eBook Gratuit

APPRENEZ compiler-construction

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#compiler-construction

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: compiler-construction

It is an unofficial and free compiler-construction ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official compiler-construction.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec le compilateur-construction

Examples

Pour commencer: Introduction

Conditions préalables

- **Avoir une connaissance approfondie d'un langage de programmation** tel que Python, C, C ++, Ruby ou tout autre langage existant.
- **Avoir votre éditeur de code préféré ou votre IDE installé** (par exemple, VSCode)
- **Restez motivé.** Construire un compilateur n'est pas facile, alors continuez à pousser; ça en vaut la peine.

Catégories de langue

Lors de la création d'un compilateur, vous devez décider lequel des deux types de langage sera le compilateur.

- **Langage de jouet:** C'est à ce moment que vous créez un langage de programmation qui ne résout pas un problème, mais est destiné à l'apprentissage. Des exemples amusants de ceux - ci sont `Whitespace` , `Lolcode` et `Brainfuck` .

- **Langage de programmation:** Ce sont les langages qui visent à résoudre un problème ou à apporter quelque chose de nouveau et unique à la table. Ceux-ci peuvent être comparés à des langages tels que `Swift` , `C++` et `Python` .

Ressources

Au cours de votre voyage, il est inévitable que vous tombiez sur quelque chose dont vous n'avez aucune idée, mais j'espère que l'une de ces ressources vous aidera.

- **Créez votre propre langage de programmation** (Ebook)
  - + Amical aux débutants
  - + Court
  - + Aidé la création de `Coffeescript` et `Rubby`
- **Compilers: Principes, techniques et outils (The Dragon Book)**
  - Contient tout ce que vous voulez savoir sur un compilateur, mais il est avancé et long
- **Modern Compiler Design** (Ebook)
  - Ceci est un autre livre très apprécié sur la conception du compilateur

Lire Démarrer avec le compilateur-construction en ligne: https://riptutorial.com/fr/compiler-construction/topic/6845/demarrer-avec-le-compilateur-construction

Chapitre 2: Bases de la construction du compilateur

Introduction

Ce sujet contiendra toutes les bases de la construction du compilateur que vous devrez connaître pour que vous puissiez commencer à créer votre propre compilateur. Cette rubrique de documentation contiendra les `2` premières des `4` sections des constructions du compilateur et le reste sera dans un autre sujet.

Les sujets qui seront couverts sont:

Analyse lexicale

Analyse

Syntaxe

- **Analyse lexicale Le** texte source est converti en types et valeurs.
- **L'analyse** des jetons source est convertie en un arbre de syntaxe abstraite (AST).

Examples

Analyseur Lexique Simple

> Dans cet exemple, je vais vous montrer comment créer un lexer de base qui créera les jetons pour une déclaration de variable entière en `python` .

Que fait l'analyseur lexical?

Lexer (analyseur lexical) a pour but de scanner le code source et de diviser chaque mot en un élément de liste. Une fois cela fait, il faut ces mots et crée une paire de type et de valeur qui ressemble à ceci `['INTEGER', '178']` pour former un jeton.

Ces jetons sont créés afin d'identifier la syntaxe de votre langue. Le but du lexer est donc de créer la syntaxe de votre langage, car tout dépend de la manière dont vous voulez identifier et interpréter les différents éléments.

Exemple de code source pour ce lexer:

```
int result = 100;
```

Code pour lexer en `python` :

```
import re                                  # for performing regex expressions

tokens = []                                # for string tokens
source_code = 'int result = 100;'.split() # turning source code into list of words

# Loop through each source code word
for word in source_code:

    # This will check if a token has datatype decleration
    if word in ['str', 'int', 'bool']:
        tokens.append(['DATATYPE', word])

    # This will look for an identifier which would be just a word
    elif re.match("[a-z]", word) or re.match("[A-Z]", word):
        tokens.append(['IDENTIFIER', word])

    # This will look for an operator
    elif word in '*-/+%=':
        tokens.append(['OPERATOR', word])

    # This will look for integer items and cast them as a number
    elif re.match(".[0-9]", word):
        if word[len(word) - 1] == ';':
            tokens.append(["INTEGER", word[:-1]])
            tokens.append(['END_STATEMENT', ';'])
        else:
            tokens.append(["INTEGER", word])

print(tokens) # Outputs the token array
```

Lors de l'exécution de cet extrait de code, le résultat doit être le suivant:

```
[['DATATYPE', 'int'], ['IDENTIFIER', 'result'], ['OPERATOR', '='], ['INTEGER', '100'],
['END_STATEMENT', ';']]
```

Comme vous pouvez le voir, tout ce que nous avons fait est de transformer un morceau de code source tel que la déclaration de variable entière en un flux de jetons de type paire de valeurs.

