IMPLEMENTASI POHON DALAM PEMROSES BAHASA PEMROGRAMAN LISP (LISt Processing) SEDERHANA

Puja Pramudya – NIM : 13506058

Jurusan Teknik Informatia ITB,Jl. Ganesha 10, Bandung e-mail : if16058@students.if.itb.ac.id

Abstract *– Makalah ini membahas tentang implementasi pohon dalam membangun sebuah pemroses bahasa pemrograman (interpreter) LISP sederhana. Bahasa LISP adalah bahasa pemrograman tingkat tinggi yang menggunakan aspek rekursif sebagai struktur kontrol utama. Pohon dengan strukturnya yang rekursif merupakan struktur yang sangat sesuai untuk menangani beberapa primitif fungsi yang didefinisikan di dalam LISP. Penggunaan pohon di dalam membangun sebuah pemroses bahasa LISP sederhana adalah terapan dari pohon biner,pohon ekspresi dan pohon keputusan untuk menangani fungsi akses elemen, penanganan list, ekspresi aritmatika,relasional,logika dan ekspresi kondisional. Dengan menggunakan pohon ,perhitungan yang sukar dapat ditangani dengan mudah secara rekursif.*

Kata Kunci: *Interpreter,rekursif,LISP,pohon,pohon ekspresi,pohon biner,pohon keputusan*

1. PENDAHULUAN

Pemroses bahasa pemrograman adalah suatu alat untuk menjalankan kode program yang ditulis dalam bahsa pemrograman tingkat tinggi. Terdapat dua cara yang paling umum, yaitu dengan kompilator atau *interpreter*. Bahasa pemrograman LISP (*LISt Processing*) diproses dengan menggunakan *interpreter*.

Interpreter bertugas menerjemahkan instruksi ke dalam bahasa antara yang kemudian akan dieksekusi program. Keuntungan interpreter adalah program tidak perlu melalui proses kompilasi yang terkadang membutuhkan waktu lama, khususnya bagi program yang memiliki sederetan instruksi sangat panjang.

Di dalam sebuah pemroses bahasa pemrograman, dapat mempunyai beberapa struktur data untuk memudahkan pemroses – dalam hal ini interpreter - melakukan aksi yang sudah ditentukan dengan kode yang bersesuaian. Penggunaan struktur data yang dipilih haruslah struktur yang dapat menunjukkan kinerja efektif, misalnya dalam kemudahan proses, kecepatan eksekusi atau kemudahan dalam memprogram pemroses bahasa pemrograman itu sendiri.

Beberapa struktur yang lazim digunakan dalam sebuah pemroses bahasa pemrograman adalah struktur senarai (*list*), tumpukan (*stack*), dan pohon (*tree*). Tumpukan dalam representasi senarai sering digunakan untuk implementasi algorirma *parsing*, evaluasi, konversi antar bentuk ekspresi,dan mengubah program rekursif menjadi non-rekursif. Sedangkan pohon sering digunakan untuk metode pencarian data, evaluasi ekspresi dan pengolahan data secara rekursif.

Nantinya akan dapat dillihat bahwa hanya dengan menggunakan struktur pohon, dapat dibangun sebuah interpreter bahasa pemrogaman LISP sederhana. Hal ini dimungkinkan karena banyak definisi rekursif dalam LISP yang sesuai dengan pohon, karena struktur pohon itu sendiri adalah rekursif.

2. DESKRIPSI BAHASA PEMROGRAMAN LISP (LIST PROCESSING)

LISP adalah keluarga bahasa pemrograman yang memiliki sejarah panjang dan penuh dengan *parenthesized syntax* (sintaks dengan tanda kurung). LISP pertama kali didesain pada tahun 1958 oleh John McCarthy sehingga menjadikan LISP menjadi bahasa kedua tertua dari keluarga bahasa pemrograman tingkat tinggi. LISP dirancang berdasarkan *lambda calculus*, ditujukan sebagai bahasa fungsional untuk memanipulasi formula simbolik.

