# PERTEMUAN 7

# ANALISIS LEKSIKAL (SCANNER)

## A. TUJUAN PEMBELAJARAN

Setelah mempelajari materi pada pertemuan ini, Mahasiswa diharap mampu memahami tugas penganalisis leksikal dalam pencarian token

## B. URAIAN MATERI

### 1. Analisis Leksikal

Tugas dasar dari penganalisis leksikal (scanner) adalah untuk memindai karakter per karakter pada kode program sumber dan memecahnya menjadi unit-unit kecil yang bermakna disebut dengan token. Analisis ini melakukan penerjemahan masukan menjadi token, hal itu dilakukan agar mempermudah proses parsing untuk dilakukan, ketimbang membuat nama untuk masing-masing fungsi dan variabel dari banyak karakter yang menyusunnya, maka tugas parsing lebih mudah karena hanya berurusan dengan sekumpulan token.

Tugas-tugas analisis leksikal (scanner) secara detail adalah:

a. Mengidentifikasi suatu bahasa yang dibangun dari semua besaran
b. Merubah ke berbagai token
c. Menentukan jenis-jenis dari token
d. Menindaklanjuti kesalahan
e. Mengurus tabel symbol
f. Pemindai (Scanner), dibuat agar mengetahui keyword, operator, dan identifier
g. Karakter dipisahkan dari bahasa sumber menjadi kelompok yang secara logis dimiliki bersama

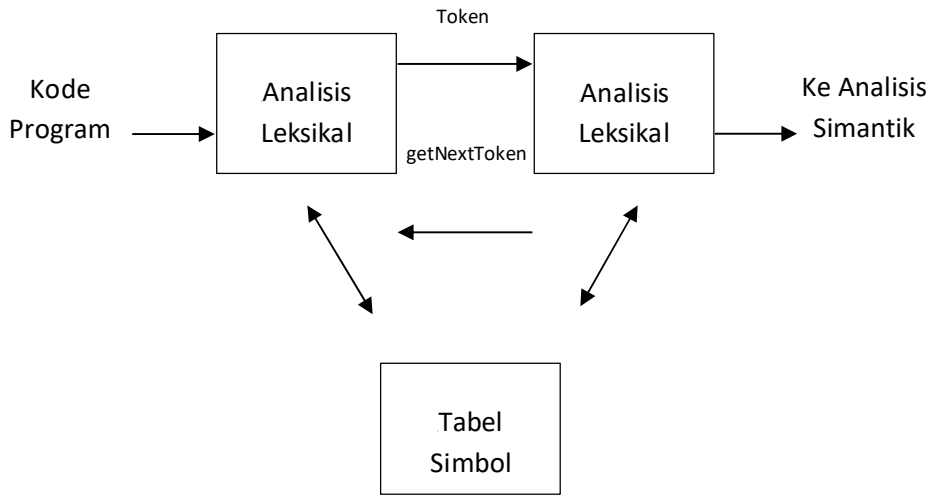
Contoh untuk besaran leksikal adalah:

a. Identifier berupa keyword, contoh If … Else, Begin … End (pascal), Float (bahasa C), dan Integer (pascal)

b. Konstanta berupa bilangan pecahan (float/real), bilangan bulat (integer), karakter, boolean (true/false), dan string
c. Operator berupa operator aritmatika (+ - * /), dan operator logika (< = >)
d. Delimiter sebagai pembatas atau pemisah, seperti kurung buka ( ( ), kurung tutup ( ) ), titik ( . ), koma ( , ), titik dua ( : ), dan titik koma ( ; )
e. White Space pembatas yang tidak dihiraukan oleh program, seperti enter, spasi, ganti baris, akhir file.

Sebagai fase pertama dari sebuah *compiler*, analisis leksikal memiliki tugas utama yaitu mendeteksi karakter yang diinput dari sumber program, mengelompokkannya ke dalam leksem, dan menghasilkan urutan token untuk setiap leksem dalam program sumber sebagai keluaran. Aliran token dikirim ke parser untuk analisis sintaks. Biasanya penganalisis leksikal berinteraksi dengan tabel simbol juga. Ketika penganalisis leksikal menemukan sebuah leksem yang merupakan sebuah pengenal, ia perlu memasukkan leksem itu ke dalam tabel simbol. Dalam beberapa kasus, informasi mengenai jenis pengenal dapat dibaca dari tabel simbol oleh penganalisis leksikal untuk membantunya dalam menentukan token yang tepat yang harus diteruskan ke parser.

