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

Oleh
Marcel Ryan Antony/13521127
Kenneth Dave Bahana/13521143
Sulthan Dzaky Alfaro/13521159

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

INSTITUT TEKNOLOGI BANDUNG

2022

DAFTAR ISI

DAFTAR ISI **2**

BAB I **3**

A. Finite Automata (FA) 3

B. Context Free Grammar (CFG) 5

Parser 5

C. Chomsky Normal Form (CNF) 6

D. Cocke-Younger-Kasami (CYK) 7

E. JavaScript 8

BAB II **11**

A. Finite Automata (FA) 11

1. FA untuk menentukan apakah penamaan variabel valid atau tidak 11

2. FA untuk menentukan apakah sebuah value valid atau tidak 12

B. Context Free Grammar (CFG) 12

BAB III **17**

A. Spesifikasi Teknis Program 17

a. Gambaran Umum 17

b. Struktur Data 17

c. Fungsi dan Prosedur 17

d. Antarmuka 20

B. Screenshot dan Analisis Hasil (Minimal 3) 21

1. Test 1 21

2. Test 2 21

3. Test 3 22

4. Test 4 23

5. Test 5 24

6. Test 6 24

7. Test 7 25

8. Test 8 26

BAB IV **27**

A. Kesimpulan 27

B. Saran 27

BAB V **28**

A. Link Repository Github 28

B. Pembagian Tugas 28

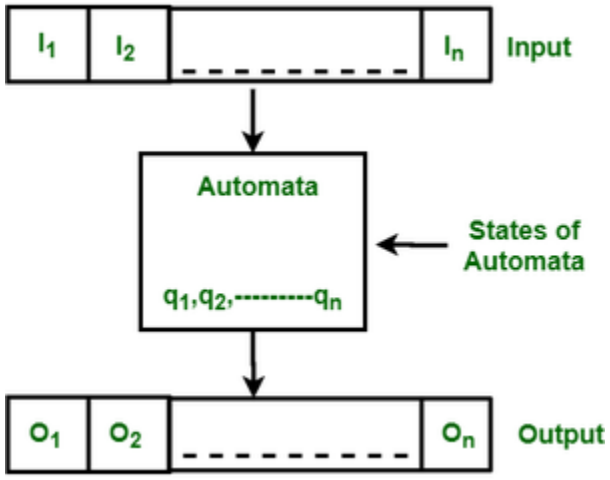
REFERENSI **29**

BAB I
TEORI DASAR

A. Finite Automata (FA)

Finite Automata atau FA adalah mesin yang paling sederhana untuk mengenali pola. FA ini merupakan mesin abstrak yang dapat ditulis sebagai sebuah tuple dengan 5 elemen. FA memiliki sepasang states dan aturan untuk berpindah dari satu state ke state lainnya bergantung juga simbol masukannya.

FA dapat dikatakan sebagai model abstrak dari komputer digital. Bagian bagian penting dalam FA dapat dilihat di gambar dibawah ini



Sumber : geeksforgeeks

Gambar diatas menunjukkan fitur automata berikut

1. Input
2. Output
3. State of Automata
4. State Relation
5. Output Relation

Sebuah FA harus terdapat:

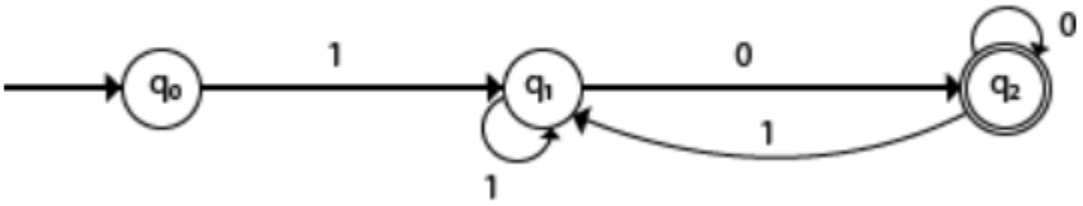
$$M = (Q, \Sigma, \delta, q, F)$$

dengan rincian sebagai berikut

1. Q = Finite set of states
2. Σ = Set dari Input Simbol
3. δ = Transition Function
4. q = Initial State
5. F = Final State

FA dapat dikategorikan menjadi 2 tipe, yaitu DFA (Deterministic Finite Automata) dan juga NFA (Non-deterministic Finite Automata). Dalam sebuah DFA, untuk setiap input yang tersedia,sebuah mesin berpindah tepat ke sebuah state. Fungsi transisi didefinisikan untuk setiap state untuk masing-masing input. Selain itu,dalam DFA tidak terdapat transisi Epsilon. Akibatnya DFA tidak dapat berpindah state jika tidak menerima input apapun.

Contoh DFA dengan input = {0,1} menerima string dengan awalan 1 dan string dengan akhiran 0.



Sumber : JavaTpoint

Yang perlu diingat, bisa terdapat banyak kemungkinan DFA untuk sebuah pattern yang sama. Namun biasanya untuk mendapatkan efisiensi tertinggi, DFA yang jumlah state nya paling sedikit yang akan dipilih.

Selain DFA, ada juga NFA. NFA mirip dengan DFA tetapi ada beberapa aturan tambahan, antara lain.

1. Epsilon diperbolehkan, sehingga dalam NFA kita bisa berpindah ke state lain tanpa menerima input apapun
2. Dapat berpindah ke state lain dengan jumlah berapapun (tidak harus 1) untuk setiap input yang diterima.

Sebenarnya NFA dan DFA adalah ekuivalen. Salah satu hal yang paling penting dalam NFA adalah semua path atau himpunan deret input yang diterima yang mengarah state menyebabkan input tersebut diterima.

Secara matematis, setiap DFA merupakan NFA, tetapi tidak berlaku untuk sebaliknya. Tetapi, NFA dapat dikonversikan menjadi DFA. NFA merupakan FA yang lebih menggunakan konsep teoritis dan DFA biasanya digunakan untuk Analisis Leksikal dalam kompiler. Akan tetapi, untuk semua DFA dan NFA dapat terdapat lebih dari satu final state.

