TP 1: Mise en œuvre d'un analyseur lexical pour le langage C⋆

1 But de cette séance de travaux pratiques

Le but de cette séance de TP est de réaliser un analyseur lexical pour un langage appelé C⋆ qui est un sous-ensemble strict du langage C. Cette séance est l'occasion de mettre en œuvre l'algorithme vu en cours pour la réalisation d'un analyseur lexical. On vous rappelle qu'il repose sur l'utilisation d'un automate déterministe à états finis. Afin d'accélérer votre développement nous allons vous fournir une grande partie du code nécessaire à la réalisation de votre compilateur.

2 Le langage C⋆

C⋆ est un langage directement inspiré du langage C. Il est cependant largement simplifié puisqu'un certain nombre de constructions présentes dans le langage C originel ont été supprimées afin de rendre possible la réalisation d'un compilateur pour ce langage dans le temps imparti à la mineure. De ce fait, un programme écrit en C⋆ peut être directement compilé par un compilateur C standard (*e.g.* GCC ou `clang`). C'est d'ailleurs ce que vous pourrez faire pour compiler votre analyseur lexical afin d'obtenir un binaire au format ELF pour un processeur x86. Vous utiliserez pour cela le compilateur GCC fourni sur votre poste de travail.

Le langage C⋆ présente les caractéristiques qui lui sont propres et qui sont détaillées dans la suite.

2.1 Types de données

C⋆ supporte seulement deux types de variables notés `uint64_t` et `uint64_t*` qui sont respectivement les entiers non signés codés sur 64 bits, et les pointeurs sur des entiers non signés 64 bits. Comme on suppose que l'architecture cible de notre compilateur est le processeur RISC-V dans sa version 64 bits, les deux types occupent tous deux exactement le même espace en mémoire (à savoir 64 bits). L'annotation de type `uint64_t` n'est pas standard en langage C, et nécessite de ce fait d'être indiqué par un moyen ou un autre au compilateur (ici GCC) qui va générer le compilateur. C'est pourquoi si vous consultez le fichier `Makefile` qui construit le compilateur vous pourrez vous rendre compte que l'option `-Duint64_t='unsigned long long'` est passée à GCC afin de lui indiquer que le type `uint64_t` est équivalent au type `unsigned long long`. L'option `-D` de GCC permet d'ajouter une directive de préprocesseur du type `#define` en début de chaque fichier traité par le compilateur, ici donc `#define uint64_t 'unsigned long long'`. La chaîne de caractères `unsigned long long` se verra donc substituée à toute occurrence de `uint64_t` lors de la phase de préprocessing de GCC.

Le langage C⋆ ne supporte pas non plus les types de données structurés du langage C comme :

— les structures de données habituellement introduites par le mot-clé `struct` ;

— les unions habituellement introduites par le mot-clé `union` ;

— les énumérations habituellement introduites par le mot-clé `enum`.

Le plus restrictif dans ce choix de conception (qui permet de limiter la complexité du compilateur) est certainement l'absence de structures de données (mot-clé `struct`) puisque celles-ci restent indispensables lors de la conception de tout logiciel un peu complexe. On est donc obligé de simuler l'existence des structures de données en ayant recours à l'arithmétique de pointeurs, disponible en C⋆. Un exemple de simulation d'une structure de données (ici une pile de fichiers) est disponible dans le fichier `preprocessor.c`. Dans cet exemple, on cherche à obtenir l'équivalent en C⋆ du code C suivant :

```
struct source_file_stack {
  source_file_stack *next;
  char *source_filename;
  int source_file_fd;
}
```

Dans le fichier `preprocessor.c`, cela se traduit par le code suivant :

```
// included file:
// +---+-----------+
// | 0 | next                        | pointer to next entry
// | 1 | source_filename    | pointer source file name
// | 2 | fd                             | source file descriptor

uint64_t* get_next(uint64_t *entry) { return (uint64_t *) *entry;}
uint64_t* get_source_filename(uint64_t *entry) { return (uint64_t*) *(entry+1);}
uint64_t get_fd(uint64_t *entry) { return *(entry+2);}

void set_next(uint64_t *entry, uint64_t *next) { *entry = (uint64_t) next; }
void set_source_filename(uint64_t *entry, uint64_t *filename) {*(entry+1)=(uint64_t)filename;}
void set_fd(uint64_t *entry, uint64_t fd) { *(entry+2) = fd; }
```

Le code documente la structure de données, en indiquant le nom des champs de la structure ainsi que le déplacement à effectuer depuis le début de la structure de données pour accéder à chacun des champs (en terme d'arithmétique de pointeurs). Puis, le développeur donne le code trois fonctions permettant d'accéder à chacun des chacun des champs de la structure en lecture puis en écriture. Si vous devez introduire de nouvelles structures de données (ce qui n'est pas strictement nécessaire pour terminer le compilateur), nous vous conseillons fortement de procéder de la même manière.

Si vous devez allouer une structure de données vous devrez appeler la fonction smalloc de la mini-librairie C et vous devrez calculer manuellement la taille en octets de la structure à allouer. Le mot-clé sizeof n'est pas supporté dans C⋆. À titre d'exemple, vous trouverez dans le fichier `preprocessor.c` du code qui alloue une structure de pile (telle que définie précédemment) et qui ressemble à ceci :

```
uint64_t *entry;
entry = smalloc(2*SIZEOFUINT64STAR + SIZEOFUINT64);
```

On alloue ainsi une zone de mémoire contiguë permettant de stocker deux pointeurs et un entier.

