Algoritma Euclidean dan Struktur Data Pohon dalam Bahasa Pemrograman LISP

Ahmad Ayyub Mustofa
Jurusan Teknik Informatika ITB, Bandung 40132, email: rekka_zan@students.itb.ac.id

Abstraksi – *Bahasa pemrograman LISP adalah bahasa pemrograman yang mengikuti kaidah pada Algoritma Euclidean. Bahasa pemrograman ini menggunakan struktur data sederhana yang disebut sebagai list, bersama dengan operator-operator sederhana dan notasi fungsi. Dengan adanya struktur list tersebut, struktur algoritma dalam bahasa pemrograman LISP dapat direpresentasikan dalam bentuk pohon (tree). Makalah ini membahas mengenai Algoritma Euclidean, struktur tree, dan aplikasinya dalam bahasa LISP.*

Kata Kunci*: algoritma, euclid, tree, LISP*

1. PENDAHULUAN

Dalam ilmu matematika terdapat sebuah algoritma terkenal yang biasa digunakan untuk menemukan PBB (Pembagi Bersama Terbesar) dari dua buah bilangan. Algoritma ini ditemukan oleh Euclid, seorang matematikawan Yunani, berabad-abad yang lampau dan kemudian didokumentasikan dalam bukunya yang berjudul "Element". Algoritma ini direpresentasikan sebagai berikut [1]:

1. Jika n = 0 maka
   m adalah PBB (m, n)
   stop
   tetapi jika n ≠ 0
   lanjutkan ke langkah 2
2. Bagilah m dengan n dan misalkan r adalah sisanya
3. Ganti nilai m dengan nilai n dan nilai n dengan nilai r, lalu ulang langkah 1

Kemudian pada tahun 1960, John McCarthy menggunakan dasar yang sama dengan Algoritma Euclidean tersebut untuk membuat sebuah bahasa pemrograman. Beliau berhasil menunjukkan bahwa hanya dengan operator-operator sederhana dan notasi pembentuk fungsi, sebuah bahasa pemrograman yang utuh dapat dibentuk. Bahasa ini kemudian diberi nama LISP, singkatan dari "List Processing."Bahasa ini disebut demikian karena pokok pikiran utama dari bahasa pemrograman ini adalah penggunaan struktur data sederhana yang disebut list [2].

2. PEMBAHASAN

2.1 Struktur Data dalam Pemrograman LISP

Struktur pemrograman LISP terdiri kode-kode dan data yang membentuk suatu ekspresi. Ekspresi ini dapat berupa sebuah atom, yaitu deretan satu atau beberapa karakter, atau sebuah list yang terdiri dari beberapa ekspresi dan boleh kosong. Di bawah ini beberapa contoh ekspresi dalam LISP :

aaa
()
(aaa)
(abc def)
(a b (c) d)

LISP bekerja dengan menempatkan sebuah ekspresi sebagai operator di awal fungsi, dan diikuti oleh argumen-argumen atau operand-operand di belakangnya. Secara umum direpresentasikan sebagai berikut :

(operator operand1 operand2 operand3 ...)

Operand yang dioperasikan dapat berupa sebuah list yang di dalamnya terdapat sebuah fungsi lain. Apabila kita wakilkan operator dengan X dan operand dengan bilangan bulat maka bentuk diatas dapat menjadi sebagai berikut :

(X 1 2 3 ...)

Dimana 1 mengandung (X 4 5 6 ...) dan memberikan bentuk keseluruhan (X (X 4 5 6 ...) 2 3 ...). Ini merupakan konsep yang sama dengan operasi dalam Algoritma Euclidean, sederhana tapi sangat efisien untuk digunakan.