B. Context Free Grammar (CFG)

Context Free Grammar adalah sebuah *grammar* formal yang memiliki aturan produksi dalam bentuk A → α, dengan A adalah simbol *non-terminal* dan α adalah sebuah terminal (α bisa kosong). Secara formal, Context Free Grammar memiliki 4 komponen yaitu, variabel, terminal, *start symbol*, dan aturan produksi. Sebuah Context Free Grammar dapat dituliskan sebagai berikut.

G = (V, T, P, S)

Dengan rincian sebagai berikut :

V : Himpunan Variabel

T : Himpunan Terminal

P : Himpunan aturan produksi

S : Start Symbol

Aturan produksi memiliki bentuk umum "Variabel → *string of variables and terminals"*.

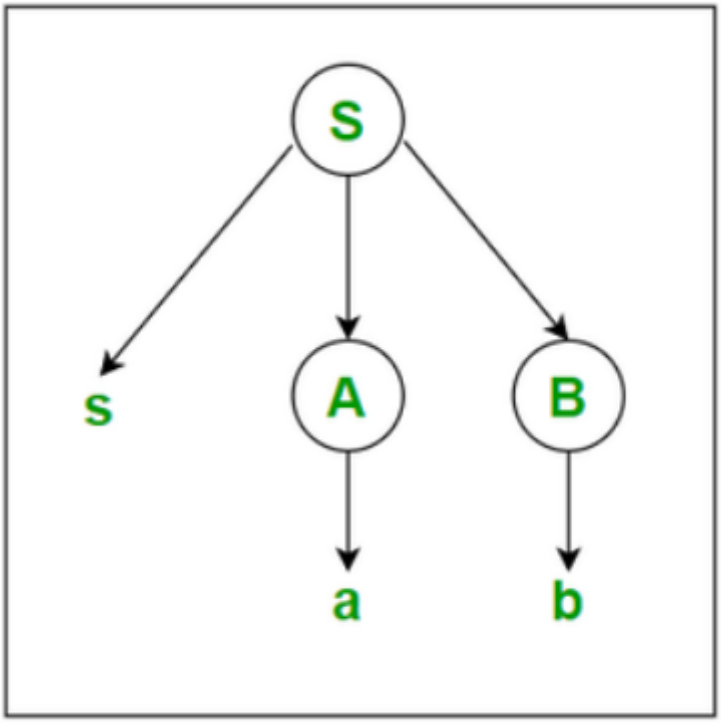
Contoh sebuah CFG : Misalkan ada sebuah CFG berikut ({S}, {a,b}, P, S), dengan S sebagai *start symbol* dan P adalah produksi yang didefinisikan sebagai P = {S → SS | aSb | ε}.

Parser

Salah satu penggunaan CFG adalah untuk pembentukan parser/ untuk menganalisis suatu sintaks. Dalam menganalisis suatu sintaks, dapat digunakan *derivation tree/parse tree* untuk mempermudah penurunan *grammar.* Contoh, terdapat G = ({S, A, B}, {a,b}, P, S) dengan P sebagai berikut :

- S → sAB
- A → a
- B → b

Parse tree dari G apabila input berupa “sab” adalah sebagai berikut :



Gambar 1.2 Parse tree untuk input “sab”

Penggambaran *parse tree* terdapat dua cara, yaitu dengan menggukan *left-most derivation* dan *right-most derivation*. *Left-most derivation* adalah teknik penurunan variabel paling kiri terlebih dahulu, dan *right-most derivation* adalah teknik penurunan variabel paling kanan terlebih dahulu.

C. Chomsky Normal Form (CNF)

Chomsky Normal Form adalah salah satu bentuk Context Free Grammar, yang memiliki aturan
produksi sebagai berikut :

- A → BC, atau
- A → a

Dimana A, B, C adalah sebuah variabel dan a adalah sebuah terminal. Artinya sebuah grammar yang memiliki bentuk CNF haruslah hanya menghasilkan 2 variabel saja atau menghasilkan 1 terminal. Berikut adalah cara untuk mengubah sebuah CFG ke dalam bentuk Chomsky Normal Form :

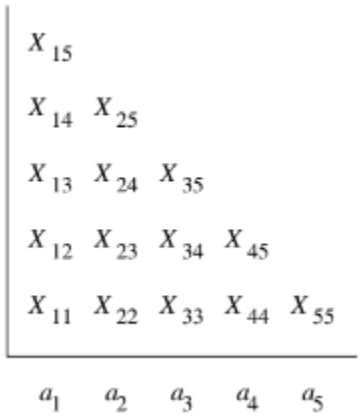
1. Menghapus ε-*productions*

2. Menghapus *unit productions*
3. Menghapus *useless variables*, yaitu
    - Variabel yang tidak menurunkan string terminal
    - Variabel yang tidak dapat dicapai dari *start symbol*
4. Ubah semua aturan produksi ke bentuk berikut :
   $A \rightarrow BC$ atau $A \rightarrow a$

D. Cocke-Younger-Kasami (CYK)

Cocke-Younger-Kasami adalah sebuah algoritma parsing untuk Context Free Grammars dengan syarat sebuah Context Free Grammar telah diubah ke bentuk Chomsky Normal Form. Algoritma Cocke-Younger-Kasami menggunakan pendekatan *bottom-up dynammic programming*. Tujuan dari algoritma Cocke-Younger-Kasami adalah untuk menentukan apakah sebuah string dapat diterima atau tidak oleh sebuah bahasa G.

