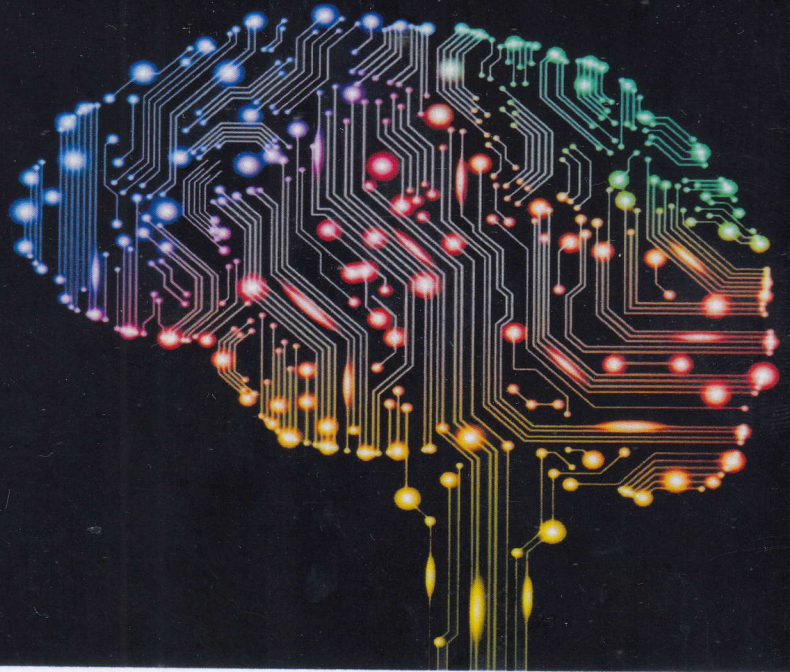


Dasar-Dasar Kecerdasan Buatan dengan Menggunakan Prolog



**Opim Salim Sitompul
Erna Budhiarti Nababan**



**Fakultas Ilmu Komputer dan Teknologi Informasi
Universitas Sumatera Utara
2017**

Daftar Isi

	Hal.
Daftar Isi	ii
 BAB I Pengenalan Prolog	 1
1.1. Pengenalan Prolog	1
1.2. <i>Propotional Logic</i>	3
1.3. <i>Predicative Logic</i>	3
1.4. Klausa dalam Prolog	4
1.5. Jenis-Jenis Klausa dalam Prolog	5
1.6. Konjungsi	8
1.7. Latihan	11
 BAB II Elemen-Elemen Prolog	 13
2.1. Objek-Objek Data	13
2.2. Atom	13
2.3. Variabel	14
2.4. Struktur	14
2.5. Operator	15
2.6. <i>Matching</i>	17
2.7. Operator Buatan Pemakai	19
2.8. Operator Buatan Pemakai	20
 BAB III Elemen-Elemen Prolog	 22
3.1. Rekursi	22
3.2. Contoh	22
3.2.1. Faktorial	22

3.2.2. Bilangan Fibonacci	24
3.2.3. <i>Database</i> Genealogi	26
3.3. <i>Backtracking</i> dan <i>Cut</i>	29
3.4. Latihan	33
 BAB IV Struktur Pohon (<i>Tree</i>)	 35
4.1. <i>Tree</i>	35
4.2. Pencarian dalam <i>Tree</i>	36
4.2.1. Pencarian <i>predecessor</i>	36
4.2.2. Pencarian <i>successor</i>	41
4.2.3. Pencarian <i>leaf</i>	41
4.2.4. Pencarian <i>root</i>	42
4.2.5. Pencarian kedalaman <i>node</i>	43
4.2.6. Pencarian <i>path</i>	44
4.3. Latihan	45
 BAB V Struktur <i>List</i>	 47
5.1. <i>List</i>	47
5.2. Manipulasi <i>List</i>	47
5.3. Operasi pada <i>List</i>	48
5.3.1. Penambahan <i>item</i> ke <i>list</i>	48
5.3.2. Pencarian <i>item</i> dalam <i>list</i>	51
5.2.3. Penghapusan <i>item</i> dari <i>list</i>	52
5.3.4. Penggabungan <i>list</i>	53
5.3.5. <i>Sublist</i>	55
5.4. Penggunaan <i>List</i> sebagai Struktur Data	57

5.5. Latihan	61
 BAB VI Representasi Pengetahuan	 63
6.1. Representasi Pengetahuan	63
6.2. <i>If-Then Rules</i>	63
6.3. <i>Backward Chaining</i>	65
6.4. Menggunakan Operator Buatan Pengguna	66
6.5. <i>Forward Chaining</i>	68
6.6. <i>Reasoning</i>	71
6.7. Latihan	72
 BAB VII <i>Semantic Network</i> dan <i>Frame</i>	 73
7.1. Pendahuluan	73
7.2. <i>Semantic Network</i>	73
7.3. <i>Frame</i>	76
7.4. Latihan	79
 BAB VIII Sistem Pakar	 81
8.1. Sistem Pakar	81
8.2. Komponen Sistem Pakar	81
8.3. Alur Kerja Sistem Pakar	82
8.4. <i>Decision Tree</i>	83
8.5. <i>Certainty Factor</i>	84
8.6. Contoh	88
8.7. Latihan	91
 Daftar Pustaka	 92
Lampiran	iv

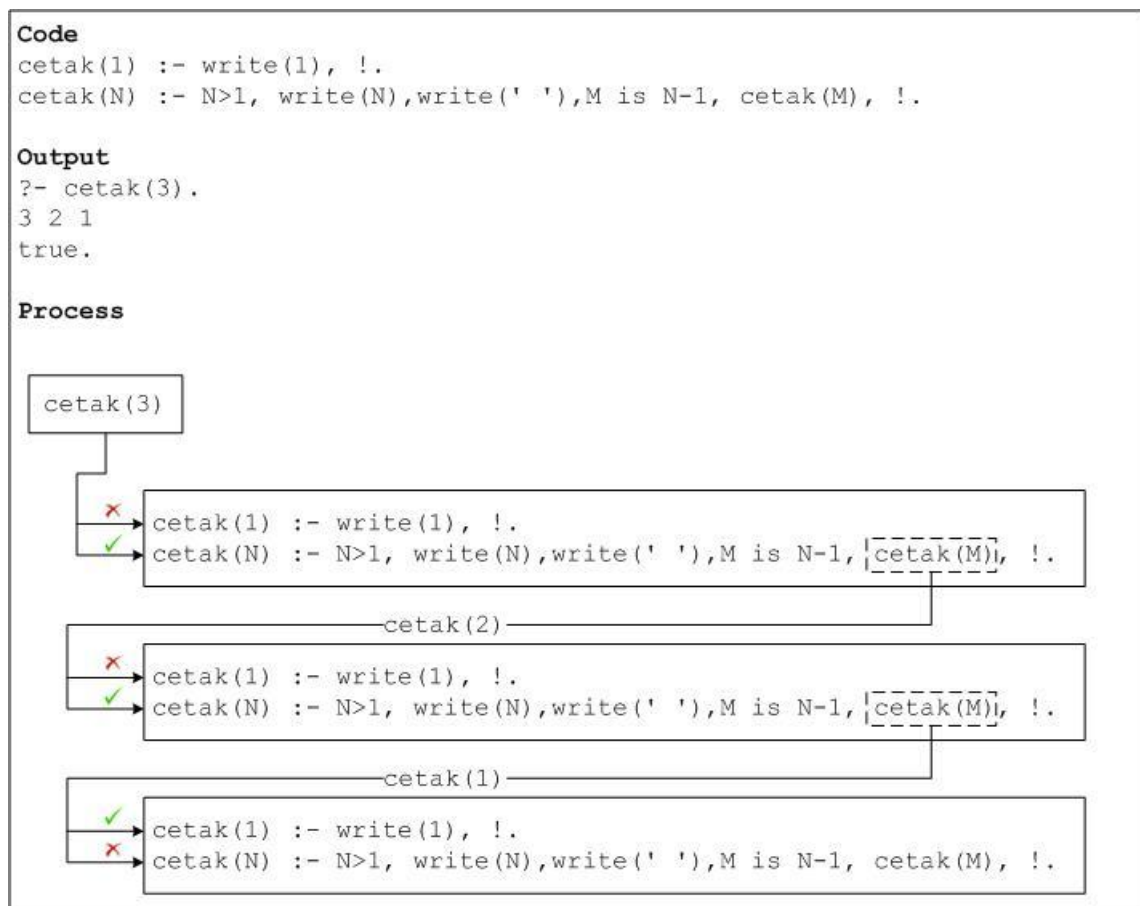
Bab I

Pengenalan Prolog

1.1 Pengenalan Prolog

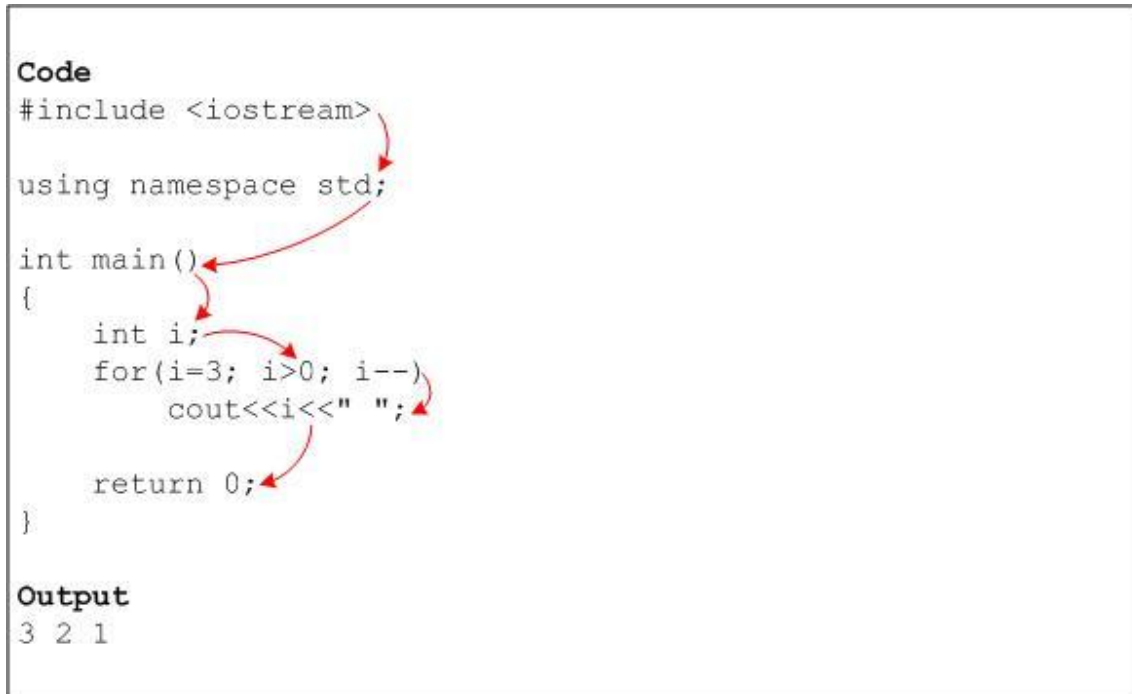
Prolog atau *Programming in Logic* adalah pemrograman simbolik non-numerik yang dipergunakan untuk pemrograman kecerdasan buatan.

Contoh pelaksanaan instruksi secara deklaratif dalam Prolog seperti pada Gambar 1.1.



Gambar 1.1 Pelaksanaan instruksi secara deklaratif dalam Prolog

Contoh pelaksanaan instruksi secara procedural dalam bahasa pemrograman lain seperti pada Gambar 1.2.



Gambar 1.2 Pelaksanaan instruksi secara prosedural

Tabel 1.1 Perbandingan Prolog dengan bahasa pemrograman lain

Parameter	Prolog	Bahasa Pemrograman Lain (misalnya C / C++)
Gaya pemrograman yang didukung	<ul style="list-style-type: none"> • Prosedural • Deklaratif 	<ul style="list-style-type: none"> • Prosedural
Konsep pemrograman	<ul style="list-style-type: none"> • Cara memperoleh keluaran. • Evaluasi relasi. • “Apa yang benar” • “Apa yang perlu dilakukan” 	<ul style="list-style-type: none"> • Melaksanakan instruksi per instruksi secara bertahap untuk memperoleh sebuah keluaran. • “Bagaimana melakukannya” • “Apa yang selanjutnya dilakukan”

1.2 Propositional Logic

Proposition adalah sebuah pernyataan yang merupakan sebuah kalimat deklaratif. Setiap proposition dapat bernilai benar atau salah. *Propositional logic* merupakan suatu susunan argumen yang valid, khususnya yang menyangkut aturan-aturan yang membuktikan suatu kesimpulan dari sekumpulan premis atau asumsi.

Contoh *propositional logic*:

P : X rajin belajar

Q : X akan menjadi pandai

Premis 1	$P \rightarrow Q$	Jika X rajin belajar maka X akan menjadi pandai
Premis 2	P	X rajin belajar
<hr/>		
Conclusion	$\therefore Q$	X akan menjadi pandai

1.3 Predicative Logic

Predicative logic atau *first-order logic* merupakan bentuk generalisasi dari *propositional logic*.

Contoh *predicative logic*:

$Px \rightarrow Qx$

$x = \text{subject} / \text{term}$

$P, Q = \text{predicate}$

Sebuah kalimat “Semua orang yang rajin belajar akan menjadi pandai” memiliki pengertian yang sama dengan “Untuk semua x , jika x rajin belajar maka x akan menjadi pandai”. Kalimat tersebut dapat ditulis dalam bentuk *predicative logic*:

$\forall x[Px \rightarrow Qx]$

dengan :

P = rajin belajar

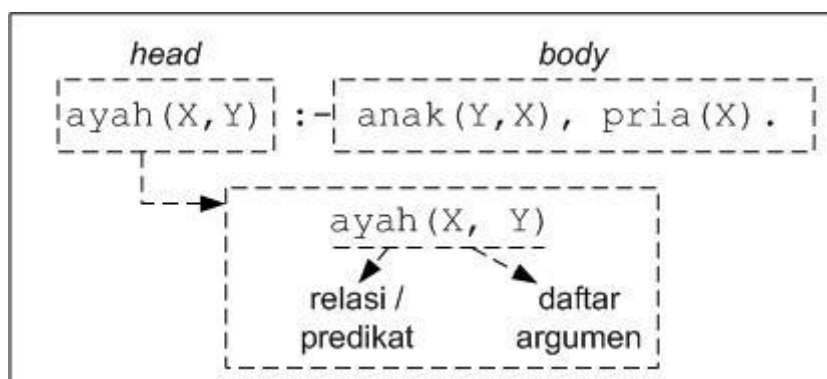
Q = akan menjadi pandai

1.4 Klausa dalam Prolog

Sebuah klausa Prolog terdiri dari *head* dan *body*. *Body* merupakan kumpulan *goal* yang dipisahkan oleh tanda koma. :

- Relasi yang terletak di depan tanda kurung atau disebut juga sebagai predikat.
- Objek-objek yang terdapat di dalam tanda kurung atau disebut juga sebagai argumen. Argumen dapat berupa objek konkrit atau konstanta (*atom*) atau objek umum (*variabel*).

Struktur sebuah klausa Prolog dapat dilihat seperti pada Gambar 1.3.



Gambar 1.3 Struktur sebuah klausa dalam Prolog

Bagian *head* dan *body* dari sebuah klausa dalam Prolog dipisahkan oleh tanda implikasi *logical* (`:-`) yang dapat diartikan sebagai “jika”. Klausa tersebut dapat dibaca seperti berikut.

“X (adalah) ayah Y **jika** Y (adalah) anak X **dan** X (adalah) pria.”

Beberapa aturan penulisan klausa dalam Prolog:

1. Nama relasi dan atom harus diawali dengan huruf kecil.

`suka(joko, tuti).` % Benar

`suka(Joko, Tuti).` % Salah

Pelanggaran pada aturan ini akan menyebabkan kesalahan seperti berikut:


```
?- suka(Joko, Tuti).
Joko = _ \
Tuti = _
```

Kesalahan tersebut terjadi karena argumen yang diberikan merupakan variabel. *Fact* dalam Prolog memuat argumen berupa atom sedangkan *rules* dalam Prolog dapat memuat argumen berupa atom maupun variabel.

2. Penulisan argumen berupa variabel harus diawali dengan huruf kapital.
3. Penulisan sebuah klausa diawali dengan relasi kemudian dilanjutkan dengan penulisan objek-objeknya dalam tanda kurung yang dipisahkan dengan koma.
4. Setiap klausa dalam Prolog diakhiri dengan tanda titik.

Kumpulan *fact* dan *rule* disimpan dalam *database* atau sering juga disebut *knowledge base*.

1.5 Jenis-Jenis Klausa dalam Prolog

Dalam Prolog, terdapat 3 jenis klausa:

- *Fact* (fakta) adalah ekspresi predikatif yang membuat pernyataan deklaratif tentang domain masalah. *Fact* memiliki *head* namun tidak memiliki *body*. Contoh *fact* dalam Prolog :

```
anak(budi, ani).      % "Budi (adalah) anak Ani".
wanita(ani).          % "Ani (adalah) wanita".
```

- *Rule* (aturan) adalah ekspresi predikatif yang menggunakan implikasi *logical* (:-) untuk mendeskripsikan relasi antara *fact*. Sebuah *rule* memiliki *head* dan *body*. Contoh *rule* dalam Prolog:

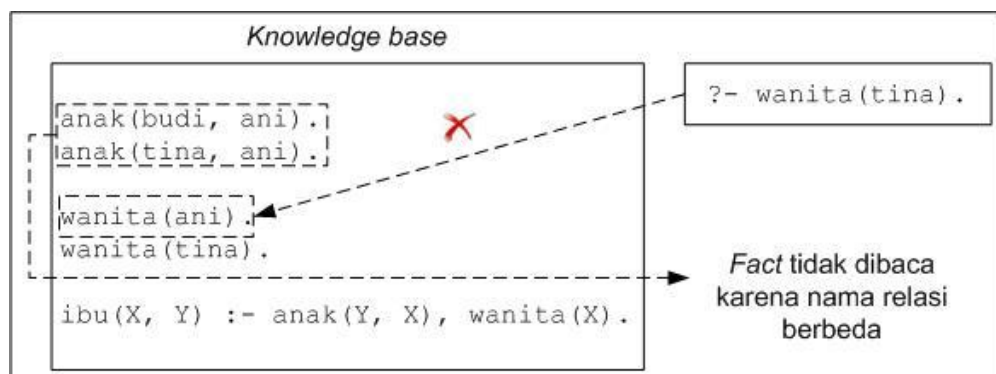
```
ibu(X, Y) :- anak(Y, X), wanita(X).
% "X (adalah) ibu Y jika Y (adalah) anak X dan X
(adalah) wanita"
```

- *Query* merupakan pertanyaan yang diajukan kepada Prolog. *Interpreter* Prolog kemudian akan memberikan jawaban berdasarkan *rules* dan *facts* dalam *database*

(*knowledge base*). *Knowledge base* diasumsikan benar untuk domain masalah tertentu. Contoh *query* dalam Prolog:

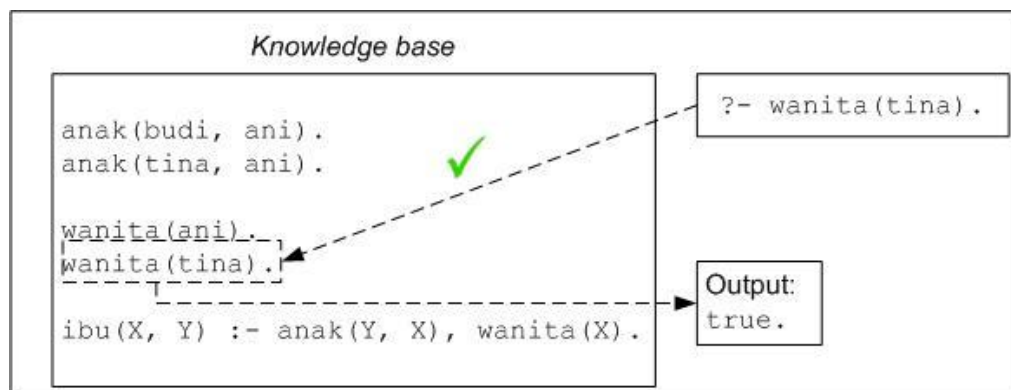
1. `?- wanita(tina).` %"Apakah Tina wanita?"
`true.`

Pada saat sebuah *query* diberikan, Prolog akan mencari apakah *query* tersebut terdapat di dalam *knowledge base* seperti tampak pada Gambar 1.4a. *Query* akan dicocokkan dengan *fact* maupun *rule* sesuai dengan nama relasi dan argumennya.



Gambar 1.4a Pencocokkan *fact* (1).

Jika *query* yang diberikan belum sesuai dengan *fact* atau *rule* yang ada, Prolog kemudian akan menelusuri *fact* dan *rule* yang ada sampai *query* yang diinginkan ditemukan seperti pada Gambar 1.4b.



Gambar 1.4b Pencocokkan *fact* (2).

Jika *query* yang diinginkan ditemukan, maka Prolog akan memberikan jawaban "*true*" atau "*yes*". Jika sampai akhir *knowledge base* masih belum

ditemukan *fact* atau *rule* yang sesuai dengan *query*, maka Prolog akan memberikan jawaban “*false*” atau “*no*”.

2. ?- anak(X,Y) .

X = budi,

Y = ani ;

X = tina,

Y = ani.

Prolog akan mencari semua *fact* dalam *knowledge base* yang memiliki predikat anak dan dua buah argumen. Apabila ditemukan relasi dengan nama dan jumlah argumen yang sesuai, maka nilai argumen tersebut akan dikembalikan. Apabila nama relasi sama, namun jumlah argumen berbeda, maka klausa tersebut tidak akan dibaca.

Contoh: anak(rika, 3, ani). %Rika anak ketiga Ani

3. ?- ibu(X, budi) .

X = ani. %"Siapa ibu Budi?"

Query yang diberikan merupakan *query* menggunakan *rule*:

ibu(X, Y) :- anak(Y, X), wanita(X) .

Prolog kemudian akan mencari *goal-goal* yang harus dipenuhi untuk menjawab *query* tersebut. Langkah-langkah pencarian *goal-goal* yang harus dipenuhi adalah sebagai berikut:

1. Pertama, variabel dan atom yang terdapat dalam *query* akan disesuaikan dengan *rule* yang ada.

ibu(X, budi) :- anak(budi, X), wanita(X) .

2. Selanjutnya, Prolog akan mencoba memenuhi *goal 1* dengan mencari *fact* dalam *knowledge base* yang memenuhi bentuk: anak(budi, X). *Fact* yang memenuhi bentuk tersebut adalah anak(budi, ani). Dengan demikian *goal 1* dipenuhi dengan nilai X = ani.
3. Prolog kemudian akan mencoba memenuhi *goal 2* yang sebelumnya telah diinstansiasi dengan nilai X = ani. Prolog akan mencoba mencari apakah

terdapat *fact* `wanita(ani)`. *Fact* tersebut ditemukan dan dengan demikian *goal* 2 dipenuhi.

4. Karena *goal* 1 dan *goal* 2 dipenuhi, maka Prolog akan memberikan jawaban `X = ani`.

1.6 Konjungsi

Konjungsi dipergunakan untuk memberikan *query-query* yang lebih kompleks. Misalnya terdapat *knowledge base* sebagai berikut:

```
anak(budi, ani).
anak(tina, ani).
anak(budi, harry).
anak(tina, harry).
```

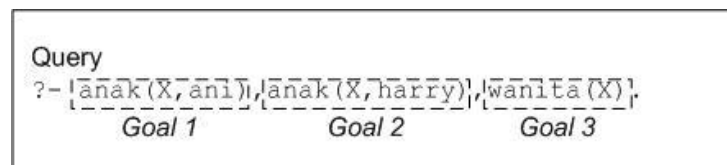
```
pria(budi).
pria(harry).
wanita(tina).
wanita(ani).
```

Apabila kita memberikan pertanyaan kepada Prolog untuk mencari “Siapa anak perempuan dari Ani dan Harry?” dengan *query*:

```
?- anak(X,ani), anak(X,harry), wanita(X).
```

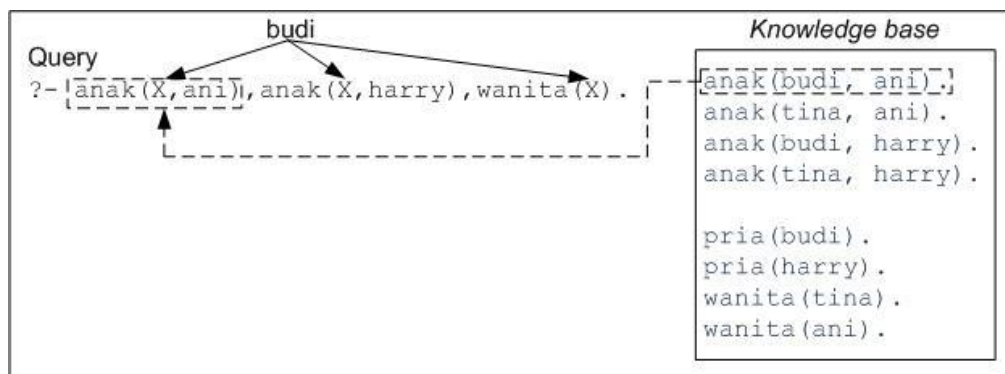
maka Prolog akan memberikan jawaban `X = tina`. Contoh tersebut menggunakan konsep *backtracking* seperti berikut:

1. Prolog akan membagi *query* yang diberikan menjadi beberapa *goal* seperti pada Gambar 1.5a.



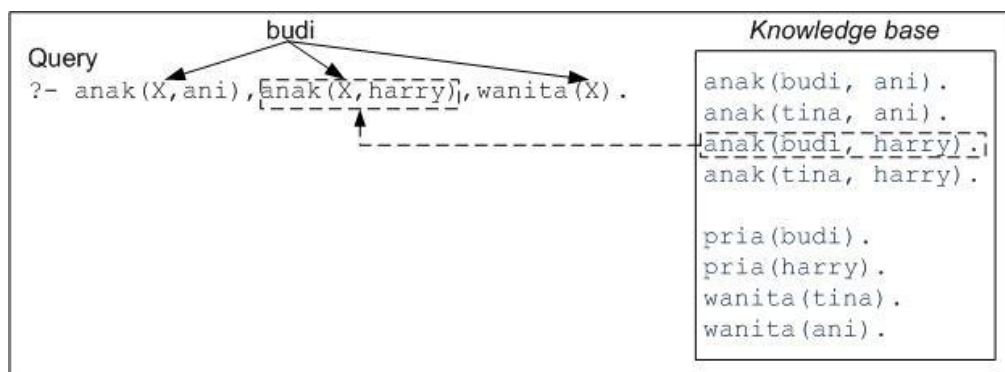
Gambar 1.5a Pembagian goal dari query.

