Parsing atau Proses Penurunan

Parsing dapat dilakukan dengan cara :

- Penurunan terkiri (*leftmost derivation*) : simbol variable yang paling kiri diturunkan (tuntas) dahulu
- Penurunan terkanan (*rightmost derivation*): variable yang paling kanan diturunkan (tuntas) dahulu
- Misalkan terdapat ingin dihasilkan string *aabbaa* dari

context free language: $S \rightarrow a\,AS \mid a,$

$A \rightarrow SbA \mid ba$

Parsing atau Proses Penurunan

Penurunan kiri :

S => aAS

=> aSbAS

=> aabAS

=> aaabbaS

=> aabbaa

Penurunan kanan :

S => aAS

=> aAa

=> aSbAa

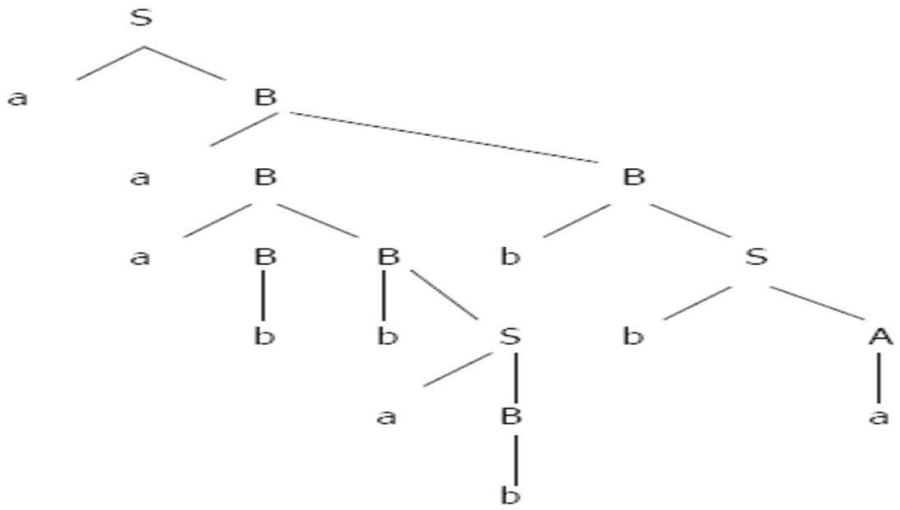
=> aSbbaa

=> aabbaa

Parsing

Misalnya:

S -> aB | bA
A -> a | aS |bAA
B -> b | bS | aBB



Penurunan untuk string aaabbabba

Dalam hal ini perlu untuk melakukan percobaan pemilihan aturan produksi yang bisa mendapatkan solusi

Metode Parsing

Perlu memperhatikan 3 hal:

- Waktu Eksekusi
- Penanganan Kesalahan
- Penanganan Kode

Parsing digolongkan menjadi:

- ***Top-Down***

  Penelusuran dari *root ke leaf* atau dari simbol awal ke simbol terminal

  metode ini meliputi:

  - *Backtrack/backup : Brute Force*
  - *No backtrack : Recursive Descent Parser*

- ***Bottom-Up***

  Metode ini melakukan penelusuran dari *leaf ke root*

Parsing: Brute force

- Memilih aturan produksi mulai dari kiri
- Meng-expand simbol non terminal sampai pada simbol terminal
- Bila terjadi kesalahan (string tidak sesuai) maka dilakukan *backtrack*
- Algoritma ini membuat pohon parsing secara top-down, yaitu dengan cara mencoba segala kemungkinan untuk setiap simbol non-terminal
- Contoh suatu language dengan aturan produksi sebagai berikut

  $S \rightarrow aAd \mid aB$

  $A \rightarrow b \mid c$

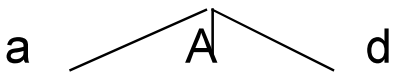
  $B \rightarrow ccd \mid ddc$

- Misal ingin dilakukan parsing untuk string ‘accd’

