

LAPORAN
TUGAS BESAR 2
IF 2124 TEORI BAHASA FORMAL DAN OTOMATA
“*COMPILER* BAHASA PYTHON”



Disusun oleh:

- | | |
|-------------------------|----------|
| - Jovan Karuna Cahyadi | 13518024 |
| - Jonathan Yudi Gunawan | 13518084 |
| - William | 13518138 |

Prodi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2019

DAFTAR ISI

DAFTAR ISI	2
DAFTAR TABEL DAN GAMBAR	4
BAB 1 DESKRIPSI MASALAH	5
1.1 Latar Belakang	5
1.2 Deskripsi Tugas	6
1.3 Input dan Output	7
1.4 Lain-lain	10
BAB 2 TEORI SINGKAT	10
2.1 Python	11
2.2 Context-Free Grammar (CFG)	12
2.2.1 Deskripsi singkat	12
2.2.2 CFG sebagai Dasar Pembentukan Parser	13
2.2.3 Ambiguitas pada CFG	13
2.3 Chomsky Normal Form (CNF)	14
2.4 Cocke–Younger–Kasami (CYK)	13
2.5 SWI-Prolog	15
2.6 Definite Clause Grammar (DCG)	16
BAB 3 ANALISIS PERSOALAN	17
3.1 Membuat CFG dan Mengimplementasikannya pada SWI-Prolog	17
3.2 Melakukan Binerisasi CFG	17
3.3 Mengkonversi CFG menjadi CNF	18
3.4 Mengimplementasikan CYK pada CNF	19
BAB 4 SPESIFIKASI TEKNIS PROGRAM	11
4.1 Struktur Program	20
4.1.1 Struktur Program CFG	20
4.1.2 Struktur Program Antara	20
4.1.3 Struktur Program Konversi CFG ke CNF	20
4.1.4 Struktur Program Implementasi CYK pada CNF	21
4.2 Kode Program	21
4.2.1 Kode Program CFG	21

4.2.2 Kode Program Antara	22
4.2.3 Kode Program Konversi CFG ke CNF	23
4.2.4 Kode Program Implementasi CYK pada CNF	23
4.3 Realisasi Program	23
BAB 5 EKSPERIMEN	24
5.1 Hasil Tangkapan Layar	24
BAB 6 KESIMPULAN, SARAN, DAN REFLEKSI	29
6.1 Hasil yang Dicapai	29
6.2 Saran dan Pengembangan	29
6.3 Refleksi	29
DAFTAR REFERENSI	30
LAMPIRAN	32

DAFTAR TABEL DAN GAMBAR

Daftar Gambar

Gambar 1 Pohon Penurunan String ‘aabbb’	13
Gambar 2 Contoh hasil <i>parse tree</i> dari pengolahan grammar dengan DCG	16
Gambar 3 Kasus Uji 1	24
Gambar 4 Kasus Uji 2	25
Gambar 5 Kasus Uji 3	25
Gambar 6 Kasus Uji 4	26
Gambar 7 Kasus Uji 5	26
Gambar 8 Kasus Uji 6	27

Daftar Tabel

Tabel 1 Daftar Kata Kunci pada Python yang Harus Dimasukkan dalam Grammar	5
Tabel 2 Struktur Program CFG	20
Tabel 3 Kode Program CFG	22
Tabel 4 Kode Program Antara	22
Tabel 5 Kode Program Konversi CFG ke CNF	23
Tabel 6 Kode Program Implementasi CYK pada CNF	23
Tabel 7 Realisasi Program	23
Tabel 8 Draft proses pembuatan CFG	33

BAB I

DESKRIPSI MASALAH

1.1 Latar Belakang

Python adalah bahasa *interpreter* tingkat tinggi (*high-level*), dan juga *general-purpose*. Python diciptakan oleh Guido van Rossum dan dirilis pertama kali pada tahun 1991. Filosofi desain pemrograman Python mengutamakan *code readability* dengan penggunaan *whitespace*-nya. Python adalah bahasa *multi-paradigm* karena mengimplementasi paradigma fungsional, imperatif, berorientasi objek, dan reflektif.

Dalam proses pembuatan program dari sebuah bahasa menjadi instruksi yang dapat dieksekusi oleh mesin, terdapat pemeriksaan sintaks atau kompilasi bahasa yang dibuat oleh programmer. Kompilasi ini bertujuan untuk memastikan instruksi yang dibuat oleh programmer mengikuti aturan yang sudah ditentukan oleh bahasa tersebut. Baik bahasa berjenis *interpreter* maupun *compiler*, keduanya pasti melakukan pemeriksaan sintaks. Perbedaananya terletak pada apa yang dilakukan setelah proses pemeriksaan (kompilasi/*compile*) tersebut selesai dilakukan.

Dibutuhkan *grammar* bahasa dan algoritma *parser* untuk melakukan kompilasi. Sudah sangat banyak *grammar* dan algoritma yang dikembangkan untuk menghasilkan *compiler* dengan performa yang tinggi. Terdapat CFG, CNF^- , CNF^+ , 2NF, 2LF, dll untuk *grammar* yang dapat digunakan, dan terdapat LL(0), LL(1), CYK, Earley's Algorithm, LALR, GLR, Shift-reduce, SLR, LR(1), dll untuk algoritma yang dapat digunakan untuk melakukan *parsing*.

1.2 Deskripsi Tugas

Pada tugas besar ini, implementasikanlah *compiler* untuk **Python** untuk *statement-statement* dan sintaks-sintaks bawaan Python dengan menggunakan algoritma **CYK (Cocke-Younger-Kasami)**. Algoritma CYK harus menggunakan *grammar* CNF (Chomsky Normal Form) sebagai *grammar* masukannya. Oleh karena itu, buatlah **terlebih dahulu *grammar* dalam CFG (Context Free Grammar)**, kemudian konversikan *grammar* CFG tersebut ke *grammar* CNF.

Berikut adalah daftar kata kunci bawaan Python yang harus terdaftar dalam *grammar* (yang dicoret tidak perlu diimplementasi). Rincian mengenai implementasi dan contohnya dapat dilihat pada [pranala ini](#).

