naturL
The NaturL Foundation

par

Simon Scatton
Vlad Argatu
Rostan Tabet
Adrian Le Comte

Produit dans le cadre :
Projet de deuxième semestre

INFOSUP

Rapport de Projet

EPITA

Villejuif

2020

Table des matières

1. Le langage naturL

1.1 Introduction

La langage naturL a été produit par quatre étudiants en première année à l'École Pour l'Informatique et les Techniques Avancées en 2020 en tant que projet de second semestre. Sa création s'inscrit dans un objectif pédagogique double : pour ses créateurs et ses utilisateurs. L'objectif principal de naturL est de facilité l'apprentissage de l'algorithmique en fournissant un langage qui se rapproche le plus du pseudo code tout en gardant une application pratique grâce à son interfaçage avec Python. En effet, tout code correctement écrit en naturL peut être transpilé (ou traduit) vers le langage Python et ensuite exécuté ou interprété. Avec naturL viens également le logiciel idL qui fait office d'interface de développement "officielle" pour la langage naturL. L'intérêt pédagogique d'un langage comme naturL réside dans une fusion entre un vérificateur de type similaire à celui d'OCaml et une syntaxe très simplifiée et proche de Python, ce qui fait de naturL un langage fortement et statiquement typé avec un système d'erreurs se prêtant plus à une utilisation algorithmique que le système d'erreurs de Python.

1.2 Utilisation du programme

1.2.1 Installation

Systèmes Unix

Pour installer naturL sur Unix il est nécessaire de télécharger le code source et de le compiler. Le code source est disponible à cette adresse. Après avoir cloné le code source il suffit d'exécuter la commande dans le README et le logiciel est installé.

Windows

Pour installer naturL sur Windows, il vous pouvez télécharger l'installateur directement sur le site web officiel de La ℕaturL Foundation.

Puis laissez vous guider par l'installateur automatique.

1.3 Syntaxe élémentaire

Tous les codes contenus dans cette section sont transpilés à via la commande naturL Unix ou le programme naturL sur Windows et exécutés à l'aide d'un interpréteur Python.

1.3.1 Déclarer et affecter une variable

Types de données

Les différents types de données primitifs présent en naturL sont précisés si dessous :

- entier : Représente un nombre entier
- reel : Représente un nombre réel (ou nombre à virgule)
- booleen : Représente une valeur de vérité (vrai ou faux)
- caractere : Représente un unique caractère.
- chaine : Représente une chaîne de caractères soit un ou plusieurs caractères (y compris les espaces).

naturL est un langage de programmation fortement typé de manière statique. Mais il existe cependant des compatibilités entre les types en naturL. Par exemple, le type entier est compatible avec le type reel ce qui permet les opérations arithmétiques entre les deux sans aucune forme de conversion ou de "cast" de l'un vers l'autre. Cependant il n'est pas possible d'ajouter un entier à un booleen en naturL.

Déclaration

En naturL, la déclaration se fait dans le champ "variables" il se forme de cette façon :

```
variables
    entier var
fin
```

Python ne nécessite pas de déclaration pour une variable mais naturL si, traduire le code précèdent en Python n'aurait donc pas beaucoup de sens.

Affectation

Pour affecter une valeur à une variable, on utilise l'opérateur d'affectation. En naturL il est représenté par une flèche vers la gauche (<-). Si l'on veut affecter la valeur 2 à la variable var il faudra donc écrire :

```
variables
    entier var
fin
var <- 2
```

Qui se traduira par le code Python suivant :

```
var = 2
```

On remarque immédiatement qu'en Python, le type n'as pas à être précisé. Il est "inféré" c'est à dire laissé indéterminé jusqu'à ce qu'une affectation le détermine [1].

Cependant si l'on essaye d'affecter une valeur d'un type différent que celui déclaré, cela lève une erreur de type. Par exemple le code suivant :

```
variables
    entier var
fin
var <- vrai
```

Renvoie

```
Erreur de Type à la ligne 4: 'var' a le type 'entier'
mais le type affecté est 'booleen'
```

Similairement :

```
variables
    entier pi
fin
pi <- 3.1415
```

1. En naturL il est possible de laisser le programme inférer le type grâce au type " ?" qui permet un polymorphisme (plus d'explications dans la partie liste)

Renvoie

```
Erreur de Type à la ligne 4: 'pi' a le type 'entier'
mais le type affecté est 'reel'
```

Pour préciser qu'un nombre est du type réel il faut rajouter un point. Par exemple 2 est un entier mais 2. est un réel :

```
variables
    reel var
fin
var <- 2.
```

Nous donne

```
var = 2.
```

Enfin on peut affecter une variable à une autre du moment que leurs types sont les mêmes ou qu'ils sont compatibles (on peut affecter un entier à un réel mais pas un réel à un entier) :

```
variables
    reel var
    reel var2
fin
var <- 2.
var2 <- var
```

```
var = 2.
var2 = var
```

Cependant le code suivant donne une erreur de type :

```
variables
    entier var
    reel var2
fin
var2 <- 2.
var <- var2
```

Donne l'erreur suivante :

```
Erreur de Type à la ligne 6: 'var' a le type 'entier' mais le type affecté est 'r
```

L'incrémentation se fait de cette manière :

```
variables
    entier var
fin
var <- 1
var <- var + 1
```

L'affectation des différents types de données se fait de manières similaire :

```
variables
  entier a
    reel b
  caractere c
  chaine d
  booleen e
fin
a <- 2
b <- 3.14
c <- 'e'
d <- "Hello world"
e <- faux
```

Nous donne en Python :

```
a = 2
b = 3.14
c = 'e'
d = "Hello world"
e = False
```

Les caractères sont représentés par des guillemets simple et les chaînes par des guillemets doubles. Les booléens sont une valeur de vérité donc soit faux soit vrai.

1.3.2 Opérations arithmétiques de base

Il existes deux types pour représenter les nombres en naturL, le type entier et le type réel. A l'origine naturL ne devait pas supporter les opérations entre ces deux types mais il a été décidé après coup que cela rendait l'utilisation du langage trop complexe. Les types entier et réels sont donc dits compatibles :

il est possible d'effectuer des opérations arithmétiques particulières entre les réels et les entiers.

Considérons le code suivant :

```
variables
    reel a
fin
a <- 2.5 + 1
```

Qui retourne en python

```
a = 3.5
```

Certains opérateurs sont plus restrictifs sur la compatibilité tu type, par exemple : la division entière n'est pas correcte avec un flottant :

```
variables
    reel a
fin
a <- 2.5 div 1
```

Qui retourne l'erreur suivante :

```
Erreur de Type à la ligne 4: Opération invalide pour les
expression de type reel et de type entier
```

Cependant il est totalement possible d'affecter le résultat d'une opération entière à un réel :

```
variables
    reel a
fin
a <- 3 div 2
```

Qui retourne en python

```
a = 1
```

Avec afficher la fonction qui permet d'écrire dans la sortie standard, transformée en la fonction print de Python.

1.3.3 Simplificateur logique et arithmétique

Le programme naturL possède un système de simplifications arithmétiques et logiques. Ainsi avant même l'exécution il va simplifier les expression qui sont simplifiable. Cette simplification automatique est accompagné d'avertissement ou de "warnings" dans le cas de la simplification des expressions booléennes afin de notifier l'utilisateur d'une possible erreur logique dans le code. Ce simplificateur a donc aussi un objectif pédagogique qui est de montrer à l'utilisateur qui apprend l'algorithmique ses erreurs de logique fondamentales que l'on commet souvent lorsque l'on débute.

Par exemple, le code :

```
variables
    entier var
fin
var <- 2 + 5 * 4
```

Nous donne

```
var = 22
```

En appliquant les priorités sur les opérations. Si l'on ajoute des parenthèses on obtient un résultat différent par exemple :

```
variables
    entier var
fin
var <- (2 + 5) * 4
```

Nous donne

```
var = 28
```

Si l'on souhaite afficher dans un interpréteur Python des variables, on peut utiliser la fonction AFFICHER qui se traduit en la fonction PRINT de Python. Par exemple :

```
variables
    entier var
fin
var <- (2 + 5) * 4
afficher(var)
```

Nous donne

```
var = 28
print(var)
```

Après exécution du script python dans un terminal on obtient :

```
28
```

La simplification des expressions données par l'utilisateur en NATURL est une étape utile dans l'optimisation du code. Elle agit en remplaçant les expressions simplifiables par leur version la plus simple lorsqu'elles sont traduites en PYTHON par le transpileur. Cette fonctionnalité permet notamment de réduire les expressions telles que *« 1 + 2 »* en *« 3 »* ou encore *« 1 = 1 »* en *« True »* permettant ainsi de traduire différentes structures de manière plus propre. On aura ainsi le code suivant :

```
si 1 = 1 alors
    a <- 56 + 4
    b <- "Hello" + " World"
fin
```

traduit par :

```
if True:
    a = 60
    b = "Hello World"
```

La simplification est encore plus poussée car elle permet aussi de simplifier au maximum les expressions logiques en prenant en compte les éléments neutres et absorbant traduisant ainsi :

```
procedure f(booleen a)
    si 1 = 1 et a alors
    fin
    si 1 = 2 et a alors
    fin
    si 1 = 1 ou a alors
    fin
    si 1 = 2 ou a alors
    fin
fin
```

par :

```
def f(a):
    if a:  # True and a
        pass
    if False: # False and a
        pass
    if True:  # True or a
        pass
    if a:  # False or a
        pass
```

Ce code sera ensuite nettoyé avec le système de warnings qui ne transpilera pas les condition systématiquement fausses. Il y aura donc pour finir le fichier PYTHON suivant :

```
def f(a):
    if a:  # True and a
        pass
    # if False supprimé par le système de warnings
    if True:  # True or a
        pass
    if a:  # False or a
        pass
```

Il y a cependant des expressions que l'on ne peut pas simplifier car le programme ne peut pas anticiper leur résultat. Avec le code suivant :

```
    fonction f(entier n) -> entier
    variables
        entier q
    debut
        q = n + 3
        retourner q
    fin
```

NATURL retournera :

```
    def f(n):
        q = n + 3
        return q
```

En effet, le traducteur code à code ne peut pas anticiper la valeur de n et donc la simplification de l'expression calculant q est impossible. Ainsi, la simplification des expressions complexes permet de clarifier le code transpilé. Ce processus permet d'avoir un code PYTHON plus propre. Il donne également la possibilité a l'utilisateur de mieux comprendre son code et de s'améliorer en évitant notamment les conditions jamais remplies ou les expressions logiques douteuses.

1.4 Les types itérables

Un type itérable est un type de donnée contenant des valeurs que l'on peut parcourir. Certains types itérables sont dit immuables, c'est à dire impossible à modifier directement. Par exemple, il est possible de concaténer deux chaînes de caractères pour en former un nouveau mais il est impossible de modifier une chaîne de caractères en soi.

1.4.1 Les listes

Les listes en naturL sont dites polymorphes mais elle ne peuvent contenir qu'un seul et même type, on parle alors de liste_de_entier ou de liste_de_booleens en fonction du type des éléments qu'elle contiennent. L'accès aux éléments d'une liste contrairement d'en Python est en base 1, ainsi le premier élément se situe à la position 1, ainsi de suite jusqu'au dernier élément qui se situe à la taille de la liste.

