Rapport sur le développement de “Earl Grey”

Ou le développement d’un micro-JS compilé vers le bytecode de la machine universelle

Hugo GUERRIER | Emilie SIAU

Avec l’aide et l’encadrement de notre professeur Darius MERCADIER

Introduction et objectifs

Le langage Earl Grey est un projet académique que nous avons dû réaliser dans le cadre de notre unité d'enseignement "Compilation Avancée". C'est un langage de programmation simple se basant sur la syntaxe de JavaScript et étant compilé vers un bytecode interprété par une machine virtuelle. Le but de ce langage n'était pas de proposer une concurrence sérieuse aux grands langages de programmation, mais de s'exercer à imaginer, développer et déployer une solution permettant d'implémenter un langage compilé.

Il a cependant été fait quelques changements quant à la grammaire demandés pour le langage par le sujet du projet :

- Il est possible dans Earl Grey de mêler les déclaration de fonction et les autre instructions
- Le mot clé "var" a été abandonné car jugé peu pertinent

Le développement de Earl Grey s'est déroulé en deux grandes étapes que nous allons vous détailler dans la suite.

La machine virtuelle

Le développement de la machine virtuelle a été la première étape du projet, elle se base sur le modèle décrit de la machine universelle d'un concours de programmation http://www.boundvariable.org/task.shtml. Le but de cette machine est de permettre l'interprétation d'un bytecode simple étant simplement composé d'entier de 32 bits contenant : le numéro d'opération, les registres à modifier, la valeur à charger si nécessaire.

La machine a donc été structurée de la manière suivante :

- Un point d'entrée, chargé de parser les arguments et d'en extraire le fichier à exécuter
- Une machine contenant et gérant sa mémoire de manière dynamique
- Un exécuteur *sûr*, faisant moult vérifications sur le bytecode
- Un exécuteur *optimisé* ne se concentrant uniquement que sur la performance

Au début évidemment, seulement la partie *sûre* était présente, elle a servi de base à la partie *optimisée* de la machine.

L'exécuteur *sûr* est donc une boucle infinie itérant sur les instructions et lançant une fonction pour chacune d'elle en agissant sur la machine. Pour chaque opération, des vérifications sont faites pour éviter des comportement imprévisible, par exemple, dans la fonction suivante, lors de l'accès à un tableau, l'indice est vérifié afin de s'assurer qu'il est bien dans le tableau, ainsi que l'existence du tableau.

```
35  // --- Do an array index fetching
36  static void _do_array_index(machine_data_t *data, int a, int b, int c) {
37      unsigned int r_b = data->registers[b];
38      unsigned int r_c = data->registers[c];
39
40      // Verify the table index
41      if(r_b < data->table_array_size) {
42          // Verify the plate index
43          if(r_c < data->table_array[r_b]->size) {
44              data->registers[a] = data->table_array[r_b]->content[r_c];
45          } else {
46              raise_machine_error(data, INDEX_OUT_OF_BOUNDS, "Tried to access a plate out of bounds");
47          }
48      } else {
49          raise_machine_error(data, INDEX_OUT_OF_BOUNDS, "Tried to access a table out of bounds");
50      }
51  }
```

L'exécuteur *optimisé* quant à lui utilise le principe de "threaded execution", il n'y a donc pas de boucle, mais des sauts dans le code à chaque fin d'instruction pour exécuter la suivante. Les fonctions ont aussi été supprimées pour laisser place à des "#define" étant donné que l'appel de fonction est plus coûteux que l'utilisation de inline. Dans le même principe, la variable servant à stocker l'instruction courante a été remplacée par un "#define" permettant de récupérer cette valeur sans passer par une affectation supplémentaire. Pour terminer, toutes les vérifications ont aussi disparu et maintenant la machine virtuelle peut se comporter de manière imprévisible si le bytecode possède un défaut. Voyons par exemple la même fonction que précédemment montrée :

```
22  // --- Inline for a array index
23  #define DO_ARRAY_INDEX \
24      R_A = data->table_array[(unsigned int) R_B]->content[(unsigned int) R_C];
```

