

SKRIPSI

*MORPHOLOGICAL PARSER* UNTUK TEKS DALAM  
BAHASA INDONESIA



Andreas Novian Dwi Triastanto

NPM: 2013730038

PROGRAM STUDI TEKNIK INFORMATIKA  
FAKULTAS TEKNOLOGI INFORMASI DAN SAINS  
UNIVERSITAS KATOLIK PARAHYANGAN  
2017



**UNDERGRADUATE THESIS**

**MORPHOLOGICAL PARSER FOR TEXTS IN BAHASA  
INDONESIA**



**Andreas Novian Dwi Triastanto**

**NPM: 2013730038**

**DEPARTMENT OF INFORMATICS  
FACULTY OF INFORMATION TECHNOLOGY AND SCIENCES  
PARAHYANGAN CATHOLIC UNIVERSITY  
2017**



## ABSTRAK

Morphological parser adalah salah satu alat dalam pengolahan bahasa alami yang berfungsi untuk membagi sebuah kata menjadi komponen-komponen penyusunnya, seperti kata dasar, awalan, akhiran, serta dapat mengenali jika kata tersebut merupakan kata ulang maupun kata majemuk. Proses ini dapat membantu mengurangi ambiguitas selama proses mengetahui makna suatu kalimat. Perangkat lunak yang dibuat pada penelitian ini dapat mengenali kata dalam bahasa Indonesia yang dibentuk dari proses morfologi berupa afiksasi, reduplikasi, dan komposisi. Leksikon dibutuhkan dalam perangkat lunak untuk menyimpan semua kata dasar dan kata turunan yang valid dalam bahasa Indonesia. Hasil dari penelitian ini menunjukkan bahwa perangkat lunak telah mampu untuk melakukan morphological parsing pada kata dalam bahasa Indonesia dan mampu menghasilkan semua kemungkinan parsing dari sebuah kata. Perangkat lunak dapat melakukan proses parsing untuk teks yang terdiri dari 100 kata dalam waktu rata-rata kurang dari 100 milidetik.

**Kata-kata kunci:** Morphological Parser, Morphological Parsing, Pengolahan Bahasa Alami, Bahasa Indonesia, Leksikon



## **ABSTRACT**

Morphological parser is one of the tools used in natural language processing to divide a word into its components, such as root, prefix, suffix, also can recognize if the word is a reduplication or compound word. This process can help reducing the ambiguity during the process of understanding the meaning of a sentence. The software built in this research can recognize a word in bahasa Indonesia that formed from the morphological process of affixation, reduplication, and composition. Lexicon is needed in the software to keep all the valid root words and their derivatives in bahasa Indonesia. The result of this research shows that the software is capable to do morphological parsing on words in bahasa Indonesia and capable of producing all possible parsing of a word. The software capable of doing parsing to text consisting of 100 words in less than 100 millisecond.

**Keywords:** Morphological Parser, Morphological Parsing, Natural Language Processing, Bahasa Indonesia, Lexicon





# DAFTAR ISI

<b>DAFTAR ISI</b>	<b>ix</b>
<b>DAFTAR GAMBAR</b>	<b>xi</b>
<b>DAFTAR TABEL</b>	<b>xiii</b>
<b>1 PENDAHULUAN</b>	<b>1</b>
1.1 Latar Belakang . . . . .	1
1.2 Rumusan Masalah . . . . .	2
1.3 Tujuan . . . . .	2
1.4 Batasan Masalah . . . . .	2
1.5 Metodologi Penelitian . . . . .	2
1.6 Sistematika Pembahasan . . . . .	3
<b>2 DASAR TEORI</b>	<b>5</b>
2.1 Morfologi . . . . .	5
2.2 Morfem . . . . .	6
2.2.1 Identifikasi Morfem . . . . .	6
2.2.2 Jenis Morfem . . . . .	8
2.2.3 Morfem Dasar, Bentuk Dasar, Akar, dan Leksem . . . . .	10
2.2.4 Morfem Afiks . . . . .	11
2.3 Proses Morfologi . . . . .	12
2.3.1 Bentuk Dasar . . . . .	12
2.3.2 Pembentuk Kata . . . . .	13
2.4 Morfofonemik . . . . .	14
2.4.1 Jenis Perubahan . . . . .	14
2.4.2 Prefiksasi ber- . . . . .	15
2.4.3 Prefiksasi me- (termasuk klofiks me-kan dan me-i) . . . . .	15
2.4.4 Prefiksasi pe- dan konfiksasi pe-an . . . . .	17
2.4.5 Prefiksasi per- dan konfiksasi per-an . . . . .	18
2.4.6 Prefiksasi ter- . . . . .	18
2.5 Peran Morphological Parser dalam Natural Language Processing[1] . . . . .	19
2.6 Struktur Data Trie . . . . .	20
2.6.1 Bitwise Trie . . . . .	20
2.6.2 Patricia Trie . . . . .	21
2.6.3 Implementasi Trie dalam Kamus . . . . .	22
2.6.4 Keunggulan Trie dibandingkan Struktur Data Lain . . . . .	23
<b>3 ANALISIS</b>	<b>25</b>
3.1 Leksikon . . . . .	25
3.2 Proses <i>Morphological Parsing</i> . . . . .	27
3.3 Morfotaktik . . . . .	29
3.4 Analisis Use Case . . . . .	31

3.5	Diagram Kelas . . . . .	34
<b>4</b>	<b>PERANCANGAN</b>	<b>37</b>
4.1	Struktur Penyimpanan Leksikon . . . . .	37
4.2	Syntax Keluaran Proses Morphological Parsing . . . . .	39
4.3	Perancangan Antarmuka . . . . .	40
4.3.1	Perancangan Antarmuka Perangkat Lunak Morphological Parser . . . . .	40
4.3.2	Perancangan Antarmuka Perangkat Lunak Lexicon . . . . .	41
4.4	Diagram Kelas Lengkap . . . . .	45
4.4.1	Kelas Parser . . . . .	46
4.4.2	Kelas Lexicon . . . . .	50
4.4.3	Kelas Node . . . . .	52
4.4.4	Kelas Combiner . . . . .	52
4.5	Diagram Aktivitas . . . . .	53
<b>5</b>	<b>IMPLEMENTASI DAN PENGUJIAN</b>	<b>57</b>
5.1	Implementasi . . . . .	57
5.2	Pengujian . . . . .	58
5.2.1	Pengujian Fungsional . . . . .	59
5.2.2	Pengujian Nonfungsional . . . . .	65
<b>6</b>	<b>KESIMPULAN DAN SARAN</b>	<b>69</b>
6.1	Kesimpulan . . . . .	69
6.2	Saran . . . . .	70
	<b>DAFTAR REFERENSI</b>	<b>71</b>
	<b>A BAHAN PENGUJIAN</b>	<b>73</b>
	<b>B KODE PROGRAM</b>	<b>75</b>

## DAFTAR GAMBAR

2.1	Morfem bebas dan terikat . . . . .	8
2.2	Morfem dasar dan afiks . . . . .	10
2.3	Trie dengan kata "trie","ini", dan "contoh" . . . . .	20
2.4	Pohon radix dengan 7 kata dengan prefix "r" . . . . .	21
2.5	Trie dengan kata "hello","hey","number","name","cat","camel", dan "cup" . . . . .	22
2.6	Penambahan kata "coding" pada trie di gambar 2.5 . . . . .	23
3.1	Struktur data trie . . . . .	26
3.2	Diagram use case perangkat lunak morphological parser . . . . .	31
3.3	Diagram kelas awal perangkat lunak morphological parser . . . . .	36
4.1	Isi dari file sayur.lxc . . . . .	38
4.2	Rancangan antarmuka perangkat lunak morphological parser . . . . .	40
4.3	Rancangan antarmuka perangkat lunak lexicon halaman home untuk user . . . . .	41
4.4	Rancangan antarmuka perangkat lunak lexicon halaman read untuk user . . . . .	42
4.5	Rancangan antarmuka perangkat lunak lexicon halaman home untuk editor . . . . .	42
4.6	Rancangan antarmuka perangkat lunak lexicon halaman create untuk editor . . . . .	43
4.7	Rancangan antarmuka perangkat lunak lexicon halaman update untuk editor . . . . .	44
4.8	Rancangan antarmuka perangkat lunak lexicon halaman delete untuk editor . . . . .	45
4.9	Diagram kelas lengkap perangkat lunak morphological parser . . . . .	46
4.10	Diagram aktivitas proses mengolah teks masukan . . . . .	54
4.11	Diagram aktivitas proses parsing pada sebuah kata . . . . .	55
5.1	Implementasi antarmuka perangkat lunak morphological parser . . . . .	58
5.2	Implementasi antarmuka perangkat lunak lexicon . . . . .	58
5.3	Hasil parsing contoh masukan pertama . . . . .	59
5.4	Hasil parsing contoh masukan kedua . . . . .	60
5.5	Hasil parsing contoh masukan pertama tanpa fitur converter . . . . .	60
5.6	Hasil parsing contoh masukan kedua tanpa fitur validator . . . . .	61
5.7	Isi leksikon sebelum kata 'warganet' dimasukkan . . . . .	62
5.8	Isi leksikon setelah kata 'warganet' dimasukkan . . . . .	63
5.9	Turunan dari kata dasar 'sayur' sebelum kata turunan 'menyayur-mayur' ditambahkan . . . . .	63
5.10	Turunan dari kata dasar 'sayur' setelah kata turunan 'menyayur-mayur' ditambahkan . . . . .	64
5.11	Isi leksikon sebelum kata 'abau' dihapus . . . . .	64
5.12	Isi leksikon setelah kata 'abau' dihapus . . . . .	65



## DAFTAR TABEL

3.1	Tabel aturan morfotaktik untuk preprefiks . . . . .	30
3.2	Tabel aturan morfotaktik untuk prefiks atau konfiks . . . . .	30
3.3	Tabel aturan morfotaktik untuk akar . . . . .	31
3.4	Tabel aturan morfotaktik untuk sufiks atau konfiks . . . . .	31
4.1	Tabel lambang bentuk turunan dalam leksikon . . . . .	37
5.1	Tabel waktu proses parsing untuk contoh masukan pertama . . . . .	66
5.2	Tabel waktu proses parsing untuk contoh masukan kedua . . . . .	66
5.3	Tabel waktu proses parsing untuk contoh masukan ketiga . . . . .	66
5.4	Tabel waktu untuk melakukan create sebuah entri baru dalam leksikon . . . . .	67
5.5	Tabel waktu untuk melakukan update sebuah entri dalam leksikon . . . . .	67
5.6	Tabel waktu untuk melakukan delete sebuah entri dalam leksikon . . . . .	67



# BAB 1

## PENDAHULUAN

### 1.1 Latar Belakang

Pengolahan bahasa alami atau dalam bahasa Inggris disebut dengan *natural language processing* (NLP) adalah cabang ilmu komputer dan linguistik yang mengkaji interaksi antara komputer dan manusia menggunakan bahasa alami. NLP sering dianggap sebagai cabang dari kecerdasan buatan dan bidang kajiannya bersinggungan dengan linguistik komputasional. Kajian NLP antara lain mencakup segmentasi tuturan (*speech segmentation*), segmentasi teks (*text segmentation*), penandaan kelas kata (*part-of-speech tagging*), serta pengawataksaan makna (*word sense disambiguation*). Salah satu alat yang digunakan oleh komputer dalam proses mengenali bahasa alami manusia adalah *morphological parser*.

*Morphological parser* berfungsi untuk membagi sebuah kata menjadi komponen-komponen penyusunnya. Proses ini dapat mengenali komponen kata seperti awalan, bentuk dasar, sisipan, dan akhiran serta dapat mengenali jika kata tersebut merupakan kata ulang maupun kata majemuk. Proses di mana *morphological parser* melakukan tugasnya dalam menguraikan kata menjadi komponen-komponen penyusunnya disebut dengan *morphological parsing*. Proses ini dapat membantu mengurangi ambiguitas selama proses mengetahui makna suatu kalimat. Sebagai contoh, kata "mengurus" bisa mempunyai makna menjadi kurus maupun mengerjakan sebuah urusan, bergantung pada apa bentuk dasar dari kata tersebut. Jika kita bisa membagi kata tersebut menjadi komponen penyusunnya, kita bisa lebih yakin mengenai makna dari kata tersebut dalam kalimat. *Morphological parsing* merupakan salah satu proses penting dalam NLP.

*Morphological parser* sudah banyak dibuat untuk beberapa bahasa yang ada di dunia, seperti bahasa Inggris, bahasa Turki, dan bahasa Bangla. Pisceldo et al. (2008) pernah membuat *morphological analyser* untuk bahasa Indonesia melalui pendekatan *two-level*, namun hanya dapat memproses kata hasil afiksasi dan reduplikasi. Dalam bahasa Indonesia, selain proses afiksasi dan reduplikasi, dikenal ada satu lagi proses morfologi yang umum dilakukan, yaitu proses komposisi. Proses komposisi adalah proses penggabungan bentuk dasar dengan bentuk dasar lain untuk mewadahi suatu "konsep" yang belum tertampung dalam sebuah kata[2]. Dalam skripsi ini, akan dibuat sebuah perangkat lunak *morphological parser* yang dapat memproses kata dalam bahasa Indonesia yang merupakan hasil proses afiksasi, reduplikasi, dan komposisi.

## 1.2 Rumusan Masalah

Sehubungan dengan latar belakang yang telah diuraikan di atas, maka dibuat rumusan masalah sebagai berikut ini.

- Bagaimana aturan morfologi bahasa Indonesia?
- Bagaimana struktur data dari *lexicon* yang digunakan pada perangkat lunak?
- Bagaimana cara mengimplementasikan aturan morfologi bahasa Indonesia ke dalam perangkat lunak?
- Bagaimana performansi dari perangkat lunak yang dihasilkan?

## 1.3 Tujuan

Tujuan dari penelitian ini adalah sebagai berikut:

- Mengetahui aturan morfologi bahasa Indonesia
- Mengetahui struktur data dari *lexicon* yang digunakan pada perangkat lunak
- Mengimplementasikan aturan morfologi bahasa Indonesia ke dalam perangkat lunak
- Mengetahui performansi dari perangkat lunak yang dihasilkan

## 1.4 Batasan Masalah

Terdapat beberapa batasan masalah untuk penelitian ini:

- Kalimat yang dapat diproses adalah kalimat dalam bahasa Indonesia yang ditulis sesuai ejaan yang disempurnakan (EYD)
- Kata yang dapat diproses adalah kata yang merupakan bentuk dasar dan kata yang dibentuk dari proses morfologi berupa afiksasi, reduplikasi, dan komposisi
- Kata yang belum ada dalam Kamus Besar Bahasa Indonesia (KBBI) dan yang bukan merupakan hasil dari proses afiksasi, reduplikasi, dan komposisi dianggap sebagai bentuk asing
- Kata yang merupakan hasil proses penyisipan (infiksasi) dan belum ada dalam KBBI tidak dapat diproses karena infiksasi dianggap sudah tidak produktif dalam bahasa Indonesia pada saat ini

## 1.5 Metodologi Penelitian

Tahap-tahap yang akan dilakukan dalam penelitian ini adalah sebagai berikut:

1. Melakukan studi literatur tentang morfologi bahasa Indonesia dan perangkat lunak *morphological parser* yang sudah ada



2. Melakukan analisis pada *morphological parser* bahasa Indonesia dan *lexicon* yang digunakan serta merancang struktur data dari *lexicon*
3. Merancang dan mengimplementasikan *lexicon* dan *morphological parser* ke dalam perangkat lunak
4. Mengumpulkan contoh kalimat dalam bahasa Indonesia sebagai bahan pengujian
5. Melakukan pengujian terhadap perangkat lunak

## 1.6 Sistematika Pembahasan

Keseluruhan bab yang disusun dalam karya tulis ini terbagi ke dalam bab-bab sebagai berikut:

1. BAB 1 - PENDAHULUAN membahas mengenai latar belakang, rumusan masalah, tujuan, batasan masalah, metodologi penelitian, dan sistematika pembahasan.
2. BAB 2 - DASAR TEORI membahas mengenai teori-teori dasar yang digunakan pada penelitian ini.
3. BAB 3 - ANALISIS membahas mengenai hasil analisis dari teori dasar dan kebutuhan dari perangkat lunak yang akan dibuat.
4. BAB 4 - PERANCANGAN membahas mengenai perancangan perangkat lunak berdasarkan hasil analisis yang telah dilakukan.
5. BAB 5 - IMPLEMENTASI DAN PENGUJIAN membahas mengenai implementasi dari hasil analisis dan perancangan perangkat lunak dan pengujian terhadap perangkat lunak yang sudah dibuat.
6. BAB 6 - KESIMPULAN DAN SARAN membahas mengenai kesimpulan dan saran dari hasil penelitian.



## BAB 2

### DASAR TEORI

Pada bab ini dijelaskan mengenai beberapa teori yang diperlukan dalam penelitian ini. Teori yang diperlukan adalah teori mengenai bahasa Indonesia, yaitu teori morfologi, morfem, proses morfologi, dan morfofonemik, lalu teori mengenai peran morphological parser dalam natural language processing, dan yang terakhir adalah teori mengenai struktur data trie yang akan digunakan dalam pembuatan perangkat lunak lexicon.

#### 2.1 Morfologi

Secara etimologi, kata *morfologi* berasal dari kata *morf* yang berarti 'bentuk' dan kata *logi* yang berarti 'ilmu'[2]. Secara harfiah, kata *morfologi* berarti 'ilmu mengenai bentuk'. Di dalam kajian linguistik, *morfologi* berarti 'ilmu mengenai bentuk-bentuk dan pembentukan kata'; sedangkan di dalam kajian biologi, *morfologi* berarti 'ilmu mengenai bentuk-bentuk sel-sel tumbuhan atau jasad-jasad hidup'. Kesamaan dari dua bidang kajian tersebut adalah keduanya mengkaji tentang bentuk.

Jika morfologi dalam kajian linguistik membicarakan tentang bentuk-bentuk dan pembentukan kata, maka segala bentuk dan jenis morfem yang merupakan satuan bentuk sebelum menjadi kata perlu dibicarakan juga. Pembicaraan mengenai pembentukan kata akan melibatkan pembicaraan mengenai komponen atau unsur pembentukan kata, yaitu morfem, baik morfem dasar maupun morfem afiks, dengan berbagai alat proses pembentukan kata, yaitu afiks dalam proses pembentukan kata melalui proses afiksasi, duplikasi atau pengulangan dalam proses pembentukan kata melalui proses reduplikasi, penggabungan dalam proses pembentukan kata melalui proses komposisi, dan sebagainya.

Ujung dari proses morfologi adalah terbentuknya *kata* dalam bentuk dan makna sesuai dengan keperluan dalam satu tindak pertuturan. Bila bentuk dan makna yang terbentuk dari satu proses morfologi sesuai dengan yang diperlukan dalam pertuturan, maka bentuknya dapat dikatakan berterima; tetapi jika tidak sesuai dengan yang diperlukan, maka bentuk itu dikatakan tidak berterima. Keberterimaan atau ketidakberterimaan bentuk itu dapat juga karena alasan sosial.

Objek kajian morfologi adalah satuan-satuan morfologi, proses-proses morfologi, dan alat-alat dalam proses morfologi itu[2]. Satuan morfologi adalah:

1. Morfem (akar atau afiks).
2. Kata.

Lalu, proses morfologi melibatkan komponen:

1. Dasar (bentuk dasar).
2. Alat pembentuk (afiks, duplikasi, komposisi).

*Morfem* adalah satuan gramatikal terkecil yang bermakna. Morfem dapat berupa akar (dasar) dan dapat pula berupa afiks. Perbedaannya, morfem berupa akar dapat menjadi dasar dalam pembentukan kata, sedangkan morfem berupa afiks hanya "menjadi" penyebab terjadinya makna gramatikal. Kemudian, *kata* adalah satuan gramatikal yang terjadi sebagai hasil dari proses morfologis. Jika berdiri sendiri, setiap kata memiliki makna leksikal dan dalam kedudukannya dalam satuan ujaran memiliki makna gramatikal.

Dalam proses morfologi, dasar atau bentuk dasar merupakan bentuk yang mengalami proses morfologi. Dasar ini dapat berupa sebuah kata dasar maupun bentuk polimorfemis (bentuk berimbuhan, bentuk ulang, atau bentuk gabungan). Alat pembentuk kata dapat berupa afiks dalam proses afiksasi, pengulangan dalam proses reduplikasi, dan penggabungan dalam proses komposisi.

## 2.2 Morfem

Morfem adalah satuan gramatikal terkecil yang memiliki makna[2]. Dengan kata terkecil berarti "satuan" itu tidak dapat dianalisis menjadi lebih kecil lagi tanpa merusak maknanya. Sebagai contoh, bentuk *membeli* dapat dianalisis menjadi dua bentuk terkecil yaitu {me-} dan {beli}. Bentuk {me-} adalah sebuah morfem, yakni morfem afiks yang secara gramatikal memiliki sebuah makna; dan bentuk {beli} juga sebuah morfem, yakni morfem dasar yang secara leksikal memiliki makna. Kalau bentuk *beli* dianalisis menjadi lebih kecil lagi menjadi *be-* dan *li*, keduanya tidak memiliki makna apapun. Jadi, keduanya bukan morfem. Contoh lain, bentuk *berpakaian* dapat dianalisis ke dalam satuan-satuan terkecil menjadi {ber-}, {pakai}, dan {-an}. Ketiganya adalah morfem, di mana {ber-} adalah morfem prefiks, {pakai} adalah morfem dasar, dan {-an} adalah morfem sufiks. Ketiganya memiliki makna. Morfem {ber-} dan morfem {-an} memiliki makna gramatikal, sedangkan morfem {pakai} memiliki makna leksikal. Perlu dicatat dalam konvensi linguistik sebuah bentuk dinyatakan sebagai morfem ditulis dalam kurung kurawal ({...}).

### 2.2.1 Identifikasi Morfem

Satuan bahasa merupakan komposit antara bentuk dan makna[2]. Oleh karena itu, untuk menetapkan sebuah bentuk adalah morfem atau bukan didasarkan pada kriteria bentuk dan makna tersebut. Hal-hal berikut dapat menjadi pedoman untuk menentukan apakah sebuah bentuk adalah morfem atau bukan.

1. Dua bentuk yang sama atau lebih memiliki makna yang sama merupakan sebuah morfem. Umpamanya kata *bulan* pada ketiga kalimat berikut adalah sebuah morfem yang sama.

- *Bulan* depan dia akan menikah.
- Sudah tiga *bulan* dia belum bayar uang SPP.
- *Bulan* November lamanya 30 hari.

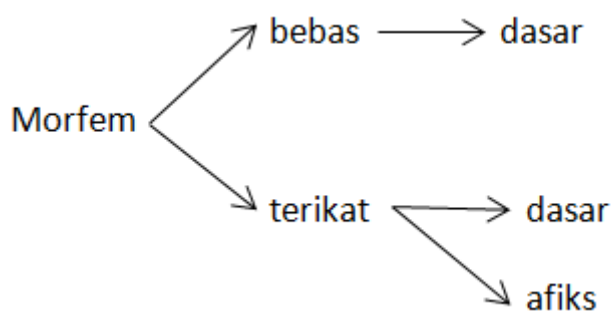
2. Dua bentuk yang sama atau lebih bila memiliki makna yang berbeda merupakan dua morfem yang berbeda. Misalnya kata *bunga* pada kedua kalimat berikut adalah dua buah morfem yang berbeda.
  - Bank Indonesia memberi *bunga* 5 persen per tahun.
  - Dia datang membawa seikat *bunga*.
3. Dua buah bentuk yang berbeda, tetapi memiliki makna yang sama, merupakan dua morfem yang berbeda. Umpamanya, kata *ayah* dan kata *bapak* pada kedua kalimat berikut adalah dua morfem yang berbeda.
  - *Ayah* pergi ke Medan.
  - *Bapak* baru pulang dari Medan.
4. Bentuk-bentuk yang mirip (berbeda sedikit) tetapi maknanya sama adalah sebuah morfem yang sama, asal perbedaan bentuk itu dapat dijelaskan secara fonologis. Umpamanya, bentuk-bentuk *me-*, *mem-*, *men-*, *meny-*, *meng-*, dan *menge-* pada kata-kata berikut adalah sebuah morfem yang sama.
  - *melihat*
  - *membina*
  - *mendengar*
  - *menyusul*
  - *mengambil*
  - *mengecat*
5. Bentuk yang hanya muncul dengan pasangan satu-satunya adalah sebuah morfem juga. Umpamanya bentuk *renta* pada konstruksi *tua renta*, dan bentuk *kuyup* pada konstruksi *basah kuyup* adalah juga morfem. Contoh lain, bentuk *bugar* pada *segar bugar*, dan bentuk *mersik* pada *kering mersik*.
6. Bentuk yang muncul berulang-ulang pada satuan yang lebih besar apabila memiliki makna yang sama adalah juga merupakan morfem yang sama. Misalnya bentuk *baca* pada kata-kata berikut adalah sebuah morfem yang sama.
  - *membaca*
  - *pembaca*
  - *pembacaan*
  - *bacaan*
  - *terbaca*
  - *keterbacaan*
7. Bentuk yang muncul berulang-ulang pada satuan bahasa yang lebih besar, apabila mempunyai bentuk bahasa yang sama namun maknanya berbeda (polisemi) merupakan morfem yang sama. Umpamanya, kata *kepala* pada kalimat-kalimat berikut memiliki makna yang berbeda, tetapi tetap merupakan morfem yang sama.

- Ibunya menjadi *kepala* sekolah di sana.
- Nomor teleponnya tertera pada *kepala* surat itu.
- *Kepala* jarum itu terbuat dari plastik.
- Setiap *kepala* mendapat bantuan sepuluh ribu rupiah.
- Tubuhnya memang besar tetapi sayang *kepalanya* kosong.

### 2.2.2 Jenis Morfem

Dalam kajian morfologi biasanya dibedakan adanya beberapa morfem berdasarkan kriteria tertentu, seperti kriteria kebebasan, keutuhan, makna, dan sebagainya. Berikut adalah jenis-jenis morfem tersebut.

1. Berdasarkan kebebasannya untuk dapat digunakan langsung dalam pertuturan, dibedakan adanya *morfem bebas* dan *morfem terikat*. Morfem bebas adalah morfem yang tanpa keterkaitannya dengan morfem lain dapat langsung digunakan dalam pertuturan. Misalnya, morfem {pulang}, {merah}, dan {pergi}. Morfem bebas ini tentunya berupa morfem dasar. Sedangkan morfem terikat adalah morfem yang harus terlebih dahulu bergabung dengan morfem lain untuk dapat digunakan dalam pertuturan. Dalam hal ini, semua afiks dalam bahasa Indonesia termasuk morfem terikat. Di samping itu, banyak juga morfem terikat yang berupa morfem dasar, seperti {henti}, {juang}, dan {geletak}. Untuk dapat digunakan, ketiga morfem ini harus terlebih dahulu diberi afiks atau digabung dengan morfem lain. Misalnya {juang} menjadi *berjuang*, *pejuang*, dan *daya juang*; *henti* harus digabung dulu dengan afiks tertentu seperti menjadi *berhenti*, *perhentian*, dan *menghentikan*; dan *geletak* harus diberi imbuhan dulu, misalnya menjadi *tergeletak*, dan *mengeletak*. Adanya morfem bebas dan terikat dapat digambarkan pada gambar 2.1 berikut.



Gambar 2.1: Morfem bebas dan terikat[2]

Berkenaan dengan bentuk dasar terikat, perlu dikemukakan catatan sebagai berikut:

*Pertama*, bentuk dasar terikat seperti *gaul*, *juang*, dan *henti* lazim juga disebut sebagai *prakategorial* karena bentuk-bentuk tersebut belum memiliki kategori sehingga tidak dapat digunakan dalam pertuturan.

*Kedua*, Verhaar (1978) juga memasukkan bentuk-bentuk seperti *beli*, *baca*, dan *tulis* ke dalam kelompok *prakategorial*, karena untuk digunakan di dalam kalimat harus terlebih dahulu diberi prefiks *me-*, prefiks *di-*, atau prefiks *ter-*. Dalam kalimat imperatif memang tanpa

imbuhan bentuk-bentuk tersebut dapat digunakan. Namun, kalimat imperatif adalah hasil transformasi dari kalimat aktif transitif (yang memerlukan imbuhan).

*Ketiga*, bentuk-bentuk seperti *renta* (yang hanya muncul dalam *tua renta*), *kerontang* (yang hanya muncul dalam *kering kerontang*), dan *kuyup* (yang hanya muncul dalam *basah kuyup*) adalah juga termasuk morfem terikat. Lalu, oleh karena hanya muncul dalam pasangan tertentu, maka disebut *morfem unik*.

*Keempat*, bentuk-bentuk yang disebut klitika merupakan morfem yang agak sukar ditentukan statusnya, apakah morfem bebas atau morfem terikat. Kemunculannya dalam pertuturan selalu terikat dengan bentuk lain, tetapi dapat dipisahkan. Umpamanya klitika *-ku* dalam konstruksi *bukuku* dapat dipisahkan sehingga menjadi *buku baruku*. Dilihat dari posisi tempatnya dibedakan adanya proklitika, yaitu klitika yang berposisi di muka kata yang diikuti seperti klitika *ku-* dalam bentuk *kubawa* dan *kauambil*. Sedangkan yang disebut enklitika adalah klitika yang berposisi di belakang kata yang dilekati, seperti klitika *-mu* dan *-nya* pada bentuk *nasibmu* dan *duduknya*.

*Kelima*, bentuk-bentuk yang termasuk preposisi dan konjungsi seperti *dan*, *oleh*, *di*, dan *karena* secara morfologis termasuk morfem bebas, tetapi secara sintaksis merupakan bentuk terikat (dalam satuan sintaksisnya).

*Keenam*, bentuk-bentuk yang oleh Kridalaksana (1989) disebut proleksem, seperti *a* (pada *asusila*), *dwi* (pada *dwibahasa*), dan *ko* (pada *kopilot*) juga termasuk morfem terikat.

