

Laporan Tugas Besar
Tugas Besar IF2124 Teori Bahasa Formal dan Otomata
Parser Bahasa JavaScript (Node.js)

Oleh

Michael Jonathan Halim / 13521124

Raynard Tanadi / 13521143

Johanes Lee / 13521148



Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
INSTITUT TEKNOLOGI BANDUNG
2022

DAFTAR ISI

DAFTAR ISI	1
BAB I	2
A. Finite Automata (FA)	2
B. Context Free Grammar (CFG)	2
C. CFG To CNF	3
D. Cocke-Younger-Kasami (CYK)	4
E. Syarat Syntax JavaScript	4
BAB II	20
A. Finite Automata (FA)	20
a. FA untuk mengecek apakah suatu ekspresi valid atau tidak	20
b. FA untuk mengecek apakah suatu variabel valid atau tidak	24
B. Context Free Grammar (CFG)	26
BAB III	37
A. Spesifikasi Teknis Program	37
a. Gambaran Umum	37
b. Struktur Data	37
c. Fungsi	38
d. Prosedur	47
e. Antarmuka	47
B. Screenshot dan Analisis Hasil	48
1. Kasus 1	48
2. Kasus 2	49
3. Kasus 3	50
4. Kasus 4	51
5. Kasus 5	51
6. Kasus 6	52
7. Kasus 7	53
8. Kasus 8	54
9. Kasus 9	54
10. Kasus 10	55
BAB IV	57
A. Link Repository Github	57
B. Pembagian Tugas	57

BAB I

TEORI DASAR

A. Finite Automata (FA)

Finite Automata atau Finite State Automata merupakan sistem model matematika dengan masukan dan keluaran diskrit yang dapat mengenali bahasa reguler. Finite Automata juga memiliki lima elemen atau tuple, yaitu input, output, himpunan state, relasi state, dan relasi output. Finite Automata sendiri memiliki state yang berhingga jumlahnya yang dapat diterima melalui input maupun dikeluarkan melalui output dan dapat berpindah dari satu state ke state lainnya jika sesuai dengan aturan yang ada. Perubahan state tersebut dinamakan fungsi transisi. Akan tetapi, Finite Automata tidak mempunyai tempat penyimpanan sehingga hanya dapat mengingat state terbaru saja. Pada saat transisi, automata tidak harus berpindah ke keadaan selanjutnya, melainkan dapat tetap di keadaan yang sama. Hal ini bergantung pada fungsi transisi yang ada. Keluaran atau status dari suatu Finite Automata sendiri hanya ada dua, yaitu 'Terima' jika string input dapat diproses dan automata mencapai state akhir atau 'Tolak' jika kebalikannya. Finite Automata sendiri dibagi menjadi dua jenis, yaitu Deterministic Finite Automata (DFA) dan Non-Deterministic Finite Automata (NFA). Terakhir, Finite Automata dapat didefinisikan dengan persamaan berikut :

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

Dimana :

- Q = himpunan state
- Σ = himpunan simbol input
- δ = fungsi transisi $\delta : Q \times \Sigma$
- S = state awal / initial state , $S \in Q$
- F = state akhir, $F \subseteq Q$

B. Context Free Grammar (CFG)

Context Free Grammar atau yang biasa adalah tata bahasa formal yang memiliki aturan produksi berupa $S \rightarrow \alpha$ dengan S merupakan pemroduksi dan simbol non-terminal atau variabel tunggal, sedangkan α adalah hasil produksi dan merupakan terminal

dan/atau variabel dan/atau ϵ . Context Free Grammar sendiri memiliki 4 elemen atau tuple, yaitu terminal, non-terminal/variabel, start symbol, dan aturan produksi sehingga Context Free Grammar dapat didefinisikan sebagai berikut :

$$G = (V, T, P, S)$$

Dimana :

- G = Grammar
- V = Himpunan non-terminal/variabel
- T = Himpunan terminal
- P = Himpunan aturan produksi
- S = Start symbol

Context Free Grammar sendiri dapat disederhanakan untuk melakukan pembatasan. Penyederhanaan ini dapat dilakukan dengan tiga langkah, yaitu eliminasi ϵ -production, eliminasi unit production, dan eliminasi useless symbol.

C. CFG To CNF

Chomsky Normal Form atau yang biasa disingkat menjadi CNF adalah salah satu bentuk normal untuk Context Free Grammar atau CFG. CFG harus disederhanakan terlebih dahulu dengan tiga langkah, yaitu eliminasi ϵ -production, eliminasi unit production, dan eliminasi useless symbol supaya CFG tidak memiliki produksi useless, produksi unit, dan produksi ϵ . CNF juga terdapat aturan tambahan, yaitu hasil produksi(ruas kanan) harus dalam bentuk 2 variabel atau 1 terminal, seperti :

- $S \rightarrow AC$
- $A \rightarrow a$
- $B \rightarrow b$
- $C \rightarrow AB \mid c$

Konversi CFG ke CNF secara lengkap sendiri dapat dilakukan dengan 4 langkah berikut :

1. Mengeliminasi ϵ -production
2. Mengeliminasi unit production
3. Mengeliminasi useless symbol
4. Ubah hasil produksi(ruas kanan) menjadi dalam bentuk 2 variabel atau 1 terminal, contoh :

- $S \rightarrow cAB \mid ab$
- $A \rightarrow aB \mid a \mid BA$
- $B \rightarrow AS \mid ab$

Menjadi :

- $S \rightarrow EF \mid CD$

- $A \rightarrow CB \mid a \mid BA$
- $B \rightarrow AS \mid CD$
- $C \rightarrow a$
- $D \rightarrow b$
- $E \rightarrow c$
- $F \rightarrow AB$

D. Cocke-Younger-Kasami (CYK)

Algoritma CYK adalah sebuah algoritma parsing yang menggunakan pemrograman dinamis dan hanya dapat bekerja jika CFG diberikan dalam bentuk CNF. Algoritma CYK sendiri merupakan salah satu algoritma yang dapat mengenali sebuah string dalam bahasa tertentu. Implementasi CYK dalam program ini menggunakan konsep dynamic programming untuk menentukan sebuah string di dalam suatu language grammar atau tidak. Kompleksitas dari CYK adalah $O(n^3 \cdot |G|)$ di mana $|G|$ adalah jumlah dari rule grammar yang telah dibuat. Suatu string akan diterima apabila starting symbol terdapat pada kotak baris ke - 1 dan kolom terakhir.

	b	a	a	b	a
b	{B}	{S,A}	ϕ	ϕ	{S,A,C}
a		{A,C}	{B}	{B}	{S,A,C}
a			{A,C}	{S,C}	{B}
b				{B}	{S,A}
a					{A,C}

Contoh Tabel CYK

E. Syarat Syntax JavaScript

❖ Break

Keyword break digunakan untuk melakukan exit dari suatu looping baik itu for loop dan while loop, juga untuk switch case. Jika break digunakan di luar loop dan switch case, maka akan dihasilkan *syntax error*.

Contoh Accepted :

```
while(true){  
    check = true;  
    if (check) {  
        break;  
    }  
}
```

Contoh Syntax Error :

```
check = true;  
if (check) {  
    break;  
}
```

Kode di atas menghasilkan *syntax error* karena `break` hanya bisa dilakukan dalam perulangan dan `switch case`.

❖ Const, Var, Let

Keyword `const`, `var`, dan `let` digunakan untuk melakukan deklarasi variabel dalam javascript. `Const` digunakan untuk mendeklarasikan sebuah konstanta yang nilainya tidak bisa diubah dan harus diinisiasi terlebih dahulu nilainya. `Var` digunakan untuk mendeklarasikan sebuah variabel dalam sebuah function scope (secara *semantics*). `Let` digunakan untuk mendeklarasikan sebuah variabel dalam sebuah block scope (secara *semantics*). Secara *syntax*, deklarasi menggunakan `const`, `var`, dan `let` dapat digunakan di dalam semua blok kode asalkan deklarasi memungkinkan.

Contoh Accepted :

```
let x = 3, y;
```

Contoh Syntax Error :

```
const x;
```

Kode di atas *syntax error* karena `const` harus diinisiasi dengan sebuah nilai.

❖ Switch, Case, Default

Keyword `switch` dan `case` digunakan untuk *control flow* dalam javascript. `Switch` dan `case` membutuhkan sebuah ekspresi sebagai argumen yang akan

digunakan untuk ekspresi kondisi di dalam case. Dalam switch case, bisa digunakan *keyword* default yang fungsionalitasnya mirip seperti else, yaitu menerima semua kondisi yang tidak memenuhi semua kondisi case.

Contoh Accepted :

```
switch(x){  
    case 1:  
        x += 1;  
        break;  
    default:  
        x = 0;  
        break;  
}
```

Contoh Syntax Error :

```
switch(){  
    case 1:  
        x += 1;  
        break;  
    default:  
        x = 0;  
        break;  
}
```

Kode di atas menghasilkan *syntax error* karena switch memerlukan sebuah argumen untuk ekspresi kondisi di case.

❖ Try, Catch, Finally

Keyword try, catch, dan finally digunakan untuk mengantisipasi kasus ketika kode tidak berjalan sesuai ekspektasi sehingga jika terjadi kesalahan, alur program akan berpindah ke *block* catch. Blok kode setelah *keyword* finally akan tetap dijalankan ketika terjadi *error* ataupun tidak. Blok kode yang mengikuti ketiga keyword ini harus diapit *curly bracket*. Selain itu, harus ada setidaknya *statement* catch ataupun *statement* finally setelah *statement* try.

Contoh Accepted :

```
try {  
    x += 2;  
} catch {  
    x -= 2;  
} finally {  
    x *= 2;  
}
```

Contoh Syntax Error :

```
try {  
    x += 2;  
}
```

Kode di atas syntax error karena try harus berpasangan dengan setidaknya catch ataupun finally .

❖ Continue

Keyword continue digunakan untuk menghentikan satu iterasi (dalam loop), lalu melanjutkan iterasi berikutnya.

Contoh Accepted :

```
for(let i = 0; i < 4; i++){  
    if(i == 1){  
        continue;  
    }  
}
```

Contoh Syntax Error :

```
if(i == 1){  
    continue;  
}
```

Kode di atas menghasilkan *syntax error* karena continue hanya dapat digunakan di dalam perulangan.

❖ Delete

Operator delete digunakan untuk menghapus properti dari objek. Delete akan menghapus nilai properti dan properti itu sendiri. Properti yang dihapus tidak dapat digunakan lagi sebelum ditambahkan. Operasi delete dapat digunakan pada semua ekspresi (secara *syntax*) tetapi hanya memiliki makna *semantics* ketika dilakukan pada properti objek.

Contoh Accepted :

```
delete object.property;
```

Contoh *Syntax Error*:

```
delete while (false) {  
    console.log(error);  
};
```

Kode di atas menghasilkan *syntax error* karena operator delete hanya dapat diikuti sebuah ekspresi.

