité un peu de poésie pour finir car nos efforts nous permettront d'observer `**` : un nombre indéfini d'étoiles.

3.3 Contre-exemples

Voici quelques chaînes qui ne sont pas des expressions régulières conformes à notre langage, et dont l'analyse devrait donc générer une erreur :


1. `*ab` — *Erreur* : L'opérateur `*` doit *suivre* quelque-chose (c'est un opérateur *postfixe*) ;
2. `[a-z+]` — *Erreur* : S'il s'était agi du caractère `+`, il aurait dû être échappé, et il ne saurait s'agir ici de l'opérateur `+` car un tel opérateur n'a pas de sens pour définir un *ensemble* de caractères *différents* par définition ;

3. a^{\sim} — *Erreur* : L'opérateur de négation doit *précéder* quelque-chose (c'est un opérateur *préfixe*) ;
4. $\backslash z$ — *Erreur* : Le caractère z ne peut être échappé.

3.4 Lecture (*parsing*) et représentation en mémoire d'une expression régulière

Lire une expression régulière, c'est, partant d'une chaîne de caractères, s'assurer qu'elle respecte notre langage des expressions régulières, puis charger cette expression régulière dans une structure de données adéquate en mémoire, afin qu'elle puisse ensuite être manipulée par la machine : nous sommes en train de vouloir *implanter* un outil défini mathématiquement.

Ce sera là l'une de vos premières tâches dans ce projet, durant l'incrément 1. Les voies pour y parvenir seront exposées au premier BE du projet.

La situation est donc la suivante : nous sommes en train d'implanter un micro-langage (celui des expressions régulières), afin de pouvoir ensuite l'utiliser pour décrire le langage qui nous intéresse (celui de l'assembleur Python). Il s'agit d'une manière de procéder absolument standard en informatique, qui est parfois présentée comme relevant de l'échaffaudage ( *code scaffolding*), exactement comme une échelle nous permet d'accéder plus facilement à des objets situés plus haut (ici dans une hiérarchie intellectuelle).

Nul doute alors que le lecteur ayant déjà chuté d'une échelle percevra avec toute l'acuité nécessaire l'importance capitale d'une implantation rigoureuse des expressions régulières. La bonne nouvelle, c'est que des expressions régulières fiables rendent la suite du projet assez triviale.

Par le vocable *fiable*, nous voulons en réalité appuyer sur l'idée qu'implanter les expressions régulières doit être un processus mathématique absolument inexorable — qui ne saurait contenir rigoureusement aucune rustine horrible (comme par exemple se dire qu'utiliser `strtok(3)` à un moment quelconque du processus serait une bonne idée).


Les conséquences *pratiques* majeures sont :

1. Vous préférerez *écrire des fonctions courtes* en C (en première approximation, une fonction *trop* longue est une fonction qui ne s'affiche pas intégralement dans l'écran que vous utilisez pour l'écrire) ;
2. Cela vous permettra de n'avoir virtuellement *aucune redite dans votre code*, et donc un seul endroit où corriger une éventuelle erreur¹ ;
3. Vous voudrez *tester votre code*, et le tester *de manière systématique* — nous vous fournissons tout un attirail, assez léger d'utilisation, pour parvenir à des *garanties de fiabilité* assez nettement supérieures à ce que vous produiriez sans.

1. La technique du *copier-coller* a été inventée au Xerox PARC par Lawrence G. Tesler en 1973 pour semer le chaos et la confusion sous couvert de faciliter la vie des gens.

3.5 Mécanisme pour reconnaître une expression régulière

Si nous avons maintenant à notre disposition le langage des expressions régulières (un outil intellectuel de nature mathématique), et la capacité de lire une expression régulière (car nous en avons donné une expression sous forme de grammaire EBNF), il nous manque le *mécanisme* par lequel vérifier si une chaîne de caractères correspond à la description donnée par l'expression régulière (et donc faire en sorte que l'outil mathématique produise automatiquement ses effets dans le monde physique).

C'est ce qu'on appelle la *mise en correspondance* ( *matching*) d'une chaîne et d'une expression régulière.

On dira qu'il y a correspondance entre la chaîne et l'expression régulière, ou que *l'expression reconnaît la chaîne*, si le *début* de la chaîne fait partie de l'ensemble des chaînes décrites par l'expression régulière.

Ainsi par exemple, l'expression régulière `0x[0-9a-fA-F]+` :

- reconnaît intégralement la chaîne `0xabcdef` (elle en consomme tous les caractères et il ne reste plus rien à analyser ensuite) ;
- ne reconnaît pas la chaîne `bonjour0xabc` (le début de la chaîne n'est pas le préfixe `0x` attendu) ;
- reconnaît la chaîne `0xabcdefzzz` jusqu'au premier caractère `z`, et il resterait donc `zzz` à analyser.

Il nous faut donc être capable de reconnaître une expression régulière au début d'une chaîne, et savoir où elle s'arrête — pour savoir où essayer de reconnaître la prochaine chaîne !

Pour cela, et sans prétendre à une quelconque exhaustivité, nous pouvons implanter un mécanisme de base d'une simplicité, d'une puissance et d'une évolutivité très satisfaisantes en prenant appui sur un code de Rob Pike [1, Ch. 1].² Nous l'avons très légèrement mis en forme pour qu'il corresponde davantage à nos besoins mais sa beauté est intacte, comme le Code 3.1 devrait vous en convaincre.

2. Le petit défi que Brian Kernighan avait posé à Rob Pike était le suivant : comment implanter le maximum de fonctionnalités des expressions régulières dans le code C le plus court possible ? Pike a quand même mis quelques heures pour produire ce petit bijou — et beaucoup plus complet que ce que nous en avons extrait.

3.5. MÉCANISME POUR RECONNAÎTRE UNE EXPRESSION RÉGULIÈRE27

CODE SOURCE 3.1 – Mécanisme de base pour reconnaître les expressions régulières pouvant contenir des caractères (y compris le caractère . (un point) désignant n'importe quel caractère) et l'opérateur * (d'après Rob Pike).

```
1  int re_match( char *regexp, char *source, char **end );
2
3  static int re_match_zero_or_more( char c, char *regexp, char *
4      source, char **end ) {
5      char *t = source;
6
7      while ( '\0' != *t && ( *t == c || '.' == c ) ) t++;
8
9      do {
10         if ( re_match( regexp, t, end ) ) return 1;
11     } while ( t-- > source );
12
13     if ( end ) *end = source;
14     return 0;
15 }
16
17 int re_match( char *regexp, char *source, char **end ) {
18     if ( !source ) {
19         if ( end ) *end = source;
20         return 0;
21     }
22
23     if ( !regexp || '\0' == regexp[ 0 ] ) {
24         if ( end ) *end = source;
25         return 1;
26     }
27     if ( '*' == regexp[ 1 ] ) {
28         return re_match_zero_or_more( regexp[ 0 ], regexp+2, source,
29             end );
30     }
31     if ( '\0' != *source &&
32         ( '.' == regexp[ 0 ] || *source == regexp[ 0 ] ) ) {
33         return re_match( regexp+1, source+1, end );
34     }
35
36     if ( end ) *end = source;
37     return 0;
38 }
```

L'idée maîtresse du Code 3.1 est de faire avancer la lecture du code source de conserve avec celle de l'expression régulière, et de s'arrêter soit quand le code source ou l'expression régulière est terminée, soit quand le code source n'est plus reconnu par l'expression régulière.

La fonction `re_match` est la fonction que nous devons appeler pour déterminer si le mot démarrant à l'adresse `source` est reconnu par l'expression régulière `regexp`. Ainsi, à chaque nouveau caractère à analyser, nous com-

mençons par regarder si nous sommes arrivés à bout de l'expression régulière (l. 23). Si c'est le cas, nous marquons le début du mot suivant et déclarons que nous avons reconnu notre expression régulière en début de chaîne.

Ensuite, la fonction `re_match` teste si le caractère suivant dans l'expression régulière est une étoile (l. 27). Si c'est le cas, on a détecté l'opérateur `*`, et on exécute la fonction `re_match_zero_or_more` pour traiter spécifiquement ce cas (cf. *infra*).

Enfin, l'on trouve le cas général : celui où l'on n'a pas d'opérateur à traiter, seulement des caractères. L'on peut avancer dans la lecture du code source (l. 31) tant que les caractères courants de l'expression régulière et du code source sont identiques (ou quand celui de l'expression régulière est un point, puisque nous avons dit qu'un point dans une expression régulière permet de désigner n'importe quel caractère).

La fonction `re_match_zero_or_more` permet donc d'implanter l'opérateur `*` du langage des expressions régulières. Au moment de l'appeler (l. 28), on lui passe en premier paramètre le caractère qu'il va s'agir de reconnaître zéro ou davantage de fois, ainsi que la suite de l'expression régulière (`regexp+2` car on se placera pour la suite de l'expression régulière après le caractère recherché et l'étoile). C'est important pour pouvoir continuer l'analyse *depuis* `re_match_zero_or_more` (cf. *infra*) et donc tenter de faire progresser la reconnaissance de la chaîne par l'expression régulière en avançant dans cette dernière.

La première étape de cette fonction `re_match_zero_or_more` consiste à avancer dans la lecture du code source tant qu'il contient le caractère passé en premier paramètre (l. 4–6). Enfin, il faut essayer de continuer la lecture du code source avec la suite de l'expression régulière (l. 9), et cela, depuis toutes les occurrences que nous avons recensées du caractère répété, en commençant par la dernière (l. 10).

En effet, comment par exemple se dépêtrer de l'expression régulière `a*a+b` sinon ? Sur la chaîne `aaaabc`, on voudrait intuitivement qu'il reste ensuite seulement `c` à analyser (et donc que notre mot extrait du code source soit `aaaab`). Mais si l'on n'essaye pas de prolonger la lecture à partir des positions précédentes, alors la fin de l'analyse de `a*` nous amènerait à `bc` (on aurait extrait le mot `aaaa`), mais comme on a un `a+` ensuite dans l'expression régulière, on déclarerait l'échec de la reconnaissance de `aaaabc` par `a*a+b` (`bc` ne contient pas de `a` au début). Alors qu'en testant aussi depuis les occurrences précédentes, on en vient nécessairement à tester si `abc` est reconnue par `a+b`, ce qui est manifestement le cas jusqu'à `c`, ce que nous voulions. L'important est de s'assurer que l'on se donne toutes les chances d'avancer le plus possible dans l'expression régulière.

Par rapport au code de Rob Pike, nous avons retiré de quoi spécifier que nous voudrions tester une expression régulière sur la *fin* d'une chaîne ou *n'importe où* dans une chaîne (car seul le *début* de la chaîne nous intéresse). Même si cela ne devrait pas trop nous impacter, nous avons opté pour la version de `re_match_zero_or_more` qui cherche à avancer d'abord dans le code source plutôt que dans l'expression régulière. Et nous avons aussi ajouté le

3.5. MÉCANISME POUR RECONNAÎTRE UNE EXPRESSION RÉGULIÈRE 29

pointeur end, qui permet de savoir où le prochain mot commencera. Nous utilisons le Code 3.2 pour illustrer.

CODE SOURCE 3.2 – Code d'illustration du Code 3.1 (regex-match.c).

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main ( int argc, char *argv[] ) {
5      char *end = NULL;
6
7      if ( argc < 3 ) {
8          fprintf( stderr, "%s regex text\n", argv[ 0 ] );
9          exit( EXIT_FAILURE );
10     }
11
12     end = argv[ 2 ];
13
14     printf( "The start of '%s' is ", argv[ 2 ] );
15
16     if ( re_match( argv[ 1 ], argv[ 2 ], &end ) ) {
17         printf( "%s, %s: '%s'.\n",
18             argv[ 1 ], *end ? "next" : "END", end );
19     }
20     else {
21         printf( "*not* %s.\n", argv[ 1 ] );
22     }
23
24     exit( EXIT_SUCCESS );
25 }
```

Voici quelques exemples d'exécution :

```
$ ./bin/regex-match.exe 'aa' 'aabbccddddd'
The start of 'aabbccddddd' is in aa, next: 'bbccddddd'.

$ ./bin/regex-match.exe 'a*' 'aabbccddddd'
The start of 'aabbccddddd' is in a*, next: 'bbccddddd'.

$ ./bin/regex-match.exe 'a*b*c*' 'aabbccddddd'
The start of 'aabbccddddd' is in a*b*c*, next: 'dddd'.

$ ./bin/regex-match.exe 'a.b*' 'aabbccddddd'
The start of 'aabbccddddd' is in a.b*, next: 'ccddddd'.

$ ./bin/regex-match.exe 'ba.b*' 'aabbccddddd'
The start of 'aabbccddddd' is *NOT* in ba.b*.
```

Comme le fait fort justement remarquer Brian Kernighan, il est remarquable

qu'un peu moins de 40 lignes de C permettent d'atteindre une telle expressivité.

3.6 Extensions du mécanisme

Le mécanisme de base décrit plus haut doit néanmoins être assez largement complété pour pouvoir être utile en pratique.

3.6.1 Ajout des opérateurs + et ?

Observez la grande modularité du Code 3.1 : il suffit d'ajouter des fonctions (mutuellement récursives) pour prendre en compte de nouveaux opérateurs : le Code 3.1 n'implante que l'opérateur *, mais il sera trivial de lui rajouter les opérateurs + et ? à l'aide des fonctions, elles-mêmes triviales, `re_match_one_or_more`³ et `re_match_zero_or_one`, respectivement.

3.6.2 Ajout des groupes de caractères

C'est la partie la plus délicate et la plus importante. Elle demande de substantielles modifications au Code 3.1, mais sa logique et sa structure resteront inchangées. On veut simplement remplacer les tests d'égalité de deux caractères par un test d'appartenance à un groupe de caractère.

Cela implique de ne plus avancer caractère par caractère dans l'expression régulière, mais groupe de caractères par groupe de caractères (et un caractère pris isolément serait alors compris comme étant seul dans son groupe). Ainsi, il ne semblerait pas inadéquat de remplacer notre expression régulière sous forme de chaîne de caractères, par une expression régulière représentée maintenant sous forme de *liste de groupes de caractères*. Et on cherchera à avancer dans cette liste pour avancer vers la reconnaissance de notre chaîne à analyser par l'expression régulière, comme on cherchait précédemment à avancer dans la chaîne de caractères de l'expression régulière.

Notez que la lecture de l'axiome `<regexp>` (cf. la grammaire de la Fig. 3.1) suggère en effet qu'une expression régulière devrait assez bien pouvoir se représenter en machine par une liste de `<char-set>` auxquels on aurait rajouté un entier codant le type d'apparition possible (exactement une fois, au plus une fois, au moins une fois, au moins zéro fois) — et l'ensemble ainsi formé imiterait assez bien la description d'un `char-group`. Libre à vous de préférer ajouter, ou pas, un autre entier pour coder éventuellement le complémentaire d'un ensemble (l'opérateur préfixe `^`). Les deux options sont possibles, d'égale non-difficulté, et n'ont absolument aucune influence sur le résultat final.

3. Que vous contruirez en ré-utilisant le code de `re_match_zero_or_more` après avoir remarqué que `a+` peut être vu comme un raccourci syntaxique pour `aa*`...

3.6.3 Lecture des expressions régulières

Cette partie et la précédente sont indissociables, mais celle-ci est néanmoins largement autonome alors nous la décrivons à part.

Le dernier problème, de taille, que nous devons résoudre est que nous n'avons pas encore de moyen d'introduire, sous forme de chaînes de caractères donc, des expressions régulières contenant des groupes de caractères.

Vous devrez donc écrire une fonction `list_t re_read(char *re)` qui renverra une liste de groupes de caractères décrivant l'expression régulière passée en paramètre sous forme de chaîne de caractères.

L'idée générale pour analyser des chaînes de caractères contenant une expression régulière s'appuie sur leur structure (justement !) régulière. En effet, une expression régulière est constituée d'une suite de trois informations : (1) un accent circonflexe de négation optionnel, (2) un (groupe de) caractère(s), suivi de (3) un éventuel opérateur (*, +, ?). Il conviendra donc d'avoir autant d'étapes, une pour la lecture de chaque information. La première étant triviale à lire, nous nous concentrons sur les suivantes.

Pour lire un (groupe de) caractère(s), il faudra d'abord déterminer si le caractère courant de l'expression régulière est un [, auquel cas nous devons lire un groupe de la forme `[^\\]`, ou pas, auquel cas nous n'avons à lire qu'un caractère isolé (par exemple, le 0 dans `0+[1-9]`). Pour lire un caractère, il faut juste faire attention au fait qu'il soit échappé ou pas mais c'est très simple à mettre en œuvre.

Pour lire les groupes de caractères, il faudra détecter une autre différence d'écriture, entre les intervalles de caractères et les caractères cette fois : les intervalles de caractères ont en effet un tiret (-) pour second caractère. Une fois connus les caractères extrémaux d'un intervalle, il n'y a plus qu'à ajouter les caractères dans l'intervalle à la structure de données décrivant le groupe. Les caractères isolés sont ajoutés tels quels à la structure de données du groupe.

Par exemple, dans le groupe `[a-z0-9\\-\\n]`, on devra détecter, dans cet ordre, les deux intervalles `a-z` et `0-9`, et les deux caractères `-` et `\\n`.

Notez encore que c'est lors de la lecture d'une expression régulière que vous devrez implanter le mécanisme d'échappement des caractères !

Enfin, observez que nous avons voulu présenter cette partie de manière systématique, conformément à l'esprit général qui doit vous habiter. Le Chap. 5 de ce document décrit une manière simple et efficace d'implanter systématiquement une grammaire EBNF (en gros : écrire une fonction par règle de grammaire). À vous de voir si vous préférez vous entraîner dès maintenant à appliquer cette méthodologie, elle sera de toute façon indispensable dans la suite pour le langage de l'assembleur Python.

3.6.4 Ajout de l'opérateur de négation ^

L'opérateur ^ signifie que ce qui suit ne peut pas apparaître. Par exemple, `^a.*` désignera toutes les chaînes de caractères non vides ne commençant pas par a.

Cet opérateur important est très facile à rajouter une fois qu'on a le support pour les groupes de caractères, car il suffit (i) d'ajouter l'information selon laquelle on devra trouver un caractère dans le groupe de caractère complémentaire (présence de `^`) ou pas, et (ii) de prendre le cas échéant le complémentaire d'un ensemble de 256 entiers (la table ASCII place dans la première moitié de cet ensemble ordonné les caractères d'imprimerie usuels qui nous intéressent, et réserve la seconde pour des extensions davantage baroques et variables).

3.6.5 Exemples d'analyse d'expressions régulières

Voici quelques exemples d'analyse d'une chaîne de caractère encodant une expression régulière :

```
$ ./bin/regex-read.exe 'STORE_FAST'
One in "S", one time.
One in "T", one time.
One in "O", one time.
One in "R", one time.
One in "E", one time.
One in "_", one time.
One in "F", one time.
One in "A", one time.
One in "S", one time.
One in "T", one time.

$ ./bin/regex-read.exe '[\+\\-]?[0-9]+'
One in "+-", zero or one time.
One in "0123456789", one or more times.

$ ./bin/regex-read.exe '[\\-\\+]?[0-9]+'
One in "-+", zero or one time.
One in "0123456789", one or more times.

$ ./bin/regex-read.exe '0x[0-9a-fA-F]+'
One in "0", one time.
One in "x", one time.
One in "0123456789abcdefABCDEF", one or more times.

$ ./bin/regex-read.exe '^[a-f\\-q-w\\+]+q^t*[x]?'
One not in "abcdef-qrstuvw+", one or more times.
One in "q", one time.
One not in "t", zero or more times.
One in "x", zero or one time.
```

Notez que vous n'êtes pas tenus de reproduire parfaitement ces sorties. Par exemple, les ensembles dont nous avons strictement besoin ne sont pas

ordonnés, mais les ensembles de caractères sont décrits par des ensembles ordonnés (la table ASCII ne fait guère que numéroté les caractères). Donc il est bien parfaitement égal d'afficher les ensembles "--" ou "+-". Dans le doute, faites au plus simple : affichez les caractères d'un groupe dans l'ordre de la table ASCII !

Et voici quelques exemples d'analyse du *début* d'une chaîne de caractères par une expression régulière passée elle aussi sous forme de chaîne de caractères :

```
$ ./bin/regex-match.exe '[\-\+]?[0-9]+\.[0-9]*' '1234'
The start of '1234' is in [\-\+]?[0-9]+\.[0-9]*, END: ''

$ ./bin/regex-match.exe '[\-\+]?[0-9]+\.[0-9]*' '-+1234'
The start of '-+1234' is *NOT* in [\-\+]?[0-9]+\.[0-9]*.

$ ./bin/regex-match.exe '[\-\+]?[0-9]+\.[0-9]*'
'-1234.321'
The start of '-1234.321' is in [\-\+]?[0-9]+\.[0-9]*, END:
'',

$ ./bin/regex-match.exe 'a*bc?^d+' 'accdefg'
The start of 'accdefg' is in a*bc?^d+, next: 'defg'.
```

3.6.6 Discussion

Écriture des chaînes de test dans le terminal

Notez que nous avons systématiquement utilisé les guillemets simples pour les exemples de chaînes ci-dessus dans le terminal (expression régulière ou texte à analyser). Cela permet de dire au terminal de ne pas appliquer sa magie ordinaire sur les chaînes qu'on lui donne, et donc de pouvoir les analyser telles qu'on s'attend à les lire. Bref, ne faites pas autrement.

Plato on the beach

Les plus platoniciens d'entre vous auraient sans doute préféré ces versions des règles suivantes :

$$\begin{aligned} \langle \text{char-set} \rangle &::= [\text{'"}] (\text{'.'} | \langle \text{char} \rangle | (\text{'('} \langle \text{group-interval} \rangle \text{'})' }) \\ \langle \text{group-interval} \rangle &::= (\langle \text{char:1} \rangle [\text{'.'} \langle \text{char:2} \rangle])^* \end{aligned}$$

FIGURE 3.2 – Coquetterie platonicienne assez largement inutile pour nous.

L'idée aurait sans doute consisté à faire en sorte que $\hat{}$ soit absolument équivalent à l'opérateur \cdot (point). Après tout, le complément de l'ensemble vide, c'est le tout. Les mêmes en déduiraient qu'une signification raisonnable pour $\hat{}$ serait d'attendre la fin de la chaîne à analyser (si bien que $\hat{}$ serait le moyen d'introduire effectivement la chaîne vide dans nos développements).

L'on peut certes voir les choses comme cela. Mais il se trouve que ce n'est pas strictement utile pour nous : on peut très bien déclarer que $\hat{}$ est une erreur de syntaxe (on impose des $\langle \text{char-set} \rangle$ non vides, ce qui n'est pas vraiment délirant, et l'on est déjà capable de détecter – mais certes pas d'attendre – la fin de la chaîne à analyser), et *idem* pour $\hat{}$ puisque l'on peut déjà exprimer la même chose avec l'opérateur point. C'est la raison pour laquelle nous vous avons proposé par défaut les règles les plus simples.

Bref, c'est sans doute effectivement d'une grande beauté formelle, mais cela nécessite tout de même de réfléchir un peu plus fort sur l'encodage d'un char-set pour que cela fonctionne.

À vous de voir ! Mais du point de vue de ce projet, la position platonicienne *hardcore* n'est rien de plus qu'une coquetterie un peu incantatoire. Et certainement pas une priorité.

Prise en compte du contexte

Une amélioration sans doute un peu moins inutile, mais pas non plus décisive, consisterait à introduire des éléments de contexte dans la lecture d'une expression régulière.

Par exemple, remarquez qu'il n'y a réellement besoin d'échapper le tiret (–) que lorsqu'on voudrait éventuellement en faire le second caractère d'un

`char-set`. De la même manière, on n'aurait réellement besoin d'échapper] que pendant la lecture d'un `char-set`. L'on pourrait multiplier les observations comme celles-là.

Une première façon de procéder pourrait alors être d'avoir une liste de caractères qu'on peut échapper, ou pas. C'est assez fruste mais cela fonctionne (vous avez le droit de faire des choses plus intelligentes). Cela donnerait les règles suivantes :

$$\langle \text{char} \rangle ::= ('\langle \text{meta} \rangle' | \langle \text{C-escaped} \rangle) | ^\langle \text{mandatory-escape} \rangle$$

$$\langle \text{free-escape} \rangle ::= '-' | ']'$$

$$\langle \text{mandatory-escape} \rangle ::= '*' | '+' | '?' | ':' | '^' | '[' | '\'$$

$$\langle \text{meta} \rangle ::= \langle \text{free-escape} \rangle | \langle \text{mandatory-escape} \rangle$$

$$\langle \text{C-escaped} \rangle ::= 'n' | 't'$$

FIGURE 3.3 – Une syntaxe un peu moins psycho-rigide (qui frise même le laxisme) pour les expressions régulières.

Mise en garde : si vous choisissez cette voie, assurez-vous que la syntaxe de base exigeant que tous les méta-caractères soient échappés continue de fonctionner. Vous avez le droit de faire mieux, mais vous devez aussi être capable de faire moins (notion de rétro-compatibilité).

Chapitre 4

Un analyseur lexical

À l'étape précédente, nous avons en réalité expliqué comment *compiler* et *exécuter* (ou exploiter, comme diraient les *cypherpunks*) *une seule* expression régulière, c'est-à-dire, respectivement, la mettre sous forme de liste de `char-group`, et utiliser cette liste pour avancer dans l'analyse en lexèmes d'un texte source. Compiler, ce n'est guère que traduire une information sous une forme qu'un automate pourra exploiter efficacement. Et c'est nous qui exploitons ainsi efficacement l'automate en retour.

La principale limitation qui saute aux yeux, c'est que nous ne pouvons pour l'instant découper un code source que s'il est constitué de chaînes reconnues par une seule expression régulière. De la même manière qu'un texte en français est constitué de noms, de verbes, de conjonctions de coordination, *etc.*, un code source va être constitué d'identifiants, d'opérateurs arithmétiques, de parenthèses et séparateurs divers, de nombres, *etc.* Nous devons donc pouvoir reconnaître plusieurs *types* de lexèmes.

Un analyseur lexical doit :

1. Charger une liste de définitions de lexèmes (nom de type et expression régulière correspondante) ;
2. Charger un fichier de code source sous forme d'une chaîne de caractères ;
3. Analyser cette chaîne en produisant la liste des premiers lexèmes reconnus par la liste des expressions régulières (le type de chaque lexème sera donc celui de l'expression régulière ayant reconnu la première un nombre non nul de caractères dans la chaîne de code source).

Le fait que notre interpréteur d'expression régulière `re_match` nous dise où poursuivre l'analyse est alors capital, et nous permet d'itérer jusqu'à la fin du code source s'il ne contient pas d'erreur, ou bien de lever une erreur de syntaxe que le programmeur doit corriger.

Notons qu'en réalité, un moteur d'expression régulière devrait toujours retourner la chaîne la plus longue. Mais nous choisissons de nous arrêter à la

première chaîne non vide reconnue afin d'avoir davantage de contrôle sur ce que nous faisons (l'ordre de test des expressions régulières en début de chaîne devient alors important, et un levier supplémentaire à notre disposition).

Mettons donc la dernière main à notre analyseur lexical (notre générateur de lexèmes donc) en commençant par souligner une faiblesse de notre implantation des expressions régulières : elle ne permet pas d'exprimer une alternative ni la notion de groupe (les parenthèses utilisées dans les règles de la grammaire de la Fig. 3.1).

Par exemple, si l'on voulait reconnaître un nombre décimal positif en virgule flottante, on pourrait utiliser l'expression régulière `[0-9]+\.[0-9]*`. Mais on pourrait aussi vouloir autoriser une écriture en notation exponentielle avec `[0-9]+\.[0-9]*[eE][\-+]?[0-9]+`. Or nous n'avons pas, au contraire des moteurs d'expressions régulières un peu sérieux, de quoi exprimer que nous souhaiterions accepter indifféremment l'une ou l'autre forme pour nos nombres.

À la vérité, il ne serait pas bien difficile d'étendre notre langage des expressions régulières pour inclure les deux. Ce serait juste trop ambitieux pour un début, et nous vous proposons une manière relativement simple et *exploitable* de pallier ces défauts.

4.1 Structure de lexème

Le problème soulevé en introduction peut être assez largement contourné en utilisant une structure de lexème comme dans le Code 4.1.

CODE SOURCE 4.1 – Structure de lexème.

```
1 struct lexem {  
2     char *type;  
3     char *value;  
4     int line;  
5     int column;  
6 };
```

En effet, en faisant en sorte que le type du lexème soit une chaîne de caractères, cela nous laisse la possibilité de nommer plusieurs expressions régulières de la même manière, et donc d'exprimer une forme limitée d'alternative, mais suffisante pour nous.

Ainsi donc, pour les nombres décimaux positifs en virgule flottante, il suffira de définir deux "sous-types" de lexèmes, chacun *portant le même nom de type*, mais différenciés par leur expression régulière : une pour les nombres exprimés sans notation exponentielle, et une seconde pour les nombres exprimés en notation exponentielle.

Concernant les coordonnées de ligne et de colonne du lexème dans le code source à analyser, on pose la convention que le numéro de ligne démarre à 1 et le numéro de colonne à 0.

4.2 Fichier de définitions de types de lexèmes

Notre analyseur lexical ira lire ses expressions régulières dans un fichier contenant, pour chaque ligne :

- Une ligne vide, ou
- Un commentaire introduit par le caractère #, jusqu'à la fin de la ligne (le caractère # étant ici le seul à devoir être échappé par \), ou
- Un couple de chaînes de caractères séparées par des blancs, la première étant le nom d'un type de lexème associé à l'expression régulière qui suit.

Pour illustrer la versatilité de la chose, nous donnons à la Fig. 4.1 la grammaire de ce format de fichier en EBNF.

