Université de Liège

INFO0946

Introduction à la Programmation

GameCode : Un exercice dont vous êtes le Héros · l'Héroïne

Fichiers Séquentiels – Concaténation de Fichiers

Benoit Donnet Simon Liénardy

23 février 2021

Préambule

Exercices

Dans ce GameCode, nous vous proposons de suivre pas à pas la résolution d'un exercice portant sur les fichiers.

Il est dangereux d'y aller seul [1] !

Partagez vos commentaires, questions, solutions alternatives sur le forum eCampus. N'hésitez jamais !

1. Référence vidéoludique bien connue des Héros.

3.1 Rappels sur les Fichiers Séquentiels

3.1.1 Généralités

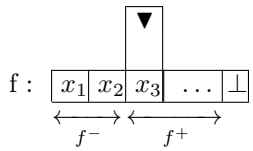


FIGURE 1 – Schéma d'un fichier séquentiel

On représente le fichier séquentiel en dessinant le ruban qui contient les données ainsi que la *tête de lecture/écriture*, symbolisée par le ▼. Un exemple de fichier est illustré par la Fig. 1.

En quoi cela diffère-t-il des tableaux ? On n'a accès qu'à un seul élément : celui sous la tête de lecture/écriture [2] (*élément courant*). Il faut de toute façon le lire pour avancer à l'élément suivant. On ne peut pas se déplacer de manière aléatoire dans le fichier (les ordinateurs en sont capables, évidemment, mais on n'envisage pas cette possibilité aujourd'hui). On peut tester si on est à la fin du fichier grâce à la valeur spéciale $\perp$ [3] qui représente le symbole `EOF` (*End Of File*).

3.1.2 Notations sur les Fichiers

Faites en sorte de bien vous **familiariser** avec ces notations ! Si vous ne les comprenez pas, relisez les slides (Chap. 3, Slides 12 → 19).

Notion	Notation
Fichier f	$f = < x_1, x_2, \ldots, x_n >$
Ens. de valeur	$x_i \in V$
Fichier vide	$f = <>$
Restriction de f à l'intervalle $[i \ldots j]$	f_i^j, vide si $j > i$
Concaténation de fichier	$f_1 \,\|\|\, f_2$
Minorant $(\forall i, 1 \leq i \leq n, a < x_i)$	$a < f$
Contient valeur $(\exists i, 1 \leq i \leq n, a = x_i)$	$a \in f$
Ne contient pas valeur $(\forall i, 1 \leq i \leq n, a \neq x_i)$	$a \notin f$
Égalité des élém. à a $(\forall i, 1 \leq i \leq n, a = x_i)$	$a = f$
¬ Égalité des élém. à a $(\exists i, 1 \leq i \leq n, a \neq x_i)$	$a \neq f$
Fin de fichier	$\perp$
Fichier déjà parcouru	f^-
Fichier encore à parcourir (compris `head` [4])	f^+

TABLE 1 – Résumé des notations pour les fichiers séquentiels.

3.1.3 Inventaire des Modules Disponibles

3.1.3.1 Ouverture (`fopen()`)

2. Si vous avez connu les VHS (cassettes vidéos), vous avez une idée pratique de ce dont on parle.
3. À prononcer « bottom »
4. `head` fait référence à l'élément courant (i.e., sous la tête de lecture/écriture).

```
/*
 * PréCondition : mode ∈ {"r", "w"} ∧ chemin valide
 * PostCondition : mode_0 = "r" ⇒ (¬eof(f) ∧ f⁻ =<> ∧ f⁺ = f)∨ mode_0 = "w" ⇒ (eof(f) ∧ f =<>)
 */
FILE fopen(char *chemin, char *mode);
```

Si le mode est `"r"` pour **R**ead, le fichier spécifié par le chemin est ouvert en lecture et la tête de lecture est positionnée sur le premier élément. Si le mode est `"w"` pour **W**rite, le fichier est ouvert en écriture et il est vide. S'il n'existe pas, il est créé avec le nom `chemin`.

Remarque Importante Ici, `FILE` est une variable de type fichier. En temps normal, ce serait un type opaque (c'est d'ailleurs le cas dans `stdio.h` et il faudrait passer un pointeur, `FILE *`). **On ne s'encombre pas de ces considérations dans ce GameCode**. On manipule `FILE` *comme si* c'était un type primitif. Consultez les slides (Chapitre 3, Slides 24, 29, 31, 36, 39) pour connaitre la traduction du pseudo-langage étudié ici et le C.

3.1.3.2 Fin de Fichier (`eof()`)

```
/*
 * PréCondition : /
 * PostCondition : eof(f) ⇔ f⁺ = < >
 */
bool eof(FILE f);
```

Le prédicat est vrai si et seulement si on a atteint la fin du fichier.

3.1.3.3 Lecture (`read()`)

```
/*
 * PréCondition : f⁻ = f_1^{i-1} ∧ f⁺ = fin ∧ ¬eof(f)
 * PostCondition : val = x_i ∧ f⁻ = f_1^i ∧ f⁺ = f_{i+1}^n
 */
void read(FILE f, type val);
```

L'élément qui est sous la tête de lecture (i.e., l'élément courant) est lu et placé dans `val`. La tête de lecture progresse ensuite vers l'élément suivant. [5]

3.1.3.4 Ecriture (`write()`)

```
/*
 * PréCondition : f =< x_1, x_2, ..., x_{i-1} > ∧ val = x_i ∧ eof(f)
 * PostCondition : f =< x_1, x_2, ..., x_{i-1} x_i > ∧ val = val_0 ∧ eof(f)
 */
void write(FILE f, type val);
```

La valeur `val` est écrite en fin de fichier. La tête d'écriture est toujours positionnée en fin de fichier suite à l'appel.

3.1.3.5 Fermeture du Fichier (`fclose()`)

```
void fclose(FILE f);
```

Ferme le fichier `f`.

5. Par rapport aux slides, la référence à `EOF` dans la PostCondition a été retirée parce que c'est vrai dans tous les cas.

3.1.4 Remarques Finales

- Quels fichiers doit-on fermer ?
  R : Tous ceux qui ont été ouverts avec `fopen()`.
- Même ceux ouverts en lecture ?
  R : Oui, c'est une habitude à prendre !
- Même si ça ne risque rien ?
  R : Oui, c'est une bonne habitude à prendre !
- Quel est le risque à ne pas fermer un fichier ?
  R : Les fonctions d'interactions avec le système de fichier, style `fprintf()` n'écrivent pas tout de suite dans le fichier lorsqu'elles sont invoquées mais écrivent d'abord dans une mémoire tampon (*buffer* en anglais). La fermeture du fichier force l'écriture du tampon dans le fichier. Un oubli de fermeture de fichier pourrait ne pas écrire le tampon et il pourrait s'ensuivre une perte de données.

3.2 Énoncé

Écrivez un module qui permet de concaténer deux fichiers contenant des valeurs entières dans un troisième.

3.2.1 Méthode de Résolution

On va procéder à la résolution de ce problème en suivant pas à pas l'approche constructive :

1. Définir de nouvelle·s notation·s (Sec. 3.3) ;
2. Analyse et découpe en SP (Sec. 3.4) ;
3. Spécifier le problème complètement (Sec. 3.5) ;
4. Chercher un Invariant de Boucle (Sec. 3.6) ;
5. Construire le programme complètement (Sec. 3.7) ;
6. Rédiger le programme final (Sec. 3.8).

3.3 Formalisation du Problème

La première étape dans la résolution du problème nécessite sa formalisation, ce qui passe par l'introduction de nouvelles notations.

Si vous voyez de quoi on parle, rendez-vous à la Section 3.3.3
Si vous ne savez comment introduire une notation, voyez la Section 3.3.1
Si vous ne voyez pas comment faire, reportez-vous à l'indice 3.3.2

3.3.1 Rappels sur la Formalisation d'un Problème

3.3.1.1 Généralités

Voici la forme que doit prendre une notation :

NomPredicat(Liste de parametres) ≡ Détail du predicat

NomPredicat est le nom du prédicat [a]. Il faut souvent trouver un nom en rapport avec ce que l'on fait. Mathématiquement parlant, cela ne changera rien de proposer un nom farfelu. Par exemple, on pourrait appeler le prédicat « Gims ». Cependant, pour des raisons de lisibilité, il est toujours préférable detrouver un nom en rapport avec la définition et le problème qu'on cherche à formaliser.

