PROJET IPI :
LR1 automata recognition C-program

Justin BABONNEAU

Lundi 10 janvier 2022

Sommaire

Chapitre 1

Introduction

Le but de ce projet est d'implanter un programme qui exécute des automates LR(1). Ces derniers servent à reconnaître tout type de langage de programmation suivant un pattern (ou schéma) bien précis.

Chapitre 2

Démarche

2.1 Analyse de la structure représentant au mieux l'automate

L'automate a besoin de :
— son nombre d'états : nb_states de type state_t
— la taille de l'alphabet (supposé égale à 128 à la base mais ici non constante) : nb_characters de type uichar_t
— le chemin du fichier de l'automate : filename de type char*
— le tableau de booléens indiquant les caractères autorisés par l'automate : allowed_characters de type boolean_t*
— le tableau d'actions de l'automate : actions de type action_t*
— le tableau de la première composante de réduit : reduce_n de type reduce_n_t*
— le tableau de la deuxième composante de réduit : reduce_c de type uichar_t*
— le tableau de décale : shift de type state_t*
— le tableau de branchement : _goto (le _ est là pour éviter le problème de nom avec la fonction goto préexistante en C) de type state_t*.

Etant donné toutes les données requises par l'automate, l'adoption d'un enregistrement de type automaton_t qui permet de stocker ces éléments dans le même objet paraissait propice.

2.2 Compréhension fine du contenu d'un fichier « .aut »

L'automate a un format bien spécifique :
— son en-tête : AUT_HEADER de format "%s %u\n"
— son caractère de fin de ligne : AUT_EOL "\n"
— son caractère de fin de séquence (pour réduit et branchement) : AUT_END_OF_SEQUENCE "\255\255\255\255\255\255"

2.3 Réflexion sur la pile évoquée dans le pseudo-code proposé : quel contenu ? pourquoi ? le plus efficace ?

La pile utilisée contient des états de type state_t car on souhaite pouvoir dépiler des états grâce à réduit. De plus, elle correspond ici à un tableau dont on ne peut récupérer que l'élément du sommet en dépilant ou empiler un élément. C'est ici le plus efficace car songer à des listes chaînées ferait intervenir un embriquement compliqué pour un simple stockage d'états (autrement dit d'entiers).

2.4 Analyse et mise en place de l'algorithme

On commence par récupérer toutes les données liées à l'automate puis à réaliser la fonction de vérification des mots pour l'automate et enfin par tout ce qui se rapporte aux graph DOT. On s'est concentré après cela sur une généralisation de ce que pouvait faire l'algorithme :
— ajout des couleurs pour rendre le programme plus compréhensible à l'œil
— ajout de l'ensemble des caractères autorisés
—

— passage à un alphabet de taille non constante
— passage à l'ensemble des caractères Unicode (contrairement à seulement ASCII)
— passage à des caractères codées sur plusieus octets

2.5 Analyse du besoin « IHM » (interface homme-machine)

Afin de pouvoir réaliser de l'auto-complétion et un historique sur les commandes du programme et avoir une interface, on a décidé d'utiliser de reprendre un squelette de l'interface readline qui avait déjà été codée et on l'a adaptée pour intégrer les fonctions relatives à la manipulation des automates.