```

<lexdefs> ::= <line-nl>*
<line-nl> ::= <line> <newline>
<line> ::= <comment>
        | <blank>
        | <def>
<newline> ::= '\n'+
<comment> ::= '#' ^\n*
<blank> ::= (' ' | '\t') +
<def> ::= <lextype> <blank> <regex>
<lextype> ::= <lex-chars> +
<regex> ::= ^[#\n]+
<lex-chars> ::= '_' | ':' | '-' | 'a' .. 'z' | 'A' .. 'Z' | '0' .. '9'

```

FIGURE 4.1 – Grammaire en EBNF du fichier de description de types de lexèmes. On comprend intuitivement que le chargement d'un fichier de définitions de lexèmes (le résultat de l'analyse en partant de <lexdefs>) doit être une liste de couples de chaînes de caractères (<lextype>, <regex>) extraites d'une même ligne.

CODE SOURCE 4.2 – Exemple de fichier de définition de types de lexèmes pour l'analyse lexicale d'un pseudo-langage quelconque – à adapter bien sûr pour notre langage assembleur Python.

```

1  # Lexems as simple regexps (first match gives lexem type!)
2
3  blank          [ \t]+
4  newline        \n+
5  comment        #^\n*
6
7  colon          :
8  semicolon      ;
9
10 # Place keywords before identifiers!
11 keyword::if     if
12 keyword::else   else
13 identifier      [a-zA-Z_][a-zA-Z_0-9]*
14
15 # Numbers
16 # Match floats by longest regexp first!
17 number::float   [0-9]+\.[0-9]*[eE][-+]?[0-9]+
18 number::float   [0-9]+\.[0-9]*
19 number::uint     [0-9]+

```

Notez que nous avons défini des noms de types de lexèmes contenant les deux caractères `::`. Il s'agit d'un moyen facile de grouper les lexèmes qui le mériteraient : par exemple les mots-clefs d'un langage avec `keyword::` et les différents types de nombres avec `number::`. Le cas échéant, nous serions donc capables de réagir à un groupe de types de lexèmes ! On pourrait par exemple n'avoir besoin que de savoir si un lexème est un nombre, sans passer en revue tous les types de nombres.

Dans le Code 4.3, nous mettons en exergue cette différence. Alors que `lexem_type_strict` vérifie si le lexème est *exactement* du type demandé ou pas, la fonction `lexem_type` ne fait que vérifier si le type du lexème *commence* par le type demandé ou pas. En étant un peu moins strict, nous avons donc là un moyen supplémentaire de rendre plus claires à la fois nos définitions de types de lexèmes, et notre code d'analyse *syntaxique* (cette fois). Toute souplesse supplémentaire est bonne à prendre, surtout à si bon marché !

CODE SOURCE 4.3 – Deux manières de tester si un lexème est d'un certain type.

```
1 int lexem_type_strict( lexem_t lex, char *type ) {  
2     return !strcmp( lex->type, type );  
3 }  
4  
5 int lexem_type( lexem_t lex, char *type ) {  
6     return lex->type == strstr( lex->type, type );  
7 }
```

4.3 Analyseur lexical

Il vous faudra donc écrire une fonction `list_t lex(char *lex_defs, char *source)` qui permettra de lire les définitions des types de lexèmes dans le fichier `lex_defs` et de procéder au découpage en lexèmes du code source contenu dans le fichier `source`, pour renvoyer la liste de ces lexèmes.

Nous prendrons donc comme règle que la première expression régulière reconnue est celle qui donnera son type au lexème. Il vous faudra donc veiller à construire votre fichier de définitions de types de lexèmes en plaçant judicieusement ces définitions dans le bon ordre, c'est-à-dire dans celui qui vous permet de reconnaître tels types de lexèmes en priorité, ou d'aller chercher par défaut le plus long (comme le ferait un moteur d'expressions régulières réel).

Lorsqu'aucune expression régulière n'est reconnue à un endroit du code source (en cas d'erreur donc), il convient de lever une erreur indiquant qu'une entrée invalide a été trouvée (et il faudra donner les numéros de ligne et de colonne de l'incident).

Nous donnons ci-dessous un exemple d'exécution de cette fonction `lex` avec les définitions de types de lexèmes du Code 4.2.

```
$ cat output/file.src
# This is a comment!

if 12 :
    print a
else :
    exit 3.14

$ ./bin/lexer.exe output/lexer.conf output/file.src
[1:0:comment] # This is a comment! [1:20:newline]

[3:0:keyword::if] if [3:2:blank] [3:3:number::uint] 12
    [3:5:blank] [3:6:colon] : [3:7:newline]
[4:0:blank] [4:2:identifiant] print [4:7:blank] [4:8:
    identifiant] a [4:9:newline]
[5:0:keyword::else] else [5:4:blank] [5:5:colon] : [5:6:
    newline]
[6:0:blank] [6:2:identifiant] exit [6:6:blank] [6:7:
    number::float] 3.14 [6:11:newline]
```

Il devient alors très facile, par exemple, de filtrer les lexèmes par leur type. Dans l'exemple ci-dessous, nous avons filtré les commentaires, les sauts de ligne et les blancs.

```
$ ./bin/lexer.exe output/lexer.conf output/file.src --no-
    blanks
[3:0:keyword::if] if [3:3:number::uint] 12 [3:6:colon] :
    [4:2:identifiant] print [4:8:identifiant] a [5:0:keyword::
    else] else [5:5:colon] : [6:2:identifiant] exit [6:7:
    number::float] 3.14
```

Chapitre 5

Analyse syntaxique


Notion de grammaire $LL(1)$

Maintenant que nous savons générer des lexèmes à partir d'un texte de code source, il faut encore s'assurer que leur enchaînement respecte la grammaire du langage dont il se réclame.

Lorsque l'on regarde comment est formée une grammaire, on a deux approches possibles pour déterminer si un code source a une syntaxe correcte :

1. l'approche montante : on part des terminaux et on essaye de remonter jusqu'à l'axiome ;
2. l'approche descendante : on part de l'axiome et on essaye de descendre jusqu'aux terminaux.

Les deux approches correspondent chacune à une éventuelle mise en forme (parfois cosmétique, parfois nettement plus lourde) de la grammaire pour se prêter à l'une ou l'autre approche. L'approche montante est adaptée aux grammaires mises en forme dite $LR(1)$ et l'approche descendante aux grammaires mises en forme dite $LL(1)$.

Une grammaire $LL(n)$ (LL pour  *left-lookahead* ou "prévision à gauche") a la propriété que la prochaine règle d'analyse peut être sélectionnée en utilisant au plus les n prochains lexèmes. Toutes les grammaires ne peuvent pas être mises en forme $LL(1)$, mais ce sera le cas des grammaires que nous utiliserons (ou alors nous utiliserons des grammaires qui pourront facilement être mise en forme $LL(1)$).

Donc dans notre cas, l'analyse du prochain lexème suffira pour nous permettre de savoir comment continuer à analyser notre liste de lexèmes pour déterminer si elle suit correctement la syntaxe du langage.

Chapitre 6

Descente récursive

Il existe pléthore de techniques pour analyser une grammaire $LL(1)$, mais nous en choisirons une qui a pour nous l'incomparable avantage d'être facile à programmer et à déboguer : l'analyse par descente récursive.

6.1 Idée générale

L'idée générale de l'analyse par descente récursive consiste à écrire une fonction d'analyse par non-terminal du langage : on va vérifier la syntaxe petit bout par petit bout en quelque sorte. Et comme nous sommes en train de mettre en œuvre une méthode d'analyse descendante, nous lancerons l'analyse syntaxique globale du code source en appelant la fonction correspondant à l'axiome du langage.

Comme les non-terminaux seront encodés par des fonctions en C, elles pourront s'appeler récursivement les unes les autres afin d'encoder les références à d'autres non-terminaux dans la grammaire (ainsi que notre discussion de la Sec. 1.4 a montré qu'il fallait un moyen d'y parvenir).

6.2 Un léger ajout à EBNF

Nous étendons le méta-langage EBNF à l'aide des méta-symboles {...} permettant d'introduire un lexème du type spécifié entre les accolades.

Cet ajout nous permet de formaliser l'obtention de lexèmes par un mécanisme extérieur (en l'occurrence pour nous : notre analyseur lexical).

La partie syntaxique du langage des expressions arithmétiques peut alors s'écrire comme dans la Fig. 6.1, qui suppose que le fichier de configuration de notre analyseur lexical contienne des expressions régulières nommées `op::plus`, `op::minus`, `op::mult`, `op::div` (les 4 opérateurs arithmétiques), `paren::left`, `paren::right` (les deux parenthèses), `number` et `identifier`.

```

<arith-expr> ::= <term> ( ( {op::sum::plus} | {op::sum::minus} ) <term> ) *
<term> ::= <s-factor> ( ( {op::prod::mul} | {op::prod::div} ) <s-factor> ) *
<s-factor> ::= [ {op::sum::plus} | {op::sum::minus} ] <factor>
<factor> ::= {number}
           | {identifiant}
           | {paren::left} <arith-expr> {paren::right}

```

FIGURE 6.1 – Grammaire en EBNF des expressions arithmétiques avec source extérieure de lexèmes de différents types.

6.3 Implantation d'une grammaire $LL(1)$ en C

Une fois qu'une grammaire $LL(1)$ est écrite en EBNF, il reste à l'implanter en C. Pour cela, il suffit de travailler avec les correspondances syntaxiques de la Tab. 6.1, au demeurant bien intuitives. Nous donnerons un exemple de mise en œuvre *infra*.

EBNF	C
concaténation	suite d'instructions
alternative ()	if(...) { ...; return ...; }
répétition (*)	while (...) {...}
répétition (+)	if (!... { return -1; }) + while (...) {...}
option ([...])	if (... {...})
non-terminal	fonction

TABLE 6.1 – Correspondances syntaxiques entre EBNF et C.

Comme on l'a expliqué, tout le charme des grammaires $LL(1)$ est qu'il suffit de pouvoir inspecter le prochain lexème pour continuer l'analyse. Vous trouverez sans doute avantageux à l'usage d'écrire les fonctions suivantes :

1. `lexem_t lexem_peek(list_t *lexems);`
2. `lexem_t lexem_advance(list_t *lexems);`
3. `int next_lexem_is(list_t *lexems, char *type);`
4. `void print_parse_error(char *msg, list_t *lexems);`

Vous vous demandez peut-être pourquoi nous avons besoin de passer notre liste de lexèmes par adresse. La réponse est que devons bien finir par *avancer* dans cette liste, et donc en mettre à jour la tête de liste au fur et à mesure de l'analyse syntaxique.

La fonction `lexem_peek` renverra le prochain lexème *utile* dans la liste de lexèmes, *mais sans l'enlever de la liste* (ce sera notamment l'occasion de

filtrer les blancs et les commentaires, qui eux seront ignorés à la volée par `lexem_peek`). On pourra notamment penser à inscrire dans le nom du type de lexème une information destinée à indiquer si le lexème doit être ignoré ou non (e.g. `blank::skip` et `comment::skip`).

La fonction `lexem_advance` retourne le premier lexème utile de la liste.

La fonction `next_lexem_is` détermine si le prochain lexème utile est bien du type demandé. À vous de voir si vous préférez utiliser `lexem_type_strict` ou `lexem_type` pour déterminer le type (cf. Code. 4.3).

La fonction `void print_parse_error(char *msg, list_t *lexems)` permettra d'afficher le message d'erreur `msg` en le situant grâce aux coordonnées du lexème en tête de la liste `*lexems`.

Regardons maintenant comment appliquer quelques correspondances de la Tab. 6.1. Commençons par l'axiome dans le Code 6.1

CODE SOURCE 6.1 – Implantation de l'axiome de la grammaire des expressions arithmétiques.

```

1  int parse_arith_expr( list_t *lexems ) {
2      printf( "Parsing arithmetic expression\n" );
3
4      if ( -1 == parse_term( lexems ) ) {
5          return -1;
6      }
7
8      while ( next_lexem_is( lexems, "op::sum" ) ||
9              next_lexem_is( lexems, "op::sub" ) ) {
10         lexem_advance( lexems );
11
12         if ( -1 == parse_term( lexems ) ) {
13             return -1;
14         }
15     }
16
17     return 0;
18 }

```

Comme vous le voyez, nous prenons la convention qu'une analyse réussie retourne 0 et qu'un échec de l'analyse sera signalé par -1.

Au regard de la règle de production de l'axiome, une expression arithmétique est un `<term>`, potentiellement concaténé avec un nombre indéfini d'expressions de type `[(op::sum::plus){op::sum::minus}]<term>` (opérateur de répétition `*`). Observez donc dans le Code. 6.1 comment on vérifie d'abord à la ligne 4 que l'on est bien parvenu à lire un `<term>`. Puis, la règle de production de l'axiome étant une concaténation d'un `<term>` avec l'expression `[(op::sum::plus|op::sum::minus)<term>]*`, il nous faudra juste ajouter la vérification de cette dernière partie après celle de la première. Or, il s'agit d'une répétition de type `*`, et suivant la Tab. 6.1 nous utilisons donc un `while` à la ligne 8.

Nous donnons maintenant dans le Code 6.2 l'implantation du non-terminal `<factor>`, qui est composé d'une triple alternative.

CODE SOURCE 6.2 – Implantation du non-terminal `<factor>` de la grammaire des expressions arithmétiques.

```

1  int parse_factor( list_t *lexems ) {
2      printf( "Parsing factor\n" );
3
4      if ( next_lexem_is( lexems, "number" ) ||
5           next_lexem_is( lexems, "identifiant" ) ) {
6          printf( "* Found factor:" );
7          lexem_print( lexem_peek( lexems ) );
8          printf( "\n" );
9          lexem_advance( lexems );
10         return 0;
11     }
12
13     if ( next_lexem_is( lexems, "paren::left" ) ) {
14         lexem_advance( lexems );
15         if ( -1 == parse_arith_expr( lexems ) ) {
16             return -1;
17         }
18
19         if ( next_lexem_is( lexems, "paren::right" ) ) {
20             lexem_advance( lexems );
21             return 0;
22         }
23
24         print_parse_error( "Missing right parenthesis", lexems );
25         return -1;
26     }
27
28     print_parse_error( "Unexpected input", lexems );
29     return -1;
30 }

```

Tout autre lexème qu'un `number`, un `identifiant` ou `paren::left` (parenthèse ouvrante) représente une entrée incorrecte. C'est notre cas par défaut (l. 28). Les deux premières alternatives sont traitées ensemble pour plus de concision (l. 4). La dernière est traitée séparément (l. 13), comme une concaténation. Dès la détection de la parenthèse ouvrante (`paren::left`), on retire ce lexème de la liste (appel à `lexem_advance`). Puis, on vérifie que l'on est bien parvenu à lire une autre `<arith-expr>`, avant de vérifier qu'une parenthèse fermante termine correctement le tout (l. 19).

6.4 Exemple : vérification d'expressions arithmétiques

En utilisant les types de lexèmes du Code 6.3, nous pouvons maintenant programmer un vérificateur de syntaxe pour les expressions arithmétiques.

CODE SOURCE 6.3 – Notre fichier de définition de types de lexèmes pour l'analyse lexicale d'une expression arithmétique.

```

1  # Lexems for arithmetic expressions.
2
3  blank          [ \t]+
4  newline        \n+
5
6  # numbers
7  number::float   [0-9]+\.[0-9]*
8  number::float::exp [0-9]+\.[0-9]*[eE][+\-]?[0-9]+
9  number::integer [0-9]+
10
11 # identifiers
12 identifier      [_a-zA-Z][_a-zA-Z0-9]*
13
14 # arithmetic operators
15 op::sum::plus    \+
16 op::sum::minus   -
17 op::prod::mul     \*
18 op::prod::div     /
19
20 # parentheses
21 paren::left      (
22 paren::right     )

```

Nous donnons ci-dessous un exemple d'analyse d'une expression arithmétique correcte. Notons que ce programme devrait retourner EXIT_SUCCESS au Terminal, puisque la syntaxe est correcte.

```
$ ./demo-parser-arith.exe "-2.2+3*(-x)"
Parsing arithmetic expression
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:1:number::float] 2.2
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:5:number::integer] 3
Parsing signed factor
Parsing factor
Parsing arithmetic expression
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:9:identifiant] x
-2.2+3*(-x) is accepted as an arithmetic expression.
```

Nous donnons à présent un autre exemple, pour une expression arithmétique incorrecte du fait d'une impossibilité de lire les lexèmes (le caractère ' ' n'apparaît jamais dans notre fichier de définitions de types de lexèmes pour l'arithmétique). Notons que ce programme devrait retourner `EXIT_FAILURE`, ou un autre entier non nul, au Terminal, puisqu'il détecte une erreur de syntaxe.

```
$ ./demo-parser-arith.exe "-2.2+3*[(-x)"
[ERROR 1:7] Garbage input.
-2.2+3*[(-x) is *NOT* accepted as an arithmetic expression.
```

Nous donnons enfin un autre exemple, pour une expression arithmétique incorrecte du fait de sa syntaxe cette fois.

```

$ ./demo-parser-arith.exe "-2.2+3*(-x)"
Parsing arithmetic expression
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:1:number::float] 2.2
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:5:number::integer] 3
Parsing signed factor
Parsing factor
[ ERROR 1:7 ] Unexpected input
-2.2+3*(-x) is *NOT* accepted as an arithmetic expression.

Left to parse on exit:  [1:7:op::prod:mul] * [1:8:paren::
    left] ( [1:9:op::sum::minus] - [1:10:identifieur] x [1:11:
    paren::right] )

```

Observez que la sortie situe correctement l'erreur au huitième caractère (numéro 7).

Chapitre 7



Un analyseur syntaxique

Jusqu'à présent, la descente récursive nous a permis d'écrire un *vérificateur* de syntaxe. C'est encourageant, parce que cela a validé notre approche générale, mais cela ne permet pas d'aller beaucoup plus loin tel quel.

En effet, nous voudrions pouvoir, en cas d'analyse syntaxique réussie, *manipuler* l'objet que nous venons de lire. Nous employons le vocable d'*objet* car, vous vous en doutez, une syntaxe peut nous permettre de décrire des choses assez différentes.

Dans l'exemple des expressions arithmétiques que nous filons depuis le début de cette partie, les objets que nous manipulons sont très simples : ce sont des relations arithmétiques entre nombres ou entre identifiants. Mais rien n'empêche, comme vous vous apprêtez à le faire, de *lire un bout de code*. Ou bien on pourrait vouloir lire un fichier décrivant des données écrites avec une certaine syntaxe (on pourrait parfaitement utiliser la descente récursive pour lire un fichier JPEG par exemple – et on aimerait bien récupérer les pixels), *etc.*

Dans le cas général, ces objets que nous lisons en même temps que nous procédons à l'analyse syntaxique devront être représentés en mémoire pour pouvoir être manipulés, et ils formeront entre eux un arbre qualifié d'*arbre de syntaxe abstraite*.

La tâche d'un analyseur syntaxique ( *parser*), plus ambitieuse que celle d'un "simple" vérificateur de syntaxe, consiste à retourner l'arbre de syntaxe abstraite ( *AST – Abstract Syntax Tree*).

7.1 Des objets de syntaxe abstraite

Au fur et à mesure qu'il mène son analyse, notre analyseur syntaxique va devoir créer des objets en mémoire. En toute généralité, il faudra être capable aussi de les afficher et de les détruire. Toute autre opération sur ces objets relèverait d'un autre outil en aval de l'analyseur syntaxique (compilateur, générateur de code, *etc.*) Ces objets devront nécessairement stocker toute l'in-

formation obtenue pendant l'analyse, y compris les informations de lien entre objets (la structure arborescente de la syntaxe analysée).

Ajouter la construction de l'arbre de syntaxe abstraite à notre vérificateur de syntaxe est assez facile, quoique cela demande un peu d'attention. En effet, puisque nous mènerons la construction de l'arbre *et* la vérification de la syntaxe en même temps, nous devrons vérifier dans la fonction appelante d'un non-terminal que la création d'un objet pour un autre non-terminal a été correctement effectuée, et prendre des mesures de libération de la mémoire en conséquence.

7.2 Exemple : syntaxe abstraite des expressions arithmétiques

Sans nous embourber davantage dans des généralités, car vous aurez compris que les informations recueillies pendant l'analyse syntaxique dépendent du type d'objet qu'on est en train de lire, passons à notre exemple des expressions arithmétiques.

Nous donnons dans le Code 7.1 la définition de l'objet que nous utilisons pour représenter une relation arithmétique (ou un nombre ou une variable).

CODE SOURCE 7.1 – Objet permettant de représenter des relations ou des entités arithmétiques (nombres et variables).

```

1  typedef enum {
2      OP_BINARY_PLUS ,
3      OP_BINARY_MINUS ,
4      OP_BINARY_MUL ,
5      OP_BINARY_DIV ,
6      OP_UNARY_MINUS ,
7      OP_UNARY_PLUS ,
8      NUMBER ,
9      VARIABLE
10 } arith_op;
11
12 typedef struct arith_ast {
13     arith_op    op;
14     union {
15         struct {
16             struct arith_ast *left;
17             struct arith_ast *right;
18         }        binary_op;
19         struct {
20             struct arith_ast *expr;
21         }        unary_op;
22         double    value;
23         char      *identifier;
24     }
25 } *arith_ast_t;

```

7.2. EXEMPLE : SYNTAXE ABSTRAITE DES EXPRESSIONS ARITHMÉTIQUES 55

Nous avons logiquement implanté les fonctions suivantes pour créer des objets arithmétiques de différents types :

1. `arith_ast_t arith_binary_op(lexem_t l, arith_ast_t left, arith_ast_t right)`, qui permet de représenter un opérateur arithmétique binaire dont on connaît les deux termes ;
2. `arith_ast_t arith_unary_op(lexem_t l, arith_ast_t factor)`, qui permet de représenter un opérateur unaire (par exemple pour `-42`) ;
3. `arith_ast_t arith_number(lexem_t l)`, qui permet de représenter une constante réelle en virgule flottante ;
4. `arith_ast_t arith_variable(lexem_t l)`, qui permet de représenter le nom d'une variable.

Dans le Code 7.2, nous avons ajouté de quoi construire l'arbre de syntaxe abstraite pour l'axiome de la grammaire des expressions arithmétiques. La structure du code est reprise exactement du Code 6.1. Pour permettre à ce code de construire un tel arbre, nous changeons les valeurs de retour des fonctions des non-terminaux pour retourner à présent un `arith_ast_t` (qui représente un objet arithmétique).

CODE SOURCE 7.2 – Implantation de la construction de l'arbre de syntaxe abstraite pour l'axiome de la grammaire des expressions arithmétiques (*cf.* Code 6.1 pour comparaison).

```
1  arith_ast_t parse_arith_expr( list_t *lexems ) {
2      arith_ast_t left;
3
4      printf( "Parsing arithmetic expression\n" );
5
6      left = parse_term( lexems );
7      if ( NULL == left ) {
8          return NULL;
9      }
10
11     while ( next_lexem_is( lexems, "op::sum" ) ) {
12         arith_ast_t right;
13         lexem_t op_lex = lexem_peek( lexems );
14
15         lexem_advance( lexems );
16
17         right = parse_term( lexems );
18         if ( NULL == right ) {
19             arith_ast_delete( left );
20             return NULL;
21         }
22
23         left = arith_binary_op( op_lex, left, right );
24     }
25
26     return left;
27 }
```

La clef pour comprendre le Code 7.2 consiste à vérifier toujours qu'un appel à une fonction d'un (autre) non-terminal n'a pas rencontré d'erreur (l. 7-18), auquel cas on ne peut continuer l'analyse et on retournera NULL, non sans avoir libéré ce qu'on a lu mais qu'on ne peut utiliser (cf. l'appel à `arit_ast_delete` après la l. 18). La construction de l'arbre syntaxique pour l'axiome implique de reconnaître un opérateur binaire (une addition ou une soustraction), donc nous devons sauver le lexème de l'opérateur (l. 13) car nous devons l'utiliser quand il aura été retiré de la liste des lexèmes (l. 23) afin de pouvoir construire le prochain membre `right` d'une addition ou d'une soustraction.

7.3 Analyseur syntaxique

Stricto sensu, notre analyseur doit retourner l'arbre de syntaxe abstraite lue dans le code source (s'il ne contient pas d'erreurs !) Nous donnons ci-dessous un exemple d'affichage de syntaxe abstraite pour une expression arithmétique.

```
Parsing arithmetic expression
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:1:number::float] 2.2
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:5:number::integer] 3
Parsing signed factor
Parsing factor
Parsing arithmetic expression
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:8:number::integer] 4
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:10:number::integer] 2
-2.2+3*(4-2) is accepted as an arithmetic expression.
  ,-- OP_UNARY_MINUS
  |   '-- NUMBER: 2.2
OP_BINARY_PLUS
  |   ,-- NUMBER: 3
  '-- OP_BINARY_MUL
      |   ,-- NUMBER: 4
      '-- OP_BINARY_MINUS
          '-- NUMBER: 2
```


Le travail de l'analyseur syntaxique s'arrête ici. Une fois qu'une syntaxe abstraite est disponible en mémoire, plusieurs options s'offrent à nous :

1. si nous venons de lire une expression arithmétique, nous pouvons par exemple l'*évaluer*, comme nous verrons ci-dessous ;
2. si nous venons de lire un fichier de données, nous pouvons *accéder* à son contenu ;
3. si nous venons de lire un code source, nous pouvons l'*interpréter*, le *compiler* ou le *formater*.

Pour finir, voici un exemple d'*évaluation* d'une expression arithmétique. Notez que cette évaluation a lieu dans un *environnement*, contenant les variables *y* et *x* récupérées depuis la ligne de commande.

Nous auront de fait, pour d'autres raisons, un environnement à maintenir à jour pour la plupart de nos manipulations ultérieures de syntaxe abstraite.

```
$ ./demo-parser-arith-ast.exe "-2.2+3*(4-x)" --eval y 2 x 5
Parsing arithmetic expression
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:1:number::float] 2.2
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:5:number::integer] 3
Parsing signed factor
Parsing factor
Parsing arithmetic expression
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:8:number::integer] 4
Parsing term
Parsing signed factor
Parsing factor
* Found factor: [1:10:identifiant] x
-2.2+3*(4-x) is accepted as an arithmetic expression.
,-- OP_UNARY_MINUS
|   '-- NUMBER: 2.2
OP_BINARY_PLUS
|   ,-- NUMBER: 3
'-- OP_BINARY_MUL
    |   ,-- NUMBER: 4
    '-- OP_BINARY_MINUS
        '-- VARIABLE: x
Eval with vars: y = 2, x = 5.
=> -5.2
```


Deuxième partie

Assembleur Python


Dans cette partie, nous présentons brièvement le modèle d'exécution de Python et nous précisons où ce projet se situerait dans la foule de techniques mobilisées pour implanter complètement Python.

Chapitre 8

Sur le fonctionnement de Python

Le langage Python a plusieurs modes de fonctionnement, dont le plus connu est l'interpréteur. Nous donnons ci-dessous un exemple de session dans l'interpréteur Python.

```
$ python
Python 2.7.18 (default, Mar  8 2021, 13:02:45)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> 4+2
6
>>> exit()
$
```

Par défaut, la commande `python` lance l'interpréteur et présente son invite de commande (`>>>`) à l'utilisateur, qui entre le texte de ses commandes. L'interpréteur exécute la commande, affiche le résultat et se met derechef en attente de la prochaine commande de l'utilisateur. On boucle donc toujours sur la même séquence de lecture, exécution (évaluation), et affichage – ce que les Anglo-Saxons nomment  *Read-Eval-Print Loop* – *REPL*.

8.1 Boucle REPL de Python

Vous vous en doutez, derrière cette mécanique simple en apparence se cache un monstre d'ingénierie informatique. Nous présentons à la Fig. 8.1 une version simplifiée et adaptée au projet de la boucle principale de l'interpréteur Python.

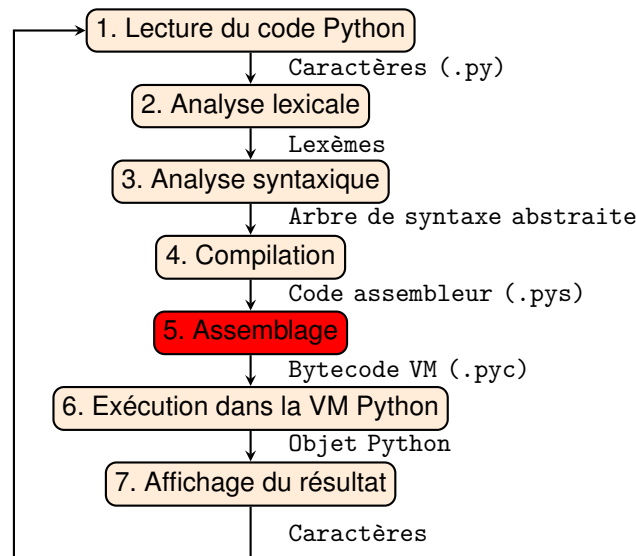



FIGURE 8.1 – Boucle simplifiée de l'interpréteur Python. En rouge : ce que nous nous proposons de réaliser dans ce projet.

L'étape 1 consiste à récupérer le texte du code Python à exécuter : il peut provenir essentiellement d'une entrée de l'utilisateur dans l'interpréteur ou bien d'un fichier dit de script (fichier `.py`).

Les étapes 2 et 3 sont décrites dans la Partie I : elles visent à obtenir une représentation exacte du code qui soit manipulable en mémoire.

L'étape 4 est une des deux plus complexes (avec la 6) : elle vise à produire un code en *assembleur* Python, au format `.pys`, qui représente le code de départ mais sous forme d'instructions élémentaires de la machine virtuelle ( *Virtual Machine* – VM) Python. Ce code assembleur est lisible par un humain : il est au format texte.

L'étape 5 est celle que nous vous proposons de réaliser dans ce projet. Elle vise donc, à partir d'un fichier texte décrivant un code en assembleur Python, à le traduire en un fichier qui sera exécutable par la machine virtuelle Python (fichier `.pyc`) et qui contiendra le code sous forme de *bytecode*. Ce fichier ne sera plus lisible par un humain mais seulement par Python.

L'étape 6 met en œuvre l'*environnement d'exécution* du bytecode et supervise son exécution jusqu'à sa terminaison. Son résultat est un objet Python.

L'étape 7 rend compte à l'utilisateur de l'exécution de sa commande.

En réalité, Python ne va pas écrire sur disque un fichier assembleur au format `.pys` : il va plutôt fusionner les étapes 4 et 5 pour les exécuter en mémoire. Mais on peut faire lire, écrire et exécuter à Python des fichiers `.pyc`.

Au reste, la grammaire de Python est publique, mais tout ce qui relève du bytecode (son format, son exécution, *etc.*) est considéré comme un *détail d'im-*

plantation – et n'est donc pas documenté.

8.2 Rappel : compilation, assemblage et édition des liens

Lorsque, comme dans l'extrait ci-dessous, vous compilez du code C, le fichier exécutable obtenu est écrit dans un format (le format ELF sous Linux) dont une partie constitue le code à exécuter, obtenu en procédant à l'édition des liens des fichiers objet (.o). Le code exécutable est constitué d'instructions machine : des octets que la machine sait décoder pour les exécuter.