2.2 Instructions

Le langage C⋆ dispose des instructions et constructions sémantiques suivantes :

- Déclaration de variables de types `uint64_t` et `uint64_t*`.
- Déclaration de tableaux de types `uint64_t` et `uint64_t*` de longueur connue par avance.
- Affectation à une variable d'une valeur calculée au moyen d'une expression arithmétique.
- Affectation à un élément d'un tableau d'une valeur calculée au moyen d'une expression arithmétique.

— Pointeurs : il est possible d'allouer de la mémoire dynamiquement, et d'utiliser l'arithmétique de pointeurs.

— Instructions conditionnelles de type `if-then` ou `if-then-else` avec la même syntaxe que celle du langage C, à savoir que les alternatives `then` et `else` peuvent comporter soit une unique instruction, soit un nouveau bloc débuté par une accolade ouvrante ({) et terminé par une accolade fermante (}).

— Boucle : la seule boucle disponible est la boucle `while` qui comme en C, peut comporter soit une unique instruction, soit un nouveau bloc débuté par une accolade ouvrante ({) et terminé par une accolade fermante (}).

— Définition de fonction : il est possible de déclarer de nouvelles fonctions dont le type de retour peut être soit `void` pour indiquer que la fonction ne retourne aucune valeur, soit `uint64_t` pour indiquer qu'elle retourne un entier non signé sur 64 bits, soit `uint64_t*` pour indiquer qu'elle retourne un pointeur. La fonction doit avoir un nombre fixe d'arguments chacun possédant un nom et un type (parmi `uint64_t` et `uint64_t*`).

— Retour de valeur : une fonction qui retourne une valeur doit utiliser l'instruction `return` pour ce faire.

— Appel de fonctions : il est possible d'appeler une fonction en tant qu'instruction ou bien en tant qu'élément d'une expression (pour les fonctions qui retournent une valeur). On doit alors passer le bon nombre d'arguments attendus ainsi que des arguments du même type que celui attendu.

— Les instructions sont séparées par des points virgule (`;`).

2.3 Logique booléenne

Le langage C⋆ n'inclut pas les opérateurs de logiques booléennes (`&&`, `||`, `!`) afin de limiter le nombre d'instructions assembleur à gérer dans la partie arrière du compilateur. Cette fonctionnalité pourrait être assez facilement ajoutée dans une version plus évoluée du langage et du compilateur, mais cela ajouterait à la complexité de la réalisation. Les seules opérateurs de comparaison et de test disponibles sont :

— `==` : pour tester l'égalité de deux valeurs entières (ou deux adresses).

— `!=` : pour tester la différence entre deux valeurs entières (ou deux adresses).

— `<` : pour tester l'infériorité stricte entre deux valeurs entières (ou deux adresses).

— `<=` : pour tester l'infériorité large entre deux valeurs entières (ou deux adresses).

— `>` : pour tester la supériorité stricte entre deux valeurs entières (ou deux adresses).

— `>=` : pour tester la supériorité large entre deux valeurs entières (ou deux adresses).

L'absence du et logique (`&&`) et du ou logique (`||`), est cependant assez pénalisante dans l'écriture de conditions complexes pour une conditionnelle ou une boucle `while`. Voici comment pallier à cette absence. Vous devez décomposer l'expression conditionnelle à tester et recourir à des motifs reposant uniquement sur la conditionnelle `if-then-else`. Par exemple pour tester la condition $\texttt{condition}_1$ `&&` $\texttt{condition}_2$, vous utiliserez la portion de code :

```
if(condition₁){
  if(condition₂){
    // Code à exécuter quand la condition condition₁ && condition₂ est vraie
  }
}
```

Si vous devez mettre en œuvre une condition du type `condition`$_1$ `||` `condition`$_2$, vous utiliserez la portion de code :

```
if(condition_1){
  // Code à exécuter quand la condition condition_1 || condition_2 est vraie
}
else{
    if(condition_2){
     // Code à exécuter quand la condition condition_1 || condition_2 est vraie
   }
}
```

Enfin si vous devez mettre en œuvre une condition du type `!condition`, vous pourrez utiliser le code suivant :

```
if(condition){
}
else{
     // Code à exécuter quand la condition !condition est vraie
}
```

Vous pourrez définir, si le besoin s'en fait sentir, des fonctions qui simulent le comportement des opérations booléennes `&&`, `||` et `!`.

Le langage C⋆ ne permet pas non plus de manipuler les valeurs bit à bit avec les opérateurs classiques du langage C tels que `&`, `|`, `!`, » ou «. Vous ne devriez cependant pas avoir besoin de ces opérateurs pour mettre en œuvre le compilateur.

2.4 Librairie C

Afin de pallier à certaines constructions manquantes dans le langage C⋆, le compilateur est livré avec une mini librairie C (fichiers `library.c` et `string.c`) qui fournit des fonctions implémentant certaines fonctionnalités manquantes dans le langage. La mini librairie C fournie par le compilateur contient (entre autres) fonctions qui pourraient vous être utiles lors du développement du compilateur. Ces fonctions peuvent être regroupées en différentes catégories.

Initialisation de la librairie La fonction `void init_library()` doit être appelée au démarrage de tout programme afin d'initialiser les données utilisées en interne par la librairie. Si l'on oublie de réaliser cet appel, il est plus que probable que votre programme se terminera par une erreur de segmentation lors d'un appel à la librairie C.