Interaksi ini ditunjukkan pada Gambar 1. Biasanya, interaksi diimplementasikan dengan meminta parser memanggil penganalisis leksikal. Panggilan tersebut, disarankan oleh perintah getNextToken, menyebabkan penganalisis leksikal membaca karakter dari inputnya hingga dapat mengidentifikasi leksem berikutnya dan menghasilkan token berikutnya, yang dikembalikan ke parser.




**Gambar 7.1** Interaksi Antara Penganalisis Leksikal Dan Parser Karena Penganalisis Leksikal Adalah Bagian Dari Kompilator Yang Membaca Sumber

Analisis leksikal juga dapat menangani beberapa hal lain, kecuali jika ditangani oleh preprocessor, seperti:

a. Penghapusan komentar.

   Komentar ditandai dengan simbol khusus dan pemindai harus mendeteksi simbol-simbol ini lalu melewati semua string sumber hingga akhir komentar ditemukan.

b. Konversi kasus.

   Dalam kebanyakan bahasa pemrograman, penggunaan huruf besar diabaikan, untuk mencegah nama seperti array size dan Array Size diperlakukan sebagai pengenal yang berbeda, pemindai mengubah seluruh huruf menjadi huruf kecil atau besar secara internal. Dalam konstanta karakter-string, tentu saja, konversi huruf besar / kecil harus dimatikan.

c. Penghapusan ruang kosong.

   Spasi adalah kode karakter apa pun yang menghasilkan ruang kosong saat program dicetak, biasanya mencakup blank, karakter tab, line feed (LF), dan carriage return (CR). Dalam sebagian besar bahasa pemrograman modern, spasi diperlukan untuk memisahkan jenis token tertentu, begitu ruang kosong telah memenuhi tujuan ini, ia harus dilewati. Fortran dirancang untuk tidak membutuhkan ruang kosong sama sekali. Contoh pernyataan:

```
do 100 i = 1, 30, 2
and do100i=1,30,2
```

keduanya legal, dan kebanyakan kompiler FORTRAN mengonversi yang pertama ke yang terakhir selama pemindaian leksikal.

d. Interpretasi dari arahan kompiler.

   Banyak bahasa pemrograman menyediakan cara untuk mengontrol operasi kompiler dari dalam program. Misalnya, di Turbo Pascal, string {$ or (* $ mark a *compiler* directive; {$ R- berarti pemeriksaan rentang akan dinonaktifkan, dan {$ R + berarti harus diaktifkan. Banyak bahasa yang menyertakan direktif. Di C, #include <stdio.h> berarti bahwa konten file bernama stdio.h harus disisipkan ke dalam kode sumber tepat di tempat petunjuk muncul. Penganalisis leksikal harus membuka file ini dan membacanya sampai habis, kemudian penganalisis harus menutup file dan kembali ke file program utama.

e. Komunikasi dengan tabel simbol.

   Kapanpun sebuah identifier dideklarasikan, *compiler* harus membuat entri tabel baru untuk itu. Entri tabel biasanya menyertakan atribut pengenal. Misalnya, tipe datanya dan ukurannya. Dalam bahasa yang sangat diketik, setiap pengenal harus dideklarasikan sebelum digunakan. Dalam bahasa seperti itu, di mana pun pengenal digunakan, pemindai harus memeriksa apakah ada entri tabel simbol untuknya dan harus mengumumkan kesalahan jika tidak ada. Jika ada entri, pemindai biasanya meneruskan alamat entri itu ke parser.

f. Persiapan daftar keluaran.

   Kebanyakan kompiler membuat file yang berisi versi kode sumber program yang telah dianotasi. Penganalisis leksikal biasanya membuat file ini. Selama pemindaian, ia harus menghitung baris-baris kode sumber, biasanya menulis setiap baris bernomor ke file daftar. Pemindai juga harus memasukkan pesan kesalahan dan peringatan sesuai kebutuhan, dan harus menyiapkan tabel data ringkasan yang muncul di akhir sebagian besar daftar sumber.

   Terkadang, penganalisis leksikal dibagi menjadi dua proses yang berjalan:

1) Pemindaian terdiri dari proses sederhana yang tidak memerlukan tokenisasi input, seperti penghapusan komentar dan pemadatan karakter spasi putih yang berurutan menjadi satu.
2) Analisis leksikal yang tepat adalah bagian yang lebih kompleks, di mana pemindai menghasilkan urutan token sebagai keluaran.