LISP banyak memberikan pengaruh pada rancangan implementasi bahasa-bahasa yang muncul di kemudian hari, misalnya ALGOL60,Logo, dan FORTH. Seiring dengan berjalannya waktu LISP berkembang menjadi media komunikasi dari komunitas *artificial intelligence*. Karakteristik inovatif dari LISP adalah :

- fungsi sebagai unit program dasar
- list sebagai struktur data dasar
- struktur data dinamik
- penggunaan ekspresi simbolik
- ekspresi rekursif dan kondisional sebagai struktur kontrol utama

Bahasa LISP merupakan bahasa ekspresi, karena – baik program maupun data – dinyatakan dalam bentuk ekspresi simbolik. Ekspresi simbolik dalam LISP dikenal dengan istilah Ekspresi-S. Ekspresi-S ditulis dalam notasi fungsional prefix. Sehingga dalam

sebuah ekspresi, operator dituliskan mendahului operannya.

Ekspresi-S terdiri dari :
Atom, jenis :

- Numerik(angka),berupa integer atau real
- Simbolik, berupa karakter atau string

List, terdiri dari :

- (
- satu/lebih ekspresi S
-)

Contoh ekspresi- S di dalam bahasa LISP :
Atom :

- Angka : 1,73,5,3.14,1.6666
- Simbo: nol,satu,nilai,a,x

List, terdiri dari :

- () atau nil
- (12 4 5)
- (a b 35 (23))
-

Dalam LISP dikenal beberapa jenis ekspresi, yaitu :

- ekspresi dasar
- ekspresi kondisional
- ekspresi rekursif

Ketiga ekspresi ini merupakan ekspresi-S jenis list.

Sebuah ekspresi, dapat bernama maupun tidak. Sebuah ekspresi terdiri dari operator dan operan. Jika operan dioperasikan sesuai dengan operatornya, maka akan menghasilkan sebuah nilai. Masing-masing operator dan operan termasuk ekspresi-S jenis atom.

3. STRUKTUR DATA POHON

Pohon dalam teori graf adalah graf tak-berarah terhubung yang tidak mengandung sirkuit. Definisi pohon menurut stukturnya, pohon adalah himpunan terbatas tidak kosong, dengan elemen yang dibedakan menjadi : sebuah elemen (disebut akar), dan elemen lain yang di bagi menjadi sub-himpunan yang merupakan pohon itu sendiri (rekursif)

Pada kebanyakan penggunaan pohon, simpul tertentu diperlakukan sebagai akar (*root)*. Begtiu sebuah simpul ditetapkan sebagai akar maka simpul- simpul lainnya dapat di capai dari akar dengan memberi arah pada sisi-sisi pohon yang mengikutinya.

Akar mempunyai derajat masuk sama dengan nol dan simpul-simpul lainnya berderajat-masuk sama dengan satu. Simpul yang mempunyai derajat keluar tidak sama dengan nol disebut simpul dalam atau cabang. Simpul yang berderajat masuk sama dengan nol disebut dengan simpul daun atau terminal. Setiap simpul di pohon dapat dicapai dari akar sengan sebuah lintasan tunggal (unik).

Beberapa istilah (terminologi) dasar – berhubungan dengan teori graf - yang harus diketahui sebelum pembahasan lebih lanjut adalah :

1. Simpul
   Elemen dari pohon yang memungkinkan akses pada sub pohon dimana simpul tersebut berfungsi sebagai AKAR

2. Anak dan Orangtua
   Misalkan *x* adalah sebauh simpul.Simpul *y* dikatakan anak simpul *x* jika ada sisi dari *x* ke *y*. Dalam hal demikian, *x* disebut orangtua *y*.

3. Lintasan (*path*)
   Lintasan dari simpul vi ke simpul vk adalah runtuttan simpul-simpul v1,v2,...,vk sedemikian sehingga vi adalah orang tua dari vi+1 untuk 1<i<k.

4. Upapohon (*subtree*)
   Misalkan *x* adalah simpul di dalam pohon T. Yang dimaksud dengan upapohon *x* dengan *x* sebagai akarnya ialah upagraf T' =(V'E') sedemikian sehingga V' mengandung *x* dan anak-anaknya serta E' mengandung sisi-sisi dalam semua lintasan yang berawal dari *x*.