```

$ cat totor.c

int main ( ) {

    return 4+2;
}

$ gcc -S totor.c -o totor.s
$ cat totor.s
        .file      "totor.c"
        .text
        .globl     main
        .type      main, @function
main:
        endbr64
        pushq      %rbp
        movq       %rsp, %rbp
        movl       $6, %eax
        popq       %rbp
        ret
        [...]
$ as totor.s -o totor.o
$ gcc totor.o -o totor
$ objdump -d totor
        [...]
0000000000401110 <main>:
    401110: 55                      push    %rbp
    401111: 48 89 e5                mov     %rsp,%rbp
    401114: c7 45 fc 00 00 00 00    movl    $0x0,-0x4(%rbp)
    40111b: b8 06 00 00 00          mov     $0x6,%eax
    401120: 5d                      pop     %rbp
    401121: c3                      retq
        [...]
$ ./totor
$ echo $?
6
$

```

Dans l'extrait ci-dessus, ce code exécutable et ses instructions apparaissent sous deux formes dans les dernières lignes : tout à droite nous avons le code en langage assembleur, et au centre, nous avons les octets qui codent ces instructions. Par exemple, l'instruction en assembleur `mov $0x6,%eax` (donc lisible par un humain, ce que vous venez de faire) est encodée en les octets `b8 06 00 00 00` (qui ne sont donc plus un texte lisible par un humain, mais adaptés à ce que sait lire et exécuter la machine).

C'est le programme `objdump -d` (le *désassembleur*) qui a l'obligeance de bien vouloir nous écrire en langage assembleur (que nous pouvons lire) les octets contenus dans le code exécutable. Et le travail du compilateur, même

8.3. ÉLÉMENTS D'ASSEMBLEUR (DIRECTIVES, ÉTIQUETTES, INSTRUCTIONS) 67

si cela vous est caché en temps normal, consiste à traduire le fichier de code C en un fichier en langage assembleur dédié à une machine donnée (dans votre cas, très probablement un PC équipé d'un processeur x86_64) et à un environnement d'exécution (Linux). Vous pouvez obtenir ce fichier en langage assembleur pour PC avec `gcc -S toto.c`, et vous pourrez le traduire au format binaire ELF à l'aide de la commande `as` (comme ci-dessus) pour en faire le fichier objet `toto.o`. Le programme `as` s'appelle un assembleur : "un assembleur est utilisé pour traduire un programme écrit en *langage assembleur* (de l'assembleur) en un format exécutable par la machine".

8.3 Éléments d'assembleur (directives, étiquettes, instructions)

8.3.1 Directives et étiquettes

Observons que le code en assembleur (contenu dans le fichier `toto.s`) semble contenir davantage d'informations que l'on en retrouve dans la version binaire. Ces informations ne sont destinées, justement, qu'au *programme assembleur* (`as`) et sont :

1. soit des *directives*, dont le nom débute par un point (`.global`, `.text`, *etc.*), qui permettent essentiellement de *structurer le code binaire* produit ;
2. soit des *étiquettes*, dont le nom est suivi de deux points (`main:`), qui permettent essentiellement de *structurer le code assembleur* en permettant des références à des *adresses* dans le code (*où* débute l'alternative `else` d'un `if` ? *où* trouver le code binaire de telle fonction ? *etc.*)

Nous suivrons dans ce projet exactement la même idée de structuration par directives et étiquettes.

8.3.2 Instructions machine

Une machine, virtuelle ou pas, ne sait exécuter qu'un certain nombre d'instructions (son *jeu* d'instructions), qui permettent de modifier l'état de la machine pour exécuter un programme. L'état d'une machine est défini par la valeur de ses registres et de sa mémoire. Une instruction machine doit donc être capable de spécifier un registre ou une adresse mémoire dont on veut changer l'état, voire une constante entière.

Si l'on prend l'exemple de l'instruction `mov $0x6,%eax`, qui sert à écrire la valeur 6 exprimée sur 32 bits dans le registre `%eax`, on constate qu'une fois encodée en binaire (`b8 06 00 00 00`), son premier octet (`b8`) sert à coder (*i*) qu'il s'agit de l'instruction `mov` (*ii*) avec le registre `%eax` comme destination de l'écriture – et on note au passage qu'il s'agit d'un encodage plutôt condensé, à l'échelle du bit, et relativement complexe. La valeur 6 quant à elle est codée sur les quatre octets suivants – c'est un encodage simple, à l'échelle de l'octet.

Une instruction machine est définie par :

1. sa *mnémonique* : son nom utilisable par un humain (`mov`) ;
2. son *opcode* : le motif binaire (un numéro mais pas toujours) qui permet à une machine de l'identifier (l'opcode est pour la machine ce que la mnémonique est pour l'humain) ;
3. ses *paramètres* (registre, adresse mémoire, constante entière) ;
4. son *format* : la manière d'encoder ses paramètres en binaire, qui dépend d'ailleurs souvent de l'opcode (et toutes les instructions une fois écrite en binaire ne font pas nécessairement la même taille comme dans l'exemple *supra*).

8.4 Notions de machine virtuelle et de bytecode

Le grand avantage de la méthode ci-dessus est que le code binaire obtenu à partir du code C est optimisé pour la machine qui va devoir l'exécuter (un PC `x86_64` de l'école) *et* pour l'environnement d'exécution (Linux dans notre cas). C'est à ces conditions que nous obtenons le code le plus rapide possible. Néanmoins, l'environnement d'exécution de Linux est différent de celui de Windows, lui-même différent de celui de MacOS, *etc.* Outre qu'il faudrait donc recompiler notre code pour l'exécuter dans ces différents environnements, ils ont surtout tous en commun de ne finalement permettre d'exécuter *que* les instructions dumicro-processeur du PC. Et il se trouve que ces instructions ne permettent pas d'exprimer nativement des tâches complexes (de gestion de la mémoire et du contrôle de l'exécution notamment), comme en ont besoin la plupart des langages visant à exonérer l'utilisateur de la gestion de la mémoire ou à lui permettre un contrôle fin de l'exécution de son code.

8.4.1 Machine virtuelle

Une solution à ce problème consiste à accepter de perdre (beaucoup) en rapidité d'exécution pour gagner en expressivité : on va imaginer et programmer un logiciel qui va se comporter comme une machine, et on parlera donc de machine virtuelle.¹ Cette machine sera dotée d'un jeu d'instructions permettant les fonctionnalités complexes souhaitées. Pendant leur exécution dans la machine virtuelle, ces instructions sont exécutées par des fonctions écrites en C, qui implantent donc la machine virtuelle sur la machine physique. En conséquence, il ne faudrait recompiler que le code C de la machine virtuelle si l'on voulait exécuter du code Python sur un autre système d'exploitation (Linux, MacOS, Windows, *etc.*)

1. L'informatique, en un sens assez large, permet de travailler sur des abstractions, fussent-elles de machines. On virtualise à peu près tout de nos jours. Mais suivant le contexte, on ne donnera pas la même signification au terme de machine virtuelle. Dans un autre contexte, une machine virtuelle est ce qui vous permet d'exécuter par exemple un Windows dans un Linux. Mais pour nous, il n'y aura de machine virtuelle que pour exécuter du code Python.

8.4.2 Bytecode

Les instructions de la machine virtuelle (VM) Python sont assez simples et relativement peu nombreuses, mais elles diffèrent d'une version de la VM Python sur l'autre : elles sont souvent les mêmes à l'intérieur d'une version majeure de Python (par exemple Python 2, versions 2.6 ou 2.7) mais pas entre versions majeures. Ainsi, le passage à Python 3.x a vu l'introduction d'un jeu d'instructions sensiblement modifié par rapport à celui de Python 2.7.

Nous avons donc des instructions peu nombreuses et qui doivent être décodées (“comprises”) rapidement par du *logiciel* (celui de la machine virtuelle – réputé bien plus lent qu'une machine physique, dans laquelle c'est le micro-processeur qui procède au décodage des instructions). Or, la plus petite unité qu'une machine permet de lire et écrire efficacement en mémoire est un octet. Lire ou écrire la valeur d'un bit dans un octet demande de faire des calculs supplémentaires (décalages de bits, “et” et “ou” logiques, *etc.*) Ainsi, concevoir l'encodage des instructions de la machine virtuelle par octets permet-il de générer un code exécutable qui sera plus rapidement décodé par le logiciel d'une machine virtuelle (ici, on sacrifie de l'espace mémoire pour ne pas perdre davantage en rapidité). C'est pour cette raison que le code exécuté par une machine virtuelle est qualifié de *bytecode* (“code binaire conçu par octets”).

8.5 Génération et exécution de bytecode Python

Pour générer le bytecode d'un bout de code Python, on peut utiliser le module Python `compileall`. La trace ci-dessous montre comment faire et comment ensuite exécuter le bytecode Python (avec exactement la même syntaxe que vous auriez employée pour du code Python non compilé). De la même manière que `gcc -c` va générer un fichier `.o` à partir d'un fichier `.c`, la commande `python2.7 -m compileall` va générer un fichier de bytecode `.pyc` à partir d'un fichier `.py`. Et on peut enfin l'exécuter avec `python totor.pyc` (ce qui nous fera démarrer à l'étape 6 de la Fig. 8.1, alors qu'on aurait démarré à l'étape 1 avec la commande `python totor.py`).

```
$ ls *.py*
totor.py
$ cat totor.py
exit( 4+2 )
$ python -m compileall totor.py
Compiling totor.py ...
$ file totor.pyc
totor.pyc: python 2.7 byte-compiled
$ ls -l totor.pyc
-rw-rw-r-- 1 cayre cayre 129 juin  8 15:19 totor.pyc
$ python totor.pyc
$ echo $?
6
$
```

8.6 Génération de code assembleur Python

Dans ce projet, votre programme devra accepter en entrée un fichier `.pys` contenant du code assembleur Python, et générer un fichier bytecode `.pyc`, que la commande `python` devra pouvoir exécuter. Il vous sera possible d'écrire un fichier `.pys` en langage assembleur Python à la main – et ce sera une bonne idée de le faire parfois, pour tester votre programme.

Toutefois, il vous est aussi possible de générer un fichier `.pys` à partir d'un fichier `.pyc`, au moyen de la commande `pyc-objdump`, que nous fournissons (cf. Annexe D). Ainsi donc, vous pourrez écrire du code en Python (fichier `.py`), générer un fichier *bytecode* avec la commande `python2.7 -m compileall <fichier.py>`, puis générer le fichier assembleur correspondant (celui que traitera votre programme) avec `pyc-objdump`, comme le montre l'exemple suivant :

```
$ python2.7 -m compileall totor.py
Compiling totor.py ...
$ pyc-objdump disasm totor.pyc > totor.pys
$
```

Chapitre 9

Machine virtuelle (VM) Python

La machine virtuelle (VM) Python est chargée d'exécuter le bytecode produit par le compilateur (dans notre cas : par l'assembleur). Elle ne sait par ailleurs gérer *que* des *objets* Python. Les objets Python peuvent être des entiers, des réels, des chaînes de caractères, des fonctions, *etc.* mais aussi du *code* : en Python, une fonction est un objet qui référence du code à exécuter (entre autres informations comme la valeur des arguments – une fonction Python est plutôt à comprendre comme un environnement d'exécution d'un objet de type code) Nous décrivons brièvement ici quelques grandes lignes de la VM Python.

9.1 Évaluation de code avec une pile

La VM Python est une machine à *pile*, comme pour Java, Lua et une foule d'autres langages, ce qui signifie qu'elle n'a qu'un seul registre (le compteur ordinal, qui contient l'adresse dans le bytecode de la prochaine instruction à exécuter) et qu'elle utilise donc une pile pour l'exécution (l'évaluation) d'un bout de code.

Cette pile est appelée *pile d'évaluation* et est notée S . Par exemple, pour évaluer $4+2$, il faudra empiler 4, puis 2 et enfin +. On a donc $S = +, 2, 4$ (le sommet de la pile étant à gauche). Au moment de l'évaluation, la VM va donc devoir exécuter l'opération au sommet de la pile (+). Le code implantant + en une fonction C va alors dépiler 4 puis 2, effectuer l'addition, et empiler le résultat. À la fin de l'évaluation, on a donc $S = 6$.

9.2 État global de la VM

Outre la pile d'évaluation S , la VM Python utilise deux autres piles :

- la pile d'appels de fonctions C ;
- la pile de blocs de code B .

Le langage C aussi utilise une pile pour les appels de fonctions. Elle sert à savoir où continuer l'exécution après qu'une fonction a terminé son exécution. On empile dans C des *cadres de pile* : des structures qui contiennent, entre autre, l'adresse de retour de la fonction courante. On trouve donc dans la pile d'appels de fonctions d'autant plus de cadres de pile qu'on compte de fonctions *actives* : les fonctions qui, à chaque instant, ont été appelées mais n'ont pas encore terminé leur exécution.

La pile de blocs de code B est par contre étrangère au langage C : on y empile et dépile des informations sur des blocs de code en cours d'exécution, correspondant à certaines constructions syntaxiques particulières. Nous nous limiterons à considérer des blocs de code correspondant à des boucles (`while`) bien que leur autre grande utilisation concerne un aspect fondamental de Python que nous passons courageusement sous silence : les exceptions.

En notant IP le compteur ordinal¹, l'état global G de la VM Python est donné par le quadruplet $G = \{IP, S, B, C\}$. Dans le cadre de ce projet, nous utiliserons surtout S (et B dans une moindre mesure), aussi l'Annexe B donnera-t-elle principalement l'évolution de S suite à l'exécution d'une instruction de la VM.

9.3 Objets Python en C

La VM Python ne sait manipuler que des objets Python. Il faut donc être capable de travailler avec leur représentation en C. La définition en C d'un véritable objet Python est assez complexe, spécialement concernant les types des objets, mais comme nous ne serons pas confrontés à la réalisation de la VM, nous pourrions simplifier considérablement la définition que nous utiliserons.

Nous donnons à la Fig. 9.1 le début de notre définition utilisée dans ce projet.

Quelques commentaires s'imposent. Notre type `pyobj_t` est défini comme un *pointeur* vers une structure qui est dite *étiquetée* car composée principalement d'un entier (l'étiquette) et d'une union (la description de chaque type Python en C).

1. IP pour *Instruction Pointer* : le registre qui contient l'adresse de la prochaine instruction à exécuter.

CODE SOURCE 9.1 – Début de définition du type C pour les objets Python.

```

1  typedef unsigned int pyobj_type;
2
3  struct pyobj_t;
4  typedef struct pyobj *pyobj_t;
5
6
7  struct pyobj {
8
9      pyobj_type      type;
10     unsigned int     refcount;
11
12     union {
13         struct {
14             pyobj_t      *value;
15             int32_t       size;
16         }               list;
17
18         struct {
19             char          *buffer;
20             int           length;
21         }               string;
22
23         py_codeblock      *codeblock;
24
25         union {
26             int32_t        integer;
27             int64_t        integer64;
28             double         real;
29             struct {
30                 double     real;
31                 double     imag;
32             }             complex;
33         }               number;
34     }                  py;
35 };

```

L'étiquette en question est le champ `type` (l. 9) : en fonction de sa valeur, on ira lire/écrire le champ correspondant de l'union `py` (démarrant l. 12). Par exemple, si l'on sait que l'on est en train de manipuler une chaîne de caractères Python dans l'objet `foo`, on accèdera à sa longueur avec `foo->py.string.length`.

Observez encore (l. 16) qu'une liste est stockée sous forme de tableau pour suivre ce que Python utilise comme représentation interne.² Nous avons également choisi une `union` pour les nombres (l. 33), mais ce n'était pas strictement nécessaire : cela sert ici uniquement à clarifier l'écriture du code C.

Enfin, nous avons dit plus haut que Python permet de stocker et manipuler du code. Pour cela, nous utilisons un *pointeur* vers une structure de type

2. Et avec la notation Python [...] pour introduire une liste, vous avez tous les éléments pour comprendre la confusion que ce langage introduit entre listes et tableaux. Mais il est vrai que Python est un langage pour les non-informaticiens...

`py_codeblock` (l. 23). En effet, stocker toute la structure plutôt que son adresse reviendrait à allouer beaucoup trop de mémoire pour chaque objet et nous choisissons donc de traiter ce type Python à part. Pour la définition du type `py_codeblock`, cf. Sec. 9.5.

Notez que le Code 9.1 ne donne pas la structure complète du type `pyobj_t`. Il vous reviendra de la compléter pour tenir compte des types manquants.

9.4 Nombre de références à un objet

Dans ce projet, il pourra vous arriver de mettre le même objet dans plusieurs listes et/ou files. Et au moment de détruire la liste ou la file, il faudrait que vous réfléchissiez si oui ou non il vous faut libérer la mémoire utilisée par un objet (par exemple, si des objets sont référencés à la fois par une liste et par une file, et que vous détruisez la liste, il ne faut pas que les objets soient libérés parce qu'ils sont aussi référencés par la file – il ne faudrait les détruire effectivement qu'au moment de la destruction ultérieure de la file).

Nous vous proposons un système simple pour uniformiser votre code. En effet, dans la Fig. 9.1, nous avons ajouté un champ `refcount` (l. 10), qui compte le nombre de fois qu'un objet Python est référencé par une structure (liste, pile, etc.) Lorsque vous insérerez (*resp.* retirerez) un objet dans une structure de données, il vous suffira d'incrémenter (*resp.* décrémenter) ce nombre de références. Lorsqu'il atteint zéro, alors on pourra effectivement le supprimer sans crainte. La fonction de destruction d'un objet devra donc vérifier la valeur de `refcount` avant de décider si oui ou non il faut appeler `free`.

9.5 Objet de type code

Nous donnons à la Fig. 9.2 une définition possible d'un objet de code Python utilisable pour décrire toutes les versions possibles d'un tel objet de code Python. Nous avons choisi de rassembler dans cette structure, entre autres, à la fois les informations concernant :

- l'en-tête du fichier `.pyc` (l'union l. 38) ;
- l'en-tête de l'objet de code lui-même (l'union l. 18) ;
- les tuples définissant les chaînes internées, les constantes, les noms de variables, etc. et le bytecode sous forme de chaîne d'octets (la structure l. 24) ;
- le code source Python (la structure l. 30).

CODE SOURCE 9.2 – Définition d'un objet Python de type code pour toutes les versions de Python.

```

1  typedef struct {
2      int          version_pyvm;
3      union {
4          struct {
5              uint32_t    arg_count;
6              uint32_t    local_count;
7              uint32_t    stack_size;
8              uint32_t    flags;
9          }           _0; /* Up to 3.7 (2.7, etc.) */
10         struct {
11             uint32_t    arg_count;
12             uint32_t    posonly_arg_count;
13             uint32_t    kwnonly_arg_count;
14             uint32_t    local_count;
15             uint32_t    stack_size;
16             uint32_t    flags;
17         }           _1; /* From 3.8 onwards */
18     }           header;
19     pyobj_t      parent;
20     struct {
21         union {
22             struct {
23                 uint32_t    magic;
24                 time_t      timestamp;
25             }           _0;
26             struct {
27                 uint32_t    magic;
28                 time_t      timestamp;
29                 uint32_t    source_size;
30             }           _1;
31             struct {
32                 uint32_t    magic;
33                 uint32_t    bitfield;
34                 time_t      timestamp;
35                 uint32_t    source_size;
36                 uint64_t    hash;
37             }           _2;
38         }           header;
39         struct {
40             pyobj_t      interned;
41             pyobj_t      bytecode;
42             pyobj_t      consts;
43             pyobj_t      names;
44             pyobj_t      varnames;
45             pyobj_t      freevars;
46             pyobj_t      cellvars;
47         }           content;
48         struct {
49             pyobj_t      filename;
50             pyobj_t      name;
51             uint32_t      firstlineno;
52             pyobj_t      lnotab;
53         }           trailer;
54     }           binary;
55 } py_codeblock;

```

Vous pouvez bien évidemment adapter cette définition si vous le souhaitez, par exemple en l'expurgeant pour ne pouvoir traiter que la version 2.7 de Python (cf. Fig. 9.3).

CODE SOURCE 9.3 – Objet de code Python 2.7 uniquement.

```
1  typedef struct {
2      int         version_pyvm;
3      struct {
4          uint32_t  arg_count;
5          uint32_t  local_count;
6          uint32_t  stack_size;
7          uint32_t  flags;
8      }           header;
9      pyobj_t      parent;
10     struct {
11         struct {
12             uint32_t  magic;
13             time_t    timestamp;
14             uint32_t  source_size;
15         }           header;
16         struct {
17             pyobj_t    interned;
18             pyobj_t    bytecode;
19             pyobj_t    consts;
20             pyobj_t    names;
21             pyobj_t    varnames;
22             pyobj_t    freevars;
23             pyobj_t    cellvars;
24         }           content;
25     }           struct {
26         pyobj_t    filename;
27         pyobj_t    name;
28         uint32_t    firstlineno;
29         pyobj_t    lnotab;
30     }           trailer;
31     }           binary;
32 } py_codeblock;
```

Chapitre 10

Spécification du format `.pys` (assembleur Python)

Le format de fichier assembleur Python, d'extension `.pys`, n'existe nulle part ailleurs dans le monde que dans ce projet, car nous sommes manifestement parmi les rares sur Terre à trouver un si grand intérêt aux détails d'implantation de Python (ce qui constitue pourtant une grande source d'enseignements). Car, comme on l'a dit, les implantations de Python fusionnent en mémoire les étapes de compilation et d'assemblage, et se passent donc de l'écriture sur disque d'un quelconque fichier en "assembleur Python" (qui n'existe donc que chez nous).

Néanmoins, ce que nous vous proposons dans ce projet a un double mérite : en établissant un langage assembleur Python, nous nous rapprochons de ce que vous mettez en œuvre lors de l'utilisation d'un compilateur C, et donc vous pouvez commencer à prendre du recul sur ce que vous avez déjà fait à l'école, mais surtout, cela devrait vous faire envisager les choses en intégrant de nouveaux paramètres.

En effet, vous aurez compris que l'implantation d'un langage comme Python revient en réalité à programmer tout un éco-système de programmes qui savent compiler, exécuter, déboguer et optimiser des programmes écrits dans ce langage.

Si l'on se plaçait du point de vue très terre-à-terre de la maîtrise des coûts de développement, il n'aurait sans doute pas été mauvais de définir un tel langage assembleur Python. En effet, cela aurait permis à l'équipe travaillant sur le compilateur de commencer à travailler en sachant quoi produire (la traduction du code Python en code assembleur Python), en même temps que l'équipe travaillant sur la mise au point de la machine virtuelle pouvait expérimenter différents encodages – et cela en ayant un certain confort de travail puisque le langage assembleur est un texte lisible par un humain. Mais le monde du logiciel libre ne fonctionne pas toujours comme cela !

10.1 Unités lexicales

Le langage assembleur Python utilise les types suivants de lexèmes.

En commençant par ceux structurant le texte du code source en assembleur :

1. `comment` : un commentaire introduit par `#` (lui-même échappé par `\#`), jusqu'à la fin de la ligne – caractère(s) de retour à la ligne exclus ;
2. `blank` : un blanc, entendu comme une suite d'espace(s) et/ou de tabulation(s) horizontale(s) ;
3. `newline` : au moins un retour à la ligne ;

Puis en ayant de quoi décrire des constantes de différents types :

4. `paren::left` : une parenthèse ouvrante ;
5. `paren::right` : une parenthèse fermante ;
6. `brack::left` : un crochet ouvrant ;
7. `brack::right` : un crocher fermant ;
8. `integer::dec` : un entier signé en décimal ;
9. `integer::hex` : un entier non signé en hexadécimal et préfixé par `0x` ;
10. `float` : un nombre réel en virgule flottante (le séparateur décimal étant un point), potentiellement en notation exponentielle ;
11. `string` : une chaîne de caractères entre double-guillemets¹ ;
12. `symbol` : un symbole constitué de lettres, de chiffres et de `_` mais ne pouvant avoir un chiffre pour premier caractère ;
13. `pycst::None`, `pycst::True` et `pycst::False` : les constantes Python `None`, `True` et `False` respectivement ;

Les mnémoniques du jeu d'instructions du bytecode :

14. `insn::nparms` : une mnémonique du bytecode nécessitant `nparms` paramètre(s) ;

Des directives pour structurer le bytecode :

15. `dir::foo` : une directive (`foo` pour l'exemple), introduite par `.foo` dans le code assembleur (son nom précédé d'un point), et dont le nom est constitué comme un `{symbol}` ;

Et enfin de quoi former des étiquettes pour structurer le code assembleur Python :

1. Si une chaîne de caractères contient des guillemets, ils devront donc être échappés par `\`.

16. `colon` : deux points (verticaux `:`).

Comme nous utilisons `lexem_type`, nous pouvons écrire `integer` pour désigner `integer::dec` ou `integer::hex` indifféremment. De même, `insn` peut-il être utilisé pour désigner n'importe quelle instruction du bytecode Python, indifféremment de son nombre de paramètres.

10.2 Grammaire formelle de l'assembleur Python

Nous supposons une source externe de lexèmes (notre analyseur lexical!) pour les types énoncés ci-dessus pour lire correctement la grammaire du langage assembleur Python donnée aux Figs. 10.1– 10.2. Remarquez que cette grammaire contient quelques structures syntaxiques similaires que vous mettez à profit pour factoriser votre code.

Nous donnons plus loin (Code 10.2) un exemple de code assembleur Python valide, pour que vous puissiez faire le lien avec la grammaire ci-dessous.

10.3 Directives

10.3.1 `.set`

La directive `.set` sert à fixer les valeurs de :

1. `version_pym` : la version du bytecode à produire (cf. Sec. 11.3);
2. `flags` : les options du bytecode (cf. Sec. C.4.2);
3. `filename` : le nom du fichier source `.py`;
4. `name` : la chaîne constante "`<module>`";
5. `source_size` [Python 3.x] : la taille du code source (`.py`) en octets;
6. `stack_size` : la taille maximale de la pile d'exécution à prévoir par la VM Python.
7. `arg_count` : le nombre total d'arguments du code;
8. `kwnonly_arg_count` [Python 3.x] : le nombre d'arguments de type mot-clé du code;
9. `posonly_arg_count` [Python 3.x] : le nombre d'arguments positionnels du code.

10.3.2 `.interned`

Python utilise une technique destinée à économiser de la mémoire : les chaînes de caractères constantes sont dites *internées*, en ce qu'elles sont stockées dans un tableau à part pendant l'exécution et qu'à chaque fois qu'on a besoin de l'une d'elle, il suffit de la référencer dans ce tableau (d'utiliser

```

<pys> ::= <eol>* <prologue> <code>

<prologue> ::= <set-directives> <interned-strings> <constants> [<names>]
               [<varnames>] [<freevars>] [<cellvars>]

<set-directives> ::= <set-version-pyvm> <set-flags> <set-filename> <set-name>
                    [<set-source-size>] <set-stack-size> <set-arg-count> [set-kwonly-arg-count]
                    [set-posonly-arg-count]

<set-version-pyvm> ::= {'dir::set'} {'blank'} {'version_pyvm'} {'blank'}
                     {'integer::dec'} <eol>

<set-flags> ::= {'dir::set'} {'blank'} {'flags'} {'blank'} {'integer::hex'} <eol>

<set-filename> ::= {'dir::set'} {'blank'} {'filename'} {'blank'} {'string'} <eol>

<set-name> ::= {'dir::set'} {'blank'} {'name'} {'blank'} {'string'} <eol>

<set-source-size> ::= {'dir::set'} {'blank'} {'source_size'} {'blank'}
                     {'integer::dec'} <eol>

<set-stack-size> ::= {'dir::set'} {'blank'} {'stack_size'} {'blank'}
                     {'integer::dec'} <eol>

<set-arg-count> ::= {'dir::set'} {'blank'} {'arg_count'} {'blank'}
                   {'integer::dec'} <eol>

<set-kwonly-arg-count> ::= {'dir::set'} {'blank'} {'kwonly_arg_count'}
                          {'blank'} {'integer::dec'} <eol>

<set-posonly-arg-count> ::= {'dir::set'} {'blank'} {'posonly_arg_count'}
                           {'blank'} {'integer::dec'} <eol>

```

FIGURE 10.1 – Grammaire en EBNF du langage assembleur Python.

son numéro) plutôt que d'en produire une nouvelle copie en mémoire pendant l'exécution.

La directive `.interned` est donc suivie des chaînes de caractères constantes, listées une par ligne et par numéro d'ordre avec lequel on s'attend à pouvoir les référencer. La table des chaînes internées est donc la table des chaînes de caractères constantes.