❖ If, Else

Statement if digunakan untuk menentukan blok kode yang akan dieksekusi jika suatu kondisi terpenuhi. Selain itu, *keyword* else digunakan untuk menentukan blok kode yang akan dieksekusi jika tidak ada kondisi yang terpenuhi. Juga terdapat *keyword* else if yang digunakan untuk menentukan blok kode yang berjalan pada kondisi selain kondisi pertama dan memungkinkan mendeklarasikan lebih dari dua kondisi. *Statement* else if dan else harus didahului *statement* if sebelumnya. Ketiga *keyword* tersebut juga dapat diikuti suatu blok kode di dalam kurung kurawal ataupun satu baris kode yang diakhiri titik tiga.

Contoh Accepted :

```
if(i == 1){  
    x = 2;  
} else{  
    x = 1;
```

```
}
```

Contoh Syntax Error :

```
else{  
    x = 1;  
}
```

Kode di atas menghasilkan *syntax error* karena *statement* else harus didahului dengan *statement* if.

❖ True, False

True dan false adalah nilai kebenaran yang bertipe boolean di dalam javascript. True dan false bisa ditulis sebagai suatu nilai dan bisa dihasilkan dari operasi *logical*.

Contoh Accepted :

```
x = true;
```

❖ For

For statement digunakan untuk mengulang blok kode beberapa kali. Untuk menggunakan keyword for, dibutuhkan 3 ekspresi. Ekspresi 1 dieksekusi satu kali sebelum eksekusi blok kode. Ekspresi 2 berguna untuk mendefinisikan kondisi untuk mengeksekusi blok kode. Terakhir, ekspresi 3 dijalankan setiap kali setelah blok kode dieksekusi. Ekspresi pertama dapat juga digantikan dengan suatu deklarasi.

Contoh Accepted :

```
for(let i = 0; i < 4; i++){  
    if(i == 1){  
        continue;  
    }  
}
```

Contoh Syntax Error :

```
for(let i = 0){  
    if(i == 1){  
        continue;  
    }  
}
```

```
}  
}
```

Kode di atas syntax error karena for harus memiliki 3 ekspresi.

❖ Function

Keyword function diikuti blok kode yang dirancang untuk melakukan suatu tugas tertentu. Function akan dijalankan ketika dipanggil. Function didefinisikan dengan *keyword* function, nama function, tanda buka kurung, parameter yang dibutuhkan (dapat kosong atau berjumlah lebih dari 1), tutup kurung, dan kode yang akan dieksekusi akan ditulis di dalam kurung kurawal. Kode yang akan dieksekusi juga dapat berupa 1 baris yang diakhiri titik koma. Berikut merupakan contoh pendefinisian function :

```
function name(parameter1, parameter2, parameter3){  
  // ini isi kode  
}
```

Nama function dapat terdiri atas huruf, angka, garis bawah, dan tanda dolar. Untuk penulisan parameter, setiap elemen parameter harus dipisahkan dengan tanda koma. Variabel parameter di dalam function akan berlaku sebagai variabel lokal.

Contoh Accepted :

```
function myFunction(a, b){  
  return a * b;  
}
```

Contoh Syntax Error :

```
function (a, b){  
  return a * b;  
}
```

Kode di atas syntax error karena function harus memiliki nama jika didefinisikan.

❖ Null

Null merupakan suatu keyword yang merepresentasikan nilai yang tidak diketahui atau kosong.

Contoh Accepted :

```
const foo = null;
```

❖ Return

Keyword `return` digunakan untuk menghentikan eksekusi fungsi dan mengembalikan nilai. `Return` sendiri hanya dapat digunakan di dalam function. `Return` dapat diikuti ataupun tidak diikuti ekspresi (jika ekspresi kosong, maka dikembalikan nilai *undefined*).

Contoh Accepted :

```
function myFunction(a){  
    return "Hello " + a;  
}
```

Contoh Syntax Error :

```
return;
```

Kode di atas syntax error karena `return` hanya boleh dilakukan di dalam function.

❖ Throw

`Throw statement` memungkinkan pengguna untuk melempar suatu kesalahan ke block `catch`. `Throw` digunakan di dalam `try` (secara *semantic*, tidak dibatasi secara *syntax*) untuk memberikan pengguna error khusus yang diinginkan. Keyword ini diikuti ekspresi yang menghasilkan nilai yang ingin dilempar dan secara *syntax*.

Contoh Accepted :

```
try {  
    throw exception;  
}  
catch(error) {  
    console.log(error);  
}
```

Contoh Syntax Error:

```
throw if (true) { };
```

Kode di atas merupakan *syntax error* karena *keyword* throw tidak diikuti suatu ekspresi.

❖ While

While statement digunakan untuk melakukan loop yang mengeksekusi blok kode selama kondisi yang ditentukan benar. Pemeriksaan kondisi akan dilakukan terlebih dahulu sebelum melakukan eksekusi blok kode.

Contoh Accepted :

```
while (i < 10){  
    Teks += "Nomor " + i;  
    i++;  
}
```

Contoh Syntax Error :

```
while (){  
    Teks += "Nomor " + i;  
    i++;  
}
```

Kode di atas syntax error karena while harus memiliki kondisi.

❖ Variable

Variable digunakan untuk menyimpan nilai data. Variabel dapat dideklarasikan dengan *keyword* const untuk variabel yang nilainya tidak bisa diubah dan bersifat tidak lokal, let untuk variabel yang nilainya dapat diubah nanti dan bersifat lokal, serta var untuk variabel yang nilainya dapat diubah nanti dan bersifat tidak lokal (tidak ada batasan secara *syntax*). Variabel hanya dapat diawali dengan huruf alfabet, simbol *underscore*, atau *dollar sign*.

Contoh Accepted :

```
let x = 5;
```

Contoh Syntax Error :

```
let @x = 5;
```

Kode di atas menghasilkan *syntax error* karena nama variabel hanya boleh diawali dengan huruf alfabet, simbol *underscore*, atau *dollar sign*.

❖ Assignment

Nomor	Operator	Contoh	Artinya
1	=	x = y	x = y
2	+=	x += y	x = x + y
3	-=	x -= y	x = x - y
4	*=	x *= y	x = x * y
5	/=	x /= y	x = x / y
6	%=	x %= y	x = x % y
7	**=	x **= y	x = x ** y

Shift Assignment Operators

Nomor	Operator	Contoh	Artinya
8	<<=	x <<= y	x = x << y
9	>>=	x >>= y	x = x >> y
10	>>>=	x >>>= y	x = x >>> y

Bitwise Assignment Operators

Nomor	Operator	Contoh	Artinya
11	&=	x &= y	x = x & y
12	^=	x ^= y	x = x ^ y
13	=	x = y	x = x y

Logical Assignment Operators

Nomor	Operator	Contoh	Artinya
14	&&=	x &&= y	x = x && (x=y)
15	=	x = y	x = x (x=y)
16	??=	x ??= y	x = x ?? (x=y)

1. Operator '=' digunakan untuk memberikan nilai ke variabel.
2. Operator '+=' digunakan untuk menambahkan nilai ke variabel.
3. Operator '-=' digunakan untuk mengurangi nilai dari variabel.
4. Operator '*=' digunakan untuk mengalikan variabel.
5. Operator '**=' digunakan untuk mengangkat variabel.
6. Operator '/=' digunakan untuk membagi variabel.
7. Operator '%=' digunakan untuk melakuakn modulo pada variabel.
8. Operator '<<=' digunakan untuk menggeser variabel ke kiri.
9. Operator '>>=' digunakan untuk menggeser variabel ke kanan (signed).
10. Operator '>>>=' digunakan untuk menggeser variabel ke kanan (unsigned).
11. Operator '&=' digunakan untuk melakukan operasi bitwise AND pada dua operan dan menugaskan hasilnya ke variabel.
12. Operator '|=' digunakan untuk melakukan operasi bitwise OR pada dua operan dan menugaskan hasilnya ke variabel.
13. Operator '^=' digunakan untuk melakukan operasi bitwise XOR pada dua operan dan menugaskan hasilnya ke variabel.
14. Operator '&&=' digunakan di antara dua nilai. Jika nilai pertama benar, maka nilai kedua diberikan.
15. Operator '||=' digunakan di antara dua nilai. Jika nilai pertama salah, maka nilai kedua diberikan.
16. Operator '??=' digunakan di antara dua nilai. Jika nilai pertama nol atau tidak terd, maka nilai kedua ditetapkan.

Contoh Accepted :

```
let x = 5;
```

```
x += 5;
```

Contoh Syntax Error :

```
let x = 5;
```

```
x 5 +=;
```

Kode di atas menghasilkan *syntax error* karena operator += membutuhkan dua operan.

❖ Comparison

Operator Perbandingan

Misal untuk x = 5, maka:

Nomor	Operator	Contoh	Hasil
1	==	x == 8	false
2	===	x === 5	true
3	!=	x != 8	true
4	!==	x !== 5	false
5	>	x > 8	false
6	<	x < 8	true
7	>=	x >= 8	false
8	<=	x <= 8	true

Operator Logika

Misal untuk x = 6, y = 3, maka:

Nomor	Operator	Contoh	Hasil
9	&&	(x < 10 && y > 1)	true
10		(x == 5 y == 5)	false
11	!	!(x == y)	true

1. Operator '==' digunakan untuk membandingkan dua variabel atau nilai. Jika kedua nilai atau variabel sama, maka true. Jika tidak, maka false.
2. Operator '===' digunakan untuk membandingkan dua variabel atau nilai. Jika kedua nilai atau variabel nilai dan tipenya sama, maka true. Jika tidak, maka false.
3. Operator '!=' digunakan untuk membandingkan dua variabel atau nilai. Jika kedua nilai atau variabel berbeda, maka true. Jika tidak, maka false.
4. Operator '!===' digunakan untuk membandingkan dua variabel atau nilai. Jika kedua nilai atau variabel nilai dan tipenya berbeda, maka true. Jika tidak, maka false.
5. Operator '>' digunakan untuk membandingkan dua variabel atau nilai. Jika nilai atau variabel pertama lebih besar dari nilai atau variabel kedua, maka true. Jika tidak, maka false.
6. Operator '<' digunakan untuk membandingkan dua variabel atau nilai. Jika nilai atau variabel pertama lebih kecil dari nilai atau variabel kedua, maka true. Jika tidak, maka false.
7. Operator '>=' digunakan untuk membandingkan dua variabel atau nilai. Jika nilai atau variabel pertama lebih besar sama dengan dari nilai atau variabel kedua, maka true. Jika tidak, maka false.
8. Operator '<=' digunakan untuk membandingkan dua variabel atau nilai. Jika nilai atau variabel pertama lebih kecil sama dengan dari nilai atau variabel kedua, maka true. Jika tidak, maka false.
9. Operator '&&' digunakan untuk membandingkan dua kondisi yang membandingkan dua variabel atau nilai. Jika kedua perbandingan benar, maka true. Jika tidak, maka false.
10. Operator '||' digunakan untuk membandingkan dua kondisi yang membandingkan dua variabel atau nilai. Jika minimal salah satu perbandingan benar, maka true. Jika tidak, maka false.
11. Operator '!' digunakan untuk melakukan not dari hasil dari kondisi perbandingan dua variabel atau nilai. Jika hasil perbandingan benar, maka menjadi not true sehingga menjadi false. Untuk sebaliknya, maka false.

Contoh Accepted :

```
x && 3;
```

Contoh Syntax Error :

```
x = = 3;
```

Kode di atas syntax error karena operator == tidak boleh terdapat spasi di antaranya.

