

Approche pratique de la théorie des langages formels.

Rapport de TIPE

Charlotte THOMAS

5 juin 2022

Résumé

L'informatique théorique et principalement l'analyse théorique des langages formels m'a toujours intéressée. L'idée de créer un langage, d'implémenter moi-même sa grammaire, son lexer, parser, interpreter et d'en calculer les propriétés théoriques est depuis toujours un sujet d'intérêt et de motivation.

Les erreurs de compilation peuvent coûter cher, et parfois se compter en vies humaines, il est donc nécessaire de développer des outils et des théories dans le domaine de la théorie des langages formels pour prévenir les bugs en amont, et protéger des vies et la société.

Pour illustrer ceci, ce rapport va parler de la création du *Baguette#* (à prononcer "Baguette Sharp") un langage de programmation exotique avec une syntaxe proche d'un BASIC, basé sur des pâtisseries. Le REPL est disponible sur OPAM *opam install baguette-sharp*, le code source est sur [GitHub](https://github.com/CharlotteThomas/baguette-sharp) sous licence MIT, enfin le site web est disponible à cette adresse <https://www.baguettesharp.fr>.

Finalement, ce rapport traitera essentiellement de l'implémentation d'une Binary Turing Machine en B#. Pour plus de renseignements généraux merci de voir les liens au ci-dessus.

Table des matières

1	Le Baguette#	2
1.1	Syntaxe vu par un exemple	2
1.2	Librairie Standard	2
2	Machine de Turing Binaire en B#	3
2.1	Définition	3
2.2	Spécification	4
2.3	Initialisation	4
2.4	Lecture du programme	5
2.5	Étape	5
2.6	Runtime de la machine	6

3	Exemple de programmes	7
3.1	left bit-shift	7
3.2	Binary adder	8
3.3	Programme sans fin	8

Table des figures

1	Résultat dans le REPL de l'opération $1 + 2$	2
2	Initialisation de la machine	5
3	Runtime de la machine	6
4	Automate représentant le bitshift	7
5	Exécution du left-bitshift	7
6	Automate représentant l'addition binaire	8
7	Dernières lignes de l'exécution de l'addition binaire	8
8	Trace d'un programme sans fin	9

1 Le Baguette#

1.1 Syntaxe vu par un exemple

La syntaxe du Baguette# est très inspiré d'un BASIC, la plus grande différence est l'absence complète d'opérateur INFIX, là où un langage "classique" sera comme ça

```
int a = 1 + 2;
```

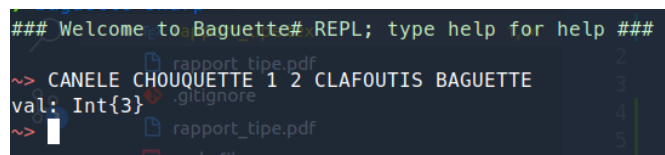
afin de calculer la somme de 1 et 2 (l'affectation sera discuté plus tard) le B# utilise une instruction comme ceci

```
ADD ( 1 2 );
```

la différence étant qu'en B# l'instruction *ADD* se nomme *CANELE*, et que les parenthèses ouvrante et fermantes sont respectivement *CHOUQUETTE* et *CLAFOUTIS*. Finalement, le point-virgule est *BAGUETTE* donc la somme est notée

```
CANELE CHOUQUETTE 1 2 CLAFOUTIS BAGUETTE
```

Le résultat est visualisé ici sur le REPL :



```
### Welcome to Baguette# REPL; type help for help ###
~> CANELE CHOUQUETTE 1 2 CLAFOUTIS BAGUETTE
val: Int{3}
```

FIGURE 1 – Résultat dans le REPL de l'opération $1 + 2$

La liste des mots clés de définitions est assez courte, il s'agit de *IF*, *LABEL*, *JUMP*. Leur utilisation est expliquée [ici](#). A l'exception de ces mots clé, toutes les instructions doivent avoir leur paramètre entre parenthèse. Les paramètres peuvent être séparés par une virgule, la *lexer* ne les vois pas. Mais ils doivent être séparés par un espace au minimum (y compris avec une virgule).

```
CANELE CHOUQUETTE 1 , 2 CLAFOUTIS BAGUETTE
```

est équivalent à

```
CANELE CHOUQUETTE 1 2 CLAFOUTIS BAGUETTE
```

Une explication complète de la syntaxe est disponible sur le [wiki-syntaxe](#).