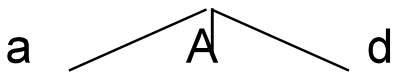
Parsing: Brute force

(i) S

(ii) S



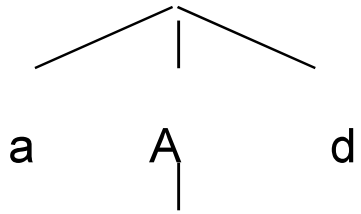
(iii) S



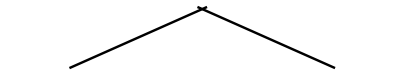
b

Terjadi kegagalan (iii), dilakukan back track

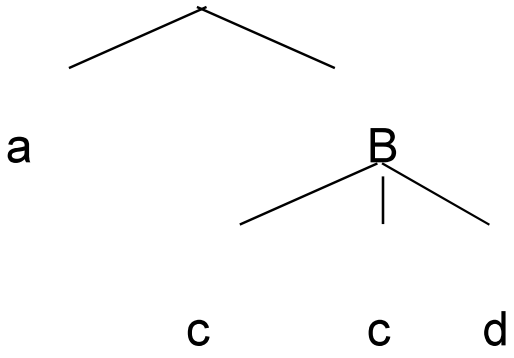
(iv) S



(v) S



(vi) S



Terjadi kegagalan lagi (iv), dilakukan back-track

Parsing: Brute force

Kelemahan dari metode-metode *brute-force*

- Mencoba untuk semua aturan produksi yang ada sehingga menjadi lambat (waktu eksekusi)
- Mengalami kesukaran untuk melakukan pembetulan kesalahan
- Memakan banyak memakan memori, dikarenakan membuat *backup* lokasi *backtrack*
- Grammar yang memiliki R*ekursif Kiri* tidak bisa diperiksa, sehingga harus diubah dulu sehingga tidak rekursif kiri, Karena rekursif kiri akan mengalami ***Loop*** yang terus-menerus

Brute force : Contoh

Terdapat grammar/tata bahasa G = (V,T,P,S), dimana

V= (“E”,”T”,”F”) Simbol NonTerminal (variable)

T= (“i”,”*”,”/” ,”+”,”-”) Simbol Terminal

S=”E” Simbol Awal / Start simbol

String yang diinginkan adalah ***i * i***

aturan produksi (P) yang dicobakan adalah

1. $E \rightarrow T \mid T + E \mid T - E$

   $T \rightarrow F \mid F * T \mid F / T$

   $F \rightarrow i$

accept (diterima)

Brute force : Contoh

2. $E \rightarrow T \mid E+T \mid E-T$

$T \rightarrow F \mid T^* F \mid T / F$

$F \rightarrow i$

accept (diterima)

- Meskipun ada rekursif kiri, tetapi tidak diletakkan sebagai aturan yang paling kiri

3. $E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T^* F \mid T / F \mid F$

$F \rightarrow i$

Rekursif kiri, program akan mengalami loop

Brute force : Aturan produksi

Aturan Produksi yang rekursif memiliki ruas kanan (hasil produksi) yang memuat simbol variabel pada ruas kiri

Sebuah produksi dalam bentuk

$A \rightarrow \beta A$ merupakan produksi rekursif kanan

β = berupa kumpulan simbol variabel dan terminal

contoh: $S \rightarrow d S$

$B \rightarrow ad B$

bentuk produksi yang rekursif kiri

$A \rightarrow A \beta$ merupakan produksi rekursif Kiri

contoh:

$S \rightarrow S d$

$B \rightarrow B ad$

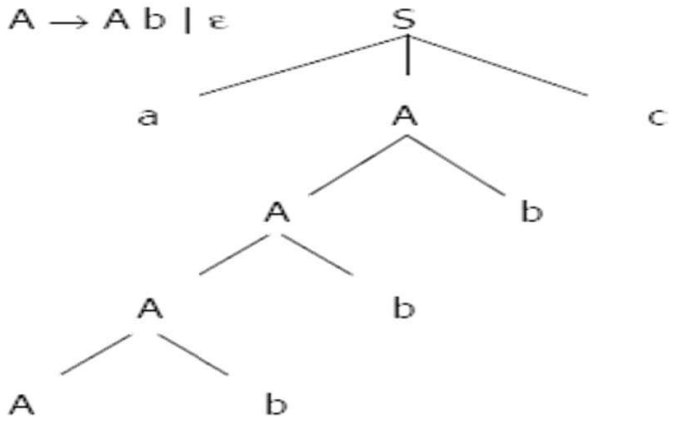
Aturan produksi : Brute force

Produksi yang rekursif kanan akan menyebabkan penurunan tumbuh kekanan, Sedangkan produksi yang rekursif kiri akan menyebabkan penurunan tumbuh ke kiri.

