Pemodelan *CNF Parser* dengan Memanfaatkan Pohon Biner

Jansen 13510611
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13516011@std.stei.itb.ac.id

***Abstrak*—CNF (*Chomsky Normal Form*) merupakan salah satu *grammar* (tata bahasa) bebas konteks (*context-free)* dalam teori bahasa formal . CNF sendiri dapat digunakan untuk membuat suatu bahasa yang dikenali oleh mesin. Bahasa-bahasa yang dapat dikenali oleh mesin seperti, PASCAL, C/C++, Java, dll. Pada pembuatan CNF, diperlukan produksi CNF untuk dapat mem-*parse* produksi CNF dalam program pengenalan bahasa yang dibuat. Pembuatan *parser* CNF dapat memanfaatkan pohon. Salah satu pemodelan *parser* CNF akan digambarkan oleh penulis pada jurnal ini.**

***Kata kunci*— Chomsky Normal Form, Tata bahasa, Bahasa formal, Pohon.**

I. PENDAHULUAN

Dalam Teori Bahasa Formal, dikenali suatu tata bahasa bebas konteks (*context-free grammar/ CFG)*. CFG merupakan suatu tata bahasa yang dapat digunakan untuk mengenali suatu bahasa. CFG akan sulit digunakan dalam alogritma pengenalan bahasa untuk digunakan dalam program. CFG ternyata dapat ditulis dalam bentuk tertentu. Salah satu penulisan lain dari CFG diperkenalkan oleh *Noam Chomsky*, yaitu CNF (*Chomsky Normal Form*).

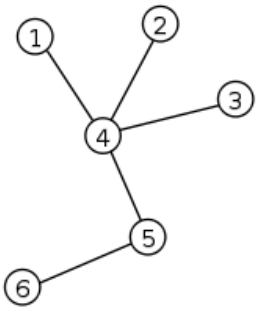
CNF merupakan salah satu *sub-set* dari CFG. Suatu CFG dapat dinyatakan dalam CNF apabila memenuhi tiga syarat. Syarat pertama, CNF tidak dapat memroduksi string kosong (*empty string/* ϵ). Kedua, CNF hanya dapat memroduksi dua variable (*non-terminal*). Ketiga, CNF hanya dapat memroduksi satu terminal. CNF kemudian dipilih sebagai suatu tata bahasa yang lebih mudah untuk diimplementasikan sebagai *parser* yang digunakan untuk mengenali suatu bahasa yang dibuat dalam program.

CNF yang dibuat untuk mengenali suatu bahasa dapat digunakan pohon. Penggunaan pohon lebih efektif karena tidak melibatkan pencarian terus-menerus. Pengenalan bahasa dalam CNF dapat digunakan *parser* secara *bottom-up*. Pendesainan produksi dapat dimodelkan berupa pohon. Pohon yang dihasilkan akan mirip dengan *binary tree* yang hanya memiliki dua cabang (non-terminal) atau satu cabang (terminal) yang bersesuaian dengan CNF yang juga hanya dapat memroduksi satu terminal atau dua non-terminal. Dengan memanfaatkan pohon, *parser* secara *bottom-up* lebih mudah dimengerti oleh program itu sendiri, karena produksi yang memanfaatkan pohon lebih teratur dan mudah untuk ditangani.

II. LANDASAN TEORI

A. Definisi Pohon

Pohon merupakan suatu graf tak-berarah yang tidak memiliki kalang. Pohon telah digunakan sejak tahun 1857 oleh matematikawan Inggris bernama Arthur Cayley. Cayley menggunakan pohon untuk menghitung tipe zat-zat kimiawi tertentu. Sama dengan *Graf*, pohon juga direpresentasikan dalam bentuk $T = (V, E)$. Pohon memiliki *vertex* yang dapat dinyatakan sebagai daratan, titik, atau simpul. Pohon juga memiliki *edge* yang dapat dinyatakan sebagai jembatan, sisi atau garis yang menghubungkan dua simpul.



Gambar 1 Pohon (Teori graf)
(Sumber: *en.wikipedia.org*)

Pada *gambar 1*, merupakan gambaran pohon yang memiliki 6 simpul dan 5 sisi.