1.5 Structures de contrôle en naturL

naturL est muni comme les autres langages traditionnels des structures de contrôle si, sinon si et sinon.

Structure si

La structure conditionnelle "si" se forme de la manière suivante en naturL :

```
si CONDITION alors
    CODE
fin
```

Et se traduit en Python par :

```
if CONDITION:
    CODE
```

Les structures de contrôle conditionnelles sont susceptibles d'être simplifiées comme montré précédemment.

Par exemple :

```
si 1 = 1 alors
    afficher(faux)
fin
si faux alors
    afficher(vrai)
fin
```

Et se traduit en Python par :

```
if True:
    print(False)
```

Avec les avertissements suivants :

```
Avertissement à la ligne 1 : Cette expression est toujours vraie
Avertissement à la ligne 4 : Cette expression est toujours fausse
```

Structure sinon

La structure conditionnelle "si" se forme de la manière suivante en naturL :

```
si CONDITION alors
    CODE
sinon
    CODE
fin
```

Et se traduit en Python par :

```
if CONDITION:
    CODE
else:
    CODE
```

Structure sinon_si

La structure conditionnelle "si" se forme de la manière suivante en naturL :

```
si CONDITION alors
    CODE
sinon_si CONDITION2 alors
    CODE
sinon
    CODE
fin
```

Et se traduit en Python par :

```
if CONDITION:
    CODE
elif CONDITION:
    CODE
else:
    CODE
```

1.6 Boucles en naturL

1.6.1 Structure tant_que

naturL donne l'opportunité à l'utilisateur d'utiliser différents types de boucles. La boucle la plus générale est la boucle *tant_que* qui correspond à la boucle *while* dans les langages traditionnels. Cette boucle répète le code donné par l'utilisateur tant que la condition est vraie. Le fonctionnement est très intuitif, la syntaxe aussi :

```
tant_que CONDITION faire
    CODE
fin
```

Le transpileur traduire ce code de la manière suivante :

```
while CONDITION:
    CODE
```

Ces boucles sont très utiles car elles permettent d'automatiser et de factoriser le code. Cependant, une mauvaise gestion de l'actualisation de la condition peut vite donner une boucle infinie. Voici un exemple de code valide

```
variables
    entier n
fin
n <- 100
tant_que n > 0 faire
    afficher(n)
fin
```

Le transpileur traduira ce code de cette manière :

```
n = 100
while n > 0:
    print(n)
```

Ce code est totalement valide. Cependant, il mène à une boucle infinie. Il faut donc faire très attention à l'utilisation de cette structure.

1.6.2 Structure pour

La Structure *pour* est une variante de *tant_ que* plus sûre mais plus limitée. Elle permet d'avoir un nombre d'opérations fini. Elle est équivalente à la boucle *for* dans les langages de programmation classiques. La syntaxe est la suivante :

```
pour VARIABLE de DEBUT jusqu_a FIN faire
    CODE
fin
```

Il est important de noter que la valeur FIN est incluse dans la boucle. Ainsi le code naturL ci-dessous :

```
variables
    entier i
fin
pour i de 1 jusqu_a 100 faire
    afficher(i)
fin
```

Sera traduit par le code Python suivant :

```
for i in range(1, 101):
    print(i)
```

2. Les modules NATURL

Rappel En NATURL comme en Python, il est possible d'importer des fichiers. Pour cela, on utilise le mot-clé **utiliser**. On se donne l'arborescence suivante :

```
main.ntl
pack
    naturl-package
    bonjour.ntl
    aurevoir.ntl
```

Avec les fichiers `main.ntl`, `naturl-package` et `mod.ntl` définis comme suit :

`main.ntl` :

```
utiliser pack

pack.bonjour()
pack.au_revoir()
```

`pack/bonjour.ntl` :

```
procedure bonjour()
debut
    afficher("Bonjour")
fin
```

`pack/aurevoir.ntl` :

```
procedure au_revoir()
debut
    afficher("Au revoir")
fin
```

`naturl-package` :

```
bonjour
aurevoir
```

Comme on le voit dans le fichier `main.ntl`, il est possible d'importer des fonctions se trouvant dans des fichiers qui eux-mêmes se trouvent dans un dossier. Cela se fait à la condition que les fichiers en question soient précisés dans un fichier `naturl-package`. Cela permet, lorsque l'on écrit une bibliothèque, de préciser si un fichier `pack/mod.ntl` s'importe avec **utiliser** `pack.mod` ou **utiliser** `pack`. Notons qu'il serait également possible d'écrire

```
utiliser pack.*
```

Ce qui permet de ne pas avoir à préfixer les fonctions importées de « `pack.` » et d'alléger le code dans certains cas.

Lorsque l'on exécute la commande

```
naturL --input main.ntl --output main.py
```

On obtient l'arborescence suivante :

```
main.ntl
main.py
pack
    naturl-package
    __init__.py
    bonjour.ntl
    bonjour.py
    aurevoir.ntl
    aurevoir.py
```

On remarque que tous les fichiers ont été traduits en Python, y compris le fichier `naturl-package`, traduit en `__init__.py`. Rappelons qu'un fichier `__init__.py` permet de définir les modules importés dans un dossier, à l'instar du fichier `naturl-package`.

2.1 La bibliothèque standard

NATURL est muni d'une bibliothèque standard, accessible depuis n'importe quel chemin sous le nom de `std`. Ainsi, si l'on souhaite importer les fonctions mathématiques de la bibliothèque standard, il suffit d'écrire

```
utiliser std.maths
```

La position de cette bibliothèque standard dans le système de fichier est indiquée par la variable d'environnement `NATURLPATH`[1]. L'installation du dossier std `std` et la définition de la variable `NATURLPATH` sont pris en charge par l'installateur sur Windows. Sur Linux et macOS, il est possible de cloner le dépôt git et d'exécuter la commande

```
dune build @install
```

Des scripts vont alors gérer la mise en place de la bibliothèque standard et de la variable `NATURLPATH`.

2.1.1 La bibliothèque graphique

Un des modules intéressants de la bibliothèque standard est le module `pixL` qui permet de manipuler des interfaces graphiques en NATURL. Les fonctions ainsi écrites sont traduites avec le module `turtle` en Python. De nombreux autres module ont été envisagés, tels que pygame mais ce choix est justifié par le fait que `turtle` soit inclus dans la bibliothèque standard de Python. De cette manière, l'exécution d'un programme transpilé de NATURL utilisant `std.pixL` ne nécessite pas de configuration supplémentaire pour Python. Voici un exemple de programme NATURL utilisant `pixL` ainsi que sa traduction en Python :

1. Cette variable d'environnement a d'autres utilités, c'est par exemple elle qui est utilisée pour accéder aux fichiers d'internationalisation

```
utiliser std.pixL.*

procedure bonjour()
debut
    afficher("Bonjour")
fin

procedure partie()
debut
fin

definir_fond("blanc")
definir_couleur("rouge")
dessiner_cercle(Vecteur(0, 0), 150)
detecter_touche_pressee(bonjour, 'b')
lancer_partie(partie, 100)
```

Qui, transpilé en Python, donne :

```
from std.pixL import *

def bonjour():
    print("Bonjour")

def partie():
    pass

couleur_de_fond("blanc")
definir_couleur("rouge")
cercle_plein(Vecteur(0, 0), 150)
detecter_touche_pressee(bonjour, 'b')
lancer_partie(partie, 100)
```

Attardons-nous sur ce code. Tout d'abord, il affiche le drapeau du Japon (un rectangle blanc avec un cercle rouge au centre). Il affiche également `bonjour` dans la sortie standard lorsque l'utilisateur appuie sur la touche `b`.

Remarquons qu'il n'est nulle part fait mention de `turtle` dans le code Python. En effet, le module est importé dans les fichiers de `std.pixL` qui, à partir de fonctions basiques de `turtle`, définissent des fonctions avec des comportements plus complexes.

Prenons un peu de temps pour détailler certains morceaux de code :

— `couleur_de_fond("blanc")` : La fonction `couleur_de_fond` est assez basique et se contente de changer le fond de la fenêtre. Remarquons néanmoins que la couleur passée en paramètre est en français et est donc traduite en anglais lors de l'appel aux fonctions de `turtle`. Les couleurs au format hexadécimal sont également supportées.

— `definir_couleur("rouge")` : La fonction `definir_couleur` permet de spécifier la couleur de toutes les prochaines formes qui vont être affichées.

— `cercle_plein(Vecteur(0,0),150)` : La fonction `cercle_plein` prend en paramètre la position du centre du cercle ainsi que son rayon. Le type `Vecteur` est défini dans `pixL`. Il existe également une fonction `cercle` permettant d'afficher seulement la bordure du cercle et qui prend les mêmes paramètres.

— `detecter_touche_pressee(bonjour,'b')` : La fonction `detecter_touche_pressee` prend en paramètre une procédure et un caractère (ou une chaîne de caractère). Elle se contente d'appeler la procédure lorsque que le caractère est pressé par l'utilisateur.

— `lancer_partie(partie,100)` : La fonction `lancer_partie` prend en paramètre une procédure et un entier x. Toute les x millisecondes, la procédure est appelée, sans bloquer l'appel des évènements et les autres fonctions qui pourraient tourner en parallèle. Ainsi, il suffit de rajouter du code dans la procédure `partie` pour rajouter du code exécuté toutes les 0.1 secondes.

3. Orienté-objet en naturL

3.1 Initialiser un type abstrait

3.1.1 Initialiser les attributs du type abstrait

Pour initialiser un type abstrait, naturL utilise le système de classe disponible en Python. Ainsi, en naturL pour créer une classe, l'utilisateur doit utiliser le mot clé *type_ abstrait* suivi du nom qu'il souhaite donner au type abstrait.

Chaque attribut d'instance doit être déclaré après le mot clé *attributs*. Les attributs peuvent avoir une valeur par défaut si nécessaire dans ce cas la valeur sera attribuée en même temps que la déclaration de l'attribut et n'auront pas besoin d'être présent dans le constructeur. Cependant, les attributs n'ayant pas de valeur par défaut devront être initialisé dans le constructeur.