**2. Kesalahan Leksikal**

Sulit bagi penganalisis leksikal untuk melakukan tanpa bantuan komponen lain, bahwa ada kesalahan kode sumber. Misalnya, jika string f i ditemukan pertama kali dalam program C dalam konteks:

fi ( a == f(x)) ...

penganalisis leksikal tidak dapat mengetahui apakah f i adalah salah eja dari kata kunci if atau pengenal fungsi yang tidak dideklarasikan. Karena f i adalah lexeme yang valid untuk id token, penganalisis leksikal harus mengembalikan id token ke parser dan membiarkan beberapa fase lain dari *compiler* - mungkin parser dalam kasus ini - menangani kesalahan karena transposisi huruf.

Namun, misalkan muncul situasi di mana penganalisis leksikal tidak dapat dilanjutkan karena pola token yang ada tidak cocok dengan awalan apa pun dari input yang tersisa. Strategi pemulihan yang paling sederhana adalah pemulihan "mode panik". Kami menghapus karakter berurutan dari input yang tersisa, hingga penganalisis leksikal dapat menemukan token yang terbentuk dengan baik di awal input apa yang tersisa. Teknik pemulihan ini mungkin membingungkan parser, tetapi dalam lingkungan komputasi interaktif mungkin cukup memadai.

Tindakan pemulihan kesalahan lainnya yang mungkin adalah:

a. Hapus satu karakter dari input yang tersisa.
b. Sisipkan karakter yang hilang ke input yang tersisa.
c. Mengganti karakter dengan karakter lain.
d. Ubah urutan dua karakter yang berdekatan.

Transformasi seperti ini dapat dicoba untuk memperbaiki input. Strategi yang paling sederhana adalah untuk melihat apakah prefiks dari input yang tersisa dapat diubah menjadi leksem yang valid dengan transformasi tunggal. Strategi ini masuk akal, karena dalam praktiknya kebanyakan kesalahan

leksikal melibatkan satu karakter. Strategi koreksi yang lebih umum adalah menemukan jumlah transformasi terkecil yang diperlukan untuk mengubah program sumber menjadi program yang hanya terdiri dari leksem yang valid, tetapi pendekatan ini dianggap terlalu mahal dalam praktiknya sehingga tidak sebanding dengan upaya yang dilakukan.

### 3. Token, Pola (Patterns), dan Leksem (Lexemes)

Pada analisis leksikal, ada tiga istilah yang terkait tetapi berbeda, yaitu:

a. Token adalah nilai simbol abstrak yang mewakili unit leksikal. Misalnya kata kunci tertentu, atau urutan karakter input yang menunjukkan pengenal. Nama token adalah simbol input yang diproses parser. Berikut ini, kami biasanya akan menulis nama token dalam huruf tebal. Kami akan sering merujuk ke token dengan nama tokennya.
b. Pola (Patterns) adalah deskripsi dari bentuk yang mungkin diambil oleh leksem token. Dalam kasus kata kunci sebagai token, polanya hanyalah urutan karakter yang membentuk kata kunci. Untuk pengenal dan beberapa token lainnya, polanya adalah struktur yang lebih kompleks yang dicocokkan dengan banyak string.
c. Leksem (Lexemes) adalah urutan karakter dalam program sumber yang cocok dengan pola token dan diidentifikasi oleh penganalisis leksikal sebagai turunan dari token itu.

Contoh nilai token, pola yang dideskripsikan secara informal, dan beberapa contoh leksem. Untuk melihat bagaimana konsep-konsep ini digunakan dalam praktiknya, dalam bahasa C di bawah ini:

```
printf("Total = %d\n", score);
```

printf dan score adalah leksem yang cocok dengan pola untuk token id, dan "Total = %d\n" adalah literal yang cocok dengan leksem.

Dalam banyak bahasa pemrograman, kelas-kelas berikut mencakup sebagian besar atau semua token:

**Tabel 7.1** Contoh Token

| Token | Deskripsi Informal | Contoh Leksem |
|---|---|---|
| If | karakter i, f | if |
| Then | karakter t, h, e, n | then |
| Perbandingan | <=, =>, ==, !=, <, > | <=, == |
| ID | huruf diikuti oleh huruf dan angka | phi, D2 |
| Angka | semua konstanta numerik | 3.14159, 6.02e23 |
| Literal | apapun kecuali menggunakan “ | “core dumped” |

a. Satu token untuk setiap kata kunci. Pola kata kunci sama dengan kata kunci itu sendiri.
b. Token untuk operator, baik secara individu atau dalam kelas seperti perbandingan token yang disebutkan pada Tabel 7.1.
c. Satu token yang mewakili semua pengenal.
d. Satu atau lebih token yang mewakili konstanta, seperti angka dan string literal.
e. Token untuk setiap simbol tanda baca, seperti tanda kurung kiri dan kanan, koma, dan titik koma.