B. Sifat-sifat Pohon

Suatu graf dapat dikatakan sebagai pohon apabil memenuhi sifat/ karakterisitik tertentu. Sifat-sifat pohon diuraikan sebagai berikut.

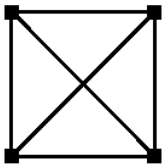
1. Suatu graf tak-berarah dikatakan pohon apabila setiap pasang simpul pada graf terhubung unik dengan satu lintasan.
2. Suatu graf terhubung memiliki $m = n - 1$ sisi.
3. Suatu graf tidak memiliki sirkuit dan penambahan satu sisi pada grad akan membuat hanya satu sirkuit.
4. Suatu graf tidak memiliki sirkuit dan memiliki$m = n - 1$ sisi.
5. Suatu graf terhubung den semua sisinya dapat dinyatakan sebagai jembatan.

C. Jenis-jenis Pohon

1. Pohon Merentang (*Spanning Tree*)

Pohon merentang dari suatu graf terhubung merupakan upagraf (*sub-graph*) merentang yang berupa pohon. Pohon merentang dapat diperoleh dengan memutuskan sirkuit pada suatu graf.

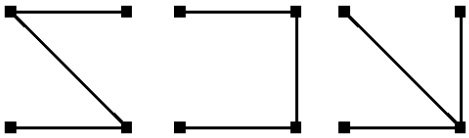
Misalkan suatu graf $G = (4,6)$, dapat dibentuk beberapa graf lainnya yang berupa pohon.



Gambar 2 Graf $G = (4,6)$

Pada graf *G*, akan dibentuk menjadi pohon dengan memutuskan sirkuit pada graf *G*. Sesuai dengan sifat pohon kedua, maka pohon dengan simpul $n = 4$, akan tepat memiliki sisi $m = n - 1 = 3$.

Pada saat pemutusan sirkuit selesai dilakukan, akan terdapat banyak bentuk dari pohon.

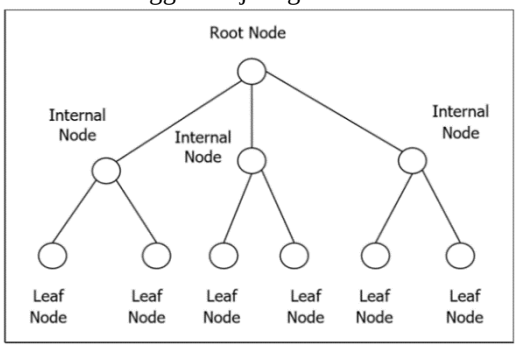


Gambar 3 Bentuk pohon dari graf *G* dengan 4 simpul

Setiap graf terhubung mempunyai paling sedikit satu buah pohon merentang.

2. Pohon Berakar (*Rooted Tree*)

Pohon berakar merupakan pohon yang satu buah simpulnya diperlakukan sebagai akar dan sisi-sisinya diberi arah sehingga menjadi graf berarah.



Gambar 4 Pohon berakar
(Sumber: *tutorialspoint.com*)

Pada pohon berakar, terdapat beberapa terminology, diantaranya:

a. Anak dan Orangtua
Suatu simpul yang menghasilkan simpul lain disebut sebagai orangtua (*parent*) dan simpul yang dihasilkan disebut anak (*child/ children*)

b. Lintasan
Lintasan adalah garis yang menghubungkan satu simpul dengan simpul yang lain.

c. Saudara kandung.
Saudara kandung merupakan istilah untuk simpul yang memiliki *parent* yang sama.

d. Upapohon
Upapohon merupakan bagian dari pohon.

e. Derajat
Derajat suatu simpul adalah jumlah dari upapohon (jumlah anak) dari suatu simpul tersebut.

f. Daun
Daun merupakan simpul yang berderajat nol (tidak mempunyai anak).

