Compilateur – Symphony

Bruno Costa & Diogo Lopes Da Silva

HE-Arc – 2021-2022 - 3 janvier 2022

François Tièche

Table des matières

1 Introduction

Dans le cadre du 1er semestre de troisième année en DLM à l'HE-Arc, il est demandé d'effectuer un projet de compilateur du thème de son choix. Les étudiants peuvent, par exemple, créer un langage de toute pièce ou encore créer un traducteur pour langages. Ce rapport contiendra l'explication et la description de chaque étape effectuée pour ce projet. Que ce soit les différentes analyses ou la logique derrière la création du langage.

Notre groupe a décidé de créer un langage nommé « Symphony ». Ce langage possède une structure simple ressemblant aux langages orientés objets très connus comme le Java ou le C#. Son but est de créer une mélodie ou un son de format MIDI selon les commandes et fonctions entrées par le programmeur.

Ainsi, imaginons que l'utilisateur souhaite créer une gamme de Do avec le son d'un synthétiseur midi, il lui suffira d'entrer les différentes notes à la suite et de compiler son code !

1.1 Exemple de la gamme de do

```
mel time = 1;
mel duration = 1;
mel volume = 100;

add_note("DO", time, duration, volume);
time = time + 1;
add_note("RE", time, duration, volume);
time = time + 1;
add_note("MI", time, duration, volume);
time = time + 1;
add_note("FA", time, duration, volume);
time = time + 1;
add_note("SOL", time, duration, volume);
time = time + 1;
add_note("LA", time, duration, volume);
time = time + 1;
add_note("SI", time, duration, volume);
time = time + 1;
add_note("DO", time, duration, volume);
```

Via ce code, l'utilisateur générera un fichier au format MIDI qui fera vibrer vos oreilles avec une jolie gamme de Do.

2 Analyse

Comme dit plus haut, le langage a été créé pour générer des pistes de sons et de mélodies via du code. Symphony est un langage qui se base sur l'utilisation de fonctions intégrées au langage. En effet, via certaines méthodes, les utilisateurs pourront ajouter une note d'un instrument MIDI, choisir un instrument spécifique ou encore modifier le tempo. Afin d'aider la création de musique, l'utilisateur aura accès à des fonctionnalités de variables et de boucles de répétition.

2.1 Variables

Pour définir une variable, l'utilisateur doit employer le mot clé "mel" (melody) avant le nom de celle-ci :

```
mel var = 10;
```

Si une variable est consultée ou modifiée avant sa définition, elle ne sera pas correcte : une erreur à la compilation apparaîtra.
Les variables sont de type très simple, elles ne peuvent contenir que des entiers ou des chaînes de caractères.

2.2 Boucles

L'utilisateur peut définir et utiliser l'équivalent des boucles "for" dans d'autres langages. Elles sont utilisées pour la répétition de notes. Comme une partition possède des milliers de notes, l'utilisateur ne peut pas se permettre de toutes les noter à la suite.
Elle est définie par le mot clé "range" ainsi que par trois valeurs :

- Une valeur de début, la boucle commencera par celle-ci.
- Une valeur de fin, dernière valeur atteinte
- Une valeur de pas, l'incrémentation que la boucle va faire entre le début et la fin

```
range(0, 10, 1) {

}
```

2.3 Fonctions

Finalement, Symphony possède plusieurs fonctions intégrées qui donnent accès à l'arsenal de création de musiques.
Ces fonctions possèdent 0, 1, 3 ou 4 paramètres :

2.3.1 Zéro paramètre

list_instrument

La fonction “list_instrument” est une fonction de “débogage” qui permet de d’afficher tous les instruments MIDI disponibles à l'utilisateur. Ainsi, lors de la création de son œuvre, le musicien pourra lancer son script pour lister les types de son qu’il peut employer. Ces instruments sont représentés par un numéro qui sera utilisable par une autre fonction décrite ci-dessous

```
list_instrument();
```