```

<interned-strings> ::= {'dir::interned'} <eol> ( {'string'} <eol> )*
<constants> ::= {'dir::consts'} <eol> ( <constant> <eol> )*
<constant> ::= {'integer'} | {'float'} | {'string'} | {'pycst'} | <tuple>
<list> ::= {'brack::left'} ( {'blank'} <constant> )* [ {'blank'} ] {'brack::right'}
<tuple> ::= {'paren::left'} ( {'blank'} <constant> )* [ {'blank'} ] {'paren::right'}
<names> ::= {'dir::names'} <eol> ( {'string'} <eol> )*
<varnames> ::= {'dir::varnames'} <eol> ( {'string'} <eol> )*
<freevars> ::= {'dir::freevars'} <eol> ( {'string'} <eol> )*
<cellvars> ::= {'dir::cellvars'} <eol> ( {'string'} <eol> )*
<code> ::= {'dir::text'} <eol> ( <assembly-line> <eol> )*
<assembly-line> ::= <insn>
| <source-lineno>
| <label>
<label> ::= {'symbol'} {'blank'} {'colon'}
<source-lineno> ::= {'dir::line'} {'blank'} {'integer::dec'}
<insn> ::= {'insn::0'}
| {'insn::1'} ( {'integer::dec'} | {'symbol'} )
<eol> ::= ([ {'blank'} ] [ {'comment'} ] [ {'newline'} ] [ {'blank'} ] )*

```

FIGURE 10.2 – Grammaire en EBNF du langage assembleur Python (suite).

10.3.3 .consts

La directive `.consts` sert à lister toutes les constantes (nombres, chaînes de caractères, *etc.*) qui sont référencées dans le code en assembleur, à nouveau une par ligne dans l'ordre dans lequel on s'attend à pouvoir les référencer par leur numéro.

C'est donc la table des constantes. Dans le bytecode, une chaîne de caractères constante est incluse dans cette table par une référence à la table des chaînes internées.

10.3.4 `.names`

La directive `.names` liste les noms de variables ou de fonctions utilisés dans le code assembleur. Ces noms sont des chaînes internées, et donc inclus dans cette liste une fois encodée en binaire par une référence à la table des chaînes internées.

10.3.5 `.text`

La directive `.text` introduit, principalement, la liste des instructions en langage assembleur Python qui doivent être traduites en bytecode.

Une instruction en assembleur Python a au plus un paramètre. Par défaut, tous les paramètres des instructions sont réputés devoir être des entiers écrits en décimal.

Cependant, nous autorisons que :

1. *[Requis]* pour les instructions de saut, l'on puisse écrire indifféremment le nom d'une étiquette ou un entier écrit en décimal codant un nombre d'octets à sauter ou une adresse à atteindre² ;
2. *[Optionnel]* pour les instructions référençant une constante ou une chaîne internée, l'on puisse écrire indifféremment l'objet à référencer ou son numéro de référence dans la table adéquate.

10.3.6 `.line`

La directive `.line`, qui peut uniquement apparaître après la directive `.text`, prend en paramètre le numéro de ligne du code source Python (dans le fichier `.py`) auquel les prochaines instructions font référence, jusqu'à la prochaine directive `.line`.

Voyez cela comme proche de l'option `-g` que vous utilisez avec `gcc` : vous incluez des informations utiles pour qu'un humain puisse naviguer dans son code source Python en fonction de l'exécution du bytecode.

Les valeurs successivement prises par le paramètre de la directive `.line` seront encodées dans un tableau d'octets appelé `lnotab` (cf. Sec. C.4.5).

10.4 Étiquettes

Dernier élément de structuration du code assembleur, les étiquettes sont des noms (pour les humains) que l'on donne à des endroits (des adresses) du code. Une fois encodée en binaire (pour la machine virtuelle Python), une adresse dans le code assembleur est un numéro d'octet dans le bytecode, commençant à zéro : ce que l'on appelle la *valeur de l'étiquette*.

Les étiquettes peuvent être utilisées comme paramètre des instructions de sauts. Ces dernières se divisent entre instructions de saut absolu et relatif.

2. Désolé pour les puristes, mais les adresses en bytecode Python sont tellement riquiqui que le décimal suffit.

Les instructions de saut absolu (`JUMP_IF_TRUE_OR_POP`, `JUMP_IF_FALSE_OR_POP`, `JUMP_ABSOLUTE`, `POP_JUMP_IF_TRUE` et `POP_JUMP_IF_FALSE`) prennent en paramètre la valeur de l'étiquette.

Il n'existe qu'une seule instruction de saut relatif, toujours vers l'avant du code (`JUMP_FORWARD`). Le paramètre de `JUMP_FORWARD` est calculé en effectuant la différence entre la valeur de l'étiquette et l'adresse de l'instruction qui suit `JUMP_FORWARD`.

Enfin, les étiquettes peuvent être utilisées pour les instructions relatives aux boucles et aux exceptions. Le paramètre de `CONTINUE_LOOP` est traité comme s'il s'agissait d'un saut absolu. Le paramètre de `FOR_ITER`, `SETUP_LOOP`, `SETUP_EXCEPT`, `SETUP_FINALLY`, `SETUP_WITH` et `CALL_FINALLY` est traité comme s'il s'agissait d'un saut relatif.

10.5 Exemple de code assembleur Python

Soit le code Python de la Fig. 10.1. Nous donnons à la Fig. 10.2 un exemple de code assembleur Python lui correspondant. Les directives apparaissent en vert. Les instructions utilisées dans ce code assembleur sont reprises de l'Annexe B.

Notez que la chaîne constante "`<module>`" est également internée (l. 11). Même si elle n'est pas utilisée par le code assembleur, la VM Python peut en avoir besoin pendant l'exécution.

Sous la directive `.consts`, nous déclarons des constantes de différents types (entier, liste, `None`, *etc.*).

Sous la directive `.names`, nous déclarons les noms des variables utilisées dans le code assembleur.

En suivant les directives `.line`, nous déduisons que le code source (du fichier `tester.py`, cf. `.set filename`) fait 7 lignes de Python (l. 39).

Quelques observations sur ce code assembleur :

1. pour référencer une constante ou une chaîne, on indique son numéro en paramètre – par exemple, le paramètre de valeur 4 pour la première instruction `LOAD_CONST` désigne la liste `((42 21) 0 1.2)` dans la directive `.consts` et la première instruction `STORE_NAME` de paramètre 0 utilisera la chaîne "a" dans la directive `.names` ;
2. tous les entiers ne codent évidemment pas nécessairement une référence dans une table, comme c'est par exemple le cas pour les instructions `UNPACK_SEQUENCE` et `BUILD_TUPLE`, pour lesquelles cet entier code une grandeur ;
3. les instructions `POP_JUMP_IF_TRUE`, `POP_JUMP_IF_FALSE` et `JUMP_FORWARD` (instructions de saut) utilisent la possibilité d'avoir un nom d'étiquette en paramètre ;
4. les instructions `PRINT_ITEM`, `PRINT_NEWLINE` et `RETURN_VALUE` ne prennent pas de paramètre ;

5. il est parfaitement légal que plusieurs étiquettes se suivent (l. 44), elles désigneront bien évidemment la même adresse (elles auront donc la même valeur) et pourront être utilisées indifféremment pour désigner le même endroit du code.

CODE SOURCE 10.1 – Exemple de code source Python.

```
1 a = 2
2 b = 4
3
4 if a < b :
5     print( a )
6 else :
7     print( b )
```

CODE SOURCE 10.2 – Code en assembleur du code Python de la Fig. 10.1.

```

1
2 .set version_pyvm      62211
3 .set flags             0x00000040
4 .set filename          "totor.py"
5 .set name              "<module>"
6 .set stack_size        2
7
8 .interned
9     "a"
10    "b"
11    "<module>"
12
13 .consts
14     2
15     4
16     None
17
18 .names
19     "a"
20     "b"
21
22 .text
23 .line 1
24     LOAD_CONST          0 # 2
25     STORE_NAME          0 # "a"
26 .line 2
27     LOAD_CONST          1 # 4
28     STORE_NAME          1 # "b"
29 .line 4
30     LOAD_NAME           0 # "a"
31     LOAD_NAME           1 # "b"
32     COMPARE_OP          0 # "<"
33     POP_JUMP_IF_FALSE   label_0
34 .line 5
35     LOAD_NAME           0 # "a"
36     PRINT_ITEM
37     PRINT_NEWLINE
38     JUMP_FORWARD        label_1
39 .line 7
40 label_0:
41     LOAD_NAME           1 # "b"
42     PRINT_ITEM
43     PRINT_NEWLINE
44 label_1:
45 coucou :
46     LOAD_CONST          2 # None
47     RETURN_VALUE



```


Chapitre 11

Spécification du format .pyc (bytecode Python)

11.1 Ordre des octets (boutisme)

Sur une machine, physique ou virtuelle, une valeur sur 32 bits (4 octets) peut être écrite en mémoire d'au moins deux manières, suivant que l'on commence par :

1. l'octet de poids le plus faible – ordre petit-boutiste ( *little-endian*, cf. Fig. 11.1);
2. l'octet de poids le plus fort – ordre gros-boutiste ( *big-endian*, cf. Fig. 11.2).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ef								be								ad								de							

FIGURE 11.1 – Représentation de la valeur 0xdeadbeef en *little-endian*.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
de								ad								be								ef							

FIGURE 11.2 – Représentation de la valeur 0xdeadbeef en *big-endian*.

Les machines amd64 ou x86 codent leurs mots en *little-endian* – alors que, par exemple, l'ordre des octets dans un mot sur Internet est en *big-endian*. Pour une liste des fonctions permettant de convertir entre l'une ou l'autre représentation boutiste et l'ordre utilisé par votre machine, voir `endian(3)`.

11.2 En-têtes des fichiers .pyc

Un fichier .pyc démarre par un en-tête, qui a évidemment le bon goût de varier d'une version de Python sur l'autre. Nous décrivons aux Figs. 11.3–11.5 les trois en-têtes possibles suivant les versions de Python.

Le format de fichier .pyc utilise des mots de 32 bits codés en *little-endian*.

Les données des en-têtes ont la signification suivante :

- `version_pyvm` : nombre magique permettant d'identifier la version de Python, cf. Sec. 11.3 ;
- `timestamp` : heure de production du fichier .pyc, cf. `time(2)` ;
- `source size` : taille du fichier .py correspondant ;
- Pour Python ≥ 3.8 : si le bit H est à 1, alors l'en-tête contient une signature (🇬🇧 *hash*) du code source sur 64 bits, et si le bit C est à 1, alors il faut vérifier que la signature du code source correspond à celle contenue dans l'en-tête.

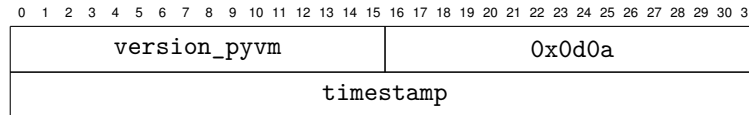


FIGURE 11.3 – En-tête d'un fichier .pyc pour Python 2.7.

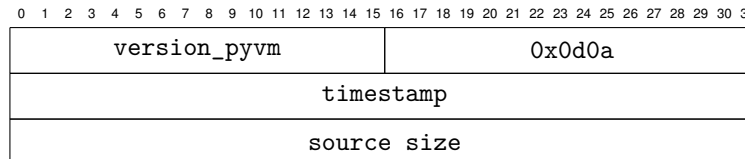


FIGURE 11.4 – En-tête d'un fichier .pyc pour Python ≥ 3.2 (PEP-3147).

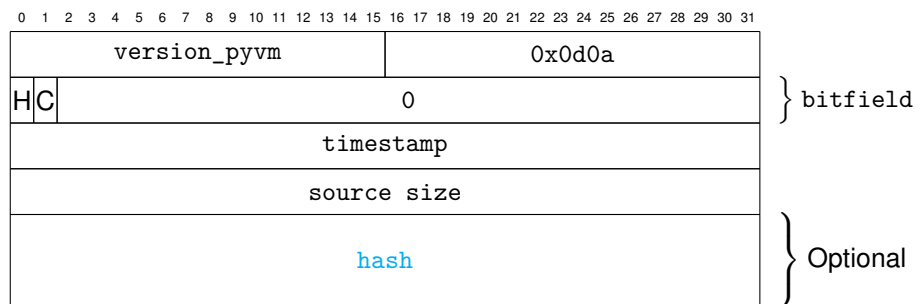


FIGURE 11.5 – En-tête d'un fichier .pyc pour Python ≥ 3.8 (PEP-0552).

La suite est tout simplement l'encodage d'un objet de code Python, cf. Annexe C.

11.3 Valeurs de version_pyvm

Pour obtenir la valeur de `version_pyvm` de votre installation Python, vous pouvez compiler un bête fichier Python et examiner la sortie.

Nous donnons ci-dessous la procédure pour Python 2.7.

```
$ python -m compileall totor.py
Compiling totor.py ...
$ hexdump -x -n 8 totor.pyc
00000000      f303 0a0d 6d4c 60bf
00000008
```

La valeur de `version_pyvm` est donc 0xf303, soit 62211.

En Python 3.x, le fichier de bytecode `.pyc` produit par la commande `python3 -m compileall fichier.py` est automatiquement placé dans un répertoire appelé `__pycache__`. C'est donc là qu'il faudra aller trouver le fichier généré. Nous donnons ci-dessous un exemple avec Python 3.8.

```
$ python3 -m compileall totor.py
$ hexdump -x -n 16 __pycache__/totor.cpython-38.pyc
00000000      0d55 0a0d 0000 0000 6d4c 60bf 000c 0000
00000010
$ wc -c totor.py
12 totor.py
```

La valeur de `version_pyvm` est donc 0x0d55, soit 3413. La valeur `source size` est 0x0c (soit 12), qui est bien la taille du fichier source Python correspondant. Observez enfin que le fichier `totor.pyc` ne contient pas de signature (le champ `bitfield` est à zéro).

Chapitre 12

Travail à réaliser

L'objectif général du projet est de produire un assembleur Python 2.7 lisant un fichier en langage assembleur Python `.pys` et écrivant le fichier binaire `.pyc` associé. Le fichier `.pyc` produit devra être correctement exécuté par la machine virtuelle Python 2.7.

Vous formerez des quadrinômes – et il y aura éventuellement quelques trinômes (mais pas de binôme). L'un(e) d'entre vous aura la responsabilité d'un(e) chef(fe) de projet. Tous les membres du groupe auront le rôle d'architecte / développeur{r,se} / testeur{r,se} / outilleur{r,se} (voir [12.2.1](#)).

Contrairement à ce qu'il se passe parfois en réalité, nous recommandons de choisir pour le rôle de chef de projet celui ou celle d'entre vous qui se sent le plus à l'aise techniquement. Le ou la chef(fe) de projet assurera notamment les fonctions de coordination / planification du projet et de communication avec l'équipe des enseignants. Il/elle aura en outre la responsabilité d'intégrer le code produit par le groupe sous la forme *exacte* décrite ci-dessous. Bien sûr, le chef de projet aura aussi sa part de développement à fournir.

Comme dans la vraie vie du dehors, vous ne serez jugés que sur vos *résultats*. Passer du temps à produire quelque-chose qui ne fonctionne pas ou n'est pas testé correctement ne constitue *pas* un résultat et ne pourra donc pas être valorisé.

12.1 Description des livrables

Le projet est organisé en quatre livrables. Les livrables 1 et 2 sont consacrés à l'analyse lexicale : le premier livrable vise à mettre en place un moteur d'expressions régulières, moteur qui sera utilisé dans le second livrable pour reconnaître et extraire les lexèmes à partir d'un fichier assembleur Python `.pys`. Le livrable 3 est dédié à l'analyse syntaxique du fichier assembleur `.pys` et le livrable 4 à la production du code binaire Python `.pyc`.

La suite de cette section expose les attendus de chaque livrable. Pour aider à démarrer le projet, le premier livrable est plus détaillé que les trois autres.

12.1.1 Livrable 1 : moteur d'expressions régulières

En partant du code *bootstrap* fourni, vous devrez produire un moteur d'expressions régulières tel qu'exposé au chapitre 3.

Plus précisément, sont attendus :

1. un programme exécutable `regex-read.exe` qui réalise le *parsing* d'une expression régulière.

Ce programme prend un unique argument en paramètre dans le Terminal : une chaîne de caractères représentant l'expression régulière.

En cas d'erreur, c'est-à-dire si la chaîne de caractères n'est pas une expression régulière valide, votre exécutable affichera autant que possible un message indiquant la nature de l'erreur dans le Terminal, puis quittera en appelant `exit(EXIT_FAILURE)`.

Sinon, il construira en mémoire la structure de données représentant l'expression régulière (*i.e.* une liste chaînée de groupes de caractères), affichera dans le Terminal la structure de l'expression régulière (c'est-à-dire le contenu de la liste chaînée), puis quittera en appelant `exit(EXIT_SUCCESS)`. Pour le formatage de la sortie, des exemples d'exécution de ce programme sont fournis au paragraphe 3.6.5.

2. un programme exécutable `regex-match.exe` qui vérifie si une expression régulière et le début d'une chaîne de caractères correspondent.

Ce programme prend deux arguments en paramètres dans le Terminal : la chaîne de caractères qui représente l'expression régulière à reconnaître, et la chaîne de caractères dans laquelle on cherche à reconnaître cette expression régulière.

En cas d'erreur, c'est-à-dire si la chaîne de caractères n'est pas une expression régulière valide, votre programme affichera autant que possible un message indiquant la nature de l'erreur dans le Terminal, puis quittera en appelant `exit(EXIT_FAILURE)`.

Sinon, il indiquera dans le Terminal si l'expression régulière et le début de la chaîne correspondent ou pas. Si elles correspondent, il indiquera également la partie de la fin de la chaîne qui n'est pas reconnue par l'expression régulière (*i.e.* le reste de la chaîne). Enfin, le programme quittera en appelant `exit(EXIT_SUCCESS)`. Pour le formatage de la sortie, des exemples d'exécution de ce programme sont fournis au paragraphe 3.6.5.

Pour parvenir à ces résultats, vous produirez également :

- bien sûr, le code C nécessaire à ces programmes.
- vos tests unitaires et vos tests d'intégration (*cf.* Annexe A). Rappelons ici **qu'un programme qui ne serait pas accompagné de tests appropriés et suffisamment complets, prouvant qu'il fonctionne correctement, ne pourrait pas être valorisé.**

Vos tests seront répartis dans plusieurs programmes principaux (`main()`) de test. L'ensemble des tests devra démontrer que chaque fonction im-

portante de votre programme fonctionne comme attendu : comportement normal si tout va bien ou détection appropriée d'éventuelles erreurs.

Quelques étapes à franchir

Voici, dans un ordre quelconque, quelques-unes des étapes qu'il vous faudra probablement franchir - **à compléter, adapter et organiser temporellement et en termes de répartition dans l'équipe** :

- (re)-lire le sujet, puis le lire à nouveau - en particulier les parties utiles à ce livrable ;
- prendre en main le *bootstrap* fourni pour démarrer : analyser le code source et le *Makefile* fournis et se former aux outils fournis dans ce bootstrap pour soutenir les tests de votre code (Cf. Annexe A) ;
- définir l'architecture du code à produire : découpage des fonctionnalités en plusieurs modules (couples de fichiers *.h* et *.c*), définition des types, prototypes et contrats des fonctions de chaque module...
- produire un découpage en tâches, identifier les points de jonction et répartir les tâches entre les membres du groupe ;
- compléter le fichier *src/queue.c* qui implante le type abstrait "file circulaire générique" et compléter les tests de ce module ;
- implanter les opérateurs d'expression régulière manquant, tels que *?*, *+*, *etc.*, et tester ;
- implanter le type abstrait "groupe de caractères", par exemple dans un couple de fichiers *include/pyas/chargroup.h* / *src/chargroup.c*, et écrire des tests unitaires pour ce module ;
- ajouter au module *regexp* (fichiers *include/pyas/regexp.h* et *src/regexp.c*) une fonction `list_t re_read(char *regexp)` ; qui, à partir d'une expression régulière passée en paramètre sous forme chaîne de caractères, retourne la liste de "groupes de caractères" qui la représente et écrire les tests de cette fonction et de ses sous-fonctions ;
- Dans le répertoire *prog*, écrire un programme principal *prog/regexp-read.c* qui réalise le *parsing* d'expression régulière et le tester ;
- modifier le prototype et le code de la fonction `int re_match(char *regexp, char *source, char **end)` ; pour l'adapter à la nouvelle représentation des expressions régulières sous forme de liste de groupes de caractères et tester ;
- adapter le programme principal *prog/regexp-match.c* qui réalise le *matching* d'expression régulière et le tester ;
- écrire vos tests unitaires et d'intégration et valider vos fonctions et programmes ;
- *etc.*, et tester ; !

Trois pistes pour l'organisation du travail

Il vous appartient de bien réfléchir à la façon d'organiser votre travail de développement dans le groupe - après une phase nécessaire de prise en main. A cet effet, pour ce livrable uniquement, nous esquissons ci-dessous trois organisations possibles très différentes. Le chemin que vous déciderez d'adopter pourra être, probablement, un mélange de ces propositions.

- **Cycle de développement en V.** Un cycle en V se caractérise par une première phase d'analyse, descendante, qui détaille progressivement chaque partie à réaliser jusqu'aux entités les plus fines et aboutit à la réalisation de ces entités et à leurs tests unitaires, puis une phase d'intégration, ascendante, qui assemble progressivement chaque entité élémentaire en entités de plus en plus complexes, en vérifiant (testant) la qualité de chaque assemblage.

Dans le contexte de ce livrable, un cycle en V pourrait conduire à :

1. définir le découpage en module ;
2. pour chaque module, définir les types et les fonctions, puis les sous-fonctions utiles à ces fonctions et ainsi de suite jusqu'aux fonctions les plus basiques dont vous aurez besoin ;
3. en bas du V, écrire d'abord les sous-fonctions les plus élémentaires. Les valider chacune par des tests unitaires ;
4. écrire ensuite les fonctions utilisant ces fonctions élémentaires et les valider par des tests, et ainsi de suite ;
5. écrire les fonctions plus importantes (c'est-à-dire `re_read()` et `re_match()` dans notre cas) et les valider par des tests d'intégration ;
6. et enfin écrire les programmes principaux `main()` attendus et les tester.

Une difficulté majeure du cycle en V est qu'il nécessite un travail précis et délicat dans la phase descendante. Si on anticipe mal les choses, on risque de définir un découpage en modules puis en fonctions qui n'est pas approprié. Il faudra alors tout refaire ! Une autre difficulté est qu'il ne permet pas de tester des fonctionnalités un peu conséquentes rapidement : il faut s'approcher du haut du "V" dans la phase ascendante pour commencer à avoir des bouts de programmes un tant soit peu utiles...

Un intérêt du cycle en V est qu'il permet assez vite de se répartir le travail. Par exemple, dès que les modules sont définis, on peut travailler en sous groupe sur chacun des modules. Un autre intérêt est qu'il s'accompagne assez facilement d'une méthodologie de tests rigoureuse. Un dernier intérêt que nous signalerons est qu'il n'y a (théoriquement, c'est à dire si l'analyse dans la phase descendante a été parfaitement exécutée !) pas besoin de modifier un code (*e.g.* une fonction) une fois qu'il a été produit et validé.

- **Cycle de développement incrémental ou "en spirale".** Un cycle en spirale se caractérise par l'implantation de versions successives du pro-

duit final (une par cycle), qui sont à chaque cycle de plus en plus complètes. Pour chaque cycle, on détermine l'objectif du cycle en identifiant la sous-partie des fonctionnalités qu'on va implanter, on modifie le logiciel pour couvrir les fonctionnalités visées, on valide par des tests. A la fin de chaque cycle, on obtient ainsi un produit "viable", qui est utilisable quoique sur une partie limitée des fonctionnalités finales attendues.

Dans le contexte de ce livrable, adopter un cycle en spirale pourrait conduire aux cycles suivants (notez toutefois que d'autres découpages en cycles sont possibles) :

- Cycle 1 : mettre en place une toute première version basique du module implantant le type abstrait "groupe de caractères", en ne gérant qu'un seul caractère et les opérateurs d'expressions régulières `*` et `.` ; tester et valider.
- Cycle 2 : écrire une première version de la fonction `re_read()` qui lit une expression régulière n'utilisant que les opérateurs `*` et `.` et construit une liste de "groupes de caractères" ; tester et valider.
- Cycle 3 : dans la fonction `re_read()`, ajouter la gestion des échappements des opérateurs d'expressions régulières (e.g. le fait que, l'expression régulière `"a*"` signifie "le caractère `'a'`, une fois, suivi du caractère `'*'`, une fois" ; et non pas "le caractère `'a'`, un nombre quelconque de fois").
- Cycle 4 : adapter la fonction `re_match()` pour qu'elle prenne en paramètre une liste chaînée de "groupes de caractères" pour décrire l'expression régulière en lieu et place d'une chaîne de caractères ; tester et valider.
- Cycle 5 : modifier le module "groupe de caractères" pour gérer également les groupes de caractères (opérateurs crochet `[` et `]`), et non pas seulement un caractère unique ; tester et valider.
- Cycle 6 : Modifier la fonction `re_read()` en conséquence ; tester et valider.
- Cycle 7 : Modifier la fonction `re_match()` en conséquence ; tester et valider.
- Cycle 8 : ajouter la prise en compte des opérateurs `+`, `?` et `^` par les fonctions `re_read()`, `re_match()` et dans le type abstrait "groupe de caractères" ; tester et valider.

Une difficulté majeure du cycle en spirale est que la mise en oeuvre d'un cycle donné conduit nécessairement à considérablement modifier le code produit au cycle précédent. Il en faut, alors, du courage ! Une autre difficulté réside dans le choix des objectifs de chaque cycle. Ils ne doivent être ni trop complexes, ni trop courts. Dans le contexte de ce livrable, viser une durée d'une heure à quelques heures pour un cycle serait appropriée.

Un intérêt du cycle en spirale est qu'il évite d'avoir à se représenter la solution complète dès le début : les difficultés sont traitées les unes après les autres. Il n'est d'ailleurs pas nécessaire de définir dès le début le contenu de chaque cycle. Par exemple, on peut, à l'issue d'un cycle

qui n'a pas pleinement convergé, décider de dédier le cycle suivant à des corrections ; ou définir l'objectif du cycle suivant en s'appuyant sur des connaissances acquises au cycle précédent. Un autre intérêt réside dans la convergence itérative vers la solution finale : à la fin de chaque cycle, on dispose d'une solution exécutable et démonstrative, quoiqu'incomplète. Enfin, le cycle en spirale se prête bien à la gestion des risques. On peut par exemple dédier un cycle pour travailler d'abord une chose délicate et ainsi lever des inconnues.

- **Développement guidé par les tests.** Cette méthode de développement est en quelque sorte orthogonale aux deux précédentes : elle peut être employée aussi bien à une étape quelconque d'un processus en "V" que dans un cycle d'un processus en spirale. Elle consiste à écrire les tests *avant* d'écrire le code et même d'y réfléchir - et ce en général par une ou des personne(s) différente(s) de celle(s) qui sont chargées d'écrire le code.

Le développement guidé par les tests présente trois intérêts majeurs - et nous ne serions trop vous encourager à vous y essayer.

Tout d'abord, prosaïquement, il garantit ... que des tests soient écrits et donc que le code produit soit effectivement testé ! Par ailleurs, il permet de bien identifier et formaliser les spécifications et la diversité des cas possibles au sein de l'équipe avant même qu'on ait commencé à réfléchir au code. Cette connaissance partagée des situations que le programme devra gérer, réifiée dans le jeu de tests, s'avère fort utile au développeur tout au long de son travail. Enfin, le développement guidé par les tests s'avère satisfaisant en ce que, durant l'écriture du code, on voit les tests être satisfaits progressivement, les uns après les autres.

il permet de capturer et de formaliser les spécifications avant même l'écriture du code, offrant ainsi une base solide pour le développement. Il offre également une couverture de tests exhaustive qui permet de détecter rapidement tout effet de bord potentiel sur le reste du code existant. Cette approche est particulièrement utile lorsque les exigences évoluent rapidement ou lorsque le risque d'impact sur d'autres parties du système est élevé.

12.1.2 Livrable 2 : analyse lexicale

Vous devrez écrire la fonction `list_t lex(char *regexp_file, char *source_file);` qui, à partir du fichier `regexp_file` contenant les définitions des expressions régulières pour les lexèmes, et d'un fichier `source_file .pys` contenant du code assembleur python 2.7, retourne la liste des lexèmes du fichier de code assembleur `source_file`, ou la liste vide si aucun lexème ne peut être formé avant la fin du fichier de code source – ce qui indiquera une erreur de syntaxe que vous devrez afficher et localiser dans le code source.

Vous fournirez également un exécutable de test appelé `lexer` prenant les deux noms de fichiers en paramètre sur la ligne de commande et affichant la liste des lexèmes (ou l'éventuelle erreur de syntaxe).

En cas d'erreur, votre exécutable appellera `exit(EXIT_FAILURE)` pour quitter. Sinon, il appellera `exit(EXIT_SUCCESS)`.

Enfin, votre rendu inclura le fichier des définitions des types de lexèmes adaptés au langage assembleur Python.

Autres attendus pour ce livrable

Vous produirez également bien sûr :

- vos tests unitaires et d'intégration (cf. Annexe A), qui vérifie aussi bien qu'un fichier `.pys` bien formé est correctement analysé et que la présence d'une erreur lexicale dans un fichier `.pys` est correctement détectée et rapportée.

12.1.3 Livrable 3 : analyse syntaxique

Vous devrez écrire la fonction `pyobj_t parse(list_t *lexems);` qui retournera un objet Python de type code, ou `NULL` en cas d'erreur de syntaxe, qui sera affichée et localisée. Cette fonction implantera l'analyse syntaxique du langage assembleur Python à partir de la liste des lexèmes du code source obtenue au premier livrable.

L'objet de code Python retourné ne contiendra pas encore le bytecode ni `lnotab`. À la place, vous stockerez la liste des lexèmes correspondant à la section `.text` du code assembleur Python.

Vous fournirez également un exécutable de test appelé `parser` qui prendra en paramètre sur la ligne de commande un fichier de code source assembleur Python et affichera l'objet de code Python ou la première erreur lexicale ou de syntaxe (localisée) éventuellement détectée dans le code assembleur Python.

En cas d'erreur, votre exécutable appellera `exit(EXIT_FAILURE)` pour quitter. Sinon, il appellera `exit(EXIT_SUCCESS)`.

Autres attendus pour ce livrable

Vous produirez également bien sûr :

- vos tests unitaires et d'intégration (cf. Annexe A), qui vérifie aussi bien qu'un fichier `.pys` bien formé est correctement analysé et que la présence d'une erreur lexicale ou syntaxique dans un fichier `.pys` est correctement détectée et rapportée.

12.1.4 Livrable 4 : génération de bytecode

Vous devrez écrire la fonction `int pyasm(pyobj_t code);` qui produira le bytecode (et la stockera dans un objet de code Python) et `lnotab` à partir des informations extraites au livrable 2. Elle retournera `-1` en cas d'erreur et `0` sinon.

Vous devrez également écrire la fonction `int pyobj_write(FILE *fp, pyobj_t obj);` permettant de sérialiser un objet Python (y compris, donc,

de type code – qui devra donc pouvoir être exécuté par la machine virtuelle Python).

Vous fournirez votre rendu final sous forme d'un exécutable appelé `pyas` prenant en paramètres sur la ligne de commande le nom du fichier assembleur Python et le nom du fichier `.pyc` voulu. Cet exécutable sera notamment chargé d'écrire l'en-tête d'un fichier `.pyc`. Son code de retour sera `EXIT_SUCCESS` (resp. `EXIT_FAILURE`) en cas de succès (resp. d'erreur, affichée et localisée).

Autres attendus pour ce livrable

Vous produirez également bien sûr :

- vos tests unitaires et d'intégration (cf. Annexe A), qui vérifient aussi bien qu'un fichier `.pys` bien formé produit un fichier `.pyc` valide, et que la présence d'une erreur lexicale ou syntaxique dans un fichier est `.pys` est correctement détectée et rapportée.

12.1.5 Options

Option livrable 1 : prise en compte du contexte dans le *parsing* d'expressions régulières

Pour le premier livrable, nous rappelons que la grammaire des expressions régulières donnée à la Figure 4.1 sous-entend que les symboles des opérateurs d'expression régulière doivent être systématiquement échappés si, dans l'expression régulière considérée, ils désignent le caractère lui-même et non pas l'opérateur. Cette disposition a été retenue car elle s'avère plus facile à implanter, mais elle rend les expressions régulières moins faciles à lire. À la Sec. 3.6.6, nous expliquons comment le langage d'expressions régulières peut être rendu plus sympathique en tenant compte du contexte lors du *parsing* de l'expression. Cette amélioration est proposée en option pour le premier livrable.

Option : prise en charge des nombres complexes

La prise en compte des nombres complexes ne figure pas dans les attendus du projet.

Optionnellement, vous pourrez supporter les complexes. Pour ce faire, nous vous laissons analyser, dans une démarche de rétro-ingénierie, comment les nombres complexes sont pris en compte dans langage assembleur `.pys`.

Option : prise en charge des fonctions Python

La prise en compte des fonctions Python n'est pas un attendu du projet. Dans un premier temps, donc, vos programmes pourront se limiter à traiter un fichier `.pys` correspondant à un script Python ne contenant pas de fonction.

Optionnellement, vous pouvez étendre votre code afin d'assembler des fichiers assembleur `.pys` issus de scripts Python dotés de fonctions Python.

Le format d'un fichier `.pyc` dispose qu'une fonction est stockée dans une constante sous forme d'un objet de type code sérialisé. Ainsi, afin d'ajouter le support pour les fonctions Python à notre assembleur, il suffit d'étendre trivialement la grammaire de la Fig. 10.2 en ajoutant un nouveau type de constante, comme décrit à la Fig. 12.1.

