

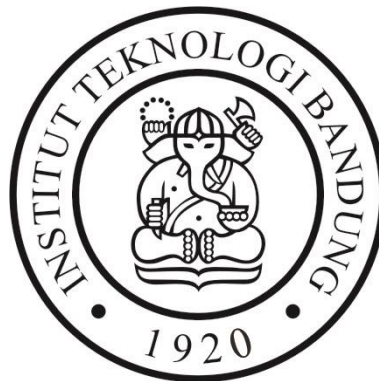
LAPORAN TUGAS BESAR

**Implementasi Compiler untuk Python Menggunakan
Konsep Context – Free – Grammar dan Finite
Automata**

Ditujukan untuk memenuhi tugas besar mata kuliah IF2124 Teori Bahasa Formal dan
Otomata pada Semester I Tahun Akademik 2021/2022

Disusun oleh:

Rifqi Naufal Abdjul	13520062
Saul Sayers	13520094
Amar Fadil	13520103



**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2021**

Daftar Isi

Daftar Isi.....	i
BAB I TEORI DASAR	1
1) Lexical Analysis (Tokenization).....	1
2) Finite Automata (FA).....	1
3) Context Free Grammar (CFG).....	4
4) Chomsky Normal Form (CNF).....	5
5) Algoritma Cocke–Younger–Kasami (CYK)	6
6) Syntax Python	7
BAB II HASIL DAN PEMBAHASAN.....	10
1) Lexical Analysis (Tokenization).....	10
2) Finite Automata (FA).....	11
3) Context – Free – Grammar (CFG).....	12
BAB III IMPLEMENTASI DAN PENGUJIAN	18
1) Spesifikasi Teknis Program.....	18
2) Uji Kasus dan Analisis.....	20
BAB IV KESIMPULAN DAN SARAN	24
1) Kesimpulan.....	24
2) Saran	24
BAB V PENUTUP.....	25
Link To Repository.....	25
Pembagian Tugas	25
Referensi	26

BAB I

TEORI DASAR

1) Lexical Analysis (Tokenization)

Lexical analysis merupakan proses mengubah rangkaian string menjadi rangkaian token. Sedangkan token merupakan gabungan dari string yang mempunyai suatu arti. Proses tokenization biasanya terbagi menjadi 2 tahap, yaitu:

1. Scanning

Pada tahap ini input string akan dipotong menjadi beberapa bagian yang dapat dievaluasi selanjutnya. Biasanya scanning berdasarkan Finite State Machine, tetapi juga bisa menggunakan pemotongan berdasarkan whitespace dan simbol.

2. Evaluating

Pada tahap ini input string akan dievaluasi menjadi token. Menggunakan evaluator suatu bagian akan di evaluasi menjadi value atau token.

Sebagai contoh, sebuah string dalam code berikut

```
net_worth_future = (assets - liabilities);
```

akan menghasilkan rangkaian token seperti berikut

```
IDENTIFIER net_worth_future  
EQUALS  
OPEN_PARENTHESIS  
IDENTIFIER assets  
MINUS  
IDENTIFIER liabilities  
CLOSE_PARENTHESIS  
SEMICOLON
```

2) Finite Automata (FA)

Finite Automata (FA), atau yang dapat disebut juga sebagai Finite State Automata (FSA), merupakan sebuah mesin abstrak berupa model sistem matematika untuk mengenali pola dari sebuah string atau set karakter. Finite State Automata merupakan mesin otomatis dari suatu bahasa paling sederhana yakni bahasa reguler. Suatu FSA disebut 'Finite State' karena memiliki state yang terbatas sehingga dapat berpindah – pindah dari suatu state ke state yang lain. Karena FSA tidak memiliki memori untuk mengingat state dan hanya bisa mengingat state terkini, maka perpindahan state pada FSA ditentukan oleh suatu fungsi transisi.

Cara kerja dari suatu Finite State Automata adalah memproses masukan string dengan membaca 1 karakter tiap saat dari kiri ke kanan yang akan dikendalikan oleh fungsi transisi sehingga mesin dapat berpindah dari suatu state ke state yang lain. Mulanya, FSA berada pada suatu state awal (*initial state*), kemudian akan berpindah setelah memproses tiap karakter dan apabila state akhirnya mencapai suatu final state maka string diterima oleh mesin FSA. Secara formal, Finite State Automata dinyatakan oleh pasangan 5 tuple, yaitu :

$$M = (Q, \Sigma, \delta, S, F)$$

di mana :

Q = Himpunan semua state / kedudukan

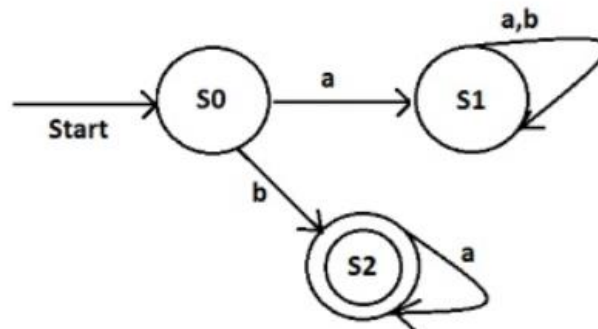
Σ = Himpunan masukan / simbol input

δ = fungsi transisi

S = State awal (Hanya satu) , $S \in Q$

F = Himpunan State Akhir (Bisa lebih dari satu), $F \cap Q$

Finite State Automata dapat direpresentasikan dalam bentuk diagram yang disebut sebagai State Transition Diagram. Diagram tersebut menunjukkan transisi dari tiap state sesuai dengan fungsi transisi. Tiap state dimodelkan sebagai sebuah lingkaran, dengan transisinya dimodelkan sebagai arah tanda panah. Initial State diindikasikan dengan diberi tanda panah dan keterangan start, sementara final state diindikasikan dengan lingkaran bergaris ganda, dan state biasa diindikasikan dengan lingkaran bergaris tunggal. Label pada tiap busur menyatakan simbol input.



Gambar 1.2.1 Contoh State Transition Diagram

Selain dalam bentuk diagram, Finite State Automata juga dapat direpresentasikan dalam bentuk tabel yang disebut sebagai State Transition Table. Secara matematis, fungsi transisi Finite State Automata dimodelkan sebagai $\delta(q_0, \sigma) = q_1$, di mana q_0 adalah state sebelum berpindah, σ adalah input yang diterima, dan q_1 adalah state setelah berpindah. Final state ditandai dengan sebuah asterisk (*) pada namanya. Apabila disajikan dalam bentuk tabel, maka semua transisi yang terjadi pada suatu Finite State Automata akan dicantumkan seperti contoh berikut :

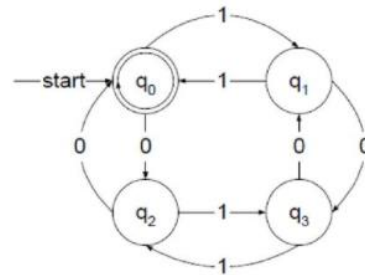
State	a	b
S0	S1	S2
S1	S1	S1
S2*	S2	-

Gambar 1.2.2 Contoh State Transition Table untuk FSA pada gambar 3.1.1

Secara umum, terdapat dua jenis Finite Automata, yaitu :

a) Deterministic Finite Automata (DFA)

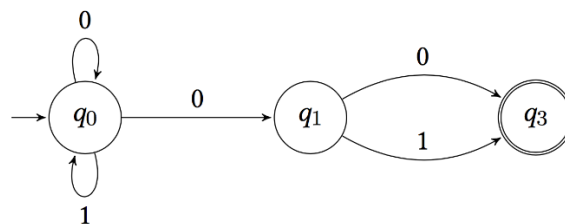
Pada deterministik Finite Automata, suatu state hanya memiliki tepat satu state berikutnya untuk setiap simbol input yang diterima. Suatu string x dinyatakan diterima bila $\delta(q_0, x)$ berada pada final state, di mana q_0 adalah initial state.