2.3.2 Un paramètre

change_instrument

“change_instrument” permet à l’utilisateur d’employer les numéros disponibles par la fonction précédente. En passant un de ceux-ci en paramètre à la fonction, la prochaine note utilisée sera de l’instrument correspondant.

```
list_instrument();
// numéro souhaitée : 32
change_instrument(32);
```

change_octave

“change_octave”, elle, va changer l’octave des prochaines notes. En effet, l’utilisateur possède un accès limité aux notes dites “écrites” : au lieu d’avoir accès à chacune d’entre-elles, il ne pourra qu’utiliser les notes entre DO et SI ainsi que leur “#”. Pour pouvoir élargir son répertoire, cette fonction est utile ! En l’utilisant, les prochaines notes disponibles appartiendront à une octave différente.

```
change_octave(3);
add_note(“DO”, 1, 1, 60);
```

change_track

Lorsque l’utilisateur ajoute une note, il doit l’ajouter sur une “track”, c’est-à-dire une piste de notes. Sur une piste, on ne peut mettre que des notes à la suite et non en même temps, c’est pourquoi, si une mélodie vient à contenir plusieurs sonorités en simultané, c’est que plusieurs pistes sont utilisées. Pour ce faire, la fonction “change_track” est à utiliser. Elle permet de changer de piste (Une mélodie ne possède que 129 pistes → 0 à 128)

```
change_track(12);
```

2.3.3 Trois paramètres

add_tempo

La fonction add_tempo permet d'ajouter un tempo sur la piste voulue. Cette fonction possède trois paramètres :

- track : la piste sur laquelle le tempo sera ajouté
- time : le temps auquel le tempo est placé
- tempo : le tempo en bpm (battements par minute)

Ainsi, l'utilisateur pourra changer de tempo aux moments voulus de sa mélodie.

```
add_tempo(12, 120, 90);
```

2.3.4 Quatre paramètres

add_note

Finalement la méthode la plus importante, sans celle-ci aucune musique ne pourrait être créée. "add_note" est là pour ajouter une note sur une piste à un moment donné. Elle possède 4 paramètres :

- note : la valeur de la note
- time : à quel moment la note apparaîtra dans la piste
- duration : la durée de la note
- volume : le volume de la note

Cependant, il existe une particularité à la valeur de la note. En effet, l'utilisateur peut employer deux méthodes pour ajouter cette valeur :

- La méthode écrite → une note peut être définie par un texte ("DO", "RE", "MI", "FA", "FA#", …). Cependant, il faut choisir la bonne octave avant de l'employée.
- La méthode valeur → une note peut être définie par une valeur MIDI (Valeur entre 0 et 127)

Note	-1	0	1	2	3	4	5	6	7	8	9
C	0	12	24	36	48	60	72	84	96	108	120
C#	1	13	25	37	49	61	73	85	97	109	121
D	2	14	26	38	50	62	74	86	98	110	122
D#	3	15	27	39	51	63	75	87	99	111	123
E	4	16	28	40	52	64	76	88	100	112	124
F	5	17	29	41	53	65	77	89	101	113	125
F#	6	18	30	42	54	66	78	90	102	114	126
G	7	19	31	43	55	67	79	91	103	115	127
G#	8	20	32	44	56	68	80	92	104	116	
A	9	21	33	45	57	69	81	93	105	117	
A#	10	22	34	46	58	70	82	94	106	118	
B	11	23	35	47	59	71	83	95	107	119	

3 Conception

Les prochains paragraphes expliqueront les différentes analyses par lesquelles le code va passer avant d'être compilé. En effet, un code est avant tout un fichier texte dont il faut traiter chaque ligne et chaque mot.
Comparons ainsi une ligne de code avec une phrase.
La première étape est l'analyse lexicale, c'est ici que les "phrases" sont décortiquées et les mots reconnus. Vient ensuite l'analyse sémantique, où nous allons détecter le sens de la phrase et vérifier s'il est correct ou non.
Finalement, il y a l'étape de la compilation et de l'analyse sémantique. Ici, les erreurs d'écriture sont gérées et le code est compilé pour créer un fichier MIDI.

