

GPS, Sous-programmes, DBC et Test

Objectifs

- Utiliser GPS, GNAT Programming Studio
- Comprendre les modes de passage des paramètres
- Savoir écrire des programmes de test simples
- Savoir exprimer les préconditions et postcondition en Ada
- Spécifier, implanter et tester des sous-programmes

1 GNAT Programming Studio

Nous utiliserons maintenant GNAT Studio, un atelier de développement qui facilite la vie du programmeur en lui permettant, via une interface utilisateur graphique, d'éditer ses fichiers sources, les compiler, les exécuter, les mettre au point...

Exercice 1 : Réparer

La version de Gnat Studio fournie avec Ubuntu 20.04 (et installée à l'N7) ne fonctionne pas. La version fournie par AdaCore a donc été installée. Pour y avoir accès, il faut modifier la variable d'environnement PATH. Pour que cette modification soit permanente, on va l'enregistrer dans le fichier \$HOME/.bashrc. On y ajoute donc la ligne suivante ¹ (au début) :

```
export PATH=/mnt/n7fs/ens/tp_cregut/gnat/bin:$PATH
```

Cette modification ne sera effective que dans les nouveaux terminaux. Ouvrir un nouveau terminal et faire :

```
which gnatstudio
```

La réponse doit être /mnt/n7fs/ens/tp_cregut/gnat/bin/gnatstudio. Sinon, relire cet exercice et vérifier que tout a été fait correctement.

Exercice 2 : Lancer

Lancer GNAT Studio avec la commande ² gnatstudio, depuis un terminal. Lors du premier lancement, GNAT Studio propose de faire une rapide configuration. On peut se contenter de la configuration par défaut.

-
1. Attention, aucun espace sauf entre export et PATH.
 2. Autrefois GNAT Stutio s'appelait GNAT Programming Studio (GPS).

Si vous l'avez lancé depuis le dossier du TP, il a lu le fichier de description du projet (fichier en .gpr) sinon, il vous propose, entre autres, d'ouvrir un projet ; il faut alors sélectionner le dossier du TP (qui contient le fichier .gpr).


Exercice 3 : Définir les préférences


Modifions quelques paramètres de GNAT Studio. Faire *Edit / Preferences*. Sélectionner *General* et choisir la valeur *Unicode UTF-8* pour *Character set*. Par défaut, GNAT Studio est configuré pour des fichiers en latin1. Dans *Editor / Ada*, définir *Default indentation* à 4, *Continuation lines* à 8 et *Record indentation* à 4. Fermer (*Close*). Les préférences sont enregistrées.

Exercice 4 : Éditer, compiler et exécuter

Vos fichiers sources sont organisés en projets (vue *Project*, à gauche figure 1).

1. *Éditer*. Déplier l'arbre des fichiers du projet pour faire apparaître les fichiers Ada (figure 1) et double-cliquer sur `premier_programme.adb`. L'éditeur s'ouvre...

2. *Compiler*. Faire *Build > Compile File* (ou cliquer sur l'icône ). Les lignes avec des erreurs apparaissent en rouge (et en orange pour les avertissements). Si une clé ou un triangle apparaît sur la gauche c'est qu'une erreur a été détectée sur la ligne. En plaçant la souris dessus, on peut voir l'erreur. En cliquant dessus, on peut la corriger si un correctif est proposé. Essayer. Il faut ensuite recompiler le fichier pour que les erreurs soient mises à jour.

3. *Produire l'exécutable*. Attention, *Compile File* compile le fichier mais ne produit pas d'exécutable. Pour obtenir un exécutable, il faut faire *Build > Project* puis choisir le fichier source pour lequel l'exécutable doit être produit, par exemple `premier_programme.adb`. C'est cependant plus rapide de faire un clic droit sur l'icône  et choisir le programme à compiler.

Remarque : Attention, n'apparaissent que les fichiers qui ont été déclarés comme programme principal pour le projet. Pour déclarer un fichier comme principal, il faut faire un clic droit sur le projet dans la vue *Project*, choisir *Project > Properties*³ puis, sélectionner *Main* dans les éléments à gauche puis, dans l'onglet *Main files*, on peut ajouter des fichiers correspondant à des programmes principaux ou en supprimer.