Manipulation des entiers (notamment signés)

- `uint64_t left_shift(uint64_t n, uint64_t b)` et `uint64_t right_shift(uint64_t n, uint64_t b)` : opérations de décalage bit à bit respectivement à gauche et à droite qui sont absentes du langage.
- `uint64_t abs(uint64_t n)` : calcul de la valeur absolue d'un nombre en supposant que le paramètre qui est passé est signé et encodé en complément à deux.
- `uint64_t signed_less_than(uint64_t a, uint64_t b)` permet d'implémenter les comparaisons sur des entiers signés (en supposant que les deux paramètres sont des entiers signés encodés en complément à deux).
- `uint64_t is_signed_integer(uint64_t n, uint64_t b)` : permet de tester si un entier `n` est signé en supposant qu'il est encodé par un complément à deux sur `b` bits.

— `uint64_t sign_extend(uint64_t n, uint64_t b)` : étend un entier `n` en entier signé sur `b` bits.

Manipulation des chaînes de caractères

— `uint64_t load_character(uint64_t* s, uint64_t i)` et `uint64_t* store_character(uint64_t* s, uint64_t i, uint64_t c)` : Le langage C⋆ ne fournit pas le type `char`. Les chaînes de caractères du langage C (du type `char *`) doivent donc être simulées par des tableaux d'entiers (de type `uint64_t *`). La première fonction permet de retrouver le $\texttt{i}^{\texttt{ième}}$ caractère stocké dans le tableau `s` en utilisant les 8 octets de chaque entrée du tableau pour stocker jusqu'à 8 caractères par entrée. La seconde fonction permet de stocker un caractère `c` à la position `i` dans le tableau `s`.

— `uint64_t string_length(uint64_t* s)` : calcule la longueur d'une chaîne de caractères avec la convention de stockage adoptée pour les chaînes de caractères.

— `uint64_t* string_copy(uint64_t* s)` : permet d'obtenir une nouvelle chaîne de caractères qui est une copie de celle passée en paramètre.

— `void string_reverse(uint64_t* s)` : renverse sur place une chaîne de caractères (sans avoir besoin d'en allouer une nouvelle).

— `uint64_t string_compare(uint64_t* s, uint64_t* t)` : comparaison de chaîne de caractères. Si la fonction renvoie la valeur 0, alors les deux chaînes sont identiques.

— `uint64_t atoi(uint64_t* s)` : fonction qui permet de convertir une chaîne de caractères qui correspond à l'écriture en base 10 d'un entier vers cet entier.

— `uint64_t* itoa(uint64_t n, uint64_t* s, uint64_t b, uint64_t a)` : fonction inverse de la précédente qui convertit un entier `n` (dans une base `b` qui doit être 2,4,8,10 ou 16) vers la chaîne de caractères qui correspondante, en respectant un alignement précisé par le paramètre `a`.

Sortie (formatée) vers la sortie sélectionnée L'ensemble des fonctions de sortie émettent leur sortie vers un descripteur de fichiers qui doit être renseigné via une variable globale [1] nommée `output_fd`. Il faut donc bien penser à positionner cette variable avant d'utiliser les fonctions en question. Concernant les sorties formatées (à la « `printf` »), les fonctions utilisent des chaînes de format qui supporte les options de format suivantes :

— `%%` : Affichage d'un pourcentage.

— `%c` : Affichage d'un caractère.

— `%s` : Affichage d'une chaîne de caractères.

— `%d` : Affichage d'un entier (en base 10).

— `%x` : Affichage d'un entier (en base 16).

— `%o` : Affichage d'un entier (en base 8).

— `%b` : Affichage d'un entier (en base 2).

— `%p` : Affichage d'une adresse mémoire (en hexadécimal).

1. Ceci mériterait d'être modifié pour passer le descripteur de fichiers en paramètres et sera traité dans une version ultérieure.

Comme le langage C⋆ ne supporte pas les fonctions variadiques [2] à l'image de la fonction `printf` du langage C, l'affichage formaté est décliné selon 6 fonctions qui ont un nombre fixe d'arguments à afficher (de un à six) :

- `void put_character(uint64_t c)` : écrit un caractère dans le fichier qui correspond à la sortie courante.
- `void print(uint64_t* s)` : écrit une chaîne de caractères dans le fichier qui correspond à la sortie courante.
- `void println()` : écrit un retour à la ligne dans le fichier qui correspond à la sortie courante.
- `void print_character(uint64_t c)` : écrit un caractère encadré par des guillemets simples (*simple quote*) dans la sortie courante.
- `void print_string(uint64_t* s)` : écrit une chaîne de caractères encadrés par guillemets doubles (*double quote*) dans la sortie courante.
- `void print_integer(uint64_t n)` : écrit un entier dans la sortie courante.
- `void printf1(uint64_t* s, uint64_t* a1)` : sortie formatée utilisant une chaîne de format passée par l'argument `s` et ne possédant qu'un unique paramètre de format (commençant par %). L'argument à afficher est passé via le paramètre `a1`.
- `void printf2(uint64_t* s, uint64_t* a1, uint64_t* a2)` : sortie formatée utilisant une chaîne de format possédant deux paramètres de format (commençant par %). Les deux arguments à afficher est passés via les paramètres `a1` et `a2`.
- `void printf3(uint64_t* s, uint64_t* a1, uint64_t* a2, uint64_t* a3)` : sortie formatée utilisant une chaîne de format possédant trois paramètres de format (commençant par %). Les trois arguments à afficher est passés via les paramètres `a1`, `a2` et `a3`.
- `void printf4(uint64_t* s, uint64_t* a1, uint64_t* a2, uint64_t* a3, uint64_t* a4)` : sortie formatée utilisant une chaîne de format possédant quatre paramètres de format (commençant par %). Les quatre arguments à afficher est passés via les paramètres `a1`, `a2`, `a3` et `a4`.
- `void printf5(uint64_t* s, uint64_t* a1, uint64_t* a2, uint64_t* a3, uint64_t* a4, uint64_t* a5)` : sortie formatée utilisant une chaîne de format possédant cinq paramètres de format (commençant par %). Les cinq arguments à afficher est passés via les paramètres `a1`, `a2`, `a3`, `a4` et `a5`.
- `void printf6(uint64_t* s, uint64_t* a1, uint64_t* a2, uint64_t* a3, uint64_t* a4, uint64_t* a5, uint64_t* a6)` : sortie formatée utilisant une chaîne de format possédant six paramètres de format (commençant par %). Les six arguments à afficher est passés via les paramètres `a1`, `a2`, `a3`, `a4`, `a5` et `a6`.

