Introduction au langage TMPL

Baptiste Mélès

13 avril 2010

Table des matières

I Introduction **2**

1 Bref historique **3**

2 Usages du TMPL **3**

3 Licence **3**

4 Téléchargement **3**

II Le langage TMPL **4**

5 Premier survol du langage TMPL **4**
5.1 Démarrer, écrire, terminer 4
5.1.1 Votre premier programme TMPL 4
5.1.2 Description du programme 5
5.2 Se déplacer 7
5.3 Lire un caractère 9
5.4 Conclusion 10

6 Ergonomie du TMPL **10**
6.1 Espaces 10
6.2 Sauts de ligne 11
6.3 Commentaires 11
6.4 Exécution directe 12

7 Précisions techniques **12**
7.1 Ordre des lignes d'instruction 12
7.2 Notion de caractère quelconque et boucle « si... alors... sinon » 13

8 Linéarisation de programmes **15**

III L'interprète `tmpl` 15
8.1 Version de l'interprète 16
8.2 Aide 16
8.3 Tabularisation d'un programme 17
8.4 Modalités d'exécution 18
8.4.1 Exécution pas à pas 18
8.4.2 Exécution explicite 18
8.4.3 Limitation du nombre d'étapes 19
8.4.4 Délai entre deux étapes 19
8.5 Programmes en ligne de commande 19
8.5.1 Exécuter une ligne de commande 19
8.5.2 Produire une chaîne de caractères 20
8.5.3 Utiliser un ruban non vierge 23
8.6 Limites de l'interprète `tmpl` 24

IV Annexes 24
8.7 Les programmes de Turing 24
8.7.1 Séquence 010101... (première formulation) 24
8.7.2 Séquence 010101... (deuxième formulation) 25
8.7.3 Séquence 001011011101111011111... 25
8.8 Messages d'erreur 25

Première partie

Introduction

Le TMPL est un langage de programmation simulant le comportement d'une machine de Turing. Les initiales TMPL signifient Turing Machine Programming Language, et le sigle se prononce comme le mot anglais *temple*.

Dans ce document, nous présenterons ce langage, puis les principales options de l'interprète `tmpl`, c'est-à-dire du logiciel qui exécute les programmes TMPL. Nous supposons que le lecteur sait déjà ce qu'est une machine de Turing[1].

Pour bien comprendre ce qui suit, il est important de bien distinguer trois choses :

- la *machine de Turing*, entité logico-mathématique, qui n'existe ni physiquement, ni informatiquement ;
- le *langage* TMPL, entité grammaticale abstraite dérivée de la machine de Turing, qui s'incarne dans des programmes et dans des listes de recommandations ;

[1] [Tur95]

– l'*interprète* **tmpl**, programme informatique exécutant des programmes rédigés en TMPL.

1 Bref historique

L'auteur de ces lignes a imaginé et réalisé le langage TMPL en mai 2008, parallèlement à la préparation d'une conférence sur Turing pour les agrégatifs de philosophie de l'Université Blaise Pascal (Clermont-Ferrand II).

2 Usages du TMPL

Comme les machines de Turing, le langage TMPL a d'abord une utilité spéculative, c'est-à-dire presque nulle. Mais il peut également servir :

– dans un cadre pédagogique : les étudiants comprendront peut-être mieux les machines de Turing s'ils peuvent les manipuler directement ;
– pour des preuves de concept en informatique théorique ;
– pour s'amuser.

3 Licence

Écrit en Perl, l'interprète `tmpl` est un logiciel libre sous licence GPL (version 3 ou supérieure). Vous pouvez donc le télécharger librement et gratuitement, et même modifier le code source puis redistribuer votre version personnalisée, à la seule condition que votre logiciel soit placé sous la même licence.

Mais si vous souhaitez absolument témoigner de votre rage ou de votre enthousiasme, rien ne vous empêche d'envoyer une carte postale à l'auteur du programme et de ce document, surtout si la carte représente des pingouins ou un beau paysage. Vous trouverez son adresse postale sur le site `http://baptiste.meles.free.fr/`.

4 Téléchargement

Vous pouvez télécharger l'interprète `tmpl` en vous rendant sur le site `http://baptiste.meles.free.fr/`, puis en suivant les liens. Il vous suffit ensuite d'installer le programme `tmpl` dans un répertoire de votre choix.

Les prérequis de cette documentation sont :

– d'avoir une connaissance de base du shell Unix ;
– de posséder Perl (version 5 ou supérieure) sur son ordinateur.

Si vous utilisez Linux ou un autre dérivé d'UNIX (y compris MacOS X), l'endroit le plus approprié pour installer le programme est sans doute le répertoire `$HOME/bin/`, où `$HOME` désigne votre répertoire personnel ; ou bien

le répertoire `/usr/bin/` si vous avez des droits d'administrateur (root) sur votre machine, et confiance en ce programme.

Sous Windows, il faut utiliser le programme depuis l'invite de commandes.

Deuxième partie
Le langage TMPL

5 Premier survol du langage TMPL

5.1 Démarrer, écrire, terminer

Un exemple simple valant tous les grands discours, voici votre premier programme en tmpl.

5.1.1 Votre premier programme TMPL

Ouvrez votre éditeur de texte favori, surtout si c'est Emacs. Tapez la ligne suivante, en respectant les majuscules (mais les espaces sont facultatives) :

```
START: >1 :STOP
```

Enregistrez maintenant cette ligne dans un fichier nommé `ecrit1.tmpl`, dans un répertoire quelconque de votre ordinateur. Nous ne saurions à ce propos trop vous recommander de créer un répertoire, nommé par exemple `tmpl-test`, dans lequel vous pourrez mener vos expérimentations, créer vos premiers programmes, etc.

Exécutez maintenant votre programme en tapant la commande

```
tmpl ecrit1.tmpl
```

Si tout se passe bien[2], vous verrez simplement ceci :

```
philonous bap ~ $ tmpl ecrit1.tmpl
1
philonous bap ~ $
```

Vous avez écrit un programme, certes modeste, dont la seule fonction est d'écrire le symbole 1 sur le ruban de la machine de Turing. Commentons maintenant, pas à pas, ce premier programme.

[2]Dans le cas contraire, c'est sans doute que votre fichier `tmpl` n'est pas exécutable (voir `chmod`), ou qu'il ne se trouve pas dans votre `$PATH`, ou encore que vous ne vous trouvez pas dans le répertoire du fichier `ecrit1.tmpl`.