```

<constant> ::= {'integer'} | {'float'} | {'string'} | {'pycst'} | <list> | <tuple> |
               <function>

<function> ::= {'dir::code_start'} {'integer'} <eol> <pys> {'dir::code_end'}

```

FIGURE 12.1 – Extension de la grammaire en EBNF du langage assembleur Python pour le support des fonctions.

L'entier qui suit la directive `.code_start` est la valeur de `firstlineno` correspondant à la première ligne de code de la fonction en question.

Au-delà de cette introduction, si vous implantez la prise en charge des fonctions Python, il vous appartiendra de préciser la façon dont elles sont gérées dans les fichiers `.pys` et `.pyc` au moyen d'une démarche de rétro-ingénierie.

12.2 Modalités de rendu et gestion de projet

Après avoir présenté *supra* ce qui relève de l'informatique pour ce projet, il nous reste à décrire ce que nous attendons concernant les aspects de la gestion et du suivi de projet, et pour les rendus.

12.2.1 Rôles dans le groupe

Dans une équipe de développement logiciel un peu conséquente, chaque membre peut se voir attribuer un ou plusieurs rôles spécifiques, en fonction de ses compétences par exemple. Ce projet étant réalisé en équipe restreinte, il ne peut être question d'un tel degré de spécialisation : chacun(e) adoptera plusieurs rôles, au fur et à mesure du déroulement du projet.

Il est toutefois intéressant que vous soyez à tout instant en mesure d'identifier le rôle que vous remplissez. Voici donc une description succincte des principaux rôles qu'il vous faudra probablement embrasser au cours du projet :

- Chef(fe) de projet. Durant ce projet, le rôle de chef(fe) de projet est le seul qui est (normalement...) attribué une fois pour toute à l'un des membres du groupe. Le/la chef(fe) de projet coordonne et planifie les activités, veille au respect du planning et déclenche un processus correctif s'il ne peut être tenu, veille à l'intégration des productions de chacun(e), s'assure que les rendus soient produits dans les temps, *etc.* Pour ce projet, le/la chef(fe) sera aussi la personne chargée de la communication avec l'équipe des enseignants.

Tous les autres rôles seront à adopter par tous les membres du groupe :

- Architecte. L'architecte est responsable de la définition de la structure du code, avant qu'il soit écrit : définition des conventions de codage (et notamment de nommage), découpage en modules (couples de fichiers `.h` et `.c`) et définition des fonctionnalités et limites de chaque module, définition des types, définitions des noms, prototypes et contrats des principales fonctions, *etc.* Pour ce projet, nous conseillons que le rôle d'architecte soit rempli collectivement (conception collective de l'architecture, avant de lancer le développement).
- Développeur/se. Il/elle produit le code, le modifie, le debug, *etc.* Pour ce projet, bien sûr, chaque membre devra adopter ce rôle, sans quoi il y a peu de chance que le projet aboutisse !
- Testeur/se. Il/elle écrit des tests, en veillant à la qualité de leur couverture (voir l'Annexe A). Pour ce projet, nous demandons que chaque membre s'essaye à ce rôle à un moment ou à un autre.
- Outilleur/se. Il/elle connaît sur le bout des doigts le fonctionnement des outils utiles à l'équipe, sait les configurer et corriger les problèmes survenant durant leur usage. Pour ce projet, les principaux outils sont : un éditeur de texte (au choix), le compilateur `gcc`, un `Makefile`, `valgrind`, `gdb`, `git` et le site Internet `gitlab`, le système de test `unittest/unittest.h` fourni, le langage `bash` du Terminal, et probablement une machine virtuelle permettant de travailler chez soi.

12.2.2 Gestion de projet

Dans la vraie vie du dehors, vous auriez à utiliser un outil dédié à la gestion de projet. Vous êtes évidemment libres de le faire de votre propre chef¹. Si vous choisissez cette solution, vous êtes tenus d'y donner accès à votre tuteur. Sinon, *a minima*, afin de ne pas vous noyer avec un outil supplémentaire, nous vous proposons plus simplement de réaliser un découpage en tâches et sous-tâches.

Une tâche portera un nom formaté de la manière suivante : `T.n.t.s`, où `n` correspond au numéro du livrable, `t` au numéro de la tâche et `s` à l'éventuel numéro de la sous-tâche. Une tâche ou une sous-tâche sera décrite par un titre court, une brève description du travail à effectuer et le nom du responsable de cette (sous-)tâche.

Par exemple :

- `T.1 "Parsing des expressions régulières" (resp. Brice Glace) : Découpage d'une expression régulière en la liste de ses composants.`
- `T.1.1 "Prise en main de unittest.c" (resp. Agathe Zblouz) : Prise en main de l'outil unittest.c et de sa documentation dans tests/howto. Analyse des tests fournis dans le bootstrap dans tests/test-regexp.c et tests/test-list.c.`

1. Parmi les plateformes proposant des versions gratuites, citons [Chanty](#) ou [nTask](#).

- T.1.1.2 "Ecriture des tests pour le *parsing* des expressions régulières" (*resp.* Mehdi Cament) : Ecriture de tests pour la nouvelle fonction `int re_read(...)` et ses sous-fonctions. Validation en commun des tests écrits.
- T.1.1.3 "Parsing des expressions régulières : prototypes des fonctions" (*resp.* Jean-Luc Touré) : Tâche réalisée en commun. Ecriture des fichiers d'en-tête `chargroup.h` et `regexp.h` pour déterminer ensemble l'interface de ces modules (c'est-à-dire les prototypes et contrats des fonctions de ces modules).
- T.1.1.4 "Module groupes de caractères" (*resp.* Claire Obscure) : Ecriture du fichier `chargroup.c` pour les expressions régulières et tests unitaires de ce module.
- T.1.1.5 "Première mise en place du *parsing* d'expression régulières" (*resp.* Camille Honnête) : *Parsing* d'expressions régulières n'utilisant que les opérateurs '*' et '.' et sans groupe de caractères dans `regexp.c` et tests unitaires.
- T.1.1.6 "Opérateur +" (*resp.* Sam Gratte) : Mise en place de l'opérateur '+' dans le *parsing* d'expressions régulières dans le fichier `regexp.c` et tests unitaires de ce module.
- ...
- T.1.X "Intégration du parsing des regexps" (*resp.* Marie-Louise van Cutsem) : Intégration du parsing des expressions régulières dans un programme exécutable, et tests d'intégration de cet exécutable.

Il est bien-sûr fortement conseillé de réfléchir en commun et avec précision au découpage en tâches au début de chaque incrément, puis à chaque fois qu'une adaptation s'avérera utile.

12.2.3 Rendu du code

Vous rendrez votre code en le poussant sur votre dépôt git. Un code qui ne compile pas ou qui plante équivaut à un code non rendu.

Corollaire : mieux vaut rendre un code peut-être pas complètement fonctionnel mais qui puisse être exécuté. Ayez donc toujours quelque-chose qui s'exécute à montrer.

Si nous n'avons pas d'exigences concernant l'outil de gestion de projet (que vous pouvez donc réaliser à la main moyennant les conventions décrites ci-dessus), nous vérifierons par contre que vous utiliserez *git correctement*.

En particulier, nous vérifierons que votre utilisation de *git* :

1. Sera équilibrée entre les membres du trinôme/quadrinôme.

En conséquence, **veillez à ce que chaque *commit* soit bien associé au membre du groupe qui réalise ce commit**. A cet effet, vous devrez soit configurer correctement chacun de vos comptes utilisateur Linux (avec `git config`, c.f. <https://git-scm.com/book/fr/v2/Personnalisation-de-Git-Configuration-de>

soit utiliser l'option `--author` de la commande `git commit` (c.f. <https://git-scm.com/docs/git-commit/>).

2. N'aura *pas* utilisé l'interface web `gitlab` pour y déposer de nouvelles versions de vos fichiers : vous êtes donc tenus d'utiliser `git` *en ligne de commande* depuis le terminal.
3. N'aura pas envoyé sur le serveur de fichiers inutiles (en particulier, pas de fichiers binaires).
4. Aura fait usage de messages explicites pour vos *commits*, comme expliqué ci-après.

Vous indiquerez dans les messages des *commits* l'identifiant (T.n.t.s) de la (sous-)tâche concernée et son état après la modification visée par le *commit*. Par exemple :

```
$ git commit -m "T.1.1: Fixed bug *bla bla*"
$ git commit -m "T.1.1: Finished"
```

Pour vous, procéder de cette manière a aussi l'avantage de pouvoir vite regarder l'historique d'une tâche dans le journal de `git` :

```
$ git log | grep T.1.1"
```

Enfin, concernant l'usage de `git`, il est conseillé :

- de faire régulièrement des `commit`, et en particulier dès qu'une étape est franchi ;
- de ne faire un `commit` que lorsque le dépôt local est dans un état propre - notamment : code pouvant être compilé et exécuté, même si incomplet.

12.3 Obtenir des conseils

Le site Web du projet précisera les modalités qui vous permettront d'obtenir du support de l'équipe enseignante.

Le support pourra concerner indifféremment des questions de compréhension du sujet et des objectifs, des problèmes de conception logiciel, des problèmes de développement (bugs, problèmes de compilation, ...), des questions de gestion de projet au sein du groupe, *etc.*

En substance (voir détails sur le site Web) :

- avant chaque BE de 2h au début de chacun des incréments, vous serez invités à poser des questions par écrit sur un espace Internet partagé. L'enseignant lira vos questions avant le BE et y répondra en début de séance.
- à tout instant, hors séance ou pendant les séances de codage, le/la chef de projet pourra ouvrir un ticket sur un espace `gitlab` dédié. Les enseignants pourront répondre soit en ligne, soit en direct durant les séances de codage. Le temps passé à répondre aux tickets sera comptabilisé, afin de garantir un certain équilibre entre les groupes.

12.4 Compétences

Dans le cadre du référentiel de compétences de l'école, ce projet est l'occasion de valider deux compétences de niveau 2 :

- 1 Concevoir ou réaliser des solutions techniques - théoriques ou expérimentales, permettant de répondre à un cahier de charges :
- 2 Savoir définir, analyser et interpréter une démarche structurée, proposer des adaptations aux techniques connues ;
- 3 Coopérer dans une équipe ou en mode projet :
 - 2 Participer à une équipe plus grande ou mettre en place des outils de gestion de projet.

Ce qui suit est la traduction en français vulgaire, et déclinée pour ce projet, de la fine exégèse à laquelle vos enseignants sont collectivement parvenus au sujet du "référentiel de compétences". Le maître mot pourrait en être une forme de professionnalisme. Nous allons préciser ce que nous entendons par là *pour l'informatique* – étant entendu qu'entrent ici en ligne de compte des spécificités culturelles propres à chaque discipline. Puis nous détaillerons pour chaque compétence.

12.4.1 Traces de l'acquisition des compétences

Il est demandé que chacun conserve des "traces" rendant compte de la façon dont il/elle aura travaillé et, espérons-le, acquis ces compétences. Ces traces ont pour objet de vous servir ultérieurement à rédiger, présenter et documenter vos "preuves de compétences".

Parmi les traces pertinentes dans le cadre de ce projet, citons :

- les documents de découpage en tâches ;
- tous les documents trace de vos réflexions personnelles et de celles du groupe ;
- la liste de vos `commit` sur le dépôt git, comme trace de ce que vous avez accompli ;
- le *log* de tous les `commit` du groupe, comme trace de la progression du groupe vers le rendu final et de son organisation ;
- les mails et autres documents échangés au sein du groupe ;
- tous les éléments qui rendraient compte du cheminement aboutissant à la résolution d'un problème particulièrement ardu (échanges au sein du groupe et avec les enseignants ; versions successives du code ; évolution des tests ; *etc.*) ;
- et enfin bien sûr le code source et les tests, éventuellement dans plusieurs versions, pour pouvoir analyser ultérieurement la progression.

12.4.2 L'informatique dans la vraie vie du dehors

Une fois n'est pas coutume, nous recourons ici à la technique sociologique de l'idéal-type chère à Max Weber pour dresser une sorte de portrait abstrait

d'un informaticien. De même qu'Hiroshima et Nagasaki ont structuré l'inconscient des physiciens dans l'histoire récente, celui des informaticiens l'a été par la contre-culture nord-américaine des années 1960 et 1970 et donc aussi par le pragmatisme américain. Un informaticien ne pourrait éventuellement consentir à reconnaître de hiérarchie que fondée sur la créativité et la maîtrise technique. Corollaire : un informaticien sera ravi de prendre sur ses heures de sommeil pour aider un *newbie* inconnu à l'autre bout du monde, tant que ce dernier aura posé son problème de manière claire. Par contre, un *noob* qui chercherait à faire faire son travail par un autre s'exposerait assez rapidement à une bordée d'acronymes vengeurs.²

Un informaticien ne répondra jamais à une question comme « *Voici mon code en attachement. Pourquoi ne fonctionne-t-il pas ?* » Si vous voulez gagner le respect de votre interlocuteur, et obtenir peut-être une réponse, la première des choses à faire est de décrire ce que vous voulez faire, et comment vous essayez de le faire (de préférence avec le bout de code minimal qui permet de reproduire le comportement fautif, ou une procédure rapide à suivre pour que votre interlocuteur puisse le reproduire facilement). En informatique comme ailleurs, le professionnalisme commence non pas seulement par être conscient de ses limites, mais aussi par l'explication en termes clairs des difficultés rencontrées devant ces limites.

Cette dernière remarque est à ce point capitale en informatique qu'elle fonde le succès du site [StackOverflow](#) : n'hésitez pas à l'utiliser, vous y trouverez, en plus d'excellentes réponses, une foule d'exemples concernant la mise en œuvre de ces saines résolutions dans vos échanges avec autrui. La première raison de ce fonctionnement de la communauté des informaticiens est que l'expérience montre que rédigés ainsi, assez peu de messages sont finalement envoyés puisque leur auteur trouve souvent sa solution lors de la rédaction... Mais la meilleure raison est qu'au moment hypothétique où l'auteur du message pas encore envoyé trouve sa solution, il ressent une forme de plaisir qui structure son esprit de manière bien plus efficace. Bref, montrez que vous avez réfléchi au problème avant d'enquiquiner les Internets pour une faute d'inattention... vous ne risquez que d'atteindre aux plaisirs supérieurs !

Cette clarté que nous venons d'exalter doit finir par infuser dans votre code. Dans une assez large mesure, du code bien écrit s'auto-documente. Les commentaires, suivant une formule de Linus Torvalds, doivent surtout concerner ce dont vous êtes très fier, et ce dont vous avez particulièrement honte. L'idée est que vous écriviez toujours du code *pour quelqu'un d'autre* : soit les autres membres du projet, soit vous-même ne serait-ce que 3 mois plus tard... Avec l'expérience, vous ferez porter votre attention sur les structures de données et la clarté du découpage en fonctions associé, car cela constitue les principaux objets de la pensée informatique.

2. Les plus connus étant bien évidemment [RTFM](#) et [PEBKAC](#).

Longue est la voie qui mène au chemin

Afin de préciser ce que nous venons d'écrire au sujet du professionnalisme à l'aide d'exemples, nous vous proposons ci-dessous de commenter quatre réalisations fonctionnellement équivalentes d'une même fonction prenant un tableau d'entiers en paramètre (ainsi que sa taille), et retournant la liste de ces entiers. Tous les informaticiens pourraient bien sûr n'être pas d'accord entre eux sur ce qui suit, mais l'important est qu'*ils le discuteraient en ces termes*. S'il ressortit de votre droit le plus strict de préférer Rabelais à Chateaubriand, l'on s'attend à ce que cette préférence se fonde sur une connaissance un peu fine de la langue ! Dans ces quatre exemples, nous explicitons exceptionnellement le code pour la gestion des maillons car cela permet une discussion plus directe.

Le Code 12.1 a au moins l'avantage de fonctionner, c'est déjà ça. En plus il ne plante pas et il est en $O(n)$. Et il est même commenté et indenté. Alors pourquoi le lecteur ricane-t-il ? Nous détaillerons ci-dessous les autres écueils dans lesquels l'auteur est tombé, mais les deux plus impardonnables ici sont : (i) le sentiment diffus que ce code va dépenser deux fois trop de temps et d'énergie, en repassant sur la liste à la fin pour l'inverser, et (ii) que l'auteur, dans sa verve, en a malencontreusement oublié les `free(3)` pour libérer les maillons de la liste temporaire `tmp`. On peut aussi reprocher à ce code de mélanger les lignes concernant la gestion du maillon `new` et celle de `tmp`. Bref, n'en jetez plus. Ce code fonctionne, mais c'est à peu près sa seule qualité : il est inefficace et cause des fuites de mémoire. Scolairement, et d'humeur guillerette et printanière, on peut imaginer monter à 7/20.

CODE SOURCE 12.1 – La principale qualité d'Hégésippe Simon est un pragmatisme qui ne demande qu'à s'étoffer. Ce code est censé créer une liste d'entiers à partir d'un tableau d'entiers.

```

1  typedef struct link_t {
2      int          val;
3      struct link_t *next;
4  } *link_t, *list_t, *queue_t;
5
6  list_t array_to_list_newbie( int *integers, int n ) {
7      list_t tmp = NULL;
8      list_t l = NULL;
9      int i = 0;
10
11     while ( i < n ) { /* Construct list in reverse order */
12         link_t new = calloc( 1, sizeof( struct link_t ) );
13         new->val = integers[i++];
14         new->next = tmp;
15         tmp = new;
16     }
17     /* Eventually reverse list: */
18     while ( tmp ) {
19         link_t new = calloc( 1, sizeof( struct link_t ) );
20         new->val = tmp->val;
21         tmp = tmp->next;
22         new->next = l;
23         l = new;
24     }
25     return l;
26 }

```

Le Code 12.2 a au moins un avantage sur le précédent : il ne repasse pas inutilement sur la structure créée. Ses commentaires se concentrent sur les structures de données en expliquant ce qu'il se passe de manière concise (l'auteur fait l'hypothèse réaliste que son lecteur sait ce qu'est une liste chaînée...) L'auteur utilise également correctement le type `size_t`, qui sert effectivement à décrire une taille (un entier positif).

L'on peut néanmoins distinguer quelques points de perfectionnement : comme on va de toute manière réécrire tous les champs du nouveau maillon, un appel à `malloc(3)` aurait suffi – et l'argument du `sizeof` aurait gagné à être en rapport avec le nom de la variable qui va récupérer l'adresse du bloc alloué. On aurait aussi aimé des accolades pour le `else` (même s'il ne contient qu'une seule instruction – car il est toujours tentant de rajouter un `printf(3)` rapidement pour déboguer... et que l'homogénéité augmente la lisibilité) Enfin et surtout, l'incréméntation de la variable de boucle `i` est faite dans un endroit qui nuit à la clarté : un `for` s'imposait ici à la place du `while`. Le Code 12.2 est écrit par quelqu'un qui chemine encore vers la simplicité. Disons que ce code mériterait 12/20. Mais ce n'est certainement pas encore le code de quelqu'un de compétent en programmation.

CODE SOURCE 12.2 – Félix-François Hébert est un gros malin ! Il a compris qu'en parcourant le tableau `integers` il fallait utiliser une file puis la convertir en liste à la fin – ce qui ne coûte quasiment rien et est très élégant !

```

1  list_t array_to_list_fun_awkward( int *integers, size_t n ) {
2      queue_t q = NULL;
3      list_t l = NULL;
4      size_t i = 0;
5
6      while ( i < n ) { /* new links are enqueued in circular list: */
7          link_t new = calloc( 1, sizeof( struct link_t ) );
8          new->val = integers[i++];
9          new->next = new;
10         if ( q ) {
11             new->next = q->next;
12             q->next = new;
13             q = new;
14         }
15         else q = new;
16     }
17     /* Eventually convert circular queue to singly-linked list: */
18     l = q->next;
19     q->next = NULL;
20     return l;
21 }
```

Le Code 12.3 est nettement meilleur : son autrice minimise le nombre d'instructions par itération (surtout, ce `if` tant honni par les architectures matérielles est éliminé de chaque itération) – elle montre qu'elle joue intelligemment avec les deux structures de données de liste et de tableau, car elle orchestre leur comportement respectif *ensemble*. Il pourrait se dégager un certain platonisme de la règle empirique suivante, mais ce n'est pas son intérêt principal : un code plus court est souvent meilleur. L'intérêt de cette règle est opérationnel : du code plus court est plus facilement compris, et donc plus facilement maintenu ou étendu. Corollaire : multipliez les fonctions courtes dans votre code ! Avec une telle concision, il eût même été possible d'omettre tout à fait le commentaire introductif (ce code peut être considéré comme auto-documenté) – mais l'autrice a manifestement conservé un attrait vivifiant pour le sarcasme. Comme nous allons parler de paranoïa dans un instant, saluons enfin dans ce code le souci presque obsessionnellement compulsif de mise en forme de ce code, qui aide l'œil à se concentrer sur son fonctionnement en lui épargnant une étape cognitivement inutile de reformatage mental. On a envie de travailler avec quelqu'un qui écrit du code comme ça. Scolairement, on est autour de 17/20. L'essentiel est là : la programmation est vue comme un exercice d'écriture mathématique, épurée et performative. Il ne manque plus grand-chose...

CODE SOURCE 12.3 – Voltairine Brewster y voit quand même un peu plus clair !
On peut faire le même boulot en 12 lignes au lieu de 20 simplement en utilisant une liste tout le long mais en parcourant `integers` à l'envers !

```
1  /*
2   Rationale: We could have done a funny and pretentious scan of
3   integers by ascending index value, yet it is simpler to produce
4   the list in reverse scan order of the array.
5  */
6  list_t array_to_list_clear_cut( int *integers, size_t n ) {
7      list_t l = NULL;
8      size_t i = 0;
9
10     for ( i = n ; i > 0 ; i-- ) {
11         link_t new = malloc( sizeof( *new ) );
12         new->val = integers[ i-1 ];
13         new->next = l;
14         l = new;
15     }
16
17     return l;
18 }
```

Le Code 12.4 est du code semi-professionnel. Il lui manque encore deux choses pour faire partie d'un projet appelé à opérer en production : une documentation formatée suivant un outil adéquat (on peut bien sûr penser à [Doxygen](#)), et une procédure de test unitaire qui valide son fonctionnement dans tous les cas de figure.

CODE SOURCE 12.4 – Sigismond Ciboulot est un être qui ne souffre plus depuis qu'il a réalisé la toute-puissance de la Loi de Murphy.