5. Derajat
   Derajat sebuah simpul adalah jumlah upapohon pada simpul tersebut.

6. Daun
   Simpul yang tidak mempunyai anak.

7. Simpul Dalam
   Simpul yang mempunyai anak.

8. Kedalaman (Tinggi)
   Panjang maksimum lintasan dari akarke daun.

Pohon berakar yang setiap simpul cabangnya paling banyak adalah n, disebut pohon *n-ary*. Pohon *n-ary* banyak digunakan diberbagai bidang ilmu maupun dalam kehidupan sehari-hari. Dalam terapannya, pohon *n-ary* digunakan sebagai model yang merepresentasikan suatu struktur.

Dalam hal khusus,seringkali dibutuhkan pohon *n-ary* yang selalu mempunyai maksimal anak simpul berjumlah 2. Pohon seperti ini disebut sebagai pohon biner. Diperkenalkan terminologi dari simpul dalam pada pohon biner, yaitu anak kiri (*left child*) dan anak kanan (*right child*). Pohon yang akarnya adalah anak kiri disebut *upapohon kiri* dan yang dari anak kanan disebut *upapohon kanan*.

Dalam penelusuran elemen-elemen pohon biner, terdapat 3 skema yaitu :

1. Preorder
    1. Kunjungi akar
    2. Telusuri anak kiri secara pre-order
    3. Telusuri anak kanan secara pre-order

2. Inorder
    1. Telusuri anak kiri secara pre-order
    2. Kunjungi akar
    3. Telusuri anak kanan secara pre-order

3. Inorder
    1. Telusuri anak kiri secara pre-order
    2. Telusuri anak kanan secara pre-order
    3. Kunjungi akar

Dapat dilihat dalam skema penelusuran ini merupakan proses yang rekursif, yang sesuai dengan struktur pohon.

4. IMPELEMENTASI POHON DALAM PEMROSES BAHASA LISP SEDERHANA

Ekspresi-S di dalam LISP hampir semuanya direpresentasikan dalam bentuk list. Namun dalam pemrosesannya, penulis menemukan bahwa penggunaan struktur pohon, khususnya biner, akan memberikan kemudahan tertentu bagi pemrogram dalam membangun sebuah interpreter LISP sederhana, khususnya yang berhubungan dengan aspek rekursifitas. Lebih lanjut lagi, untuk pemroses LISP sederhana, hanya dengan struktur ini sudah cukup untuk melakuan evaluasi ekspresi dari primitf-primtif yang didefiniskan di dalam LISP. Disarankan agar dalam pengembangan interpreter LISP ini menggunakan bahasa yang mendukung pemrograman dinamik, agar penggunaan pohon dengan *pointer* dapat lebih leluasa dilakukan.

Beberapa primitif yang akan diimplementasikan untuk pemroses bahasa LISP sederhana adalah :

1. Operasi List
    - CAR
    - CDR
    - LIST
    - APPEND
    - CONS
2. Evaluasi ekspresi matematika,logika dan relasi
3. Ekspresi kondisional
4. Fungsi Eval
5. *Call function*

4.1 Akses Elemen : Primitif CAR dan CADR

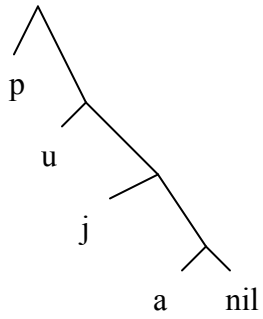
Primitif **CAR** adalah sebuah fungsi yang akan mengembalikan elemen pertama dari suatu list. Apabila list-nya sederhana, maka kembaliannya adalah atom, apabila tidak maka kembaliannya adalah list.

Primitif **CDR** adalah sebuah fungsi yang akan mengembalikan semua elemen list kecuali elemen pertamanya. Kembalian dari CDR adalah list.

