TP 3: Écriture d'une grammaire et génération d'un arbre de syntaxe abstraite

Vous l'avez remarqué lors du TP 2, écrire un analyseur syntaxique à la main est fastidieux. Lors de cette séance de TP, nous mettons à votre disposition un outil qui génère automatiquement les tables NULL, FIRST et FOLLOW, ainsi que la table de prédiction LL(1). Il vous suffira d'écrire la grammaire et l'outil génèrera automatiquement l'analyseur syntaxique (sous forme de code C⋆).

On se concentrera ici sur un sous-ensemble de C⋆ très simple, où les programmes sont simplement composés d'affectations et d'expressions (pas de conditionnelle, de boucle, de fonctions). Dans un premier temps, vous écrirez du code pour générer un arbre de syntaxe abstraite (AST). Dans un second temps, vous re-transformerez cet AST en format textuel (vous devriez retrouver le programme source), puis vous écrirez un interpréteur pour ce langage.

Du code vous est fourni pour ce TP sur `https://github.com/ftronel/tl-compilation` dans le répertoire `parser-generator`.

1 Générateur d'analyseur syntaxique

1.1 Format de la grammaire

Vous écrirez votre grammaire dans le format suivant, dans un fichier `.g`. On vous donne ci-dessous le contenu d'un fichier `expr_grammar.g` qui donne la grammaire d'un petit langage d'expressions (un sous-ensemble des expressions C⋆).

```
tokens SYM_EOF SYM_IDENTIFIER SYM_INTEGER SYM_PLUS SYM_MINUS SYM_ASTERISK
    ↪ SYM_LPARENTHESIS SYM_RPARENTHESIS SYM_ASSIGN SYM_SEMICOLON SYM_RETURN
non-terminals S ASSIGNS ASSIGN EXPR TERM TERMS FACTOR FACTORS
rules
S -> ASSIGNS SYM_RETURN EXPR SYM_EOF
ASSIGNS -> ASSIGN ASSIGNS
ASSIGNS ->
ASSIGN -> SYM_IDENTIFIER SYM_ASSIGN EXPR SYM_SEMICOLON
EXPR -> TERM TERMS
TERM -> FACTOR FACTORS
TERMS -> SYM_PLUS TERM TERMS
TERMS -> SYM_MINUS TERM TERMS
TERMS ->
FACTOR -> SYM_INTEGER
FACTOR -> SYM_IDENTIFIER
FACTOR -> SYM_LPARENTHESIS EXPR SYM_RPARENTHESIS
FACTORS -> SYM_ASTERISK FACTOR FACTORS
FACTORS ->
```

Décortiquons ce fichier :

- La première ligne déclare les tokens que l'on traitera. Vous reconnaîtrez des tokens de C⋆. La ligne commence par le mot clé **tokens** puis on place la liste de tous les tokens séparés par des espaces.
- La seconde ligne déclare les non-terminaux que l'on va définir. On commence par le mot-clé **non-terminals** puis on place les différents non-terminaux, séparés par des espaces.
- La troisième ligne contient le mot-clé **rules**. Cela signifie que l'on va commencer à écrire des règles de production pour chacun des non-terminaux.

— Les lignes suivantes sont des définition de règles de grammaire. Elles ont toutes la même forme, à savoir un non-terminal, suivi d'une flèche (–>), suivi d'une liste de tokens et non-terminaux. Par exemple, la première règle `S –> ASSIGNS SYM_RETURN EXPR SYM_EOF` déclare que le non-terminal `S` (l'*axiome* de notre grammaire) peut être produit par la séquence formée par le non-terminal `ASSIGNS`, suivi du token `SYM_RETURN`, suivi du non-terminal `EXPR`, et enfin suivi du token `SYM_EOF`, indiquant la fin du fichier. Certains non-terminaux ont plusieurs règles ; dans ce cas, on écrira plusieurs règles qui commencent par ce non-terminal à gauche de la flèche. Certaines règles peuvent dériver le mot vide, comme `TERMS –>` par exemple ; dans ce cas, on n'écrit rien à droite de la flèche.