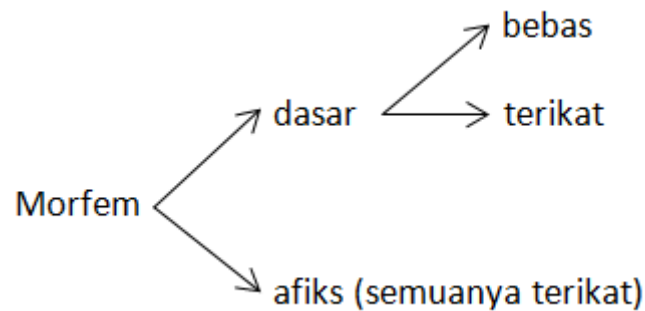
2. Berdasarkan keutuhan bentuknya dibedakan adanya *morfem utuh* dan *morfem terbagi*. Morfem utuh secara fisik merupakan satu-kesatuan yang utuh. Semua morfem dasar, baik bebas maupun terikat, serta prefiks, infiks, dan sufiks termasuk morfem utuh. Sedangkan yang dimaksud morfem terbagi adalah morfem yang fisiknya terbagi atau disisipi morfem lain. Karenanya semua konfiks (seperti *pe-an*, *ke-an*, dan *per-an*) adalah termasuk morfem terbagi. Namun, mengenai morfem terbagi ini ada dua catatan yang perlu diperhatikan.

*Pertama*, semua konfiks adalah morfem terbagi; tetapi pada bentuk *ber-an* ada yang berupa konfiks dan ada yang bukan konfiks. Jika kata dalam bentuk *ber-an* tidak memiliki arti ketika hanya ditambahkan prefiks *ber-* atau sufiks *-an* saja, maka bentuk *ber-an* tersebut adalah berupa konfiks. Namun, jika kata tersebut memiliki arti ketika hanya ditambahkan prefiks *ber-* atau sufiks *-an* saja, maka bentuk *ber-an* tersebut adalah berupa *klofiks* (akronim dari kelompok afiks). Contoh, kata *bermunculan* adalah dasar *muncul* ditambahkan konfiks *ber-an* sementara kata *berpakaian* adalah prefiks *ber-* yang ditambahkan pada bentuk *pakaian*.

*Kedua*, dalam bahasa Indonesia ada afiks yang disebut infiks, yaitu afiks yang ditempatkan di tengah (di dalam kata). Umpamanya infiks *-el-* pada dasar *tunjuk* menjadi kata *telunjuk*. Di sini infiks itu memecah morfem *tunjuk* menjadi dua bagian, yaitu *t-el-unjuk*. Dengan demikian morfem *t-unjuk* menjadi morfem terbagi, bukan morfem utuh.

3. Berdasarkan kemungkinan menjadi dasar dalam pembentukan kata, dibedakan *morfem dasar* dan *morfem afiks*. Morfem dasar adalah morfem yang dapat menjadi dasar dalam suatu proses morfologi. Misalnya, morfem {beli}, {makan}, dan {merah}. Namun, perlu dicatat bentuk dasar yang termasuk dalam kategori preposisi dan konjungsi tidak pernah mengalami proses afiksasi. Sedangkan, yang tidak dapat menjadi dasar, melainkan hanya sebagai pembentuk disebut morfem afiks, seperti morfem {me-}, {-kan}, dan {pe-an}. Berdasarkan pembagian

ini, maka dapat dibuat gambar 2.2 berikut.



Gambar 2.2: Morfem dasar dan afiks[2]

4. Berdasarkan ciri semantik dibedakan adanya *morfem bermakna leksikal* dan *morfem tak bermakna leksikal*. Sebuah morfem disebut bermakna leksikal karena di dalam dirinya, secara inheren, telah memiliki makna. Semua morfem dasar bebas, seperti {makan}, {pulang}, dan {pergi} termasuk morfem bermakna leksikal. Sebaliknya, morfem afiks seperti {ber-}, {ke-}, dan {ter-} termasuk morfem tak bermakna leksikal. Morfem bermakna leksikal dapat langsung menjadi unsur dalam pertuturan, sementara morfem tidak bermakna leksikal tidak dapat.

Dikotomi morfem bermakna leksikal dan tidak bermakna leksikal ini, untuk bahasa Indonesia timbul masalah. Morfem-morfem seperti {juang}, {henti}, dan {gaul} memiliki makna leksikal atau tidak. Kalau dikatakan memiliki makna leksikal, pada kenyataannya morfem-morfem itu belum dapat digunakan dalam pertuturan sebelum mengalami proses morfologi. Kalau dikatakan tidak bermakna leksikal, pada kenyataannya morfem-morfem tersebut bukan afiks.

### 2.2.3 Morfem Dasar, Bentuk Dasar, Akar, dan Leksem

Morfem dasar, bentuk dasar (lebih lazim dasar (*base*) saja), akar, dan leksem adalah empat istilah yang lazim digunakan dalam kajian morfologi. Namun, seringkali digunakan secara kurang cermat, malah seringkali berbeda. Oleh karena itu, ada baiknya istilah-istilah tersebut dibicarakan dulu sebelum pembicaraan mengenai proses-proses morfologi.

Istilah *morfem dasar* biasanya digunakan sebagai dikotomi dengan morfem afiks. Jadi, bentuk-bentuk seperti {beli}, {juang}, dan {kucing} adalah morfem dasar. Morfem dasar ini ada yang termasuk morfem bebas seperti {beli}, {kucing}, dan {pulang}; tetapi ada pula yang termasuk morfem terikat, seperti {juang}, {henti}, dan {tempur}. Sedangkan morfem afiks seperti {ber-}, {di-}, dan {-an} jelas semuanya termasuk morfem terikat seperti dijelaskan pada gambar 2.2 di atas.

Sebuah morfem dasar dapat menjadi bentuk dasar atau dasar (*base*) dalam suatu proses morfologi. Artinya, morfem dasar dapat diberi afiks tertentu dalam proses afiksasi, dapat diulang dalam proses reduplikasi, atau dapat digabung dengan morfem yang lain dalam suatu proses komposisi atau pemajemukan.



Istilah *bentuk dasar* atau *dasar (base)* biasanya digunakan untuk menyebut sebuah bentuk yang menjadi dasar dalam suatu proses morfologi. Bentuk dasar ini dapat berupa morfem tunggal, tetapi dapat juga berupa gabungan morfem. Umpamanya pada kata *berbicara* yang terdiri dari morfem {ber-} dan morfem {bicara}; maka morfem {bicara} adalah menjadi bentuk dasar dari kata *berbicara* itu, yang kebetulan juga berupa morfem dasar. Pada kata *dimengerti* bentuk dasarnya adalah *mengerti*, dan pada kata *keanekaragaman* bentuk dasarnya adalah *aneka ragam*. Pada bentuk reduplikasi *rumah-rumah* bentuk dasarnya adalah *rumah*, pada bentuk reduplikasi *berlari-lari* bentuk dasarnya *berlari*, dan pada bentuk reduplikasi *kemerah-merahan* bentuk dasarnya adalah *kemerahan*. Lalu, pada komposisi *sate ayam* bentuk dasarnya adalah *sate*, pada komposisi *ayam betina* bentuk dasarnya adalah *ayam*, dan pada komposisi *pasar induk* bentuk dasarnya adalah *pasar*. Jadi, bentuk dasar adalah bentuk yang langsung menjadi dasar dalam suatu proses morfologi. Wujudnya dapat berupa morfem tunggal, dapat juga berupa bentuk polimorfemis (terdiri dari dua morfem atau lebih).

Istilah *akar (root)* digunakan untuk menyebut bentuk yang tidak dapat dianalisis lebih jauh lagi. Artinya, akar adalah bentuk yang tersisa setelah semua afiksnya ditanggalkan. Misalkan pada kata *memberlakukan* setelah semua afiksnya ditanggalkan (yaitu prefiks *me-*, prefiks *ber-*, dan sufiks *-kan*) dengan cara tertentu, maka yang tersisa adalah akar *laku*. Akar *laku* ini tidak dapat dianalisis lebih jauh lagi tanpa merusak makna akar tersebut. Contoh lain, kata *keberterimaan* kalau semua afiksnya ditanggalkan akan tersisa akarnya yaitu bentuk *terima*. Bentuk *terima* ini pun tidak dapat dianalisis lebih jauh lagi.

Istilah *leksem* ada digunakan dalam dua bidang kajian linguistik, yaitu bidang *morfologi* dan bidang *semantik*. Dalam kajian morfologi, leksem digunakan untuk mewadahi konsep "bentuk yang akan menjadi kata" melalui proses morfologi. Umpamanya bentuk PUKUL (dalam konvensi 'morfologi' leksem ditulis dengan huruf kapital semua) adalah sebuah leksem yang akan menurunkan kata-kata seperti *memukul*, *dipukul*, *terpukul*, *pukulan*, *pemukul*, dan *pemukulan*. Sedangkan dalam kajian semantik leksem adalah satuan bahasa yang memiliki sebuah makna. Jadi, bentuk-bentuk seperti *kucing*, *membaca*, *matahari*, *membanting tulang*, dan *sumpah serapah* adalah leksem.

Dari bentuk *leksem* ada bentuk-bentuk turunannya, yaitu *leksikon*, *leksikal*, *leksikologi*, dan *leksikografi*. Istilah leksikon dalam arti 'kumpulan leksem' dapat dipadankan dengan istilah *kosakata* atau *perbendaharaan kata*.

#### 2.2.4 Morfem Afiks

Sudah dijelaskan pada subbab 2.2.2 bahwa morfem afiks adalah morfem yang tidak dapat menjadi dasar dalam pembentukan kata, tetapi hanya menjadi unsur pembentuk dalam proses afiksasi. Dalam bahasa Indonesia dibedakan adanya morfem afiks yang disebut:

1. *Prefiks*, yaitu afiks yang dibubuhkan di kiri bentuk dasar, yaitu prefiks *ber-*, prefiks *me-*, prefiks *per-*, prefiks *pe-*, prefiks *di-*, prefiks *ter-*, prefiks *se-*, dan prefiks *ke-*.
2. *Infiks*, yaitu afiks yang dibubuhkan di tengah kata, biasanya pada suku awal kata, yaitu infiks *-el-*, infiks *-em-*, dan infiks *-er-*.
3. *Sufiks*, adalah afiks yang dibubuhkan di kanan bentuk dasar, yaitu sufiks *-kan*, sufiks *-i*, dan sufiks *-an*.

4. *Konfiks*, yaitu afiks yang dibubuhkan di kiri dan di kanan bentuk dasar secara bersamaan karena konfiks ini merupakan satu kesatuan afiks. Konfiks yang ada dalam bahasa Indonesia adalah konfiks *ke-an*, konfiks *ber-an*, konfiks *pe-an*, konfiks *per-an*, dan konfiks *se-nya*.
5. *Klitika*<sup>1</sup>, adalah imbuhan yang dalam ucapan tidak mempunyai tekanan sendiri dan tidak merupakan kata karena tidak dapat berdiri sendiri. Jadi, klitika merupakan bentuk yang selalu terikat pada bentuk (kata) lain. Dilihat dari posisi tempatnya, dibedakan adanya *proklitika*, yaitu klitika yang berposisi di sebelah kiri kata yang diikuti seperti klitika *ku-* dan *kau-* dalam bentuk *kubawa* dan *kauambil*. Sedangkan yang disebut *enklitika* adalah klitika yang berposisi di belakang kata yang dilekati, seperti klitika *-ku*, *-mu*, *-nya*, dan *-lah* pada bentuk *bukuku*, *nasibmu*, *duduknya*, dan *pergilah*. Ada juga bentuk klitika yang ditulis terpisah dari kata yang diimbuhan, yaitu klitika *pun* pada bentuk *kami pun*.
6. Dalam bahasa Indonesia ada bentuk kata yang *berklofiks*, yaitu kata yang dibubuhi afiks pada kiri dan kanannya; tetapi pembubuhannya itu tidak sekaligus, melainkan bertahap. Kata-kata berklofiks dalam bahasa Indonesia adalah yang berbentuk *me-kan*, *me-i*, *memper-*, *memper-kan*, *memper-i*, *ber-kan*, *di-kan*, *di-i*, *diper-*, *diper-kan*, *diper-i*, *ter-kan*, dan *ter-i*.
7. Dalam ragam nonbaku ada afiks nasal yang direalisasikan dengan nasal *m-*, *n-*, *ny-*, *ng-*, dan *nge-*. Kridalaksana (1989) menyebut afiks nasal ini dengan istilah *simulfiks*. Contoh: *nulis*, *nyisir*, *ngambil*, dan *ngecat*.

## 2.3 Proses Morfologi

Proses morfologi pada dasarnya adalah proses pembentukan kata dari sebuah bentuk dasar melalui pembubuhan afiks (dalam proses afiksasi), pengulangan (dalam proses reduplikasi), dan penggabungan (dalam proses komposisi)[2]. Prosedur ini berbeda dengan analisis morfologi yang menceraikan kata (sebagai satuan sintaksis) menjadi bagian-bagian atau satuan-satuan yang lebih kecil. Sebagai contoh, jika dilakukan analisis morfologi terhadap kata *berpakaian*, mula-mula kata *berpakaian* dianalisis menjadi bentuk *ber-* dan *pakaian*; lalu bentuk *pakaian* dianalisis lagi menjadi bentuk *pakai* dan *-an*. Dalam proses morfologi, prosedurnya dibalik: mula-mula dasar *pakai* diberi sufiks *-an* menjadi *pakaian*. Kemudian kata *pakaian* itu diberi prefiks *ber-* menjadi *berpakaian*. Jadi, kalau analisis morfologi menceraikan data kebahasaan yang ada, sedangkan proses morfologi mencoba menyusun dari komponen-komponen kecil menjadi sebuah bentuk yang lebih besar yang berupa kata kompleks atau kata yang polimorfemis.

Proses morfologi melibatkan komponen bentuk dasar dan alat pembentuk kata (afiksasi, reduplikasi, dan komposisi).

### 2.3.1 Bentuk Dasar

Pada subbab 2.2.3 telah disinggung bahwa *bentuk dasar* adalah bentuk yang kepadanya dilakukan proses morfologi. Bentuk dasar dapat berupa akar seperti *baca*, *pahat*, dan *juang* pada kata *membaca*, *memahat*, dan *berjuang*. Dapat pula berupa bentuk polimorfemis seperti bentuk *bermakna*, *berlari*, dan *jual beli* pada kata *kebermaknaan*, *berlari-lari*, dan *berjual beli*.

<sup>1</sup>[id.wikibooks.org/wiki/Bahasa\\_Indonesia/Klitika](https://id.wikibooks.org/wiki/Bahasa_Indonesia/Klitika)

Dalam proses reduplikasi, bentuk dasar dapat berupa akar, seperti akar *rumah* pada kata *rumah-rumah*, akar *tinggi* seperti pada kata *tinggi-tinggi*, dan akar *marah* pada kata *marah-marah*. Dapat juga berupa kata berimbuhan seperti *menembak* pada kata *menembak-nembak*, kata berimbuhan *bangunan* pada kata *bangunan-bangunan*, dan kata berimbuhan *kemerahan* pada kata *kemerah-merahan*. Dapat juga berupa kata gabung seperti *rumah sakit* pada kata *rumah-rumah sakit*, dan *anak nakal* pada kata *anak-anak nakal*.

Dalam proses komposisi, bentuk dasar dapat berupa akar *sate* pada kata *sate ayam*, *sate padang*, dan *sate lontong*; dapat berupa dua buah akar seperti akar *kampung* dan akar *halaman* pada kata *kampung halaman*, atau akar *tua* dan akar *muda* pada kata *tua muda*.

Ada perbedaan bentuk antara *pelajar* dan *pengajar*. Menurut kajian tradisional dan struktural bentuk dasar dari kedua kata itu adalah sama, yaitu akar *ajar*. Dalam kajian proses di sini bentuk dasar kedua kata itu tidaklah sama. Bentuk dasar kata *pelajar* adalah *belajar* sedangkan bentuk dasar kata *pengajar* adalah *mengajar*. Ini dikarenakan makna gramatikal kata *pelajar* adalah 'orang yang belajar' sedangkan makna gramatikal kata *pengajar* adalah 'orang yang mengajar'. Contoh lain, bentuk dasar kata *penyatuan* adalah *menyatukan* karena makna *penyatuan* adalah 'hal/proses menyatukan'. Sedangkan bentuk dasar kata *persatuan* adalah *bersatu* atau *mempersatukan* karena makna gramatikalnya adalah 'hal bersatu' atau 'hal mempersatukan'. Namun, secara teoretis dapat juga dikatakan bentuk dasar kata *pelajar* dan *pengajar* adalah sama yaitu *ajar*; tetapi bentuk *pelajar* dibentuk dari dasar *ajar* melalui verba *belajar*, sedangkan *pengajar* dibentuk dari dasar *ajar* melalui verba *mengajar*. Demikian juga kata *penyatuan* dibentuk dari dasar *satu* melalui verba *menyatukan*, sedangkan kata *persatuan* dibentuk dari dasar *satu* melalui verba *bersatu* atau *mempersatukan*.

Dari uraian di atas, jelas bahwa konsep *bentuk dasar* tidak sama dengan pengertian *morfem dasar* atau *kata dasar*. Ini dikarenakan bentuk dasar dapat juga berupa bentuk-bentuk polimorfemis.

### 2.3.2 Pembentuk Kata

Komponen kedua dalam proses morfologi adalah alat pembentuk kata. Sejauh ini alat pembentuk kata dalam proses morfologi adalah (a) afiks dalam proses afiksasi, (b) pengulangan dalam proses reduplikasi, dan (c) penggabungan dalam proses komposisi.

Dalam proses afiksasi sebuah afiks diimbuhkan pada bentuk dasar sehingga hasilnya menjadi sebuah kata. Umpamanya pada dasar *baca* diimbuhkan afiks *me-* sehingga menghasilkan kata *membaca* yaitu sebuah verba transitif aktif; pada dasar *juang* diimbuhkan afiks *ber-* sehingga menghasilkan verba intransitif *berjuang*.

Berkenaan dengan jenis afiksnya, proses afiksasi dibedakan atas *prefiksasi*, yaitu proses pembubuhan prefiks, *konfiksasi* yakni proses pembubuhan konfiks, *sufiksasi* yaitu proses pembubuhan sufiks dan *infiksasi* yakni proses pembubuhan infiks. Perlu dicatat dalam bahasa Indonesia proses infiksasi sudah tidak produktif lagi. Dalam hal ini perlu juga diperhatikan adanya *klofiksasi*, yaitu kelompok afiks yang proses afiksasinya dilakukan bertahap. Misalnya pembentukan kata *menangisi*, mula-mula pada dasar *tangis* diimbuhkan sufiks *-i*; setelah itu baru dibubuhkan prefiks *me-*.

Proses prefiksasi dilakukan oleh prefiks *ber-*, *me-*, *pe-*, *per-*, *di-*, *ter-*, *ke-*, dan *se-*; infiksasi dilakukan oleh infiks *-el-*, *-em-*, dan *-er-*; sufiksasi dilakukan sufiks *-an*, *-kan*, dan *-i*; konfiksasi dilakukan oleh konfiks *pe-an*, *per-an*, *ke-an*, *se-nya*, dan *ber-an*; dan klofiksasi dilakukan oleh klofiks

1 *me-kan, me-i, memper-, memper-kan, memper-i, ber-kan, di-kan, di-i, diper-, diper-kan, diper-i,*  
 2 *ter-kan, dan ter-i.*

3 Alat pembentuk kedua adalah pengulangan bentuk dasar yang digunakan dalam proses redupli-  
 4 kasi. Hasil dari proses reduplikasi ini lazim disebut dengan istilah *kata ulang*. Secara umum dikenal  
 5 adanya tiga macam pengulangan, yaitu pengulangan secara utuh, pengulangan dengan perubahan  
 6 bunyi vokal maupun konsonan, dan pengulangan sebagian.

7 Alat pembentuk ketiga adalah penggabungan sebuah bentuk pada bentuk dasar yang ada  
 8 dalam proses komposisi. Penggabungan ini juga merupakan alat yang banyak digunakan dalam  
 9 pembentukan kata karena banyaknya konsep yang belum ada wadahnya dalam bentuk sebuah kata.  
 10 Misalnya, bahasa Indonesia hanya punya sebuah kata untuk berbagai macam warna merah. Oleh  
 11 karena itulah dibentuk gabungan kata seperti *merah jambu, merah darah, dan merah bata*.

## 12 2.4 Morfofonemik

13 Morfofonemik (disebut juga morfofonologi) adalah kajian mengenai terjadinya perubahan bunyi  
 14 atau perubahan fonem sebagai akibat dari adanya proses morfologi, baik proses afiksasi, proses  
 15 reduplikasi, maupun proses komposisi[2]. *Fonem* adalah satuan bunyi terkecil (dalam kajian fonologi)  
 16 yang dapat membedakan makna kata. Morfofonemik dalam pembentukan kata bahasa Indonesia  
 17 terutama terjadi dalam proses afiksasi. Dalam proses reduplikasi dan komposisi hampir tidak ada.  
 18 Dalam proses afiksasi pun terutama, hanya dalam prefiksasi *ber-*, prefiksasi *me-*, prefiksasi *pe-*,  
 19 prefiksasi *per-*, konfiksasi *pe-an*, konfiksasi *per-an*, dan sufiksasi *-an*.

20 Berikut adalah beberapa jenis perubahan fonem dan bentuk-bentuk morfofonemik pada beberapa  
 21 proses morfologi.

### 22 2.4.1 Jenis Perubahan

23 Dalam bahasa Indonesia ada beberapa jenis perubahan fonem berkenaan dengan proses morfologi  
 24 ini. Di antaranya adalah proses:

- 25 1. *Pemunculan fonem*, yakni munculnya fonem (bunyi) dalam proses morfologi yang pada  
 26 mulanya tidak ada. Misalnya, dalam proses pengimbuhan prefiks *me-* pada dasar *baca* akan  
 27 memunculkan bunyi sengau [m] yang semula tidak ada.  
 28  $me + baca \rightarrow membaca$
- 29 2. *Pelesapan fonem*, yakni hilangnya fonem dalam suatu proses morfologi. Misalnya, dalam  
 30 proses pengimbuhan prefiks *ber-* pada dasar *renang*, maka bunyi [r] yang ada pada prefiks *ber-*  
 31 dilesapkan. Juga, dalam proses pengimbuhan "akhiran" *-wan* pada dasar *sejarah*, maka fonem  
 32 /h/ pada dasar *sejarah* itu dilesapkan. Contoh lain, pada proses pengimbuhan "akhiran" *-nda*  
 33 pada dasar *anak*, maka fonem /k/ pada dasar *anak* dilesapkan atau dihilangkan.  
 34  $ber + renang \rightarrow berenang$   
 35  $sejarah + wan \rightarrow sejarawan$   
 36  $anak + nda \rightarrow ananda$

37 Dalam beberapa tahun terakhir ada juga gejala pelesapan salah satu fonem yang sama yang  
 38 terdapat pada akhir kata dan awal kata yang mengalami proses komposisi. Misalnya.

*pasar + raya → pasaraya*

*ko + operasi → koperasi*

3. *Peluluhan fonem*, yakni luluhnya sebuah fonem serta disenyawakan dengan fonem lain dalam suatu proses morfologi. Umpamanya, dalam pengimbuhan prefiks *me-* pada dasar *sikat*, maka fonem /s/ pada kata *sikat* itu diluluhkan dan disenyawakan dengan fonem nasal /ny/ yang ada pada prefiks *me-*. Hal yang sama juga terjadi pada proses pengimbuhan prefiks *pe-*.

*me + sikat → menyikat*

*pe + sikat → penyikat*

4. *Perubahan fonem*, yakni berubahnya sebuah fonem atau sebuah bunyi, sebagai akibat terjadinya proses morfologi. Umpamanya, dalam pengimbuhan prefiks *ber-* pada dasar *ajar* terjadi perubahan bunyi, di mana fonem /r/ berubah menjadi fonem /l/.

*ber + ajar → belajar*

## 2.4.2 Prefiksasi ber-

Morfofonemik dalam proses pengimbuhan prefiks *ber-* berupa: (a) pelepasan fonem /r/ pada prefiks *ber-*; (b) perubahan fonem /r/ pada prefiks *ber-* menjadi fonem /l/; dan (c) pengekaln fonem /r/ yang terdapat prefiks *ber-* itu.

1. Pelepasan fonem /r/ pada prefiks *ber-* itu terjadi apabila bentuk dasar yang diimbuhi mulai dengan fonem /r/, atau suku pertama bentuk dasarnya berbunyi [er]. Misalnya:

*ber + renang → berenang*

*ber + ragam → beragam*

*ber + racun → beracun*

*ber + kerja → bekerja*

*ber + ternak → beternak*

*ber + cermin → bercermin*

2. Perubahan fonem /r/ pada prefiks *ber-* menjadi fonem /l/ terjadi bila bentuk dasarnya akar *ajar*; tidak ada contoh lain.

*ber + ajar → belajar*

3. Pengekaln fonem /r/ pada prefiks *ber-* tetap /r/ terjadi apabila bentuk dasarnya bukan yang ada pada poin 1 dan 2 di atas.

*ber + obat → berobat*

*ber + korban → berkorban*

*ber + getah → bergetah*

*ber + lari → berlari*

*ber + tamu → bertamu*

## 2.4.3 Prefiksasi me- (termasuk klofiks me-kan dan me-i)

Morfofonemik dalam proses pengimbuhan dengan prefiks *me-* dapat berupa: (a) pengekaln fonem; (b) penambahan fonem; dan (c) peluluhan fonem.

1. Pengekalan fonem di sini artinya tidak ada fonem yang berubah, tidak ada yang dihapuskan dan tidak ada yang ditambahkan. Hal ini terjadi apabila bentuk dasarnya diawali dengan konsonan /r, l, w, y, m, n, ng, dan ny/. Contoh:

*me + rawat → merawat*

*me + lirik → melirik*

*me + wasiat → wasiat*

*me + yakin → meyakinkan*

*me + makan → memakan*

*me + nanti → menanti*

*me + nganga → nganga*

*me + nyanyi → nyanyi*

2. Penambahan fonem, yakni penambahan fonem nasal /m, n, ng, dan nge/. Penambahan fonem nasal /m/ terjadi apabila bentuk dasarnya dimulai dengan konsonan /b/, /f/, dan /v/. Umpamanya:

*me + baca → membaca*

*me + buru → memburu*

*me + fitnah → memfitnah*

*me + fokus → memfokus(kan)*

*me + vonis → memvonis*

Penambahan fonem nasal /n/ terjadi apabila bentuk dasarnya dimulai dengan konsonan /c/, /d/, dan /j/. Umpamanya:

*me + cari → mencari*

*me + dengar → mendengar*

*me + jual → menjual*

Penambahan fonem nasal /ng/ terjadi apabila bentuk dasarnya dimulai dengan konsonan /g, h, dan kh/ dan huruf vokal /a, i, u, e, dan o/. Contoh:

*me + goda → menggoda*

*me + hina → menghina*

*me + khayal → mengkhayal*

*me + ambil → mengambil*

*me + iris → mengiris*

*me + ukur → mengukur*

*me + elak → mengelak*

*me + obral → mengobral*

Penambahan fonem nasal /nge/ terjadi apabila bentuk dasarnya hanya terdiri dari satu suku kata. Misalnya:

*me + bom → mengebom*

*me + cat → mengecat*

*me + lap → mengelap*

3. Peluluhan fonem terjadi apabila prefiks *me-* diimbuhkan pada bentuk dasar yang dimulai dengan konsonan bersuara /s, k, p, dan t/. Dalam hal ini konsonan /s/ diluluhkan dengan

nasal /ny/, konsonan /k/ diluluhkan dengan nasal /ng/, konsonan /p/ diluluhkan dengan nasal /m/, dan konsonan /t/ diluluhkan dengan nasal /n/. Contoh:

*me + sikat → menyikat*

*me + kirim → mengirim*

*me + pilih → memilih*

*me + tolong → menolong*

#### 2.4.4 Prefiksasi *pe-* dan konfiksasi *pe-an*

Morf fonemik dalam proses pengimbuhan dengan prefiks *pe-* dan konfiks *pe-an* sama dengan morf fonemik yang terjadi dalam proses pengimbuhan dengan prefiks *me-*, yaitu (a) pengeklalan fonem; (b) penambahan fonem; dan (c) peluluhan fonem.

1. Pengeklalan fonem, artinya tidak ada perubahan fonem, dapat terjadi apabila bentuk dasarnya diawali dengan konsonan /r, l, y, w, m, n, ng, dan ny/. Contoh:

*pe + rawat → perawat*

*pe + latih → pelatih*

*pe + yakin → peyakin*

*pe + waris → pewaris*

*pe - an + manfaat → pemanfaatan*

*pe - an + nanti → penantian*

*pe + nganga → penganga*

*pe + nyanyi → penyanyi*

2. Penambahan fonem, yakni penambahan fonem nasal /m, n, ng, dan nge/ antara prefiks dan bentuk dasar. Penambahan fonem nasal /m/ terjadi apabila bentuk dasarnya diawali oleh konsonan /b/. Contoh:

*pe + baca → pembaca*

*pe + bina → pembina*

*pe + buru → pemburu*

Penambahan fonem nasal /n/ terjadi apabila bentuk dasarnya diawali oleh konsonan /c/, /d/, dan /j/. Contoh:

*pe + cari → pencari*

*pe + dengar → pendengar*

*pe + jual → penjual*

Penambahan fonem nasal /ng/ terjadi apabila bentuk dasarnya diawali dengan konsonan /g, h, dan kh/ dan vokal /a, i, u, e, o/. Contoh:

*pe + gali → penggali*

*pe + hambat → penghambat*

*pe + khianat → pengkhianat*

*pe + angkat → pengangkat*

*pe + inap → penginap*

*pe + usir → pengusir*

*pe + elak → pengelak*

*pe + obral → pengobral*

Penambahan fonem nasal /nge/ terjadi apabila bentuk dasarnya berupa bentuk dasar satu suku. Contoh:

*pe + bom → pengebom*

*pe + cat → pengecat*

*pe + lap → pengelap*

3. Peluluhan fonem, apabila prefiks *pe-* (atau *pe-an*) diimbuhkan pada bentuk dasar yang diawali dengan konsonan tak bersuara /s, k, p, dan t/. Dalam hal ini konsonan /s/ diluluhkan dengan nasal /ny/, konsonan /k/ diluluhkan dengan nasal /ng/, konsonan /p/ diluluhkan dengan nasal /m/, dan konsonan /t/ diluluhkan dengan nasal /n/. Contoh:

*pe + saring → penyaring*

*pe + kumpul → pengumpul*

*pe + pilih → pemilih*

*pe + tulis → penulis*

#### 2.4.5 Prefiksasi *per-* dan konfiksasi *per-an*

Morfofonemik dalam pengimbuhan prefiks *per-* dan konfiks *per-an* dapat berupa: (a) pelesapan fonem /r/ pada prefiks *per-* itu; (b) perubahan fonem /r/ dari prefiks *per-* itu menjadi fonem /l/; dan (c) pengekal fonem /r/ tetap /r/.

