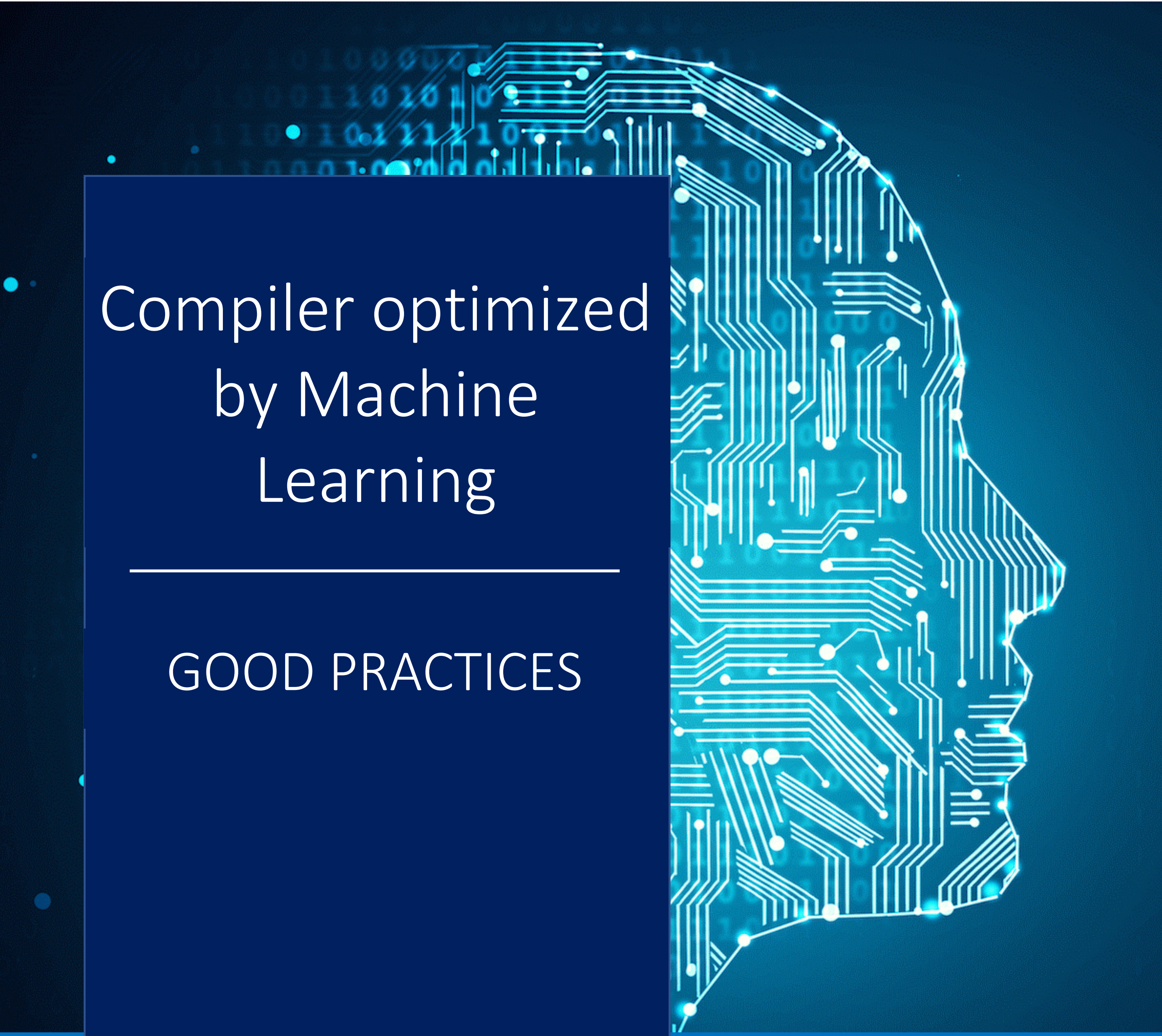

Compiler optimized by Machine Learning

GOOD PRACTICES

Date : 24/04/2021
GIACINTI Florian
MEURIN Jérémy

INDEX

Préambule

Motivation

Par où commencer lorsque l'on cherche à écrire un meilleur code ? Et comment maintenons-nous la qualité et la cohérence du code lorsque nous travaillons en équipe ?
La révision de code est un excellent moyen de procéder, par le biais de retours d'information et de partage des connaissances. Cependant, il s'agit d'une approche réactive, qui a tendance à ne pas être systématique et difficile à faire évoluer. En instaurant et en créant/adaptant une référence partagée pour les méthodes et la qualité, une équipe peut surmonter ces problèmes.
Il est rarement pratique pour tout le monde ou pour une personne de contrôler chaque ligne de code, tout en poursuivant son travail et ses responsabilités. En tant que relecteurs, nous avons tous notre propre expérience, nos connaissances techniques/de projet et notre niveau de compétence. Nous pouvons en tirer davantage en les partageant avec les autres plutôt qu'en appliquant les nôtres individuellement.
Ceci est un guide personnel, basé sur certaines des choses que je cherche. Tout le monde ne sera pas d'accord avec tout cela, mais je pense que c'est majoritairement raisonnable.
J'espère que cela vous aidera à rédiger un meilleur code, à être un contrôleur plus efficace et à susciter votre intérêt dans un apprentissage ultérieur.

Contrôle du code

Le contrôle du code est plus qu'un processus de contrôle, mais beaucoup de choses que nous recherchons sont simples et il y en a trop à retenir. Les check-lists nous aident à nous souvenir et informent les contributeurs dès le départ des attentes. Cela aide les contrôleurs à avoir plus de temps à consacrer aux choses les plus importantes et les moins simples, et à réduire le temps consacré aux révisions.
Ces éléments pouvant être automatisés devraient l'être (à l'aide d'outils tels qu'ESLint et ses plugins, ou checkstyle). Les détections et les corrections manuelles ne sont pas fiables, ne s'adaptent pas bien et consomment de l'attention qui serait mieux dépensée ailleurs.