3. On peut aussi le faire depuis la barre de menu : *Project > Edit Project Properties*.

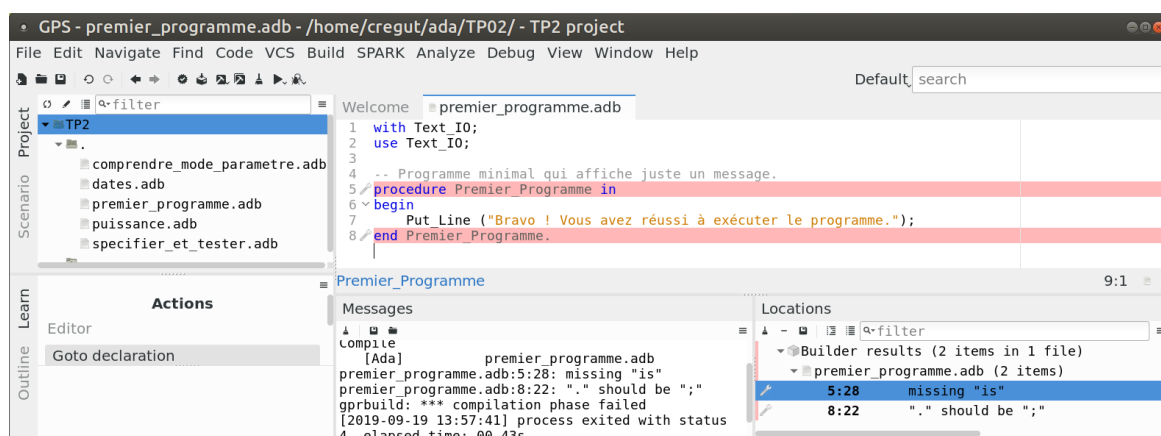




FIGURE 1 – GNAT Studio

4. *Exécuter*. Pour exécuter, on peut faire *Build > Run* et choisir le programme à exécuter. Il est cependant plus pratique de faire un clic droit sur l'icône  et choisir le fichier car il sera automatiquement compilé avant d'être exécuté. Cliquer directement sur cette icône, (compile et) exécute à nouveau le dernier programme exécuté. Exécuter `premier_programme`.

5. *Modifier le programme*. Changer le message qui est affiché par le programme et l'exécuter.

Le nouveau message doit apparaître... sinon c'est certainement que vous avez oublié d'engendrer l'exécutable (Build) !

6. *Nettoyer*. Pour nettoyer le dossier des fichiers engendrés automatiquement, on peut faire *Build > Clean > Clean All* ou cliquer sur l'icône .

2 Modes de passage des paramètres

Exercice 5 : Modes de passage des paramètres

Dans cet exercice, nous considérons le programme `comprendre_mode_parametre.adb`.

1. Lire ce programme et répondre aux questions qu'il contient. On donnera la réponse directement dans le fichier sous forme d'un commentaire en fin de ligne.

2. Une fois les commentaires ajoutés, compiler le programme pour vérifier les réponses données à la question précédente.

3 Les sous-programmes en Ada : bonnes pratiques

Voyons quelques bonnes pratiques lors de la définition de sous-programmes en Ada. Dans ces exercices, nous nous appuyons sur le programme `specifier_et_tester.adb`.

Exercice 6 : Appeler un sous-programme

1. Lire la spécification du sous-programme `Pgcd` puis indiquer si le code de la procédure `Utiliser_Pgcd` est correct ou non.

2. Compiler, puis exécuter pour confirmer votre réponse. Que se passe-t-il ? On ne corrigera pas le programme ! Pour arrêter le programme sous GNAT Studio, fermer la vue montrant son exécution (en cliquant sur la petite croix, sur la droite de l'onglet). Elle est nommée *Run : specifier_et_tester*.

Exercice 7 : Formaliser les préconditions et postconditions en Ada

Ada permet de formaliser les préconditions et les postconditions grâce au `with` que l'on peut mettre avant le `is` d'un sous-programme.

1. Lire et comprendre la précondition (Pre) et la postcondition (Post) de `Pgcd`. On remarque que préconditions et postconditions sont exprimées sous forme d'expressions booléennes Ada. Que signifie `Pgcd'Result` ?

2. Lire et comprendre la postcondition de `Permuter`. Que signifie `A'Old` et `B'Old` ?

Exercice 8 : Instrumenter les préconditions et postconditions

Par défaut, ces préconditions et postconditions ne sont pas exploitées. On peut demander au

compilateur Ada (option `-gnata` comme *gnat assertion*) de les utiliser pour vérifier qu'elles sont respectées lors de l'exécution du programme : le compilateur va engendrer du code supplémentaire qui vérifie les préconditions à l'entrée du sous-programme et les postconditions à la sortie. Pour positionner cette option avec GNAT Studio, faire un clic droit sur le projet, choisir *Properties*, puis *Switches* puis *Ada*. Mettre « `-gnata -gnatwa -g` » dans la zone de saisie, puis *Valider*.