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Tabel 1 Daftar Kata Kunci pada Python yang Harus Dimasukkan dalam Grammar

Hal-hal ini tidak perlu kalian masukkan ke dalam *grammar* atau diimplementasikan:

1. Arti semantik dari *input* (mis. walaupun kelas Foo belum pernah didefinisikan, `bar = Foo()` atau `bar.a_method()` diperbolehkan)
2. Arti semantik dari *method* (mis. jumlah parameternya)
3. Regex dalam bentuk apapun, seperti r-string (mis. `r'123'`)
4. F-string (mis. `f'aku suka tbfo'`)
5. *End of statement* dengan menggunakan titik koma (;)

1.3 Input dan Output

1. Teks input dibaca melalui sebuah file eksternal, dengan nama bebas, dijadikan parameter eksekusi program
2. Keluaran berupa “Accepted”, atau “Syntax Error”.
3. (BONUS) Jika ditolak, tampilkan sebuah output (bebas) yang memudahkan kalian untuk mencari penyebab kesalahan. (5)
4. (BONUS) Menerima *type hinting*. (5)

Berikut adalah contoh input-output yang mungkin:

input1.txt (*highlight biru adalah type hinting*)

```
def get_rule_category(rule: dict) -> str:
    ''' Get rule category in string. This category is also a key for its
    corresponding dictionary.

    Input(dict) = rule
    Output(str) = category of the rule

    Example:
    Input      = {'producer': 'N', 'product': ['people']}
    Output     = 'terminal'
    '''
    rule_product = rule[PRODUCT_KEY]
    if len(rule_product) == 0:
        return EPSILON_RULE_KEY
    elif len(rule_product) == 1:
        if rule_product[0].islower():
            return TERMINAL_RULE_KEY
        else:
            return UNARY_RULE_KEY
    elif len(rule_product) == 2:
        return BINARY_RULE_KEY
    else:
        return N_ARIES_RULE_KEY
```

input2.txt

```
from PIL import Image
import pytesseract
import time
import keyboard
import random

IMAGE_INPUT_NAME = 'input.jpg'
FILE_OUTPUT_NAME = 'result.txt'
ACCEPTED_ASCII_LIST = [x for x in range(32, 127)]

def extract_text_from_image():
    PIPE_ASCII = 124

    time_start = time.perf_counter()
    text = pytesseract.image_to_string(Image.open(IMAGE_INPUT_NAME))
    with open(FILE_OUTPUT_NAME, 'w') as f_out:
        for c in text:
            ascii = ord(c)
            if ascii == PIPE_ASCII:
                f_out.write('I')
            elif ascii in ACCEPTED_ASCII_LIST:
                f_out.write(c)
            else:
                f_out.write(' ')
    time_end = time.perf_counter()
    print('\nDone! Executed in: {:.2f} seconds'.format(time_end -
time_start))
```


input3.txt (*highlight kuning adalah lokasi-lokasi kesalahan*)

```
from PIL import asdf Image

-var = 'input.jpg'
ACCEPTED_ASCII_LIST = [x for x in range((32, 127))]

def extract_text_from_image():
    PIPE_ASCII = 124

    time_start = time.perf_counter()
    text = pytesseract.image_to_string(Image.open(IMAGE_INPUT_NAME))
    with open(FILE_OUTPUT_NAME, 'asdf') as f_out:
        for c in text:
            ascii = ord(c)
            if ascii == PIPE_ASCII:
                f_out.write('I')
            elif ascii in ACCEPTED_ASCII_LIST:
                f_out.write[c]
            else ascii == 1:
                f_out.write(' ')
    time_end = time.perf_counter()
    print('\nDone! Executed in: {:.2f} seconds'.format(time_end -
time_start))
```

Contoh 1

main.exe input1.txt

Accepted

Penjelasan:

Terdapat *type hinting* yang adalah bonus

Contoh 2

main.exe input2.txt

Accepted

Contoh 3

main.exe input3.txt

Syntax Error

Penjelasan:

Terdapat kesalahan pada lokasi-lokasi yang telah ditandai.

1.4 Lain-lain

1. Versi Python yang dijadikan referensi adalah **Python 3.7**
2. Dalam pengerjaan tugas ini, belajar dari sumber-sumber lain diperbolehkan, tetapi **persalinan atau kecurangan dalam bentuk apapun sangat dilarang. Pelaku kecurangan akan ditindak tegas.**
3. Bahasa yang digunakan **bebas**.

BAB II

DASAR TEORI

2.1 Python

Python adalah sebuah bahasa pemrograman tingkat tinggi yang mudah untuk dibaca dan diimplementasikan. Python sendiri bersifat *open source* sehingga bebas digunakan untuk berbagai keperluan, baik keperluan komersial maupun pribadi. Python sendiri dapat digunakan di berbagai *platform* seperti Mac, Windows dan Unix dan telah bisa dipakai di bahasa lain seperti Java dan .NET.

Python sendiri banyak dianggap sebagai bahasa *scripting*, seperti Ruby atau Perl dan banyak digunakan untuk membuat aplikasi berjenis Web beserta isinya. Python sendiri mendukung banyak teknik penanganan gambar, baik gambar 2D atau gambar 3D. Python sendiri juga memiliki banyak modul - modul, baik bawaan maupun buatan pihak ketiga.

Berbeda dengan kode program yang ditulis dengan bahasa yang dikompilasi, *script* yang ditulis dalam Python bisa dijalankan secara langsung dengan *interpreter* tanpa perlu adanya kompilasi terhadap program. Tidak hanya itu, blok program di bahasa pemrograman Python menggunakan indentasi *whitespace* (space dan tab).

2.2 Context-Free Grammar (CFG)

2.2.1 Deskripsi Singkat

CFG atau *Context Free Grammar* adalah tata bahasa formal di mana setiap aturan produksi adalah dalam bentuk $A \rightarrow B$ dimana A adalah pemroduksi, dan B adalah hasil produksi. Batasannya hanyalah ruas kiri adalah sebuah simbol variabel. Dan pada ruas kanan bisa berupa terminal, symbol, variabel ataupun ϵ , Contoh aturan produksi yang termasuk CFG adalah seperti berikut ini :

- $X \rightarrow bY \mid Za$
- $Y \rightarrow aY \mid b$
- $Z \rightarrow bZ \mid \epsilon$

CFG adalah tata bahasa yang mempunyai tujuan sama seperti halnya tata bahasa regular yaitu merupakan suatu cara untuk menunjukkan bagaimana menghasilkan suatu untai-untai dalam sebuah bahasa.

CFG perlu disederhanakan dengan tujuan untuk melakukan pembatasan sehingga tidak menghasilkan pohon penurunan yang memiliki kerumitan yang tak perlu atau aturan produksi tak berarti. Berikut merupakan langkah-langkah dalam melakukan penyederhanaan CFG :

- Eliminasi ϵ -production
- Eliminasi unit production
- Eliminasi useless symbol

Latar belakang CFG terinspirasi dari bahasa natural manusia, ilmuwan-ilmuwan ilmu komputer yang mengembangkan bahasa pemrograman turut serta memberikan grammar (pemrograman) secara formal. Grammar ini diciptakan secara bebas-konteks dan disebut *Context Free Grammar (CFG)*. Hasilnya, dengan pendekatan formal ini, kompiler suatu bahasa pemrograman dapat dibuat lebih mudah dan menghindari ambiguitas ketika parsing bahasa tersebut. Contoh desain CFG untuk *parser*, misal : $B \rightarrow BB \mid (B) \mid \epsilon$ untuk mengenali bahasa dengan hanya tanda kurung $\{‘(’, ‘)’\}$ sebagai terminal-nya. Proses parsing adalah proses pembacaan string dalam bahasa sesuai CFG tertentu, proses ini harus mematuhi aturan produksi dalam CFG tersebut.

