EBook Gratuito

APPRENDIMENTO compiler-construction

Free unaffiliated eBook created from
Stack Overflow contributors.

#compiler-construction

Sommario

Di

Capitolo 1: Iniziare con la compilazione del compilatore

Examples

Per iniziare: Introduzione

Prerequisiti

- **Avere una conoscenza approfondita di un linguaggio di programmazione** come Python, C, C ++, Ruby o una qualsiasi delle altre lingue là fuori.
- **Avere il proprio editor di codice preferito o IDE installato** (uno di questi esempi è VSCode)
- **Rimani motivato.** Costruire un compilatore non è facile, quindi continua a spingere; ne vale la pena.

Categorie linguistiche

Quando si crea un compilatore, è necessario decidere quale dei 2 tipi di linguaggio sarà il compilatore.

- **Linguaggio giocattolo:** questo è quando crei un linguaggio di programmazione che non risolve un problema, ma è per l'apprendimento. Esempi divertenti di questi sono `Whitespace` , `Lolcode` e `Brainfuck` .
- **Linguaggio di programmazione:** sono le lingue che mirano a risolvere un problema o portano qualcosa di nuovo e unico al tavolo. Questi possono essere paragonati a linguaggi come `Swift` , `C++` e `Python` .

risorse

Durante il tuo viaggio, è inevitabile che ti imbatti in qualcosa di cui non hai idea, ma spero che una di queste risorse ti aiuti.

- **Crea il tuo linguaggio di programmazione** (Ebook)
  - + Cordiale per i principianti
  - + breve
  - + Aiutato la creazione di `Coffeescript` e `Rubby`
- **Compilatori: principi, tecniche e strumenti (il libro del drago)**
  - Contiene tutto ciò che vorresti sapere su un compilatore, ma è avanzato e una lettura lunga
- **Modern Compiler Design** (Ebook)

- Questo è un altro libro molto apprezzato sul design del compilatore

Leggi Iniziare con la compilazione del compilatore online: https://riptutorial.com/it/compiler-construction/topic/6845/iniziare-con-la-compilazione-del-compilatore

Capitolo 2: Nozioni di base sulla costruzione del compilatore

introduzione

Questo argomento conterrà tutte le nozioni di base sulla costruzione del compilatore che dovrai conoscere per poter iniziare a creare il tuo compilatore. Questo argomento di documentazione conterrà le prime `2` sezioni su `4` nelle costruzioni del compilatore e il resto sarà in un argomento diverso.

Gli argomenti che saranno trattati sono:

Analisi lessicale

parsing

Sintassi

- **Analisi lessicale** il testo sorgente viene convertito in tipo e valore token.
- **L'analisi** dei token di origine viene convertita in un albero di sintassi astratto (AST).

Examples

Semplice analizzatore lessicale

> In questo esempio ti mostrerò come creare un lexer di base che creerà i token per una dichiarazione di variabile intera in `python` .

Cosa fa l'analizzatore lessicale?

Lo scopo di un lexer (analizzatore lessicale) è di scansionare il codice sorgente e suddividere ogni parola in una voce di elenco. Una volta fatto, prende queste parole e crea una coppia di tipo e valore che assomiglia a questo `['INTEGER', '178']` per formare un token.

Questi token vengono creati per identificare la sintassi della tua lingua, quindi l'intero punto del lexer è creare la sintassi della tua lingua poiché tutto dipende da come vuoi identificare e interpretare elementi diversi.

Esempio di codice sorgente per questo lexer:

```
int result = 100;
```

Codice per 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
```

Quando si esegue questo snippet di codice, l'output dovrebbe essere il seguente:

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

Come puoi vedere, tutto ciò che abbiamo fatto è trasformare un pezzo di codice sorgente come la dichiarazione della variabile intera in un flusso di token di token di tipo e valore coppia.

Scopriamolo

1. Iniziamo con la libreria di regex di importazione perché sarà necessaria per verificare se certe parole corrispondono a un certo modello di espressioni regolari.

2. Creiamo una lista vuota chiamata `tokens` . Questo sarà usato per memorizzare tutti i token che creiamo.

3. Abbiamo diviso il nostro codice sorgente che è una stringa in un elenco di parole in cui ogni parola nella stringa separata da uno spazio è una voce di elenco. Quindi li memorizziamo in una variabile chiamata `source_code` .

4. Iniziamo a scorrere la nostra lista `source_code` parola per parola.

5. Ora eseguiamo il nostro primo controllo:

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

   Quello che controlliamo qui è un tipo di dati che ci dirà che tipo sarà la nostra variabile.

6. Dopodiché eseguiremo altri controlli come quello sopra, identificando ogni parola nel nostro codice sorgente e creando un token per esso. Questi token verranno quindi passati al parser per creare un Abstract Syntax Tree (AST).

   Se vuoi interagire con questo codice e giocarci, ecco un link al codice in un compilatore online https://repl.it/J9Hj/latest

Parser semplice

Questo è un parser semplice che analizzerà un flusso di token di dichiarazione di variabili intere che abbiamo creato nell'esempio precedente Simple Lexical Analyzer. Questo parser sarà anche codificato in python.

Cos'è un parser?

Il parser è il processo in cui il testo di origine viene convertito in un albero di sintassi astratto (AST). È inoltre incaricato di eseguire la convalida semantica che elimina le affermazioni sintatticamente corrette che non hanno senso, ad esempio codice irraggiungibile o dichiarazioni duplicate.

Token di esempio:

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

Codice per 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)
```

Di conseguenza, la seguente parte di codice dovrebbe produrre questo risultato:

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

Come puoi vedere tutto ciò che fa il parser dai token del codice sorgente trova un pattern per la dichiarazione di variabile (in questo caso) e crea un oggetto con esso che mantiene le sue proprietà come `type` , `name` e `value` .

Scopriamolo

1. Abbiamo creato la variabile `ast` che manterrà l'AST completo.
2. Abbiamo creato la variabile `token` esempio che contiene i token creati dal nostro lexer che ora deve essere analizzato.
3. Successivamente, passiamo in rassegna ogni token ed eseguiamo alcuni controlli per trovare determinati token e formare il nostro AST con loro.
4. Creiamo variabili per tipo e valore per la leggibilità
5. Ora eseguiamo assegni come questo:

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

   che cerca un tipo di dati e lo aggiunge all'AST. Continuiamo a farlo per il valore e il nome che quindi genereranno un AST `VariableDecleration` completo.

   Se vuoi interagire con questo codice e giocarci, ecco un link al codice in un compilatore online https://repl.it/J9IT/latest

Leggi Nozioni di base sulla costruzione del compilatore online: https://riptutorial.com/it/compiler-construction/topic/10816/nozioni-di-base-sulla-costruzione-del-compilatore

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con la compilazione del compilatore	Community, RyanM, TriskalJM
2	Nozioni di base sulla costruzione del compilatore	RyanM