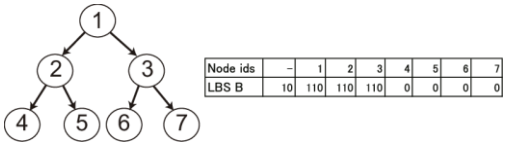
g. Simpul dalam
Simpul dalam merupakan simpul yang memiliki anak.

h. Aras (*level*)
Akar pohon utama berderajat nol, simpul yang dihasilkan berurutan berderajat lebih satu dari derajat sebelumnya.

i. Tinggi atau Kedalaman
Tinggi atau kedalaman suatu pohon dapat dilihat dari aras maksimum suatu pohon.

3. Pohon Terurut (*Orderd Tree*)

Pohon terurut merupakan pohon berakar yang urutan anak-anaknya penting.

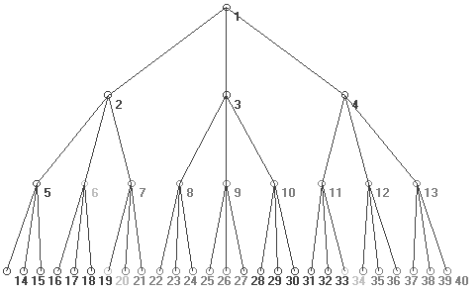


Node ids	–	1	2	3	4	5	6	7
LBS B	10	110	110	110	0	0	0	0

Gambar 5 Pohon terurut
(Sumber: *researchgate.net*)

4. Pohon *n-ary*

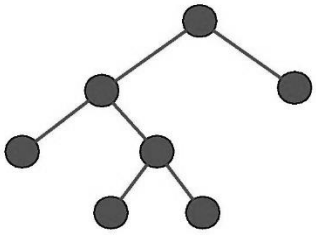
Pohon *n-ary* merupakan pohon berakar yang memiliki paling banyak *n* anak pada setiap simpulnya. Pohon *n-ary* dikatakan teratur atau penuh (*full*) jika setiap orangtuanya memiliki tepat *n* anak.



Gambar 6 Pohon *n-ary* dengan *level* 3
(Sumber: *dembinhyen.free.fr*)

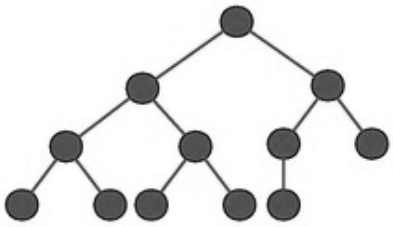
5. Pohon biner (*Binary Tre)*

Pohon biner merupakan pohon *n-ary* dengan $n = 2$. Setiap simpul pada pohon biner mempunyai paling banyak dua anak. Anak pada pohon biner dibedakan menjadi anak kiri (*left child*) dan anak kanan *(right child*). Pohon biner juga merupakan pohon terurut karena memiliki perbedaan urutan anak.



Gambar 7 Pohon biner penuh
(Sumber: *en.wikipedia.org*)

Pada pohon biner, terdapat istilah pohon biner seimbang. Pohon biner seimbang adalah kondisi pada saat tinggi upapohon kiri dan tinggi upapohon kanan seimbang. Perbedaan ketinggian upapohon maksimal bernilai satu.



Gambar 8 Pohon biner seimbang
(Sumber: *en.wikipedia.org*)

D. Definisi CNF (Chomsky Normal Form)

CNF memiliki bentuk yang sama seperti CFG. CNF memiliki 4 tuple, yang dapat dituliskan:

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

Ide dasar dalam CFG/CNF adalah dengan menggunakan variable untuk menyatakan suatu himpunan string. Variable tersebut dapat dinyatakan secara rekursif. Aturan dari rekursif yang digunakan dalam produksi hanya melibatkan konkatenasi. Aturan lain dari variable yaitu dengan menggabungkan (union).

Variable (non-terminal) merupakan suatu himpunan terbatas yang terdiri dari simbol yang setiap simbol menyatakan suatu tata bahasa. Terminal adalah suatu simbol dari alfabet dari bahasa yang dinyatakan. Produksi merupakan derivasi dari suatu variable ke variable lain atau terminal. Start symbol adalah variable pertama yang digunakan untuk memulai membaca suatu bahasa.