Le constructeur doit être placé après le mot clé *methodes*. Ce mot clé lance la section dans laquelle l'utilisateur devra entrer les méthodes de l'instance. Le constructeur doit cependant respecter quelques règles pour être reconnu comme tel par le transpileur. En effet, le constructeur doit être une fonction ayant pour nom *nouveau* et pour type de retour le nom de la classe qui sera ici initialisé comme un type. De plus, le constructeur doit utiliser le mot clé *instance* pour faire référence aux attributs de l'instance et devra retourner *instance*. Le constructeur sera traduit par la méthode python _ _ *init* _ _.

```
type_abstrait MyType
    attributs
        entier a
        reel b
        chaine c
    methodes
```

```
        fonction nouveau(entier a, reel b, chaine c) ->
           MyType
        debut
            instance a <- a
            instance b <- b
            instance c <- c
            retourner instance
        fin
    fin
fin
```

```
class MyType:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c
```

Comme on peut le remarquer, le comportement est assez intuitif. Dans l'exemple suivant, on utilise un attribut ayant une valeur par défaut.

```
type_abstrait MyType
    attributs
        entier a <- 42
        reel b
        chaine c
    methodes
        fonction nouveau(reel b, chaine c) -> MyType
        debut
            instance b <- b
            instance c <- c
            retourner instance
        fin
    fin
fin
```

De manière cohérente, le code est traduit ainsi :

```
class MyType:
    def __init__(self, b, c):
        self.a = 42
        self.b = b
        self.c = c
```

3.1.2 Fonctionnement du mot clé instance

De manière très intuitive, ce mot clé est une sorte de référence vers l'instance de la classe qui exécute la méthode. Elle joue entre autre le rôle du mot clé *self* en PYTHON. Ainsi, ce mot clé est utilisé dans une méthode lorsque celle-ci change un attribut de l'instance. Par conséquent ce mot clé ne fait sens que lorsqu'il est utilisé à l'intérieur d'un type abstrait. Pour illustrer cela prenons l'exemple de ce code :

```
type_abstrait Personnage
    attributs
        booleen en_vie <- vrai
        entier age
        reel argent
    methodes
        fonction nouveau(entier age, reel argent) ->
           Personnage
        debut
            instance age <- age
            instance argent <- argent
            retourner instance
        fin
        procedure anniversaire()
        debut
            instance age <- instance age + 1
        fin
    fin
fin
```

Ce code nécessite donc de modifier, dans la procédure *anniversaire*, l'âge de l'instance du type abstrait *personnage*. La procédure nécessite donc de manipuler les attributs propres a l'instance d'où la nécessité d'introduire un mot clé qui exprime cela. NATURL traduira donc le code ainsi :

```
    class Personnage:
    def __init__(self, age, argent):
        self.en_vie = True
        self.age = age
        self.argent = argent


    def anniversaire(self):
        self.age = self.age + 1
```

Il faut aussi préciser que le mot clé *instance* a un sens sémantique plus large en tant qu'il reflète l'entièreté de l'instance et non seulement les attributs. C'est pourquoi il est plus intuitif de préciser « retourner instance » à la fin du constructeur. Cette ligne rappelle également à l'utilisateur que le constructeur n'est pas qu'un outil magique sorti du merveilleux monde de la programmation, il s'agit avant tout d'une fonction, plus précisément d'une méthode, renvoyant l'instance de la classe que l'utilisateur veut initialiser.

3.1.3 Créations des méthodes

Les méthodes sont, en apparence, de simples fonctions. Elles appartiennent cependant à une instance d'une classe et peuvent en conséquence interagir avec les attributs de la classe à travers le mot clé *instance*.

Afin de déclarer des méthodes, l'utilisateur doit préciser le champ *methodes* après le champ *attributs*. La première méthode doit être le constructeur, cependant les méthodes suivantes sont spécifiques au type abstrait que l'utilisateur souhaite définir.

Une fois le mot clé entré et le constructeur défini, l'utilisateur peut déclarer et définir ses propres méthodes en respectant bien sûr les règles de définition et de déclaration des fonctions et des procédure en NATURL. Ainsi, le code suivant :

```
type_abstrait MyType
    attributs
        entier a
        reel b
        chaine c
    methodes
```

```
        fonction nouveau(entier a, reel b, chaine c) ->
           Mytype
        debut
            instance a <- a
            instance b <- b
            instance c <- c
            retourner instance
        fin
        procedure compter()
        debut
            tant_que instance a > 0 faire
                afficher(instance a)
                instance a <- instance a - 1
            fin
        fin
        fonction mult(reel x) -> reel
        debut
            retourner x*instance b
        fin
    fin
fin
```

Sera traduit par le transpileur de cette manière :

```
class Mytype:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def compter(self):
        while self.a > 0:
            print(self.a)
            self.a = self.a - 1

    def mult(self, x):
        return x * self.b
```

Il faut également bien prendre en compte le fait que les méthodes définies à travers NATURL ne peuvent pas être statiques.

3.2 Utiliser un type abstrait défini

Il est possible d'initier l'instance d'un type abstrait en donnant comme valeur la sortie d'une fonction portant le nom du type abstrait à une variable ayant comme type le nom du type abstrait. Pour être plus explicite, le code suivant initialise une instance du type abstrait défini ci-dessus.

```
variables
    Mytype mon_type
fin

mon_type <- Mytype(42,23.42, "Hello World")
```

On peut observer qu'une fois défini, un type abstrait agit comme un type classique de NATURL. Le nom de la classe correspond également au constructeur qui permet d'initialiser à proprement parler l'instance de la classe. Il y a donc une sorte de dualité concernant le nom du type défini.

Bien sur le constructeur est avant tout une fonction, par conséquent, il ne faut pas oublier de passer les arguments et de capturer la sortie dans une variable du bon type.

Vous pouvez aussi accéder aux attributs de l'instance du type abstrait en utilisant la notation « . » après le nom de la variable de l'instance. Cette écriture permet de manipuler simplement les différents attributs même en dehors des méthodes de l'instance et ce traduit très bien avec les mécanisme de la programmation orientée objets en PYTHON. Elle est également utilisée pour appeler les méthodes de l'instance. En effet, les fonctions sont en quelques sortes des variables, il est donc logique d'avoir ce type de comportement pour les appels de méthodes. Voici quelques exemples d'utilisation :

```
mon_type.compter()
mon_type.b <- mon_type.mult(3.1415)
mon_type.a
```

ce qui donne :

```
mon_type.compter()
mon_type.b = mon_type.mult(3.1415)
mon_type.a
```

Voici en guise de résumé l'exemple complet ainsi que sa version transpilée par NATURL.

```
type_abstrait Mytype
    attributs
        entier a
        reel b
        chaine c
    methodes
        fonction nouveau(entier a, reel b, chaine c) ->
           Mytype
        debut
            instance a <- a
            instance b <- b
            instance c <- c
            retourner instance
        fin
        procedure compter()
        debut
            tant_que instance a > 0 faire
                afficher(instance a)
                instance a <- instance a - 1
            fin
        fin
        fonction mult(reel x) -> reel
        debut
            retourner x*instance b
        fin
    fin
fin

variables
    Mytype mon_type
fin

mon_type <- Mytype(42,23.42, "Hello World")
mon_type.compter()
mon_type.b <- mon_type.mult(3.1415)
mon_type.a
```

Ce code nous donne le code en PYTHON suivant :

```
class Mytype:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c



    def compter(self):
        while self.a > 0:
            print(self.a)
            self.a = self.a - 1


    def mult(self, x):
        return x * self.b



mon_type = Mytype(42, 23.42, "Hello World")
mon_type.compter()
mon_type.b = mon_type.mult(3.1415)
mon_type.a
```

4. idL

4.1 Introduction

idL est un IDE qui à été pensé et écrit pour transpiler, exécuter et écrire en NATURL. Cet IDE présente des fonctionnalités qui permettront à l'utilisateur de coder de manière ergonomique. Ainsi, il pourra comprendre ses erreurs en temps réel puisqu'elles seront affichées pendant son édition. idL utilise des fonctionnalités qui sont dépendantes du protocole LSP (language server protocol) qui permet de mettre a disposition de l'utilisateur des fonctionnalités assez avancées comme le soulignage des erreurs directement dans le code. Nous parlerons de ces fonctionnalités dans la partie "Fonctionalités d'éditeur".

4.2 Fonctionnalités d'édition

L'IDE est muni de nombreuses fonctionnalités qui peuvent être qualifiées de fonctionnalités d'édition, c'est à dire celles qui ne sont pas liées au code de l'utilisateur. Nous allons les énumérer et les présenter ci-après.

4.2.1 Ouverture

Lorsque l'utilisateur ouvre idL, il arrive sur une page d'accueil. Sur cette dernière il pourra voir trois boutons. Le premier bouton lui permet d'ouvrir les fichiers récemment ouverts. Les fichiers récemment ouverts. A noter que les fichiers récemment ouverts sont les fichiers qui ont étés ouverts par l'utilisateur lors d'une précédente utilisation de idL. Ces fichiers seront alors tous les fichiers qui sont sauvegardés sur l'ordinateur de l'utilisateur et qui étaient ouverts lors de la fermeture d'idL. Il est aussi possible d'ouvrir un fichier déjà existant à l'aide du deuxième bouton (Note : il n'est pas possible de faire l'inverse

et d'ouvrir idL à partir d'un fichier existant. Il est impératif de passer par l'application idL en elle même). Si l'utilisateur choisi d'ouvrir un fichier depuis idL ou bien d'en créer un mais annule cette procédure, il pourra tout de même le faire depuis idL. Cependant, si l'utilisateur a annulé sa procédure d'ouverture ou de création certaines fonctionnalités de idL ne seront pas présente (affichage des erreurs en temps réel) ce bug est mineur et ne nuit pas à l'utilisation de idL et sera corrigé dans les versions suivantes. En conclusion, la meilleure manière de profiter de idL actuellement est d'utiliser un fichier sauvegardé.

4.2.2 Fenêtre principale

La page principale est décomposée en trois parties principales. La zone à gauche est la zone de texte dans laquelle l'utilisateur peut écrire son code en naturL. A droite il verra son code python apparaître lorsqu'il transpilera son code naturL. En bas se situe la console. Celle-ci est utile à l'utilisateur puisqu'elle donne des informations sur le code de l'utilisateur. Si jamais le code contient une erreur et que l'utilisateur veut transpiler son code, une erreur en rouge s'affichera dans la console. Sinon seul le message "Transpilation réussie" apparaîtra s'il peut être traduit. Les avertissements ne seront pas indiqués pour la transpilation. En revanche si l'utilisateur choisi d'exécuter son code et qu'il présente des avertissements ; les avertissements et le résultat de l'exécution s'afficheront. A noter que partout dans idL les avertissements s'affichent en orange et les erreurs en rouge.

Le bouton de transpilation est le bouton bleu et jaune et le bouton d'exécution est le bouton orange. Le bouton carré rouge permet d'annuler un processus en cours (cette fonctionnalité a été implémenté dans le cas ou l'utilisateur créé une boucle infinie et qu'il souhaite l'arrêter ou bien s'il souhaite simplement annuler un processus pour X raisons).

La bande orange est la bande sur laquelle sont écris les noms de fichiers ouverts, cela est assez pratique puisqu'il il est possible d'ouvrir plusieurs onglets à la fois. Ouvrir un onglet ou en fermer se fait simplement en appuyant sur les boutons prévus à cet effet. Il est possible d'ouvrir un nouvel onglet sans nouveau fichier, ainsi idL présente une option bac à sable. Laissez moi clarifier cela. Cette fonctionnalité permet à l'utilisateur d'éditer, transpiler, exécuter un fichier non sauvegardé. Cependant il faut faire attention. Microsoft limite le nombre de caractères passés dans l'entrée standard. Ainsi si le fichier est trop gros la transpilation ne se fera pas et aucun message d'erreur ne s'affiche.