Selanjutnya Prolog akan menelusuri *fact* dan *rule* yang ada untuk mencoba memenuhi *goal-goal* tersebut. Pada saat Prolog menemukan *fact* yang sesuai dengan *query* yang diberikan, Prolog akan meninstansiasi variabel yang ada dengan nilai argumen yang terdapat pada *fact* tersebut seperti pada Gambar 1.5b. Dengan demikian nilai variabel X akan diinstansiasi dengan budi.



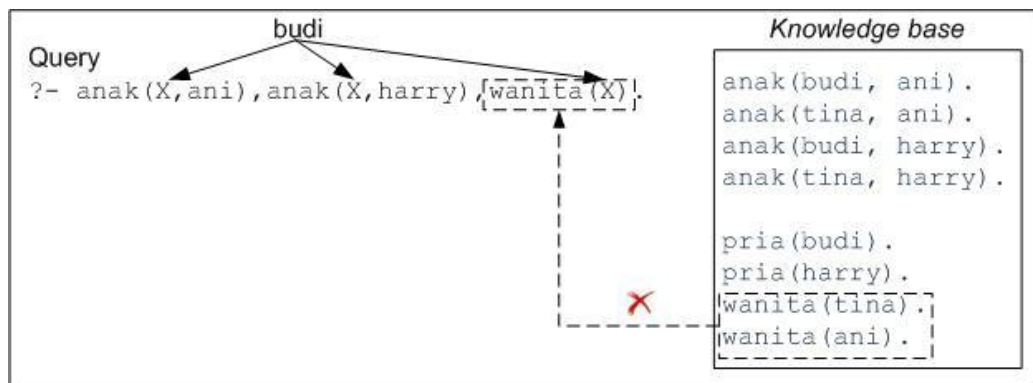
Gambar 1.5b Goal 1 berhasil.

2. *Goal 2* juga berhasil dipenuhi seperti tampak pada Gambar 1.5c.



Gambar 1.5c Goal 2 berhasil.

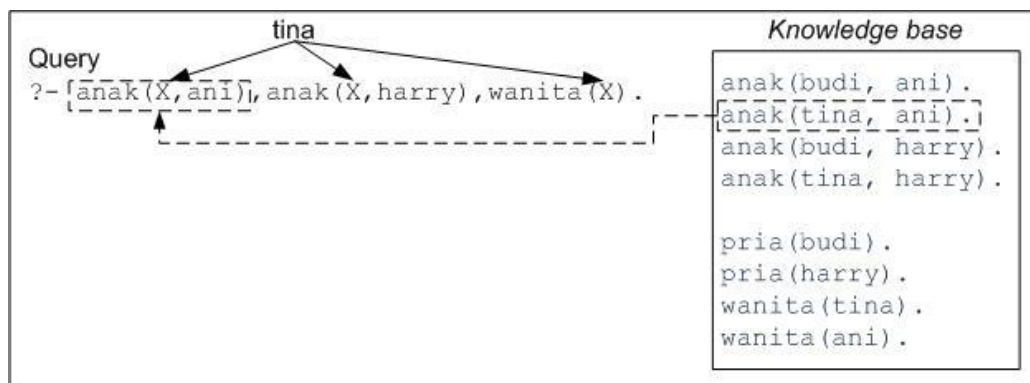
3. Pada Gambar 1.5d terlihat bahwa *goal 3* gagal dipenuhi karena tidak terdapat *fact* wanita(budi) dalam *knowledge base*.



Gambar 1.5d Goal 3 gagal.

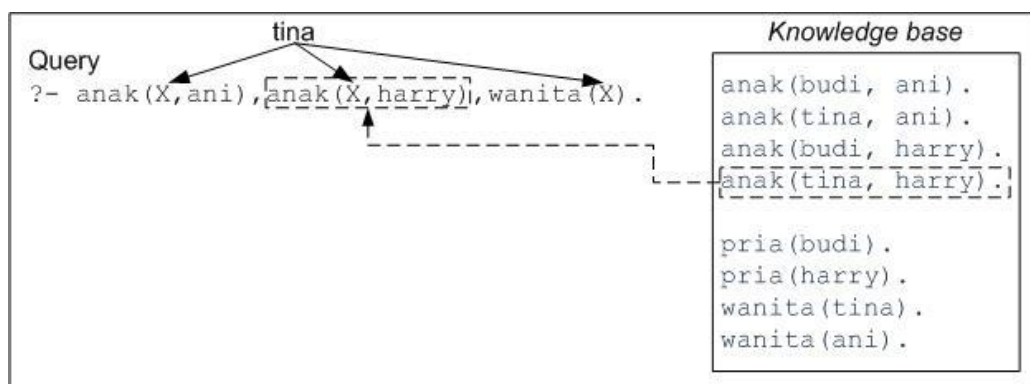
Dengan demikian, Prolog akan melakukan proses *backtracking*. Lupakan nilai *X* dan kembali berupaya memenuhi *goal 1*.

4. *Goal 1* berhasil dipenuhi dan variabel *X* diinstansiasi dengan *tina* seperti pada Gambar 1.5e.



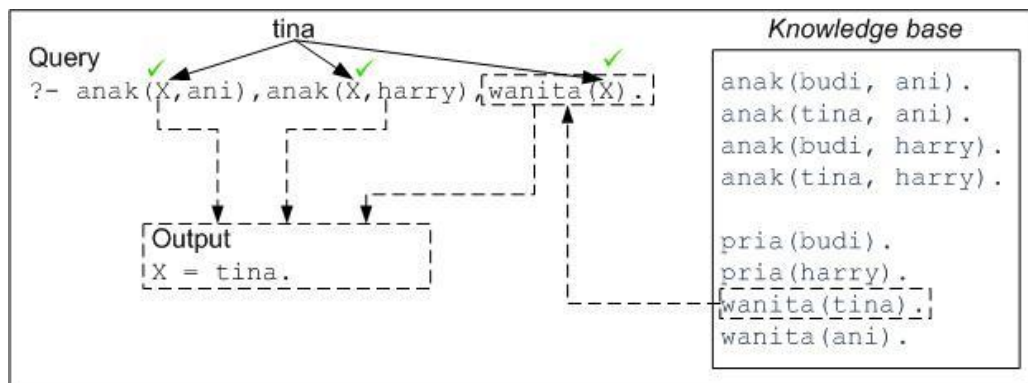
Gambar 1.5e Goal 1 berhasil.

5. *Goal 2* berhasil dipenuhi seperti pada Gambar 1.5f.



Gambar 1.5f Goal 2 berhasil.

6. *Goal 3* juga berhasil dipenuhi seperti pada Gambar 1.5g.

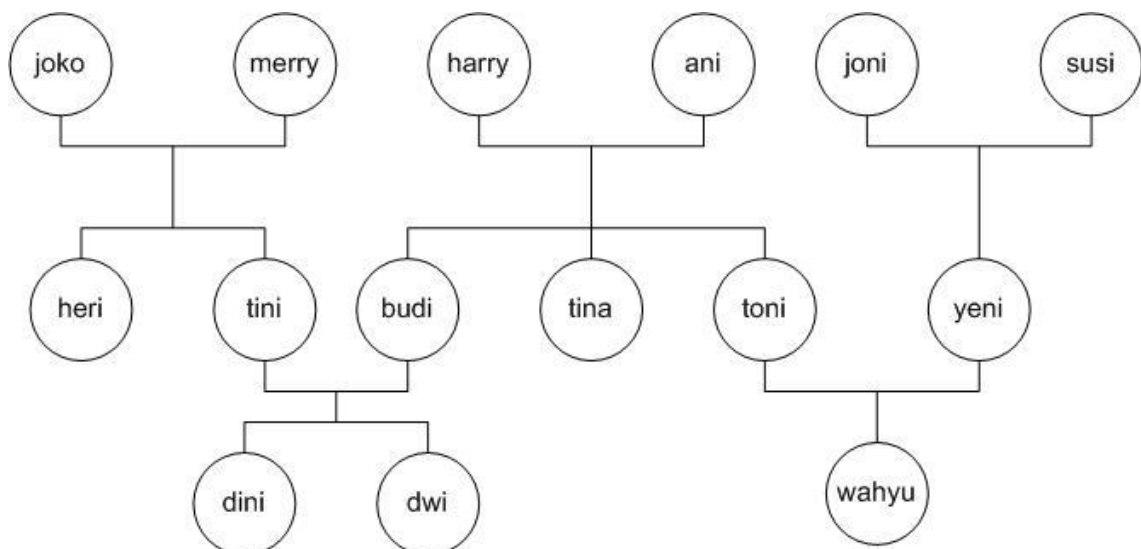


Gambar 1.5g. Goal 3 berhasil.

Karena semua *goal* telah berhasil dipenuhi, Prolog akan memberikan jawaban `X = tina.`

1.7 Latihan

- Tentukan makna, jenis klausa, relasi, serta argumen dan jenis argumen dari klausa-klausa Prolog berikut:
 - `beli(X, Y) :- harga(X, N), N < 50000, Y < 5.`
 - `punya(tuti, sepeda).`
 - `orang('Harry', 'Susanto', tanggal_lahir(21, 3, 1986), bekerja('PT. Maju Sejahtera', 4500000)).`
 - `?- orang(A, B, C, D).`
- Buatlah sebuah *knowledge base* berisi *fact* dari pohon keluarga berikut:



Anda hanya boleh menggunakan predikat anak, pria, dan wanita dalam *knowledge base* tersebut.

Catatan:

- Joko, Harry, Joni, Heri, Budi, Toni, Dwi, dan Wahyu adalah pria.
 - Merry, Ani, Susi, Tini, Tina, Yeni, dan Dini adalah wanita.
3. Lengkapilah *rule* berikut lalu tambahkan ke dalam *knowledge base* pada soal nomor 2 sehingga *rule* tersebut dapat dipergunakan untuk menentukan panggilan seseorang (Contoh: Jika X merupakan anak Y, Z merupakan saudara Y, dan Z merupakan pria, maka Z adalah paman X).

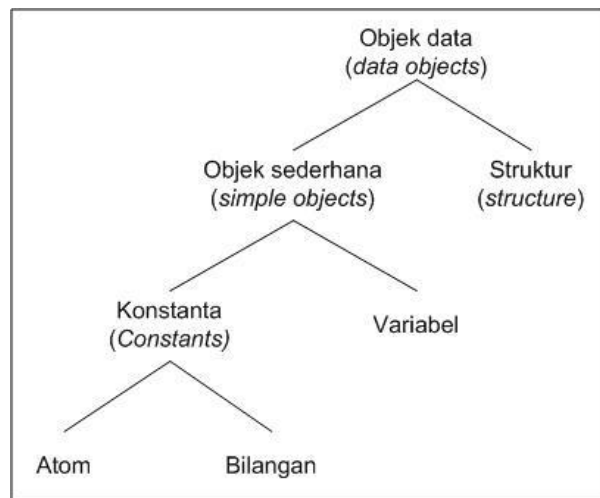
- | | |
|-----------------------------|----------------------------|
| a) ibu(X, Y) | % X ibu dari Y |
| b) ayah(X, Y) | % X ayah dari Y |
| c) seorang_ibu(X) | % X seorang ibu |
| d) seorang_ayah(X) | % X seorang ayah |
| e) seorang_anaklelaki(X) | % X seorang anak lelaki |
| f) seorang_anakperempuan(X) | % X seorang anak perempuan |
| g) adik_perempuan(X, Y) | % X adik perempuan Y |
| h) abang(X, Y) | % X abang Y |
| i) kakek(X, Y) | % X kakek Y |
| j) nenek(X, Y) | % X nenek Y |
| k) sepupu(X, Y) | % X sepupu Y |
| l) ipar_perempuan(X, Y) | % X adik ipar perempuan Y |
| m) bibi(X, Y) | % X bibi Y |

Bab II

Elemen-Elemen Prolog

2.1 Objek-Objek Data

Klasifikasi objek dalam Prolog dapat dilihat seperti pada Gambar 2.1.



Gambar 2.1 Klasifikasi objek Prolog

2.2 Atom

Atom adalah objek dalam Prolog yang memiliki nilai berupa konstanta. Atom dapat dibentuk dengan beberapa cara:

1. Karakter yang ditulis dengan diawali huruf kecil, digit, dan karakter *underscore*('_'). Contoh:

```
hari      budi      rumahsakit      barang_berharga      a25
a_25      a_
```

2. Untaian karakter khusus. Contoh:

```
<-->      ==>      ...      ...      ::=
```

3. Atom dapat juga berupa untaian karakter (*string*) yang ditulis dengan diapit tanda kutip tunggal. Contoh:

```
'Lorem ipsum dolor sit ame'      'Indonesia'
```

4. Atom juga dapat berupa sebuah *list* kosong: []

2.3 Variabel

Variabel adalah objek dalam Prolog berupa *string* yang dapat berisi angka, *underscore*, huruf, dan harus diawali dengan huruf besar atau karakter *underscore*. Contoh variabel:

```
X      Siapa      _2      _21_abc
```

Variabel dapat juga berupa tanda *underscore* tunggal ('_') yang menyatakan variabel *anonymous* yang dapat bernilai apa saja. Variabel ini umumnya dipergunakan untuk mengetahui apakah di dalam *knowledge base* terdapat *fact* atau *rule* dengan argumen tertentu tanpa memperdulikan nilai argumen tersebut.

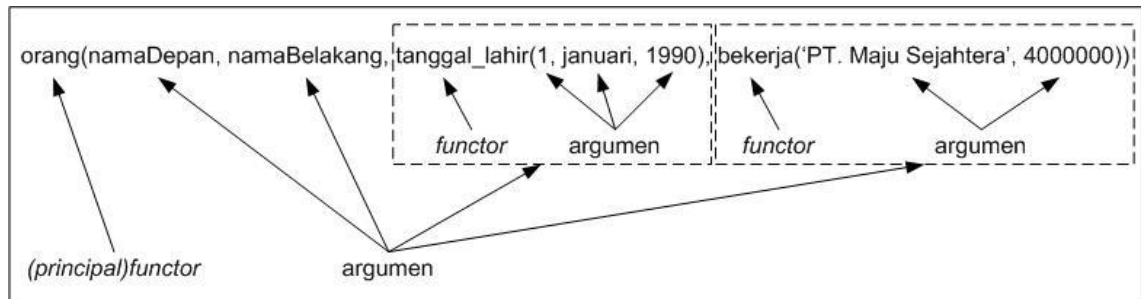
```
seorang_ibu(X) :- anak(_, X), wanita(X).
```

Rule tersebut menggunakan variable *anonymous* dan memiliki arti “X (adalah) ibu jika X memiliki anak dan X (adalah) wanita.” Pada *rule* tersebut kita tidak perlu mengetahui siapa anak X, melainkan kita hanya perlu mengetahui apakah terdapat *fact* atau *rule* yang menunjukkan bahwa X memiliki seorang anak.

2.4 Struktur

Struktur adalah sebuah objek tunggal yang terdiri dari objek-objek lain. Struktur juga disebut *compound terms* atau *complex terms*. Untuk mengkombinasikan komponen-komponen ke dalam satu objek tunggal kita harus memilih sebuah *functor*. Dalam Prolog, *functor* adalah elemen syntaksis yang dipergunakan untuk membangun sebuah struktur (*compound terms*) dari *terms* yang lebih sederhana. Gambar 2.2 menunjukkan penggunaan *functor* dalam membangun sebuah struktur. Dalam gambar tersebut juga terlihat

sebuah *principal functor* atau dikenal juga dengan sebutan *top-level functor* yang dipergunakan sebagai *functor* utama dalam membangun struktur.



Gambar 2.2 Contoh *functor* dan *principal functor*

2.5 Operator

Operator merupakan tanda yang dipergunakan sehingga program menjadi lebih mudah dibaca. Tanpa menggunakan operator, operasi $2+3-4$ harus ditulis dalam bentuk $-(+(2,3), 4)$ yang akan mengakibatkan program menjadi lebih sulit dibaca. Operator juga berfungsi untuk mengembalikan nilai dari operasi tertentu, misalnya operasi aritmatika. Beberapa operator yang terdapat dalam Prolog adalah sebagai berikut:

+	penjumlahan
-	pengurangan
*	perkalian
/	pembagian
//	pembagian bilangan bulat (<i>integer</i>)
mod	modulo atau sisa pembagian

Selain operator aritmatika, di dalam Prolog juga terdapat sebuah operator *assignment* (penugasan) yang dinyatakan dengan *is*. Operator ini berguna untuk

mendeklarasikan dan memberikan nilai sebuah variabel. Contoh penggunaan operator *is*:

```
?- X = 2+5.
```

```
X = 2+5.
```

```
?- X is 2+5.
```

```
X = 7.
```

Dalam Prolog juga terdapat *relational* yang dipergunakan untuk membandingkan argumen. Operator relasional dalam Prolog adalah sebagai berikut:

```
>      lebih besar
```

```
<      lebih kecil
```

```
>=     lebih besar atau sama dengan
```

```
=<     lebih kecil atau sama dengan
```

```
:=     sama dengan
```

```
=\=    tidak sama dengan
```

Contoh penggunaan operator tersebut adalah:

```
?- X is 3, Y is 4, X:=Y.
```

```
false.
```

```
?- X is 1995 mod 12, X >= 10.
```

```
false.
```



```
?- 200 * 5 =< 1000.
```

```
true.
```

Prolog juga memiliki operator logika yang sebagai berikut:

```
,      dan
;      atau
:-     hanya jika
not    tidak
```

Baik operator relasional dan operator logika dalam Prolog berfungsi untuk membandingkan argumen-argumen yang diberikan kemudian mengembalikan nilai berupa *boolean* (*true* atau *false*).

2.6 Matching

Prolog menerapkan operasi *matching* untuk menginstansiasi sebuah *term*. Dua buah *term* sesuai (*match*) apabila keduanya sama(setara) atau *term* tersebut memiliki variabel dapat diinstansiasi sehingga menghasilkan *term* yang setara atau sama. Beberapa syarat agar objek dapat *match* adalah:

1. Jika dua buah objek yaitu X dan Y adalah konstanta, maka X dan Y *match* jika dan hanya jika keduanya merupakan objek yang sama.
2. Jika X merupakan variabel dan Y merupakan objek sembarang, maka X dan Y *match* dan X diinstansiasi ke Y. Demikian pula sebaliknya, jika Y merupakan variabel, maka Y diinstansiasi ke X. Jika X dan Y merupakan variabel, maka keduanya akan diinstansiasi ke satu sama lain atau dengan kata lain memiliki nilai yang sama.
3. Jika X dan Y merupakan struktur maka keduanya *match* jika dan hanya jika:
 - a. Keduanya memiliki *principal functor* yang sama,
 - b. Semua argumen yang bersangkutan sesuai, dan

- c. Instansiasi variabel harus sesuai. Jika X telah diinstansiasi dengan sebuah nilai, variabel tersebut tidak mungkin diinstansiasi dengan nilai lain.

Contoh *match* adalah sebagai berikut:

```
?- orang>NamaDepan, hariyanto , tanggal_lahir(D1,
'Januari', Y1), kerja('PT. Maju Sejahtera', 50000)) =
orang(X, Y, tanggal_lahir(D2, M2, 1990), Z).

NamaDepan = X,

D1 = D2,

Y1 = 1990,

Y = hariyanto,

M2 = 'Januari',

Z = kerja('PT. Maju Sejahtera', 50000).
```

Pada contoh tersebut, terdapat sebuah *principal functor* yaitu *orang* yang membentuk sebuah struktur. Di dalam struktur tersebut terdapat 2 *functor* lain yaitu *tanggal_lahir* dan *kerja*. Karena *principal functor* yang dimiliki oleh kedua *term* sama, argumen-argumen yang dimiliki kedua *term* tersebut akan diinstansiasi. Instansiasi yang terjadi adalah:

- NamaDepan dan X saling diinstansiasi satu sama lain karena keduanya merupakan variabel. Keduanya memiliki nilai yang sama.
- D1 dan D2 saling diinstansiasi satu sama lain.
- Y1 diinstansiasi ke atom bernilai 1990.
- Y diinstansiasi ke atom bernilai hariyanto.
- M2 diinstansiasi ke atom bernilai 'Januari'.
- Z diinstansiasi ke klausa bernilai kerja('PT. Maju Sejahtera', 50000).

Dalam *matching*, instansiasi dapat dilakukan tidak hanya kepada atom atau variabel, namun dapat juga dilakukan kepada klausa.

2.7 Operator Buatan Pemakai

Dalam Prolog, pengguna dapat mendefinisikan sendiri suatu operator. Sebuah definisi operator harus dibuat sebelum adanya ekspresi lain. Struktur penulisan operator buatan pengguna adalah:

```
:- op(NilaiPresedensi, JenisOperator, NamaOperator).
```

Presedensi adalah nilai di antara 1-1200. Terdapat tiga jenis operator yaitu:

1. Operator *infix*: $x f x$ $x f y$ $y f x$
2. Operator *prefix*: $f x$ $f y$
3. Operator *postfix*: $x f$ $y f$

f menyatakan *functor* atau nama operator yang dipergunakan. x menyatakan argumen yang presedensinya harus lebih rendah daripada presedensi operator. y menyatakan argumen yang presedensinya lebih rendah atau sama dengan presedensi operator.

Pendefinisian operator buatan pengguna tidak menunjukkan operasi atau tindakan apapun, melainkan hanya menentukan bagaimana penggunaan operator tersebut secara sintaksis. Sebuah fakta yang sebelumnya ditulis sebagai berikut:

```
makan(monyet, pisang).
```

dapat ditulis menggunakan operator buatan pengguna menjadi:

```
:- op(600, xfx, makan).
```

```
monyet makan pisang.
```

Dengan menggunakan operator buatan pengguna, kita dapat menanyakan *query*:

```
monyet makan Apa.
```

dan memperoleh jawaban `Apa = pisang`.

Operator dengan nilai presedensi paling tinggi akan menjadi *principal functor*. Sebagai contoh, terdapat definisi operator sebagai berikut:

$:- \text{op}(800, \text{xfx}, <==>) .$

$:- \text{op}(700, \text{xfx}, \vee) .$

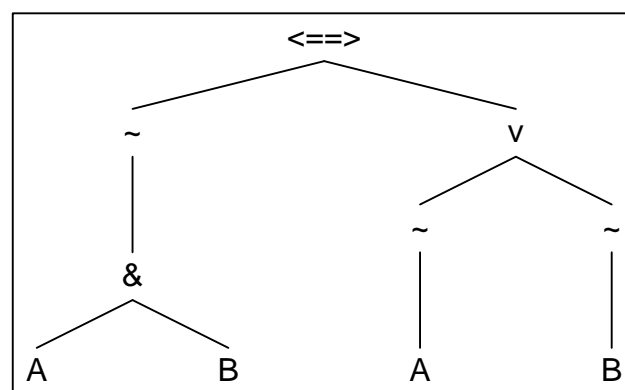
$:- \text{op}(600, \text{xfx}, \&) .$

$:- \text{op}(500, \text{xfx}, \sim) .$

Definisi operator tersebut dapat dipergunakan untuk menyatakan fakta logika:

$\sim (A \& B) <==> \sim A \vee \sim B .$

Operator $<==>$ memiliki nilai presedensi tertinggi yaitu 800 dan dengan demikian, operator tersebut akan menjadi *principal functor* seperti pada Gambar 2.3.



Gambar 2.3 Presedensi operator

2.8 Latihan

1. Beberapa makhluk hidup memiliki ciri-ciri sebagai berikut:

Monyet:

- Monyet **makan** pisang
- Monyet **melahirkan**
- Monyet **berumur** 45 tahun

Kura-kura:

- Kura-kura **makan** rumput laut
- Kura-kura **bertelur**
- Kura-kura **berumur** 70 tahun

Burung pelikan:

- Burung pelikan **makan** ikan
- Burung pelikan **bertelur**
- Burung pelikan **berumur** 25 tahun

Lainnya:

- Pisang (adalah) **tumbuhan**
- Rumput laut (adalah) **tumbuhan**
- Ikan (adalah) **hewan**

Selanjutnya buatlah *facts* dan *rules* untuk menjawab *query-query* berikut:

- monyet makan Apa.
- Apa makan ikan.
- Apa melahirkan.
- HewanApa herbivora. %herbivora adalah hewan yang memakan tumbuhan
- monyet berumur X.
- Apa hidup_paling_lama. %binatang dengan umur paling lama
- Apa tumbuhan.

2. Buatlah *rules* untuk:

a) Melakukan operasi aritmatika sederhana. Contoh:

- `?- arith('+', 2, 3, Z) .`
`Z = 5.`
- `?- arith('/', 2, 3, Z) .`
`Z = 0.6666666666666666.`
- `?- arith('-', 2, 3, Z) .`
`Z = -1.`
- `?- arith('*', 2, 3, Z) .`
`Z = 6.`

- Menghitung nilai mutlak dari sebuah bilangan.
- Menghitung nilai maksimal dari dua bilangan.

d) Menghitung nilai median dari tiga bilangan.

Bab III

Rekursi

3.1 Rekursi

Rekursi adalah sebuah pendefinisian sesuatu dalam bentuk pemanggilan dirinya sendiri. Syarat-syarat yang diperlukan dalam rekursi adalah:

1. Terdapat *base case* atau *terminal* yaitu bagian yang menentukan akhir dari rekursi.
2. Terdapat *recursive case* yaitu bagian yang memanggil dirinya sendiri. Pada saat rekursi terjadi, terjadi penyederhaan struktur pemanggilan di mana struktur pemanggilan berikutnya harus lebih sederhana dari yang sebelumnya.

Struktur rekursi yang memanggil dirinya sendiri dapat dipergunakan untuk melakukan perulangan perintah tertentu. Proses rekursi akan menyederhanakan sampai *base case* lalu selanjutnya mundur kembali ke proses sebelumnya.