2.2 Algoritma Euclidean dalam LISP

Berikut ini adalah realisasi Algoritma Euclidean dalam notasi *pseudo-code* [1]:

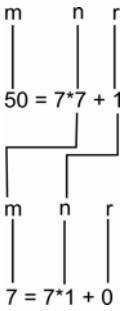
```
procedure Euclidean (input m, n : integer, output PBB : integer)
{
Mencari PBB (m, n) dengan syarat m dan n bilangan tak negatif dan m ≥ n
Masukan : m dan n, m ≥ n, dan m, n ≥ 0
}
Deklarasi
r : integer
Algoritma
while n ≠ 0 do
        r ← m mod n
        m ← n
        n ← r
```

endwhile
{
n = 0, maka PBB (m, n) = m
}
PBB ← m

Untuk lebih jelasnya, berikut ini adalah contoh aplikasi algoritma tersebut. Kita berikan masukan untuk algoritma tersebut bilangan integer 50 dan 7. Maka hasilnya adalah sebagai berikut :

m ← 50
n ← 7
n ≠ 0
    r ← 50 mod 7 = 1
    m ← 7
    n ← 1
n ≠ 0
    r ← 7 mod 1 = 0
    m ← 1
    n ← 0
n = 0
    stop
PBB = m

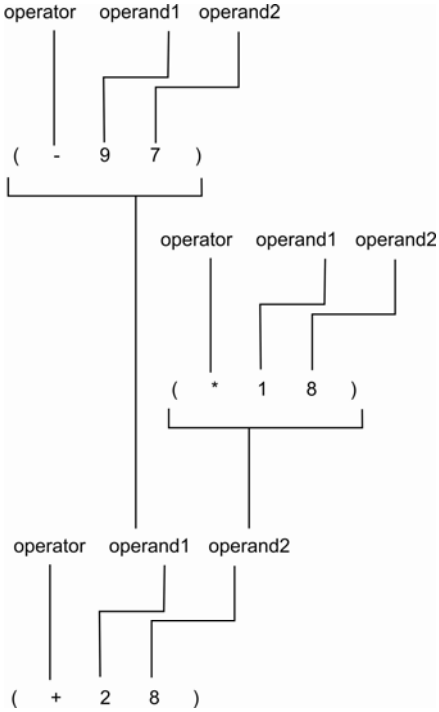
Apabila prosedur di atas kita gambarkan dalam sebuah bagan, akan menjadi seperti ini :



Aplikasi dari Algoritma Euclidean ini dapat kita lihat ketika kita memroses sebuah fungsi dalam LISP dimana operand yang ada mengandung fungsi lain yang harus dioperasikan terlebih dahulu. Kita akan melihat contoh kasusnya pada operasi aritmatika dengan input berikut :

(+ (- 9 7) (* 1 8))

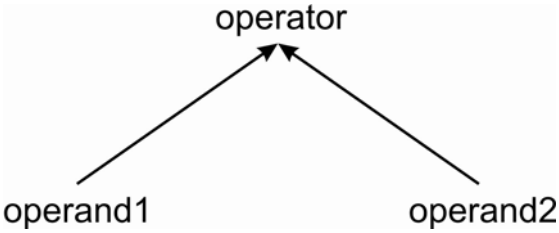
Fungsi diatas sejatinya merupakan sebuah fungsi dengan bentuk dasar (operator operand1 operand2) dimana operator adalah tanda plus (+), sedangkan operand1 dan operand2 berupa ekspresi yang mengandung fungsi. Operasi ini berjalan sesuai dengan kaidah Algoritma Euclidean sebagaimana dalam bagan berikut :