Misalkan contoh ekspresi masukan adalah :

```
(p u j a)
```

Maka dari masukan, akan dibuat sebuah pohon dengan elemen-elemennya berupa atom sebagai seperti gambar berikut :



Gambar1.a:Pohon bentukan input

Jika dilakukan penyimpanan elemen sepert ini, dengan mudah terlihat bahwa primitif *CAR* ternyata adalah akses elemen terhadap anak kiri, dan primitif *CDR* adalah akses elemen anak kanan sampai bertemu nil. Penelurusan dilakukan secara pre-order. Dengan kata lain, CDR akan mengambil upapohon kanan dari pohon elemen ,sedangkan CAR akan mengambil upapohon kiri.

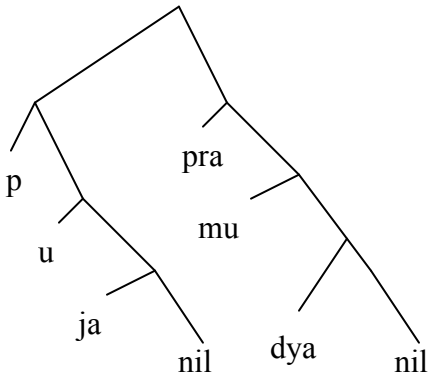
```
(car (p u j a))
→ p

(cdr (p u j a))
→ (u j a)
```

Apabila pengguna menggunakan primitif berkalang :

```
(cdr (cdr (cdr (p u j a) ) ) )
→ (a)
```

Operasi demikian akan mudah ditangani dengan cara rekursif dengan penulusuran preorder.
Perhatikan bila masukan berupa list yang tidak sederhana :

```
( (p u ja) pra mu dya)
```



Gambar1.b:Pohon bentukan input

Operasi CAR tetep benar dan akan menghasilkan (p u j a) yang dikerjakan secara rekursif sebagai akses anak kiri secara preorder, yang akan mengirimkan upapohon kiri dari elemenn.

4.2 Primitif CONS,LIST dan APPEND

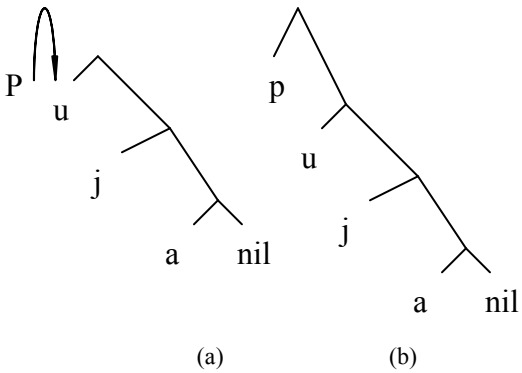
Primitif **CONS** adalah fungsi yang akan menghasilkan sebuah list dengan cara menambahkn sebuah atom atau list ke dalam suatu list,hasilnya adalah list.

Primitif **LIST** adalah fungsi yang akan menghasilkan sebuah list dari atom atau kumpulan list.

Primitif **APPEND** adalah fungsi yang akan menghasilkan sebuah list dengan cara menggabungkan dua list.

Pada dasarnya 3 primitif ini merupakan operasi dasar terhadap list, yaitu membuat list baru, *insert* elemen dan gabung list. Untuk mengimple- mentasikannya dalam struktur pohon ternyata sangat mudah.

Misalkan `(cons p (u j a) )`, akan menghasilkan `(p u j a)` . Algoritmanya adalah :
1. Buat P sebagai simpul daun
2. Buat sebuah pohon baru dengan P sebagai anak kiri dan list sebagai anak kanan.

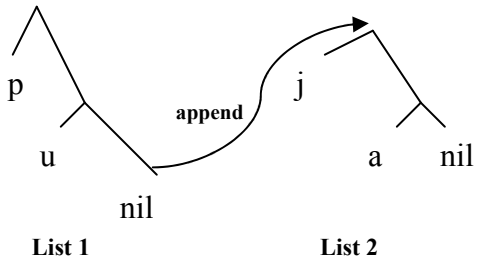


Gambar 2.
(a) Keadaan awal,elemen belum digabung
(b) Keadaan akhir setelah proses

Untuk primitif *LIST*, maka dari elemen masukan akan dibuat sebuah pohon dengan bentuk yang sama dengan pohon diatas. Dapat dikerjakan secara rekursif atau iteratif sampai elemen habis.