❖ String

Tipe string digunakan untuk menyimpan dan memanipulasi teks. String adalah nol atau lebih karakter yang ditulis di dalam tanda kutip. Tanda kutip dapat berupa kutip satu maupun kutip dua. Tanda kutip dapat digunakan di dalam string jika tidak sama dengan tanda kutip yang digunakan untuk membuka dan menutup string (atau digunakan *escape character* untuk tanda kutip yang sama seperti yang dijelaskan pada paragraf selanjutnya).

Panjang string dapat didapatkan dengan menggunakan 'length'. Selain itu, terdapat karakter escape backslash (\) untuk mengubah karakter khusus menjadi karakter string, yaitu sebagai berikut :

Nomor	Code	Hasil
1	\'	'
2	\"	"
3	\\	\

Selain itu, terdapat beberapa escape sequence lainnya, yaitu :

Nomor	Code	Hasil
1	\n	Enter (new line)

Contoh Accepted :

```
const foo = "string";
```

Contoh Syntax Error :

```
const foo = "string";
```

Kode di atas menghasilkan *syntax error* karena string harus diapit oleh tanda petik yang sejenis.

❖ Array

Array adalah variabel khusus yang dapat menampung lebih dari satu nilai. Deklarasi array dapat menggunakan array literal dan dengan kata kunci `const`. Dalam array, spasi dan new line tidak penting, serta deklarasi dapat terdiri atas beberapa baris. Saat mendeklarasi array, pengguna juga dapat memberikan elemen-elemen dari array tersebut.

Untuk mengakses elemen, diperlukan index dari elemen pada array tersebut. Selain itu, pengguna juga dapat mengganti elemen - elemen dari array yang sudah dideklarasikan. Untuk mengakses array lengkap, dapat mengakses dengan mengacu pada nama array tersebut.

Array dapat berisi bermacam-macam tipe data, seperti *objects*, *functions*, *arrays*, dan lain-lain. Selain itu, untuk mengetahui panjang dari array, pengguna dapat menggunakan properti `'length'`. Pengguna juga dapat menggunakan metode `'push(element yang ingin ditambahkan)'` untuk menambahkan elemen ke array.

Contoh Accepted :

```
const nama = ["Budi", "Bob", "Ayu"];
```

Contoh Syntax Error :

```
const = ["Budi", "Bob", "Ayu"];
```

Kode di atas menghasilkan *syntax error* karena nama array tidak didefinisikan.

❖ Object

Objek merupakan wadah untuk nilai bernama yang disebut properti. Objek tidak mementingkan spasi dan new line. Selain itu, deklarasi objek dapat ditulis dalam beberapa baris.

Properti sendiri adalah pasangan nama dan nilai dalam objek. Untuk mengakses properti objek, dapat dilakukan dengan `namaObjek.namaProperti` atau `namaObjek["namaProperti"]`.

Objek juga memiliki metode yang merupakan tindakan yang dapat dilakukan pada objek. Metode disimpang sebagai definisi fungsi dalam properti. Untuk mengakses metode, dapat dilakukan dengan `namaObjek.namaMetode()` .

Contoh Accepted :

```
const car = { type: "Fiat", model: "500", color: "white" };
```

Contoh Syntax Error :

```
const car = { type: "Fiat" model: "500" color: "white" };
```

Kode di atas menghasilkan *syntax error* karena setiap atribut objek harus dipisah dengan tanda koma.

BAB II

HASIL

A. Finite Automata (FA)

- a. FA untuk mengecek apakah suatu ekspresi valid atau tidak

```
# Making FA to check expression is valid or not

# Start State
def Expression_state_1(char):
    # Checking the first character of the variable
    state = 0
    if((ord(char) == 33) or (ord(char) == 36) or
(ord(char) >= 65 and ord(char) <= 90) or (ord(char) == 95)
or (ord(char) >= 97 and ord(char) <= 122)):
        state = 2
    elif(ord(char) >= 48 and ord(char) <= 57):
        state = 4
    else:
        state = 3
    return state

# First Final State
def Expression_state_2(char):
    # Checking the rest character of the variable
    state = 0
    if((ord(char) == 36) or (ord(char) >= 48 and ord(char)
<= 57) or (ord(char) >= 65 and ord(char) <= 90) or
(ord(char) == 95) or (ord(char) >= 97 and ord(char) <=
122)):
        state = 2
    elif(ord(char) == 32):
        state = 5
    elif(ord(char) == 37 or ord(char) == 42 or ord(char)
== 43 or ord(char) == 45 or ord(char) == 47):
        state = 6
    elif(ord(char) == 38):
        state = 7
    elif(ord(char) == 124):
        state = 8
    elif(ord(char) == 60 or ord(char) == 62):
```

```

        state = 9
    elif(ord(char) == 61 or ord(char) == 33):
        state = 10
    else:
        state = 3
    return state

def Expression_state_3(char):
    # Dead state (Expression not valid)
    state = 0
    if(char):
        state = 3
    else:
        state = 3
    return state

# Second Final State
def Expression_state_4(char):
    # Checking if the number consist of number only or not
    state = 0
    if(ord(char) >= 48 and ord(char) <= 57):
        state = 4
    elif(ord(char) == 32):
        state = 5
    elif(ord(char) == 37 or ord(char) == 42 or ord(char)
== 43 or ord(char) == 45 or ord(char) == 47):
        state = 6
    elif(ord(char) == 38):
        state = 7
    elif(ord(char) == 124):
        state = 8
    elif(ord(char) == 60 or ord(char) == 62):
        state = 9
    elif(ord(char) == 61 or ord(char) == 33):
        state = 10
    else:
        state = 3
    return state

def Expression_state_5(char):
    # Checking if blank or operator is valid or not
    state = 0
    if(ord(char) == 32):

```

```

        state = 5
        elif(ord(char) == 37 or ord(char) == 42 or ord(char)
== 43 or ord(char) == 45 or ord(char) == 47):
            state = 6
            elif(ord(char) == 38):
                state = 7
            elif(ord(char) == 124):
                state = 8
            elif(ord(char) == 60 or ord(char) == 62):
                state = 9
            elif(ord(char) == 61 or ord(char) == 33):
                state = 10
        else:
            state = 3
        return state

def Expression_state_6(char):
    # Checking if blank or make sure there is number or
    variable after operator
    state = 0
    if(ord(char) == 32):
        state = 6
    elif((ord(char) == 33) or (ord(char) == 36) or
(ord(char) >= 65 and ord(char) <= 90) or (ord(char) == 95)
or (ord(char) >= 97 and ord(char) <= 122)):
        state = 2
    elif(ord(char) >= 48 and ord(char) <= 57):
        state = 4
    else:
        state = 3
    return state

def Expression_state_7(char):
    # Checking if there is another & after & (&) or make
    sure there is number or variable after &
    state = 0
    if((ord(char) == 36) or (ord(char) >= 65 and ord(char)
<= 90) or (ord(char) == 95) or (ord(char) >= 97 and
ord(char) <= 122)):
        state = 2
    elif(ord(char) >= 48 and ord(char) <= 57):
        state = 4
    if(ord(char) == 38 or ord(char) == 32):

```

```

        state = 6
    else:
        state = 3
    return state

def Expression_state_8(char):
    # Checking if there is another | after | (||) or make
    # sure there is number or variable after |
    state = 0
    if((ord(char) == 36) or (ord(char) >= 65 and ord(char)
    <= 90) or (ord(char) == 95) or (ord(char) >= 97 and
    ord(char) <= 122)):
        state = 2
    elif(ord(char) >= 48 and ord(char) <= 57):
        state = 4
    elif(ord(char) == 124 or ord(char) == 32):
        state = 6
    else:
        state = 3
    return state

def Expression_state_9(char):
    # Checking if there is = after < or > (<= or >=) or
    # make sure there is number or variable after < or >
    state = 0
    if((ord(char) == 36) or (ord(char) >= 65 and ord(char)
    <= 90) or (ord(char) == 95) or (ord(char) >= 97 and
    ord(char) <= 122)):
        state = 2
    elif(ord(char) >= 48 and ord(char) <= 57):
        state = 4
    elif(ord(char) == 61 or ord(char) == 32):
        state = 6
    else:
        state = 3
    return state

def Expression_state_10(char):
    # Checking if there is = after = (==)
    state = 0
    if(ord(char) == 61):
        state = 6
    else:

```



```

        state = 3
    return state

# FA to check if an expression is valid or not
def CheckExpression(expression):
    state = 1
    for char in expression:
        if(state == 1):
            state = Expression_state_1(char)
        elif(state == 2):
            state = Expression_state_2(char)
        elif(state == 3):
            state = Expression_state_3(char)
        elif(state == 4):
            state = Expression_state_4(char)
        elif(state == 5):
            state = Expression_state_5(char)
        elif(state == 6):
            state = Expression_state_6(char)
        elif(state == 7):
            state = Expression_state_7(char)
        elif(state == 8):
            state = Expression_state_8(char)
        elif(state == 9):
            state = Expression_state_9(char)
        elif(state == 10):
            state = Expression_state_10(char)

    if(state == 2 or state == 4):
        return True
    else:
        return False

```

- b. FA untuk mengecek apakah suatu variabel valid atau tidak

```

# Making FA to check if a variable is valid or not

# Start state (var_state_1)
def Var_state_1(char):
    # Checking the first character of the variable
    state = 0

```

```

        if((ord(char) == 36) or (ord(char) >= 65 and ord(char)
<= 90) or (ord(char) == 95) or (ord(char) >= 97 and
ord(char) <= 122)):
            state = 2
        else:
            state = 3
        return state

# Final State (var_state_2)
def Var_state_2(char):
    # Checking the rest character of the variable
    state = 0
    if((ord(char) == 36) or (ord(char) >= 48 and ord(char)
<= 57) or (ord(char) >= 65 and ord(char) <= 90) or
(ord(char) == 95) or (ord(char) >= 97 and ord(char) <=
122)):
        state = 2
    else:
        state = 3
    return state

# State 3 (var_state_3)
def Var_state_3(char):
    # Dead state (variable name not valid)
    state = 0
    if(char):
        state = 3
    else:
        state = 3
    return state

# Function FA for checking a variable name valid or not
def CheckVariable(variable_name):
    state = 1 # Start State
    for char in variable_name:
        if(state == 1):
            state = Var_state_1(char)
        elif(state == 2):
            state = Var_state_2(char)
        elif(state == 3):
            state = Var_state_3(char)
    if(state == 2):
        return True