Gestion dynamique de la mémoire La gestion dynamique de la mémoire se résume à son allocation. La désallocation (via une fonction telle que `free` en langage C) n'est pas encore supportée par le compilateur (et surtout par la mini librairie C). Ceci est tolérable dans la mesure où l'on souhaite ici fabriquer un logiciel dont le temps d'exécution est borné et donc la totalité de la mémoire qu'il aura alloué sera libérée à la fin de son exécution. Les fonctions d'allocation sont les suivantes :

2. Fonction possédant un nombre variables d'argument

— `uint64_t* smalloc(uint64_t size)` : Allocation d'une zone mémoire de taille `size`. Attention la sémantique de cette fonction diffère de la fonction `malloc` du langage C dans la mesure où en cas d'erreur d'allocation (plus assez de mémoire), la fonction termine le programme plutôt que de renvoyer le pointeur nul. Par ailleurs, elle peut renvoyer n'importe quelle adresse (et en particulier l'adresse $0x0$) dans le cas où le paramètre `size` vaut zéro.

— `uint64_t* zalloc(uint64_t size)` : Allocation d'une zone mémoire de taille `size` remplie d'octets nuls. Cette fonction possède la même sémantique que la précédente en cas d'erreur d'allocation, et dans le cas où le paramètre `size` vaut zéro.

2.5 Compilation séparée

Le compilateur que vous devez construire ne supporte pas la compilation séparée. Ceci signifie qu'il ne sera pas capable de produire des fichiers objets (fichier au format ELF et d'extension `.o`) qui pourraient à leur tour être réunis pour former un exécutable autonome. Il ne sera capable de produire que des fichiers directement exécutables (au format ELF). En particulier, nous n'attendons pas que votre compilateur fournissent de fonctionnalités d'éditions de liens. Par contre, il devra pouvoir compiler d'un seul coup un ensemble de fichiers (unités de compilation) qui pris séparément ne peuvent pas être compilés pour produire directement un exécutable, mais qui forment, considérés ensemble, un programme complet (c'est une première étape pour pouvoir supporter la compilation séparée mais ce n'est pas suffisant).

Le mot clé `extern` Par ailleurs, le langage C⋆ supporte (tout comme le langage C) le mot-clé `extern`. Ce mot-clé permet de déclarer l'existence d'une variable ou d'une fonction avant sa définition complète. Ceci est très utile pour la compilation séparée (comme le supporte GCC pour le langage C), ou la compilation en une fois d'un ensemble d'unités de compilation (comme le supporte le compilateur que vous devez produire pour le langage C⋆). En effet lorsque l'on découpe une application en une multitude d'unités de compilation, il n'est pas rare qu'une unité de compilation fasse référence à une fonction ou une variable (globale) définie dans une autre unité de compilation. Dans ce cas, il est utile pour le compilateur de pouvoir connaître l'adresse de cette variable ou de cette fonction.

Le cas des variables Pour les variables et dans le cas bien particulier du langage C⋆, il est facile de déterminer l'adresse d'une variable en réservant l'espace voulu dès sa première occurrence car tous les types occupent le même espace en mémoire (64 bits). Cependant, on voit bien que cela ne fonctionne plus en langage C, car tous les types n'ont pas la même taille (pensez aux structures de données par exemple). De ce fait par souci de compatibilité avec le langage C et le compilateur GCC on va s'astreindre à déclarer les variables avec leur type avant de les utiliser.

Le cas des fonctions Pour les fonctions la situation en langage C⋆ est la même qu'en langage C. On ne peut pas connaître la taille qu'occupera le code d'une fonction avant d'avoir compilé cette fonction. Et on ne peut pas lui allouer une adresse précise sans connaître la taille qu'occupera son code. Ce problème est résolu de manière très différente par les compilateurs de langage C et par le compilateur que vous allez développer :

— En langage C, la plupart des compilateurs (sinon tous) vont supporter la compilation séparée. C'est donc l'éditeur de liens qui corrigera durant la phase d'éditions de liens l'ensemble des appels à une fonction (instruction `call` en langage d'assemblage x86 par exemple) lorsque cet appel a été compilé avant de connaître l'adresse de la fonction. Pour cela, il faut noter dans

le fichier objet l'ensemble des positions dans le binaire qui nécessite ce que l'on appelle une relocation.