Algorima untuk primitif LIST :
1. Cek elemen,jika nil proses selesai
2. Insert sebagai anak kiri
3. Buat anak kanan

Primitif *APPEND* merupakan persoalan insert elemen,dimana elemen yang di insert juga berupa pohon. Misallan list `(p u)` akan di append dengan list `( j a )`.Ilustrasi dari primitif ini adalah :



Gambar3 : Ilustrasi pohon untuk primitif append

Algoritma untuk primitf append :
1. Telusuri pohon yang mengandung list 1 hingga bertemu nil
2. Anak kanan yang bernilai nil ditunjuk ke akar dari pohon yang mengandung list 2

4.3 Ekspresi Matematika, Logika dan Relasional

Terapan pohon yang penting di dalam interpreter adalah untuk mengevaluasi ekspresi matematika, logika dan relasional. Salah satu keuntungan pohon adalah dalam mengevaluasi suatu ekspresi, kita dapat mengerjakannya secara rekursif. Namun, yang perlu diperhatikan sebenarnya adalah bagaimana membangun sebuah pohon ekspresi dari masukan pengguna yang berupa notasi prefix. Begitu pohon ekspresi terbentuk, evaluasi dapat mudah dikerjakan.

Algoritma pembentukan pohon ekspresi ternyata juga tidak terlalu sulit. Yang perlu diperhatikan adalah dalam sebuah ekspresi aritmatika terdapat dua komponen, yaitu operator dan operan. Komponen inilah yang akan dijadikan *mark*, dalam pembentukan pohon ekspresi.

Salah satu algoritma pembentukan pohon ekspresi, dengan bantuan sebuah *stack*,adalah sebagai berikut: :
1. Balikkan ekspresi masukan
2. Periksa elemen
3. Jika operan, buat simpul daun,lalu PUSH alamat simpul ke *stack*
4. Jika operator,buat simpul,lalu :
   - POP alamat simpul daun, dan *assign* sebagai anak kiri
   - POP alamat simpul daun, dan *assign* sebagai anak kanan
   - PUSH alamat simpuk ke *stack*
5. Jika masih ada elemen teruskan ke langkah 2
6. Jika tidak POP alamat simpul,dan alamat itu lah yang akan menjadi akar dari pohon ekspresi

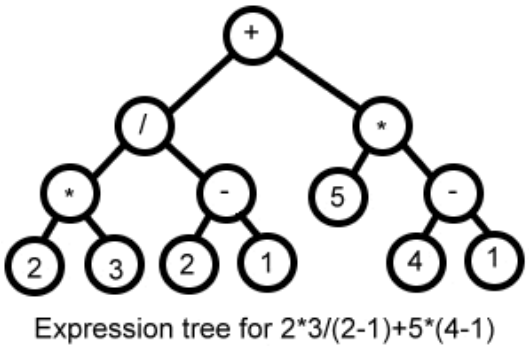
Begitu pohon ekspresi telah terbentuk,maka untuk mengevaluasinya kita tinggal melakukan operasi secara rekursif. Misalkan fungsi untuk evaluasi pohon

ekspresi kita namakan `eval`,maka algoritmanya adalah :

1. Jika simpul berisi operan, maka kembalikan isi simpul {basis}
2. Jika tidak berisi operator {rekurens}
   - Eval anak kiri
   - Eval anak kanan
   - Operasikan sesuai dengan operator

Terlihat di dalam fungsi `eval` terdapat penggunaan fungsi itu sendiri. Fungsi ini bersifat rekursif.

Contoh pohon ekspresi matematika:



Gambar 4: Pohon ekspresi matematika

Untuk ekspresi relasional, maka pada akar akan terdapat operator >,< ,= atau kombinasi keduanya. Pada evaluasi ekspresi relasional, pohon aritmatika harus tetap digunakan. Yaitu untuk evaluasi ekspresi matematika.

Untuk ekspresi logika, maka pada akar akan terdapat operator AND,OR dan NOT. Pada evaluasi logika, akan menggunakan evaluasi relasional dan aritmatika pula.