Dans ce cas éventuel, vous devez sauvegarder le fichier. Cela fonctionne de la même manière pour les fichiers sauvegardés. A la seule différence près que nous vous sauvegardons ce gros fichier automatiquement. De plus lorsque le fichier est trop gros, un fichier python est créé car il n'est pas possible de récupérer la sortie standard trop volumineuse ainsi le fichier python est lu directement pour éviter tout problème.

L'utilisateur peut donc ouvrir, créer, sauvegarder, sauvegarder sous le texte qu'il édite. Ces fonctions ouvrent une boîte de dialogue qui donnent à l'utilisateur la possibilité de faire ce qu'il a demandé, ces fonctionnalités sont basiques et évidentes nous ne nous étendrons pas dessus.

Il est possible de modifier la langue d'idL ainsi que des messages d'erreurs de la console. Cependant, le protocole LSP ne permet pas de modifier la langue du serveur en cours d'utilisation de idL. Cela signifie simplement que les messages d'erreurs que l'utilisateur verra apparaître dans la zone de texte pour le code naturL ne seront pas traduits. Ceux dans la console le seront quoi qu'il arrive. Pour changer la langue des messages d'erreurs en direct il faut, changer la langue puis relancer idL.

L'utilisateur peut aussi rechercher du texte dans la zone de texte naturL. Cette rechercher se fait a partie de la position du curseur dans la zone de texte. Cela signifie qu'avant de cliquer dans la zone de texte pour la recherche au milieu de la fenetre de idL, l'utilisateur doit au préalable placer son curseur quelque part dans le texte. La recherche se fait ainsi a partir de la position du curseur et il y a donc une recherche de toutes les occurrences du mot (en fait de la chaîne ; cette dernière peut être quelconque) jusqu'à la fin du texte. A chaque occurrence trouvée le mot est selectionné et si le fichier est long et que le mot recherché trouvé ne se situe pas dans la zone visibme par l'utilisateur alors un défilement vertical automatique est activé pour le rendre visible pour l'utilisateur.

L'utilisateur peut aussi agrandir son texte en utilsant CTRL + molette. Il peut aussi réinitialiser l'agrandissement avec le raccourci CTRL + 0 (zéro). Tous les raccourcis claviers sont présents dans le tableau à la fin du docuement.

Une fenêtre de paramètres est disponible, celle-ci permet de modifier la couleur de la coloration synthaxique. Cette dernière est sauvegardée de telle manière a ce que l'utilisateur récupère ses derniers réglages au lancement de idL, il est aussi possible de revenir aux couleurs d'origines (à termes il sera possible de donner a l'utilisateur de sauvegarder plusieurs fichiers de coloration synthaxique). La langue est sauvegardée de la même manière que les couleurs. Au lancement de idL les réglages précédemment sauvegardés

sont appliqués.

L'utilisateur peut aussi évidemment fermer idL. Lorsque l'utilisateur ferme idL, tous les fichiers qui ont déjà étés sauvegardés quelque part dans l'ordinateur sont sauvegardés et fermés automatiquement. Ceux-ci seront réouverts a l'ouverture de idL. Ensuite si l'utilisateur a des bacs à sable d'ouverts et qu'il a écrit des choses à l'intérieur, idL proposera à l'utilisateur de les sauvegarder s'il le souhaite.

4.2.3 Fonctionnalités disponibles grâce au protocole LSP

Le protocole LSP a permi l'implémentation de superbes fonctionnalités à idL afin de le rendre plus pratique à utiliser. Parmis ces fonctionnalités on compte :

- Le "jump to definition" : La fonctionnalité "jump to definition" permet à un utilisateur en cliquant sur un usage ou un appel d'une fonction particulière de remonter à sa déclaration dans le code, même si cette fonction se situe dans un autre fichier. C'est utile par exemple pour connaître les valeurs à passer dans un constructeur d'un type_abstrait.
- Autocompletion "intelligente" : l'autocompletion "intelligente" permet maintenant d'ajouter au mots de l'autocomplétion les différentes fonctions et ou variables disponibles dans le contexte actuel.
- Signature des fonctions : avec l'autocomplétion intelligente viens la signature des fonctions ainsi on peut consulter le type des paramètres et le type d'une retour d'une fonction dans la fenêtre d'auto-complétion.
- Reformat : l'outil de reformatage permet d'indenter et de mettre en forme le code de manière correcte et ce, sans qu'il soit nécessairement transpilable.
- Message d'erreurs et avertissement : le protocole LSP permet l'analyse syntaxique et sémantique en temps réel du code, permettant ainsi de souligner les erreurs de syntaxe, ou encore les avertissement de bonne pratique.

5. Le Site Web

5.1 Les informations essentielles

Le site est disponible à l'adresse suivante : . Il a pour but de renseigner sur l'identité des membres du groupe, la chronologie du projet ainsi que de fournir des liens de téléchargement et d'installation pour les programmes du projet. Le site présente également l'histoire de la création et explique les origines du projet.

5.2 Le plan du site

Le site web est organisé en plusieurs pages. Chaque page est divisée en deux parties. La partie de droite contient des liens utiles. Il s'agit des liens permettant à l'utilisateur de : se rendre à la page d'accueil, d'accéder à la page *github* du projet, d'accéder au fichier PDF contenant le rapport de soutenance, d'accéder au fichier PDF contenant la documentation et d'accéder à la page contenant les liens de téléchargement. La partie droite contient la page en question et permet donc d'explorer le site.

5.2.1 La page d'accueil

La page d'accueil contient l'histoire de la fondation du groupe. À cet endroit, un lien vers le site de l'école et vers une page contenant une frise chronologique est disponible. Cette section explique donc d'où vient l'idée du projet naturL/idL ainsi que les premiers pas du groupe. Il y a également une brève section décrivant en une phrase naturL et idL et donne les liens d'accès au git respectifs. La dernière section contient l'image de chaque membre ainsi que des liens menant aux pages de présentations de chaque membre.

5.2.2 Les pages de présentation

Il existe une page de présentation par membre. Ces pages sont également divisée en deux. A droite, on retrouve a peu près les mêmes éléments que sur les autres pages du site à l'exception du logo naturL qui est remplacé par la photo du membre du groupe. Il y a également un lien vers le github du membre en question en plus.

Sur la partie droite, chaque membre a écrit un petit texte de présentation et a décoré la page d'une citation personnelle.

5.2.3 La page de Chronologie

Cette page est assez simple. Elle contient une frise chronologique de l'évolution du projet. Chaque zone de texte énumère les changements apporté par la mise à jour. Il y a également des images de l'état d'idL qui illustre bien l'avancée du projet ainsi que les nouveautés de naturL. Cette page n'est certes pas très explicative mais elle a le mérite d'être assez explicite, claire et précise.

5.2.4 Page de téléchargement

Sur cette page l'utilisateur peut choisir une manière d'installer le projet en fonction de son système d'exploitation. Pour Linux, la page renvoie vers le git de naturL où le fichier *REAMDE* explique les étapes de l'installation. Pour MacOS, l'utilisateur peut télécharger directement un fichier écrit en bash et l'exécuter depuis le terminal. Dans les deux cas, l'installation configure naturL et crée une commande *naturL* dans la console. Pour Windows, un installateur est directement disponible. Windows est le seul système d'exploitation qui permet la compilation d'idL. Par conséquent, idL est seulement disponible pour cette plateforme.

6. Outils et technologies utilisées

6.1 Github pages

Le site internet est disponible via la technologie *Github pages*. Cet outil permet de maintenir le site web disponible sur internet. De plus,cette méthode est très pratique car le code source du site internet est contenu dans un dépôt Github ce qui permet au développement du site de bénéficier des mêmes avantages que n'importe quel projet `git`.

Github pages utilise Jekyll pour construire les site web que le service héberge. Jekyll est un outil de génération de site internet statique. Il offre quelques fonctionnalités pratique que nous avons utilisé pour insérer les langages du web dans le site.

Github pages permet donc à l'utilisateur de créer entièrement un site internet et supporte même les langages de programmation html, JavaScript et CSS. De cette manière, le site internet du projet peut être rendu dynamique et relativement beau et agréable pour l'utilisateur. Ainsi, cette technologie permet au site internet du projet d'être hébergé par github ce qui le rend accessible et visible sur internet.

6.2 Github et Git

The NaturL Foundation sur Github c'est plus de 400 commits et 20 000 lignes de codes écrites entre les différents éléments du programme naturL. C'est aussi 20 bug fixés et plus de 6 branches de développement.

7. Conclusions

7.1 Rostan

Ce projet est le plus gros projet sur lequel Rostan ait eu l'occasion de travailler, et aussi le plus intéressants.

L'un des points les plus passionnants a sans doute été la création de l'algorithme du parseur dédié à NATURL. Cela a également permis à Rostan de se découvrir un réel intérêt pour le langage OCaml et de manière plus générale le paradigme de programmation fonctionnelle. Cela a également été l'occasion de se documenter sur des notions de programmation et de mathématiques fascinantes.

Un autre enseignement important de ce projet concerne le travail en groupe. Étant donné que Rostan a eu la tâche d'implémenter le protocole LSP côté client, il a dû travaillé de manière conjointe avec Adrian et corriger avec lui des bugs causés par les deux projets à la fois : NATURL et IDL. L'implémentation du protocole LSP a de plus nécessité beaucoup de rigueur étant donné que les requêtes et les réponses devaient se conformer strictement aux spécifications officielles du protocole [1].

Il faut ajouter à cela le fait que ce projet a permis à Rostan une meilleure connaissance du fonctionnement du langage Python. Il a en effet fallu une bonne compréhension du système d'imports de Python pour gérer tous les cas (imports relatifs / absolus, fichier `__init__.py`, etc.).

Dans l'ensemble, ce projet a permis à Rostan d'appronfondir ces connaissances dans de nombreux domaines et de développer un intérêt pour la programmation fonctionnelle.

1. Dans l'objectif de pouvoir utiliser le serveur avec tout client LSP et pas seulement IDL. Grâce à cela, le serveur LSP de NATURL fonctionne très bien avec Sublime Text

7.2 Simon

Ce premier est le premier projet conséquent en équipe pour Simon. En plus de sa responsabilité de chef de projet tiré à la courte paille, il a du mettre en place les sytèmes de compilation autonome et de répartition du code entre les utilisateurs.

7.2.1 Le travail en groupe

La dynamique de groupe durant le projet était bonne et globalement, tout le monde motiveait tout le monde. Personne n'as vraiment tout fait, personne n'as rien fait non plus, le rythme était bon bien que moins soutenu sur la fin mais vers la fin de mois d'Avril il était devenu presque naturel pour nous d'échanger entre pull requests, branches, messages de commits et issues via Github... Inimaginable il y a 3 mois....

Un autre aspect du travail de groupe qui a, selon Simon, été bien développé c'est l'entraide et le partage de l'expérience. Ainsi certains membres du groupe étaient plus expérimentés que d'autres dans certains langages et ont partagé leur expériences avec les moins familiers.