**4. Atribut untuk Token**

Ketika lebih dari satu leksem dapat cocok dengan suatu pola, penganalisis leksikal harus menyediakan fase kompilator selanjutnya informasi tambahan tentang leksem tertentu yang cocok. Misalnya, pola untuk nomor token cocok dengan 0 dan 1, tetapi sangat penting bagi pembuat kode untuk mengetahui leksem mana yang ditemukan dalam program sumber. Jadi, dalam banyak kasus penganalisis leksikal kembali ke parser tidak hanya nama token, tetapi nilai atribut yang mendeskripsikan lexeme yang diwakili oleh token, nama token memengaruhi keputusan penguraian, sedangkan nilai atribut memengaruhi terjemahan token setelah penguraian.

Kami akan berasumsi bahwa token memiliki paling banyak satu atribut terkait, meskipun atribut ini mungkin memiliki struktur yang menggabungkan beberapa informasi. Contoh paling penting adalah token id, di mana kita perlu mengasosiasikan dengan token banyak informasi. Biasanya, informasi tentang identifier _ e.g.^ leksemnya, tipenya, dan lokasi di mana ia pertama kali ditemukan (jika pesan kesalahan tentang pengenal itu harus dikeluarkan) disimpan dalam tabel simbol. Jadi, nilai atribut yang sesuai untuk pengenal adalah penunjuk ke entri tabel-simbol untuk pengenal tersebut.

Biasanya masalah rumit saat mengenali token adalah mengingat pola yang menggambarkan leksem-leksem sebuah token, relatif mudah untuk mengenali leksem-lexem yang cocok ketika mereka muncul pada input. Namun, dalam beberapa bahasa tidak segera terlihat ketika kita telah melihat sebuah instance dari sebuah lexeme yang sesuai dengan sebuah token. Contoh berikut diambil dari Fortran, dalam format tetap yang masih diizinkan di Fortran 90. Dalam pernyataan:

```
DO  5  I = 1.25
```

tidak jelas bahwa leksem pertama adalah D05I, turunan dari token pengenal, sampai kita melihat titik setelah 1. Perhatikan bahwa kosong dalam format tetap Fortran diabaikan (konvensi kuno). Seandainya kita melihat koma dan bukan titik, kita akan memiliki pernyataan:

```
DO  5  I = 1,25
```

di mana leksem pertama adalah kata kunci DO.

Nama token dan nilai atribut terkait pada pernyataan Fortran:

```
E = M * C ** 2
```

ditulis di bawah ini sebagai urutan pasangan.

<id, penunjuk ke input tabel simbol untuk E>

<assign_op>

<id, penunjuk ke input tabel simbol untuk M>

<mult_op>

<id, penunjuk ke input tabel simbol untuk C>

<exp_op>

<angka, nilai integer 2>

Perhatikan bahwa dalam pasangan tertentu, terutama operator, tanda baca, dan kata kunci, tidak diperlukan nilai atribut. Dalam contoh ini, nomor token telah diberi atribut bernilai integer. Dalam praktiknya, kompiler tipikal malah akan menyimpan string karakter yang mewakili konstanta dan digunakan sebagai nilai atribut untuk nomor penunjuk ke string tersebut.

Setiap token adalah urutan karakter yang mewakili unit informasi dalam program sumber. Contoh umumnya adalah kata kunci, seperti if dan while, for dan do, yang merupakan string huruf tetap. Program, ketika tiba di masukan ke kompilator, seperti satu string karakter yang sangat besar. Contohnya seperti:

```
for k := 1 to max do
x[k] := 0;
```

dan sebelum kompilator memulai menerjemahkan seperti di atas, ia juga harus mengidentifikasi elemen yang berarti dari pernyataan tersebut. Kita harus membuat ide alternatif yang lebih tepat untuk masalah yang dihadapi kompilator dengan menulis pernyataan tersebut dalam biner, menggunakan representasi karakter ASCII:

```
09 66 6F 72 20 69 20 3A 3D 20 31 20 64 6F 20 6D
61 78 20 64 6F 0D 0A 09 20 20 20 78 5B 69 5D 20
3A 3D 20 30 3B 0D 0A
```