≡ est le symbole mathématique signifiant « identique à ». Il sert à introduire le détail du prédicat.

Liste de paramètres C'est une liste de symboles qui pourront être utilisés dans la définition du prédicat. Comment cela fonctionne-t-il ? Le prédicat peut être utilisé en écrivant son nom ainsi qu'une valeur particulière de paramètre. On remplace cette valeur dans la définition du prédicat et on regarde si c'est vrai ou faux.

Détail du prédicat C'est une combinaison de symboles logiques qui fait intervenir, le plus souvent, les paramètres. Si nécessaire, il est aussi possible d'utiliser des quantificateurs.

a. Pour rappel, un prédicat est une fonction mathématique produisant une valeur booléenne.

Exemple :

$$\text{Pair}(X) \equiv X \ \% \ 2 = 0$$

Le nom du prédicat est « Pair » et prend un argument. $Pair(4)$ remplace $4 \ \% \ 2 = 0$ qui est vrai et $Pair(5)$ signifie $5 \ \% \ 2 = 0$ qui est faux. Le choix du nom du prédicat est en référence à sa signification : il est vrai si X est pair.

▷ **Exercice** Si vous avez compris, vous pouvez définir facilement le prédicat Impair(X).

3.3.1.2 Opérateurs Logiques

Les opérateurs logiques usuels sont les suivants :

$True$ correspond la propriété "vrai" ;

$False$ correspond à la propriété "faux" ;

$\neg$ est l'opérateur de négation. Dès lors, $\neg p$ signifie "non p" ;

$\wedge$ est l'opérateur de conjonction. Dès lors, $p \wedge q$ signifie "p et q" ;

$\vee$ est l'opérateur de disjonction. Dès lors, $p \vee q$ signifie "p ou q" ;

$\Longrightarrow$ est l'opérateur d'implication. Dès lors, $p \implies q$ signifie que si p est vraie, alors q aussi. A noter aussi que si p est fausse, alors $p \implies q$ est vraie, quelque soit q (i.e., du faux on peut conclure ce qu'on veut).

Le Tableau 2 reprend la table de vérité pour les différents opérateurs logiques.
A noter que $p \iff q$ est équivalent à $(p \implies q) \wedge (q \implies p)$.
Enfin, la priorité des opérateurs logiques est la suivante ($<$ signifie "plus prioritaire") :

$$\neg \ < \ \wedge, \vee \ < \ \implies, \iff$$

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \implies q$	$p \iff q$
Vrai	Vrai	Faux	Vrai	Vrai	Vrai	Vrai
Vrai	Faux	Faux	Faux	Vrai	Faux	Faux
Faux	Vrai	Vrai	Faux	Vrai	Vrai	Faux
Faux	Faux	Vrai	Faux	Faux	Vrai	Vrai

TABLE 2 – Table de vérité pour les opérateurs logiques usuels.

3.3.1.3 Quantificateurs Logiques

Quantificateur Universel

Soient P et Q, deux prédicats, et (l) une liste (non vide) d'identificateurs.

$$\forall\ (l) \cdot (P \implies Q)$$

est un prédicat représentant la *quantification universelle*. La signification (en français) du prédicat est "pour tout i tel que P alors Q (est vrai)".

Dans le cadre du cours, nous allons utiliser la notation (relaxée) suivante :

$$\forall\ (l), P, Q$$

La quantification universelle peut être vue comme un *sucre syntaxique* simplifiant une suite de conjonctions. Par exemple, le prédicat :

$$\forall\ i, 0 \leq i \leq N-1, T[i] > 0$$

peut se comprendre comme suit :

$$T[0] > 0 \wedge T[1] > 0 \wedge\ \ldots\ \wedge T[N-1] > 0$$

Enfin, lorsque la variable prend ses valeurs sur l'ensemble vide, la quantification est vraie, quel que soit le terme P. Par exemple :

$$\forall\ i, 0 \leq i \leq -1, T[i] = 0 == True$$

Quantificateur Existentiel

Soient P et Q, deux prédicats, et (l) une liste (non vide) d'identificateurs.

$$\exists\ (l) \cdot (P \wedge Q)$$

est un prédicat représentant la *quantification existentielle*. La signification (en français) du prédicat est "il existe (au moins) un i tel que P pour lequel Q (est vrai)".

Dans le cadre du cours, nous allons utiliser la notation (relaxée) suivante :

$$\exists\ (l), P, Q$$

La quantification existentielle peut être vue comme un *sucre syntaxique* simplifiant une suite de disjonctions. Par exemple, le prédicat :

$$\exists\ i, 0 \leq i \leq N-1, T[i] > 0$$

peut se comprendre comme suit :

$$T[0] > 0 \vee T[1] > 0 \vee\ \ldots\ \vee T[N-1] > 0$$

3.3.1.4 Quantificateurs Numériques

Somme

La somme d'une suite de termes est représentée à l'aide de la notation Σ.
Ainsi, par exemple :

$$0 + 1 + 2 + \ \ldots \ + N - 1$$

sera représenté de manière beaucoup plus condensée par

$$\sum_{i=0}^{N-1} i$$

Par définition, la somme sur un intervalle vide vaudra toujours le neutre de l'addition, soit 0 (i.e., somme à zéro terme). Soit :

$$\sum_{i=0}^{-1} i = 0$$

Produit

Le produit d'une suite de termes est représentée à l'aide de la notation Π.
Ainsi, par exemple :

$$T[0] \times T[1] \times T[2] \times \ \ldots \ \times T[N-1]$$

sera représenté de manière beaucoup plus condensée par

$$\prod_{i=0}^{N-1} T[i]$$

Par définition, le produit sur un intervalle vide vaudra toujours le neutre de la multiplication, soit 1 (i.e., produit à zéro terme)

$$\prod_{i=0}^{-1} T[i] = 1$$

Minimum/Maximum

Le minimum (respectivement maximum) d'une suite de termes est représenté à l'aide la notation min (respectivement max).

$$min_P Q$$

où P et Q sont des prédicats

Par exemple :

$$minimum\{T[0], T[1], \ldots, T[N-1]\}$$

sera adéquatement représenté par :

$$min_{0 \leq i \leq N-1} \ (T[i])$$

Dénombrement

Soient P et Q, deux prédicats, et i une variable libre [a]

$$\#i \cdot (P|Q)$$

est un prédicat représentant le *quantificateur de dénombrement*. La signification (en français) du prédicat est "le nombre de fois où le prédicat Q est vrai lorsque P (est vrai)".

a. Pour rappel, une variable est dite *libre* quand le nom que l'on utilise pour définir un objet a de l'importance (il s'agit typiquement d'une variable non introduite par un quantificateur). A l'inverse, une variable est dite *liée* si le nom que l'on utilise pour définir un objet n'a pas d'importance (il s'agit typiquement d'une variable introduite par une quantificateur).

Par exemple, un prédicat indiquant le nombre de valeurs à 0 dans un tableau `T` à `N` valeurs entières se présente comme suit :

$$\#i \cdot (0 \leq i \leq N-1|T[i] = 0)$$

Suite de l'Exercice

À vous ! Formalisez le problème et passez à la Sec. 3.3.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 3.3.2

3.3.2 Indice

Pour formaliser un problème, il faut répondre aux deux questions suivantes :

- De quoi parle le Problème ?
- Quel est le Sujet ?

La réponse devrait mettre en évidence les éléments essentiels à la résolution du problème.
Ensuite, une bonne idée est d'illustrer le fonctionnement de la notation sur un dessin, qui sera ensuite traduit en prédicat.

Suite de l'Exercice

À vous ! Définissez le problème et passez à la Sec. 3.3.3.

3.3.3 Mise en Commun de la Formalisation du Problème

En relisant l'énoncé, on a :

> Écrivez une fonction qui retourne la concaténation de deux fichiers.

Avons-nous besoin d'une nouvelle notation ? Non, il suffit relire les notations sur les fichiers pour s'en convaincre.

