Enseirb-Matmeca
Filière Informatique

Projet de compilation
Rapport final

Description du compilateur `DJ_compil`

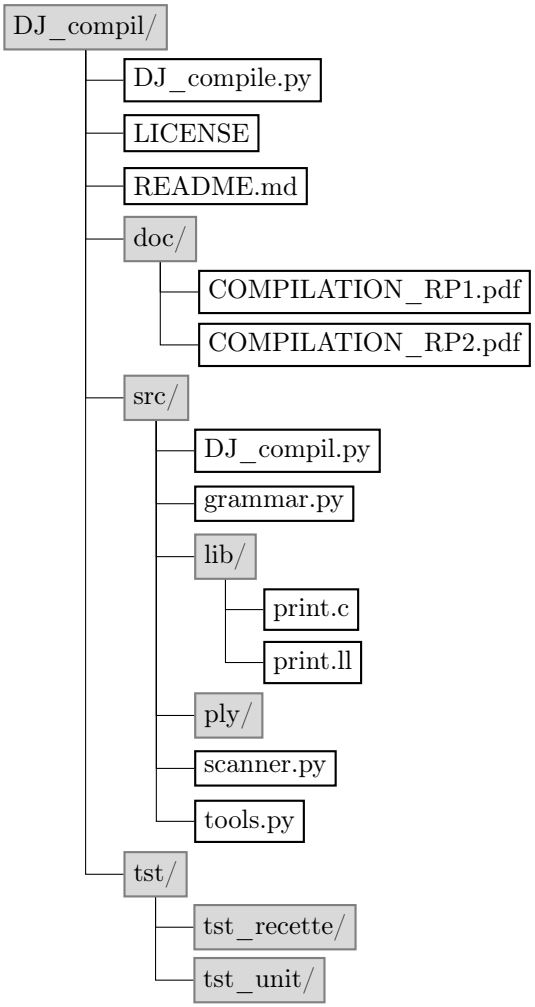
Auteurs :
Florian Le Vern
Sylvain Cassiau

20 décembre 2015

1 Contexte

Pour implémenter notre compilateur `DJ_compil`, nous avons opté pour l'utilisation du langage `Python`. Nous avons pour cela utilisé l'implémentation de `Lex` et `Yacc` pour `Python` nommée `Ply` [1].
L'avantage de ce langage est qu'il contient déjà les structures de données que nous avons besoin d'utiliser dans le cadre de notre compilateur et pour nos vérifications. Notamment les dictionnaires, qui nous ont servi de table de hachage, les listes, la possibilité de créer des classes pour représenter les types avec leurs attributs associés, le tout sans avoir à gérer la mémoire ou à implémenter des structures équivalentes en `C`.

Voici l'arborescence de notre archive :



Le fichier principal de notre compilateur est `DJ_compile.py` situé à la racine de l'archive. Pour compiler un ou des fichier(s) en langage `dj` (comme un exemple du dossier `tst`), et les lier à d'éventuels fichiers `ll` ou `c`, la syntaxe est la suivante :

```
python DJ_compile.py [-o binaire_de_sortie] fichiers_source.<dj|c|ll> ...
```

1. `http://www.dabeaz.com/ply/`

L'option `-o binaire_de_sortie` est facultative. Si elle n'est pas précisée, l'exécutable créé prendra le nom de `a.out`. En modifiant la première ligne du fichier `DJ_compile.py` et en lui donnant les droit d'exécution, il est possible de l'utiliser directement comme un programme.

Ce script fait appel à `src/DJ_compil.py`, qui converti les fichiers écrit en `dj` vers le langage `llvm`, à l'aide de `src/scanner.py` et `src/grammar.py`. Ces deux fichiers contiennent l'implémentation du scanner et du parseur pour le langage `dj`. Il s'agit du coeur de ce projet. Ils sont accompagnés de `src/tools.py` qui est un recueil de fonctions qui permettent de modéliser les données manipulées par le parseur et le scanner. Il ajoute ensuite les librairies d'affichage que nous avons développé pour le langage `dj` situées dans le dossier `src/lib`, puis il cherche une installation des compilateurs `llc` et `gcc` ou de `clang` pour compiler et lier les fichiers obtenus entre eux, afin de générer un exécutable.

2 Fonctionnalités du compilateur

Ce qui est reconnu et ce qui ne l'est pas Les erreurs relevées - portée - syntax - types

2.1 Types

2.1.1 Les valeurs

Le langage `dj` ne comporte que quatre types de base : char, int, float, void.
Des conversions implicites sont possibles pour les opérations et les affectations. On utilise pour cela un tableau à double entrée :

+, -, *, /	char	int	float
char	char	int	float
int	int	int	float
float	float	float	float

Le type `void` n'est pas utilisable dans des expressions, mais uniquement dans les valeurs de retour de fonction.

Le langage fait la différence entre `int` et `char` principalement pour gérer les chaînes de caractères (*cf.* tableaux), mais ces deux types sont convertibles implicitement l'un vers l'autre.