1.2 Suite de tests

Dans le répertoire `expr/tests`, vous trouverez une dizaine de programmes de test. Examinez-les. Lesquels vous paraissent syntaxiquement corrects ? Sémantiquement corrects ?

Remplissez le tableau ci-dessous. Indiquez dans la première colonne si le programme est syntaxiquement correct (c'est-à-dire s'il est reconnu par la grammaire). Indiquez dans la seconde colonne si le programme est sémantiquement correct (c'est-à-dire s'il va s'exécuter correctement ou non). Dans le cas où la sémantique est définie, donnez également le résultat attendu dans cette colonne.

	syntaxiquement correct	sémantique
test1.e		
test2.e		
test3.e		
test4.e		
test5.e		
test6.e		
test7.e		
test8.e		
test9.e		

1.3 Compiler le générateur d'analyseur syntaxique

Le générateur d'analyseur syntaxique vous est fourni sous la forme d'un binaire compatible avec les systèmes Linux 64-bit. Si vous voulez l'utiliser sur d'autres plateformes, il faudra auparavant le compiler. Il s'agit d'un programme OCaml, que vous pouvez compiler en tapant simplement `make` dans le dossier adéquat, pourvu que vous ayez une installation d'Ocaml [1] et de Menhir [2].

1.4 Utiliser le générateur d'analyseur syntaxique

Pour générer l'analyseur syntaxique (le *parser*) correspondant à votre grammaire, entrez la commande suivante dans le terminal :

```
$ ./parser_generator -g expr_grammar.g -pc expr_parser.c -ph expr_parser.h -pdir expr -t table.html
```

Les options sont les suivantes :

— `-g <file>` : indique dans quel fichier se trouve votre grammaire. Il s'agit du fichier `.g` que vous avez écrit ;

1. `https://ocaml.org/`
2. `http://gallium.inria.fr/~fpottier/menhir/`

— −pc <file> : indique le nom du fichier dans lequel sera écrit le parser généré par l'outil (.c) ;

— −ph <file> : indique le nom du fichier dans lequel sera écrit l'entête du parser généré par l'outil (.h) ;

— −pdir <file> : indique le répertoire dans lequel les deux précédents fichiers seront écrits.

— −t <file> : indique le nom du fichier dans lequel les tables NULL, FIRST et FOLLOW seront écrites. Il s'agit d'un fichier HTML.

Observez les fichiers générés.

1.4.1 Le fichier HTML avec les différentes tables.

Ouvrez le fichier HTML dans un navigateur. Vous devriez y trouver :

1. votre grammaire (ignorez la colonne remplie de **return** NULL; pour le moment, nous y reviendrons plus tard [3]) ;
2. la table NULL. Pour chaque non-terminal, la table vous indique s'il peut dériver le mot vide ou non.
3. la table FIRST. Pour chaque non-terminal, la table vous indique par quels tokens il peut commencer.
4. la table FOLLOW. Pour chaque non-terminal, la table vous indique quels tokens peuvent suivre.
5. la table LL. Pour chaque non-terminal (ligne du tableau), et pour chaque token (colonne du tableau), cette table indique la règle à suivre. Si votre grammaire est effectivement LL(1) (c'est le cas de celle fournie en exemple, expr_grammar.g), il ne devrait y avoir qu'une seule règle dans chaque case du tableau. Les règles sont différenciées selon que le token appartient à l'ensemble FIRST du non-terminal (en bleu) ou à l'ensemble FOLLOW (en rouge). Les conflits sont mis en évidence par un fond rouge clair dans la case concernée. Si vous enlevez le token SYM_RETURN de la règle S, vous aurez un conflit pour le non-terminal ASSIGNS et le token SYM_IDENTIFIER.