2.2.2 CFG sebagai Dasar Pembentukan *Parser*

CFG menjadi dasar dalam pembentukan suatu *parser*/proses analisis sintaksis. Bagian sintaks dalam suatu kompilator kebanyakan didefinisikan dalam tata bahasa bebas konteks. Pohon penurunan (*derivation tree*/*parse tree*) berguna untuk menggambarkan simbol-simbol variabel menjadi simbol-simbol terminal setiap simbol variabel akan diturunkan menjadi terminal sampai tidak ada yang belum tergantikan.

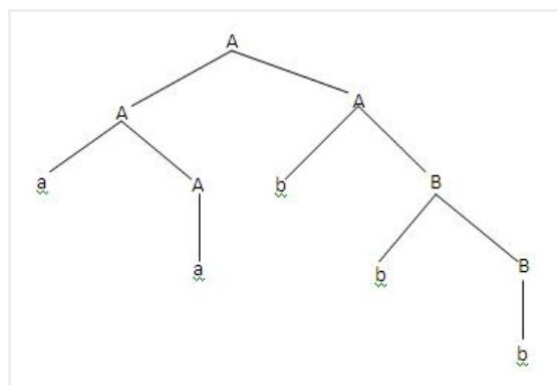
Contoh, terdapat CFG dengan aturan produksi sebagai berikut dengan simbol awal S :

$$S \rightarrow AB$$

$$A \rightarrow aA \mid a$$

$$B \rightarrow bB \mid b$$

Maka jika ingin dicari gambar *pohon penurunan* dengan string : 'aabbb' hasilnya adalah seperti di bawah :



Gambar 1 Pohon Penurunan String 'aabbb'

Proses penurunan / *parsing* bisa dilakukan dengan cara sebagai berikut :

- Penurunan terkiri (*leftmost derivation*): simbol variabel terkiri yang diperluas terlebih dahulu.
- Penurunan ter-kanan (*rightmost derivation*) : simbol variabel terkanan yang diperluas terlebih dahulu.

Grammar sbb :

$$S \rightarrow aAS \mid a$$

$$A \rightarrow SbA \mid ba$$

Untuk memperoleh string 'aabbaa' dari grammar diatas dilakukan dengan cara :

- Penurunan ter kiri: $S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS \Rightarrow aabbaa$
- Penurunan ter-kanan : $S \Rightarrow aAS \Rightarrow aAa \Rightarrow aSbAa \Rightarrow aAbbaa \Rightarrow aabbaa$

2.2.3 Ambiguitas pada CFG

Ambiguitas terjadi bila terdapat lebih dari satu pohon penurunan yang berbeda untuk memperoleh suatu string.

Misalkan terdapat tata bahasa sebagai berikut :

$$S \rightarrow A \mid B$$

$$A \rightarrow a$$

$$B \rightarrow a$$

Untuk memperoleh untai 'a' bisa terdapat dua cara penurunan sebagai berikut :

$$S \Rightarrow A \Rightarrow a$$

$$S \Rightarrow B \Rightarrow a$$

Sebuah string yang mempunyai lebih dari satu pohon sintaks disebut *string ambigu* (*ambiguous*). Grammar yang menghasilkan paling sedikit sebuah string ambigu disebut *grammar ambigu*.

2.3 Chomsky Normal Form (CNF)

Salah satu bentuk dari Context Free Grammar (CFG) adalah Chomsky Normal Form (CNF). Bentuk ini dipakai untuk menulis bentuk CFG karena bentuk ini memudahkan pemahaman dan pembuktian CFG. Selain itu, bentuk ini juga memudahkan penurunan pembuktian suatu string masuk ke dalam suatu CFG atau tidak dengan memakai *binary tree*. Suatu CFG telah berada dalam keadaan Chomsky Normal Form (CNF) jika seluruh aturan produksi telah memenuhi kondisi berikut :

- Suatu non-terminal menghasilkan suatu terminal ($X \rightarrow x$).
- Suatu non-terminal menghasilkan 2 non-terminal ($X \rightarrow YZ$).
- Start symbol menghasilkan ϵ . ($S \rightarrow \epsilon$).

Cara mengkonversi CFG menjadi CNF :

1. Membuat suatu variabel start baru untuk CFG.
2. Eliminasi ϵ dari aturan produksi CFG.

3. Hilangkan seluruh aturan unit pada CFG.
4. Sederhanakan aturan yang memiliki lebih dari 2 (dua) terminal ataupun variabel.

2.4 Cocke-Younger-Kasami (CYK)

Cocke-Younger-Kasami-Algorithm adalah suatu algoritma *parsing* yang efisien untuk context-free grammars. Hal ini membuat algoritma Cocke-Younger-Kasami (CYK) sangat efektif untuk *parsing* CFG yang ada dalam bentuk CNF. Algoritma CYK dapat digunakan untuk mendeteksi apakah suatu kata merupakan bagian dari *context-free grammar* $w \in \Sigma^*$.

Berikut merupakan algoritma CYK yang digunakan untuk mengetahui apakah suatu kata merupakan bagian dari CFG atau tidak :

1. Siapkan tabel segitiga atas sesuai persoalannya (string yang ingin dicocokkan dengan CFG).
2. Untuk baris pertama, lihat ruas kiri pada aturan P untuk persoalannya / string.
3. Kemudian masukkan ke kolom di baris berikutnya aturan produksi yang menyusun string tersebut.
4. Lakukan langkah 3 sampai mencapai kolom terakhir
5. Jika pada kolom terakhir diisi dengan aturan produksi start (S), maka string tersebut merupakan bagian dari CFG yang ingin dicocokkan.

Algoritma pseudocode untuk CYK :

```

begin
  for i:= 1 to n do
     $V_i := \{A \mid A \text{ a aturan produksi dimana simbol ke- } i \text{ adalah } a \}$ ;
    for j:= 2 to n do
      for i:= 1 to (n-j+1) do
        begin
           $V_{ij} := \emptyset$ ;
          for k:=1 to (j - 1) do
             $V_{ij} := V_{ij} \cup \{ A \mid A \text{ BC adalah suatu produksi, dimana B di } V_{ik} \text{ dan C di } V_{i+k,j-k} \}$ 
          end
        end
      end
    end
  end
end

```

2.5 SWI-Prolog

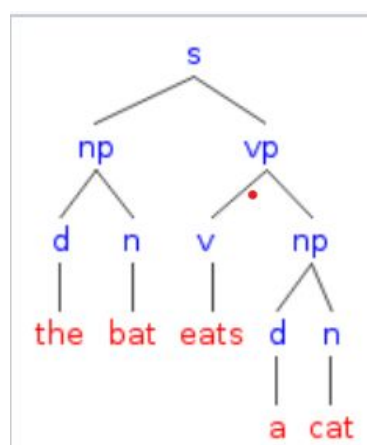
SWI-Prolog merupakan suatu bentuk implementasi gratis dari bahasa pemrograman Prolog. Umumnya bahasa ini digunakan di bidang akademis dalam hal yang berbasis logika

dan juga implementasi terhadap Artificial Intelligence (AI). Meskipun demikian, penggunaan utama dari Swi-Prolog terdapat pada pembuatan aplikasi (*application development*). Seperti namanya, Swi-Prolog berbasis bahasa pemrograman logika Prolog yang banyak digunakan dalam pemrograman-pemrograman berbasis logika. Bahasa Prolog pertama kali diciptakan oleh Alain Colmerauer dan Robert Kowalski sekitar tahun 1972 dalam upaya untuk menciptakan suatu bahasa pemrograman yang memungkinkan pernyataan - pernyataan berbasis logika dapat dimengerti dan dijalankan pada komputer.