Tout nettoyer et recompiler (pour être sûr que les nouvelles options sont prises en compte).

On a utilisé l'option `-g` pour conserver dans l'exécutable des liens vers les fichiers sources Ada et les numéros de ligne. Ceci sera utile pour la suite quand on utilisera le débogueur.



Exécuter le programme. Que se passe-t-il ?

Conclusion : Toujours compiler avec l'option `-gnata` (et `-g`) et `-gnatwa`, bien sûr !

Important : Si le programme s'arrête sur une précondition non vérifiée, l'erreur est dans le code de l'appelant (ou dans la précondition). S'il s'arrête sur une postcondition non vérifiée, l'erreur est dans le sous-programme qui porte cette postcondition (ou dans la postcondition).

Exercice 9 : Trouver l'appelant qui n'a pas respecté la précondition

Quand le programme détecte une précondition non vérifiée, il s'arrête et n'affiche que la ligne où la précondition est spécifiée. Il serait utile de savoir où a été fait l'appel à ce sous-programme car c'est là que l'appel incorrect a été fait. On peut y arriver en utilisant un débogueur.

Faire *Debug > Initialize > spécifier_et_tester* ou faire un clic droit sur l'icône . La perspective *Debug* apparaît avec la pile des appels (*call stack*) de sous-programmes à droite⁴ et le débogueur en bas (gdb). De nouveaux boutons apparaissent () pour, de gauche à droite, lancer l'exécution (*Start/Continue*), interrompre l'exécution (*Interrupt*), arrêter le débogueur (*Terminate*), ligne par ligne (*Next*), terminer l'appel courant (*Finish*), exécuter instruction par instruction (*Step*), etc.

Lancer le programme en cliquant sur lecture puis en validant (pas de d'arguments attendus). La première instruction du programme principal passe sur fond vert. C'est la prochaine instruction à exécuter.

On peut faire apparaître les variables locales et les arguments en faisant *Debug > Data > Display Local Variables* et *Debug > Data > Display Arguments*.

Pour interrompre l'exécution si une exception se produit, taper dans la vue débogueur :

```
(gdb) catch exception
```

Cliquer de nouveau sur lecture. L'exécution s'arrête sur la levée de l'exception. On peut alors cliquer sur les lignes affichées dans la vue pile des appels. La ligne du bas (*spécifier_et_tester*) correspond au programme principal. Quand on clique dessus, on voit l'instruction en cours d'exécution (appel *Utiliser_Pgcd* du programme). La ligne au dessus correspond à l'instruction de *Utiliser_Pgcd* qui en cours d'exécution (*Resultat := Pgcd (0, 10)*) et donc l'appel à *Pgcd*. Celle au dessus correspond à la vérification de la précondition de *Pgcd* qui a déclenché la levée de l'exception. Les lignes au dessus sont dans la bibliothèque Ada. On sait donc où est l'appel incorrect : Première instruction de *Utiliser_Pgcd* !

4. Faire *Debug > Data > Call Stack* si elle n'apparaît pas.

Remarque : Dans la vue du débogueur, on peut taper `bt full` pour voir les appels des sous-programmes et la valeurs des paramètres et variables locales.

Exercice 10 : Écrire un programme de test

Les exemples et tests identifiés lors de la spécification du programme devraient être traduits en programmes de test de manière à pouvoir facilement les exécuter pour détecter des erreurs dans le programme lors de son développement initial mais aussi tout au long de sa vie et en particulier lors des évolutions⁵ qu'il subira inévitablement. Voyons comment écrire assez simplement⁶ des programmes de test pour les sous-programmes de notre programme. On parle de *test unitaire*.

Chaque programme de test sera une procédure dont le nom commence⁷ par `Tester_`, par exemple `Tester_Pgcd`, et dans laquelle on utilisera l'instruction :

```
pragma Assert (condition);
```

`condition` est une expression booléenne qui devrait être vraie. Si elle est fausse une erreur est détectée (soit dans le sous-programme testé, soit dans le test). Cette instruction n'est prise en compte par le compilateur que si le programme est compilé avec l'option `-gnata`. Dans ce cas, elle ne fait rien si la condition est vraie ; elle lève une exception (et donc généralement arrête le programme) si la condition est fausse.

Remarque : On pourrait aussi utiliser `pragma Assert` pour les conditions exprimées comme un commentaire après une boucle :

```
1  ...
2  while N > 0 loop
3      ...
4  end loop;
5  pragma Assert (N <= 0);
```

1. Lire les procédures `Tester_Pgcd` et `Tester_Performance_Pgcd`. Elles correspondent à des programmes de test de la fonction `Pgcd`. Chaque ligne correspond à un test de cette fonction. On constate que l'on peut définir plusieurs procédures de test pour un même sous-programme.