2.1.2 Les fonctions

Le langage `dj` reconnaît des objets de type "fonction". Nous avons fait le choix dans ce langage de masquer tout aspect de pointeur. Nous avons donc retiré le caractère `*` qui se trouvait dans la notation du type "fonction" que nous avions initialement spécifié (`type_retour(*)(type_arg, ...)`). Des variable ayant un type "fonction" sont donc affectable, retournable, et passables en paramètre. Bien sûr, dans l'implémentation `llvm`, ce sont des pointeurs de fonctions qui sont utilisés.

Voici par exemple comment déclarer une telle variable, ici une fonction prenant un tableau de flottants et un caractère en paramètres et retournant un entier :

```
int(float[],char) f;
```

Après avoir été affectée, `f` est ensuite utilisable comme tout autre fonction déclarée.

Une fonction peut prendre en paramètre et retourner des éléments de n'importe quel type du langage (valeurs, fonctions, et tableaux), à l'exception de `void` comme type d'un argument.

Une fonction retournant `void` doit obligatoirement comporter l'instruction `return;`.

2.1.3 Les tableaux

Tout comme pour les fonctions, nous avons souhaité masquer l'utilisation de pointeurs. Le langage `dj` comprend donc un type d'objet "tableau", dont le type est noté `type_des_elements[]`. Nous avons fait le choix de ne pas séparer les `[]` du nom du type comme ce serait le cas en C par exemple. On déclare dont un tableau comme suit :

```
int[] t;
```

Et non pas `int t[];`

Dans l'implémentation `llvm`, nous utilisons une structure pour représenter un tableau. Celle-ci comprend deux champs : taille, et pointeur vers le buffer des éléments.

```
%float.array = type { i32 , float* }
```

Lorsqu'un tableau est créé sans spécifier de taille, comme `t` ci-dessus, sa taille est initialisée à 0, et aucun buffer ne lui est associé. Par contre, si une taille est spécifiée, comme pour `u` ci-dessous, la taille est initialisée, et un appel à `malloc` est effectué pour créer un buffer, dont d'adresse est stockée dans la structure.

```
int[10] u;
```

On accède ensuite aux éléments du tableau comme en C à l'aide du sélecteur `[indice]`. Une fonctions `size()` est fournie par le langage, et retourne le nombre d'éléments du tableau passé en paramètre.

Les chaînes de caractères sont des tableaux de `char`, elles sont donc de type `char[]`, et sont manipulables comme n'importe quel autre tableau. Le langage fournit une technique d'initialisation des tableaux de caractères comme suit :

```
char[] str = "ma chaine\n";
```

Les chaînes notées entre guillemets peuvent contenir des caractères échappés avec un \. Dans notre implémentation, elles sont stockées comme tableau `llvm` global, et copiées dans le buffer correspondant avec `memcpy` lors de la création d'une chaîne.

Il est important de noter que le type des éléments d'un tableau peut être n'importe lequel (à l'exception de `void`), y compris d'autres tableaux. Cela permet de gérer des tableaux multi-dimensionnels. La notion de taille ne faisant pas partie du type d'un tableau, un tableau peut contenir d'autre tableaux qui ont le même types les uns les autres, mais des tailles éventuellement différentes. Il faut cependant faire attention en déclarant des tableaux multi-dimensionnels :

```
int[3][5] t;
```

En effet, seule la dernière taille spécifiée sera allouée. Par exemple, ici, `t` sera un tableau de taille 5 contenant des tableaux d'entiers. Mais les tableaux d'entiers contenus ne sont pas alloués. Il est donc nécessaire de faire une boucle pour les allouer un par un ensuite.

2.2 Structures de contrôle

Le langage `dj` comprend les structures de contrôle `if` suivi ou non de `else`, `for`, `while`, et `do-while`, qui s'utilisent comme en C.

Les boucles `for`, `while`, et `do-while` supportent les instructions `break;` et `continue;`

Les boucles `for` ne permettent pas de créer une nouvelle variable dans leur initialisation.

Ces structures sont réalisées en `llvm` à l'aide de labels et d'instructions `br`.

2.3 Opérations

Les opérations définies sur les type "valeur" sont `+, -, *, /, %`, ainsi que les affectations correspondantes `+=, -=, *=, /=, %=`. Ces types sont également comparables avec les opérateurs `==, !=, <, <=, >, >=`.

Les entiers disposent des opérateur postfixe et suffixe `++` et `- -`.

Les entiers servent de booléens, où 0 est la valeur fausse, le reste étant vrai. Les entiers supportent donc également les opérations `&&, ||, !`.

Plus généralement, les objets sont affectables avec l'opérateur `=` s'ils ont le même type ou si une conversion implicite existe. L'affectation se fait par copie simple, champ par champ. Pour les valeurs, cela correspond à une copie classique. Pour les fonctions, à une copie du pointeur de fonction. Et pour les tableaux, une copie de la taille et du pointeur vers le buffer. On notera ici qu'on ne recopie pas les contenus des buffers, mais seulement le pointeur vers le buffer. Après une affectation, les deux tableaux partagent donc le même buffer.

