FREE eBook

LEARNING compiler-construction

Free unaffiliated eBook created from
Stack Overflow contributors.

#compiler-construction

Table of Contents

About

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

Chapter 1: Getting started with compiler-construction

Examples

Getting Started: Introduction

Prerequisites

- **Have a strong grasp of a programming language** such as Python, C, C++, Ruby, or any of the other languages out there.
- **Have your favorite code editor or IDE installed** (one such example is VSCode)
- **Stay motivated.** Constructing a compiler is not easy, so keep pushing; it's worth the effort.

Language Categories

When making a compiler, you need to decide which of 2 types of language the compiler will be.

- **Toy language:** This is when you make a programming language which doesn't fix an issue, but is for learning. Fun examples of these are `Whitespace`, `Lolcode`, and `Brainfuck`.

- **Programming language:** These are the languages which aim to solve a problem or bring something new and unique to the table. These can be compared to languages like `Swift`, `C++`, and `Python`.

Resources

During your journey, it is inevitable that you will stumble over something which you have no idea about, but hopefully, one of these resources will aid you.

- **Create Your Own Programming Language** (Ebook)
  - +Friendly to beginners
  - +Short
  - +Aided the creation of `Coffeescript` and `Rubby`
- **Compilers: Principles, Techniques, and Tools (The Dragon Book)**
  - Contains everything you'd ever want to know about a compiler, but it's advanced and a long read
- **Modern Compiler Design** (Ebook)
  - This is another highly praised book on compiler design

Read Getting started with compiler-construction online: https://riptutorial.com/compiler-construction/topic/6845/getting-started-with-compiler-construction

Chapter 2: Basics of Compiler Construction

Introduction

This topic will contain all the basics in compiler construction that you will need to know so that you can get started in making your own compiler. This documentation topic will contain the first `2` out of `4` sections in compiler constructions and the rest will be in a different topic.

The topics which will be covered are:

Lexical Analysis

Parsing

Syntax

- **Lexical Analysis** the source text is converted to type and value tokens.
- **Parsing** the source tokens are converted to an abstract syntax tree (AST).

Examples

Simple Lexical Analyser

> In this example I will show you how to make a basic lexer which will create the tokens for a integer variable declaration in `python`.

What does the lexical analyser do?

The purpose of a lexer (lexical analyser) is to scan the source code and break up each word into a list item. Once done it takes these words and creates a type and value pair which looks like this `['INTEGER', '178']` to form a token.

These tokens are created in order to identify the syntax for your language so the whole point of the lexer is to create the syntax of your language as it all depends on how you want to identify and interpret different items.

Example source code for this lexer:

```
int result = 100;
```

Code for 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
```

When running this code snippet the output should be the following:

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

As you can see all we did is turn a piece of source code such as the integer variable declaration into a token stream of type and value pair tokens.

Let's break it down

1. We begin of by import regex library because it will be needed when checking if certain words match a certain regex pattern.
2. We create an empty list called `tokens`. This will be used to store all of the tokens we create.
3. We split our source code which is a string into a list of words where every word in the string separated by a space is a list item. We then store those in a variable called `source_code`.
4. We start looping through our `source_code` list word by word.
5. We now perform our first check:

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

What we check for here is a datatype which will tell us what type our variable will be.

6. After that we perform more checks like the one above identifying each word in our source code and creating a token for it. These tokens will then be passed on to the parser to create an Abstract Syntax Tree (AST).

   If you want to interact with this code and play with it here is a link to the code in an online compiler https://repl.it/J9Hj/latest

Simple Parser

This is a simple parser which will parse an integer variable declaration token stream which we created in the previous example Simple Lexical Analyser. This parser will also be coded in python.

What is a parser?

The parser is the process in which the source text is converted to an abstract syntax tree (AST). It is also in charge of performing semantical validation which is weeding out syntactically correct statements that make no sense, e.g. unreachable code or duplicate declarations.

Example tokens:

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

Code for 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)
```

The following piece of code should output this as a result:

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

As you can see all that the parser does is from the source code tokens finds a pattern for the variable declaration (in this case) and creates an object with it which holds its properties like `type`, `name` and `value`.

Let's break it down

1. We created the `ast` variable which will hold the complete AST.
2. We created the examples `token` variable which holds the tokens that were created by our lexer which now needs to be parsed.
3. Next, we loop through each token and perform some checks to find certain tokens and form our AST with them.
4. We create variable for type and value for readability
5. We now perform checks like this one:

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

   which looks for a datatype and adds it to the AST. We keep doing this for the value and name which will then result in a full `VariableDecleration` AST.

   If you want to interact with this code and play with it here is a link to the code in an online compiler https://repl.it/J9IT/latest

Read Basics of Compiler Construction online: https://riptutorial.com/compiler-construction/topic/10816/basics-of-compiler-construction

Credits

S. No	Chapters	Contributors
1	Getting started with compiler-construction	Community, RyanM, TriskalJM
2	Basics of Compiler Construction	RyanM