Representasi ini menempatkan kita pada kerugian yang sama seperti komputer, di mana huruf, simbol, dan spasi yang sudah dikenal hilang. Apa yang dilihat kompilator adalah urutan string bit ini, dan tugas penganalisis leksikal adalah memecah pernyataan ini menjadi token dan analisis leksikal terkadang juga disebut tokenizing. Dalam pernyataan ini, unit yang berarti adalah

kata kunci : for, to, do

pengenal : i, x, max

konstanta : 0, 1

operator : = :

tanda baca : ;

tanda kurung : [ , ]

Tokenisasi relatif mudah di sebagian besar bahasa pemrograman modern, karena pemrogram diharuskan memisahkan banyak bagian pernyataan dengan karakter kosong atau tab (spasi), blank ini memudahkan kompilator untuk menentukan di mana satu token berakhir dan token berikutnya dimulai.

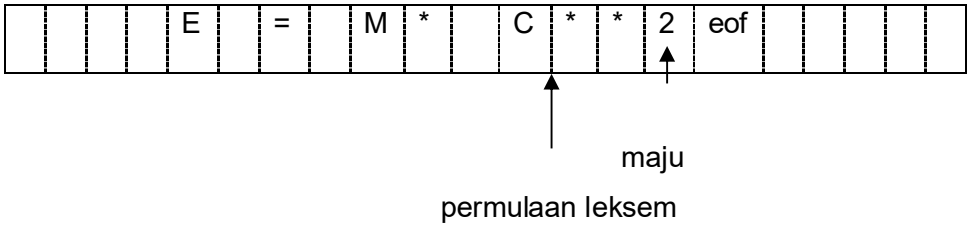
**5. Penyangga Masukkan (Input Buffering)**

Sebelum membahas masalah mengenali leksem dalam input, mari kita periksa beberapa cara agar tugas sederhana namun penting membaca program sumber dapat dipercepat. Tugas ini dipersulit oleh fakta bahwa kita sering kali harus melihat satu atau lebih karakter di luar leksem berikutnya sebelum kita dapat yakin bahwa kita memiliki leksem yang tepat, ada banyak situasi di mana kita perlu melihat setidaknya satu karakter tambahan di depan. Misalnya, kita tidak dapat memastikan bahwa kita telah melihat akhir pengenal sampai kita melihat karakter yang bukan huruf atau angka, dan oleh karena itu bukan bagian dari leksem untuk *id*. Di bahasa C, operator satu karakter seperti -, =, atau < juga bisa menjadi awal dari operator dua karakter seperti ->, ==, atau <=. Jadi, kami akan memperkenalkan skema dua buffer yang menangani lookahead besar dengan aman. Kami kemudian mempertimbangkan peningkatan yang melibatkan "penjaga" yang menghemat waktu memeriksa ujung buffer.

a. Pasangan Penyangga (Buffer Pairs)

Karena jumlah waktu yang dibutuhkan untuk memproses karakter dan sejumlah besar karakter yang harus diproses selama kompilasi program sumber yang besar, teknik buffering khusus telah dikembangkan untuk

mengurangi jumlah overhead yang diperlukan untuk memproses karakter input tunggal. Skema penting melibatkan dua buffer yang dimuat ulang secara bergantian, seperti yang disarankan pada Gambar 7.2.




**Gambar 7.2** Menggunakan Sepasang Input Buffer

Dua petunjuk dari gambar di atas adalah:

1) Petunjuk permulaan leksem, menandai awal dari leksem saat ini, yang sejauh mana kita coba tentukan.
2) Petunjuk maju memindai ke depan sampai pola yang cocok ditemukan.

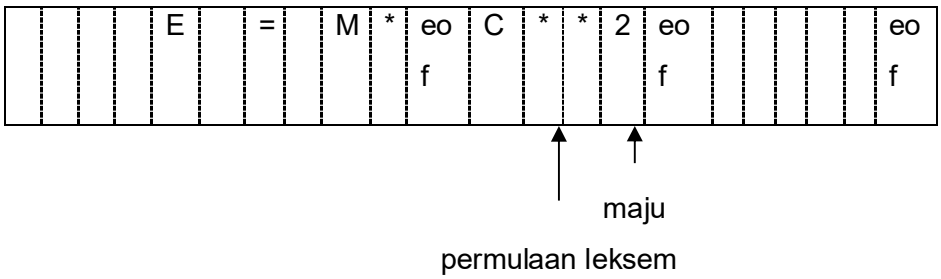
Setelah leksem berikutnya ditentukan, lalu diatur ke karakter di ujung kanannya. Kemudian, setelah leksem direkam sebagai nilai atribut dari sebuah token yang dikembalikan ke parser, permulaan leksem disetel ke karakter segera setelah leksem yang baru ditemukan. Pada Gambar 7.2, kita melihat ke depan telah melewati akhir leksem berikutnya, ** (operator eksponen Fortran), dan satu posisi harus ditarik ke kiri.