1. Pelesapan fonem /r/ terjadi apabila bentuk dasarnya dimulai dengan fonem /r/, atau suku kata pertamanya /er/. Contoh:

*per + ringan → peringan*

*per + rendah → perendah*

*per + ternak → peternak*

*per + kerja → pekerja*

2. Perubahan fonem /r/ menjadi /l/ terjadi apabila bentuk dasarnya berupa kata *ajar*.

*per + ajar → pelajar*

3. Pengekal fonem /r/ terjadi apabila bentuk dasarnya bukan yang disebutkan pada poin 1 dan 2 di atas. Contoh:

*per + kaya → perkaya*

*per + kecil → perkecil*

*per + lambat → perlambat*

*per + tegas → pertegas*

#### 2.4.6 Prefiksasi *ter-*

Morfofonemik dalam proses pengimbuhan dengan prefiks *ter-* dapat berupa: (a) pelesapan fonem /r/ dari prefiks *ter-* itu; (b) perubahan fonem /r/ dari prefiks *ter-* itu menjadi fonem /l/; dan (c) pengekal fonem /r/ itu.



- 1 1. Pelepasan fonem dapat terjadi apabila prefiks *ter-* diimbuhkan pada bentuk dasar yang dimulai  
2 dengan konsonan /r/. Misalnya:  
3 *ter + rasa → terasa*  
4 *ter + rangkum → terangkum*  
5 *ter + rebut → terebut*
- 6 2. Perubahan fonem /r/ pada prefiks *ter-* menjadi fonem /l/ terjadi apabila prefiks *ter-* itu  
7 diimbuhkan pada bentuk dasar *anjur*.  
8 *ter + anjur → telanjur*
- 9 3. Pengekalan fonem /r/ pada prefiks *ter-* tetap menjadi /r/ apabila prefiks *ter-* itu diimbuhkan  
10 pada bentuk dasar yang bukan disebutkan pada poin 1 dan 2 di atas. Contoh:  
11 *ter + dengar → terdengar*  
12 *ter + jauh → terjauh*  
13 *ter + lempar → terlempar*  
14 *ter + baik → terbaik*

## 2.5 Peran Morphological Parser dalam Natural Language Processing[1]

Dalam bahasa Inggris, ketika kita ingin menulis sebuah kata benda plural, untuk sebagian besar kasus kita hanya tinggal menambahkan huruf 's' di belakang kata benda yang dimaksud. Misalnya, kita ingin membuat bentuk plural dari kata 'book' kita hanya perlu menambahkan huruf 's' di belakangnya sehingga menjadi kata 'books' yang merupakan bentuk plural dari 'book'. Namun, hal itu tidak berlaku jika kita ingin membuat bentuk plural dari kata 'baby', 'goose', atau 'fish'. Bentuk plural dari kata 'baby', 'goose', dan 'fish' adalah 'babies', 'geese', dan 'fish'.

Untuk mengenali bentuk 'books' dapat dipisahkan menjadi dua buah morfem 'book' dan 's' diperlukan proses yang disebut dengan **morphological parsing**. Dalam ranah ilmu temu kembali informasi (*information retrieval*), proses yang sama untuk memetakan bentuk 'books' menjadi 'book' disebut dengan *stemming*. Morphological parsing atau stemming tidak hanya dapat memisahkan bentuk plural dari kata benda, tapi juga dapat untuk bentuk lain seperti bentuk kata kerja berakhiran 'ing' dalam contoh kata 'going', 'talking', dan 'eating'. Ketika dilakukan proses parsing terhadap bentuk tersebut akan didapatkan kata kerja 'go', 'talk', dan 'eat' yang ditambahkan morfem 'ing'.

Muncul pertanyaan, kenapa kita tidak simpan saja semua bentuk plural dari semua kata benda dan semua bentuk 'ing' dari semua kata kerja dalam bahasa Inggris yang ada dalam kamus? Alasan utamanya adalah karena bentuk 'ing' adalah sufiks yang sangat produktif, yang berarti bentuk tersebut dapat diterapkan pada semua kata kerja. Sama halnya dengan bentuk plural 's' yang bisa diterapkan di hampir semua kata benda. Oleh karena itu, ide untuk menyimpan semua bentuk plural dan bentuk 'ing' akan sangat tidak efisien.

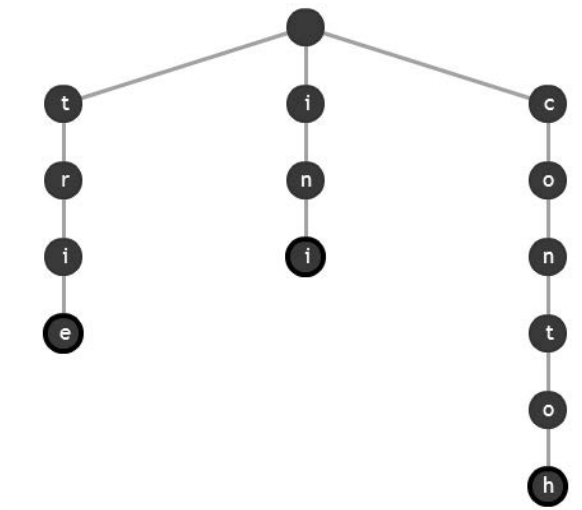
Morphological parsing dibutuhkan tidak hanya untuk temu kembali informasi. Kita membutuhkan proses tersebut untuk supaya mesin dapat mengerti bahwa kata 'va' dan 'aller' dalam bahasa Perancis keduanya diterjemahkan menjadi kata kerja 'go' dalam bahasa Inggris. Kita juga membutuhkan proses tersebut untuk mengecek ejaan, karena dengan aturan morfologi kita dapat

- 1 menentukan bahwa kata 'misclam' dan 'antiundoggingly' bukan merupakan kata yang valid dalam
- 2 bahasa Inggris.

## 3 2.6 Struktur Data Trie

4 *Trie* adalah struktur data berupa pohon terurut untuk menyimpan suatu himpunan string di mana  
 5 setiap node pada pohon tersebut mengandung awalan (prefix) yang sama[3]. Kata "trie" berasal  
 6 dari kata *retrieval* yang berarti pengambilan. Struktur data trie ditemukan oleh seorang professor  
 7 di MIT bernama Edward Fredkin. Trie sering digunakan pada masalah komputasi yang melibatkan  
 8 penyimpanan dan pencarian string. Trie memiliki sejumlah keunggulan dibanding struktur data lain  
 9 untuk memecahkan masalah serupa terutama dalam hal kecepatan dan memori yang digunakan.

10 Dalam trie, tidak ada node yang menyimpan kunci yang terkait dengan node tersebut, sebaliknya,  
 11 posisinya di pohon menunjukkan kunci apa yang terkait dengannya. Setiap keturunan dari sebuah  
 12 node memiliki prefix yang sama dengan string yang diwakilkan oleh node tersebut, dan akar  
 13 menandakan sebuah string kosong.



Gambar 2.3: Trie dengan kata "trie", "ini", dan "contoh"[3]

14 Gambar 2.3 di atas adalah contoh representasi struktur data trie yang menyimpan tiga buah  
 15 string, yaitu "trie", "ini", dan "contoh". Dari gambar tersebut kita bisa mendapat gambaran mengenai  
 16 kompleksitas waktu yang diperlukan untuk mencari sebuah kata dalam trie. Jika panjang kata  
 17 terpanjang dalam trie adalah  $L$ , maka untuk mencari sebuah kata dalam trie memerlukan waktu  
 18 terburuk  $O(L)$

### 19 2.6.1 Bitwise Trie

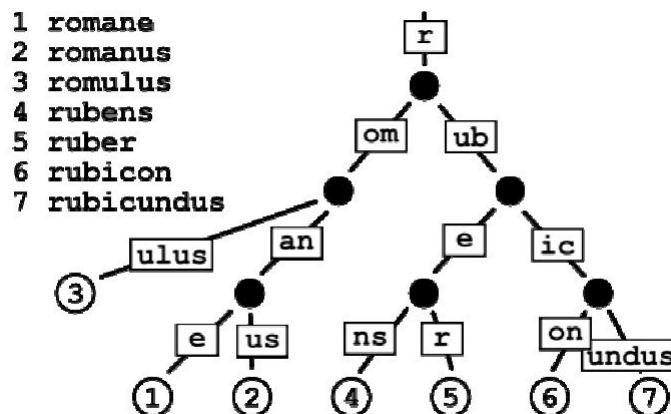
20 Bitwise trie adalah salah satu variasi dari trie yang memiliki banyak kesamaan dengan trie berbasis  
 21 karakter biasa, kecuali dalam representasi dengan bit individual yang biasanya digunakan untuk  
 22 traversal secara efektif dan membentuk sebuah pohon biner. Secara umum, implementasinya  
 23 menggunakan fungsi khusus CPU untuk dapat secara cepat mencari himpunan bit dengan panjang  
 24 tertentu. Nilai ini lalu akan digunakan sebagai entri dari tabel dengan indeks 32 atau 64 yang

1 menunjuk kepada elemen pertama dalam bitwise trie dengan sejumlah bilangan 0 di depan. Proses  
 2 pencarian selanjutnya akan dilakukan dengan mengetes setiap bit dalam kunci dan memilih anak[0]  
 3 atau anak[1] sesuai aturan hingga pencarian berakhir.

4 Walaupun proses ini mungkin terdengar lambat, tetapi sangat fleksibel karena kurangnya  
 5 ketergantungan terhadap *register* dan oleh karena itu pada kenyataannya melakukan eksekusi  
 6 dengan sangat baik pada CPU modern.

## 7 2.6.2 Patricia Trie

8 PATRICIA adalah variasi lain dari trie yang merupakan singkatan dari Practical Algorithm  
 9 To Retrieve Information Coded In Alphanumeric. PATRICIA trie sendiri lebih dikenal dengan  
 10 sebutan *pohon radix* atau *radix tree*. Pohon radix bisa diartikan secara sederhana sebagai trie yang  
 11 kompleksitas ruangnya lebih efisien, di mana setiap node yang hanya memiliki satu anak digabung  
 12 dengan anaknya sendiri. Hasilnya adalah setiap node paling dalam paling tidak memiliki 2 anak.  
 13 Tidak seperti trie biasa, anak bisa diberi label deretan karakter maupun satu karakter. Ini membuat  
 14 pohon radix jauh lebih efisien untuk jumlah string yang sedikit (terutama jika stringnya cukup  
 15 panjang) dan untuk himpunan string yang memiliki prefix sama yang panjang.



Gambar 2.4: Pohon radix dengan 7 kata dengan prefix "r"[3]

16 Pohon radix memiliki fasilitas untuk melakukan operasi-operasi berikut, yang mana setiap  
 17 operasinya memiliki kompleksitas waktu terburuk  $O(k)$ , di mana  $k$  adalah panjang maksimum string  
 18 dalam himpunan.

- 19 • Pencarian: Mencari keberadaan suatu string pada himpunan string. Operasi ini sama dengan  
 20 pencarian pada trie biasa kecuali beberapa sisi mengandung lebih dari satu karakter.
- 21 • Penyisipan: Menambahkan sebuah string ke pohon. Kita mencari tempat yang tepat di pohon  
 22 untuk menyisipkan elemen baru. Jika sudah ada sisi yang memiliki prefix sama dengan string  
 23 masukan, kita akan memisahkan nya menjadi dua sisi dan memprosesnya. Proses pemisahan  
 24 ini meyakinkan bahwa tidak ada node yang memiliki anak lebih banyak dari jumlah karakter  
 25 string yang ada.

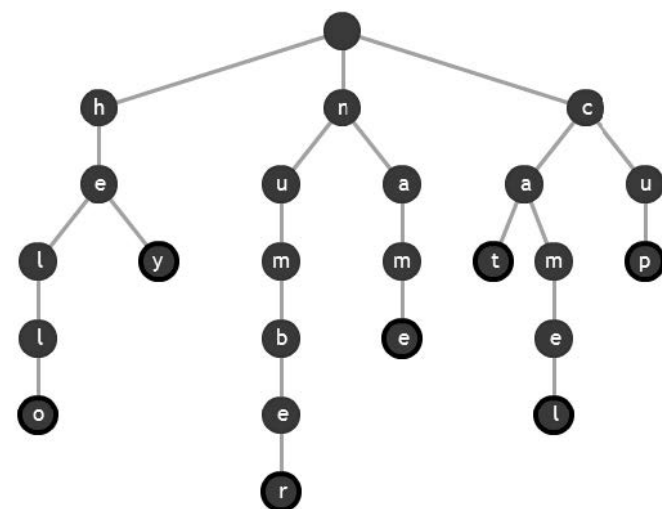
- Hapus: Menghapus sebuah string dari pohon. Pertama kita menghapus daun yang berkaitan. Lalu, jika orangtuanya hanya memiliki satu anak lagi, kita menghapus orangtuanya dan menggabungkan sisi yang saling terhubung
- Cari anak: Mencari string terbesar yang lebih kecil dari string masukan, sesuai dengan urutan alfabet.
- Cari orang tua: Mencari string terkecil yang lebih besar dari string masukan, sesuai dengan urutan alfabet.

Pengembangan yang umum dari pohon radix yaitu menggunakan node dua warna, hitam dan putih. Untuk mengecek apakah sebuah string masukan sudah ada di dalam pohon, pencarian dimulai dari puncak, dan terus menelusuri setiap sisi sampai tidak ada lagi jalan. Jika node akhir dari proses ini berwarna hitam, berarti pencarian gagal, jika node berwarna putih berarti pencarian telah berhasil. Hal ini membuat kita bisa menambahkan string dalam jumlah banyak yang memiliki prefix yang sama dengan elemen di pohon dengan menggunakan node putih, lalu menghapus sejumlah pengecualian untuk menghemat memori dengan cara menambahkan elemen string baru dengan node hitam.

### 2.6.3 Implementasi Trie dalam Kamus

Salah satu implementasi dari struktur data trie yang paling populer adalah dalam kamus. Kamus terdiri dari kumpulan kata-kata yang sudah terurut menaik berdasarkan urutan alfabet. Dalam perkembangannya saat ini sudah banyak kamus yang hadir dalam bentuk perangkat lunak, yang bisa digunakan di komputer ataupun di telepon genggam. Kamus dalam bentuk perangkat lunak tentunya memiliki fitur-fitur yang memudahkan pengaksesannya, antara lain pencarian kata dan penambahan kata ke kamus.

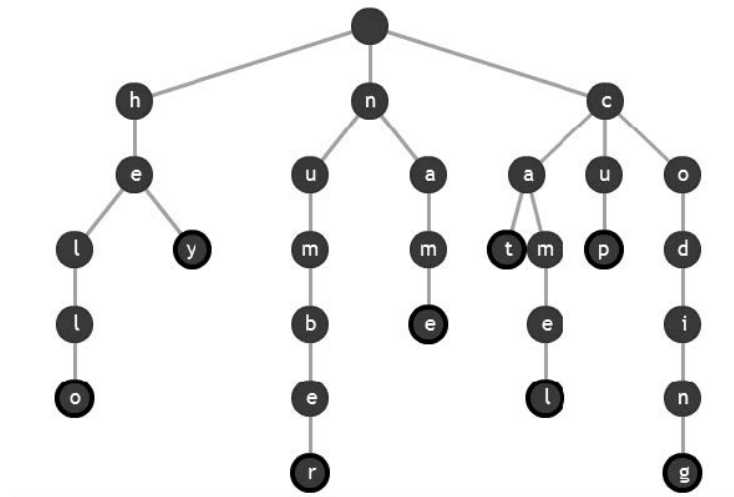
Berikut adalah ilustrasi kamus dengan struktur data trie.



Gambar 2.5: Trie dengan kata "hello", "hey", "number", "name", "cat", "camel", dan "cup"<sup>[3]</sup>

Dapat dibayangkan, ketika kita mencari sebuah kata dalam kamus, kita akan mulai dengan karakter pertama dari kata tersebut. Jika tidak ada anak dari akar yang nilainya sama dengan karakter itu, maka langsung disimpulkan kata tidak ada di kamus. Jika ada, akan ditelusuri terus sampai ke dasar dari trie, jika kata ditemukan, maka akan dikembalikan info dari node itu, sedangkan jika tidak ketemu kita bisa mengembalikan kata yang memiliki prefix sama dengan kata yang dicari sebagai saran pencarian.

Sementara itu penambahan kata akan berakibat penambahan cabang baru bila kata itu belum ada sebelumnya.



Gambar 2.6: Penambahan kata "coding" pada trie di gambar 2.5[3]

#### 2.6.4 Keunggulan Trie dibandingkan Struktur Data Lain

Trie sebagai turunan dari pohon memiliki keunggulan dibandingkan stuktur data yang sering digunakan untuk persoalan yang sama, yakni pohon pencarian biner dan tabel hash.

Berikut keunggulan utama trie dibandingkan pohon pencarian biner:

- Pencarian kunci dengan trie lebih cepat. Mencari sebuah kunci dengan panjang  $m$  menghabiskan waktu dengan kasus terburuk  $O(m)$ . Pohon pencarian biner melakukan  $O(\log(n))$  perbandingan kunci, dimana  $n$  adalah jumlah elemen di dalam pohon, karena pencarian pada pohon biner bergantung pada kedalaman pohon, yang mana bernilai logaritmik terhadap jumlah kunci pencarian apabila pohonnya seimbang. Oleh karena itu pada kasus terburuk, sebuah pohon biner menghabiskan waktu  $O(m \log n)$ , yang mana pada kasus terburuk juga  $\log n$  akan mendekati  $m$ . Operasi sederhana yang digunakan trie pada saat pencarian karakter, seperti penggunaan array index menggunakan karakter, juga membuat pencarian dengan trie menjadi lebih cepat.
- Trie menggunakan ruang lebih sedikit jika memuat string pendek dalam jumlah besar, karena kunci tidak disimpan secara eksplisit dan node dipakai bersama oleh kunci yang memiliki prefix yang serupa.
- Trie bisa memiliki fitur untuk menghitung kesamaan prefix terpanjang, yang membantu untuk mencari penggunaan kunci bersama terpanjang dari karakter-karakter yang unik.

Berikut keunggulan utama trie dibanding tabel hash:

- Trie bisa melakukan pencarian kunci yang paling mirip hampir sama cepatnya dengan pencarian kunci yang tepat, sementara tabel hash hanya bisa mencari kunci yang sama tepat karena tidak menyimpan hubungan antara kunci.
- Trie lebih cepat secara rata-rata untuk menyisipkan elemen baru dibandingkan dengan tabel hash. Hal ini terjadi karena tabel hash harus membangun ulang indeksinya ketika tabel sudah penuh, yang mana menghabiskan waktu sangat banyak. Oleh karena itu, trie memiliki kompleksitas waktu terburuk yang batasnya lebih konsisten, yang mana merupakan salah satu unsur penting pada jalannya sebuah program.
- Trie bisa diimplementasikan sedemikian sehingga tidak memerlukan memori tambahan. Tabel hash harus selalu memiliki memori tambahan untuk menyimpan pengindeksan tabel hash.
- Pencarian kunci bisa jauh lebih cepat jika fungsi hashing dapat dihindarkan. Trie bisa menyimpan kunci bertipe integer maupun pointer tanpa perlu membuat fungsi hashing sebelumnya. Hal ini membuat trie lebih cepat daripada tabel hash pada hampir setiap kasus karena fungsi hash yang baik sekalipun cenderung overhead ketika melakukan hashing pada data yang hanya berukuran 4 sampai 8 byte.
- Trie bisa menghitung kesamaan prefix terpanjang, sedangkan tabel hash tidak.

## BAB 3

## ANALISIS

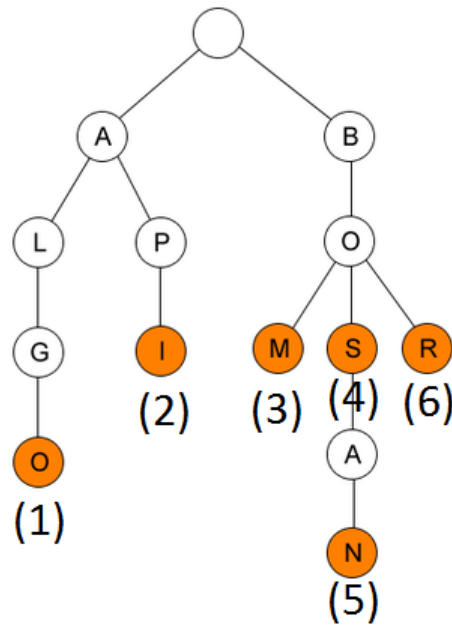
Pada bab ini dijelaskan mengenai hasil analisis yang dilakukan untuk penelitian ini, yaitu analisis mengenai leksikon, proses morphological parsing, morfotaktik, use case dan skenario dari perangkat lunak, dan yang terakhir diagram kelas dari perangkat lunak yang akan dibangun.

### 3.1 Leksikon

Leksikon, seperti ditulis pada subbab 2.2.3, dapat dipadankan dengan istilah *kosakata* atau *perbendaharaan kata*. Leksikon dibutuhkan pada proses *morphological parsing* untuk mengetahui apakah sebuah kata yang sedang diproses adalah sebuah kata dasar yang valid atau tidak dalam bahasa Indonesia. Leksikon menyimpan kumpulan kata dasar dan turunannya untuk nantinya diakses ketika proses *morphological parsing* dilakukan. Kata turunan adalah kata yang merupakan hasil dari proses morfologi berupa afiksasi, reduplikasi, dan komposisi dari kata dasar yang bersangkutan.

Leksikon dalam proses *morphological parsing* harus bisa diakses dengan cepat dan efektif. Hal ini dikarenakan leksikon akan diakses sangat sering dalam proses ini. Leksikon akan diakses sekitar 3-5 kali untuk setiap kata yang sedang diproses. Oleh karena itu, leksikon perlu disimpan pada struktur data yang memungkinkan waktu akses yang cepat supaya keseluruhan proses dapat dijalankan dalam waktu yang masuk akal.

Struktur data yang saat ini terkenal paling cepat untuk diakses adalah struktur data *trie*. Trie adalah struktur data berbentuk pohon yang menyimpan himpunan string yang jika ditelusuri setiap node mulai dari akar hingga daun akan membentuk suatu string yang merupakan kunci yang kita cari. Setiap string yang dihasilkan dari node awal yang sama akan mempunyai awalan (prefiks) yang sama, karena itulah trie disebut juga pohon prefiks.



Gambar 3.1: Struktur data trie

Struktur data trie yang digambarkan pada bagan 3.1 menyimpan enam string kunci dari dua buah awalan, yaitu string "A" dan "B". Jika kita telusuri dari node akar "A" sampai node daun "O", kita akan mendapat string "ALGO" yang ditandai dengan nomor (1). String lain yang disimpan pada contoh tersebut adalah string "API" pada nomor (2), string "BOM" pada nomor (3), string "BOS" pada nomor (4), string "BOSAN" pada nomor (5), dan string "BOR" pada nomor (6).

Perlu diperhatikan bahwa sebuah string kunci tidak harus disimpan dengan node terakhir ada pada posisi daun, seperti pada string "BOS" pada nomor 4. Node terakhir pada string tersebut merupakan node internal. Penyimpanan seperti ini bisa dilakukan dengan menandai setiap node yang merupakan akhir dari sebuah string yang membentuk kata.

Ada dua jenis kata yang disimpan dalam leksikon, yaitu kata dasar dan kata turunan. Contoh kata dasar adalah kata 'sapu', 'makan', dan 'kerja' sementara contoh kata turunan adalah kata 'menyapu', 'makan-makan', dan 'kerja bakti'. Kata-kata turunan ini adalah kata yang merupakan hasil dari proses morfologi berupa afiksasi, reduplikasi, atau komposisi. Kata turunan disimpan sebagai bagian dari kata dasar dan dapat diakses ketika dibutuhkan.

Dalam Kamus Besar Bahasa Indonesia, ada beberapa kata yang merupakan hasil dari proses morfologi yang sudah diserap dan dianggap sebagai sebuah kata dasar. Contohnya adalah kata 'gerigi' yang merupakan hasil penyisipan infiks -er- pada kata 'gigi', kata 'abu-abu' yang merupakan hasil reduplikasi dari kata 'abu', dan kata 'rumah sakit' yang merupakan hasil komposisi dari kata 'rumah' dan kata 'sakit'. Dalam kasus tersebut, untuk kata yang merupakan hasil penyisipan infiks akan disimpan sebagai sebuah kata dasar dalam leksikon sementara untuk kata yang merupakan hasil reduplikasi dan komposisi akan disimpan sebagai kata turunan dari kata dasar yang bersangkutan.

Untuk kata turunan yang merupakan hasil afiksasi berupa pengimbuhan klitika, kata tersebut tidak disimpan sebagai bentuk turunan dalam leksikon karena dianggap terlalu produktif. Contohnya adalah kata 'bukumu' tidak disimpan sebagai bentuk turunan dari kata 'buku' dan kata 'mobilnya' tidak disimpan sebagai bentuk turunan dari kata 'mobil'. Sementara ada kasus di mana klitika -nya



merupakan bagian dari konfiks *se-nya*, di mana konfiks adalah gabungan antara prefiks dan sufiks yang diimbuhkan secara bersamaan. Timbul kerancuan apakah *-nya* di sini akan dianggap sebagai klitika atau sufiks. Untuk penelitian kali ini, bentuk *-nya* akan dianggap sebagai klitika karena *-nya* sebagai klitika dianggap lebih produktif daripada sebagai sufiks dalam kesatuan konfiks *se-nya*.

Dalam Kamus Besar Bahasa Indonesia Dalam Jaringan (KBBI daring)<sup>1</sup>, kata dasar dan kata turunan disimpan secara terpisah namun keduanya dapat dicari melalui kolom pencarian. Sementara pada Kamus Besar Bahasa Indonesia Luar Jaringan (KBBI luring)<sup>2</sup>, hanya kata dasar saja yang bisa dicari melalui kolom pencarian namun semua kata turunannya juga disimpan sebagai bagian dari sebuah kata dasar. Pada penelitian kali ini akan digunakan struktur penyimpanan dan pencarian seperti pada KBBI luring.

Struktur penyimpanan seperti pada KBBI luring memungkinkan untuk mengenali perbedaan antara kata dasar dan kata yang telah melalui proses morfologi seperti afiksasi, reduplikasi, dan komposisi. Perangkat lunak yang dirancang pada penelitian ini harus dapat menentukan apakah sebuah kata merupakan kata dasar yang valid atau tidak dalam bahasa Indonesia. Pencarian kata dasar dalam leksikon harus dapat melakukan hal tersebut sehingga pencarian hanya bisa dilakukan terhadap kata dasar saja dan tidak dengan kata turunannya. Kata turunan perlu disimpan dalam leksikon untuk melakukan validasi terhadap hasil dari proses parsing yang dilakukan oleh perangkat lunak.

## 3.2 Proses *Morphological Parsing*

Pada subbab 2.3 telah dibahas mengenai proses morfologi, yang pada dasarnya adalah proses pembentukan kata melalui beberapa proses, yaitu pembubuhan afiks (afiksasi), pengulangan (reduplikasi), dan penggabungan (komposisi). Proses *morphological parsing* merupakan kebalikan dari proses morfologi. Masukan bagi proses *morphological parsing* adalah kata atau kalimat yang telah melalui proses morfologi dan keluarannya adalah komponen-komponen penyusunnya.

Proses *morphological parsing* untuk setiap kata dalam masukan dapat dituliskan sebagai berikut:

1. Periksa leksikon, jika kata tersebut ada dalam leksikon, masukkan sebagai salah satu kemungkinan keluaran
2. Periksa adanya kemungkinan afiks, baik itu prefiks, sufiks, maupun konfiks. Pisahkan afiks yang ditemukan dengan komponen kata yang lain dan lakukan proses parsing pada komponen kata tersebut
3. Periksa adanya simbol penghubung (-), yang menandakan hasil proses reduplikasi, lalu lakukan proses parsing terhadap bentuk dasar dari kata tersebut
4. Jika ada kata yang mengikuti, periksa kemungkinan kata yang sedang diproses dan kata yang mengikuti adalah dua kata hasil komposisi dengan melakukan pengecekan terhadap bentuk dasar dari kata tersebut

Leksikon yang dibuat dalam perangkat lunak ini juga menyimpan kata turunan yang valid dari setiap kata dasar yang ada. Setelah proses parsing selesai dilakukan, leksikon dapat melakukan

<sup>1</sup><https://kbbi.kemdikbud.go.id/>

<sup>2</sup><http://ebsoft.web.id/kbbi-kamus-besar-bahasa-indonesia-offline-gratis/>

1 validasi apakah kata turunan yang sudah diproses benar merupakan kata turunan yang valid dari  
 2 kata dasar yang bersangkutan.

3 Sebagai contoh, jika dilakukan proses *morphological parsing* pada kata 'kemerah-merahan', maka  
 4 prosesnya adalah sebagai berikut:

- 5 • Periksa leksikon, kata tersebut tidak ditemukan dalam leksikon
- 6 • Periksa kemungkinan afiks, ditemukan kemungkinan konfiks {ke-an} dan klofiks {ke-an},  
 7 lakukan proses parsing terhadap kata 'merah-merah'
- 8 • Ditemukan simbol penghubung (-) sehingga diketahui kata tersebut adalah hasil proses  
 9 reduplikasi. Pisahkan kata dan lakukan proses parsing sehingga didapat hasilnya adalah  
 10 reduplikasi dari kata dasar 'merah'
- 11 • Didapat dua kemungkinan hasil, yaitu reduplikasi kata 'merah' diikuti konfiksasi {ke-an} dan  
 12 reduplikasi kata 'merah' diikuti klofiksasi {ke-an}
- 13 • Lakukan validasi pada leksikon, dan didapatkan kata turunan yang valid adalah reduplikasi  
 14 kata 'merah' diikuti konfiksasi {ke-an}
- 15 • Hasil akhir proses parsing adalah bentuk dasar {merah} + reduplikasi + konfiks {ke-an}

16 Untuk kata dengan kemungkinan hasil parsing lebih dari satu, seperti kata 'beruang', prosesnya  
 17 adalah sebagai berikut:

- 18 • Periksa leksikon, ditemukan bentuk dasar {beruang}, masukkan sebagai salah satu kemung-  
 19 kinan keluaran
- 20 • Periksa kemungkinan afiks pada kata 'beruang'
- 21 • Didapatkan prefiks {ber-} + bentuk dasar {uang}, masukkan sebagai salah satu kemungkinan  
 22 keluaran
- 23 • Periksa kemungkinan adanya fonem yang dilesapkan pada bentuk dasar, yaitu fonem 'r', dan  
 24 didapatkan prefiks {ber-} + bentuk dasar {ruang}, masukkan sebagai salah satu kemungkinan  
 25 keluaran
- 26 • Lakukan validasi pada leksikon terhadap kata turunan dari bentuk dasar {uang} dan {ruang}
- 27 • Hasil akhir proses parsing adalah bentuk dasar {beruang}, prefiks {ber-} + bentuk dasar  
 28 {uang}, dan prefiks {ber-} + bentuk dasar {ruang}

29 Bentuk-bentuk yang tidak secara khusus ada dalam bahasa Indonesia seperti angka, nama orang,  
 30 dan kata dalam bahasa asing ditulis sebagai *bentuk asing* dalam keluaran dari proses parsing.

