Kostenloses eBook

LERNEN

compiler-construction

Free unaffiliated eBook created from
Stack Overflow contributors.

#compiler-construction

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit der Compilerbauweise

Examples

Erste Schritte: Einführung

Voraussetzungen

- **Beherrschen Sie eine Programmiersprache** wie Python, C, C ++, Ruby oder eine der anderen Sprachen.
- Lassen **Sie Ihren bevorzugten Code-Editor oder Ihre bevorzugte IDE** (z. B. VSCode) **installieren.**
- **Bleib motiviert.** Einen Compiler zu konstruieren ist nicht einfach, also schieben Sie weiter; es ist die Mühe wert.

Sprachkategorien

Wenn Sie einen Compiler erstellen, müssen Sie entscheiden, welche der beiden Sprachtypen der Compiler sein wird.

- **Spielzeugsprache:** Dies ist, wenn Sie eine Programmiersprache erstellen, die ein Problem nicht behebt, sondern zum Lernen dient. Beispiele dafür sind `Whitespace` , `Lolcode` und `Brainfuck` .
- **Programmiersprache:** Dies sind die Sprachen, die darauf abzielen, ein Problem zu lösen oder etwas Neues und Einzigartiges an den Tisch zu bringen. Diese können mit Sprachen wie `Swift` , `C++` und `Python` verglichen werden.

Ressourcen

Während Ihrer Reise ist es unvermeidlich, dass Sie über etwas stolpern, von dem Sie keine Ahnung haben, aber hoffentlich hilft Ihnen eine dieser Ressourcen.

- **Erstellen Sie Ihre eigene Programmiersprache** (Ebook)
  - + Freundlich für Anfänger
  - + Kurz
  - + Unterstützung bei der Erstellung von `Coffeescript` und `Rubby`
- **Compiler: Prinzipien, Techniken und Werkzeuge (The Dragon Book)**
  - Enthält alles, was Sie jemals über einen Compiler wissen möchten, aber es ist fortgeschritten und langwierig
- **Modernes Compiler-Design** (Ebook)

- Dies ist ein weiteres hochgelobtes Buch über das Compiler-Design

Erste Schritte mit der Compilerbauweise online lesen: https://riptutorial.com/de/compiler-construction/topic/6845/erste-schritte-mit-der-compilerbauweise

Kapitel 2: Grundlagen des Compilerbaus

Einführung

Dieses Thema enthält alle Grundlagen der Compiler-Erstellung, die Sie kennen müssen, damit Sie einen eigenen Compiler erstellen können. Dieses Dokumentationsthema enthält die ersten `2` von `4` Abschnitten in Compilerkonstruktionen. Der Rest wird in einem anderen Thema behandelt.

Folgende Themen werden behandelt:

Lexikalische Analyse

Parsing

Syntax

- **Lexikalische Analyse** Der Quelltext wird in Typ- und Wert-Token konvertiert.
- **Das Analysieren** der Quell-Token wird in einen abstrakten Syntaxbaum (AST) umgewandelt.

Examples

Einfacher Lexical Analyzer

> In diesem Beispiel zeige ich Ihnen, wie Sie einen Basis-Lexer erstellen, der die Token für eine Integer-Variablendeklaration in `python` .

Was macht der lexikalische Analysator?

Der Zweck eines Lexers (lexikalischer Analysator) besteht darin, den Quellcode zu scannen und jedes Wort in ein Listenelement aufzuteilen. Sobald dies erledigt ist, werden diese Wörter verwendet und ein Typ- und Wertepaar erstellt, das folgendermaßen aussieht `['INTEGER', '178']`, um ein Token zu bilden.

Diese Token werden erstellt, um die Syntax für Ihre Sprache zu ermitteln. Der Lexer muss also die Syntax Ihrer Sprache erstellen, da alles davon abhängt, wie Sie verschiedene Elemente identifizieren und interpretieren möchten.