La machine virtuelle est indépendante de l'exécuteur, et offre des fonctions permettant aux exécuteurs de créer, stocker et supprimer des tableaux de manière dynamique et optimisée. La machine possède 8 registres entiers, étant tous stockés dans un tableau et un tableau de tableau d'entier de taille variable. Une fonction permet de créer un nouveau tableau comme vous pouvez le voir sur la prise d'écran suivante :

```
48  unsigned int allocate_table(machine_data_t *data, unsigned int size) {
49
50      // Prepare the new index
51      unsigned int new_table_index;
52
53      // Check if there is a free space in the array
54      if(data->free_start != NULL) {
55
56          new_table_index = (unsigned int) (data->free_start - data->table_array);
57          data->free_start = (table_t **) *data->free_start;
58
59      } else {
60
61          new_table_index = data->table_array_size;
62
63          // Check the array size and double it if needed
64          if(data->table_array_size >= data->table_array_cap) {
65              _double_table_array(data);
66          }
67
68          data->table_array_size++;
69
70      }
71
72      // Create a new table and increase the size
73      data->table_array[new_table_index] = (table_t *) calloc(size + 1, sizeof(int));
74      data->table_array[new_table_index]->size = size;
75
76      return new_table_index;
77
78  }
```

L'autre fonction, elle, permet de supprimer un tableau et de l'ajouter dans la freelist. La gestion de la mémoire est effectuée à partir d'une freelist, une liste chaînée ou chaque élément contient l'adresse du prochain, quand un élément du tableau de tableaux est supprimé, on ajoute donc son adresse au début de la freelist, et lorsqu'on allouera un nouveau tableau, il sera indexé à la première place de libre si il y en a une dans la freelist. En faisant cela, on évite donc de parcourir la liste, ce qui serait extrêmement coûteux.

```c
// --- Function to free a plate table
void free_table(machine_data_t *data, unsigned int index) {

    free(data->table_array[index]);
    data->table_array[index] = NULL;

    if(index == data->table_array_size - 1) {

        // Handle the size changments
        data->table_array_size--;

    } else {

        // Update the free list
        data->table_array[index] = (table_t *) data->free_start;
        data->free_start = &data->table_array[index];

    }

}
```

A la fin de l'exécution du fichier, la machine virtuelle parcourt la mémoire allouée dynamiquement et la libère de manière à éviter les fuites de mémoire. En effet, la machine virtuelle ayant pour but d'être déployé sur le plus d'OS possible, elle doit assurer que la mémoire est bien nettoyée à la fin de son exécution et ne pas se reposer sur les mécanismes de nettoyage du système.

La machine virtuelle est testée par le fichier sandmark.umz, un fichier fourni par le site du concours de programmation susdit. Il s'assure du bon fonctionnement de la machine ainsi que de son efficacité dans tous ses aspects.

Le compilateur

Tout d'abord, voici les exemples de compilation du micro-JS vers la Machine Universelle demandés (faits à la main, ils ne seront pas générés exactement de la même manière par le compilateur qui lui utilisera la pile etc) :

- 2 + 3 :

```
1: ORTHO $0 2
2: ORTHO $3 3
3: ADD $0 $0 $3
```

- 2 + 3 + 4 :

```
1: ORTHO $0 2
2: ORTHO $3 3
```

```
3: ADD $0 $0 $3
4: ORTHO $3 4
5: ADD $0 $0 $3
```

- 2000000000 + 2 :

```
1: ORTHO $3, 0
2: ORTHO $4, 5
3: LOAD $3, $4
4: 2000000000
5: ORTHO $4, 4
6: INDEX $4, $3, $4
```

- print "abc"

```
1: ORTHO $3, 97
2: OUTPUT $3
3: ORTHO $3, 98
4: OUTPUT $3
5: ORTHO $3, 99
6: OUTPUT $3
```

L'analyseur syntaxique du langage a été réalisé à l'aide de flex et bison, deux outils permettant, en C, de créer un analyseur lexical et syntaxique aboutissant à un arbre de syntaxe abstraite. Nous avons modifié quelques règles par rapport à ce qui nous était demandé pour permettre plus de libertés avec le langage, avec l'accord de notre professeur. Il a aussi été réalisé un méthode permettant de donner la ligne et le type lorsqu'une erreur de syntaxe est détectée, dans le futur de ce compilateur, il faudra aussi faire en sorte qu'il ne s'arrête pas si il rencontre une erreur, mais qu'il retourne toutes les erreurs présentes dans le fichier.

Quant au fichier *compiler.c* en lui-même, la première piste a été un unique parcours de l'AST avec une écriture directe du bytecode dans le fichier de retour. Seulement, nous nous sommes vite rendus compte qu'il n'allait pas être possible de fonctionner ainsi pour des opérations plus complexes que des simples additions sur des entiers : c'est l'implémentation de la compilation du *if* utilisant des labels qui nous a fait comprendre que ce n'était pas possible de ne faire qu'un seul passage. En effet, on ne peut pas prédire en combien d'instructions va tenir la conséquence ou même l'alternative du if, et donc il est difficile de calculer la taille du saut à faire dans le programme.