Chomsky Normal Form (CNF) merupakan suatu tata bahasa perkembangan dari *Context-free Grammar* (CFG). Suatu CFG $G = (V, \Sigma, R, S)$ dapat dikatakan sebagai bentuk CNF apabila setiap produksi *R* memenuhi salah satu bentuk berikut:

1. $A \rightarrow BC$, $A, B, C \in V, B \neq S$, dan $C \neq S$.
2. $A \rightarrow a$, $A \in V$, dan $a \in \Sigma$.
3. $S \rightarrow \epsilon$, S merupakan *start-symbol*.

Pada CNF terdapat suatu teorema yang dinyatakan sebagai berikut:

"*Misalkan Σ merupakan suatu alfabet dan $L \subseteq \Sigma*$ merupakan CFG. Pasti terdapat suatu* CFG *dalam bentuk CNF dengan bahasa L*".

E. Lexical Analysis (Tokenization)

Lexical Analysis (*tokenization*) merupakan suatu proses mengubah suatu kalimat/ masukan menjadi token-token. Program yang melakukan *tokenization*/ *lexical analysis* disebut sebagai *lexer/ tokenizer*. Suatu *lexer* secara umum digabungkan dengan *parser*.

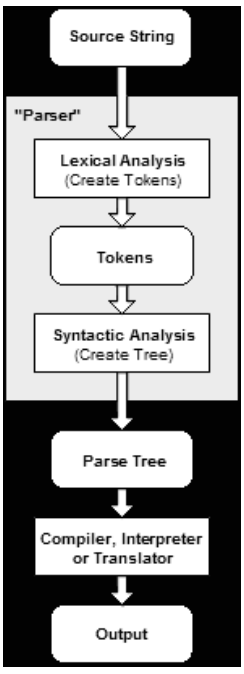
Suatu *token* didapat dinyatakan sebagai pasangan yang mengandung nake token dan nilai token. Nama token merupakan kategori dati *lexical unit*.

Pada CNF, suatu token dapat dibuat dari terminal (daun dari suatu cabang pohon) CFG/CNF.

F. Parser CNF

Parser adalah sebuah komponen perangkat lunak yang menerima masukan data (umumnya teks) dan membentuk suatu struktur data. Parsing data dapat dilakukan dengan cara lain, atau digabungkan menjadi hanya satu langkah. Parser yang dibuat umumnya dilakukan dengan membaca token yang telah diubah dari suatu string tertentu.

Proses dari suatu parsing dapat digambarkan sebagai berikut:



Gambar 9 Diagram *Parser*
(Sumber: *en.wikipedia.org*)

Tahap pertama adalah men-*generate* string input menjadi token (*tokenization*) atau dikenal dengan istilah *lexical analysis*. Pengubahan token ini dilakukan dengan membagi string input menjadi suatu simbol unik yang didefinisikan oleh tata bahasa. *Lexical analysis* kemudian akan menghasilkan keluaran berupa token.

Tahap berikutnya adalah *parsing* atau dikenal dengan *syntactic analysis*. Parsing ini mengecek apakah token membentuk suatu ekspresi yang diterima oleh tata bahasa tersebut atau tidak. Token yang diperiksa dapat berupa *terminal* pada CFG/CNF yang kemudian di-*parse* sesuai dengan kondisi yang memenuhi.

Tahap akhir adalah *semantic parsing* atau Analisa. Tahap ini akan menghasilkan keluaran apakah input *valid* atau tidak dan melakukan aksi-aksi bersesuaian dengan keluarannya.

G. Jenis-jenis Parser

Fungsi parser secara umum adalah untuk menyatakan bagaimana suatu masukan dapat diturunkan dari *start-symbol* suatu dari suatu tata bahasa. Parser secara umum dapat dilakukan dengan dua cara:

1. *Top-down Parsing*
   Top-down parsing dapat dilakukan dengan mencari turunan paling kiri (*left-most derivations*) dari suatu masukan string. Pencarian dilakukan dalam pohon *parse* dengan menggunakan ekspansi *top-down* dari aturan tata bahasa formal.
   Pada *top-down* parsing, pembacaan token dimulai dari membaca token paling kiri ke kanan menuju ke *start-symbol*.

