Le langage naturL
The ℕaturL Foundation

par

Simon Scatton
Vlad Argatu
Rostan Tabet
Adrian Le Comte

Documentation et Manuel Utilisateur

EPITA

Villejuif

2020

à Élie Brami pour ses heures de conseil et de participation à ce projet.

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
```

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
```

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)

```
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 'reel'
```

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.