2.6 Definite Clause Grammar (DCG)

DCG *Grammar rules* merupakan salah satu bentuk aturan bentuk cara untuk mengungkapkan bentuk bahasa, baik dalam bentuk *natural languages (formal language)* dalam bahasa pemrograman Prolog. Konsep yang dipakai dalam pendefinisian bentuk *grammar* dalam DCG menyerupai konsep - konsep *attribute grammar*. Klausa - klausa *grammar* yang didefinisikan di DCG dapat dianggap sebagai suatu *axiom* validitas dari suatu kalimat, serta apakah kalimat tersebut dapat di dibuat dari aturan - aturan yang didefinisikan di klausa - klausa DCG (memakai *parse tree*).

Salah satu penggunaan utama DCG adalah untuk melakukan parsing (mendeteksi kesalahan sintaks) terhadap suatu kalimat terhadap aturan yang telah ditetapkan untuk kalimat tersebut (*grammar*). Dikarenakan mekanisme parsing yang hampir sama antara DCG dan CFG, maka pola - pola parsing dengan menggunakan CFG dapat juga diterapkan pada DCG.



Gambar 2 Contoh hasil *parse tree* dari pengolahan grammar dengan DCG

BAB III

ANALISIS PERSOALAN

3.1 Membuat CFG dan Mengimplementasikannya pada SWI-Prolog

Kami memulai pengerjaan dengan membuat draft kasar CFG yang akan kami buat (lihat lampiran 1). Selanjutnya kami mencoba memikirkan bahasa pemrograman apa yang akan kami gunakan untuk mengimplementasikan CFG tersebut. Awalnya kami ingin menggunakan Python3 dengan library lark, namun ternyata kami tidak diperbolehkan menggunakan library - library selain library bawaan bahasa tersebut. Akhirnya kami memilih untuk menggunakan bahasa Prolog, lebih tepatnya SWI-Prolog.

Alasan kami memilih menggunakan bahasa tersebut adalah karena sifat Prolog yang secara alamiah akan mencari semua “kemungkinan” / “parse tree” yang mungkin dapat dihasilkan, sehingga kami berpikir akan cukup mudah bila digunakan untuk mengimplementasikan CFG dalam bahasa tersebut. Beruntung, SWI-Prolog memiliki library bawaan bernama Definite Clause Grammar (DCG). Namun tidak semudah memasukkan CFG ke dalam prolog saja, namun kami juga perlu belajar menggunakan DCG serta mengintegrasikan cara menggunakan DCG dengan bahasa Prolog.

Pada laman resmi Python, terdapat referensi manual bagaimana Python melakukan kompilasi bahasanya, mulai dari identifikasi atom, token, compound, statement, hingga pembentukan block (pada laman, block tersebut dinamakan *saint*), sehingga pembuatan CFG untuk mengkompilasi bahasa Python pun menjadi lebih mudah.

3.2 Melakukan binerisasi CFG

Sebagian besar CFG yang kami buat sudah pendek (3-4 variabel) sehingga proses binerisasi kami lakukan secara manual, yaitu dengan membuat rule antara dari rule 1 ke rule 2, misalnya:

```
Float -> int . int
```

menjadi,

```
Float -> int float_1
```

```
Float_1 -> . int
```

Proses binerisasi ini dilakukan oleh kami untuk memudahkan guna meningkatkan efisiensi dari program serta mempermudah proses konversi dari *grammar* bentuk CFG ke dalam *grammar* bentuk CNF

3.3 Mengkonversi CFG menjadi CNF

Selanjutnya kami mengkonversi CFG yang sudah dibinerisasi pada langkah sebelumnya menjadi CNF, yaitu dengan melakukan langkah - langkah konversi CFG ke CNF pada *grammar* CFG kami. Kami menemui kesulitan di tahap menghilangkan epsilon dari *production rule* karena CFG kami menggunakan cukup banyak epsilon. Kami memutuskan untuk membuat *script* dengan bahasa Python untuk melakukan konversi CFG ke CNF. Pada konversi ini juga dilakukan penyesuaian format CFG dari bahasa pemrograman Prolog(DCG) ke dalam bentuk format yang dapat menjadi input bagi algoritma CYK kami, yaitu dengan mengubah ‘-->’ menjadi ‘->’ serta mengubah ‘;’ (or) menjadi rule terpisah dengan nama yang sama, menghilangkan ‘.’ di akhir rule (karena titik tersebut hanya merupakan sintaks prolog), serta mengubah terminal yang tadinya diapit kutip dua “” menjadi kutip satu “”.

Proses pengkonversian bentuk *grammar* CFG ke bentuk CNF dilakukan dengan menyimpan terminal, variabel dan aturan - aturan produksi di suatu file eksternal. File eksternal ini kemudian akan dimasukkan sebagai parameter pada saat menggunakan script *convert.py*. Script tersebut kemudian akan mengubah bentuk *grammar* CFG ke bentuk CNF melalui 4 tahap yakni:

1. Mengeliminasi produksi epsilon pada aturan produksi
2. Menghilangkan aturan - aturan produksi yang merupakan *unit production*
3. Menghilangkan aturan - aturan produksi yang tidak berguna
4. Menjadikan hasil produksi dari aturan produksi berbentuk 2 non-terminal atau 1 terminal

Hasil *grammar* CNF yang dihasilkan script tersebut akan disimpan di file eksternal lainnya. File eksternal yang menyimpan hasil pengkonversian *grammar* CFG ke bentuk CNF inilah yang kemudian akan dijalankan oleh algoritma cyk untuk meng-*parse* kode uji Python.

Selanjutnya kami sadar bahwa algoritma cyk ini hanya dapat memproses karakter satu demi satu, sehingga terminal - terminal berupa kata, seperti “from”, “while”, dan lain lain, harus diubah, yaitu dengan cara dipecah menjadi beberapa rule yang menyatakan kata tersebut, contoh:

`“False”`

diubah menjadi,

```
char_False --> char_F, charalse.  
charalse --> char_a, charlse.  
charlse --> char_l, charse.  
charse --> char_s, char_e.  
char_F --> “F”.  
char_a --> “a”.  
char_l --> “l”.  
char_s --> “s”.  
char_e --> “e”.
```

Hal yang sama juga dilakukan untuk operator yang lebih dari 1 karakter, misalnya “**”, “+=”, “/=", dll, namun untuk operator tidak dapat diberi nama sesuai dengan operatornya, contoh:

`char_** (tidak valid)`

maka kami mengubah nama variabel dengan ascii yang bersesuaian, menjadi

`char_4242`

3.4 Mengimplementasikan Algoritma CYK pada CNF

Setelah CFG yang kami buat telah terkonversi sempurna menjadi bentuk Chomsky Normal Form, kami dapat melakukan algoritma CYK pada CFG tersebut. Implementasi algoritma CYK dilakukan dengan bahasa Python berdasarkan pseudocode yang telah dibahas di bab 2.4.

