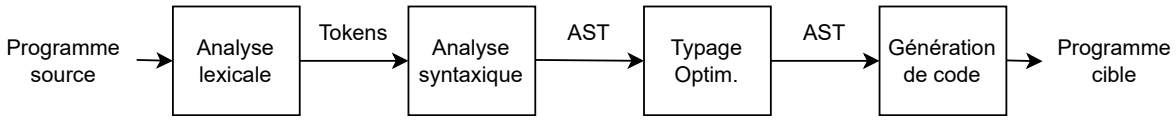
Projet de compilateur

Partie 4 : Génération de code et finalisation du compilateur

Cours Turing

Introduction

Pour cette dernière partie du projet, vous allez terminer l'implémentation de votre compilateur pour `ALAN`. Jusqu'à présent, vous avez implémenté un lexer, un parser et un vérificateur de types pour le langage `ALAN`, qui permettent de transformer un programme écrit en `ALAN` en un arbre de syntaxe abstraite (AST) et de vérifier que le programme est bien typé. Il ne reste plus qu'à ajouter une dernière étape à notre compilateur : la génération de code. Le langage cible de votre compilateur, dans un souci de simplicité, est le langage Python. Nous allons donc nous atteler à générer du code Python à partir du code `ALAN`.



Étapes

Notre travail du jour sera décomposé en plusieurs étapes :

1. Correction d'une erreur dans le code fourni pour le parser.
2. Ajout de la génération de code à votre compilateur.
3. Modification du fichier `main.py` pour assembler toutes les parties de votre compilateur.
4. Ajout du support des noms de fonctions primitives (comme `input` ou `print`).
5. Implémentation de quelques programmes en `ALAN` pour tester votre compilateur.
6. Implémentation d'améliorations supplémentaires (facultatif).

1 Génération de code

Tout d'abord : Bonne nouvelle ! Vous n'aurez pas aujourd'hui à écrire vous-même un générateur de code de zéro. Une implémentation de base, pleinement fonctionnelle, vous est déjà fournie. Comme première étape, je vous invite donc à télécharger le fichier `codegen.py` depuis Moodle et à le placer à la racine de votre projet.

Ouvrez ensuite le fichier `codegen.py` dans votre éditeur de code pour jeter un oeil à l'implémentation. La fonction principale de ce fichier, `generate`, implémente la génération de code Python à partir d'un AST. Pour ceci, la fonction utilise du *pattern matching* afin de traiter les différents types de noeuds de l'arbre. Notez que la fonction est implémentée de manière récursive.

Le fichier `codegen.py` exporte cette même fonction `generate`. La fonction prend en paramètre l'AST et renvoie le code Python généré sous la forme d'une chaîne de caractères. C'est en faisant appel à cette fonction que vous allez pouvoir obtenir le code Python d'un AST donné dans votre compilateur.

Le code est déjà fonctionnel, mais il est possible de l'améliorer pour qu'il génère du code plus efficace ou plus lisible.

2 Implémentation de `main.py`

La prochaine étape consiste à modifier le fichier `main.py` pour donner un point d'entrée à notre compilateur.

Ouvrez le fichier `main.py` et faites en sorte de :

1. Lire un fichier `ALAN` en entrée. Cela peut être fait via un fichier passé en argument en ligne de commande, ou, plus simplement, via une entrée standard lue grâce à la fonction `input`.
2. Utiliser le lexer et le parser pour obtenir l'AST du code `ALAN` lu. Pour cela, vous avez normalement simplement à importer la classe `TokenStream` et la fonction `parse_expression` du fichier `parser.py`. En principe, il suffit de faire

   ```
   expr = parse_expression(TokenStream(text))
   ```

   pour obtenir l'AST. En cas d'erreur, affichez un message d'erreur et arrêtez le programme.
3. Utiliser la méthode `type` de l'AST pour vérifier que le code `ALAN` est correct. Donnez comme argument un environnement de typage vide (`TypeEnv()`, disponible dans `typechecking.py`).
4. Appeler la fonction `generate` du fichier `codegen.py` pour obtenir le code Python correspondant à l'AST.
5. Afficher le code Python généré, ou l'écrire dans un fichier.

Une fois le code Python généré, vous pouvez l'exécuter de la façon habituelle, en utilisant la commande `python` ou `python3`. Il est aussi possible d'appeler directement la fonction `exec` de Python pour exécuter le code généré, transformant ainsi votre compilateur en un interpréteur.

Pour plus de détails sur la gestion des fichiers en Python, vous pouvez consulter la documentation officielle : `https://docs.python.org/fr/3.11/tutorial/inputoutput.html#reading-and-writing-files`. De même, si vous souhaitez offrir une interface plus avancée à votre compilateur, vous pouvez consulter la documentation de la bibliothèque `argparse` de Python : `https://docs.python.org/fr/3/library/argparse.html`.

3 Support des fonctions primitives

Pour le moment, notre compilateur ne supporte pas totalement les fonctions primitives comme `input` ou `print`. Il y a bien un noeud qui correspond à ces fonctions dans l'AST (`Primitive`), mais le parser ne produit pas ces noeuds. À la place, le parser produit des noeuds de type `Identifier` pour les noms de fonctions primitives comme tout comme pour les autres identifiants.