Maju ke depan mengharuskan kita menguji terlebih dahulu apakah kita telah mencapai akhir salah satu buffer, dan jika demikian, kita harus memuat ulang buffer lain dari masukan, dan maju ke awal buffer yang baru dimuat. Selama kita tidak perlu melihat jauh ke depan dari leksem yang sebenarnya sehingga jumlah panjang leksem ditambah jarak yang kita lihat ke depan lebih besar dari N, kita tidak akan pernah menimpa leksem dalam buffer sebelum menentukannya.

b. Penjaga (Sentinels)

Jika kita menggunakan skema buffer pairs seperti yang dijelaskan, kita harus memeriksa, setiap kali kita maju, bahwa kita belum pindah dari salah satu buffer; jika kita melakukannya, maka kita juga harus memuat ulang buffer lainnya. Jadi, untuk setiap karakter yang dibaca, kami membuat dua tes, satu untuk akhir buffer, dan satu lagi untuk menentukan karakter apa

yang dibaca (yang terakhir mungkin merupakan cabang multiway). Kita bisa menggabungkan tes buffer-end dengan tes untuk karakter saat ini jika kita memperluas setiap buffer untuk menahan karakter sentinel di akhir. Sentinel adalah karakter khusus yang tidak dapat menjadi bagian dari program sumber, dan pilihan yang wajar adalah karakter eof.




**Gambar 7.3** Sentinel Diakhir Setiap Buffer

Gambar 7.3 menunjukkan pengaturan yang sama seperti gambar 7.2, tetapi dengan penambahan sentinel. Perhatikan bahwa eof tetap menggunakannya sebagai penanda untuk akhir seluruh masukan. Setiap eof yang muncul selain di akhir buffer berarti input berada di akhir.

## 6. Spesifikasi Token

Ekspresi reguler adalah notasi penting untuk menentukan pola leksem. Meskipun mereka tidak dapat mengungkapkan semua pola yang mungkin, mereka sangat efektif dalam menentukan jenis pola yang sebenarnya kita butuhkan untuk token.

a. String dan Bahasa

Alfabet adalah kumpulan simbol yang terbatas. Contoh simbol yang umum adalah huruf, angka, dan tanda baca. Himpunan {0,1} adalah alfabet biner. ASCII adalah contoh penting alfabet; itu digunakan di banyak sistem perangkat lunak. Uni-code, yang mencakup sekitar 100.000 karakter dari alfabet di seluruh dunia, adalah contoh penting alfabet lainnya.

String di atas alfabet adalah urutan simbol terbatas yang diambil dari alfabet itu. Dalam teori bahasa, istilah "kalimat" dan "kata" sering digunakan sebagai sinonim untuk "string". Panjang string $s$, biasanya ditulis $|s|$, adalah banyaknya kemunculan simbol dalam $s$. Misalnya, pisang adalah untaian

panjang enam. String kosong, dilambangkan dengan $\epsilon$, adalah string dengan panjang nol.

Bahasa adalah kumpulan string yang dapat dihitung di atas beberapa alfabet tetap. Definisi ini sangat luas. Bahasa abstrak seperti $\emptyset$, himpunan kosong, atau $\{\epsilon\}$, himpunan yang hanya berisi string kosong, adalah bahasa di bawah definisi ini. Begitu juga himpunan dari semua program C yang dibentuk dengan baik secara sintaksis dan himpunan dari semua kalimat bahasa Inggris yang secara tata bahasa benar, meskipun dua bahasa terakhir sulit untuk ditentukan dengan tepat. Perhatikan bahwa definisi "bahasa" tidak mengharuskan makna apa pun diberikan ke string dalam bahasa tersebut.

Jika $x$ dan $y$ adalah string, maka rangkaian dari $x$ dan $y$, dilambangkan dengan $xy$ , adalah string yang dibentuk dengan menambahkan $y$ ke $x$. Misalnya, jika x = rumah dan $y$ = kucing, maka $xy$ = rumah kucing. String kosong adalah identitas di bawah rangkaian; yaitu, untuk setiap string $s, \epsilon s = s\epsilon = s$.