BAB IV

SPESIFIKASI TEKNIS PROGRAM

4.1 Struktur Program

Source code kami dapat dibagi menjadi 4. Program CFG, konversi CFG ke CNF, implementasi CYK pada CNF, serta beberapa program antara.

4.1.1 Struktur Program CFG

Program kami terdiri dari sebuah file driver (cfg.pl) dengan 3 file input (input<x>.txt) dan 18 file database yang dibagi menjadi 3 tipe:

Atom	Statement	Compound
Datatype	Argument	Block
Number	Comment	Class
Operator	Expression	If_else
String	Function	Loop
Terminals	Import	With_as
Util	Parameter	
	Variable	

Tabel 2 Struktur Program CFG

4.1.2 Struktur Program Antara

Program ini terdiri dari beberapa script, yaitu

- Script_parsing_kwd
- Script_parsing_cfg
- Script_parsing_cnf_to_cyk (converter.py)
- Benchmark CYK

4.1.3 Struktur Program Konversi CFG ke CNF

Program kami terdiri dari 2 file utama, yaitu file convert.py untuk melakukan konversi *grammar* dari bentuk CFG ke dalam bentuk CNF, dan utility.py yang berfungsi untuk mendukung kerja dari program convert.py

4.1.4 Struktur Program Implementasi CYK pada CNF

Program untuk mengimplementasikan CYK pada CNF dibuat dalam bahasa python, yaitu `cyk.py` untuk melakukan *parsing* dengan algoritma CYK, serta terdapat 2 file txt yaitu `grammar.txt` yang berisi *grammar-grammar* dan `input.txt` yang merupakan inputan yang akan di cek apakah dapat di compile.

4.2 Kode Program

4.2.1 Kode Program CFG

File	Isi File
Datatype	Tipe data dasar seperti angka, set, dictionary, list, tuple, dan None
Number	Tipe data angka, yaitu integer, float, bilangan kompleks, scientific, serta boolean yang merupakan integer dalam Python
Operator	Kumpulan operator yang valid pada python serta pengelompokan operator tersebut seperti prefix, postfix, infix, logical, bitwise, comparator, dll.
String	String yang valid dalam python, yaitu kumpulan karakter yang diapit dengan tanda kutip (tunggal maupun ganda), termasuk multi-line string yaitu diapit dengan tiga tanda kutip (tunggal maupun ganda).
Terminals	Keyword dan representasi CFG nya (lihat bab 3.3), serta semua terminal yang mungkin (alfabet, angka, dan berbagai simbol)
Util	Kombinasi karakter yang sering dipakai dalam CFG seperti kurung buka, kurung tutup, satu atau lebih spasi, enter yang disambung dengan indentasi, dll
Argument	Argumen yang valid bagi sebuah pemanggilan fungsi
Comment	Komentar yang valid, yaitu seperti sebuah string yang diawali dengan # dan diakhiri dengan \n
Expression	Ekspresi aritmatik yang valid, aturan mengikuti aturan matematika standar dengan beberapa jenis operator yang mungkin, yang sudah dibuat di file <code>operator.pl</code> , namun ditambah dengan “not” untuk negasi dari ekspresi tersebut. Ekspresi yang valid juga termasuk ekspresi di dalam kurung yang bisa saja berulang / nested.
Function	Nama fungsi yang valid, pembuatan fungsi yang valid, serta beberapa statement yang berhubungan dengan fungsi, seperti <code>raise</code> dan <code>return</code> .
Import	Statement import yang valid pada python, termasuk <code>from .. import ..</code> , <code>import .. as ..</code> , <code>from .. import *</code> , dan

	kombinasi lainnya
Parameter	Parameter yang valid, termasuk kurung buka dan kurung tutup nya. Termasuk juga <code>*argument</code> dan <code>**keyword_argument</code> .
Variable	Variabel yang valid yaitu diawali dengan <code>_</code> atau alfabet, lalu diikuti dengan sejumlah alfabet / <code>_</code> / angka. Selain itu pada python juga dimungkinkan menggunakan <code>.</code> untuk menyambung variabel
Block	Sebuah block yang valid pada python, namun dapat menerima indentasi berapa banyak pun. Program ini juga memvalidasi block agar tidak kosong.
Class	Class yang valid
If_else	If statement yang utuh dan valid yaitu if, diikuti dengan 0 atau lebih elif, lalu diikuti dengan 0 atau 1 else.
Loop	Loop dalam python seperti while dan for. Beserta statement yang ada di dalamnya seperti break dan continue.
With_as	Statement with yang valid yaitu <code>with expressions as expressions</code> .

Tabel 3 Kode Program CFG

4.2.2 Kode Program Antara

File	Isi File
Parsing keyword	Mengubah terminal yang lebih dari satu karakter menjadi CFG yang bersesuaian seperti yang telah dijelaskan pada bab 3.3
Parsing cfg	Mengubah <i>grammar</i> Swi-Prolog yang terdapat dalam bahasa Prolog agar lebih mudah untuk dibentuk ke dalam bentuk CFG yang akan dikonversi ke dalam bentuk CNF
Parsing to CYK (format.py)	Mengubah format <i>grammar</i> CNF yang dihasilkan dari proses konversi CFG ke CNF agar lebih mudah diproses pada script algoritma parser CYK.
Benchmark CYK	Menghitung perkiraan waktu yang diperlukan untuk menjalankan algoritma CYK kami untuk suatu file input tertentu

Tabel 4 Kode Program Antara

4.2.3 Kode Program Konversi CFG ke CNF

File	Isi File
convert.py	Kode program yang berisikan metode - metode dalam mengubah <i>grammar</i> untuk mengkompilasi bahasa Python dari bentuk CFG ke dalam bentuk CNF. Program ini berisikan 4 tahap cara pengubahan bentuk CFG, yakni penghilangan epsilon, penghapusan unit production, pembuangan rule yang kurang berguna dan pengubahan production rule ke dalam bentuk 2 non-terminal atau 1 terminal
utility.py	Kode program yang berisikan fungsi - fungsi bantuan untuk mendukung kinerja dari file convert.py

Tabel 5 Kode Program Konversi CFG ke CNF

4.2.4 Kode Program Implementasi CYK pada CNF

File	Isi File
cyk.py	Kode program yang berisikan metode untuk <i>parsing</i> kata dari input.txt dengan algoritma CYK sebagai metode <i>parsingnya</i> dan grammar.txt sebagai <i>rules</i> .
grammar.txt	File ini berisi <i>grammar-grammar</i> dalam bentuk CNF
input.txt	File ini berisi inputan yang akan di test apakah dapat di compile di python 3.7

