EBook Gratis

APRENDIZAJE compiler-construction

Free unaffiliated eBook created from
Stack Overflow contributors.

#compiler-construction

Tabla de contenido

Acerca de 1

Capítulo 1: Comenzando con la compilación de construcción 2

Examples 2

Comenzando: Introducción 2

Prerrequisitos 2

Categorías de idiomas 2

Recursos 2

Capítulo 2: Conceptos básicos de la construcción del compilador 4

Introducción 4

Sintaxis 4

Examples 4

Analizador léxico simple 4

¿Qué hace el analizador léxico? 4

Vamos a descomponerlo 5

Analizador simple 6

¿Qué es un analizador? 6

Vamos a descomponerlo 7

Creditos 9

Acerca de

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

Capítulo 1: Comenzando con la compilación de construcción

Examples

Comenzando: Introducción

Prerrequisitos

- **Tenga un conocimiento sólido de un lenguaje de programación** como Python, C, C ++, Ruby o cualquiera de los otros lenguajes que existen.
- **Tener su editor de código favorito o IDE instalado** (uno de estos ejemplos es VSCode)
- **Mantenerse motivado.** Construir un compilador no es fácil, así que sigue empujando; Vale la pena el esfuerzo.

Categorías de idiomas

Al crear un compilador, debe decidir cuál de los dos tipos de lenguaje será el compilador.

- **Lenguaje de juguete:** esto es cuando creas un lenguaje de programación que no soluciona un problema, pero es para aprender. Ejemplos divertidos de estos son `Whitespace` , `Lolcode` y `Brainfuck` .

- **Lenguaje de programación:** estos son los lenguajes que intentan resolver un problema o aportar algo nuevo y único a la tabla. Estos pueden compararse con lenguajes como `Swift` , `C++` y `Python` .

Recursos

Durante su viaje, es inevitable que tropiece con algo de lo que no tiene idea, pero con suerte, uno de estos recursos lo ayudará.

- **Crea tu propio lenguaje de programación** (Ebook)
  - + Amigable con los principiantes
  - + Corto
  - + Ayudó en la creación de `Coffeescript` y `Rubby`
- **Compiladores: Principios, Técnicas y Herramientas (El Libro del Dragón)**
  - Contiene todo lo que siempre querría saber sobre un compilador, pero es avanzado y una lectura larga
- **Diseño Compilador Moderno** (Ebook)
  - Este es otro libro altamente elogiado en el diseño del compilador

Lea Comenzando con la compilación de construcción en línea: https://riptutorial.com/es/compiler-construction/topic/6845/comenzando-con-la-compilacion-de-construccion

Capítulo 2: Conceptos básicos de la construcción del compilador

Introducción

Este tema contendrá todos los elementos básicos de la construcción del compilador que deberá conocer para poder comenzar a crear su propio compilador. Este tema de documentación contendrá las primeras `2` de `4` secciones en las construcciones del compilador y el resto estará en un tema diferente.

Los temas que se tratarán son:

Análisis léxico

Análisis

Sintaxis

- **En el análisis léxico,** el texto de origen se convierte en tokens de tipo y valor.
- **El análisis de** los tokens de origen se convierte en un árbol de sintaxis abstracta (AST).

Examples

Analizador léxico simple

> En este ejemplo, te mostraré cómo hacer un lexer básico que creará los tokens para una declaración de variable entera en `python` .

¿Qué hace el analizador léxico?

El propósito de un lexer (analizador léxico) es escanear el código fuente y dividir cada palabra en un elemento de la lista. Una vez hecho esto, toma estas palabras y crea un par de tipo y valor que se parece a esto `['INTEGER', '178']` para formar un token.

Estos tokens se crean para identificar la sintaxis de su idioma, por lo que todo el objetivo del lexer es crear la sintaxis de su idioma, ya que todo depende de cómo desee identificar e interpretar los diferentes elementos.

Código fuente de ejemplo para este lexer:

```
int result = 100;
```

Código para 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
```

Cuando se ejecuta este fragmento de código, la salida debe ser la siguiente:

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

Como puede ver, todo lo que hicimos fue convertir un fragmento de código fuente, como la declaración de variable de entero, en una secuencia de tokens de tipo y valor.

Vamos a descomponerlo

1. Comenzamos con la importación de la biblioteca de expresiones regulares porque será necesario cuando se compruebe si ciertas palabras coinciden con un determinado patrón de expresiones regulares.

2. Creamos una lista vacía llamada `tokens` . Esto se utilizará para almacenar todos los tokens que creamos.

3. Dividimos nuestro código fuente, que es una cadena en una lista de palabras donde cada palabra en la cadena separada por un espacio es un elemento de la lista. Luego los almacenamos en una variable llamada `source_code` .

4. Comenzamos a recorrer nuestra lista de `source_code` palabra por palabra.

5. Ahora realizamos nuestro primer control:

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

   Lo que buscamos aquí es un tipo de datos que nos dirá qué tipo de variable será nuestra variable.

6. Después de eso, realizamos más verificaciones como la anterior, identificando cada palabra en nuestro código fuente y creando un token para ella. Estos tokens se pasarán al analizador para crear un árbol de sintaxis abstracta (AST).

   Si desea interactuar con este código y jugar con él, aquí hay un enlace al código en un compilador en línea https://repl.it/J9Hj/latest

Analizador simple

Este es un analizador simple que analizará una secuencia de token de declaración de variable de entero que creamos en el ejemplo anterior Simple Lexical Analyzer. Este analizador también se codificará en python.

¿Qué es un analizador?

El analizador es el proceso en el que el texto de origen se convierte en un árbol de sintaxis abstracta (AST). También se encarga de realizar la validación semántica, que está eliminando las afirmaciones sintácticamente correctas que no tienen sentido, por ejemplo, código inalcanzable o declaraciones duplicadas.

Fichas de ejemplo:

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

Código para el analizador en '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)
```

El siguiente fragmento de código debería generar esto como resultado:

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

Como puede ver, todo lo que hace el analizador es desde el código fuente. Los tokens encuentran un patrón para la declaración de variable (en este caso) y crean un objeto con él que contiene sus propiedades como `type` , `name` y `value` .

Vamos a descomponerlo

1. Creamos la variable `ast` que contendrá el AST completo.
2. Creamos la variable de `token` ejemplos que contiene los tokens creados por nuestro lexer que ahora necesita ser analizado.
3. A continuación, repasamos cada token y realizamos algunas comprobaciones para encontrar ciertos tokens y formar nuestro AST con ellos.
4. Creamos variable por tipo y valor por legibilidad.
5. Ahora realizamos chequeos como este:

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

   que busca un tipo de datos y lo agrega al AST. Continuamos haciendo esto para el valor y el nombre que luego resultarán en una `VariableDecleration` AST completa.

   Si desea interactuar con este código y jugar con él, aquí hay un enlace al código en un compilador en línea https://repl.it/J9IT/latest

Lea Conceptos básicos de la construcción del compilador en línea: https://riptutorial.com/es/compiler-construction/topic/10816/conceptos-basicos-de-la-construccion-del-compilador

Creditos

S. No	Capítulos	Contributors
1	Comenzando con la compilación de construcción	Community, RyanM, TriskalJM
2	Conceptos básicos de la construcción del compilador	RyanM