5.1.2 Description du programme

L'état `START:` Le programme `ecrit1.tmpl` commence par

```
START:
```

Il s'agit d'un mot spécial du langage, qui désigne l'état de départ de la machine. Un programme devrait contenir au moins une fois l'instruction `START:`, à défaut de quoi il ne ferait rien du tout. Ceci dit, quoique stérile, omettre le `START:` n'en est pas moins légal, et ne générera aucun bogue. Un programme sans `START:` équivaut, ni plus ni moins, à un programme vide[3].

Les états de machine Plus généralement, un mot suivi du signe deux-points (`:`) et placé en début de ligne désigne un **état** de la machine — pour adopter le lexique de Turing, c'est une « m-configuration ». Un état de machine est un nom quelconque, contenant un nombre quelconque de caractères non nuls (donc sans espace). Exemples :

```
START:
épinards_surgelés:
```

```
白馬非馬:
```

```
1729:
abc123:
```

À l'exception de `START:`, qui est réservé, le choix des noms d'état est totalement libre. La casse est prise en compte, si bien que vous pouvez utiliser les noms `start:`, `sTaRt:` et `StArT:` en toute impunité, sans risquer la moindre interférence avec l'état initial de la machine ; mais vous vous exposez à des maux de crâne.

L'écriture sur le ruban Notre programme contient ensuite l'instruction

```
>1
```

qui signifie « écrire le symbole `1`[4] ». Si vous remplacez 1 par n'importe quel autre caractère, la machine l'écrira. La principale contrainte — nous verrons les autres plus tard — est que ce caractère doit être unique. Vous pouvez écrire

```
>9
```

[3] Libre à vous de créer des programmes vides, mais à résultat égal, autant ne pas en faire du tout...

[4] Ceux qui connaissent un peu le shell UNIX reconnaîtront le symbole `>` qui sert à rediriger la sortie d'un programme, typiquement dans un fichier — donc, d'une certaine façon, à « écrire ».

ou même

```
>爱
```

mais pas

```
>10
```

Pour écrire 10, vous devrez écrire 1 puis 0 ; nous verrons plus loin comment faire.

Il est possible d'effacer un caractère en écrivant l'instruction `>` sans argument. Voici par exemple un programme, `efface1.tmpl`, qui écrit un symbole avant de l'effacer (vous comprendrez plus loin la fonction du mot `suite` dans ce programme) :

```
# Écrire 1 et l'effacer.
START: >1 :suite
suite: >  :STOP
```

Une case vide est dans le langage TMPL strictement équivalente à une case contenant une espace. On peut considérer qu'au départ, par défaut, le ruban de la machine de Turing est rempli d'espaces.

Certains caractères doivent être « protégés » afin de ne pas interférer avec la syntaxe de TMPL. Comme dans le shell Unix, l'antislash joue le rôle de protecteur. Ainsi, pour écrire un point-virgule, un antislash etc., on tapera :

```
>\;
>\\
```

Par paresse ou dans le doute, vous pouvez aussi protéger tous les caractères que vous voulez, y compris ceux qui n'en ont pas besoin : `>\a`, `>\5`, etc.

Le changement d'état de la machine La ligne de notre premier programme se termine par le mot

```
:STOP
```

Un mot précédé du signe deux-points et placé en fin de ligne désigne un **changement d'état** de notre machine. Exemples :

```
:STOP
:abc123
```

```
:白馬非馬
```

```
:1729
:a
```

En l'occurrence, `:STOP` est un mot spécial du langage TMPL qui prescrit l'arrêt de la machine. Le plus court programme terminant est donc :

```
# Programme vide, qui se termine instantanément.
START: :STOP
```

Un programme « standard » contient au moins une fois `START:`, et généralement au moins une fois l'instruction `:STOP`. Mais vous pouvez parfaitement concevoir un programme sans `START:` — un programme vide —, aussi bien qu'un programme sans `:STOP`, tel qu'une boucle infinie[5].

Toute ligne du programme doit impérativement contenir un état et un changement d'état. Notez que les deux états ne sont pas nécessairement distincts : on peut donc parfaitement écrire des lignes récursives comme les suivantes :

```
# Ces lignes sont vides et constituent des boucles infinies.
abc123:  :abc123
START:   :START
```

Vous avez donc réalisé votre premier programme en TMPL. Félicitations ! N'hésitez pas, d'ores et déjà, à mener des expériences, à écrire de petits programmes en modifiant ou en ôtant tel ou tel mot.

5.2 Se déplacer

Jusqu'ici, votre usage de la machine de Turing est assez limité : vous pouvez seulement démarrer, écrire un caractère, et terminer. En d'autres termes, vous ne pouvez manipuler qu'une seule case du ruban de la machine. Passons maintenant à des programmes un peu plus complexes, et voyons comment exécuter plusieurs instructions successives, et se déplacer sur le ruban.

Se déplacer d'une case Enregistrez dans le fichier `123.tmpl` le programme suivant.

```
START:   >1  ->  :suite
suite:   >2  ->  :suite2
suite2:  >3      :STOP
```

On reconnaît ici l'état `START:` et le changement d'état `:STOP`, qui vous sont désormais familiers. Mais cette fois, on ne passe pas directement de `START:` à `:STOP` au sein de la même ligne d'instruction.

[5] Tous les programmes dépourvus de `:STOP` sont infinis, mais la réciproque n'est pas vraie : un programme peut être infini pour la simple et bonne raison qu'il ne passe jamais par `:STOP`, quand bien même ce changement d'état figure sur l'une des lignes du code.