31 Beberapa contoh yang sudah dibahas di atas adalah contoh proses parsing yang dilakukan pada  
 32 sebuah kata dalam bahasa Indonesia. Perangkat lunak *morphological parser* yang dirancang pada  
 33 penelitian ini akan dapat memproses tidak hanya kata tapi juga kalimat dan paragraf yang ditulis  
 34 dalam bahasa Indonesia. Proses parsing pada kalimat dan paragraf memerlukan beberapa langkah  
 35 tambahan yaitu:

- 1 1. Hilangkan tanda baca yang tidak diperlukan dalam proses parsing. Tanda baca yang diperlukan
- 2 dalam proses parsing hanya tanda baca penghubung kata (-) sebagai tanda hasil proses
- 3 reduplikasi
- 4 2. Gantikan tanda baca yang dihilangkan dengan karakter kosong (" ")
- 5 3. Pisahkan setiap kata berdasarkan karakter spasi yang memisahkan kata supaya proses parsing
- 6 dapat dilakukan untuk setiap kata yang sudah dipisahkan

### 7 3.3 Morfotaktik

8 Pada subbab 2.2.3 dan 2.2.4 telah dijelaskan mengenai morfem dasar dan morfem afiks. Morfem  
 9 dasar adalah morfem yang dapat menjadi dasar dalam suatu proses morfologi. Sementara morfem  
 10 afiks adalah morfem yang tidak dapat menjadi dasar dalam pembentukan kata, tetapi hanya menjadi  
 11 unsur pembentuk dalam proses afiksasi. Kedua morfem tersebut dapat digabungkan dalam proses  
 12 morfologi berupa proses afiksasi untuk membentuk sebuah kata. Proses penggabungan antara kedua  
 13 morfem tersebut tidak boleh dilakukan secara sembarangan sehingga diperlukan aturan khusus  
 14 yang mengatur penggabungan antara morfem dasar dan morfem afiks. Aturan ini disebut dengan  
 15 morfotaktik.

16 Morfem afiks terdiri dari prefiks, sufiks, dan konfiks. Jenis-jenis prefiks adalah prefiks *ber-*,  
 17 prefiks *me-*, prefiks *per-*, prefiks *pe-*, prefiks *di-*, prefiks *ter-*, prefiks *se-*, dan prefiks *ke-*. Jenis-jenis  
 18 sufiks adalah sufiks *-kan*, sufiks *-i*, dan sufiks *-an*. Jenis-jenis konfiks adalah konfiks *ke-an*, konfiks  
 19 *ber-an*, konfiks *pe-an*, konfiks *per-an*, dan konfiks *se-nya*. Bahasa Indonesia membolehkan sebuah  
 20 kata untuk dibentuk dengan dua buah prefiks, seperti contoh kata 'memperkuat' yang dibentuk dari  
 21 kata dasar 'kuat' dengan dua buah prefiks, yaitu prefiks *me-* dan prefiks *per-*. Afiks yang diletakkan  
 22 sebelum prefiks ini akan disebut dengan *preprefiks*.

23 Selain afiks yang sudah disebutkan, ada juga morfem afiks yang disebut dengan klitika. Berda-  
 24 sarkan letaknya, dibedakan adanya *proklitika*, yaitu klitika yang berposisi di sebelah kiri kata yang  
 25 diikuti dan *enklitika* adalah klitika yang berposisi di belakang kata yang dilekati. Contoh proklitika  
 26 adalah klitika *ku-* dan *kau-* dalam bentuk *kubawa* dan *kauambil*. Sementara contoh enklitika adalah  
 27 klitika *-ku*, *-mu*, *-nya*, dan *-lah* pada bentuk *bukuku*, *nasibmu*, *duduknya*, dan *pergilah*. Secara  
 28 pertuturan, klitika pada umumnya diletakkan di paling kiri atau di paling kanan dari sebuah kata,  
 29 sehingga tidak mungkin ada morfem afiks atau morfem dasar lain yang diletakkan di sebelah kiri  
 30 proklitika atau di sebelah kanan enklitika pada sebuah kata.

31 Aturan morfotaktik tidak hanya mengatur tentang proses penggabungan antara morfem dasar  
 32 dengan morfem afiks, namun juga mengatur tentang proses pengulangan morfem dasar dalam  
 33 proses reduplikasi dan proses penggabungan antara morfem dasar dengan morfem dasar lain dalam  
 34 proses komposisi. Proses reduplikasi dan komposisi pun tidak boleh dilakukan secara sembarangan  
 35 sehingga proses tersebut perlu juga diatur dalam aturan morfotaktik.

36 Aturan morfotaktik yang digunakan pada penelitian ini akan dijelaskan melalui tabel. Akan  
 37 ada beberapa tabel untuk mewakili setiap jenis morfem, yaitu preprefiks, prefiks, akar, sufiks, dan  
 38 konfiks. Setiap tabel akan menunjukkan komponen tersebut boleh didahului oleh komponen apa  
 39 saja dan boleh diikuti oleh komponen apa saja. Komponen 'null' berarti boleh tidak didahului

- 1 atau diikuti oleh komponen apapun. Berikut adalah tabel aturan morfotaktik untuk preprefiks,  
 2 prefiks/konfiks, akar, dan sufiks/konfiks.

Pendahulu	Preprefiks	Pengikut
proklitika null	me- di-	per- ke- ber- ter-
proklitika null	pe-	ber-
proklitika null	ke-	ber- pe-
proklitika null	ber-	ke- pe-

Tabel 3.1: Tabel aturan morfotaktik untuk preprefiks

Pendahulu	Prefiks/Konfiks	Pengikut
proklitika me- di- null	per- ter-	akar
proklitika null	me- di- se-	akar
proklitika me- di- ber- null	ke-	akar
proklitika me- di- pe- ke- null	ber-	akar
proklitika ke- ber- null	pe-	akar

Tabel 3.2: Tabel aturan morfotaktik untuk prefiks atau konfiks

Pendahulu	Akar	Pengikut
proklitika prefiks null	akar	konfiks sufiks reduplikasi komposisi enklitika null

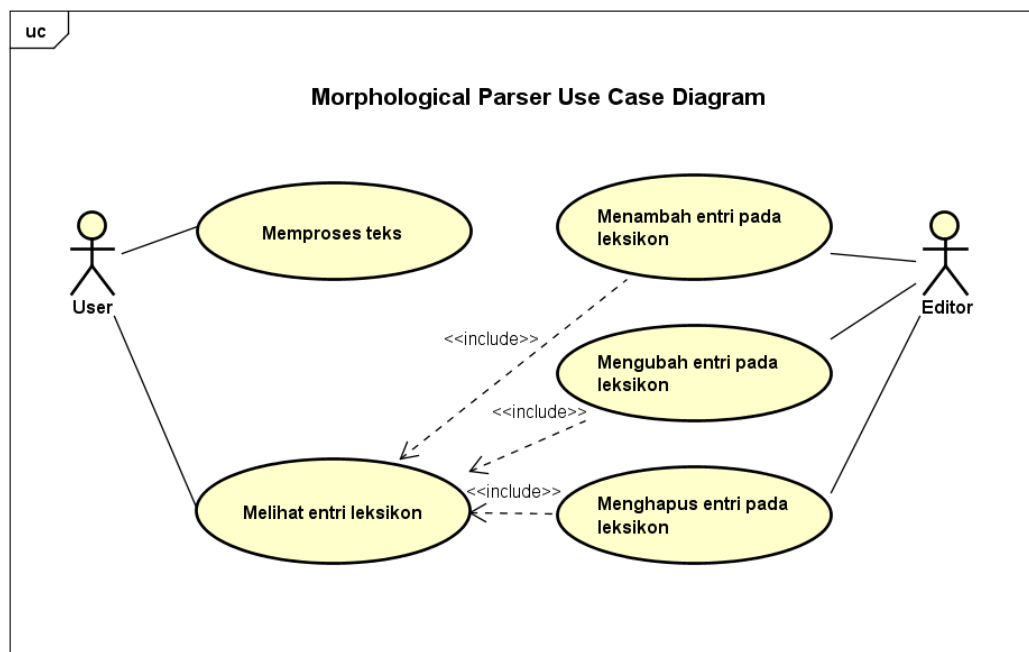
Tabel 3.3: Tabel aturan morfotaktik untuk akar

Pendahulu	Sufiks/Konfiks	Pengikut
akar	-kan -an -i	reduplikasi komposisi enklitika null

Tabel 3.4: Tabel aturan morfotaktik untuk sufiks atau konfiks

### 3.4 Analisis Use Case

Perangkat lunak *morphological parser* yang akan dibangun dapat memproses masukan berupa teks dalam bahasa Indonesia yang dapat dimasukkan ke dalam perangkat lunak melalui dua cara, melalui kolom masukan dan melalui file teks. Perangkat lunak juga memiliki leksikon yang isinya dapat dilihat oleh user. Selain itu, terdapat user khusus yang disebut dengan editor yang dapat melakukan penambahan, pengubahan, dan penghapusan entri pada leksikon melalui perangkat lunak ini. Fitur-fitur ini dapat digambarkan dalam diagram use case pada gambar 3.2.



Gambar 3.2: Diagram use case perangkat lunak morphological parser

Dari diagram tersebut dapat dituliskan use case scenario sebagai berikut:

## MEMPROSES TEKS

**Name:** Memproses teks

**Actors:** User

**Goals:** User berhasil memproses teks melalui sistem

**Precondition:** Teks sudah disiapkan

**Steps:**

Actor actions	System responses
1. User memilih pilihan untuk memasukkan teks melalui file	2. Sistem menampilkan kotak dialog untuk memilih file
3. User mengarahkan kotak dialog ke direktori tempat file teks masukan	
4. User menekan tombol "OK"	5. Sistem menampilkan isi file ke dalam kolom masukan
6. User menekan tombol "Proses"	7. Sistem menampilkan hasil proses ke dalam kolom keluaran

**Alternate flow:**

Actor actions	System responses
1a. User menulis teks ke dalam kolom masukan 2a. User menekan tombol "Proses"	3a. Sistem menampilkan hasil proses ke dalam kolom keluaran

## MELIHAT ENTRI LEKSIKON

**Name:** Melihat entri leksikon

**Actors:** User

**Goals:** User dapat melihat semua entri leksikon yang ada dalam sistem

**Precondition:** Sistem sudah memuat leksikon ke dalam program

**Steps:**

Actor actions	System responses
1. User memilih pilihan untuk melihat leksikon	2. Sistem menampilkan leksikon yang ada dalam sistem

## MENAMBAH ENTRI PADA LEKSIKON

**Name:** Menambah entri pada leksikon

**Actors:** Editor

**Goals:** Editor berhasil menambah entri pada leksikon

**Precondition:** Sistem sudah memuat leksikon ke dalam program

**Steps:**

Actor actions	System responses
1. Editor memilih pilihan untuk menambah entri pada leksikon	2. Sistem menampilkan form untuk menambah entri pada leksikon
3. Editor mengisikan entri baru pada form	5. Sistem mengeluarkan keterangan "Entri berhasil dimasukkan"
4. Editor menekan tombol "OK"	

#### Alternate flow:

Actor actions	System responses
	5a. Sistem mengeluarkan keterangan "Format pengisian entri salah, ulangi lagi"

### MENGUBAH ENTRI PADA LEKSIKON

**Name:** Mengubah entri pada leksikon

**Actors:** Editor

**Goals:** Editor berhasil mengubah entri pada leksikon

**Precondition:** Sistem sudah memuat leksikon ke dalam program

**Steps:**

Actor actions	System responses
1. Editor memilih entri leksikon yang akan diubah	
2. Editor memilih pilihan untuk mengubah entri pada leksikon	3. Sistem menampilkan form untuk mengubah entri pada leksikon
4. Editor melakukan perubahan entri pada form	
5. Editor menekan tombol "OK"	6. Sistem mengeluarkan keterangan "Entri berhasil diubah"

#### Alternate flow:

Actor actions	System responses
	6a. Sistem mengeluarkan keterangan "Format pengisian entri salah, ulangi lagi"

### MENGHAPUS ENTRI PADA LEKSIKON

**Name:** Menghapus entri pada leksikon

**Actors:** Editor

**Goals:** Editor berhasil menghapus entri pada leksikon

**Precondition:** Sistem sudah memuat leksikon ke dalam program

**Steps:**

Actor actions	System responses
1. Editor memilih entri leksikon yang akan dihapus 2. Editor memilih pilihan untuk menghapus entri pada leksikon 4. Editor menekan tombol "OK"	3. Sistem menampilkan kotak dialog persetujuan menghapus entri 5. Sistem mengeluarkan keterangan "Entri berhasil dihapus"

### 3.5 Diagram Kelas

Berdasarkan analisis yang telah dilakukan, kelas-kelas yang akan dibuat untuk perangkat lunak morphological parser adalah sebagai berikut.

**Kelas Parser** berfungsi untuk melakukan proses parsing terhadap sebuah kalimat atau paragraf dalam bahasa Indonesia.

Atribut yang terdapat dalam kelas ini adalah:

- *parseResult*: bertipe String dan menyimpan hasil parsing dari kalimat atau paragraf yang menjadi masukan
- *lexicon*: bertipe objek dari kelas Lexicon dan merupakan objek yang digunakan oleh kelas Parser untuk mengakses fitur leksikon dari perangkat lunak

Method yang terdapat dalam kelas ini adalah:

- *Parser*: konstruktor tanpa parameter untuk membuat objek dari kelas Parser.
- *isRootWord*: method dengan sebuah parameter bertipe String dan kembalian bertipe boolean untuk menentukan apakah kata yang ada di parameter merupakan kata dasar atau bukan. Method ini memanggil method *searchInTree* dari kelas Lexicon.
- *processFromText*: method dengan sebuah parameter bertipe String dan kembalian bertipe String untuk melakukan proses parsing terhadap teks yang ada di parameter.
- *processFromFile*: method dengan sebuah parameter bertipe String dan kembalian bertipe String untuk melakukan proses parsing terhadap isi file dari path yang ada di parameter.
- *checkPrefiks*: method dengan sebuah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak prefiks dalam String kata yang diberikan di parameter.
- *checkSufiks*: method dengan sebuah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak sufiks dalam String kata yang diberikan di parameter.
- *checkRedup*: method dengan sebuah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan reduplikasi pada String kata yang diberikan di parameter.



- 1 • *checkKonfiks*: method tanpa parameter dan kembalian bertipe void untuk melakukan pengecekan adanya kemungkinan kombinasi prefiks dan sufiks yang membentuk konfiks pada atribut hasil parsing.
- 2
- 3
- 4 • *checkKomposisi*: method dengan sebuah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan komposisi pada String kata yang diberikan di parameter.
- 5
- 6 • *checkKomposisi*: method dengan sebuah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan kemungkinan komposisi antara kata yang sedang diproses dengan String kata yang diberikan di parameter.
- 7
- 8

9 **Kelas Lexicon** berfungsi untuk menyimpan kumpulan kata dasar dan kata turunan yang digunakan selama proses morphological parsing berlangsung.

10 Atribut yang terdapat dalam kelas ini adalah:

- 12 • *roots*: bertipe array of Node dan menyimpan kumpulan akar dari pohon node yang menyimpan kata dasar yang valid dalam bahasa Indonesia
- 13
- 14 • *components*: bertipe array of String dan menyimpan kumpulan kata turunan untuk setiap kata dasar dalam bahasa Indonesia
- 15

16 Method yang terdapat dalam kelas ini adalah:

- 17 • *Lexicon*: konstruktor tanpa parameter untuk membuat objek dari kelas Lexicon.
- 18 • *insertRoot*: method dengan sebuah parameter bertipe String dan kembalian bertipe void untuk memasukkan sebuah kata dasar baru ke dalam leksikon.
- 19
- 20 • *updateRoot*: method dengan dua buah parameter bertipe String dan kembalian bertipe void untuk mengubah kata dasar lama dalam parameter menjadi kata dasar baru dalam parameter.
- 21
- 22 • *deleteRoot*: method dengan sebuah parameter bertipe String dan kembalian bertipe void untuk menghapus sebuah kata dasar dalam parameter.
- 23
- 24 • *insertComponent*: method dengan dua buah parameter bertipe String dan kembalian bertipe void untuk memasukkan sebuah kata turunan baru dalam parameter ke kata dasar dalam parameter.
- 25
- 26
- 27 • *updateComponent*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk mengubah kata turunan lama dalam parameter menjadi kata turunan baru dalam parameter untuk kata dasar dalam parameter.
- 28
- 29
- 30 • *deleteComponent*: method dengan dua buah parameter bertipe String dan kembalian bertipe void untuk menghapus sebuah kata turunan dalam parameter dari kata dasar dalam parameter.
- 31
- 32 • *searchInTree*: method dengan sebuah parameter bertipe String dan kembalian bertipe boolean untuk mencari kata dasar di parameter dalam pohon Node.
- 33
- 34 • *printAllWordInTree*: method tanpa parameter dan kembalian bertipe String untuk mencetak semua kata yang disimpan dalam pohon Node ke dalam sebuah String.
- 35

**Kelas Node** berfungsi untuk menyimpan satu karakter dalam pohon Node.

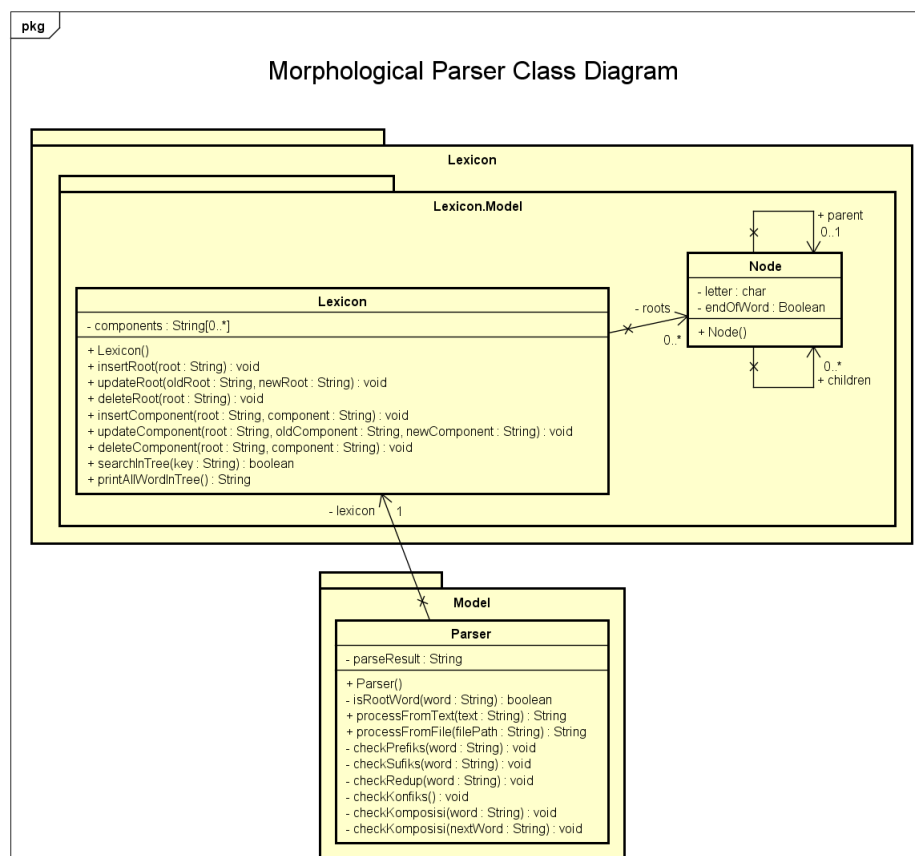
Atribut yang terdapat dalam kelas ini adalah:

- *letter*: bertipe char dan menyimpan sebuah karakter
- *endOfWord*: bertipe boolean dan menyimpan keterangan apakah node ini merupakan karakter akhir dari sebuah kata dalam pohon atau tidak
- *children*: bertipe array of Node dan menyimpan kumpulan node yang merupakan anak dari node ini
- *parent*: bertipe node dan menyimpan sebuah node yang merupakan parent dari node ini

Method yang terdapat dalam kelas ini adalah:

- *Node*: konstruktor tanpa parameter untuk membuat objek dari kelas Node.

Gambar 3.3 berikut adalah diagram kelas awal yang dibuat untuk perangkat lunak ini.



Gambar 3.3: Diagram kelas awal perangkat lunak morphological parser

## BAB 4

### PERANCANGAN

Pada bab ini dijelaskan mengenai beberapa perancangan yang dilakukan dalam penelitian ini, yaitu perancangan struktur penyimpanan leksikon, syntax keluaran proses morphological parsing, antarmuka perangkat lunak, diagram kelas lengkap dari perangkat lunak, dan yang terakhir adalah diagram aktivitas dari perangkat lunak yang akan dibangun.

#### 4.1 Struktur Penyimpanan Leksikon

Leksikon yang dirancang pada perangkat lunak morphological parser ini akan menyimpan kata dasar dan kata turunan yang valid dalam bahasa Indonesia. Kata dasar secara khusus akan dimuat ke dalam program dalam sebuah struktur data trie supaya dapat diakses dengan cepat dan efektif. Sementara kata turunan akan diakses setelah proses parsing selesai untuk melakukan validasi terhadap hasil dari proses parsing. Kata dasar dan kata turunan tersebut harus disimpan dalam file khusus supaya dapat dimuat dan diakses oleh program ketika program dijalankan.

Semua kata dasar disimpan pada sebuah file yang bernama 'roots' berekstensi '.lxc' pada sebuah folder dalam program. Setiap entri kata dasar dipisahkan oleh karakter enter dan disimpan terurut berdasarkan urutan abjad. Untuk menyimpan kata turunan dari setiap kata dasar, dibuat sebuah file khusus dengan nama file sama dengan kata dasar dan berekstensi '.lxc' yang disimpan pada folder yang sama. Isi dari setiap file tersebut adalah kata dasar diikuti oleh semua kemungkinan kata turunan yang dapat dibentuk dari kata dasar yang bersangkutan.

Seperti dibahas pada subbab 2.5, penyimpanan kata turunan tidak bisa dilakukan dengan menulis semua bentuk turunan secara langsung karena akan sangat tidak efisien, terutama untuk bentuk turunan dari afiks yang sangat produktif seperti prefiks *ber-* dan prefiks *me-*. Perlu struktur khusus untuk menyimpan semua kata turunan dengan efisien dalam setiap file kata dasar. Oleh karena itu, dirancang beberapa lambang leksikon seperti dapat dilihat pada tabel 4.1 berikut.

Bentuk	Lambang leksikon
Komposisi	@
Reduplikasi	^
Prefiks	[
Sufiks	]
Konfiks	# ..-

Tabel 4.1: Tabel lambang bentuk turunan dalam leksikon

Untuk menyimpan kata turunan 'sayur bening' yang merupakan hasil proses komposisi dari

1 kata 'sayur' digabung dengan kata 'bening', kita bisa menambahkan bentuk '@bening' dalam file  
 2 'sayur.lxc'. Untuk menyimpan kata 'sayur-mayur' yang merupakan hasil reduplikasi berubah bunyi  
 3 dari kata 'sayur', kita bisa menuliskan bentuk '^mayur'. Sementara untuk bentuk turunan hasil dari  
 4 proses prefiksasi dan sufiksasi seperti kata 'menyayur' dan 'sayuran', dapat disimpan dalam bentuk  
 5 '[me' dan ']an'. Untuk kata yang merupakan hasil dari proses konfiksasi seperti kata 'kesatuan', kita  
 6 bisa menyimpannya dalam file 'satu.lxc' dengan bentuk '# ke-an'.

7 Suatu kata turunan dapat dibentuk dari lebih dari satu proses morfologi, misalnya kata 'sayur-  
 8 sayuran' merupakan kata dasar 'sayur' yang dilakukan reduplikasi utuh menjadi 'sayur-sayur' lalu  
 9 dibubuhkan sufiks *-an* menjadi bentuk 'sayur-sayuran'. Untuk menyimpan bentuk tersebut, setiap  
 10 proses dapat dipisahkan dengan simbol '+' dengan proses yang lebih dulu dikerjakan ditulis lebih  
 11 dahulu. Untuk menyimpan bentuk reduplikasi utuh, kita bisa menyimpannya dengan bentuk '^2'.  
 12 Kata turunan 'sayur-sayuran' disimpan dalam file 'sayur.lxc' dengan bentuk '^2+]an'. Untuk kasus  
 13 kata yang merupakan hasil dari proses reduplikasi dan afiksasi, penulisan bentuk turunan selalu  
 14 proses reduplikasi ditulis lebih dahulu baru diikuti oleh proses afiksasi.

15 Gambar 4.1 berikut adalah contoh isi dari file 'sayur.lxc' yang berisi semua kata turunan dari  
 16 kata dasar 'sayur'.

```

1 sayur
2 @asam
3 @bening
4 ^mayur
5 [me
6 ]an
7 ^2+]an
8
```

Gambar 4.1: Isi dari file sayur.lxc

17 Pada kasus di mana terdapat lebih dari satu prefiks seperti pada kata 'memperkuat', maka  
 18 prefiks yang lebih dulu dibubuhkan pada kata dasar ditulis terlebih dahulu. Kata 'memperkuat'  
 19 disimpan dalam file 'kuat.lxc' dengan bentuk '[per+[me'. Sementara untuk kasus pada proses  
 20 klofiksasi, di mana ada prefiks dan sufiks yang diimbuhkan tetapi pengimbuhanannya tidak sekaligus,  
 21 urutan penulisannya adalah sufiks ditulis lebih dahulu dari prefiks. Contohnya pada kata 'berlarian'  
 22 disimpan dalam file 'lari.lxc' dalam bentuk ']an+[ber'.

23 Pada kasus kata yang merupakan hasil dari proses komposisi dan prefiksasi, seperti pada kata  
 24 'bekerja bakti', proses yang lebih dulu ditulis adalah proses yang melekat pada kata dasarnya  
 25 yaitu proses prefiksasi *ber-* pada kata 'kerja'. Kata 'bekerja bakti' disimpan dalam file 'kerja.lxc'  
 26 dengan bentuk '[ber+@bakti'. Sementara pada contoh kasus kata yang merupakan hasil dari proses  
 27 komposisi dan konfiksasi, seperti pada kata 'pertanggungjawaban', proses yang lebih dulu ditulis  
 28 adalah proses komposisi kata 'tanggung' dengan kata 'jawab'. Kata 'pertanggungjawaban' disimpan  
 29 dalam file 'tanggung.lxc' dengan bentuk '@jawab+# per-an'. Hal ini berlaku juga untuk kata yang  
 30 merupakan hasil dari proses komposisi dan klofiksasi, seperti pada kata 'menanggungjawab' yang  
 31 disimpan dengan bentuk '@jawab+]i+[me'.

## 4.2 Syntax Keluaran Proses Morphological Parsing

Pada subbab 2.2 disebutkan bahwa dalam konvensi linguistik sebuah bentuk dinyatakan sebagai morfem ditulis dalam kurung kurawal ( $\{\dots\}$ ). Proses morphological parsing merupakan proses memisahkan sebuah kata menjadi morfem-morfem penyusunnya. Oleh karena itu, keluaran dari proses morphological parsing sebaiknya mengikuti konvensi linguistik di mana setiap morfem penyusun kata ditulis dalam kurung kurawal ( $\{\dots\}$ ).

Morfem penyusun kata dalam proses morfologi dapat terdiri dari beberapa jenis, yaitu morfem dasar dan morfem afiks dalam proses afiksasi, morfem dasar dengan dirinya sendiri dalam proses reduplikasi, dan morfem dasar dengan morfem dasar lain dalam proses komposisi. Morfem afiks dibagi menjadi tiga jenis, yaitu prefiks, sufiks, dan konfiks.

Proses morphological parsing yang dirancang pada penelitian ini akan menghasilkan keluaran berupa bentuk dasar diikuti oleh proses morfologi yang dilakukan kepada bentuk dasar tersebut. Sebagai contoh, jika masukan adalah kata 'pertanggungjawaban' maka keluaran dari proses morphological parsing terhadap kata tersebut adalah bentuk dasar {tanggung} diikuti proses komposisi {jawab} lalu diikuti proses konfiksasi {per-an}. Untuk menyederhanakan keluaran, kata 'proses' tidak ditulis, kata 'diikuti' diganti dengan simbol '+', dan proses seperti 'konfiksasi' hanya ditulis 'konfiks' saja, sehingga keluaran dari proses tersebut menjadi bentuk dasar {tanggung} + komposisi {jawab} + konfiks {per-an}. Untuk kata yang merupakan hasil reduplikasi, seperti kata 'buah-buahan', hasil proses parsing terhadap kata tersebut adalah bentuk dasar {buah} + reduplikasi {2} + sufiks {an}. Bentuk '2' dalam reduplikasi berarti reduplikasi utuh.

Seperti dijelaskan pada subbab 3.1, leksikon menyimpan setiap kata turunan yang valid dari sebuah kata dasar supaya perangkat lunak dapat melakukan validasi terhadap hasil dari proses parsing. Oleh karena itu, keluaran dari proses parsing harus menggunakan simbol yang sama dengan leksikon supaya hasil dari proses parsing dapat dibandingkan dengan kata turunan yang disimpan dalam leksikon untuk melakukan validasi. Dengan menggunakan simbol pada tabel 4.1, keluaran dari proses parsing terhadap kata 'pertanggungjawaban' adalah bentuk dasar 'tanggung' ditambah proses morfologi '@jawab+# per-an'. Dengan demikian, perangkat lunak dapat melakukan validasi terhadap bentuk '@jawab+# per-an' apakah merupakan bentuk turunan yang valid atau tidak dari kata 'tanggung' dalam leksikon.

Sesuai analisis yang dilakukan pada subbab 3.1, kata turunan yang merupakan hasil proses afiksasi berupa pengimbuhan klitika tidak disimpan dalam leksikon. Oleh karena itu, kata tersebut harus dapat dikenali dan diproses dalam perangkat lunak tanpa melalui proses validasi dalam leksikon. Untuk melakukan proses tersebut diperlukan simbol khusus untuk menandai dan menyimpan klitika dalam hasil proses parsing. Untuk menandai proklitika, yaitu klitika yang berposisi di muka kata yang diikuti, digunakan simbol '\$'. Sementara untuk menandai enklitika, yaitu klitika yang berposisi di belakang kata yang dilekati, digunakan simbol '%'.

