Projet compilation 2024-2025

Consignes :

- Lisez bien tout le sujet avant de commencer à coder.
- Déposez avant le **vendredi 20 juin à 23h59** une archive au format `.tar.gz`, contenant votre code, vos éventuels fichiers de test, ainsi que votre rapport au **format PDF**, dans le dépôt `proc_fisa_2425` sur `http://exam.ensiie.fr`.

L'objectif de ce sujet est de développer un mini-interpréteur permettant de manipuler des fonctions logiques, définies soit par une formule, soit par une table de vérité.

1 Description du langage

1.1 Définitions préliminaires

Dans tout le sujet, les booléens seront représentés par les caractères 0 (pour `faux`) et 1 (pour vrai).

Le langage utilisé pour définir et manipuler des fonctions logiques comporte les mots-clés suivants :

`and`	`formula`	`table`
`at`	`list`	`varlist`
`define`	`not`	`xor`
`eval`	`or`	

Votre programme devra reconnaître ces mots-clés, qu'ils soient écrits en majuscules, en minuscules, ou avec un mélange de majuscules et minuscules.

Pour des raisons de simplicité, on se limitera à des fonctions dont le nombre de variables sera compris entre 0 et 8 (inclus). Les noms des fonctions, tout comme ceux des variables, seront de chaînes de caractères contenant uniquement des lettres, des chiffres et le symbole `_`. Ces noms devront toujours commencer par une lettre, et ne pourront pas être un des mots-clés ci-dessus.

Pour définir une fonction, on pourra utiliser une formule booléenne. Ces formules sont définies inductivement comme suit :

- Les constantes `0` et `1` sont des formules ;
- Une variable est une formule ;
- Si f est une formule, alors `! f` et `not f` sont des formules représentant la négation de f ;
- Si f et g sont des formules, alors
  - `f | g`, `f || g` et `f or g` sont des formules représentant le OU logique de f et g,
  - `f & g`, `f && g` et `f and g` sont des formules représentant le ET logique de f et g,
  - `f ^ g` et `f xor g` sont des formules représentant le XOR logique de f et g,
  - `f => g` est une formule représentant f implique g ;
- Enfin, si f est une formule, alors `( f )` est une formule équivalente à f.

On pourra aussi définir une fonction à l'aide de sa table de vérité. Une telle table sera donnée par une suite de valeurs booléennes séparées par des blancs et entre accolades. Ainsi, l'implication est définie par

x	y	x => y
0	0	1
0	1	1
1	0	0
1	1	1

et la table associée à la formule `x => y` est donc représentée par `{ 1 1 0 1 }` . Il faudra faire attention à l'ordre des valeurs booléennes entre les accolades, qui est lié à l'ordre des variables. Par exemple, la table associée à la fonction `g(y,x) = x => y` est représentée par `{ 1 0 1 1 }` car, dans ce contexte, la première variable est y.

1.2 Commandes de l'interpréteur

Cette section décrit les cinq commandes prévues pour notre mini-interpréteur.

La commande **define** permet de définir une fonction. Le mot-clé `define` doit être suivi d'un nom de fonction, d'une liste optionnelle de variables (entre parenthèses et séparées par des virgules), du symbole `=`, et enfin d'une formule logique ou d'une table en suivant la syntaxe décrite à la section précédente. Ainsi, on peut par exemple écrire :

```
define f = x or !z
define g(y,x) = x => y
define h(a,b,c) = { 0 1 1 0 1 0 0 1 }
define nand = { 1 1 1 0 }
```

Dans le cas où il n'y a pas de liste de variables, les noms par défaut sont dans l'ordre `x`, `y`, `z`, `s`, `t`, `u`, `v` et `w`.

La commande **list** permet d'afficher les noms des fonctions préalablement définies à l'aide de la commande **define**. Cette commande est appelée en entrant simplement le mot-clé `list`. En reprenant l'exemple précédent, le résultat attendu est :

commande	résultat
`list`	`f g h nand`

Vous êtes libre d'afficher les noms de fonctions dans l'ordre qui vous arrange.