Au début, notre machine est dans l'état `START:`. Elle exécute donc la première ligne d'instruction, qui consiste à écrire le symbole `1`. Vient ensuite, toujours sur la première ligne, le symbole

```
->
```

composé d'un tiret et du signe « plus grand que ». Il signifie « se déplacer d'une case vers la droite » ; il imite en effet une flèche tournée dans cette direction. Comme vous pouvez vous en douter, pour se déplacer vers la gauche, on utilise le symbole

```
<-
```

Dans le programme qui nous intéresse actuellement, nous nous déplaçons donc d'une case vers la droite. Enfin, nous passons à l'état `:suite`. Notre ruban présente l'état suivant, où le symbole « `_` » désigne l'emplacement actuel de la tête de lecture sur le ruban :

1	_

Notre machine étant à l'état `suite`, elle va lire chaque ligne d'instruction pour voir s'il convient de l'exécuter. La première ligne ne correspond pas, car elle commence par `START:` et non par `suite:`. Mais la deuxième ligne est la bonne ; la machine écrit donc `2` à l'emplacement actuel, puis se déplace vers la droite d'une case, et passe à l'état `:suivant2`. Notre ruban ressemble maintenant à ceci :

1	2	_

Enfin, on exécute la troisième ligne du programme, qui écrit `3` sur le ruban, et la machine s'arrête. Le ruban porte donc la suite de caractères

1	2	3

On peut également se déplacer d'un certain nombre de cases vers la gauche ou la droite en indiquant le nombre après la direction. Par défaut, la valeur de l'argument est 1 (on ne se déplace que d'une case vers la gauche ou vers la droite). Exemples :

```
->
->0
->2
->15
```

Comme vous pouvez le constater, ce nombre peut contenir plusieurs chiffres. On peut donc se déplacer en une seule fois d'autant de cases que l'on peut le souhaiter. Si le nombre est 0, la tête ne se déplace pas (comme si vous n'aviez pas utilisé du tout la fonction de déplacement). Il est impossible d'utiliser un nombre négatif, une fraction, un irrationnel, un nombre complexe, etc. : les choses sont déjà suffisamment compliquées avec des entiers naturels.

C'est au moyen de l'instruction de déplacement, et d'elle seule, que l'on peut écrire des mots de plusieurs caractères. Voici donc à quoi peut ressembler un programme affichant les mots `Hello world!`.

```
START: >H  ->   :2
2:     >e  ->   :3
3:     >l  ->   :4
4:     >l  ->   :5
5:     >o  ->2  :6
6:     >w  ->   :7
7:     >o  ->   :8
8:     >r  ->   :9
9:     >l  ->   :10
10:    >d  ->2  :11
11:    >!       :STOP
```

5.3 Lire un caractère

Il ne reste plus qu'une seule fonction à découvrir : comment lire un caractère. Étudions le programme `lit1.tmpl` :

```
START:    >1 :suite
suite: <0 >2 :STOP
suite: <1    :STOP
```

La première ligne ne devrait plus avoir de secret pour vous : je démarre, j'écris `1`, je passe à l'état `:suite`. La deuxième ligne se lit de la façon suivante : si je suis dans l'état `suite:` ET que je lise le caractère `0`, alors j'écris `2` et je m'arrête. La troisième ligne, quant à elle, signifie : si je suis dans l'état `suite:` ET que je lis le caractère `1`, alors je m'arrête. Notre machine, si nous exécutons le programme, écrira donc d'abord le nombre `1`. Ensuite, elle n'exécutera pas la deuxième ligne, car l'une des deux conditions n'est pas remplie : nous sommes dans le bon état, mais ne lisons pas le bon caractère. En revanche, la troisième ligne est exécutable, car elle satisfait les deux exigences. Par conséquent, la machine aura écrit `1` avant de s'arrêter, sans écrire le caractère `2`.

Le signe `<`, comme dans le shell Unix, signifie « lecture ». Son mode de fonctionnement est exactement analogue à celui du signe d'écriture `>` :

– on protège les caractères spéciaux (notamment le point-virgule, dont nous décrirons l'utilité à la page 15, et l'antislash, mais aussi n'importe quel caractère) avec l'antislash ;
– employé sans argument, le signe est à interpréter comme la lecture d'un caractère nul ou d'une espace.

Voyons cela avec le programme `lit_blanc.tmpl` :

```
START: <1 >2 :STOP
START: <  >0 :STOP
```

La première ligne signifie : si je suis dans l'état `START:` et que je lis le caractère `1`, alors j'écris `2` et je m'arrête. La seconde se lit : si je suis dans l'état `START:` et que je ne lis aucun caractère, alors j'écris `0` et je m'arrête. Exécutez ce programme : le ruban portera le signe `0`. En effet, le ruban étant vierge au démarrage de la machine, ce n'est pas la première, mais la deuxième ligne qui est exécutée.

5.4 Conclusion

À ce stade, vous connaissez déjà, ou peu s'en faut, toute la grammaire de base du langage TMPL ! Vous savez démarrer la machine, partir d'un état, lire et écrire un caractère, vous déplacer, changer d'état, et arrêter la machine. Une machine de Turing ne demande rien de plus. Vous pouvez donc retenir qu'une ligne typique d'instructions en TMPL présente l'aspect suivant :

```
etat:       <x       >y     ->z      :suite
```

où `etat:` et `:suite` ne comportent aucune espace, `x` et `y` représentent chacun zéro ou un seul caractère, et `z` un nombre entier naturel quelconque, ou bien aucun caractère. Toutes les instructions, à l'exception de l'état de la machine et du changement d'état, sont facultatives.

6 Ergonomie du TMPL

Étudions maintenant quelques fonctionnalités annexes du TMPL relatives à l'ergonomie du langage, qui prétendent conférer un minimum de lisibilité à cette grammaire relativement indigeste. Mais ne vous prenez pas à rêver : si les machines de Turing constituent un outil théorique infiniment précieux et élégant, elles n'en font pas moins passer l'assembleur pour un miracle de limpidité, ce qui n'est pas peu dire.

6.1 Espaces

Vous pouvez insérer autant d'espaces et de tabulations que vous le souhaitez au début, au milieu ou à la fin d'une ligne (mais pas à l'intérieur

d'un nom d'état). Le langage les ignore purement et simplement. Allons plus loin : non seulement elles ne sont pas interdites, mais nous ne saurions trop vous les recommander, tant un programme aéré gagne en lisibilité. Qu'il vous suffise de comparer le programme

```
START:<>1->:suite
suite:>2->:suite2
suite2:>3:STOP
```

avec le programme suivant, qui produit rigoureusement le même résultat :

```
START:     <     >1     ->    :suite
suite:           >2     ->    :suite2
suite2:          >3           :STOP
```

Comme vous pouvez le constater par vous-même, la structure du second apparaît bien plus nettement que celle du premier. Il est donc vivement conseillé d'abuser des espaces et des tabulations dans votre code, en TMPL comme dans la plupart des langages informatiques.