Chapitre 3

Découpage en modules

3.1 Graphe des dépendances

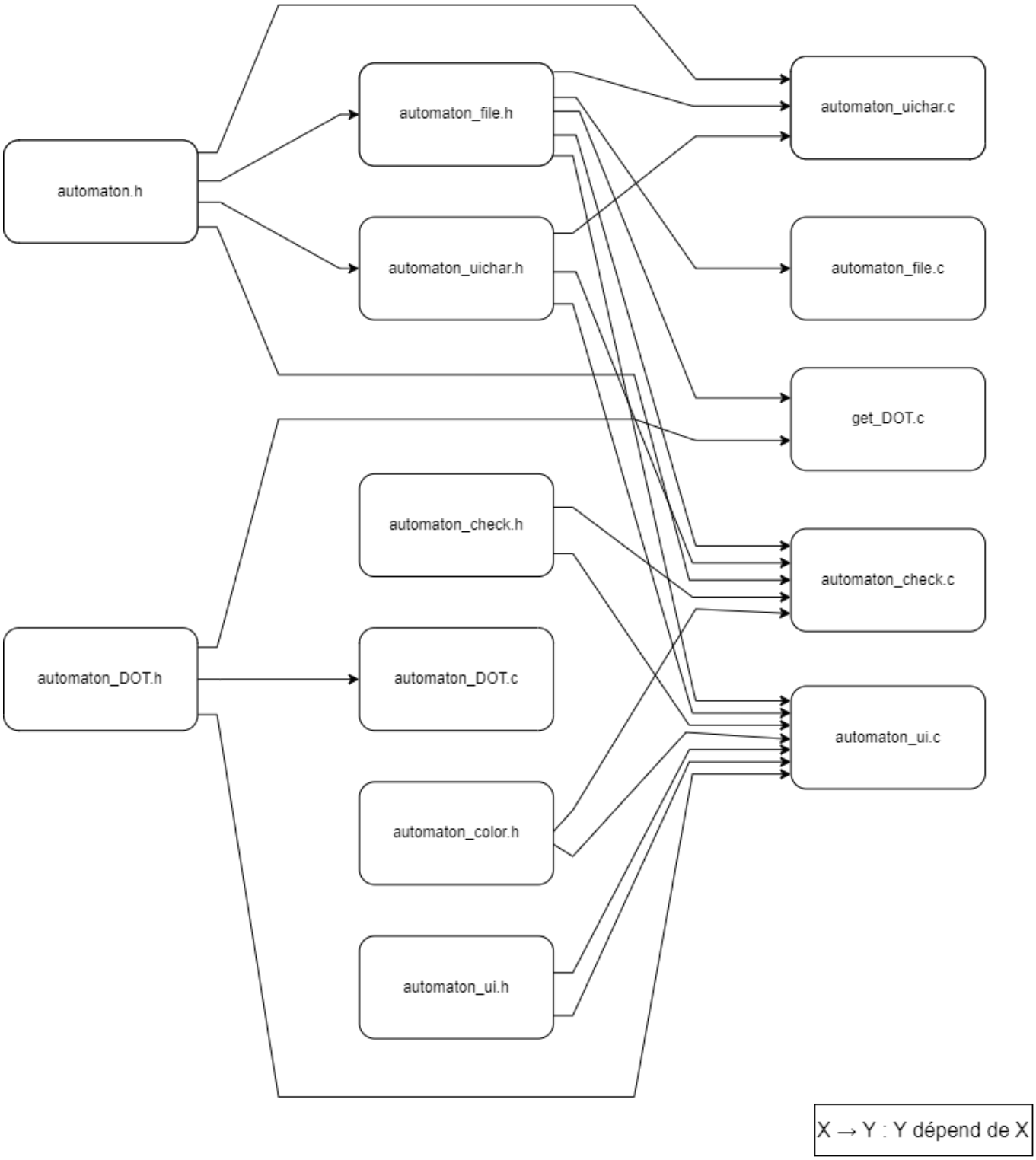


FIGURE 3.1 – Dépendances inter-module

3.2 La gestion de l'objet automate et son peuplement depuis le fichier

La gestion de l'objet automate et son peuplement depuis le fichier sont gérés par automaton_file.c. Ce dernier contient :

- une fonction de debug : dumpAutomaton
- une fonction de libération de la mémoire : freeAut
- une fonction d'allocation de la mémoire liée à l'enregistrement de l'automate : allocAutomaton
- la fonction qui lit le fichier de l'automate : loadAutomatonFromFile

3.3 Mise en œuvre de l'algorithme de l'automate

Ceci est réalisé par la fonction isword de automaton_check.c qui vérifie si la chaîne entrée correspond à un mot lisible par l'automate.

3.4 L'interface utilisateur reprise et adaptation de Fileman

Cela est effectué par automaton_ui.c Il a fallu y rajouter les fonctions de chargement de l'automate com_load_automaton_from_file, d'utilisation du programme com_automaton_isword (vérification de compatibilité d'un mot avec l'automate), com_automaton_DOT (génération du graphe DOT) et com_automaton_DOT_x (affichage du graphe DOT) et com_automaton_version (version du programme de lecture des automates LR1) de suivi de la version du programme.