4.4 Ekspresi Kondisional

Di dalam LISP dikenal 2 primitif untuk ekspresi kondisional,yaitu :

1. IF
2. COND

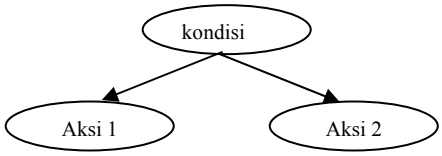
Primtif *IF* digunakan jika hanya terdapat 2 kondisi yang saling disjoin, sedangkan primitif COND digunakan jika terdapat lebih dari 2 kondisi.

Notasinya IF dalam LISP adalah :

```
(IF   (kondisi)
      (aksi1)
      (aksi2))
```

Untuk evaluasi ekspresi kondisional, kita menggunakan terapan pohon biner yang lain, yaitu pohon keputusan (*decision tree)*. Untuk primitif IF, strukturnya merupakan sebuah pohon yang terdiri dari akar, anak kiri dan anak kanan. Dimana akar berisikan kondisi,anak kiri adalah aksi jika kondisi terpenuhi, dan anak kanan adalah aksi jika selain tidak[1].

Ilustrasi dari ekspresi IF dalam representasi pohon adalah sebagai berikut :



Gambar 4 : Pohon keputusan

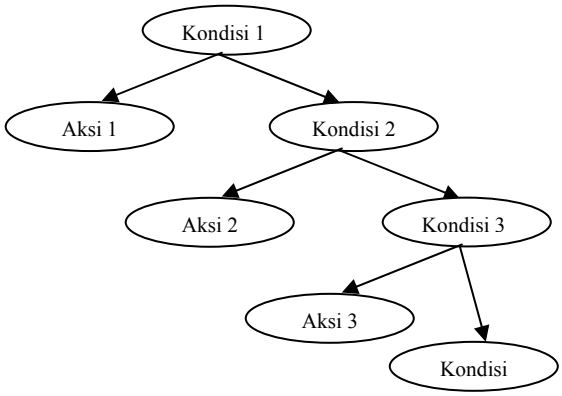
Aspek pemeriksaan kondisi dapat dikerjakan secara terpisah,yaitu dikerjakan dengan pohon ekspresi relasional atau logika. Kemudian hasil dari ekspresi dilemparkan ke pohon keputusan, yang akan dijadikana acuan apakah akan mengerjakan aksi-1 atau aksi-2.

Seperti telah disebutkan di atas, bahwa untuk kondisi yang lebih dari satu maka di dalam LISP digunakan primitif COND. Notasinya adalah :

```
(COND (kondisi-1 aksi 1)
      (kondisi-2 aksi 2)
      (kondisi-3 aksi 3)
      (kondisi-4 aksi 4)
      .........
)
```

Untuk melakukan pengerjaan primitif *cond* dapat dilakukan dengan beberapa alternatif, dengan menggunakan senarai dari pohon atau pohon yang dapat memiliki kedalaman sesuai dengan jumlah kondisi.

Ilustrasi pohon evaluasi untuk ekspresi kondisional lebih dari 2 kondisi adalah :



Gambar 5 : Pohon keputusan lebih dari 2 kondisi

[1] Pemilihan aksi jika kondisi terpenuhi tidak harus selalu dikiri,tergantung selera pemrogram

Proses evaluasi akan dikerjakan dengan urutan kondisi 1. Begitu terpenuhi, akan dikerjakan aksi 1, jika tidak lanjut ke kondisi 2, begitu seterusnya, hingga semua kondisi diperiksa atau terdapat kondisi yang memenuhi. Dengan kata lain evaluasi primitif *cond* dilakukan secara rekursif, diibaratkan sebagai if berkalang.

4.5 Fungsi Eval

Di dalam LISP, didefinisikan fungsi eval, yaitu fungsi yang akan memperlihatkan urutan pengerjakan ekspresi aritmatika. Dengan fungsi ini akan terlihat urutan pengerjaan berdasarkan tingkat kedalaman pohon.

Untuk implementasi fungsi eval, dilakukan dengan menggunakan pohon ekspresi. Ini tidak lain adalah menambahkan perintah *print* atau output untuk melakukan interaksi ke layar. Setiap satu ekspresi selesai dikerjakan, ditulis, sehingga pada akhirnya akan terlihat urutan pengerjaan suatu ekspresi aritmatika.