Cara kerja Algoritma Cocke-Younger-Kasami adalah untuk string dengan panjang N, buatlah sebuah tabel T dengan ukuran NxN. Sebagai contoh apabila terdapat string dengan panjang 5 maka akan terbentuk tabel T sebagai berikut :



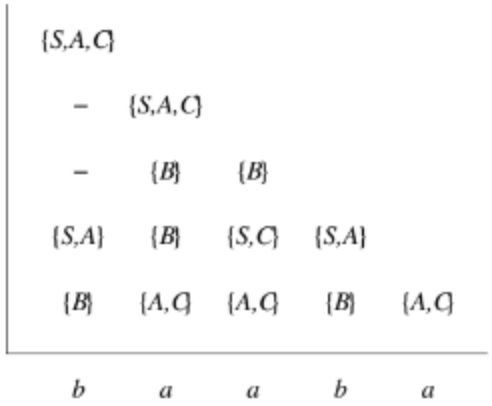
Gambar 1.3.1 Tabel CYK untuk string dengan panjang 5

Pengisian tabel dilakukan dengan cara mengisi baris paling bawah terlebih dahulu dan kemudian naik. Kemudian apabila baris paling atas mengandung *start symbol* berarti string diterima.

Contoh terdapat sebuah bahasa G dalam bentuk CNF sebagai berikut G = ({S, A, B, C}, {a, b}, P, S) dengan P sebagai berikut:

$$S \rightarrow AB \mid BC$$
$$A \rightarrow BA \mid a$$
$$B \rightarrow CC \mid b$$
$$C \rightarrow AB \mid a$$

Untuk input string “baaba” maka akan membentuk tabel CYK sebagai berikut :



Gambar 1.3.2 Tabel CYK dengan input string “baaba” dan bahasa G

Dapat dilihat pada tabel CYK diatas *start symbol* dari G ada di dalam baris teratas sehingga string “baaba” diterima bahasa G.

E. JavaScript

Javascript adalah sebuah bahasa pemrograman tingkat tinggi yang dapat digunakan untuk mengimplementasikan fitur-fitur kompleks kepada web dan juga dapat digunakan untuk kalkulasi, manipulasi, dan validasi data. Bahasa javascript umumnya digunakan untuk pembuatan fitur-fitur kompleks di web seperti apabila suatu tombol di klik di sebuah web maka sesuatu akan terjadi.

Beberapa *syntax* Javascript adalah :

- Sintaks *var* untuk mendeklarasi sebuah variabel
- Sintaks *let* untuk mendeklarasi sebuah blok variabel
- Sintaks *const* untuk mendeklarasi sebuah blok konstanta
- Sintaks *if* untuk memberi syarat pada suatu blok kode
- Sintaks *else if*, digunakan setelah *if*
- Sintaks *else*, digunakan setelah *if* atau *else if*
- Sintaks *switch* untuk menandai sebuah blok berisi pernyataan yang akan dieksekusi dalam kasus yang berbeda
- Sintaks *case* digunakan setelah *switch*
- Sintaks *default* digunakan setelah *switch case* dan digunakan apabila tidak ada kasus yang memenuhi pada *switch*
- Sintaks *throw* digunakan untuk membuat jenis error baru
- Sintaks *for* untuk mengiterasi objek yang dapat diiterasi sampai habis dan untuk menandai sebuah blok kode untuk dieksekusi di loop
- Sintaks *while* untuk mengeksekusi sebuah blok kode selama pernyataan bernilai benar
- Sintaks *break* untuk menghentikan loop pada sebuah blok kode

- Sintaks *continue* digunakan untuk melewatkan 1 iterasi dalam loop
- Sintaks *try* untuk memperbolehkan suatu pengecualian di dalam blok kode untuk nanti diatasi oleh *catch*, sintaks *try* juga memastikan blok kode di dalam sintaks *finally* akan dijalankan
- Sintaks *catch* digunakan setelah *try* berguna untuk menangkap error yang terjadi di *try* dan mengeksekusi blok kode di *catch*
- Sintaks *function* untuk mendeklarasi sebuah fungsi yang nantinya akan mengembalikan sebuah value
- Sintaks *return* digunakan setelah *function* untuk mengembalikan sebuah value dari *function*
- Sintaks *delete* untuk menghapus sebuah properti dari sebuah objek
- Sintaks *false* untuk menandakan sebuah pernyataan salah di blok kode
- Sintaks *true* untuk menandakan sebuah pernyataan benar di blok kode
- Sintaks *null* untuk menandakan sebuah variabel tidak memiliki value
- Sintaks *import* untuk memasukkan modul file lain agar dapat digunakan pada program yang sedang dibuat
- Sintaks *class* untuk membuat class baru

Beberapa operator pada Javascript adalah :

- Operator = untuk meng-assign value ke sebuah variabel
- Operator + untuk menambahkan value
- Operator * untuk mengalikan value
- Operator / untuk membagi value
- Operator ** untuk memangkatkan value
- Operator % untuk mencari sisa pembagian dari value
- Operator == untuk membandingkan apakah sisi kanan dan sisi kiri sama atau tidak
- Operator === untuk membandingkan apakah tipe dan value sisi kanan dan sisi kiri sama atau tidak
- Operator != untuk membandingkan apakah sisi kanan atau sisi kiri berbeda
- Operator !== untuk membandingkan apakah tipe dan value sisi kanan dan sisi kiri berbeda
- Operator > untuk mengecek apakah sisi kiri lebih besar dari sisi kanan
- Operator < untuk mengecek apakah sisi kanan lebih besar dari sisi kiri
- Operator >= untuk mengecek apakah sisi kiri lebih besar atau sama dengan sisi kanan
- Operator <= untuk mengecek apakah sisi kanan lebih besar atau sama dengan sisi kiri
- Operator ? untuk meng-assign sebuah value ke variabel berdasarkan boolean
- Operator && mengembalikan true hanya jika semua kondisi true
- Operator || mengembalikan true jika salah satu kondisi true

- Operator ! untuk mengembalikan *not* dari sebuah kondisi

Selain itu penamaan variabel dalam Javascript memiliki beberapa aturan sebagai berikut :

- Penamaan variabel hanya boleh menggunakan angka, huruf, *underscore*, dan *dollar sign*
- Nama variabel harus diawali huruf atau *underscore* atau *dollar sign*
- Huruf kapital dan huruf tidak kapital dalam variabel dianggap sebagai hal yang berbeda
- Penamaan variabel tidak bisa sama dengan sintaks sintaks yang ada di javascript seperti break, continue, dll.

