Projet de compilateur

Partie 0 : Conception d'un interpréteur

Cours Turing

Introduction

Pour ces 5 dernières semaines de cours, nous allons nous atteler à concevoir un compilateur pour `ALAN`[1], un langage de programmation inventé de toutes pièces pour ce cours. Pour les quatre prochains cours, nous allons suivre dans les grandes lignes les principales phases d'un compilateur :

1. La phase d'analyse lexicale.
2. La phase d'analyse syntaxique.
3. La phase de vérification des types.
4. La phase de génération de code.

Avant de se plonger dans ces quatre étapes, nous allons nous intéresser aujourd'hui à la conception d'un *interpréteur* pour `ALAN`. Un interpréteur est un programme qui permet directement d'exécuter un programme (et non de le traduire dans un autre langage). Concevoir un interpréteur nous permettra de nous familiariser avec notre nouveau langage de programmation ainsi qu'avec la notion d'AST. Afin de simplifier cette étape, nous allons directement écrire les programmes sous forme d'AST en Python, ce qui permettra de nous passer des phases d'analyse lexicale et d'analyse syntaxique normalement nécessaires à un interpréteur. Nous nous passerons aussi de la phase de vérification des types.

Pour l'étape du jour, nous n'allons pas aborder la question de la syntaxe concrète de notre langage, c'est-à-dire de la façon dont les programmes sont représentés de manière textuelle. Nous allons uniquement aborder sa syntaxe abstraite, et plus particulièrement la représentation de programme sous forme d'arbre (AST).

Notre langage de programmation

`ALAN`, le langage source de notre compilateur, et de notre interpréteur de ce jour, est un langage inventé de toutes pièces pour ce cours. Il s'agit d'un langage volontairement minimaliste (sans être simpliste). Le langage est statiquement typé, ce qui signifie que les expressions du langage ont toute un type qui est calculé et vérifié lors de la compilation. Le langage supporte

1. Approachable Language for Assiduous Neophytes

les fonctions d'ordre supérieur, c'est-à-dire que les fonctions sont des valeurs comme les autres, qui peuvent être stockées dans des variables, passées comme argument à un appel de fonction ou retournées d'une fonction. Comme autres valeurs de base, `ALAN` supporte les nombres entiers, les chaînes de caractères, les booléens et `None`.

Fichiers du projet

Un certain nombre de fichiers sont fournis pour vous lancer dans ce projet relativement conséquent. Vous trouverez une archive avec ces fichiers sur Moodle. Veuillez les extraire dans un dossier de travail et l'ouvrir avec votre éditeur de code favori (comme VSCode par exemple).

Étapes du jour

Pour aujourd'hui, vous allez concevoir un interpréteur pour notre langage de programmation. Cette conception se fera en plusieurs étapes qui sont indiquées ci-dessous et détaillées dans la suite de ce document.

1. Prise de connaissance du code fourni,
2. Implémentation de la méthode d'évaluation,
3. Ajout des opérations *et* et *ou*,
4. Implémentation de programmes simples en `ALAN`.

1 Prise de connaissance du code fourni

La première étape du jour consiste à prendre connaissance des fichiers fournis, à savoir :

1. `main.py`, le point d'entrée de notre interpréteur,
2. `trees.py`, la définition de l'AST,
3. `env.py`, la définition de la notion d'environnement,
4. `primitives.py`, les fonctions primitives du langage,
5. `test.py`, une batterie de tests pour l'étape du jour.

1.1 `main.py`

Dans le fichier `main.py`, observez comment peut se décrire un simple programme à l'aide de constructions telles que `Application`, `Primitive` et `Literal`. Notez aussi la façon dont on appellera la méthode d'évaluation afin d'obtenir le résultat d'une expression.

1.2 `trees.py`

Dans le fichier `trees.py`, vous trouverez la définition des différents types d'expressions du langage, à savoir :

`Literal` Une expression littérale, c'est-à-dire une valeur directement écrite. Les valeurs sont soit des entiers, des chaînes de caractères, des booléens ou `None`. Le constructeur prend comme argument directement la valeur en Python.

`Variable` Une référence à une variable. Le constructeur prend comme argument le nom de la variable, une chaîne de caractères Python. Au moment de l'évaluation, la valeur de la variable sera lue dans l'environnement.