La concaténation de fichiers est déjà définie :

$$f_{\text{concat}} = f_1 \,||\, f_2$$

f_{concat} est la concaténation de f_1 et f_2

Alerte : Studentite aigüe

Si vous aviez d'autres notations, vous êtes atteint de studentite aigüe ! Vous cherchez midi à quatorze heures et vous vous complaisez dans une solution trop complexe. Reposez-vous et passez votre tour ! Éloignez-vous de votre feuille d'exercice pendant 15 minutes [a]. Reprenez ensuite le travail.

a. Sérieux, faites-le.

Suite de l'exercice

On peut maintenant passer à l'analyse du problème.

3.4 Découpe en Sp

La deuxième étape de la résolution du problème est l'*analyse du problème*, nous amenant à une découpe en Sp.

Si vous voyez de quoi on parle, rendez-vous à la Section 3.4.3
Si vous ne savez pas ce qu'est l'analyse du problème, allez à la Section 3.4.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 3.4.2

3.4.1 Rappels sur l'Analyse

L'étape d'*analyse* permet de réfléchir à la structuration du code en appliquant une *approche systémique*, i.e., la découpe du problème principal en différents sous-problèmes (SP) et la façon dont les SP interagissent les uns avec les autres. C'est cette interaction entre les SP qui permet la structure du code.

Un SP correspond à une tâche particulière que le programme devra effectuer dans l'optique d'une résolution complète du problème principal. Si la tâche correspond à un module, il faudra le spécifier. Il est impératif que chaque SP dispose d'un nom (pertinent) ou d'une courte description de la tâche qu'il effectue.

Alerte : Microscopisme

Inutile de tomber, ici, dans une découpe en SP trop fine (ou *microscopique*). Le bon niveau de granularité, pour un SP, est la boucle ou un *module*.
Une erreur classique est de considérer la déclaration des variables comme un SP à part entière.

L'agencement des différents SP doit permettre de résoudre le problème principal, i.e., à la fin du dernier SP, la PostCondition du problème doit être atteint. De même, le premier SP doit s'appuyer sur les informations fournies par la PréCondition. On envisage deux formes d'agencement des SP :

1. *Linéraire* : $\text{SP}_i \rightarrow \text{SP}_j$. Dans ce cas, le SP_j est exécuté après le SP_i. Typiquement, la PostCondition du SP_i sert de PréCondition au SP_j.
2. *Inclusion* : $\text{SP}_j \subset \text{SP}_i$. L'exécution du SP_j est incluse dans celle du SP_i. Cela signifie que le SP_j est invoqué plusieurs fois dans le SP_i. Typiquement, le SP_j est un traitement exécuté lors de chaque itération du SP_i. Par exemple, le SP_i est une boucle et, son corps, comprend une exécution du SP_j (qui lui aussi peut être une boucle).

Attention, les deux agencements ne sont pas exclusifs. On peut voir apparaître les différents agencements au sein d'un même problème.

Suite de l'Exercice

À vous ! Analysez le problème et passez à la Sec. 3.4.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 3.4.2

3.4.2 Indice

Pour découper le problème principal, il convient de repartir de l'énoncé

Vous devez, bien entendu, veiller à ce que votre découpe soit pertinente : ni trop haut niveau (e.g., SP_1 : résoudre le problème), ni trop bas niveau (e.g., SP_1 déclarer des variables). Dans tous les cas, après exécution du dernier SP, le problème général doit être résolu (i.e., le tableau est trié par ordre croissant).

Suite de l'Exercice

À vous ! Définissez le problème et passez à la Sec. 3.3.3.

3.4.3 Mise en Commun de l'Analyse du Problème

Concaténer deux fichiers dans un fichier résultat consiste :

1. À recopier f_1 dans f_{concat}
2. À recopier f_2 dans f_{concat}, à la suite de f_1

Les deux opérations sont exactement les mêmes ! On peut donc les externaliser dans un SP.
On aurait donc deux SP :

SP$_0$. C'est le problème principal.

SP$_1$. Ce SP ajoute en fin de fichier le contenu d'un autre fichier.

Le gros du travail se fera dans le SP$_1$. Le SP$_0$ se contentera d'invoquer deux fois le SP$_1$: une première fois pour recopier f_1 dans f_{concat}, une deuxième fois pour recopier f_2 à la suite de de f_1 dans f_{concat}.

Suite de l'exercice

On peut maintenant passer à la spécification du problème.

3.5 Spécifications

Il s'agit, maintenant, de proposer une interface pour le module.

Si vous voyez de quoi on parle, rendez-vous à la Section 3.5.3
Si vous ne savez pas ce qu'est une spécification, allez à la Section 3.5.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 3.5.2

3.5.1 Rappels sur les Spécifications

3.5.1.1 Généralités

Une spécification se définit en deux temps :

La PréCondition implémente les suppositions. Elle caractérise donc les conditions initiales du module, i.e., les propriétés que doivent respecter les valeurs en entrée (i.e., paramètres effectifs) du module. Elle se définit donc sur les paramètres formels (qui seront initialisés avec les paramètres effectifs). La PréCondition doit être satisfaite avant l'exécution du module.

La PostCondition implémente les certifications. Elle caractérise les conditions finales du résultat du module. Dit autrement, la PostCondition décrit le résultat du module sans dire comment il a été obtenu. La PostCondition sera satisfaite après l'invocation.

A l'instar de la définition d'un problème, un module doit toujours avoir une PostCondition (un module qui ne fait rien ne sert à rien) mais il est possible que le module n'ait pas de PréCondition (e.g., le module ne prend pas de paramètres formels).

La spécification d'un module se met, en commentaires, avec le prototype du module. L'ensemble (i.e., spécification + prototype) forme l'*interface* du module.

Contrairement au cours INFO0946, les spécifications d'un module seront, dorénavant, exprimées de manière formelle (i.e., à l'aide de prédicats et de notations).

3.5.1.2 Construction d'une Interface

Pour construire l'interface d'un module, il est souhaitable de commencer par faire un dessin. Le dessin, qui naturellement sera lié à Invariant Graphique, doit représenter des situations particulières du problème à résoudre. La situation initiale (i.e., avant la première instruction de la Zone 1/Init) doit aider pour trouver le prototype du module, ainsi que la PréCondition. La situation finale (i.e., après exécution de la Zone 3/End) doit vous aider à trouver la PostCondition.

En s'appuyant sur les dessins, il suffit ensuite de répondre à trois questions pour construire l'interface d'un module :

Question 1 : Quels sont les objets dont j'ai besoin pour atteindre mon objectif ? Répondre à cette question permet de construire le prototype du module (éventuel type de retour, identificateur du module, paramètres formels).

Question 2 : Quel est l'objectif du module ? Répondre à cette question permet d'obtenir la PostCondition, représentée à l'aide d'un prédicat.

Question 3 : Quelles sont les contraintes sur les paramètres ? Répondre à cette question permet d'obtenir la PréCondition, représentée à l'aide d'un prédicat.

L'avantage de faire un dessin est, bien entendu, de dresser un pont avec l'Invariant Graphique, mais aussi de faciliter la production d'un prédicat (il "suffit" de traduire le dessin en un prédicat).

3.5.1.3 Exemple

A titre d'exemple, essayons de spécifier une fonction qui permet de calculer le produit de tous les entiers entre des bornes fournies, i.e., `a` et `b` (avec $b > a$).

Le problème peut s'illustrer comme indiqué à la Fig. 2. On dessine une droite (représentant les nombres qu'on va manipuler) avec les bornes de l'intervalle d'intérêt. La flèche bleue indique ce qu'on doit calculer.

Question 1
Pour atteindre l'objectif, nous avons simplement besoin des bornes de l'intervalle. Sur la Fig. 2, on peut clairement voir que le calcul du produit s'étale entre `a` et `b`. Ce sont les seuls éléments "extérieurs" dont on a besoin pour résoudre le problème. `a` et `b` seront donc les paramètres formels du module.

