Approche pratique de la théorie des langages formels.
Oral de TIPE

Charlotte THOMAS

5 juin 2022

Résumé

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 cette présentation se concentre sur le *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 sous licence MIT, enfin le site web est disponible à cette adresse `https://www.baguettesharp.fr`

Résumé

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 cette présentation se concentre sur le *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 sous licence MIT, enfin le site web est disponible à cette adresse `https://www.baguettesharp.fr`

Tableau des matières

Présentation générale

Le Baguette# a une syntaxe proche d'un BASIC à quelques exceptions près, qui ont été choisies dès le début.

- L'absence d'opérateurs INFIX (par exemple 1+2)
- Toutes les instructions ont un nom de pâtisserie (pour le rendre moins simple et plus éxotique)
- Chaque structure (condition, boucle, fonction) a une syntaxe complètement différente que les autres
- Enfin, tous les algorithmes ainsi que leurs implémentations ont été codées à la main sans avoir recours aux bibliothèques en OCaml (telles que OCamlLex et Menhir).

Présentation générale

Le Baguette# a une syntaxe proche d'un BASIC à quelques exceptions près, qui ont été choisies dès le début.

- L'absence d'opérateurs INFIX (par exemple 1+2)
- Toutes les instructions ont un nom de pâtisserie (pour le rendre moins simple et plus éxotique)
- Chaque structure (condition, boucle, fonction) a une syntaxe complètement différente que les autres
- Enfin, tous les algorithmes ainsi que leurs implémentations ont été codées à la main sans avoir recours aux bibliothèques en OCaml (telles que OCamlLex et Menhir).

Présentation générale

Le Baguette# a une syntaxe proche d'un BASIC à quelques exceptions près, qui ont été choisies dès le début.

- L'absence d'opérateurs INFIX (par exemple 1+2)
- Toutes les instructions ont un nom de pâtisserie (pour le rendre moins simple et plus éxotique)
- Chaque structure (condition, boucle, fonction) a une syntaxe complètement différente que les autres
- Enfin, tous les algorithmes ainsi que leurs implémentations ont été codées à la main sans avoir recours aux bibliothèques en OCaml (telles que OCamlLex et Menhir).

Présentation générale

Le Baguette# a une syntaxe proche d'un BASIC à quelques exceptions près, qui ont été choisies dès le début.

- L'absence d'opérateurs INFIX (par exemple 1+2)
- Toutes les instructions ont un nom de pâtisserie (pour le rendre moins simple et plus éxotique)
- Chaque structure (condition, boucle, fonction) a une syntaxe complètement différente que les autres
- Enfin, tous les algorithmes ainsi que leurs implémentations ont été codées à la main sans avoir recours aux bibliothèques en OCaml (telles que OCamlLex et Menhir).

Algorithmes

Le B# est pour l'instant uniquement interprété il passe donc par plusieurs étapes standard afin d'être évalué correctement

- Le fichier source / la ligne de code est lue est passée par le **lexer** pour en former une liste de token
- La liste de Token est passée par le **parser** pour former un arbre de syntaxe abstraite (AST)
- L'AST est ensuite interprété en suivant la structure de l'arbre en profondeur (un algorithme sensiblement proche d'un DFS)

Algorithmes

Le B# est pour l'instant uniquement interprété il passe donc par plusieurs étapes standard afin d'être évalué correctement

- Le fichier source / la ligne de code est lue est passée par le **lexer** pour en former une liste de token
- La liste de Token est passée par le **parser** pour former un arbre de syntaxe abstraite (AST)
- L'AST est ensuite interprété en suivant la structure de l'arbre en profondeur (un algorithme sensiblement proche d'un DFS)

Algorithmes

Le B# est pour l'instant uniquement interprété il passe donc par plusieurs étapes standard afin d'être évalué correctement

- Le fichier source / la ligne de code est lue est passée par le **lexer** pour en former une liste de token
- La liste de Token est passée par le **parser** pour former un arbre de syntaxe abstraite (AST)
- L'AST est ensuite interprété en suivant la structure de l'arbre en profondeur (un algorithme sensiblement proche d'un DFS)