```

```

else:
    return False

```

B. Context Free Grammar (CFG)

```

#TERMINALS
break  const  case  catch  continue  default  delete
else   false  finally  for  function  if  let  null
return  switch  throw  try  true  var  while  new  Error
( ) { } [ ] ; = + - * / % < > &
^ | ? ! \ ' /* */ \\ \" \` a b c d e
f g h i j k l m n o p q r s t u v
w x y z A B C D E F G H I J K L
M N O P Q R S T U V W X Y Z . ' "
# $ % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ _ ` { | } ~
€ , „ † ‡ ^ % Š ‹
Œ Ž ‘ ’ “ ” • – — ~ ™ š › œ ž Ÿ ĩ
ç ₣ ₧ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³
´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿ À Á Â Ã
Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ
Ö × Ø Ù Ú Û Ü Ý Þ ß à á â ã ä å
æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö
÷ ø ù ú û ü ý þ ÿ

#VARIABLES
ASSIGNMENT_OP  COMPARISON_OP  BINARY_ARITHMETIC_OP
UNARY_ARITHMETIC_OP  BINARY_BITWISE_OP  UNARY_BITWISE_OP
BINARY_LOGICAL_OP  UNARY_LOGICAL_OP  BINARY_OP  UNARY_OP
SINGLE_INT  INT  BIGINT  FLOAT  NUMBER  ALPHABET
PRINTABLE_ASCII_WITHOUT_QUOTATION_AND_ESCAPE
PRINTABLE_ASCII_WITHOUT_QUOTATION
PRINTABLE_ASCII_WITHOUT_FORMAT_HEAD
PRINTABLE_ASCII_WITHOUT_DOUBLE_QUOTATION_HEAD
PRINTABLE_ASCII_WITHOUT_SINGLE_QUOTATION_HEAD
ASCII_WORD_WITHOUT_DOUBLE_QUOTATION_HEAD
ASCII_WORD_WITHOUT_SINGLE_QUOTATION_HEAD
ASCII_WORD_WITHOUT_FORMAT_HEAD
PRINTABLE_ASCII_WITHOUT_QUOTATION_TAIL
PRINTABLE_ASCII_WITHOUT_FORMAT_TAIL
PRINTABLE_ASCII_WITHOUT_DOUBLE_QUOTATION_TAIL
PRINTABLE_ASCII_WITHOUT_SINGLE_QUOTATION_TAIL
ASCII_WORD_WITHOUT_DOUBLE_QUOTATION_TAIL
ASCII_WORD_WITHOUT_SINGLE_QUOTATION_TAIL

```

```

ASCII_WORD_WITHOUT_FORMAT_TAIL  ASCII_WORD  FORMAT_WORD
DOUBLE_QUOTE_STRING  SINGLE_QUOTE_STRING  FORMAT_STRING
BOOLEAN  STRING  VARIABLES_FIRST_CHAR  VARIABLES_SUFFIX
VARIABLES  PARAMETER_LIST  FUNC_CALL  OBJ_ATTR_CALL
OBJ_FUNC_CALL  OBJ_CALL  PRIMITIVE  DATA  ARRAY_LIST  ARRAY
  OBJ_ELMT  OBJ  LOGICAL_EXPRESSION  ARITHMETIC_EXPRESSION
STRING_EXPRESSION  EXPRESSION  ASSIGNMENT
DECLARATION_KEYWORD  DECLARATION  RETURN_STATEMENT
THROW_STATEMENT  STATEMENT  LINE  BODY_BLOCK  CONDITION
CONDITIONAL_BLOCK  ONLY_IF_STATEMENT  ELIF_STATEMENT
ELSE_STATEMENT  IF_STATEMENT  LOOP_LINE  LOOP_BLOCK
CONDITIONAL_LOOP_BLOCK  WHILE_STATEMENT  ARRAY_PARAM_LIST
ARRAY_PARAM  OBJECT_PARAM_ELMT  OBJECT_PARAM_LIST
OBJECT_PARAM  FOR_CONDITION  FOR_STATEMENT  PARAMETER_ELMT
PARAMETER_LIST  PARAMETER  FUNCTION  TRY_STATEMENT
ONLY_TRY_STATEMENT  CATCH_STATEMENT  FINALLY_STATEMENT
CASE_LINE  CASE_BLOCK  SWITCH_CASE  SWITCH_DEFAULT
SWITCH_BLOCK  SWITCH_STATEMENT  FUNCTION_LINE  FUNCTION_BLOCK
  CODE  FUNCTION_IF_STATEMENT  FUNCTION_CONDITIONAL_BLOCK
FUNCTION_ONLY_IF_STATEMENT  FUNCTION_ELIF_STATEMENT
FUNCTION_ELSE_STATEMENT  FUNCTION_CONDITIONAL_BLOCK
FUNCTION_LOOP_LINE  FUNCTION_LOOP_BLOCK
FUNCTION_CONDITIONAL_LOOP_BLOCK  FUNCTION_WHILE_STATEMENT
FUNCTION_FOR_STATEMENT  FUNCTION_TRY_STATEMENT
FUNCTION_ONLY_TRY_STATEMENT  FUNCTION_CATCH_STATEMENT
FUNCTION_FINALLY_STATEMENT  FUNCTION_CASE_LINE
FUNCTION_CASE_BLOCK  FUNCTION_SWITCH_CASE
FUNCTION_SWITCH_DEFAULT  FUNCTION_SWITCH_BLOCK
FUNCTION_SWITCH_STATEMENT  MULTI_LINE_COMMENT  ASCII_WORD
BLANK  DECLARATION_ELMT  DECLARATION_LIST  ARGUMENT_LIST
PRINTABLE_ASCII_WITHOUT_FORMAT_WITHOUT_OPENCURLY_HEAD
PRINTABLE_ASCII_WITHOUT_FORMAT_WITHOUT_DOLLAR_HEAD
PRINTABLE_ASCII_WITHOUT_FORMAT_WITHOUT_OPENCURLY_TAIL
PRINTABLE_ASCII_WITHOUT_FORMAT_WITHOUT_DOLLAR_TAIL
FORMAT_WORD_HEAD  FORMAT_WORD_TAIL  OBJ_LIST  ARR_CALL
ARR_IDX  LOOP_IF_STATEMENT  LOOP_IF_STATEMENT
LOOP_CONDITIONAL_BLOCK  LOOP_ONLY_IF_STATEMENT
LOOP_ELIF_STATEMENT  LOOP_ELSE_STATEMENT  LOOP_TRY_STATEMENT
LOOP_ONLY_TRY_STATEMENT  LOOP_CATCH_STATEMENT
LOOP_FINALLY_STATEMENT  LOOP_CASE_LINE  LOOP_CASE_BLOCK
LOOP_SWITCH_CASE  LOOP_SWITCH_DEFAULT  LOOP_SWITCH_BLOCK
LOOP_SWITCH_STATEMENT  CONDITIONAL_EXPRESSION
CONST_DECLARATION  CONST_DECLARATION_ELMT

```

```

CONST_DECLARATION_LIST  ASSIGNED_EXPRESSION  TRY_LINE
TRY_BLOCK  LOOP_TRY_LINE  LOOP_TRY_BLOCK  FUNCTION_TRY_LINE
FUNCTION_TRY_BLOCK  ARRAY_KEY_LIST
FUNCTION_LOOP_CONDITIONAL_BLOCK  FUNCTION_LOOP_ELIF_STATEMENT
FUNCTION_LOOP_ELSE_STATEMENT  FUNCTION_LOOP_IF_STATEMENT
FUNCTION_LOOP_ONLY_IF_STATEMENT  FUNCTION_LOOP_CASE_LINE
FUNCTION_LOOP_CASE_BLOCK  FUNCTION_LOOP_SWITCH_CASE
FUNCTION_LOOP_SWITCH_DEFAULT  FUNCTION_LOOP_SWITCH_BLOCK
FUNCTION_LOOP_SWITCH_STATEMENT  FUNCTION_LOOP_TRY_LINE
FUNCTION_LOOP_TRY_BLOCK  FUNCTION_LOOP_TRY_STATEMENT
FUNCTION_LOOP_ONLY_TRY_STATEMENT  FUNCTION_LOOP_CATCH_STATEMENT
FUNCTION_LOOP_FINALLY_STATEMENT
#PRODUCTIONS
ASSIGNMENT_OP -> =  + =  - =  * =  / =  % =  * * =  < < =
    > > =  > > > =  & =  ^ =  | =  & & =  | | =  ? ? =
COMPARISON_OP -> = =  ! =  = = =  ! = =  >  > =  <  < =
BINARY_ARITHMETIC_OP -> %  -  +  * *  /  *
UNARY_ARITHMETIC_OP -> + +  - -
BINARY_BITWISE_OP -> &  |  ^  < <  > >  > > >
UNARY_BITWISE_OP -> ~
BINARY_LOGICAL_OP -> & &  | |  COMPARISON_OP  ? ?
UNARY_LOGICAL_OP -> !
BINARY_OP -> ASSIGNMENT_OP  COMPARISON_OP
BINARY_ARITHMETIC_OP  BINARY_BITWISE_OP  BINARY_LOGICAL_OP
UNARY_OP -> UNARY_ARITHMETIC_OP  UNARY_BITWISE_OP
UNARY_LOGICAL_OP
SINGLE_INT -> 0  1  2  3  4  5  6  7  8  9
INT -> SINGLE_INT  INT INT
BIGINT -> INT n
FLOAT -> INT . INT  INT .  . INT
NUMBER -> INT  BIGINT  FLOAT
ALPHABET -> a  b  c  d  e  f  g  h  i  j  k  l  m  n
    o  p  q  r  s  t  u  v  w  x  y  z  A  B  C  D
E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U
    V  W  X  Y  Z
PRINTABLE_ASCII_WITHOUT_QUOTATION_AND_ESCAPE -> ALPHABET
SINGLE_INT  \'  \"  \\  \`  !  #  %  &  (  )  *  +
-  /  :  ;  <  =  >  ?  @  [  ]  ^  _  |  ~  €  ,
"  ...  †  ‡  ^  %  Š  <  Œ  Ž  '  '  "  "  .  -
-  ~  ™  š  >  œ  ž  Ÿ  ¡  ¢  £  ¤  ¥  ¦  §  ¨  ©
a  «  ¬  ®  ¯  °  ±  ²  ³  ´  µ  ¶  ·  ¸  ¹  º
»  ¼  ½  ¾  ¿  À  Á  Â  Ã  Ä  Å  Æ  Ç  È  É  Ê  Ë
Ì  Í  Î  Ï  Ð  Ñ  Ò  Ó  Ô  Õ  Ö  ×  Ø  Ù  Ú  Û