7.2.2 La projet en lui même

Simon étant passionné de recherche, voulant lui même être enseignant c'est surtout l'aspect pédagogique du langage de programmation développé qui l'a intéressé. Une sorte de vulgarisation algorithmique que l'on retrouve dans le document en annexe (malheureusement pas totalement terminé pour cette soutenance) : la manuel pédagogique, tutoriel complet sur le langage naturL et les débuts de la programmation. Enfin, Simon aimerais continuer à développer le projet après la dernière soutenance, produire une version encore plus aboutie avec une généricité de types, améliorer le vérificateur de types et les messages d'erreurs.

7.2.3 Ce qu'on retiendra du code de Simon dans naturL

— Le tokenizer et le reformatage pour ce qui touche à la grammaire et à l'analyse syntaxe
— La gestion des messages d'erreurs et de leur traduction à l'aide d'un interfaçage d'un fichier JSON

— La structure d'évaluation des structures de contrôle.
— La base de la librairie standard.
— Dans l'idL : la gestion des paramètres utilisateurs et l'utilisation des DataContract dans la sauvegarde des données personnalisés (nottament la coloration syntaxique).

7.3 Adrian

Pour Adrian cette expérience est nouvelle. Il a pu créer une application WPF pour la première fois et a appris beaucoup de choses grâce à ce projet. Ce projet lui a permit d'apprendre beaucoup de choses sur la programmation et la patience.

7.3.1 La projet lui même

Travailler avec le framework WPF de windows peut parfois apporter de la joie lorsque nous sommes stupéfaits par son efficacité mais parfois de très grandes frustrations puisque ce dernier est tout de même assez vieux. En effet ce que l'on pense être facile à implémenter en WPF est en fait parfois le plus difficile, en effet, certaines fonctionnalités que l'on penserait présentes ne le sont pas et ainsi doivent être ré implémentées. Par exemple pour les onglets. Comme je l'ai expliqué dans le rapport précédent un onglet est une ressource statique en WPF ainsi il n'y a pas de moyen prévu pour en rajouter par l'utilisateur. Adrian a donc tout ré implementé de zéro avec de l'aide de ses camarades. Cela lui à donc donné envie de créer sa propre librairie pour gérer les onglets dans une fenêtre. Ainsi il a été difficile de rendre une interface la plus moderne possible avec un framework qui a été créé il y a plus d'une dizaine d'années. Ainsi tout a été personnalisé, la fenêtre est uniquement de notre création, nous n'aimions pas les boutons présents pour fermer la fenêtre ? Adrian les a remplacés

7.3.2 Ce qu'on retiendra du code de Adrian dans naturL

Adrian avait dès le début du projet envie de créer une interface qui se rapprocherait le plus possible d'un IDE. Nous pouvons aujourd'hui dire avec notre groupe que l'interface grâce à la récente implémentation du protocole LSP est un IDE de développement pour le language naturL. Adrian a donc pu s'occuper de la majeure partie de idL avec de l'aide de ses camarades pour l'implémentation de certaines fonctionnalités qui ne sont pas forcément faciles à mettre en place lorsque l'on est débutant en programmation. Ainsi grâce a l'expérience de ses collègues et amis de projet Adrian a pu terminer un IDE assez pratique et ergonomique pour de programmer en naturL et en est très fier.

7.4 Vlad

Ce projet est le premier gros projet de Vlad. Jamais auparavant il n'avait travaillé sur la même chose pendant plusieurs mois. Les capacités développés au cours du développement sont très précieuse pour initier au domaine de la programmation.

NATURL est un projet très instructif car il nécessite de comprendre plusieurs langages de programmations. Le premier est bien évidemment OCAML qui gère le transpileur. Ensuite, il faut une maîtrise de PYTHON qui permet d'effectuer une traduction sans erreurs et un code optimal respectant les normes syntaxiques de PYTHON. S'ajoute à cela la nécessité de formaliser la syntaxe de NATURL et l'utilisation de C# pour l 'interface graphique. Cette pluralité des langages de programmation utilisées permet de développer de solides connaissances dans ces langages en plus d'ouvrir Vlad à de nouvelles manières de résoudre les problèmes.

Cette ouverture est notamment due à la manipulation d'un langage fonctionnel comme OCAML et de langages impératifs comme PYTHON et même NATURL. De plus, le travail en équipe nécessite l'utilisation de différents outils de collaboration tels que git et LaTeX. Une fois de plus, le projet a permis à Vlad de s'exercer sur ses différents outils et d'acquérir des connaissances fondamentales dans les outils les plus utilisés dans l'industrie.

Qui plus est, un tel projet permet aussi de s'initier à la dynamique du travail en groupe. Cette mécanique est essentielle mais pose souvent quelques difficultés dont la principale difficulté rencontrée lors du travail qui est l'utilisation du même fichier, des mêmes fonctions, par plusieurs développeurs. En effet, sur un fichier fondamental de la conception du projet et devait être au point en premier car ce dernier permet à la fois la traduction des mots-clés du langage NATURL en PYTHON et la gestion de certaines erreurs syntaxiques. Ainsi, Rostan, Simon et Vlad ont travaillé en même temps sur ce fichier essentiel au bon fonctionnement du transpileur. Mais le travail d'équipe implique aussi l'utilisation de fonctions et de structures faites par d'autres membres. Ainsi, une autre difficulté a très clairement été le fait de comprendre le code des autres. En effet, il est assez chronophage de lire et comprendre du code écrit par autrui avant de pouvoir résoudre les bugs et de pouvoir implémenter de nouvelles fonctionnalités pour le transpileur. De plus,

un nouveau défi apparaît lors de la compréhension globale du fonctionnement du programme. En effet, les fonctions ont été écrites par différents membres du groupe. Par conséquent, la communication au sein de l'équipe est un défi mineur mais qu'il faut relever pour gagner de précieuses heures. De plus, la résolution de bugs est plus complexe notamment à cause de l'augmentation de la quantité de code avec l'avancement du projet. De ce fait, si un bug est causé par une des fonctions écrites à l'origine de la phase de développement, sa résolution est très délicate car elle peut causer l'apparition d'autres bugs.

Par conséquent, la modifications d'une fonction auxiliaire pour résoudre un bug dans une fonction bien précise pouvait en entraîner un dans une autre fonction écrite par un autre membre. Ceci donne lieu à la perte de nombreuses heures de programmation mais permet également l'émergence de nouvelles approches pour surmonter les défis posés par ce projet. S'ajoute à cela la difficulté liée à la phase de création initiale du projet. En effet, beaucoup de changements concernant les structures de données utilisées ont eu lieu. Ces modifications, bien que nécessaires, ont causé beaucoup de bugs et d'erreurs ce qui a coûté d'innombrables heures aux développeurs du groupes.

Mais le travail de groupe présente aussi de nombreux avantages et notamment la possibilité de résoudre plusieurs problèmes assez rapidement en divisant correctement les tâches. Il y a aussi la facilité d'apprendre de nouveaux concepts qui vient avec le travail d'équipe. En effet, les niveaux, notamment en OCamL, dans le groupe étant très hétérogènes, ce projet permet facilement au moins bons d'apprendre des meilleurs. Et c 'est principalement pour cette raison que Vlad a beaucoup progressé dans la programmation en OCamL.

De plus, la résolution de bugs et de problèmes peut parfois être facilité par le travail d'équipe. En effet, la collaboration permet d'avoir plusieurs approches et donne l'opportunité au groupe de trouver la meilleure solution. De ce fait, il est rarement question de développeurs qui passent des jours a essayer de résoudre un bug qu'ils ne comprennent absolument pas car généralement, à plusieurs la compréhension du groupe est plus rapide.

L'ajout de nouvelles fonctionnalités est aussi facilité car tous les membres ont une vue d'ensemble du projet et sont ainsi inspirés par les idées des autres. Le travail d'équiper à permis à Vlad de créer plusieurs fonctionnalités améliorant naturL grâce aux interactions du groupes. Parmi elles il y a la simplification d'expressions, qui permet de rendre un code simple et plus

lisible, et l'implémentions de la création de types abstraits qui permet au transpileur de pleinement exploiter le potentiel de PYTHON.

8. Annexes

Ci joint en annexe le manuel pédagogique de naturL en cours de rédaction par Simon Scatton.

8.1 Introduction

La langage naturL a été produit par quatre étudiants en première année à l'École Pour l'Informatique et les Techniques Avancées en 2020 en tant que projet de second semestre. Sa création s'inscrit dans un objectif pédagogique double : pour ses créateurs et ses utilisateurs. L'objectif principal de naturL est de faciliter l'apprentissage de l'algorithmique en fournissant un langage qui se rapproche le plus du pseudo-code français. naturL est destiné aux élèves de début d'enseignement supérieur, du lycée et du collège.

8.1.1 Qu'est ce que la programmation

8.1.2 Penser comme un programmeur

Avant de débuter ce manuel pédagogique j'aimerais revenir sur la définition d'un programmeur et qu'est ce que programmer. La programmation est d'abord un mode de penser qui s'acquiert avec l'exercice : l'activité essentielle d'un programmeur est de résoudre des problèmes, et ce, à l'aide d'outils : la machine, le langage. Cette activité est complexe et nécessite une approche de réflexion importante. En effet, écrire du code ne suffit souvent pas à faire de vous un programmeur, ce qui importe c'est de réfléchir son code et répondre au problème de la manière qui nous semble la plus appropriée.

8.1.3 Les outils

Les langages de programmation sont des outils pour faire le pont entre la machine et l'Homme. En effet, il serait difficile pour un humain d'écrire dans le langage que la machine comprends : le code machine ou langage machine. Ce code machine est une suite de 0 et de 1 qui sont interprétés par le processeur. Cependant, il nous est impossible de lire et de comprendre le code machine de manière intuitive, c'est donc pour ça que nous avons ajouté ce qu'on appelle des "couches d'abstractions" : des étapes intermédiaires dans la création d'un code machine afin de nous permettre de réaliser ce dernier plus facilement. Il existe plusieurs couches d'abstraction entre le code machine et le langage de programmation qu'on ne va pas détailler ici mais il est important de comprendre que le code que l'on écrit dans un langage de programmation ne ressemble pas du tout à la finalité du programme. Entre les deux se passe un processus de appelé compilation décrit ci dessous.

Un compilateur est un programme qui transforme un code source en un code objet. Généralement, le code source est écrit dans un langage de programmation (le langage source), il est dit "de haut niveau d'abstraction", et est facilement compréhensible par l'humain. Le code objet est généralement écrit en langage de plus bas niveau (appelé langage cible), par exemple un langage d'assemblage ou langage machine, afin de créer un programme exécutable par une machine. Pour créer le langage cible ce compilateur doit analyser et traduire le code source et traduire entièrement l'intégralité du code écrit dans le langage cible. Un interpréteur se distingue d'un compilateur par le fait que, pour exécuter un programme, les opérations d'analyse et de traductions sont réalisées à chaque exécution du programme (par un interprète) plutôt qu'une fois pour toutes (par un compilateur). L'interprétation repose sur l'exécution dynamique du programme par un autre programme (l'interprète), plutôt que sur sa conversion en un autre langage (par exemple le langage machine) ; elle évite la séparation du temps de conversion et du temps d'exécution, qui sont simultanés.On différencie un programme dit script, d'un programme dit compilé :

— un programme script est exécuté à partir du fichier source via un interpréteur de script
— un programme compilé est exécuté à partir d'un bloc en langage machine issu de la traduction du fichier source.