3 Choix d'implémentation

3.1 La table des symboles

L'utilisation de python permet de gérer facilement les nouveaux contextes. En effet, nous disposons dans le fichier `tools.py` d'une classe `Context`. Celle-ci permet de créer un nouveau contexte ayant pour contexte englobant le contexte précédent, de repasser au contexte d'ordre supérieur, d'ajouter un symbole et son adresse au contexte courant, de gérer les différents labels et registres créés. Ainsi, des appels judicieux dans `grammar.py` aux méthodes de cette classe permettent de gérer la portée des variables vis-à-vis des différentes fonctions, des blocs `if`, `else`, `for` ou `while`.

3.2 Map et reduce

Les fonctions `map` et `reduce` se veulent par définition polymorphes. Comme ni `llvm`, ni le langage `dj` ne prennent en compte cette fonctionnalité, qui correspondrait à des `void *` en `C`, nous avons opté pour la génération par le compilateur d'une fonction `map` et d'une fonction `reduce` par type d'argument. Le code de ces fonctions est ensuite rajouté au programme. Ainsi, à chaque appel d'une fonction `map` ou `reduce` dans le code `dj`, le compilateur vérifie si la fonction correspondante n'a pas déjà été générée, et au choix la génère ou retourne le nom de celle existante.

Ces fonctions `map` et `reduce` sont séquentielles. A aucun moment dans ce projet nous ne vectorisons de code, ou n'utilisons plusieurs threads. En effet, les instructions de base de `llvm` telles `add`, ou `mul` supportent les vecteurs, mais un appel à fonction doit explicitement prendre des vecteurs en paramètre. Il faudrait donc pour chaque fonction générée, créer une version séquentielle et une version vectorisée. Or toute fonction n'est pas vectorisable (dès qu'elle contient un if par exemple). N'ayant réalisé l'ampleur de cette problématique que tardivement, nous avons préféré fournir toutes les fonctionnalitées que nous avions spécifiées originalement, plutôt que nous lancer dans cette vectorisation.

4 Résultats et perspectives

4.1 Ce qui fonctionne

A quelques corrections de syntaxe près dans la grammaire, tous les tests que nous avions fournis au début du projet passent avec succès. Au cours des tests, nous n'avons pas remarqué d'instabilité particulière, de plantage ou de comportement erratique du compilateur. Dans l'ensemble, celui-ci vérifie correctement au cours de la compilation les données qu'il manipule et émet les messages d'erreur adéquats sans planter.

La structure de l'implémentation du compilateur nous semble relativement propre, et peut permettre de rajouter des fonctionnalités au langage `dj` sans nécessiter de grand remaniement.

4.2 Éléments retirés des spécifications

Nous avions prévu de fournir une fonction `free` qui permet de libérer la mémoire du buffer d'un tableau et de remettre sa taille à 0. Mais cela ne remettrait pas trivialement la taille des autres tableaux utilisant ce buffer à zéro. Nous avons préféré masquer l'aspect mémoire et pointeur dans le langage `dj`. C'est pourquoi nous ne fournissons pas cette fonction. Nous trouvons davantage dans l'optique du langage d'implémenter un ramasse miettes libérant les buffers qui ne sont plus pointés par une tableau.

4.3 Fonctionnalitées non implémentées

Certaines fonctionnalités prévues dans les spécifications ont toujours lieue d'être mais nous n'avons pas eu le temps de les implémenter. Cependant, nous avons des pistes de solutions pour le faire et avec un peu plus de temps nous y serions parvenu. Ainsi, la structure de contrôle `switch` n'apparaît pas dans notre grammaire. Il suffirait de rajouter la reconnaissance des mots clés correspondants (`switch`, `case` et `default`) et d'utiliser l'instruction `switch` de `llvm` en s'inspirant du `if`.

Il n'est pas possible de déclarer un index dans l'initialisation de la boucle `for`.

Les tableaux sont alloués avec `malloc` sans que l'utilisateur ne s'en apperçoive mais ils ne sont jamais libérés ! Notre compilateur crée donc à ce jour des fuites mémoires pour chaque utilisation de tableau. Nous aurions aimé ajouter un ramasse miettes pour libérer la mémoire quand elle n'est plus utilisée par aucune variable. Cela aurait été facilement réalisable en ajoutant un pointeur vers un entier global pour chanque buffer à notre structure de tableau correspondant au nombre de variable l'utilisant. Lorsqu'un tableau n'est plus utilisé (après affectation d'un autre tableau ou à la fin d'un contexte) cet entier serait décrémenté et s'il atteint zero alors le buffer serait libéré.

4.4 Fonctionnalités à compléter

Comme présenté à la fin de la section sur les types tableaux, l'allocation de tableaux multi-dimensionnels pourrait être complétée par une allocation des tableaux contenus.

Ainsi dans une potentielle deuxième version de notre compilateur, nous aurions implémenté tout ce qui est cité ci-dessus.