3.2 Contoh

3.2.1 Faktorial

Perhitungan faktorial merupakan salah satu contoh kasus yang dapat diselesaikan dengan menggunakan rekursi. Sebuah bilangan N memiliki nilai faktorial yang dapat dihitung dengan rumus:

$$N! = N * (N-1) * (N-2) * \dots * 2 * 1$$

dengan nilai dari $0! = 1$ dan $1! = 1$.

Secara rekursif, nilai faktorial sebuah bilangan dapat dihitung dengan cara:

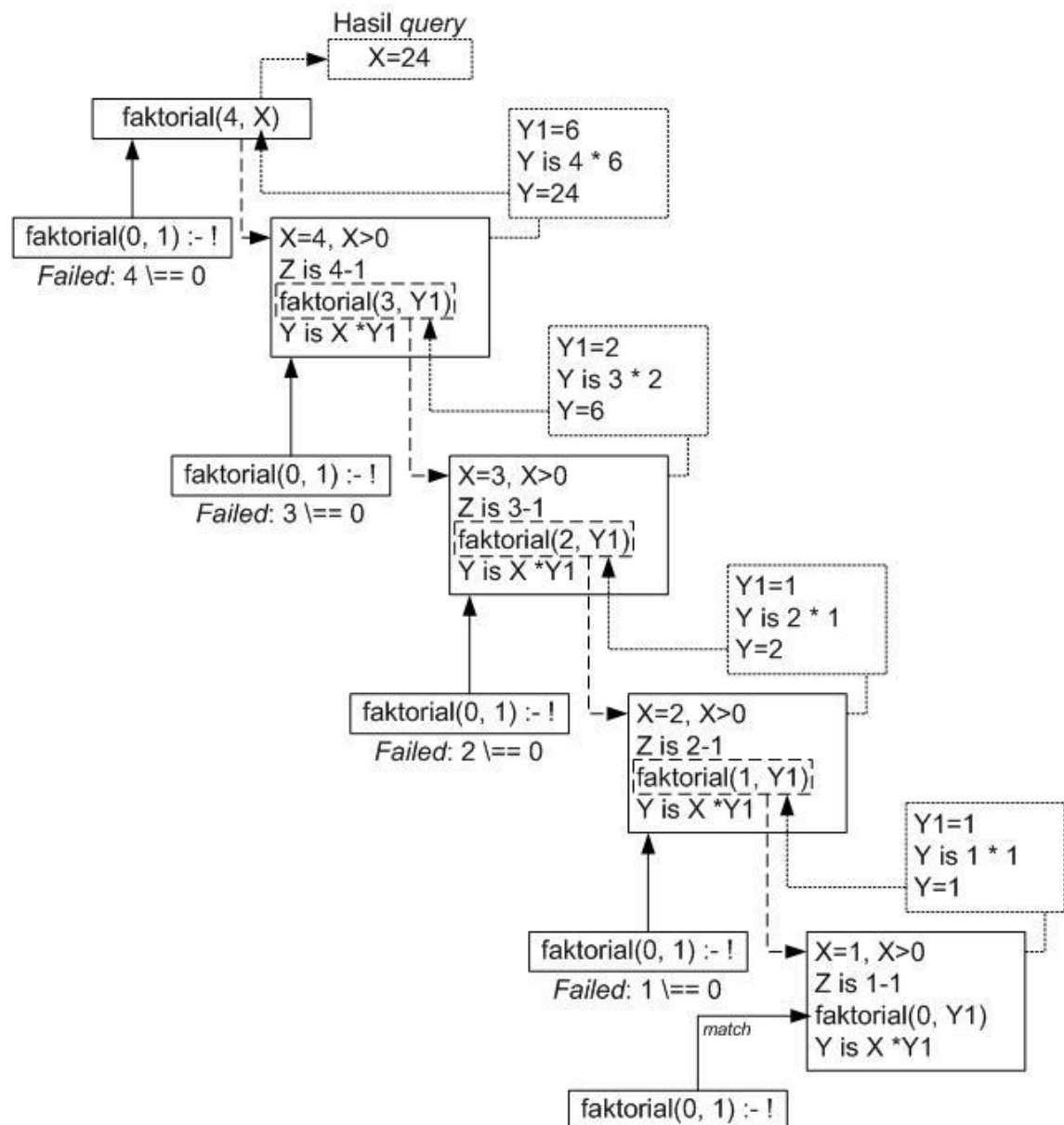
$$N! = N * (N - 1) !$$

Dalam Prolog, rumus tersebut dapat diimplementasi dengan *rules*:

```
faktorial(0,1) :- !.
```

```
faktorial(X,Y) :- X > 0, Z is X-1, faktorial(Z, Y1), Y
is X * Y1.
```

Selanjutnya, apabila kita menanyakan sebuah *query*: `?- faktorial(4, X).` maka kita akan memperoleh jawaban `X=24`. Proses rekursif untuk perhitungan faktorial berlangsung seperti pada Gambar 3.1.



Gambar 3.1 Perhitungan faktorial secara rekursif

Rule pertama merupakan *base case* yang menunjukkan bahwa perhitungan faktorial akan berhenti ketika sampai pada perhitungan $0!$. *Rule* kedua merupakan proses rekursif dengan *goal* pertama yang harus dipenuhi adalah apabila X (bilangan yang diberikan) memiliki nilai lebih besar dari 0. Jika *goal* tersebut dipenuhi, nilai X akan dikurangi dengan 1 dan disimpan di variabel Z . Pemanggilan rekursi selanjutnya menggunakan variabel Z sebagai argumen. Hasil setiap perhitungan akan disimpan di $Y1$ lalu kemudian dikalikan dengan bilangan semula (X) dan diberikan kepada B untuk dikembalikan sebagai jawaban.

3.2.2 Bilangan Fibonacci

Bilangan Fibonacci adalah bilangan yang memiliki syarat:

$$f(0) = 1$$

$$f(1) = 1$$

$$f(n) = f(n-1) + f(n-2), \quad n \geq 2$$

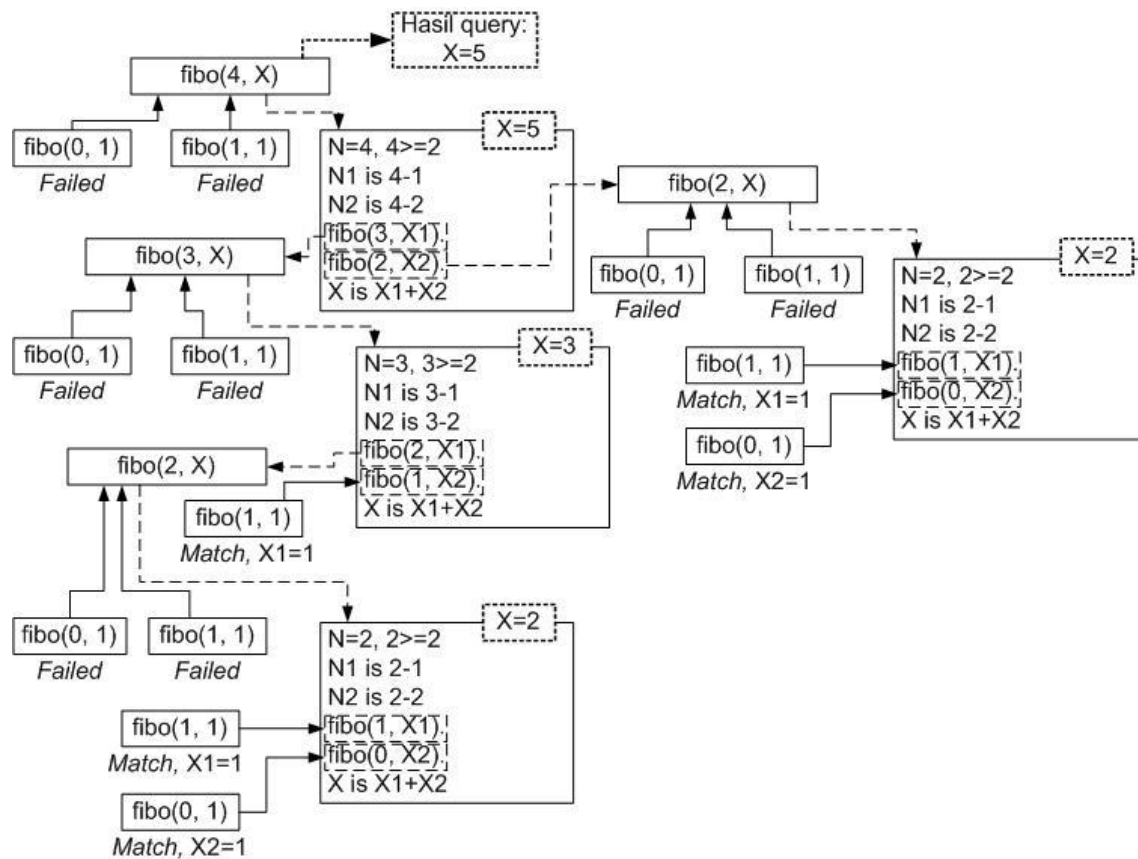
Dengan demikian, perhitungan suku ke- n bilangan Fibonacci dapat diimplementasi dengan *rules*:

```
fibonacci(0, 1) :- !.
```

```
fibonacci(1, 1) :- !.
```

```
fibonacci(N, X) :- N >= 2, N1 is N-1, N2 is N-2, fibonacci(N1, X1), fibonacci(N2, X2), X is X1 + X2.
```

Rule pertama dan kedua merupakan *base case* yang akan mengembalikan nilai 1 jika menerima $n=1$ atau $n=0$. *Rule* ketiga merupakan proses rekursif untuk $n \geq 2$. Dalam rekursif ini, dilakukan pemanggilan rekursif sebanyak 2 kali dan dikenal dengan nama *doubly recursive*. Dengan demikian, apabila kita memberikan *query* `?-fibonacci(4, X)`, kita akan memperoleh jawaban $X=5$. Proses rekursif ketiga *rules* tersebut seperti pada Gambar 3.2.



Gambar 3.2 Pencarian suku ke-n deret Fibonacci secara rekursif

Ketiga *rules* tersebut dipergunakan untuk menentukan suku ke-n deret Fibonacci. Kita dapat menambahkan *rules* untuk mencetak deret Fibonacci sampai suku ke-n. *Rules* yang ditambahkan adalah:

```
deret_fibo(N) :- print_fibo(0, N).

print_fibo(N,N) :- !, fibo(N, X), write(X).

print_fibo(I,N) :- I < N, I1 is I+1, fibo(I, X),
write(X), write(' '), print_fibo(I1, N).
```

Rule pertama merupakan *rule* yang dipergunakan sebagai *query*. Jika *rule* tersebut dipanggil, *rule* tersebut kemudian akan memanggil *rule* `print_fibo` dengan mengirimkan 0 dan N sebagai argumen. Pengiriman nilai 0 sebagai argumen berfungsi seperti bagian inisialisasi pada struktur perulangan. *Rule* kedua merupakan *base case* untuk mengakhiri rekursi jika perulangan telah sampai pada suku ke-n. *Rule* ketiga

merupakan bagian rekursi untuk nilai $I < N$. Jika *goal* tersebut dipenuhi, maka *rule* tersebut akan meminta nilai suku ke- i deret Fibonacci dan mencetaknya kemudian melakukan pemanggilan rekursi berikutnya.

3.2.3 Database Genealogi

Dalam *database* Genealogi, terdapat dua relasi yaitu *ancestor* dan *descendant*. *Ancestor* memiliki dua definisi yaitu:

1. Orangtua (*parent*).
2. (Secara rekursif) orangtua dari *ancestor* (*parent of ancestor*).

Berdasarkan kedua definisi tersebut, kita dapat membuat dua buah *rules* yaitu:

```
%base case

ancestor(X, Y) :- parent(X, Y).

%recursive case

ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Apabila kita memiliki *fact*:

```
parent(andi, budi).
parent(andi, ani).
parent(budi, joko).
parent(joko, susi).
```

Dengan menggunakan kedua *rules* tersebut, kita dapat memperoleh relasi *ancestor*:

```
?- ancestor(X, Y).

X = andi,
```

Y = budi ;

X = andi,

Y = ani ;

X = budi,

Y = joko ;

X = joko,

Y = susi ;

X = andi,

Y = joko ;

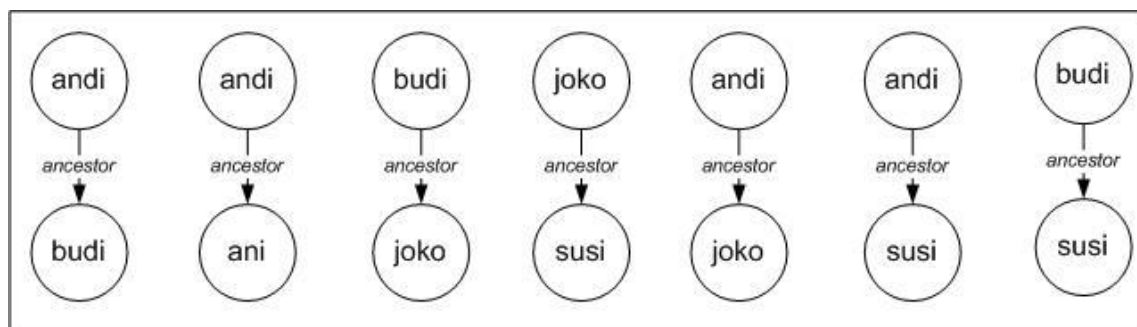
X = andi,

Y = susi ;

X = budi,

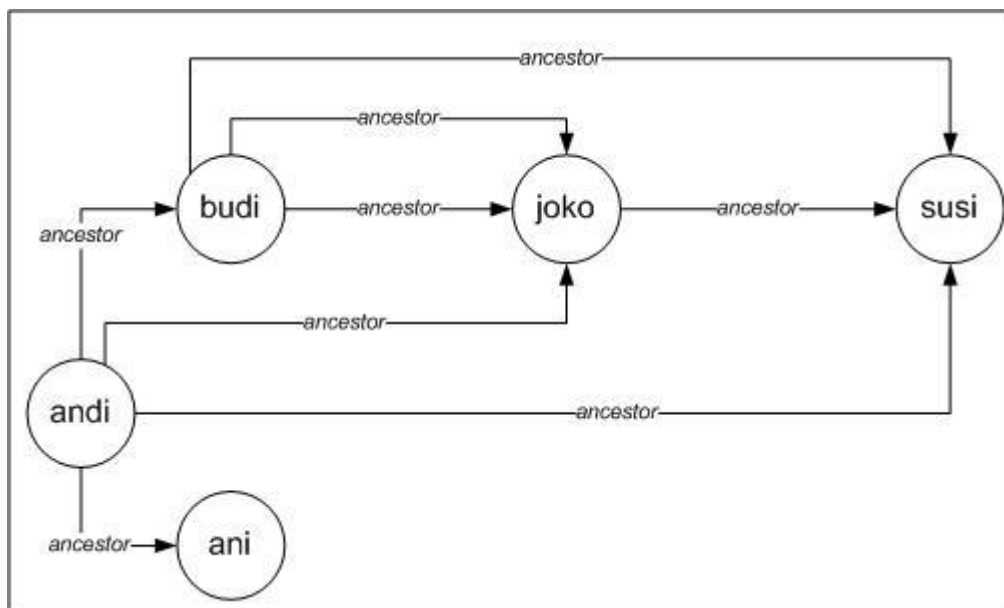
Y = susi.

Berdasarkan hasil *query*, tampak seolah-olah bahwa relasi *ancestor* hanya bersifat 1 tingkat seperti pada Gambar 3.3.



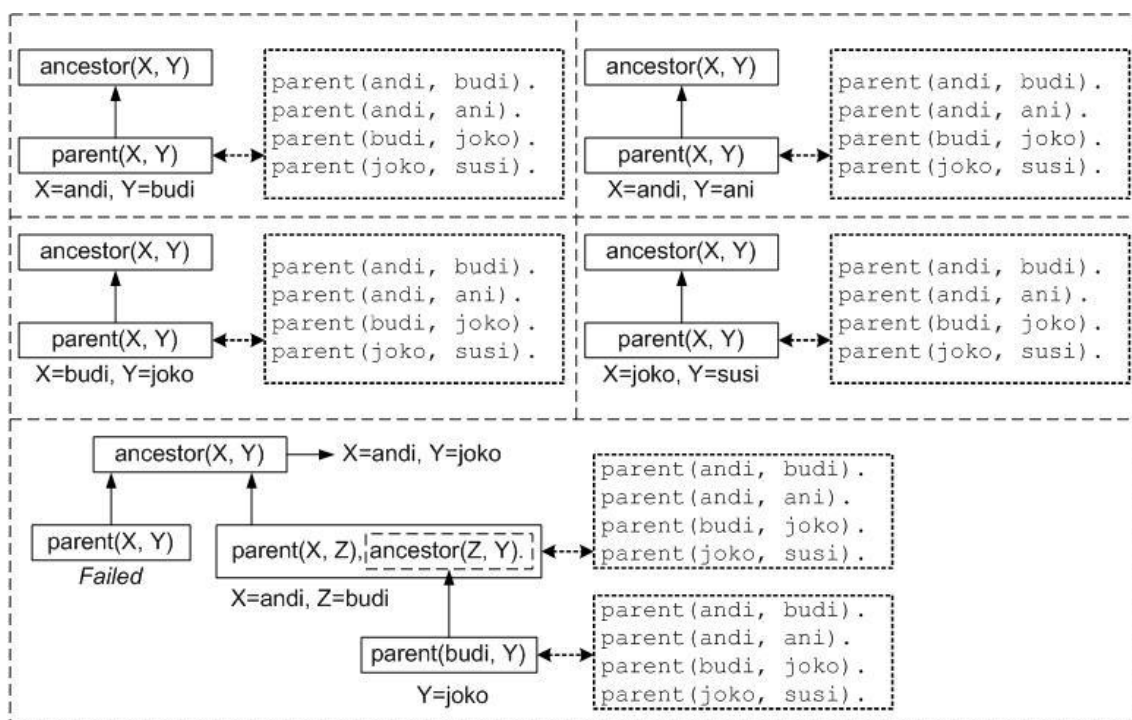
Gambar 3.3 Relasi *ancestor* 1 level

Akan tetapi, relasi *ancestor* sebenarnya bersifat lebih kompleks serta memiliki relasi langsung dan tidak langsung seperti pada Gambar 3.4.



Gambar 3.4 Relasi *ancestor* langsung dan tidak langsung

Proses eksekusi *query* untuk semua hasil relasi *ancestor* seperti pada Gambar 3.5.



Gambar 3.5a Penelusuran hasil eksekusi *query* (1).

titik sebelumnya di mana pilihan untuk *matching* dibuat. Proses *backtracking* akan mengakibatkan Prolog mungkin melupakan nilai dari beberapa argumen yang dipergunakan. Sebagai contoh, apabila kita memiliki *knowledge base* sebagai berikut:

```
orangtua(ani, joko).
```

```
orangtua(budi, joko).
```

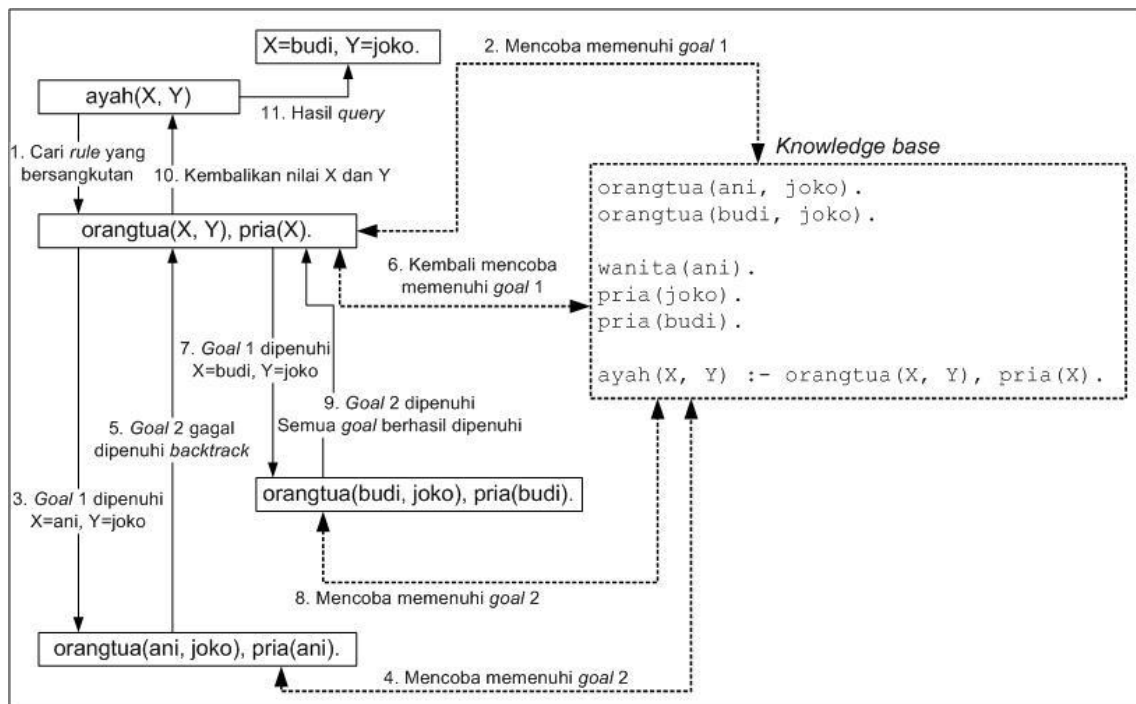
```
wanita(ani).
```

```
pria(joko).
```

```
pria(budi).
```

```
ayah(X, Y) :- orangtua(X, Y), pria(X).
```

Apabila kita memberikan *query*: `?- ayah(X, Y).` proses eksekusi *query* seperti pada Gambar 3.6.



Gambar 3.6 Proses eksekusi *query* dan *backtrack*.

Proses *backtracking* yang terjadi terlihat pada langkah 5 di mana X yang telah diinstansiasi dengan `ani` gagal memenuhi *goal* 2 yaitu `pria(ani)`. Karena *goal* 2 gagal dipenuhi, Prolog akan melakukan *backtrack* dan melupakan nilai yang telah diinstansiasi.

Operator *cut* merupakan atom yang dipergunakan untuk mengkontrol proses *backtracking* otomatis yang dimiliki Prolog. Jika Prolog menemukan operator *cut* dalam sebuah *rule* maka Prolog tidak akan melakukan *backtrack* terhadap pilihan yang telah dipilih. Dengan kata lain, apabila Prolog telah menemukan nilai yang *match* dengan *rules* atau *facts* tertentu, maka nilai tersebut akan dianggap sebagai satu-satunya nilai yang ada dan nilai lainnya akan diabaikan. Perbandingan hasil *query* antara *rule* dengan operator *cut* dengan *rule* tanpa operator *cut* dapat dilihat pada Tabel 3.1 di mana dalam tabel tersebut terdapat dua program Prolog yang berisi *facts* yang sama dan dua *rules* yang hampir sama. *Rule* pada program pertama tidak menggunakan operator *cut* sedangkan *rule* pada program kedua menggunakan operator *cut*.

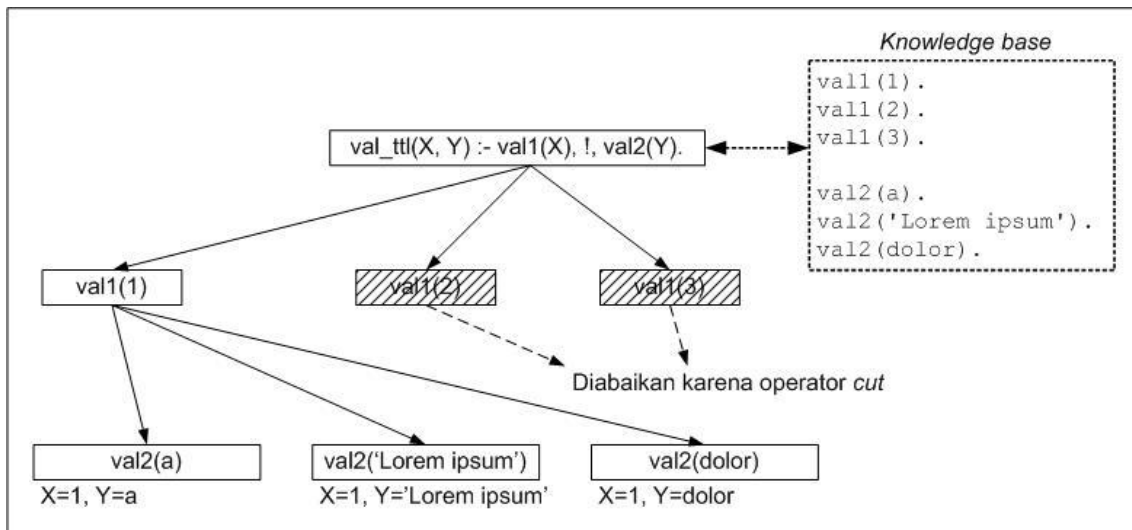
Tabel 3.1 Perbandingan hasil *query*.

Parameter	Tanpa operator <i>cut</i>	Dengan operator <i>cut</i>
<i>Facts</i>		<code>val1(1).</code> <code>val1(2).</code> <code>val1(3).</code> <code>val2(a).</code> <code>val2('Lorem ipsum').</code> <code>val2(dolor).</code>
<i>Rule</i>	<code>val_ttl(X, Y) :-</code> <code>val1(X), val2(Y).</code>	<code>val_ttl(X, Y) :-</code> <code>val1(X), !, val2(Y).</code>

Tabel 3.1 Perbandingan hasil *query* (lanjutan).

Parameter	Tanpa operator <i>cut</i>	Dengan operator <i>cut</i>
Hasil <i>Query</i>	<pre> ?- val_ttl(X,Y) . X = 1, Y = a ; X = 1, Y = 'Lorem ipsum' ; X = 1, Y = dolor ; X = 2, Y = a ; X = 2, Y = 'Lorem ipsum' ; X = 2, Y = dolor ; X = 3, Y = a ; X = 3, Y = 'Lorem ipsum' ; X = 3, Y = dolor. </pre>	<pre> ?- val_ttl(X,Y) . X = 1, Y = a ; X = 1, Y = 'Lorem ipsum' ; X = 1, Y = dolor. </pre>

Pada Tabel 3.1 terlihat bahwa terdapat hasil *query* yang cukup signifikan hanya dengan penambahan sebuah operator *cut*. Penggunaan operator *cut* mengakibatkan proses *backtracking* tidak akan dilakukan terhadap bagian *rule* *val1*(X). *Fact* pertama dengan relasi *val1* yang ditemukan yaitu *val1*(1) akan dianggap sebagai satu-satunya nilai yang ada untuk relasi *val1*. *Fact* *val1*(2) dan *val1*(3) akan diabaikan karena dalam *rule* *val_ttl* ditemukan operator *cut* setelah *val1*(X). Hal ini akan mengakibatkan nilai X yang merupakan argumen dalam relasi *val1* tidak akan mengalami proses *backtrack*. Proses eksekusi *query* dengan operator *cut* seperti diberikan pada Gambar 3.7.



Gambar 3.7 Proses eksekusi *query* dengan operator *cut*.

3.4 Latihan

1. Buatlah *rule* untuk mencetak deret bilangan sebagai berikut:

```
?- cetak(4, 3, 10).
```

```
4 7 10 13 16 19 22 25 28 31
```

Kemudian jelaskan proses rekursi yang dipergunakan untuk mencetak deret tersebut.

2. Buatlah *rule* untuk mencetak segitiga sebagai berikut:

```
?- cetak(3, '*' ).
```

```
*
```

```
**
```

```
***
```

3. Perkalian dua buah bilangan dapat diselesaikan dengan penjumlahan secara rekursif sebagai berikut:

```
4 * 3 = 3 + (3 * 3)
```

```
3 * 3 = 3 + (2 * 3)
```

```
2 * 3 = 3 + (1 * 3)
```

```
1 * 3 = 3 + (0 * 3)
```

```
0 * 3 = 0
```

(0 dikalikan dengan semua bilangan akan menghasilkan 0)

Buatlah *rule* untuk mensimulasikan perkalian dua buah bilangan dengan melakukan penjumlahan secara rekursif.

4. Buatlah *rule* untuk memeriksa apakah sebuah bilangan merupakan bilangan prima.

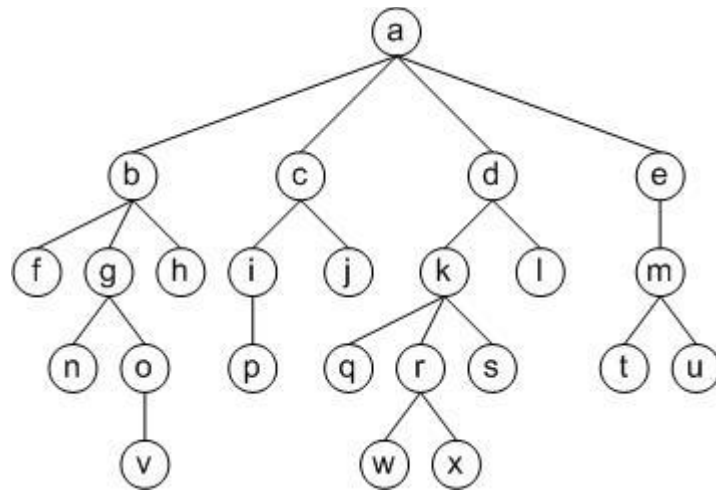
Bab IV

Struktur Pohon (*Tree*)

4.1 *Tree*

Tree adalah struktur data yang berbentuk menyerupai sebuah pohon. Struktur data ini terdiri dari kumpulan *node* dan didefinisikan secara rekursif. Pendefinisian secara rekursif menyebabkan setiap cabang dari *tree* akan merujuk pada struktur yang lain.

Terdapat sebuah pohon keturunan seperti yang diberikan pada Gambar 4.1.



Gambar 4.1 Contoh struktur *tree*

Struktur pohon pada Gambar 4.1 dapat dinyatakan dengan *facts* sebagai berikut:

<code>is_parent(a, b).</code>	<code>is_parent(b, h).</code>
<code>is_parent(a, c).</code>	<code>is_parent(c, i).</code>
<code>is_parent(a, d).</code>	<code>is_parent(c, j).</code>
<code>is_parent(a, e).</code>	<code>is_parent(d, k).</code>
<code>is_parent(b, f).</code>	<code>is_parent(d, l).</code>
<code>is_parent(b, g).</code>	<code>is_parent(e, m).</code>

<code>is_parent(g, n).</code>	<code>is_parent(m, t).</code>
<code>is_parent(g, o).</code>	<code>is_parent(m, u).</code>
<code>is_parent(i, p).</code>	<code>is_parent(o, v).</code>
<code>is_parent(k, q).</code>	<code>is_parent(r, w).</code>
<code>is_parent(k, r).</code>	<code>is_parent(r, x).</code>
<code>is_parent(k, s).</code>	

Sebuah *tree* tersusun dari beberapa *node*. Pada Gambar 4.1 terdapat sebuah *node* yang merupakan *node* awal atau dalam *tree* dikenal dengan istilah *root*. Percabangan dari sebuah *node* akan merujuk ke struktur lain secara rekursif.

4.2 Pencarian dalam Tree

4.2.1 Pencarian *predecessor*

Salah satu pencarian yang dapat diterapkan dalam *tree* adalah pencarian *predecessor*. *Predecessor* adalah semua *node* yang merupakan pendahulu dari *node* tertentu. Pencarian *predecessor* dapat dilakukan dengan menggunakan *rules*:

```
is_pred(X, Y) :- is_parent(X, Y).

is_pred(X, Y) :- is_parent(X, Z), is_pred(Z, Y).
```

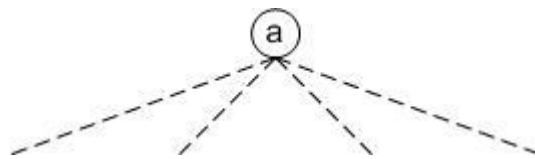
Prolog menerapkan proses pencarian *default* berupa *Depth First Search*. *Depth First Search* (DFS) adalah proses penyelesaian sebuah *tree* atau graf dengan melakukan penelusuran sampai *node* paling dalam (*leaf*) terlebih dahulu sebelum melakukan *backtrack*. Ketika kita memberikan *query* `?- is_pred(a, X)` kita akan memperoleh jawaban:

<code>X = b ;</code>	<code>X = d ;</code>	<code>X = f ;</code>	<code>X = h ;</code>
<code>X = c ;</code>	<code>X = e ;</code>	<code>X = g ;</code>	<code>X = n ;</code>

$X = o ;$	$X = p ;$	$X = r ;$	$X = m ;$
$X = v ;$	$X = k ;$	$X = s ;$	$X = t ;$
$X = i ;$	$X = l ;$	$X = w ;$	$X = u ;$
$X = j ;$	$X = q ;$	$X = x ;$	$\text{false}.$

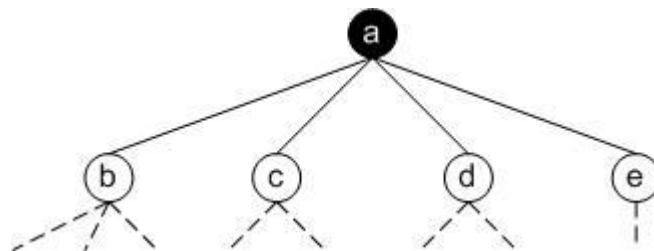
Proses penelusuran *tree* pada Gambar 4.1 berlangsung sebagai berikut:

1. Keadaan awal *tree*.



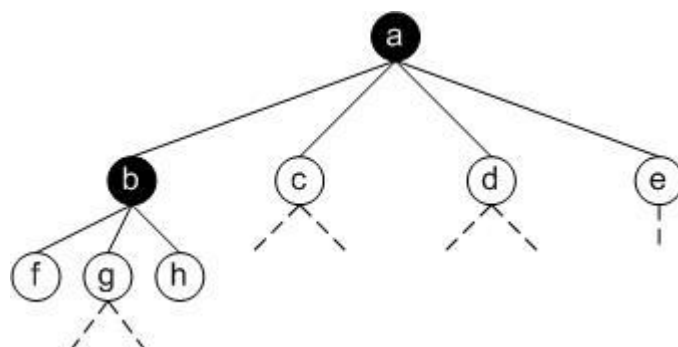
Gambar 4.2a Keadaan *tree*

2. Setelah mengekspansi *node a*



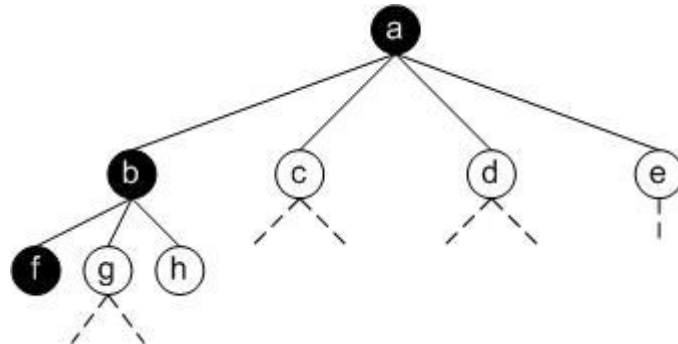
Gambar 4.2b Keadaan *tree*

3. Setelah mengekspansi *node b*



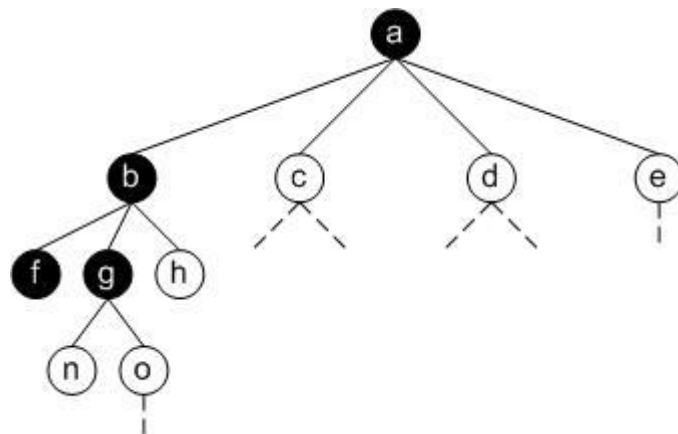
Gambar 4.2c Keadaan *tree*

4. Pencarian berhenti di *node f* karena *f* merupakan *leaf*. Prolog akan melakukan *backtrack* ke *node* sebelumnya yaitu *node b*.



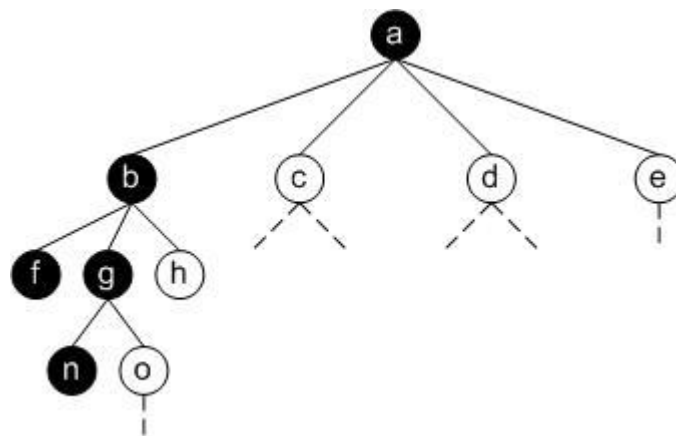
Gambar 4.2d Keadaan tree

5. Setelah mengekspansi *node g*



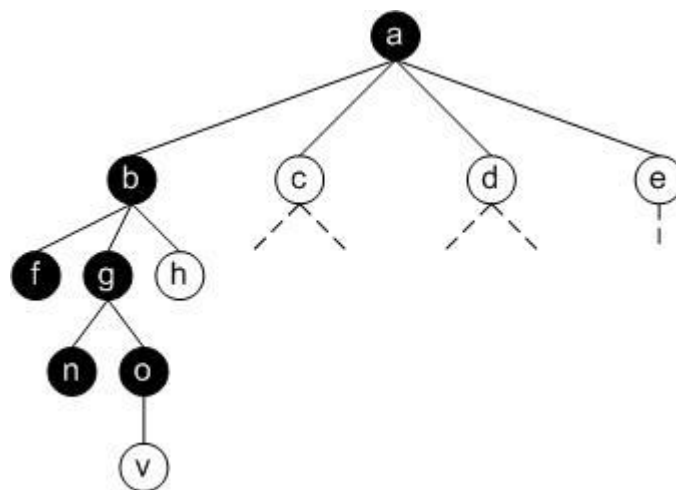
Gambar 4.2e Keadaan tree

6. Pencarian berhenti di *node n* karena *n* merupakan *leaf*. Prolog akan melakukan *backtrack* ke *node* sebelumnya yaitu *node g*.



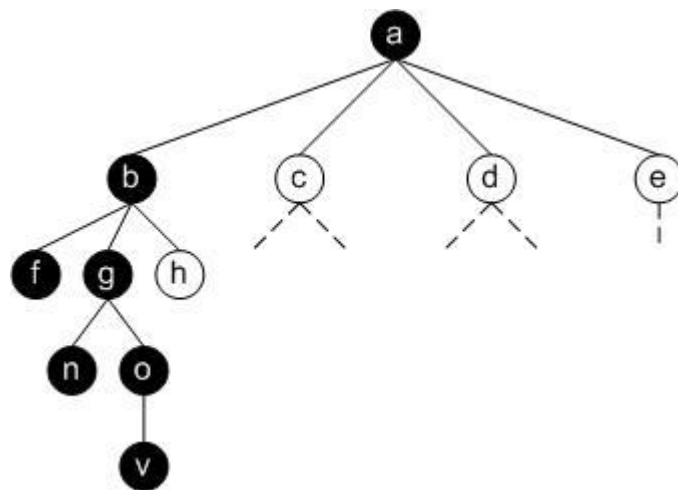
Gambar 4.2f Keadaan *tree*

7. Setelah mengekspansi *node o*



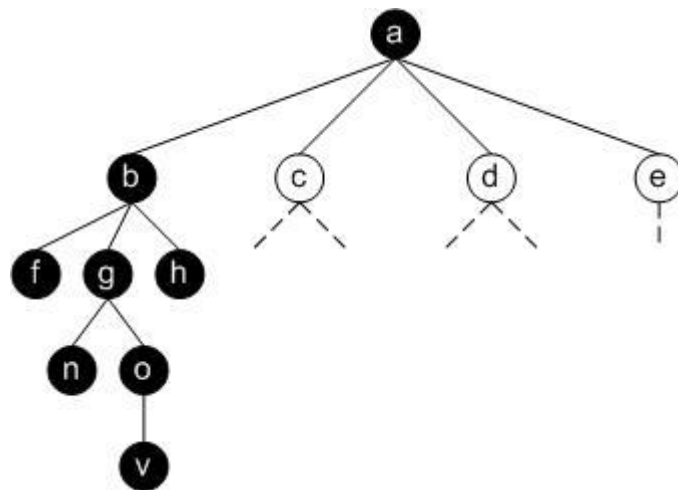
Gambar 4.2g Keadaan *tree*

8. Pencarian berhenti di *node v* karena *v* merupakan *leaf*. Prolog akan melakukan *backtrack* ke *node* sebelumnya yaitu *node o*. Akan tetapi *node o* tidak memiliki *node* yang dapat ditelusuri lagi. Prolog akan kembali melakukan *backtrack* ke *node* sebelumnya yaitu *node g*. Pada *node g* juga tidak ditemukan *node* lain yang dapat ditelusuri lagi. Prolog akan kembali melakukan *backtrack* ke *node b*.



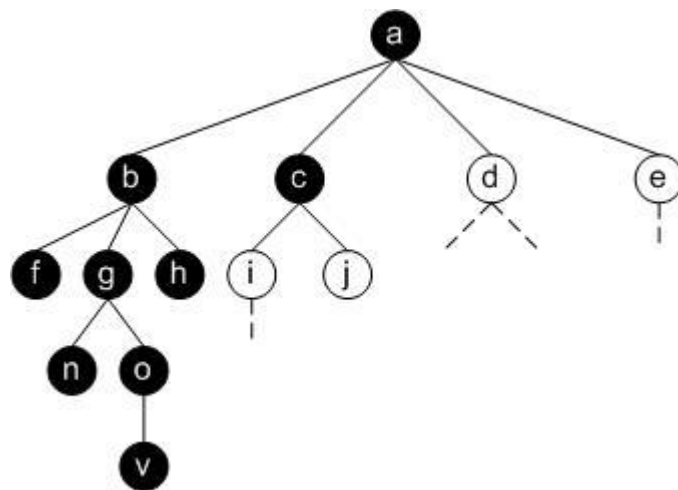
Gambar 4.2h Keadaan *tree*

9. Pencarian kemudian dilanjutkan ke *node* h.



Gambar 4.2i Keadaan *tree*

10. Setelah semua *node* yang merupakan ekspansi dari *node* b ditelusuri, penelusuran akan dilakukan untuk *node* lain. Dalam hal ini terhadap *node* c.



Gambar 4.2j Keadaan *tree*

4.2.2 Pencarian *successor*

Sama halnya dengan pencarian *predecessor*, kita dapat melakukan pencarian *successor*. *Successor* adalah *node* yang merupakan percabangan dari sebuah *node*. Pencarian *successor* dapat dilakukan dengan *rules*:

```
is_succ(X, Y) :- is_parent(Y, X).
```

```
is_succ(X, Y) :- is_parent(Y, Z), is_succ(X, Z).
```

4.2.3 Pencarian *leaf*

Leaf adalah *node* yang tidak memiliki *successor*. Kita dapat mencari *node* mana saja yang merupakan *leaf* dengan menambahkan *rules*:

```
is_node(X) :- is_parent(X, _) ; is_parent(_, X).
```

```
has_succ(X) :- is_succ(_, X).
```

```
is_leaf(X) :- is_node(X), not(has_succ(X)).
```

Sehingga apabila kita dapat memberikan *query* sebagai berikut:

```
?- is_leaf(v) .
```

```
true.
```

```
?- is_leaf(X) .
```

```
X = f ;
```

```
X = h ;
```

```
X = j ;
```

```
X = l ;
```

```
X = n ;
```

```
X = p ;
```

```
X = q ;
```

```
X = s ;
```

```
X = t ;
```

```
X = u ;
```

```
X = v ;
```

```
X = w ;
```

```
X = x.
```

4.2.4 Pencarian *root*

Root adalah *node* awal dalam *tree*. Sebuah *root* tidak memiliki *predecessor*. Sama seperti untuk mencari *leaf*, kita dapat mencari *root* dari sebuah *tree* dengan *rules*:

```
has_pred(X) :- is_pred(_, X).
```

```
is_root(X) :- is_node(X), not(has_pred(X)), !.
```

Setelah menambahkan kedua *rules* tersebut, kita dapat memberikan *query*:

```
?- is_root(X).
```

```
X = a.
```

```
?- is_root(a).
```

```
true.
```

4.2.5 Pencarian kedalaman *node*

Dengan memanfaatkan *rules* `is_root`, kita dapat menghitung kedalaman dari sebuah *node* pada *tree* dengan menambahkan *rules*:

```
depth(X, 0) :- is_root(X), !.
```

```
depth(X, D) :- is_parent(P, X), depth(P, D1), D is D1+1.
```

Rule pertama merupakan *base case* di mana kedalaman dari *root* adalah 0. *Rule* kedua merupakan *recursive case* untuk mengakumulasikan kedalaman *node* tersebut secara rekursif dengan mengunjungi *parent node*.

Selanjutnya kita dapat memberikan *query* sebagai berikut:

```
?- depth(u, D).
```

```
D = 3.
```

```
?- depth(a, D).
```

```
D = 0.
```



```
?- depth(x, D).
```

```
D = 4.
```

```
?- depth(b, D).
```

```
D = 1.
```

4.2.6 Pencarian *path*

Untuk lebih jauh, kita dapat melakukan pencarian *path* dalam sebuah *tree*. Pencarian *path* merupakan cara untuk menelusuri *tree* sehingga diperoleh *node-node* apa saja yang dilalui untuk mencapai *node* yang dicari. Pencarian *path* dilakukan dengan *rules*:

```
path(X) :- travel(X), write(X), !.
```

```
travel(X) :- is_root(X), !.
```

```
travel(X) :- is_parent(Y, X), travel(Y), write(Y),  
write('-->'), !.
```

Path-path yang diperoleh sebagai hasil *query* adalah sebagai berikut:

```
?- path(t).
```

```
a-->e-->m-->t
```

```
true.
```

```
?- path(a).
```

```
a
```

```
true.
```

```
?- path(g) .
```

```
a-->b-->g
```

```
true.
```

4.3 Latihan

1. Buatlah sebuah *database* berisi *facts* sebagai berikut:

```
byCar(auckland,hamilton) .
```

```
byCar(hamilton,raglan) .
```

```
byCar(valmont,saarbruecken) .
```

```
byCar(valmont,metz) .
```

```
byTrain(metz,frankfurt) .
```

```
byTrain(saarbruecken,frankfurt) .
```

```
byTrain(metz,paris) .
```

```
byTrain(saarbruecken,paris) .
```

```
byPlane(frankfurt,bangkok) .
```

```
byPlane(frankfurt,singapore) .
```

```
byPlane(paris,losAngeles) .
```

```
byPlane(bangkok,auckland) .
```

```
byPlane(singapore,auckland) .
```

```
byPlane(losAngeles, auckland) .
```

Selanjutnya pergunakan *facts* yang ada untuk menyelesaikan soal nomor 2-4.

2. Tambahkan *rule* `travel(X, Y, T)` yang dapat dipergunakan untuk memeriksa transportasi T yang dapat dipergunakan untuk pergi dari X ke Y. Contoh:

```
?- travel(valmont, metz, How) .
```

```
How = car .
```

3. Tambahkan *rule* `travel(X, Y)` yang dapat dipergunakan untuk memeriksa apakah mungkin untuk pergi dari X ke Y. Contoh:

```
?- travel(valmont, paris) .
```

```
true .
```

4. Tambahkan *rule* `path(X, Y)` untuk mencari *path* dari X ke Y. Contoh:

```
?- path(paris, hamilton) .
```

```
[paris]=plane=>[losAngeles]=plane=>[auckland]=car=>[hami  
lton]
```

```
true .
```

Hasil *query* tersebut menunjukkan bahwa seseorang dapat pergi dari Paris ke Hamilton dengan cara:

- Pertama-tama, dia harus naik pesawat dari Paris ke Los Angeles.
- Selanjutnya dia harus naik pesawat dari Los Angeles ke Auckland.
- Dan terakhir dia harus naik mobil dari Auckland ke Hamilton.

Bab V

Struktur *List*

5.1 *List*

List adalah struktur data dalam Prolog yang berisi deretan *item*. *List* ditulis dengan diapit tanda kurung $[a_0, a_1, a_2, \dots, a_{n-1}, a_n]$ dengan a_i merupakan elemen *list*. Sebuah *list* terdiri dari *head* dan *tail*. *Head* adalah *item* pertama *list* sedangkan *tail* adalah *item* selebihnya. Contoh sebuah *list*: `[apel, jeruk, mangga, jambu]`

5.2 Manipulasi *List*

List dapat dimanipulasi dengan cara memisahkan *head* dan *tail* menggunakan sebuah simbol *vertical bar*. Sebagai contoh, apabila kita melakukan *parsing* terhadap sebuah *list* dengan *query*:

```
?- [Head|Tail]=[apel, jeruk, mangga, jambu].
```

Kita akan memperoleh jawaban:

```
Head = apel,
```

```
Tail = [jeruk, mangga, jambu].
```

Hasil *parsing* tersebut menunjukkan variabel *Head* diinstansiasi dengan *item* pertama *list* (*head* dari *list*) dengan nilai `apel` dan variabel *Tail* diinstansiasi dengan *item* selebihnya *list* (*tail* dari *list* dan merupakan sebuah *list*) yaitu `[jeruk, mangga, jambu]`. Beberapa contoh *list* beserta *head* dan *tail*-nya seperti pada Tabel 5.1.

Tabel 5.1 *List* beserta *head* dan *tail*.

List	Head	Tail
<code>[[tikus, kucing, anjing]]</code>	<code>[tikus, kucing, anjing]</code>	<code>[]</code>
<code>[makan, [tidur, bangun]]</code>	<code>makan</code>	<code>[[tidur, bangun]]</code>

Tabel 5.1 *List* beserta *head* dan *tail* (lanjutan).

List	Head	Tail
[[joko, tri], susi, budi]	[joko, tri]	[susi, budi]
[apel]	apel	[]
[[joko]]	[joko]	[]

Tabel 5.2 memperlihatkan proses *match* pada dua buah *list*:

Tabel 5.2 *Matching* pada *list*.

List 1	List 2	Instansiasi
[[joko, tri], susi, budi]	[X Y]	X = [joko, tri], Y = [susi, budi].
[apel]	[X Y]	X = apel, Y = [].
[[joko]]	[X Y]	X = [joko], Y = [].
[[joko, tri], susi, budi]	[X [Y Z]]	X = [joko, tri], Y = susi, Z = [budi].
[[apel, X], mangga]	[[Z, jambu] Y]	X = jambu, Z = apel, Y = [mangga].
[X, Y Z]	[joko, tuti, tika, siti]	X = joko, Y = tuti, Z = [tika, siti].

5.3 Operasi dalam *List*

5.3.1 Penambahan *item* ke *list*

Sebuah *item* dapat ditambahkan ke dalam *list* sehingga menjadi sebuah *list* baru. Penambahan *item* dapat dilakukan pada beberapa bagian *list* yaitu:

1. Penambahan *item* di depan *list*. Penambahan *item* di depan *list* dapat dilakukan dengan *rule* sebagai berikut:

```
tambah_depan(Item, List ,Hasil) :- Hasil = [Item|List].
```

Sehingga apabila kita memberikan *query*:

```
?- tambah_depan(durian, [jambu, mangga, apel], L).
```

Kita akan memperoleh jawaban $L = [\text{durian}, \text{jambu}, \text{mangga}, \text{apel}]$.

Proses penambahan *item* di depan *list* seperti diilustrasikan dalam Gambar 5.1.



Gambar 5.1 Proses penambahan *item* di depan *list*.

Penambahan *item* di depan *list* akan mengakibatkan *item* yang ditambahkan akan menjadi *head* dari *list* yang baru dan *list* akan menjadi *tail* dari *list* yang baru. *Rule* untuk menambahkan *item* di depan *list* dapat diubah menjadi bentuk yang lebih singkat yaitu:

```
tambah_depan(Item, List , [Item|List]).
```

2. Penambahan *item* di akhir *list*. Untuk menambahkan *item* di akhir *list* kita dapat menggunakan *rule*:

```
tambah_akhir(Item, [], [Item]) :- !.
```

```
tambah_akhir(Item, [Head | Tail], [Head | L]) :-
```

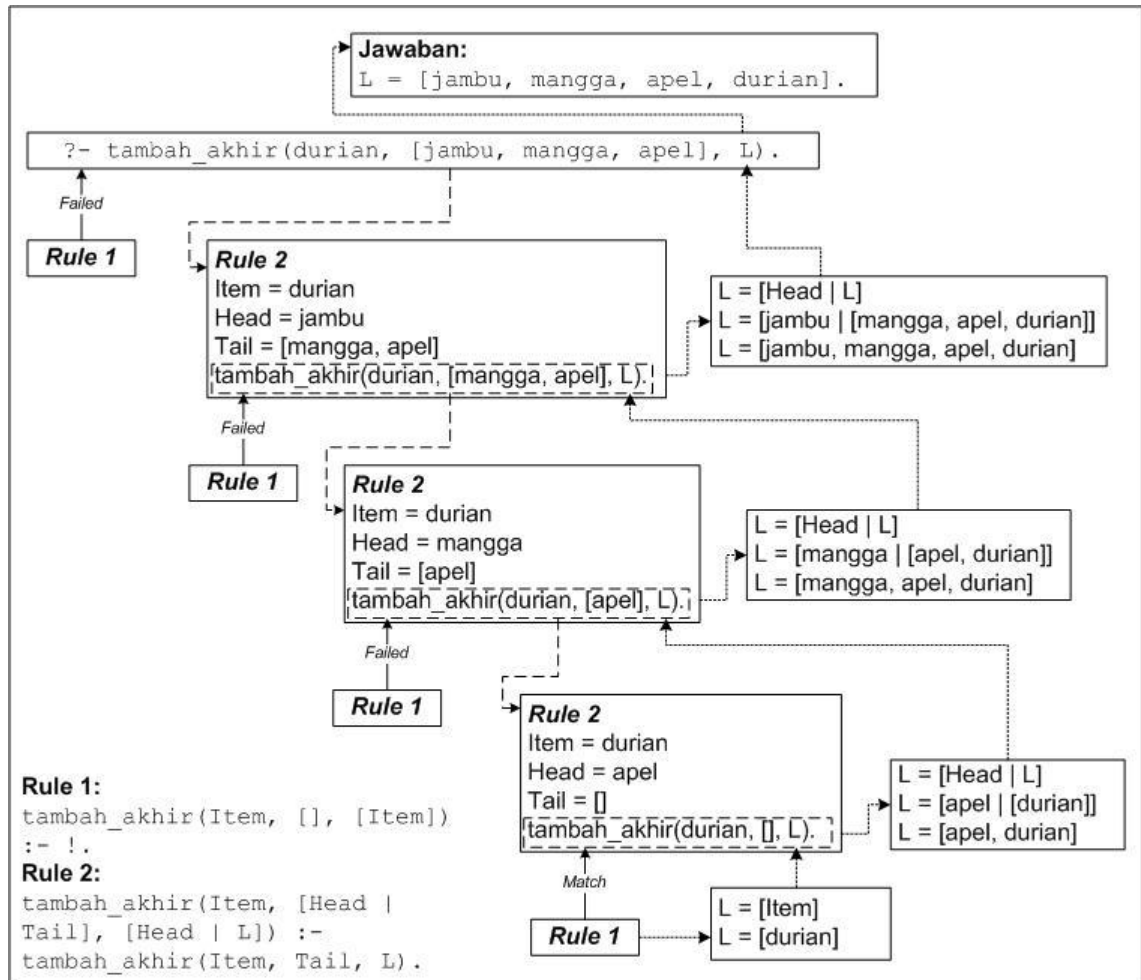
```
tambah_akhir(Item, Tail, L).
```

Apabila kita menanyakan *query*:

?- tambah_akhir(durian, [jambu, mangga, apel], L).

Maka kita akan mendapatkan jawaban $L = [jambu, mangga, apel, durian]$.

Proses penambahan *item* di akhir *list* seperti diilustrasikan dalam Gambar 5.2.



Gambar 5.2 Proses penambahan *item* di akhir *list*.

Untuk melakukan penambahan *item* di akhir *list*, kita menggunakan dua buah *rule*. *Rule* pertama merupakan *base case* di mana apabila *item* akan ditambahkan pada sebuah *list* kosong, maka akan diperoleh sebuah *list* berisi *item* itu sendiri. *Rule* kedua merupakan *recursive case* di mana apabila sebuah *item* akan ditambahkan ke *list* yang bukan merupakan *list* kosong, maka *list* tersebut harus disederhanakan terlebih dahulu. Penyederhanaan *list* dilakukan dengan cara membagi *list* menjadi

head dan *tail*. Kemudian, penyederhanaan kembali dilakukan terhadap *tail* dari *list* secara rekursif sampai pada *base case* yang akan menambah *item* tersebut ke dalam *list* kosong tersebut. Selanjutnya, pada masing-masing proses rekursif, akan dilakukan penambahan *head* di awal *list*.

5.3.2 Pencarian *item* dalam *list*

Kita dapat melakukan pencarian dalam sebuah *list* untuk mengetahui apakah sebuah *item* terdapat dalam *list* tersebut atau tidak. Untuk melakukan pencarian dalam *list* kita dapat menambah *rules* sebagai berikut:

```
cari(Item, [Item|_]) :- !.
cari(Item, [_|Tail]) :- cari(Item, Tail).
```

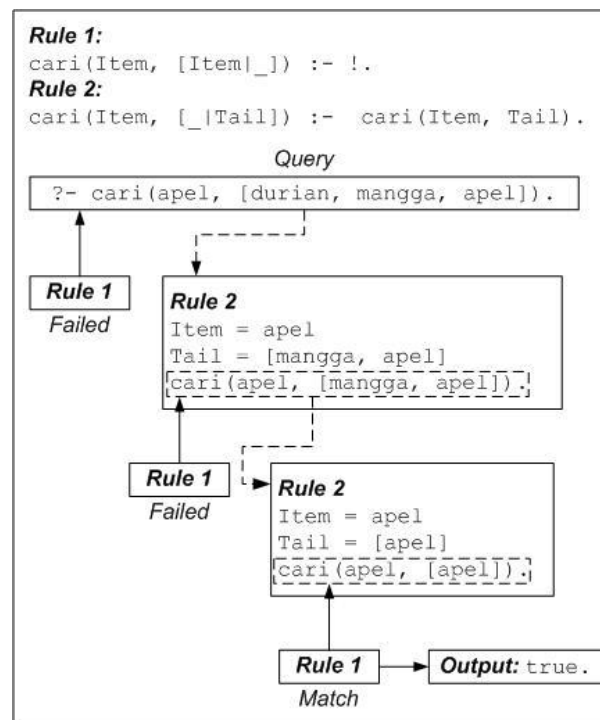
Apabila kita memberikan *query*:

```
?- cari(apel, [durian, mangga, apel]).
```

Maka kita akan mendapatkan jawaban `true`.

Rule pertama merupakan *base case* di mana pencarian akan berhenti jika *item* yang dicari merupakan *head* dari *list*. *Rule* kedua merupakan *recursive case* di mana apabila *item* yang dicari bukan merupakan *head* dari *list* maka pencarian akan dilakukan kembali secara rekursif terhadap *tail* dari *list*. Kedua *rules* tersebut hanya dipergunakan untuk melakukan pencarian terhadap *item* yang terdapat dalam *list* atau selama *list* masih ada (tidak kosong). Namun, secara implisit dapat disimpulkan jika pencarian dilakukan sampai *list* habis atau kosong (`[]`) dan *item* yang dicari tidak ditemukan, maka *item* tidak terdapat dalam *list*. Jawaban `true` yang diberikan oleh Prolog menandakan bahwa terdapat *rule* yang sesuai atau dalam kasus ini jawaban tersebut memiliki arti yang sama dengan "*item* ditemukan dalam *list*". Demikian sebaliknya, jawaban `false` yang diberikan Prolog menandakan bahwa tidak ada *rules* yang sesuai atau dalam kasus ini memiliki arti yang sama dengan "*item* tidak ditemukan dalam *list*".

Proses penelusuran *item* dalam *list* seperti diilustrasikan dalam Gambar 5.3.



Gambar 5.3 Proses pencarian *item* di dalam *list*.

5.3.3 Penghapusan *item* dari *list*

Penghapusan *item* dari *list* bertujuan untuk membuang *item* yang tidak diinginkan dari *list*. Untuk menghapus sebuah *item* dari *list* kita dapat menggunakan *rules* sebagai berikut:

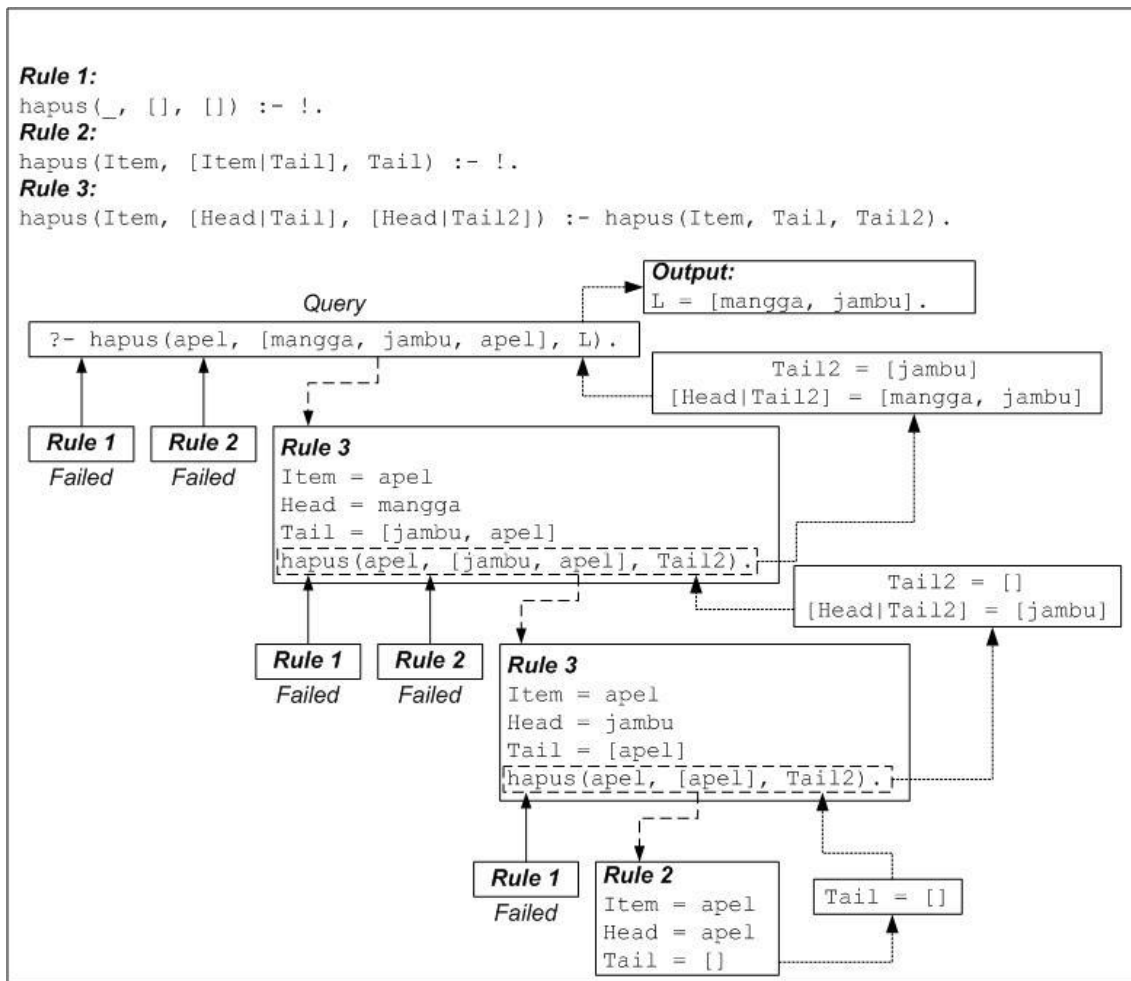
```
hapus(_, [], []) :- !.
hapus(Item, [Item|Tail], Tail) :- !.
hapus(Item, [Head|Tail], [Head|Tail2]) :- hapus(Item,
Tail, Tail2).
```

Apabila kita memberikan *query*:

```
?- hapus(apel, [mangga, jambu, apel], L).
```

Maka kita akan memperoleh jawaban `L = [mangga, jambu]`.

Proses penghapusan *item* dari *list* seperti diilustrasikan dalam Gambar 5.4.



Gambar 5.4 Penghapusan item dari list.

Rule pertama merupakan rule opsional yang dipergunakan agar Prolog tidak mengembalikan nilai `false` apabila Prolog tidak menemukan *item* yang ingin dihapus dari *list*. Rule kedua merupakan *base case* di mana apabila Prolog menemukan *item* yang ingin dihapus sebagai *head* dari sebuah *list* maka Prolog akan mengembalikan *tail* dari *list* tersebut. Rule ketiga merupakan *recursive case* di mana apabila *item* yang ingin dihapus bukan merupakan *head* dari *list*, Prolog akan mencari *item* yang ingin dihapus pada *tail* dari *list* secara rekursif.

5.3.4 Penggabungan list

Penggabungan *list* atau *concatenate* digunakan untuk menggabungkan dua buah *list* sehingga dihasilkan sebuah *list* baru. Penggabungan *list* dapat dilakukan dengan *rules*:

```
gabung([], L2, L2).
```

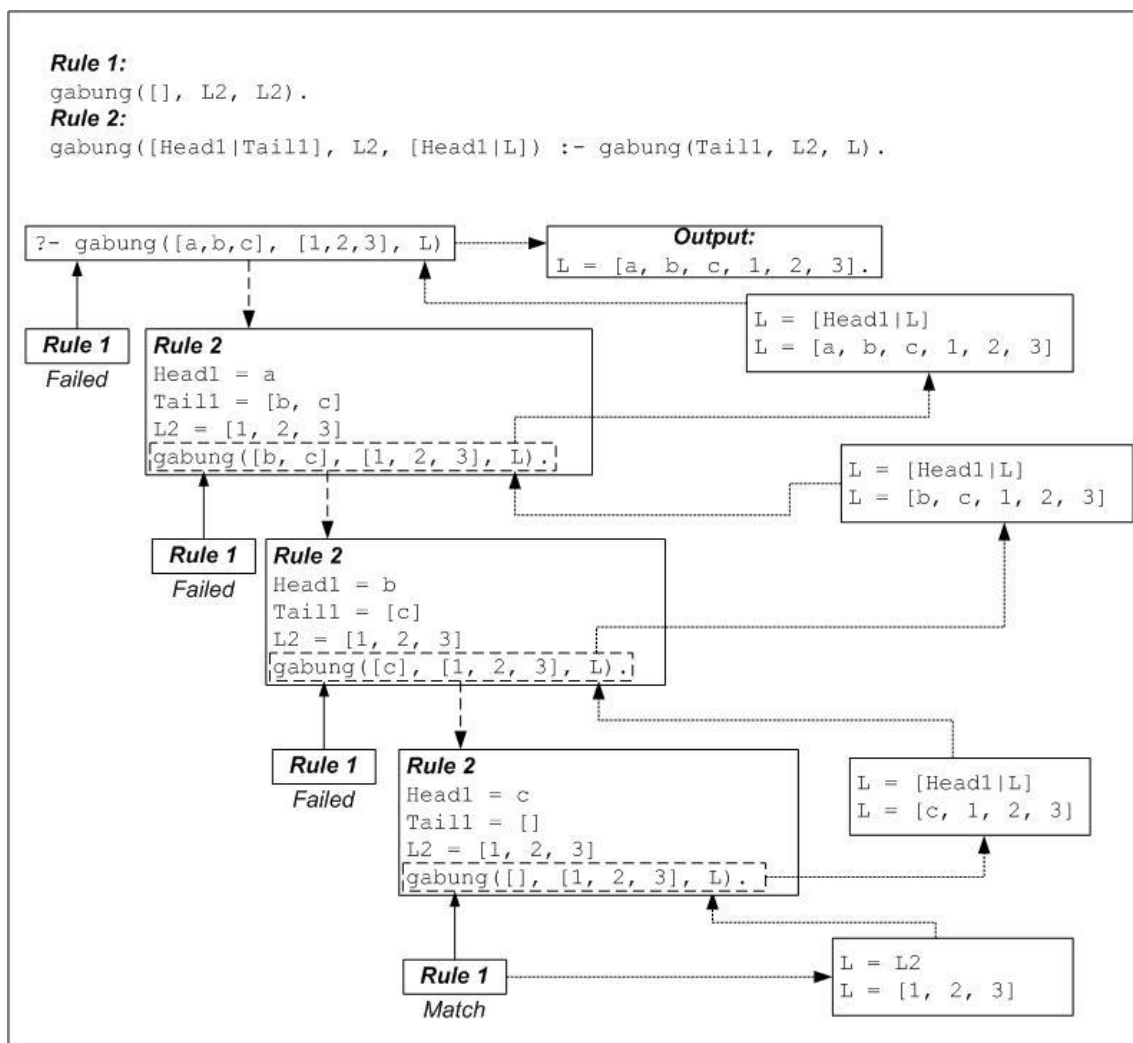
```
gabung([Head1|Tail1], L2, [Head1|L]) :- gabung(Tail1,
L2, L).
```

Apabila kita memberikan *query*:

```
?- gabung([a,b,c], [1,2,3], L).
```

Kita akan memperoleh jawaban $L = [a, b, c, 1, 2, 3]$.

Proses penggabungan *list* seperti diilustrasikan pada Gambar 5.5.



Gambar 5.5 Penggabungan *list*.

Rule pertama merupakan *base case* di mana apabila sebuah *list* digabung dengan *list* lain, maka akan menghasilkan *list* itu sendiri. *Rule* kedua merupakan *recursive case*

yang dipergunakan menggabungkan *head* dari *list* pertama dan hasil penggabungan *tail* dari *list* pertama dengan *list* kedua.

Kita juga dapat melihat kemungkinan *list* apa saja yang digabungkan untuk menghasilkan sebuah *list* baru dengan memberikan *query*:

```
?- gabung(L1, L2, [a,b,c,d]).
L1 = [],
L2 = [a, b, c, d] ;
L1 = [a],
L2 = [b, c, d] ;
L1 = [a, b],
L2 = [c, d] ;
L1 = [a, b, c],
L2 = [d] ;
L1 = [a, b, c, d],
L2 = [] ;
false.
```

5.3.5 Sublist

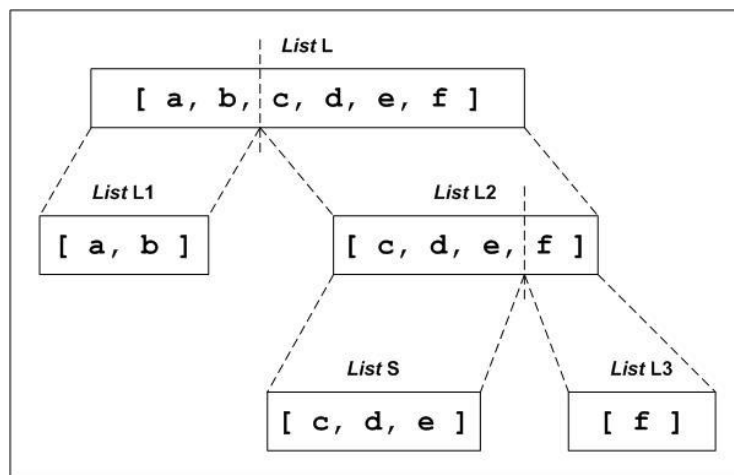
Sebuah *list* S merupakan *sublist* dari *list* L apabila:

1. *List* L1 dan *list* L2 dapat digabungkan sehingga menghasilkan *list* L.
2. *List* S dapat digabungkan dengan sebuah *list* L3 membentuk *list* baru L2 dan

Dengan demikian, berdasarkan pengertian *sublist* tersebut, kita dapat membuat *rule* untuk mencari *sublist* yaitu:

```
sublist(S, L) :- gabung(_, L2, L), gabung(S, _, L2).
```

Berdasarkan *rule* tersebut, apabila S digabungkan dengan sebuah *list* L3, maka akan menghasilkan *list* L2. *List* L2 ini selanjutnya digabungkan dengan *list* L1 sehingga menghasilkan *list* L. Proses penggabungan ini seperti diberikan pada Gambar 5.6.



Gambar 5.6 Ilustrasi *sublist*.

Dalam hal ini, kita dapat mengabaikan nilai L1 dan L3 karena kita hanya perlu mengetahui bahwa terdapat *list* yang dapat digabungkan dengan S dan L2 tanpa perlu mengetahui isi dari *list* tersebut.

Beberapa *query* yang dapat diberikan pada *rule* tersebut adalah:

```
?- sublist([b,c], [a,b,c,d]).
true .
```

```
?- sublist([b,d], [a,b,c,d]).
false.
```

```
?- sublist(S, [a, b, c, d]).
```

```
S = [] ;
```

```
S = [a] ;
```

```
S = [a, b] ;
```

```
S = [a, b, c] ;
```

```
S = [a, b, c, d] ;
```

```
S = [] ;
```

```
S = [b] ;
```

```
S = [b, c] ;
```

```
S = [b, c, d] ;
```

```
S = [] ;
```

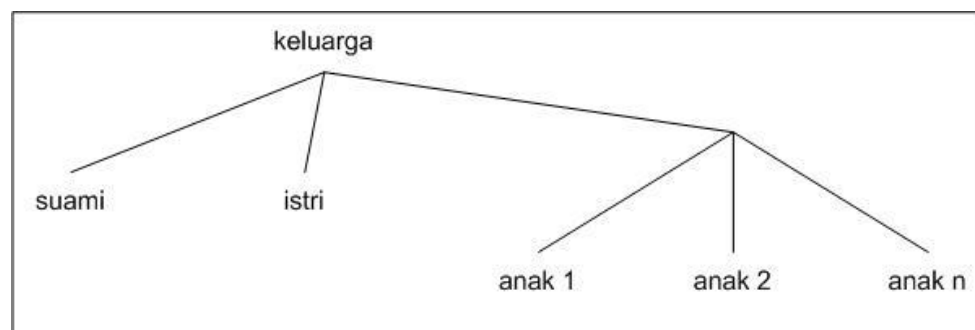
```

S = [c] ;
S = [c, d] ;
S = [] ;
S = [d] ;
S = [] ;
false.

```

5.4 Penggunaan *List* sebagai Struktur Data

List dapat dipergunakan sebagai struktur data. Salah satu contoh penggunaan *list* sebagai struktur data adalah pada *database* keluarga. Sebuah **keluarga** terdiri dari satu orang **ayah**, satu orang **ibu**, dan **anak-anak** seperti diberikan pada Gambar 5.7.



Gambar 5.7 Struktur keluarga

Dengan demikian kita dapat membuat *database* dengan menggunakan relasi **keluarga** sebagai *principal functor*. Relasi **keluarga** dapat ditulis dengan argumen pertama sebagai **suami**, argumen kedua sebagai **istri**, dan argumen ketiga sebagai **anak-anak** sebagai berikut:

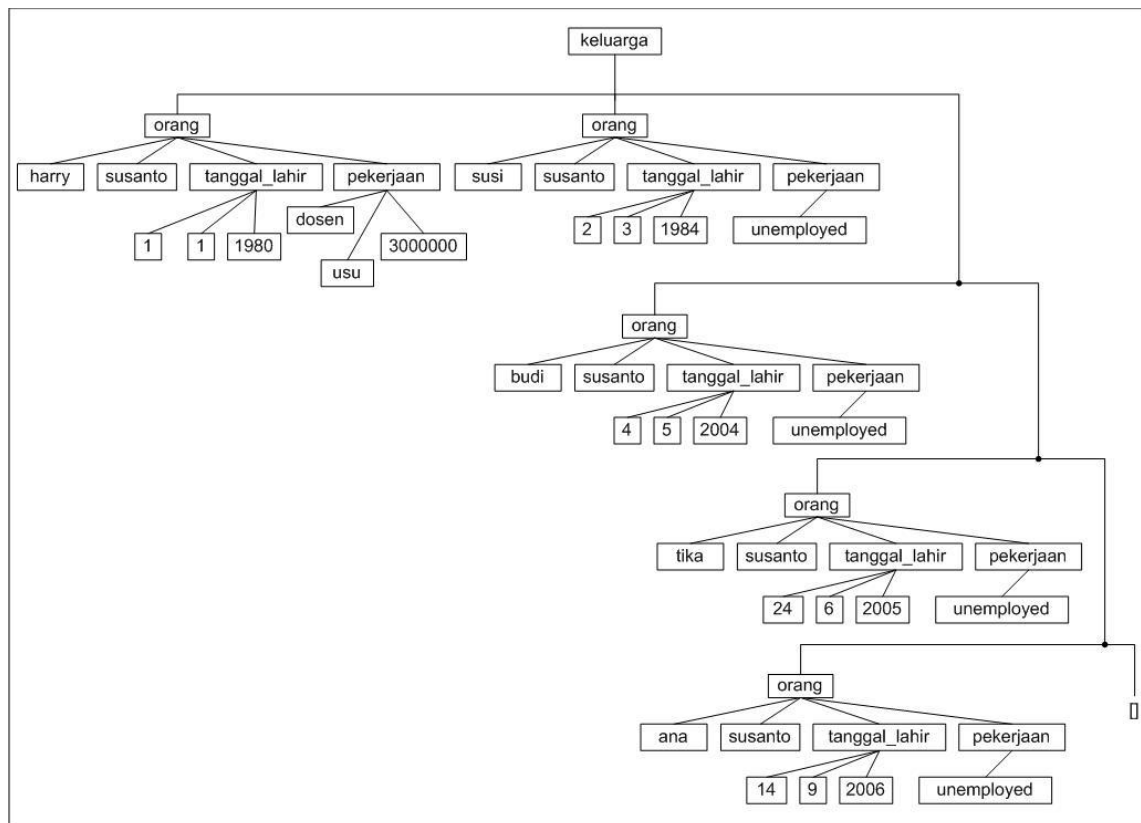
```
keluarga(Suami, Istri, Anak).
```

Beberapa hal yang dipergunakan sebagai acuan dalam penulisan relasi **keluarga** adalah:

1. Dalam sebuah keluarga, dapat terdapat lebih dari satu orang anak. Oleh karena itu, argumen Anak dalam struktur keluarga merupakan sebuah *list*.

2. Ayah, ibu, dan anak-anak dalam keluarga merupakan **orang**. Setiap **orang** memiliki nama depan, nama belakang (pada umumnya mengikuti nama belakang ayah), tanggal lahir, dan pekerjaan.

Misalnya kita memiliki sebuah pohon keluarga seperti diberikan pada Gambar 5.8.



Gambar 5.8 Pohon keluarga

Pohon keluarga tersebut dapat diubah menjadi sebuah bentuk *fact* sebagai berikut:

```
keluarga (
  orang(harry,      susanto,      tanggal_lahir(1,1,1980),
  pekerjaan(dosen,  usu,  3000000)),
  orang(susi,      susanto,      tanggal_lahir(2,3,1984),
  pekerjaan(unemployed)),
  [
    orang(budi,      susanto,      tanggal_lahir(4,5,2004),
    pekerjaan(unemployed)),
```

```

    orang(tika,      susanto,      tanggal_lahir(24,6,2005),
    pekerjaan(unemployed)),
    orang(ana,      susanto,      tanggal_lahir(14,9,2006),
    pekerjaan(unemployed))
]
).
```

Nilai argumen *unemployed* menyatakan bahwa orang tersebut tidak bekerja. Berdasarkan *fact* tersebut, kita dapat merujuk pada argumen-argumen dalam relasi tersebut dengan *query*:

```
?- keluarga(Suami,Istri,Anak).
```

Jawaban yang diperoleh dari *query* tersebut adalah:

```

Suami = orang(harry, susanto, tanggal_lahir(1, 1, 1980),
pekerjaan(dosen, usu, 3000000)),
Istri = orang(susi, susanto, tanggal_lahir(2, 3, 1984),
pekerjaan(unemployed)),
Anak = [orang(budi, susanto, tanggal_lahir(4, 5, 2004),
pekerjaan(unemployed)),      orang(tika,      susanto,
tanggal_lahir(24, 6, 2005),      pekerjaan(unemployed)),
orang(ana,      susanto,      tanggal_lahir(14, 9, 2006),
pekerjaan(unemployed))].
```

Kita juga dapat melakukan penambahan beberapa *rules* untuk mengolah informasi dalam *database* tersebut lebih lanjut. *Rules* yang dapat ditambahkan adalah sebagai berikut:

```

%X adalah suami
suami(X) :- keluarga(X, _, _).
```

```

%X adalah istri
istri(X) :- keluarga(_, X, _).
```



```

%X adalah anak
anak(X) :- keluarga(_, _, Anak), member(X, Anak).