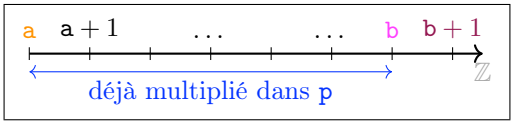


FIGURE 2 – Exemple de représentation graphique du problème à résoudre.

La Fig. 2 illustre aussi le fait qu'on manipule des entiers ($\mathbb{Z}$). a et b seront donc de type `int`.

Enfin, on peut adéquatement nommer le module `produit()`, ce qui permet de naturellement décrire ce qu'il fait.

Tout ceci nous permet de dériver le prototype du module :

```
1 produit(int a, int b);
```

Question 2

L'objectif du module est de calculer (et donc retourner) le produit de tous les entiers dans l'intervalle $[a, b]$. Soit :

$$a \times (a+1) \times \cdots \times (b-1) \times b$$

On peut représenter ce produit de manière plus condensée en utilisant le quantificateur numérique du produit : $\prod$. Soit :

$$\prod_{i=a}^{b} i$$

Puisque a et b sont de type `int`, le produit résultat sera aussi de type `int`. Le module `produit()` est donc une fonction retournant un `int`. On représente cela de la façon suivante :

$$produit = \prod_{i=a}^{b} i$$

Enfin, `a` et `b` ne sont pas modifiés par le module.

A ce stade, on obtient donc l'interface incomplète suivante :

```
1 /*
2  * PostCondition : produit = ∏_{i=a}^{b} i ∧ a = a_0 ∧ b = b_0
3  */
4 int produit(int a, int b);
```

Question 3

Enfin, nous pouvons déterminer si il existe des conditions sur les paramètres. C'est bien le cas ici. L'énoncé du problème et la Fig. 2 indiquent clairement que b est strictement plus grand que a. On peut le représenter de la façon suivante :

$$b > a$$

Interface
Au final, l'interface du module est la suivante :

```
/*
 * PréCondition : b > a
 * PostCondition : produit = ∏_{i=a}^{b} i ∧ a = a_0 ∧ b = b_0
 */
int produit(int a, int b);
```

Suite de l'Exercice

À vous ! Proposez les interfaces pour les Sp et passez à la Sec. 3.5.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 3.5.2

3.5.2 Indice

Pour le module, répondez aux trois questions en vous appuyant sur un schéma. Le schéma permettra d'aisément traduire la situation particulière (PréCondition ou PostCondition) sous forme formelle.

Ne soyez pas ambigu dans la PréCondition, ni dans la PostCondition. Au contraire, soyez le plus précis possible (rappelez-vous, un prédicat est une formulation mathématique et, dès lors, ne peut souffrir d'une quelconque ambiguïté).

Une fois que c'est fait, rédigez l'interface du module.

Suite de l'Exercice

À vous ! Spécifiez les Sp et passez à la Sec. 3.5.3.

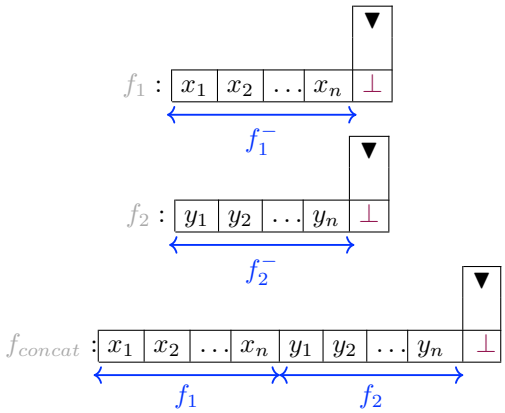
3.5.3 Mise en Commun des Spécifications

Il faut maintenant donner l'interface (i.e., spécifications + prototype) de chacun des SP.

- spécification du SP_0 ;
- spécification du SP_1 ;

3.5.3.1 Spécification du SP_0

On peut représenter le module de la façon suivante :



En entrée, on veut que les chaînes de caractères indiquant les noms des trois fichiers (les deux fichers à concaténer et le fichier résultat) soient initialisés. Soit :

$$\text{PréCondition} \equiv nom\_fichier1\ init \wedge nom\_fichier2\ init \wedge nom\_concat\ init$$

L'objectif du module est de concaténer les deux fichiers, comme décrit dans le schéma ci-dessus. Dans cette situation, f_{concat} contient la concaténation des deux fichiers f_1 et f_2. Ces fichiers ne sont pas modifiés. On peut appeler le module `fconcat()`. On obtient donc :

$$\text{PostCondition} \equiv f_{concat} = f_1 \,||\, f_2 \wedge f_1 = f_{1_0} \wedge f_2 = f_{2_0}$$

Au final, l'interface de la procédure est :

```
1 /*
2  * PréCondition : nom_fichier1 init ∧ nom_fichier2 init ∧ nom_concat init
3  * PostCondition : f_concat = f1 || f2 ∧ f1 = f1_0 ∧ f2 = f2_0
4  */
5 void fconcat(char *nom_fichier1, char *nom_fichier2, char* nom_concat);
```

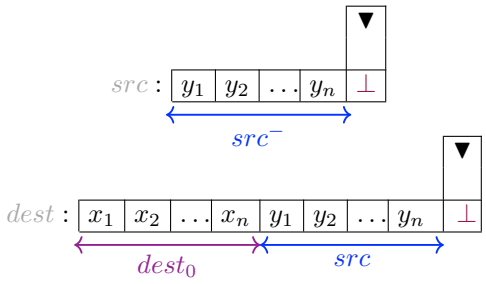
Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- spécification du SP_1 ;
- Invariants de Boucle des modules.

3.5.3.2 Spécification du SP_1

On peut représenter le module de la façon suivante :



`src` est le fichier source à recopier à la fin de `dest`. En entrée, on a donc besoin de deux fichiers déjà ouverts : `src` (en lecture) et `dest` en écriture. On ajoute toujours à la fin de `dest`, donc la tête de lecture/écriture doit forcément se trouver sur $\perp$.

On obtient donc la PréCondition suivante :

$$\text{PréCondition} \equiv dest^+ = <> \wedge \text{mode}(src) = "r" \wedge \text{mode}(dest) = "w"$$

La notation mode$(\cdot)$ indique le mode d'ouverture du fichier en argument (i.e., soit "r", soit "w").

L'objectif du SP_1 est de recopier (ou de *concaténer*) `src` dans `dest`. Nous pouvons donc appeler le module `fappend()`.[6] Le module ne retourne rien. C'est donc une procédure. L'objectif du module est clairement décrit par le schéma ci-dessus : il faut concaténer `src` à `dest`, sans modifier le contenu initial de `dest`. Ceci nous donne la PostCondition suivante :

$$\text{PostCondition} \equiv dest = dest_0 \wedge dest = dest_0 \,||\, src$$

Au final, l'interface de la procédure est :

```
/*
 * PréCondition : dest+ = <> ∧ mode(src) = "r" ∧ mode(dest) = "w"
 * PostCondition : dest = dest0 ∧ dest = dest0 || src
 */
void fappend(FILE src, FILE dest);
```

Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- spécification du SP_0 ;
- Invariants de Boucle des modules.

6. *to append* signifie, en français, *ajouter*.

3.6 Invariants de Boucle

L'écriture de code impliquant un traitement itératif (i.e., boucle) nécessite au préalable la proposition d'un Invariant de Boucle.

Si vous voyez de quoi on parle, rendez-vous à la Section 3.6.3
Si vous ne savez pas ce qu'est un Invariant de Boucle, allez la Section 3.6.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 3.6.2

3.6.1 Rappels sur l'Invariant de Boucle

3.6.1.1 Généralités

Un Invariant de Boucle est une **propriété** vérifiée à chaque évaluation du Gardien de Boucle.
L'Invariant de Boucle présente un **résumé** de tout ce qui a déjà été calculé, jusqu'à maintenant (i.e., jusqu'à l'évaluation courante du Gardien de Boucle), par la boucle.

Le fait que l'Invariant de Boucle soit une propriété est important : ce n'est pas du code, ce n'est pas exclusivement destiné au langage C. Tout programme qui inclut une boucle doit s'appuyer sur un Invariant de Boucle.