Untuk hasil parsing yang merupakan bentuk asing, seperti dijelaskan pada subbab 3.2, penulisan hasil parsing dalam simbol adalah dengan simbol '!' diikuti kata yang sedang diproses. Contohnya, hasil parsing dari kata 'netizen' adalah '!netizen' atau setelah dilakukan konversi ke dalam kata-kata yang dimengerti oleh manusia menjadi bentuk asing {netizen}.

Berdasarkan penjelasan di atas, dapat disimpulkan ada dua jenis keluaran dari proses morphological parsing, yaitu keluaran dalam bentuk simbol seperti dalam leksikon dan keluaran dalam

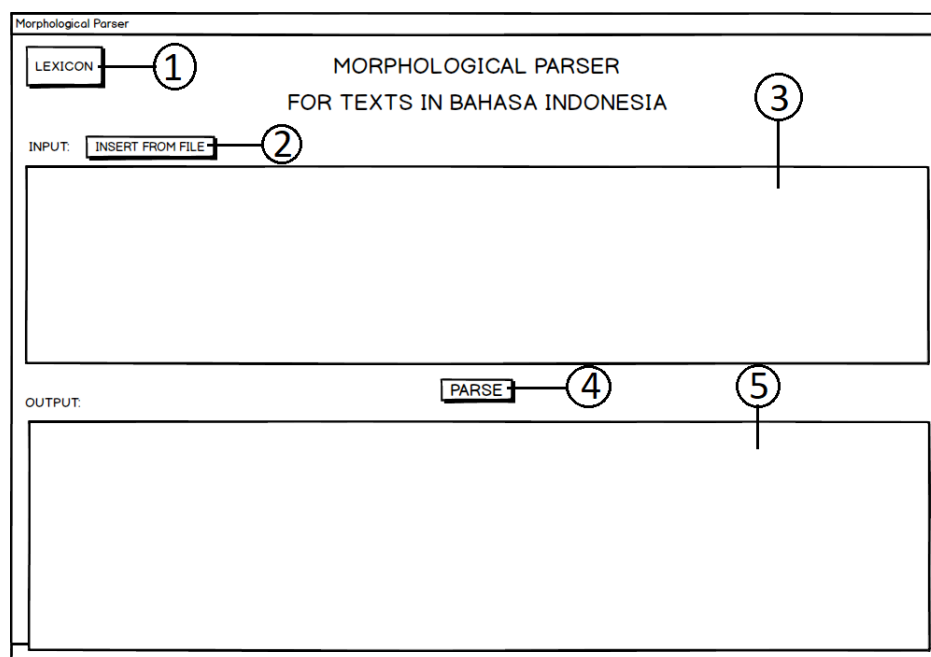
1 bentuk kata-kata yang dapat dimengerti oleh manusia. Perangkat lunak pertama kali membuat  
 2 keluaran dalam bentuk simbol yang kemudian divalidasi oleh leksikon. Keluaran yang dianggap  
 3 tidak valid akan dibuang oleh perangkat lunak sementara keluaran yang dianggap valid kemudian  
 4 diterjemahkan menjadi keluaran berupa kata-kata yang dimengerti oleh manusia.

### 5 4.3 Perancangan Antarmuka

6 Antarmuka yang akan dibuat terdiri dari dua jenis, yaitu untuk perangkat lunak morphological  
 7 parser dan perangkat lunak lexicon. Sesuai use case yang diuraikan pada subbab 3.4, perangkat  
 8 lunak lexicon memiliki dua jenis user, yaitu user biasa dan editor. Antarmuka untuk perangkat  
 9 lunak morphological parser terdiri dari sebuah *frame* sementara untuk perangkat lunak lexicon  
 10 terdiri dari enam buah *frame* yang masing-masing mewakili sebuah fitur untuk sebuah user dari  
 11 perangkat lunak. Berikut adalah penjelasan untuk setiap rancangan antarmuka yang dibuat.

#### 12 4.3.1 Perancangan Antarmuka Perangkat Lunak Morphological Parser

13 Gambar 4.2 berikut adalah rancangan antarmuka yang dibuat untuk perangkat lunak morphological  
 14 parser.



Gambar 4.2: Rancangan antarmuka perangkat lunak morphological parser

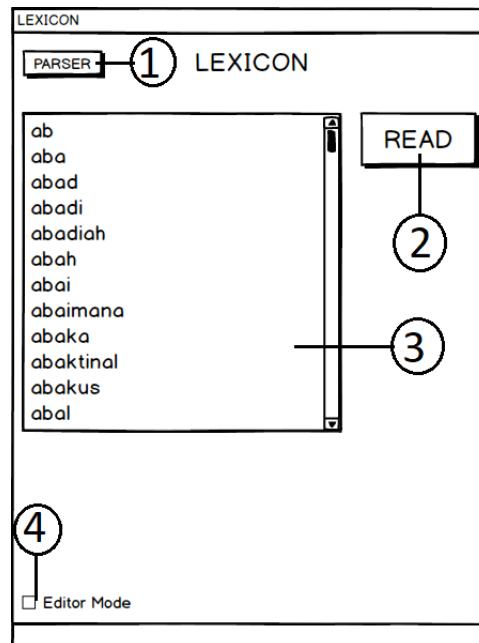
15 Penjelasan untuk setiap objek dalam *frame* di atas adalah sebagai berikut.

- 16 1. Tombol *Lexicon*: untuk mengakses perangkat lunak lexicon
- 17 2. Tombol *Insert From File*: untuk memuat isi dari sebuah file txt ke dalam kolom masukan
- 18 dari perangkat lunak
- 19 3. Kolom masukan: untuk menulis masukan dari proses parsing

4. Tombol *Parse*: untuk melakukan proses parsing terhadap teks dalam kolom masukan dan mengeluarkan hasilnya pada kolom keluaran
5. Kolom keluaran: untuk menampilkan keluaran dari proses parsing

#### 4.3.2 Perancangan Antarmuka Perangkat Lunak Lexicon

Gambar 4.3 berikut adalah rancangan antarmuka yang dibuat untuk perangkat lunak lexicon pada halaman home untuk user.

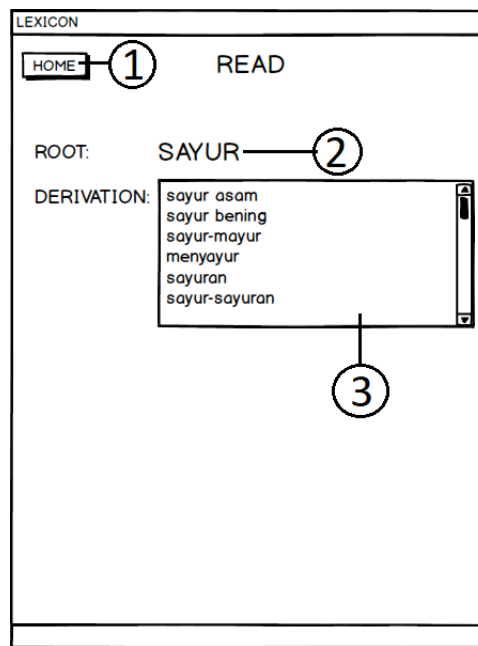


Gambar 4.3: Rancangan antarmuka perangkat lunak lexicon halaman home untuk user

Penjelasan untuk setiap objek dalam *frame* di atas adalah sebagai berikut.

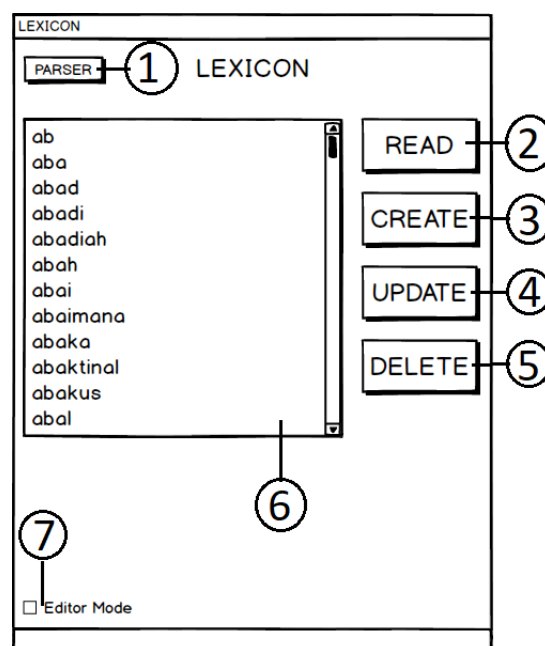
1. Tombol *Parser*: untuk mengakses perangkat lunak morphological parser
2. Tombol *Read*: untuk melihat kata turunan dari sebuah entri kata dasar pada leksikon
3. Kolom entri leksikon: untuk menampilkan semua entri kata dasar yang disimpan dalam leksikon
4. Tombol *Editor mode*: untuk mengaktifkan atau mematikan mode editor pada perangkat lunak lexicon

Gambar 4.4 berikut adalah rancangan antarmuka yang dibuat untuk perangkat lunak lexicon pada halaman read untuk user.



Gambar 4.4: Rancangan antarmuka perangkat lunak lexicon halaman read untuk user

- 1 Penjelasan untuk setiap objek dalam *frame* di atas adalah sebagai berikut.
- 2 1. Tombol *Home*: untuk kembali ke halaman home dari perangkat lunak lexicon
- 3 2. Label root: untuk menampilkan kata dasar yang sedang dilihat saat ini
- 4 3. Kolom entri kata turunan: untuk menampilkan semua entri kata turunan dari kata dasar
- 5 yang disimpan dalam leksikon
- 6 Gambar 4.5 berikut adalah rancangan antarmuka yang dibuat untuk perangkat lunak lexicon
- 7 pada halaman home untuk editor.



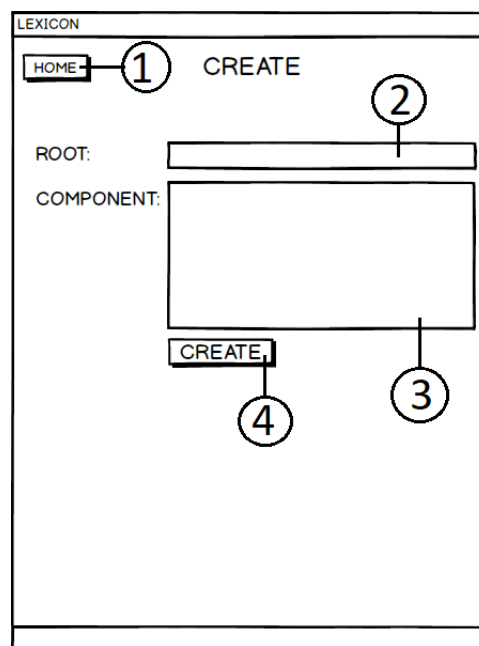
Gambar 4.5: Rancangan antarmuka perangkat lunak lexicon halaman home untuk editor



Penjelasan untuk setiap objek dalam *frame* di atas adalah sebagai berikut.

1. Tombol *Parser*: untuk mengakses perangkat lunak morphological parser
2. Tombol *Read*: untuk melihat kata turunan dari sebuah entri kata dasar pada leksikon
3. Tombol *Create*: untuk memasukkan entri baru pada leksikon
4. Tombol *Update*: untuk mengubah entri pada leksikon
5. Tombol *Delete*: untuk menghapus entri pada leksikon
6. Kolom entri leksikon: untuk menampilkan semua entri kata dasar yang disimpan dalam leksikon
7. Tombol *Editor mode*: untuk mengaktifkan atau mematikan mode editor pada perangkat lunak lexicon

Gambar 4.6 berikut adalah rancangan antarmuka yang dibuat untuk perangkat lunak lexicon pada halaman create untuk editor.

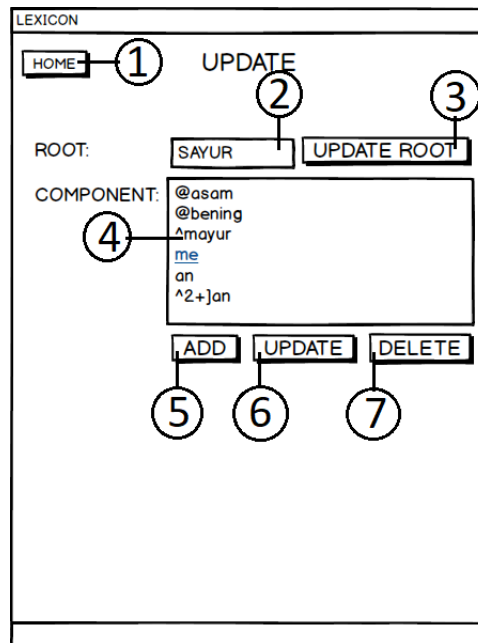


Gambar 4.6: Rancangan antarmuka perangkat lunak lexicon halaman create untuk editor

Penjelasan untuk setiap objek dalam *frame* di atas adalah sebagai berikut.

1. Tombol *Home*: untuk kembali ke halaman home dari perangkat lunak lexicon
2. Kolom kata dasar: untuk memasukkan kata dasar dari entri yang akan dibuat
3. Kolom kata turunan: untuk memasukkan kata turunan dari entri yang akan dibuat
4. Tombol *Create*: untuk memasukkan entri baru yang sudah dibuat ke dalam leksikon

- 1 Gambar 4.7 berikut adalah rancangan antarmuka yang dibuat untuk perangkat lunak lexicon  
 2 pada halaman update untuk editor.

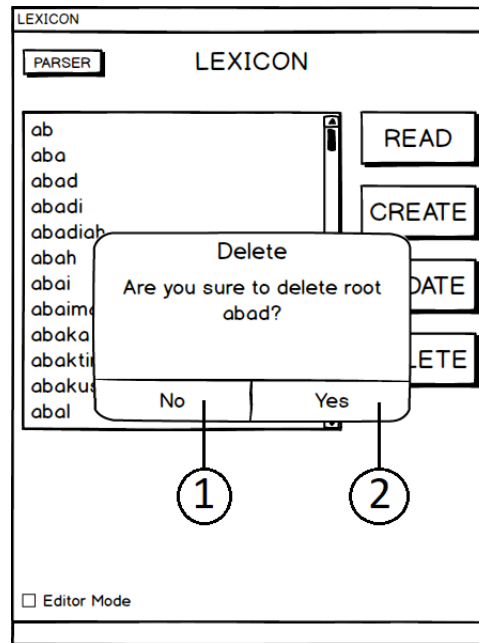


Gambar 4.7: Rancangan antarmuka perangkat lunak lexicon halaman update untuk editor

- 3 Penjelasan untuk setiap objek dalam *frame* di atas adalah sebagai berikut.

- 4 1. Tombol *Home*: untuk kembali ke halaman home dari perangkat lunak lexicon  
 5 2. Kolom kata dasar: untuk mengubah kata dasar dari entri  
 6 3. Tombol *Update Root*: untuk memasukkan entri kata dasar baru ke dalam leksikon  
 7 4. Kolom kata turunan: untuk mengubah kata turunan dari entri  
 8 5. Tombol *Add*: untuk menambahkan entri kata turunan baru pada kata dasar  
 9 6. Tombol *Update*: untuk mengubah entri kata turunan pada kata dasar  
 10 7. Tombol *Delete*: untuk menghapus entri kata turunan pada kata dasar

- 11 Gambar 4.8 berikut adalah rancangan antarmuka yang dibuat untuk perangkat lunak lexicon  
 12 pada halaman delete untuk editor.

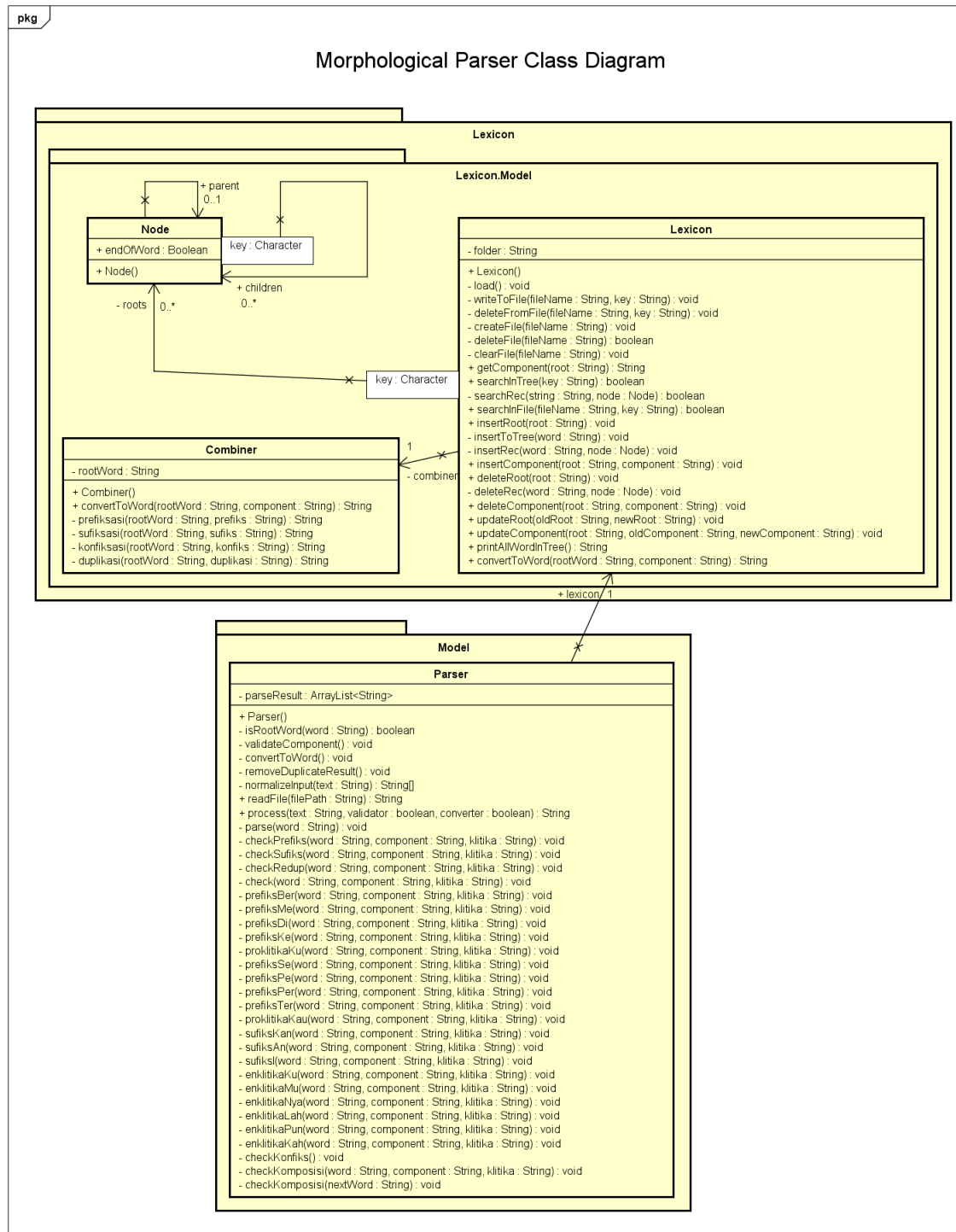


Gambar 4.8: Rancangan antarmuka perangkat lunak lexicon halaman delete untuk editor

- 1 Penjelasan untuk setiap objek dalam *frame* di atas adalah sebagai berikut.
- 2 1. Tombol *No*: untuk membatalkan penghapusan entri kata dasar dari leksikon
- 3 2. Tombol *Yes*: untuk melakukan konfirmasi penghapusan entri kata dasar dari leksikon

#### 4.4 Diagram Kelas Lengkap

- 5 Pada subbab 3.5 telah diuraikan analisis mengenai kelas-kelas yang akan dibuat untuk perangkat
- 6 lunak morphological parser. Pada subbab ini akan ditambahkan beberapa kelas dan method untuk
- 7 melengkapi kelas-kelas yang sudah diuraikan sebelumnya. Gambar 4.9 adalah diagram kelas lengkap
- 8 yang dibuat untuk perangkat lunak morphological parser.



Gambar 4.9: Diagram kelas lengkap perangkat lunak morphological parser

#### 1 4.4.1 Kelas Parser

2 Kelas ini berfungsi untuk melakukan proses parsing terhadap sebuah kalimat atau paragraf dalam  
3 bahasa Indonesia.

4 Atribut yang terdapat dalam kelas ini adalah:

- 5 • *parseResult*: bertipe ArrayList of String untuk menyimpan beberapa kemungkinan hasil

parsing dari setiap kata dalam kalimat atau paragraf yang menjadi masukan

- *lexicon*: bertipe objek dari kelas *Lexicon* dan merupakan objek yang digunakan oleh kelas *Parser* untuk mengakses fitur leksikon dari perangkat lunak

Method yang terdapat dalam kelas ini adalah:

- *Parser*: konstruktor tanpa parameter untuk membuat objek dari kelas *Parser*.
- *isRootWord*: method dengan sebuah parameter bertipe *String* dan kembalian bertipe boolean untuk menentukan apakah kata yang ada di parameter merupakan kata dasar atau bukan. Method ini memanggil method *searchInTree* dari kelas *Lexicon*.
- *validateComponent*: method tanpa parameter dan kembalian bertipe void untuk melakukan validasi terhadap hasil parsing dengan melakukan pengecekan kata turunan dalam leksikon.
- *convertToWord*: method tanpa parameter dan kembalian bertipe void untuk melakukan konversi hasil parsing dari simbol leksikon ke kata-kata yang dapat dimengerti oleh manusia.
- *removeDuplicateResult*: method tanpa parameter dan kembalian bertipe void untuk menghapus hasil parsing yang memiliki duplikat pada atribut hasil parsing.
- *normalizeInput*: method dengan sebuah parameter bertipe *String* dan kembalian bertipe array of *String* untuk melakukan normalisasi pada teks pada parameter dengan membuat semua karakter huruf menjadi huruf kecil, membuang karakter yang tidak diperlukan, dan memisahkan teks berdasar karakter spasi. Karakter yang diperlukan adalah karakter huruf kecil (a..z), karakter pemisah (-), dan karakter angka (0..9).
- *readFile*: method dengan sebuah parameter bertipe *String* dan kembalian bertipe *String* untuk isi file dari path yang ada di parameter dan mengembalikan isinya.
- *process*: method dengan sebuah parameter bertipe *String*, dua buah parameter bertipe boolean, dan kembalian bertipe *String* untuk melakukan proses parsing terhadap teks yang ada di parameter. Fitur validator dan converter untuk hasil dari proses parsing dapat dinyalakan dan dimatikan bergantung pada nilai dari parameter validator dan converter.
- *parse*: method dengan sebuah parameter bertipe *String* dan kembalian bertipe void untuk melakukan parsing pada sebuah kata dalam parameter. Hasil parsing disimpan dalam atribut *parseResult*.
- *checkPrefiks*: method dengan tiga buah parameter bertipe *String* dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak prefiks dalam *String* kata yang diberikan di parameter. Parameter *component* dan *klitika* menyimpan *component* dan *klitika* yang sudah ditemukan ketika method ini dipanggil.
- *checkSufiks*: method dengan tiga buah parameter bertipe *String* dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak sufiks dalam *String* kata yang diberikan di parameter. Parameter *component* dan *klitika* menyimpan *component* dan *klitika* yang sudah ditemukan ketika method ini dipanggil.

- 1     • *checkRedup*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void  
2       untuk melakukan pengecekan reduplikasi dalam String kata yang diberikan di parameter.  
3       Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan  
4       ketika method ini dipanggil.
- 5     • *check*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk  
6       melakukan pengecekan terhadap semua kemungkinan proses morfologi dalam String kata yang  
7       diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika  
8       yang sudah ditemukan ketika method ini dipanggil.
- 9     • *prefiksBer*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void  
10       untuk melakukan pengecekan ada atau tidak prefiks ber- dalam String kata yang diberikan di  
11       parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah  
12       ditemukan ketika method ini dipanggil.
- 13    • *prefiksMe*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void  
14       untuk melakukan pengecekan ada atau tidak prefiks me- dalam String kata yang diberikan di  
15       parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah  
16       ditemukan ketika method ini dipanggil.
- 17    • *prefiksDi*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void  
18       untuk melakukan pengecekan ada atau tidak prefiks di- dalam String kata yang diberikan di  
19       parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah  
20       ditemukan ketika method ini dipanggil.
- 21    • *prefiksKe*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void  
22       untuk melakukan pengecekan ada atau tidak prefiks ke- dalam String kata yang diberikan di  
23       parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah  
24       ditemukan ketika method ini dipanggil.
- 25    • *proklitikaKu*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void  
26       untuk melakukan pengecekan ada atau tidak proklitika ku- dalam String kata yang diberikan  
27       di parameter. Parameter component dan klitika menyimpan component dan klitika yang  
28       sudah ditemukan ketika method ini dipanggil.
- 29    • *prefiksSe*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void  
30       untuk melakukan pengecekan ada atau tidak prefiks se- dalam String kata yang diberikan di  
31       parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah  
32       ditemukan ketika method ini dipanggil.
- 33    • *prefiksPe*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void  
34       untuk melakukan pengecekan ada atau tidak prefiks pe- dalam String kata yang diberikan di  
35       parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah  
36       ditemukan ketika method ini dipanggil.
- 37    • *prefiksPer*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void  
38       untuk melakukan pengecekan ada atau tidak prefiks per- dalam String kata yang diberikan di

parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.

- *prefiksTer*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak prefiks ter- dalam String kata yang diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.

- *proklitikaKau*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak proklitika kau- dalam String kata yang diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.

- *sufiksKan*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak sufiks -kan dalam String kata yang diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.

- *sufiksAn*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak sufiks -an dalam String kata yang diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.

- *sufiksI*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak sufiks -i dalam String kata yang diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.

- *enklitikaKu*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak enklitika -ku dalam String kata yang diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.

- *enklitikaMu*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak enklitika -mu dalam String kata yang diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.

- *enklitikaNya*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak enklitika -nya dalam String kata yang diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.

- *enklitikaLah*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak enklitika -lah dalam String kata yang diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.

- *enklitikaPun*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak enklitika -pun dalam String kata yang diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.
- *enklitikaKah*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan ada atau tidak enklitika -kah dalam String kata yang diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.
- *checkKonfiks*: method tanpa parameter dan kembalian bertipe void untuk melakukan pengecekan adanya kemungkinan kombinasi prefiks dan sufiks yang membentuk konfiks pada atribut hasil parsing.
- *checkKomposisi*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan komposisi dalam String kata yang diberikan di parameter. Parameter component dan klitika menyimpan component dan klitika yang sudah ditemukan ketika method ini dipanggil.
- *checkKomposisi*: method dengan sebuah parameter bertipe String dan kembalian bertipe void untuk melakukan pengecekan kemungkinan komposisi antara kata yang sedang diproses dengan String kata yang diberikan di parameter.

#### 4.4.2 Kelas Lexicon

Kelas ini berfungsi untuk menyimpan kumpulan kata dasar dan kata turunan yang digunakan selama proses morphological parsing berlangsung. Seperti telah diuraikan pada subbab 4.1, kata dasar disimpan dalam sebuah file bernama 'roots.lxc' dan kata turunan untuk setiap kata dasar disimpan dalam file bernama sama dengan kata dasar yang bersangkutan. File 'roots.lxc' akan dimuat oleh kelas ini setiap kali program dijalankan dan kata dasar akan disimpan dalam trie yang berbentuk pohon node supaya pencarian kata dasar dapat dilakukan dengan cepat dan efektif.

Atribut yang terdapat dalam kelas ini adalah:

- *roots*: bertipe map of Node dengan key adalah sebuah karakter dan menyimpan kumpulan akar dari pohon node yang menyimpan kata dasar yang valid dalam bahasa Indonesia
- *folder*: bertipe String dan menyimpan path dari folder tempat file leksikon berada

Method yang terdapat dalam kelas ini adalah:

- *Lexicon*: konstruktor tanpa parameter untuk membuat objek dari kelas Lexicon.
- *load*: method tanpa parameter dan kembalian bertipe void untuk memuat semua kata dasar dalam leksikon dan menyimpannya ke dalam trie.
- *writeToFile*: method dengan dua buah parameter bertipe String dan kembalian bertipe void untuk menulis parameter key ke dalam file dengan nama file dalam parameter fileName.



- 1     • *deleteFromFile*: method dengan dua buah parameter bertipe String dan kembalian bertipe  
2       void untuk menghapus parameter key dari dalam file dengan nama file dalam parameter  
3       fileName.
- 4     • *createFile*: method dengan sebuah parameter bertipe String dan kembalian bertipe void untuk  
5       membuat sebuah file baru dengan nama file dalam parameter fileName.
- 6     • *deleteFile*: method dengan sebuah parameter bertipe String dan kembalian bertipe boolean  
7       untuk menghapus sebuah file dengan nama file dalam parameter fileName dan mengembalikan  
8       status keberhasilan penghapusan file.
- 9     • *clearFile*: method dengan sebuah parameter bertipe String dan kembalian bertipe void untuk  
10       mengosongkan isi sebuah file dengan nama file dalam parameter fileName.
- 11    • *getComponent*: method dengan sebuah parameter bertipe String dan kembalian bertipe String  
12       untuk mengembalikan semua komponen kata turunan dari sebuah kata dasar dalam parameter  
13       root.
- 14    • *searchInTree*: method dengan sebuah parameter bertipe String dan kembalian bertipe boolean  
15       untuk mencari ada atau tidak kata dasar pada parameter key dalam pohon Node.
- 16    • *searchRec*: method dengan sebuah parameter bertipe String, sebuah parameter bertipe objek  
17       dari kelas Node, dan kembalian bertipe boolean untuk melakukan pencarian secara rekursif  
18       dalam pohon Node dengan parameter string adalah kata yang dicari dan node adalah node  
19       yang sedang ditelusuri saat ini.
- 20    • *searchInFile*: method dengan dua buah parameter bertipe String dan kembalian bertipe boo-  
21       lean untuk melakukan pencarian dari parameter key dalam file dengan nama pada parameter  
22       fileName.
- 23    • *insertRoot*: method dengan sebuah parameter bertipe String dan kembalian bertipe void  
24       untuk memasukkan kata dasar baru dalam parameter root ke dalam leksikon.
- 25    • *insertToTree*: method dengan sebuah parameter bertipe String dan kembalian bertipe void  
26       untuk memasukkan sebuah kata dalam parameter word ke dalam pohon Node.
- 27    • *insertRec*: method dengan sebuah parameter bertipe String, sebuah parameter bertipe objek  
28       dari kelas Node, dan kembalian bertipe void untuk memasukkan kata secara rekursif dalam  
29       pohon Node dengan parameter string adalah kata yang dimasukkan dan node adalah node  
30       yang sedang ditelusuri saat ini.
- 31    • *insertComponent*: method dengan dua buah parameter bertipe String dan kembalian bertipe  
32       void untuk memasukkan sebuah kata turunan baru dalam parameter component ke kata dasar  
33       dalam parameter root.
- 34    • *deleteRoot*: method dengan sebuah parameter bertipe String dan kembalian bertipe void  
35       untuk menghapus sebuah kata dasar dalam parameter root dari pohon Node.