Contoh: Context free Grammar dengan aturan produksi sebagai berikut:

$S \rightarrow a\ A\ c$

$A \rightarrow A\ b \mid \varepsilon$



Aturan produksi : Brute force

Dalam Banyak penerapan tata-bahasa, ***rekursif kiri*** tidak diinginkan,

Untuk menghindari penurunan kiri yang looping, perlu dihilangkan sifat rekursif, dengan langkah-langkah sebagai berikut:

- Pisahkan Aturan produksi yang rekursif kiri dan yang tidak; misalnya

  Aturan produksi yang **rekursif kiri**

  $A \rightarrow A\ \alpha 1 \mid A\ \alpha 2 \mid \ldots \mid A\ \alpha n$

  Aturan produksi yang ***tidak*** **rekursif kiri**

  $A \rightarrow \beta 1 \mid \beta 2 \mid \ldots \mid \beta n$

Aturan produksi : Brute force

- lakukan per-ganti-an aturan produksi yang rekursif kiri, sebagai berikut:

  1. $A \rightarrow \beta 1\ Z \mid \beta 2\ Z \mid \ldots \mid \beta n\ Z$

  2 $Z \rightarrow \alpha 1 \mid \alpha 2 \mid \ldots \mid \alpha n$

  3 $Z \rightarrow \alpha 1\ Z \mid \alpha 2\ Z \mid \ldots \mid \alpha n\ Z$

Aturan produksi : Brute force

- Pergantian dilakukan untuk setiap aturan produksi dengan simbol ruas kiri yang sama, bisa muncul variabel Z1, Z2 dst, sesuai dengan variabel yang menghasilkan rekurisif kiri

Contoh: Tata Bahasa Context free

$S \rightarrow Sab \mid aSc \mid dd \mid ff \mid Sbd$

- Pisahkan aturan produksi yang rekursif kiri

  $S \rightarrow Sab \mid Sbd$

  Ruas Kiri untuk S: $\alpha 1=ab$, $\alpha 2=bd$

- Aturan Produksi yang tidak rekursif kiri

  $S \rightarrow aSc \mid dd \mid ff$

  dari situ didapat untuk Ruas Kiri untuk S: $\beta 1 = aSc$, $\beta 2 = dd$, $\beta 3= ff$

Aturan produksi : Brute force

- Langkah berikutnya adalah penggantian yang rekursif kiri

S → Sab | Sbd, dapat digantikan dengan

1. S → aSc***Z1*** | dd***Z1*** | ff***Z1***
2. Z1 → ab | bd
3. Z1 → ab***Z1*** | bd***Z1***

- Hasil akhir yang didapat setelah menghilangkan rekursif kiri adalah sebagai Berikut:

S → aSc | dd | ff

S → aSc***Z1*** | dd***Z1*** | ff***Z1***

Z1 → ab | bd

Z1 → ab***Z1*** | bd***Z1***

Aturan produksi : Brute force

- Kalau pun tidak mungkin menghilangkan rekursif kiri dalam penyusunan aturan produksi maka produksi rekursif kiri diletakkan pada bagian belakang atau terkanan, hal ini untuk menghindari looping pada awal ***proses parsing***

- Metode ini jarang digunakan, karena semua kemungkinan harus ditelusuri, sehingga butuh waktu yang cukup lama serta memerlukan memori yang besar untuk penyimpanan stack (backup lokasi backtrack)

- Metode ini digunakan untuk aturan produksi yang memiliki alternatif yang sedikit

Parsing: Recursive Descent Parser

Parsing dengan ***Recursive Descent Parser***

- Salah satu cara untuk meng-aplikasikan bahasa context free
- Simbol terminal maupun simbol variabelnya sudah bukan sebuah karakter
- Besaran leksikal sebagai simbol terminalnya, besaran syntax sebagai simbol variablenya /non terminalnya
- Dengan cara penurunan secara recursif untuk semua variabel dari awal sampai ketemu terminal
- Tidak pernah mengambil token secara mumdur (back tracking)
- Beda dengan turing yang selalu maju dan mundur dalam melakukan *parsing*

Aturan Produksi memakai Recursif Descent :

- Semua simbol variabel dijadikan prosedur/fungsi
- Jika ketemu simbol terminal pada aturan produksi , maka panggil prosedurnya
- Penelusuran bersifat top down mengikuti sintaks sesuai pola pada diagram sintaks
- Fungsi/prosedur ditulis untuk setiap non terminal dari suatu produksi. Setiap fungsi/prosedur akan melemparkan nilai benar atau salah bergantung pada apakah fungsi tersebut mengenali substring yang diterima sebagai ekspansi dari non terminal.