Contoh penamaan variabel yang valid dalam Javascript :

- _akukamu
- $a21
- a332bi

Contoh penamaan variabel yang tidak valid dalam Javascript :

- 26Lol
- Halo-hai
- Var 1

BAB II
HASIL

A. Finite Automata (FA)

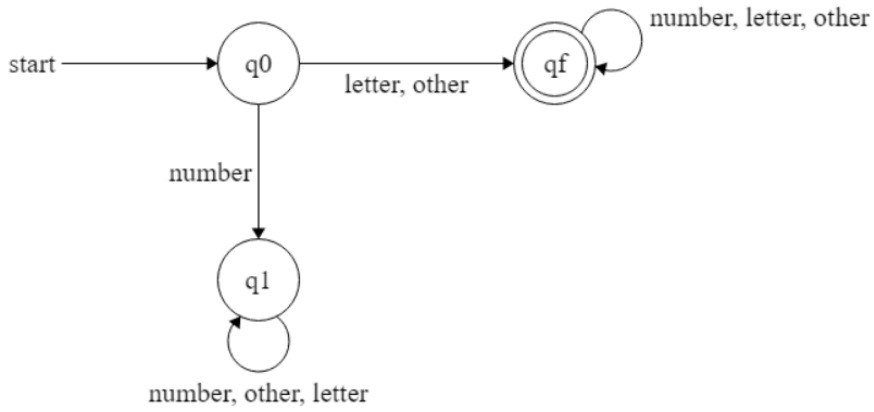
1. FA untuk menentukan apakah penamaan variabel valid atau tidak

FA yang digunakan dalam menentukan penamaan variabel valid atau tidak adalah sebagai berikut :

A = ({q0,q1,qf}, {number, letter, other}, δ, q0, {qf})

Fungsi transisi (δ):
δ (q0, number) = q1
δ (q0, letter) = qf
δ (q0, other) = qf
δ (q1, number) = q1
δ (q1, letter) = q1
δ (q1, other) = q1
δ (qf, number) = qf
δ (qf, letter) = qf
δ (qf, other) = qf

Dengan rincian sebagai berikut :
number = {0,1,2,3,4,5,6,7,8,9}
letter=
{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}
other = {_, $}
FA apabila digambarkan sebagai diagram transisi adalah sebagai berikut:



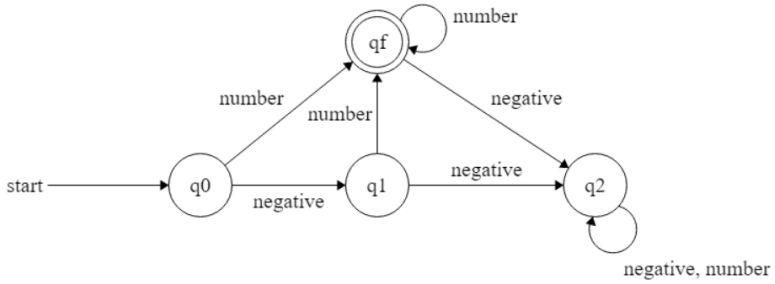
2. FA untuk menentukan apakah sebuah value valid atau tidak

FA yang digunakan dalam menentukan penamaan variabel valid atau tidak adalah sebagai berikut :

A = ({q0,q1,q2,qf}, {number, negative}, δ, q0, {qf})

Fungsi transisi (δ):
δ (q0, number) = qf
δ (q0, negative) = q1
δ (q1, number) = qf
δ (q1, negative) = q2
δ (q2, number) = q2
δ (q2, negative) = q2
δ (qf, number) = qf
δ (qf, negative) = q2

Dengan rincian sebagai berikut :
number = {0,1,2,3,4,5,6,7,8,9}
negative = {-}
FA apabila digambarkan sebagai diagram transisi adalah sebagai berikut:



B. Context Free Grammar (CFG)

Terminals:
( ) { } [ ] . ? , + - * / % = < > ! " ' : ; ~ ^ null true false function if else var delete let const return do while for break continue try catch finally throw new
Variables:
START_STATE START ALPHABET NUM OTHER COMMENT FREE FIRST MIDCHAR NAME VART VAR CONSTT CONST LETT LET VLC POST_NUM NUMBER FVAR FALSET FALSE TRUET TRUE BOOLEAN STRING_IN STRING_VAL STRING ARRAY_VAL ARRAY OBJECT_VAL OBJECT DELETET DELETE RETURNT RETURN BREAKT BREAK FUNCTIONT FUNC_PARAMETER FUNC_NAME FUNC_PARAMETER_CALL FUNC_CALL FUNC_BLOCK FUNCTION CALL_FUNC VALUE INCL DECL A_OP B_OP C_OP OR_OP L_OP T_OP OP ARITH ARITH_OPERATOR ARITH_ALL EQUAL_OPERATOR BOOLEAN_OPERATOR BITWISE_NUMBER BITWISE_OPERATOR TENARY_OPERATOR OPERATORS BLOCK ALLRET_CONDITION ALLBREAKRET_CONDITION ALLB_CONDITION BREAK_STATE BC_STATE RETURN_STATE ALL_STATE FUNC_BREAK_STATE

FUNC_B_STATE FUNC_RETURN_STATE FUNC_ALL_STATE FUNC_STATE EXPRESSION EXPRESSION_STATE EXPRESSION_BODY ELSET ELSEIF_CONDITION ELSE IFT IF_CONDITION IF DOT DO LOOP_BC_STATE LOOP_STATE WHILET WHILE_CONDITION WHILE_BLOCK WHILE_HEADER WHILE FORT FOR_EXPRESSION FOR_BODY FOR_CONDITION FOR_BLOCK FOR NEWT THROWT THROWN THROW_ALL THROW TRY_STATE TRY_BLOCK TRYT TRY_MUST TRY CATCHT CATCH_EXPRESSION CATCH FINALLYT FINALLY CASET CASE NEW_CASE DEFAULTT DEFAULT NEW_DEFAULT SWITCHT SWITCH_STATE SWITCH_BLOCK SWITCH CONDITIONS