Alerte : Ce que l'Invariant de Boucle n'est pas

De manière générale, un Invariant de Boucle

- n'est pas une instruction exécutée par l'ordinateur ;
- n'est pas une directive comprise par le compilateur ;
- n'est pas une preuve de correction du programme ;
- est indépendant du Gardien de Boucle ;
- ne garantit pas que la boucle se termine [a] ;
- n'est pas une assurance de l'efficacité du programme.

a. Seule la Fonction de Terminaison vous permet de garantir la terminaison

3.6.1.2 Sémantique Formelle

La *sémantique* formelle de l'Invariant de Boucle est présentée ici :

```
1  // {PréCondition}
2  Init
3  // {Inv}
4  while(B){
5       // {Inv ∧ B}
6       Iter
7       // {Inv}
8  }
9  // {Inv ∧ ¬B}
10 End
11 // {PostCondition}
```

Init correspond à la Zone 1. Ce sont donc les instructions qui, partant de la PréCondition, doivent amener la boucle et, donc, l'Invariant de Boucle.

B correspond au Gardien de Boucle. B doit être vrai pour pouvoir entrer dans la boucle.

Iter correspond à la Zone 2. Il s'agit donc du Corps de la Boucle. Ce sont donc les instructions qui doivent être répétées pour faire avancer le problème. Dans Iter, le point de départ est la situation particulière $\{\text{Inv} \wedge B\}$ et l'objectif, après la dernière intstruction du Corps de la Boucle, est de restaurer l'Invariant de Boucle.

¬ B correspond au Critère d'Arrêt. Quand ¬ B est vérifié, alors on sort de la boucle.

End correspond à la Zone 3. Il s'agit des instructions qui, une fois la boucle terminée (i.e., $\{\text{Inv} \wedge \neg B\}$), amène la PostCondition.

3.6.1.3 Règles pour l'Élaboration d'un Invariant Graphique

Bien que le cours INFO0947 ait pour objectif de présenter la programmation sous l'angle de ses fondements mathématiques (et, donc, par conséquence un Invariant de Boucle est présenté sous la forme d'un prédicat), il est toujours souhaitable de passer par l'étape dessin (et donc, l'Invariant Graphique).

Nous rappelons, ici, les 7 règles pour l'élaboration d'un Invariant Graphique satisfaisant (en les illustrants avec la Fig. 3) :

Règle 1 : réaliser un dessin pertinent par rapport au problème (la droite des entiers dans la Fig. 3) et le nommer ($\mathbb{Z}$ dans la Fig. 3) ;

Règle 2 : placer sur le dessin les bornes de début et de fin (a, b, et b +1 dans la Fig. 3) ;

Règle 3 : placer une ligne de démarcation qui sépare ce qu'on a déjà calculé dans les itérations précédentes de ce qu'il reste encore à faire (la ligne rouge sur la Fig. 3). Si nécessaire, le dessin peut inclure plusieurs lignes de démarcation ;

Règle 4 : étiqueter proprement chaque ligne de démarcation avec, e.g., une variable (i sur la Fig. 3) ;

Règle 5 : décrire ce que les itérations précédentes ont déjà calculé. Ceci implique souvent d'introduire de nouvelles variables ("déjà multiplié dans p" dans la Fig. 3) ;

Règle 6 : identifier ce qu'il reste à faire dans les itérations suivantes ("à multiplier" dans la Fig. 3) ;

Règle 7 : Toutes les structures identifiées et variables sont présentes dans le code (a, b, i, et p sur la Fig. 3) [7].

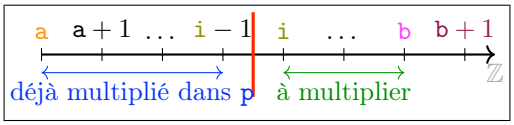


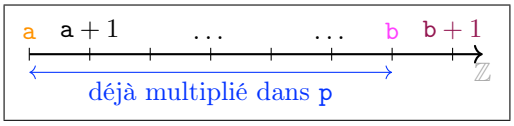
FIGURE 3 – Exemple d'Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

3.6.1.4 Comment Trouver un Invariant Graphique ?

C'est la grande question. On peut envisager plusieurs méthodes mais, dans le cadre d'un GAMECODE et du cours, il nous semble plus pertinent d'insister sur la méthode de *l'éclatement de la* POSTCONDITION. La technique est, finalement, assez simple :

1. repartir de la représentation graphique du problème et, en particulier, de la POSTCONDITION ;
2. appliquer les règles de conception d'un Invariant Graphique sur le dessin représentant la POSTCONDITION ;

Par exemple, l'Invariant Graphique du problème de calcul du produit des entiers entre deux bornes peut facilement être inféré de la représentation graphique du problème. Reprenons là ici :



On voit bien qu'on dispose, ainsi, d'une situation particulière (en général, la situation à la sortie de la boucle). Naturellement, le dessin s'accomode déjà de la Règle 1 et Règle 2. Placer une ligne de démarcation (Règle 3) naturellement rétrécit la zone bleue à une situation générale (Règle 5). Il suffit d'étiqueter correctement cette ligne de démarcation avec une variable (Règle 4) et, ensuite, de rajouter l'information sur la zone encore à traiter dans les itérations suivantes (Règle 6). L'Invariant Graphique ainsi produit est alors complet et cohérent avec les règles de conception. Il ne reste plus qu'à le transformer en Invariant Formel.

7. Plus de détails sur le lien entre l'Invariant de Boucle et la construction du code dans le rappel associé

3.6.1.5 De l'Invariant Graphique à l'Invariant Formel

A des fins d'exemple, traduisons l'Invariant Graphique donné à la Fig. 3 en un Invariant Formel. Un Invariant Graphique bien formé (i.e., qui respecte les 7 règles) contient toutes les informations nécessaires pour un Invariant Formel. Passer de l'un à l'autre est donc une simple quesiton de traduction d'un langage à l'autre (i.e., du dessin aux mathématiques).

$$\textsc{Inv} \equiv \mathtt{a} \leq \mathtt{i} \leq \mathtt{b} + 1 \tag{1}$$

$$\wedge\, \mathtt{p} = \prod_{j=a}^{\mathtt{i}-1} j \tag{2}$$

$$\wedge\, \mathtt{a} = \mathtt{a}_0 \tag{3}$$

$$\wedge\, \mathtt{b} = \mathtt{b}_0 \tag{4}$$

Ce qui signifie :

1. la variable de parcours, `i`, est dans l'intervalle d'intérêt ([`a`, `b`]). Cela correspond à la Règle 2 et la Règle 4 de l'Invariant Graphique ;
2. `p` contient le produit jusqu'à maintenant. Cela correspond à Règle 5. A noter que le produit se fait entre `a` et `i-1`, ce qui implicitement positionnne la variable de parcours, `i`, par rapport à la ligne de démarcation (Règle 3 et Règle 4) ;
3. `a` n'est pas modifié par le traitement itératif ;
4. `b` n'est pas modifié par le traitement itératif ;
5. les différentes parties de l'Invariant Formel sont connectées à l'aide de l'opérateur de conjonction ($\wedge$) ;
6. la partie encore à traiter (Règle 6) est implicite dans l'Invariant Formel ;
7. toutes les variables et structures identifiées dans l'Invariant Graphique sont bien présentes dans l'Invariant Formel (Règle 7).

Suite de l'Exercice

À vous ! Proposez les Invariants de Boucle pour les Sp et passez à la Sec. 3.6.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 3.6.2

3.6.2 Indice

Contrairement au cours INFO0946, les Invariants de Boucle sont ici proposés sous la forme de prédicats mathématiques. Il est néanmoins compliqué de proposer, directement, le prédicat.

L'approche la plus raisonnable reste donc de partir sur un Invariant Graphique et de le traduire, ensuite, sous la forme d'un Invariant Formel. Pour le Invariant Graphique, il suffit de repartir des règles définies dans le cours INFO0946 (en les adaptant aux fichiers – par exemple la ligne de démarcation sera représentée par la tête de lecture/écriture). Dans l'état actuel des développements, le Gli ne vous permet pas encore de dessiner un fichier séquentiel.