```

Ü Ý Þ ß à á â ã ä å æ ç è é ê ë ì
 í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü
 ý þ ÿ BLANK } .
 PRINTABLE_ASCII_WITHOUT_QUOTATION ->
 PRINTABLE_ASCII_WITHOUT_QUOTATION_AND_ESCAPE \ \$ { `
 PRINTABLE_ASCII_WITHOUT_FORMAT_HEAD ->
 PRINTABLE_ASCII_WITHOUT_QUOTATION_AND_ESCAPE " ' \
 PRINTABLE_ASCII_WITHOUT_DOUBLE_QUOTATION_HEAD ->
 PRINTABLE_ASCII_WITHOUT_QUOTATION '
 PRINTABLE_ASCII_WITHOUT_SINGLE_QUOTATION_HEAD ->
 PRINTABLE_ASCII_WITHOUT_QUOTATION "
 PRINTABLE_ASCII_WITHOUT_FORMAT_WITHOUT_OPENCURLY_HEAD ->
 PRINTABLE_ASCII_WITHOUT_FORMAT_HEAD \$
 PRINTABLE_ASCII_WITHOUT_FORMAT_WITHOUT_DOLLAR_HEAD ->
 PRINTABLE_ASCII_WITHOUT_FORMAT_HEAD {
 ASCII_WORD_WITHOUT_DOUBLE_QUOTATION_HEAD ->
 PRINTABLE_ASCII_WITHOUT_DOUBLE_QUOTATION_HEAD
 ASCII_WORD_WITHOUT_DOUBLE_QUOTATION_HEAD
 ASCII_WORD_WITHOUT_DOUBLE_QUOTATION_HEAD
 ASCII_WORD_WITHOUT_SINGLE_QUOTATION_HEAD ->
 PRINTABLE_ASCII_WITHOUT_SINGLE_QUOTATION_HEAD
 ASCII_WORD_WITHOUT_SINGLE_QUOTATION_HEAD
 ASCII_WORD_WITHOUT_SINGLE_QUOTATION_HEAD
 ASCII_WORD_WITHOUT_FORMAT_HEAD ->
 PRINTABLE_ASCII_WITHOUT_FORMAT_HEAD
 ASCII_WORD_WITHOUT_FORMAT_HEAD ASCII_WORD_WITHOUT_FORMAT_HEAD
 PRINTABLE_ASCII_WITHOUT_QUOTATION_TAIL ->
 PRINTABLE_ASCII_WITHOUT_QUOTATION_AND_ESCAPE \$ { `
 PRINTABLE_ASCII_WITHOUT_FORMAT_TAIL ->
 PRINTABLE_ASCII_WITHOUT_QUOTATION_AND_ESCAPE " '
 PRINTABLE_ASCII_WITHOUT_FORMAT_WITHOUT_OPENCURLY_TAIL ->
 PRINTABLE_ASCII_WITHOUT_FORMAT_TAIL \$
 PRINTABLE_ASCII_WITHOUT_FORMAT_WITHOUT_DOLLAR_TAIL ->
 PRINTABLE_ASCII_WITHOUT_FORMAT_TAIL {
 PRINTABLE_ASCII_WITHOUT_DOUBLE_QUOTATION_TAIL ->
 PRINTABLE_ASCII_WITHOUT_QUOTATION_TAIL '
 PRINTABLE_ASCII_WITHOUT_SINGLE_QUOTATION_TAIL ->
 PRINTABLE_ASCII_WITHOUT_QUOTATION_TAIL "
 ASCII_WORD_WITHOUT_DOUBLE_QUOTATION_TAIL ->
 PRINTABLE_ASCII_WITHOUT_DOUBLE_QUOTATION_TAIL
 ASCII_WORD_WITHOUT_DOUBLE_QUOTATION_TAIL
 ASCII_WORD_WITHOUT_DOUBLE_QUOTATION_TAIL
 ASCII_WORD_WITHOUT_SINGLE_QUOTATION_TAIL ->

```

PRINTABLE_ASCII_WITHOUT_SINGLE_QUOTATION_TAIL
ASCII_WORD_WITHOUT_SINGLE_QUOTATION_TAIL
ASCII_WORD_WITHOUT_SINGLE_QUOTATION_TAIL
ASCII_WORD_WITHOUT_FORMAT_TAIL ->
PRINTABLE_ASCII_WITHOUT_FORMAT_TAIL
ASCII_WORD_WITHOUT_FORMAT_TAIL ASCII_WORD_WITHOUT_FORMAT_TAIL
ASCII_WORD -> PRINTABLE_ASCII_WITHOUT_QUOTATION " '
ASCII_WORD ASCII_WORD
FORMAT_WORD_HEAD -> ASCII_WORD_WITHOUT_FORMAT_TAIL $ {
ASSIGNED_EXPRESSION } ASCII_WORD_WITHOUT_FORMAT_HEAD $
PRINTABLE_ASCII_WITHOUT_FORMAT_WITHOUT_OPENCURLY_HEAD
PRINTABLE_ASCII_WITHOUT_FORMAT_WITHOUT_DOLLAR_HEAD {
FORMAT_WORD_HEAD FORMAT_WORD_HEAD
FORMAT_WORD_TAIL -> ASCII_WORD_WITHOUT_FORMAT_TAIL $ {
ASSIGNED_EXPRESSION } ASCII_WORD_WITHOUT_FORMAT_HEAD
ASCII_WORD_WITHOUT_FORMAT_TAIL $
PRINTABLE_ASCII_WITHOUT_FORMAT_WITHOUT_OPENCURLY_TAIL
PRINTABLE_ASCII_WITHOUT_FORMAT_WITHOUT_DOLLAR_TAIL {
FORMAT_WORD_TAIL FORMAT_WORD_TAIL
FORMAT_WORD -> FORMAT_WORD_TAIL FORMAT_WORD_HEAD FORMAT_WORD
{ FORMAT_WORD FORMAT_WORD $ $ {
DOUBLE_QUOTE_STRING -> "
ASCII_WORD_WITHOUT_DOUBLE_QUOTATION_HEAD
ASCII_WORD_WITHOUT_DOUBLE_QUOTATION_TAIL " "
ASCII_WORD_WITHOUT_DOUBLE_QUOTATION_TAIL " " "
SINGLE_QUOTE_STRING -> '
ASCII_WORD_WITHOUT_SINGLE_QUOTATION_HEAD
ASCII_WORD_WITHOUT_SINGLE_QUOTATION_TAIL ' '
ASCII_WORD_WITHOUT_SINGLE_QUOTATION_TAIL ' ' '
FORMAT_STRING -> ` FORMAT_WORD ` ` `
STRING -> DOUBLE_QUOTE_STRING SINGLE_QUOTE_STRING
FORMAT_STRING
BOOLEAN -> true false
VARIABLES_FIRST_CHAR -> ALPHABET _ $
VARIABLES_SUFIX -> SINGLE_INT VARIABLES_FIRST_CHAR
VARIABLES_SUFIX VARIABLES_SUFIX
VARIABLES -> VARIABLES_FIRST_CHAR VARIABLES_FIRST_CHAR
VARIABLES_SUFIX
ARRAY_LIST -> ASSIGNED_EXPRESSION , ASSIGNED_EXPRESSION ,
ARRAY_LIST , ARRAY_LIST
ARRAY -> [ ARRAY_LIST ]
OBJ_ELMT -> VARIABLES VARIABLES : ASSIGNED_EXPRESSION
OBJ_LIST -> OBJ_ELMT OBJ_ELMT , OBJ_ELMT , OBJ_LIST

```

```

OBJ -> { OBJ_LIST } { }
ARGUMENT_LIST -> ASSIGNED_EXPRESSION ASSIGNED_EXPRESSION ,
ARGUMENT_LIST
FUNC_CALL -> VARIABLES ( ARGUMENT_LIST ) VARIABLES ( )
OBJ_ATTR_CALL -> VARIABLES VARIABLES . OBJ_ATTR_CALL
VARIABLES ? . OBJ_ATTR_CALL
OBJ_FUNC_CALL -> OBJ_ATTR_CALL . FUNC_CALL OBJ_ATTR_CALL ? .
FUNC_CALL
OBJ_CALL -> OBJ_ATTR_CALL OBJ_FUNC_CALL
ARR_IDX -> [ ASSIGNED_EXPRESSION ] ARR_IDX ARR_IDX
ARR_CALL -> VARIABLES ARR_IDX OBJ_ATTR_CALL ARR_IDX
PRIMITIVE -> BOOLEAN NUMBER STRING ARRAY OBJ null
DATA -> PRIMITIVE VARIABLES FUNC_CALL OBJ_CALL ARR_CALL
LOGICAL_EXPRESSION -> EXPRESSION LOGICAL_EXPRESSION
BINARY_LOGICAL_OP LOGICAL_EXPRESSION UNARY_LOGICAL_OP
LOGICAL_EXPRESSION
ARITHMETIC_EXPRESSION -> EXPRESSION ARITHMETIC_EXPRESSION
BINARY_ARITHMETIC_OP ARITHMETIC_EXPRESSION
ARITHMETIC_EXPRESSION BINARY_BITWISE_OP ARITHMETIC_EXPRESSION
UNARY_ARITHMETIC_OP ARITHMETIC_EXPRESSION BLANK
UNARY_ARITHMETIC_OP ARITHMETIC_EXPRESSION
ARITHMETIC_EXPRESSION UNARY_ARITHMETIC_OP
ARITHMETIC_EXPRESSION UNARY_ARITHMETIC_OP BLANK
UNARY_BITWISE_OP ARITHMETIC_EXPRESSION
STRING_EXPRESSION -> STRING ( STRING_EXPRESSION )
STRING_EXPRESSION + STRING_EXPRESSION
CONDITIONAL_EXPRESSION -> EXPRESSION ? ASSIGNED_EXPRESSION :
ASSIGNED_EXPRESSION
EXPRESSION -> DATA LOGICAL_EXPRESSION ARITHMETIC_EXPRESSION
STRING_EXPRESSION CONDITIONAL_EXPRESSION ASSIGNMENT (
EXPRESSION ) ( - EXPRESSION ) ( + EXPRESSION ) delete
ASSIGNED_EXPRESSION
ASSIGNED_EXPRESSION -> EXPRESSION function PARAMETER {
FUNCTION_BLOCK } function VARIABLES PARAMETER { FUNCTION_BLOCK
} ASSIGNED_EXPRESSION ? ASSIGNED_EXPRESSION :
ASSIGNED_EXPRESSION ASSIGNED_EXPRESSION , ASSIGNED_EXPRESSION
( ASSIGNED_EXPRESSION ) ( - ASSIGNED_EXPRESSION ) ( +
ASSIGNED_EXPRESSION )
ASSIGNMENT -> EXPRESSION ASSIGNMENT_OP ASSIGNED_EXPRESSION
CONST_DECLARATION_ELMT -> VARIABLES = ASSIGNED_EXPRESSION
CONST_DECLARATION_LIST -> CONST_DECLARATION_ELMT
CONST_DECLARATION_ELMT , CONST_DECLARATION_LIST
CONST_DECLARATION -> const CONST_DECLARATION_LIST