Gambar 1.2.3 Contoh State Diagram dari sebuah DFA

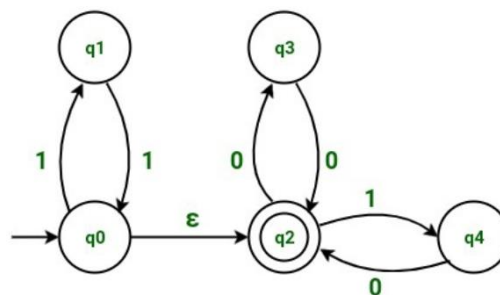
b) Non-deterministic Finite Automata (NFA)

Pada Non-deterministic Finite Automata, suatu state bisa memiliki 0, 1, ataupun lebih transisi untuk setiap simbol input yang diterima. Suatu string diterima oleh NFA bila terdapat suatu urutan transisi sehubungan dengan masukan string tersebut dari initial state menuju final state. Dengan demikian, untuk NFA, harus dicoba semua kemungkinan yang ada hingga ada agar terdapat suatu urutan transisi untuk mencapai final state.



Gambar 1.2.4 Contoh State Diagram dari sebuah NFA

Struktur dari NFA dapat diperluas dengan memberikan keberadaan transisi antara dua state yang terjadi secara spontan. Jenis NFA ini dinamai sebagai Epsilon-NFA atau ϵ – NFA. Pada Epsilon-NFA ini, memungkinkan untuk merubah state tanpa perlunya membaca sebuah input. Transisi ini juga dikatakan sebagai transisi ϵ karena tidak bergantung pada suatu input ketika melakukan transisi.

Gambar 1.2.5 Contoh State Diagram dari sebuah ϵ – NFA

3) Context Free Grammar (CFG)

Context Free Grammar adalah sebuah tata bahasa formal yang tidak memiliki batasan pada hasil produksinya. Secara formal, sebuah Context Free Grammar didefinisikan sebagai pasangan 4 tupel, yaitu:

$$G = (V, \Sigma, S, P)$$

di mana :

V = Himpunan variabel atau simbol non-terminal

Σ = Himpunan simbol terminal

S = Simbol start

P = Production rule

Setiap aturan produksi dalam Context Free Grammar memiliki bentuk sebagai berikut :

$$A \rightarrow \alpha$$

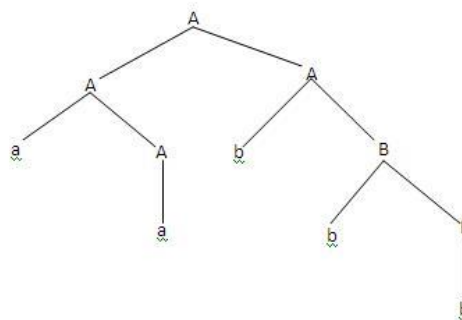
Di mana A adalah sebuah simbol nonterminal, $A \in V$, dan α adalah string of terminals dan/atau nonterminals, $\alpha \in \{V \cup \Sigma\}^*$. Bahasa yang dibuat oleh Context Free Grammar disebut sebagai Context Free Languages. Context Free Grammar memiliki tujuan yang sama dengan bahasa reguler yakni untuk menunjukkan bagaimana menghasilkan suatu untai – untai dalam sebuah bahasa. Berikut adalah contoh dari hasil aturan produksi Context Free Grammar :

$$S \rightarrow AB$$

$$A \rightarrow aA \mid a$$

$$B \rightarrow bB \mid b$$

Cara kerja dari Context Free Grammar dimulai dari simbol start, kemudian diterapkan rules of production untuk mendapatkan hasil produksi yang lain. Hal tersebut dilakukan berulang kali hingga didapatkan string hanya terdiri dari simbol terminal. Karena cara kerjanya tersebut, Context Free Grammar menjadi dasar pembentukan suatu parser atau analisis sintaksis. Bagian sintaks dari suatu compiler biasanya didefinisikan dalam Context Free Grammar. Parse tree atau Derivation tree berguna dalam proses parsing dari simbol – simbol variabel menjadi simbol – simbol terminal di mana setiap simbol variabel akan diturunkan menjadi terminal sampai tidak ada yang belum tergantikan. Contoh dari sebuah parse tree adalah sebagai berikut :

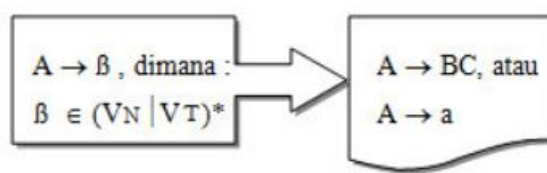


Gambar 1.3.1 Contoh parse tree dari contoh aturan produksi untuk mendapatkan string 'aabb'

4) Chomsky Normal Form (CNF)

Context Free Grammar (CFG) memiliki bentuk lain yakni bentuk Normal Chomsky atau Chomsky Normal Form (CNF). Bentuk normal Chomsky dapat dibuat dari sebuah tata bahasa bebas konteks yang telah mengalami penyederhanaan yaitu penghilangan produksi useless, unit, dan ϵ . Dengan kata lain, suatu tata bahasa bebas konteks dapat dibuat menjadi bentuk normal Chomsky dengan syarat tata Context Free Grammar tersebut tidak memiliki useless productions, tidak memiliki unit productions, dan tidak memiliki null productions. Selain itu, produksi dalam bentuk Chomsky Normal Form hanya bisa menghasilkan tepat satu buah terminal saja, atau dua buah non – terminal. Dengan kata lain, Bentuk CNF dapat direpresentasikan seperti berikut :

$$A \rightarrow BC \text{ atau } A \rightarrow a$$

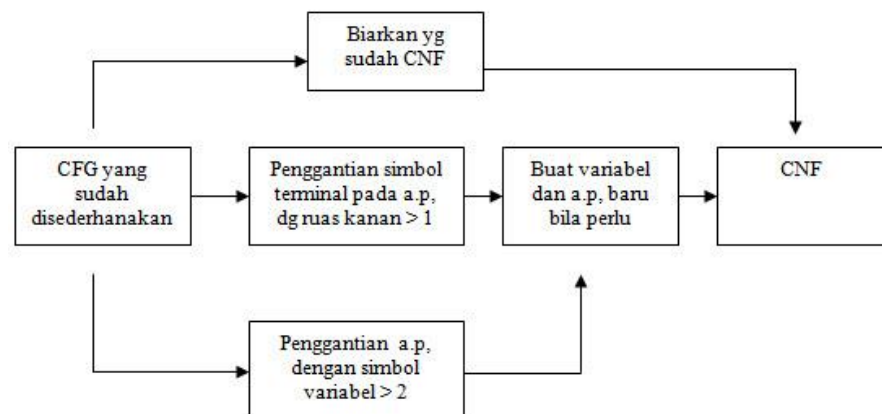


Transformasi CFG Ke CNF

Gambar 1.4.1 Konversi bentuk CFG ke CNF

Langkah – langkah untuk mengkonversi Context Free Grammar menjadi bentuk Chomsky Normal Form adalah sebagai berikut:

- Untuk tiap produksi yang bentuknya sudah sesuai dengan bentuk CNF, kita biarkan
- Mencari tiap useless production dan membuangnya
- Mencari tiap null productions (produksi yang memiliki epsilon) dengan cara mensubstitusi kemungkinan produksi variabel tersebut di semua produksi lain lalu menghapus null productions itu sendiri
- Mencari tiap unit productions dan menyingkatnya
- Untuk tiap produksi yang ruas kanannya memiliki terminal tetapi panjang ruas kanannya lebih dari 1, dilakukan penggantian produksi
- Untuk tiap produksi yang ruas kanannya memiliki tidak memiliki terminal tetapi panjangnya lebih dari 2, dilakukan penggantian produksi
- Penggantian produksi digunakan dengan mensubstitusi 2 variabel menjadi 1 variabel baru, lalu menambahkan produksi dari variabel baru itu sendiri



Gambar 1.4.2 Diagram alur proses konversi CFG menjadi CNF

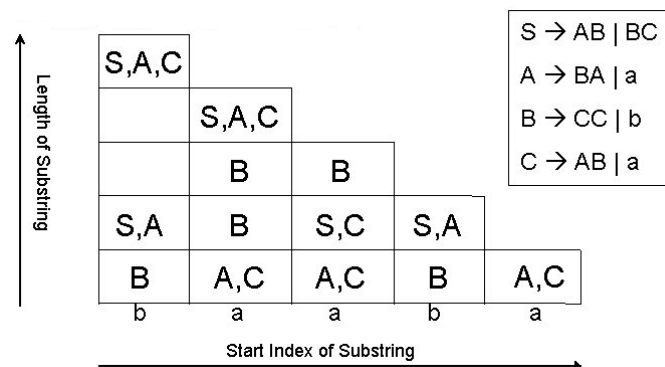
5) Algoritma Cocke–Younger–Kasami (CYK)

Algoritma Cocke – Younger – Kasami (CYK atau CKY) adalah algoritma parsing yang sangat efisien untuk Context Free Grammars. Hal ini mempermudah untuk menentukan masalah penerimaan dari sebuah kata oleh mesin bergantung bahasa yang dibuat oleh Context Free Grammar. Algoritma ini bisa digunakan untuk mengecek apakah sebuah kata $w \in \Sigma^*$ termasuk sebuah bagian dari bahasa Context Free Language yang disediakan dalam bentuk CNF.

Secara informal, cara kerja algoritma ini adalah:

- Membuat sebuah tabel berukuran $N \times N$, di mana N adalah panjang dari kata yang diinput kemudian kenuliskan kata pada baris pertama tabel dan menambahkan simbol non-terminal pada bagian bawah yang mendeduksi simbol terminal
- Untuk tiap cell dalam tabel secara vertikal mulai dari atas dan turun ke kotak bawahnya untuk dicek bersamaan dengan pasangan kotak lain yang naik secara diagonal.
- Untuk tiap langkah b, gabungkan tiap kotak yang bersesuaian dan cek apakah kombinasinya muncul dalam grammar tersebut
- Apabila muncul, maka tambahkan sisi kiri non-terminal ke kotak tersebut
- Apabila semua step sudah terlaksana dan start symbol terkandung pada kotak terbawah pada tabel tersebut, maka kata tersebut dapat diturunkan oleh grammar yang diberikan.

Dengan demikian, algoritma ini memperhitungkan semua kemungkinan subsequence dari huruf yang menambahkan non-terminal baru K ke tabel jika sequence dari huruf tersebut bisa diproduksi dari non-terminal K . Untuk setiap subsequence dengan panjang lebih besar sama dengan dari 2, maka algoritma ini memperhitungkan semua partisi dari subsequence yang mungkin menjadi dua bagian, dan mengecek apakah ada aturan produksi dari bentuk A menjadi BC di dalam grammar di mana B dan C masing – masing dapat memproduksi bagiannya sendiri.



Gambar 1.5.1 Contoh tabel CYK untuk untuk suatu string “baaba”

6) Syntax Python

Python bahasa pemrograman tingkat tinggi yang *intrepeted* dan *general purpose* dengan semantik yang dinamis. Karena struktur data nya yang berlevel tinggi dan karakteristik dynamic typing dan dynamic bindingnya, maka pengembangan aplikasi dan program menggunakan python menjadi lebih mudah dan berlaku juga untuk scripting untuk menggabungkan beberapa komponen yang lain. Syntax dalam python bersifat sederhana dan mudah untuk dipelajari sehingga menekankan pada keterbacaan oleh pengguna. Python mendukung beberapa modul dan package yang mendorong modularitas program. Interpreter Python dan library standarnya tersedia dan terdistribusi secara open source serta secara gratis untuk semua platform besar. Pada Python, tidak terdapat langkah kompilasi sehingga mempercepat proses edit, test, dan debugging. Debugging dalam Python sangat mudah, sebuah bug atau input yang salah tidak akan membuat segmentation fault, melainkan *raise an exception*. Saat program tidak menerima *exception* tersebut, maka interpreter akan mencetak stack trace tersebut.

Python mendukung pemrograman yang *multi-paradigm*, termasuk pemrograman berorientasi objek, pemrograman fungsional, pemrograman imperatif, dan masih banyak lagi. Python memiliki banyak sekali fitur yang didukung dan diakses oleh beberapa keywords yang tersedia dalam Python, di antaranya adalah :

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	

Gambar 1.6.1 Daftar kata kunci bawaan Python

Python juga memiliki struktur data sederhana yang luas, dan juga struktur data *collection* yang sangat membantu. Untuk yang bertipe sederhana, terdapat beberapa tipe seperti integer, floating point, bilangan desimal, bilangan kompleks, string (bersifat immutable), dan boolean (True dan False). Untuk yang bertipe collection, Terdapat list (berupa dynamic array), tuples, set, dan dictionaries yang sangat bermanfaat untuk mapping keys dan itemsnya. Python menyediakan banyak library bawaan yang membantu programmernya untuk memproses dan handle tiap struktur data tersebut.

Literals dalam Python dapat dibagi menjadi 3 jenis. Pertama adalah strings yang terbagi menjadi beberapa bagian, yakni normal strings yang diapit dengan satu atau dua tanda petik atau multiline strings yakni multiline strings yang diapit oleh tiga tanda petik. Yang kedua

adalah numbers dan dapat berupa integer ataupun float. Yang ketiga adalah lists, tuples, set, dictionaries untuk menampung beberapa elemen.

Selain itu, jenis operasi yang bisa dilakukan dalam Python juga terbagi menjadi 3. Yang pertama adalah operasi aritmatika, termasuk penjumlahan (+), pengurangan (-), pembagian asli (/), perkalian (*), perpangkatan (**), dan pembagian dengan pembulatan ke bawah (//). Yang kedua adalah operasi perbandingan, yakni sama dengan (==), tidak sama dengan (!=), kurang dari (<), kurang dari sama dengan (<=), lebih dari sama dengan (>=), is, is not, in, dan not in. Yang ketiga adalah operasi logika, di mana Python akan menganggap nilai kosong sebagai 0, None, "", ataupun list / set kosong. Selain itu, akan dievaluasi sebagai true.

Adapun fitur – fitur lain yang menjadi keunggulan dalam Python adalah:

- Memiliki tata bahasa yang mudah dipelajari
- Didukung sistem pengelolaan memori secara otomatis sehingga membutuhkan kinerja saat *coding*
- Terdapat banyak fasilitas pendukung sehingga mudah dan cepat saat pengoperasiannya.
- Mampu berorientasi kepada objek
- Mudah dikembangkan dengan menciptakan modul – modul baru, yang juga dapat ditulis dalam Python.