Python est un langage interprété, ainsi on parle le plus souvent de "script python" (même si dans certains cas il est possible de compiler du Python).

Cet ouvrage se veut explicatif mais vous retrouverez des explications sur ces processus dans des ouvrages plus généralistes sur la compilation. Beaucoup d'aspects sont ici expressément laissés de côté comme la notion de code assembleur mentionnée mais que l'on ne définira pas par soucis de simplicité.

Maintenant que vous savez grossièrement ce qu'est un langage de programmation, parlons du code en lui même

8.1.4 Programmer : la syntaxe, la sémantique, les erreurs

Durant l'exercice de la programmation vous serez amené à écrire dans différents langages, ces langages comme le français et l'anglais et possèdent chacun une *syntaxe* particulière. La syntaxe est l'ensemble des mots et de leur assemblage qui forme un code source. Ce code source peut être valide syntaxiquement ou contenir des erreurs (appelées erreur de syntaxe). Les erreurs de syntaxe seront la première difficulté à affronter en tant que débutant dans un langage particulier. Par exemple, en français, une phrase doit toujours commencer par une majuscule et se terminer par un point. Un manquement à cette règle de base constitue une erreur de syntaxe. Gardez à l'esprit que les mots et les symboles utilisés n'ont aucune signification en eux-mêmes : ce ne sont que des suites de codes destinés à être convertis automatiquement en nombres binaires. Par conséquent, il vous faudra être très attentifs à respecter scrupuleusement la syntaxe du langage. Finalement, souvenez-vous que tous les détails ont de l'importance. Il faudra en particulier faire très attention à la casse (c'est-à-dire l'emploi des majuscules et des minuscules) et à la ponctuation. Toute erreur à ce niveau (même minime en apparence, tel l'oubli d'une virgule, par exemple) peut modifier considérablement la signification du code, et donc le déroulement du programme. Afin de déterminer la source d'une erreur de syntaxe il sera utile de lire les "messages d'erreurs" ! Ces messages qui vont sont adressés par le compilateur ou l'interpréteur sont important et vous permettrons à vous habituer à repérer rapidement vos erreurs. Le second type d'erreur est l'erreur sémantique ou erreur de logique. S'il existe une erreur de ce type dans un de vos programmes, celui-ci s'exécute parfaitement, en ce sens que vous n'obtenez aucun message d'erreur, mais le résultat n'est pas celui que vous attendiez : vous obtenez autre chose. En réalité, le programme fait exactement ce que vous lui avez dit de faire. Le problème est que ce que vous lui avez dit de faire ne correspond pas à ce que

vous vouliez qu'il fasse. La séquence d'instructions de votre programme ne correspond pas à l'objectif poursuivi. La sémantique (la logique) est incorrecte. Rechercher des fautes de logique peut être une tâche ardue. Il vous faudra analyser patiemment ce qui sort de la machine et tâcher de vous représenter une par une les opérations qu'elle a effectuées, à la suite de chaque instruction.

Le troisième type d'erreur est l'erreur en cours d'exécution (Run-time error), qui apparaît seulement lorsque votre programme fonctionne déjà, mais que des circonstances particulières se présentent (par exemple, votre programme essaie de lire un fichier qui n'existe plus). Ces erreurs sont également appelées des exceptions, parce qu'elles indiquent en général que quelque chose d'exceptionnel (et de malencontreux) s'est produit. Vous rencontrerez ce type d'erreurs lorsque vous programmerez des projets de plus en plus volumineux, et vous apprendrez plus loin dans ce manuel qu'il existe des techniques particulières pour les gérer.

8.1.5 Fonctionnement

naturL n'est pas un langage de programmation comme les autres : il est dit "transpilé". Un langage transpilé est un langage de programmation qui, pour s'exécuter, va être traduit dans un autre langage de programmation ici : Python. naturL est donc avant-tout un langage voué à être traduit dans un langage de programmation "traditionnel". C'est d'ailleurs là tout son intérêt, il permet à son utilisateur de lier directement le code qu'il écrit et son implémentation dans un langage traditionnel. Dans les parties qui suivent nous allons essayer de couvrir pas à pas toutes les fonctionnalités que comprends naturL et d'expliquer leur fonctionnement.

naturL comme dit précédemment s'inscrit dans une démarche pédagogique d'apprentissage de l'algorithmique et donc la syntaxe de naturL est simplifiée et ressemble à ce qu'on appelle le pseudo code. Le pseudo code est une manière de décrire un programme de manière abstraite sans implémentation particulière dans un langage de programmation, c'est un outil descriptif afin de mieux cerner la complexité d'implémentation d'un algorithme en particulier. L'écriture en pseudo-code permet souvent de bien prendre toute la mesure de la difficulté de la mise en œuvre de l'algorithme, et de développer une démarche structurée dans la construction de celui-ci. En effet, son aspect descriptif permet de décrire avec plus ou moins de détail l'algorithme, permettant de ce fait de commencer par une vision très large et de passer outre temporairement certains aspects complexes, ce que n'offre pas la programmation directe.

Cependant le pseudo code naturL est beaucoup plus limité parce qu'il possède un nombre limité d'instructions et est soumise à une syntaxe et formalisation particulière. Dans la conception de ce langage nous avons donc du faire des choix car derrière ce code naturL il y a une contrainte matérielle qu'est l'implémentation de ce langage et sa traduction dans un langage conventionnel comme le Python. On voit donc bien son intérêt qui réside dans la comparaison entre l'implémentation et la pensée du code. On peut penser naturL comme un outil de réflexion algorithmique mais pas probablement pas comme un langage utilisable dans l'industrie ou la recherche (bien qu'il soit Turing-Complet [1]).

8.2 Les bases du langage naturL

8.2.1 Utilisation

Un fichier naturL possède l'extension *.ntl* il est éditable comme une fichier texte sans condition particulières. Les fichiers *.ntl* sont passés au programme naturL [2]. Le programme naturL va ressortir la traduction Python de ce code si la syntaxe et la sémantique sont valides, sinon il retourne un message d'erreur indiquant la position de l'erreur et sa nature.

Dans la section suivante nous allons définir les bases de la syntaxe naturL.

8.2.2 Mot clés réservés

Le langage possède une quantité non négligeable de mots clé définissant les instructions, les types etc. Nous y reviendrons plus tard mais en voici une liste exaustive :

- La définition de fonction : fonction / procedure
- La définition de variables : variables
- Les boucles : tant_que / pour / pour_chaque /
- de / allant / jusqu_a
- Les accès conditionnels : si / sinon / sinon_si / alors
- Les fermetures et ouvertures de structures : fin / debut
- retourner
- Les opérateurs logiques : et / ou / non / vrai / faux

1. Un langage est dit Turing-Complet s'il peut implémenter n'importe quelle machine de Turing
2. Voir la documentation : Utilisation de naturL

— Les opérateurs mathématiques : div
— Les types : entier, reel, chaine, charactere, liste, booleen
— Les listes typées : liste_de_(reel/entier/booleen/chaines)

8.3 Données et variables

8.3.1 Qu'est ce qu'une variable ?

Pour comprendre ce qu'est une variable et comment manipuler celles-ci, il faut commencer par comprendre comment notre ordinateur fait pour stocker des données. En théorie, un ordinateur est capable de stocker tout type d'information. Mais comment est-il possible de réaliser un tel miracle alors qu'il ne s'agit finalement que d'un amas de circuits électriques ? Peut-être avez-vous déjà entendu le proverbe suivant : « si le seul outil que vous avez est un marteau, vous verrez tout problème comme un clou » (Abraham Maslow). Hé bien, l'idée est un peu la même pour un ordinateur : ce dernier ne sachant utiliser que des nombres, il voit toute information comme une suite de nombres.

L'astuce consiste donc à transformer une information en nombre pour que l'ordinateur puisse la traiter, autrement dit la numériser. Différentes techniques sont possibles pour atteindre cet objectif, une des plus simples étant une table de correspondance. Cependant un ordinateur compte en base 2 et il est donc nécessaire d'adapter notre système numérique à ce format. La base correspond au nombre de chiffres disponibles pour représenter un nombre. En base 10, nous disposons de dix chiffres : zéro, un, deux, trois, quatre, cinq, six, sept, huit et neuf. En base deux, nous en avons donc... deux : zéro et un. Pour ce qui est de compter, c'est du pareil au même : nous commençons par épuiser les unités : 0, 1 ; puis nous passons aux dizaines : 10, 11 ; puis aux centaines : 100, 101, 110, 111 ; et ainsi de suite. Un chiffre binaire (un zéro ou un un) est appelé un bit en anglais. Il s'agit de la contraction de l'expression « binary digit ». Nous l'emploierons assez souvent dans la suite de ce cours par souci d'économie.

Mais pourquoi utiliser la base deux et non la base dix ?

Parce que les données circulent sous forme de courants électriques. Or, la tension de ceux-ci n'étant pas toujours stable, il est difficile de réaliser un système fiable sachant détecter dix valeurs différentes. Par contre, c'est parfaitement possible avec deux valeurs : il y a du courant ou il n'y en a pas.

Sans rentrer dans trop de détails, ces données sont stockés dans une partie de l'ordinateur qu'on appelle mémoire. Cependant comment accéder aux données dans cette mémoire et comment les récupérer de manière particulière ? Chaque valeur de la mémoire se voit attribuer un nombre unique, une adresse, qui va permettre de le sélectionner et de l'identifier parmi tous les autres. Imaginez la mémoire de l'ordinateur comme une immense armoire, qui contiendrait beaucoup de tiroirs (les cases mémoires) pouvant chacun contenir un nombre. Chaque tiroir se voit attribuer un numéro pour le reconnaître parmi tous les autres. Nous pourrions ainsi demander quel est le contenu du tiroir numéro 27. Pour la mémoire, c'est pareil. Chaque case mémoire a un numéro : son adresse. Plus généralement, toutes les mémoires disposent d'un mécanisme similaire pour retrouver les données. Aussi, vous entendrez souvent le terme de référence qui désigne un moyen (comme une adresse) permettant de localiser une donnée. Il s'agit simplement d'une notion plus générale.

Tout cela est bien sympathique, mais manipuler explicitement des références (des adresses si vous préférez) est un vrai calvaire, de même que de s'évertuer à calculer en base deux. Heureusement pour nous, les langages de programmation (et notamment Python), se chargent d'effectuer les conversions pour nous et remplacent les références par des variables.

Une variable correspondra à une portion de mémoire, appelée objet, à laquelle nous donnerons un nom. Ce nom permettra d'identifier notre variable, tout comme une référence permet d'identifier une portion de mémoire parmi toutes les autres. Nous allons ainsi pouvoir nommer les données que nous manipulons, chacun de ces noms étant remplacés lors de la compilation par une référence (le plus souvent une adresse).

8.3.2 Déclarer et affecter une variable

Types de données

Les données dans un ordinateur sont représentés par leur "type" par exemple : un nombre entier et un nombre à virgule ne sera pas stocké de la même manière dans la mémoire. Ainsi il existe différent types de données. Les types de données défini en naturL sont les types suivants :

— entier : Représente un nombre entier
— reel : Représente un nombre réel (ou nombre à virgule)
— booleen : Représente une valeur de vérité (vrai ou faux)

— caractere : Répresente un unique caractère.
— chaine : Représente une chaine de caractères soit un ou plusieurs caractères (y compris les espaces).