Faisons le décomposer

1. Nous commençons par la bibliothèque regex par import car il sera nécessaire pour vérifier si certains mots correspondent à un certain modèle de regex.

2. Nous créons une liste vide appelée `tokens` . Cela sera utilisé pour stocker tous les jetons que nous créons.

3. Nous divisons notre code source qui est une chaîne en une liste de mots où chaque mot de la chaîne séparée par un espace est un élément de liste. Nous stockons ensuite ceux-ci dans une variable appelée `source_code` .

4. Nous commençons à parcourir notre liste `source_code` mot par mot.

5. Nous effectuons maintenant notre premier contrôle:

```
if word in ['str', 'int', 'bool']:
   tokens.append(['DATATYPE', word])
```

   Ce que nous vérifions ici est un type de données qui nous indiquera quel type sera notre variable.

6. Après cela, nous effectuons plus de contrôles, comme celui ci-dessus, en identifiant chaque mot de notre code source et en créant un jeton pour celui-ci. Ces jetons seront ensuite transmis à l'analyseur pour créer un arbre de syntaxe abstraite (AST).

   Si vous voulez interagir avec ce code et y jouer, voici un lien vers le code dans un compilateur en ligne https://repl.it/J9Hj/latest

Simple Parser

Il s'agit d'un analyseur simple qui analyse un flux de jetons de déclaration de variables entières que nous avons créé dans l'exemple précédent, Simple Lexical Analyzer. Cet analyseur sera également codé en python.

Qu'est-ce qu'un analyseur?

L'analyseur est le processus dans lequel le texte source est converti en arbre de syntaxe abstraite (AST). Il est également chargé d'effectuer une validation sémantique qui élimine les instructions syntaxiquement correctes qui n'ont aucun sens, par exemple du code inaccessible ou des déclarations en double.

Exemple de jetons:

```
[['DATATYPE', 'int'], ['IDENTIFIER', 'result'], ['OPERATOR', '='], ['INTEGER', '100'],
['END_STATEMENT', ';']]
```

Code pour l'analyseur dans 'python3':

```
ast = { 'VariableDecleration': [] }

tokens = [ ['DATATYPE', 'int'], ['IDENTIFIER', 'result'], ['OPERATOR', '='],
           ['INTEGER', '100'], ['END_STATEMENT', ';'] ]

# Loop through the tokens and form ast
for x in range(0, len(tokens)):

    # Create variable for type and value for readability
    token_type  = tokens[x][0]
    token_value = tokens[x][1]
```

```
    # This will check for the end statement which means the end of var decl
    if token_type == 'END_STATEMENT': break

    # This will check for the datatype which should be at the first token
    if x == 0 and token_type == 'DATATYPE':
        ast['VariableDecleration'].append( {'type': token_value} )

    # This will check for the name which should be at the second token
    if x == 1 and token_type == 'IDENTIFIER':
        ast['VariableDecleration'].append( {'name': token_value} )

    # This will check to make sure the equals operator is there
    if x == 2 and token_value == '=': pass

    # This will check for the value which should be at the third token
    if x == 3 and token_type == 'INTEGER' or token_type == 'STRING':
        ast['VariableDecleration'].append( {'value': token_value} )

print(ast)
```

Le morceau de code suivant devrait afficher ceci comme résultat:

```
{'VariableDecleration': [{'type': 'int'}, {'name': 'result'}, {'value': '100'}]}
```

Comme vous pouvez voir tout ce que fait l'analyseur à partir du code source, les jetons trouvent un modèle pour la déclaration de variable (dans ce cas) et créent avec lui un objet qui contient ses propriétés comme `type` , `name` et `value` .

Faisons le décomposer

1. Nous avons créé la variable `ast` qui contiendra l'AST complet.
2. Nous avons créé la variable `token` examples qui contient les jetons créés par notre lexer et qui doivent maintenant être analysés.
3. Ensuite, nous parcourons chaque jeton et effectuons des vérifications pour trouver certains jetons et former notre AST avec eux.
4. Nous créons une variable pour le type et la valeur pour la lisibilité
5. Nous effectuons maintenant des vérifications comme celle-ci:

   ```
   if x == 0 and token_type == 'DATATYPE':
        ast['VariableDecleration'].append( {'type': token_value} )
   ```

   qui recherche un type de données et l'ajoute à l'AST. Nous continuons à le faire pour la valeur et le nom, ce qui se traduira alors par un AST `VariableDecleration` complète.

   Si vous souhaitez interagir avec ce code et jouer avec, voici un lien vers le code dans un compilateur en ligne https://repl.it/J9IT/latest

Lire Bases de la construction du compilateur en ligne: https://riptutorial.com/fr/compiler-construction/topic/10816/bases-de-la-construction-du-compilateur

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec le compilateur-construction	Community, RyanM, TriskalJM
2	Bases de la construction du compilateur	RyanM