Productions:

START_STATE -> START_STATE START_STATE | START ; | CONDITIONS | COMMENT

START -> OPERATORS | VLC | CALL_FUNC | null | DELETE

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

NUM -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

OTHER -> ! | @ | # | $ | % | ^ | & | * | (|) | - | = | _ | + | [|] | \ | { | } | . | ? | < | > | ~ | |

COMMENT -> /* */

FREE -> FREE FREE | FIRST | MIDCHAR | OTHER

FIRST -> ALPHABET | $ | _

MIDCHAR -> MIDCHAR MIDCHAR | NUM | FIRST

NAME -> FIRST MIDCHAR | FIRST | - NAME

VART -> var

VAR -> VART NAME

CONSTT -> const

CONST -> CONSTT NAME

LETT -> let

LET -> LETT NAME

VLC -> VAR | CONST | LET

POST_NUM -> POST_NUM POST_NUM | NUM

NUMBER -> - POST_NUM | POST_NUM

FVAR -> VLC | NAME

FALSET -> false

FALSE -> FALSET | null

TRUET -> true

TRUE -> TRUET

BOOLEAN -> TRUE | FALSE | NAME | NUMBER | FUNCTION | ! BOOLEAN

STRING_IN -> \ "

STRING_VAL -> STRING_VAL STRING_VAL | ALPHABET | NUM | OTHER | OBJECT | START | NAME | FREE | STRING_IN STRING_VAL STRING_IN

STRING -> STRING STRING | ' STRING_VAL ' | " STRING_VAL " | ' ' | " "

ARRAY_VAL -> ARRAY_VAL , ARRAY_VAL | VALUE

ARRAY -> [ARRAY_VAL]

OBJECT_VAL -> OBJECT_VAL , OBJECT_VAL | NAME : VALUE

OBJECT -> { OBJECT_VAL }

DELETET -> delete
DELETE -> DELETET NAME . NAME | DELETET NAME | DELETET NAME [NUM]
RETURNT -> return
RETURN -> RETURNT VALUE ; | RETURNT ; | RETURN (VALUE) ;
BREAKT -> break
BREAK -> BREAKT ;
CONTINUET -> continue
CONTINUE -> CONTINUET ;
FUNCTIONT -> function
FUNC_PARAMETER -> FUNC_PARAMETER , FUNC_PARAMETER | NAME
FUNC_NAME -> NAME (FUNC_PARAMETER) | NAME ( )
FUNC_PARAMETER_CALL -> FUNC_PARAMETER_CALL , FUNC_PARAMETER_CALL
| NAME | VALUE | NAME . NAME
FUNC_CALL -> NAME (FUNC_PARAMETER_CALL) | NAME ( )
FUNC_BLOCK -> { FUNC_STATE } | BLOCK
FUNCTION -> FUNCTIONT FUNC_NAME FUNC_BLOCK
CALL_FUNC -> NAME . FUNC_NAME | FUNC_NAME
VALUE -> NUMBER | FVAR | BOOLEAN_OPERATOR | STRING | ARRAY | FUNC_CALL |
TENARY_OPERATOR | BITWISE_OPERATOR | null
INCL -> NAME + + | + + NAME
DECL -> NAME - - | - - NAME
A_OP -> + | - | % | / | * | * *
B_OP -> & | ^ | ~ | < < | > > | > > > | OR_OP
C_OP -> = = | = = = | ! = | ! = = | > | < | > = | < = | ?
OR_OP -> |
L_OP -> & & | OR_OP OR_OP | !
T_OP -> ?
OP -> A_OP | C_OP | OR_OP | L_OP | T_OP
ARITH -> ARITH A_OP ARITH | VALUE
ARITH_OPERATOR -> FVAR A_OP ARITH
ARITH_ALL -> ARITH | ARITH_OPERATOR
EQUAL_OPERATOR -> FVAR = ARITH_ALL | VALUE = ARITH_ALL | FVAR | VALUE
BOOLEAN_OPERATOR -> BOOLEAN_OPERATOR C_OP BOOLEAN_OPERATOR |
BOOLEAN | VALUE
BITWISE_NUMBER -> FVAR | NUMBER
BITWISE_OPERATOR -> BITWISE_NUMBER B_OP BITWISE_NUMBER
TENARY_OPERATOR -> NAME T_OP VALUE : VALUE | (BOOLEAN_OPERATOR) T_OP
VALUE : VALUE | VALUE T_OP T_OP VALUE | VALUE T_OP T_OP = VALUE
OPERATORS -> ARITH_OPERATOR | EQUAL_OPERATOR | BOOLEAN_OPERATOR |
TENARY OPERATOR | INCL | DECL | BITWISE_OPERATOR
BLOCK -> { START_STATE }
ALLRET_CONDITION -> IF_CONDITION | ELSEIF_CONDITION | ELSET
ALLBREAKRET_CONDITION -> WHILE_CONDITION | FOR_CONDITION
ALLB_CONDITION -> SWITCHT EXPRESSION_BODY