Introduction

The document includes the best practices to adopt concerning the production of code within the project as well as the way to produce it and also the best practices of the functioning of the Gitlab of the project.

Software development

Software development is the study, design, construction, transformation, development, maintenance and improvement of software.
This work is done by employees of software publishers, IT services and engineering companies (SSII), independent workers (freelancers) and members of the open source software community.
Software is created step by step by a team of engineers in accordance with a set of specifications established by a client or an internal team. The software is broken down into different modules and a project manager, or architect, is responsible for the coherence of the whole.
Different activities allow us to take note of the user's expectations, to create a theoretical model of the software, which will serve as a construction plan, then to build the software, to control its good functioning and its adequacy to the need. The planning and the distribution of the works allow to anticipate the delay and the cost of manufacture.
The software is accompanied by an installation procedure, a procedure for verifying correct installation, documentation (sometimes created automatically from comments placed for this purpose in the source code) and a team to assist with deployment and maintenance, referred to as support.
In addition to the analysis, design, construction and testing work, an acceptance procedure - a simulated acquisition - will determine whether the software can be considered usable.

Source : Wikipedia

Using Gitlab

Use of issues

To access the issue board, click on the left tab, then on "Issue" and then on "Boards" to see the Kanban view:

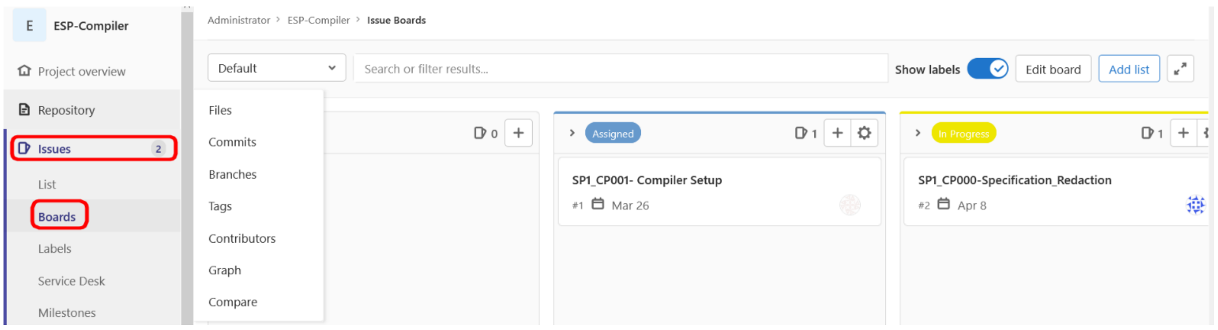

Figure 1 - Kanban view of a GitLab project

An Issue contains several pieces of information:

- A name that refers to a specification number of the associated technical document in the form "SPX_PROJYYY - feature description" with X the sprint number, PROJ the project ID and YYY the technical specification number.
- One or more assignees to solve the issue (since the gitlab is in non-professional form, the participants have the equivalent of assignee roles).
- A description of the issue (usually a summary of the technical specification).
- A rendering date.
- A linked development branch.
- Follow-up messages/comments.

A current status or tag following:
- Open : Issue created but not yet assigned to a member by the project manager.
- Assigned : Issue assigned to a member but not yet started.
- In Progress : Issue under development.
- In Test: Issue undergoing testing before being validated.
- Done - Wait to be merged: Issue waiting to be merged on the master branch.
- Closed : Issue closed and merged on the master branch.

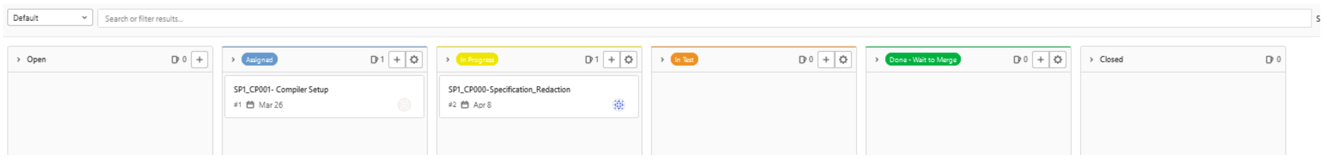
Default
Search or filter results...
Open
Assigned
In Progress
In Test
Done - Wait to Merge
Closed
SP1_CP001- Compiler Setup
#1 Mar 26
SP1_CP000-Specification_Redaction
#2 Apr 8

Figure 2 - View of the branches associated with the exits

When the issue is created, a branch will automatically be associated to it for the development of the associated functionalities. It is important to keep the Issue progress up to date (assigned label, comments, ...).

MANDATORY: Every Friday night, update your Issue in Progress (Issue with a label In Progress or In Test) with a comment on the progress of the work (functionality developed/implemented), difficulties and risks of delay.

Commit, push and merge management

General rules:

- Commits must be done fairly regularly in order to limit losses and minimize code conflicts.
- No push should be done on a master branch.
- Merges that create conflicts must be analyzed and resolved manually.

The header of the commit messages must take the following form:

[SPX_PROJYYY][DESC] Header of commit

Detailed description of the changes

With X the sprint number, PROJ the project ID and YYY the technical specification number. The [DESC] fields must be replaced by one or more of the following tags:

- [func] : add a feature.
- [edit] : modification of a feature.
- [del] : delete a feature.
- [fix] : fix a bug.
- [refa] : refactor code.
- [doc] : documentation / add comment.
- [misc] : when no tag above matches.

Example:
[SP1_CP000][func,doc] Debug mode Added to Lexer
[func] add printStep() to Lexer main function
[doc] add printStep() description in ReadMe

Branch management

Branch naming convention

SPX_PROJYYY-Specification_Desc

- X : Sprint number
- PROJ : Project Identifier [ML | CP]
- YYY : Specification Identifier (refer to technical specification number)

Exemple : SP1_CP001-Compiler_Setup

Specifications

The purpose of this chapter is to propose a set of conventions and rules to facilitate the understanding and therefore the maintenance of the code.
These rules are not to be followed explicitly to the letter: they are only presented to encourage developers to define and use rules in the realization of the code, especially in the context of teamwork. The proposed rules are those commonly used. However, there is no absolute rule and everyone can use all or part of the proposed rules.
The definition of conventions and rules is important for several reasons:
Most of the time spent on coding is spent on the evolutionary and corrective maintenance of an application
It is not always, or even rarely, the author of the code who performs these maintenances. These rules facilitate the readabilitý and thus the understanding of the code
The content of this document is largely inspired by the coding conventions historically proposed by Sun.