1.2 Librairie Standard

Une liste complète des instructions est visible sur le [wiki](#) et un tutoriel sur l'utilisation basique est disponible [ici](#). La librairie standard du langage implémente les usages suivants :

- Opérations Mathématiques sur les entiers et flottants
- Algèbre booléenne

- Lecture de l'entrée standard, écriture sur la sortie standard
- Manipulation de tableaux
- Manipulation de chaînes de caractères

Le langage est typé faiblement, et avec inférence de type, les types primitifs suivants sont implémentés :

- Nombres entiers, flottants (NB : aucune distinction n'est faite entre les deux en général)
- Chaînes de caractères (NB : les caractères sont des chaînes de caractère de longueur 1)
- Booléens (*CUPCAKE* est *true* et *POPCAKE* est *false*)
- Null (le *unit* de OCaml)

Enfin, le langage implémente les tableaux, à savoir qu'ils peuvent être *non-homogènes* donc il est possible de faire un tableau contenant deux entiers, trois flottants et quatre chaînes de caractères. Cependant, comme ce sont des tableaux ils sont de *taille fixe* et ils héritent du caractère *mutable* des tableaux de OCaml.

2 Machine de Turing Binaire en B#

2.1 Définition

Une définition formelle d'une machine de Turing est énoncé par Lewis dans *Elements of the Theory of Computation* en 1982, il définit une machine de Turing comme un 5-uplet $(K, \Sigma, \delta, s, H)$ avec K l'ensemble fini d'états possible de la machine, Σ est l'alphabet qui contient le caractère nul et le caractère marquant le début du ruban, $s \in K$ est l'état initial de la machine, δ est la fonction de transition des états formellement $(K - H) \times \Sigma \rightarrow K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$ (elle prend en entrée un couple formé d'un état non sortant et d'un état du ruban lu et renvoi un couple du nouvel état de la machine et du nouvel état à écrire ou s'il faut bouger à gauche ou à droite, ce qui revient à un nouvel état à écrire ET s'il faut bouger à gauche/droite), et enfin $H \subset K$ est l'ensemble des états finaux dit "de halte". (*Elements of the Theory of Computation, Chapter 4, Definition 4.1.1, p. 181*)

Ici est utilisé une machine de Turing dite, *binaire*, avec le caractère 2 comme caractère vide, il y a un seul état sortant dénoté \bar{H} (pour éviter de le confondre avec le H de la définition, ils sont ensuite confondus) les autres états sont des entiers. Les caractères $r, l, *$ respectivement signifient aller à droite, aller à gauche, et ne pas bouger. Enfin la fonction δ est découpé en trois fonctions $\delta_1, \delta_2, \delta_3$ donnant respectivement le nouvel état, ce qu'il faut écrire sur le ruban, et comment il faut bouger la tête on a donc quelque chose comme ci-dessous. On utilise des matrices pour les fonctions (matrice de tailles (n,3)), un nombre pour l'étape actuelle et une chaîne de caractère pour le ruban, tout ceci pour

pouvoir être implémenté en Baguette#!

$$\begin{aligned}K &= H \cup Q & Q &\subset \mathbb{N} \\ \Sigma &= \{0, 1, 2\} \\ s &= 0 \\ H &= \{\bar{H}\} \\ \delta_1 &: Q \times \Sigma \rightarrow K \\ \delta_2 &: Q \times \Sigma \rightarrow \Sigma \\ \delta_3 &: Q \times \Sigma \rightarrow \{r, l, *\}\end{aligned}$$

Un moyen simple et ludique de montrer qu'un langage est Turing-complet est d'implémenter un émulateur d'une machine de Turing universelle dans ce langage.

La machine de Turing et les programmes ont été adapté depuis ce tutoriel sur une machine de turing universelle binaire en python.