6.2 Sauts de ligne

Autant les espaces sont tolérés et encouragés, autant les sauts de ligne sont interdits à l'intérieur d'une instruction. Le programme

```
START:     <0      >1
->         :STOP
```

vous renverra tout simplement l'erreur suivante :

```
philonous bap ~ $ tmpl saut_ligne.tmpl
Erreur ("saut_ligne.tmpl":1) : Format de ligne invalide.
```

6.3 Commentaires

Abordons maintenant par une instruction qui ne se trouve pas dans la machine telle que Turing l'a décrite : les commentaires. Afin de rendre vos programmes plus lisibles, vous pouvez en effet insérer des remarques qui ne seront ni lues, ni exécutées. Pour cela, votre ligne doit commencer par le caractère # (dièse). Exemple :

```
# Ceci est un commentaire ; il ne sera pas lu par notre machine de
# Turing.
```

On peut également placer du texte en commentaire en l'encadrant par les signes /* et */. Exemple :

```
START: /* J'écris 1 puis me déplace */ >1 ->:suite
suite: /* J'écris 2 et termine */ >2 :STOP
```

Cette modeste fonctionnalité devrait vous épargner bien des tourments.

6.4 Exécution directe

On peut trouver quelque profit à écrire sur la première ligne du programme quelque chose comme :

```
#!/home/bap/bin/tmpl
```

— à supposer naturellement que le programme `tmpl` soit situé dans le répertoire `/home/bap/bin/`. On peut également utiliser la forme suivante :

```
#!/usr/bin/env tmpl
```

Vous pourrez ainsi exécuter le programme `123.tmpl` en tapant par exemple

```
./123.tmpl
```

au lieu de

```
/home/bap/bin/tmpl 123.tmpl
```

Veillez à ce que le programme soit bien exécutable, par exemple en utilisant `chmod` sous Unix.

7 Précisions techniques

7.1 Ordre des lignes d'instruction

Un ordre arbitraire L'ordre des lignes d'instruction dans un programme est, dans une large mesure, arbitraire. On peut donc presque toujours permuter sans dommage les lignes d'un programme. Le programme

```
START:  >1  ->  :suite
suite:  >2  ->  :suite2
suite2: >3      :STOP
```

est donc strictement équivalent au programme

```
START:  >1  ->  :suite
suite2: >3      :STOP
suite:  >2  ->  :suite2
```

qui est à son tour identique au programme

```
suite:  >2  ->  :suite2
suite2: >3      :STOP
START:  >1  ->  :suite
```

et au programme

```
suite2: >3      :STOP
suite:  >2  ->  :suite2
START:  >1  ->  :suite
```

Vous pouvez donc écrire votre programme dans l'ordre que vous souhaitez. Mais nous vous recommandons tout de même de suivre, dans la mesure du possible, un ordre un peu logique, et ce pour deux raisons, l'une purement psychologique — votre code sera bien plus lisible, partant plus simple à déboguer —, l'autre proprement technique : en suivant un ordre logique, vous éviterez les surprises — entendez par là : les bogues — liées au caractère glouton des lignes d'instruction. Précisons ce dernier point.

Des lignes gloutonnes Il existe un seul cas de figure où l'ordre des lignes ait une importance : celui où une ligne est « engloutie » par une autre. Étudions le programme suivant, que nous appellerons `glouton.tmpl` :

```
START: >1  :STOP
START: >2  :STOP
```

Ce programme écrit `1` et s'arrête. Si en revanche on permutait l'ordre des deux dernières lignes de ce programme, alors notre ruban porterait le nombre `2`. Les deux lignes sont en effet concurrentes, et la première dans l'ordre du programme « engloutit » l'autre. Il s'agit ni plus ni moins d'un ordre de priorité. C'est le seul cas dans lequel l'ordre des lignes d'un programme ait un effet. Mais si aucun couple de lignes de votre programme ne crée de concurrence, alors l'ordre est rigoureusement arbitraire.

7.2 Notion de caractère quelconque et boucle « si... alors... sinon »

Avant d'exécuter une ligne d'instructions — c'est-à-dire, typiquement, d'écrire un symbole ou de se déplacer —, notre machine effectue deux vérifications :

1. que son état soit bien celui qui est indiqué au début de la ligne ;
2. qu'elle lise bien sous la tête de lecture le caractère qu'on lui indique.

Par exemple, prenons le programme suivant, qui écrit d'abord un 0, puis, s'il lit un 0 (ce qui, convenons-en, sera nécessairement le cas), écrit un 1 à la place :

```
# J'écris d'abord 0.
START:      >0   :boucle
# Puis, si je suis dans l'état "boucle" ET que je lis 0, alors j'écris 1.
boucle: <0  >1   :STOP
```

On peut également réaliser le programme infini suivant, qui remplace un 0 par 1 et vice versa :

```
START:       >0  :boucle
boucle:  <1  >0  :boucle
boucle:  <0  >1  :boucle
```

Si l'on suit le déroulement de ce programme, nous voyons qu'il exécute d'abord la ligne de l'état `START:`, donc écrit `0`. Il entre alors dans l'état « `:boucle` », comme indiqué à la fin de la même ligne. Il va donc chercher si l'une des lignes correspondant à l'état « `boucle:` » peut être exécutée. La deuxième ligne de notre programme ne remplit pas les conditions nécessaires, car elle ne peut être exécutée que si le programme lit le symbole `1` ; or, nous avons écrit, pour l'instant, le symbole `0`. La ligne suivante, en revanche, peut être exécutée, car nous sommes bien dans l'état « `boucle:` » et lisons bien le caractère `0`. Nous pouvons donc, comme le veut la troisième ligne du programme, écrire le symbole `1`. Ensuite, nous voyons que la deuxième ligne peut être exécutée, puis la troisième, puis la deuxième, etc. à l'infini. Notre machine écrira puis effacera des `0` et des `1` pour l'éternité.