Jika kita menganggap penggabungan sebagai produk, kita dapat mendefinisikan "eksponensial" dari string sebagai berikut. Definisikan $s^0$ menjadi $\epsilon$, dan untuk semua $i > 0$, definisikan $s^i$ menjadi $s^{i-1}s$. Karena $\epsilon s = s$, maka $s^1 = s$. Kemudian $s^2 = ss, s^3 = sss$, dan seterusnya.

b. Operasi di Bahasa

Dalam analisis leksikal, operasi terpenting pada bahasa adalah union, concatenation, dan closure, yang didefinisikan secara formal pada Gambar 4. Union adalah operasi yang lazim di set. Rangkaian bahasa adalah semua string yang dibentuk dengan mengambil string dari bahasa pertama dan string dari bahasa kedua, dengan segala cara yang memungkinkan, dan menggabungkannya. Penutupan (Kleene) dari bahasa $L$, dilambangkan dengan $L^*$, adalah himpunan string yang Anda dapatkan dengan menggabungkan $L$ nol atau lebih. Perhatikan bahwa $L^0$, "rangkaian $L$ nol kali," didefinisikan sebagai $\{\epsilon\}$, dan secara induktif, $L^i$ adalah $L^{i-1}L$. Akhirnya, penutupan positif, dilambangkan dengan $L^+$, adalah sama dengan penutupan Kleene, tetapi tanpa suku $L^0$. Artinya, $\epsilon$ tidak akan berada di $L^+$ kecuali di $L$ itu sendiri.

**Tabel 7.2** Definisi Operasi pada Bahasa

| Operasi | Definisi dan Notasi |
|---|---|
| Union dari $L$ dan $M$ | $L \cup M = \{s \mid s\ ada\ di\ L\ atau\ ada\ di\ M\}$ |
| Concatenation dari $L$ dan $M$ | $LM = \{st \mid s\ ada\ di\ L\ dan\ t\ ada\ di\ M\}$ |
| Penutupun Kleene dari $L$ | $L^* = \cup_{i=0}^{\infty} L^i$ |
| Penutupan Positif dari $L$ | $L^+ = \cup_{i=1}^{\infty} L^i$ |

c. Ekspresi Reguler

Misalkan kita ingin mendeskripsikan himpunan pengenal C yang valid. Ini hampir persis seperti bahasa yang dijelaskan pada butir (5) di atas; satu-satunya perbedaan adalah bahwa garis bawah disertakan di antara huruf-huruf.

Dalam Contoh 3.3, kami dapat mendeskripsikan pengidentifikasi dengan memberikan nama untuk kumpulan huruf dan angka dan menggunakan gabungan operator bahasa, penggabungan, dan penutupan. Dalam notasi ini, jika letter_ ditetapkan untuk mewakili huruf apa pun atau garis bawah, dan digit_ ditetapkan untuk mewakili digit apa pun, maka kita dapat mendeskripsikan bahasa pengenal C dengan:

*letter_ (letter_ | digit) **

Bagian vertikal di atas berarti penyatuan, tanda kurung digunakan untuk mengelompokkan sub ekspresi, bintang berarti "nol atau lebih kemunculan", dan penjajaran letter_ dengan sisa ekspresi menandakan penggabungan.

Ekspresi reguler dibuat secara rekursif dari ekspresi reguler yang lebih kecil, menggunakan aturan yang dijelaskan di bawah ini. Setiap ekspresi reguler $r$ menunjukkan bahasa $L(r)$, yang juga didefinisikan secara rekursif dari bahasa yang dilambangkan dengan sub ekspresi $r$. Berikut adalah aturan yang menentukan ekspresi reguler di beberapa alfabet $\sum$ dan bahasa yang ditunjukkan oleh ekspresi tersebut.

Ada dua aturan yang menjadi dasar:

1) $\epsilon$ adalah ekspresi reguler, dan $L(\epsilon)$ adalah $\{\epsilon\}$, yaitu bahasa yang anggota tunggalnya adalah string kosong.
2) Jika $a$ adalah simbol $\sum$, maka $\alpha$ adalah ekspresi reguler, dan $L\,(a) = \{a\}$, yaitu bahasa dengan satu string, panjang satu, dengan a di satu posisinya. Perhatikan bahwa menurut ketentuan, kami menggunakan huruf miring untuk simbol, dan huruf tebal untuk ekspresi reguler yang sesuai.