3.5 Analyse du format « DOT » et déclinaison à partir de la structure choisie pour l'automate

L'analyse du format DOT est faite par automaton_DOT.c (qui construit le code du graphe de l'automate) et get_DOT.c (qui construit les fichiers « .dot »).

3.6 Couleurs pour l'affichage

Tout le nécessaire pour colorer les messages est contenu dans automaton_color.h.

Chapitre 4

Difficultés rencontrées

4.1 Analyse du fichier « .aut »

L'analyse du fichier « .aut » a lieu en 6 étapes :

1. récupération de l'en-tête au format AUT_HEADER (fscanf)
2. lecture des nb_states×nb_characters actions (fread) et d'un AUT_EOL (fscanf)
3. obtention des nb_state reduce_n (fread) puis passage d'un AUT_EOL (fscanf)
4. récupération des nb_state reduce_c (fread) suivi d'un AUT_EOL (fscanf)
5. lecture de nb_states×nb_characters états de décale se terminant par un AUT_END_OF_SEQUENCE
6. obtention de nb_states×nb_characters états de branchement se terminant par un AUT_END_OF_SEQUENC

4.2 Fin de séquence « \255\255\255 » en octal différent de « \xFF\xFF\xFF » (octal \377\377\377)

On a opté pour une solution portant sur l'analyse hexadécimale du contenu d'un fichier « .aut » (# od –t x1 <fichier>) afin de confirmer la valeur octale « \255 »

4.3 Mise en œuvre de l'algorithme

Dans la mise en œuvre de l'algorithme, il faut bien penser dans le cas du « decale » à conserver l'information de « s' alpha » avant de faire le décalage (sinon on ne peut aboutir).

4.4 Affichage de la restitution de l'analyse visuellement parlant

Afin d'obtenir un résultat explicite à l'œil et relativement stylisé, on a fait le choix d'opter pour une solution faisant intervenir un affichage coloré sur les terminaux « modernes » ce qui utilise des fonctions avancées des terminaux (inconvénient d'enlever la retro-compatibilité).

4.5 Construction du « .dot »

On a rencontré des difficultés à constituer les chaînes de caractères contigus pour une même transition (éviter de citer a,b,c,d,e,f,g,...z, et privilégier a-Z, A-Z). Cependant, la solution nécessitait de ne pas réafficher les caractéres déjà traités dans les cas de transition précédents : c'est pourquoi un tableau état correspondant à chaque caractère et à chaque état a été mis en place.

4.6 A l'affichage, problème des caractères parasites (hors de l'ASCII) entrés par l'utilisateur

A la base, l'affichage des caractères hors ASCII était impossible (cela causait un crash : "Segmentation fault"). En revanche, la solution utilisée ici prépare l'évolution vers un alphabet autre

que ASCII (stockage des caractères sur des « unsigned int » plutôt que des « char » ce qui a forcé à ajouter des fonctions de conversion de 'chaînes de char' aux 'chaînes de 'unsigned int'.

4.7 Adaptation du chargement du fichier « .aut » à un alphabet multibyte (unicode)

La solution retenue repose sur l'hypothèse d'un format étendu (par exemple le premier 'a' du fichier donnant le nombre byte de l'alphabet (ici un seul). Mais on s'arrête à la limite des états stockés sur un byte dans le fichier. Mais on pourrait songer à avoir davantage d'états.

4.8 Affichage X11 pas si simple à gérer

Sur Windows, l'affichage des graphes est un peu compliqué à mettre en place. Il faut en effet créer un serveur X11 (tests réalisés avec « VcXsrv » et puis « VNC Server ») et paramétrer correctement le programme et la variable d'environnement DISPLAY.

4.9 Passer en tâche de « fond » du lancement de Graphviz 'dot' pour l'affichage X11

Il ne fallait pas oublier l'esperluette () lors de l'utilisation de la fonction dot de Graphviz pour laisser la main à l'utilisateur après l'utilisation de !xDOT.