Un Invariant de Boucle décrit particulièrement ce que l'on a déjà fait au fil des itérations précédentes. Dans le cadre d'un fichier séquentiel, on possède une notation des éléments déjà vus : f^-. L'Invariant de Boucle parlera donc **quasi toujours** de f^-. Ce qu'il reste à calculer correspond au traitement à appliquer à f^+.

Une bonne technique consiste à retravailler la PostCondition (cfr. Chapitre 2, Slide 29) et d'y faire apparaitre f^-. Il faut ensuite se poser la question : « à l'itération i, de quelle·s information·s ai-je besoin pour traiter le i^{e} élément que je m'apprête à lire dans le fichier ? » (dans ce cas, on a évidemment $long(f^-) = i-1$).

Pensez à vérifier quel(s) Sp a besoin d'un Invariant de Boucle.

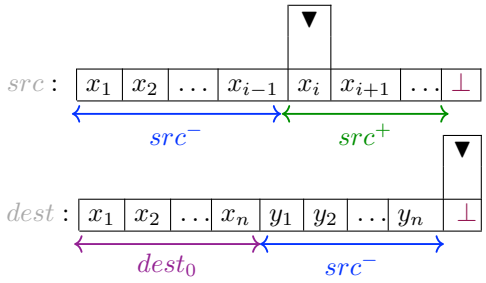
Suite de l'Exercice

À vous ! Proposez l'Invariant de Boucle et passez à la Sec. 3.6.3.

3.6.3 Mise en Commun des Invariants de Boucle

Nous avons deux Sp mais seul le Sp_1 nécessite un traitement itératif, et donc une construction basée sur l'Invariant de Boucle.

Commençons donc par proposer un Invariant Graphique pour le Sp_1 (construction basée sur l'éclatement de la PostCondition) :



On dessine deux fichiers. On indique que ce qu'on a déjà parcouru (lu) dans le fichier source *src* a été écrit dans le fichier destination *dest*. On est constamment en train d'écrire dans *dest*, donc on est toujours à la fin du fichier. On décrit la concaténation.

L'Invariant Formel pour le Sp_1 est naturellement dérivé de l'Invariant Graphique :

$$\text{Inv} \equiv dest = dest_0 \,||\, src^- \wedge eof(dest)$$

Suite de l'exercice

On peut maintenant passer à la construction du code.

3.7 Construction du Code

Une fois l'Invariant Formel proposé, on peut construire le code en adoptant l'approche constructive et on précisant, à chaque étape, les assertions intermédiaires.

Si vous voyez de quoi on parle, rendez-vous à la Section 3.7.3
Si vous ne savez pas comment fonctionne l'approche constructive, allez la Section 3.7.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 3.7.2

3.7.1 Rappels sur l'Approche Constructive

3.7.1.1 Généralités

Voici les différentes étapes de l'Approche Constructive.

INIT De la PréCondition, il faut arriver dans une situation où l'Invariant de Boucle est vrai.

B Il faut trouver d'abord le Critère d'Arrêt $\neg B$ et en déduire le Gardien de Boucle, B.

ITER Il faut faire progresser le Corps de la Boucle puis restaurer l'Invariant de Boucle.

END Vérifier si la PostCondition est atteinte si $\{Inv \wedge \neg B\}$, sinon, amener la PostCondition.

Fonction de Terminaison Il faut donner une Fonction de Terminaison pour montrer que la boucle se termine (cfr. INFO0946, Chapitre 3, Slides 97 → 107).

Tout ceci doit être justifié sur base de l'Invariant de Boucle, à l'aide d'assertions intermédiaire. Notez que l'ordre dans lequel vous faites ces étapes n'a pas d'importance !

On se retrouvera donc dans la situation suivante :

```
1  // {PréCondition}
2  Init
3  // {Inv}
4  while(B){
5      // {Inv ∧ B}
6      Iter
7      // {Inv}
8  }
9  // {Inv ∧ ¬B}
10 End
11 // {PostCondition}
```

Pour chaque étape, on démarre d'un point de départ et doit dériver les instructions (ainsi que les assertions intermédiaires) permettant d'arriver à l'objectif. Ainsi :

{PréCondition} Init {Inv} : Init est donc un bloc d'instructions pour lequel {PréCondition} est la précondition (i.e., point de départ) et {Inv} la postcondition (i.e., point d'arrivée). À nous de trouver comment arriver au but ({Inv}) en partant du point de départ ({PréCondition}). A noter que Init peut être éventuellement vide. Dans ce cas, il faut démontrer que {PréCondition} $\implies$ {Inv}.

{Inv $\wedge$ B} Iter {Inv} : Iter est donc un bloc d'instructions pour lequel $\{Inv \wedge B\}$ est la précondition (i.e., le point de départ) et {Inv} la postcondition (i.e., le point d'arrivée). À nous de trouver comment arriver au but ({Inv}) en partant du point de départ ({Inv $\wedge$ B}) tout en faisant avancer le problème. Il faut donc garantir qu'on a conservé notre Invariant de Boucle à la fin du Corps de la Boucle.

{Inv $\wedge \neg$B} End {PostCondition} : End est donc un bloc d'instructions pour lequel $\{Inv \wedge \neg B\}$ est la précondition (i.e., point de départ) et {PostCondition} la postcondition (i.e., le point d'arrivée). À nous de trouver comment arriver au but ({PostCondition}) en partant du point de départ ({Inv $\wedge$ $\neg B$}). A noter que End peut être éventuellement vide. Dans ce cas, il faut démontrer que $\{Inv \wedge \neg B\}$ $\implies$ {PostCondition}.

On voit donc bien que l'Invariant de Boucle est l'étape clé autour de laquelle s'articule non seulement la construction de la boucle mais aussi ce qui se passe avant et après la boucle. L'Invariant de Boucle est donc la colle entre les différentes parties du code.

Alerte : Oubli

Attention, pour chaque partie, on n'oublie pas d'indiquer les assertions intermédiaires. C'est obligatoire afin de s'assurer que les différentes parties s'enchaînent correctement.

3.7.1.2 Exemple

Afin d'illustrer l'approche constructive, travaillons sur un exemple. Il s'agit de rédiger une fonction qui permet de calculer le produit de tous les entiers entre des bornes fournies, i.e., `a` et `b` (avec $b > a$).

Pour rappel, l'interface de la fonction est la suivante :

```
/*
 * PréCondition : b > a
 * PostCondition : produit = ∏_{i=a}^{b} i ∧ a = a0 ∧ b = b0
 */
int produit(int a, int b);
```

L'Invariant Graphique est le suivant :

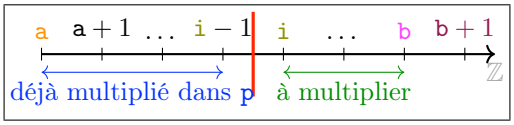


FIGURE 4 – Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

L'Invariant Formel est le suivant :

$$\begin{aligned} \text{INV} \equiv\ & \mathtt{a} \leq \mathtt{i} \leq \mathtt{b}+1 \\ & \wedge\, \mathtt{p} = \prod_{j=a}^{\mathtt{i}-1} j \\ & \wedge\, \mathtt{a} = \mathtt{a}_0 \\ & \wedge\, \mathtt{b} = \mathtt{b}_0 \end{aligned}$$

INIT

Sur base de l'Invariant Formel :

$$\text{INV} : \mathtt{a} \leq \mathtt{i} \leq \mathtt{b}+1 \wedge \mathtt{p} = \prod_{j=a}^{\mathtt{i}-1} j \wedge \mathtt{a} = \mathtt{a}_0 \wedge \mathtt{b} = \mathtt{b}_0$$

il faut rendre ceci vrai en partant de la PRÉCONDITION (i.e., $\mathtt{b} > \mathtt{a}$). Cela consiste donc à :

- déclarer les variables nécessaires. Soit `i` (la variable d'itération) et `p` (l'accumulateur pour le produit cumulatif) ;
- initialiser les variables. La valeur de `i` est donnée par $\mathtt{a} \leq \mathtt{i} \leq \mathtt{b}+1$. La valeur de `p` en découle naturellement.