BREAK_STATE -> BREAK_STATE BREAK_STATE | START_STATE | BREAK
BC_STATE -> BC_STATE BC_STATE | START_STATE | BREAK | CONTINUE
RETURN_STATE -> RETURN_STATE RETURN_STATE | START_STATE | RETURN
ALL_STATE -> BC_STATE | RETURN
FUNC_BREAK_STATE -> NEW_CASE | NEW_DEFAULT NEW_CASE | NEW_CASE
NEW_DEFAULT
FUNC_B_STATE -> FUNC_B_STATE FUNC_B_STATE | BREAK_STATE | RETURN |
ALLRET_CONDITION { FUNC_RETURN_STATE } | ALLBREAKRET_CONDITION {
FUNC_ALL_STATE } | ALLB_CONDITION { FUNC_BREAK_STATE }
FUNC_RETURN_STATE -> FUNC_RETURN_STATE FUNC_RETURN_STATE |
RETURN_STATE | THROW | ALLRET_CONDITION { FUNC_RETURN_STATE } |
ALLBREAKRET_CONDITION { FUNC_ALL_STATE } | ALLB_CONDITION {
FUNC_BREAK_STATE }
FUNC_ALL_STATE -> FUNC_ALL_STATE FUNC_ALL_STATE | ALL_STATE | THROW |
ALLBREAKRET_CONDITION { FUNC_ALL_STATE } | ALLRET_CONDITION {
FUNC_ALL_STATE } | ALLB_CONDITION { FUNC_BREAK_STATE }
FUNC_STATE -> FUNC_STATE FUNC_STATE | ALLRET_CONDITION {
FUNC_RETURN_STATE } | ALLBREAKRET_CONDITION { FUNC_ALL_STATE } |
ALLB_CONDITION { FUNC_BREAK_STATE } | FUNC_RETURN_STATE | THROW
EXPRESSION_STATE -> BOOLEAN_OPERATOR | INCL | DECL | EQUAL_OPERATOR |
NAME
EXPRESSION -> (EXPRESSION) | EXPRESSION_STATE
EXPRESSION_BODY -> (EXPRESSION)
ELSET -> else
ELSEIF_CONDITION -> ELSET IFT EXPRESSION_BODY
ELSE -> ELSEIF_CONDITION BLOCK | ELSET BLOCK
IFT -> if
IF_CONDITION -> IFT EXPRESSION_BODY
IF -> IF_CONDITION BLOCK | ELSEIF_CONDITION BLOCK
DOT -> do
DO -> DOT BLOCK
LOOP_BC_STATE -> LOOP_BC_STATE LOOP_BC_STATE | BC_STATE |
ALLRET_CONDITION { LOOP_BC_STATE }
LOOP_STATE -> LOOP_STATE LOOP_STATE | BC_STATE | ALLRET_CONDITION {
LOOP_BC_STATE }
WHILET -> while
WHILE_CONDITION -> WHILET EXPRESSION_BODY
WHILE_BLOCK -> { LOOP_STATE } | BLOCK
WHILE_HEADER -> WHILE_CONDITION WHILE_BLOCK
WHILE -> DO WHILE_CONDITION | WHILE_HEADER
FORT -> for
FOR_EXPRESSION -> OPERATORS
FOR_BODY -> (EXPRESSION ; EXPRESSION ; EXPRESSION)
FOR_CONDITION -> FORT FOR_BODY

FOR_BLOCK -> { LOOP_STATE } | BLOCK
FOR -> FOR_CONDITION FOR_BLOCK | FOR_CONDITION
NEWT -> new
THROWT -> throw
THROWN -> THROWT VALUE
THROW_ALL -> THROWN | THROWT NEWT FUNC_NAME | IF_CONDITION THROWN | ELSEIF_CONDITION THROWN | WHILE_CONDITION THROWN | FOR_CONDITION THROWN
THROW -> THROW_ALL ;
TRY_STATE -> TRY_STATE TRY_STATE | START_STATE | THROW
TRY_BLOCK -> { TRY_STATE }
TRYT -> try
TRY_MUST -> CATCH | CATCH FINALLY
TRY -> TRYT TRY_BLOCK TRY_MUST
CATCHT -> catch
CATCH_EXPRESSION -> CATCHT (NAME)
CATCH -> CATCH_EXPRESSION TRY_BLOCK
FINALLYT -> finally
FINALLY -> FINALLYT TRY_BLOCK
CASET -> case
CASE -> CASE CASE | CASET VALUE : BREAK_STATE
NEW_CASE -> NEW_CASE NEW_CASE | CASET VALUE : FUNC_B_STATE
DEFAULTT -> default
DEFAULT -> DEFAULTT : BREAK_STATE
NEW_DEFAULT -> DEFAULTT : FUNC_B_STATE
SWITCHT -> switch
SWITCH_STATE -> CASE | DEFAULT CASE | CASE DEFAULT
SWITCH_BLOCK -> { SWITCH_STATE }
SWITCH -> SWITCHT EXPRESSION_BODY SWITCH_BLOCK
CONDITIONS -> IF | WHILE | FOR | FUNCTION | TRY | SWITCH

BAB III
IMPLEMENTASI DAN PENGUJIAN

A. Spesifikasi Teknis Program

a. Gambaran Umum

Algoritma parser kami memiliki beberapa langkah utama yaitu :

1. Membuat CFG
2. Mengubah CFG menjadi CNF
3. Mengubah CNF menjadi dictionary
4. Untuk tiap line dari program yang akan dicek, semua akan dipecah sesuai dengan sintaks yang sudah kami buat
5. Lakukan pengecekan sintaks dengan algoritma CYK
6. Hasil dari CYK akan menentukan apakah sintaks dari program valid atau tidak

b. Struktur Data

Untuk langkah 1, CFG kami akan disimpan dalam sebuah file txt. Hal ini kami lakukan agar penyimpanan CFG lebih mudah dan modifikasi serta pembuatan CFG menjadi lebih mudah.