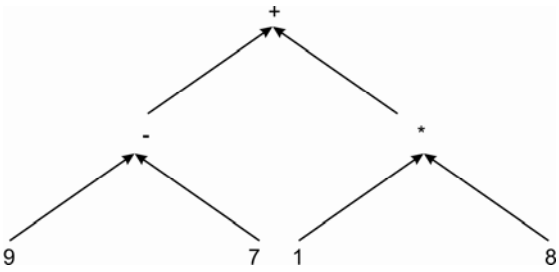
Dari kedua bagan diatas kita dapat melihat kesamaan dalam struktur fungsi dan prosedurnya. Dalam algoritma LISP, operator dapat diumpakan sebagai m, sedangkan operand-operand sebagai n, r atau variabel lainnya. Perbedaan antara kedua struktur ini terletak pada caranya menempatkan nilai ke dalam variabel dan cara pengambilannya. Algoritma Euclidean menggunakan variabel-variabel yang sudah ditetapkan jumlahnya dan isinya selalu berubah setiap kali melakukan *looping*. Sedangkan dalam LISP, variabel-variabel tersebut jumlahnya bisa bervariasi, tergantung dari kebutuhan operasi yang akan dijalankan.

2.3 Struktur Data Tree dalam LISP

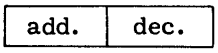
Secara umum, bentuk sebuah fungsi dalam LISP adalah sebuah list berisi operand-operand berupa ekspresi, dan kesemua operand tersebut berpusat pada operator yang diletakkan di awal list tersebut. Karena semua operand dioperasikan bergantung pada operator yang ada, maka kita dapat mengandaikan sebuah hubungan subordinasi antara operator dan operand dalam suatu fungsi, dimana operator akan menempati posisi yang lebih tinggi (menjadi *"parent"*), sedangkan operand menempati posisi yang lebih rendah (menjadi *"children"*). Hubungan dalam suatu fungsi sederhana dengan satu operator dan dua operand dapat kita umpamakan sebuah graf berarah dengan tiga simpul sebagaimana berikut ini :



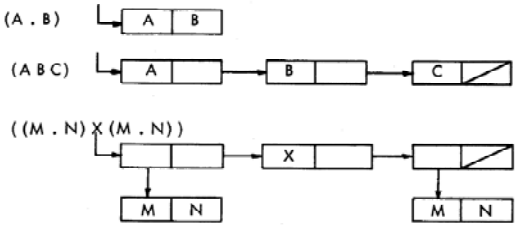
Dari contoh sebelumnya, kita dapat memasukkan fungsi lain ke dalam operand-operand tersebut. Misalnya untuk fungsi (+ (- 9 7) (* 1 8)), kita akan mendapatkan graf berikut :



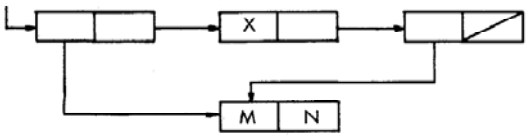
Hubungan-hubungan dalam fungsi yang di dalamnya terdapat fungsi lain (dengan kata lain fungsi bertingkat) seperti ini akan memberikan struktur berupa pohon. Pohon yang dihasilkan dapat berupa pohon biner maupun pohon ener, tergantung dari berapa banyak variabel yang digunakan di dalam fungsi tersebut.
LISP menyimpan bentuk data berupa list sebagai bagian dari sebuah pohon dalam komputer dengan setiap elemennya berisi sebuah *address* dan *decrement*. Sebuah elemen dalam list dapat kita representasikan sebagai kotak yang mengandung dua elemen tersebut.



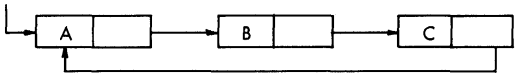
Sebuah list dapat diekspresikan dengan menggunakan sebuah pointer yang mengacu pada car (elemen pertama) dari ekspresi list tersebut, dan memberikan pointer yang mengacu pada cdr (elemen sisanya) dari list tersebut pada *decrement*-nya seperti di bawah ini.



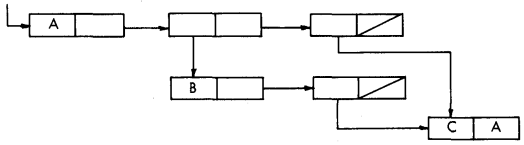
Sebuah list juga dapat mengandung subekspresi-subekspresi yang sama, misalnya ((M .N) X (M . N).