Tabel 6 Kode Program Implementasi CYK pada CNF

4.3 Realisasi Program

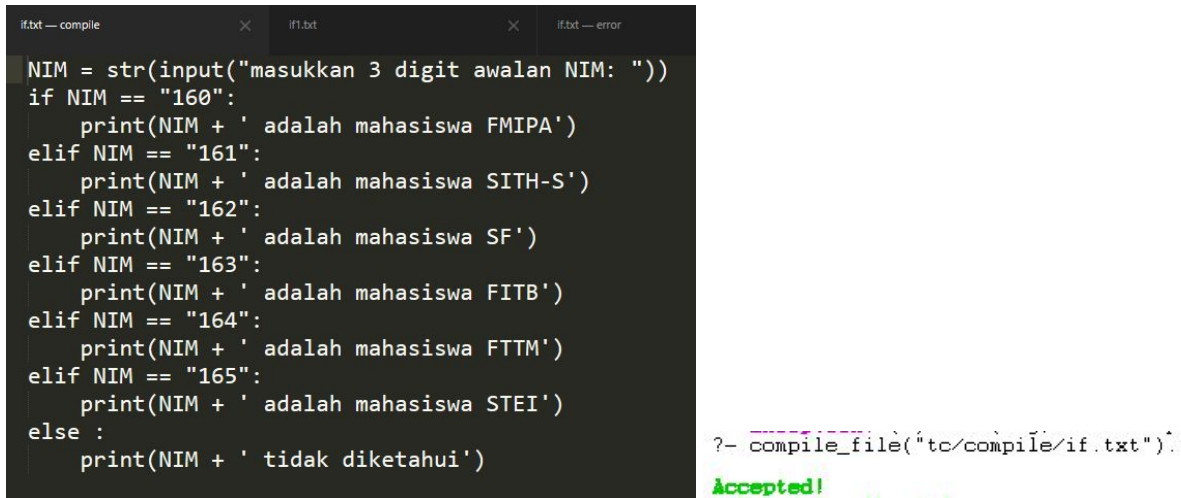
Program CFG	Program CFG telah kami pastikan lengkap dan benar dengan menggunakan program yang kami buat dalam bahasa SWI-Prolog.
Program antara	Program antara berjalan dengan baik dan bisa menjadi input bagi program konversi CFG ke CNF
Konversi CFG ke CNF	Program ini berhasil hanya untuk skala yang kecil (misal: menerima integer, float, operator simpel, nama variabel, dll), namun gagal ketika ada cukup banyak epsilon seperti block statement if, while, dll.
Algoritma CYK	Program ini telah kami pastikan berjalan dengan baik dengan menggunakan rule simpel seperti menerima integer, float, operator simpel, nama variabel, dll. Program kami juga dapat mengeluarkan parse tree yang didapatkannya.

Tabel 7 Realisasi Program

BAB V

EKSPERIMEN

5.1 Hasil Tangkapan Layar



```
if.txt — compile  if1.txt  if.txt — error
NIM = str(input("masukkan 3 digit awalan NIM: "))
if NIM == "160":
    print(NIM + ' adalah mahasiswa FMIPA')
elif NIM == "161":
    print(NIM + ' adalah mahasiswa SITH-S')
elif NIM == "162":
    print(NIM + ' adalah mahasiswa SF')
elif NIM == "163":
    print(NIM + ' adalah mahasiswa FITB')
elif NIM == "164":
    print(NIM + ' adalah mahasiswa FTTM')
elif NIM == "165":
    print(NIM + ' adalah mahasiswa STEI')
else :
    print(NIM + ' tidak diketahui')
```

```
?- compile_file("tc/compile/if.txt").
Accepted!
```

Gambar 3 Kasus Uji 1

Kasus uji 1 berisi if statement yang benar. Program ini benar karena berisi sebuah statement if yang valid yaitu sebuah if clause, diikuti dengan 0 atau lebih elif clause, yang diikuti dengan 0 atau 1 else clause.

Selain itu, di dalam if block juga terdapat sebuah fungsi print dengan parameter ekspresi penambahan string dengan variabel NIM. Semua string valid karena diapit dengan jenis kutip yang sama dan memiliki pasangannya sebelum end line (`\n`).

Di baris paling atas juga terdapat sebuah statement berupa fungsi di dalam fungsi, yang di dalamnya terdapat string dengan varian yang berbeda yaitu diapit double quote. Program kami tetap dapat membacanya.

Perlu diperhatikan juga setelah if keyword terdapat ekspresi dengan operator `'=='` antara variabel dengan string. Ekspresi tersebut juga merupakan ekspresi yang valid sehingga input mengeluarkan verdict Accepted.

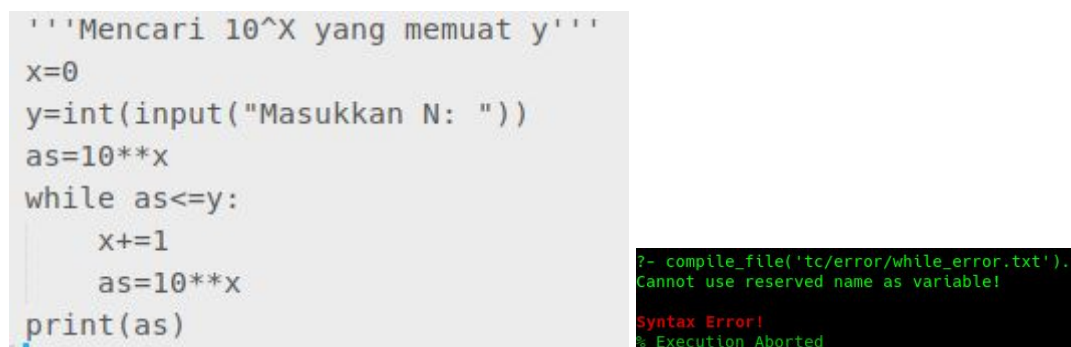


```
1 # TestCase Error
2 x=int(input("Masukkan nilai x: "))
3 y=int(input("Masukkan nilai y: "))
4 if(x==0 and y==0):
5     print(''+str(x)+'',''+str(y)+'',"ada di titik pusat.")
6 elif(x>0 and y>0):
7     print(''+str(x)+'',''+str(y)+'',"ada di kuadran 1.")
8 elif(x<0 and y>0):
9     print(''+str(x)+'',''+str(y)+'',"ada di kuadran 2.")
10 elif(x<0 and y<0):
11     print(''+str(x)+'',''+str(y)+'',"ada di kuadran 3.")
12 elif(x>0 and y<0):
13     print(''+str(x)+'',''+str(y)+'',"ada di kuadran 4.")
14 else:
```

?- compile_file("tc/error/if.txt").
a block must contain something
Syntax Error!