%penghasilan
penghasilan(orang(_, _, _, pekerjaan(unemployed)), 0).
penghasilan(orang(_, _, _, pekerjaan(_, _, X)), X).

%total penghasilan keluarga
total_penghasilan([], 0).
total_penghasilan([X|Tail], P) :- penghasilan(X, P1),
total_penghasilan(Tail, P2), P is P1+P2.

```

Kemudian kita dapat mengajukan *query*:

```

?- suami(X).
X = orang(harry, susanto, tanggal_lahir(1, 1, 1980),
pekerjaan(dosen, usu, 3000000)).

?- istri(X).
X = orang(susi, susanto, tanggal_lahir(2, 3, 1984),
pekerjaan(unemployed)).

?- anak(X).
X = orang(budi, susanto, tanggal_lahir(4, 5, 2004),
pekerjaan(unemployed)) ;
X = orang(tika, susanto, tanggal_lahir(24, 6, 2005),
pekerjaan(unemployed)) ;
X = orang(ana, susanto, tanggal_lahir(14, 9, 2006),
pekerjaan(unemployed)).

?-          keluarga(Suami,          Istri,
Anak),total_penghasilan([Suami, Istri| Anak], P).

```

```

Suami = orang(harry, susanto, tanggal_lahir(1, 1, 1980),
pekerjaan(dosen, usu, 3000000)),
Istri = orang(susi, susanto, tanggal_lahir(2, 3, 1984),
pekerjaan(unemployed)),
Anak = [orang(budi, susanto, tanggal_lahir(4, 5, 2004),
pekerjaan(unemployed)), orang(tika, susanto,
tanggal_lahir(24, 6, 2005), pekerjaan(unemployed)),
orang(ana, susanto, tanggal_lahir(14, 9, 2006),
pekerjaan(unemployed))],
P = 3000000 .

```

5.5 Latihan

1. Buatlah relasi untuk:
 - a. `len(List, Len)`. %Jumlah *item* dalam *list*
 - b. `max(List, Max)`. %Nilai *item* maksimal dalam *list*
 - c. `sum(List, Sum)`. %Total *item* dalam *list*
 - d. `avg(List, Avg)`. %Nilai rata-rata *item* dalam *list*
2. Buatlah *rules* untuk melakukan penambahan sebuah *item* pada posisi ke-*n* sebuah *list*. Contoh:


```

?- tambah(durian, [apel, mangga, jambu], 2, L).
L = [apel, durian, mangga, jambu].

```
3. Modifikasilah *rules* pada **5.3.2. Pencarian *item* dalam *list*** sehingga hasil pencarian tidak hanya berupa nilai *boolean* namun berupa posisi *item* dalam *list*. Jika *item* tidak ditemukan dalam *list*, kembalikan nilai 0. Jika *item* ditemukan dalam *list*, kembalikan nilai posisi *item*. Contoh:


```

?- cari(durian, [apel, mangga, jambu], Posisi).
Posisi = 0.
?- cari(mangga, [apel, mangga, jambu], Posisi).
Posisi = 2.

```
4. *Rules* pada **5.3.3. Penghapusan *item* dalam *list*** hanya akan melakukan penghapusan sebuah *item* dari *list*. Apabila kita memberikan *query*:


```

?- hapus(apel, [apel, mangga, apel, apel, jambu], L).

```

`L = [mangga, apel, apel, jambu].`

Modifikasilah *rules* tersebut sehingga *rules* tersebut dapat menghapus semua *item* yang dipilih dari *list*. Contoh:

`?- hapus(apel, [apel, mangga, apel, apel, jambu], L).`

`L = [mangga, jambu].`

5. Tambahkan *rule* ke dalam *database* keluarga pada **5.4. Penggunaan *List* sebagai Struktur Data** untuk:
 - a. menghitung jumlah anak dalam sebuah keluarga.
 - b. mencari nama semua orang yang ada di dalam database.

Bab VI

Representasi Pengetahuan

6.1 Representasi Pengetahuan

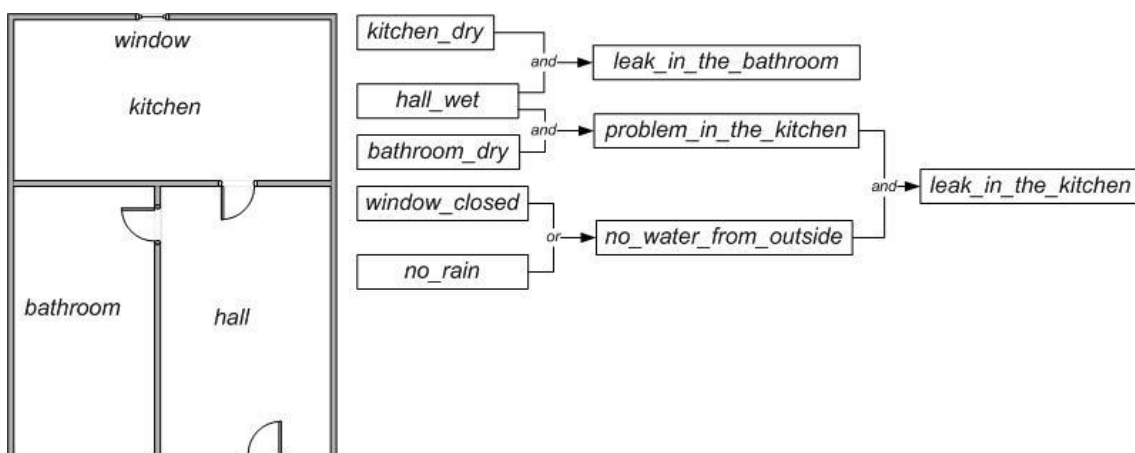
Representasi pengetahuan adalah suatu cara untuk menangkap ciri-ciri *domain* masalah dan menyajikan informasi dalam bentuk yang dapat dimanfaatkan sistem komputer untuk menyelesaikan sebuah pekerjaan yang kompleks.

6.2 If-Then Rules

If-Then rules merupakan aturan formal yang dipergunakan untuk merepresentasikan pengetahuan. *If-Then rules* dalam memiliki beberapa interpretasi yaitu:

- *If* kondisi P *then* kesimpulan C.
- *If* situasi S *then* aksi A.
- *If* kondisi C1 dan C2 berlaku *then* kondisi C3 tidak berlaku.

Sebagai contoh, terdapat *knowledge base* untuk mendiagnosa kebocoran seperti yang diilustrasikan pada Gambar 6.1.



Gambar 6.1 Ilustrasi *knowledge base* untuk diagnosa kebocoran

Berdasarkan ilustrasi pada Gambar 6.1 tersebut, kita dapat memperoleh *knowledge base* manual:

*If kitchen is dry **and** hall is wet **then** there is a leak in the bathroom.*

*If hall is wet **and** bathroom is dry **then** there is a problem in the kitchen.*

*If window is closed **or** there is no rain **then** there is no water from outside.*

*If there is a problem in the kitchen **and** there is no water from outside **then** there is a leak in the kitchen.*

Pengetahuan dalam bentuk *if-then rules* tersebut dapat direpresentasikan dalam bentuk *rules* di Prolog sebagai berikut:

```
leak_in_bathroom :- hall_wet, kitchen_dry, !.
problem_in_the_kitchen :- hall_wet, bathroom_dry, !.
no_water_from_outside :- window_closed; no_rain, !.
leak_in_the_kitchen      :-      problem_in_the_kitchen,
no_water_from_outside, !.
```

Selanjutnya kita dapat menambahkan *facts* berikut:

```
hall_wet.
bathroom_dry.
window_closed.
```

Facts tersebut dapat ditambahkan dengan dua cara yaitu:

1. Secara manual dengan menambahkan ketiga *facts* tersebut dalam *knowledge base*.
2. Menggunakan perintah `assert`. Contoh:

```
?- assert(hall_wet).
true.
```

```
?- assert(bathroom_dry).
true.
```

```
?- assert(window_closed).
true.
```

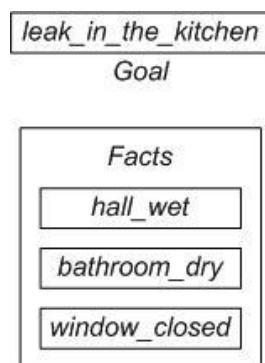
Setelah ketiga *fact* tersebut ditambahkan, kita dapat memberikan *query*:

```
?- leak_in_the_kitchen.
true.
```

6.3 Backward Chaining

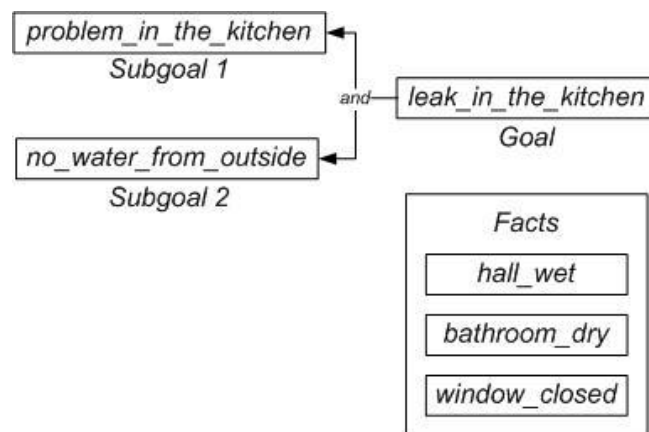
Secara *default*, Prolog mengimplementasikan metode *backward chaining* untuk melakukan penarikan kesimpulan. Metode *backward chaining* adalah suatu proses penarikan kesimpulan berdasarkan *goal* dan oleh karena itu dikenal juga dengan istilah *goal-driven*. *Goal* yang ingin dipenuhi pertama-tama akan dibagi menjadi beberapa *subgoal* yang harus dipenuhi. Contoh *backward chaining*:

1. Terdapat sebuah *goal* yang ingin dipenuhi yaitu `leak_in_the_kitchen` dan 3 *facts* yaitu `hall_wet`, `bathroom_dry`, dan `window_closed` seperti pada Gambar 6.2a.



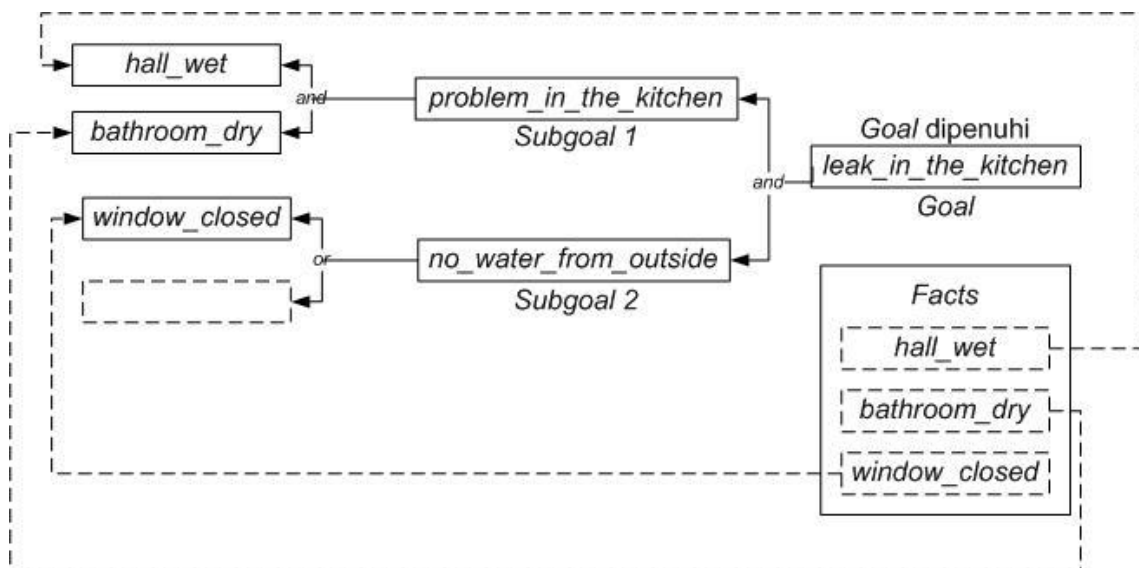
Gambar 6.2a Backward chaining

2. *Goal* `leak_in_the_kitchen` akan dibagi menjadi dua *subgoal* yaitu `problem_in_the_kitchen` dan `no_water_from_outside`. Kedua *subgoal* tersebut harus dipenuhi agar kesimpulan `leak_in_the_kitchen` dapat ditarik seperti pada Gambar 6.2b.



Gambar 6.2b Backward chaining

3. *Subgoal* yang ada sesuai dengan *facts* yang dimiliki. Dengan demikian kedua *subgoal* berhasil dipenuhi. Karena *subgoal* berhasil dipenuhi maka kesimpulan pada *goal* juga dipenuhi seperti terlihat pada Gambar 6.2c.



Gambar 6.2c Backward chaining

6.4 Menggunakan Operator Buatan Pengguna

Representasi *rules* dalam sintaks Prolog mengakibatkan keterbatasan orang yang dapat membaca *rules* tersebut. Seorang pakar untuk suatu domain masalah belum tentu memahami sintaks Prolog yang ada. Oleh karena itu pengetahuan yang direpresentasikan dalam *rules* Prolog dapat ditulis dengan operator buatan pengguna

sehingga mempermudah pakar untuk memahami, menambah atau memodifikasi *knowledge base*.

Penambahan operator buatan pengguna adalah sebagai berikut:

```
:- op(800, fx, if) .
:- op(700, xfx, then) .
:- op(300, xfy, or) .
:- op(200, xfy, and) .
```

Setelah operator tersebut ditambahkan, maka *rules* yang ada dapat diubah menjadi:

```
if hall_wet and kitchen_dry then leak_in_bathroom.
if      hall_wet      and      bathroom_dry      then
problem_in_the_kitchen.
if window_closed or no_rain then no_water_from_outside.
if problem_in_the_kitchen and no_water_from_outside then
leak_in_the_kitchen.
```

Selanjutnya *facts* yang ada ditulis dengan relasi untuk membedakannya dengan *rules* sebagai berikut:

```
fact(hall_wet) .
fact(bathroom_dry) .
fact(window_closed) .
```

Selanjutnya ditambahkan sebuah relasi baru yaitu `is_true(P)` dengan `P` merupakan *fact* yang ada atau *fact* yang diperoleh berdasarkan *rules* yang ada.

Perubahan program tersebut akan menghasilkan hasil akhir sebagai berikut:

```
:- op(800, fx, if) .
:- op(700, xfx, then) .
:- op(300, xfy, or) .
:- op(200, xfy, and) .
```



```

is_true(P) :- fact(P).
is_true(P) :- if Condition then P, is_true(Condition),
!.
is_true(P1 and P2) :- is_true(P1), is_true(P2), !.
is_true(P1 or P2) :- is_true(P1); is_true(P2), !.

if hall_wet and kitchen_dry then leak_in_bathroom.
if      hall_wet      and      bathroom_dry      then
problem_in_the_kitchen.
if window_closed or no_rain then no_water_from_outside.
if problem_in_the_kitchen and no_water_from_outside then
leak_in_the_kitchen.

fact(hall_wet).
fact(bathroom_dry).
fact(window_closed).

```

6.5 Forward Chaining

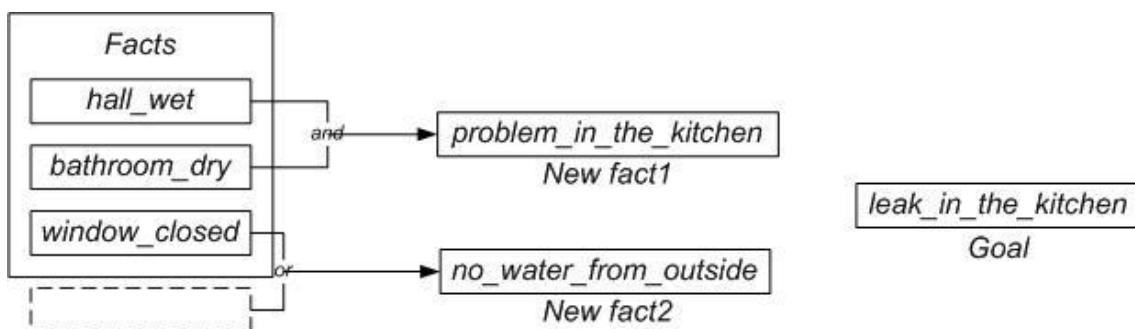
Forward chaining adalah metode untuk menarik kesimpulan dengan mengacu pada data yang ada dan oleh karena itu sering juga dikenal dengan sebutan *data-driven*. Dengan menggunakan metode *forward chaining*, kita akan dapat memperoleh *fact* baru dari *fact* yang telah ada ataupun menarik kesimpulan dengan menggunakan *fact* yang telah ada. Contoh *forward chaining*:

1. Terdapat sebuah *goal* yang ingin dipenuhi yaitu `leak_in_the_kitchen` dan 3 *facts* yaitu `hall_wet`, `bathroom_dry`, dan `window_closed` seperti pada Gambar 6.3a.

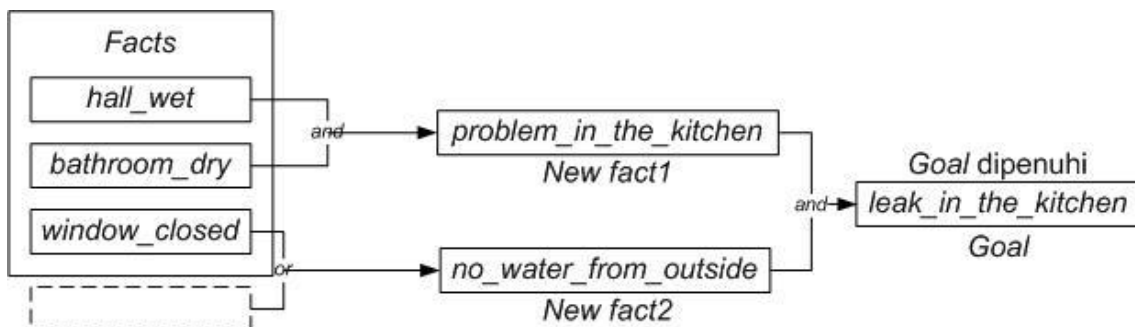
Gambar 6.3a *Forward chaining*

2. Pada Gambar 6.3b terlihat bahwa diperoleh dua *facts* baru berdasarkan *fact* sebelumnya yaitu:

- a. *Fact* `problem_in_the_kitchen` dari *rule*: if `hall_wet` and `bathroom_dry` then `problem_in_the_kitchen`.
- b. *Fact* `no_water_from_outside` dari *rule*: if `window_closed` or `no_rain` then `no_water_from_outside`.

Gambar 6.3b *Forward chaining*

3. Berdasarkan kedua *facts* baru tersebut *goal* yang ada berhasil dipenuhi seperti pada Gambar 6.3c.

Gambar 6.3c *Forward chaining*

Implementasi *forward chaining* sebagai berikut:

```
:- dynamic fact/1.
:- op(800, fx, if).
:- op(700, xfx, then).
:- op(300, xfy, or).
:- op(200, xfy, and).
```

```
forward :- (new_derived_fact(F), !, write('Derived: '),
write(F), nl, assert(fact(F)), forward); (write('No more
facts'), nl).
```

```
new_derived_fact(Concl)    :-    if    Cond    then    Concl,
not(fact(Concl)), composed_fact(Cond).
```

```
composed_fact(Cond) :- fact(Cond).
composed_fact(Cond1 and Cond2) :- composed_fact(Cond1),
composed_fact(Cond2).
composed_fact(Cond1 or Cond2) :- composed_fact(Cond1);
composed_fact(Cond2).
```

```
if hall_wet and kitchen_dry then leak_in_bathroom.
if      hall_wet      and      bathroom_dry      then
problem_in_the_kitchen.
if window_closed or no_rain then no_water_from_outside.
if problem_in_the_kitchen and no_water_from_outside then
leak_in_the_kitchen.
```

```
fact(hall_wet).
fact(bathroom_dry).
fact(window_closed).
```

Klausa “:- dynamic fact/1” menyatakan bahwa klausa dengan relasi bernama *fact* dan memiliki 1 argumen bersifat dinamis. Ini berarti klausa tersebut dapat ditambah (dengan perintah *assert*) ataupun dibuang (dengan perintah *retract*) dari memori. Dalam *forward chaining*, klausa *dynamic* diperlukan karena terdapat kemungkinan munculnya *fact* baru dari *fact* yang telah ada dan perlu ditambahkan ke dalam memori untuk kemudian diakses kembali.

6.6 Reasoning

Reasoning adalah suatu cara memberikan penjelasan mengapa atau bagaimana sebuah kesimpulan diperoleh. Sebagai contoh, kesimpulan terdapat kebocoran di dapur diperoleh dari:

1. Terdapat masalah di dapur yang disimpulkan dari fakta *hall* yang basah dan kamar mandi yang kering.
2. Tidak ada air dari luar yang disimpulkan dari fakta tidak ada hujan.

Rules berikut ditambahkan untuk *backward chaining* agar program tersebut mampu memberikan *reasoning*.