Nous avons donc réfléchi à plusieurs possibilités, à faire plusieurs passages sur l'AST, et notre professeur nous l'a confirmé plus précisément lorsque nous lui en avons parlé. Nous devions donc faire trois passages pour compiler entièrement le programme :

- Un premier pour transformer l'AST en un tableau d'instructions potentiellement labellisées,

- Un deuxième pour associer les labels à leurs numéros de ligne,
- Un troisième pour remplacer les labels par leur numéro de ligne et écrire par la même occasion le bytecode dans le fichier de retour.

Cela a été un gros travail de restructuration et d'abstraction mais une fois bien comprise et assimilée, cette façon de faire a pris tout son sens et est apparue bien plus simple que ce que l'on pouvait s'imaginer.

Nous avons donc nos fonctions de visite d'AST, qui appellent des fonctions qui forgent le tableau d'instructions labellisées. Un exemple avec la traduction du '==', algorithme que nous avons mis au point grâce aux opérateurs *NAND* et *COND_MOVE* (et donc différent de celui proposé par notre professeur) :

```
422     case EQEQ:
423         // Algorithm based on the universal logical operator NAND, following Boole's algebra's rules :
424         _nand(data, TMP2, TMP1, ACC, -1);   // r = NAND(x, y)
425         _nand(data, TMP1, TMP2, TMP1, -1);  // r_x = NAND(r, x)
426         _nand(data, ACC, TMP2, ACC, -1);    // r_y = NAND(r, y)
427         _nand(data, TMP1, TMP1, ACC, -1);   // not_res = NAND(r_x, r_y)
428         // Interpretation of not_res :
429         // (not_res = 0) : x == y
430         // (not_res = 1) : x != y
431         // Initialisation of the result to true (1)
432         _ortho(data, ACC, 1, -1, -1);
433         // Put it at false (0) if not_res = 1
434         _cond_move(data, ACC, 0, TMP1, -1);
435         break;
```

Voici un autre exemple, qui implémente l'opérateur '<' (lower than), demandé dans les questions bonus :

```
443     case LT:
444         // If x/y = 0, then x < y
445         _division(data, TMP1, TMP1, ACC, -1);
446         _ortho(data, ACC, 1, -1, -1);
447         _ortho(data, TMP2, 0, -1, -1);
448         _cond_move(data, ACC, TMP2, ACC, -1);
449         break;
```

On constate que les fonctions appelées prennent 3 catégories d'arguments : tout d'abord *data*, qui permet de faire passer les informations générales liées à la compilation, ensuite une suite de registres définis dans des "#define" au début du fichier, et finalement un label numérique, mis à la valeur -1 lorsque l'instruction n'est pas labellisée, et lorsqu'elle est labellisé cette valeur est associée à un numéro de label (grâce à un compteur dans *data*).

Le cas de l'opérateur *ORTHO* est un peu plus particulier : premièrement il s'agit de notre seul opérateur pouvant être utilisé avec des valeurs immédiates, et deuxièmement c'est donc le seul à qui l'on peut passer un label en valeur à charger.

Voici la représentation de ces instructions en une structure :

```
26  // --- Structure to contain an instruction
27  typedef struct {
28      enum {STD_OP, ORTHO_OP, BIGINT} op_type;
29      union {
30          struct {
31              int opcode;
32              int a;
33              int b;
34              int c;
35          } std_op;
36
37          struct {
38              int opcode;
39              int a;
40              int val;
41              char val_is_target_lbl;
42          } ortho_op;
43
44          int big_int;
45      } content;
46  } instruction;
```

Il y a donc :

- D'une part les opérateurs standards accompagnés de leur opcode (numéro d'opérateur) et des numéros de registre A, B et C, le tout permettant de représenter l'opération en bytecode ensuite.
- D'autre part, l'opérateur spécial *ORTHO*, qui est composé de son opcode, son numéro de registre A, sa valeur *val*, qui contiendra selon le pseudo booléen *val_is_target_lbl* soit une valeur immédiate standard (un entier), soit un numéro de label cible.
- Finalement, un simple "grand" entier (sur 32 bits, car le maximum chargé par l'opérateur *ORTHO* est sur 25 bits).