Untuk langkah 2, perubahan CFG menjadi CNF akan dilakukan dengan menggunakan prosedur convertCFG yang ada di file CFGtoCNF.py

Untuk langkah 3, perubahan dari CNF menjadi dictionary juga akan dilakukan di dalam prosedur convertCFG yang ada di file CFGtoCNF.py

Untuk langkah 4, pemisahan untuk tiap line program yang akan di cek akan dilakukan dengan menggunakan prosedur splitcode yang ada di file splitcode.py

Untuk langkah 5 dan 6, akan dilakukan pengecekan dengan menggunakan prosedur Cyk di cyk.py dan setelah itu akan keluar output “Accepted Syntax :D” apabila sintaks program yang di cek valid

c. Fungsi dan Prosedur

Nama	Parameter	Output	Penjelasan
readGrammarFile	file yang akan dibaca	Terminal, Variables, Productions	Pada function ini grammar akan dibaca dan semua terminal, variabel,

			dan aturan produksi akan dimasukkan ke dalam list Terminal, Variables, Productions
isUnit	Rules, Variables	boolean	Function ini digunakan untuk menentukan apakah sebuah aturan produksi bersifat unit (menghasilkan 1 variabel saja)
isOneTerminal	Rules, Variables, Terminal	boolean	Function ini digunakan untuk menentukan apakah sebuah aturan produksi menghasilkan 1 terminal saja
replaceStart	Productions, Variables	Productions	Function ini digunakan untuk mengganti start symbol menjadi S0, berjaga-jaga apabila ada rule yang menghasilkan start symbol sebelumnya
makeDict	Productions, Variables, Terminal	dictionary	Function ini berguna untuk membuat dictionary untuk semua terminal dengan tiap terminal menjadi key dictionary tersebut
replaceTerminal	Productions, Variables, Terminal	Productions	Function ini digunakan untuk mengubah semua aturan produksi yang menghasilkan variabel dan juga terminal contoh : A ->Sa menjadi A ->

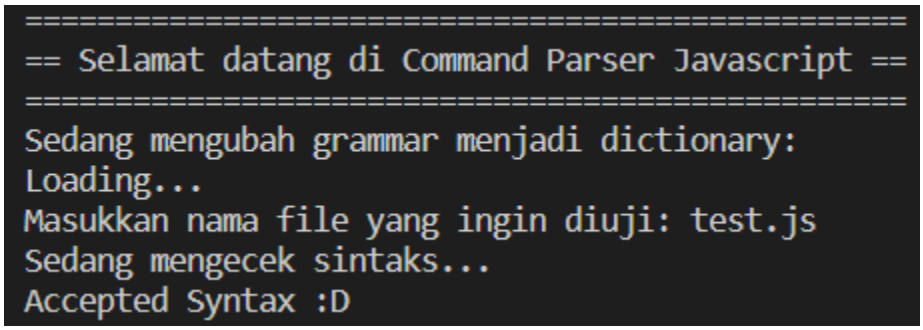
			SZ, Z -> a
remove2PlusVariable	Productions, Variables	Productions	Function ini digunakan untuk menghapus aturan produksi yang menghasilkan 2 variabel lebih menjadi 2 variabel atau 1 terminal saja
pisahUnit	Rules, Variables	Non_unit productions	Function ini berguna untuk memisahkan rule yang bersifat unit (menghasilkan 1 variabel saja) dengan yang tidak
removeUnit	Productions, Variables	Productions	Function ini digunakan untuk menghapus unit production dalam aturan produksi grammar
ProdDict	Productions	prod_dict	Function ini digunakan untuk mengubah aturan produksi menjadi dictionary
convertCFG	-	Productions	Function ini digunakan untuk mengubah sebuah CFG menjadi bentuk CNF dan mengubah bentuk CNF menjadi sebuah dictionary dengan menggunakan fungsi-fungsi yang telah dibuat diatas
splitcode	File yang akan dibaca	List of Operator and variable	Function ini akan membaca file yang nantinya tiap operator dan variable akan

			dipisah dan dimasukkan ke dalam list
CYK	Output splitcode, cnf	String	Function ini merupakan metode untuk mengecek syntax code eror atau accepted dengan menggunakan metode Cocke-Younger-Kasami.

d. Antarmuka

Berikut file-file utama pada program kami :
CFGtoCNF.py
CYK.py
FA.py
splitcode.py
main.py
grammar.txt
Cara menjalankan program kami sangat simpel, cukup dengan melakukan run pada program main.py yang telah kami buat dan masukkan nama file javascript yang ingin di cek sintaksnya.

Jika file javascript yang dicoba merupakan sebuah kode yang valid maka outputnya sebagai berikut



Jika file javascript yang dicoba memiliki sintaks kode yang tidak valid di dalamnya maka akan didapatkan output sebagai berikut

```
================================================
== Selamat datang di Command Parser Javascript ==
================================================
Sedang mengubah grammar menjadi dictionary:
Loading...
Masukkan nama file yang ingin diuji: test4.js
Sedang mengecek sintaks...
Syntax Error D:
```

B. Screenshot dan Analisis Hasil (Minimal 3)

1. Test 1

Berikut isi file test 1

```
function do_something(x) {
  // This is a 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";
  }
}
```

Berikut hasil program untuk file test 1

```
================================================
== Selamat datang di Command Parser Javascript ==
================================================
Sedang mengubah grammar menjadi dictionary:
Loading...
Masukkan nama file yang ingin diuji: test.js
Sedang mengecek sintaks...
Accepted Syntax :D
```

Karena file test 1 sudah memenuhi kriteria sintaks Javascript, maka parser menilai file tersebut merupakan file javascript yang valid

2. Test 2

Berikut isi file test 2

```
function do_something(x){
    // This is a comment
    if (a > 2){
        x = 3;
        return x;
    }
    else if a >= 4{    '('
        return x;
    }
    return "ok";
}
```

Berikut hasil program untuk file test 2

```
================================================
== Selamat datang di Command Parser Javascript ==
================================================
Sedang mengubah grammar menjadi dictionary:
Loading...
Masukkan nama file yang ingin diuji: test2.js
Sedang mengecek sintaks...
Syntax Error D:
```

File test 2 tidak valid karena pernyataan pada else if tidak diberikan tanda kurung. Perbaikan agar sintaks pada file 2 valid adalah dengan menambahkan tanda kurung menjadi else if (a >= 4).