3.1 Analyse lexicale

Dans l'analyse lexicale on s'occupe de vérifier que la grammaire de notre code est correcte. Chaque ligne de code est séparée en mots selon des critères bien définis. Symphony est un langage simple qui possède une logique lexicale bien à lui. En premier lieu, Symphony possède les traits d'un langage de base : son analyse lexicale détecte ainsi les éléments de base connus via des expressions régulières (regex).

- Les identifiants → élément qui permet de reconnaître un nom de variable ou un nom de fonction
- De simple nombres
- Des chaînes de caractères sous la forme **"string"**
- Les retours à la ligne

Finalement, il doit également pouvoir reconnaître des caractères ou éléments qui désignent une action, un séparateur, une opération mathématique ou un regroupement.

- Les parenthèses
- Les additions et multiplications
- Les points virgules
- Les virgules
- Les accolades
- Le signe égal

Afin d'assurer une validité du code logique, l'analyse lexicale ignore également aussi les espaces et les tabulations.
Quand une séquence ne correspond à aucune de celles mentionnées plus haut, le package lexem renvoie un feedback indiquant une erreur lexicale.

3.2 Analyse syntaxique

Le but de l'analyse syntaxique est de vérifier que l'analyse précédente, l'analyse lexicale, ait récupéré des "mots" qui ensemble ont un sens. En effet, après l'analyse lexicale il faut pouvoir donner un sens à nos lignes de code.
Pour que ceci fonctionne, il faut créer un arbre syntaxique (AST). Chacun des nœuds de l'arbre représente une règle, une partie syntaxique du langage.
Symphony peut accepter différents types de règles syntaxiques, de la boucle ou simplement de la déclaration de variable, c'est dans ce paragraphe que ces différentes règles seront expliquées.

3.2.1 Règles de définition du programme

Les deux premières règles permettent de définir un programme depuis sa racine. La première indique que le programme peut être représenté par un "statement" et la seconde que le programme peut être représenté par un "statement" et un programme. Ainsi nous avons une déclaration récursive qui met en place un programme entier (comprenant tous les "statements" du code).

3.2.1.1 Statement

Depuis le début, le terme statement est évoqué, mais qu'est-ce qu'il représente ? Pour Symphony, un statement peut-être plusieurs choses.

- Une structure : La boucle range décrite plus haut
- Une assignation de variable
- Un appel de fonction

3.2.2 Assignation

L'assignation de variable fonctionne de deux manières :

- Déclaration de variable via le mot clé "mel"
- Assignation de valeur à une variable déjà déclarée

3.2.3 Structure

La seule structure du langage Symphony est la boucle range(start, end, step). Pour son bon fonctionnement, il a fallu créer un node spécifique dans le fichier AST.py. Ce node permet de gérer la range.

```
class RangeNode(Node):
    type = "range"
```

3.2.4 Expressions

Les expressions correspondent aux valeurs du langage. Par exemple un nombre entier ou une chaîne de caractères. Si l'utilisateur souhaite utiliser une valeur ou simplement la stocker dans une variable, il doit passer par la déclaration d'expression.
Les expressions sont donc :

- Un nombre entier
- Une chaîne de caractère
- Un identifiant → Un nom de variable permettant de récupérer sa valeur

De plus, il existe deux petites fonctionnalités supplémentaires aux expressions.
La possibilité d'avoir des nombres négatifs et les opérations basiques entre les nombres entiers : addition, soustraction, division et multiplication.