En fonction de l'usage que l'on veut faire de la variable, le type de donnée et les opérations possible avec cette variable vont varier. Il existe cependant des compatibilités entre les types en naturL. Par exemple, le type entier est compatible avec le type reel ce qui permet les opérations arithmétiques entre les deux sans aucune forme de conversion de l'un vers l'autre. Cependant il n'est pas possible d'ajouter un entier à un booleen en naturL. La raison de cette impossibilité viens du fait que le typage en naturL est dit fort. C'est à dire que globalement les types sont très peu compatibles entre eux contrairement à d'autres langages de programmation.

Déclaration

La déclaration d'une variable sert à réserver une "case mémoire" pour cette dernière et à lui affecter une adresse. La taille de cette case mémoire dépends du type de donnée, un reel est plus lourd à stocker en mémoire car il prends plus de place avec la virgule. En naturL, la déclaration se fait dans le champ "variables" il se forme de cette façon :

```
variables
    entier var
fin
```

Python ne nécessite pas de déclaration pour une variable mais naturL si, traduire le code précèdent en Python n'aurait donc pas beaucoup de sens.

Affectation

Il est maintenant l'heure de donner une valeur à notre variable ! Pour faire ceci, on utilise l'opérateur d'affectation. En naturL il est représenté par une flèche vers la gauche (<-). Si l'on veut affecter la valeur 2 à la variable var il faudra donc écrire :

```
variables
    entier var
fin
var <- 2
```

Qui se traduira par le code Python suivant :

```
var = 2
```

On remarque immédiatement qu'en Python, le type n'as pas à être précisé. Il est "inféré" c'est à dire laissé indéterminé jusqu'à ce qu'une affectation le détermine [3].

Cependant si l'on essaye d'affecter une valeur d'un type différent que celui déclaré, cela lève une erreur de type. Par exemple le code suivant :

```
variables
    entier var
fin
var <- vrai
```

Renvoie

```
Erreur de Type à la ligne 4: 'var' a le type 'entier'
mais le type affecté est 'booleen'
```

Similairement :

```
variables
    entier pi
fin
pi <- 3.1415
```

Renvoie

```
Erreur de Type à la ligne 4: 'pi' a le type 'entier'
mais le type affecté est 'reel'
```

Pour préciser qu'un nombre est du type réel il faut rajouter un point. Par exemple 2 est un entier mais 2. est un réel :

```
variables
    reel var
fin
var <- 2.
```

Nous donne

```
var = 2.
```

3. En naturL il est possible de laisser le programme inférer le type grâce au type " ?" qui permet le polymorphisme (plus d'explications dans la partie liste)

Enfin on peut affecter une variable à une autre du moment que leurs types sont les mêmes ou qu'ils sont compatibles (on peut affecter un entier à un réel mais pas un réel à un entier) :

```
variables
    reel var
    reel var2
fin
var <- 2.
var2 <- var
```

```
var = 2.
var2 = var
```

Cependant le code suivant donne une erreur de type :

```
variables
    entier var
    reel var2
fin
var2 <- 2.
var <- var2
```

Donne l'erreur suivante :

```
Erreur de Type à la ligne 6: 'var' a le type 'entier' mais le type affecté est 'r
```

Une instruction particulièrement utile que vous verrez dans beaucoup de langages est la suivante :

```
variables
    entier var
fin
var <- 1
var <- var + 1
```

Cela signifie que la nouvelle valeur de var sera sa valeur + 1 soit 2 ici. Cette instruction est particulièrement pratique dans les structures de contrôle dites "boucles" que nous verrons plus tard.

L'affectation des différents types de données se fait de manières similaire :

```
variables
  entier a
```

```
    reel b
  caractere c
  chaine d
  booleen e
fin
a <- 2
b <- 3.14
c <- 'e'
d <- "Hello world"
e <- faux
```

Nous donne en Python :

```
a = 2
b = 3.14
c = 'e'
d = "Hello world"
e = False
```

Les caractères sont représentés par des guillemets simple et les chaînes par des guillemets doubles. Les booleen sont une valeur de vérité donc soit faux soit vrai.

8.4 Opérations sur les variables

8.4.1 Simplification automatique et affichage

naturL possède un système de simplification automatique, ainsi une expression arithmétique simplifiable est simplifiée avant sa traduction. Ainsi il est tout à fait possible de laisser naturL faire des calculs arithmétiques simples par exemple :

```
variables
    entier var
fin
var <- 2 + 5 * 4
```

Nous donne

```
var = 22
```

En appliquant les priorités sur les opérations. Si l'on ajoute des parenthèses on obtient un résultat différent par exemple :

```
variables
    entier var
fin
var <- (2 + 5) * 4
```

Nous donne

```
var = 28
```

Si l'on souhaite afficher dans un interpréteur Python des variables, on peut utiliser la fonction AFFICHER qui se traduit en la fonction PRINT de Python. Par exemple :

```
variables
    entier var
fin
var <- (2 + 5) * 4
```

Nous donne

```
var = 28
print(var)
```

Après exécution du script python dans un terminal on obtient :

```
28
```

Comme dit précédemment tous les types ne sont pas compatibles entre eux. Cependant entre les variables d'un même type il existe différentes opérations.

8.4.2 Les opérations sur les types numériques : entier et réel

Il est possible d'effectuer des opérations arithmétiques entre les variables de type numériques comme les entiers et les réels, cependant toutes ne sont pas possibles. Prenons pour exemple le code suivant :

```
variables
    reel a
fin
a <- 2.5 + 1
```

Qui retourne en python

```
a = 3.5
```

Cependant la division entière n'est pas correcte avec un flottant :

```
variables
    reel a
fin
a <- 2.5 div 1
```

Qui retourne l'erreur suivante :

```
Erreur de Type à la ligne 4: Opération invalide pour les
expression de type reel et de type entier
```

Cependant il est totalement possible d'affecter le résultat d'une opération entière à un réel :

```
variables
    reel a
fin
a <- 3 div 2
```

Qui retourne en python

```
a = 1
```

8.4.3 Les opérations entre les types caractères et chaînes de caractères

Les chaînes de caractères sont des types dit "immuables" ont ne peut donc pas en modifier le contenu de manière spécifique une fois qu'ils sont formés. Cependant, on peut effectuer des opérations avec les chaînes de caractères. L'opération principale que l'on peut faire est la concaténation le type chaîne est compatible avec le caractère par concaténation mais l'inverse n'est pas possible en effet une variable du type caractère ne peut contenir qu'un seul caractère. Ainsi :

```
variables
    chaine s
fin
s <- "Hello" + ", world!"
```

Retourne en python

```
s = "Hello, world!"
```

Comme dit précédemment la concaténation entre une chaîne et un caractère est gauche :

```
variables
    chaine s
fin
s <- "Hello : " + 'c'
```

Retourne :

```
s = "Hello : c"
```

Mais :

```
variables
    chaine s
fin
s <- "Hello : " + 'c'
```

Retourne :

```
s = "Hello : c"
```

Cependant :

```
variables
    caractere c
fin
c <- 's' + 'd'
```

Lève une erreur de type :

```
Erreur de Type à la ligne 4: 'c' a le type 'caractere' mais
le type affecté est 'chaine'
```

8.4.4 Les opérations logiques : le type booléen

Les opérations logiques sont la base de la logique combinatoire et des programmes informatiques. Elle s'évaluent [4] à une valeur de vérité : vrai ou faux.

Il existe plusieurs opérateurs logiques qui permettent ce qu'on appelle les accès conditionnels que nous verrons après dans les "structures de contrôle".

Avec le type booléen viens 3 opération principales : et, ou ainsi que non.

L'opérateur non

Le plus simple est l'opérateur de négation non, il inverse la valeur de vérité. Ainsi non vrai et faux et non faux est vrai. Un exemple ?

```
variables
    booleen b
fin
b <- non vrai
```

Retourne en Python

```
b = False
```

On remarque que les booléens en Python s'écrivent avec une majuscule : vrai s'écrit True en Python et faux False.

L'opérateur ou

L'opérateur ou est un opérateur dit absorbant par le vrai : si une deux des conditions est vraie alors il évalue à vrai sinon faux. Il est important de savoir que cet opérateur est évalué de manière séquentielle donc si la première partie d'un ou est vrai alors le compilateur ne va pas chercher à évaluer la valeur de vérité de l'autre partie parce qu'il est sur que la totalité de l'expression évaluera à vrai. Par exemple :

```
variables
    booleen b
fin
b <- vrai ou faux
```

4. L'évaluation d'une expression ou d'une opération correspond au résultat final d'un calcul qu'il soit logique ou mathématique : 1 + 2 évalue à 3

Retourne en Python

```
b = True
```

Cependant si les deux conditions sont fausses :

```
variables
    booleen b
fin
b <- faux ou faux
```

Retourne en Python

```
b = False
```

L'opérateur et

L'opérateur et est un opérateur dit absorbant par le faux : si une deux des conditions est fausse alors il évalue à faux sinon vrai. Il également est important de savoir que comme pour le ou cet opérateur est évalué de manière séquentielle donc si la première partie d'un et est fausse alors le compilateur ne va pas chercher à évaluer la valeur de vérité de l'autre partie parce qu'il est sur que la totalité de l'expression évaluera à faux. Par exemple :

```
variables
    booleen b
fin
b <- faux et vrai
```

Retourne en Python

```
b = False
```

Cependant si les deux conditions sont vraies :

```
variables
    booleen b
fin
b <- vrai ou vrai
```

Retourne en Python

```
b = True
```

8.5 Structure de contrôle

Dans notre premier chapitre, nous avons vu que l'activité essentielle d'un programmeur est la résolution de problèmes. Or, pour résoudre un problème informatique, il faut toujours effectuer une série d'actions dans un certain ordre. La description structurée de ces actions et de l'ordre dans lequel il convient de les effectuer s'appelle un algorithme. Le « chemin » suivi par naturL à travers un programme est appelé un flux d'exécution, et les constructions qui le modifient sont appelées des instructions de contrôle de flux. Les structures de contrôle sont les groupes d'instructions qui déterminent l'ordre dans lequel les actions sont effectuées. En programmation moderne, il en existe seulement trois.

8.5.1 Séquence d'instructions

Sauf mention explicite, les instructions d'un programme s'exécutent les unes après les autres, dans l'ordre où elles ont été écrites à l'intérieur du script. Cette affirmation peut vous paraître banale et évidente à première vue. L'expérience montre cependant qu'un grand nombre d'erreurs sémantiques dans les programmes d'ordinateur sont la conséquence d'une mauvaise disposition des instructions. Plus vous progresserez dans l'art de la programmation, plus vous vous rendrez compte qu'il faut être extrêmement attentif à l'ordre dans lequel vous placez vos instructions les unes derrière les autres.