`Assignment` Une assignation à une variable. Le constructeur prend deux arguments : le nom de la variable et l'expression à calculer pour obtenir sa valeur.

`Primitive` Une référence à une construction primitive du langage. Le nom de la primitive est à donner comme argument au constructeur.

`Block` Un bloc de code, avec un certain nombre de déclarations de variables et d'expressions. Une déclaration de variable consiste en un nom de variable et un type. Toutes les variables déclarées sont visibles dans l'entièreté du bloc.

`IfThenElse` Une expression conditionnelle, équivalent à `if`/`else` en Python. La constructeur prend trois arguments : la condition, l'expression si la condition est vraie, l'expression si la condition est fausse.

`While` Une boucle `while`. Le constructeur prend deux arguments : la condition et le corps de la boucle.

`Abstraction` Une fonction anonyme, similaire à `lambda` en Python. Le constructeur prend deux arguments : une liste de déclarations de variables (les paramètres de la fonction) et le corps de la fonction. La fonction peut être assignée à une variable pour lui donner un nom.

`Application` Application d'une fonction à des arguments, autrement dit appel d'une fonction. Le constructeur prend deux arguments : la fonction à appliquer et la liste des arguments. Notez que la fonction à appliquer, tout comme les arguments, sont des expressions quelconques. La fonction à appeler peut être contenue dans une variable, être le résultat d'un appel de fonction ou de tout autre expression susceptible de résulter en une fonction.

1.3 `env.py`

Le fichier `env.py` contient la définition de la class `Env`, qui sert à définir des environnements. Un environnement représente une collection de noms de variables, chacun associé à une valeur. Les environnements peuvent être étendus pour permettre temporairement d'y ajouter des entrées. Cette fonctionnalité sera très utile dans le cas des blocs de code avec des déclarations de variables, ou dans le cas des fonctions avec des paramètres. Observez les différentes méthodes de la classe `Env` et essayez de comprendre leur usage.

1.4 `primitives.py`

Dans le fichier `primitives.py` vous trouverez un dictionnaire, `PRIM_VALS` associant chaque nom de primitive à une valeur en Python. Le nom de la primitive correspond à l'argument donné au constructeur de la classe `Primitive`. Observez quelles sont les primitives proposées.

1.5 `test.py`

Finalement, dans le fichier `test.py`, vous trouverez toute une série de tests pour l'interpréteur que vous allez concevoir aujourd'hui. Les tests présents dans ce fichier vous permettent aussi de vous familiariser avec la façon de construire des expressions. Vous êtes libres d'y ajouter des tests.

2 Implémentation de la méthode d'évaluation

Abordons maintenant la question de l'évaluation des expressions. Le but de l'évaluation est de calculer la valeur d'une expression de notre langage. Pour représenter les valeurs, nous utiliserons directement une valeur de Python. Ainsi, on utilisera des entiers Python pour représenter une valeur entière ou encore une fonction Python pour représenter une valeur fonction.

La fonction d'évaluation prendra la forme d'une méthode de la classe `Expression` et de ses diverses sous-classes. La méthode a deux paramètres : `self`, la référence à l'objet, et `env`, l'environment dans lequel l'expression doit être évaluée.

Votre but dans cette étape est d'implémenter la méthode `eval` pour les différentes sous-classes d'`Expression`. Après avoir implémenté chaque cas, testez votre code à l'aide des tests fournis dans `test.py`. Profitez aussi de l'opportunité pour ajouter des tests supplémentaires.

Ci-dessous vous trouverez des instructions pour vous aider à implémenter la méthode `eval` pour les diverses constructions du langage.

`Literal` Lorsque l'on évalue une expression littérale, la valeur à retourner est directement celle passée en argument au constructeur.

`Variable` La valeur d'une variable est directement donnée par l'environnement.

`Assignment` Lors de l'évaluation d'une assignation de variable, on commence par évaluer l'expression qui donne la valeur. Ensuite, l'environment est mis-à-jour afin de refléter la nouvelle valeur. Enfin, comme résultat, la nouvelle valeur de la variable est retournée comme valeur de l'expression.

`Primitive` Pour les primitives, il suffit de retourner la valeur indiquée dans le dictionnaire des valeurs primitives.