2.2 Spécification

Malheureusement, on ne peut pas avoir une RAM infinie dans la réalité. Voici la spécification de la machine qui est implémentée :

- Le ruban est de longueur 1000, initialisée avec 1000 "2" comme la machine est binaire le 2 est considéré comme étant le caractère nul (une case vide)
- L'input est placée à partir de la place 500 dans le ruban, on place la tête au rang 500, au début de l'input.
- Les programmes sont une suite d'instructions de la forme STATE-READ-WRITE-TRANSLATION-NEWSTATE. Par exemple, pour dire que si on lit un 1 en étant dans l'état 0 alors on écrit un 2, on va à droite et on passe dans l'état 1 on écrira $0 - 1 - 2 - r - 1$.
- On limite le tout à 100 states différents pour des raisons de performance, mais on peut augmenter en changeant la taille de la matrice.
- On appelle l'état de sortie "H". Pour les déplacements : 'r' déplace la tête d'une case vers la droite, 'l' d'une case vers la gauche et '*' ne bouge pas la tête.

2.3 Initialisation

Voici une image du code de l'initialisation.

```

//Turing machine initialization
//Creation of three 100 by 3 matrices to represent the transition function
//Declaration of the initial state and the tape with the head at the first bit of the input
QUATREQUART CHOUQUETTE PARISBREST state PARISBREST , PARISBREST 0 PARISBREST CLAFOUTIS BAGUETTE
CROISSANT CHOUQUETTE PARISBREST Please type the input tape PARISBREST CLAFOUTIS BAGUETTE
QUATREQUART CHOUQUETTE PARISBREST tape PARISBREST , GAUFFREDELIEGE CHOUQUETTE GAUFFREDELIEGE CHOUQUETTE BUCHE CHOUQUETTE 500 , PARISBREST
QUATREQUART CHOUQUETTE PARISBREST head PARISBREST , 500 CLAFOUTIS BAGUETTE
CROISSANT CHOUQUETTE PARISBREST Please type the number of line of the program PARISBREST CLAFOUTIS BAGUETTE
QUATREQUART CHOUQUETTE PARISBREST n PARISBREST , ECLAIR CHOUQUETTE CLAFOUTIS CLAFOUTIS BAGUETTE
QUATREQUART CHOUQUETTE PARISBREST i PARISBREST , 0 CLAFOUTIS BAGUETTE
QUATREQUART CHOUQUETTE PARISBREST delta0 PARISBREST , TARTEALARHUBARBE CHOUQUETTE 100 , 3 , 0 CLAFOUTIS BAGUETTE
QUATREQUART CHOUQUETTE PARISBREST delta1 PARISBREST , TARTEALARHUBARBE CHOUQUETTE 100 , 3 , 0 CLAFOUTIS BAGUETTE
QUATREQUART CHOUQUETTE PARISBREST delta2 PARISBREST , TARTEALARHUBARBE CHOUQUETTE 100 , 3 , 0 CLAFOUTIS BAGUETTE

```

FIGURE 2 – Initialisation de la machine

En première ligne on initialise l'état initial de la machine à 0, puis on demande l'input tape (donc l'input). La ligne suivante n'est pas affichée entièrement, elle initialise le ruban comme étant de longueur 500, puis on met l'input, puis on remet 500 caractères vides (donc des 2). Le plus important ici est les appels à l'instruction *TARTEALARHUBARBE* qui construit une matrice de taille $n \times p$ (ici 100 par 3) initialisée avec des 0. Ces trois matrices serviront à stocker le programme. Deux autres variables qui vont aider pour la boucle de lecture du programme sont initialisées : n , le nombre de lignes du programme, et i à la valeur 0.

2.4 Lecture du programme

Pour lire le programme sur l'entrée standard, il nous faut émuler une boucle conditionnelle, que le langage n'implémente pas nativement. Pour cela nous aurons besoin des deux variables déclarées précédemment, n et i , de l'instruction permettant de construire des labels *ICECREAM*, de l'instruction permettant de JUMP à un label *PAINVIENNOIS* ainsi que d'un test conditionnel ici *SABLE*.

La boucle fait ceci dans l'ordre :

- Lecture de l'entrée standard
- Découpe de la chaîne de caractère à chaque caractère "-"
- Remplissage des trois matrices avec les programmes correspondants
- Incrémentation de i
- Vérification de i est supérieur ou égal à n
- Si $i \geq n$, alors : quitter le programme
- Sinon : exécuter un JUMP à lui-même

Ces étapes, une fois codées en B#, rendent 23 lignes de code (longues) disponible ici jusqu'à la ligne 23.