Dengan hubungan tersebut, sebuah list tidak perlu didefinisikan dua kali. Apabilaterdapat list yang identik dengan list yang telah didefinisikan, maka untuk memanggil list tersebut cukup dengan mengacu kepada pointer tempat list tersebut berada.
Sebuah list juga dapat berupa list sirkuler. List seperti ini tidak umum digunakan, tetapi bisa muncul sebagai hasil dari operasi tertentu. Misalnya :



List di atas akan mengembalikan nilai (A B C A B ...).
Apabila pembentukan list diterjemahkan ke dalam bahasa *assembly*, maka pembentukannya akan melalui proses sebagai berikut untuk list (A (B (C . A)) (C . A)) :



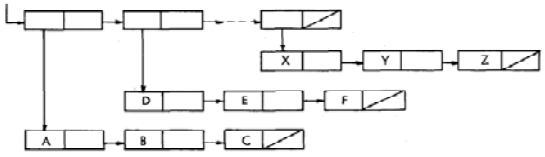
Pointer yang kosong atau NIL akan dimasukkan ke dalam address 000000, sedangkan setiap ekspresi tidak kosong akan mengacu ke sebuah pointer ke address dari nilai ekspresi tersebut dan *decrement*-nya.

```
10425    0 67352 0 65451      -A,,-*-1
10426    0 67351 0 67350      -*-2,,-*-1
10427    0 00000 0 67346      -*-3
10430    0 67347 0 65450      -B,,-*-1
10431    0 00000 0 67346      -*-1
10432    0 65451 0 65447      -C,,-A
```

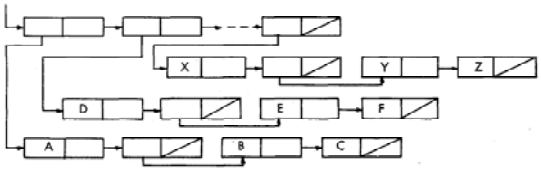
Beberapa keuntungan menggunakan struktur data tree seperti ini antara lain :

1. Dengan menggunakan metode susunan *tree* seperti ini, maka ukuran dan jumlah ekspresi yang akan ditangani oleh program tidak perlu ditentukan dari awal. Struktur data *tree* ini memungkinkan operasi dengan ekspresei-ekspresi yang terus membesar.
2. Register yang sudah selesai digunakan dapat langsung dikosongkan, sehingga dpat menghemat memori yang terpakai saat menjalankan program. Apabila struktur data disimpan secara linier, maka akan sulit untuk ditentukan kapan register yang dipakai di dalam fungsi dapat dikembalikan untuk dipakai dalam proses-proses lain.
3. Sebuah ekspresi yang menjadi subekspresi dari ekspresi lainnya hanya perlu dialokasikan sekali saja.

Berikut ini adalah contoh pembentukan list yang lebih kompleks. Terdapat dua bentuk umum list kompleks yang di dalamnya memuat ekspresi-ekspresi yang menjadi subekspresi list tersebut :
Yang pertama adalah bentuk ((A B C) (D E F) ... (X Y Z)). Pembentukan list tersebut direpresentasikan sebagai berikut :



Bentuk umum kedua adalah :



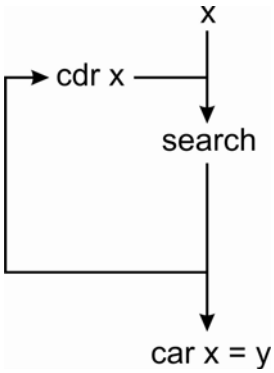
List yang akan terbentuk adalah list ((A (B C)) (D (E F)) ... (X (Y Z)))