3.2.5 Fonctions

Pour les fonctions il a également fallu ajouter une node "FuncNode" pour les représenter. Plusieurs cas de fonctions sont analysés selon le nombre de paramètres comme indiqué précédemment. Chaque paramètre d'une fonction est représenté par une expression et le nom de la fonction est un identifier.

```
class FuncNode(Node):
    def __init__(self, func_name, children=None):
        Node.__init__(self, children)
        self.func_name = func_name

    def __repr__(self):
        return "%s (%s)" % (self.func_name, tuple(self.children))
```

3.3 Analyse sémantique

Plusieurs points sont vérifiés dans le parseur et le compilateur. Le compilateur s'assure que la valeur de début de la boucle soit plus petite que celle de fin si le pas est positif, au contraire si le pas est négatif il faut que le début soit supérieur à la fin.

Si le programmeur veut accéder à une variable, le compilateur vérifie que cette variable existe avant d'y accéder. Le compilateur vérifie également que l'appel d'une fonction avec un nom connu ait le nombre de paramètres correspondant. Dans les fonctions, des checks des valeurs sont effectués. Par exemple, l'octave ne peut pas aller en dessous de -1 et au-delà de 9. La track ne peut pas aller en dessous de 0 et dépasser 128. Si on indique la note directement en integer, on vérifie si elle se trouve entre 0 et 127. Si on indique le nom de la note, on regarde dans un table associative si le nom est valide, s'il l'est on calcule la valeur de la note en fonction de l'octave puis on vérifie si cette note se trouve entre 0 et 127.

Pour finir, si le programmeur appelle une fonction, quelle qu'elle soit, avec un nombre de variables inconnues, le parseur va de son côté rapporter l'erreur. De cette façon, seules des fonctions avec 0, 1, 3 ou 4 paramètres vont passer le test du parseur.

4 Guide Utilisateur

Le guide d'installation des modules python nécessaires se trouve dans le README.MD sur le repos gitlab.
https://gitlab-etu.ing.he-arc.ch/bruno.costa/symphony/-/blob/main/README.md

Il faut d'abord créer un fichier texte et coder le comportement de notre programme.
Des exemples se trouvent dans https://gitlab-etu.ing.he-arc.ch/bruno.costa/symphony/-/tree/main/Test%20Files
Pour toutes informations supplémentaires sur la construction d'un fichier source, se référer au point 2. Analyse.

Pour récupérer le compilateur il faut se rendre sur git et cloner le projet avec un ***git clone lien_repos_clone***.

Une fois le projet récupéré, on peut lancer le fichier symphony_compiler.py dans une console en donnant comme paramètre le chemin du fichier source.

Le fichier .mid sera créé à la racine du projet Symphony et sera nommé selon le nom du fichier source : ***file_name.mid***.

Pour lancer le fichier mdi, nous recommandons de l'ouvrir avec VLC Media Player :
https://www.videolan.org/vlc/index.fr.html

5 Résultats obtenus

Pour nos tests nous avons utilisés les différents fichiers se trouvant dans https://gitlab-etu.ing.he-arc.ch/bruno.costa/symphony/-/tree/main/Test%20Files

Notamment *input6.txt* qui rassemble tout, du changement de tempo, changement d'octave, de track, des variables ainsi que des boucles.
Quand on lance le fichier symphony_compiler.py, qui est la version compilée du fichier *input6.txt,* on remarque que les sonorités sont comparables à ce qui se trouve dans le fichier texte. Ainsi on peut conclure que les résultats obtenus sont les résultats escomptés et que notre compilation fonctionne comme voulue.

6 Conclusion

Pour conclure, on peut affirmer que les différents points du cahier des charges ont été respectés. En effet, Symphony crée un fichier midi à partir de son propre langage en passant par un compilateur python et tous les éléments pour pouvoir créer un son complet sont présents dans le langage.
Des vérifications d'erreur sont également mises en place comme expliqué dans l'analyse sémantique pour s'assurer du bon déroulement de la compilation. Certains points sont toujours améliorables, notamment reconnaître des fonctions avec *n* paramètres ou bien ajouter plus d'analyse sémantique pour les erreurs qui persistent. Globalement le projet est dans un très bon état et totalement utilisable.