Akan tetapi, penulisan sintaks dalam Python juga harus diperhatikan agar tidak terjadi kesalahan saat eksekusi program. Termasuk penulisan nama variabel, indentasi, dan ketepatan dalam penulisan fungsi, ataupun sintaks – sintaks lainnya. Berikut adalah contoh penulisan syntax yang benar dalam bahasa Python :

```
#Inisialisasi fungsi dan prosedur
def func(param) : #param boleh kosong
    <algoritma>
    return <hasil> #apabila prosedur tidak perlu return

#Percabangan
if <cond1> :
    <then1>
elif <cond2> :
    <then2>
else :
    <then3>

#Pengulangan
for i in range(N) :
    <algoritma>
while <cond1> :
    <then1>

#iterasi tiap item pada list ataupun set
for item in <list atau set> :
    <algoritma>

#Assign value ataupun inisialisasi struktur data bertipe
collection kepada sebuah variabel
```

```
x = 5
y = 1.7
kata = "hello"
inilist = []
inilist2 = [0 for i in range(N)] #Agar langsung menginisialisasi
list dengan tipe elemen yang sama
inidict = {}
```

Selain itu untuk perbandingan, berikut adalah contoh dari beberapa kesalahan sintaks yang umum terjadi :

```
# Lupa indentasi saat inialisasi fungsi ataupun percabangan
def func(x) :
return x+4

if x > 0:
print("positif")

# Lupa menuliskan beberapa keyword, misalnya def saat inialisasi fungsi
func(x) :
return x+4

#Terdapat beberapa salah ketik atau salah penggunaan keyword
if x>0 :
    print("positif")
else if x <0 : # Seharusnya elif
    print("negatif")
else :
    print("nol").

#Penulisan print yang lupa diakhiri dengan tanda petik, atau jumlah kurung buka dan
kurung tutup yang tidak sesuai
print("Hello)
x = int(input("Masukkan bilangan x:"))
```

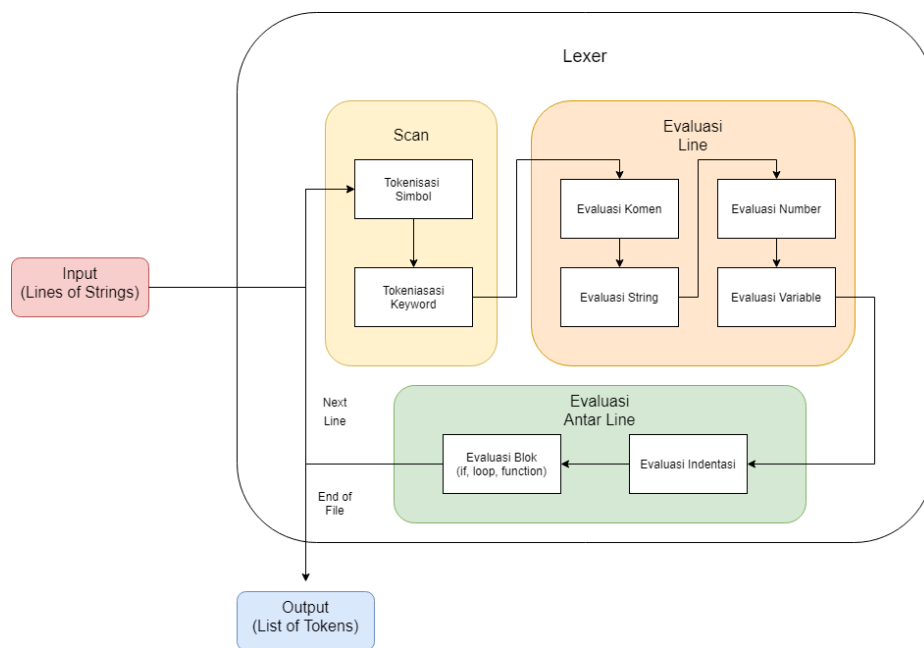
Dengan demikian, untuk mempermudah programmer menentukan ketelitian dan ketepatan saat penulisan program, diperlukan suatu parser untuk mengevaluasi apakah program yang diketik sudah sesuai dengan sintaks yang disediakan oleh Python. Salah satu parser sederhana untuk Python dapat diimplementasikan menggunakan grammar CFG yang nantinya akan dikonversi menjadi CNF untuk dapat dievaluasi menggunakan algoritma CYK sebagaimana prosesnya telah dijelaskan sebelumnya.

BAB II

HASIL DAN PEMBAHASAN

1) Lexical Analysis (Tokenization)

Proses Tokenization digunakan untuk merubah file input menjadi token yang dapat di proses menggunakan algoritma selanjutnya. Kami membagi proses tokenization menjadi 3 bagian besar yang akan dibagi menjadi proses proses kecil lagi. Untuk penggambaran proses tokenization yang kami implementasi, dapat dilihat pada diagram dibawah ini.



Gambar 2.1.1 Penggambaran implementasi proses tokenization

Penjelasan detail dari proses pada diagram dapat dilihat sebagai berikut.

- i. Tokenisasi Simbol
Memotong string berdasarkan simbol dan menyisipkan token diantara pemotongannya.
- ii. Tokenisasi Keyword
Mengganti seluruh string yang sesuai dengan keyword dengan token yang sesuai.
- iii. Evaluasi Komen
Melakukan penghapusan untuk seluruh elemen berdasarkan comment_flag yang akan diaktifkan oleh token TRIPLEQUOTE ("" dan """) dan token HASHTAG (#).
- iv. Evaluasi String
Melakukan penggantian seluruh elemen string yang di apit oleh token QUOTE (‘ dan “) menjadi token STRING (""") sesuai flag yang diaktifkan token tersebut.
- v. Evaluasi Number
Melakukan penggantian seluruh elemen string yang bersifat numerical, menjadi token NUMBER (0-9)
- vi. Evaluasi Variable
Melakukan penggantian seluruh elemen string yang memenuhi language FA yang telah dibuat.
- vii. Evaluasi indentasi

Menghitung indentasi kode dan meningkatkan indentasi berdasarkan flag `indent_flag` yang akan di-raise oleh token khusus yang menghasilkan blok kode (if, loop, function, class, dll.).

viii. Evaluasi blok

Meng-evaluasi token yang mengaktifkan flag blok (if, loop, function) dan token yang membutuhkan flag blok.

2) Finite Automata (FA)

Finite automata digunakan dalam pengecekan apakah nama variable yang berada dalam suatu assignment atau persamaan adalah sesuai dengan aturan.

Kami menentukan automata yang digunakan adalah Nondeterministic Finite Automata atau NFA untuk mempersingkat desain automata yang digunakan. Automata yang kami buat memiliki state sebanyak 2, yakni state q_1 sebagai start state yang hanya bisa menerima uppercase/lowercase letters ataupun underscore lalu menuju ke final state, dan state q_2 sebagai final state yang bisa menerima uppercase, lowercase, numerical, dan underscore menuju ke dirinya sendiri. Untuk input yang tidak sesuai dengan fungsi transisi akan membawa automata ke deadstate, sehingga nama variabel tidak bisa digunakan. Dengan demikian, kami mendefinisikan automata tersebut sebagai berikut.

$$A = (\{q_1, q_2\}, \Sigma, \delta, \{q_1\}, \{q_2\})$$

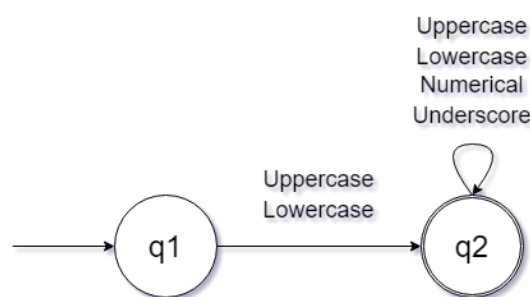
dimana,

$$\Sigma = \{a - z, A - Z, 0 - 9, _\}$$

$\delta =$

	Uppercase	Lowercase	Numerical	Underscore
$\rightarrow q_1$	q2	q2	-	-
*q2	q2	q2	q2	q2