Lorsqu'il est employé sans argument, le signe `<` signifie que l'on lit un caractère vide (ou une espace. Le comportement des instructions `>` et `<` est donc identique : les employer sans argument revient à leur donner pour argument une espace. À titre d'exemple, vous pouvez analyser le comportement du programme suivant :

```
START: < >1 :suite
suite: <1 > :suite2
suite2: < >2 :STOP
```

La possibilité de représenter le signe nul permet également, en creux, d'exprimer la notion de « caractère (non nul) quelconque ». Soit en effet le programme suivant :

```
START: < >1 :STOP
START:   >2 :STOP
```

Sous une apparence simple se cache un fonctionnement peu trivial. Au départ est exécutée la première ligne, car elle suppose l'état `START:` et que sous la tête de lecture se trouve un caractère vide, comme le requiert l'instruction `<` employée sans argument. Elle écrit donc le caractère `1` sur cette case du ruban, puis nous renvoie vers l'état `:START`. Que fait alors le programme ? Il vérifie si la première ligne du programme est applicable. Mais elle ne l'est plus ! Car il ne suffit pas d'être dans l'état `START:` pour l'exécuter, encore faudrait-il que la case soit vide ; et comme nous venons d'y inscrire un `1`, elle ne l'est plus. Par conséquent, le programme va vérifier si

la deuxième ligne est exécutable. Comme elle ne s'intéresse pas au caractère situé sous la tête de lecture (l'instruction `<` y est absente), elle est applicable : la machine écrit donc `2` et s'arrête.

Et c'est ainsi que l'on code en TMPL la notion de « caractère quelconque » ! En vertu du principe des lignes gloutonnes, la première ligne va absorber tous les caractères nuls, la seconde tous ceux qui auront survécu au test de la première ligne — soit tous les caractères non nuls.

Généralisons. Cette technique permet de faire émerger la notion classique de boucle « si... alors... sinon » (*if... then... else...*). Le début du programme cité plus haut équivaut en effet à ce que l'on exprimerait dans maint langage de la façon suivante :

```
si caractere nul
     alors ecrire 1
     sinon ecrire 2
```

À titre d'exercice, nous vous laissons deviner comment exclure, non pas un, mais deux types de caractères (par exemple le caractère nul et le nombre 1).

8 Linéarisation de programmes

Nous avons vu qu'une ligne d'instructions ne pouvait être étalée sur plus d'une ligne (cf. *supra*, p. 11). Mais à l'inverse, il est tout à fait possible d'en écrire plusieurs sur une seule et même ligne. Autant vous avertir tout de suite que les programmes y perdent en lisibilité ! Mais cette fonctionnalité est d'un grand prix théorique, car elle permet de linéariser les programmes comme le fait Turing dans son article de 1936[6], et donc d'utiliser comme programme la sortie du ruban d'un autre programme (cf. p. 20), etc.

Pour écrire plusieurs lignes d'instructions sur une même ligne de texte, il suffit de les séparer par des points-virgules. L'un des programmes examinés plus haut deviendrait ainsi :

```
START: < >1 :suite; suite: <1 > :suite2; suite2: < >2 :STOP
```

ou encore, pour gagner en concision ce que l'on perd en lisibilité :

```
START:<>1:suite;suite:<1>:suite2;suite2:<>2:STOP
```

En général, cette fonctionnalité, précieuse pour la logique et l'informatique théorique, sera peu utilisée par les utilisateurs normaux du langage. Mais existent-ils seulement ?

[6] [Tur95], sections 5 et suivantes, p. 63 *sq.*

Troisième partie

L'interprète `tmpl`

L'interprète `tmpl` a pour principale fonction d'exécuter vos programmes TMPL ; mais il propose également en option quelques fonctionnalités supplémentaires...

8.1 Version de l'interprète

L'option `-v` affiche la version de `tmpl` que vous utilisez actuellement.

```
philonous bap ~ $ tmpl -v
tmpl 1.0
philonous bap ~ $
```

L'information est d'importance, car le langage TMPL devrait recevoir une extension importante dans les versions 2.0 et supérieures : la possibilité de créer des fonctions.

8.2 Aide

L'option `-h` affiche l'aide de `tmpl`. C'est un pense-bête pratique pour réviser les options les plus exotiques de notre interprète.

```
philonous bap ~ $ tmpl -h
tmpl : interprète du langage tmpl (Turing Machine Programming Language).
Version 1.0

Syntaxe :
tmpl [-v|-h]
tmpl [-p|-t|-x|-d secondes|-s étapes] [prog1 prog2 ...]

Options
  -v            Version de tmpl
  -h            Cette aide
  -p            Attend que l'on appuie sur Entrée entre deux opérations
  -t            Affichage du programme sous forme de tableau
  -x            Exécution eXplicite du programme (numéro de ligne)
  -d secondes   Nombre (entier ou décimal) de secondes entre deux
                instructions
  -s étapes     Nombre (entier) d'instructions à exécuter
philonous bap ~ $
```

8.3 Tabularisation d'un programme

L'option `-t` de `tmpl`, suivie d'un ou plusieurs noms de programmes, est très précieuse pour le débogage de programmes TMPL. Elle en affiche en effet la structure sous forme de tableau. En ceci, l'affichage se rapproche de celle que propose Turing dans son article.

Prenons le programme suivant, conçu pour afficher à l'infini la suite de nombres

0	1	0	1	0	1	0	1	0	1	0	1	...

Les lecteurs de Turing auront vraisemblablement reconnu le premier programme proposé dans la « Théorie des nombres calculables[7] ». Nous l'enregistrons dans un fichier `0101.tmpl` (l'extension *tmpl*, que nous adoptons par convention, est absolument arbitraire : elle n'a strictement aucune signification pour notre interprète).

```
#!/home/bap/bin/tmpl

# Si je lis un 0, j'avance de 2 cases et écris 1.
START:                  :b
b:      <0      ->2     :b2
b2:         >1          :b

# Si je lis un 1, j'avance de deux cases et écris 0.
b:      <1      ->2     :b3
b3:         >0          :b

# Et si je ne lis rien du tout, alors j'écris 0.
b:      <   >0          :b
```

Un programme difficile à lire et à comprendre au premier coup d'œil. Nous pouvons en construire une représentation plus régulière, indépendante des partis pris stylistiques du développeur (espaces, tabulations, commentaires, sauts de lignes...), avec `tmpl -t` :

```
philonous bap ~ $ ./tmpl -t 0101.tmpl
Ligne  État            Lect.  Écr.  Dépl.   Nouvel état
-------------------------------------------------------
4      START                                  b
5      b               0             2       b2
6      b2                      1             b
9      b               1             2       b3
10     b3                      0             b
```

[7] [Tur95], section 3, p. 55.

```
13     b               " "    0             b
philonous bap ~ $
```

Dans ce tableau, le symbole `" "` représente un blanc (une espace).