La commande **varlist** permet d'afficher les noms des variables pour une fonction donnée en argument. Les résultats attendus sur les fonctions définies ci-dessus sont les suivants :

commande	résultat
`varlist f`	`x y z`
`varlist g`	`y x`
`varlist h`	`a b c`
`varlist nand`	`x y`

On notera que, comme la définition de `f` contient le troisième nom par défaut des variables, il s'agit d'une fonction à trois variables (même si la définition n'utilise pas explicitement `y`).

La commande **eval** permet d'évaluer une fonction. Le mot-clé `eval` doit être suivi d'un nom de fonction, du mot-clé `at` et d'une liste de valeurs booléennes séparées par des espaces. Le tableau suivant contient quelques exemples :

commande	résultat
`eval g at 0 1`	`0`
`eval h at 1 1 0`	`0`

La commande **table** permet d'afficher la table de vérité d'une fonction. Pour l'appeler, il faut utiliser le mot-clé `table` suivi d'un nom de fonction. Les résultats attendus sont :

commande	résultat
`table f`	`{ 1 0 1 0 1 1 1 1 }`
`table g`	`{ 1 0 1 1 }`
`table h`	`{ 0 1 1 0 1 0 0 1 }`
`table nand`	`{ 1 1 1 0 }`

2 Travail demandé

Vous devrez réaliser :
— un analyseur lexical (fichier `lexer.l`)
— un analyseur syntaxique (fichier `parser.y`)
— un programme principal pour lier le tout.

Dans un premier temps, votre objectif va être d'avoir un programme qui reconnaît les commandes valides, et qui affiche un message d'erreur sur la saisie d'une commande invalide.

Dans un deuxième temps, vous devrez implanter les différentes fonctionnalités décrites dans la partie 1.2. Vous serez peut-être aussi amené à faire quelques hypothèses simplificatrices. Dans ce cas, vous indiquerez ces hypothèses dans votre rapport.

Enfin, vous pourrez implanter une ou plusieurs des fonctionnalités suivantes :
— une gestion raisonnable des erreurs afin, en particulier, d'éviter que votre programme s'arrête en cas de faute de frappe ;
— un mode non-interactif, dans lequel votre programme prend en argument le nom d'un fichier contenant les données à lire. En cas d'erreur de syntaxe dans ce mode, il est préférable d'afficher le numéro de la première ligne contenant une erreur et de s'arrêter là ;
— la nouvelle commande **formula**, permettant d'afficher une formule logique correspondant à la fonction passée en argument. Les résultats pourront par exemple être :

commande	résultat
`formula f`	`x or !z`
`formula g`	`x => y`
`formula h`	`(!a & !b & c) | (!a & b & !c) | (a & !b & !c) | (a & b & c)`
`formula nand`	`(!x & !y) | (!x & y) | (x & !y)`

Lorsque la définition d'une fonction a été faite à partir d'une formule, il paraît naturel d'afficher cette formule. Dans le cas d'une définition par table, on se contentera d'afficher n'importe quelle formule qui convient ;
— la possibilité de définir une fonction à partir d'une autre, rendant valide

```
define ff(x,y) = x or !y
define gg(a,b,c,d) = a or ff(b, ff(c, d) )
```

par exemple.

3 Conseils

Assez rapidement, vous ne devriez plus avoir besoin de toucher à vos lexèmes (définis dans `parser.y`) et vos expressions régulières (définies dans `lexer.l`).

Comme indiqué dans la section précédente, il est vivement recommandé de s'attaquer d'abord à la partie purement grammaire de `parser.y` (et donc, ne pas mettre de traitement dans un premier temps).

La grammaire (dans `parser.y`) pourra être amenée à évoluer un peu au cours du projet, selon les structures de données que vous allez utiliser pour stocker les formules et les tables de vérité (et l'ordre dans lequel vous allez remplir ces structures).

Il est fortement conseillé de créer (et de rendre) des fichiers de test. Ces tests devraient en particulier permettre de vérifier facilement :
— la qualité (et les limites) de votre analyseur syntaxique,
— le bon fonctionnement des commandes implantées.