On obtient donc la séquence suivante :

```
// {b > a}
int i, p;
// {i =?? ∧ p =?? ∧ a = a0 ∧ b = b0 ∧ b > a}
i = a;
// {i = a ∧ p =?? ∧ a = a0 ∧ b = b0 ∧ b > a}
// {a ≤ i ≤ b + 1 ∧ p =?? ∧ a = a0 ∧ b = b0}
p = 1;
// {a ≤ i ≤ b + 1 ∧ p = 1 ∧ a = a0 ∧ b = b0}
// {⇒ INV}
```

On peut écrire la ligne 6 car, par la PréCondition, $\texttt{b} > \texttt{a}$. Soit $\texttt{a} \leq \texttt{b} + 1$ (les deux expressions sont identiques).

La transition ligne 8 → ligne 9 est permise car

$\texttt{p} = 1$

$\texttt{p} = \prod_{j=\texttt{a}}^{\texttt{i}-1}$

`i` est initialisé à `a` (ligne 4), ce qui donne $\texttt{p} = \prod_{j=\texttt{a}}^{\texttt{a}-1}$. Soit un produit à 0 termes. Par définition, `p` vaut le neutre de la multiplication, soit 1.

B et Fonction de Terminaison

Le Critère d'Arrêt (¬ B) est donné par le schéma suivant :

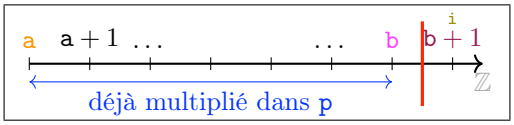


Figure 5 – Exemple d'Invariant Graphique pour le calcul du produit des entiers entre deux bornes.

C'est confirmé par la partie suivante de l'Invariant Formel :

$\texttt{a} \leq \texttt{i} \leq \texttt{b} + 1$

On a donc :

$\neg B \equiv \texttt{i} = \texttt{b} + 1$

On en dérive naturellement le Gardien de Boucle (B) :

$B \equiv \texttt{i} \leq b$

Pour la Fonction de Terminaison, il suffit d'estimer la taille de la zone "encore à traiter" dans l'Invariant Graphique. Soit : $(b - a + 1) - (i - a)$. Ce qui donne :

$f \equiv b - i + 1$

Iter

```
1 while(i <= b){
2   // {Inv ∧ B ≡ a ≤ i ≤ b + 1 ∧ p = ∏_{j=a}^{i-1} j ∧ a = a0 ∧ b = b0 ∧ i <= b}
3   // {a ≤ i ≤ b ∧ p = ∏_{j=a}^{i-1} j ∧ a = a0 ∧ b = b0}
4   p *= i;
5   // {a ≤ i ≤ b ∧ p = [(∏_{j=a}^{i-1} j) × i] ∧ a = a0 ∧ b = b0}
6   // {a ≤ i ≤ b ∧ p = ∏_{j=a}^{i} j ∧ a = a0 ∧ b = b0}
```

```
7     i++;
8     // {Inv}
9   }
```

La transition ligne 2 → ligne 3 est naturelle dans le sens où il s'agit de réécrire le prédicat (ligne 3) de sorte qu'il intègre complètement $\{\text{Inv} \wedge B\}$ (ligne 2).

Cette situation de départ nous indique qu'il faut faire progresser le problème, soit accumuler encore une valeur dans le produit cumulatif (ligne 4). Cette instruction nous conduit à la situation décrite à la ligne 6 (après réécriture du prédicat à la ligne 5).

Pour restaurer l'Invariant Formel (qui est l'objectif à atteindre pour Iter), il suffit d'incrémeter `i` d'un unité, ce qui permettra de restaurer $\mathtt{a} \leq \mathtt{i} \leq \mathtt{b}+1$ et $\mathtt{p} = \prod_{j=a}^{\mathtt{i}-1} j$.

End

À la sortie de la boucle, on est dans la situation $\{\text{Inv} \wedge \neg B\}$. Soit :

$$\mathtt{a} \leq \mathtt{i} \leq \mathtt{b}+1 \wedge \mathtt{p} = \prod_{j=a}^{\mathtt{i}-1} j \wedge \mathtt{a} = \mathtt{a}_0 \wedge \mathtt{b} = \mathtt{b}_0 \wedge \mathtt{i} = \mathtt{b}+1$$

On voit que la variable d'itération (`i`) prend une valeur particulière dans l'intervalle $\mathtt{a} \leq \mathtt{i} \leq \mathtt{b}+1 : \mathtt{b}+1$. On peut donc réécrire le prédicat en remplaçant la variable `i` par sa valeur particulière `b` $+1$. Soit :

$$\mathtt{p} = \prod_{j=a}^{\mathtt{b}+1-1} j \wedge \mathtt{a} = \mathtt{a}_0 \wedge \mathtt{b} = \mathtt{b}_0$$

Et encore :

$$\mathtt{p} = \prod_{j=a}^{\mathtt{b}} j \wedge \mathtt{a} = \mathtt{a}_0 \wedge \mathtt{b} = \mathtt{b}_0$$

Cette situation correspond à la PostCondition, à la différence près que `p` contient le produit cumulatif au lieu de la fonction `produit()` ! Dès lors, à la sortie de la boucle, il nous suffit de retourner la valeur de `p`.

Ce qui nous donne :

```
1  // {Inv ∧ ¬B}
2  return p;
3  // {PostCondition}
```

Code Complet

```
1  int produit(int a, int b){
2    // {PréCondition}
3    int i, p;
4    i = a;
5    p = 1;
6    // {Inv}
7    while(i <= b){
8      p *= i;
9      i++;
10   }//fin while - i
11
12   return p;
13   // {PostCondition}
14 }//fin produit()
```

Suite de l'Exercice

À vous ! Construisez le code pour les SP et passez à la Sec. 3.7.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 3.7.2

3.7.2 Indice

L'ordre de résolution des Sp n'a pas d'importance. Pour chaque Sp, il faut cependant construire le code en suivant :

Init De la PréCondition, il faut arriver dans une situation où l'Invariant de Boucle est vrai.

B Il faut trouver d'abord le Critère d'Arrêt $\neg B$ et en déduire le Gardien de Boucle.

Iter Il faut faire progresser le programme puis restaurer l'Invariant de Boucle.

End Vérifier si la PostCondition est atteinte si $\{\text{Inv} \wedge \neg B\}$, sinon, amener la PostCondition.

Fonction de Terminaison Il faut donner une Fonction de Terminaison pour montrer que la boucle se termine.

Tout ceci doit être justifié sur base de l'Invariant de Boucle, à l'aide d'assertions intermédiaire. Notez que l'ordre dans lequel vous faites ces étapes n'a pas d'importance !

Suite de l'Exercice

À vous ! Construisez le code pour chaque Sp et passez à la Sec. 3.7.3.

3.7.3 Mise en Commun de l'Approche Constructive

On peut construire le code de chaque SP.

- Construction du SP_0 ;
- Construction du SP_1.

3.7.3.1 Construction de `fconcat()` (SP_0)

Pour rappel, l'interface de la procédure est la suivante :

```
/*
 * PréCondition : nom_fichier1 init ∧ nom_fichier2 init ∧ nom_concat init
 * PostCondition : f_concat = f1 || f2 ∧ f1 = f1_0 ∧ f2 = f2_0
 */
void fconcat(char *nom_fichier1, char *nom_fichier2, char* nom_concat);
```

Le SP_0 n'a pas de traitement itératif, donc pas d'Invariant de Boucle. Ceci n'empêche nullement d'appliquer l'approche constructive, sur une "simple" suite d'instructions séquentielles. Cela signifie qu'on part de la PréCondition avec, comme objectif, d'atteindre la PostCondition. $\{\text{PréCondition}_{fappend}\}$ et $\{\text{PostCondition}_{fappend}\}$ sont respectivement la PréCondition et la PostCondition de `fappend()`.