— En langage C⋆, par une astuce que nous verrons plus tard, le compilateur se souvient via une structure de données appelée table des symboles qu'une fonction a été déclarée (mot-clé `extern`) mais pas encore définie (on ne connaît donc pas son adresse réelle). Lorsqu'une unité de compilation fait appel à une telle fonction il va utiliser une adresse qui n'est pas la bonne (puisqu'elle est inconnue). Ce n'est que lorsque la fonction aura été réellement définie qu'il viendra corriger le binaire pour l'ensemble des appels à cette fonction (qui ont été astucieusement chaînés les uns aux autres).

Par ailleurs, la déclaration d'une fonction avant sa définition permet de vérifier que chaque appel à cette fonction est bien effectué avec le bon nombre d'arguments et du bon type (ici entier ou pointeur). Ceci permet aussi de vérifier que l'on n'utilise pas une fonction ne retournant aucune valeur dans une expression, ce qui n'aurait aucun sens.

Les fichiers d'entête Une bonne pratique consiste donc à créer pour chaque unité de compilation, un fichier d'entête portant le même nom que l'unité de compilation considérée, mais dont le suffixe sera `.h`). Ce fichier contiendra un ensemble de déclarations :

— chaque variable globale définie dans l'unité de compilation à laquelle elle est associée.

— chaque fonction dont la visibilité doit être extérieure à l'unité de compilation (et qui peut donc être appelée par une autre unité de compilation).

Toute unité de compilation qui a besoin d'utiliser une variable globale ou une fonction définie dans une autre unité de compilation, doit simplement comporter une directive d'inclusion (`#include`), située généralement en début de fichier [3] pour indiquer d'inclure l'ensemble des déclarations contenues dans le fichier d'entête.

3 Matériel fourni

3.1 Code

Le code fourni pour cette séance est disponible sur le dépôt Git [4] public. Il est composé des fichiers suivants dont le rôle est brièvement indiqué :

— `main-cstar.c` : Il s'agit du fichier qui contient la fonction `main` du compilateur. Elle parse de manière minimaliste les arguments passés au compilateur. Pour cette séance de TP, le seul paramètre compris est le paramètre `-c` qui permet d'indiquer une liste de fichiers de code en C⋆ (extension `.c`) et/ou d'entête (`.h`) à compiler. La compilation se bornera à une analyse lexicale pour cette séance.

— `globals.c` : un ensemble de variables globales qui contiennent des constantes initialisées dans ce fichier et utilisées à plusieurs endroits du compilateur (final).

— `string.c` : un ensemble de variables globales qui contiennent des constantes initialisées dans ce fichier en relation avec la gestion des chaînes de caractères et utilisées à plusieurs endroits du compilateur (final).

— `library.c` : ce compilateur est destiné à produire (finalement) des exécutables auto suffisants au format ELF pour le processeur RISC-V. Le compilateur ne supporte pas la liaison

3. Même si ce n'est pas obligatoire.
4. `https://github.com/ftronel/tl-compilation`

dynamique de librairie. Il a donc recours à la liaison statique. En particulier une librairie C minimaliste est fournie qui permet entre autres d'afficher des chaînes de caractères sur la sortie standard, d'ouvrir, lire et écrire des fichiers. Cette librairie suppose l'existence des fonctions suivantes :

- `exit` : pour sortir d'un programme en renvoyant un code d'erreur.
- `open` : pour ouvrir un nouveau fichier et renvoyer un descripteur de fichier.
- `close` : pour fermer un fichier précédemment ouvert.
- `read` : pour lire dans un fichier précédemment ouvert.
- `write` : pour écrire dans un fichier précédemment ouvert.
- `malloc` : pour allouer dynamiquement de la mémoire. Vous noterez que dans sa version actuelle il n'est pas possible de libérer de la mémoire allouée. Ceci simplifie le gestionnaire de mémoire mais aboutit à des fuites de mémoire.

Ceci signifie que lorsque vous compilerez votre compilateur pour obtenir un programme qui fonctionne sur processeur Intel, il faudra le lier avec la librairie C du système. Ultérieurement lorsque vous serez capable de produire un compilateur complet celui-ci générera un binaire au format ELF pour processeur RISC-V qui contiendra la version de cette mini librairie C ainsi qu'une implémentation directement codée en assembleur RISC-V des 6 fonctions précédentes qui se contenteront de réaliser un appel système (pour le noyau Linux fonctionnant sur processeur RISC-V).

- `arguments.c` : ce fichier contient des fonctions permettant de récupérer les arguments passées à la fonction `main`.
- `preprocessor.c` : un ensemble de fonctions capables de simuler le comportement du préprocesseur du langage C, mais uniquement pour la directive `#include` (les autres directives telles `#define` ou encore `#ifdef` ne sont pas supportées).
- `parser.c` : un analyseur syntaxique minimaliste qui ne fait qu'appeler en séquence l'analyseur lexical pour qu'il lui fournisse le prochain lexème (ou *token*) reconnu. Il affiche ce lexème.
- `compiler.c` : ce fichier se contente de compiler l'ensemble des fichiers qui lui sont passés sur la ligne de commande en appelant l'analyseur syntaxique sur chacun d'eux.
- `lexer.c` : ce fichier contient l'analyseur lexical. C'est ce fichier (et uniquement lui que vous devez modifier lors de cette première séance).

Prenez connaissance de ce code afin de vous familiariser avec l'architecture du compilateur. L'ensemble du code est écrit en C⋆ et recourt abondamment à des variables globales. Ceci n'est évidemment pas recommandé de manière générale mais permet ici de maintenir le code suffisamment petit et permet aussi de pallier l'absence de structures de données.