Tabel 2.2.1 Fungsi transisi dari FA pengecekan nama variabel



Gambar 2.2.1 Diagram State dari FA pengecekan nama variabel

Contoh beberapa penamaan variabel setelah dievaluasi oleh FA:

- IniVariabel135 --> Accepted
- 135IniVariabel --> Rejected
- _Halogais --> Accepted
- V@riabel --> Rejected
- Ini_Variabel --> Accepted

3) Context – Free – Grammar (CFG)

Context Free Grammar diimplementasikan berdasarkan referensi grammar asli dari Python yang telah dimodifikasi sesuai kebutuhan. Syntax python secara umum terdiri dari statement dan expression. Expression memiliki precedence dalam CFG dan dapat dipakai dengan composing beberapa Test (yang juga dapat berupa atom dan operator biner serta aritmatika). Expression valid pada python terdiri atas:

A. Atom

Atom merupakan syntax fundamental terkecil pembentuk statement. Atom berisikan token literal (NAME, NUMBER, beberapa STRING, ELLIPSIS, NONE, TRUE, FALSE), list, set, atau dictionary. Atom dapat memiliki satu atau lebih trail berupa list subscription dan slicing (`x[0]`, `x[:]`, `dst`), atribut dari object (DOT NAME, seperti `x.y.z`), dan function call (`x()`, `x(bruh)`). Untuk assignment sendiri, atom hanya bisa berupa nama yang mungkin diikuti dengan beberapa trail berupa list subscription dan slicing serta atribut dari object saja.

B. Test

Test terdiri dari ekspresi logika yang juga berisi ternary (`and`, `or`, `not`, `x if y else z`) dan operator pembandingan (`<`, `<=`, `>`, `>=`, `==`, `!=`, `<>`, `in`, `not in`, `is`, `is not`).

C. Biner dan Aritmatika

Terdiri dari operator aritmatika (`**`, `+`, `-`, `*`, `@`, `/`, `%`, `//`) dan operasi biner (`>>`, `<<`, `&`, `^`, `|`, `~`)

D. List Subscription dan Slicing

Berupa ekspresi sebelum-sebelumnya yang mungkin kosong dan mungkin dipisahkan dengan maksimum dua buah titik dua di dalam kurung siku.

E. Assignment

Berupa ekspresi assignment (`=`) dan augmented assignment (operator aritmatika/biner sebelumnya ditambah assignment).

F. Spread operator

Terdiri dari star (`*`) untuk list atau double star (`**`) untuk dict.

Dalam program ini, grammar dimodifikasi sehingga produksi dimulai dengan new line, simple statement dengan newline, atau compound statement dengan newline, dan semua input dimasukkan per baris untuk performa terbaik. Statement terdiri atas Simple statement dan Compound statement:

A. Simple Statement

Simple statement terdiri dari beberapa small statement yang dipisahkan dengan titik koma dan mungkin diakhiri dengan titik koma. Small statement terdiri atas statement berikut:

- Expression
- Pass

- Control Flow (Break, Continue, Return, Raise)
- Import (Import Name, Import Name As Name, From Name Import Name)

B. Compound Statement

Compound statement terdiri dari:

- If (If, Elif, Else)
- While (While, While-Else)
- For (For-In, For-In-Else)
- With
- Function (Def)
- Class

Compound statement dapat diakhiri dengan simple statement.

Hasil dari Context Free Grammar yang telah kami buat terkait implementasi compiler bahasa Python didefinisikan sebagai tuple seperti berikut:

$$G = (V, \Sigma, S, P)$$

Di mana elemen dari tuple nya adalah :

i) **Variables (V)**

Berikut adalah tabel yang berisi semua simbol variabel yang dapat memiliki produksi juga dalam CFG

Start	AugAssign	CompOp	TermOp	ArithOp
FactorOp	ShiftOp	MString	Atom	DotEllipsis
DotName	DotAsName	DotAsNames	DotEllipsisName	DotEllipsisOneM
FromExpr	ImportAsName	MImportAsNames	ImportAsNames	ImportExpr
AtomTrail	MAtomTrail	AtomExpr	Power	Factor
Term	ArithExpr	ShiftExpr	XorExpr	Expr
Comparison	NotTest	AndTest	OrTest	Test
MTests	Tests	Comparison	NotTest	AndTest
OrTest	Test	MTest	Tests	StarExpr
ExprStmt	TestStarExpr	ExprStmt	TestStarExpr	TestsStarExpr
MTestsStarExpr	TestsStarExprStmt	EstarExpr	MESstarExpr	Exprs
DStarExpr	TestDStarExpr	MTestDStarExpr	Params	ClassParams
Args	MArgs	Arg	List	DictSet
Dict	DictMake	Set	SetMake	SubList
MSub	Sub	Slice	SliceTwo	CompList
CompFor	CompIf	Stmt	MStmt	SimpleStmt
MultiSimpleStmt	SmallStmt	Return	Raise	Flow
Import	ImportName	ImportFrom	Section	CompoundStmt
IfStmt	If	Elif	Else	While
For	With	WithExpr	WithItem	Class
Function				

ii) **Terminal Symbols (Σ)**

Berikut adalah tabel yang berisi semua simbol terminal yang menterminasi grammar CFG

NL	ADDA	SUBA	MULA	MATMULA
DIVA	MODA	BANDA	BORA	BXORA
BSHLA	BSHRA	POWA	IDIVA	LT
GT	EQ	GEQT	LEQT	NEQ
IN	IS	NOT	MUL	MATMUL
DIV	MOD	IDIV	ADD	SUB
TILDE	SHR	SHL	STRING	LP
RP	LB	RB	LC	RC
NAME	NUMBER	ELLIPSIS	NONE	TRUE
FALSE	DOT	AS	POW	BAND
BXOR	BOR	AND	OR	IF
ELSE	COMMA	COLON	ASSIGN	FOR
SEMICOLON	PASS	RETURN	RAISE	FROM
BREAK	CONTINUE	IMPORT	ELIF	WHILE
CLASS	DEF			

iii) **Start Symbol (S) : Start**

String atau Tokens yang dimasukkan hanya bisa accepted oleh sebuah context free language apabila masukan tersebut dapat dimulai dari variabel Start. Dengan demikian, maka variabel start harus berada pada level tertinggi pada tabel CYK yang menunjukkan bahwa string atau tokens secara penuh dapat dibentuk dan berasal dari start.