```



```

DECLARATION_KEYWORD -> let    var
DECLARATION_ELMT -> VARIABLES    VARIABLES = ASSIGNED_EXPRESSION
DECLARATION_LIST -> DECLARATION_ELMT    DECLARATION_ELMT ,
DECLARATION_LIST
DECLARATION -> DECLARATION_KEYWORD DECLARATION_LIST
CONST_DECLARATION
THROW_STATEMENT -> throw ASSIGNED_EXPRESSION    throw new Error (
ASSIGNED_EXPRESSION )
RETURN_STATEMENT -> return ASSIGNED_EXPRESSION    return
STATEMENT -> EXPRESSION    DECLARATION
LINE -> IF_STATEMENT    WHILE_STATEMENT    FOR_STATEMENT
TRY_STATEMENT    SWITCH_STATEMENT    FUNCTION    STATEMENT ;
MULTI_LINE_COMMENT    ;
BODY_BLOCK -> LINE    BODY_BLOCK BODY_BLOCK    { BODY_BLOCK }
MULTI_LINE_COMMENT -> /* ASCII_WORD */
CONDITION -> ( ASSIGNED_EXPRESSION )
CONDITIONAL_BLOCK -> CONDITION { BODY_BLOCK }    CONDITION LINE
ONLY_IF_STATEMENT -> if CONDITIONAL_BLOCK
ELIF_STATEMENT -> else ONLY_IF_STATEMENT    ELIF_STATEMENT
ELIF_STATEMENT
ELSE_STATEMENT -> else { BODY_BLOCK }    else LINE
IF_STATEMENT -> ONLY_IF_STATEMENT    ONLY_IF_STATEMENT
ELIF_STATEMENT    ONLY_IF_STATEMENT ELSE_STATEMENT
ONLY_IF_STATEMENT ELIF_STATEMENT ELSE_STATEMENT
LOOP_LINE -> LOOP_IF_STATEMENT    WHILE_STATEMENT    FOR_STATEMENT
    LOOP_TRY_STATEMENT    LOOP_SWITCH_STATEMENT    FUNCTION
STATEMENT ;    MULTI_LINE_COMMENT    ;    break ;    continue ;
LOOP_BLOCK -> LOOP_LINE    LOOP_BLOCK LOOP_BLOCK    { LOOP_BLOCK }

LOOP_IF_STATEMENT -> LOOP_ONLY_IF_STATEMENT
LOOP_ONLY_IF_STATEMENT LOOP_ELIF_STATEMENT
LOOP_ONLY_IF_STATEMENT LOOP_ELSE_STATEMENT
LOOP_ONLY_IF_STATEMENT LOOP_ELIF_STATEMENT LOOP_ELSE_STATEMENT
LOOP_CONDITIONAL_BLOCK -> CONDITION { LOOP_BLOCK }    CONDITION
LOOP_LINE
LOOP_ONLY_IF_STATEMENT -> if LOOP_CONDITIONAL_BLOCK
LOOP_ELIF_STATEMENT -> else LOOP_ONLY_IF_STATEMENT
LOOP_ELIF_STATEMENT LOOP_ELIF_STATEMENT
LOOP_ELSE_STATEMENT -> else { LOOP_BLOCK }    else LOOP_LINE
LOOP_TRY_LINE -> LOOP_LINE    THROW_STATEMENT ;
LOOP_TRY_BLOCK -> LOOP_TRY_LINE    LOOP_TRY_BLOCK LOOP_TRY_BLOCK
    { LOOP_TRY_BLOCK }
LOOP_TRY_STATEMENT -> LOOP_ONLY_TRY_STATEMENT

```

```

LOOP_CATCH_STATEMENT LOOP_ONLY_TRY_STATEMENT
LOOP_CATCH_STATEMENT LOOP_FINALLY_STATEMENT
LOOP_ONLY_TRY_STATEMENT LOOP_FINALLY_STATEMENT
LOOP_ONLY_TRY_STATEMENT -> try { LOOP_TRY_BLOCK }
LOOP_CATCH_STATEMENT -> catch ( VARIABLES ) { LOOP_TRY_BLOCK }
catch { LOOP_TRY_BLOCK }
LOOP_FINALLY_STATEMENT -> finally { LOOP_TRY_BLOCK }
LOOP_CASE_LINE -> LOOP_LINE
LOOP_CASE_BLOCK -> LOOP_CASE_LINE LOOP_CASE_BLOCK
LOOP_CASE_BLOCK { LOOP_CASE_BLOCK }
LOOP_SWITCH_CASE -> case ASSIGNED_EXPRESSION : LOOP_CASE_BLOCK
LOOP_SWITCH_CASE LOOP_SWITCH_CASE
LOOP_SWITCH_DEFAULT -> default : LOOP_CASE_BLOCK
LOOP_SWITCH_BLOCK -> LOOP_SWITCH_CASE LOOP_SWITCH_CASE
LOOP_SWITCH_DEFAULT
LOOP_SWITCH_STATEMENT -> switch ( ASSIGNED_EXPRESSION ) {
LOOP_SWITCH_BLOCK }
CONDITIONAL_LOOP_BLOCK -> CONDITION { LOOP_BLOCK } CONDITION
LOOP_LINE
WHILE_STATEMENT -> while CONDITIONAL_LOOP_BLOCK do LOOP_BLOCK
while CONDITION ; do LOOP_LINE while CONDITION ;
ARRAY_PARAM_LIST -> PARAMETER_ELMT , PARAMETER_ELMT ,
ARRAY_PARAM_LIST , ARRAY_PARAM_LIST
ARRAY_PARAM -> [ ARRAY_PARAM_LIST ]
ARRAY_KEY_LIST -> ASSIGNED_EXPRESSION ASSIGNED_EXPRESSION ,
ARRAY_KEY_LIST
OBJECT_PARAM_ELMT -> VARIABLES VARIABLES : PARAMETER_ELMT [
ARRAY_KEY_LIST ] : PARAMETER_ELMT
OBJECT_PARAM_LIST -> OBJECT_PARAM_ELMT OBJECT_PARAM_ELMT ,
OBJECT_PARAM_ELMT , OBJECT_PARAM_LIST
OBJECT_PARAM -> { OBJECT_PARAM_LIST } { }
PARAMETER_ELMT -> VARIABLES VARIABLES = ASSIGNED_EXPRESSION
ARRAY_PARAM OBJECT_PARAM
PARAMETER_LIST -> PARAMETER_ELMT PARAMETER_ELMT ,
PARAMETER_LIST PARAMETER_ELMT ,
PARAMETER -> ( PARAMETER_LIST ) ( )
FOR_CONDITION -> ( ASSIGNED_EXPRESSION ; ASSIGNED_EXPRESSION ;
ASSIGNED_EXPRESSION ) ( DECLARATION ; ASSIGNED_EXPRESSION ;
ASSIGNED_EXPRESSION )
FOR_STATEMENT -> for FOR_CONDITION { LOOP_BLOCK } for
FOR_CONDITION LOOP_LINE
FUNCTION -> function VARIABLES PARAMETER { FUNCTION_BLOCK }
function VARIABLES PARAMETER ;

```

```

FUNCTION_BLOCK -> FUNCTION_LINE    FUNCTION_BLOCK FUNCTION_BLOCK
    { FUNCTION_BLOCK }
FUNCTION_LINE -> MULTI_LINE_COMMENT    FUNCTION_IF_STATEMENT
FUNCTION_WHILE_STATEMENT    FUNCTION_FOR_STATEMENT
FUNCTION_TRY_STATEMENT    FUNCTION_SWITCH_STATEMENT    FUNCTION
STATEMENT ;    RETURN_STATEMENT ;    THROW_STATEMENT ;    ;
FUNCTION_IF_STATEMENT -> FUNCTION_ONLY_IF_STATEMENT
FUNCTION_ONLY_IF_STATEMENT FUNCTION_ELIF_STATEMENT
FUNCTION_ONLY_IF_STATEMENT FUNCTION_ELSE_STATEMENT
FUNCTION_ONLY_IF_STATEMENT FUNCTION_ELIF_STATEMENT
FUNCTION_ELSE_STATEMENT
FUNCTION_CONDITIONAL_BLOCK -> CONDITION { FUNCTION_BLOCK }
CONDITION FUNCTION_LINE
FUNCTION_ONLY_IF_STATEMENT -> if FUNCTION_CONDITIONAL_BLOCK
FUNCTION_ELIF_STATEMENT -> else FUNCTION_ONLY_IF_STATEMENT
FUNCTION_ELIF_STATEMENT FUNCTION_ELIF_STATEMENT
FUNCTION_ELSE_STATEMENT -> else { FUNCTION_BLOCK }    else
FUNCTION_LINE
FUNCTION_LOOP_LINE -> FUNCTION_LINE    break ;    continue ;
FUNCTION_LOOP_IF_STATEMENT    FUNCTION_LOOP_SWITCH_STATEMENT
FUNCTION_LOOP_TRY_STATEMENT
FUNCTION_LOOP_BLOCK -> FUNCTION_LOOP_LINE    FUNCTION_LOOP_BLOCK
FUNCTION_LOOP_BLOCK    { FUNCTION_LOOP_BLOCK }
FUNCTION_CONDITIONAL_LOOP_BLOCK -> CONDITION {
FUNCTION_LOOP_BLOCK }    CONDITION FUNCTION_LOOP_LINE
FUNCTION_WHILE_STATEMENT -> while
FUNCTION_CONDITIONAL_LOOP_BLOCK    do FUNCTION_LOOP_BLOCK while
CONDITION ;    do FUNCTION_LOOP_LINE while CONDITION ;
FUNCTION_FOR_STATEMENT -> for FOR_CONDITION {
FUNCTION_LOOP_BLOCK }    for FOR_CONDITION FUNCTION_LOOP_LINE
FUNCTION_TRY_LINE -> FUNCTION_LINE
FUNCTION_TRY_BLOCK -> FUNCTION_TRY_LINE    FUNCTION_TRY_BLOCK
FUNCTION_TRY_BLOCK    { FUNCTION_TRY_BLOCK }
FUNCTION_TRY_STATEMENT -> FUNCTION_ONLY_TRY_STATEMENT
FUNCTION_CATCH_STATEMENT    FUNCTION_ONLY_TRY_STATEMENT
FUNCTION_CATCH_STATEMENT FUNCTION_FINALLY_STATEMENT
FUNCTION_ONLY_TRY_STATEMENT FUNCTION_FINALLY_STATEMENT
FUNCTION_ONLY_TRY_STATEMENT -> try { FUNCTION_TRY_BLOCK }
FUNCTION_CATCH_STATEMENT -> catch ( VARIABLES ) {
FUNCTION_TRY_BLOCK }    catch { FUNCTION_TRY_BLOCK }
FUNCTION_FINALLY_STATEMENT -> finally { FUNCTION_TRY_BLOCK }
FUNCTION_CASE_LINE -> FUNCTION_LINE    break ;
FUNCTION_CASE_BLOCK -> FUNCTION_CASE_LINE    FUNCTION_CASE_BLOCK