8.4 Modalités d'exécution

8.4.1 Exécution pas à pas

Pour le débogage d'un programme, il est souvent utile de suivre pas à pas son exécution. L'option `-p` vous en offre la possibilité. Le programme exécute une ligne d'instructions et attend que vous appuyez sur Entrée pour continuer.

8.4.2 Exécution explicite

L'option `-x` lance une exécution explicite du programme, c'est-à-dire qu'elle met à chaque étape en valeur la règle qui est appliquée. Cette option est particulièrement utile pour le débogage : lorsque le programme n'accomplit pas ce que vous attendez de lui, cette option vous donne les moyens de savoir pourquoi.

Soit le programme `ifthenelse.tmpl` suivant :

```
START: < >1 :START
START:   >2 :STOP
```

Le mode explicite donnera le résultat suivant :

```
philonous bap $ tmpl -x ifthenelse.tmpl
1. (1:1) 1
2. (2:1) 2
2
philonous bap $
```

Les étapes de l'exécution sont décomptées par des nombres suivis d'un point (ici `1.` puis `2.`). Entre parenthèses est indiquée l'instruction que l'on exécute : `(2:1)` signifie que l'on a exécuté la première instruction de la deuxième ligne (nous vous rappelons que l'on peut inscrire plusieurs lignes d'instructions sur une seule ligne de texte : elles seront donc numérotées `(2:1)`, `(2:2)`, `(2:3)`, etc.). Enfin, avant de s'arrêter, le programme inscrit, comme à l'accoutumée, l'état final du ruban (ici `2`).

8.4.3 Limitation du nombre d'étapes

Il est parfois utile de limiter l'exécution d'un programme à un nombre restreint d'étapes, notamment si le programme est infini et que l'on n'a pas l'éternité devant soi. L'option `-s`, suivie d'un nombre entier naturel (éventuellement nul, mais cela n'a pas grand intérêt), le permet :

```
philonous bap $ tmpl -s 5 0101.tmpl
0 1
philonous bap $ tmpl -s 10 0101.tmpl
0 1 0 1 0
philonous bap $
```

8.4.4 Délai entre deux étapes

On peut configurer le délai d'attente entre deux lignes d'instruction avec l'option `-d`, suivie d'un nombre (entier ou décimal, supérieur ou égal à 0) de secondes :

```
philonous bap $ tmpl -d 5 0101.tmpl
0
0
0 1
0 1
0 1 0
0 1 0
0 1 0 1
```

etc.

Les commandes qui précèdent (`-p`, `-x -s`, `-d`) peuvent être — parfois avantageusement — cumulées :

```
philonous bap $ tmpl -p -d 1 -s 15 -x 0101.tmpl
```

8.5 Programmes en ligne de commande

8.5.1 Exécuter une ligne de commande

L'interprète `tmpl` permet d'exécuter des programmes enregistrés dans un fichier, mais aussi des programmes tapés en ligne de commande, au moyen de l'option `-e`. Voyez plutôt :

```
philonous bap $ echo "START: >1 :STOP" | tmpl -e
1
philonous bap $
```

Cette ligne de commande procure en entrée à `tmpl` un programme court et simple, qu'il exécute à la volée. Pour des programmes plus élaborés, on tirera profit de la possibilité de linéariser les programmes au moyen des points-virgules (toute la commande doit se trouver sur une seule ligne) :

```
philonous bap $  echo "START::b; b:<0->2:b2; b2:>1:b; b:<1->2:b3; b3:>0:b; b:<>0:b"
| tmpl -e -s 10
0 1 0 1 0
philonous bap $
```

(J'ai dit que c'était possible, je n'ai jamais dit que c'était lisible...)

Quiconque connaît un peu Unix déduira de ce qui précède une stricte équivalence entre les deux commandes suivantes :

```
tmpl foo.tmpl
tmpl -e < foo.tmpl
```

Mais pourquoi utiliser l'option `-e` quand on peut s'en passer ?

8.5.2 Produire une chaîne de caractères

L'option `-e` possède une sœur en l'apparemment inoffensive option `-l`. Celle-ci, suivie d'une chaîne de caractères d'une longueur quelconque, produit un code TMPL permettant de l'afficher. Exemple :

```
philonous bap $ tmpl -l 'Hello world !'
START: >\H -> :2; 2: >\e -> :3; 3: >\l -> :4; 4: >\l -> :5; 5: >\o ->
:6; 6: > -> :7; 7: >\w -> :8; 8: >\o -> :9; 9: >\r -> :10; 10: >\l ->
:11; 11: >\d -> :12; 12: > -> :13; 13: >\! -> :14; 14: :STOP
philonous bap $
```

L'utilité de cette fonction ne saute certes pas aux yeux. Et pourtant elle est immense ! Car elle permet de créer des machines de Turing simulant d'autres machines de Turing ; en d'autres termes, le couple des options `-e` et `-l` reproduit l'astuce essentielle de l'article publié par Turing en 1936 : la faculté pour une machine de Turing d'en simuler une autre[8].

Voyez plutôt la séquence suivante :

```
philonous bap $ tmpl -l 'Hello world !' | tmpl -e
Hello world !
philonous bap $
```

Cela paraît un moyen bien étrange de réaliser l'équivalent d'un trivial

```
echo 'Hello world !'
```

[8] [Tur95], section 6, p. 66–68.

Et de fait, la chaîne `Hello world!` est d'abord codée en TMPL par l'option `-l`, avant d'être décodée par l'option `-e` — le tout en une seule ligne, et sans que le code TMPL en question n'apparaisse.

Pouquoi s'arrêter en si bon chemin ? Les esprits les plus tordus ne résisteront pas à l'envie de couper les cheveux en deux cent cinquante-six par la séquence suivante... Commençons en douceur avec le programme `1.tmpl` :

```
START: >1 :STOP
```

Nous obtenons très naturellement :

```
philonous bap $ cat 1.tmpl
START: >1 :STOP
philonous bap $ tmpl 1.tmpl
1
philonous bap $
```

Mais attention, les choses vont se compliquer maintenant ! Nous allons afficher à l'écran un *programme écrivant le programme* qui écrit `1`. Ce qui, dans le shell Unix, s'exprime ainsi :

```
philonous bap $ tmpl -l "‘cat 1.tmpl‘"
START: >\S -> :2; 2: >\T -> :3; 3: >\A -> :4; 4: >\R -> :5; 5: >\T ->
:6; 6: >\: -> :7; 7: > -> :8; 8: >\> -> :9; 9: >\1 -> :10; 10: > -> :11;
11: >\: -> :12; 12: >\S -> :13; 13: >\T -> :14; 14: >\O -> :15; 15: >\P
-> :16; 16: :STOP
philonous bap $
```