2. *Bottom-up Parsing*
   Suatu *parser* dapat dimulai dari input awal dan manulis ulang hingga menuju *start-symbol*. *Parser* digunakan untuk menglokalisasi suatu elemen yang paling dasar. Salah satu *parser* yang menggunakan *bottom-up parsing* adalah *LR parser*.

III. METODOLOGI

A. Merancang *Lexer/ Tokenizer*

Pada pembuatan parser, terlebih dahulu harus mendesain *tokenizer/ lexer*. tujuan dari pembuatan *lexer* adalah untuk memudahkan pembacaan input dari suatu string. Berikut adalah *pseudo-code* pada *lexer*.

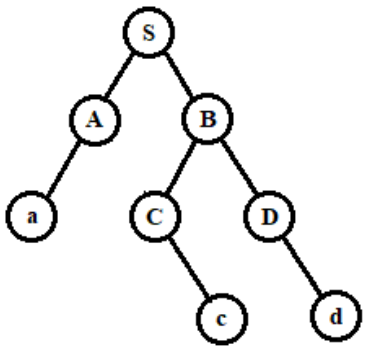
```
lengthToken <- 0
while not eof(files) do
  input(files,s);
  IgnoreBlank(s);
  for each i = 1 to s.length()
    if (s condititon to split) then
       lengthToken <- lengthToken + 1
    make a new token
    else
    add s[i] to current token
```

Pada awalnya panjang token diinisialisasi dengan 0 untuk menandakan sebagai banyaknya token yang ada dalam string input. Kemudian membaca token dari file eksternal. Apabila pada file belum *NULL*, maka pembacaan akan terus dilakukan hingga *NULL*. Pembacaan file dilakukan dalam satu baris, kemudian disimpan dalam suatu *variable 's'*. String yang ditampung dalam s kemudian dibaca karakter per karakter. Apabila dalam karakter tersebut terdapat karakter-karakter tertentu yang menyatakan kondisi untuk "menghentikan" pembacaan, maka panjang token akan ditambah satu, kemudian string yang telah ditampung pada suatu variabel akan ditampung pada token bertipe array string. Apabila kondisi untuk "menghentikan" pembacaan belum terbaca, maka akan karakter yang dibaca akan ditambahkan pada string token yang sedang dibaca.

B. Merancang Parser CNF

Setelanh perancangan *lexer* selesai dilakukan, berikutnya merancang *parser* CNF yang akan digunakan dalam *parsing*. Pada kasus ini, CNF yang dirancang dapat dibilang cukup unik, dan tetap memenuhi kondisi CNF.

Salah satu cara perancangan *parser* CNF dengan memanfaatkan pohon biner (*binary tree*).



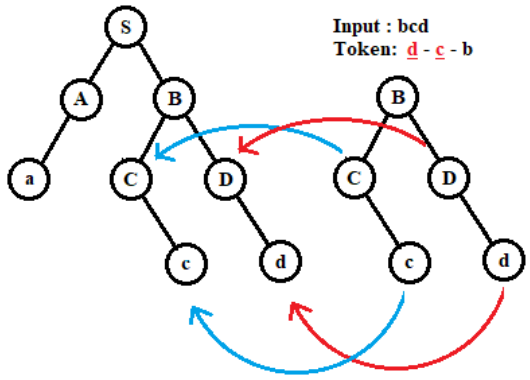
Gambar 10 Pemodelan *parser* CNF dengan menggunakan binary tree

Pemodelan CNF ini harus unik dan tidak boleh ada lebih dari satu variabel yang memroduksi terminal maupun non-terminal yang sama.

Setelah perancangan *parser* CNF selesai dilakukan, *parser* akan lebih mudah mencari produksi yang dibuat. Ide ini didasari dengan mengambil dua token terakhir, apabila suatu produksi menghasilkan dua terminal yang sama, maka produksi itu disimpan untuk kemudian dibandingkan lagi dengan terminal berikutnya. Apabila tidak ditemukan produksi yang sesuai, maka bahasa itu tidak *valid*.