Lexer

La liste complète des tokens est disponible dans base/token.ml,

```
(**Type of tokens*)
type token_type =
  | LEFT_PARENTHESIS
  | RIGHT_PARENTHESIS
  | KEYWORD of string
  | QUOTE
  | SEMI_COLON
  | INT_TOKEN of int
  | FLOAT_TOKEN of float
  | NULL_TOKEN
  | STRING_TOKEN of string
  | BOOL_TOKEN of bool
  | ARRAY_BEGIN
  | PARAM_BEGIN
  | ARRAY_END
  | PARAM_END
  | COMMENT
  | COMMA;;
```

Figure – Type des tokens

Lexer - Algorithme

1. Le fichier d'entrée est lue et les sauts de ligne sont enlevé
2. La chaîne de caractère est coupé en une liste de mot (à chaque blanc)
3. Chaque mot est lu et est comparé à un algorithme de reconnaissance des *tokens*
4. On répète pour tous les mots et est ajouté à la liste et on répète récursivement
5. On renvoie la liste renversée (pour conserver l'ordre de lecture)

```
string -> Token.token_type list
let generate_token input_string =
  let lst = String.split_on_char ' ' input_string in
  let rec aux acc quotes lst = match lst with
    | [] -> List.rev acc
    | t::q -> let token,add = read_token t ((quotes mod 2) = 1)
      in aux (token::acc) (quotes+add) q
  in aux [] 0 lst;;
```

Figure – Implémentation du *lexer* principal

Lexer - Algorithme

1. Le fichier d'entrée est lue et les sauts de ligne sont enlevé
2. La chaîne de caractère est coupé en une liste de mot (à chaque blanc)
3. Chaque mot est lu et est comparé à un algorithme de reconnaissance des *tokens*
4. On répète pour tous les mots et est ajouté à la liste et on répète récursivement
5. On renvoie la liste renversée (pour conserver l'ordre de lecture)

```
string -> Token.token_type list
let generate_token input_string =
  let lst = String.split_on_char ' ' input_string in
  let rec aux acc quotes lst = match lst with
    | [] -> List.rev acc
    | t::q -> let token,add = read_token t ((quotes mod 2) = 1)
      in aux (token::acc) (quotes+add) q
  in aux [] 0 lst;;
```

Figure – Implémentation du *lexer* principal

Lexer - Algorithme

1. Le fichier d'entrée est lue et les sauts de ligne sont enlevé
2. La chaîne de caractère est coupé en une liste de mot (à chaque blanc)
3. Chaque mot est lu et est comparé à un algorithme de reconnaissance des *tokens*
4. On répète pour tous les mots et est ajouté à la liste et on répète récursivement
5. On renvoie la liste renversée (pour conserver l'ordre de lecture)

```
string -> Token.token_type list
let generate_token input_string =
  let lst = String.split_on_char ' ' input_string in
  let rec aux acc quotes lst = match lst with
    | [] -> List.rev acc
    | t::q -> let token,add = read_token t ((quotes mod 2) = 1)
      in aux (token::acc) (quotes+add) q
  in aux [] 0 lst;;
```

Figure – Implémentation du *lexer* principal

Lexer - Algorithme

1. Le fichier d'entrée est lue et les sauts de ligne sont enlevé
2. La chaîne de caractère est coupé en une liste de mot (à chaque blanc)
3. Chaque mot est lu et est comparé à un algorithme de reconnaissance des *tokens*
4. On répète pour tous les mots et est ajouté à la liste et on répète récursivement
5. On renvoie la liste renversée (pour conserver l'ordre de lecture)

```
string -> Token.token_type list
let generate_token input_string =
  let lst = String.split_on_char ' ' input_string in
  let rec aux acc quotes lst = match lst with
    | [] -> List.rev acc
    | t::q -> let token,add = read_token t ((quotes mod 2) = 1)
      in aux (token::acc) (quotes+add) q
  in aux [] 0 lst;;
```

Figure – Implémentation du *lexer* principal

Lexer - Algorithme

1. Le fichier d'entrée est lue et les sauts de ligne sont enlevé
2. La chaîne de caractère est coupé en une liste de mot (à chaque blanc)
3. Chaque mot est lu et est comparé à un algorithme de reconnaissance des *tokens*
4. On répète pour tous les mots et est ajouté à la liste et on répète récursivement
5. On renvoie la liste renversée (pour conserver l'ordre de lecture)

```
string -> Token.token_type list
let generate_token input_string =
  let lst = String.split_on_char ' ' input_string in
  let rec aux acc quotes lst = match lst with
    | [] -> List.rev acc
    | t::q -> let token,add = read_token t ((quotes mod 2) = 1)
      in aux (token::acc) (quotes+add) q
  in aux [] 0 lst;;
```