Si vous suivez toujours le raisonnement, nous allons maintenant exécuter exactement la même commande, mais en écrivant la sortie dans un fichier nommé `2.tmpl`, au lieu de l'afficher à l'écran :

```
philonous bap $ tmpl -l "‘cat 1.tmpl‘" > 2.tmpl
philonous bap $ cat 2.tmpl
START: >\S -> :2; 2: >\T -> :3; 3: >\A -> :4; 4: >\R -> :5; 5: >\T ->
:6; 6: >\: -> :7; 7: > -> :8; 8: >\> -> :9; 9: >\1 -> :10; 10: > -> :11;
11: >\: -> :12; 12: >\S -> :13; 13: >\T -> :14; 14: >\O -> :15; 15: >\P
-> :16; 16: :STOP
philonous bap $
```

On vérifie que le programme `2.tmpl` écrive bien le programme désiré :

```
philonous bap $ tmpl 2.tmpl
START: >1 :STOP
philonous bap $
```

Et on en conclut à juste titre que l'on peut l'exécuter par un petit détour :

```
philonous bap $ tmpl 2.tmpl | tmpl -e
1
philonous bap $
```

Vous pouvez jouer à créer ensuite les programmes `3.tmpl`, `4.tmpl` etc., selon le même schéma :

```
philonous bap $ tmpl -l "`cat 2.tmpl`" > 3.tmpl
philonous bap $ tmpl -l "`cat 3.tmpl`" > 4.tmpl
```

La taille des programmes continue d'enfler de manière exponentielle. À titre indicatif, `3.tmpl` est déjà très long :

```
START: >\S -> :2; 2: >\T -> :3; 3: >\A -> :4; 4: >\R -> :5; 5: >\T ->
:6; 6: >\: -> :7; 7: > -> :8; 8: >\> -> :9; 9: >\\ -> :10; 10: >\S ->
:11; 11: > -> :12; 12: >\- -> :13; 13: >\> -> :14; 14: > -> :15; 15: >\:
-> :16; 16: >\2 -> :17; 17: >\; -> :18; 18: > -> :19; 19: >\2 -> :20;
20: >\: -> :21; 21: > -> :22; 22: >\> -> :23; 23: >\\ -> :24; 24: >\T ->
:25; 25: > -> :26; 26: >\- -> :27; 27: >\> -> :28; 28: > -> :29; 29: >\:
-> :30; 30: >\3 -> :31; 31: >\; -> :32; 32: > -> :33; 33: >\3 -> :34;
34: >\: -> :35; 35: > -> :36; 36: >\> -> :37; 37: >\\ -> :38; 38: >\A ->
:39; 39: > -> :40; 40: >\- -> :41; 41: >\> -> :42; 42: > -> :43; 43: >\:
-> :44; 44: >\4 -> :45; 45: >\; -> :46; 46: > -> :47; 47: >\4 -> :48;
48: >\: -> :49; 49: > -> :50; 50: >\> -> :51; 51: >\\ -> :52; 52: >\R ->
:53; 53: > -> :54; 54: >\- -> :55; 55: >\> -> :56; 56: > -> :57; 57: >\:
-> :58; 58: >\5 -> :59; 59: >\; -> :60; 60: > -> :61; 61: >\5 -> :62;
62: >\: -> :63; 63: > -> :64; 64: >\> -> :65; 65: >\\ -> :66; 66: >\T ->
:67; 67: > -> :68; 68: >\- -> :69; 69: >\> -> :70; 70: > -> :71; 71: >\:
-> :72; 72: >\6 -> :73; 73: >\; -> :74; 74: > -> :75; 75: >\6 -> :76;
76: >\: -> :77; 77: > -> :78; 78: >\> -> :79; 79: >\\ -> :80; 80: >\: ->
:81; 81: > -> :82; 82: >\- -> :83; 83: >\> -> :84; 84: > -> :85; 85: >\:
-> :86; 86: >\7 -> :87; 87: >\; -> :88; 88: > -> :89; 89: >\7 -> :90;
90: >\: -> :91; 91: > -> :92; 92: >\> -> :93; 93: > -> :94; 94: >\- ->
:95; 95: >\> -> :96; 96: > -> :97; 97: >\: -> :98; 98: >\8 -> :99; 99:
>\; -> :100; 100: > -> :101; 101: >\8 -> :102; 102: >\: -> :103; 103: >
-> :104; 104: >\> -> :105; 105: >\\ -> :106; 106: >\> -> :107; 107: > ->
:108; 108: >\- -> :109; 109: >\> -> :110; 110: > -> :111; 111: >\: ->
:112; 112: >\9 -> :113; 113: >\; -> :114; 114: > -> :115; 115: >\9 ->
:116; 116: >\: -> :117; 117: > -> :118; 118: >\> -> :119; 119: >\\ ->
:120; 120: >\1 -> :121; 121: > -> :122; 122: >\- -> :123; 123: >\> ->
:124; 124: > -> :125; 125: >\: -> :126; 126: >\1 -> :127; 127: >\0 ->
:128; 128: >\; -> :129; 129: > -> :130; 130: >\1 -> :131; 131: >\0 ->
:132; 132: >\: -> :133; 133: > -> :134; 134: >\> -> :135; 135: > ->
:136; 136: >\- -> :137; 137: >\> -> :138; 138: > -> :139; 139: >\: ->
:140; 140: >\1 -> :141; 141: >\1 -> :142; 142: >\; -> :143; 143: > ->
```

```
:144; 144: >\1 -> :145; 145: >\1 -> :146; 146: >\: -> :147; 147: > ->
:148; 148: >\> -> :149; 149: >\\ -> :150; 150: >\: -> :151; 151: > ->
:152; 152: >\- -> :153; 153: >\> -> :154; 154: > -> :155; 155: >\: ->
:156; 156: >\1 -> :157; 157: >\2 -> :158; 158: >\; -> :159; 159: > ->
:160; 160: >\1 -> :161; 161: >\2 -> :162; 162: >\: -> :163; 163: > ->
:164; 164: >\> -> :165; 165: >\\ -> :166; 166: >\S -> :167; 167: > ->
:168; 168: >\- -> :169; 169: >\> -> :170; 170: > -> :171; 171: >\: ->
:172; 172: >\1 -> :173; 173: >\3 -> :174; 174: >\; -> :175; 175: > ->
:176; 176: >\1 -> :177; 177: >\3 -> :178; 178: >\: -> :179; 179: > ->
:180; 180: >\> -> :181; 181: >\\ -> :182; 182: >\T -> :183; 183: > ->
:184; 184: >\- -> :185; 185: >\> -> :186; 186: > -> :187; 187: >\: ->
:188; 188: >\1 -> :189; 189: >\4 -> :190; 190: >\; -> :191; 191: > ->
:192; 192: >\1 -> :193; 193: >\4 -> :194; 194: >\: -> :195; 195: > ->
:196; 196: >\> -> :197; 197: >\\ -> :198; 198: >\O -> :199; 199: > ->
:200; 200: >\- -> :201; 201: >\> -> :202; 202: > -> :203; 203: >\: ->
:204; 204: >\1 -> :205; 205: >\5 -> :206; 206: >\; -> :207; 207: > ->
:208; 208: >\1 -> :209; 209: >\5 -> :210; 210: >\: -> :211; 211: > ->
:212; 212: >\> -> :213; 213: >\\ -> :214; 214: >\P -> :215; 215: > ->
:216; 216: >\- -> :217; 217: >\> -> :218; 218: > -> :219; 219: >\: ->
:220; 220: >\1 -> :221; 221: >\6 -> :222; 222: >\; -> :223; 223: > ->
:224; 224: >\1 -> :225; 225: >\6 -> :226; 226: >\: -> :227; 227: > ->
:228; 228: >\: -> :229; 229: >\S -> :230; 230: >\T -> :231; 231: >\O ->
:232; 232: >\P -> :233; 233: :STOP
```