`Block` La première chose à effectuer lorsque l'on évalue un bloc est de créer un nouvel environment qui étend l'environment courant. Dans ce nouvel environnement, l'on déclare ensuite une nouvelle variable pour chaque déclaration de l'environment. Attention, la liste des déclarations contient à la fois le nom des variables mais aussi leur type. L'environnement, au moment de l'évaluation, ne se soucie que du nom de la variable.

Ensuite, les expressions sont évaluées une à une dans ce nouvel environnement. La valeur de la dernière de ces expressions est retournée comme valeur pour le bloc entier.

`IfThenElse` Lors de l'évaluation d'une expression conditionnelle, on commence toujours par évaluer la condition. Si la condition est vraie, on procède à l'évaluation de l'expression donnée dans ce cas précis. Dans le cas contraire, on procède à l'évaluation de l'autre expression. Dans les deux cas, le résultat de cette évaluation est retourné comme résultat de l'expression conditionnelle.

`While` Lors de l'évaluation d'une boucle `while`, on commence toujours par évaluer la condition. Si la condition est vraie, on évalue le corps de la boucle puis l'on recommence. Lorsque finalement la condition est fausse on s'arrête. La valeur d'une expression `while` est toujours `None`.

`Abstraction` La définition de la méthode `eval` pour la classe `Abstraction` vous est fournie. Néanmoins, essayez de comprendre l'implémentation donnée.

Pour commencer, on définit une fonction Python qui admet un nombre quelconque d'arguments (qui seront stockés dans la liste args). C'est cette fonction qui servira de valeur à l'évaluation de cette expression.

Cette fonction, à chaque fois qu'elle sera exécutée, commencera par étendre l'environment dans lequel la fonction a été *définie* pour y déclarer les paramètres de la fonction et leur associer les valeurs reçues en arguments à l'appel. Ensuite, la fonction procédera à l'évaluation du corps de la fonction dans ce nouvel environnement.

`Application` Lors de l'évaluation d'une application de fonction, on commence par évaluer la fonction elle-même afin d'obtenir (en principe) une fonction Python. Ensuite, on procède à l'évaluation successive des différents arguments de la fonction. Finalement, on appelle la fonction Python obtenues avec les valeurs obtenue pour les arguments. Point technique en Python : pour *dérouler* une liste de valeurs `args` et passer chaque valeur en tant qu'argument à un appel à une fonction `fun`, on note `fun(*args)`. La valeur retournée par cet appel de fonction est utilisée comme valeur pour l'expression.

3 Ajout des opérations *et* et *ou*

L'étape d'après est d'implémenter les opérateurs *et* et *ou* pour notre langage. L'idée de base serait de les rajouter parmi les primitives, ce qui aurait pour avantage d'être très simple. Cependant, dans notre langage, nous allons décider d'évaluer de manière un petit peu particulière *et* et *ou*.

Dans le cas de *et*, on commencera par évaluer la première expression. Si le résultat de la première expression est faux, on ne se donnera pas la peine d'évaluer la seconde expression, et on retournera directement la valeur fausse. Dans le cas où le résultat de la première expression est vrai, on procèdera à l'évaluation de la seconde expression et on retournera son résultat.

De manière similaire pour *ou* : dans le cas où la première expression est *vraie*, on ne procédera pas à l'évaluation de la seconde expression.

On parle de *short-circuit evaluation* pour faire référence à cette méthode d'évaluation que l'on adopte généralement dans les langages de programmation (y compris Python).

Pour implémenter ces deux opérateurs, on procédera donc à la création de deux sous-classes d'`Expression` : une première appelée `And` et une seconde appelée `Or`. Votre travail consiste à implémenter ces deux classes et leur méthode d'évaluation de zéro. Vous êtes aussi encouragés à ajouter des tests dans le fichier `test.py` afin de vérifier le bon fonctionnement de ces deux opérateurs.

4 Implémentation de programmes simples en ALAN

Une fois la fonction d'évaluation d'expression terminée et les deux opérateurs *et* et *ou* implémentés, il ne vous reste plus qu'à programmer dans votre langage ! Voici quelques idées de programmes à implémenter :

— Définition et utilisation d'une fonction `factorielle`,
— Jeu du *plus petit/plus grand*,
— Le traditionnel *fizzbuzz* !

Ceci n'est qu'une liste de suggestions, n'hésitez pas à faire preuve de créativité !