Chapitre 5

Limitations et améliorations envisagées

5.1 Etats de l'automate sur 1 octet dans le fichier « .aut »

On a limité le nombre de l'états de l'automate à $256 = 2^8$ (de 0 à 255) donc stocké sur 1 octet mais il se pourrait qu'il y en ait davantage.

5.2 Alphabet multibytes à « peaufiner »

L'alphabet de l'automate pourrait contenir des caractères sur plus de 4 octets ce qui ne fonctionnerait pas en l'état.

5.3 Affichage « color » à durcir pour un fonctionnement plus robuste

Lorsque un crash survient ou que l'utilisateur interrompt la session (en particulier sur Linux), le shell de l'utilisateur conserve la couleur de la dernière chaîne de texte affichée. On n'a pas réussi à résoudre cela mais cela pourrait être une piste d'amélioration : on voudrait qu'un printf(COLOR_RESET) ait lieu en cas de crash ou d'interruption causée par l'utilisateur. Cependant, ce problème n'a pas lieu sur WSL et lors d'un arrêt propre du binaire avec la commande !quit.

Annexe A

Annexes

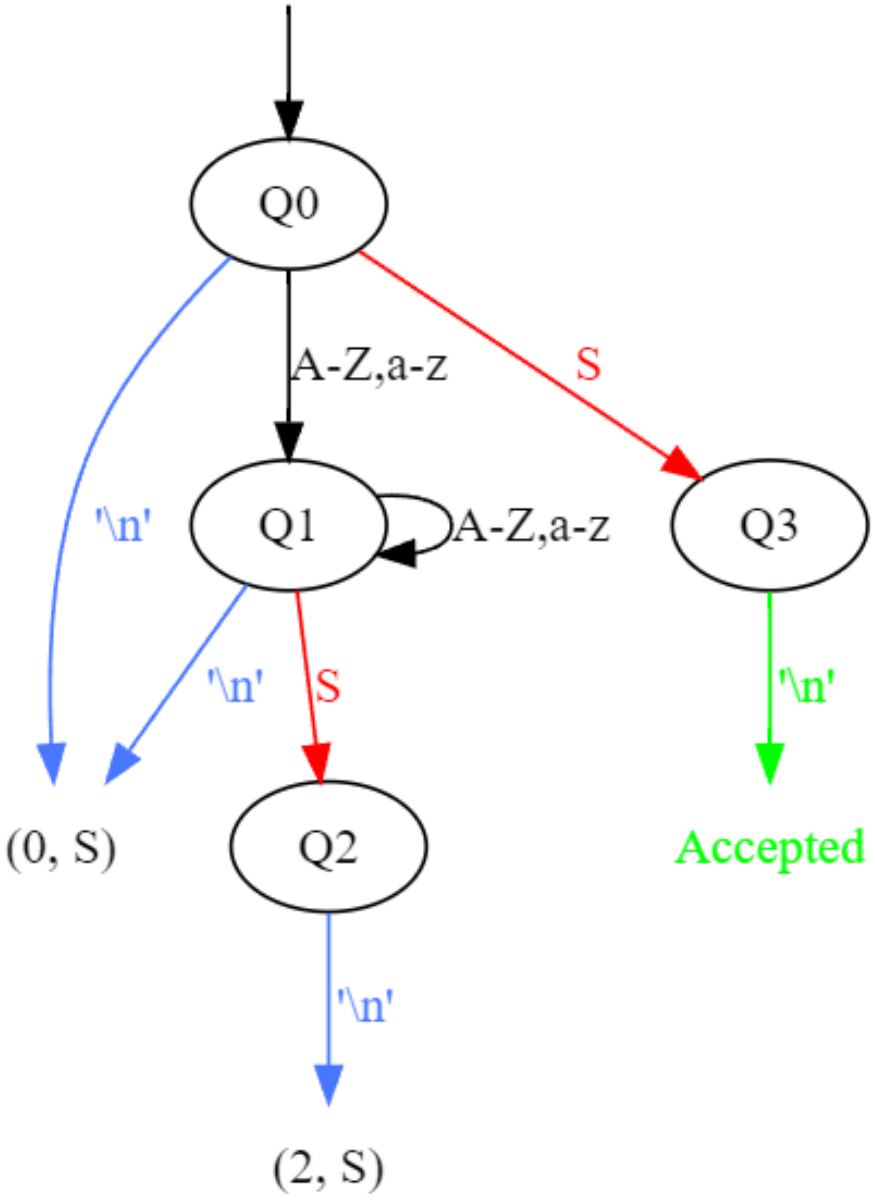


Figure A.1 – Graphe DOT de word.aut

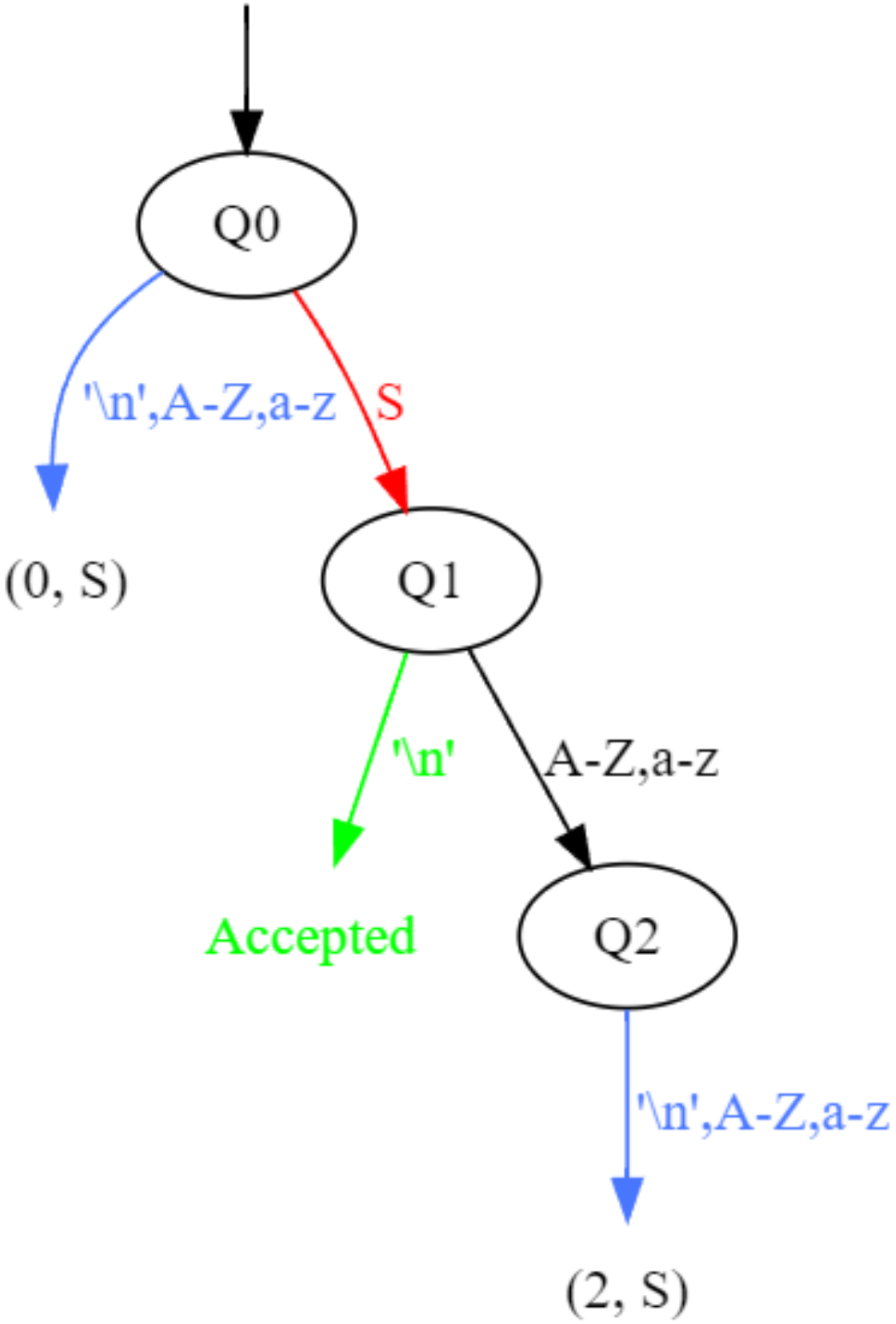
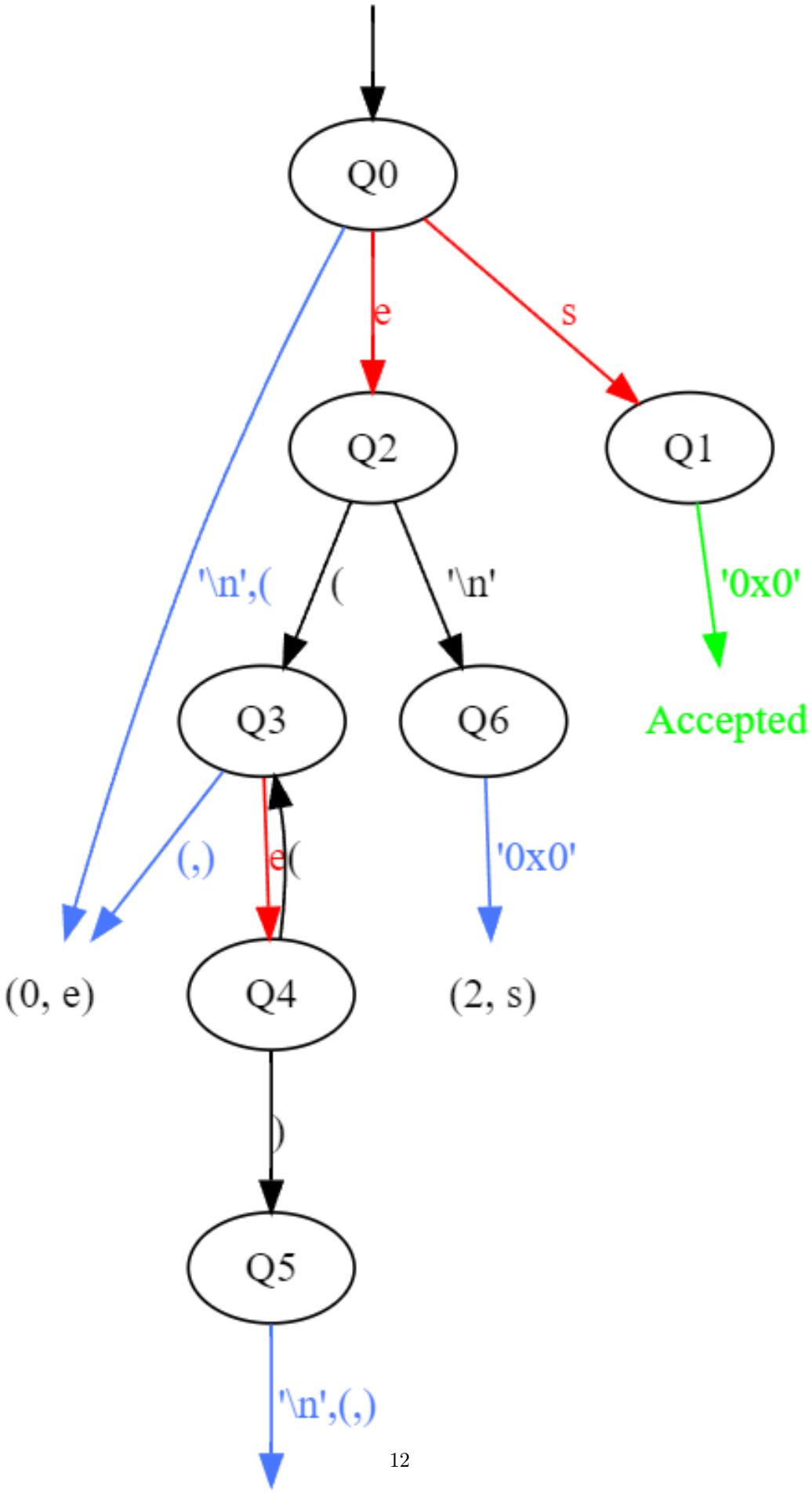