Figure – Implémentation du *lexer* principal

Parser/AST

L'AST en lui même est un arbre à n-enfant défini comme ceci en OCaml, bien que générique dans la définition, le type utilisé en pratique est un *parameters ast*

```
(**Type argument: every primitive type the language recognize*)
type arguments = Str of string | I of int | Nul of unit | D of float | Bool of bool;;

(**Type parameters: every structure the language recognize*)
type parameters = CallExpression of string
                | Argument of arguments
                | GOTO of string
                | LOAD of string
                | Exception of bag_exception
                | Label of string
                | IF | COND | Array
                | TBL of parameters array
                | Function of string * string list;;

(**Type AST*)
type 'a ast = Nil | Node of 'a * ('a ast) list;;
```

Figure – Type de l'AST et des paramètres

Parser - Algorithme

L'algorithme du *parser* est assez classique, bien qu'il ne soit pas exactement un algorithme connu (tout faire depuis 0 sans utiliser d'algorithme déjà utilisé) il est classique pour un parser récursif en mémorisant le *dernier token* lu pour pouvoir tirer lors du contexte, ci-après l'implémentation de quand le parser lit une parenthèse ouvrante, qui va être présenté en détail

```
let parse_line lst =
  let rec aux last_token acc lst =
    match lst with
    | [] -> lst,List.rev acc
    (*basic handling*)
    | Token.LEFT_PARENTHESIS::q ->
        (match last_token with
        (*call expression*)
        | Token.STRING_TOKEN s -> let rest,accs = aux Token.LEFT_PARENTHESIS [] q in
          let acc' = if List.length acc > 0 then List.tl acc else [] in
          (aux Token.NULL_TOKEN (Node(CallExpression s, accs)::acc') rest)

        | Token.PARAM_END -> let rest,accs = aux Token.LEFT_PARENTHESIS [] q in rest,accs
        (*remaining*)
        | _ -> aux Token.LEFT_PARENTHESIS acc q)
```

Figure – Handler de la lecture du token de parenthèse ouvrante

Parser - Parenthèse ouvrante

La condition de sortie est généralement une parenthèse fermante, en effet la validation du parenthésage est automatique lors de l'interprétation, il est donc présumé que le programme est bien parenthésé.

1. On lit une parenthèse ouvrante, on match le token précédent
2. Si c'est une chaîne de caractère on décide que c'est une instruction, une *call expression*
   - On relance l'algorithme de parsing avec un accumulateur vide (pour avoir les paramètres de l'expression)
   - On supprime la tête de la liste (qui est généralement le nom de l'instruction) si la liste est non-vide
   - On continue l'algorithme sur la suite de la liste de token en ajoutant une Node *call expression* à l'arbre accumulé
3. Si le dernier token est une accolade fermante (qui signifie la fin des paramètres d'une déclaration de fonction) alors on relance l'algorithme et on renvoie l'accumulateur et le reste de la liste de token
4. Sinon on continue l'algorithme en sautant la parenthèse

Parser - Parenthèse ouvrante

La condition de sortie est généralement une parenthèse fermante, en effet la validation du parenthésage est automatique lors de l'interprétation, il est donc présumé que le programme est bien parenthésé.

1. On lit une parenthèse ouvrante, on match le token précédent
2. Si c'est une chaîne de caractère on décide que c'est une instruction, une *call expression*
   - On relance l'algorithme de parsing avec un accumulateur vide (pour avoir les paramètres de l'expression)
   - On supprime la tête de la liste (qui est généralement le nom de l'instruction) si la liste est non-vide
   - On continue l'algorithme sur la suite de la liste de token en ajoutant une Node *call expression* à l'arbre accumulé
3. Si le dernier token est une accolade fermante (qui signifie la fin des paramètres d'une déclaration de fonction) alors on relance l'algorithme et on renvoie l'accumulateur et le reste de la liste de token
4. Sinon on continue l'algorithme en sautant la parenthèse

Parser - Parenthèse ouvrante

La condition de sortie est généralement une parenthèse fermante, en effet la validation du parenthésage est automatique lors de l'interprétation, il est donc présumé que le programme est bien parenthésé.