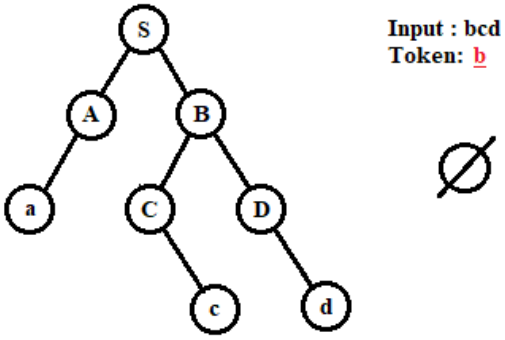
Misalkan suatu CNF:

```
S -> AB
B -> CD
A -> a
C -> c
D -> d
```



Gambar 11 Ilustrasi *parsing token* (1)

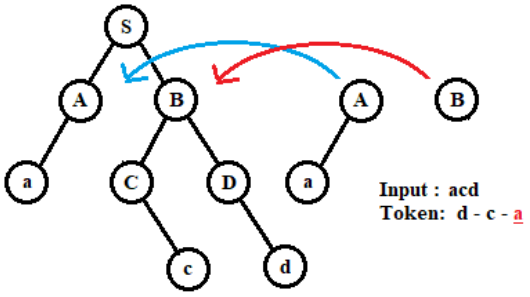
Karena input dua token sama dengan *main-tree* maka hasil akan disimpan variabel produksi yang menghasilan C dan D, yaitu B. Kemudian akan dilakukan pembacaan token berikutnya lagi untuk dibandingkan dengan produksi berikutnya.



Gambar 12 Ilustrasi *parsing token* (2)

Pada saat pencarian, tidak ditemukan produksi yang menghasilkan terminal 'b'. Oleh karena itu program dapat menyimpulkan bahwa masukan tidak *valid*.

Sebaliknya apabila masukan yang diterima 'acd', maka program akan mengecek dua token terakhir dari masukan (sama seperti gambar 11). Kemudian diambil lagi token berikutnya untuk dibandingkan.



Gambar 13 Ilustrasi *parsing token* (3)

Pada kasus diatas pada saat membandingkan variabel A dengan variabel B, ditemukan suatu variabel S yang memroduksi A dan B, yaitu S (*start-symbol*). Karena posisi produksi sekarang berada pada *start-symbol* dan tidak ada lagi token, maka *parsing* telah selesai dilakukan dan kondisi *valid* telah terpenuhi. Oleh karena itu, string '*acd*' *valid* sesuai dengan tata bahasa yang dirancang.

CNF yang dirancang cukup unik, karena anak paling kiri (*left-child*) dari suatu simpul akan selalu menghasilkan terminal, sedangkan anak paling kanan (*right-child*) dari suatu simpul akan menghasilkan dua variabel lainnya yang juga unik. Anak paling kanan dari suatu simpul juga dapat men-*derive* suatu terminal untuk kondisi terminal akhir dalam suatu bahasa.

Algoritma yang dirancang akan mengecek seluruh produksi sekarang dengan terminal berikutnya. Apabila terdapat produksi yang menghasilkan *left-child* dan *right-child* yang bersesuaian, maka produksi tersebut disimpan sebagai produksi sekarang dan membandingkannya lagi dengan token berikutnya, hingga *error* atau mencapai *start-symbol* dan token kosong.

Pada saat *error*, maka bahasa tersebut tidak dikenali, sedangkan pada saat token kosong dan sudah mencapai simbol awal, maka pencarian selesai dan bahasa tersebut dikenali oleh tata bahasa yang telah dibuat (masukan string *valid*).