Pour ajouter le support des fonctions primitives, il y a deux manières de procéder :

1. Modifier le parser pour qu'il produise des noeuds de type `Primitive` pour les fonctions primitives comme `print`, `input`, `tostr` ou encore `toint`. Pour arriver à cela, il faudra aussi modifier le lexer afin de reconnaître les noms de fonctions primitives et produire un token distinct pour ces noms.

   Cette solution est la plus simple à mettre en place mais elle nécessite que vous modifiiez le lexer (ajouter un type de token, une règle) et le parser (tenir compte de l'existence du nouveau token pour les expressions basiques).

   À noter que cette solution apporte une légère restriction : il n'est pas possible, pour les utilisateurs du langage, de définir des identifiants qui ont le même nom que des fonctions primitives.

2. Concevoir un *namer* qui transforme les noeuds de type `Identifier` en noeuds de type `Primitive` pour les noms de fonctions primitives dans le cas où ces noms ne sont pas déclarés dans le contexte du noeud.

   Cette solution est plus complexe à mettre en place mais elle permet de garder le comportement actuel du lexer et du parser. Elle permet aussi aux utilisateurs du langage de définir des identifiants qui ont le même nom que des fonctions primitives.

   Pour implémenter cette solution, vous pouvez ajouter un nouveau fichier `namer.py` à votre projet. Ce fichier devra contenir une fonction `name` qui prend en paramètre un AST et renvoie un AST modifié. Cette fonction devra transformer les noeuds de type `Identifier` en noeuds de type `Primitive` pour les noms de fonctions primitives *non déclarés* dans le contexte du noeud. Il y a donc encore une fois la notion d'environnement à prendre en compte.

   Faites aussi attention à ne pas perdre d'informations lors de la transformation. Par exemple, si vous avez indiqué une position dans le code source pour un noeud, il est important de conserver cette information dans le nouveau noeud.

En terme de difficulté, la première solution est plus simple à mettre en place mais la deuxième solution est plus flexible. La deuxième solution pourrait être l'occasion de mettre en pratique le pattern matching sur un exemple concret.

4 Programmation en ALAN

Pour tester votre compilateur, je vous invite à écrire quelques programmes en `ALAN` et à les compiler à l'aide de votre compilateur maison. Inspectez le code Python généré et exécutez-le pour vérifier que le programme fonctionne correctement.

Voici quelques idées de programmes à écrire :

— Un programme qui affiche un message à l'utilisateur et lui demande de saisir un nombre. Le programme doit ensuite afficher le double de ce nombre.
— Un programme qui affiche la somme des entiers de 1 à 100.
— Un programme qui affiche les 10 premiers termes de la suite de Fibonacci.
— Un programme qui joue au jeu du plus ou moins avec l'utilisateur.

5 Améliorations supplémentaires

Si vous avez terminé toutes les étapes précédentes et que vous avez encore du temps, voici quelques idées d'améliorations supplémentaires que vous pourriez apporter à votre compilateur :

1. Ajout d'une primitive pour la génération de nombres entiers pseudo-aléatoires. Une telle primitive pourrait être utile pour la création de jeux ou de simulations, voire plus comme vous avez eu l'occasion de le voir cette année.
2. De meilleurs messages d'erreur, avec des informations plus précises sur la position de l'erreur dans le code source.
3. Ajout des commentaires dans le langage `ALAN`. Vous pourriez par exemple utiliser le symbole `#` pour permettre d'insérer un commentaire qui s'étend jusqu'à la fin de la ligne.
4. Modification du générateur de code pour qu'il produise du code plus efficace. Le générateur de code actuel est assez simpliste et produit du code Python qui n'est pas toujours très efficace. Par exemple, pour une addition, le générateur de code actuel produit du code Python qui ressemble à

   ```
   (lambda x, y: x + y)(4, 7)
   ```

   alors qu'il serait plus efficace de simplement produire

   ```
   4 + 7
   ```

   voire même simplement l'expression constante `11`.
5. Ajout des listes à votre langage. Vous pouvez commencer par ajouter une primitive pour la création de listes vides, puis ajouter des primitives pour l'ajout d'éléments à une liste, la récupération d'un élément à un index donné, etc.

La liste ci-dessus n'est de loin pas exhaustive. Si vous avez d'autres idées d'améliorations, n'hésitez pas à les implémenter. Il s'agit de votre compilateur, et vous êtes libre de l'adapter à vos besoins et à vos envies !

Conclusion

Vous voilà arrivés à la fin de ce projet de compilateur, félicitations ! J'espère que, malgré les difficultés rencontrées, vous avez apprécié ce projet et que vous avez appris beaucoup de choses en chemin. Le projet sur lequel vous avez travaillé est un projet complet de compilateur, qui couvre toutes les étapes de la compilation, de l'analyse lexicale à la génération de code. C'est un projet très ambitieux et un exemple de programme non-trivial comme on peut être amené à en écrire dans le monde professionnel ou académique. J'espère que la réalisation de ce projet vous aura donné une compréhension plus profonde du fonctionnement des compilateurs et des langages de programmation, et surtout donné l'envie d'en apprendre plus sur le sujet !