1.4.2 Le parser généré pour votre grammaire.

Le parser est séparé en un fichier d'entête et un fichier de code C. Dans notre exemple, ces fichiers sont expr/expr_parser.h et expr/expr_parser.c. Le fichier .h contient les déclarations des fonctions d'analyse syntaxique de chaque non-terminal de la grammaire. Pour chaque non-terminal NT, une fonction uint64_t* parse_NT(); est déclarée.

Dans le fichier .c, ces fonctions sont définies. Pour parser un non-terminal NT (fonction parse_NT()), on calcule la liste lt des tokens pour lesquels on a une entrée dans la table LL. Pour chaque token t appartenant à cette liste lt, si le symbole donné par l'analyseur lexical est égal à t (symbol == t), alors on applique la règle donnée par la table LL dans la case (NT,t). Appliquer une règle signifie appeler l'analyseur correspondant pour chaque non-terminal présent dans la règle et consommer les tokens lorsqu'ils apparaissent dans la règle. Plus précisément, pour appliquer une règle X −> aYbZc, on écrira le code C suivant :

```
eat(a);
parse_Y();
eat(b);
parse_Z();
eat(c);
```

3. TODO : forward reference

où eat est une fonction qui consomme un token particulier (et échoue avec une erreur de syntaxe lorsque ce token n'est pas présent dans le flux de lexèmes), et parse_XXX() sont les fonctions définies pour les autres non-terminaux.

1.5 Compiler l'analyseur syntaxique obtenu

Le code qui vous est fourni (dans le répertoire expr/) est une partie du code du compilateur de C⋆, dont le fichier parser.c a été modifié pour fonctionner avec le fichier expr_parser.c que vous avez généré. Compilez l'ensemble des fichiers C :

```
$ cd expr
$ make
```

Vous obtenez un exécutable cstar qui parse des programmes dans notre petit langage d'expressions.

1.6 Lancer l'analyseur syntaxique

L'exécutable cstar que vous venez de compiler permet d'analyser lexicalement et syntaxiquement des programmes écrits dans notre grammaire d'expressions (expr_grammar.g). Pour ce faire, entrez la commande suivante :

```
expr $ ./cstar -c <file>
```

Par exemple, pour lancer le premier test fourni, il faudra entrer :

```
expr $ ./cstar -c tests/test1.e
Parsed expression :
À compléter.
Result = 0.
expr $
```

Vous devriez obtenir la sortie que l'on vous montre. L'affichage du programme est pour l'instant À compléter. et l'évaluation rend toujours 0. Par contre, le fichier est syntaxiquement valide, donc vous n'obtenez pas d'erreur. En revanche, si vous lancez le compilateur sur un fichier vide, alors la syntaxe ne sera pas valide :

```
expr $ ./cstar -c /dev/null
expr/cstar: syntax error in /dev/null in line 1: error while parsing S
Expected one of { identifier, return} but got 'end␣of␣file' instead.
```

Vous pouvez lancer le compilateur sur l'ensemble des tests en lançant la cible test du Makefile :

```
expr $ make test
```

Vérifiez que le compilateur obtient le même diagnostic que vous sur l'ensemble des tests. (Comparez avec le tableau rempli en 1.2).

2 Génération d'un AST

Nous allons à présent générer un arbre de syntaxe abstraite (*Abstract Syntax Tree*, ou AST) pour notre langage.

2.1 Structures de données en C⋆

2.1.1 Tableaux

```
uint64_t* array_get(uint64_t* p, uint64_t i);
void array_set(uint64_t* p, uint64_t i, uint64_t* v);
```

Pour manipuler des tableaux en C⋆, nous utiliserons la fonction `array_get(p,i)` qui a le même effet que `p[i]` en C standard, c'est-à-dire accéder à la ième case du tableau `p`. De la même manière, on utilisera la fonction `array_set(p,i,v)` pour stocker la valeur `v` à l'indice `i` du tableau `p`, *i.e.* comme si on écrivait `p[i] = v;` en C.