1. On lit une parenthèse ouvrante, on match le token précédent
2. Si c'est une chaîne de caractère on décide que c'est une instruction, une *call expression*
   - On relance l'algorithme de parsing avec un accumulateur vide (pour avoir les paramètres de l'expression)
   - On supprime la tête de la liste (qui est généralement le nom de l'instruction) si la liste est non-vide
   - On continue l'algorithme sur la suite de la liste de token en ajoutant une Node *call expression* à l'arbre accumulé
3. Si le dernier token est une accolade fermante (qui signifie la fin des paramètres d'une déclaration de fonction) alors on relance l'algorithme et on renvoie l'accumulateur et le reste de la liste de token
4. Sinon on continue l'algorithme en sautant la parenthèse

Parser - Parenthèse ouvrante

La condition de sortie est généralement une parenthèse fermante, en effet la validation du parenthésage est automatique lors de l'interprétation, il est donc présumé que le programme est bien parenthésé.

1. On lit une parenthèse ouvrante, on match le token précédent
2. Si c'est une chaîne de caractère on décide que c'est une instruction, une *call expression*
   - On relance l'algorithme de parsing avec un accumulateur vide (pour avoir les paramètres de l'expression)
   - On supprime la tête de la liste (qui est généralement le nom de l'instruction) si la liste est non-vide
   - On continue l'algorithme sur la suite de la liste de token en ajoutant une Node *call expression* à l'arbre accumulé
3. Si le dernier token est une accolade fermante (qui signifie la fin des paramètres d'une déclaration de fonction) alors on relance l'algorithme et on renvoie l'accumulateur et le reste de la liste de token
4. Sinon on continue l'algorithme en sautant la parenthèse

Parser - Parenthèse ouvrante

La condition de sortie est généralement une parenthèse fermante, en effet la validation du parenthésage est automatique lors de l'interprétation, il est donc présumé que le programme est bien parenthésé.

1. On lit une parenthèse ouvrante, on match le token précédent
2. Si c'est une chaîne de caractère on décide que c'est une instruction, une *call expression*
   - On relance l'algorithme de parsing avec un accumulateur vide (pour avoir les paramètres de l'expression)
   - On supprime la tête de la liste (qui est généralement le nom de l'instruction) si la liste est non-vide
   - On continue l'algorithme sur la suite de la liste de token en ajoutant une Node *call expression* à l'arbre accumulé
3. Si le dernier token est une accolade fermante (qui signifie la fin des paramètres d'une déclaration de fonction) alors on relance l'algorithme et on renvoie l'accumulateur et le reste de la liste de token
4. Sinon on continue l'algorithme en sautant la parenthèse

Parser - Parenthèse ouvrante

La condition de sortie est généralement une parenthèse fermante, en effet la validation du parenthésage est automatique lors de l'interprétation, il est donc présumé que le programme est bien parenthésé.

1. On lit une parenthèse ouvrante, on match le token précédent
2. Si c'est une chaîne de caractère on décide que c'est une instruction, une *call expression*
   - On relance l'algorithme de parsing avec un accumulateur vide (pour avoir les paramètres de l'expression)
   - On supprime la tête de la liste (qui est généralement le nom de l'instruction) si la liste est non-vide
   - On continue l'algorithme sur la suite de la liste de token en ajoutant une Node *call expression* à l'arbre accumulé
3. Si le dernier token est une accolade fermante (qui signifie la fin des paramètres d'une déclaration de fonction) alors on relance l'algorithme et on renvoie l'accumulateur et le reste de la liste de token
4. Sinon on continue l'algorithme en sautant la parenthèse

Parser - Parenthèse ouvrante

La condition de sortie est généralement une parenthèse fermante, en effet la validation du parenthésage est automatique lors de l'interprétation, il est donc présumé que le programme est bien parenthésé.

1. On lit une parenthèse ouvrante, on match le token précédent
2. Si c'est une chaîne de caractère on décide que c'est une instruction, une *call expression*
   - On relance l'algorithme de parsing avec un accumulateur vide (pour avoir les paramètres de l'expression)
   - On supprime la tête de la liste (qui est généralement le nom de l'instruction) si la liste est non-vide
   - On continue l'algorithme sur la suite de la liste de token en ajoutant une Node *call expression* à l'arbre accumulé
3. Si le dernier token est une accolade fermante (qui signifie la fin des paramètres d'une déclaration de fonction) alors on relance l'algorithme et on renvoie l'accumulateur et le reste de la liste de token
4. Sinon on continue l'algorithme en sautant la parenthèse

Parser version complète I

(À sauter sauf si demande du jury).
En réalité l'algorithme est bien plus gros, la parenthèse ouvrante n'est qu'un moment clef parmi d'autre, l'algorithme entier est expliqué ci-dessous si besoin.