Un autre écueil rencontré sur le chemin de la compilation a été la labellisation des instructions à compiler : par exemple, comment faire pour labelliser la première instruction du corps d'un while, alors que l'on lance seulement la fonction de compilation de celui-ci ? On ne peut pas seulement passer un argument supplémentaire qui serait le label puisque lors de la non-validation de la condition du while, on doit sauter à la fin du while, où il n'y a aucune instruction, et on ne peut pas revenir en arrière dans l'AST pour labelliser la prochaine instruction après cette compilation de ce while, etc…

Ainsi nous avons choisi tout simplement de créer des instructions neutres (un "`ORTHO TMP1, 0`" accompagné du label cible) lorsqu'il y en avait besoin dont le but serait simplement de porter le label.

Vous pouvez voir l'application concrète de ces arrangements au *while* ci-dessous :

```
253         case WHILE_STMT:
254             lbl_x = data->nb_lbl++; // while_cond
255             lbl_y = data->nb_lbl++; // while_body
256             lbl_z = data->nb_lbl++; // while_end
257 
258             // --- lbl_while_cond :
259             // Labelise
260             _ortho(data, TMP1, 0, lbl_x, -1);
261             // Compilation of the condition expression
262             _compile_expr(stmt->content.while_stmt.cond, data);
263             // Load the body & end labels
264             _ortho(data, TMP2, 0, -1, lbl_z);
265             _ortho(data, TMP3, 0, -1, lbl_y);
266             // Test the result of the condition
267             _cond_move(data, TMP2, TMP3, ACC, -1);
268             // Jump/Loading
269             _ortho(data, TMP1, 0, -1, -1);
270             _load_prog(data, TMP1, TMP2, -1);
271 
272             // --- lbl_while_body :
273             // Labelise
274             _ortho(data, TMP1, 0, lbl_y, -1);
275             // Compilation of the body expression
276             _compile_stmts(stmt->content.while_stmt.body, data);
277             // Jump back to the condition
278             _ortho(data, TMP1, 0, -1, -1);
279             _ortho(data, TMP2, 0, lbl_x, -1);
280             _load_prog(data, TMP1, TMP2, -1);
281 
282             // --- lbl_while_end :
283             // Labelise
284             _ortho(data, TMP1, 0, lbl_z, -1);
285             break;
```

Par faute de temps et malgré l'intérêt porté au projet, nous n'avons malheureusement pas pu finir le compilateur (en ayant pourtant des pistes pour les différentes instructions comme le *let* et les fonctions qui utiliseraient judicieusement la pile et les différents passages de compilation pour créer leurs propres piles de la bonne taille), et aucun jeu de test n'est encore présent. Il serait pertinent pour la suite de proposer quelques tests pour s'assurer du bon fonctionnement de l'analyseur syntaxique et du fait qu'il relève bien les fautes de syntaxe.

Conclusion et suite du projet

Ce projet était un véritable défi, tant au niveau du développement que de l'organisation du travail. Nous avons essayé de fournir le travail le plus propre et clair possible pour que la relecture et l'éventuelle continuité du projet ne pose pas de problème et soit même facilitée. Vous pouvez voir en effet que beaucoup de travail a été réalisé sur les commentaires, les instructions de compilation et d'utilisation, mais aussi sur diverses options comme le fait de pouvoir afficher l'AST ou d'enregistrer dans un fichier l'exécution complète du bytecode. Cependant, ce langage micro-JS n'est pas fini, et il manque de nombreuses choses pour qu'il devienne un langage informatique complet et puisse être utilisé. De plus, le compilateur n'est donc pas finalisé, mais nous avons structuré les choses de façon à ce qu'il puisse être continué aisément.

Une évolution possible de ce langage, étant donné la nature de la machine virtuelle, serait de faire un langage purement fonctionnel ayant une syntaxe C-like, permettant d'aborder la programmation fonctionnelle sans quitter le confort et la fluidité de lecture des langages plus "traditionnels". Le compilateur devra être complété et on peut également penser à des optimisations possibles dans le code (passages répétés à chaque fonction de l'argument *data* qui pourraient être évités), ainsi qu'à un découpage du fichier (par exemple en trois fichiers pour les trois passages de compilation) pour la lisibilité et la modularité. Il serait également intéressant et pratique de créer un système d'erreurs (déjà initié dans notre structure *compiler_data_t*), et enfin de créer une réelle banque de tests complète.