2.1.2 Tuples : paires, triplets, quadruplets, quintuplets

Nous aurons besoin de tuples, qui nous permettront d'encoder des arbres.

```
uint64_t* pair(uint64_t* fst, uint64_t* snd);
uint64_t* triple(uint64_t* fst, uint64_t* snd, uint64_t* thr);
uint64_t* quadruple(uint64_t* fst, uint64_t* snd, uint64_t* thr, uint64_t* frth);
uint64_t* quintuple(uint64_t* fst, uint64_t* snd, uint64_t* thr, uint64_t* frth,
    uint64_t* fif);

uint64_t* fst(uint64_t* p);
uint64_t* snd(uint64_t* p);
uint64_t* thr(uint64_t* p);
uint64_t* frth(uint64_t* p);
uint64_t* fif(uint64_t* p);
```

Les fonctions `pair`, `triple`, `quadruple`, `quintuple` créent des tuples avec tous leurs arguments et rendent un pointeur vers cette nouvelle structure. Les accesseurs `fst`, `snd`, `thr`, `frth`, `fif` permettent de récupérer le premier, le deuxième, le troisième, le quatrième et le cinquième élement du tuple passé en paramètre, respectivement.

2.1.3 Listes

Enfin, on souhaite aussi pouvoir manipuler des listes pour notre AST.

```
uint64_t* cons(uint64_t* elt, uint64_t* l);
uint64_t* nil;

uint64_t* get_elt(uint64_t* l);
uint64_t* next_elt(uint64_t* l);
```

La fonction `cons(hd,tl)` construit une liste avec `hd` comme tête et `tl` comme queue de liste. Pour parcourir une liste, on utilisera la fonction `get_elt(l)` qui retourne l'élément en tête de liste et la fonction `next_elt(l)` qui retoure un pointeur vers la queue de la liste.

Par exemple, si on a une liste `lstring` de chaînes de caractères, on peut les afficher toutes avec ce morceau de code :

```
uint64_t* l = lstring;
uint64_t* elt;
while(l){
  elt = get_elt(l);
  print(elt);
  l = next_elt(l);
}
```

2.2 Grammaire à actions

Nous allons enrichier notre grammaire avec des actions.

Dans le fichier `expr_grammar.g`, modifiez chaque règle en ajoutant en fin de ligne, du code C⋆ entre accolades. Par exemple, on pourra écrire pour la première règle :

```
S -> ASSIGNS SYM_RETURN EXPR SYM_EOF { return pair($1,$3); }
```

Dans l'action (le code C⋆ entre accolades), on accède aux valeurs retournées pour chacun des terminaux et non-terminaux avec les variables `$1`, `$2`, ... où `$i` correspond au i-ème terminal ou non-terminal de la règle. Pour le terminal `SYM_IDENTIFIER`, `$i` contient la chaîne de caractères correspondant à l'identifiant. Pour le terminal `SYM_INTEGER`, `$i` contient la chaîne de caractères correspondant à l'entier lu (utiliser la fonction `atoi` pour la transformer en entier).

Dans l'exemple ci-dessus, on rend donc une paire avec :

— premier élément : le résultat rendu par le non-terminal `ASSIGNS` ;

— deuxième élément : le résultat rendu par le non-terminal `EXPR` (`$3` parce que c'est le 3ème élément de la règle).

Concrètement, on obtient une paire avec d'une part, la liste des affectations de variables ; et d'autre part l'expression retournée par le programme.

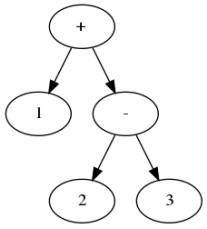
2.3 Construction de l'AST

Activité 1. Ajoutez des actions à la grammaire `expr_grammar.g` pour construire un AST.

Les indications qui suivent ne sont que des conseils. Vous pouvez sans doute faire autrement.