```

1  #include <assert.h>
2
3  /*
4   - Returns the empty list if !integers or n == 0 (extra security).
5   - Irrecoverable errors: negative array size or failed memory
      allocation.
6  */
7  list_t array_to_list_paranoid( int *integers, int n ) {
8      list_t lst = NULL;
9      int i = 0;
10
11     assert( n >= 0 );
12
13     if ( NULL == integers ) {
14         return NULL; /* Empty list on any form of no data */
15     }
16     /* Construct list in reverse scan of array; */
17     for ( i = n ; i > 0 ; i-- ) {
18         link_t new = malloc( sizeof( *new ) );
19         assert( NULL != new );
20         new->val = integers[ i-1 ];
21         new->next = lst;
22         lst = new;
23     }
24
25     return lst;
26 }

```

Mais détaillons plutôt les différences avec le Code 12.3, car il est évident qu'ils partagent la même structure de parcours du tableau par indice décroissant. Ces différences font que son auteur code plus vite car il code davantage en sécurité. Voyons donc comment opère cette saine paranoïa.

Deux aspects doivent retenir notre attention : (i) l'abstraction des erreurs irrécupérables de celles qui le sont, et (ii) la programmation défensive qui permet de traiter ces dernières.

À notre modeste niveau, une erreur irrécupérable en C est traitée avec la macro `assert(3)`, qui arrête immédiatement l'exécution du programme en signalant une erreur si la condition passée en paramètre est fausse. Dans ce cas, sont affichées à l'écran la condition non validée et sa ligne dans le code C pour déboguer plus rapidement. La macro `assert(3)` est désactivée si le symbole `NDEBUG` est passé au compilateur, ce qui suggère que la définition (ou pas) de ce symbole `NDEBUG` permet de distinguer entre du code de production qui s'en dispense pour aller (souvent marginalement) plus vite, et du code de mise au point, qui effectue davantage de vérifications.

Le retour du type `int` en lieu et place de `size_t` témoigne que les choses peuvent ne pas être aussi tranchées qu'on pourrait l'espérer. Ce court paragraphe vise à exposer les termes du débat, et il ne nous paraît pas pertinent

de prendre parti. Bien évidemment, une taille est un entier positif, et l'on aimerait le signifier dans l'interface en utilisant le type `size_t`. Nous privilégierions donc ici la clarté du code. En utilisant par contre le type `int`, la clarté du code en pâtit indéniablement, mais *parce que l'on utilise `assert(3)`* avec l'on peut espérer augmenter en contrepartie sa résilience. En effet, nous détectons ici une erreur que le type `size_t` ne permet pas de détecter : celle où l'utilisateur de la fonction, par erreur donc, en viendrait à calculer et/ou utiliser une taille négative. La limite de cette approche, outre de ternir la clarté, est que les tailles de tableaux effectivement utilisables sont limitées à `INT_MAX` (ce qui est déjà très confortable). L'on aurait donc très bien pu conserver le type `size_t`, mais utiliser `int` nous permet aussi d'illustrer plus facilement la distinction importante entre erreurs récupérables et irrécupérables.

Remarquons maintenant qu'il est possible que l'utilisateur de notre fonction choisisse de passer un pointeur nul pour `integers` afin de signifier l'absence de données, mais pour une raison que nous ignorons, la taille `n` serait strictement positive... La technique de programmation dite défensive consiste à traiter le plus tôt possible tous les cas potentiellement problématiques, et si possible en respectant le *Principe de moindre surprise*, qui stipule que du code doit se comporter *par défaut* de la manière la moins inattendue. C'est le cas de notre `if` de la l. 13 qui fait retourner toujours la liste vide, peu importe *pourquoi* il n'y pas de données à placer dans une liste (soit parce que `integers` est nul, soit parce que c'est `n` qui l'est).

Enfin, notre auteur utilise la convention dite Yoda : il n'écrit pas `if (integers == NULL)` ("*si `integers` est `NULL`*"), mais `if(NULL == integers)` (avec l'accent de Maître Yoda : "*si `NULL` `integers` être*"). La raison est que notre auteur se méfie de lui-même et cherche à se prémunir d'une typo désastreuse comme `if(integers = NULL)`.

Du coup, on fait comment ?

Que retenir de ce dernier code pour ce projet ? Vous n'allez pas vous lancer dans des optimisations rusées pour optimiser la vitesse d'exécution en production (les vraies optimisations sont d'ailleurs bien souvent algorithmiques...) Vous vous contenterez plutôt déjà d'un code qui fonctionne de bout en bout. Aussi, nous vous recommandons de ne compiler qu'en mode de débogage (et en ne définissant pas le symbole `NDEBUG`).

N'hésitez pas à (ab)user de la macro `assert(3)` notamment pour vérifier les préconditions de vos fonctions. De même, placez des tests de programmation défensive pour les erreurs récupérables : il s'agit de s'assurer que toutes les conditions sont remplies avant de poursuivre le traitement dans la fonction. Par défaut, faites tourner votre code dans `Valgrind` : moins vous avez de fautes d'accès à la mémoire, plus vite vous avancerez.

Souvenez-vous toujours que le grand jeu du langage C consiste à rester dans les cases de la mémoire que nous déclarons vouloir utiliser.

Quelques derniers conseils qui valent de l'or :

- Tout d'abord, mettre en place un canal de communication efficace dans

- le groupe. Un espace partagé pour centraliser tous les documents (sur le `git` ?), ainsi qu'un *chat* ou un petit forum seraient, par exemple, utiles.
- Pour le code, adopter dans le groupe des *conventions de nommage*. Exemple :
 - `nom_termine_par_t` pour les types ;
 - `underscore_entre_mots` ou `motsReperesParUneMajuscule` pour les variables et fonctions ;
 - `MAJUSCULE` pour les constantes ;
 - Dans un module donné, commencer le nom de toutes les fonctions par le nom du module ;
 - *etc.*
 - Accorder du soin au choix des noms de fichiers, types, fonctions, variables et constantes. En particulier, choisir des noms signifiants, qui aideront à (re)lire votre code.
 - Ne pas utiliser de valeur constante en dur dans le code ; préférer des constantes nommées, définies avec `#define`.
 - Viser la concision et la simplicité. Vous allez construire sur ce code et le reprendre, il doit donc être compréhensible.
 - User et abuser de `valgrind`. Un programme ne doit produire *aucune* erreur d'accès mémoire et, autant que possible, aucune fuite mémoire.
 - Vous n'avez pas terminé lorsque votre code fonctionne. Vous avez terminé lorsque votre code qui fonctionne est simplifié, nettoyé des scories et des verrues qui vous ont permis de parvenir au succès, testé et commenté.
 - Et enfin : faites des tests et conservez vos tests ! Voir l'Annexe A pour des détails.

12.4.3 Compétence 1.2 : "concevoir ou réaliser des solutions techniques - théoriques ou expérimentales [...]"

Maintenant que certains traits culturels en informatique sont posés, et que nous avons vu comment ils trouvent à s'appliquer même et surtout à du code, voyons comment ils se déclinent suivant la première compétence. Ce sera rapide.

L'expression de la pensée informatique est prise dans une tension entre un formalisme forcené et une exposition plus fluide et plus analogique des idées et des outils. Nous avons choisi cette seconde option pour ce projet : les premières parties sont rédigées de manière à ce que vous ayez à fournir un travail minimal d'appropriation et d'adaptation de nouveaux outils intellectuels.

Il eût bien sûr été loisible pour nous de recouvrir tout cela d'une couche bien hermétique de formalisme abscons. Nous y aurions sans doute même pris un peu de plaisir. Mais selon nous, l'intérêt, pourtant *essentiel*, de la formalisation ne peut être pleinement apprécié qu'après une première expérience avec de nouveaux outils.

C'est la raison pour laquelle nous avons filé tout au long de ce sujet l'exemple des expressions arithmétiques pour illustrer les analyses lexicale et syntaxique.

Ainsi, valider cette compétence dans ce projet signifiera que vous aurez découvert de nouveaux outils, même de manière relativement informelle, et que vous aurez été capables de les adapter avec succès au problème que vous aviez à résoudre.

12.4.4 Compétence 3.2 : "coopérer dans une équipe ou en mode projet"

Les traits culturels que nous avons esquissés ne dessinent pas seulement un *habitus* propice au succès technique d'un projet, ils guident aussi les modes de résolution des conflits inter-personnels qui pourraient survenir dans la conduite d'un projet.

En informatique, celui qui a tort, c'est celui dont le code plante, ne fonctionne pas, produit des fautes ou des fuites mémoire, prend beaucoup trop de temps à s'exécuter, ou est mal écrit. C'est une école d'humilité.

Il est donc plus sage de considérer que tout le monde, et soi-même au premier chef, a un peu tort dès le début, et de se dire que par défaut, la bonne solution, tant technique qu'humaine, passe par davantage de travail personnel et de discussion. S'énervier est une perte de temps. Travailler à s'améliorer, rarement. Et ça fluidifie les relations sociales.

Voilà pour ce qui nous semble relever de la coopération. C'est finalement se montrer efficacement volontaire et de bonne foi !

Notre exégèse a bien sûr quelques lacunes. En particulier, le sens hypothétique attaché à la conjonction de coordination *ou* dans la locution « *ou en mode projet* » ne laisse pas de nous surprendre. Nous avons par contre quelques idées bien arrêtées concernant le « *mode projet* ».

Passant outre la pauvreté de la locution, qui laisserait penser qu'il existerait un mode d'existence spécifique à l'être humain lorsqu'il participe à une réalisation collective, il faut donc encore et toujours préciser.

Pour l'informatique, et au niveau de pratique qui est globalement le vôtre, on peut penser que répartir votre temps consacré à ce projet de la manière suivante ne serait pas trop déraisonnable :

1. 50% : travail en solo, implantation et tests de modules ;
2. 20% : travail en duo, jusqu'à ce qu'il y ait convergence entre les tests et le code testé ;
3. 10% : échanges de messages techniques brefs appelant des réponses précises et circonstanciées (« *Je sais que tel bug est dû à telles lignes dans mon code, j'espère pousser un commit dessus avant 18h.* ») ;
4. 20% : discussion en groupe complet sur l'état d'avancement, chacun doit repartir avec une vue claire du travail à réaliser seul et pour quand (*i.e.* l'interface et les spécifications du module qui lui est confié).

Cela reste bien entendu indicatif. De notre point de vue, le problème principal à considérer ici est l'obtention d'un équilibre acceptable pour chacun dans le partage du temps entre le travail et la communication. En continuant, nous

entrerions dans le royaume luxuriant de la diversité des caractères humains, où les expansifs dilettantes côtoient les anachorètes abusivement émotifs.

Alors simplifions. Nous ne voyons finalement que quatre règles à appliquer pour que les temporalités de chacun trouvent à s'agencer efficacement :

1. Répondre rapidement à une sollicitation, idéalement dans les 3h maximum – ou même plus tôt pour signifier la prise en compte du message qui vous a été adressé si vous sentez que son traitement complet nécessitera davantage de temps (dans ce cas, fixez vous-même explicitement une date ferme de réponse complète au plus tôt)³ ;
2. Ne pas refuser de s'entendre dire que l'on sollicite trop tel ou telle autre membre du projet, auquel cas il faut discuter et parvenir à un accord bilatéral sur des modalités communes de communication ;
3. Réserver du temps collectif (mais pas trop...) à l'organisation du travail ;
4. Construire collectivement une compréhension aussi objective que possible du processus de travail : découpage en tâches, état d'avancement...

De votre point de vue, vous avez là ce qui est culturellement accepté comme la norme en matière de participation sociale minimale à un projet. Cela vous permet donc de discuter entre vous à partir de ce qui est acceptable et de ce qui ne l'est pas.

Pour finir de traduire en français vulgaire : lors de la conduite d'un projet, vous êtes dans la même galère qui doit avancer dans la tempête – *et vous avez le droit d'éprouver de la Schadenfreude en jetant les touristes et les gens de mauvaise foi par-dessus bord.*

12.4.5 Validation des compétences

Si nous avons pris la peine de rédiger ces lignes au sujet des compétences, avec espérons-nous toute la minutie et la couleur requises, c'est pour que vous compreniez que cette partie de l'évaluation sanctionnera des qualités qui, sans autre explication, pourraient passer pour éminemment arbitraires.

De notre point de vue, il s'agissait de décrire aussi finement que possible ce qui fait que vous serez perçu comme un élément moteur dans un groupe, ou comme un baleineau échoué sur la grève. Mais nous avons aussi assez décrit, croyons-nous, en quoi tendre vers ces comportements culturels d'informaticiens (recherche de simplicité, accumulation de précautions, clarté et précision dans l'expression générale, honnêteté, émulation plutôt que compétition, etc.) permet de travailler *mieux* et de parvenir plus sûrement à des résultats opérationnels tangibles.

3. Si un problème est bloquant pour un membre du projet, il faut qu'il ou elle obtienne rapidement une estimation du temps disponible pour d'autres tâches, avant de pouvoir reprendre une fois que le problème est résolu. Pendant ce temps qui ne peut pas être mort, le membre bloqué du projet peut rédiger les tests d'un module, avancer sur le codage d'un autre, etc.

Or, comme la validation de la compétence 1.2 dans ce projet passe par l'acquisition *opérationnelle* de connaissances nouvelles pour vous⁴, on imagine mal comment un projet qui ne fonctionne pas permettrait de valider l'une ou l'autre compétence.

Inversement, nous serons évidemment enclins à valider l'une et l'autre compétences en cas de succès incontestable pour votre projet. Mais il appartiendra à vos encadrants de moduler cette impression (concernant un travail collectif) par d'éventuelles anecdotes au sujet d'Untel qui n'aurait manifestement pas compris un traître mot du sujet et s'en contenterait piteusement, ou qui aurait disparu corps et biens dans un silence radio assourdissant et récurrent. Nous vous devons aussi un minimum d'honnêteté sur ce qui vous attend ensuite... À la lecture de ces quelques principes d'une portée finalement assez générale, vous sentez d'ailleurs peut-être déjà que vous aurez probablement l'occasion d'en expérimenter les bienfaits dans d'autres moments de votre vie que seulement professionnels.

En résumé, et sauf cas nécessitant manifestement un traitement particulier, la validation de ces compétences devrait présenter une assez forte corrélation statistique avec votre note obtenue lors de l'évaluation fonctionnelle principale de votre travail.

12.5 Notation

La note finale du module sera constituée d'une partie de contrôle continu, d'une partie de contrôle final (examen - soutenance) et d'une modulation entre membres du groupes.

La note de contrôle continue, appelée CC, sera déterminée d'une part par la moyenne des notes obtenues à des très courts questionnaires auxquels il vous sera demandé de répondre au début de certaines séances de tutorat de 2h, d'autre part par une note de livrables.

Les encadrants vous attribueront la note globale de livrable comme suit :

- fonctionnalités du code rendu (60%) ;
- propreté du code, pertinence des tests, utilisation des outils tels que `git`, `valgrind`, *etc.* (20%) ;
- gestion de projet (20%).

Les enseignants ne vous feront un retour que sur votre premier livrable.

Un code non rendu (*cf.* définition ci-dessus) ne peut donc rapporter que les points de la gestion de projet. Afin de garder intacte votre motivation, vous ne connaîtrez votre note de livrables qu'à la toute fin du projet.

L'examen consistera principalement en une étape de recette sur chacun des livrables (nous ferons passer des tests à votre code en votre présence – d'où l'importance de rendre des exécutables conformes pour chaque livrable),

4. En gros, vous aurez montré que vous êtes capables de transformer du PDF qui parle d'une idée en français, en du code C qui la fait fonctionner dans le monde physique ! Accessoirement, cela vous procure un avantage concurrentiel sur un réseau de neurones.

une visite de portions choisies de votre code et des questions. Cette évaluation donnera une note sur 20 appelée T.

Nous vous demanderons enfin de vous répartir 30 (*resp.* 40) points fictifs de participation pour chaque membre du trinôme (*resp.* quadrinôme). Ainsi : $P_1 + P_2 + P_3 = 30$ (*resp.* $P_1 + P_2 + P_3 + P_4 = 40$).

La note finale N_i de chaque membre du trinôme/quadrinôme sera calculée à l'aide d'une sigmoïde, *cf.* la formule ci-dessous pour les trinômes :

$$N_i = \min \left\{ 20, \frac{CC + T}{2} \times \left(\frac{1}{2} + \frac{1}{1 + e^{-\frac{1}{3}(P_i - 10)}} \right) \right\}.$$

Et pour les quadrinômes :

$$N_i = \min \left\{ 20, \frac{CC + T}{2} \times \left(\frac{1}{2} + \frac{1}{1 + e^{-\frac{1}{4}(P_i - 10)}} \right) \right\}.$$

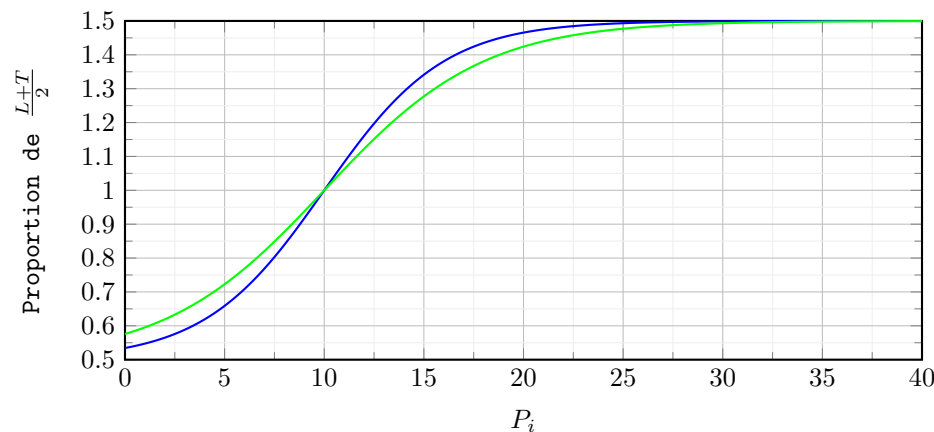


FIGURE 12.2 – Fonction d'équilibre dans le trinôme : $\frac{1}{2} + \frac{1}{1 + e^{-\frac{1}{3}(P_i - 10)}}$ en bleu – et dans le quadrinôme : $\frac{1}{2} + \frac{1}{1 + e^{-\frac{1}{4}(P_i - 10)}}$ en vert.

La note $\frac{CC+T}{2}$ représente ce que vaut votre projet pour lui-même, en tenant compte de sa progression. La Fig. 12.2 reprend la proportion de $\frac{CC+T}{2}$ en fonction de l'équilibre dans le trinôme ou le quadrinôme : si vous vous attribuez 10 points fictifs à chacun, tout le monde sera réputé avoir fourni la même quantité de travail *utile* et vous obtiendrez tous la note $\frac{CC+T}{2}$. Si un des membres du trinôme n'obtenait par exemple que 5 points fictifs, il n'obtiendrait finalement que 65.5% de $\frac{CC+T}{2}$. Notez que cette formule implique qu'il n'est rentable pour personne qu'un membre du groupe cherche à réaliser seul l'intégralité du projet.

Annexe A

Tests

Sauf exception, il s'agit de votre premier projet un peu conséquent en programmation, et surtout en groupe. L'un des objectifs du projet est que vous développiez votre savoir-faire en matière de Génie Logiciel et en particulier en matière de *test* – et le succès de cet apprentissage sera évalué, au même titre que votre projet lui-même.

Espérer que du code non testé va fonctionner relève de la pensée magique. Autrement dit, tester un programme est aussi important que de le concevoir puis de le développer. De même, les jeux de tests (ensemble des tests écrits pour valider un programme ou une partie de son code) ont autant de valeur que le code du programme testé lui-même.

Tester du code de manière exhaustive, c'est-à-dire sur tous les cas possibles, est difficile – et en général par principe impossible. Il existe néanmoins des stratégies et un savoir-faire associé. En génie logiciel, le *test* regroupe l'ensemble des méthodes et outils permettant de valider (ou invalider !) une partie ou la totalité d'un logiciel, c'est-à-dire de vérifier s'il se comporte comme il le devrait.

Nous n'avons pas ici pour but de présenter le vaste monde du test logiciel. Nous partageons plus simplement quelques conseils pratiques pour que vous testiez "un peu (plus) correctement" vos programmes dans le cadre de votre projet 2A. Pour ce faire, nous présentons quelques principes généraux, quelques conseils formant une stratégie basique de test de votre code qui peut déjà se montrer assez redoutablement efficace, puis l'outil de test que nous vous fournissons à cet effet dans le *bootstrap* et qu'il vous faudra prendre en main.

A.1 Notion de jeu de test et d'oracle

Concevoir un jeu de test consiste à produire d'une part une série de données d'entrée, d'autre part les résultats attendus pour ces données d'entrée. On appelle oracle d'un test ces résultats attendus. Un oracle peut typiquement

être une valeur numérique attendue, une chaîne de caractères dont il est attendu qu'elle soit affichée dans le terminal, un code de retour (*exit code*) d'un programme, *etc.*

Tester le programme, c'est ensuite vérifier que le programme (ou la partie concernée du programme) se comporte comme attendu : pour chaque test du jeu de tests, il s'agit de vérifier si les sorties produites par le programme correspondent à l'oracle.

A.2 Tests fonctionnels (ou "d'intégration") et tests unitaires

L'on distingue classiquement deux types de tests, qui ont chacun une utilité particulière dans un projet : les tests fonctionnels (ou "d'intégration") et les tests unitaires.

Les **tests unitaires** sont des tests de bas niveau, qui testent isolément une petite partie d'un programme : une fonction, un petit groupe de fonctions, un type abstrait et le module qui l'implante, *etc.* Chaque test unitaire va alors correspondre à la validation d'une propriété de vos fonctions. On écrira, par exemple, un ou des jeu(x) de tests unitaires pour les fonctions du module `list`, *etc.* Idéalement, chaque partie de votre projet (*e.g.* chacune des fonctions d'un module `.h` / `.c` unique qui implante un type abstrait de donnée, une fonction spécifique, *etc.*) devra être testée isolément au moyen de tests unitaires.

Les **tests fonctionnels** sont des tests de plus haut niveau, sur des exécutables dont le comportement global est censé converger, par grandes étapes fonctionnelles successives, vers l'exécutable applicatif final. Dans ce projet, vous serez amenés à mettre en place plusieurs programmes exécutables : pour le *parsing* d'expressions régulières, le *matching* d'expression régulières, l'analyseur lexical, l'analyseur syntaxique et encore pour l'assembleur – et ce dernier exécutable pourrait bien être le résultat de votre projet ! Pour chacun de ces exécutables, il vous revient de tester s'ils fonctionnent correctement au moyen de jeux de tests fonctionnels. Les tests fonctionnels représentent ainsi des buts de haut niveau à atteindre.

A.3 Notion de test positif et de test négatif

Il convient de tester bien sûr le fonctionnement normal du programme ou de la partie testée, c'est-à-dire le fait qu'il fournit la réponse attendue lorsque les données en entrée sont correctes. On parle alors de "tests positifs". Mais il faut aussi tester la capacité du programme (ou de la partie du code testée) à détecter les erreurs, c'est-à-dire le fait que si une erreur s'est glissée dans les données en entrée, celle-ci est bien détectée et traitée comme il convient. Par exemple : une expression régulière invalide, une erreur syntaxique dans un fichier en langage assembleur, *etc.* On parle alors de "tests négatifs".

A.4 Quelques conseils

- Tester un programme demande du temps. Peut-être autant qu'écrire le programme lui-même ! Ce n'est pas du temps perdu. C'est un temps nécessaire.
- Pour tester son programme, il ne suffit pas de l'exécuter un petit nombre de fois avec quelques jeux de données. Il faut au contraire *bien réfléchir aux jeux de test*. Deux objectifs que vous pouvez vous fixer sont :
 1. d'essayer de tester "tous les cas possibles". C'est le plus souvent impossible en pratique, puisqu'il y a par principe une infinité de données d'entrée possibles ! Mais cet objectif doit néanmoins vous guider dans la conception de vos tests.
 2. d'essayer que "toutes les lignes de code du programme sont exécutées au moins une fois par au moins un des tests" (e.g. tous les cas des conditionnelles, toutes les boucles, etc.) Là encore, ce n'est pas (du tout) évident, mais cet objectif peut vous guider dans la conception de vos tests. Pour information, l'ensemble des lignes de code effectivement exécutées lorsqu'on a lancé tous les tests d'un jeu de tests est appelé la "couverture" du jeu de tests.
- Vos jeux de tests doivent être conservés sur le dépôt `git`, au même titre que le code source. Les tests sont en effet une ressource précieuse ! Il s'agit ici d'abord de conserver la trace de tout ce qu'on a fait pour tester – notez d'ailleurs que ceci sera regardé par votre évaluateur lors de l'examen final. Mais il s'agit aussi de pouvoir plus tard relancer facilement tous les tests déjà écrits, pour s'assurer que le logiciel n'a pas perdu en qualité au fur et à mesure de ses évolutions. C'est ce qu'on appelle "faire des tests de non-régression". Dans ce projet, vous allez devoir faire évoluer votre code, parfois en le reprenant plusieurs semaines avoir l'avoir écrit. Il sera alors extrêmement utile de pouvoir re-tester automatiquement les parties modifiées pour vérifier si de nouvelles erreurs n'auraient pas été malencontreusement introduites par l'implantation d'une nouvelle fonctionnalité ou par la correction d'un problème.
- En accumulant des tests et en les faisant s'exécuter régulièrement (ou en cas de doute), les tests vous permettent de capitaliser sur votre code : vous faites apparaître le plus tôt possible l'existence d'un problème et vous minimiserez le temps passé à rechercher l'origine de ce problème. De plus, la liste des tests s'allongeant, vous observerez un effet psychologique rassurant en contemplant tout ce qui fonctionne dans votre code. Cela permet également de mesurer l'état d'avancement global du codage.
- Il est généralement admis que l'efficacité d'une stratégie de test est meilleure lorsque la personne qui écrit les tests n'est **pas** celle qui écrit le code correspondant. Si l'on considère par exemple deux modules A et B d'un code, on peut imaginer que Bananuphe doive coder le module

A et écrire les jeux de test du module B, alors que Golinduche codera le module B et écrira les jeux de test du module A. Cette stratégie offre plusieurs avantages : chacun(e) aura une vision claire de l'ensemble du projet, et deux personnes auront contribué à chaque module, de sorte que la personne qui code ne sera pas tentée de négliger tel ou tel cas de figure auquel elle n'aurait pas pensé mais que son/sa sympathique camarade aurait repéré.

- Il peut s'avérer également utile d'écrire des tests de façon collaborative. En effet, l'écriture d'un jeu de tests est un moyen très efficace de se mettre d'accord sur les objectifs à atteindre et d'en construire une vision commune.
- Il est enfin souvent très bénéfique d'écrire les jeux de tests en amont du développement. On parle alors en Génie Logiciel de *développement guidé par les tests*. Cela permet de réfléchir à la variété des cas possibles et de clarifier les buts à atteindre, avant même de réfléchir au code. Puis on écrit le programme, de telle sorte que progressivement il satisfasse tous les tests préalablement conçus. Il s'avère au demeurant très réconfortant de constater comment, progressivement, les tests sont satisfaits.

En complément de ces quelques conseils, nous fournissons dans le *bootstrap* deux jeux de tests pour certaines des fonctionnalités du *bootstrap*, que nous avons poussés jusqu'à un niveau que nous souhaitons vous voir adopter. Nous vous conseillons de lire en détail ces fichiers, et de vous en inspirer pour vos propres tests :

- Le premier, dans le fichier `tests/test-list.c`, teste le module implantant le type abstrait "listes chaînées génériques".
- Le second, dans le fichier `tests/test-regexp.c`, teste la fonction `re_match(...)` fournie, qui réalise le *matching* d'une expression régulière simple (uniquement avec les opérateurs '*' et '.') et d'une chaîne de caractères.

A.5 Le système de test fourni

Le *bootstrap* fourni au début du projet est doté d'un module `unittest` (fichiers `unittest.h` / `unittest.c`) qui permet d'écrire et exécuter des tests jusqu'à un niveau de détail que nous considérons adéquat et suffisant pour ce projet.

Cet outil est rapide à mettre en œuvre, et fournit qui plus est de manière native une aide appréciable au débogage.

Écriture des tests

La documentation permettant d'écrire des tests avec cet outil est fournie dans le répertoire `tests/howto`. Elle est à lire attentivement. Commencez par lire / compiler / exécuter les fichiers `01_test-examples-pass.c` et `02_test-examples-fail.c`,

les autres fichiers (`ZZ_test-implementation-*`) illustrent des techniques d'un peu plus haut niveau et nécessitent d'avoir déjà pratiqué pour en tirer pleinement profit.

Voici toutefois une première introduction à l'outil.

Pour créer des tests unitaires en C avec cet outil, il suffit d'écrire un programme exécutable `main()` dans un fichier `.c`, et d'utiliser les fonctions du fichier `unittest.h`.

Le fichier `.c` sera typiquement placé dans le répertoire `tests/` du code fourni, ou dans un sous-répertoire.

Tout d'abord, il faut inclure `<unittest/unittest.h>` et l'initialiser en appelant la fonction `unit_test`. Ensuite, vous créez une fonction par suite de tests unitaires.

Ici, les tests unitaires sont regroupés en *suites* de tests unitaires. Une suite de tests est introduite avec `test_suite(suite_descr)`.

Dans une suite de test, les deux premiers outils à prendre en main pour produire un test unitaire sont :

- `test_assert(cond, test_descr)`, pour tester une propriété en validant une condition (typiquement un oracle numérique) ;
- soit à l'aide de l'une des fonctionnalités `test_oracle_check*(test_descr, oracle)` lorsque l'oracle est une sortie (chaîne de caractères) que le code testé est censé écrire.

Nous vous donnons un premier exemple couvrant ces premières fonctionnalités de `unittest` dans le Code A.1.

Notez que d'autres fonctionnalités plus avancées sont disponibles. Elles sont documentées dans le répertoire `tests/howto`.

Rappelons que vous trouverez également dans l'embryon de code fourni deux programmes de test dans le répertoire `tests` :

- `test-list.c` : ensemble de tests unitaires pour le module qui implante le type abstrait "listes chaînées génériques" fourni ;
- `test-regex.c` : ensemble de tests unitaires pour le *matching* d'expressions régulières basiques, telles que présentées dans ce sujet (opérateurs `*` et `.` uniquement).

Il vous appartient de lire ces deux fichiers `.c` (pour comprendre comment ils font usage des fonctionnalités d'`unittest`) et de compiler puis d'exécuter ces deux programmes de test unitaire (pour vérifier que tous les tests passent). Puis, bien sûr, d'écrire à votre tour d'autres fichiers de tests unitaires au fur et à mesure du projet.

CODE SOURCE A.1 – Comment écrire des tests unitaires en C avec le code fourni.