Gambar 4 Kasus Uji 2

Kasus uji 2 berisi if statement yang salah. Walaupun string-string beserta operator infix di dalam parameter fungsi print sudah benar, masih terdapat suatu kesalahan dalam program ini yang terletak pada baris terakhir dari blok program. Kasus if dan elif statement sudah benar, namun else tidak memiliki blok program yang benar dan tidak mengandung statement apapun. Program kami juga dapat memberitahu tipe kesalahan yang terjadi, namun belum dapat memberikan lokasi kesalahan tersebut.



```
'''Mencari 10^X yang memuat y'''
x=0
y=int(input("Masukkan N: "))
as=10**x
while as<=y:
    x+=1
    as=10**x
print(as)
```

?- compile_file('tc/error/while_error.txt').
Cannot use reserved name as variable!
Syntax Error!
% Execution Aborted

Gambar 5 Kasus Uji 3

Kasus uji 3 berisikan while loop yang salah. Walaupun syntax keseluruhan dari format block while loop kode di atas cenderung benar, namun dalam kompilasi kode Python tidak diperkenankan untuk menggunakan keyword (reserved word) sebagai nama variabel. Penggunaan huruf *as* sebagai variabel telah melanggar aturan penggunaan keyword sebagai variabel tersebut, menyebabkan syntax error saat kompilasi kode berikut.

```
import pandas as pd
import numpy as np
from sklearn import model_selection, preprocessing
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
```

```
% Execution Aborted
?- compile_file('tc/compile/import1.txt').
Accepted!
% Execution Aborted
```

Gambar 6 Kasus Uji 4

Kasus uji 4 berisikan potongan kode yang lulus uji kompilasi kode Python. Dalam potongan kode tersebut, dapat dilihat bahwa terdapat 2 jenis template yang dipakai untuk mengimport suatu modul, yakni `import <nama_modul> as <alias>` dan `from <modul> import <isi_modul>`. Karena potongan kode di atas telah memenuhi persyaratan syntax Python, maka potongan kode di atas lulus uji kompilasi syntax Python.

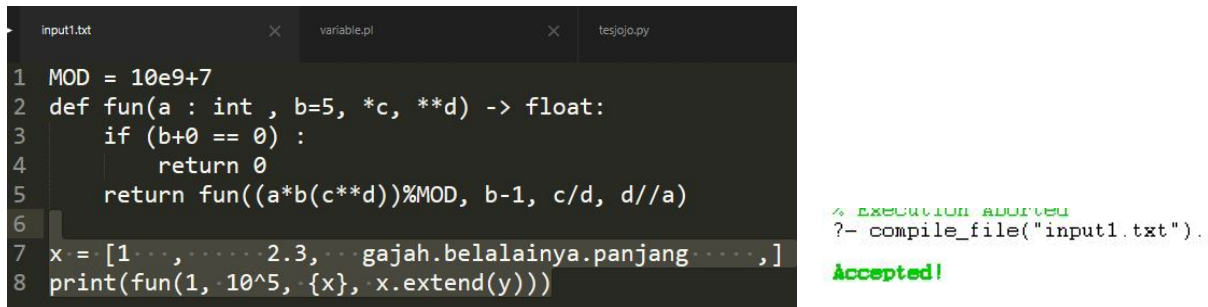
```
1  '''
2  Mencari apakah X merupakan bilangan prima
3  '''
4  x=int(input("Masukkan X: "))
5  if x<=1: #x kurang dari sama dengan 1
6      print(x,"bukan bilangan prima")
7  for i in range(2,x+1):
8      if x>1:
9          if x%i==0:
10             if x==i:
11                 print(x,"adalah bilangan prima")
12             else:
13                 print(x,"bukan bilangan prima")
14             x=0
```

```
?- compile_file("tc/compile/forcompile.txt").
Accepted!
% Execution Aborted
?-
```

Gambar 7 Kasus Uji 5

Kasus uji 5 berisikan for loop yang benar. Dalam potongan kode tersebut dapat dilihat bahwa ada sebuah for loop dengan isi nested if. Terdapat variabel x yang merupakan hasil inputan yang merupakan integer, yang digunakan untuk mencari apakah x tersebut merupakan bilangan prima atau bukan.

Untuk di dalam for terdapat pula variabel x dan i yang digunakan untuk menentukan apakah x itu bilangan prima ataupun tidak, jika x setelah di `mod(%) i` merupakan 0 dan x adalah i (menggunakan logical operator `'=='`) maka x merupakan bilangan prima, apabila bukan, maka x bukan bilangan prima, oleh sebab itu kasus uji 5 accepted karena sesuai dengan syntax python.

The image shows a code editor with three tabs: 'input1.txt', 'variable.pl', and 'tesjojo.py'. The 'tesjojo.py' tab is active, displaying a Python script. The script defines a function 'fun' with parameters 'a' (int), 'b' (default 5), '*c', and '**d' (returning float). It includes an if-statement to return 0 if 'b+0 == 0', and a recursive call 'return fun((a*b(c**d))%MOD, b-1, c/d, d//a)'. Below the function, a list 'x' is assigned with elements '1', '2.3', and a string 'gajah.belalainya.panjang'. The script ends with a print statement: 'print(fun(1, 10^5, {x}, x.extend(y)))'. To the right of the code, the execution output is shown: '% EXECUTION ADDRESS' followed by '?- compile_file("input1.txt").' and 'Accepted!' in green text.

```
1 MOD = 10e9+7
2 def fun(a : int , b=5, *c, **d) -> float:
3     if (b+0 == 0) :
4         return 0
5     return fun((a*b(c**d))%MOD, b-1, c/d, d//a)
6
7 x = [1, 2.3, 'gajah.belalainya.panjang',]
8 print(fun(1, 10^5, {x}, x.extend(y)))
```

% EXECUTION ADDRESS
?- compile_file("input1.txt").
Accepted!

Gambar 8 Kasus Uji 6

Kasus uji 6 berisikan sebuah assignment, sebuah blok fungsi yang berisi sebuah if statement dengan dua return statement, serta sebuah assignment list dan sebuah fungsi print.

Assignment pada baris 1 adalah assignment sebuah angka dengan scientific notation ($10e9+7 = 1.000.000.007$). Pada baris 2 terdapat sebuah definisi fungsi fun dengan empat parameter, masing-masing memiliki tipe yang berbeda. Pertama adalah parameter biasa (a), namun dilengkapi dengan type hinting (: int). Selanjutnya adalah parameter dengan default value, ditandai dengan assignment operator (=). Ketiga ada argument (args) parameter yaitu sebuah variabel dengan * di depannya. Terakhir adalah keyword argument (kwargs) parameter yaitu sebuah variabel dengan ** di depannya. Di belakang parameter terdapat sebuah type hinting lain, yaitu menunjukkan hasil return fungsi tersebut, yakni float.

Di dalam fungsi fun, terdapat sebuah statement if dengan ekspresi yang melibatkan operator logical == dan operator +. Setelah itu ada sebuah statement return yang mengembalikan nilai 0. Di baris 5, terdapat sebuah return statement lain, dengan ekspresi yang cukup panjang dan rumit. Ada sebuah fungsi dengan parameter di dalamnya yang melibatkan banyak operator, namun semua operator tersebut adalah valid secara sintaks.