8.5.2 Structure de contrôle si

La structure de contrôle si permet de réguler le flux d'exécution d'un programme en fonction d'une valeur de vérité. En naturL elle se forme de cette manière :

```
si condition alors
    CODE
fin
```

Le code contenu dans la partie CODE est exécuté si et seulement si la condition condition est évaluée à vrai. Autrement dit un si vrai rédigeras toujours le flux d'exécution à "l'intérieur" du si tandis qu'un si faux ne le rédigeras jamais. Nous avons donc besoin de nouveaux opérateurs qui nous renvoient des valeurs booléenes si nous voulons écrire du code utile. Ces

opérateurs sont les opérateurs de comparaison. En naturL ils sont représentés comme en mathématiques par exemple le code suivant :

```
variables
    entier a
fin
a <- 2
si a = 2 alors
    a <- 3
fin
afficher(a)
```

Se lit de cette manière : "si la valeur de a est à 2 alors affecter 3 à deux, ensuite quoi qu'il arrive, afficher a". Ce code assez inutile [5] en soi nous donne une bonne idée du chemin que va prendre du flux d'exécution. En Python ce code se traduit par :

```
a = 2
if a == 2:
    a = 3
print(a)
```

On remarque plusieurs choses sur ce code : l'égalité est déjà occupé par l'affectation en Python, c'est le double égal qui représente l'opérateur d'égalité (comme dans beaucoup de langages de programmation d'ailleurs). De plus le mot clé "alors" est remplacé par un " :". Enfin, il n'y a pas de fin pour dire où s'arrête les instructions à l'intérieur du si. Ainsi comment l'interpréteur Python peut-il savoir si l'instruction "print(a)" est dans la condition ou en dehors ? Grâce au niveau d'"indentation" : les espaces. En effet, on voit que l'instruction a = 3 est légèrement avancée par rapport à la ligne précédente. C'est ce qu'on appelle une indentation et elle est primordiale en Python. Je vous invite à vous renseigner plus en détail sur le langage Python et son système d'indentation qui ne seras pas détaillé dans ce manuel.

Les opérateurs de comparaisons en naturL ainsi que leurs homologues en Python sont détaillés dans le code ci dessous. Comme dit précédemment ce sont les mêmes qu'en mathématiques. Il suffit dans sa tête de se poser une question par exemple la comparaison "a $<$ 2" peut se comprendre comme "Est ce que a est inférieur strictement à 2 ?".

5. Les plus perspicaces d'entre vous voient déjà le problème de logique derrière ce code. La valeur est à l'origine affectée à 2 donc la condition est inutile : elle est toujours vraie

```
variables
    entier a
fin
a <- 2
si a = 2 alors
    a <- 3
fin
si a <= 2 alors
  a <- 4
fin
si a < 2 alors
  a <- 5
fin
si a > 2 alors
  a <- 6
fin
si a >= 2 alors
  a <- 7
fin

afficher(a)
```

```
a = 2
if a == 2:
    a = 3
if a <= 2:
    a = 4
if a < 2:
    a = 5
if a > 2:
    a = 6
if a >= 2:
    a = 7
print(a)
```

Remarquons ici que ces blocs d'instructions se comportent de manière séquentielle ainsi la valeur de a que l'on compare au fur et à mesure change. [6]

8.5.3 Structure de contrôle sinon

La structure de contrôle sinon viens directement en lien avec la structure si. Elle permet d'exécuter une instruction particulière si la condition du si qui le précède n'est pas vérifiée. Par exemple le code suivant :

```
variables
    entier a
fin
a <- 2
si a = 2 alors
    afficher(2)
sinon
```

6. Petit exercice, quelle est la valeur de a à la fin de l'exécution du programme si dessus, réponse à la page suivante en note de bas de page

```
    afficher(3)
fin
```

Se traduit en python par le code suivant :

```
a = 2
if a == 2:
    print(2)
else:
    print(3)
```

En fonction de la valeur de a, on obtient un résultat différent. Si a vaut 2 alors on affiche 2 sinon on affiche 3. la condition sinon est donc une extension de la condition si.

On obtient le diagramme d'exécution suivant [7] :

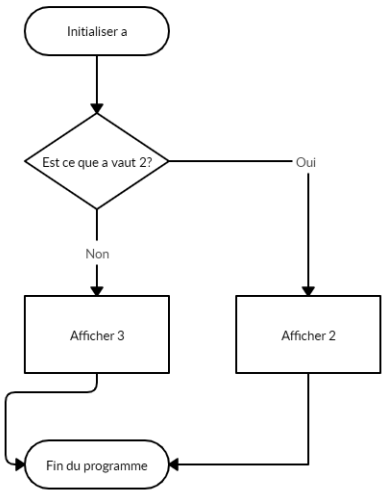

FIGURE 8.1 – Diagramme d'exécution du programme ci dessus

8.5.4 La structure de contrôle sinon_si

La structure de contrôle sinon_si est une fusion entre le sinon et le si. Elle permet de créer des conditions multiples pour un même chemin. Reprenons le

7. Un diagramme d'exécution est une manière de représenter un les possibles exécutions d'un algorithme. La réponse à la question de la page précédente est 7, a vaut 2 puis 3, 6 et enfin 7

code précédent.

```
variables
    entier a
fin
a <- 2
si a = 2 alors
    a <- 3
fin
si a <= 2 alors
  a <- 4
fin
si a < 2 alors
  a <- 5
fin
si a > 2 alors
  a <- 6
fin
si a >= 2 alors
  a <- 7
fin

afficher(a)
```

```
a = 2
if a == 2:
    a = 3
if a <= 2:
    a = 4
if a < 2:
    a = 5
if a > 2:
    a = 6
if a >= 2:
    a = 7
print(a)
```

Si l'on change les si par des sinon_si on obtient :

```
variables
    entier a
fin
a <- 2
si a = 2 alors
    a <- 3
sinon_si a <= 2
   alors
  a <- 4
sinon_si a < 2 alors
  a <- 5
sinon_si a > 2 alors
  a <- 6
sinon_si a >= 2
   alors
  a <- 7
fin

afficher(a)
```

```
a = 2
if a == 2:
    a = 3
elif a <= 2:
    a = 4
elif a < 2:
    a = 5
elif a > 2:
    a = 6
elif a >= 2:
    a = 7
print(a)
```

Observons la différence d'un point de vue de l'exécution.
Le premier code est exécuté séquentiellement, on a donc par ordre d'exécution :

1. On assigne 2 à a
2. Est ce que a vaut 2 ? Oui donc on assigne 3 à a. a vaut donc 3 à partir de maintenant.
3. Est ce que a est inférieur ou égal à 2 ? Non il vaut désormais 3
4. Est ce que a est inférieur strictement à 2 ? Non
5. Est ce que a est supérieur strictement à 2 ? Oui car il vaut 3, a vaut donc 6 désormais.
6. Est ce que a est supérieur ou égal à 2 ? Oui il vaut 6 donc il vaut désormais 7.
7. afficher(a) -> 7

Tandis que pour le deuxième code l'exécution se fait se cette manière :

1. On assigne 2 à a
2. Est ce que a vaut 2 ? Oui donc on assigne 3 à a. a vaut donc 3 à partir de maintenant.
3. On ignore toutes les instructions sinon_si car elles sont exécutés seulement si la condition précédente n'est pas satisfaite ce qui n'est pas le cas.

4. afficher(a) -> 3

Pour les plus visuels d'entre vous voila les deux diagrammes d'exécutions pour ces deux programmes.

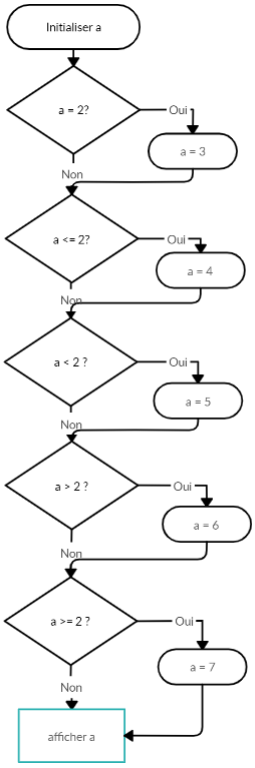

FIGURE 8.2 – Programme 1

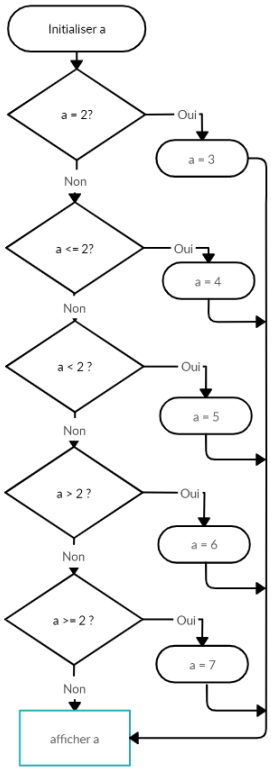

FIGURE 8.3 – Programme 2

8.6 Boucles

L'une des tâches que les ordinateurs effectuent le mieux c'est la répétition d'une instruction.

8.6.1 La boucle tant_que

En programmation, on appelle boucle un système d'instructions qui permet de répéter un certain nombre de fois (voire indéfiniment) toute une série

d'opérations. NATURL propose deux instructions particulières pour construire des boucles : l'instruction pour, très puissante, que nous étudierons dans la prochaine partie, et l'instruction tant_que que nous allons découvrir tout de suite. Considérons le code ci-dessous :

```
variables
  entier a
fin
a <- 1
tant_que (a <= 10) faire
  afficher(a)
  a <- a + 1
fin
```

Qui se traduit en Python par le code suivant :

```
a = 1
while a <= 10:
    print(a)
    a = a + 1
```

On remarque plusieurs choses : il y a un mot clé en plus comme tout à l'heure ici c'est le mot clé faire, cependant, on garde une chose commune : l'expression booléenne. En fait il faut lire l'instruction tant_que de cette manière : tant que cette condition est vraie faire. Nous avons ainsi construit notre première boucle de programmation, laquelle répète un certain nombre de fois le bloc d'instructions entre le faire et le fin. Voici comment cela fonctionne :

- Avec l'instruction tant_que, Python commence par évaluer la validité de la condition fournie entre parenthèses (celles-ci sont optionnelles, nous ne les avons utilisées que pour clarifier notre explication).
- Si la condition se révèle fausse, alors tout le bloc qui suit est ignoré et le programme reprends sa lecture après le mot clé fin.
- Si la condition est vraie, alors Python exécute tout le bloc d'instructions constituant le corps de la boucle, c'est-à-dire :
  - l'instruction a <- a + 1 qui incrémente d'une unité le contenu de la variable a (ce qui signifie que l'on affecte à la variable a une nouvelle valeur, qui est égale à la valeur précédente augmentée d'une unité).
  - l'appel de la fonction afficher() pour afficher la valeur courante de la variable a.

— Lorsque ces deux instructions ont été exécutées, nous avons assisté à une première itération, et le programme boucle, c'est-à-dire que l'exécution reprend à la ligne contenant l'instruction tant_que. La condition qui s'y trouve est à nouveau évaluée, et ainsi de suite. Dans notre exemple, si la condition a <= 10 est encore vraie, le corps de la boucle est exécuté une nouvelle fois et le bouclage se poursuit.