```
1  #include <unittest/unittest.h>
2
3  void exemple_test_assert( void ) {
4      test_suite( "Examples with test_assert" );
5
6      test_assert( 1 == 1,
7                  "Can assert numerical equality" );
8      test_assert( 1 > 0 && 0 < 1,
9                  "Can assert numerical order" );
10     test_assert( 5 == 3+2,
11                 "Addition works" );
12 }
13
14 void exemple_test_oracle( void ) {
15     test_suite( "Example with text oracles" );
16
17     test_oracle_start( stdout );
18     printf( "%d + %d = %d", 2, 3, 5 );
19     test_oracle_check( "Can check standard output", "2 + 3 = 5" );
20 }
21
22 int main ( int argc, char *argv[] ) {
23
24     unit_test( argc, argv );
25
26     exemple_test_assert();
27     exemple_test_oracle();
28
29     exit( EXIT_SUCCESS );
30 }
```

Aide au débogage

L'exécutable d'un test unitaire ainsi rédigé accepte les options `-v` et `-g` (c'est pour cela qu'il faut passer `argc` et `argv` à `unit_test`). Les deux options peuvent être cumulées avec `-vg`.

L'option `-v` (ou `--verbose`) permet d'afficher l'exécution de chaque test dans chaque suite de tests. Si cette option est absente, seul un message concernant la suite de tests en cours est affiché.

L'option `-g` (ou `--debug`) permet de signifier qu'un test est appelé à être exécuté dans un débogueur.

À la fin de l'exécution d'un programme de tests unitaires, le nombre de tests ayant échoué est affiché s'il est non nul.

Annexe B

Jeux d'instructions de la VM Python

Les instructions données *infra* proviennent de `python-2.7.18` et `python-3.8.8`. Elles devraient donc être très largement compatibles avec une version à peu près à jour de Python 2.7 ou Python 3.x. Nous avons repris des informations de unpyc.sourceforge.net, que nous avons parfois mises à jour (instruction `LIST_APPEND` notamment, mais pas que) et auxquelles nous avons ajouté les opcodes pour Python 3.x.

Une instruction est décrite par sa mnémonique, son éventuel paramètre et ses opcodes sous forme de deux chiffres hexadécimaux préfixés par `0x`, d'abord pour Python 2.7 et ensuite pour Python 3.x. Si une instruction n'existe pas dans la version ciblée de la VM, l'opcode est noté avec des tirets : `----`. Voici le gabarit utilisé pour décrire une instruction :

MNEMO [opt. parm]

Opcodes : 2.7 | 3.x

Le texte qui suit spécifie le rôle de l'instruction.

Avant l'exécution d'une instruction, la pile d'exécution S est réputée être constituée de n objets Python et S_k est l'objet en $k^{\text{ième}}$ position dans S . Le sommet de la pile est S_0 :

$S = S_0, S_1, \dots, S_{n-1}$

Les instructions d'opcode strictement inférieur à `0x5A` ne prennent pas d'argument.

Le décodeur de bytecode intègre une variable `parm_high` contenant les 16 bits de poids forts à ajouter aux 16 bits de poids faible du paramètre encodé dans le bytecode pour obtenir la valeur réellement utilisée pour l'argument de l'instruction. `parm_high` est remis à zéro après l'exécution de chaque instruction, sauf s'il s'agit de l'instruction `EXTENDED_ARG`.

B.1 Encodage des instructions Python

B.1.1 Python 2.7

Les instructions d'opcode supérieur ou égal à 0x5a admettent un paramètre, codé sur deux octets. Une instruction occupera donc un ou trois octets dans le bytecode suivant qu'elle aura 0 ou 1 paramètre. Il s'agit donc d'un jeu d'instructions de longueurs variables.

B.1.2 Python 3.x

Toutes les instructions occupent deux octets dans le bytecode. Il s'agit donc d'un jeu d'instructions de longueur fixe.

Lorsqu'une instruction n'admet pas d'argument, l'octet qui suit son opcode est 0x00.

B.2 Spécification des instructions

STOP_CODE	Opcodes : 0x00 0x00
Indique la fin du code au compilateur, inutilisé par l'interpréteur.	
POP_TOP	Opcodes : 0x01 0x01
Dépile l'objet au sommet de la pile.	
$B \leftarrow B \quad S \leftarrow S_1, S_2, \dots, S_{n-1}$	
ROT_TWO	Opcodes : 0x02 0x02
Échange les deux objets au sommet de la pile.	
$B \leftarrow B \quad S \leftarrow S_1, S_0, S_2, \dots, S_{n-1}$	
ROT_THREE	Opcodes : 0x03 0x03
Déplace les deuxième et troisième objets au sommet de la pile et déplace en troisième position l'objet qui était au sommet de la pile.	
$B \leftarrow B \quad S \leftarrow S_1, S_2, S_0, S_3, \dots, S_{n-1}$	

DUP_TOP	Opcodes : 0x04 0x04
Duplique l'objet au sommet de la pile.	
$B \leftarrow B \quad S \leftarrow S_0, S_0, S_1, \dots, S_n$	

DUP_TOP_TWO	Opcodes : ---- 0x05
Duplique les deux objets au sommet de la pile.	
$B \leftarrow B \quad S \leftarrow S_0, S_1, S_0, S_1, \dots, S_{n-1}$	

ROT_FOUR	Opcodes : 0x05 0x06
Déplace les deuxième, troisième et quatrième objets au sommet de la pile et déplace en quatrième position l'objet qui était au sommet de la pile.	
$B \leftarrow B \quad S \leftarrow S_1, S_2, S_3, S_0, S_4, \dots, S_{n-1}$	

NOP	Opcodes : 0x09 0x09
Ne fait rien.	
$B \leftarrow B \quad S \leftarrow S$	

UNARY_POSITIVE	Opcodes : 0x0a 0x0a
Implémente l'opérateur unaire +.	
$B \leftarrow B \quad S \leftarrow +S_0, S_1, \dots, S_{n-1}$	

UNARY_NEGATIVE	Opcodes : 0x0b 0x0b
Implémente l'opérateur unaire -.	
$B \leftarrow B \quad S \leftarrow -S_0, S_1, \dots, S_{n-1}$	

UNARY_NOT	Opcodes : 0x0c 0x0c
Implémente l'opérateur unaire not.	
$B \leftarrow B \quad S \leftarrow \text{not } S_0, S_1, \dots, S_{n-1}$	

UNARY_CONVERT	Opcodes : 0x0d ----
Implémente l'opérateur unaire '... '.	
$B \leftarrow B \quad S \leftarrow 'S_0', S_1, \dots, S_{n-1}$	

UNARY_INVERT	Opcodes : 0x0f 0x0f
Implémente l'opérateur unaire ~.	
$B \leftarrow B \quad S \leftarrow \sim S_0, S_1, \dots, S_{n-1}$	

BINARY_MATRIX_MULTIPLY	Opcodes : ---- 0x10
Dépile les deux matrices au sommet de la pile et empile le résultat de la multiplication matricielle.	
$B \leftarrow B \quad S \leftarrow S_1 \times S_0, S_2, \dots, S_{n-1}$	

INPLACE_MATRIX_MULTIPLY	Opcodes : ---- 0x11
Dépile les deux matrices au sommet de la pile et empile le résultat de la multiplication matricielle (version "en place").	
$B \leftarrow B \quad S \leftarrow S_1 \times S_0, S_2, \dots, S_{n-1}$	

BINARY_POWER	Opcodes : 0x13 0x13
Implémente l'opérateur binaire **.	
$B \leftarrow B \quad S \leftarrow S_1 ** S_0, S_1, S_2, \dots, S_{n-1}$	

BINARY_MULTIPLY	Opcodes : 0x14 0x14
Implémente l'opérateur binaire *.	
$B \leftarrow B \quad S \leftarrow S_1 * S_0, S_1, S_2, \dots, S_{n-1}$	
BINARY_DIVIDE	Opcodes : 0x15 ----
Implémente l'opérateur binaire /.	
$B \leftarrow B \quad S \leftarrow S_1 / S_0, S_1, S_2, \dots, S_{n-1}$	
BINARY_MODULO	Opcodes : 0x16 0x16
Implémente l'opérateur binaire %.	
$B \leftarrow B \quad S \leftarrow S_1 \% S_0, S_1, S_2, \dots, S_{n-1}$	
BINARY_ADD	Opcodes : 0x17 0x17
Implémente l'opérateur binaire +.	
$B \leftarrow B \quad S \leftarrow S_1 + S_0, S_1, S_2, \dots, S_{n-1}$	
BINARY_SUBTRACT	Opcodes : 0x18 0x18
Implémente l'opérateur binaire -.	
$B \leftarrow B \quad S \leftarrow S_1 - S_0, S_1, S_2, \dots, S_{n-1}$	
BINARY_SUBSCR	Opcodes : 0x19 0x19
Implémente l'opérateur binaire [...].	
$B \leftarrow B \quad S \leftarrow S_1[S_0], S_1, S_2, \dots, S_{n-1}$	
BINARY_FLOOR_DIVIDE	Opcodes : 0x1a 0x1a
Implémente l'opérateur binaire //.	
$B \leftarrow B \quad S \leftarrow S_1 // S_0, S_1, S_2, \dots, S_{n-1}$	

BINARY_TRUE_DIVIDE

Opcodes : 0x1b | 0x1b

Implémente l'opérateur binaire / lorsque `from __future__ import division` n'est pas en vigueur.

$$B \leftarrow B \quad S \leftarrow S_1/S_0, S_1, S_2, \dots, S_{n-1}$$

INPLACE_FLOOR_DIVIDE

Opcodes : 0x1c | 0x1c

Implémente l'opérateur binaire // (version "en place").

$$B \leftarrow B \quad S \leftarrow S_1//S_0, S_1, S_2, \dots, S_{n-1}$$

INPLACE_TRUE_DIVIDE

Opcodes : 0x1d | 0x1d

Implémente l'opérateur binaire / lorsque `from __future__ import division` n'est pas en vigueur (version "en place").

$$B \leftarrow B \quad S \leftarrow S_1/S_0, S_1, S_2, \dots, S_{n-1}$$

SLICE

Opcodes : 0x1e | ----

Implémente le découpage [:].

$$B \leftarrow B \quad S \leftarrow S_0[:], S_1, S_2, \dots, S_{n-1}$$

SLICE_PLUS_1

Opcodes : 0x1f | ----

Implémente le découpage [n:].

$$B \leftarrow B \quad S \leftarrow S_1[S_0:], S_1, S_2, \dots, S_{n-1}$$

SLICE_PLUS_2

Opcodes : 0x20 | ----

Implémente le découpage [:n].

$$B \leftarrow B \quad S \leftarrow S_1[:S_0], S_1, S_2, \dots, S_{n-1}$$

SLICE_PLUS_3

Opcodes : 0x21 | ----

Implémente le découpage $[m:n]$. $B \leftarrow B \quad S \leftarrow S_2[S_1:S_0], S_1, S_2, \dots, S_{n-1}$ **STORE_SLICE**

Opcodes : 0x28 | ----

Implémente l'écriture d'un découpage $[:]$. $B \leftarrow B \quad S \leftarrow S_0[:] \leftarrow S_1, S_1, S_2, \dots, S_{n-1}$ **STORE_SLICE_PLUS_1**

Opcodes : 0x29 | ----

Implémente l'écriture d'un découpage $[n:]$. $B \leftarrow B \quad S \leftarrow S_0, S_1[S_0:] \leftarrow S_2, S_2, S_3, \dots, S_{n-1}$ **STORE_SLICE_PLUS_2**

Opcodes : 0x2a | ----

Implémente l'écriture d'un découpage $[:n]$. $B \leftarrow B \quad S \leftarrow S_0, S_1[:S_0] \leftarrow S_2, S_2, S_3, \dots, S_{n-1}$ **STORE_SLICE_PLUS_3**

Opcodes : 0x2b | ----

Implémente l'écriture d'un découpage $[m:n]$. $B \leftarrow B \quad S \leftarrow S_0, S_1, S_2[S_1:S_0] \leftarrow S_3, S_3, S_4, \dots, S_{n-1}$ **DELETE_SLICE**

Opcodes : 0x32 | ----

Implémente la destruction d'un découpage $del \ [:]$. $B \leftarrow B \quad S \leftarrow S_0 \leftarrow delS_0[:], S_1, S_2, \dots, S_{n-1}$ **GET_AITER**

Opcodes : ---- | 0x32

Dépile l'objet asynchrone au sommet de la pile et empile son l'itérateur.

 $B \leftarrow B \quad S \leftarrow S_0.iter, S_1, S_2, \dots, S_{n-1}$

DELETE_SLICE_PLUS_1

Opcodes : 0x33 | ----

Implémente la destruction d'un découpage `del [n:]`.

$$B \leftarrow B \quad S \leftarrow S_0, S_1 \leftarrow \text{del}S_1[S_0:], S_2, S_3, \dots, S_{n-1}$$
GET_ANEXT

Opcodes : ---- | 0x33

Dépile l'objet asynchrone au sommet de la pile et empile la méthode `next` de l'itérateur de l'objet asynchrone.

$$B \leftarrow B \quad S \leftarrow \text{iter.next}, S_1, S_2, \dots, S_{n-1}$$
DELETE_SLICE_PLUS_2

Opcodes : 0x34 | ----

Implémente la destruction d'un découpage `del [:n]`.

$$B \leftarrow B \quad S \leftarrow S_0, S_1 \leftarrow \text{del}S_1[:S_0], S_2, S_3, \dots, S_{n-1}$$
BEFORE_ASYNC_WITH

Opcodes : ---- | 0x34

NON DOCUMENTÉE.

DELETE_SLICE_PLUS_3

Opcodes : 0x35 | ----

Implémente la destruction d'un découpage `del [m:n]`.

$$B \leftarrow B \quad S \leftarrow S_0, S_1, S_2 \leftarrow \text{del}S_2[S_1:S_0], S_3, S_4, \dots, S_{n-1}$$
BEGIN_FINALLY

Opcodes : ---- | 0x35

Empile `NULL` en vue de l'utilisation ultérieure de `END_FINALLY`, `POP_FINALLY`, `WITH_CLEANUP_START` et `WITH_CLEANUP_FINISH`.

$$B \leftarrow B \quad S \leftarrow \text{NULL}, S_0, S_1, \dots, S_{n-1}$$

STORE_MAP

Opcodes : 0x36 | ----

La clef en S_0 et la valeur en S_1 forment un couple ajouté au dictionnaire en S_2 , puis S_0 et S_1 sont dépilés.

$$B \leftarrow B \quad S \leftarrow S_2 \cup \{S_0 : S_1\}, S_3, \dots, S_{n-1}$$
END_ASYNC_FOR

Opcodes : ---- | 0x36

Sort d'une boucle asynchrone. L'exception en S_0 , la valeur de retour en S_1 et la trace d'appels de fonctions en S_2 sont dépilées et utilisées pour restaurer un état compatible avec la suite de l'exécution.

$$B \leftarrow B \quad S \leftarrow S_3, S_4, \dots, S_{n-1}$$
INPLACE_ADD

Opcodes : 0x37 | 0x37

Implémente l'opérateur binaire + (version "en place").

$$B \leftarrow B \quad S \leftarrow S_1 S_0, S_1, S_2, \dots, S_{n-1}$$
INPLACE_SUBTRACT

Opcodes : 0x38 | 0x38

Implémente l'opérateur binaire - (version "en place").

$$B \leftarrow B \quad S \leftarrow S_1 - S_0, S_1, S_2, \dots, S_{n-1}$$
INPLACE_MULTIPLY

Opcodes : 0x39 | 0x39

Implémente l'opérateur binaire * (version "en place").

$$B \leftarrow B \quad S \leftarrow S_1 * S_0, S_1, S_2, \dots, S_{n-1}$$
INPLACE_DIVIDE

Opcodes : 0x3a | ----

Implémente l'opérateur binaire / (version "en place").

$$B \leftarrow B \quad S \leftarrow S_1 / S_0, S_1, S_2, \dots, S_{n-1}$$

INPLACE_MODULO

Opcodes : 0x3b | 0x3b

Implémente l'opérateur binaire % (version "en place") lorsque `from __future__ import division` n'est pas en vigueur.

$$B \leftarrow B \quad S \leftarrow S_1 \% S_0, S_1, S_2, \dots, S_{n-1}$$
STORE_SUBSCR

Opcodes : 0x3c | 0x3c

Implémente l'écriture à un indice [].

$$B \leftarrow B \quad S \leftarrow S_0, S_1[S_0] S \leftarrow S_2, S_2, S_3, \dots, S_{n-1}$$
DELETE_SUBSCR

Opcodes : 0x3d | 0x3d

Implémente la suppression à un indice [].

$$B \leftarrow B \quad S \leftarrow S_0, \text{del} S_1[S_0] S, S_2, S_3, \dots, S_{n-1}$$
BINARY_LSHIFT

Opcodes : 0x3e | 0x3e

Implémente l'opérateur binaire <<.

$$B \leftarrow B \quad S \leftarrow S_1 << S_0, S_1, S_2, \dots, S_{n-1}$$
BINARY_RSHIFT

Opcodes : 0x3f | 0x3f

Implémente l'opérateur binaire >>.

$$B \leftarrow B \quad S \leftarrow S_1 >> S_0, S_1, S_2, \dots, S_{n-1}$$
BINARY_AND

Opcodes : 0x40 | 0x40

Implémente l'opérateur binaire &.

$$B \leftarrow B \quad S \leftarrow S_1 \& S_0, S_1, S_2, \dots, S_{n-1}$$

BINARY_XOR	Opcodes : 0x41 0x41
Implémente l'opérateur binaire \wedge .	
$B \leftarrow B \quad S \leftarrow S_1 \wedge S_0, S_1, S_2, \dots, S_{n-1}$	

BINARY_OR	Opcodes : 0x42 0x42
Implémente l'opérateur binaire $ $.	
$B \leftarrow B \quad S \leftarrow S_1 S_0, S_1, S_2, \dots, S_{n-1}$	

INPLACE_POWER	Opcodes : 0x43 0x43
Implémente l'opérateur binaire $**$ (version "en place").	
$B \leftarrow B \quad S \leftarrow S_1 ** S_0, S_1, S_2, \dots, S_{n-1}$	

GET_ITER	Opcodes : 0x44 0x44
Dépile l'itérable en S_0 et empile son itérateur.	
$B \leftarrow B \quad S \leftarrow S_0.iter, S_0, S_1, S_2, \dots, S_{n-1}$	

GET_YIELD_FROM_ITER	Opcodes : ---- 0x45
Dépile l'itérable S_0 et empile son itérateur.	
$B \leftarrow B \quad S \leftarrow S_0.iter, S_1, \dots, S_{n-1}$	

PRINT_EXPR	Opcodes : 0x46 0x46
Utilisée seulement en mode interpréteur interactif.	
Dépile l'objet au sommet de la pile et l'affiche.	
$B \leftarrow B \quad S \leftarrow S_1, S_2, \dots, S_{n-1}$	

PRINT_ITEM	Opcodes : 0x47 ----
Affiche l'objet au sommet de la pile sur la sortie standard <code>sys.stdout</code> .	
$B \leftarrow B \quad S \leftarrow S$	

LOAD_BUILD_CLASS	Opcodes : ---- 0x47
Empile la fonction native (dans les builtins) <code>__build_class__</code> .	
$B \leftarrow B \quad S \leftarrow \text{__build_class_}, S_0, S_1, \dots, S_{n-1}$	

PRINT_NEWLINE	Opcodes : 0x48 ----
Passe à la ligne suivante sur la sortie standard <code>sys.stdout</code> .	
$B \leftarrow B \quad S \leftarrow S$	

YIELD_FROM	Opcodes : ---- 0x48
Empile la prochaine valeur <code>retval</code> du générateur en S_1 exécuté avec le paramètre dépilé S_0 .	
$B \leftarrow B \quad S \leftarrow \text{retval}, S_1, S_2, \dots, S_{n-1}$	

PRINT_ITEM_TO	Opcodes : 0x49 ----
Affiche l'objet en seconde position de la pile (S_1) sur la sortie représentée par l'objet au sommet de la pile.	
$B \leftarrow B \quad S \leftarrow S$	

GET_AWAITABLE	Opcodes : ---- 0x49
NON DOCUMENTÉE.	

PRINT_NEWLINE_TO	Opcodes : 0x4a ----
Passe à la ligne suivante sur la sortie représentée par l'objet au sommet de la pile.	
$B \leftarrow B \quad S \leftarrow S$	

INPLACE_LSHIFT

Opcodes : 0x4b | 0x4b

Implémente l'opérateur binaire << (version "en place").

$$B \leftarrow B \quad S \leftarrow S_1 \ll S_0, S_1, S_2, \dots, S_{n-1}$$

INPLACE_RSHIFT

Opcodes : 0x4c | 0x4c

Implémente l'opérateur binaire >> (version "en place").

$$B \leftarrow B \quad S \leftarrow S_1 \gg S_0, S_1, S_2, \dots, S_{n-1}$$

INPLACE_AND

Opcodes : 0x4d | 0x4d

Implémente l'opérateur binaire & (version "en place").

$$B \leftarrow B \quad S \leftarrow S_1 \& S_0, S_1, S_2, \dots, S_{n-1}$$

INPLACE_XOR

Opcodes : 0x4e | 0x4e

Implémente l'opérateur binaire ^ (version "en place").

$$B \leftarrow B \quad S \leftarrow S_1 \wedge S_0, S_1, S_2, \dots, S_{n-1}$$

INPLACE_OR

Opcodes : 0x4f | 0x4f

Implémente l'opérateur binaire | (version "en place").

$$B \leftarrow B \quad S \leftarrow S_1 | S_0, S_1, S_2, \dots, S_{n-1}$$

BREAK_LOOP

Opcodes : 0x50 | ----

Implémente la sortie anticipée d'une boucle par break.

$$B \leftarrow B \quad S \leftarrow S$$

WITH_CLEANUP

Opcodes : 0x51 | ----

NON DOCUMENTÉE.

WITH_CLEANUP_START	Opcodes : ---- 0x51
NON DOCUMENTÉE.	
LOAD_LOCALS	Opcodes : 0x52 ----
Empile une référence aux variables locales dans la portée courante.	
$B \leftarrow B \quad S \leftarrow \text{locals}, S_0, S_1, \dots, S_{n-1}$	
WITH_CLEANUP_FINISH	Opcodes : ---- 0x52
NON DOCUMENTÉE.	
RETURN_VALUE	Opcodes : 0x53 0x53
Retourne S_0 à la fonction appelante.	
$B \leftarrow B \quad S \leftarrow \emptyset$	
IMPORT_STAR	Opcodes : 0x54 0x54
Charge dans l'espace local de noms tous les symboles (dont le nom ne commence pas par <code>_</code>) contenus dans le module au sommet de la pile. Le module est ensuite dépilé.	
$B \leftarrow B \quad S \leftarrow S_1, S_2, \dots, S_{n-1}$	
EXEC_STMT	Opcodes : 0x55 ----
Implémente <code>exec S_2 S_1 S_0.</code>	
$B \leftarrow B \quad S \leftarrow S$	
SETUP_ANNOTATIONS	Opcodes : ---- 0x55
Dans les variables locales, crée un dictionnaire pour les annotations s'il n'existe pas déjà.	
$B \leftarrow B \quad S \leftarrow S$	

YIELD_VALUE

Opcodes : 0x56 | 0x56

Dépile l'objet au sommet de la pile et le retourne comme valeur renvoyée par un générateur.

$$B \leftarrow B \quad S \leftarrow S_1, S_2, \dots, S_{n-1}$$
POP_BLOCK

Opcodes : 0x57 | 0x57

Dépile la pile de blocs de contrôle.

$$B \leftarrow B_1, B_2, \dots, B_{n-1} \quad S \leftarrow S$$
END_FINALLY

Opcodes : 0x58 | 0x58

Implémente `finally`.

$$B \leftarrow B \quad S \leftarrow S$$
BUILD_CLASS

Opcodes : 0x59 | ----

Crée une nouvelle classe. S_0 contient le dictionnaire des méthodes de la classe, S_1 contient le tuple des noms des classes de base et S_2 est le nom de la classe.

$$B \leftarrow B \quad S \leftarrow S$$
POP_EXCEPT

Opcodes : ---- | 0x59

Sort d'un bloc de contrôle `except`. Le contrôle est restauré à partir de l'exception en S_0 , de la valeur de retour en S_1 et de la trace d'appels de fonctions en S_2 , qui sont dépilées.

$$B \leftarrow B_1, B_2, \dots, B_{n-1} \quad S \leftarrow S_3, S_4, \dots, S_{n-1}$$
STORE_NAME name_ref

Opcodes : 0x5a | 0x5a

Implémente `co_names[name_idx] ← S_0` .

$$B \leftarrow B \quad S \leftarrow S$$

DELETE_NAME name_idx Opcodes : 0x5b | 0x5b

Implémente `del co_names[name_idx]`.

$B \leftarrow B \quad S \leftarrow S$

UNPACK_SEQUENCE count Opcodes : 0x5c | 0x5c

Déballe l'objet au sommet de la pile pour en empiler les `count` premiers objets pris de droite à gauche.

$B \leftarrow B \quad S \leftarrow S_{0,0}, S_{0,1}, \dots, S_{0,\text{count}-1}, S_0, S_0, \dots, S_{n-1},$

FOR_ITER delta Opcodes : 0x5d | 0x5d

S_0 doit être un itérateur.

Appelle la méthode `next` de l'itérateur S_0 .

Si une nouvelle valeur `val` est générée, alors elle est empilée et :

$B \leftarrow B \quad S \leftarrow \text{val}, S_0, S_1, \dots, S_{n-1}$

Sinon on dépile S_0 et le compteur ordinal est incrémenté de `delta` :

$B \leftarrow B \quad S \leftarrow S_1, \dots, S_{n-1}$

UNPACK_EX count Opcodes : ---- | 0x5e

Dépille l'itérable en S_0 et empile les valeurs de ses $p = 1 + \text{count} \& 0x00ff + \text{count} \gg 8$ prochaines itérations.

$B \leftarrow B \quad S \leftarrow S_0.\text{iter}(0), \dots, S_0.\text{iter}(p-1), \dots, S_1, S_2, \dots, S_{n-1}$

LIST_APPEND 1 Opcodes : 0x5e | 0x91

Dépille S_0 et l'ajoute à la fin de la liste S_{l-1} .

$B \leftarrow B \quad S \leftarrow S_1, \dots, S_l.\text{append}(S_0), S_{l+1}, \dots, S_{n-1}$

STORE_ATTR name_idx Opcodes : 0x5f | 0x5f

Implémente `$S_0.\text{co_names}[\text{name_idx}] \leftarrow S_1$` .

$B \leftarrow B \quad S \leftarrow S_0.\text{co_names}[\text{name_idx}] \leftarrow S_1, S_1, S_2, \dots, S_{n-1}$

DELETE_ATTR name_idx	Opcodes : 0x60 0x60
Implémente $S_0.co_names[name_idx] \leftarrow S_1$.	
$B \leftarrow B \quad S \leftarrow delS_0.co_names[name_idx], S_1, S_2, \dots, S_{n-1}$	

STORE_GLOBAL name_idx	Opcodes : 0x61 0x61
Implémente $globals[name_idx] \leftarrow S_0$.	
$B \leftarrow B \quad S \leftarrow S$	

DELETE_GLOBAL name_idx	Opcodes : 0x62 0x62
Implémente $del\ globals[name_idx]$.	
$B \leftarrow B \quad S \leftarrow S$	

DUP_TOPX count	Opcodes : 0x63 ----
$1 \leq count \leq 5$.	
Duplique count objets sur la pile.	
$B \leftarrow B \quad S \leftarrow S_0, \dots, S_{count-1}, S_0, \dots, S_{count-1}, S_{count}, \dots, S_{n-1}$	

LOAD_CONST const_idx	Opcodes : 0x64 0x64
Empile $co_consts[const_idx]$.	
$B \leftarrow B \quad S \leftarrow co_consts[const_idx], S_0, S_1, \dots, S_{n-1}$	

LOAD_NAME name_idx	Opcodes : 0x65 0x65
Empile la valeur associée à $co_names[name_idx]$.	
$B \leftarrow B \quad S \leftarrow co_names[name_idx].value, S_0, S_1, \dots, S_{n-1}$	

BUILD_TUPLE tuple_size Opcodes : 0x66 | 0x66

Construit un tuple constitué des tuple_size premiers objets retirés de la pile, et l'empile.

$$B \leftarrow B \quad S \leftarrow (S_0, \dots, S_{\text{tuple_size}-1}), S_{\text{tuple_size}}, \dots, S_{n-1}$$

BUILD_LIST list_size Opcodes : 0x67 | 0x67

Construit une liste constituée des list_size premiers objets retirés de la pile, et l'empile.

$$B \leftarrow B \quad S \leftarrow [S_0, \dots, S_{\text{tuple_size}-1}], S_{\text{tuple_size}}, \dots, S_{n-1}$$

BUILD_SET set_size Opcodes : 0x68 | 0x68

Construit un ensemble constitué des list_size premiers objets retirés de la pile, et l'empile.

$$B \leftarrow B \quad S \leftarrow \{S_0, \dots, S_{\text{tuple_size}-1}\}, S_{\text{tuple_size}}, \dots, S_{n-1}$$

BUILD_MAP map_size Opcodes : 0x69 | 0x69

Empile un nouveau dictionnaire contenant les map_size couples de clef puis valeur au sommet de la pile.

$$B \leftarrow B \quad S \leftarrow \text{map}, S_{2 \times \text{map_size}}, \dots, S_{n-1}$$

LOAD_ATTR name_idx Opcodes : 0x6a | 0x6a

Remplace S_0 par $\text{getattr}(S_0, \text{co_names}[\text{name_idx}])$.

$$B \leftarrow B \quad S \leftarrow \text{getattr}(S_0, \text{co_names}[\text{name_idx}]), S_1, \dots, S_{n-1}$$

COMPARE_OP op Opcodes : 0x6b | 0x6b

Effectue l'opération booléenne de comparaison compare, de numéro op, sur S_1 et S_0 .

$$B \leftarrow B \quad S \leftarrow S_1 \text{ compare } S_0, S_2, S_3, \dots, S_{n-1}$$

IMPORT_NAME name_idx Opcodes : 0x6c | 0x6c

Importe le module `co_names[name_idx]`, qui est empilé. L'espace de noms courant n'est pas modifié (STORE_FAST requis).

$B \leftarrow B$ $S \leftarrow \text{module}, S_0, S_1, \dots, S_{n-1}$

IMPORT_FROM name_idx Opcodes : 0x6d | 0x6d

Empile l'attribut `co_names[name_idx]` du module au sommet de la pile.

$B \leftarrow B$ $S \leftarrow \text{attribute}, S_0, S_1, \dots, S_{n-1}$

JUMP_FORWARD offset Opcodes : 0x6e | 0x6e

Incrémente le compteur ordinal de la valeur offset.

$B \leftarrow B$ $S \leftarrow S$

JUMP_IF_FALSE_OR_POP address Opcodes : 0x6f | 0x6f

Si S_0 n'est pas évalué à True, alors le compteur ordinal prend la valeur address et la pile est inchangée :

$B \leftarrow B$ $S \leftarrow S_0, S_1, \dots, S_{n-1}$

Sinon il est dépilé :

$B \leftarrow B$ $S \leftarrow S_1, S_2, \dots, S_{n-1}$

JUMP_IF_TRUE_OR_POP address Opcodes : 0x70 | 0x70

Si S_0 est évalué à True, alors le compteur ordinal prend la valeur address et la pile est inchangée :

$B \leftarrow B$ $S \leftarrow S_0, S_1, \dots, S_{n-1}$

Sinon il est dépilé :

$B \leftarrow B$ $S \leftarrow S_1, S_2, \dots, S_{n-1}$

JUMP_ABSOLUTE address Opcodes : 0x71 | 0x71

Le compteur ordinal prend la valeur address.

$B \leftarrow B \quad S \leftarrow S$

POP_JUMP_IF_FALSE address Opcodes : 0x72 | 0x72

Dépile S_0 . S'il n'est pas évalué à True, alors le compteur ordinal prend la valeur address.

$B \leftarrow B \quad S \leftarrow S_1, S_2, \dots, S_{n-1}$

POP_JUMP_IF_TRUE address Opcodes : 0x73 | 0x73

Dépile S_0 . S'il est évalué à True, alors le compteur ordinal prend la valeur address.

$B \leftarrow B \quad S \leftarrow S_1, S_2, \dots, S_{n-1}$

LOAD_GLOBAL name_idx Opcodes : 0x74 | 0x74

Empile `co_names[name_idx]`.

$B \leftarrow B \quad S \leftarrow \text{co_names}[\text{name_idx}], S_0, S_1, \dots, S_{n-1}$

CONTINUE_LOOP address Opcodes : 0x77 | ----

Continue une boucle à cause d'un continue.

Le paramètre address doit être l'adresse d'une instruction FOR_ITER.

Le compteur ordinal prend la valeur address.

$B \leftarrow B \quad S \leftarrow S$

SETUP_LOOP delta Opcodes : 0x78 | ----

Empile un bloc de contrôle pour une boucle. Le bloc englobe les instructions depuis l'adresse de SETUP_LOOP jusqu'à celle située à SETUP_LOOP + delta (octets de bytecode).

$B \leftarrow \text{loop_block}, B_0, B_1, \dots, B_{n-1} \quad S \leftarrow S$

SETUP_EXCEPT δ Opcodes : 0x79 | ----

Empile un bloc de contrôle pour une clause try/except, SETUP_EXCEPT + δ est l'adresse de la première instruction du premier bloc except.

$B \leftarrow \text{try_block}, B_0, B_1, \dots, B_{n-1} \quad S \leftarrow S$

SETUP_FINALLY δ Opcodes : 0x7a | 0x7a

Empile un bloc de contrôle pour une clause try/except, SETUP_EXCEPT + δ est l'adresse de la première instruction du bloc finally.

$B \leftarrow \text{try_block}, B_0, B_1, \dots, B_{n-1} \quad S \leftarrow S$

LOAD_FAST var_idx Opcodes : 0x7c | 0x7c

Empile le contenu de la variable locale `co_varnames[var_idx]`.

$B \leftarrow B \quad S \leftarrow \text{co_varnames}[\text{var_idx}].\text{value}, S_0, S_1, \dots, S_{n-1}$

STORE_FAST var_idx Opcodes : 0x7d | 0x7d

Stocke S_0 dans la variable locale `co_varnames[var_idx]` :
 $\text{co_varnames}[\text{var_idx}].\text{value} \leftarrow S_0$

$B \leftarrow B \quad S \leftarrow S$

DELETE_FAST var_idx Opcodes : 0x7e | 0x7e

Détruit la variable locale `co_varnames[var_idx]`.

$B \leftarrow B \quad S \leftarrow S$

RAISE_VARARGS argc Opcodes : 0x82 | 0x82

$0 \leq \text{argc} \leq 3$.

Lève une exception. argc indique le nombre d'arguments. Le gestionnaire de l'exception trouvera la trace d'appels dans S_2 , le paramètre dans S_1 et l'exception dans S_0 .

$B \leftarrow B \quad S \leftarrow S$

CALL_FUNCTION *argc* Opcodes : 0x83 | 0x83

Empile la valeur de retour *retval* d'une fonction à exécuter. Le nombre d'arguments positionnels de la fonction est donné par *argc*&0x00ff et le nombre d'arguments mots-clefs (kwargs) est donné par *argc*>>8. Avant l'appel, la pile contient d'abord les arguments mots-clefs : pour chaque paramètre mot-clef, la valeur se situe au-dessus de la clef sur la pile. Sous les arguments mots-clefs se trouvent les arguments positionnels (l'argument le plus à droite d'abord). Enfin, on trouve sous les arguments positionnels l'objet de type fonction à appeler.

$$B \leftarrow B \quad S \leftarrow \text{retval}, S_0, S_1, \dots, S_{n-1}$$

MAKE_FUNCTION *argc* Opcodes : 0x84 | 0x84

Empile un objet de type fonction. La fonction est réputée utiliser *argc* paramètres par défaut, trouvés sous S_0 qui contient le code de la fonction à exécuter.

$$B \leftarrow B \quad S \leftarrow \text{function}, S_0, S_1, \dots, S_{n-1}$$

BUILD_SLICE *argc* Opcodes : 0x85 | 0x85

Le paramètre *argc* vaut 2 ou 3.

Empile un objet de type découpage. Si *argc* vaut 2 alors :

$$B \leftarrow B \quad S \leftarrow \text{slice}(S_1, S_0), S_0, S_1, S_2, \dots, S_{n-1}$$

Sinon :

$$B \leftarrow B \quad S \leftarrow \text{slice}(S_2, S_1, S_0), S_0, S_1, S_2, \dots, S_{n-1}$$

MAKE_CLOSURE *argc* Opcodes : 0x86 | ----

Empile un objet de type fonction (une fermeture est une fonction référençant $v > 0$ variables libres, donc l'attribut *func_closure* de l'objet fonction est mis à jour en conséquence). S_0 contient le code de la fermeture à exécuter. Dessous, on trouve les v objets qui sont les cellules correspondant aux variables libres. La fonction utilise *argc* paramètres par défaut, dont les valeurs sont sous les cellules.

$$B \leftarrow B \quad S \leftarrow \text{function}, S_0, S_1, \dots, S_{n-1}$$

LOAD_CLOSURE $fvar_idx$ Opcodes : 0x87 | 0x87

Empile la cellule `freevars[fvar_idx]` de la variable libre de numéro `fvar_idx`.

$B \leftarrow B \quad S \leftarrow freevars[fvar_idx], S_0, S_1, \dots, S_{n-1}$

LOAD_DEREF $fvar_idx$ Opcodes : 0x88 | 0x88

Empile l'objet contenu dans la cellule `freevars[fvar_idx]`.

$B \leftarrow B \quad S \leftarrow freevars[cell_idx].value, S_0, S_1, \dots, S_{n-1}$

STORE_DEREF $fvar_idx$ Opcodes : 0x89 | 0x89

Dépile S_0 pour le stocker dans la cellule `freevars[fvar_idx]`.

`freevars[fvar_idx].value` $\leftarrow S_0$.

$B \leftarrow B \quad S \leftarrow S_1, S_2, \dots, S_{n-1}$

DELETE_DEREF $fvar_idx$ Opcodes : ---- | 0x8a

Détruit l'objet dans la cellule `freevars[fvar_idx]`.

$B \leftarrow B \quad S \leftarrow S$

CALL_FUNCTION_VAR $argc$ Opcodes : 0x8c | ----

Le paramètre `argc` est interprété comme pour `CALL_FUNCTION`. S_0 contient la liste des arguments variables, suivie par les arguments mots-clefs et les arguments positionnels.

$B \leftarrow B \quad S \leftarrow S$

CALL_FUNCTION_KW $argc$ Opcodes : 0x8d | 0x8d

Le paramètre `argc` est interprété comme pour `CALL_FUNCTION`. S_0 contient le dictionnaire des arguments mots-clefs, suivi par les arguments mots-clefs explicites et les arguments positionnels.

$B \leftarrow B \quad S \leftarrow S$

CALL_FUNCTION_VAR_KW *argc* Opcodes : 0x8e | ----

Le paramètre *argc* est interprété comme pour **CALL_FUNCTION**. S_0 contient le dictionnaire des arguments mots-clefs, suivi par la liste des arguments variables, puis par les arguments mots-clefs explicites et les arguments positionnels.

$B \leftarrow B \quad S \leftarrow S$

CALL_FUNCTION_EX *has_kwargs* Opcodes : ---- | 0x8e

Si *has_kwargs* vaut 1, alors le dictionnaire des arguments mots-clefs en S_0 est dépilé. Ensuite le tuple des arguments positionnels est dépilé et la fonction qui suit voit sa valeur de retour *retval* empilée.

Si *has_kwargs* vaut 1 :

$B \leftarrow B \quad S \leftarrow \text{retval}, S_2, S_3, \dots, S_{n-1}$

Sinon :

$B \leftarrow B \quad S \leftarrow \text{retval}, S_1, S_2, \dots, S_{n-1}$

SETUP_WITH *delta* Opcodes : 0x8f | 0x8f

Équivalent à **SETUP_FINALLY** mais en "normalisant" l'exception.

$B \leftarrow \text{try_block}, B_0, B_1, \dots, B_{n-1} \quad S \leftarrow S$

EXTENDED_ARG *extend* Opcodes : 0x91 | 0x90

Fournit le support pour les paramètres de plus de 16 bits :

$\text{parm_high} \leftarrow \text{extend}$

$B \leftarrow B \quad S \leftarrow S$

SET_ADD *s* Opcodes : 0x92 | 0x92

Dépile S_0 et l'ajoute à l'ensemble dans S_{s-1} .

$B \leftarrow B \quad S \leftarrow S_1, \dots, S_s \cup \{S_0\}, S_{s+1}, \dots, S_{n-1}$

MAP_ADD *m*

Opcodes : 0x93 | 0x93

Dépile la clef S_0 et la valeur S_1 , et en ajoute le couple au dictionnaire dans S_{m-1} .

$$B \leftarrow B \quad S \leftarrow S_2, \dots, S_{m+1} \cup \{S_0 : S_1\}, S_{m+2}, \dots, S_{n-1}$$

LOAD_CLASSDEREF

Opcodes : ---- | 0x94

NON DOCUMENTÉE.

BUILD_LIST_UNPACK *count*

Opcodes : ---- | 0x95

Dépile les *count* premiers objets pour les placer dans une liste qui sera empilée.

$$B \leftarrow B \quad S \leftarrow [S_0, \dots, S_{\text{count}-1}], S_{\text{count}}, \dots, S_{n-1}$$

BUILD_MAP_UNPACK *map_size*

Opcodes : ---- | 0x96

Empile un nouveau dictionnaire contenant les *map_size* couples clef/valeur au sommet de la pile.

$$B \leftarrow B \quad S \leftarrow \text{map}, S_{2 \times \text{map_size}}, \dots, S_{n-1}$$

BUILD_MAP_UNPACK_WITH_CALL *map_size* Opcodes : ---- | 0x97

NON DOCUMENTÉE.

BUILD_TUPLE_UNPACK *count*

Opcodes : ---- | 0x98

Dépile les *count* premiers objets pour les placer dans un tuple qui sera empilé.

$$B \leftarrow B \quad S \leftarrow (S_0, \dots, S_{\text{count}-1}), S_{\text{count}}, \dots, S_{n-1}$$

BUILD_SET_UNPACK count Opcodes : ---- | 0x99

Dépile les `count` premiers objets pour les placer dans un tuple qui sera empilé.

$$B \leftarrow B \quad S \leftarrow \{S_0, \dots, S_{\text{count}-1}\}, S_{\text{count}}, \dots, S_{n-1}$$

SETUP_ASYNC_WITH Opcodes : ---- | 0x9a

NON DOCUMENTÉE.

FORMAT_VALUE fmt_spec Opcodes : ---- | 0x9b

Empile un formatage texte en dépilant éventuellement les spécifications de formatage en S_0 . L'objet à formater est dépilé puis sa chaîne formatée est empilée.

Si présence de spécifications de formatage :

$$B \leftarrow B \quad S \leftarrow \text{string}, S_2, S_3, \dots, S_{n-1}$$

Sinon :

$$B \leftarrow B \quad S \leftarrow \text{string}, S_1, S_2, \dots, S_{n-1}$$

BUILD_CONST_KEY_MAP map_size Opcodes : ---- | 0x9c

Le tuple de clefs en S_0 est dépilé pour former, à l'aide des valeurs qui sont dépilées, les couples du nouveau dictionnaire qui sera empilé.

$$B \leftarrow B \quad S \leftarrow \text{map}, S_{\text{map_size}}, \dots, S_{n-1}$$

BUILD_STRING count Opcodes : ---- | 0x9d

Dépile les `count` premiers objets caractères Unicode pour les placer dans une chaîne Unicode qui sera empilée.

$$B \leftarrow B \quad S \leftarrow "S_0, \dots, S_{\text{count}-1}", S_{\text{count}}, \dots, S_{n-1}$$

BUILD_TUPLE_UNPACK_WITH_CALL n Opcodes : ---- | 0x9e

NON DOCUMENTÉE.

LOAD_METHOD name_idx Opcodes : ---- | 0xa0

NON DOCUMENTÉE.

CALL_METHOD argc Opcodes : ---- | 0xa1

NON DOCUMENTÉE.

CALL_FINALLY offset Opcodes : ---- | 0xa2

Empile l'adresse de l'instruction suivante. La valeur du compteur ordinal est incrémentée de la valeur offset.

$$B \leftarrow B \quad S \leftarrow \text{return_adress}, S_0, S_1, \dots, S_{n-1}$$

POP_FINALLY Opcodes : ---- | 0xa3

NON DOCUMENTÉE.

Annexe C

Sérialisation des objets Python

Nous décrivons dans ce chapitre comment écrire des objets Python pour en faire une série d'octets que la VM Python pourra relire afin de les charger en mémoire. On parle alors de *sérialisation* des objets Python. De nombreux autres formats, notamment dans le monde des technologies Web, existent pour sérialiser des données (JSON, XML, *etc.*)

C.1 Principe de la sérialisation en Python

Le principe général est la suivant : un objet Python est écrit à l'aide d'un marqueur, un octet qui indique le type de l'objet Python, suivi du contenu de l'objet en question.

Nous donnons à la Tab. [C.1](#) la liste des marqueurs reconnus par Python. Tous les objets ne seront pas à sérialiser dans ce projet. Nous donnons dans la suite la procédure de sérialisation des objets Python utilisés dans ce projet.

C.2 Sérialisation des scalaires

C.2.1 Sérialisation des constantes

Les constantes (`null`, `True`, `False`, `None`) sont codées uniquement par leur marqueur.

C.2.2 Sérialisation des nombres

Un entier (`INT`) est codé avec les quatre octets de sa valeur en *little endian*. Dans ce projet, on ne produira pas d'entiers de type `INT64` ou `LONG`.

Un réel peut être codé de deux manières : soit en écrivant les huit octets du `double` codant sa valeur (`BINARY_FLOAT`), soit en écrivant la chaîne de caractères de sa valeur (`FLOAT`). Idéalement, le choix entre les deux se fait de manière à minimiser la place occupée par les données sérialisées.

Un nombre complexe (`BINARY_COMPLEX` et `COMPLEX`) se code exactement comme un réel, en ajoutant simplement la partie imaginaire à la suite de la partie réelle. Si un nombre complexe n'a pas de partie imaginaire, il est codé comme un réel s'il a une partie fractionnaire, et comme un entier sinon.

C.2.3 Sérialisation des chaînes de caractères

Par défaut, Python utilise en interne la représentation Unicode des caractères (un sur-ensemble des codes ASCII). Nous n'aurons pas cette prétention de gérer l'Unicode et nous nous bornerons à utiliser l'alphabet latin. En conséquence, les types `STRING`, `ASCII`, `SHORT_ASCII`, `UNICODE` et `*INTERNE`d seront pour nous synonymes.

Lorsque le nom du type contient `SHORT`, le marqueur est suivi d'*un octet* indiquant la taille de la chaîne, sinon il est suivi de *quatre octets* pour coder la taille de la chaîne. Une chaîne longue (de plus de 255 caractères) est donc introduite avec trois octets de plus qu'une chaîne courte.

Dans le but de ne pas utiliser trop de mémoire, Python prévoit de pouvoir interner des chaînes de caractères : une chaîne dite *internée* est stockée par la VM Python dans un espace séparé, en lecture seule, qui permet de faire des références à ces chaînes. On indique qu'une chaîne doit être internée en utilisant un type de chaîne contenant `INTERNE`d (`INTERNE`d, `ASCII_INTERNE`d ou `SHORT_ASCII_INTERNE`d). Les chaînes internées sont indicées par leur ordre d'apparition dans le bytecode.

On fait référence à une chaîne internée en utilisant le type `STRINGREF` suivi de son indice.

C.3 Sérialisation des tuples et des listes

Une liste en Python est en réalité plus proche d'un tableau que de ce que le reste du vaste monde appelle une liste, que Python appelle un tuple. Mais bref...

Pour coder une liste (`LIST`) ou un tuple (`TUPLE`), on commence par coder le nombre d'éléments de la liste ou du tuple avec quatre octets, suivis du codage des éléments. Python 3.x prévoit un marqueur spécial pour les petits tuples (`SMALL_TUPLE`) contenant moins de 256 éléments et dont le nombre d'éléments peut donc être codé à l'aide d'un seul octet (exactement comme pour les petites chaînes).

Dans ce projet, nous ne supporterons pas les dictionnaires (`DICT`) ni les ensembles (`SET` et `FROZENSET`).

C.4 Sérialisation du code

Un fichier `.pyc` est constitué d'un en-tête (cf. Sec. 11.2) suivi de la sérialisation d'un objet de type `CODE`. Et cet objet va contenir la sérialisation des objets Python utilisés par le code (constantes, chaînes, tuples, etc.)

Si vous avez suivi ce que nous avons raconté au sujet des grammaires formelles, on peut dire qu'un objet de type `CODE` joue le rôle d'axiome pour la sérialisation.

C.4.1 En-tête d'un objet de code Python



Un objet de code Python est encodé en commençant par écrire les entiers non signés suivants (sans marqueur) :

1. le nombre total d'arguments du code (`arg_count`);
2. [Python \geq 3.8] : le nombre d'arguments positionnels (`posonly_arg_count`);
3. [Python \geq 3.8] : le nombre d'arguments mots-clefs (`kwonly_arg_count`);
4. le nombre de variables locales (`local_count`);
5. la taille de la pile nécessaire pour l'exécution (`stack_size`);
6. les propriétés du code (`flags`), cf. Sec. C.4.2 *infra*.

C.4.2 Le champ `flags` de l'en-tête

L'entier codant les propriétés du bout de code à exécuter est formé par un OU logique entre les valeurs de la Tab. C.2.

Notez que le compilateur Python fixe la valeur du champ `flags` et que le plus souvent, il vous suffira de la recopier – sauf si vous voulez jouer avec !

Comme un peu d'humour ne nuit jamais, remarquez le nom du drapeau `FUTURE_BARRY_AS_BDFL` : le développement de Python a été supervisé jusqu'au 12 juillet 2018 par son inventeur Guido van Rossum, que la communauté des développeurs Python a surnommé le *dictateur bienveillant à vie* ( *BDFL* – *Benevolent Dictator For Life*). Van Rossum ayant été épuisé par près de 20 ans de cette tâche éprouvante consistant au surplus à gérer des *égos surdimensionnés*, il faut conclure qu'une campagne secrète pour le prochain BDFL s'est opérée ici en faveur de Barry Warsaw, surnommé *FLUFL* ( *Friendly Language Uncle For Life*). Plus sérieusement, le [PEP-0401](#) (qui date tout de même du premier avril 2009...) remplace aussi dans Python 3.x l'opérateur `!=` par `<>` pour tester la non-égalité.¹

C.4.3 Corps d'un objet de code Python

Une fois l'en-tête écrit, il faut écrire le corps de l'objet Python, qui est constitué :

1. Autre exemple de blague dans une norme, dont l'IETF est coutumière : l'[erreur 418](#) dans le protocole HTTP.

1. des instructions compilées, sous forme d'une chaîne (STRING) ;
2. du tuple des constantes (directive `.consts`) ;
3. du tuple des noms de symboles (directive `.names`) ;
4. du tuple des noms de variables (directive `.varnames`) ;
5. du tuple des noms de variables libres (directive `.freevars`) ;
6. du tuple des cellules (directive `.cellvars`).

Si un tuple est vide, il est tout de même écrit, mais avec une taille de zéro.

C.4.4 Fin d'un objet de code Python

Enfin, il faut écrire les informations suivantes, qui constituent la fin d'un objet de code Python :

1. la chaîne (STRING) du nom de fichier source Python (`filename`) ;
2. la chaîne (STRING) du nom du code (`name`) ;
3. les quatre octets de l'entier donnant le numéro de la première ligne dans le code Python (`firstlineno`) – ce sera souvent la valeur 1 pour nous ;
4. la chaîne `lnotab` (STRING) codant les directives `.line` (et donc permettant de savoir quelles instructions correspondent à quelle ligne de code source Python), *cf. infra*.

C.4.5 `lnotab`

Nous ne supporterons pas toutes les finesses de l'encodage de `lnotab`, pour nous contenter de son fonctionnement général.

Le tableau d'octets (STRING) `lnotab` est constitué de paires d'entiers codant alternativement les incréments entre adresses dans le bytecode et entre lignes de code Python.

Nous donnons à la Tab. C.3 un exemple de correspondance entre l'adresse dans le bytecode et sa ligne de code source Python (par exemple, la première ligne du code source a nécessité 6 octets d'instructions diverses).

Le codage de `lnotab` est tout bêtement formé des différences par rapport à la paire précédente, et on fait l'hypothèse d'une première paire non écrite constituée de deux zéros.

Ainsi, dans l'exemple de la Tab. C.3, aurons-nous les valeurs suivantes pour `lnotab` : 0, 1, 6, 1, 44, 5.

Il s'agit d'un codage différentiel, très utilisé par ailleurs (et appelé DPCM en multimédia, *cf.* l'encodage des valeurs moyennes d'un bloc JPEG par exemple).

Comme chaque valeur est un octet, nous passons courageusement sous silence l'encodage de différences supérieures à 255. Le lecteur avide de précision et muni d'aspirine pourra consulter le fichier [Objects/lnotab_notes.txt](#) du code source de Python pour les détails horribles.

C.5 Une spécificité de Python 3.x

De la même manière que des chaînes peuvent être internées pour économiser de la mémoire, Python 3.x introduit en outre la possibilité d'indexer tout type d'objet. Les objets indexés sont indicés par leur ordre d'apparition dans le bytecode et on y fait référence à l'aide du type (REF) suivi de l'indice de l'objet indexé.

Un objet est indexé en *marquant le marqueur de type* : on ajoute la valeur 0x80 au marqueur. Par exemple, une chaîne (STRING) de marqueur 0x73 (la valeur du code ASCII 's') sera indexée en écrivant le marqueur de type 0xf3 (0x73+0x80).

Type	Marqueur	Min. version	Observation
NULL	'0'	2.7	Absence d'objet
NONE	'N'	2.7	None
FALSE	'F'	2.7	False
TRUE	'T'	2.7	True
INT	'i'	2.7	Entier signé sur 4 octets
INT64	'I'	2.7	Entier signé sur 8 octets, plus généré
FLOAT	'f'	2.7	Chaîne d'un réel (max : 17 caractères)
BINARY_FLOAT	'g'	2.7	Réel binaire sur 8 octets (double)
COMPLEX	'x'	2.7	Deux chaînes pour un complexe
BINARY_COMPLEX	'y'	2.7	Deux réels pour un complexe
STRING	's'	2.7	Chaîne
REF	'r'	3.x	Référence à un objet
STRINGREF	'R'	2.7	Référence à une chaîne internée
TUPLE	'('	2.7	Tuple
SMALL_TUPLE	')'	3.x	Petit tuple
LIST	'['	2.7	Liste
Dict	'{'	2.7	Dictionnaire
SET	'<'	2.7	Ensemble
ASCII	'a'	3.x	Chaîne ASCII → Unicode
ASCII_INTERNEDED	'A'	3.x	<i>idem</i> mais à interner
SHORT_ASCII	'z'	3.x	Petite chaîne ASCII → Unicode
SHORT_ASCII_INTERNEDED	'Z'	3.x	<i>idem</i> , à interner
CODE	'c'	2.7	Objet de code
STOP_ITER	'S'	2.7	Arrêt d'un itérateur
ELLIPSIS	'.'	2.7	...
LONG	'l'	2.7	Entier signé en base 15
UNICODE	'u'	2.7	Chaîne Unicode
INTERNEDED	't'	2.7	<i>idem</i> , à interner
UNKNOWN	'?'	2.7	Objet de type inconnu
FROZENSET	'>'	2.7	Ensemble en lecture seule

TABLE C.1 – Liste des marqueurs de sérialisation des objets Python.

Propriété	Valeur
OPTIMIZED	0x0001
NEWLOCALS	0x0002
VARARGS	0x0004
VARKEYWORDS	0x0008
NESTED	0x0010
GENERATOR	0x0020
NOFREE	0x0040
COROUTINE	0x0080
ITERABLE_COROUTINE	0x0100
ASYNC_GENERATOR	0x0200
FUTURE_DIVISION	0x20000
FUTURE_ABSOLUTE_IMPORT	0x40000
FUTURE_WITH_STATEMENT	0x80000
FUTURE_PRINT_FUNCTION	0x100000
FUTURE_UNICODE_LITERALS	0x200000
FUTURE_BARRY_AS_BDFL	0x400000
FUTURE_GENERATOR_STOP	0x800000
FUTURE_ANNOTATIONS	0x1000000

TABLE C.2 – Propriétés possibles d'un objet de code Python.

Offset dans le bytecode	Numéro de ligne du code Python
0	1
6	2
50	7

TABLE C.3 – Exemple de valeurs à encoder dans `lnotab`.

Annexe D

Outil disponible

D.1 pyc-objdump

Voici l'aide de l'outil pyc-objdump :