Contoh :

```
( (+ (+ 2 4) (- 5 1) ) )
((+ 6 4))
(10)
```

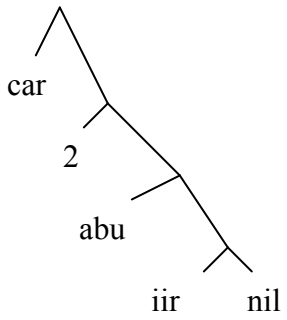
4.6 Call Function

Bagaimana LISP bekerja untuk memanggil primitf LISP yang ada berdasarkan input dari user ? Secara sederhana pemanggilan fungsi dapat diimplementasikan dengan cara penggunaan fungsi CAR dan CDR kembali. Setiap input user, dapat secara dipandang sebagai *list of symbol*.

Misalkan input user adalah :

```
(car ( 2 abu iir ) )
```

Dibangun sebuah pohon untuk pencocokan fungsi sebagai berikut :



Gambar 6 : Pencocokan ekspresi

Dengan fungsi CAR, maka jenis fungsi yang diinginkan user dapat diambil lalu dicocokkan dengan daftar fungsi yang tersedia. Sedangkan dengan fungsi CDR argumen atau parameter dapat diambil lalu dikerjakan sesudai dengan fungsi yang cocok.

5. KESIMPULAN

Dalam membangun sebuah interpreter bahasa pemrograman,diperlukan pemilihan struktur data yang tepat agar dalam pemrosesannya sederhana dan efektif. Dalam hal membangun LISP (List Proccessing) interpreter sederhana dengan bahasa pemrograman tingkat tinggi harus dicari struktur yang memudahkan pemrosesan sintaks LISP yang banyak menggunakan aspek rekursifitas. Penggunaan pohon dalam kasus ini merupakan pilihan yang sesuai dengan struktur yang diperlukan.

Pohon merupakan salah satu bagian dari aplikasi graf, yang strukturnya rekursif. Dengan menggunakan pohon, komputasi yang sulit dapat ditangani dengan mudah. Implementasi pohon dalam pemroses bahasa LISP sederhana adalah untuk menangani fungsi primitif yang didefinisikan di dalam LISP. Fungsi-fungsi tersebut adalah :

- Akses elemen : CAR,CDR
- Primitif CONS,LIST,dan APPEND
- Ekspresi aritmatika,relasional dan logika
- Ekspresi kondisional
- Fungsi eval
- Call function

Primitif jenis akses elemen,CONS,LIST dan APPEND diterapkan dengan pohon biner yang tidak seimbang,dimana tiap elemen akan menempati satu tempat di anak kiri setiap simpul dalam, dan melakukan insert elemen pohon . Ekspresi aritmatika,relasional,logika dan fungsi evaluasi menggunakan konsep pohon ekspresi dengan pengerjaan secara preorder. Ekspresi kondisional menggunakan terapan pohon keputusan. Call function atau metode pemanggilan dan pencocokan fungsi dilakukan dengan notasi akses elemen CAR dan CDR.

DAFTAR REFERENSI

[1] Munir, Rinaldi. (2003). Matematika Diskrit. Departemen Teknik Informatika, Institut Teknologi Bandung.

[2] Liem, Inggriani. (2003). Catatan Kuliah Algoritma dan Pemrograman. Departemen Teknik Informatika, Institut Teknologi Bandung.

[3] Purwanti, Sri. (2003). Diktat LISP. Departemen Teknik Informatika, Institut Teknologi Bandung.

[4] McCarthy, John. (1985). LISP 1.5 Programmer's Manual. The Computation Center and Research Laboratory of Electronic,Massachusetts Institute of Technology.

[5] Wikipedia. Lisp (programming language). *http://en.wikipedia.org/wiki/Lisp_%28programming_language%29* Tanggal akses 20 Desember 2007 pukul 20.00.

[6] *Trees. http://www.ibiblio.org/ obp/thinkCS/ python/english/chap20.htm . Tanggal akses 20 Desember 2007 pukul 20.03*