- *deleteRec*: method dengan sebuah parameter bertipe String, sebuah parameter bertipe objek dari kelas Node, dan kembalian bertipe void untuk menghapus kata secara rekursif dalam pohon Node dengan parameter string adalah kata yang dihapus dan node adalah node yang sedang ditelusuri saat ini.
- *deleteComponent*: method dengan dua buah parameter bertipe String dan kembalian bertipe void untuk menghapus sebuah kata turunan dalam parameter component dari kata dasar dalam parameter root.
- *updateRoot*: method dengan dua buah parameter bertipe String dan kembalian bertipe void untuk mengubah kata dasar lama dalam parameter oldRoot menjadi kata dasar baru dalam parameter newRoot.
- *updateComponent*: method dengan tiga buah parameter bertipe String dan kembalian bertipe void untuk mengubah kata turunan lama dalam parameter oldComponent menjadi kata turunan baru dalam parameter newComponent untuk kata dasar dalam parameter root.
- *printAllWordInTree*: method tanpa parameter dan kembalian bertipe String untuk mencetak semua kata yang disimpan dalam pohon Node ke dalam sebuah String.
- *convertToWord*: method dengan dua buah parameter bertipe String dan kembalian bertipe String untuk mengubah kata dasar dalam parameter rootWord dan komponen dalam parameter component menjadi kata-kata yang dapat dimengerti oleh manusia. Method ini memanggil method *convertToWord* dari kelas *Combiner*.

#### 4.4.3 Kelas Node

Kelas ini berfungsi untuk merepresentasikan satu node dalam pohon Node.

Atribut yang terdapat dalam kelas ini adalah:

- *endOfWord*: bertipe boolean dan menyimpan keterangan apakah node ini merupakan karakter akhir dari sebuah kata dalam pohon atau tidak
- *children*: bertipe map of Node dengan key adalah sebuah karakter dan menyimpan kumpulan node yang merupakan anak dari node ini
- *parent*: bertipe node dan menyimpan sebuah node yang merupakan parent dari node ini

Method yang terdapat dalam kelas ini adalah:

- *Node*: konstruktor tanpa parameter untuk membuat objek dari kelas Node.

#### 4.4.4 Kelas Combiner

Kelas ini berfungsi untuk melakukan konversi sebuah kata turunan yang disimpan dalam leksikon dari bentuk dalam simbol leksikon menjadi kata yang dapat dimengerti oleh manusia. Ini berfungsi supaya user dari program dapat melihat kata turunan yang disimpan dalam leksikon dalam bentuk

kata yang dapat dimengerti dan bukan dalam bentuk simbol leksikon. Kelas ini digunakan dalam fitur Read yang dimiliki oleh perangkat lunak Lexicon.

Atribut yang terdapat dalam kelas ini adalah:

- *rootWord*: bertipe String dan menyimpan kata dasar dari kata yang sedang diproses

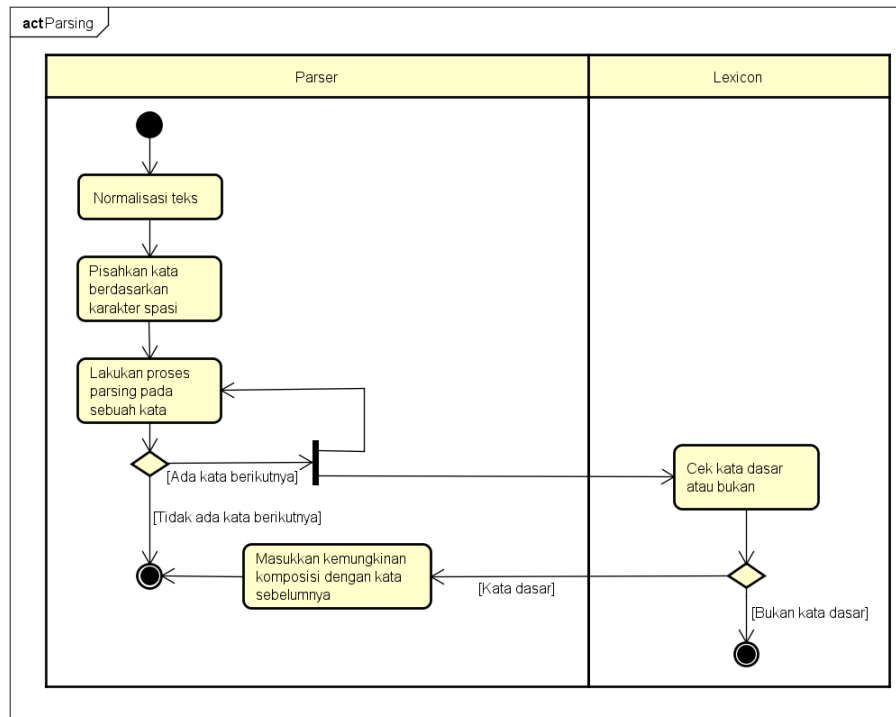
Method yang terdapat dalam kelas ini adalah:

- *Combiner*: konstruktor tanpa parameter untuk membuat objek dari kelas Combiner.
- *convertToWord*: method dengan dua buah parameter bertipe String dan kembalian bertipe String untuk mengubah kata dasar dalam parameter *rootWord* dan komponen dalam parameter *component* menjadi kata-kata yang dapat dimengerti oleh manusia.
- *prefiksasi*: method dengan dua buah parameter bertipe String dan kembalian bertipe String untuk melakukan proses prefiksasi pada kata dasar dalam parameter *rootWord* dan prefiks dalam parameter *prefiks*.
- *sufiksasi*: method dengan dua buah parameter bertipe String dan kembalian bertipe String untuk melakukan proses sufiksasi pada kata dasar dalam parameter *rootWord* dan sufiks dalam parameter *sufiks*.
- *konfiksasi*: method dengan dua buah parameter bertipe String dan kembalian bertipe String untuk melakukan proses konfiksasi pada kata dasar dalam parameter *rootWord* dan konfiks dalam parameter *konfiks*.
- *duplikasi*: method dengan dua buah parameter bertipe String dan kembalian bertipe String untuk melakukan proses reduplikasi pada kata dasar dalam parameter *rootWord* dan jenis reduplikasi dalam parameter *duplikasi*.

## 4.5 Diagram Aktivitas

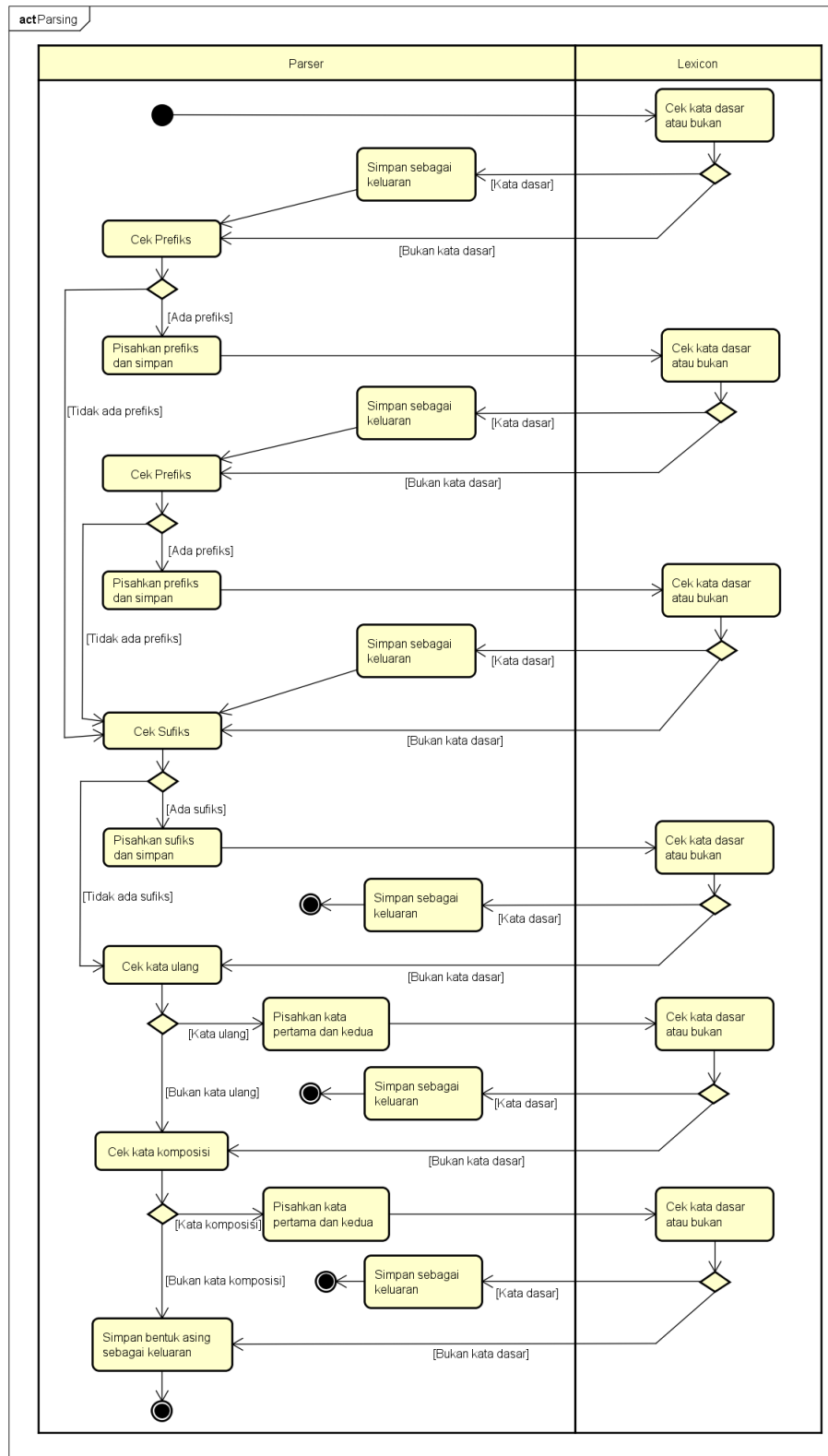
Untuk menjelaskan proses morphological parsing yang akan dikerjakan oleh perangkat lunak, dibuat dua buah diagram aktivitas, yang pertama adalah diagram aktivitas untuk proses mengolah teks masukan dan yang kedua adalah diagram aktivitas untuk proses parsing pada sebuah kata.

Gambar 4.10 berikut adalah diagram aktivitas yang dibuat untuk proses mengolah teks masukan, khususnya untuk yang terdiri dari lebih dari satu kata.



Gambar 4.10: Diagram aktivitas proses mengolah teks masukan

- 1 Gambar 4.11 berikut adalah diagram aktivitas yang dibuat untuk proses parsing pada sebuah
- 2 kata.



Gambar 4.11: Diagram aktivitas proses parsing pada sebuah kata



## BAB 5

### IMPLEMENTASI DAN PENGUJIAN

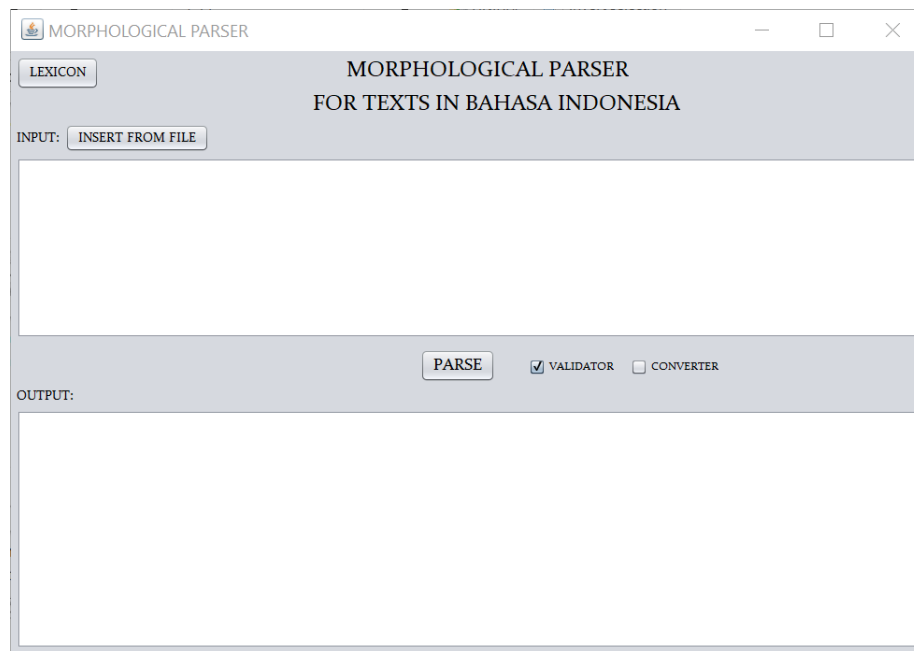
Pada bab ini dijelaskan mengenai implementasi dari seluruh hasil analisis dan perancangan yang telah dilakukan pada bab-bab sebelumnya dan pengujian yang dilakukan untuk hasil implementasi tersebut. Hasil pengujian akan digunakan untuk mengukur performansi dan kualitas dari perangkat lunak yang dibuat.

#### 5.1 Implementasi

Pada subbab 4.4 telah dirancang beberapa kelas yang menjadi bagian dari perangkat lunak morphological parser. Implementasi dari kelas-kelas tersebut menjadi sebuah perangkat lunak akan dilakukan dengan menggunakan bahasa pemrograman Java. Implementasi meliputi keseluruhan method dan atribut untuk setiap kelas yang sudah dirancang. Pada kelas Lexicon dan Node, terdapat atribut yang memiliki tipe map of Node dengan key sebuah karakter, atribut ini diimplementasikan dengan hash table pada kelas HashMap yang dimiliki oleh bahasa Java.

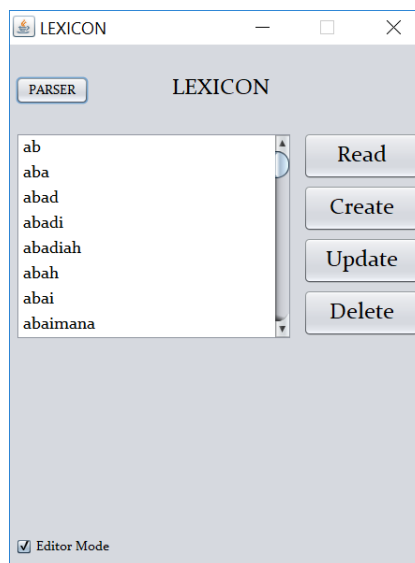
Penyimpanan leksikon, seperti dibahas pada subbab 4.1, dilakukan dalam file khusus berekstensi '.lxc'. Supaya perangkat lunak dapat melakukan proses baca dan tulis dari dan ke file tersebut, diperlukan suatu implementasi dari proses baca tulis file oleh perangkat lunak. Proses ini diimplementasikan dalam bahasa Java dengan kelas BufferedReader dan kelas BufferedWriter yang menggunakan objek Reader dari kelas FileReader dan kelas FileWriter.

Untuk implementasi dari rancangan antarmuka perangkat lunak, dilakukan dengan menggunakan kelas JFrame dalam bahasa Java. Sebagai tambahan dari perancangan yang sudah dilakukan, ditambahkan fitur untuk mengaktifkan dan mematikan fitur validator dan converter dari perangkat lunak. Fitur validator adalah fitur untuk melakukan validasi hasil parsing pada kata turunan dalam leksikon, seperti yang dibahas pada subbab 3.1. Sementara fitur converter adalah fitur untuk melakukan konversi hasil parsing dari simbol leksikon menjadi kata-kata yang dapat dimengerti oleh manusia. Gambar 5.1 berikut adalah implementasi antarmuka untuk perangkat lunak morphological parser.



Gambar 5.1: Implementasi antarmuka perangkat lunak morphological parser

1 Gambar 5.2 berikut adalah implementasi antarmuka untuk perangkat lunak lexicon.



Gambar 5.2: Implementasi antarmuka perangkat lunak lexicon

## 2 5.2 Pengujian

3 Pengujian yang dilakukan pada perangkat lunak yang dibuat dalam penelitian ini dibagi menjadi  
 4 dua bagian. Bagian pertama merupakan pengujian fungsional yang akan menguji kesesuaian  
 5 antara implementasi dari perangkat lunak dengan kebutuhan. Bagian kedua merupakan pengujian  
 6 nonfungsional yang akan menguji kualitas dari perangkat lunak.



### 5.2.1 Pengujian Fungsional

Beberapa aspek yang akan diuji pada pengujian fungsional adalah sebagai berikut.

- hasil normalisasi teks masukan
- hasil proses parsing
- proses create, update, dan delete entri leksikon

Pada tahap pengujian hasil normalisasi teks masukan dan hasil proses parsing, contoh masukan yang digunakan adalah:

- Mengisi kemerdekaan Indonesia tanggal 17 Agustus adalah tanggung jawab setiap warga
- Ayah menyuruh, "Jangan main-main dengan makanan beku!"

Hasil normalisasi dari teks masukan tersebut adalah sebagai berikut.

- mengisi kemerdekaan indonesia tanggal 17 agustus adalah tanggung jawab setiap warga
- ayah menyuruh jangan main-main dengan makanan beku

Hasil parsing dari kedua contoh masukan tersebut dapat dilihat pada gambar 5.3 dan gambar 5.4.

```

1 MENGISI:
2 Bentuk Dasar {isi} + Prefiks {me};
3
4 KEMERDEKAAN:
5 Bentuk Dasar {merdeka} + Konfiks {ke-an};
6
7 INDONESIA:
8 Bentuk Dasar {indonesia};
9
10 TANGGAL:
11 Bentuk Dasar {tanggal};
12
13 17:
14 Bentuk Asing {17};
15
16 AGUSTUS:
17 Bentuk Dasar {agustus};
18
19 ADALAH:
20 Bentuk Dasar {adalah};
21 Bentuk Dasar {ada} + Enklitika {lah};
22
23 TANGGUNG:
24 Bentuk Dasar {tanggung};
25 Bentuk Dasar {tanggung} + Komposisi {jawab};
26
27 JAWAB:
28 Bentuk Dasar {jawab};
29
30 SETIAP:
31 Bentuk Dasar {tiap} + Prefiks {se};
32
33 WARGA:
34 Bentuk Dasar {warga};

```

Gambar 5.3: Hasil parsing contoh masukan pertama

```

1  AYAH:
2  Bentuk Dasar {ayah};
3
4  MENYURUH:
5  Bentuk Dasar {suruh} + Prefiks {me};
6
7  JANGAN:
8  Bentuk Dasar {jangan};
9
10 MAIN-MAIN:
11 Bentuk Dasar {main} + Reduplikasi {2};
12
13 DENGAN:
14 Bentuk Dasar {dengan};
15
16 MAKANAN:
17 Bentuk Dasar {makan} + Sufiks {an};
18 Bentuk Dasar {makan} + Sufiks {an} + Komposisi {beku};
19
20 BEKU:
21 Bentuk Dasar {beku};

```

Gambar 5.4: Hasil parsing contoh masukan kedua

1 Seperti dijelaskan pada subbab 5.1, perangkat lunak yang dibuat memiliki fitur untuk mengak-  
2 tifkan dan mematikan fitur validator dan converter untuk memproses hasil dari proses parsing. Hasil  
3 parsing pada contoh di atas menggunakan kedua fitur validator dan converter. Hasil dari proses  
4 parsing terhadap masukan yang sama akan berbeda ketika salah satu atau kedua fitur tersebut  
5 dimatikan. Berikut adalah beberapa contoh keluaran dari masukan yang sama dengan salah satu  
6 fitur tersebut dimatikan.

7 Gambar 5.5 berikut adalah hasil parsing dari contoh masukan pertama yang diproses tanpa  
8 fitur converter.

```

1  MENGISI:
2  isi+[me;
3
4  KEMERDEKAAN:
5  merdeka+#ke-an;
6
7  INDONESIA:
8  indonesia;
9
10 TANGGAL:
11 tanggal;
12
13 17:
14 !17;
15
16 AGUSTUS:
17 agustus;
18
19 ADALAH:
20 adalah;
21 ada+%lah;
22
23 TANGGUNG:
24 tanggung;
25 tanggung+@jawab;
26
27 JAWAB:
28 jawab;
29
30 SETIAP:
31 tiap+[se;
32
33 WARGA:
34 warga;

```

Gambar 5.5: Hasil parsing contoh masukan pertama tanpa fitur converter

9 Gambar 5.6 berikut adalah hasil parsing dari contoh masukan kedua yang diproses tanpa fitur  
10 validator.

```

1  AYAH:
2  Bentuk Dasar {ayah};
3
4  MENYURUH:
5  Bentuk Dasar {suruh} + Prefiks {me};
6  Bentuk Dasar {suruh} + Prefiks {me} + Komposisi {jangan};
7
8  JANGAN:
9  Bentuk Dasar {jangan};
10
11 MAIN-MAIN:
12 Bentuk Dasar {main} + Reduplikasi {2};
13 Bentuk Dasar {main} + Reduplikasi {2} + Komposisi {dengan};
14
15 DENGAN:
16 Bentuk Dasar {dengan};
17
18 MAKANAN:
19 Bentuk Dasar {makan} + Sufiks {an};
20 Bentuk Dasar {mak} + Sufiks {an} + Sufiks {an};
21 Bentuk Dasar {makan} + Sufiks {an} + Komposisi {beku};
22 Bentuk Dasar {mak} + Sufiks {an} + Sufiks {an} + Komposisi {beku};
23
24 BEKU:
25 Bentuk Dasar {beku};
26 Bentuk Dasar {ku} + Prefiks {ber};

```

Gambar 5.6: Hasil parsing contoh masukan kedua tanpa fitur validator

Ada beberapa jenis kata yang merupakan kata valid dalam bahasa Indonesia namun diproses secara kurang tepat dalam perangkat lunak ini, yaitu:

- sekuatnya
- tari-menari
- pertama-tama
- ngomong-ngomong

Kata 'sekuatnya' merupakan kata yang dibentuk dari bentuk dasar {kuat} + konfiks {se-nya}. Namun, seperti dibahas pada subbab 3.1, bentuk {nya} dianggap sebagai klitika dan bukan dianggap sebagai sufiks dalam kesatuan konfiks {se-nya}. Oleh karena itu, kata 'sekuatnya' diproses dalam perangkat lunak menjadi bentuk dasar {kuat} + prefiks {se} + enklitika {nya}.

Kata 'tari-menari' merupakan contoh kata yang dibentuk dari proses reduplikasi dari bentuk dasar {tari} lalu dilakukan proses prefiksasi, namun proses prefiksasi dilakukan bukan pada bentuk dasarnya melainkan pada kata ulangnya. Umumnya, proses reduplikasi yang diikuti proses prefiksasi pada bentuk dasar {tari} dan prefiks {me} akan menghasilkan kata 'menari-nari'. Sementara, pada kasus ini yang diharapkan adalah kata 'tari-menari'. Untuk bentuk kata seperti 'tari-menari' ini diproses dalam perangkat lunak seperti memproses kata ulang berubah bunyi seperti 'sayur-mayur', sehingga hasil prosesnya menjadi bentuk dasar {tari} + reduplikasi {menari}.

Kata 'pertama-tama' merupakan contoh kata yang dibentuk dari proses reduplikasi dari bentuk dasar, namun reduplikasi tidak dilakukan secara utuh maupun berubah bunyi melainkan hanya sebagian. Sama halnya dengan bentuk 'tari-menari', bentuk ini diproses dalam perangkat lunak seperti memproses kata ulang berubah bunyi, sehingga hasil prosesnya menjadi bentuk dasar {pertama} + reduplikasi {tama}.

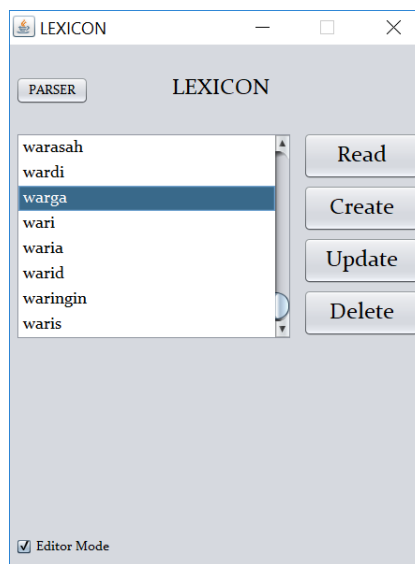
Kata 'ngomong-ngomong' merupakan contoh kata yang dibentuk dari bentuk dasar {omong} melalui proses prefiksasi dan proses reduplikasi, namun prefiks yang digunakan bukan prefiks yang baku. Prefiks yang digunakan pada bentuk ini disebut dengan prefiks nasal, seperti dijelaskan pada

1 subbab 2.2.4. Prefiks nasal ini tidak dapat diproses dalam perangkat lunak sehingga bentuk seperti  
2 kata 'ngomong-ngomong' dianggap sebagai bentuk asing.

3 Pengujian fungsional selanjutnya akan dilakukan pada proses create, update, dan delete entri  
4 leksikon, contoh masukan yang digunakan adalah:

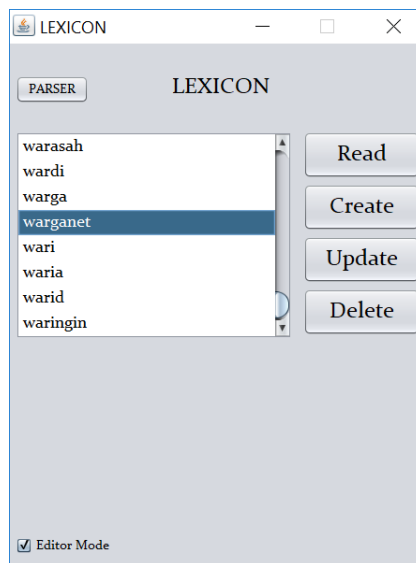
- 5 • Kata dasar 'warganet'
- 6 • Kata turunan 'menyayur-mayur' pada kata dasar 'sayur'
- 7 • Kata dasar 'abau'

8 Proses create akan dilakukan dengan menggunakan kata dasar 'warganet' yang merupakan  
9 serapan dari kata 'netizen' dalam bahasa Inggris. Gambar 5.7 berikut adalah isi dari leksikon  
10 sebelum kata tersebut dimasukkan.



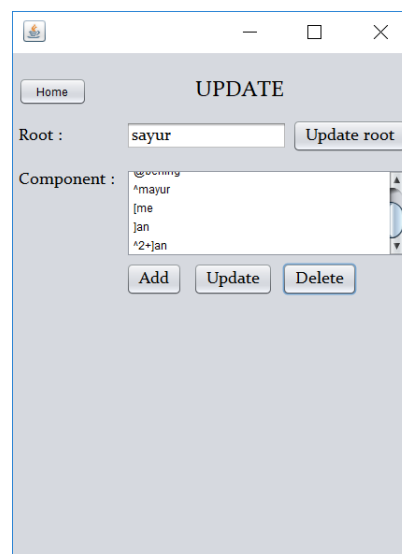
Gambar 5.7: Isi leksikon sebelum kata 'warganet' dimasukkan

11 Gambar 5.8 berikut adalah isi dari leksikon setelah kata 'warganet' dimasukkan.



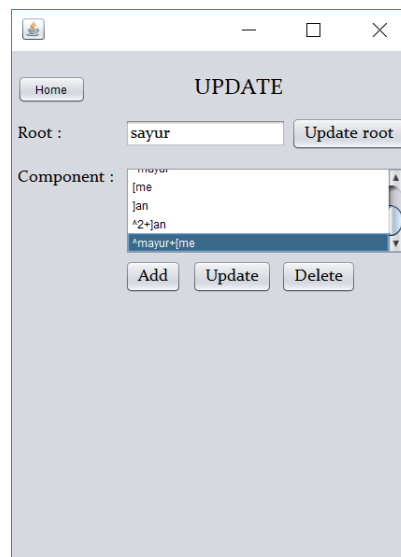
Gambar 5.8: Isi leksikon setelah kata 'warganet' dimasukkan

- 1 Proses update akan dilakukan pada kata dasar 'sayur' dengan menambahkan kata turunan
- 2 'menyayur-mayur'. Gambar 5.9 berikut adalah isi dari leksikon pada kata dasar 'sayur' sebelum
- 3 kata tersebut ditambahkan.



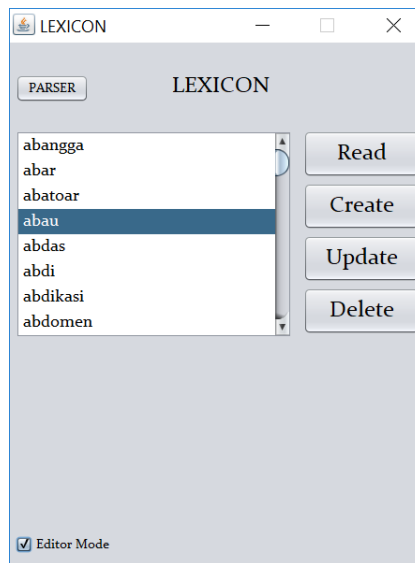
Gambar 5.9: Turunan dari kata dasar 'sayur' sebelum kata turunan 'menyayur-mayur' ditambahkan

- 4 Gambar 5.10 berikut adalah isi dari leksikon pada kata dasar 'sayur' setelah kata turunan
- 5 'menyayur-mayur' ditambahkan.



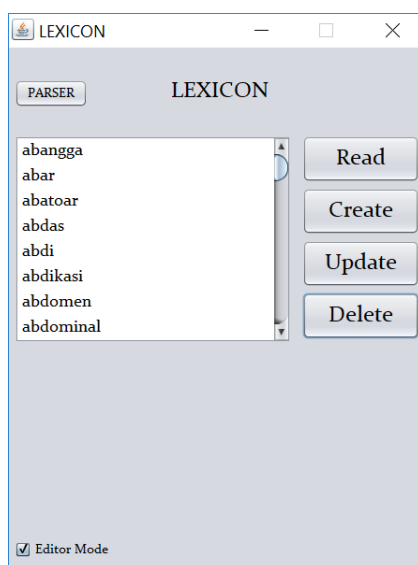
Gambar 5.10: Turunan dari kata dasar 'sayur' setelah kata turunan 'menyayur-mayur' ditambahkan

- 1 Proses delete akan dilakukan pada kata dasar 'abau'. Gambar 5.11 berikut adalah isi dari
- 2 leksikon sebelum kata tersebut dihapus.