```
$ pyc-objdump --help

pyc-objdump -- A PYC objdump.

Synopsis: pyc-objdump action file.pyc [options]

Actions:

    dump                Dump PYC info (default action).
    disasm              Disassemble PYC bytecode.

Options:

    [-c|--code]         Select function (separator: colon).
    [--pretty]          Pretty print instead of PYS output.
    [--easy]            Jump insns have numeric args.
    [-v|-vv|-vvv]      Gradually increase verbose level.
```

Vous pouvez l'utiliser pour générer des fichiers .pys à partir de fichiers .pyc à l'aide de la commande disasm.

```
$ python2.7 -m compileall totor.py
Compiling totor.py ...
$ pyc-objdump disasm totor.pyc > totor.pys
$
```

En ajoutant l'option `--easy`, vous obtiendrez un code assembleur avec des valeurs numériques à la place des étiquettes. C'est peut-être plus facile pour commencer.

Les options `-v|-vv|-vvv` permettent d'augmenter progressivement le niveau de verbosité de la sortie, notamment pour observer le décodage des octets d'un fichier `.pyc`, ce qui sera en retour utile pour mettre au point votre génération du bytecode.

Cet outil fournit un support quasi-exhaustif pour Python 2.7 et relativement complet pour Python 3.x (pas de support pour les objets indexés notamment).

D.2 Installation

`pyc-objdump` est disponible pour les architectures `amd64` sur les distributions Linux basées sur Debian (Ubuntu, Mint, *etc.*)

Voici les commandes à entrer pour ajouter le dépôt et le logiciel :

1. `sudo wget -O /etc/apt/sources.list.d/uvolante.sources
https://www.uvolante.org/apt/uvolante.sources`
2. `sudo apt update`
3. `sudo apt install pyc-objdump`

Liste des codes source

3.1	Mécanisme de base pour reconnaître les expressions régulières pouvant contenir des caractères (y compris le caractère . (un point) désignant n'importe quel caractère) et l'opérateur * (d'après Rob Pike).	27
3.2	Code d'illustration du Code 3.1 (<code>regexp-match.c</code>).	29
4.1	Structure de lexème.	38
4.2	Exemple de fichier de définition de types de lexèmes pour l'analyse lexicale d'un pseudo-langage quelconque – à adapter bien sûr pour notre langage assembleur Python.	40
4.3	Deux manières de tester si un lexème est d'un certain type.	41
6.1	Implantation de l'axiome de la grammaire des expressions arithmétiques.	47
6.2	Implantation du non-terminal <code><factor></code> de la grammaire des expressions arithmétiques.	48
6.3	Notre fichier de définition de types de lexèmes pour l'analyse lexicale d'une expression arithmétique.	49
7.1	Objet permettant de représenter des relations ou des entités arithmétiques (nombres et variables).	54
7.2	Implantation de la construction de l'arbre de syntaxe abstraite pour l'axiome de la grammaire des expressions arithmétiques (cf. Code 6.1 pour comparaison).	55
9.1	Début de définition du type C pour les objets Python.	73
9.2	Définition d'un objet Python de type code pour toutes les versions de Python.	75
9.3	Objet de code Python 2.7 uniquement.	76
10.1	Exemple de code source Python.	84
10.2	Code en assembleur du code Python de la Fig. 10.1.	85
12.1	La principale qualité d'Hégésippe Simon est un pragmatisme qui ne demande qu'à s'étoffer. Ce code est censé créer une liste d'entiers à partir d'un tableau d'entiers.	106
12.2	Félix-François Hébert est un gros malin ! Il a compris qu'en parcourant le tableau <code>integers</code> il fallait utiliser une file puis la convertir en liste à la fin – ce qui ne coûte quasiment rien et est très élégant !	107

12.3 Voltairine Brewster y voit quand même un peu plus clair! On peut faire le même boulot en 12 lignes au lieu de 20 simplement en utilisant une liste tout le long mais en parcourant <code>integers</code> à l'envers!	108
12.4 Sigismond Ciboulot est un être qui ne souffre plus depuis qu'il a réalisé la toute-puissance de la Loi de Murphy.	109
A.1 Comment écrire des tests unitaires en C avec le code fourni. . .	122

Table des figures

1.1	Grammaire en EBNF des expressions arithmétiques avec expressions parenthésées (<i>cf.</i> la dernière alternative du non-terminal <code><factor></code>) et variables (dont le nom ne peut pas démarrer par un chiffre, <i>cf.</i> le non-terminal <code><identifieur></code>).	18
3.1	Grammaire en EBNF des expressions régulières du projet. . . .	22
3.2	Coquetterie platonicienne assez largement inutile pour nous. . .	34
3.3	Une syntaxe un peu moins psycho-rigide (qui frise même le laxisme) pour les expressions régulières.	35
4.1	Grammaire en EBNF du fichier de description de types de lexèmes. On comprend intuitivement que le chargement d'un fichier de définitions de lexèmes (le résultat de l'analyse en partant de <code><lexdefs></code>) doit être une liste de couples de chaînes de caractères (<code><lextype></code> , <code><regex></code>) extraites d'une même ligne.	39
6.1	Grammaire en EBNF des expressions arithmétiques avec source extérieure de lexèmes de différents types.	46
8.1	Boucle simplifiée de l'interpréteur Python. En rouge : ce que nous nous proposons de réaliser dans ce projet.	64
10.1	Grammaire en EBNF du langage assembleur Python.	80
10.2	Grammaire en EBNF du langage assembleur Python (suite). . .	81
11.1	Représentation de la valeur <code>0xdeadbeef</code> en <i>little-endian</i>	87
11.2	Représentation de la valeur <code>0xdeadbeef</code> en <i>big-endian</i>	87
11.3	En-tête d'un fichier <code>.pyc</code> pour Python 2.7.	88
11.4	En-tête d'un fichier <code>.pyc</code> pour Python ≥ 3.2 (PEP-3147).	88
11.5	En-tête d'un fichier <code>.pyc</code> pour Python ≥ 3.8 (PEP-0552).	88
12.1	Extension de la grammaire en EBNF du langage assembleur Python pour le support des fonctions.	99

12.2 Fonction d'équilibre dans le trinôme : $\frac{1}{2} + \frac{1}{1+e^{-\frac{1}{3}(P_i-10)}}$ en bleu –	
et dans le quadrinôme : $\frac{1}{2} + \frac{1}{1+e^{-\frac{1}{4}(P_i-10)}}$ en vert.	115

Bibliographie

[1] Andy Oram and Greg Wilson. *Beautiful Code*. O'Reilly, USA.