iv) **Rules of Productions (P)**

```
# First grammar line should be the start production
# New file input
Start -> NL | SimpleStmt NL | CompoundStmt NL

# Composed unit productions and it's recursive productions
# e.g. Zero or More ( $A^*$ ), One or More ( $A^+$ ), denoted by  $M^*$ 
AugAssign -> ADDA | SUBA | MULA | MATMULA | DIVA | MODA | BANDA | BORA |
BXORA | BSHLA | BSHRA | POWA | IDIVA
CompOp -> LT | GT | EQ | GEQT | LEQT | NEQ | IN | NOT IN | IS | IS NOT
TermOp -> MUL | MATMUL | DIV | MOD | IDIV
ArithOp -> ADD | SUB
FactorOp -> ArithOp | TILDE
ShiftOp -> SHR | SHL
MString -> STRING | MString STRING
Atom -> LP List RP | LB List RB | LC DictSet RC | NAME | NUMBER |
MString | ELLIPSIS | NONE | TRUE | FALSE
DotEllipsis -> DOT | ELLIPSIS
DotName -> NAME | DotName DOT NAME
DotAsName -> DotName | DotName AS NAME
DotAsNames -> DotAsName COMMA DotAsNames | DotAsName
DotEllipsisName -> DotEllipsis DotEllipsisName | DotName
DotEllipsisOneM -> DotEllipsis DotEllipsisOneM | DotEllipsis
FromExpr -> DotEllipsisName | DotEllipsisOneM
ImportAsName -> NAME | NAME AS NAME
```



```
MImportAsNames -> COMMA ImportAsName MImportAsNames | e
ImportAsNames -> ImportAsName MImportAsNames | ImportAsName
MImportAsNames COMMA
ImportExpr -> MUL | LP ImportAsNames RP | ImportAsNames

# Atomic Expression
AtomAssignTrail -> LB SubList RB | DOT NAME
MAtomAssignTrail -> AtomAssignTrail MAtomAssignTrail | e
AtomTrail -> Params | AtomAssignTrail
MAtomTrail -> AtomTrail MAtomTrail | e
AtomExpr -> Atom MAtomTrail
AtomAssignExpr -> NAME MAtomAssignTrail
Power -> AtomExpr | AtomExpr POW Factor
Factor -> FactorOp Factor | Power
Term -> Factor | Term TermOp Factor

# Binary and Arithmetic Expression
ArithExpr -> Term | ArithExpr ArithOp Term
ShiftExpr -> ArithExpr | ShiftExpr ShiftOp ArithExpr
AndExpr -> ShiftExpr | AndExpr BAND ShiftExpr
XorExpr -> AndExpr | XorExpr XOR AndExpr
Expr -> XorExpr | Expr BOR XorExpr

# Comparison and Logical Expression (Test)
# Also combine it with Binary and Arithmetic Expression
Comparison -> Expr | Comparison CompOp Expr
NotTest -> NOT NotTest | Comparison
AndTest -> NotTest | AndTest AND NotTest
OrTest -> AndTest | OrTest OR AndTest
Test -> OrTest | OrTest IF OrTest ELSE Test
MTests -> COMMA Test | e
Tests -> Test MTests | Test MTests COMMA

# Assignment (+Augmented) Expression
# Support for inline expression (test, incl. bin & arith)
# and also star (spread) expression
StarExpr -> MUL Expr
ExprStmt -> AtomAssignExpr AugAssign Tests | AssignExpr
TestStarExpr -> Test | StarExpr
TestsStarExpr -> TestStarExpr MTestsStarExpr | TestStarExpr
MTestsStarExpr COMMA
MTestsStarExpr -> COMMA TestStarExpr MTestsStarExpr | e
AssignExpr -> AtomAssignExpr ASSIGN AssignExpr | Tests

# Multiple Expressions
# Including Star Expression
EStarExpr -> Expr | StarExpr
MESTarExpr -> COMMA EStarExpr MESTarExpr | e
Exprs -> EStarExpr MESTarExpr | EStarExpr MESTarExpr COMMA

# Spread Operator for Dict (**)
DStarExpr -> POW Expr
TestDStarExpr -> Test COLON Test | DStarExpr
```

```

MTestDStarExpr -> COMMA TestDStarExpr MTestDStarExpr | e

# Arguments and Parameters
Params -> LP RP | LP Args RP
Args -> Arg | Arg MArgs | Arg MArgs COMMA
MArgs -> COMMA Args MArgs | e
Arg -> Test | Test CompFor | NAME ASSIGN Test | POW NAME | MUL NAME
# Differentiate Class/Func Define Arguments
# It does not accept CompFor and Test as its args (only name allowed)
ArgsDef -> ArgDef | ArgDef MArgsDef | ArgDef MArgsDef COMMA
MArgsDef -> COMMA ArgsDef MArgsDef | e
ArgDef -> NAME | NAME ASSIGN Test | POW NAME | MUL NAME
ParamsDef -> LP RP | LP ArgsDef RP
ClassParams -> ParamsDef | e

# List/Dict/Set Definition
List -> TestStarExpr CompFor | TestsStarExpr | e
DictSet -> Dict | Set | e
Dict -> TestDStarExpr DictMake
DictMake -> CompFor | MTestDStarExpr | MTestDStarExpr COMMA
Set -> TestStarExpr SetMake
SetMake -> CompFor | MTestsStarExpr | MTestsStarExpr COMMA

# List Subscription (Indexing) and Slicing
SubList -> Sub MSub | Sub MSub COMMA
MSub -> COMMA Sub MSub | e
# Subscription and Slicing Type:
# 1. Indexing (e.g. x[Test])
# 2. 1D Slicing (e.g. x[:, x[Test:], x[:,Test], x[Test:Test])
# 3. 2D Slicing (e.g. x[A:B:], x[A:B:Test])
Sub -> Test | Slice | Slice SliceTwo
Slice -> COLON | Test COLON | COLON Test | Test COLON Test
SliceTwo -> COLON | COLON Test

# Comprehension
CompList -> CompFor | CompIf
CompFor -> FOR Exprs IN OrTest | FOR Exprs IN OrTest CompList
CompIf -> IF OrTest | IF OrTest CompList

# Basic building block
# One Statement, could be Simple or Compound
Stmt -> SimpleStmt | CompoundStmt

# 1. Simple Statement: Small Statement OR Multiple Small Statement
separated by semicolon
# Also support trailing semicolon
SimpleStmt -> SmallStmt | SmallStmt SEMICOLON | SmallStmt
MultiSimpleStmt | SmallStmt MultiSimpleStmt SEMICOLON
MultiSimpleStmt -> SEMICOLON SmallStmt MultiSimpleStmt | e
SmallStmt -> ExprStmt | PASS | Flow | Import
# Small Statement Types
# A. Expressions in Statement
# B. Pass

```

```
# C. Control Flow (Break, Continue, Return, Raise)
Return -> RETURN | RETURN Tests
Raise -> RAISE | RAISE Test | RAISE Test FROM Test
Flow -> BREAK | CONTINUE | Return | Raise
# D. Import (Import, From Import)
Import -> ImportName | ImportFrom
ImportName -> IMPORT DotAsNames
ImportFrom -> FROM FromExpr IMPORT ImportExpr

# 2. Compound Statement: If/While/For/With/Func Statement
# Each consist of one Section that could be either:
# - One simple statement in the same line
# - Multiple Any Statement in different line (min. 1)
Section -> SimpleStmt | e
CompoundStmt -> IfStmt | While | For | With | Func | Class
# Compound Statement Types
# A. If
IfStmt -> If | Elif | Else
If -> IF Test COLON Section | IF Test COLON Section
Elif -> ELIF Test COLON Section
Else -> ELSE COLON Section
# B. While
While -> WHILE Test COLON Section | WHILE Test COLON Section ELSE COLON
Section
# C. For
For -> FOR Exprs IN Tests COLON Section | FOR Exprs IN Tests COLON
Section ELSE COLON Section
# D. With
With -> WITH WithExpr COLON Section
WithExpr -> WithItem | WithExpr COMMA WithItem
WithItem -> Test | Test AS Expr
# F. Class
Class -> CLASS NAME ClassParams COLON Section
# G. Function
Func -> DEF NAME ParamsDef COLON Section
```

BAB III IMPLEMENTASI DAN PENGUJIAN

1) Spesifikasi Teknis Program

Kami mengimplementasi seluruh fungsi kami buat dalam bentuk modul bernama 'tbfo' yang dibagi menjadi 2 submodul yaitu, lexer dan parser, ditambah dengan program utama. Dimana lexer berisikan modul lexer.py dan konfigurasi keywords.txt dan parser berisikan fa.py dan cyk.py

Kami membungkus seluruh modul dalam bentuk class sesuai dengan fungsionalitasnya. Kami merasa dengan mengimplementasi OOP, akan membuat program lebih rapih dan jelas bagiannya. Kami mengimplementasikan 5 class, dengan penjelasan tiap class sebagai berikut :

a) Class Variable Checker

Merupakan class yang setara dengan Finite Automata. Mengimplementasi sistem NFA, mulai dari definisi formalnya hingga proses penerimaan languagenya.