Gambar 5.11: Isi leksikon sebelum kata 'abau' dihapus

- 3 Gambar 5.12 berikut adalah isi dari leksikon setelah kata 'abau' dihapus.



Gambar 5.12: Isi leksikon setelah kata 'abau' dihapus

Pengujian fungsional yang dilakukan untuk beberapa aspek di atas berjalan dengan baik dan sesuai dengan yang diharapkan. Perlu dicatat, bahwa untuk menggunakan fitur validator ketika melakukan proses parsing pada perangkat lunak, supaya hasil parsing akurat dan sesuai harapan kata yang diproses harus sudah disimpan dalam leksikon, baik itu kata dasar maupun kata turunan. Pada penelitian ini, belum semua kata dasar dan kata turunan yang valid dalam bahasa Indonesia disimpan dalam leksikon. Hal ini dikarenakan ada terlalu banyak kata dalam bahasa Indonesia, khususnya kata turunan. Perangkat lunak dapat digunakan tanpa fitur validator namun hasil dari proses parsing tidak akan seakurat jika menggunakan fitur validator.

### 5.2.2 Pengujian Nonfungsional

Beberapa aspek yang akan diuji pada pengujian nonfungsional adalah sebagai berikut.

- waktu yang dibutuhkan untuk melakukan proses parsing
- waktu yang dibutuhkan untuk melakukan create, update, dan delete entri leksikon

Pada tahap pengujian waktu proses parsing, contoh masukan yang digunakan adalah:

- File txt berisi 10 kata
- File txt berisi 20 kata
- File txt berisi 100 kata

Isi file txt yang digunakan pada pengujian ini dapat dilihat pada lampiran [A](#).

Untuk setiap contoh masukan, dilakukan proses parsing sebanyak 5 kali dan setiap proses parsing selesai dilakukan akan dicatat waktu prosesnya. Dari data waktu proses parsing tersebut akan diambil rata-rata waktu prosesnya.

Tabel [5.1](#) berikut berisi waktu proses parsing dan rata-ratanya untuk contoh masukan pertama.

Proses ke-	Waktu (dalam milidetik)
1	29
2	3
3	4
4	3
5	3
<b>Rata-rata</b>	<b>8,4</b>

Tabel 5.1: Tabel waktu proses parsing untuk contoh masukan pertama

1      Tabel 5.2 berikut berisi waktu proses parsing dan rata-ratanya untuk contoh masukan kedua.

Proses ke-	Waktu (dalam milidetik)
1	50
2	8
3	8
4	9
5	9
<b>Rata-rata</b>	<b>16,8</b>

Tabel 5.2: Tabel waktu proses parsing untuk contoh masukan kedua

2      Tabel 5.3 berikut berisi waktu proses parsing dan rata-ratanya untuk contoh masukan ketiga.

Proses ke-	Waktu (dalam milidetik)
1	280
2	41
3	30
4	36
5	38
<b>Rata-rata</b>	<b>85</b>

Tabel 5.3: Tabel waktu proses parsing untuk contoh masukan ketiga

3      Dapat dilihat dari hasil pengujian terhadap waktu proses parsing di atas, kenaikan rata-rata  
4 waktu proses untuk contoh masukan pertama, kedua, dan ketiga berbanding lurus dengan jumlah  
5 kata yang diproses. Dapat disimpulkan bahwa waktu untuk melakukan proses parsing bergantung  
6 pada banyaknya kata yang diproses. Semakin banyak kata yang diproses waktu yang dibutuhkan  
7 untuk melakukan proses tersebut akan semakin lama.

8      Dapat dilihat juga pada setiap tabel waktu proses untuk setiap contoh masukan, terjadi  
9 penurunan waktu yang signifikan antara proses pertama ke proses kedua untuk proses pada contoh  
10 masukan yang sama. Dapat disimpulkan bahwa ketika dilakukan beberapa kali proses pada masukan  
11 yang sama maka waktu prosesnya akan turun cukup signifikan. Hal ini dimungkinkan karena bahasa  
12 Java menyimpan data hasil proses yang pernah dijalankan sebelumnya.

13      Pada tahap pengujian waktu untuk melakukan create, update, dan delete entri leksikon, akan  
14 dilakukan dengan melakukan masing-masing proses tersebut sebanyak 3 kali untuk kata yang sama.  
15 Dari data waktu proses tersebut akan diambil rata-rata waktunya.

16      Tabel 5.4 berikut berisi waktu untuk melakukan create sebuah entri baru pada leksikon dan  
17 rata-ratanya.



Proses ke-	Waktu (dalam milidetik)
1	5488
2	5382
3	5267
<b>Rata-rata</b>	<b>5379</b>

Tabel 5.4: Tabel waktu untuk melakukan create sebuah entri baru dalam leksikon

1      Tabel 5.5 berikut berisi waktu untuk melakukan update sebuah entri pada leksikon dan rata-  
 2      ratanya.

Proses ke-	Waktu (dalam milidetik)
1	7
2	3
3	2
<b>Rata-rata</b>	<b>4</b>

Tabel 5.5: Tabel waktu untuk melakukan update sebuah entri dalam leksikon

3      Tabel 5.6 berikut berisi waktu untuk melakukan delete sebuah entri dari leksikon dan rata-  
 4      ratanya.

Proses ke-	Waktu (dalam milidetik)
1	5244
2	5055
3	5083
<b>Rata-rata</b>	<b>5127,33</b>

Tabel 5.6: Tabel waktu untuk melakukan delete sebuah entri dalam leksikon

5      Dapat dilihat dari hasil pengujian terhadap waktu untuk melakukan create, update, dan delete  
 6      entri leksikon di atas, waktu yang dibutuhkan untuk melakukan create dan delete cukup lama, yaitu  
 7      sekitar 5000 milidetik atau 5 detik, sementara waktu yang dibutuhkan untuk melakukan update  
 8      sangat singkat, yaitu sekitar 4 milidetik. Hal ini dikarenakan untuk melakukan create dan delete  
 9      dibutuhkan proses berupa menelusuri pohon node untuk membuat node yang diperlukan ketika  
 10      melakukan create atau untuk menghapus node ketika melakukan delete. Selain itu, juga dibutuhkan  
 11      waktu yang cukup lama untuk mencetak keseluruhan kata yang ada dalam pohon node untuk  
 12      dimasukkan ke dalam file 'roots.lxc' yang menyimpan semua kata dasar. Untuk melakukan update  
 13      dapat diselesaikan dalam waktu singkat karena proses ini hanya berupa menulis kata turunan baru  
 14      ke dalam file kata dasar yang bersangkutan.



## BAB 6

### KESIMPULAN DAN SARAN

Pada bab ini dijelaskan mengenai beberapa kesimpulan yang dapat diambil dari penelitian ini dan diberikan beberapa saran yang dapat dilakukan untuk memperbaiki dan mengembangkan penelitian ini selanjutnya.

#### 6.1 Kesimpulan

Beberapa kesimpulan yang dapat diambil dari penelitian ini adalah sebagai berikut.

1. Aturan morfologi dalam bahasa Indonesia terdiri atas beberapa hal, yaitu:

- Proses morfologi dalam bahasa Indonesia terdiri dari beberapa jenis, yaitu proses pembubuhan afiks dalam proses afiksasi, pengulangan dalam proses reduplikasi, dan penggabungan dalam proses komposisi
- Terdapat beberapa aturan dalam proses morfologi bahasa Indonesia, yaitu morfofonemik yang mengatur perubahan bunyi atau perubahan fonem ketika dilakukan proses morfologi dan morfotaktik yang mengatur suatu morfem boleh digabungkan dengan morfem apa saja

2. Leksikon yang digunakan dalam perangkat lunak memiliki spesifikasi sebagai berikut:

- Leksikon menyimpan seluruh kata dasar beserta kata turunan untuk setiap kata dasar yang valid dalam bahasa Indonesia dengan menggunakan struktur data trie dan diimplementasikan dalam perangkat lunak dengan objek dari kelas HashMap yang dimiliki oleh bahasa pemrograman Java
- Perangkat lunak lexicon yang dibuat dapat menyimpan kata turunan yang merupakan hasil dari proses morfologi berupa afiksasi, reduplikasi, atau komposisi

3. Implementasi dari aturan morfologi bahasa Indonesia ke dalam perangkat lunak adalah sebagai berikut:

- Implementasi dari aturan morfofonemik dilakukan dalam perangkat lunak morphological parser dengan menghasilkan semua kemungkinan hasil parsing berdasarkan perubahan fonem yang valid dari kata masukan yang diproses
- Implementasi dari aturan morfotaktik dilakukan dalam perangkat lunak morphological parser dengan melakukan validasi hasil parsing melalui perangkat lunak lexicon

4. Performansi dari perangkat lunak yang dihasilkan adalah sebagai berikut:

- Perangkat lunak morphological parser yang dibuat dapat melakukan proses parsing dalam waktu yang masuk akal, yaitu kurang dari 100 milidetik untuk teks yang terdiri dari 100 kata
- Perangkat lunak morphological parser yang dibuat dapat menghasilkan semua kemungkinan hasil parsing yang valid dari sebuah kata
- Perangkat lunak lexicon yang dibuat dapat melakukan create, update, dan delete untuk entri kata dasar dan kata turunan yang disimpan dalam leksikon
- Operasi create dan delete pada perangkat lunak lexicon membutuhkan waktu cukup lama, yaitu sekitar 5 detik, dikarenakan perangkat lunak harus menelusuri node dalam pohon node dan harus mencetak ulang keseluruhan kata dalam leksikon setelah operasi tersebut selesai dilakukan

## 6.2 Saran

Beberapa saran yang dapat dilakukan untuk memperbaiki dan mengembangkan penelitian ini selanjutnya adalah sebagai berikut.

- Melengkapi seluruh kata turunan untuk setiap kata dasar dalam leksikon supaya semua kata dalam bahasa Indonesia dapat diproses dan divalidasi dengan tepat.
- Menambahkan kelas kata seperti kata benda, kata kerja, kata sifat, dan lain-lain untuk setiap kata dasar dan kata turunan yang disimpan dalam leksikon supaya dapat digunakan dalam proses lebih lanjut dalam pengolahan bahasa alami.

## DAFTAR REFERENSI

- [1] Jurafsky, D. dan Martin, J. H. (2009) *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*, 2nd edition. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- [2] Chaer, A. (2008) *Morfologi Bahasa Indonesia (Pendekatan Proses)*. Rineka Cipta, Jakarta.
- [3] Najogie, R. D. (2010) Pengenalan trie dan aplikasinya. *Makalah IF2091 Struktur Diskrit - Sem. I Tahun 2010/2011*, **1**, 91–95.



## **LAMPIRAN A**

### **BAHAN PENGUJIAN**

Pada tahap pengujian nonfungsional, isi dari file txt yang digunakan adalah sebagai berikut.

1. Bakso atau baso adalah bola daging yang ada di Indonesia.
2. Bakso atau baso adalah bola daging yang lazim ditemukan pada masakan Indonesia. Bakso dibuat dari daging sapi dan tepung tapioka.
3. Bakso atau baso adalah jenis bola daging yang lazim ditemukan pada masakan Indonesia. Bakso umumnya dibuat dari campuran daging sapi giling dan tepung tapioka, akan tetapi ada juga bakso yang terbuat dari daging ayam, ikan, atau udang bahkan daging kerbau. Dalam penyajiannya, bakso umumnya disajikan panas-panas dengan kuah kaldu sapi bening, dicampur mi, bihun, taoge, tahu, terkadang telur dan ditaburi bawang goreng dan seledri. Bakso sangat populer dan dapat ditemukan hampir di seluruh wilayah Indonesia; dari gerobak pedagang kaki lima hingga restoran besar. Berbagai jenis bakso sekarang banyak ditawarkan dalam bentuk makanan beku yang dijual di pasar swalayan ataupun mal-mal.





# LAMPIRAN B

## KODE PROGRAM

Listing B.1: Parser.java

```
1 package Model;
2
3 import Lexicon.Model.Lexicon;
4 import java.io.BufferedReader;
5 import java.io.FileReader;
6 import java.io.IOException;
7 import java.util.ArrayList;
8
9 /**
10  *
11  * @author Andreas Novian
12  */
13 public class Parser {
14
15     //an object of a lexicon
16     public Lexicon lexicon;
17
18     private final ArrayList<String> parseResult;
19
20     public Parser(Lexicon lexicon) throws IOException {
21         this.lexicon = lexicon;
22         this.parseResult = new ArrayList<>();
23     }
24
25     /**
26      * Method to search whether a word is a valid root word in lexicon tree or
27      * not
28      *
29      * @param word a word to check
30      * @return true if and only if word is a valid root word in lexicon tree
31      */
32     private boolean isRootWord(String word) {
33         if (!word.equalsIgnoreCase("")) {
34             return lexicon.searchInTree(word.toLowerCase());
35         } else {
36             return false;
37         }
38     }
39
40     /**
41      * Check each component in tempResult, cross check with lexicon, delete from
42      * tempResult if not valid based on lexicon
43      *
44      * @throws IOException
45      */
46     private void validateComponent() throws IOException {
47         String rootWord;
48         String component;
49         String line;
50         boolean valid;
51         String[] words;
52
53         for (int i = 0; i < this.parseResult.size(); i++) {
54             component = "";
55             line = this.parseResult.get(i);
56             words = line.split("\\\\+");
57             rootWord = words[0];
58             for (int j = 1; j < words.length; j++) {
59                 if (words[j].charAt(0) != '$' && words[j].charAt(0) != '%') {
60                     component += words[j] + "+";
61                 }
62             }
63
64             if (!component.equalsIgnoreCase("")) {
65                 component = component.substring(0, component.length() - 1);
66                 if (isRootWord(rootWord)) {
67                     valid = this.lexicon.searchInFile(rootWord, component);
68                     if (!valid) {
69                         this.parseResult.remove(line);
70                         i--;
71                     }
72                 } else {
73                     this.parseResult.remove(line);
74                     i--;
75                 }
76             }
77         }
78     }
79 }
```

```

76     }
77 }
78 }
79
80 /**
81  * To convert from String "malu+#per-kan+[me+$mu] to "Prefiks [me] + Bentuk
82  * Dasar [malu] + Konfiks [per-kan] + Klitika [mu]"
83  */
84 private void convertToWord() throws IOException {
85     String line, result;
86     String[] words;
87
88     for (int i = 0; i < this.parseResult.size(); i++) {
89         result = "";
90         line = this.parseResult.get(i);
91         words = line.split("\\\\+");
92
93         for (String word : words) {
94             switch (word.charAt(0)) {
95                 case '@':
96                     result += "Komposisi_" + word.substring(1) + "_+";
97                     break;
98                 case '^':
99                     result += "Reduplikasi_" + word.substring(1) + "_+";
100                    break;
101                 case '[':
102                     result += "Prefiks_" + word.substring(1) + "_+";
103                     break;
104                 case ']':
105                     result += "Sufiks_" + word.substring(1) + "_+";
106                     break;
107                 case '#':
108                     result += "Konfiks_" + word.substring(1) + "_+";
109                     break;
110                 case '$':
111                     result += "Proklitika_" + word.substring(1) + "_+";
112                     break;
113                 case '%':
114                     result += "Enklitika_" + word.substring(1) + "_+";
115                     break;
116                 case '!':
117                     result += "Bentuk_Asing_" + word.substring(1) + "_+";
118                     break;
119                 default:
120                     result += "Bentuk_Dasar_" + word + "_+";
121                     break;
122             }
123         }
124
125         result = result.substring(0, result.length() - 3);
126         this.parseResult.remove(i);
127         this.parseResult.add(i, result);
128     }
129 }
130
131 private void removeDuplicateResult() {
132     String l1, l2;
133
134     //remove same item
135     for (int i = 0; i < this.parseResult.size(); i++) {
136         l1 = this.parseResult.get(i);
137         for (int j = i + 1; j < this.parseResult.size(); j++) {
138             l2 = this.parseResult.get(j);
139             if (l1.equalsIgnoreCase(l2)) {
140                 this.parseResult.remove(j);
141                 if (i > 0) {
142                     i--;
143                 }
144                 j--;
145             }
146         }
147     }
148 }
149
150 /**
151  * Normalize text before entering parse process
152  *
153  * @param text
154  * @return String[] containing each word in text but in lowercase, and only
155  * character a..z, 0..9, and - allowed
156  */
157 private String[] normalizeInput(String text) {
158     String[] tempArray;
159     String tempWord;
160     String input = "";
161     String[] words;
162     char c;
163
164     text = text.toLowerCase();
165     tempArray = text.split("\\\\s");
166     for (String word : tempArray) {
167         if (!word.equalsIgnoreCase("")) {
168             tempWord = "";
169             for (int i = 0; i < word.length(); i++) {
170                 c = word.charAt(i);
171                 //a..z
172                 if (c >= 97 && c <= 122) {
173                     tempWord += (char) c;
174                 }

```

```

175         //0..9
176         if (c >= 48 && c <= 57) {
177             tempWord += (char) c;
178         }
179         //symbol -
180         if (c == 45) {
181             tempWord += (char) c;
182         }
183     }
184     if (!tempWord.equalsIgnoreCase("")) {
185         input += tempWord + " ";
186     }
187 }
188 }
189 input = input.trim();
190 //System.out.println(input);
191 words = input.split("\\s");
192 return words;
193 }
194
195 /**
196  * Method to do parsing process a line of text by divide them into each word
197  * separated by space character, then parse each word
198  *
199  * @param text line of text to parse
200  * @param validator
201  * @param converter
202  * @return a list of all the possible parse of each word in text
203  *
204  * @throws java.io.IOException
205  */
206 public String process(String text, boolean validator, boolean converter) throws IOException {
207     String result = "";
208     String words[] = normalizeInput(text);
209     String word;
210     ArrayList<String> oneWord = new ArrayList<>(), twoWord = new ArrayList<>();
211
212     for (int i = 0; i < words.length; i++) {
213         oneWord.clear();
214         twoWord.clear();
215         this.parseResult.clear();
216
217         word = words[i];
218         if (!word.equalsIgnoreCase("")) {
219             parse(word.toLowerCase());
220             if (i < words.length - 1) {
221                 this.checkKomposisi(words[i + 1]);
222             }
223
224             this.removeDuplicateResult();
225
226             if (validator) {
227                 this.validateComponent();
228             }
229
230             if (this.parseResult.isEmpty()) {
231                 this.parseResult.add("!" + word);
232             }
233
234             for (int j = 0; j < this.parseResult.size(); j++) {
235                 if (this.parseResult.get(j).contains("&")) {
236                     twoWord.add(this.parseResult.get(j));
237                 } else {
238                     oneWord.add(this.parseResult.get(j));
239                 }
240             }
241
242             this.parseResult.clear();
243             this.parseResult.addAll(oneWord);
244             this.parseResult.addAll(twoWord);
245
246             if (converter) {
247                 this.convertToWord();
248             }
249
250             if (!oneWord.isEmpty()) {
251                 result += word.toUpperCase() + ":\n";
252                 for (int j = 0; j < oneWord.size(); j++) {
253                     result += this.parseResult.get(j) + ";\n";
254                 }
255                 result += "\n";
256             }
257             if (!twoWord.isEmpty()) {
258                 result += word.toUpperCase() + " " + words[i + 1].toUpperCase() + ":\n";
259                 for (int j = 0; j < this.parseResult.size(); j++) {
260                     result += this.parseResult.get(j) + ";\n";
261                 }
262                 result += "\n";
263             }
264         }
265     }
266     result = result.trim();
267     return result;
268 }
269
270 /**
271  * Method to do parsing process a line of text by divide them into each word
272  * separated by space character, then parse each word
273  *

```

```

274  * @param filePath path to the input file
275  * @return a list of all the possible parse of each word in text
276  *
277  * @throws java.io.IOException
278  */
279  public String readFile(String filePath) throws IOException {
280      String text = "";
281
282      try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
283          String currentLine;
284          while ((currentLine = br.readLine()) != null) {
285              if (!currentLine.equalsIgnoreCase("")) {
286                  text += currentLine;
287                  text = text.trim();
288                  text += " ";
289              }
290          }
291      }
292
293      return text.trim();
294  }
295
296  /**
297   * Method to perform parse operation of a word
298   *
299   * @param word word to parse
300   */
301  private void parse(String word) throws IOException {
302      boolean isAWord = true;
303      for (int i = 0; i < word.length(); i++) {
304          int c = (int) word.charAt(i);
305          if (c < 97 || c > 122) {
306              isAWord = false;
307          }
308          if (c == 45) {
309              isAWord = true;
310          }
311      }
312
313      if (isAWord) {
314          this.check(word, "", "");
315      }
316
317      if (this.parseResult.isEmpty()) {
318          this.parseResult.add("!" + word);
319      }
320  }
321
322  /**
323   * Check all the possible prefiks, including klitika, even when combined
324   *
325   * @param word word to check
326   * @param klitika any klitika found
327   * @param prefiks any prefiks found previously
328   */
329  private void checkPrefiks(String word, String component, String klitika) throws IOException {
330      //two letters prefiks
331      if (word.length() > 2) {
332          String c2 = word.substring(0, 2);
333          String w2 = word.substring(2);
334
335          if (c2.equalsIgnoreCase("be")) {
336              prefiksBer(w2, component, klitika);
337          } else if (c2.equalsIgnoreCase("me")) {
338              prefiksMe(w2, component, klitika);
339          } else if (c2.equalsIgnoreCase("di")) {
340              prefiksDi(w2, component, klitika);
341          } else if (c2.equalsIgnoreCase("ke")) {
342              prefiksKe(w2, component, klitika);
343          } else if (c2.equalsIgnoreCase("ku")) {
344              proklitikaKu(w2, component, klitika);
345          } else if (c2.equalsIgnoreCase("se")) {
346              prefiksSe(w2, component, klitika);
347          } else if (c2.equalsIgnoreCase("pe")) {
348              prefiksPe(w2, component, klitika);
349          } else if (c2.equalsIgnoreCase("te")) {
350              prefiksTer(w2, component, klitika);
351          }
352      }
353
354      //three letters prefiks
355      if (word.length() > 3) {
356          String c3 = word.substring(0, 3);
357          String w3 = word.substring(3);
358
359          if (c3.equalsIgnoreCase("per") || c3.equalsIgnoreCase("pel")) {
360              prefiksPer(w3, component, klitika);
361          } else if (c3.equalsIgnoreCase("kau")) {
362              proklitikaKau(w3, component, klitika);
363          }
364      }
365  }
366
367  /**
368   * Check all the possible sufiks, including klitika, even when combined ex.
369   * makananmu
370   *
371   * @param word word to check

```

```

373  * @param klitika any klitika found
374  * @param prefiks any prefiks found previously
375  */
376  private void checkSufiks(String word, String component, String klitika) throws IOException {
377      if (word.length() > 2) {
378          String c3 = word.substring(word.length() - 3);
379          String w3 = word.substring(0, word.length() - 3);
380
381          if (c3.equalsIgnoreCase("kan")) {
382              sufiksKan(w3, component, klitika);
383          } else if (c3.equalsIgnoreCase("nya")) {
384              enklitikaNya(w3, component, klitika);
385          } else if (c3.equalsIgnoreCase("lah")) {
386              enklitikaLah(w3, component, klitika);
387          } else if (c3.equalsIgnoreCase("pun")) {
388              enklitikaPun(w3, component, klitika);
389          } else if (c3.equalsIgnoreCase("kah")) {
390              enklitikaKah(w3, component, klitika);
391          }
392      }
393      if (word.length() > 1) {
394          String c2 = word.substring(word.length() - 2);
395          String w2 = word.substring(0, word.length() - 2);
396
397          if (c2.equalsIgnoreCase("an")) {
398              sufiksAn(w2, component, klitika);
399          } else if (c2.equalsIgnoreCase("ku")) {
400              enklitikaKu(w2, component, klitika);
401          } else if (c2.equalsIgnoreCase("mu")) {
402              enklitikaMu(w2, component, klitika);
403          }
404      }
405      if (word.length() > 0) {
406          String c1 = word.substring(word.length() - 1);
407          String w1 = word.substring(0, word.length() - 1);
408
409          if (c1.equalsIgnoreCase("i")) {
410              sufiksI(w1, component, klitika);
411          }
412      }
413  }
414
415  /**
416   * Check if word is reduplication or not based on the present of "-."
417   *
418   * @param word word to check
419   * @param klitika any klitika found
420   * @param prefiks any prefiks found previously
421   * @throws IOException
422   */
423  private void checkRedup(String word, String component, String klitika) throws IOException {
424      String temp;
425      boolean haveResult = false;
426      if (word.contains("-.")) {
427          String[] words = word.split("-.");
428
429          if (words[0].equalsIgnoreCase(words[1])) {
430              if (isRootWord(words[0])) {
431                  temp = words[0] + "+^2" + component + klitika;
432                  this.parseResult.add(temp);
433                  haveResult = true;
434              }
435              //this.check(words[0], component + "+^2", klitika);
436          } else {
437              if (isRootWord(words[0])) {
438                  String c1 = words[0].substring(0, 1); //first word first letter
439                  String c2 = words[1].substring(0, 1); //second word first letter
440                  String c3 = words[1].substring(0, 2); //second word first two letters
441                  String w1 = words[1].substring(1); //second word minus c2
442                  String w2 = words[1].substring(2); //second word minus c3
443
444                  if (isRootWord(words[1])) {
445                      temp = words[0] + "+^" + words[1] + component + klitika;
446                      this.parseResult.add(temp);
447                  } else if (c1.equalsIgnoreCase("k")) {
448                      if (c3.equalsIgnoreCase("ng")) {
449                          if (words[0].equalsIgnoreCase("k" + w2)) {
450                              temp = words[0] + "+^2" + component + klitika;
451                              this.parseResult.add(temp);
452                              haveResult = true;
453                          }
454                      }
455                  } else if (c1.equalsIgnoreCase("t")) {
456                      if (c2.equalsIgnoreCase("n")) {
457                          if (words[0].equalsIgnoreCase("t" + w1)) {
458                              temp = words[0] + "+^2" + component + klitika;
459                              this.parseResult.add(temp);
460                              haveResult = true;
461                          }
462                      }
463                  } else if (c1.equalsIgnoreCase("s")) {
464                      if (c3.equalsIgnoreCase("ny")) {
465                          if (words[0].equalsIgnoreCase("s" + w2)) {
466                              temp = words[0] + "+^2" + component + klitika;
467                              this.parseResult.add(temp);
468                              haveResult = true;
469                          }
470                      }
471                  } else if (c1.equalsIgnoreCase("p")) {

```

```

472         if (c2.equalsIgnoreCase("m")) {
473             if (words[0].equalsIgnoreCase("p" + w1)) {
474                 temp = words[0] + "+"^2 + component + klitika;
475                 this.parseResult.add(temp);
476                 haveResult = true;
477             }
478         }
479     }
480
481     if (!haveResult) {
482         temp = words[0] + "+"^ + words[1] + component + klitika;
483         this.parseResult.add(temp);
484     }
485 }
486 }
487 }
488 }
489
490 private void check(String word, String component, String klitika) throws IOException {
491     //if word is found in lexicon
492     if (isRootWord(word)) {
493         this.parseResult.add(word + component + klitika);
494     }
495     //afixed word must be 3 or more letters
496     if (word.length() > 3) {
497         //prefiks check, including sufiks check
498         checkPrefiks(word, component, klitika);
499
500         //only sufiks check
501         checkSufiks(word, component, klitika);
502
503         //reduplication check
504         checkRedup(word, component, klitika);
505     }
506 }
507
508 private void prefiksBer(String word, String component, String klitika) throws IOException {
509     //ex. beragam
510     this.check(word, "+[ber" + component, klitika);
511
512     if (word.length() > 3) {
513         if (word.charAt(0) == 'r' || word.charAt(0) == 'l') {
514             //ex. beranak, belajar
515             word = word.substring(1);
516             this.check(word, "+[ber" + component, klitika);
517         }
518     }
519 }
520
521 private void prefiksMe(String word, String component, String klitika) throws IOException {
522     //me..
523     this.check(word, "+[me" + component, klitika);
524
525     if (word.length() > 3) {
526         //mem..;
527         if (word.charAt(0) == 'm') {
528             word = word.substring(1);
529             this.check(word, "+[me" + component, klitika);
530             this.check("p" + word, "+[me" + component, klitika);
531         }
532
533         //men..;
534         if (word.charAt(0) == 'n') {
535             word = word.substring(1);
536             this.check(word, "+[me" + component, klitika);
537             this.check("t" + word, "+[me" + component, klitika);
538
539             if (word.length() > 3) {
540                 //meng..;
541                 if (word.charAt(0) == 'g') {
542                     word = word.substring(1);
543                     this.check(word, "+[me" + component, klitika);
544                     this.check("k" + word, "+[me" + component, klitika);
545
546                     if (word.length() > 3) {
547                         //menge..
548                         if (word.charAt(0) == 'e') {
549                             word = word.substring(1);
550                             this.check(word, "+[me" + component, klitika);
551                         }
552                     }
553                 }
554                 //meny..;
555                 if (word.charAt(0) == 'y') {
556                     word = word.substring(1);
557                     //this.check(word, "+[me" + component, klitika);
558                     this.check("s" + word, "+[me" + component, klitika);
559                 }
560             }
561         }
562     }
563 }
564
565 private void prefiksDi(String word, String component, String klitika) throws IOException {
566     this.check(word, "+[di" + component, klitika);
567 }
568
569 private void prefiksKe(String word, String component, String klitika) throws IOException {
570     this.check(word, "+[ke" + component, klitika);