3.2 Les binaires

Nous vous fournissons par ailleurs deux binaires :

- cstar.x86 : il s'agit du binaire du compilateur du langage C⋆ complètement finalisé. Le binaire est au format ELF pour les processeurs INTELx86 64 bits. Il permet de compiler du code C⋆ et de produire des fichiers au format ELF pour des processeurs RISC-V 64 bits.
- mipster.x86 : il s'agit d'un émulateur de processeur RISC-V 64 bits. Il permet de simuler l'exécution de binaire ELF pour ce processeur.

Vous pouvez donc tester si votre développement est bien conforme et compilable en langage C⋆ (ce qui est un des attendus du TP). Par ailleurs vous pourrez tester le fonctionnement de vos binaires comme s'ils s'exécutaient sur la plateforme RISC-V dont nous ne possédons malheureusement pas pour le moment de matériel sur lequel charger directement le code.

4 Prise en main du langage C⋆

4.1 Compilation du compilateur

Une première étape consiste à recompiler le code de l'embryon de compilateur qui vous est fourni à l'aide de GCC. Pour vous faciliter la vie nous fournissons un fichier `Makefile` qui s'occupe de recompiler l'ensemble du code. Pour cela, il suffit de faire un :

```
user@hostname:~/ git clone ...
user@hostname:~/ cd lexeur
user@hostname:~/lexeur/ ls -la
arguments.c  compiler.c  globals.c  lexer.c  library.c  main-cstar.c  parser.c  preprocessor.c  string.c
arguments.h  compiler.h  globals.h  lexer.h  library.h  Makefile      parser.h  preprocessor.h  string.h
user@hostname:~/lexeur/ make
cc [...] -Duint64_t='unsigned␣long␣long' -fno-builtin arguments.c -c arguments.c
cc [...] -Duint64_t='unsigned␣long␣long' -fno-builtin compiler.c -c compiler.c
cc [...] -Duint64_t='unsigned␣long␣long' -fno-builtin globals.c -c globals.c
cc [...]  -Duint64_t='unsigned␣long␣long' -fno-builtin lexer.c -c lexer.c
cc [...] -Duint64_t='unsigned␣long␣long' -fno-builtin library.c -c library.c
cc [...] -Duint64_t='unsigned␣long␣long' -fno-builtin parser.c -c parser.c
cc [...] -Duint64_t='unsigned␣long␣long' -fno-builtin preprocessor.c -c preprocessor.c
cc [...] -Duint64_t='unsigned␣long␣long' -fno-builtin string.c -c string.c
cc [...] -Duint64_t='unsigned␣long␣long' -fno-builtin main-cstar.c -c main-cstar.c
<command-line>: warning: return type of 'main' is not 'int' [-Wmain]
main-cstar.c:51:10: note: in expansion of macro 'main'
 uint64_t main(uint64_t argc, uint64_t* argv) {
          ^~~~
<command-line>: warning: return type of 'main' is not 'int' [-Wmain]
main-cstar.c:51:10: note: in expansion of macro 'main'
 uint64_t main(uint64_t argc, uint64_t* argv) {
          ^~~~
cc [...] -Duint64_t='unsigned␣long␣long' -fno-builtin arguments.o compiler.o globals.o lexer.o library.o parser.o prepro
```

4.2 Hello World

Vous pouvez créer un premier programme pour tester le compilateur `cstar.x86` ainsi que l'émulateur `mipster.x86`. Pour cela créez un fichier nommé `hello.c` contenant le code suivant :

```
#include "string.h"
#include "library.h"

uint64_t main(uint64_t argc, uint64_t *argv){
  init_library();
  printf1("%s\n", "Hello␣World");
}
```

Puis compilez ce code à l'aide du compilateur `cstar.x86`. Pour rappel (cf section 2.5) le compilateur `cstar.x86` ne supporte pas la compilation séparée. Vous devez donc indiquer sur la ligne

de commande l'ensemble des fichiers en langage C⋆ à compiler ensemble pour produire un binaire. Afin d'intégrer la mini librairie C à votre programme vous devez nécessairement compiler les fichiers suivants : `arguments.c`, `globals.c`, `string.c` et `library.c`. L'ordre dans lequel vous indiquez les fichiers n'a aucune importance. Le compilateur les traitera dans cet ordre, mais s'il devait manquer une définition de variables ou de fonctions au moment d'émettre le code final, l'éditeur de liens (primitif) intégré au compilateur final vous avertira des définitions manquantes (ou éventuellement multiples). Le compilateur ne connaît que deux options :

— `-c` `fichier`$_1$`.c ... fichier`$_n$`.c` : cette option permet d'indiquer la liste des fichiers C⋆ à compiler ensemble.