- i. Attributes
 - 1.startState
Berisikan state awal dari NFA.
 - 2.finalState
Berisikan final state dari NFA.
 - 3.state
Berisikan seluruh state yang ada di NFA.
 - 4.sym
Berisikan seluruh symbol/karakter yang diterima NFA
 - 5.delta
Berisikan seluruh transisi antar state sesuai dengan symbol yang diterima NFA.
- ii. Methods
 - 1.Constructor
Merupakan metode konstruktor yang menetapkan seluruh atribut quintuple dari NFA ke dalam object.
 - 2.check(inp) -> boolean
Menerima suatu string yang akan dievaluasi menggunakan atribut quintuple NFA objek. Mengembalikan True jika string merupakan bahasa yang diterima NFA, dan mengembalikan False jika tidak.

b) Class Symbol

Merupakan class token yang digunakan untuk membedakan string masukan dengan token yang sudah di proses oleh Lexer.

- i. Attributes
 - 1.name
Berisikan nama dari token, con: "SPACE" atau "SEMICOLON".
 - 2.symbol
Berisikan simbol dari token con: " " atau " ;".
- ii. Methods
 - 1.Constructor
Merupakan metode konstruktor yang menetapkan masukan nama dan simbol ke dalam objek.

c) Class Lexer

Merupakan class Lexer yang digunakan untuk merubah input string menjadi list of tokens yang akan diproses selanjutnya oleh class lain.

- i. Attributes
 - 1.symbols, keywords, whitespace, dan konstanta lainnya.
berisikan simbol yang digunakan untuk menggantikan string yang sesuai dengan objek simbol yang sesuai
 - 2.varCheck
merupakan objek variable checker untuk melakukan check kepada nama variable yang ada.
 - 3.flags
merupakan flag yang digunakan untuk mengunifikasi token menjadi satu, seperti komen, string, dan number.
- ii. Methods
 - 1.lex(line) -> List of Symbol
Merupakan metode untuk lexical analysis terhadap satu baris string. Tahapan proses ini telah dijelaskan pada bab sebelumnya.
 - 2.lex_lines(lines) -> List of Symbol
Merupakan metode untuk lexical analysis terhadap list of string. Tahapan proses ini telah dijelaskan pada bab sebelumnya.

d) Class CFG

Merupakan Class CFG yang digunakan untuk merubah grammar CFG ke dalam bentuk Chomsky Normal Form sehingga bisa dilakukan algoritma CYK.

- i. Attributes
 - 1.variables
Merupakan list dari variable (non-terminal) yang berada di objek CFG
 - 2.terminals
Merupakan list dari terminal yang berada di grammar
 - 3.productions
Merupakan list of productions yang berada di objek CFG
 - 4.start_variables
Merupakan start variable yang berada di objek CFG
- ii. Methods
 - 1.to_cyk()
Merupakan metode untuk mengkonversi grammar yang sudah dalam bentuk CNF ke format yang sesuai untuk diterima oleh metode cyk. Metode ini mengembalikan hasil produksinya berupa dictionaries yang memiliki key pemproduksi dan memiliki itemnya list yang berisi hasil produksi. Struktur data disesuaikan sebagai dictionary agar setelah mendapatkan items yang sesuai, maka langsung dapat dipetakan pada dictionary untuk mengoptimasi proses searching saat metode cyk.
 - 2.to_cnf()
Merupakan metode untuk mengkonversi grammar CFG yang sudah diperoleh menjadi bentuk CNF nya dengan mengatasi null productions, useless productions, dan unit productions nya. Kemudian, metode ini akan mengubah productions yang memiliki hasil produksi lebih dari 2 atau memiliki 2 atau lebih terminals dengan

mensubstitusi nya dengan variabel baru lalu menambahkan rule baru yang bersesuaian tersebut.

3. `sanitize(lines)`

Merupakan metode untuk mensanitasi baris grammar dari sebuah text files.

Dengan kata lain, metode ini mengabaikan komentar, baris kosong, dan adanya trailing line breaks.

4. `parse(prod_lines)`

Merupakan metode untuk melakukan parsing terhadap list of string agar menjadi format grammar yang tepat. List of string harus di sanitasi terlebih dahulu dengan fungsi sebelumnya (tanpa comment tanpa empty line, tanpa trailing linebreaks).

e) Class CYK

Merupakan class CYK yang digunakan untuk melakukan parsing dari sebuah list of tokens yang merupakan hasil proses dari class Lexer dan mengecek apakah tokens tersebut diterima oleh bahasa yang dibuat oleh class CFG.

i. Attributes

1. `grammar`

Merupakan grammar CFG yang didapat dari class CFG

2. `terminals`

Merupakan list yang isinya adalah production yang menghasilkan terminal

3. `variables`

Merupakan dictionary yang isinya adalah hasil dari production yang menghasilkan variabel, digunakan saat hashing untuk proses pengisian tabel saat parsing.

4. `verbose`

Berisikan boolean yang mengalihkan percetakan hasil tabel dari hasil parsing.

ii. Methods

1. Constructor

Merupakan metode konstruktor yang menetapkan masukan grammar, terminals, `variable_cache`, dan `verbose` kepada object.

2. `cross_product(left,right) -> list of list`

Merupakan metode untuk melakukan konkatenasi antara variabel subsequent string kiri dan variabel subsequent string kanan dalam bentuk list dan mengembalikan list yang berisi hasil tersebut.

3. `parse(self, tokens) -> boolean`

Merupakan metode untuk melakukan parsing dari sebuah list of tokens dan mengembalikan boolean yang mengecek apakah hasilnya diterima oleh grammar atau tidak. Tahapan proses ini telah dijelaskan pada bab sebelumnya.

2) Uji Kasus dan Analisis

a) tes1.py

```

tests > tes1.py > ...
1  a = int(input("Masukkan X: "))
2
3  if a < 0 :
4      print("X adalah bilangan negatif")
5  else:
6      print("X adalah bilangan nol")
7  else:
8      if a%2 == 0:
9          print("X adalah bilangan positif genap")
10         else :
11             print("X adalah bilangan positif ganjil")

```

```

Syntax Error in line: 7
Invalid Syntax
"else:"
Time: 2.09 ms
PS D:\Kuliah\SEM 3\Teori Bahasa Formal dan Otomata (TBFO)\Python-Grammar-Checker>

```

Gambar 3.2.1 dan 3.2.2 Hasil eksperimen uji kasus 1

Dalam gambar di atas, kami menampilkan hasil parsing menggunakan program kami untuk uji kasus yang pertama. Dari hasil screenshot tersebut, hasil yang didapat sesuai dengan hasil yang diharapkan, yaitu "Syntax Error in Line: 7" dan "Invalid syntax: Else". Pada percobaan ini, kami berusaha mengevaluasi apakah grammar berhasil mendeteksi sintaks error adanya keyword ELSE yang tidak didahului oleh keyword IF. Pada CFG yang telah kami buat sudah mendefinisikan rules tertentu sehingga ELSE harus didahului oleh sebuah IF dan saling berkaitan satu sama lain dan memiliki semantik tertentu. Untuk sintaks sebelum dan setelah line 7 tidak dideteksi kesalahan, karena sebelumnya percabangan dilakukan dengan benar.

b) tes2.py

```

tests > tes2.py > [x]
1  x = 5
2  list_x = [1,2,3]
3  yeah = "#ini comment di dalem string"
4  # "Ini string di dalem comment"
5  list_x[True or False | x and 5 + 1 if x+1 == 3 else print(5)] = list_x[...]
6  6ix9nine = "n-word"