Ces étapes terminées, le programme est lu et stocké dans les 3 matrices.

2.5 Étape

Il est ensuite nécessaire d'implémenter une *étape* du programme, là aussi implémenté avec un label pour faire plus "propre" et pour séparer les différentes fonctionnalités du

programme – les labels ne sont pas des *fonctions* à proprement parler, et le *scopes* des variables est global, permettant d’émuler un peu le fonctionnement de véritables fonctions avec uniquement des labels. L’instruction la plus utilisée sera *TARTEAUXFRAISES* qui permet d’accéder à l’élément n d’un tableau.

Une étape doit faire ceci :

- Vérifier qu’on est bien dans un état différent de "H". La suite est dans le cas ou cette assertion est vraie
- Lire le ruban avec la tête de lecture pour obtenir l’état du ruban appelé e , l’état de la machine est s
- Récupérer ce que l’on doit écrire depuis la première matrice aux coordonnées (s, e)
- Récupérer dans quelle direction aller depuis la seconde matrice aux coordonnées (s, e)
- Récupérer le nouvel état de la machine depuis la troisième matrice aux coordonnées (s, e)
- Modifier le ruban en écrivant ce que l’on doit écrire
- Mettre à jour l’état de la machine avec le nouvel état
- Mettre à jour la position de la tête selon si on a lu r , l ou $*$
- Afficher le ruban (en enlevant les 2) et l’état de la machine

L’implémentation en B# de ce code est disponible une fois encore sur GitHub [ici](#) jusqu’à la ligne 58 Ce morceau de code fait uniquement *une* étape. Il faut maintenant implémenter la boucle principale.

2.6 Runtime de la machine

Maintenant qu’une étape de la machine a été implémentée, il faut répéter les étapes tant que l’état de la machine n’est pas "H". On verra dans les exemples comment le programme se comporte face à un programme qui ne s’arrête pas. Là encore nous avons affaire à une boucle conditionnelle que l’on peut implémenter à l’aide d’un label et d’un test conditionnel.

```
ICECREAM PARISBREST run PARISBREST
MUFFIN
  PAINVIENNOIS PARISBREST step PARISBREST
  SABLE CHAUSSONAUXPOMMES CHOUQUETTE TIRAMISU CHOUQUETTE MADELINE CHOUQUETTE PARISBREST state PARISBREST CLAFOUTIS , PARISBREST H PARISBREST CLAFOUTIS CLAFOUTIS
  FRAMBOISIER MUFFIN
    PAINVIENNOIS PARISBREST run PARISBREST
  COOKIES BAGUETTE
COOKIES BAGUETTE
```

FIGURE 3 – Runtime de la machine

Qui peut se traduire par "tant que l’état n’est pas 'H', on exécute step". Enfin, nous avons besoin d’ajouter un "JUMP "run"" juste après l’initialisation pour lancer le programme :

```
PAINVIENNOIS PARISBREST run PARISBREST
```

3 Exemple de programmes

3.1 left bit-shift

Le left-bitshift est une multiplication par 2 en binaire, c'est "bouger" tous les bits vers la gauche et ajouter un 0 à la fin. On peut faire ceci avec l'algorithme suivant :

- Si on est dans l'état 0 et qu'on lit un 0 on écrit un 0, on bouge à droite, et on reste dans l'état 0
- Si on est dans l'état 1 et qu'on lit un 1 on écrit un 1, on bouge à droite, et on reste dans l'état 0
- Si on est dans l'état 0 et qu'on lit un 2 on écrit un 0, on ne bouge pas et passe en état "H"

Sous forme d'automate (les transitions sont d'état de machine à état de machine, étiquetées par l'état du ruban lu)

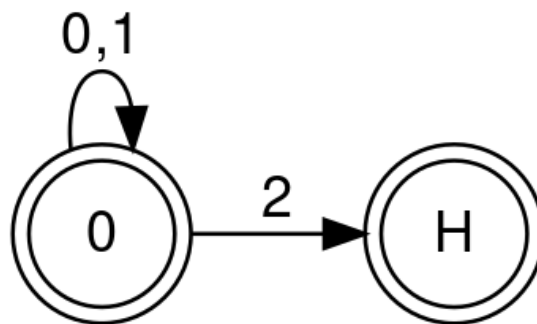


FIGURE 4 – Automate représentant le bitshift

Et sous forme de programme pour notre machine :

```
0-0-0-r-0
0-1-1-r-0
0-2-0-*-H
```