... tout cela pour écrire simplement `1` au moyen de la commande

```
philonous bap $ tmpl 3.tmpl | tmpl -e | tmpl -e
1
philonous bap $
```

Sous des dehors futiles, ces fonctionnalités devraient théoriquement permettre la création d'une machine de Turing universelle, c'est-à-dire d'un programme TMPL capable d'exécuter tout programme TMPL. L'auteur de ces lignes n'a pas eu le courage d'en entreprendre la programmation, mais si vous vous en sentez la force, n'hésitez pas...

8.5.3 Utiliser un ruban non vierge

Par défaut, le ruban utilisé par `tmpl` est d'abord vierge. Mais il est possible de fournir à notre machine de Turing un ruban déjà partiellement rempli, en ayant recours à l'option `-i`, suivie de la série de caractères que doit porter le ruban. Après le remplissage du ruban et avant l'exécution du programme, la tête de lecture se positionne au début de cette chaîne de caractères. Ces quelques exemples devraient suffire à rendre intelligible le fonctionnement de l'option `-i` :

```
philonous bap $ tmpl vide.tmpl

philonous bap $ tmpl -i 123456789 vide.tmpl
123456789
philonous bap $ tmpl helloworld.tmpl
Hello world!
philonous bap $ tmpl -i 123456789 helloworld.tmpl
Hello6world!
```

8.6 Limites de l'interprète `tmpl`

La machine de Turing, entité logico-mathématique, ne connaît pas de limitation empirique, que ce soit en rapidité, en espace disponible, etc. Mais l'interprète `tmpl` partage les limites de son support matériel et logiciel. Il est donc limité par la mémoire vive de votre ordinateur, la capacité de calcul de votre processeur, les caractères reconnus par Unicode, et autres limites inhérentes à Perl.

Turing n'osait en rêver, `tmpl` l'a fait : les caractères chinois — entre autres — sont parfaitement reconnus[9]. En revanche, certains caractères ne le sont pas tout à fait : la tabulation, les sauts de ligne, etc. En outre, le caractère nul n'est pas distingué de l'espace. D'un point de vue théorique, les symboles logiques étant arbitraires, le dommage est négligeable. Et d'un point de vue pratique, personne ne songerait à utiliser TMPL pour quoi que ce soit d'utile. Gageons donc que ces lacunes n'incommoderont personne.

Quatrième partie
Annexes

8.7 Les programmes de Turing

À titre d'exemple, nous présenterons ici les premires programmes décrits par Turing dans l'article de 1936[10].

8.7.1 Séquence 010101... (première formulation)

```
START: >0 -> :c
c:        -> :e
e:     >1 -> :f
f:        -> :START
```

[9] [Tur95], section 9, p. 78 : « le chinois [...] tend à avoir une infinité dénombrable de symboles ».

[10] [Tur95], section 3, p. 54–57.

8.7.2 Séquence 010101... (deuxième formulation)

```
START: <  >0     :START
START: <0    ->2 :a
a:        >1     :START
START: <1    ->2 :b
b:        >0     :START
```

8.7.3 Séquence 0010110111011111011111...

```
START:          >H      ->1     :b2
b2:                     >H      ->1             :b3
b3:                     >0      ->2             :b4
b4:                     >0      <-2             :o
o:              <1              ->1             :o2
o2:                     >x      <-3             :o
o:              <0                              :q
q:              <0              ->2             :q
q:              <1              ->2             :q
q:              <       >1      <-1             :p
p:              <x      >       ->1             :q
p:              <H              ->1             :f
p:              <               <-2             :p
f:              <0              ->2             :f
f:              <1              ->2             :f
f:              <       >0      <-2             :o
```

8.8 Messages d'erreur

`Erreur : Fichier "foo.tmpl" inexistant.` Le programme n'existe pas à l'endroit demandé.

`Impossible de lire le fichier "foo.tmpl". Avez-vous les droits pour lire ce fichier ?` Le fichier existe mais n'est pas accessible en lecture.

`Erreur ("foo.tmpl":3:1) : Format de ligne invalide.` La première instruction de la ligne 3 du fichier `foo.tmpl` n'est pas valide. Vérifiez sa syntaxe.

`Impossible de fermer le fichier "foo.tmpl".` Rien de dramatique en général, mais `tmpl` n'a pas réussi à fermer le fichier.

Références

[Tur95] Alan Turing. Théorie des nombres calculables, suivie d'une application au problème de la décision. In Alan Turing and Jean-Yves Girard, editors, *La Machine de Turing*, Points Sciences, pages 47–104. Seuil, Paris, 1995. trad. et annotation par Julien Basch.