Бесплатная электронная книга

УЧУСЬ compiler-construction

Free unaffiliated eBook created from
Stack Overflow contributors.

#compiler-construction

Около

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

глава 1: Начало работы с составлением компилятора

Examples

Начало работы: Введение

Предпосылки

- **У вас есть сильное понимание языка программирования,** такого как Python, C, C ++, Ruby или любой другой язык.
- **Создайте свой любимый редактор кода или IDE** (одним из таких примеров является VSCode)
- **Оставайтесь мотивированными.** Построение компилятора непросто, поэтому продолжайте нажимать; это стоит усилий.

Категории языков

При создании компилятора вам нужно решить, какой из двух типов языка будет компилятором.

- **Язык игрушек:** это когда вы создаете язык программирования, который не фиксирует проблему, но предназначен для обучения. Забавные примеры из них `Whitespace` , `Lolcode` и `Brainfuck` .
- **Язык программирования:** это языки, которые направлены на решение проблемы или приносят что-то новое и уникальное в таблицу. Их можно сравнить с такими языками, как `Swift` , `C++` и `Python` .

Ресурсы

Во время вашего путешествия неизбежно, что вы наткнетесь на то, о чем не знаете, но, надеюсь, один из этих ресурсов поможет вам.

- **Создайте свой собственный язык программирования** (Ebook)
  - + Дружелюбный к новичкам
  - + Short
  - + `Coffeescript` создать `Coffeescript` и `Rubby`
- **Составители: принципы, методы и инструменты (Книга Дракона)**

    - Содержит все, что вы когда-либо хотели бы знать о компиляторе, но оно продвинуто и долго читается
- **Современный дизайн компилятора** (Ebook)
    - Это еще одна высоко оцененная книга о дизайне компилятора

Прочитайте Начало работы с составлением компилятора онлайн: https://riptutorial.com/ru/compiler-construction/topic/6845/начало-работы-с-составлением-компилятора

глава 2: Основы построения компилятора

Вступление

В этом разделе будут содержаться все основы построения компилятора, которые вам нужно знать, чтобы вы могли приступить к созданию собственного компилятора. Этот раздел документации будет содержать первые `2` из `4` разделов в конструкциях компилятора, а остальные будут в другой теме.

Темы, которые будут рассмотрены:

Лексический анализ

анализ

Синтаксис

- **Лексический анализ** исходный текст преобразуется в токены типа и значения.
- **Разбор** исходных токенов преобразуется в абстрактное синтаксическое дерево (AST).

Examples

Простой лексический анализатор

> В этом примере я покажу вам, как сделать базовый лексер, который будет создавать токены для объявления целочисленной переменной в `python` .

Что делает лексический анализатор?

Цель лексера (лексический анализатор) - сканировать исходный код и разбивать каждое слово на элемент списка. После этого он принимает эти слова и создает пару типа и значения, которая выглядит так `['INTEGER', '178']` чтобы сформировать токен.

Эти маркеры создаются для того, чтобы идентифицировать синтаксис для вашего языка, поэтому вся лексер должен создать синтаксис вашего языка, поскольку все зависит от того, как вы хотите идентифицировать и интерпретировать разные элементы.

Пример исходного кода для этого лексера:

```
int result = 100;
```

Код для lexer в `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
```

При выполнении этого фрагмента кода вывод должен быть следующим:

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

Как вы можете видеть, все, что мы сделали, это превратить часть исходного кода, например объявление целочисленной переменной в токен токенов типа и знака.

Давайте сломаем его

1. Мы начинаем с библиотеки импорта регулярных выражений, потому что это будет необходимо при проверке соответствия определенных слов определенному шаблону регулярного выражения.

2. Мы создаем пустой список, называемый `tokens` . Это будет использоваться для хранения всех токенов, которые мы создаем.

3. Мы разделили наш исходный код, который представляет собой строку, в список слов, где каждое слово в строке, разделенное пробелом, является элементом списка. Затем мы сохраняем их в переменной `source_code` .

4. Мы начинаем циклически перебирать список `source_code` по слову.

5. Теперь мы выполним первую проверку:

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

   Здесь мы проверяем тип данных, который укажет нам, какой тип нашей переменной будет.

6. После этого мы выполняем больше проверок, как указано выше, определяя каждое слово в нашем исходном коде и создавая для него токен. Затем эти токены передаются в парсер для создания абстрактного дерева синтаксиса (AST).

   Если вы хотите взаимодействовать с этим кодом и играть с ним, вот ссылка на код в онлайн-компиляторе https://repl.it/J9Hj/latest

Простой парсер

Это простой синтаксический анализатор, который будет анализировать поток токенов целочисленной переменной, который мы создали в предыдущем примере Simple Lexical Analyzer. Этот анализатор также будет закодирован в python.

Что такое парсер?

Парсер - это процесс, в котором исходный текст преобразуется в абстрактное синтаксическое дерево (AST). Он также отвечает за выполнение семантической проверки, которая отбирает синтаксически правильные утверждения, которые не имеют никакого смысла, например недостижимый код или дубликаты деклараций.

Пример жетонов:

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

Код для парсера в '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)
```

Следующий фрагмент кода должен выводить это в результате:

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

Поскольку вы можете видеть все, что делает синтаксический анализатор, из токенов исходного кода находит шаблон для объявления переменной (в данном случае) и создает с ним объект, который сохраняет свои свойства, такие как `type` , `name` и `value` .

Давайте сломаем его

1. Мы создали переменную `ast` которая будет содержать полный AST.
2. Мы создали переменную `token` примеров, которая содержит токены, которые были созданы нашим лексером, который теперь нуждается в анализе.
3. Затем мы прокручиваем каждый токен и выполняем некоторые проверки, чтобы найти определенные токены и сформировать наш ACT с ними.
4. Мы создаем переменную для типа и значения для удобочитаемости
5. Теперь мы выполняем проверки, подобные этому:

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

который ищет тип данных и добавляет его в AST. Мы продолжаем делать это для значения и имени, которое затем приведет к полной `VariableDecleration` ASTM.

Если вы хотите взаимодействовать с этим кодом и играть с ним, вот ссылка на код в онлайн-компиляторе https://repl.it/J9IT/latest

Прочитайте Основы построения компилятора онлайн: https://riptutorial.com/ru/compiler-construction/topic/10816/основы-построения-компилятора

кредиты

S. No	Главы	Contributors
1	Начало работы с составлением компилятора	Community, RyanM, TriskalJM
2	Основы построения компилятора	RyanM