```

```

571     }
572
573     private void proklitikaKu(String word, String component, String klitika) throws IOException {
574         this.check(word, component, klitika + "+$ku");
575     }
576
577     private void prefiksSe(String word, String component, String klitika) throws IOException {
578         this.check(word, "+[se" + component, klitika);
579     }
580
581     private void prefiksPe(String word, String component, String klitika) throws IOException {
582         this.check(word, "+[pe" + component, klitika);
583         prefiksPer(word, component, klitika);
584
585         if (word.length() > 3) {
586             //pem..;
587             if (word.charAt(0) == 'm') {
588                 word = word.substring(1);
589                 this.check(word, "+[pe" + component, klitika);
590                 this.check("p" + word, "+[pe" + component, klitika);
591             }
592
593             //pen..;
594             if (word.charAt(0) == 'n') {
595                 word = word.substring(1);
596                 this.check(word, "+[pe" + component, klitika);
597                 this.check("t" + word, "+[pe" + component, klitika);
598
599                 if (word.length() > 3) {
600                     //peng..;
601                     if (word.charAt(0) == 'g') {
602                         word = word.substring(1);
603                         this.check(word, "+[pe" + component, klitika);
604                         this.check("k" + word, "+[pe" + component, klitika);
605
606                         if (word.length() > 3) {
607                             //penge..
608                             if (word.charAt(0) == 'e') {
609                                 word = word.substring(1);
610                                 this.check(word, "+[pe" + component, klitika);
611                             }
612                         }
613                     }
614                     //peny..;
615                     if (word.charAt(0) == 'y') {
616                         word = word.substring(1);
617                         //this.check(word, "+[pe" + component, klitika);
618                         this.check("s" + word, "+[pe" + component, klitika);
619                     }
620                 }
621             }
622         }
623     }
624
625     private void prefiksPer(String word, String component, String klitika) throws IOException {
626         this.check(word, "+[per" + component, klitika);
627     }
628
629     private void prefiksTer(String word, String component, String klitika) throws IOException {
630         this.check(word, "+[ter" + component, klitika);
631
632         if (word.length() > 3) {
633             if (word.charAt(0) == 'r') {
634                 word = word.substring(1);
635                 this.check(word, "+[ter" + component, klitika);
636             }
637         }
638     }
639
640     private void proklitikaKau(String word, String component, String klitika) throws IOException {
641         this.check(word, component, klitika + "+$kau");
642     }
643
644     private void sufiksKan(String word, String component, String klitika) throws IOException {
645         String temp;
646         if (isRootWord(word)) {
647             temp = word + "+]kan" + component + klitika;
648             this.parseResult.add(temp);
649             this.checkKonfiks();
650         }
651         this.checkKomposisi(word, "+]kan" + component, klitika);
652         this.checkKonfiks();
653         this.check(word, "+]kan" + component, klitika);
654     }
655
656     private void sufiksAn(String word, String component, String klitika) throws IOException {
657         String temp;
658         if (isRootWord(word)) {
659             temp = word + "+]an" + component + klitika;
660             this.parseResult.add(temp);
661             this.checkKonfiks();
662         }
663         this.checkKomposisi(word, "+]an" + component, klitika);
664         this.checkKonfiks();
665         this.check(word, "+]an" + component, klitika);
666     }
667
668     private void sufiksI(String word, String component, String klitika) throws IOException {
669         String temp;

```

```

670     if (isRootWord(word)) {
671         temp = word + "+" + i + component + klitika;
672         this.parseResult.add(temp);
673         this.checkKonfiks();
674     }
675     this.checkKomposisi(word, "+" + i + component, klitika);
676     this.checkKonfiks();
677     this.check(word, "+" + i + component, klitika);
678 }
679
680 private void enklitikaKu(String word, String component, String klitika) throws IOException {
681     String temp;
682     if (isRootWord(word)) {
683         temp = word + component + "+%ku" + klitika;
684         this.parseResult.add(temp);
685     }
686     this.check(word, component, "+%ku" + klitika);
687 }
688
689 private void enklitikaMu(String word, String component, String klitika) throws IOException {
690     String temp;
691     if (isRootWord(word)) {
692         temp = word + component + "+%mu" + klitika;
693         this.parseResult.add(temp);
694     }
695     this.check(word, component, "+%mu" + klitika);
696 }
697
698 private void enklitikaNya(String word, String component, String klitika) throws IOException {
699     String temp;
700     if (isRootWord(word)) {
701         temp = word + component + "+%nya" + klitika;
702         this.parseResult.add(temp);
703     }
704     this.check(word, component, "+%nya" + klitika);
705 }
706
707 private void enklitikaLah(String word, String component, String klitika) throws IOException {
708     String temp;
709     if (isRootWord(word)) {
710         temp = word + component + "+%lah" + klitika;
711         this.parseResult.add(temp);
712     }
713     this.check(word, component, "+%lah" + klitika);
714 }
715
716 private void enklitikaPun(String word, String component, String klitika) throws IOException {
717     String temp;
718     if (isRootWord(word)) {
719         temp = word + component + "+%pun" + klitika;
720         this.parseResult.add(temp);
721     }
722     this.check(word, component, "+%pun" + klitika);
723 }
724
725 private void enklitikaKah(String word, String component, String klitika) throws IOException {
726     String temp;
727     if (isRootWord(word)) {
728         temp = word + component + "+%kah" + klitika;
729         this.parseResult.add(temp);
730     }
731     this.check(word, component, "+%kah" + klitika);
732 }
733
734 private void checkKonfiks() {
735     String rootWord, component, line;
736     for (int i = 0; i < this.parseResult.size(); i++) {
737         rootWord = "";
738         line = this.parseResult.get(i);
739         if (line.contains("+")) {
740             for (int j = 0; j < line.indexOf("+"); j++) {
741                 rootWord += line.charAt(j);
742             }
743             component = line.substring(line.indexOf("+") + 1);
744
745             String temp;
746             if (component.contains("]an+[ber") {
747                 temp = component.replace("]an+[ber", "#ber-an");
748                 this.parseResult.add(rootWord + "+" + temp);
749             } else if (component.contains("]an+[ke") {
750                 temp = component.replace("]an+[ke", "#ke-an");
751                 this.parseResult.add(rootWord + "+" + temp);
752             } else if (component.contains("]an+[per") {
753                 temp = component.replace("]an+[per", "#per-an");
754                 this.parseResult.add(rootWord + "+" + temp);
755             } else if (component.contains("]an+[pe") {
756                 temp = component.replace("]an+[pe", "#pe-an");
757                 this.parseResult.add(rootWord + "+" + temp);
758             } else if (component.contains("]nya+[se") {
759                 temp = component.replace("]nya+[se", "#se-nya");
760                 this.parseResult.add(rootWord + "+" + temp);
761             }
762         }
763     }
764 }
765
766 /**
767  * To check if a word is first komposisi then afixed ex. pertanggungjawaban
768  *

```



```

769 |     * @param word word to check
770 |     * @param klitika any klitika found
771 |     * @param prefiks any prefiks found previously
772 |     */
773 |     private void checkKomposisi(String word, String component, String klitika) {
774 |         String rootWord = "";
775 |         String temp;
776 |         for (int i = 0; i < word.length() - 1; i++) {
777 |             rootWord += word.charAt(i);
778 |             temp = word.substring(i + 1);
779 |             if (isRootWord(rootWord)) {
780 |                 if (isRootWord(temp)) {
781 |                     temp = rootWord + "+" + temp + component + klitika;
782 |                     if (temp.contains("[") && temp.contains("]") && !temp.contains("-")) {
783 |                         this.parseResult.add(temp);
784 |                     }
785 |                 }
786 |             }
787 |         }
788 |     }
789 | }
790 |
791 | /**
792 |  * To check if a word is first afixed then komposisi
793 |  */
794 |     private void checkKomposisi(String nextWord) {
795 |         if (isRootWord(nextWord)) {
796 |             String line, newLine;
797 |             int size = this.parseResult.size();
798 |             for (int i = 0; i < size; i++) {
799 |                 line = this.parseResult.get(i);
800 |                 newLine = line + "+" + nextWord + " ";
801 |                 this.parseResult.add(newLine);
802 |             }
803 |         }
804 |     }

```

Listing B.2: Lexicon.java

```

1 | package Lexicon.Model;
2 |
3 | import java.io.BufferedReader;
4 | import java.io.BufferedWriter;
5 | import java.io.File;
6 | import java.io.FileNotFoundException;
7 | import java.io.FileReader;
8 | import java.io.FileWriter;
9 | import java.io.IOException;
10 | import java.util.ArrayList;
11 | import java.util.HashMap;
12 |
13 | /**
14 |  * Class to represent the lexicon used in Morphological Parser Lexicon keeps
15 |  * root words in Bahasa Indonesia and it's morphological component
16 |  *
17 |  * @author Andreas Novian
18 |  */
19 | public class Lexicon {
20 |
21 |     //each letter of a roots stored in a single node
22 |     private final HashMap<Character, Node> roots;
23 |
24 |     //to read from file
25 |     private BufferedReader br;
26 |
27 |     //to write into file
28 |     private BufferedWriter bw;
29 |
30 |     //the folder where the lxc file is in the project folder
31 |     private final String folder;
32 |
33 |     private final Combiner combiner;
34 |
35 |     public Lexicon() throws FileNotFoundException, IOException {
36 |         this.roots = new HashMap<>();
37 |         this.folder = "lxc/";
38 |         this.combiner = new Combiner();
39 |         load();
40 |     }
41 |
42 |     /**
43 |      * Method to initialize the lexicon tree based on roots in the "roots" file
44 |      *
45 |      * @throws FileNotFoundException
46 |      * @throws IOException
47 |      */
48 |     private void load() throws FileNotFoundException, IOException {
49 |         this.br = new BufferedReader(new FileReader(folder + "roots.lxc"));
50 |         String currentLine;
51 |         while ((currentLine = br.readLine()) != null) {
52 |             if (!currentLine.equalsIgnoreCase("")) {
53 |                 this.insertToTree(currentLine);
54 |             }
55 |         }
56 |         this.br.close();
57 |     }
58 | }
59 | /**

```

```

60      * Method to write a String into a lexicon file (.lxc)
61      *
62      * @param fileName name of the lexicon file
63      * @param key String to be entered
64      *
65      * @throws IOException
66      */
67      private void writeToFile(String fileName, String key) throws IOException {
68          String path = folder + fileName + ".lxc";
69          this.bw = new BufferedWriter(new FileWriter(path, true));
70          if (!key.equalsIgnoreCase("")) {
71              this.bw.write(key + "\n");
72          }
73          this.bw.flush();
74          this.bw.close();
75      }
76
77      /**
78      * Method to delete a String from a lexicon file
79      *
80      * @param fileName name of the lexicon file
81      * @param key String to be deleted
82      *
83      * @throws FileNotFoundException
84      * @throws IOException
85      */
86      private void deleteFromFile(String fileName, String key) throws FileNotFoundException, IOException {
87          String content = "";
88          this.br = new BufferedReader(new FileReader(folder + fileName + ".lxc"));
89          String currentLine;
90          while ((currentLine = br.readLine()) != null) {
91              if (!currentLine.equalsIgnoreCase(key) && !currentLine.equalsIgnoreCase("")) {
92                  content += currentLine + "\n";
93              }
94          }
95          this.br.close();
96          content = content.trim();
97          this.clearFile(fileName);
98          this.writeToFile(fileName, content);
99      }
100
101      /**
102      * Method to make a new lexicon file for a new root word entry
103      *
104      * @param fileName new root word entry, as the new lexicon file name
105      *
106      * @throws IOException
107      */
108      private void createFile(String fileName) throws IOException {
109          String path = folder + fileName + ".lxc";
110          this.bw = new BufferedWriter(new FileWriter(path));
111          this.bw.flush();
112          this.bw.close();
113      }
114
115      /**
116      * Method to delete a lexicon file
117      *
118      * @param fileName root word, name of the lexicon file
119      * @return true if and only if the file is successfully deleted
120      */
121      private boolean deleteFile(String fileName) {
122          String path = folder + fileName + ".lxc";
123          boolean status = new File(path).delete();
124          if (status) {
125              System.out.println("File_" + fileName + "_deleted");
126          } else {
127              System.out.println("Fail_to_delete_" + fileName);
128          }
129          return status;
130      }
131
132      /**
133      * Method to clear the content of a lexicon file
134      *
135      * @param fileName name of the lexicon file
136      *
137      * @throws IOException
138      */
139      private void clearFile(String fileName) throws IOException {
140          String path = folder + fileName + ".lxc";
141          this.bw = new BufferedWriter(new FileWriter(path, false));
142          this.bw.write("");
143          this.bw.flush();
144          this.bw.close();
145      }
146
147      /**
148      * Method to get all the morphological component from a single root
149      *
150      * @param root root word, name of the lexicon file
151      * @return String containing all components from a single root
152      *
153      * @throws FileNotFoundException
154      * @throws IOException
155      */
156      public String getComponent(String root) throws FileNotFoundException, IOException {
157          String content = "";
158          this.br = new BufferedReader(new FileReader(folder + root + ".lxc"));

```

```

159 |         this.br.readLine();
160 |         String currentLine;
161 |         while ((currentLine = br.readLine()) != null) {
162 |             content += currentLine + "\n";
163 |         }
164 |         content = content.trim();
165 |         this.br.close();
166 |         return content;
167 |     }
168 |
169 |     /**
170 |      * Method to search if a key is already in the tree or not
171 |      *
172 |      * @param key root word as a key to search in tree
173 |      * @return true if and only if the root word is already in the tree
174 |      */
175 |     public boolean searchInTree(String key) {
176 |         if (roots.containsKey(key.charAt(0))) {
177 |             if (key.length() == 1 && roots.get(key.charAt(0)).endOfWord) {
178 |                 return true;
179 |             }
180 |             return searchRec(key.substring(1), roots.get(key.charAt(0)));
181 |         } else {
182 |             return false;
183 |         }
184 |     }
185 |
186 |     /**
187 |      * Private method to recursively search in the lexicon tree
188 |      *
189 |      * @param string root word
190 |      * @param node discovered node
191 |      * @return true whether the value of attribute endOfWord in param node
192 |      */
193 |     private boolean searchRec(String string, Node node) {
194 |         if (string.length() == 0) {
195 |             return node.endOfWord;
196 |         }
197 |         if (node.children.containsKey(string.charAt(0))) {
198 |             return searchRec(string.substring(1), node.children.get(string.charAt(0)));
199 |         } else {
200 |             return false;
201 |         }
202 |     }
203 |
204 |     /**
205 |      * Method to search if a String key is in a file or not
206 |      *
207 |      * @param fileName name of the file
208 |      * @param key String to search
209 |      * @return true if and only if key String is in the file
210 |      *
211 |      * @throws FileNotFoundException
212 |      * @throws IOException
213 |      */
214 |     public boolean searchInFile(String fileName, String key) throws FileNotFoundException, IOException {
215 |         this.br = new BufferedReader(new FileReader(folder + fileName + ".lxc"));
216 |         String currentLine;
217 |         while ((currentLine = br.readLine()) != null) {
218 |             if (currentLine.equalsIgnoreCase(key)) {
219 |                 return true;
220 |             }
221 |         }
222 |         this.br.close();
223 |         return false;
224 |     }
225 |
226 |     /**
227 |      * Method to insert a new root word to lexicon, including creating a new
228 |      * file for the entry
229 |      *
230 |      * @param root the root word to be entered
231 |      * @throws IOException
232 |      */
233 |     public void insertRoot(String root) throws IOException {
234 |         //only insert if the root word is not yet in the tree
235 |         if (!this.searchInTree(root)) {
236 |             this.insertToTree(root);
237 |
238 |             //refresh the roots file
239 |             this.clearFile("roots");
240 |             String allWords = this.printAllWordInTree();
241 |             this.writeToFile("roots", allWords);
242 |
243 |             this.createFile(root);
244 |             this.writeToFile(root, root);
245 |         }
246 |     }
247 |
248 |     /**
249 |      * Method to insert a word into the tree
250 |      *
251 |      * @param word word to be entered
252 |      */
253 |     private void insertToTree(String word) {
254 |         if (!roots.containsKey(word.charAt(0))) {
255 |             roots.put(word.charAt(0), new Node());
256 |         }
257 |         insertRec(word.substring(1), roots.get(word.charAt(0)));

```

```

258     }
259
260     /**
261     * Private method to recursively insert a word to the tree
262     *
263     * @param word word to be entered
264     * @param node discovered node
265     */
266     private void insertRec(String word, Node node) {
267         final Node nextChild;
268         if (node.children.containsKey(word.charAt(0))) {
269             nextChild = node.children.get(word.charAt(0));
270         } else {
271             nextChild = new Node();
272             nextChild.parent = node;
273             node.children.put(word.charAt(0), nextChild);
274         }
275         if (word.length() == 1) {
276             nextChild.endOfWord = true;
277         } else {
278             insertRec(word.substring(1), nextChild);
279         }
280     }
281
282     /**
283     * Method to write morphological component of a root word into the lexicon
284     * file
285     *
286     * @param root root word, also the name of the lexicon file
287     * @param component morphological component to be entered
288     *
289     * @throws IOException
290     */
291     public void insertComponent(String root, String component) throws IOException {
292         if (!this.searchInFile(root, component)) {
293             this.writeToFile(root, component);
294         }
295     }
296
297     /**
298     * Method to delete a root word from lexicon Delete root from the tree and
299     * from the file
300     *
301     * @param root root word to be deleted
302     *
303     * @throws IOException
304     */
305     public void deleteRoot(String root) throws IOException {
306         this.deleteFile(root);
307         deleteRec(root.substring(1), roots.get(root.charAt(0)));
308
309         this.clearFile("roots");
310         String allWords = this.printAllWordInTree();
311         this.writeToFile("roots", allWords);
312     }
313
314     /**
315     * Private method to recursively delete a root word from the tree
316     *
317     * @param word word to be deleted
318     * @param node discovered node
319     */
320     private void deleteRec(String word, Node node) {
321         final Node nextChild;
322         nextChild = node.children.get(word.charAt(0));
323         if (word.length() == 1) {
324             nextChild.endOfWord = false;
325         } else {
326             deleteRec(word.substring(1), nextChild);
327         }
328     }
329 }
330
331 /**
332 * Method to delete morphological component of a root word from the file
333 *
334 * @param root root word, also name of the file
335 * @param component morphological component to be deleted
336 *
337 * @throws FileNotFoundException
338 * @throws IOException
339 */
340 public void deleteComponent(String root, String component) throws FileNotFoundException, IOException {
341     this.deleteFromFile(root, component);
342 }
343
344 /**
345 * Method to update a root word
346 *
347 * @param oldRoot root to update
348 * @param newRoot new updated root
349 *
350 * @throws IOException
351 */
352 public void updateRoot(String oldRoot, String newRoot) throws IOException {
353     //can't update to a root thats already in the tree
354     if (this.searchInTree(newRoot)) {
355         return;
356     }

```

```

357 |
358 |         this.insertRoot(newRoot);
359 |         String content = "";
360 |         this.br = new BufferedReader(new FileReader(folder + oldRoot + ".lxc"));
361 |         this.br.readLine();
362 |         String currentLine;
363 |         while ((currentLine = br.readLine()) != null) {
364 |             if (!currentLine.equalsIgnoreCase("")) {
365 |                 content += currentLine + "\n";
366 |             }
367 |         }
368 |         this.br.close();
369 |
370 |         content = content.trim();
371 |         this.writeToFile(newRoot, content);
372 |
373 |         this.deleteRoot(oldRoot);
374 |     }
375 |
376 |     /**
377 |      * Method to update morphological component from a root word
378 |      *
379 |      * @param root root to update
380 |      * @param oldComponent morphological component to update
381 |      * @param newComponent new updated morphological component
382 |      *
383 |      * @throws FileNotFoundException
384 |      * @throws IOException
385 |      */
386 |     public void updateComponent(String root, String oldComponent, String newComponent) throws FileNotFoundException, IOException {
387 |         String content = "";
388 |         this.br = new BufferedReader(new FileReader(folder + root + ".lxc"));
389 |         String currentLine;
390 |         while ((currentLine = br.readLine()) != null) {
391 |             if (currentLine.equalsIgnoreCase(oldComponent)) {
392 |                 content += newComponent + "\n";
393 |             } else if (!currentLine.equalsIgnoreCase("")) {
394 |                 content += currentLine + "\n";
395 |             }
396 |         }
397 |         this.br.close();
398 |         content = content.trim();
399 |         this.clearFile(root);
400 |         this.writeToFile(root, content);
401 |     }
402 |
403 |     /**
404 |      * Method to print all root word in the tree
405 |      *
406 |      * @return String containing all root word separated with new line char
407 |      */
408 |     public String printAllWordInTree() {
409 |         ArrayList<String> words = new ArrayList<>();
410 |         String result = "";
411 |         Node node;
412 |         String word;
413 |         int letterInt;
414 |         char letterChar;
415 |
416 |         for (int ascii = 97; ascii < 123; ascii++) {
417 |             if (roots.containsKey((char) ascii)) {
418 |                 node = roots.get((char) ascii);
419 |                 word = "" + (char) ascii;
420 |                 letterInt = 97;
421 |                 do {
422 |                     letterChar = (char) letterInt;
423 |                     if (node.children.containsKey(letterChar)) {
424 |                         word += letterChar;
425 |                         node = node.children.get(letterChar);
426 |                         letterInt = 97;
427 |                         if (node.endOfWord) {
428 |                             words.add(word);
429 |                         }
430 |                     } else {
431 |                         if (letterInt < 122) {
432 |                             letterInt++;
433 |                         } else {
434 |                             char prevLetter = word.charAt(word.length() - 1);
435 |                             letterInt = (int) prevLetter;
436 |                             letterInt++;
437 |                             word = word.substring(0, word.length() - 1);
438 |                             node = node.parent;
439 |                         }
440 |                     }
441 |                 } while (node != null);
442 |             }
443 |         }
444 |
445 |         for (int i = 0; i < words.size(); i++) {
446 |             result += words.get(i) + "\n";
447 |         }
448 |         result = result.trim();
449 |         return result;
450 |     }
451 |
452 |     public String convertToWord(String rootWord, String component) {
453 |         return combiner.convertToWord(rootWord, component);
454 |     }
455 | }

```

Listing B.3: Node.java

```

1 package Lexicon.Model;
2
3 import java.util.HashMap;
4
5 /**
6  *
7  * @author Andreas Novian
8  */
9 public class Node {
10
11     public Boolean endOfWord;
12     public HashMap<Character, Node> children;
13     public Node parent;
14
15     public Node() {
16         this.endOfWord = false;
17         this.children = new HashMap<>();
18     }
19 }

```

Listing B.4: Combiner.java

```

1 /*
2  * To change this license header, choose License Headers in Project Properties.
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6 package Lexicon.Model;
7
8 import java.io.IOException;
9
10 /**
11  *
12  * @author Andreas Novian
13  */
14 public class Combiner {
15
16     private String rootWord;
17
18     public Combiner() throws IOException {
19         this.rootWord = "";
20     }
21
22     public String convertToWord(String rootWord, String component) {
23         this.rootWord = rootWord;
24         String result = rootWord;
25
26         if (!component.equalsIgnoreCase("")) {
27             String[] comp = component.split("\\\\+");
28             char symbol;
29
30             for (String i : comp) {
31                 symbol = i.charAt(0);
32                 i = i.substring(1);
33
34                 switch (symbol) {
35                     case '@':
36                         result += " " + i;
37                         break;
38                     case '^':
39                         result = duplikasi(result, i);
40                         break;
41                     case '[':
42                         result = prefiksasi(result, i);
43                         break;
44                     case ']':
45                         result = sufiksasi(result, i);
46                         break;
47                     case '#':
48                         result = konfiksasi(result, i);
49                         break;
50                     default:
51                         break;
52                 }
53             }
54         }
55
56         return result;
57     }
58
59     private String prefiksasi(String rootWord, String prefiks) {
60         String result = rootWord;
61         char c1 = rootWord.charAt(0);
62         char c2 = rootWord.charAt(1);
63
64         //to determine whether rootWord is one syllable or not
65         boolean oneSyl = false;
66         int countSyl = 0;
67         char temp;
68         for (int i = 0; i < rootWord.length(); i++) {
69             temp = rootWord.charAt(i);
70             if (temp == 'a' || temp == 'i' || temp == 'u' || temp == 'e' || temp == 'o') {
71                 countSyl++;
72             }
73         }
74         if (countSyl == 1) {
75             oneSyl = true;
76         }
77     }
78 }

```

```

76 |     }
77 |
78 |     if (prefiks.equalsIgnoreCase("me")) {
79 |         switch (c1) {
80 |             case 'b':
81 |             case 'f':
82 |             case 'v':
83 |                 result = "mem" + rootWord;
84 |                 break;
85 |             case 'c':
86 |             case 'd':
87 |             case 'j':
88 |                 result = "men" + rootWord;
89 |                 break;
90 |             case 's':
91 |                 if (c2 == 'y') {
92 |                     result = "men" + rootWord;
93 |                 } else {
94 |                     if (result.equalsIgnoreCase(this.rootWord)) {
95 |                         result = "meny" + rootWord.substring(1);
96 |                     } else {
97 |                         result = "meny" + rootWord;
98 |                     }
99 |                 }
100 |                 break;
101 |             case 'g':
102 |             case 'h':
103 |             case 'a':
104 |             case 'i':
105 |             case 'u':
106 |             case 'e':
107 |             case 'o':
108 |             case 'q':
109 |                 result = "meng" + rootWord;
110 |                 break;
111 |             case 'k':
112 |                 if (c2 == 'h') {
113 |                     result = "meng" + rootWord;
114 |                 } else {
115 |                     if (result.equalsIgnoreCase(this.rootWord)) {
116 |                         result = "meng" + rootWord.substring(1);
117 |                     } else {
118 |                         result = "meng" + rootWord;
119 |                     }
120 |                 }
121 |                 break;
122 |             case 'p':
123 |                 if (result.equalsIgnoreCase(this.rootWord)) {
124 |                     result = "mem" + rootWord.substring(1);
125 |                 } else {
126 |                     result = "mem" + rootWord;
127 |                 }
128 |
129 |                 if (rootWord.equalsIgnoreCase("punya") || rootWord.equalsIgnoreCase("perkara")) {
130 |                     result = "mem" + rootWord;
131 |                 }
132 |                 break;
133 |             case 't':
134 |                 if (result.equalsIgnoreCase(this.rootWord)) {
135 |                     result = "men" + rootWord.substring(1);
136 |                 } else {
137 |                     result = "men" + rootWord;
138 |                 }
139 |                 break;
140 |             default:
141 |                 result = "me" + rootWord;
142 |                 break;
143 |         }
144 |         if (oneSyl) {
145 |             result = "menge" + rootWord;
146 |         }
147 |     } else if (prefiks.equalsIgnoreCase("pe")) {
148 |         switch (c1) {
149 |             case 'b':
150 |             case 'f':
151 |             case 'v':
152 |                 result = "pem" + rootWord;
153 |                 break;
154 |             case 'c':
155 |             case 'd':
156 |             case 'j':
157 |                 result = "pen" + rootWord;
158 |                 break;
159 |             case 's':
160 |                 if (c2 == 'y') {
161 |                     result = "pen" + rootWord;
162 |                 } else {
163 |                     if (result.equalsIgnoreCase(this.rootWord)) {
164 |                         result = "peny" + rootWord.substring(1);
165 |                     } else {
166 |                         result = "peny" + rootWord;
167 |                     }
168 |                 }
169 |                 break;
170 |             case 'g':
171 |             case 'h':
172 |             case 'a':
173 |             case 'i':
174 |             case 'u':

```

```

175     case 'e':
176     case 'o':
177     case 'q':
178         result = "peng" + rootWord;
179         break;
180     case 'k':
181         if (c2 == 'h') {
182             result = "peng" + rootWord;
183         } else {
184             if (result.equalsIgnoreCase(this.rootWord)) {
185                 result = "peng" + rootWord.substring(1);
186             } else {
187                 result = "peng" + rootWord;
188             }
189         }
190         break;
191     case 'p':
192         if (result.equalsIgnoreCase(this.rootWord)) {
193             result = "pem" + rootWord.substring(1);
194         } else {
195             result = "pem" + rootWord;
196         }
197         break;
198     case 't':
199         if (result.equalsIgnoreCase(this.rootWord)) {
200             result = "pen" + rootWord.substring(1);
201         } else {
202             result = "pen" + rootWord;
203         }
204         break;
205     default:
206         result = "pe" + rootWord;
207         break;
208 }
209 if (oneSyl) {
210     result = "penge" + rootWord;
211 }
212 } else if (prefiks.equalsIgnoreCase("per")) {
213     String syl = rootWord.substring(0, 3);
214     boolean erSyl = syl.substring(1).equalsIgnoreCase("er");
215     switch (c1) {
216         case 'r':
217             result = "pe" + rootWord;
218             break;
219         default:
220             result = "per" + rootWord;
221             break;
222     }
223     if (rootWord.equalsIgnoreCase("ajar")) {
224         result = "pelajar";
225     }
226     if (erSyl) {
227         result = "pe" + rootWord;
228     }
229 } else if (prefiks.equalsIgnoreCase("ber")) {
230     String syl = rootWord.substring(0, 3);
231     boolean erSyl = syl.substring(1).equalsIgnoreCase("er");
232     switch (c1) {
233         case 'r':
234             result = "be" + rootWord;
235             break;
236         default:
237             result = "ber" + rootWord;
238             break;
239     }
240     if (rootWord.equalsIgnoreCase("ajar")) {
241         result = "belajar";
242     }
243     if (erSyl) {
244         result = "be" + rootWord;
245     }
246 } else if (prefiks.equalsIgnoreCase("ter")) {
247     switch (c1) {
248         case 'r':
249             result = "te" + rootWord;
250             break;
251         default:
252             result = "ter" + rootWord;
253             break;
254     }
255 } else if (prefiks.equalsIgnoreCase("memper")) {
256     String syl = rootWord.substring(0, 3);
257     boolean erSyl = syl.substring(1).equalsIgnoreCase("er");
258
259     if (erSyl) {
260         result = "mempe" + rootWord;
261     } else {
262         result = "memper" + rootWord;
263     }
264 } else if (prefiks.equalsIgnoreCase("diper")) {
265     String syl = rootWord.substring(0, 3);
266     boolean erSyl = syl.substring(1).equalsIgnoreCase("er");
267
268     if (erSyl) {
269         result = "dipe" + rootWord;
270     } else {
271         result = "diper" + rootWord;
272     }
273 }

```



```
274 |  
275 |     //for prefix other than specified above  
276 |     if (result.equalsIgnoreCase(rootWord)) {  
277 |         result = prefiks + rootWord;  
278 |     }  
279 |  
280 |     return result;  
281 | }  
282 |  
283 | private String sufiksasi(String rootWord, String sufiks) {  
284 |     if (rootWord.contains("_")) {  
285 |         rootWord = rootWord.replace("_", "");  
286 |     }  
287 |  
288 |     String result = rootWord;  
289 |     result = result + sufiks;  
290 |  
291 |     return result;  
292 | }  
293 |  
294 |  
295 | private String konfiksasi(String rootWord, String konfiks) {  
296 |     String[] comp = konfiks.split("-");  
297 |  
298 |     if (rootWord.contains("_")) {  
299 |         rootWord = rootWord.replace("_", "");  
300 |     }  
301 |  
302 |     String result = prefiksasi(rootWord, comp[0]);  
303 |     result = sufiksasi(result, comp[1]);  
304 |  
305 |     return result;  
306 | }  
307 |  
308 | private String duplikasi(String rootWord, String duplikasi) {  
309 |     String result = rootWord;  
310 |  
311 |     if (duplikasi.equalsIgnoreCase("2")) {  
312 |         result = result + "-" + rootWord;  
313 |     } else {  
314 |         result = result + "-" + duplikasi;  
315 |     }  
316 |  
317 |     return result;  
318 | }  
319 | }
```