```

```

FUNCTION_CASE_BLOCK    { FUNCTION_CASE_BLOCK }
FUNCTION_SWITCH_CASE -> case ASSIGNED_EXPRESSION :
FUNCTION_CASE_BLOCK    FUNCTION_SWITCH_CASE FUNCTION_SWITCH_CASE
FUNCTION_SWITCH_DEFAULT -> default : FUNCTION_CASE_BLOCK
FUNCTION_SWITCH_BLOCK -> FUNCTION_SWITCH_CASE
FUNCTION_SWITCH_CASE FUNCTION_SWITCH_DEFAULT
FUNCTION_SWITCH_STATEMENT -> switch ( ASSIGNED_EXPRESSION ) {
FUNCTION_SWITCH_BLOCK }
FUNCTION_LOOP_IF_STATEMENT -> FUNCTION_LOOP_ONLY_IF_STATEMENT
FUNCTION_LOOP_ONLY_IF_STATEMENT FUNCTION_LOOP_ELIF_STATEMENT
FUNCTION_LOOP_ONLY_IF_STATEMENT FUNCTION_LOOP_ELSE_STATEMENT
FUNCTION_LOOP_ONLY_IF_STATEMENT FUNCTION_LOOP_ELIF_STATEMENT
FUNCTION_LOOP_ELSE_STATEMENT
FUNCTION_LOOP_CONDITIONAL_BLOCK -> CONDITION {
FUNCTION_LOOP_BLOCK }    CONDITION FUNCTION_LOOP_LINE
FUNCTION_LOOP_ONLY_IF_STATEMENT -> if
FUNCTION_LOOP_CONDITIONAL_BLOCK
FUNCTION_LOOP_ELIF_STATEMENT -> else
FUNCTION_LOOP_ONLY_IF_STATEMENT    FUNCTION_LOOP_ELIF_STATEMENT
FUNCTION_LOOP_ELIF_STATEMENT
FUNCTION_LOOP_ELSE_STATEMENT -> else { FUNCTION_LOOP_BLOCK }
else FUNCTION_LOOP_LINE
FUNCTION_LOOP_CASE_LINE -> FUNCTION_LOOP_LINE
FUNCTION_LOOP_CASE_BLOCK -> FUNCTION_LOOP_CASE_LINE
FUNCTION_LOOP_CASE_BLOCK FUNCTION_LOOP_CASE_BLOCK {
FUNCTION_LOOP_CASE_BLOCK }
FUNCTION_LOOP_SWITCH_CASE -> case ASSIGNED_EXPRESSION :
FUNCTION_LOOP_CASE_BLOCK    FUNCTION_LOOP_SWITCH_CASE
FUNCTION_LOOP_SWITCH_CASE
FUNCTION_LOOP_SWITCH_DEFAULT -> default :
FUNCTION_LOOP_CASE_BLOCK
FUNCTION_LOOP_SWITCH_BLOCK -> FUNCTION_LOOP_SWITCH_CASE
FUNCTION_LOOP_SWITCH_CASE FUNCTION_LOOP_SWITCH_DEFAULT
FUNCTION_LOOP_SWITCH_STATEMENT -> switch ( ASSIGNED_EXPRESSION
) { FUNCTION_LOOP_SWITCH_BLOCK }
FUNCTION_LOOP_TRY_LINE -> FUNCTION_LOOP_LINE
FUNCTION_LOOP_TRY_BLOCK -> FUNCTION_LOOP_TRY_LINE
FUNCTION_LOOP_TRY_BLOCK FUNCTION_LOOP_TRY_BLOCK {
FUNCTION_LOOP_TRY_BLOCK }
FUNCTION_LOOP_TRY_STATEMENT -> FUNCTION_LOOP_ONLY_TRY_STATEMENT
FUNCTION_LOOP_CATCH_STATEMENT    FUNCTION_LOOP_ONLY_TRY_STATEMENT
FUNCTION_LOOP_CATCH_STATEMENT FUNCTION_LOOP_FINALLY_STATEMENT
FUNCTION_LOOP_ONLY_TRY_STATEMENT

```

```

FUNCTION_LOOP_FINALLY_STATEMENT
FUNCTION_LOOP_ONLY_TRY_STATEMENT -> try {
FUNCTION_LOOP_TRY_BLOCK }
FUNCTION_LOOP_CATCH_STATEMENT -> catch ( VARIABLES ) {
FUNCTION_LOOP_TRY_BLOCK } catch { FUNCTION_LOOP_TRY_BLOCK }
FUNCTION_LOOP_FINALLY_STATEMENT -> finally {
FUNCTION_LOOP_TRY_BLOCK }
TRY_LINE -> LINE    THROW_STATEMENT ;
TRY_BLOCK -> TRY_LINE    TRY_BLOCK TRY_BLOCK    { TRY_BLOCK }
TRY_STATEMENT -> ONLY_TRY_STATEMENT CATCH_STATEMENT
ONLY_TRY_STATEMENT CATCH_STATEMENT FINALLY_STATEMENT
ONLY_TRY_STATEMENT FINALLY_STATEMENT
ONLY_TRY_STATEMENT -> try { TRY_BLOCK }
CATCH_STATEMENT -> catch ( VARIABLES ) { TRY_BLOCK }    catch {
TRY_BLOCK }
FINALLY_STATEMENT -> finally { TRY_BLOCK }
CASE_LINE -> LINE    break ;
CASE_BLOCK -> CASE_LINE    CASE_BLOCK CASE_BLOCK    { CASE_BLOCK }

SWITCH_CASE -> case ASSIGNED_EXPRESSION : CASE_BLOCK
SWITCH_CASE SWITCH_CASE
SWITCH_DEFAULT -> default : CASE_BLOCK
SWITCH_BLOCK -> SWITCH_CASE    SWITCH_CASE SWITCH_DEFAULT
SWITCH_STATEMENT -> switch ( ASSIGNED_EXPRESSION ) {
SWITCH_BLOCK }
CODE -> BODY_BLOCK

```

BAB III

IMPLEMENTASI DAN PENGUJIAN

A. Spesifikasi Teknis Program

a. Gambaran Umum

1. Membuat CodeSplitter
2. Membuat CFG
3. Membuat FA untuk variabel dan ekspresi
4. Mengubah CFG menjadi CNF
5. Memasukan hasil CNF menjadi dictionary
6. Melakukan algoritma CYK dengan menggunakan dictionary berisi CNF untuk mengecek apakah syntax error atau accepted
7. Mengembalikan output syntax error atau accepted

b. Struktur Data

i. Array

1. Terminal

Terminal didefinisikan dalam grammar yang telah dibuat di file CFG.txt. Setelah dibaca oleh fungsi ReadGrammar, terminal akan disimpan dalam sebuah array yang bernama terminals.

Contoh : ['a','b','c']

2. Variables

Variables didefinisikan dalam grammar yang telah dibuat di file CFG.txt. Setelah dibaca oleh fungsi ReadGrammar, variabel akan disimpan dalam sebuah array yang bernama variables.

Contoh : ['CODE','STRING','NUMBER']

3. Production

Production didefinisikan dalam grammar yang telah dibuat di file CFG.txt. Setelah dibaca oleh fungsi ReadGrammar, production akan disimpan dalam sebuah array yang bernama productions dalam bentuk set (himpunan) dengan 2 elemen, elemen pertama menyimpan variabel dan elemen kedua menyimpan production yang diturunkan.

Contoh : [('CODE','BODY_BLOCK'),('BOOLEAN','true')]

ii. Dictionary (Hash Table)

1. Terminal to Variables

Dictionary Terminal to Variables dibuat di dalam prosedur CYK yang ditujukan untuk memudahkan skema pencarian variabel yang menurunkan satu terminal dalam algoritma table filling CYK dari semula

$O(n)$ menjadi $O(1)$. Key dalam dictionary memiliki tipe data string dan value dalam dictionary memiliki tipe data array of string.

Contoh : {'a': ['ALPHABET', 'ASCII_WORD']}

2. Two Variables to Variable

Dictionary Two Variables to Variable dibuat di dalam prosedur CYK yang ditujukan untuk memudahkan skema pencarian variabel yang menurunkan dua variabel dalam algoritma table filling CYK dari semula $O(n)$ menjadi $O(1)$. Key dalam dictionary memiliki tipe data set dan value dalam dictionary memiliki tipe data array of string.

Contoh : {'ALPHABET', 'ALPHABET': ['STRING']}

3. CNF

Dictionary CNF dibuat di dalam convertCFGToCNF untuk memudahkan proses CYK karena diperlukan skema traversal di dalam setiap production yang dibuat sehingga penggunaan dictionary dapat memetakan tiap production lebih rapih dan cepat. Key dalam dictionary memiliki tipe data string dan value dalam dictionary memiliki tipe data array of array of string.

Contoh : {'ALPHABET': [['a'], ['b'], ['c']]}

c. Fungsi

I. Modul GrammarReader

```
def ReadGrammar(relativePath):
```

```
    """
```

```
    Function to read grammar from CFG.txt
```

```
    Args:
```

```
        relativePath (string): relative path to txt
```

```
    Returns:
```

```
        terminals (array): list of terminals
```

```
        variables (array): list of variables
```

```
        productions (array): list of productions
```

```
    """
```

```
def IsEpsilonProd(body):
```

```
    """
```

```
    Function that returns true if a production derivatives  
    epsilon
```

<pre> Args: body (list): list of production Returns: boolean : true if a production derivatives epsilon """ </pre>
<pre> def IsEpsilonVar(currentVar, prodsDict, variables): """ Function that returns true if a variables derivatives epsilon Args: currentVar (string): variable prodsDict (dictionary): dictionary of productions variables (array): list of variables Returns: boolean : true if a variable derivatives epsilon """ </pre>
<pre> def IsNullable(currentVar, prodsDict, variables, processedVar): """ Function that return true if a variable is nullable Args: currentVar (string): variable prodsDict (dictionary): dictionary of production variables (string): list of variables processedVar (array): list of processed variables Returns: boolean : true if a variable is nullable """ </pre>
<pre> def GenerateFromNullable(body, prodsDict, variables): """ Generate New Production From Eliminating Epsilon Args: body (array): production prodsDict (dictionary): dictionary of productions variables (array): list of variables </pre>


```
Returns:
    newBodies (array) : new production
"""
```

```
def EliminateEpsilon(productions, variables):
    """
    Eliminate Epsilon Production

    Args:
        productions (array): list of productions
        variables (array): list of variables

    Returns:
        newProds (array) : new production
    """
```

```
def IsUnitBody(body, variables):
    """
    Function that returns true if a production derivatives
    one variable

    Args:
        body (array): production
        variables (array): list of variables

    Returns:
        boolean : true if production derivatives one
        variable only
    """
```

```
def IsUnitPairs(currentPair, prodsDict, variables,
processedPair):
    """
    Function that returns true if currentPair in unitPairs

    Args:
        currentPair (set): production
        prodsDict (dictionary): dictionary of productions
        variables (array): list of variable
        processedPair (array): list of processed pair

    Returns:
        boolean : true if currentPair in unitPairs
    """
```

<pre> """ </pre>
<pre> def EliminateUnit(productions, variables): """ Function to eliminate unit productions Args: productions (array): list of productions variables (array): list of variables Returns: newProds (array) : list of new productions """ </pre>
<pre> def isDerivateTerminal(production, variables, productions): """ Function that returns true if a variable derivatives one terminal Args: production (array): production variables (array): list of variables productions (array): list of productions Returns: boolean : True if a variable derivatives one terminal """ </pre>
<pre> def eliminateUselessVariable(productions, variables): """ Function to eliminate useless variable Args: productions (array): list of productions variables (array): list of variables Returns: newProds (array) : list of new productions """ </pre>
<pre> def ConvertToDict (productions): """ </pre>

<p>Function to convert list of productions to dictionary</p> <p>Args:</p> <p> productions (array): list of productions</p> <p>Returns:</p> <p> dictionary (dict): dictionary of productions</p> <p>"""</p>
<pre>def ConvertToCNF(productions, variables, terminals): """ Function to convert CFG to CNF Args: productions (array): list of productions variables (array): list of variables terminals (array): list of terminals Returns: result (array): list of new productions """</pre>
<pre>def convertCFGtoCNF(): """ Main Function converting CFG to CNF Returns: productionsFix (dictionary) : dictionary of CNF """</pre>
<p>II. Module CodeSplitter</p>
<pre>def Read_file(inputFile): """ Function to read file Args: inputFile(string): name of file Returns: isiFile(string): content of file """</pre>
<pre>def Code_splitter(inputFile): """</pre>

```

Function to split code into terminals

Args:
    inputFile (string): file name

Returns:
    resultbeneranbangetbanget (array): array of
terminals of string code
"""

```

III. Module CheckVariable

```

def Var_state_1(char):
    """
    State 1 of FA Check Variable

    Args:
        char (char): char of variable

    Returns:
        state (int): next state
    """