Contoh :

Grammar dengan BNF :

<program> ::= t_PROG t_ID t_SEMICOL <block> t_DOT

<block> ::= t_BEGIN <statement> {t_SEMICOL <statement>}
t_END

<statement> ::= t_ID t_ASS <simple exp> |
t_IF <exp> t_THEN <statement>|
t_IF <exp> t_THEN <statement> t_ELSE
<statement>

<exp> ::= <simple_exp> t_EQ <simple exp> |
<simple_exp> t_LT <simple_exp>|
<simple_exp> t_GT <simple_exp>

Dst…….

Penggalan program untuk grammar tsb

```
Procedure Program
Begin
If token=t_PROG then
Begin
        Scan;
        If token=t_ID then
        Begin
                Scan;
                If token=t_SEMICOL then
                Begin
                        Scan;
                        Block; {panggil prosedur block}
                        If token=t_DOT then
                        Begin
                                Scan;
                        End;
                end;
        end;
end;
end;
```

```
Procedure Block
Begin
If token=t_BEGIN then
begin
        scan;
        Statement; {panggil prosedur statement}
        While token=t_SEMICOL do
        Begin
                Scan;
                Statement;
        End;
        If token=t_END then
                scan
end;
end;
```

```
Procedure Statement
Begin
If token=t_ID then
Begin
        Scan;
        If token=t_ASS then
        Begin
                Scan;
                Simple_exp;
        end
end
else
if token = t_IF then
begin
        Scan;
        Exp;
        If token = t_THEN then
        Begin
                Scan;
                Statement;
                If token=t_ELSE  then
                Begin
                        Scan;
                        Statement;
                End;
        End;
End;
End;
```

```
Procedure Exp;
Begin
Simple_Exp;
If (token=t_EQ) OR (token=t_LT) OR (token=t_GT) then
Begin
        Scan;
        Simple_Exp;
End;
End;
```

Semantics Analyser

- Proses ini merupakan proses kelanjutan dari proses kompilasi sebelumnya, yaitu analisa leksikal (scanning) dan analisa sintaks (parsing)
- Bagian terakhir dari tahapan analisis adalah analisis semantik
- Memanfaatkan pohon sintaks yang dihasilkan dari *parsing*
- Proses analisa sintak dan analisa semantik merupakan dua proses yang sangat erat kaitannya, dan sulit untuk dipisahkan

Semantics Analyser

Contoh : A := (A+B) * (C+D)

- *Parser* hanya akan mengenali simbol-simbol ‘:=‘, ‘+’, dan ‘*’, parser tidak mengetahui makna dari simbol-simbol tersebut

- Untuk mengenali makna dari simbol-simbol tersebut, Compiler memanggil routin semantics

Semantics Analyser

Untuk mengetahui makna, maka routin ini akan memeriksa:

- Apakah variabel yang ada telah didefinisikan sebelumnya
- Apakah variabel-variabel tersebut tipenya sama
- Apakah operand yang akan dioperasikan tersebut ada nilainya, dan seterusnya
- Menggunakan tabel simbol
- Pemeriksaan bisa dilakukan pada tabel *identifier*, tabel *display*, dan tabel *block*

Semantics Analyser

Pengecekan yang dilakukan dapat berupa:

- Memeriksa penggunaan nama-nama (keberlakuannya)
  - **Duplikasi**

    Apakah sebuah nama terjadi pendefinisian lebih dari dua kali. Pengecekan dilakukan pada bagian pengelolaan block
  - **Terdefinisi**

    Apakah nama yang dipakai pada program sudah terdefinisi atau belum. Pengecekan dilakukan pada semua tempat kecuali block

- Memeriksa tipe

  Melakukan pemeriksaan terhadap kesesuaian tipe dalam *statement - statement* yang ada, Misalnya bila terdapat suatu operasi, diperiksa tipe operand nya

Semantics Analyser

Contohnya;

- expresi yang mengikut **IF** berarti tipenya boolean, akan diperiksa tipe *identifier* dan tipe ekspresinya
- Bila ada operasi antara dua operand maka *tipe operand* pertama harus bisa dioperasikan dengan *operand* yang kedua

Analisa Semantic sering juga digabungkan dengan *intermediate code* yang akan menghasilkan *output intermediate code*.

Intermediate code ini nantinya akan digunakan pada proses kompilasi berikutnya (pada bagian *back end compilation*)