L'expression `x + y` pourra être représentée par le triplet (`EADD`, `ast_x`, `ast_y`), où `ast_x` est l'AST correspondant à l'expression `x` (idem pour `y`) et `EADD` est une constante que l'on aura défini au préalable.

L'analyse syntaxique des expressions est sans doute la partie la plus complexe de cette grammaire. En effet, pour rendre notre grammaire LL(1), on a découpé les expressions en termes et en facteurs, ce qui donne des règles peu naturelles. Pour cette raison, on ne pourra pas construire directement des morceaux d'AST, notamment pour les règles `TERMS -> SYM_PLUS TERM TERMS` ou `TERMS -> SYM_MINUS TERM TERMS`. Nous vous suggérons, pour ces règles, de générer une liste de paires de la forme `[(ADD, 2); (SUB, 3)]`. Pour la règle `EXPR -> TERM TERMS`, il vous faudra alors combiner le résultat du non-terminal `TERM` et le résultat du non-terminal `TERMS` pour former une expression correcte. Par exemple, pour l'expression `1 + 2 - 3`, vous devrez aboutir à un arbre qui ressemble à celui-ci :



Plus concrètement, la règle pour le non-terminal `EXPR` aura la forme suivante :

```
EXPR -> TERM TERMS {
  // $1 contient l'AST correspondant à l'expression 1
  // $2 contient une liste de paires ([(ADD,2),(SUB,3)])
  return XXX($1,$2);
}
```

Vous pouvez définir de nouvelles fonctions par exemple dans ast.c. N'oubliez pas d'ajouter la signature de vos nouvelles fonctions dans ast.h pour qu'elles soient accessibles depuis les autres fichiers.

3 Affichage des programmes

Maintenant que vous avez construit un AST, utilisez-le ! Et pour vérifier qu'il est construit correctement, écrivez une fonction qui re-transforme l'AST en une représentation textuelle. Vous pourrez ainsi comparer la sortie de votre afficheur avec le programme original et vous convaincre que votre construction d'AST et votre afficheur sont corrects.

Activité 2. Complétez le fichier expr_dump.c.

Vous utiliserez notamment les fonctions d'affichage (print, printf1, printf2...) déclarées dans library.h et dont la spécification vous a été donnée dans le sujet du TP 1 (accessible sur le dépôt git).

Testez votre afficheur sur les programmes de test (dans le répertoire tests) en tapant make **test**. Vous pouvez également ajouter vos tests personnels dans ce répertoire.

4 Évaluation des programmes

Dans cette partie, vous allez construire un interpréteur pour les programmes reconnus par la grammaire expr_grammar.g.

Vous aurez besoin de modéliser l'état du programme pour écrire l'interpréteur, c'est-à-dire maintenir une structure de données qui stocke les valeurs des différentes variables du programme. On propose de modéliser cet état par une liste d'associations. Une liste d'associations est une liste de paires, où chaque paire est constituée d'un identifiant de la variable et d'une valeur entière. Vous réutiliserez les fonctions de manipulations de listes et de paires définies dans ast.c et introduites dans la section 2.1.

L'état sera stocké dans une variable globale state. On aura besoin d'une fonction lookup qui cherche dans l'état la valeur d'une variable et d'une fonction set qui met à jour l'état pour associer une nouvelle valeur à une variable donnée.

Par exemple, lookup((uint64_t* "foo") dans l'état [("bar",2),("foo",3)] retournera 3 et set((uint64_t*)"bar",8) dans le même état modifiera l'état en [("bar",8),("foo",3)]. Pour finir set((uint64_t*)"baz",42) dans l'état obtenu modifiera l'état en [("baz",42),("bar",8),("foo",3)].

```
uint64_t* state;
uint64_t lookup(uint64_t* id);
void set(uint64_t* id, uint64_t v);
```

Activité 3. Complétez le fichier expr_eval.c.