Beispiel-Quellcode für diesen Lexer:

```
int result = 100;
```

Code für Lexer in `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
```

Bei der Ausführung dieses Code-Snippets sollte die Ausgabe Folgendes sein:

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

Wie Sie sehen, können Sie einen Quellcode wie die Deklaration der Ganzzahlvariablen in einen Token-Stream von Typ- und Wertpaartoken umwandeln.

Lass es uns brechen

1. Wir beginnen mit dem Import von Regex-Bibliotheken, da dies erforderlich ist, wenn geprüft wird, ob bestimmte Wörter einem bestimmten Regex-Muster entsprechen.

2. Wir erstellen eine leere Liste namens `tokens` . Hiermit werden alle von uns erstellten Token gespeichert.

3. Wir teilen unseren Quellcode, der eine Zeichenfolge ist, in eine Liste von Wörtern auf, wobei jedes durch Leerzeichen getrennte Wort in der Zeichenfolge ein Listenelement ist. Wir speichern diese dann in einer Variablen namens `source_code` .

4. Wir fangen an, unsere `source_code` Liste Wort für Wort `source_code` .

5. Wir führen jetzt unsere erste Prüfung durch:

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

   Wir überprüfen hier einen Datentyp, der uns sagt, welcher Typ unsere Variable sein wird.

6. Danach führen wir weitere Prüfungen durch, wie z. B. die oben beschriebene, die jedes Wort in unserem Quellcode identifiziert und ein Token dafür erstellt. Diese Token werden dann an den Parser übergeben, um einen Abstract Syntax Tree (AST) zu erstellen.

   Wenn Sie mit diesem Code interagieren und damit spielen möchten, finden Sie hier einen Link zum Code in einem Online-Compiler https://repl.it/J9Hj/latest

Einfacher Parser

Dies ist ein einfacher Parser, der einen ganzzahligen Deklarationstoken-Datenstrom analysiert, den wir im vorherigen Beispiel des Simple Lexical Analyzer erstellt haben. Dieser Parser wird auch in Python codiert.

Was ist ein Parser?

Der Parser ist der Prozess, in dem der Quelltext in einen abstrakten Syntaxbaum (AST) umgewandelt wird. Es ist auch für die Durchführung einer semantischen Validierung zuständig, die aussagekräftige syntaktisch korrekte Anweisungen aussortiert, z. B. nicht erreichbaren Code oder doppelte Deklarationen.

Beispielmarker:

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

Code für Parser in '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)
```

Der folgende Code sollte dies als Ergebnis ausgeben:

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

Wie Sie sehen können, findet der Parser nur in den Quelltext-Tokens ein Muster für die Variablendeklaration (in diesem Fall) und erstellt damit ein Objekt, das seine Eigenschaften wie `type` , `name` und `value` .

Lass es uns brechen

1. Wir haben die `ast` Variable erstellt, die den vollständigen AST enthält.
2. Wir haben die Beispiel- `token` Variable erstellt, die die Token enthält, die von unserem Lexer erstellt wurden und nun analysiert werden müssen.
3. Als Nächstes durchlaufen wir jedes Token und führen einige Überprüfungen durch, um bestimmte Token zu finden und mit ihnen unser AST zu bilden.
4. Wir erstellen eine Variable für Typ und Wert für die Lesbarkeit
5. Wir führen jetzt Prüfungen wie diese durch:

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

   die nach einem Datentyp sucht und ihn dem AST hinzufügt. Wir machen dies weiterhin für den Wert und den Namen, die dann zu einer vollständigen `VariableDecleration` AST führen.

   Wenn Sie mit diesem Code interagieren und damit spielen möchten, finden Sie hier einen Link zum Code in einem Online-Compiler https://repl.it/J9IT/latest

Grundlagen des Compilerbaus online lesen: https://riptutorial.com/de/compiler-construction/topic/10816/grundlagen-des-compilerbaus

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit der Compilerbauweise	Community, RyanM, TriskalJM
2	Grundlagen des Compilerbaus	RyanM