Soit :

```
1  void fconcat(char *nom_fichier1, char *nom_fichier2, char *nom_concat){
2    // {PréCondition}
3    FILE f1 = fopen(nom_fichier1, "r");
4    // {nom_fichier2 init ∧ nom_concat init ∧ mode(f1) = "r"}
5    FILE f2 = fopen(nom_fichier2, "r");
6    // {nom_concat init ∧ mode(f1) = "r" ∧ mode(f2) = "r"}
7    FILE f_concat = fopen(nom_concat, "w");
8    // {mode(f1) = "r" ∧ mode(f2) = "r" ∧ mode(f_concat) = "w"}
9    // {f_concat = < > ∧ eof(f_concat) ∧ f1 = f1+ ∧ f2 = f2+ ∧ mode(f1) = "r" ∧ mode(f2) = "r
       ∧ mode(f_concat) = "w"}
10   // {⇒ PréCondition_fappend}
11   fappend(f1, f_concat);
12   // {PostCondition_fappend : f_concat = f_concat_0 || f1 = < > || f1 = f1 ∧ eof(f_concat) ∧ f1 = f1_0}
13   // {⇒ PréCondition_fappend}
14   fappend(f2, f_concat);
15   // {PostCondition_fappend : f_concat = f_concat_0 || f2 = f1 || f2 ∧ f1 = f1_0 ∧ f2 = f2_0}
16   fclose(f1);
17   fclose(f2);
18   fclose(f_concat)
19   // {PostCondition}
20 }
```

Le passage de la ligne 9 à la ligne 10 est valide car la PréCondition du SP_1 est la suivante :

$$dest^+ = < > \wedge \text{mode}(src) = "r" \wedge \text{mode}(dest) = "w"$$

La situation à la ligne 9 englobe bien la PréCondition du SP_1. On peut donc l'invoquer. Après l'invocation, on adapte la PostCondition du SP_1 à ses paramètres effectifs (`f_1` et `f_concat`). Cette situation particulière correspond parfaitement à la PréCondition du SP_1 qu'on peut, de nouveau, invoquer. A nouveau, il faut adapter la PostCondition du SP_1 à ses paramètres effectifs (`f_2` et `f_concat`).

Partant de là (ligne 15), nous avons atteint la PostCondition du SP_0. Il nous reste donc à fermer proprement les fichiers.

Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- construction du SP_1 ;
- mise en commun du code.

3.7.3.2 Construction de `fappend()` (SP_1)

Pour rappel, l'interface de la procédure est la suivante :

```
/*
 * PréCondition : dest+ = < > ∧ mode(src) = "r" ∧ mode(dest) = "w"
 * PostCondition : dest = dest0 ∧ dest = dest0 || src
 */
void fappend(FILE src, FILE dest);
```

L'Invariant Formel est le suivant :

$$\text{Inv} \equiv dest = dest_0 \,||\, src^- \wedge eof(dest)$$

Init

Sur base de l'Invariant Formel :

$$\text{Inv} \equiv dest = dest_0 \,||\, src^- \wedge eof(dest)$$

il faut rendre ceci vrai en partant de la PréCondition. Cela consiste donc à :

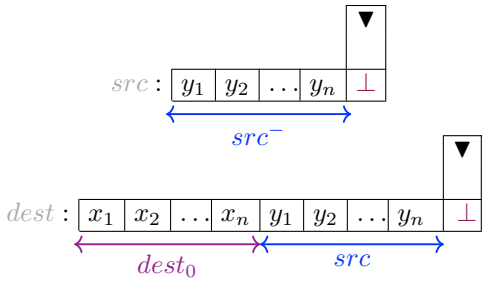
- déclarer les variables nécessaires. Avec la procédure `fappend()`, les deux fichiers sont déjà ouverts (cfr. PréCondition). Il faut par contre déclarer la variable `val` qui servira à la lecture de `src` et à l'écriture de `dest`. Puisque les fichiers contiennent des valeurs entières (cfr. énoncé), `val` sera de type `int`.
- initialiser la variable. Initialiser `val` n'a aucun intérêt car son seul objectif est de servir de réceptacle à la lecture des valeurs dans `src` pour être ensuite écrit dans `dest`.

On obtient donc :

```
// {PréCondition}
int val;
// {dest+ = < > ∧ mode(src) = "r" ∧ mode(dest) = "w" ∧ val =??}
// {dest+ = < > ∧ src− = < > ∧ dest = dest0 ∧ eof(dest)}
// {dest = dest0 || src− ∧ eof(dest)}
// {⇒ Inv}
```

B et Fonction de Terminaison

A la sortie de la boucle, on aura atteint la situation particulière suivante :



On voit clairement que le Critère d'Arrêt est le suivant :

$$\neg B \equiv eof(src)$$

On en dérive naturellement le Gardien de Boucle :

$$B \equiv \neg eof(src)$$

La Fonction de Terminaison est assez facilement obtenue, en s'appuyant sur la notation définie dans l'autre GameCode portant sur les fichiers :

$$f \equiv long(src^+)$$

ITER

Nous partons de la situation particulière suivante :

$$\text{INV} \wedge B \equiv dest = dest_0 \,||\, src^- \wedge eof(dest) \wedge \neg eof(src)$$

On peut réécrire cela de la façon suivante :

$$dest = dest_0 \,||\, src^- \wedge src^+ \neq <> \wedge eof(dest)$$

Cela signifie qu'il reste, au moins, un élément de `src` à concaténer à la fin de `dest`. Cela implique de lire la valeur courante dans `src` (possible car cette situation particulière correspond à la PRÉCONDITION de `read()` et la copier dans `dest`. {PRÉCONDITION$_{read}$}, {PRÉCONDITION$_{write}$}, {POSTCONDITION$_{read}$} et {POSTCONDITION$write$} sont respectivement la PRÉCONDITION et la POSTCONDITION de `read()` et `write()`.

Soit :

```
while(!eof(src)){
    // {INV ∧ B ⇒ PRÉCONDITION_read ∧ eof(dest)}
    read(src, val);
    // {POSTCONDITION_read : src⁻ = src⁻⁻ || < val > ∧ dest = dest0 || src⁻⁻ ∧ eof(dest) ⇒ PRÉCONDITION_write}
    write(dest, val);
    // {src⁻ = src⁻⁻ || < val > ∧ dest = dest0 || src⁻⁻ || < val > ⇒ dest = dest0 || src⁻ ∧ eof(dest)}
    // {⇒ INV}
}
```

On lit l'élément sous la tête de lecture de `src` (i.e., l'élément courant) et on l'écrit dans `dest`. Ce faisant, l'Invariant de Boucle est déjà rétabli. On passe donc à l'itération suivante. Avant d'appeler `read`, on se met dans une situation où sa PRÉCONDITION est vraie (c'est-à-dire $\neg eof(src)$, garantie par le Gardien de Boucle). Après l'appel, par l'approche constructive, la POSTCONDITION est vérifiée. On est toujours dans une situation où `write` peut être appelée car `eof(dest)` est garanti par l'Invariant de Boucle. Après son appel, par l'approche constructive, sa POSTCONDITION est respectée et `val` est rajoutée en dernière position du fichier de destination.

END

À la sortie de la boucle, nous sommes dans la situation suivante :

$$\text{INV} \wedge \neg B \equiv dest = dest_0 \,||\, src^- \wedge eof(dest) \wedge eof(src)$$

Cela signifie donc que src^- vaut tout le fichier `src`. On peut donc réécrire de la façon suivante :

$$dest = dest_0 \,||\, src \wedge eof(dest)$$

Il n'y a donc rien à faire puisqu'on a atteint la POSTCONDITION.

Code Complet du SP$_1$

```
void fappend(FILE src, FILE dest){
  // {PRÉCONDITION}
  int val;

  // {INV}
  while(! eof(src)){
    read(src, val);
    write(dest, val);
  }//fin while - i
}//fin fappend()
```

Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- construction du SP_0 ;
- mise en commun du code.

3.8 Code Complet

```
void fappend(FILE src, FILE dest){
  int val;

  while(! eof(src)){
    read(src, val);
    write(dest, val);
  }//fin while - i
}//fin fappend()

void fconcat(char *nom_fichier1, char *nom_fichier2, char *nom_concat){
  FILE f1 = fopen(nom_fichier1, "r");
  FILE f2 = fopen(nom_fichier2, "r");
  FILE f_concat = fopen(nom_concat, "w");

  fappend(f1, f_concat);
  fappend(f2, f_concat);

  fclose(f1);
  fclose(f2);
  fclose(f_concat)
}//fin fconcat()
```