Di luar fungsi tersebut ada assignment sebuah list ke dalam variabel x. Walaupun terdapat banyak spasi di antara elemen2 nya, program kami tetap dapat membacanya sebagai sebuah list. Di dalamnya juga terdapat dot operator yang digunakan untuk mengambil method sebuah class, sehingga dapat menjadi nama variabel yang valid. Ada pula trailing comma yang juga valid di bahasa Python.

Baris terakhir ada sebuah statement yang memanggil fungsi print. yang memanggil fungsi lain yaitu fun, dengan 4 argumen, di dalam argumennya tidak harus variabel, namun bisa juga ekspresi yang panjang dan rumit. Bisa juga sebuah datatype seperti set ({x}). Bisa

juga sebuah method seperti `x.extend`, yang di dalamnya memanggil fungsi lain lagi. Keempat variabel ini valid sintaksnya dan dapat dicompile. Semua kurung juga memiliki pasangannya dan tidak memiliki new line (`\n`) di antaranya.

BAB VI

KESIMPULAN

6.1 Hasil yang Dicapai

Pada tugas kali ini, kami berhasil membuat *Compiler (Grammar Checker)* Python. Kami membuat program ini dengan bahasa SWI-Prolog untuk mengecek CFG dan menggunakan bahasa Python untuk melakukan konversi ke CNF dan untuk memasukkannya dalam algoritma CYK. Namun, program kami berjalan dengan sangat lambat karena memiliki banyak rule setelah dikonversikan dalam bentuk CNF serta program kami juga belum sempurna karena program konversi ke CNF kami baru bisa mengatasi aturan yang simpel saja (lihat bab 4.3). Untuk mengatasi hal ini, kami telah mempersiapkan sebuah program yang dapat didemokan yang dibuat dengan menggunakan bantuan DCG (*Definite Clause Grammar*) yang merupakan modul bawaan dari bahasa SWI-Prolog.

6.2 Saran dan Pengembangan

Untuk pengembangan ke depannya, program sebenarnya dapat menggunakan algoritma yang lain agar lebih cepat dalam memproses kata-kata. Seperti advanced string matching algorithm, dll. Selain itu, dalam bahasa Python sebenarnya juga sudah terdapat library untuk lexing dan tokenizing agar proses parsing lebih mudah dilakukan.

Sebaiknya spesifikasi tugas diperiksa terlebih dahulu sebelum dirilis dan diuji terlebih dahulu oleh beberapa orang, minimal mengecek apakah tugas tersebut memang dapat diselesaikan dengan metode yang diajarkan. Menurut kami, bobot tugas ini terlalu berat untuk jumlah sks nya, terlebih lagi bila dibandingkan dengan tugas tahun-tahun lalu.

6.3 Refleksi

Melakukan proses kompilasi suatu kode program ternyata tidaklah mudah. Seharusnya untuk melakukan tugas kali ini, diperlukan pemahaman mengenai materi yang bersangkutan lebih matang. Selain itu, ada baiknya pemahaman - pemahaman mengenai materi - materi di bidang lain dapat kami terapkan dalam tugas ini. Pemanfaatan waktu yang lebih baik lagi juga dapat menjadi refleksi untuk kami dalam menghadapi tugas tugas serupa berikutnya.

DAFTAR REFERENSI

- Anonymous. 2019. *Python Data Types*.
<https://www.tutorialsteacher.com/python/python-data-types> diakses pada 14 November 2019.
- _____. 2019. *The Python Language References*.
<https://docs.python.org/3.7/reference> diakses pada 14 November 2019.
- _____. 2013. *List of Keywords in Python*.
https://www.programiz.com/python-programming/keyword-list#from_import diakses pada 14 November 2019.
- _____. 2010. *Python*. <https://techterms.com/definition/python> diakses pada 14 November 2019.
- _____. 2018. *Converting Context Free Grammar to Chomsky Normal Form*.
<https://www.geeksforgeeks.org/converting-context-free-grammar-chomsky-normal-form/> diakses pada 14 November 2019.
- _____. 2012. *Reference Manual*.
https://www.swi-prolog.org/pldoc/doc_for?object=manual diakses pada 14 November 2019.
- _____. 2016. *SWI-Prolog*. <https://www.swi-prolog.org> diakses pada 27 November 2019.
- _____. 2016. *DCG Grammar rules*.
<https://www.swi-prolog.org/pldoc/man?section=DCG> diakses pada 27 November 2019.
- _____. 2006. *Definite clause grammar*.
https://en.wikipedia.org/wiki/Definite_clause_grammar#Parsing_with_DCGs diakses pada 27 November 2019.
- Aulia, Alvina. 2018. *Penyederhanaan Context Free Grammar*.
<https://socs.binus.ac.id/2018/12/20/penyederhanaan-context-free-grammar/> diakses pada 14 November 2019.
- Bacon, Dave. 2011. *Introduction to Formal Methods in Computer Science Chomsky Normal Form*. <https://courses.cs.washington.edu/courses/cse322/08au/lec14.pdf> diakses pada 14 November 2019.

- Eisele, Robert. 2018. *The CYK Algorithm*. <https://www.xarg.org/tools/cyk-algorithm/> diakses pada 14 November 2019.
- Fairuzabadi, Muhammad. 2011. *Context Free Grammar*. <https://fairuzelsaid.wordpress.com/2011/06/16/tbo-context-free-grammar-cfg/> diakses pada 14 November 2019.
- Nogatz, Falco, Dietmar Seipel, Salvador Abreu. 2019. *Definite Clause Grammars with Parse Trees: Extension for Prolog*. OASlcs-SLATE-2019-7.pdf.
- Ogborn, Anne. 2013. *Using Definite Clause Grammars in SWI-Prolog*. <http://www.pathwayslms.com/swipltuts/dcg/> diakses pada 14 November 2019.

LAMPIRAN

kwd :	expr > is -> expr
class	
class -> var -> classblock	not
	not -> expr
classblock	
newblock -> *(funtions expr)	with, as
	with expr {as var}
return	
return -> sentence	if, elif, else
	if -> var -> if_block -> *elifs -> elses
continue	
continue -> endline	elifs
	elif -> var -> if_block
for, in	
for -> var -> in -> iterables -> new_block	elses
	else -> if_block
def	
def -> method -> newblock	if_block
	newblock -> *sentence pass
from	
from -> var -> import -> var * -> newline	break
	raise
while	
while -> var -> loopblock	newblock
	: -> newline -> indent
loopblock	
newblock -> (*sentence) break	method
	var -> (-> *param ->) -> {'->' -> datatype
and, or, is	e } ->seq

param {* ** e } -> var -> { : -> datatype} datatype num seq dict mutable immutable None False (int, 0) True (int, 1) num Int float complex	list string tuple var (_ alphabet) -> *(alphabet _ angka e) angka 1..9 alphabet lowercase uppercase lowercase a..Z uppercase A..Z newblock
---	--

Tabel 8 Draft proses pembuatan CFG