— `-o programme` : le nom du fichier binaire à produire.

```
user@hostname:~/lexeur/
user@hostname:~/ ./cstar.x86 -c arguments.c globals.c string.c library.c hello.c -o hello
./cstar.x86: compiling arguments.c with starc
./cstar.x86: 808 characters read in 51 lines and 0 comments
./cstar.x86: with 649(80.32%) characters in 176 actual symbols
./cstar.x86: 3 global variables, 8 procedures, 0 string literals
./cstar.x86: 4 calls, 7 assignments, 0 while, 2 if, 6 return
./cstar.x86: compiling string.c with starc
./cstar.x86: 10875 characters read in 392 lines and 67 comments
./cstar.x86: with 6200(57.10%) characters in 1687 actual symbols
./cstar.x86: 49 global variables, 62 procedures, 3 string literals
./cstar.x86: 42 calls, 36 assignments, 8 while, 15 if, 9 return
./cstar.x86: compiling globals.c with starc
./cstar.x86: 414 characters read in 11 lines and 5 comments
./cstar.x86: with 188(45.41%) characters in 50 actual symbols
./cstar.x86: 56 global variables, 62 procedures, 3 string literals
./cstar.x86: 0 calls, 0 assignments, 0 while, 0 if, 0 return
./cstar.x86: compiling library.c with starc
./cstar.x86: 20047 characters read in 728 lines and 108 comments
./cstar.x86: with 11646(58.90%) characters in 3337 actual symbols
./cstar.x86: 56 global variables, 62 procedures, 10 string literals
./cstar.x86: 147 calls, 53 assignments, 9 while, 47 if, 49 return
./cstar.x86: compiling hello.c with starc
./cstar.x86: 5617 characters read in 156 lines and 23 comments
./cstar.x86: with 4185(74.51%) characters in 857 actual symbols
./cstar.x86: 56 global variables, 62 procedures, 12 string literals
./cstar.x86: 2 calls, 0 assignments, 0 while, 0 if, 0 return
./cstar.x86: symbol table search time was 2 iterations on average and 4030 in total
./cstar.x86: 14664 bytes generated with 3438 instructions and 784 bytes of data
./cstar.x86: init:    lui: 1(0.20%), addi: 1379(40.11%)
./cstar.x86: memory:  ld: 627(18.23%), sd: 594(17.27%)
./cstar.x86: compute: add: 144(4.18%), sub: 86(2.50%), mul: 20(0.58%), divu: 21(0.61%), remu: 13(0.37%)
./cstar.x86: control: sltu: 72(2.90%), beq: 85(2.47%), jal: 327(9.51%), jalr: 61(1.77%), ecall: 8(0.23%)
./cstar.x86: 14664 bytes with 3438 instructions and 784 bytes of data written into hello
```

Vous pouvez vérifier le type de fichier ELF produit :

```
user@hostname:~/lexeur/file ./hello
./hello: ELF 64-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically linked, no section header
```

Vous pouvez à présent exécuter ce code via l'émulateur `mipster.x86` :

```
user@hostname:~/lexeur/
user@hostname:~/ ./mipster.x86 -l hello -m 4
./mipster.x86: 14664 bytes with 3438 instructions and 784 bytes of data loaded from ./hello
./mipster.x86: executing ./hello with 4MB physical memory on mipster
Hello World
./mipster.x86: ./hello exiting with exit code 0 and 0.00MB mallocated memory
./mipster.x86: terminating ./hello with exit code 0
./mipster.x86: summary: 7664 executed instructions and 0.20MB mapped memory
./mipster.x86: init:    lui: 1(0.10%), addi: 2741(35.76%)
./mipster.x86: memory:  ld: 1808(23.59%), sd: 999(13.30%)
./mipster.x86: compute: add: 491(6.40%), sub: 205(2.67%), mul: 329(4.29%), divu: 54(0.70%), remu: 42(0.54%)
./mipster.x86: control: sltu: 149(1.94%), beq: 159(2.70%), jal: 463(6.40%), jalr: 203(2.64%), ecall: 20(0.26%)
./mipster.x86: profile: total,max(ratio%)@addr,2max,3max
./mipster.x86: calls:   205,55(26.82%)@0xF68,30(14.63%)@0x2F8,30(14.63%)@0x1100
./mipster.x86: loops:   75,63(84.00%)@0x30F0,11(14.66%)@0x1B28,1(1.33%)@0x2070
./mipster.x86: loads:   1808,64(3.53%)@0x30F4,64(3.53%)@0x30F8,63(3.48%)@0x3104
./mipster.x86: stores:  999,63(6.30%)@0x3140,63(6.30%)@0x3150,55(5.50%)@0xF70
```

5 Travail restant à effectuer

Des squelettes d'analyseurs lexical et syntaxique vous sont fournis. L'analyseur syntaxique (que vous devrez compléter dans les prochaines séances) se contente de récupérer les tokens envoyés par l'analyseur lexical et de les afficher sur la sortie standard. Vous n'avez normalement rien à modifier dans son code.

Concernant l'analyseur lexical, il est déjà pourvu de plusieurs fonctions :

— `get_character()` : lit le prochain caractère du fichier source actuellement en train d'être compilé.

— `find_next_character()` : lit le fichier tant que le prochain caractère est un espace, ou une tabulation, ou bien un commentaire. Les styles de commentaires acceptés sont :

1. les commentaires sur une ligne qui débutent par `//`.
2. les commentaires qui peuvent potentiellement s'étaler sur plusieurs lignes et qui débutent par `/*` et se terminent par `*/`.

Tant que l'on est dans cette situation, cette fonction consomme des caractères du fichier à compiler et les ignore. Elle peut renvoyer un lexème à l'analyseur syntaxique si elle trouve que le prochain caractère est un signe de division `/` (du fait que ce caractère est un préfixe des symboles de commentaires) qui n'est ni suivi du caractère `/`, ni du caractère `*`.

— `is_character_new_line()` : teste si un caractère est un retour à la ligne, c'est-à-dire le caractère ASCII *Carriage Return* ou *Line Feed*.

— `is_character_whitespace()` : teste si un caractère est une espace (ou une tabulation).

— `is_character_letter()` : teste si un caractère est une lettre (minuscule ou majuscule).