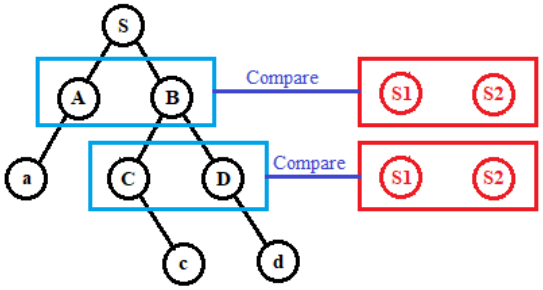
Berikut tampilan *pseudo-code parser*:

```
  //mencari variable yang menghasilkan terminal S
  function TermVar(S: string; T: Term): string

  //membandingkan variabel[terminal] (S1) dengan variabel (S2)
  function lookVar(S1, S2: string; V: NonTerm): string;

  for each i = 1 to N
  //N total productions
    lookVar(S1,S2)
    //compare S1 and S2, looking for a variable which derives both S1 and S2
  if not found then
    input invalid
  else
    S1 <- next(terminal)
    TermVar(S1)
    S2 <- Variable which derives S1,S2
    input valid
```

Berdasarkan algorima yang telah dibuat, program hanya akan membandingkan satu terminal dan satu variabel. Karena produksi pada CNF bersifat unik, maka tepat hanya satu variabel yang akan dihasilkan. Sehingga program akan berjalan lebih efisien. Kompleksitas pada pencarian dengan membandingkan terminal dan variabel menjadi linier sesuai dengan banyaknya produksi CNF yang dibuat. Kompleksitas pada program pencarian produksi adalah $O(n)$.



Gambar 14 Ilustrasi perbandingan dua variabel

Dengan algoritma yang dibuat, maka hanya ada dua opsi, terdapat variabel atau tidak. Jika tidak terdapat, maka program langsung menampilkan *error message*. Jika terdapat maka program melanjutkan ke proses berikutnya hingga mencapai simbol awal.

IV. KESIMPULAN

Dalam merancang CNF, dapat digunakan pohon sehingga CNF yang dirancang unik. Pemodelan *parser CNF* yang dirancang dapat digunakan teori pohon sebagai alat bantu dalam mem-*parse* string yang telah diubah menjadi *token* oleh *lexer/ tokenizer*. Pencarian dengan pohon akan lebih efektif karena hanya membandingkan variabel[terminal] dan variable dengan anak paling kanan (*right child*) dan anak paling kiri (*left child*) pada suatu tree. Pada saat diperoleh kesamaan, maka akan langsung mengambil *parent* (simpul yang menghasilkan *right* dan *left child*). Pencarian akan berhenti apabila ditemukan simbol awal dan token kosong atau tidak ditemukan suatu parent dari *right child* dan *left child*. Kompleksitas waktu yang dimakan juga lebih sedikit yaitu $O(n)$.

VII. UCAPAN TERIMA KASIH

Pertama-tama penulis mengucapkan puji syukur kepada Tuhan Yang Maha Esa. Atas berkat, rahmat dan karunia-Nya penulis dapat menyelesaikan makalah yang berjudul "*Pemodelan CNF Parser dengan Memanfaatkan Pohon Biner"*. Penulis juga berterima kasih kepada kedua orangtua yang telah membimbing penulis dan memberi kesempatan bagi penulis untuk melanjutkan studi di Institut Teknologi Bandung. Selanjutnya penulis berterima kasih kepada dosen yang memberi tugas ini, Dr. Ir. Rinaldi Munir, M.T., dan juga kepada dosen pengajar yaitu Dra. Harlili S., M.Sc. Atas pengajaran beliau kepada penulis mengenai matematika diskrit, penulis mampu menulis makalah ini dengan baik. Penulis juga berterima kasih kepada teman-teman yang telah memberi semangat dan juga mendukung penulis.

REFERENSI

[1] *https://en.wikipedia.org/wiki/Chomsky_normal_form/* diakses pada tanggal 1 Desember 2017 pukul 09.19.

[2] Maheshwari, A., & Smid, M., "*Introduction to Theory of Computation*", Ottwawa, 2014, ch 3.

[3] Rosen, Kenneth.H, "*Discrete Mathematics and Its Applications – 7th ed.*", New York: McGraw-Hill, 2012, ch. 11.

[4] *https://en.wikipedia.org/wiki/Parsing* diakses pada tanggal 2 Desember 2017 pukul 15.29.

[5] *https://en.wikipedia.org/wiki/Lexical_analysis* diakses pada tanggal 2 Desember 2017 pukul 17.48

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 2 Desember 2017

Jansen - 13516011