1. On vérifie les conditions de sortie (liste vide)
2. Si on lit une parenthèse ouvrante
   - 2.1 On match le token précédent
   - 2.2 Si c'est une chaîne de caractère on décide que c'est une instruction, une *call expression*
     - On relance l'algorithme de parsing avec un accumulateur vide (pour avoir les paramètres de l'expression)
     - On supprime la tête de la liste (qui est généralement le nom de l'instruction) si la liste est non-vide
     - On continue l'algorithme sur la suite de la liste de token en ajoutant une Node *call expression* à l'arbre accumulé

Parser version complète II

2.3 Si le dernier token est une accolade fermante (qui signifie la fin des paramètres d'une déclaration de fonction) alors on relance l'algorithme et on renvoie l'accumulateur et le reste de la liste de token

2.4 Sinon on continue l'algorithme en sautant la parenthèse

3. Si on lit une accolade ouvrante (paramètre de fonction)

3.1 On match le token précédent

3.2 Si c'est une chaîne de caractère

- On relance l'algorithme de parsing avec un accumulateur vide (pour avoir les paramètres de l'expression)
- On supprime la tête de la liste (qui est généralement le nom de l'instruction) si la liste est non-vide
- On continue l'algorithme sur la suite de la liste de token en ajoutant une Node *call expression* à l'arbre accumulé

3.3 Sinon on continue en sautant l'accolade

4. Si on est sur une accolade fermante on sort avec l'accumulateur et la liste restante

Parser version complète III

5. Si on est sur un début de tableau (crochet ouvrant) alors on appelle le parser en profondeur et on continue en ajoutant une Node de type Array à l'accumulateur
6. Si on tombe sur une parenthèse fermante on sort avec l'accumulateur et la liste restante
7. Si on tombe sur une fin de tableau (crochet fermant) idem
8. Si on tombe sur un point virgule, idem
9. Si on tombe sur une virgule, on la saute (pour prendre en compte le lexer secondaire)
10. Si on tombe sur un IF alors on appelle le parser en profondeur et on ajoute une Node IF à l'accumulateur
11. Si on tombe sur un BEGIN on vérifie que le token précédent est un keyword et on continue
12. Si on tombe sur un THEN on vérifie qu'on a un IF avant et si oui on ajoute une node de type condition

Parser version complète IV

13. Si on tombe sur un LABEL alors on part en profondeur pour le nom du label et son contenu et on ajoute une node de type label
14. Si on tombe sur un GOTO alors on vérifie le nom et on ajoute une node de type GOTO
15. Idem pour un LOAD
16. Si on tombe sur un QUOTE alors on construit la chaîne de caractère jusqu'à tomber sur le quote fermant
17. Si on tombe sur un string, idem
18. Si on tombe sur un int, on ajoute une node de type argument de type int
19. Si on tombe sur un float, on ajoute une node de type argument de type float
20. Si on tombe sur un boolean, on ajoute une node de type argument de type boolean
21. Sinon on sort en renvoyant la liste et l'accumulateur

Machine de Turing Universelle Binaire - Définition I

Bien entendu créer un langage s'avère uniquement utile si on est capable d'écrire une quantité intéressante d'algorithme dans ce langage, pour cela l'outil théorique de la *Machine de Turing Universelle* est très utile.

Une machine de Turing est un *modèle abstrait*, pensé originellement par le précurseur des sciences de l'informatique et mathématicien **Alan Turing** lors de ses recherches sur la *calculabilité*, une définition intuitive d'une telle machine est par une tête de lecture et un ruban de longueur infini à droite, la tête de lecture lit la "case" et d'après un tableau d'action bouge à droite/gauche/pas, écris quelque chose sur la case (ou pas) et change la machine d'état.