```
:- op(800, xfx, <=) .

is_true(P, P) :- fact(P) .
is_true(P, P <= CondProof) :- if Cond then P,
is_true(Cond, CondProof) .
is_true(P1 and P2, Proof1 and Proof2) :- is_true(P1,
Proof1), is_true(P2, Proof2) .
is_true(P1 or P2, Proof) :- is_true(P1, Proof);
is_true(P2, Proof) .
```

Sehingga apabila diberikan *query* sebagai berikut:

```
?- is_true(leak_in_the_kitchen, How) .
How = (leak_in_the_kitchen <=
(problem_in_the_kitchen <= hall_wet and bathroom_dry) and
(no_water_from_outside <= window_closed)) .
```

maka akan diperoleh penjelasan bagaimana kesimpulan tersebut diperoleh.

6.7 Latihan

1. Representasikanlah uraian berikut dalam bentuk *IF-THEN rules*:
 - X adalah makhluk hidup jika X merupakan manusia, hewan, ataupun tumbuhan.
 - X adalah makhluk hidup, jika X bernafas dan dapat bergerak.
 - Mamalia, reptil, amfibi, ikan, dan burung adalah hewan.
 - Ikan bernafas dengan insang, memiliki sisik, ovipar, dan memiliki sirip.
 - Burung bernafas dengan paru-paru, ovipar, dan memiliki sayap.
 - Mamalia bernafas dengan paru-paru dan vivipar.
 - Reptil bernafas dengan paru-paru, memiliki sisik dan ovovivipar.
 - Amfibi bernafas dengan paru-paru dan vivipar.
 - Tumbuhan dapat menghasilkan oksigen.
 - Manusia dan hewan tidak dapat menghasilkan oksigen.
2. Ubahlah *IF-THEN rules* pada soal nomor 1 menjadi *rules* dalam Prolog dengan menambahkan operator buatan pengguna.

Bab VII

Semantic Network dan Frame

7.1 Pendahuluan

Semantic net dan *frame* adalah *framework* yang dipergunakan menyajikan pengetahuan. Kedua *framework* tersebut merepresentasikan pengetahuan bukan dalam bentuk representasi berbasis aturan namun menggunakan cara yang lebih terstruktur. Kedua *framework* ini juga menganut mekanisme *inheritance*.

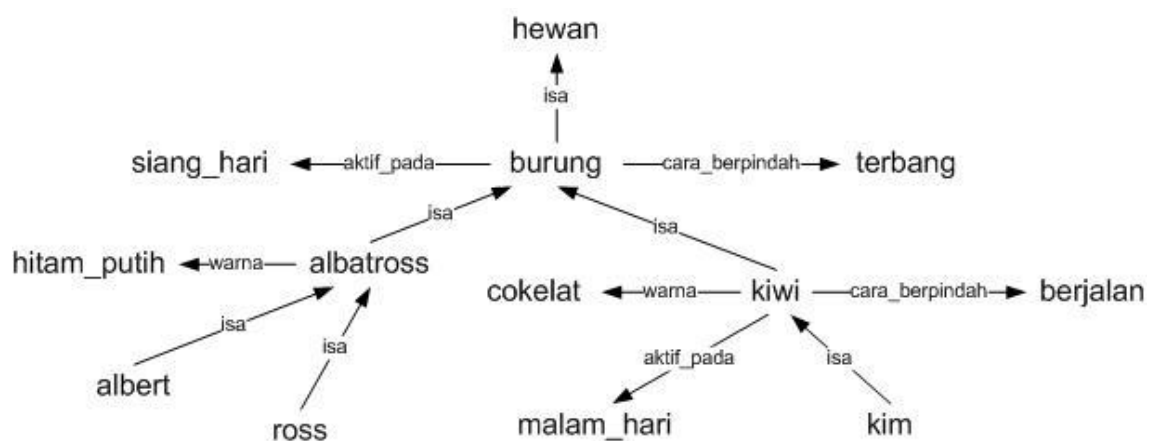
7.2 *Semantic Network*

Semantic network adalah sebuah representasi pengetahuan dalam bentuk *entity* dan relasi yang disajikan dalam bentuk graf. Sebagai contoh, terdapat *facts* sebagai berikut:

- Burung adalah hewan.
- Secara umum, burung aktif pada siang hari.
- Secara umum, burung berpindah dengan cara terbang.
- Albatross adalah sejenis burung.
- Albatros memiliki warna hitam putih.
- Albert adalah seekor albatross.
- Ross adalah seekor albatross.
- Kiwi adalah sejenis.
- Kiwi memiliki warna cokelat.
- Kiwi berpindah dengan cara terbang.
- Kiwi aktif pada malam hari.
- Kim adalah seekor kiwi.

Berdasarkan *facts* tersebut, kita dapat membuat sebuah graf yang merepresentasikan *semantic network*. Setiap *entity* dalam *facts* tersebut diwakili oleh *node* dan relasi diwakili oleh *link* dalam graf. *Link* dipergunakan untuk menghubungkan *node* yang satu

dengan *node* yang lain. Sebuah *node* dapat memiliki *slot* yang ditunjukkan oleh *link* yang memiliki nama. *Facts* tersebut dapat direpresentasikan dalam bentuk graf seperti pada Gambar 7.1.



Gambar 7.1 Contoh *semantic network*

Pada Gambar 7.1 yang termasuk *node* adalah burung, hewan, terbang, dan sebagainya. *Node* burung memiliki tiga *link* yang menyatakan terdapat tiga *slot* yaitu *isa* yang menyatakan relasi "adalah sejenis/seekor", *cara_berpindah* yang menyatakan relasi "cara berpindah tempat", dan *aktif_pada* yang menyatakan "waktu aktif". *Slot* *isa* berisi nilai hewan, *slot* *aktif_pada* berisi nilai siang_hari, dan *slot* *cara_berpindah* yang berisi nilai terbang.

Semantic network juga menganut konsep *inheritance* atau pewarisan sifat. *Inheritance* memungkinkan sebuah *instance* mewarisi sifat-sifat objek yang ada di kelasnya. Sebagai contoh, albert dan ross memiliki sifat dapat berpindah dengan terbang walaupun keduanya tidak memiliki *slot* *cara_berpindah* secara eksplisit. Hal ini dimungkinkan karena albert dan ross mewarisi *slot* *cara_berpindah* dari albatross dan albatross mewarisi *slot* *cara_berpindah* dari burung.

Semantic network tersebut dapat diimplementasikan dalam bentuk program sebagai berikut:


```
isa(burung, hewan).
```

```
isa(albatross, burung).
```

```
isa(kiwi, burung).
```

```
isa(kim, kiwi).
```

```
isa(ross, albatross).
```

```
isa(albert, albatross).
```

```
cara_berpindah(burung, terbang).
```

```
cara_berpindah(kiwi, berjalan).
```

```
cara_berpindah(X, Method) :- isa(X, SuperX),
cara_berpindah(SuperX, Method).
```

```
aktif_pada(burung, siang_hari).
```

```
aktif_pada(kiwi, malam_hari).
```

```
aktif_pada(X, Active) :- isa(X, SuperX),
aktif_pada(SuperX, Active).
```

```
warna(kiwi, coklat).
```

```
warna(albatross, hitam_putih).
```

```
warna(X, Color) :- isa(X, SuperX), warna(SuperX, Color).
```

```
fact(Fact) :- Fact, !.
```

```
fact(Fact) :- Fact =..[Rel, Arg1, Arg2], isa(Arg1,
SuperArg), SuperFact =..[Rel, SuperArg, Arg2],
fact(SuperFact).
```

Beberapa *query* yang dapat diajukan adalah:

```
?- fact(isa(ross, X)).
```

```
X = albatross.
```

```
?- fact(cara_berpindah(kim, X)).
X = berjalan.
```

```
?- aktif_pada(albert, When).
When = siang_hari .
```

Dalam program tersebut terdapat sebuah predikat yang disebut '*univ*' di mana predikat ini berfungsi untuk menguraikan sebuah *term* menjadi *term* baru. Sebagai contoh:

```
?- Fact =.. [umur, 20].
Fact = umur(20).

?- persegi_panjang(10, 5) =.. L.
L = [persegi_panjang, 10, 5].
```

7.3 Frame

Frame adalah kumpulan *node* dan *slot* dari *semantic network*. *Frame* dapat juga dianggap sebagai sebuah struktur data yang komponennya terdiri dari *slot*. Sama halnya dengan *semantic network*, *frame* juga memiliki karakteristik *inheritance*. Selain *inheritance*, *frame* juga memiliki karakteristik lain yaitu *demon*. *Demon* dapat dipergunakan untuk menghitung nilai dari sebuah *slot* apabila *slot* tersebut tidak memiliki nilai.

Informasi seperti yang dipergunakan dalam *semantic network* dapat dituliskan dalam bentuk *frame* sebagai berikut:

```
FRAME: burung
a_kind_of: hewan
cara_berpindah: terbang
aktif_pada: siang_hari
```

```
FRAME: albatross
a_kind_of: burung
warna: hitam_putih
ukuran: 115
```

```
FRAME: kiwi
a_kind_of: burung
cara_berpindah: berjalan
aktif_pada: malam_hari
warna: cokelat
ukuran: 40
```

Salah satu *instance* dari sebuah kelas yang ditulis dalam bentuk *frame* adalah:

```
FRAME: Albert
instance_of: albatross
ukuran: 120
```

Contoh implementasi dalam *frame* adalah:

```
% Frame burung
burung(a_kind_of, hewan).
burung(cara_berpindah, terbang).
burung(aktif_pada, siang_hari).
```

```
%Frame albatross
albatross(a_kind_of, burung).
albatross(warna, hitam_putih).
albatross(ukuran, 115).
```

```
%Frame kiwi
kiwi(a_kind_of, burung).
kiwi(cara_berpindah, berjalan).
```

```

kiwi(aktif_pada, malam_hari).
kiwi(ukuran, 40).
kiwi(warna, coklat).

%Frame Albert
albert(instance_of, albatross).
albert(ukuran, 120).

%Frame Ross
ross(instance_of, albatross).
ross(ukuran, 40).

%Frame hewan
hewan(relative_size,
execute(relative_size(Object, Value), Object, Value)).

value(Frame, Slot, Value) :- Query =..[Frame, Slot,
Value], call(Query), !.
value(Frame, Slot, Value) :- parent(Frame, ParentFrame),
value(ParentFrame, Slot, Value).
value(Frame, Slot, Value) :- value(Frame, Frame, Slot,
Value).
value(Frame, SuperFrame, Slot, Value) :- Query =..
[SuperFrame, Slot, Information], call(Query),
process(Information, Frame, Value), !.
value(Frame, SuperFrame, Slot, Value) :-
parent(SuperFrame, ParentSuperFrame), value(Frame,
ParentSuperFrame, Slot, Value).

process(execute(Goal, Frame, Value), Frame, Value) :- !,
call(Goal).
process(Value, _, Value).

```

```
parent(Frame, ParentFrame) :- (Query =..[Frame,
a_kind_of, ParentFrame]; Query =..[Frame, instance_of,
ParentFrame]), call(Query).
```

```
relative_size(Object, RelativeSize) :- value(Object,
ukuran, ObjSize), value(Object, instance_of, ObjClass),
value(ObjClass, ukuran, ClassSize), RelativeSize is
ObjSize/ClassSize * 100.
```

Query-query yang dapat diajukan adalah:

```
?- value(albert, aktif_pada, AlbertTime).
AlbertTime = siang_hari .
```

```
?- value(kiwi, aktif_pada, KiwiTime).
KiwiTime = malam_hari.
```

```
?- value(ross, relative_size, R).
R = execute(relative_size(_G1709, _G1710), _G1709,
_G1710) ;
R = 34.78260869565217 ;
false.
```

7.4 Latihan

1. Buatlah sebuah *knowledge base* berdasarkan uraian berikut:

Semua hewan mempunyai kulit. Ikan adalah sejenis hewan, burung adalah jenis hewan lainnya, dan mamalia adalah jenis hewan yang lain pula. Biasanya ikan mempunyai insang dan bisa berenang, sedangkan burung mempunyai sayap dan bisa terbang. Burung dan ikan biasanya bertelur, sedangkan mamalia tidak. Meskipun ikan hiu adalah sejenis ikan, ikan ini tidak bertelur—melainkan melahirkan anak. Salmon adalah jenis ikan lainnya dan ikan ini bisa dimakan. Kenari adalah sejenis burung dan berwarna kuning. Burung kasuari adalah sejenis burung yang sangat

tinggi tetapi tidak bisa terbang, hanya bisa berjalan. Mamalia umumnya bergerak dengan cara berjalan—misalnya kerbau. Selain menghasilkan susu, kerbau juga bisa dimakan. Akan tetapi, tidak semua mamalia bergerak dengan cara berjalan. Misalnya kelelawar, yang bergerak dengan cara terbang.

2. Gambarkanlah model *semantic network* untuk soal nomor 1.
3. Implementasikanlah *knowledge base* pada soal nomor 1 dalam bentuk *semantic network*.
4. Implementasikanlah *knowledge base* pada soal nomor 1 dalam bentuk *frame*.

Bab VIII

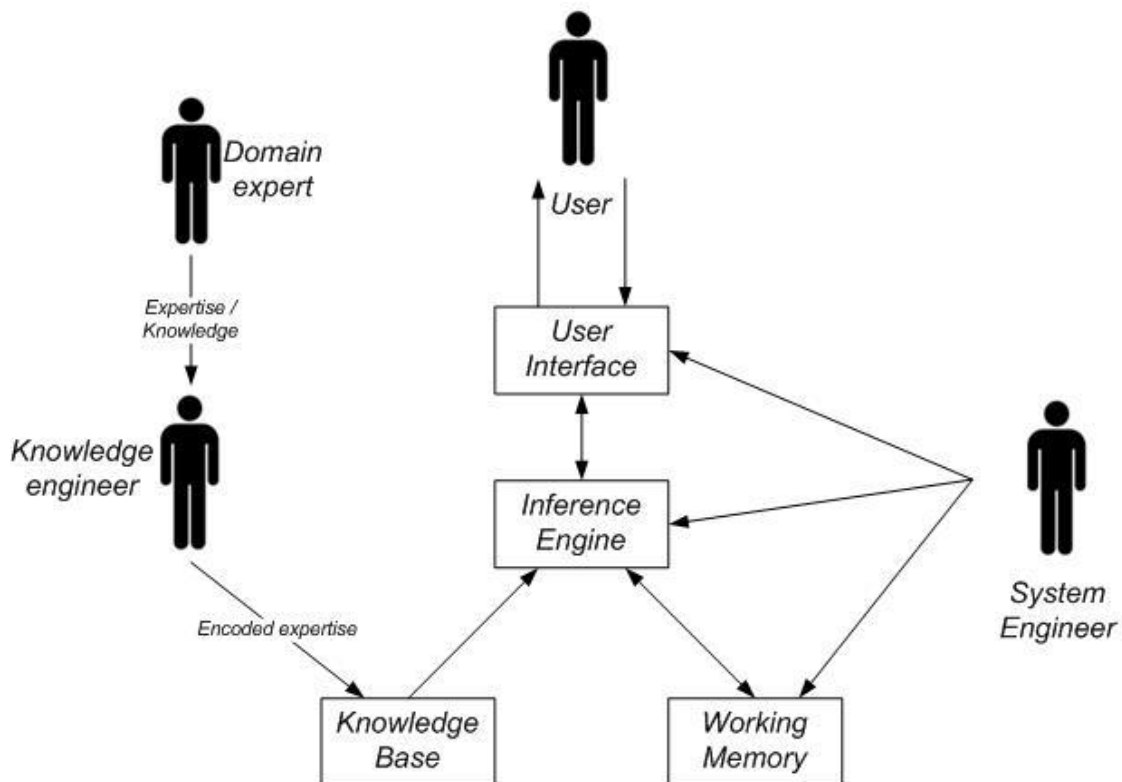
Sistem Pakar

8.1 Sistem Pakar

Sistem pakar adalah suatu aplikasi yang dapat dipergunakan untuk memecahkan suatu masalah dan memiliki kemampuan untuk mengambil keputusan layaknya seorang pakar.

8.2 Komponen Sistem Pakar

Sistem pakar memiliki komponen-komponen seperti yang terlihat pada Gambar 8.1.



Gambar 8.1 Komponen Sistem Pakar

Berdasarkan Gambar 8.1 komponen-komponen sistem pakar adalah:

1. *Knowledge base*: Database berisi pengetahuan yang diperoleh dari pakar yang telah diubah menjadi bentuk *rules*.
2. *Inference engine*: Sebuah *coding system* yang menghubungkan *rules* dalam *knowledge base* dengan data dalam *working memory*.
3. *Working memory*: Berisi data spesifik yang berhubungan dengan suatu masalah.
4. *User interface*: Penghubung antara pengguna dengan sistem.

Adapun individu-individu di dalam sistem pakar yaitu:

1. *User*: orang yang menggunakan sistem pakar.
2. *Domain expert*: orang yang memiliki pengetahuan tentang suatu domain masalah.
3. *Knowledge engineer*: orang yang menerima pengetahuan dari *domain expert* dan mengubahnya menjadi *rules* untuk kemudian disimpan dalam *knowledge base*.
4. *System engineer*: orang yang merancang *user interface* dan *working memory* serta mengintegrasikan *user interface*, *inference engine*, *knowledge base*, dan *working memory*.

8.3 Alur Kerja Sistem Pakar

Seorang *domain expert* yang memiliki pengetahuan tentang suatu domain masalah memberikan pengetahuannya kepada *knowledge engineer*. *Knowledge engineer* kemudian akan mengubah pengetahuan yang diperoleh dari *domain expert* menjadi *rules* yang dapat dipergunakan oleh *inference engine* dan menyimpannya dalam *knowledge base*. *System engineer* kemudian akan merancang *user interface* dan *working memory* lalu selanjutnya mengintegrasikan *user interface*, *inference engine*, *working memory*, dan *knowledge base*.

Ketika *user* menggunakan sistem pakar, *user* akan diberikan pertanyaan-pertanyaan yang berhubungan dengan domain masalah. Jawaban yang diberikan *user* akan disimpan dalam *working memory*. Setelah jawaban yang diberikan *user* telah mencukupi untuk menarik sebuah kesimpulan, *inference engine* akan mengambil semua data yang ada dalam *working memory*, lalu kemudian mencari kesimpulan mana yang sesuai dengan data tersebut berdasarkan *rules* dalam *knowledge base*. Setelah

memperoleh kesimpulan, *inference engine* akan memberikan kesimpulan tersebut kepada *user* melalui *user interface*.

8.4 Decision Tree

Decision tree atau pohon keputusan adalah sebuah *tree* yang menentukan bagaimana sebuah keputusan diambil. Keputusan yang diambil dengan menggunakan *decision tree* diperoleh dengan melakukan eliminasi terhadap jawaban-jawaban lain dengan menggunakan pertanyaan yang mempersempit ruang pencarian. *Leaf* dalam *decision tree* berisi jawaban atau keputusan yang berhasil diperoleh setelah menelusuri *tree*.

Seandainya terdapat *knowledge base* sebagai berikut:

*If it has fur **then** it's a mammal.*

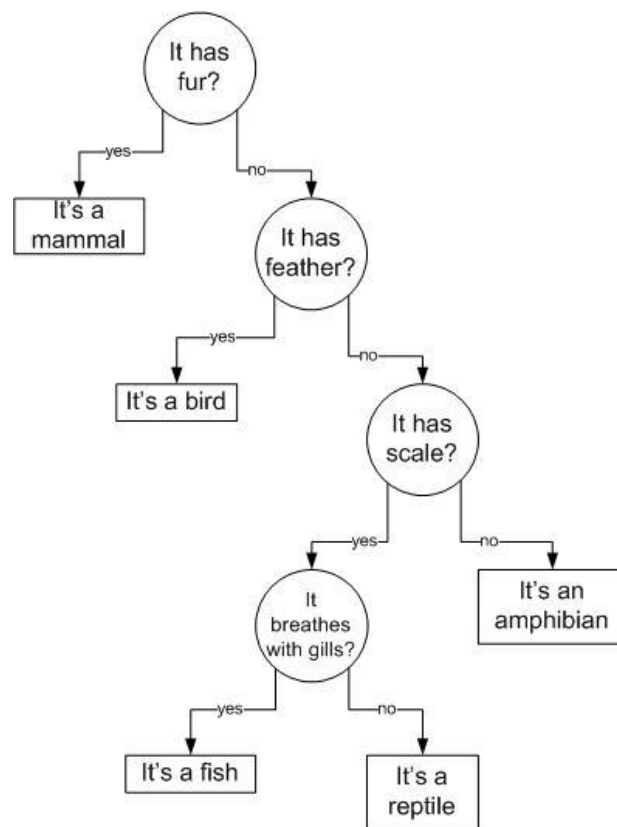
*If it has feathers **then** it's a bird.*

*If it has scales **and** it breathes with gills **then** it's a fish.*

*If it has scales **and** it doesn't breathe with gills **then** it's a reptile.*

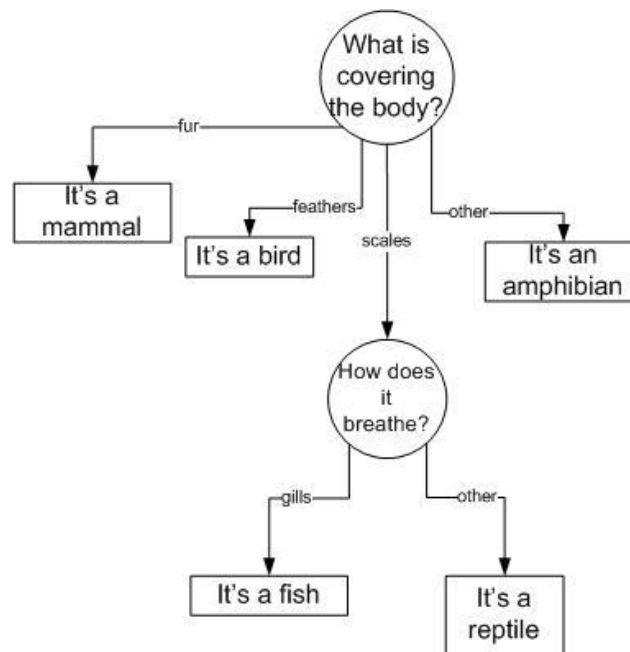
*If it doesn't have fur, feathers, **or** scales **then** it's an amphibian.*

Knowledge base tersebut dapat direpresentasikan dalam bentuk *decision tree* seperti pada Gambar 8.2a dan Gambar 8.2b.



Gambar 8.2a Binary Decision Tree

Sebuah *decision tree* dapat berbentuk *binary* ataupun *multiple branches*. Gambar 8.2a dan Gambar 8.2b merupakan *decision tree* untuk klasifikasi jenis hewan. *Binary decision tree* seperti pada Gambar 8.2a hanya mendukung dua jawaban yaitu *yes* dan *no*. Sedangkan *multiple branches binary tree* seperti pada Gambar 8.2b memungkinkan lebih dari dua jawaban untuk pertanyaan yang diajukan.



Gambar 8.2b Multiple Branches Decision Tree

8.5 Certainty Factor

Certainty factor adalah sebuah bentuk permodelan yang dipergunakan untuk mengatasi ketidakpastian dalam sebuah sistem berbasis *rules*. *Certainty factor* berfungsi untuk:

- Tingkat kepastian/ketidakpastian dari sebuah hipotesis H dengan bukti/gejala E.
- Tingkat kepercayaan/ketidakpercayaan terhadap sebuah bukti E.

Certainty factor dapat dihitung menggunakan Persamaan 1:

$$CF(H, E) = \frac{MB(H, E) - MD(H, E)}{1 - \min(MB(H, E) - MD(H, E))} \quad (1)$$

Dengan:

CF adalah *certainty factor* untuk hipotesis H yang disebabkan oleh bukti E.

MB adalah *measure of increased belief* untuk H yang disebabkan oleh E.

MD adalah *measure of increased disbelief* untuk H yang disebabkan oleh E.

$MB(H, E)$ adalah *measure of increased belief* yaitu tingkat peningkatan kepercayaan terhadap hipotesis H yang dihasilkan setelah observasi bukti E dan dapat dihitung dengan Persamaan 2:

$$MB(H, E) = \begin{cases} 1 & \text{jika } P(H) = 1 \\ \frac{P(H|E) - P(H)}{1 - P(H)} & \text{lainnya} \end{cases} \quad (2)$$

Demikian sebaliknya, $MD(H, E)$ adalah *measure of increased disbelief* yaitu tingkat peningkatan ketidakpercayaan terhadap hipotesis H yang dihasilkan setelah observasi bukti E dan dapat dihitung dengan Persamaan 3:

$$MD(H, E) = \begin{cases} 1 & \text{jika } P(H) = 0 \\ \frac{P(H) - P(H|E)}{P(H)} & \text{lainnya} \end{cases} \quad (3)$$

Di mana $P(H|E) - P(H)$ menunjukkan peningkatan kemungkinan H setelah memperoleh bukti E dan sebaliknya $P(H) - P(H|E)$ menunjukkan penurunan kemungkinan H setelah memperoleh bukti E .

Karakteristik dari MB , MD dan CF seperti diberikan pada Tabel 8.1.

Tabel 8.1 Karakteristik MB , MD , dan CF .

Karakteristik	Nilai
<i>Range nilai</i>	$0 \leq MB \leq 1$ $0 \leq MD \leq 1$ $-1 \leq CF \leq 1$
Hipotesis <i>certain true</i>	$MB = 1$
$P(H E) = 1$	$MD = 0$ $CF = 1$

Tabel 8.1 Karakteristik MB , MD , dan CF (lanjutan).

Karakteristik	Nilai
Hipotesis <i>certain false</i>	$MB = 0$

$P(H E) = 0$	$MD = 1$
	$CF = -1$
Tidak memiliki bukti(<i>evidence</i>)	$MB = 0$
$P(H E) = P(H)$	$MD = 0$
	$CF = 0$

Certainty factor memiliki nilai di antara -1 sampai dengan 1. Nilai positif menunjukkan kecondongan terhadap hipotesis sedangkan nilai negatif menunjukkan kecondongan pada negasi hipotesis. Nilai *certainty factor* dapat diinterpretasikan seperti pada Tabel 8.2.

Tabel 8.2 Interpretasi nilai *certainty factor*.

<i>Term</i>	<i>Certainty Factor</i>
<i>Definitely not</i>	-1.0
<i>Almost certainly not</i>	-0.8
<i>Probably not</i>	-0.6
<i>Maybe not</i>	-0.4
<i>Unknown</i>	-0.2 sampai +0.2
<i>Maybe</i>	+0.4
<i>Probably</i>	+0.6
<i>Almost certainly</i>	+0.8
<i>Definitely</i>	+1.0

Adapun metode untuk mengkombinasikan beberapa bukti dalam *antecedent* sebuah *rule* seperti pada Tabel 8.3.

Tabel 8.3 Kombinasi beberapa bukti.

Bukti (<i>Evidence</i>) E	<i>Antecedent Certainty</i>
$E_1 \text{ AND } E_2$	$\min [CF(E_1, e), CF(E_2, e)]$
$E_1 \text{ OR } E_2$	$\max [CF(E_1, e), CF(E_2, e)]$
NOT E	$-CF(E, e)$

Perhitungan *certainty factor* untuk aturan:

IF E THEN H

dapat dihitung dengan Persamaan 4:

$$CF(H, e) = CF(E, e) \cdot CF(H, E) \quad (4)$$

dengan:

$CF(E, e)$ adalah *certainty factor* untuk bukti e di mana e merupakan bukti yang belum pasti yang menggantikan bukti E.

$CF(H, E)$ adalah *certainty factor* dari hipotesis H dengan asumsi bahwa bukti yang ada telah pasti atau $CF(E, e) = 1$.

$CF(H, e)$ adalah *certainty factor* untuk hipotesis H berdasarkan bukti tidak pasti e.

Certainty factor untuk sebuah hipotesis dapat digabung baik secara seri maupun paralel.

1. Penggabungan *certainty factor* secara seri.

Penggabungan nilai *certainty factor* secara seri dilakukan jika dengan menggunakan *rule* R_1 dan bukti E_1 kita dapat menarik hipotesis H_1 dan H_1 dipergunakan sebagai bukti untuk *rule* R_2 untuk menarik hipotesis H_2 atau mengikuti aturan:

R_1 : IF E_1 THEN H_1 . (CF_1)

R_2 : IF H_1 THEN H_2 . (CF_2)

maka kita dapat melakukan penggabungan nilai *certainty factor* secara seri sehingga kita dapat menarik hipotesis H_2 berdasarkan bukti E_1 sebagai berikut:

IF E_1 THEN H_2 . ($CF_{combine}$)

Perhitungan nilai $CF_{combine}$ seperti dalam Persamaan 5:

$$CF_{combine} = CF_1 \cdot CF_2 \quad (5)$$

2. Penggabungan *certainty factor* secara paralel.

Penggabungan nilai *certainty factor* secara paralel dilakukan jika sebuah hipotesis H dapat diperoleh melalui lebih dari satu *rule* dengan nilai *certainty factor* yang berbeda atau mengikuti aturan:

R_1 : IF E_1 THEN H. (CF_1)

R_2 : IF E_2 THEN H. (CF_2)

Penggabungan *certainty factor* secara paralel mengikuti Persamaan 6:

$$CF_{combine} = \begin{cases} CF_1 + CF_2 - CF_1 \cdot CF_2 & CF_1, CF_2 > 0 \\ \frac{CF_1 + CF_2 + CF_1 \cdot CF_2}{1 - \min[|CF_1|, |CF_2|]} & CF_1, CF_2 < 0 \end{cases} \quad (6)$$

8.6 Contoh

Misalkan terdapat sebuah kasus sebagai berikut:

An animal which has fur, breathes with lungs, and reproduces by giving birth is almost certainly a mammal.

Kasus tersebut dapat diartikan sebagai berikut:

Seekor binatang yang memiliki bulu, bernafas dengan paru-paru, dan bereproduksi dengan melahirkan hampir dipastikan merupakan seekor mamalia.

Kasus tersebut dapat diubah menjadi *IF-THEN rule* sebagai berikut:

IF

it has fur AND

it breathes with lungs AND

it reproduces by giving birth

THEN

it is a mammal. (CF=0,8)

Berdasarkan *rule* tersebut kita dapat memperoleh tiga bukti dan sebuah hipotesis yaitu:

E_1 : *it has fur.*

E_2 : *it breathes with lungs.*

E_3 : *it reproduces by giving birth.*

H : *it is a mammal.*

$$CF(H, E) = CF(H, E_1 \cap E_2 \cap E_3) = 0,8$$

Dalam hal ini, CF=0,8 merupakan *attenuation factor* yaitu nilai *certainty factor* yang dipergunakan jika semua bukti (E_1 , E_2 , dan E_3) diketahui dengan pasti.

$$CF(H, E_1) = CF(H, E_2) = CF(H, E_3) = 1$$

Apabila bukti tidak diketahui dengan kepastian, maka nilai CF=0,8 tidak dapat dipergunakan dan harus dihitung kembali. Sebagai contoh:

E_1 : **maybe** *it has fur.*

E_2 : **probably** *it breathes with lungs.*

E_3 : *it reproduces by giving birth.*

Perubahan pada bukti E_1 dan E_2 mengakibatkan perubahan nilai *certainty factor* untuk masing-masing bukti tersebut menjadi:

$$CF(E_1, e) = 0,4$$

$$CF(E_2, e) = 0,6$$

$$CF(E_3, e) = 1$$

Selanjutnya berdasarkan aturan pada Tabel 8.3 dan Persamaan 4 nilai *certainty factor* dapat dihitung kembali sebagai berikut:

$$\begin{aligned} CF(E,e) &= CF(E_1 \cap E_2 \cap E_3) = \min[CF(E_1,e), CF(E_2,e), CF(E_3,e)] \\ &= \min[(0,4), (0,6), (1)] = 0,4 \\ CF(H,e) &= CF(E,e) \cdot CF(H,E) = 0,4 \cdot 0,8 = 0,32 \end{aligned}$$

Contoh kasus lainnya adalah sebagai berikut:

*Mr. Holmes receives a telephone call from his neighbor Mr. Watson stating that he hears an alarm sound from the direction of Mr. Holmes' house. If there's an alarm, **probably** there's a buglar in his house. Preparing to rush home, Mr. Holmes recalls that Mr. Watson is known to be a tasteless practical joker and **maybe** he is playing a prank. Therefore, he decides to first call his other neighbor, Mrs. Gibbons, whose information is **almost certainly** reliable.*

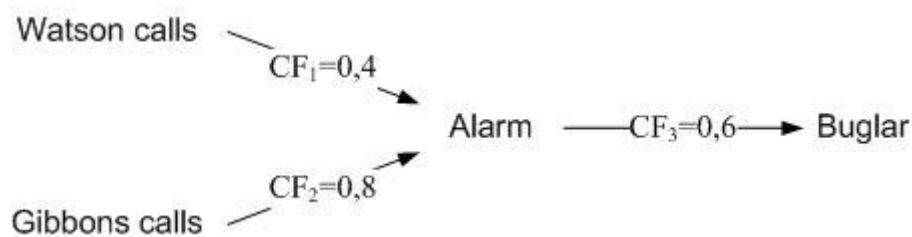
Berdasarkan kasus tersebut kita dapat membuat *IF-THEN rules* sebagai berikut:

R₁: IF Watson calls THEN there's an alarm. (CF₁=0,4)

R₂: IF Gibbons calls THEN there's an alarm. (CF₂=0,8)

R₃: IF there's an alarm THEN there's a buglar. (CF₃=0,6)

Rules tersebut dapat digambarkan dalam *inference network* seperti pada Gambar 8.3.



Gambar 8.3 *Inference network* berdasarkan *IF-THEN rules* **R₁**, **R₂**, dan **R₃**.

Pada Gambar 8.3 terlihat bahwa hipotesis alarm berbunyi dapat diperoleh dari dua *rules* yang berbeda yaitu apabila Watson menelepon dan apabila Gibbons menelepon. Oleh karena itu, kita dapat menggabungkan nilai kedua CF tersebut sesuai Persamaan 6:

R₄: IF Watson calls and Gibbons calls THEN there's an alarm. (CF₄)

$$CF_1, CF_2 > 0$$

$$CF_4 = CF_1 + CF_2 - CF_1 \cdot CF_2 = (0,4) + (0,8) - (0,4)(0,8) = 0,88$$

Selanjutnya kita dapat menggabungkan *rules* R₄ dan R₃ sehingga diperoleh R₅ menggunakan penggabungan CF secara seri seperti pada Persamaan 5 sebagai berikut:

R₄: IF Watson calls and Gibbons calls THEN there's an alarm. (CF₄=0,88)

R₃: IF there's an alarm THEN there's a buglar. (CF₃=0,6)

R₅: IF Watson calls and Gibbons calls THEN there's a buglar. (CF₅)

$$CF_5 = CF_4 \cdot CF_3 = (0,88)(0,6) = 0,528$$

8.7 Latihan

1. Buatlah sebuah sistem pakar untuk menyelesaikan masalah tertentu. Contoh:
 - a. Diagnosa penyakit.
 - b. Rekomendasi pariwisata.
 - c. dan sebagainya.

Dengan syarat:

- Memiliki *knowledge base* dan *working memory*.
- Memiliki *user interface* yang terintegrasi dengan *knowledge base* dan *working memory* serta Prolog sebagai *inference engine*.
- Sistem tersebut mampu memberikan *reasoning* terhadap suatu kesimpulan yang diperoleh.
- Menerapkan perhitungan *certainty factor* untuk setiap *rules* dan kesimpulan yang diperoleh.
- Menerap metode *backward chaining* atau *forward chaining* untuk mengambil keputusan.

Selanjutnya, sertakan pula laporan untuk sistem tersebut yang memuat:

- Pengetahuan yang diperoleh dari *domain expert*.
- Hasil pengolahan pengetahuan tersebut menjadi *IF-THEN rules*.
- *Decision tree* yang memuat bagaimana kesimpulan diperoleh sesuai dengan *rules* yang ada.
- Hasil perhitungan *certainty factor* untuk setiap kondisi dan kesimpulan yang ada berdasarkan pengolahan pengetahuan dari *domain expert*.
- Bentuk implementasi *IF-THEN rules* tersebut dalam *knowledge base*.

Daftar Pustaka

- Benkard, M. 2013. *Implementation of a PROLOG-like Logic Programming Language Based on B&D Search*. Institut Für Informatik: München.
- Blackburn, P., Bos, J., & Striegnitz, K. 2001. *Learn Prolog Now!*. (Online) <http://cs.union.edu/~striegnk/learn-prolog-now/html/index.html> (16 Maret 2015).
- Heckerman, D. 1987. *The Certainty Factor Model*. Departments of Computer Science and Pathology University of Southern California: Los Angeles.
- Negnevitsky, M. 2005. *Artificial Intelligence: A Guide to Intelligent System*. Pearson Education Limited: Edinburgh Gate.
- Sitompul, O. S. 2013. *Prolog : Pengantar Teknik Pemrograman Kecerdasan Buatan*. USU Press: Medan.
- Wielemaker, J. 2017. Swi-Prolog 7.4 Reference Manual. Creative Commons Attribution-Share Alike 3.0 Unported License : California.
- Arzaki, M. 2015. *Pemograman Logika dengan Prolog*. Fakultas Informatika Telkom University : Bandung.

SWI Prolog

1.8 Pengenalan SWI Prolog

Tahun 1987 merupakan awal dimulainya pengembangan SWI-Prolog. SWI-Prolog merupakan *software* gratis yang dapat digunakan untuk mengimplementasikan bahasa pemrograman prolog. *Software* ini telah banyak digunakan dalam berbagai penelitian dan pendidikan serta beberapa aplikasi komersial.

Sociaal-Wetenschappelijke Informatica (“*Social Science Informatics*”) adalah akronim dari SWI, merupakan nama lain dari grup riset *Human-Computer Studies* di *University of Amsterdam*, Belanda. SWI-Prolog ditulis oleh Jan Wielemaker (Arzaki, 2015).

Kebanyakan implementasi dari bahasa prolog yang dirancang hanya untuk melayani satu set terbatas dalam sebuah kasus. SWI-Prolog tidak ada pengecualian untuk aturan ini. Posisi SWI-Prolog itu sendiri terutama sebagai sebuah lingkungan prolog untuk ‘ pemrograman dalam skala besar ’ dan penggunaannya pada suatu kasus dimana ia memainkan peran utama dalam sebuah aplikasi, yaitu bertindak sebagai perekat antara komponen. Pada saat yang sama, SWI-Prolog bertujuan untuk menyediakan lingkungan produksi *prototyping* yang cepat.

1.9 Instalasi SWI-Prolog

Untuk proses instalasi SWI-Prolog, aplikasi ini dapat diunduh langsung secara gratis di lingkungan sistem operasi Windows dan Macintosh dengan cara mengunduh file instalasi yang terdapat pada link <http://www.swi-prolog.org/download/stable>. Instalasi dapat dilakukan secara mudah seperti instalasi aplikasi atau program pada umumnya khususnya instalasi pada sistem operasi Windows XP/ Vista/ 7/ 8/ 10 baik 32 maupun 64-bit. Pengguna juga dapat menentukan lokasi atau direktori instalasi sesuai dengan kebutuhan, contohnya di C:\Program Files\swipl.

Namun proses penginstalan SWI-Prolog di sistem operasi LINUX terdapat perbedaan. Instalasi SWI-Prolog dapat dilakukan via PPA (*Personal Package Archive*). Buka Terminal, kemudian ketikkan perintah sebagai berikut:

```
$ sudo apt-add-repository ppa:swi-prolog/stable
```

```
$ sudo apt-get update
$ sudo apt-get install swi-prolog
```

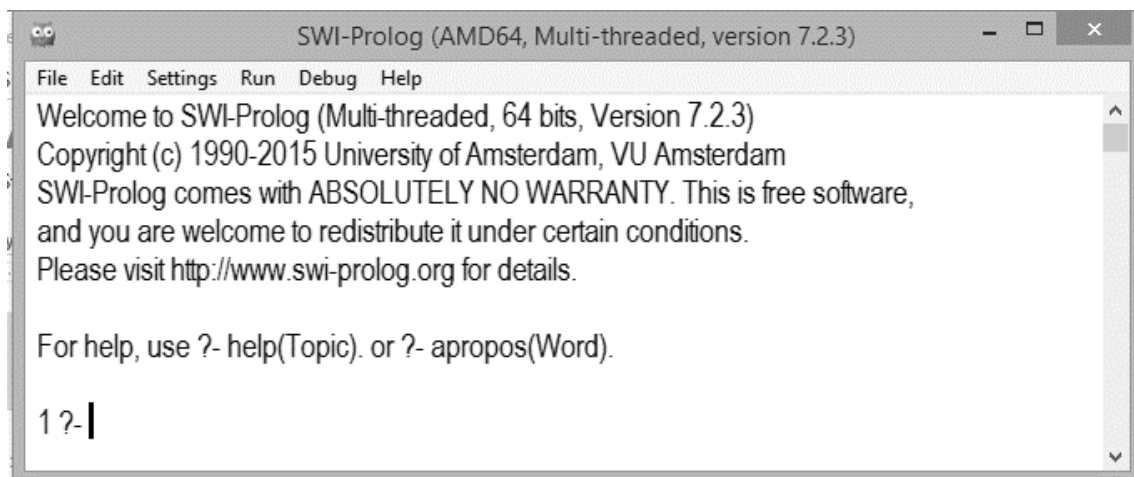
Setelah Instalasi, untuk dapat menjalankan SWI-Prolog melalui terminal dapat dilakukan dengan mengetikkan perintah berikut:

```
$ swipl
?- emacs.
```

Perintah `?- emacs.` ini berfungsi untuk menjalankan GUI pada linux yang serupa dengan GUI pada windows.

Instalasi untuk sistem operasi Linux dengan distro selain Ubuntu dapat dilakukan dengan mengunjungi situs <http://www.swi-prolog.org/build/LinuxDistro.txt>.

1.10 Pemakaian SWI-Prolog (Interpreter).



Gambar L1. Tampilan *interpreter* pada swipl

Ketika membuka SWI-Prolog maka akan muncul tampilan seperti diatas yang disebut sebagai Interpreter. Pada Interpreter biasa diberikan *queri* – *queri* atau perintah untuk mendapatkan suatu hasil. Contoh ketika mengetikkan perintah berikut pada interpreter:

```
1 ?-write('Hello World'),nl, write('ini mencoba Prolog').
```

Maka SWI-Prolog akan mengeluarkan:

```
Hello World
Ini mencoba Prolog
```

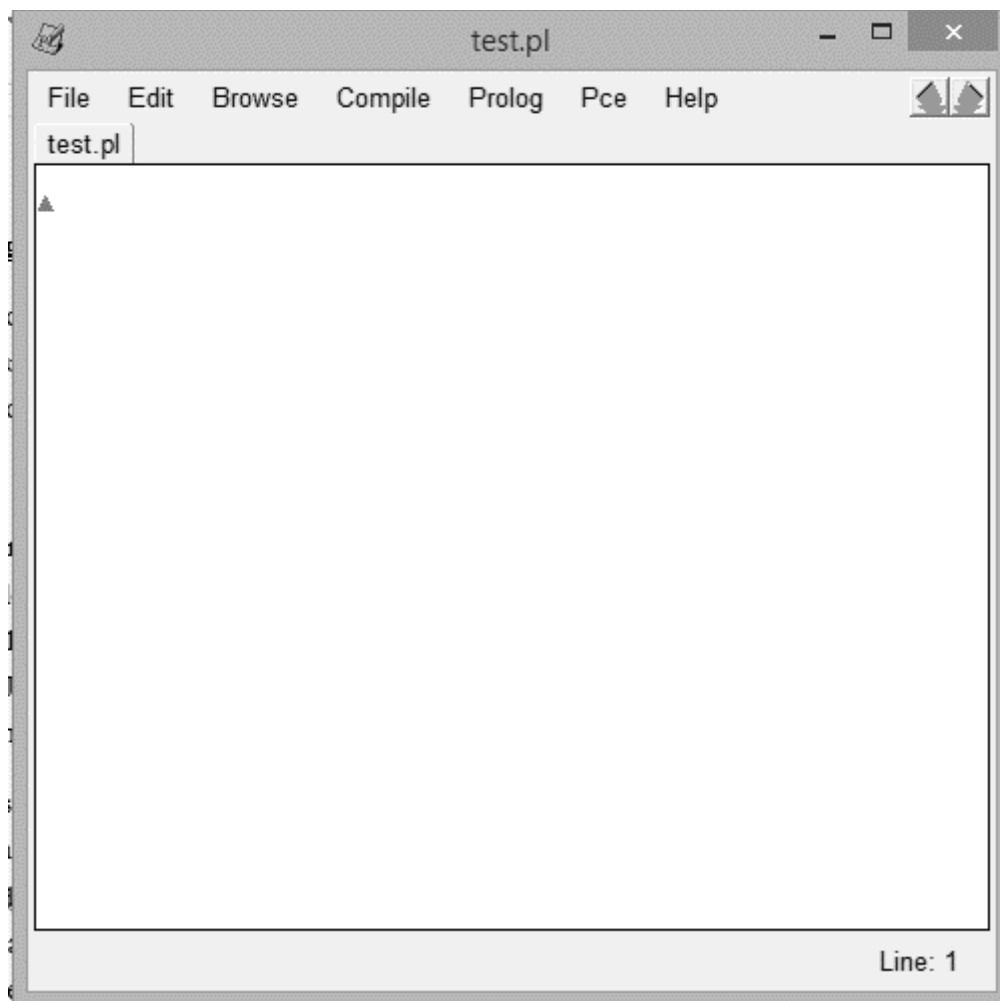
Perintah *write* dan *nl* merupakan dua contoh *build-in-predicate* (BIP) pada SWI-Prolog, *write('xxx')* berfungsi untuk mengeluarkan tulisan *xxx* dan *nl* berfungsi untuk memindahkan keluaran berikutnya ke baris yang baru. Dan yang harus diingat perintah prolog harus diakhiri oleh tanda titik (.).

Berikut beberapa penjelasan submenu yang terdapat pada menu utama interpreter:

- *File (Reload Modified files)*
Fungsi submenu ini adalah untuk memuat ulang semua sumber file yang telah dimodifikasi dengan menggunakan perintah *make/0*.
- *File (Navigator)*
Membuka *explorer* seperti melihat file prolog dan predikat didalamnya.
- *Setting (Font)*
Memungkinkan untuk mengubah font konsol.
- *Setting (User init file)*
Mengedit file personalisasi pengguna. Jika tidak ada file tersebut, maka pertama kali menginstall file *default* sebagai *.pl* yang berisi pengaturan umum digunakan dalam komentar..
- *Setting (Stack Sizes)*
Memungkinkan untuk mendefinisikan ukuran maksimum dimana berbagai tumpukan prolog yang dibiarkan berkembang.
- *Run (Interrupt)*
Berfungsi untuk mencoba menghentikan proses prolog yang sedang berjalan.
- *Help*
Menu bantuan yang menyediakan berbagai titik awal untuk dokumen terkait. Biasanya item atau submenu ditandai dengan (www) kemudai membuka browser internet bawaan dan mengarahkannya ke halaman website SWI-Prolog.

1.11 Menulis program pada SWI-Prolog

Cara penulisan program prolog pada SWI-Prolog dapat dilakukan dengan menggunakan editor teks apapun (seperti notepad atau notepad++), namun SWI-Prolog juga telah menyediakan editor yang dapat memeriksa syntax program prolog yang dibuat.



Gambar L2. Tampilan *text editor* pada swipl

Cara membuat skrip program prolog pada windows dapat dilakukan dengan membuka aplikasi SWI-Prolog, kemudian pilih menu *File* lalu pilih submenu *New* kemudian beri nama file sesuai kebutuhan (pastikan jenis file adalah *Prolog Source* atau program berekstensi *.pl*). Jika nama *file* sudah ditentukan maka teks editor yang muncul siap digunakan untuk menuliskan program prolog.

Cara membuat skrip program prolog pada linux ubuntu melalui terminal dapat dilakukan dengan mengetikkan perintah *swipl*, setelah masuk ke SWI-Prolog ketikkan perintah *emacs*. (dengan diakhiri tanda baca titik). Langkah berikutnya adalah kemudian pilih menu *File* lalu pilih submenu *New* kemudian beri nama file sesuai kebutuhan (pastikan jenis file adalah *Prolog Source* atau program berekstensi *.pl*). Jika nama *file* sudah ditentukan maka teks editor yang muncul siap digunakan untuk menuliskan program prolog.

1.12 Cara Menjalankan Program prolog

Baik di sistem operasi windows dan linux cara menjalankan program pada editor teks dapat dilakukan dengan memilih menu *Compile* lalu pilih submenu *Make* dan setelah itu pilih menu *Compile* lalu pilih submenu *Compile Buffer* (jika terdapat kesalahan pada penulisan sintaksis program biasanya akan muncul pop-up warning atau error). Jika berhasil maka *Interpreter* (atau terminal pada Linux) dapat mengeluarkan peringatan apakah proses kompilasi telah sukses dilakukan atau error karena terdapat kesalahan seperti penulisan sintaksis program prolog.

1.13 Menulis dan menjalankan program dengan SWIPL GUI Version

1. Buatlah sebuah file prolog

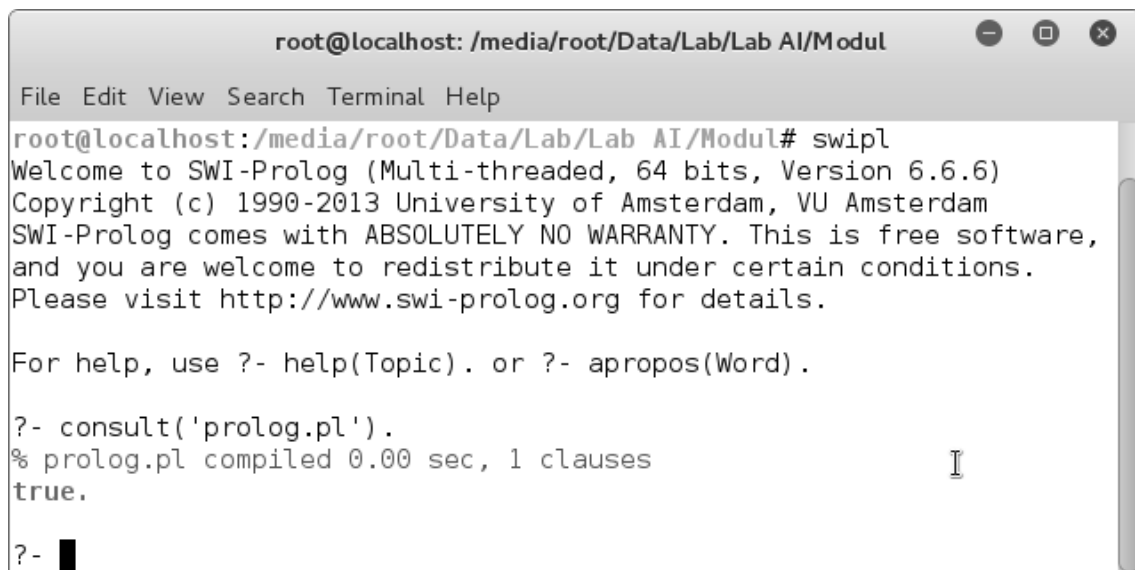
```
root@localhost:~# touch nama_file.pl
```

2. Isilah file tersebut dengan perintah prolog

```
root@localhost:~# nano nama_file.pl
> write('Hello World'),nl, write('ini mencoba Prolog').
```

3. Terakhir simpan file tersebut dan jalankan interpreter swipl

```
root@localhost:~# swipl
> consult('nama_file').
```



```
root@localhost: /media/root/Data/Lab/Lab AI/Modul
File Edit View Search Terminal Help
root@localhost:/media/root/Data/Lab/Lab AI/Modul# swipl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.6.6)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- consult('prolog.pl').
% prolog.pl compiled 0.00 sec, 1 clauses
true.

?- 
```

Gambar L3. Tampilan interpreter pada SWIPL GUI Version

Praktikum Artificial Intelligence

Nama : _____
 NIM : _____
 Kom : _____
 Tanggal : _____
 Pertemuan ke : _____
 Topik : _____

Nilai

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

Lined writing area with horizontal ruling lines.



Opm Salim Sitompul pertama kali mengenal dunia pemrograman pada tahun 1984 ketika mengikuti kursus Pengenalan Komputer dan Teknik Pemrograman di Pusat Komputer USU Medan. Ketika itu beliau masih berstatus sebagai mahasiswa jurusan Matematika FMIPA USU. Setelah memperoleh gelar sarjananya pada Tahun 1985, beliau merintis kariernya sebagai dosen di jurusan

FMIPA USU Medan pada tahun 1987. Obsesinya untuk menjadi ahli dalam bidang ilmu komputer membawanya ke Program Studi Pascasarjana S2 Ilmu Komputer Universitas Indonesia, Jakarta pada tahun 1988 yang bekerja sama dengan University of Maryland, University College, College Park, USA pada tahun 1989 guna mengikuti kuliah-kuliah master selama satu tahun. Gelar Magister Ilmu Komputer diraihinya pada tahun 1990. Pada tahun 1992, beliau mengikuti berbagai short course di University of Colorado, Division of Continuing Education Boulder, Colorado, USA seperti Introduction to Unix, Intermediate Unix, Introduction to C Programming, Intermediate C Programming, Introduction to C ++ Programming, dan Intermediate C ++ Programming. pada tahun 1994 beliau juga mengikuti pelatihan The Third Country Training Programme in Information Technology, Structured System Analysis and Design Methodology (SSADM) selama tiga bulan di Institute of Computer Technology, University of Colombo, Sri Lanka. Beliau memperoleh gelar Doktor dalam bidang Information Science and Department of Information Science Universiti Kebangsaan Malaysia



Erna Budhiarti Nababan menyelesaikan S1 (pendidikan Teknik Elektro - Arus Lemah) tahun 1985, lalu mengajar di TTUC Bandung untuk mata kuliah Teknik Digital. Tahun 1987, mengajar mata kuliah programming BASIC dan PASCAL di Jurusan Pendidikan Teknik Elektro IKIP Bandung (Sekarang UPI). Pindah Ke Medan Dan Mengajar di STT Harapan (1998) untuk review mata kuliah Programming C. Melanjutkan studi dalam bidang Teknologi Informasi jurusan Industrial Computing dengan kekhususan Kecerdasan Buatan dan menyelesaikan program Doktor dalam Science dan Management System, masih dalam bidang Teknologi Informasi, dengan kekhususan Kecerdasan Buatan dalam permasalahan optimasi. Sejak 2010 hingga sekarang, mengajar Di S1 Dan S2, Universitas Sumatera Utara untuk matakuliah Kecerdasan Buatan, Kecerdasan Komputasional dan Soft Computing.

Buku Dasar - Dasar Kecerdasan Buatan dengan Menggunakan Prolog ini dibuat sebagai acuan praktikum mata kuliah Kecerdasan Buatan dan Sistem Cerdas dengan menggunakan bahasa pemrograman Prolog. Melalui buku ini penulis berharap pembaca mampu memahami konsep pemrograman deklaratif Prolog dan mampu merancang sistem cerdas sesuai dengan materi-materi yang ada dalam buku ini. Penulis juga meng harapkan kritik dan saran pembaca sehingga buku ini dapat menjadi le-

ISBN 979-458-960-8



9 789794 589601

9 0000

usupress.usu.ac.id