2. Après avoir mis en commentaire l'appel à `Utiliser_Pgcd` dans le programme principal, compiler et exécuter le programme. Ne pas oublier `-gnata` !

Que se passe-t-il ?

3. Lire la procédure `Tester_Permuter` et en déduire comment l'on fait pour tester une procédure (qui n'a pas de valeur et ne peut donc pas apparaître dans la condition d'un `Assert`).

4 Appliquons ce que nous venons de voir...

Exercice 11 : Puissance entière d'un nombre réel

Nous souhaitons écrire une fonction qui calcule la puissance entière d'un nombre réel sans utili-

5. Ces évolutions peuvent être liées à la correction d'erreurs identifiées lors de l'utilisation du programme. On parle de *maintenance corrective*. Cependant, les évolutions sont souvent dues à la prise en compte de nouveaux besoins des utilisateurs. On parle alors de *maintenance évolutive*. Les tests sont alors utiles pour pouvoir vérifier que les corrections ou les évolutions n'ont pas introduit de nouvelles erreurs. On parle de *test de non régression*.

6. Ce n'est pas la solution optimale mais les autres solutions proposées par Ada sont lourdes à mettre en œuvre.

7. C'est une convention, pas une obligation.

ser l'opérateur puissance du langage Ada (**). Nous écrirons deux versions de cette fonction, la première récursive, la seconde itérative. Pour la version itérative, nous nous appuierons sur une fonction qui calcule la puissance dans le cas où l'exposant est positif.

1. Écrire le code de la fonction `Puissance_Positive_Iteratif`. Exécuter le programme. La procédure de test `Tester_Puissance_Positive_Iteratif` ne doit pas signaler d'erreurs.
2. Compléter les préconditions et écrire le code de la fonction `Puissance_Iteratif`. On s'appuiera bien sûr sur la fonction précédente.

Exécuter le programme. `Tester_Puissance_Iteratif` ne doit pas signaler d'erreurs.

3. Rappeler quelle est la structure générale d'un sous-programme récursif.
4. Indiquer un moyen qui permet de garantir la terminaison d'un sous-programme récursif
5. Compléter la spécification et écrire le code de la fonction `Puissance_Rekursif`. Exécuter le programme. La procédure de test `Tester_Puissance_Rekursif` ne doit pas signaler d'erreurs.
6. Est-ce que le programme principal est correct au vu de la spécification des fonctions qui calculent la puissance ?
7. Corriger le programme principal pour qu'il fonctionne correctement si l'utilisateur saisit 0.0 pour le nombre et -3 pour l'exposant. On ne doit pas modifier le premier niveau de raffinement du programme principal.
8. Exécuter le programme avec -1.0 pour le nombre et 1234567891 pour l'exposant. Qu'en conclure ?

Exercice 12 : Spécifier, implanter et tester des sous-programmes

Lors de la spécification du sous-programme, on formalisera grâce aux Pre et Post d'Ada les préconditions et les postconditions (dans la mesure du possible). On écrira aussi de manière systématique des procédures de test, de préférence avant d'écrire le code du sous-programme correspondant.

1. Spécifier, implanter et tester⁸ un sous-programme qui calcule le nombre de jours d'un mois. Le mois sera représenté par un entier (1 pour janvier, 12 pour décembre).
2. On décide de représenter une date par trois entiers, un pour le numéro de jour dans le mois (de 1 à 31), le second par le numéro du mois (de 1 à 12) et le troisième pour l'année.

Spécifier, implanter et tester un sous-programme qui permet de modifier une date pour passer au lendemain. Par exemple, si une date correspond au 5 octobre 2017, cette date deviendra, après exécution de ce sous-programme, le 6 octobre 2017.

3. Dans la question précédente, on a représenté une date par trois entiers. Quelles sont les inconvénients de cette modélisation ?

Remarque : Dans cet exercice on a explicitement demandé de « spécifier, implanter et tester » mais ceci doit être fait systématiquement dès que l'on définit un sous-programme même quand ce n'est pas demandé explicitement !

8. Les tests doivent être écrits dès que la spécification du sous-programme est terminée. Cependant, ils ne pourront être exécutés que quand le sous-programme sera implanté. En Ada, on commencera par spécifier le sous-programme, on le définira avec corps vide (avec la seule instruction `null` pour une procédure ou avec un `return` d'une valeur du bon type dans le cas d'une fonction), on écrira les programmes, on compilera le tout (les tests échoueront), on implantera le sous-programme et, enfin, on le testera.