Machine de Turing Universelle Binaire - Définition

Une définition formelle 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 $(K - H) \times \Sigma \to K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$ (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*)

Machine de Turing Universelle Binaire - Implémentation I

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 confondu) 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

Machine de Turing Universelle Binaire - Implémentation II

chaîne de caractère pour le ruban, tout ceci pour pouvoir être implémenté en Baguette# !

$$K = H \cup Q \qquad Q \subset \mathbb{N}$$
$$\Sigma = \{0, 1, 2\}$$
$$s = 0$$
$$H = \{\bar{H}\}$$
$$\delta_1 : Q \times \Sigma \to K$$
$$\delta_2 : Q \times \Sigma \to \Sigma$$
$$\delta_3 : Q \times \Sigma \to \{r, l, *\}$$

Machine de Turing Universelle Binaire - Implémentation en B# I

En omettant la boucle lisant l'entrée clavier pour le programme l'important est la partie *étape* dont le code en B# est disponible sur le compte Github dans le fichier *examples/turing.baguette* à la ligne 26 La boucle principale de *runtime* s'éxécute juste après la lecture du programme qui a lieu après l'initialisation, et ce tant que l'état de la machine n'est pas *H*

```
//Turing machine initialization
//Creation of three 100 by 3 matrices to represent the transition function