Code formatting

IF

```
# Déclaration d'une variable de test.
x = 5

# Si x inférieur à 10.
if x < 10:

    #Incerémantion
    x = x + 1

    # Code à executer.

```

An IF conditional should be written as follows. This is an example of a simple conditional. The comment must be associated with the line. Do not forget the opening and closing brackets. Ternaries are not allowed.

ELSE

```
# Déclaration d'une variable de test.
x = 5

# Si x inférieur à 10.
if x < 10:

    #Incerémantion
    x = x + 1

    # Code à executer.

else:

```

An ELSE conditional should be written as follows. This is an example of a simple conditional. The comment must be associated with the line. Do not forget the opening and closing brackets. Ternaries are not allowed.

WHILE

```python
# Déclaration d'une variable de test.
x = 5

# Tant que x inféeieur à 10.
while x < 10:

  #Incerémantion
  x = x + 1

  # Code à executer.



```

A WHILE conditional should be written as follows.
This is an example of a simple infinite double.
A WHILE loop must always have an exit gate.
The comment must be associated with the line. Do not forget the opening and closing brackets.

FOR

```python
# Déclaration d'une variable de test.
liste = [1,5,10,15,20,25]

# Pour touts les nombres dans la liste.
for i in liste:

  #Incerémantion
  x = x + 1

  # Code à executer.

```

A FOR conditional must be written in one of the following ways.
The comment must be associated with the line.
Do not forget the opening and closing brackets.

The comments

Language

The language of comments throughout the code is ENGLISH.

Typography

The comment writes the associated line of code. It explains nothing more or nothing less. All lines of code without any exception must be commented. A comment as by a capital letter and ends with a period like a basic sentence. You should put a space between the character that declares the comment and the first character of your comment to improve visibilitý.

```python
# I'm a good commentary.
# i'M not a good commentary
#I'm not a good commentary

```

Best practices

If your comments are done correctly, a person with no knowledge of code can read and understand your code.
A good practice is if possible to comment your code before you write it.

Don't forget if you go back over the commented code to change the logic to re-comment the code with the new logic. The comment should always explain in French what the line of code does.

It is usual and a bad habit in companies to say I will do it later. It is an upstream step of any development.

The function headers

The functions must have comments so that we can know how it works, its objectives and what it returns without even having to read the code of the latter.
In order to standardize the whole code, all functions must respect the following header:

```python
# NOM_DE_LA_FONCTION (Respect du formatage du nom)
#
# @description: Description en quelques mots de la fonction
# @param param1: Explication du paramètre.
# @param param2: Explication du paramètre.
# @return (Type de variable a retourner) (Dans quels cas ?).

def NOM_DE_LA_FONCTION(param1, param2):

    # Code à executer.

    # Si debug.
    if DEBUG:

      # Affichage du debug
      print("Je suis un debug")

    # On retourne l'addition des valeurs passès en paramètres.
    return (param1 + param2)


 # Appel de fonction.
 NOM_DE_LA_FONCTION(VAR1, VAR2)

```

Naming of functions

If it concerns a report that must contain or return a boolean
Prefix with is or has : isLoading, hasPlayed, isNotDone, hasNotPosted...

Prefix with the keyword IS if it is a getter on a boolean

Example: isAvaible()

Prefix with the GET keyword if it is a getter on something other than a boolean

Example : getDate()

Prefix with the SET keyword if it is a setter

Example: setDate()

If the function name has several words, it must be capitalized. Example : Function that performs a process on a creation date.