d. Definisi Reguler

Untuk kenyamanan notasi, kami mungkin ingin memberi nama pada ekspresi reguler tertentu dan menggunakan nama tersebut dalam ekspresi berikutnya, seolah-olah nama itu adalah simbol itu sendiri. Jika $\sum$ adalah alfabet lambang dasar, maka definisi reguler adalah sebagai berikut:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
…
$$d_n \rightarrow r_n$$

dimana:

1) Setiap $d_i$ adalah simbol baru, tidak di $\sum$ dan tidak sama dengan yang lainnya di $d$, dan
2) Setiap $r_i$ adalah ekspresi reguler di atas alfabet $\sum \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

e. Ekstensi Ekspresi Reguler

Sejak Kleene memperkenalkan ekspresi reguler dengan operator dasar untuk penyatuan, penggabungan, dan penutupan Kleene pada tahun 1950-an, banyak ekstensi telah ditambahkan ke ekspresi reguler untuk meningkatkan kemampuannya untuk menentukan pola string. Di sini kami menyebutkan beberapa ekstensi notasi yang pertama kali dimasukkan ke dalam utilitas Unix seperti Lex. yang sangat berguna dalam penganalisis spesifikasi leksikal.

1) *Satu atau lebih contoh*. Unary, operator postfix $+$ mewakili penutupan positif dari ekspresi reguler dan bahasanya. Artinya, jika $r$ adalah ekspresi reguler, maka $(r)^+$ menunjukkan bahasa $\big(L(r)\big)^+$. Operator $+$ memiliki prioritas dan asosiasi yang sama dengan operator $*$. Dua

hukum aljabar yang berguna, $r^* = r^+|\epsilon$ dan $r^+ = rr^* = r^*r$ menghubungkan penutupan Kleene dan penutupan positif.

2) *Nol atau satu contoh*. Operator postfix unary? berarti "nol atau satu kejadian". Artinya, $r$? setara dengan $r|\epsilon$, atau dengan kata lain, $L\,(r?) = L(r) \cup \{\epsilon\}$ = L (r) U {e}. Itu? operator memiliki prioritas dan asosiatif yang sama dengan $*$ dan $+$.

3) *Kelas karakter*. Ekspresi reguler $d_1|d_2|\ldots|d_n$, di mana $d_i$ adalah setiap simbol alfabet, dapat diganti dengan sederhana $[d_1 d_2 \ldots d_n]$. Lebih penting lagi, ketika $d_1, d_2, \ldots, d_n$ adalah suatu bentuk urutan logis, misalnya, huruf besar yang berurutan, huruf kecil, atau angka, kita dapat menggantinya dengan $d_i - d_n$, yaitu, hanya yang pertama dan terakhir dipisahkan oleh sebuah tanda hubung. Jadi, $[def]$ adalah singkatan dari $d \mid e \mid f$, dan $[a - z]$ adalah singkatan dari $d \mid e \mid \ldots \mid z$.

## C. SOAL LATIHAN

1. Buatlah kesimpulan definisi Analisis Leksikal sesuai dengan bahasa Anda sendiri berdasarkan para ahli selain yang telah dijelaskan pada modul ini!
2. Menurut Anda, bagaimana cara kerja Analisis Leksikal, jelaskan!
3. Jelaskan bahasa yang dilambangkan dengan ekspresi reguler berikut:
   a. $b(b|c) * b$.
   b. $((\epsilon|b)c *) *$.
   c. $b * cb * cb * cb *$.

## D. REFERENSI

Alfred V. Aho, Monica S., Ravi Sethi, Jeffrey D. (2007). *Compilers "Priciples, Techniques, & Tools"*, 2nd Edition. Boston, San Francisco, New York.

Jean-Paul Tremblay, Paul G. Sorenson. (1995). *"The Theory and Practice of Compiler Writing"*, United State of America.

Kenneth C. Louden. (1997). "*Compiler Construction: Principles and Practice"*.

Thomas W. Parsons. (1992). *"Introduction to Compiler Construction"*. New York: Computer Science Press.

Jean-Paul Tremblay & Paul G. Sorenson. (1985). "*The Theory and Practice of Compiler Writing"*. United State of America: International Edition.

Hari Soetanto. (2004). "*Teknik Kompilasi*". Fakultas Teknologi Informasi Universitas Budi Luhur.

# GLOSARIUM

**Compiler** adalah suatu program yang menerjemahkan bahasa program ( source code) kedalam bahasa objek (obyek code).

**Ekstensi file (file extensions)** adalah sekumpulan karakter yang ditambahkan di posisi akhir nama file.