//Declaraton 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 , PARISBRE
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 – Initialisation de la machine

Machine de Turing Universelle Binaire - Implémentation en B# II

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

Figure – Runtime

Résultats - Bitshift

Maintenant on peut présenter quelques programmes qui tourne sur cette machine de Turing. Sous la forme d'algorithme et l'automate d'état associé. Ici un bitshift,

- 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"

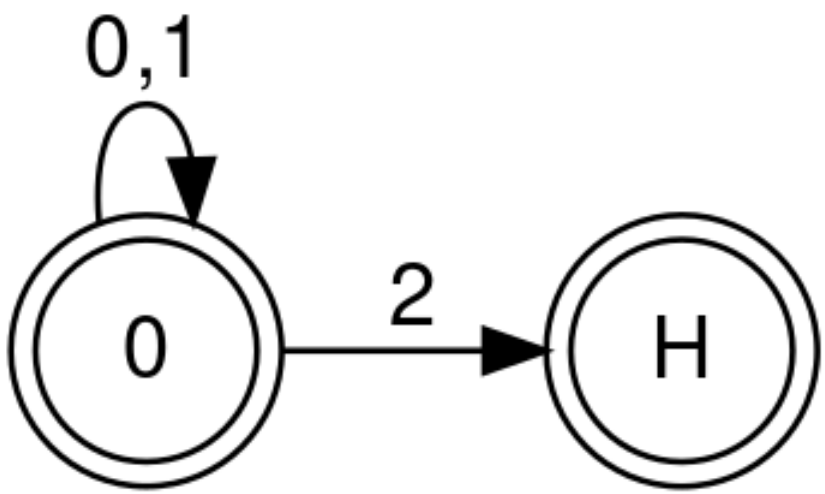

Figure – Automate d'état du bitshift

Résultats - Bitshift

Maintenant on peut présenter quelques programmes qui tourne sur cette machine de Turing. Sous la forme d'algorithme et l'automate d'état associé. Ici un bitshift,

- 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"

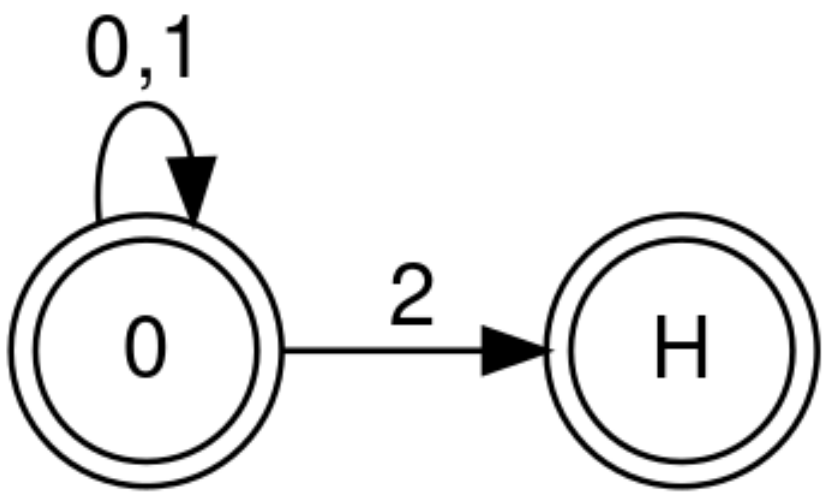

Figure – Automate d'état du bitshift

Résultats - Bitshift

Maintenant on peut présenter quelques programmes qui tourne sur cette machine de Turing. Sous la forme d'algorithme et l'automate d'état associé. Ici un bitshift,

- 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"

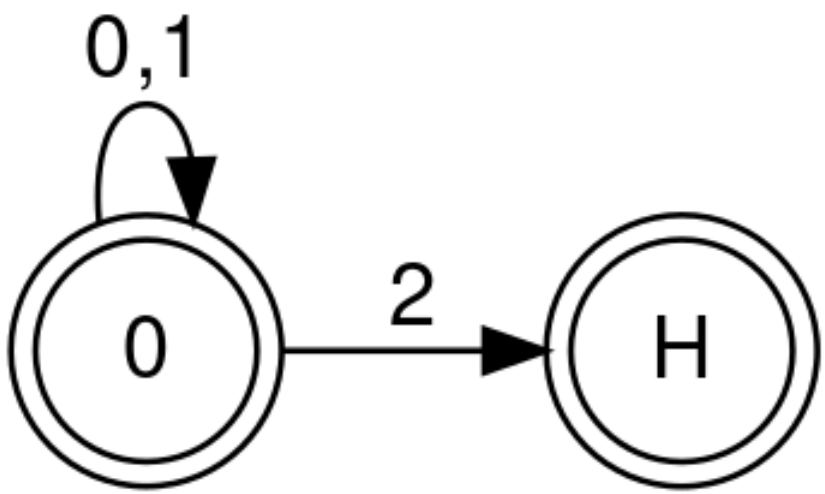

Figure – Automate d'état du bitshift

Résultats - Adder

Ou encore un binary adder qui en automate d'état rend

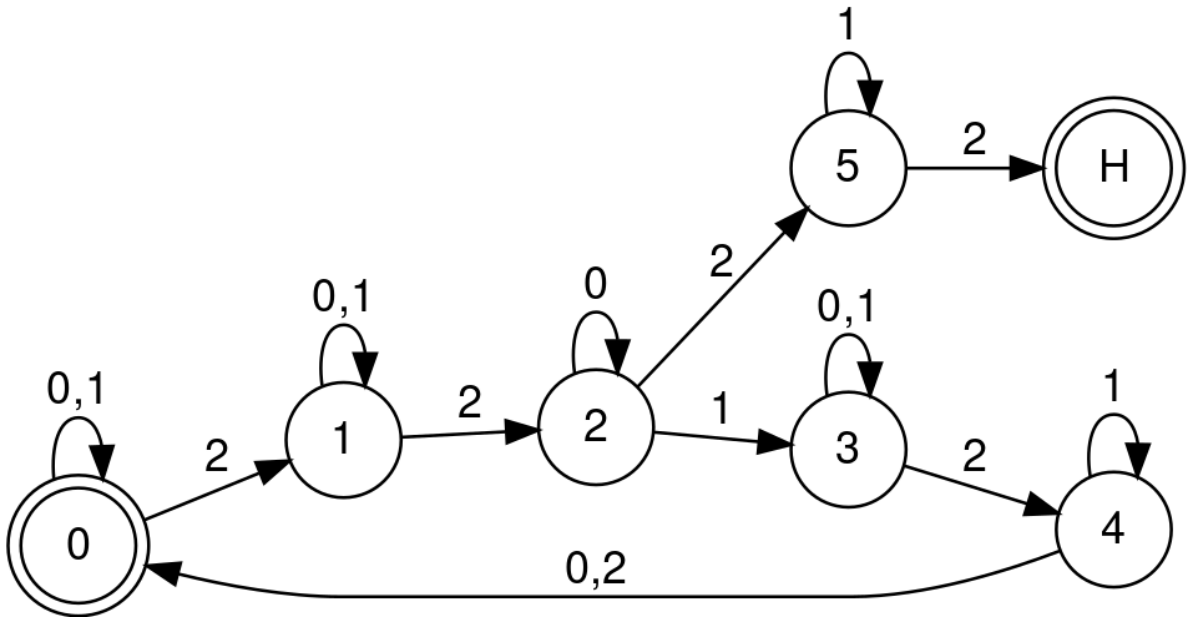

Figure – Automate d'état du binary adder