2.4 Fungsi Rekursif

Salah satu kekuatan dalam bahasa pemrograman LISP adalah kemampuan untuk menerima definisi fungsi yang menggunakan fungsi dari dirinya sendiri. Fungsi seperti ini disebut fungsi rekursif. Hal seperti ini bisa dilakukan dengan memanggil nama atau label dari fungsi tersebut (apabila telah didefinisikan sebelumnya) di dalam fungsi itu sendiri, atau dengan cara tidak langsung, menggunakan definisi fungsi berantai yang akan berujung pada fungsi-fungsi yang asli. Fungsi ini mengikuti kaidah Algoritma Euclidean dimana fungsi tersebut akan mengulangi dirinya sendiri terhadap hasil operasinya apabila hasil yang diinginkan belum terpenuhi.
Penggunaan fungsi rekursif ini sangat efisien, karena kita tidak perlu mendefinisikan fungsi baru untuk menjalankan proses yang sama berulangkali. Dengan demikian proses *coding* dapat dipercepat dan memori yang diperlukan untuk menyimpan program dapat dihemat.
Salah satu contoh penggunaan fungsi rekursif ada pada fungsi pencarian/*search.* Berikut ini contoh fungsi *search* sederhana sebuah atom dalam sebuah list menggunakan fungsi car dan cdr dalam LISP. Fungsi car adalah fungsi yang akan mengembalikan elemen pertama dari suatu list. Elemen pertama tersebut adalah ekspresi yang berupa atom ataupun list. Sedangkan fungsi cdr adalah kebalikan dari fungsi car. Fungsi cdr akan mengembalikan nilai suatu list kecuali elemen pertamanya. Fungsi *search* dalam notasi *pseudo-code* adalah sebagai berikut.

```
function search (input x : list, y : atom output : boolean)
while search ← false do
        car x
        if car x = y then
                true
        {
        Mengembalikan nilai true apabila nilai y yang dicari terdapat dalam car x
        }
        else
                search cdr x
        {
        Melakukan search terhadap elemen sisanya apabila y belum didapati
        }
```

Fungsi *search* di atas akan mencocokkan atom pertama yang ditemukan dalam list yang dioperasikan dengan atom y yang dicari. Apabila atom y tersebut sama dengan elemen pertama dari list, maka fungsi akan mengembalikan nilai true. Sedangkan apabila atom y belum ditemukan, maka fungsi akan mencari di dalam sisa dari list tersebut setelah diambil atom pertamanya pada operasi sebelumnya.
Hubungan dalam fungsi rekursif ini dapat direpresentasikan dalam bagan berikut :



3. KESIMPULAN

Salah satu struktur data yang dapat digunakan untuk merepresentasikan suatu fungsi adalah struktur pohon atau *tree*. Bahasa pemrograman LISP menggunakan struktur ini untuk mengalokasikan ekspresi-ekspresi yang dibentuk dalam suatu susunan yang disebut *list.*
Untuk mengeksekusi fungsi-fungsi yang terdapat di dalamnya, bahasa pemrograman LISP menggunakan Algoritma Euclidean. Algoritma ini merupakan algoritma yang dapat berulang selama hasil yang diinginkan belum didapatkan .
Algoritma Euclidean ini juga memungkinkan bahasa LISP untuk melakukan fungsi rekursif, dimana fungsi tersebut akan memanggil dirinya sendiri dalam kondisi tertentu. Secara sederhana fungsi rekursif adalah menggunakan hasil dari fungsi tersebut untuk digunakan sebagai operand dari fungsi itu sendiri.

DAFTAR REFERENSI

[1] Munir, Renaldi, *Diktat Kuliah IF2153 Matematika Diskrit Edisi Keempat,* Program Studi Teknik Informatika Institut Teknologi Bandung, Bandung, 2006

[2] Graham, Paul, *The Roots of LISP (draft),* 2002

[3] McCarthy, John, *LISP 1.5 Programmer's Manual*, The M.I.T. Press Massachusetts Institute of Technology, Cambridge, Massachusetts, 1985