Dont l'exécution donne :

```
> load examples/turing.baguette
Please type the input tapes
101
Please type the number of line of the program
3
0-0-0-r-0
0-1-1-r-0
0-2-0-*-H
Step (tape,state): (101,0)
Step (tape,state): (101,0)
Step (tape,state): (101,0)
Step (tape,state): (1010,H)
Final (tape,state,head): (1010,H,503)
```

FIGURE 5 – Exécution du left-bitshift

3.2 Binary adder

Un additionneur binaire est bien plus complexe, le programme fait 17 lignes (et ne sera pas expliqué ici), rapidement il se sert du deuxième chiffre comme d'un compteur pour additionner un par un au premier chiffre jusqu'à ce que le compteur atteigne 0. La représentation sous forme d'automate donne ceci

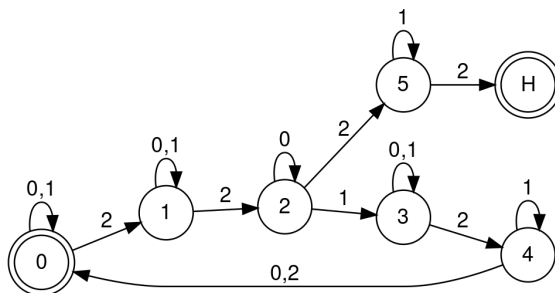


FIGURE 6 – Automate représentant l'addition binaire

Les 17 lignes du programme sont disponibles [ici](#). L'exécution pour l'addition de 10 et 8 (en binaire 1010 et 1000) donne 10010 ce qui est bien $16+2 = 18$.

```
Step (tape,state): (100100111,2)
Step (tape,state): (100101111,2)
Step (tape,state): (100101111,5)
Step (tape,state): (10010111,5)
Step (tape,state): (1001011,5)
Step (tape,state): (100101,5)
Step (tape,state): (10010,5)
Step (tape,state): (10010,H)
Final (tape,state,head): (10010,H,509)
```

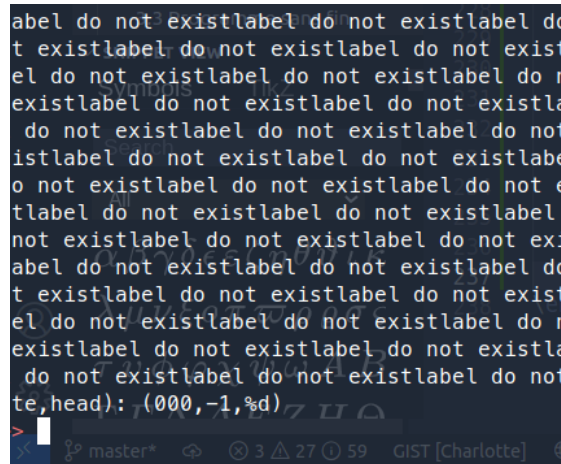
FIGURE 7 – Dernières lignes de l'exécution de l'addition binaire

3.3 Programme sans fin

Prenons un programme simple qui ne s'arrête pas :

```
0-0-0-* -1
1-0-0-* -0
```

Qui n'écrit rien reste sur place et constamment passe de l'état 0 à 1 et vice-versa. L'exécuter fait que l'interpréteur affiche pleins d'erreur jusqu'à s'arrêter et retourner sur le REPL.



```
abel do not existlabel do not existlabel do
t existlabel do not existlabel do not exist
el do not existlabel do not existlabel do n
existlabel do not existlabel do not existla
do not existlabel do not existlabel do not
istlabel do not existlabel do not existlabe
o not existlabel do not existlabel do not e
tlabel do not existlabel do not existlabel
not existlabel do not existlabel do not exi
abel do not existlabel do not existlabel do
t existlabel do not existlabel do not exist
el do not existlabel do not existlabel do n
existlabel do not existlabel do not existla
do not existlabel do not existlabel do not
te,head): (000,-1,%d)
>
```

FIGURE 8 – Trace d'un programme sans fin