— `is_character_digit()` : teste si un caractère si un caractère est un chiffre (entre 0 et 9).

— `is_character_letter_or_digit_or_underscore()` : teste si un caractère est une lettre, ou un chiffre, ou le caractère souligné (« _ »).

— `is_character_not_double_quote_or_new_line_or_eof()` : teste si un caractère n'est pas un double guillemet (« " »), un retour à la ligne ou la fin du fichier.

— `identifier_or_keyword()` : teste si l'identifiant (qui est une chaîne de caractères) associé à un lexème est un mot-clé du langage ou bien un identifiant (nom de variable ou nom de fonction).

— `handle_escape_sequence()` : prend en charge les caractères d'échappement (\t, \n, \b, \', \", \% et \\) dans les chaînes de caractères afin de remplacer les deux caractères par un seul dont le code ASCII correspond au caractère échappé.

— `get_symbol()` : Il s'agit de la fonction que vous devez compléter. Elle doit correspondre à l'automate illustré par la figure 1. Cette fonction doit rendre la main à chaque fois qu'elle a reconnu un symbole en mettant à jour la variable globale nommée `symbol` en utilisant les constantes définies à cet usage et nommée `SYM_...`

À vous de jouer ! Assurez-vous bien de produire du code écrit en C⋆. Pour cela vous pouvez utiliser la cible du fichier Makefile intitulée test.cstar qui tente de recompiler votre code en utilisant le compilateur cstar complet.

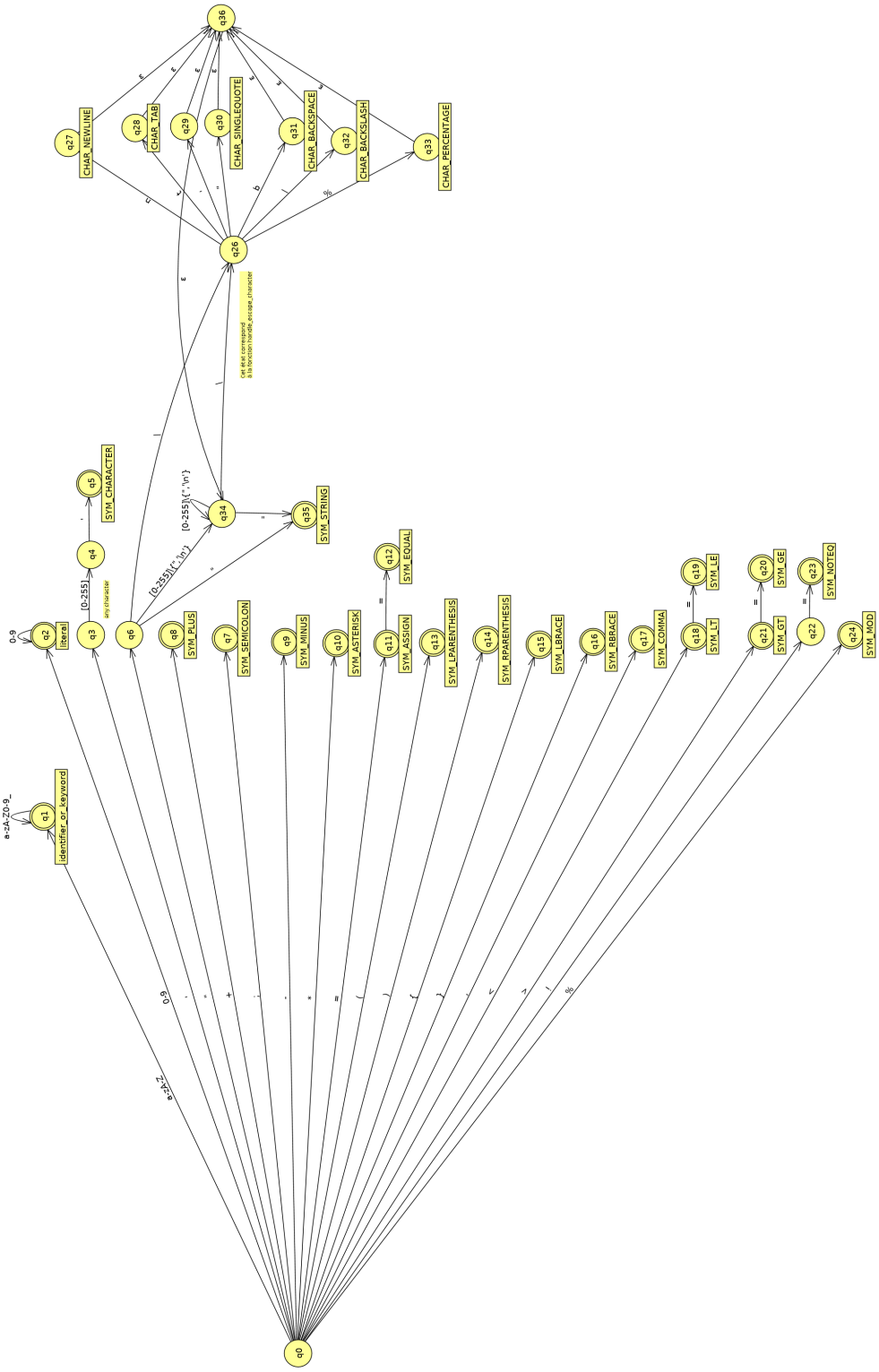

FIGURE 1 – L'automate qu'il faut mettre en œuvre dans la fonction `get_symbol()`.