FIGURE A.2 – Graphe DOT de word_bis.aut


Q0
e
s
Q2
Q1
'\n',(
(
'\n'
'0x0'
Q3
Q6
Accepted
(,)
e(
'0x0'
(0, e)
Q4
(2, s)
)
Q5
'\n',(,)

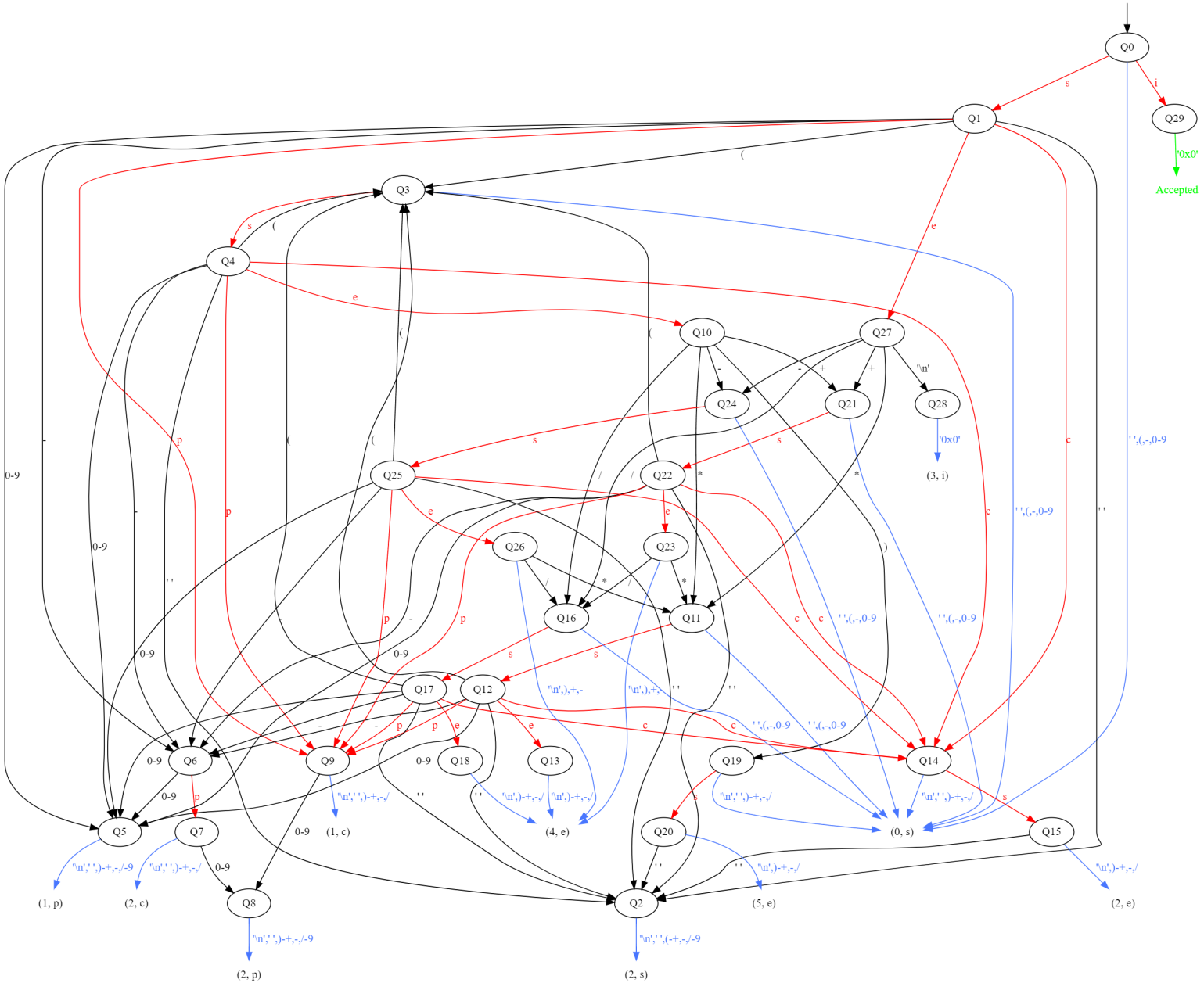


FIGURE A.4 – Graphe DOT de arith.aut