Name : processDateCreation()
Ban : process_date_creation()

Files headers

Each file must have as the first thing to read a small header.
This is to explain what the file is used for and provide some information such as its version or who wrote the file.

```
#!/usr/bin/env python3

# NOM_DU_FICHIER.py
#
# @description: Que fait ce fichier ? Quels but ?
# @version: 0.1.1 (Sous 3 digits) Version du document.
# @author: Nom complet ou pseudonyme.
# @contact: Email si vous souhaitez laisser un moyen de vous
contacter.
# @date : Date de création du fichier.

```

Code indentation

In order to share your code via Git and for a better visibility and compatibility of the code between the different development interfaces, we will use an indentation standard.
For a better readability of the code, you must indent your code. This is commonly called "tabulating" your code. You will have to configure your software so that a

tabulation corresponds to 2 spaces. Donc sur les fichiers il ne doit exister aucune tabulation, mais que des espaces. Si vous configurer correctement votre logiciel, quand vous allez appuyer sur la touche tabulation, elle va mettre deux espaces à la place.

Code debugging

To improve the debugging of the code as well as the speed to activate and remove it, it is necessary that all the comments of the code are under condition like the following screenshot.
The comments should be displayed only if the global constant of the file is active.
Take the following example.

```python
#!/usr/bin/env python3

# NOM_DU_FICHIER.py
#
# @description: Que fait ce fichier ? Quels but ?
# @version: 0.1.1 (Sous 3 digits) Version du document.
# @author: Nom complet ou pseudonyme.
# @contact: Email si vous souhaitez laisser un moyen de vous contacter.
# @date : Date de création du fichier.


# Importations des librairies externes.
import os
import time
import subprocess

# Constantes
MY_CONSTANTE = "Je suis une constante"

# Variable de debug (Actif = voir les debugs)
DEBUG = False

# Int qui contient une valeur.
VAR1 = 2

# Int qui contient une valeur.
VAR2 = 3



# NOM_DE_LA_FONCTION (Respect du formatage du nom)
#
# @description: Description en quelques mots de la fonction
# @param param1: Explication du paramètre.
# @param param2: Explication du paramètre.
# @return (Type de variable a retourner) (Dans quels cas ?).

def NOM_DE_LA_FONCTION(param1, param2):

  # Code à executer.

  # Si debug.
```

```
 if DEBUG:

  # Affichage du debug
  print("Je suis un debug")

 # On retourne l'addition des valeurs passès en paramètres.
 return (param1 + param2)


# Appel de fonction.
NOM_DE_LA_FONCTION(VAR1, VAR2)

```

Code complexity

To measure the complexity of the code, we will use the unit of cyclomatic complexity.

The cyclomatic complexity of a method is defined by the number of linearly independent paths that can be taken in this method.
More simply, it is the number of decision points of the method (if, case, while, ...) + 1 (the main path).
The cyclomatic complexity of a method is at least 1, since there is always at least one path.

The cyclomatic complexity of a method increases proportionally to the number of decision points. A method with a high cyclomatic complexity is more difficult to understand and maintain.
A cyclomatic complexity that is too high (above 30) indicates that the method needs to be refactored.
A cyclomatic complexity lower than 30 can be acceptable if the method is sufficiently tested.

The cyclomatic complexity is linked to the notion of "code coverage", i.e. the coverage of the code by the tests. Ideally, a method should have a number of unit tests equal to its cyclomatic complexity in order to have a "code coverage" of 100%. This means that each path of the method has been tested.

Source : http://www-igm.univ-mlv.fr/~dr/XPOSE2008/Measuring%20the%20quality%20of%20source%20code%20-%20Algorithms%20and%20tools/cyclomatic-complexity.html
So we will allow a cyclomatic complexity of 6 per function.
To reduce its complexity, we just have to divide its code into functions with shorter and more precise processing.

Glossary

Gitlab: GitLab is a free git-based forge software offering wiki functionality, a bug tracking system, continuous integration and continuous delivery.

Conditional: In computer science, a conditional statement, (also called conditional expression), is a function of a programming language, which performs different calculations or actions, depending on the evaluation of a Boolean condition, namely true or false.

Table of illustrations