3. Test 3

Berikut isi file test 3

```
function do_something(x){
    This is a comment    U
    if (a > 2){
        x = 3;
        return x;
    }
    else if (a >= 4) {
        return x;
    }
    return "ok";
}
```

Berikut hasil program untuk file test 3

```
================================================
== Selamat datang di Command Parser Javascript ==
================================================
Sedang mengubah grammar menjadi dictionary:
Loading...
Masukkan nama file yang ingin diuji: test2.js
Sedang mengecek sintaks...
Syntax Error D:
```

File test 3 tidak valid dikarenakan terdapat sebuah tulisan yang bukan merupakan sintaks valid pada javascript, yang harusnya berupa komentar apabila dibaca. Karena tidak diberikan operator komentar, maka program tidak valid

4. Test 4

Berikut isi file test 4

```
var x = 4;
var y = 5;
while (x <= 9) {
    for (i = 0; i < 9; i++) {
        y++;
    }
    x++;
}
```

Berikut hasil program untuk file test 4

```
================================================
== Selamat datang di Command Parser Javascript ==
================================================
Sedang mengubah grammar menjadi dictionary:
Loading...
Masukkan nama file yang ingin diuji: test3.js
Sedang mengecek sintaks...
Accepted Syntax :D
```

Karena penggunaan while loop dan for loop pada file test 4 telah memenuhi sintaks javascript, maka parser menilai file tersebut adalah file javascript yang valid.

5. Test 5

Berikut adalah file test 5

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

Berikut hasil program untuk file test 5

```
================================================
== Selamat datang di Command Parser Javascript ==
================================================
Sedang mengubah grammar menjadi dictionary:
Loading...
Masukkan nama file yang ingin diuji: test4.js
Sedang mengecek sintaks...
Syntax Error D:
```

File test 5 tidak valid karena penamaan variabel pada file test 5 memiliki spasi diantaranya yang bukan merupakan sintaks valid javascript. Perbaikan yang dapat membuat file test 5 benar adalah dengan mengganti line diatas menjadi let xy = 3;

6. Test 6

Berikut adalah file test 6

```
let result =0;
try {
  result = add(10,20);
} catch (e) {
  console.log(e.message);
} finally {
  console.log(result);
}
```

Berikut hasil program untuk file test 6

```
================================================
== Selamat datang di Command Parser Javascript ==
================================================
Sedang mengubah grammar menjadi dictionary:
Loading...
Masukkan nama file yang ingin diuji: test6.js
Sedang mengecek sintaks...
Accepted Syntax :D
```

File test 6 sudah valid karena penggunaan try, catch, dan finally sudah sesuai dengan sintaks javascript, maka parser akan menilai file tersebut sebagai file javascript yang valid

7. Test 7

Berikut adalah file test 7

```
function delete_first(x) {
    let x = [1,3];
    delete x[0];
    return x;
}
```

Berikut hasil program untuk file test 7

```
================================================
== Selamat datang di Command Parser Javascript ==
================================================
Sedang mengubah grammar menjadi dictionary:
Loading...
Masukkan nama file yang ingin diuji: test4.js
Sedang mengecek sintaks...
Accepted Syntax :D
```

Pada file test 7 karena penggunaan sintaks-sintaks sudah memenuhi sintaks javascript, maka parser menilai file tersebut sebagai file javascript yang valid

8. Test 8

Berikut adalah file test 8

```
switch( i )
{
    case 3:
        n++;
        break;
    case 0 :
        x++;
        break;
    case 1 :
        p++;
        break;
    default:
        break;
}
```

Berikut adalah hasil program untuk file test 8

```
================================================
== Selamat datang di Command Parser Javascript ==
================================================
Sedang mengubah grammar menjadi dictionary:
Loading...
Masukkan nama file yang ingin diuji: test7.js
Sedang mengecek sintaks...
Accepted Syntax :D
```

Pada file test 8, penggunaan sintaks switch-case telah sesuai dengan sintaks javascript, maka parser akan menilai file sebagai file javascript yang valid

BAB IV
KESIMPULAN DAN SARAN

A. Kesimpulan

Implementasi CFG dan FA untuk parser bahasa pemrograman tertentu sangat sulit untuk dilakukan karena banyaknya aturan produksi yang harus dibuat. Oleh karena itu, program yang kami buat hanya dapat *menghandle* kasus-kasus *basic* untuk kode javascript. Untuk kasus besar, program kami masih mengalami beberapa kendala, bug, dan kesalahan evaluasi dikarenakan pembuatan CFG kami yang masih belum mengatasi segala kemungkinan sintaks pada bahasa Javascript.

B. Saran

Saran untuk kelompok kami sendiri adalah dalam pembuatan parser bahasa pemrograman tertentu, lebih baik lain kali kami lebih fokus membuat CFG nya terlebih dahulu, karena pembuatan CFG yang gagal akan membuat program gagal seluruhnya. Saran untuk tugas besar kali ini adalah akan lebih baik apabila ada referensi tentang pembuatan CFG untuk bahasa pemrograman dan juga beberapa referensi untuk algoritma CNF dan CYK.

BAB V
LINK REPOSITORY GITHUB DAN PEMBAGIAN TUGAS

A. Link Repository Github

https://github.com/MarcelRyan/Tubes-TBFO

B. Pembagian Tugas

NIM	Nama	Tugas
13521127	Marcel Ryan Antony	CFGtoCNF.py, FA.py, laporan
13521145	Kenneth Dave Bahana	Grammar
13521159	Sulthan Dzaky Alfaro	splitcode.py , Cyk.py, laporan

REFERENSI

https://awesomeopensource.com/project/adelmassimo/CFG2CNF
https://www.geeksforgeeks.org/introduction-of-finite-automata/
https://www.w3schools.com/js/js_syntax.asp
https://www.geeksforgeeks.org/cyk-algorithm-for-context-free-grammar/
https://www.geeksforgeeks.org/cocke-younger-kasami-cyk-algorithm/
https://pynative.com/python-regex-split/