```

```

Syntax Error in line: 6
Invalid variable : 6ix9nine.
"6ix9nine = "n-word""
Time: 4.50 ms
PS D:\Kuliah\SEM 3\Teori Bahasa Formal dan Otomata (TBFO)\Python-Grammar-Checker>

```

Gambar 3.2.3 dan 3.2.4 Hasil eksperimen uji kasus 2

Dalam gambar di atas, kami menampilkan hasil parsing menggunakan program kami untuk uji kasus yang kedua. Dari hasil screenshot tersebut, hasil yang didapat sesuai dengan hasil yang diharapkan, yaitu "Syntax Error in Line: 6" dan "Invalid Variable : 6ix9nine". Disini program berhasil membedakan comment di dalam string dan string di dalam comment, dan indexing yang sedikit

ekstrim. Program juga terlihat mampu melakukan pengecekan nama variabel menggunakan konsep Finite Automata, terbukti bahwa terdeteksinya sebuah variabel yang secara konvensi tidak sesuai dengan penamaan variabel dalam Python.

c) tes3.py

```
tests > tes3.py > x > x
1  with x:
2      print(yeah)
3      class x:
4          if (x == 5):
5              puts("this is python")
6          else:
7              def x(yeah):
8                  print(yeah)
```

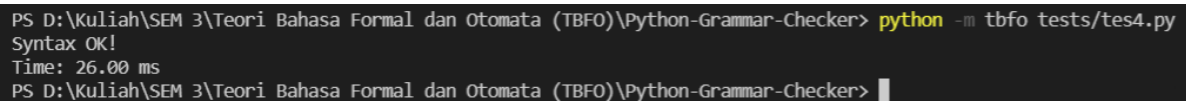
```
Syntax Error in line: 8
Indentation Error
"print(yeah)"
Time: 1.97 ms
PS C:\Users\rifqi\Data\File Kuliah\Sem 3\TBFO\Tubes\Python-Grammar-Checker> []
```

Gambar 3.2.5 dan 3.2.6 Hasil eksperimen uji kasus 3

Dalam gambar di atas, kami menampilkan hasil parsing menggunakan program kami untuk uji kasus yang ketiga. Dari hasil screenshot tersebut, hasil yang didapat sesuai dengan hasil yang diharapkan, yaitu "Syntax Error in line: 8", dengan detail Indentation error dikarenakan harus meningkatkan indentasi setelah pemanggilan keyword def. Disini program dapat mengenali indentasi dan jenis block of code yang ada di python.

d) tes4.py

```
tests > tes4.py > ...
1  def do_something(x):
2      ''' This is a sample multiline comment
3      ...
4      if x == 0 + 1:
5          return 0
6      elif x + 4 == 1:
7          return None
8      elif x:
9          print("a")
10     else:
11         return 2
12         """
13     elif x == 32:
14         return 4
15     else:
16         return "Doodoo"
17         """
18     def bruh(x):
19         x[True or False if 1 == 2 else 3 and 4:1+None+3 and 5:None and False or True] = 5
20
```

```
PS D:\Kuliah\SEM 3\Teori Bahasa Formal dan Otomata (TBFO)\Python-Grammar-Checker> python -m tbfo tests/tes4.py
Syntax OK!
Time: 26.00 ms
PS D:\Kuliah\SEM 3\Teori Bahasa Formal dan Otomata (TBFO)\Python-Grammar-Checker>
```

Gambar 3.2.7 dan 3.2.8 Hasil eksperimen uji kasus 1

Dalam gambar di atas, kami menampilkan hasil parsing menggunakan program kami untuk uji kasus yang keempat. Dari hasil screenshot tersebut, hasil yang didapat sesuai dengan hasil yang diharapkan, yaitu “Syntax OK!” yang berarti compile berhasil. Disini program berhasil mengabaikan multiline comment dan if else statement biasa.

BAB IV

KESIMPULAN DAN SARAN

1) Kesimpulan

Pada tugas besar mata kuliah IF2124 Teori Bahasa Formal dan Otomata ini, telah berhasil diimplementasikan hasil pembelajaran Finite Automata dan Context Free Grammar untuk membuat sebuah compiler bahasa Python dan pengecekan nama variabel dalam Python.

Compiler bahasa Python yang kami buat ini dilakukan dengan melakukan leksikal analisis terlebih dahulu dari tiap baris masukan menjadi tokens, kemudian menggunakan Algoritma Cocke-Younger-Kasami dengan Context – Free – Grammar yang telah dibuat untuk melakukan parsing terhadap list of tokens yang telah diperoleh tadi. Dengan demikian, dapat ditentukan apakah masukan memiliki sintaks yang salah ataupun tidak. Untuk pengecekan nama variabel, cukup dengan membuat Finite Automata yang menyimpan state dan fungsi transisi untuk tiap karakter nama variabelnya.

Dari program yang telah kami buat, terlihat bahwa program dapat berjalan cukup baik mengingat keterbatasan waktu ataupun kendala lainnya terkait pembuatan compiler bahasa Python. Namun, masih terdapat beberapa value yang tidak disupport oleh grammar yang telah dibuat karena memang tidak menampung keyword – keyword tersebut. Dengan demikian, kelompok dapat menyimpulkan bahwa dengan mengerjakan Tugas Besar IF2124 Teori Bahasa Formal dan Otomata Semester 1 Tahun 2021/2022 ini, dapat diketahui bahwa kegiatan pengecekan sintaks dan nama variabel dari masukan file Python dapat dilakukan dengan menerapkan konsep Finite Automata dan Context Free Grammar.

2) Saran

Hasil tugas ini dapat dijadikan lebih baik dengan penambahan beberapa value yang penting dalam python dikarenakan terbatasnya waktu, seperti: keyword `async/await`, multiline continuation (argumen tidak dalam satu baris), multiline string, dan lain-lain. Selain itu, lebih baik apabila intensitas daripada uji kasus diperbesar karena Python memiliki syntax yang sangat luas sehingga perlu pengujian tiap kasus sehingga tidak ada yang terlewat. Agar pengujian dapat dilakukan secara lebih akurat, lebih baik apabila pengujian dilakukan secara step by step. Juga agar compiler kami mencakup lebih banyak lagi kasus uji yang sebelumnya disupport, grammar dapat dikembangkan lagi agar semakin menyerupai compiler Python yang asli.

BAB V

PENUTUP

Link To Repository

Link to Github Repository : <https://github.com/rifqi2320/Python-Grammar-Checker>

Pembagian Tugas

Pembagian Tugas		
Nama	NIM	Tugas
Rifqi Naufal Abdjul	13520062	Lexer dan FA
Saul Sayers	13520094	Algoritma CYK
Amar Fadil	13520103	Grammar CFG dan Algoritma CFG to CNF

Referensi

The Python Language Reference for Python 3.6 (2021, 5 September). Diakses pada 21 November, 2021. Dari <https://docs.python.org/3.6/reference/>

TELAAH TEORITIS FINITE STATE AUTOMATA DENGAN PENGUJIAN HASIL PADA MESIN OTOMATA (2011, 1 Januari). Diakses pada 22 November, 2021. Dari <https://media.neliti.com/media/publications/226255-telaah-teoritis-finite-state-automata-de-8b056b07.pdf>

CYK Parsing over Distributed Representations (2020, 5 Oktober). Diakses pada 22 November, 2012. Dari <https://www.mdpi.com/1999-4893/13/10/262/htm>

Generating all permutations by context-free grammars in Chomsky normal form (2006, 21 Maret). Diakses pada 23 November 2021. Dari <https://doi.org/10.1016/j.tcs.2005.11.010>

John E. Hopcroft; Rajeev Motwani; Jeffrey D. Ullman (2003). *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley