Rapport de projet

COMPILATEUR ET INTERPRÉTEUR DE LANGAGE « NATUREL »

Etienne COTTE ; Kilyan LE GALLIC
Méthodologie de la programmation
27 Janvier 2022

Abstract :

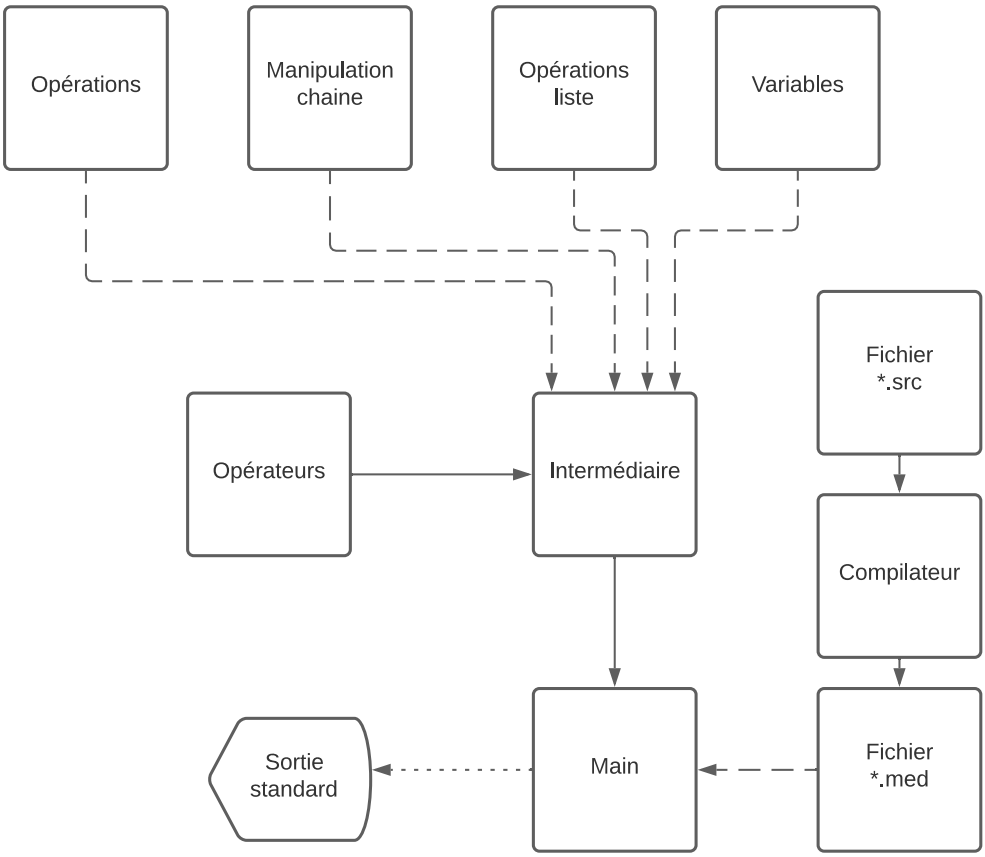
L'objectif de ce rapport est de formaliser les enseignements tirés de ce projet ainsi que disposer d'un point de référence dans l'avancement du projet. Ce rapport contient la présentation des solutions choisies, des principaux algorithmes, l'architecture de données, la stratégie de tests, le pilotage opérationnel du projet, ainsi qu'un bilan sur les principales métriques sortant du projet.

Introduction

L'objectif de ce projet est de disposer d'un compilateur écrit en Ada compilant des programmes écrits dans un langage source proche du langage algorithmique usuel en des programmes écrits en code intermédiaire à la syntaxe proche du BASIC, ainsi que d'un interpréteur permettant d'exécuter un code en langage intermédiaire de manière transparente pour l'utilisateur.

Table des matières

Architecture de l'application



Le module **Opérateurs** est chargé de proposer l'interprétation des opérateurs arithmétiques de bases {+ ; - ; / ; * }, des opérations logiques de bases {AND ; OR ; > ; < ; = ; >= ; <= }, des opérations définies sur les caractères { successeur ; prédécesseur } et des branchements logiques { GOTO ; GOTO IF }

Le module **Intermédiaire** est chargé de la lecture du fichier, du parsing des variables, du parsing des instructions, de la gestion et l'affichage des variables et des instructions, de l'interprétation des instructions et de l'affichage convivial de la mémoire et des instructions.

Le fichier **Main.adb** est l'endpoint utilisateur où il pourra renseigner le fichier à interpréter via l'invite de commande.

Choix réalisés

Le premier choix réalisé au cours de ce projet a été de développer l'application en plusieurs dossiers (interpréteur, main, compilateur) afin de garder une structure de fichiers cohérente et lisible dans le projet. Ce choix a requis d'adapter le processus de compilation des sources en indiquant à gnat quel(s) répertoire(s) utilisé(s) lors de la compilation des librairies. L'option permettant de compiler un fichier .adb avec des librairies externes est **-I<*chemin_relatif_vers_directory_repertory*>**.

Le second choix fut d'héberger le processus de contrôle de source sur la plateforme GitHub afin de permettre un partage et une sauvegarde des sources, dans le but de faciliter le développement et le pilotage du projet par l'équipe. L'utilisation des systèmes de releases nous a permis de figer les versions du projet en fonction de l'avancement afin d'éviter un lourd processus de rollback en cas de régression.

Le troisième choix réalisé a été de ne manipuler que des entiers au cœur de notre mémoire afin de disposer d'une unique instance de notre mémoire dans notre programme. Cela implique que les caractères sont convertis en entier afin de pouvoir être utilisés au sein du langage intermédiaire.

Le choix a été fait de déployer un pipeline sur une instance GitLab afin d'exécuter la compilation et le passage des tests à chaque dépôt de nouvelle release afin de conserver toutes les fonctionnalités développées.

Principaux algorithmes

Opérateur d'affectation

```
Procedure affectation(ptrInstruction : in out
T_List_Instruction) is
   begin
      -- Structure d'une ligne d'instructions
      -- z op x y
      if ptrInstruction.all.ptrIns.all.operandes.z.all.isConstant
then
        raise Variable_Constante;
      else
        if
(ptrInstruction.all.ptrIns.all.operandes.z.all.typeVariable =
ptrInstruction.all.ptrIns.all.operandes.x.all.typeVariable) then

ptrInstruction.all.ptrIns.all.operandes.z.all.valeurVariable :=
ptrInstruction.all.ptrIns.all.operandes.x.all.valeurVariable;
         else
           raise Type_Incorrect;
        end if;
      end if;
   end affectation;
```

Création de variables temporaire

```
function creer_variable_tmp (nomVariable : in
Unbounded_String; isCaractere : in boolean)
return T_Ptr_Variable is

      valeurVariableTmp : integer;
      nomVariableTmp : Unbounded_String;
      typeVariableTmp : Unbounded_String;

   begin

      if (isCaractere) then
        valeurVariableTmp :=
Character'Pos(element(nomVariable,1));
        typeVariableTmp :=
To_Unbounded_String("Caractere");
      else
        valeurVariableTmp :=
Integer'Value(To_String(nomVariable));
        typeVariableTmp :=
To_Unbounded_String("Entier");
      end if;

      nomVariableTmp :=
To_Unbounded_String("Tmp");

      return new
T_Variable'(valeurVariableTmp,
typeVariableTmp, nomVariableTmp, false);

   end creer_variable_tmp;
```

Recherche d'une variable dans la mémoire

```
function rechercher_variable (variables : in T_List_Variable;
nomVariable : in Unbounded_String) return T_List_Variable is

      copy : T_List_Variable;

   begin

      copy := variables;
      while copy.all.prev /= null loop -- Retour au début de la
liste
        copy := copy.all.prev;
      end loop;
      while copy /= null and then copy.all.ptrVar /= null and
then copy.all.ptrVar.all.nomVariable /= nomVariable loop
        copy := copy.all.next;
      end loop;
      if copy = null then
        raise Variable_Non_Trouvee;
      else
        return copy;
      end if;
```

Modification du compteur programme

```
procedure
changerInstructionParNumero(ptrInstruction :
in out T_List_Instruction; numInstruction : in
integer) is
      instuction_not_found: Exception;
   begin
      if (ptrInstruction = null) then
        raise instuction_not_found;
      elsif
(ptrInstruction.all.ptrIns.all.numInstruction <
numInstruction) then
        ptrInstruction := ptrInstruction.all.next;

changerInstructionParNumero(ptrInstruction,
numInstruction);
      elsif
(ptrInstruction.all.ptrIns.all.numInstruction >
numInstruction) then
        ptrInstruction := ptrInstruction.all.prev;

changerInstructionParNumero(ptrInstruction,
numInstruction);
      else
        null;
      end if;
      exception
        when instuction_not_found =>
          put("GOTO : la ligne précisée n'existe
pas dans le fichier");

   end changerInstructionParNumero;
```

Principaux types de données

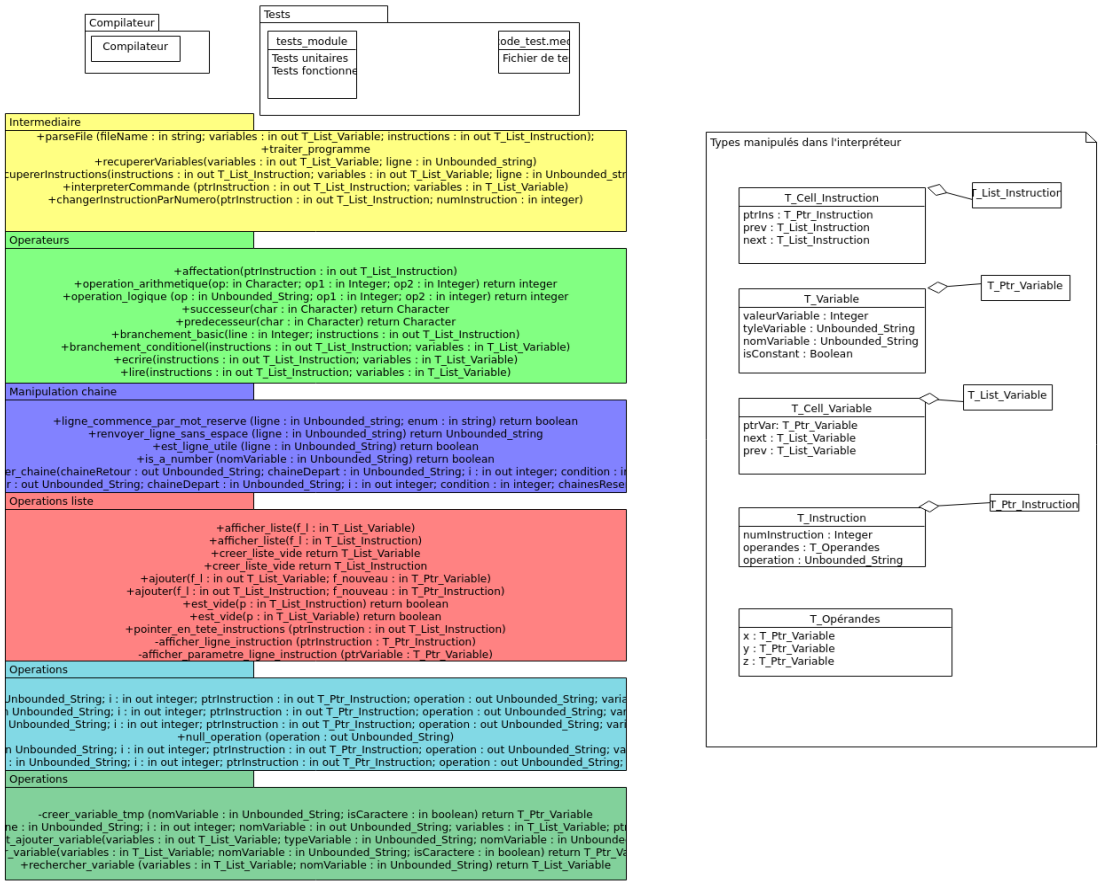


Figure 1 : Diagramme des types manipulés pour l'interpréteur

Les variables sont stockées dans une mémoire instanciée via une liste chaînée permettant de disposer d'exécuter des codes avec un nombre de variables inconnu avant interprétation et de dépasser le besoin du langage Ada de limiter la taille d'un tableau à l'instanciation.

Les variables sont également accessibles depuis la liste chaînée d'instructions, ce qui permet une interprétation plus simple du code lors de l'exécution.

Stratégie de test

La stratégie de tests déployée s'est concentrée sur l'utilisation d'une suite de tests unitaires afin d'assurer le test Rouge Vert et d'un fichier de tests fonctionnel par module afin d'éviter la régression et d'assurer la solidité du code.

Dans un souci de simplicité, les tests unitaires et fonctionnels ont été compilés dans un seul fichier par module développé, et les modules autres qu'operateurs ont été testés dans le fichier test_intermediaire.adb.

Afin de s'assurer de la non-régression des tests, nous avons utilisés un pipeline sur GitLab afin d'automatiser les tests.

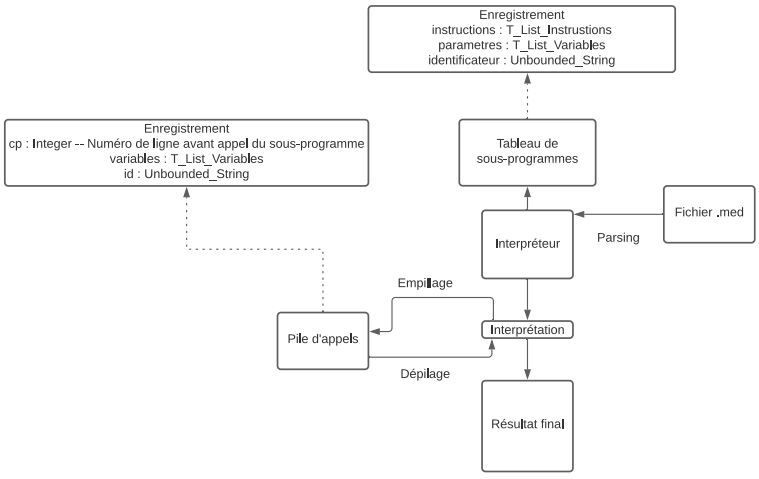
Pour tester le comportement de notre application, nous avons également écrits plusieurs programmes de tests en langage intermédiaire couvrant la totalité des fonctionnalités attendues et des erreurs levées/corrigées lors de l'exécution.

Difficultés rencontrées et solutions considérées

L'une des premières difficultés rencontrées a été le traitement des caractères dans la structure des variables définie dans la première partie du projet. En effet, la valeur d'une variable ne peut recevoir qu'une valeur du type Integer d'Ada, ce qui empêchait notre idée initiale de pouvoir stocker un caractère de manière conviviale dans le tas mémoire des variables. L'une des solutions envisagées a été de donner le type des valeurs en tant que paramètre de généricité mais cela forçait l'instanciation d'un tas mémoire pour chaque type pouvant être utilisé dans le langage intermédiaire. La solution finale a été d'utiliser la position ASCII d'un caractère, valeur de type Integer dans le langage Ada, et l'attribut de type afin de déterminer si la variable manipulée est un entier ou un caractère. L'intérêt de cette solution repose sur l'existence en Ada d'un cercle commutatif sur le transtypage Entier Caractère via les opérations Val et Pos du type Character.

La seconde difficulté de conception rencontrée fut le traitement des tableaux dans la seconde partie. En effet, le seul type utilisable dans une variable est un entier, et contrairement aux caractères, Ada ne permet pas de convertir un tableau en entier. L'idée initiale fut de définir une opération bijective permettant un « transtypage » d'un entier vers un tableau mais en raison de l'absence d'une application mathématique permettant cette opération, la décision fut prise de considérer les tableaux de la même façon qu'Ada, c'est-à-dire de taille fixée dans la déclaration. Cela permet de dérouler le tableau case par case dans le tas mémoire, en utilisant l'attribut de nommage de la variable en tant qu'identificateur du tableau et de l'index, via une notation au format NomTableau[IndexTableau].

Le besoin de disposer de la possibilité de déclarer des sous-programmes est également une difficulté rencontrée lors de la réalisation de ce projet. Le choix a été fait de réutiliser l'architecture d'un programme main utilisée dans le projet en parsant les sous-programmes avant exécution. Lors de l'exécution, les appels aux sous-programmes empileraient le compteur programme et les sous-programmes modifieraient la mémoire globale du programme.



Organisation opérationnelle

Lors du pilotage de ce projet, l'attribution des tasks s'est faite de manière agile selon l'avancement du projet et des intérêts de chacun des membres de l'équipe.

Kilyan s'est chargé de définir les spécifications et de réaliser les raffinages des méthodes principales, ainsi que de la réalisation des programmes de tests unitaires et fonctionnels et des fichiers en langage source/intermédiaire pour la première et deuxième partie.

Etienne s'est chargé de définir les structures de données adaptées à chaque utilisation ainsi que de développer les algorithmes de parsing, d'interprétation et d'assainissement des variables et des instructions.

Bilan technique

Après réalisation de ce projet, la connaissance de l'équipe dans le langage Ada ainsi qu'en solution logicielle se trouve considérablement augmentée. Les questions considérées et l'objectif du projet sortant du scope habituel des membres de l'équipe, plus habitués à des langages de plus haut niveau et à mémoire dynamique, les conventions de codage acquises par l'équipe sauront se révéler clés pour de futurs projets.

Bilan personnel de l'équipe

Etienne :

La réalisation de ce projet m'a permis de développer mes capacités de travail au sein d'une équipe ainsi que de l'utilisation de Git avec des fichiers régulièrement modifiés par toute l'équipe. L'utilisation d'un langage aussi restrictif qu'Ada m'a également permis de mettre en place des méthodes de programmation afin d'éviter de corriger constamment des erreurs de codes et de conception.

Kilyan :

Le sujet de ce projet permettant de réfléchir sur des problèmes de représentation des données et sur l'organisation d'un projet rapide en équipe réduite, cela m'a permis d'appréhender de nouvelles méthodes et de mettre en place les méthodes de programmation vue avec M. Ait-Ameur. De plus, l'écriture d'un interpréteur étant radicalement opposé à mon domaine habituel de la programmation full-stack, elle constitue une expérience enrichissante et ne pourrait être que bénéfique dans chaque réflexion future.