```

```

def Var_state_2(char):
    """
    State 2 of FA Check Variable

    Args:
        char (char): char of variable

    Returns:
        state (int): next state
    """

```

```

def Var_state_3(char):
    """
    State 3 of FA Check Variable

    Args:
        char (char): char of variable

    Returns:
        state (int): next state
    """

```

```

def CheckVariable(variable_name):
    """
    Function for FA Check Variable

    Args:
        variable_name (string): variable name

    Returns:
        boolean : True if variable name valid
    """

```

IV. Module CheckExpression

```

def Expression_state_1(char):
    """
    State 1 of FA Check Expression

    Args:
        char (char): char of expression

    Returns:
        state (int): next state
    """

```

```

def Expression_state_2(char):
    """
    State 2 of FA Check Expression

    Args:
        char (char): char of expression

    Returns:
        state (int): next state
    """

```

```

def Expression_state_3(char):
    """
    State 3 of FA Check Expression

    Args:
        char (char): char of expression

    Returns:
        state (int): next state

```

<pre> """ </pre>
<pre> def Expression_state_4(char): """ State 4 of FA Check Expression Args: char (char): char of expression Returns: state (int): next state """ </pre>
<pre> def Expression_state_5(char): """ State 5 of FA Check Expression Args: char (char): char of expression Returns: state (int): next state """ </pre>
<pre> def Expression_state_6(char): """ State 6 of FA Check Expression Args: char (char): char of expression Returns: state (int): next state """ </pre>
<pre> def Expression_state_7(char): """ State 7 of FA Check Expression Args: char (char): char of expression Returns: state (int): next state </pre>

<pre> """ </pre>
<pre> def Expression_state_8(char): """ State 8 of FA Check Expression Args: char (char): char of expression Returns: state (int): next state """ </pre>
<pre> def Expression_state_9(char): """ State 9 of FA Check Expression Args: char (char): char of expression Returns: state (int): next state """ </pre>
<pre> def Expression_state_10(char): """ State 10 of FA Check Expression Args: char (char): char of expression Returns: state (int): next state """ </pre>
<pre> def CheckExpression(expression): """ Function FA to check expression valid or not Args: expression (string) : expression Returns: boolean : True if expression valid </pre>

```
"""
```

d. Prosedur

I. Module CYK

```
def CYK(splittedCode, CNF):  
    """  
    Function algorithm of CYK Table Filling  
  
    Args:  
        splittedCode (array): list of terminals of string  
code  
        CNF (dictionary): dictionary of CNF  
    """
```

e. Antarmuka

Loading...

JavaScript Parser

Made by:

1. Michael Jonathan Halim | 13521124
2. Raynard Tanadi | 13521143
3. Johanee Lee | 13521148

Welcome To Our Javascript Parser!

Please Input Your File Javascript (type EXIT to exit):

Please Input Your File Javascript (type EXIT to exit): inputAcc.js

Parsing...

Accepted

Please Input Your File Javascript (type EXIT to exit):


```
Please Input Your File Javascript (type EXIT to exit): inputReject.js

Parsing...

Syntax Error

Please Input Your File Javascript (type EXIT to exit): 

Please Input Your File Javascript (type EXIT to exit): laporan.js
['const', 'f', 'o', 'o', '=', 'â€œ', 's', 't', 'r', 'i', 'n', 'g', 'â€™', ';']

Parsing...

Terminal not defined

Please Input Your File Javascript (type EXIT to exit): 

Please Input Your File Javascript (type EXIT to exit): EXIT

Thank You! Have A Nice Day :)
```

B. Screenshot dan Analisis Hasil

1. Kasus 1

Input

```
function do_something(x) {
  // This is a sample comment
  if (x == 0) {
    return 0;
  } else if (x + 4 == 1) {
    if (true) {
      return 3;
    } else {
      return 2;
    }
  } else if (x == 32) {
    return 4;
  } else {
    return "Momen";
  }
}
```

Hasil

```
Please Input Your File Javascript (type EXIT to exit): inputAcc.js
Parsing...
Accepted
```

Analisis Kasus :

Input merupakan syntax yang valid.

2. Kasus 2

Input :

```
function do_something(x) {
  // This is a sample multiline comment
  if (x == 0) {
    return 0;
  } else if x + 4 == 1 {
    if (true) {
      return 3;
    } else {
      return 2;
    }
  } else if (x == 32) {
    return 4;
  } else {
    return "Momen";
  }
}
```

Hasil :

```
Please Input Your File Javascript (type EXIT to exit): inputReject.js
Parsing...
Syntax Error
```

Analisis Kasus :

Input merupakan syntax yang tidak valid karena kondisi if harus diapit oleh tanda kurung agar valid. Kesalahan terletak pada `x + 4 == 1`.

3. Kasus 3

Input :

```
// program to check if the string is palindrome or not

function checkPalindrome(string) {

    // find the length of a string
    const len = string.length;

    // loop through half of the string
    for (let i = 0; i < len / 2; i++) {

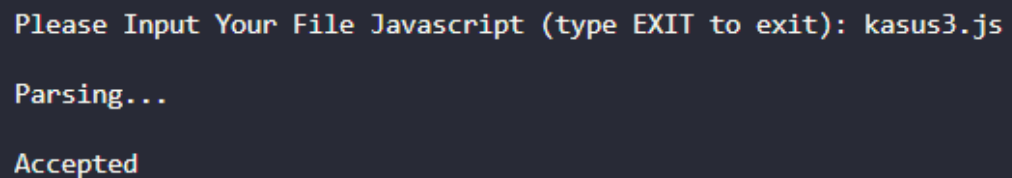
        // check if first and last string are same
        if (string[i] !== string[len - 1 - i]) {
            return 'It is not a palindrome';
        }
    }
    return 'It is a palindrome';
}

// take input
const string = prompt('Enter a string: ');

// call the function
const value = checkPalindrome(string);

console.log(value);
```

Hasil :



```
Please Input Your File Javascript (type EXIT to exit): kasus3.js
Parsing...
Accepted
```

Analisis Kasus :

Input merupakan syntax yang valid.

4. Kasus 4

Input :

```
// program to create JavaScript object using object
literal
const person = {
  name: "John",
  age: 20,
  hobbies: ["reading", "games", "coding"],
  greet: function () {
    console.log("Hello everyone.");
  },
  score: {
    maths: 90,
    science: 80,
  },
};
```

Hasil :

```
Please Input Your File Javascript (type EXIT to exit): kasus4.js
Parsing...
Accepted
```

Analisis Kasus :

Input merupakan syntax yang valid.

5. Kasus 5

Input :

```
// program to create a multiline strings

// using the + operator
const message = 'This is a long message\n' +
```

```
'that spans across multiple lines\n' +  
'in the code.';  
  
console.log(message);
```

Hasil :

```
Please Input Your File Javascript (type EXIT to exit): kasus5.js  
  
Parsing...  
  
Accepted
```

Analisis Kasus :

Input merupakan syntax yang valid.

6. Kasus 6

Input :

```
// program to generate fibonacci series up to a certain  
number  
  
// take input from the user  
const number = 10;  
let n1 = 0, n2 = 1, nextTerm;  
nextTerm = n1 + n2;  
  
while (nextTerm <= number) {  
    n1 = n2;  
    n2 = nextTerm;  
    nextTerm = n1 + n2;  
}  
console.log(nextTerm);
```

Hasil :

```
Please Input Your File Javascript (type EXIT to exit): kasus6.js
Parsing...
Accepted
```

Analisis Kasus :

Input merupakan syntax yang valid.

7. Kasus 7

Input :

```
//JavaScript program to swap two variables

//take input from the users
let a = prompt('Enter the first variable: ');
let b = prompt('Enter the second variable: ');

//create a temporary variable
let temp;

//swap variables
temp = a;
a = b;
b = temp;

console.log(`The value of a after swapping: ${a}`);
console.log(`The value of b after swapping: ${b}`);
```

Hasil :

```
Please Input Your File Javascript (type EXIT to exit): kasus7.js
Parsing...
Syntax Error
```

Analisis Kasus :

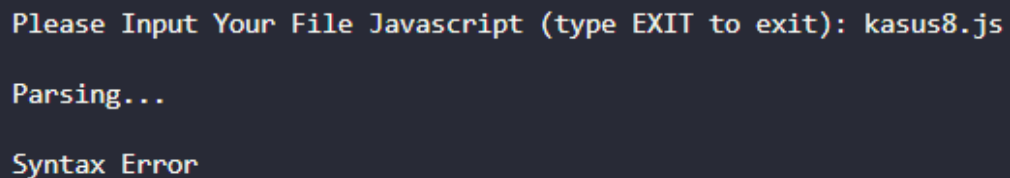
Input merupakan syntax error karena dollar sign dalam string harus memiliki tanda kurung kurawal yang berpasangan sehingga ``${a}`` seharusnya menjadi ``${a}``.

8. Kasus 8

Input :

```
for (let i = 1; i++) {  
    if( number1 % i == 0 && number2 % i == 0) {  
        hcf = i;  
    }  
}
```

Hasil :



```
Please Input Your File Javascript (type EXIT to exit): kasus8.js  
Parsing...  
Syntax Error
```

Analisis Kasus :

Input merupakan syntax error karena dalam for harus memiliki 3 ekspresi sehingga kesalahan terletak pada `for (let i = 1; i++)`.

9. Kasus 9

Input :

```
// program to find the factorial of a number  
function 1factorial(x) {  
  
    // if number is 0  
    if (x == 0) {  
        return 1;  
    }  
  
    // if number is positive  
    else {  
        return x * factorial(x - 1);  
    }  
}
```

```
}  
}
```

Hasil :

```
Please Input Your File Javascript (type EXIT to exit): kasus9.js  
Parsing...  
Syntax Error
```

Analisis Kasus :

Input merupakan syntax error karena nama fungsi tidak boleh diawali angka sehingga kesalahan terletak pada function 1factorial(x).

10. Kasus 10

Input :

```
// program to check if an array contains a specified value  
const array = ['you', 'will', 'learn', 'javascript'];  
const hasValue = array.includes('javascript');  
  
// check the condition  
if(hasValue) {  
    console.log('Array contains a value.');} else {  
    console.log('Array does not contain a value.');}
```

Hasil :

```
Please Input Your File Javascript (type EXIT to exit): kasus10.js  
Parsing...  
Syntax Error
```


Analisis Kasus :

Input merupakan syntax error karena pada array seharusnya diapit oleh kurung siku buka dan tutup sehingga kesalahan terdapat pada `const array = ['you', 'will', 'learn', 'javascript'];`.

BAB IV
LINK REPOSITORY GITHUB DAN PEMBAGIAN TUGAS

A. Link Repository Github

<https://github.com/maikeljh/TubesTBFOParserah>

B. Pembagian Tugas

NIM	Nama	Tugas
13521124	Michael Jonathan Halim	Main Program, FA, CYK, Debugging (Keseluruhan), Laporan
13521143	Raynard Tanadi	Code Splitter, Eliminate Useless Variable, Convert CFG to CNF, Debugging (Keseluruhan), Laporan
13521148	Johannes Lee	Membuat CFG, Baca CFG, Eliminasi E production, Eliminasi Unit production, Debugging (Keseluruhan), Laporan