OTOMATA 09

LEXICAL ANALYSIS & SYNTAX ANALYSIS

Mahasiswa mengetahui metode dan urutan proses di dalam tahap analisis leksikal & analisis sintaks

Materi Pertemuan

- Lexical Analysis Overview
- Lexical Analysis Generator
- Studi Kasus : Lex
- The Role of Parsing
- Syntax Errors & Handling
- General Parsing Methods
- Top-Down Parsing Approaches :
  - Brute-Force Approach
  - Recursive-Descent Parsing
  - Top-Down Parsing with Limited Backup
- Studi Kasus : LL(1) Grammar

Lexical Analysis (1)

Secara khusus, Lexical Analyzer (Scanner) memiliki tugas :

1. Melakukan pembacaan program, karakter per karakter, dan membentuk rangkaian token;
2. Mengabaikan komentar dan membuang white space (blank space, tab, dan newline character);
3. Mengelompokkan token-token sesuai dengan klasifikasi/ besarannya;
4. Mengisi record-record tabel simbol;
5. Mengenali kesalahan leksikal;

Lexical Analysis (2)

Jika tugasnya 'hanya' membaca source program, mengapa tahap lexical analysis tidak digabung saja dengan sintaks analysis ?

- **Menyederhanakan desain compiler**

  jika digabung, bagaimana menangani white space dalam grammar/ parsing? Dengan terpisah, maka kita dapat mengelimnir terlebih dahulu white space tersebut sebelum di-parsing

- **Memudahkan peningkatan kemampuan compiler**

  dengan terpisah, maka kita lebih mudah mencari metode untuk mempercepat pembacaan karakter source program (pembacaan adalah fase terlama dalam proses kompilasi)

- **Menyederhanakan pemrosesan kompilasi**

  karakter 'aneh-aneh' yang diperkenankan untuk dipergunakan oleh suatu bahasa pemrograman, dapat dilokalisir di dalam tahap leksikal analysis saja (untuk kemudian direpresentasikan menjadi token tertentu

Lexical Analysis (3)

Bentuk kesalahan leksik yang kerap kali terjadi adalah kekurang sempurnaan penulisan string oleh programmer. Sehingga scanner sulit mengidentifikasi jenis lexeme yang ditemukan. Misalnya perintah pencabangan "IF" yang ditulis "FI" dsb.

Sementara bentuk error-recovery yang dapat dilakukan oleh scanner meliputi, antara lain :

- menghapus karakter berlebih;
- menyisipkan karakter yang kurang;
- mengganti karakter yang salah;
- menukar posisi 2 adjacent characters;

Note :

Lexeme = rangkaian karakter (string) yang baru saja dibaca oleh scanner

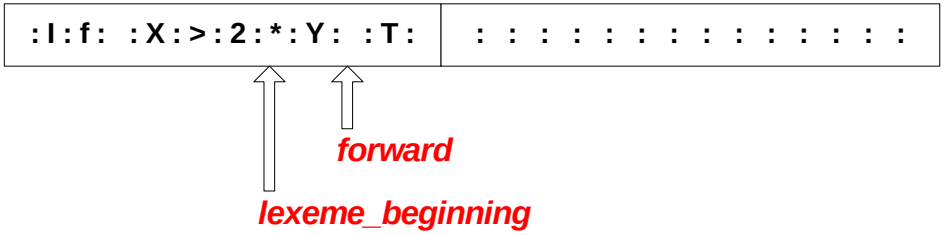
Token = lexeme yang telah dapat diidentifikasi jenisnya (sehingga dapat dimasukkan ke dalam tabel simbol)

Lexical Analysis (4)

Dengan issue bahwa proses pembacaan source program membutuhkan waktu terlama, maka perlu dicarikan algoritma yang dapat mempercepat proses tersebut :

Buffer Pairs

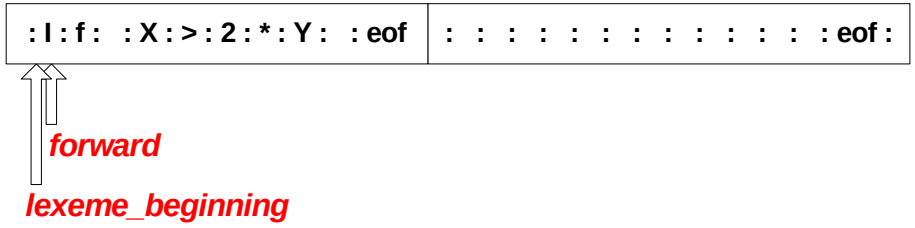
- sebuah buffer disiapkan sebanyak 2 x N karakter
- sejumlah N karakter dimasukkan ke buffer pertama
- 2 pointer dipekerjakan : ***lexeme_beginning*** dan ***forward***
- ***forward*** bergerak maju per karakter untuk mendapatkan pola token (misal sejumlah n karakter, termasuk white space)
- ***lexeme_beginning*** bergerak maju per karakter untuk mendapatkan token tersebut (sejumlah n-1 karakter, tidak termasuk white space)
- jika ***forward*** mencapai batas buffer, maka sejumlah N karakter dimasukkan ke buffer berikutnya

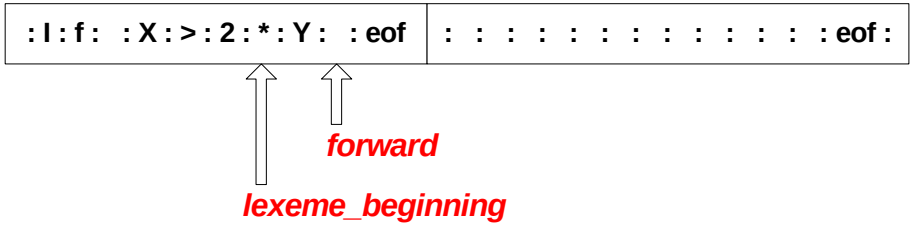


Lexical Analysis (5)

Sentinels

- sebuah buffer disiapkan sebanyak 2 x N karakter
- sejumlah N karakter dimasukkan ke buffer pertama
- 2 pointer dipekerjakan : ***lexeme_beginning*** dan ***forward***
- ***forward*** bergerak maju per karakter untuk mendapatkan pola token (misal sejumlah n karakter, termasuk white space)
- ***lexeme_beginning*** bergerak maju per karakter untuk mendapatkan token tersebut (sejumlah n-1 karakter, tidak termasuk white space)
- jika ***forward*** membaca tanda batas buffer (berupa special character : ***eof***), maka sejumlah N karakter dimasukkan ke buffer berikutnya





Lexical Analysis (6)

Beberapa pendekatan yang dapat digunakan untuk membangun Lexical Analyzer, yaitu :

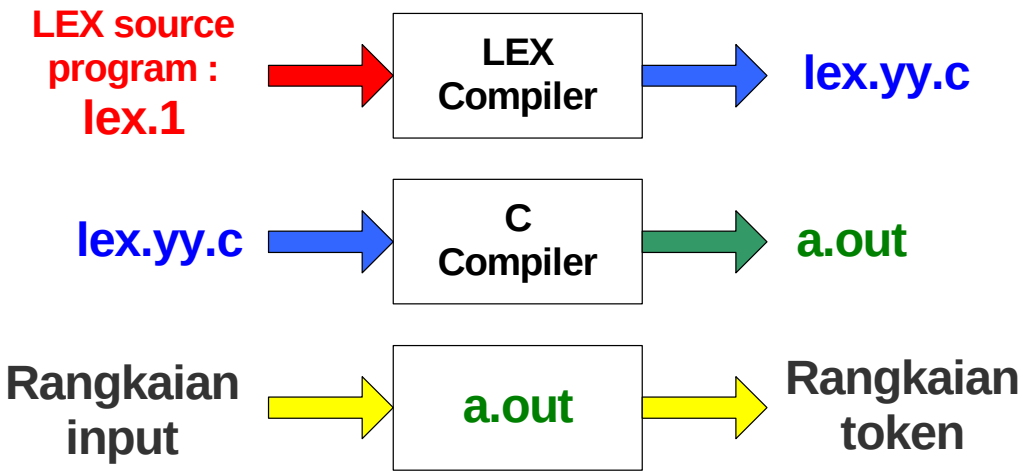
1. Menggunakan Lexical Analyzer Generator, seperti LEX. Kita tinggal mengembangkan model seperti yang kita kehendaki, dan fasilitas LEX yang akan menangani pembacaan dan buffering input string;

2. Menulis Lexical Analyzer melalui bahasa pemrograman konvensional atau hi-level language dan memanfatkan fasilitas I/O bahasa pemrograman tersebut untuk membaca input string;

3. Menulis Lexical Analyzer dalam bahasa Assembly dan merancang sendiri fasilitas pembacaan dan bufering input string;

Lexical Analyzer Generator (1)

- LEX (Lex Compiler) adalah sebuah tool untuk mengembangkan scanner.
- LEX dibuat oleh **Lesk** dan **Schmidt** pada tahun 1975;
- Mekanisme kerja LEX adalah sebagai berikut :



- Program LEX memiliki 3 bagian, yaitu :

  declaration
  % %
  transition rules
  % %
  user subprograms

Lexical Analyzer Generator (2)

contoh :

1. Menspesifikasikan bahasa

stmt ➔ **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| ε

expr ➔ term relop term
| term

term ➔ id
| num

relop ➔ < | <= | = | <> | > | >=

id ➔ letter (letter | digit) *

num ➔ digit$^+$ (**.** Digit$^+$) ? (**E** (+ | -) ? Digit$^+$) ?

letter ➔ **A** | **B** | . . . | **Z** | **a** | **b** | . . . | **z**

digit ➔ **0** | **1** | **2** | . . . | **9**

Lexical Analyzer Generator (3)

2. Mendefinisikan nilai atribut untuk setiap token

Regular Expression	Token	Attribute-Value
ws	-	
if	t_if	
then	t_then	
else	t_else	
id	t_id	pointer to table entry
num	t_num	pointer to table entry
<	t_relop	LT
<=	t_relop	LE
=	t_relop	EQ
<>	t_relop	NE
>	t_relop	GT
>=	t_relop	GE

Lexical Analyzer Generator (4)

3. Mengembangkan scanner generator dengan menggunakan LEX

```
%{ /* pendefinisian dari konstanta LT, LE, EQ, NE, GT, GE, IF, THEN, ELSE, ID, NUM, RELOP */ %}

/* pendefinisian regular */
delim           [ \ t \ n ]
ws              { delim }+
letter          [ A – Z a – z ]
digit           [ 0 – 9 ]
id              { letter } ( { letter } | { digit } )*
num             { digit }+ ( \ . { digit }+ ) ? ( E [ + \ - ] ? { digit }+ ) ?

%%
{ ws }          { /* no action and no return 8? }
if              { return ( IF ) }
then            { return ( THEN ) }
else            { return ( ELSE ) }
{ id }          { yylval = install_id(); return ( ID ) }
{ num }         { yylval = install_num(); return ( NUM ) }
“ < “           { yylval = LT; return ( RELOP ) }
“ <= “          { yylval = LE; return ( RELOP ) }
“ = “           { yylval = EQ; return ( RELOP ) }
“ <> “          { yylval = NE; return ( RELOP ) }
“ > “           { yylval = GT; return ( RELOP ) }
“ >= “          { yylval = GE; return ( RELOP ) }
%%
```

Lexical Analyzer Generator (5)

```
install_id() {
/* prosedur untuk melakukan instalasi leksi, yang karakter pertamanya ditunjuk oleh yytext, dan
    yang memiliki panjang yyleng, ke dalam tabel simbol, dan mengirimkan pointer */
}

install_num() {
/*  serupa dengan instal_id(), tetapi ini untuk angka */
}
```

untuk lebih jelasnya, anda dapat mencari dan mempelajari reference manual LEX yang ditulis oleh Lesk and Schmidt.

The Role of Parsing

Setiap bahasa pemrograman pasti memiliki aturan-aturan yang menjelaskan bagaimana struktur sintaktik dari sebuah program yang baik.

Sebagai contoh, program dalam bahasa Pascal dibangun dari kumpulan blok. Sementara blok disusun dari statement-statement. Statemen disusun dari kumpulan ekspresi. Ekspresi disusun dari rangkaian token, dst.

Struktur sintaks dari sebuah bahasa pemrograman umumnya dijelaskan melalui context-free grammar atau BNF.

Syntax Analyzer (Parser) menerima inputan berupa rangkaian token dari Scanner. Parser akan memverifikasi apakah strings of tokens tersebut dapat di-generated oleh grammar bahasa ybs.

Syntax Errors & Handling

Seandainya compiler hanya memproses/meng-compile program-program yang benar, maka tentu saja merancang dan membangun compiler akan menjadi pekerjaan yang lebih mudah.

Namun kenyataannya, programmer seringkali (sengaja atau tidak sengaja) membuat kesalahan dalam menulis program.

Dan sebuah compiler yang baik seharusnya dapat membantu programmer di dalam mengidentifikasi bentuk dan letak kesalahan.

Bentuk-bentuk kesalahan sintaktik yang mungkin timbul, antara lain adalah :

- kekuranglengkapan perintah (begin tanpa end, dll);
- kekuranglengkapan simbol (., ;, dll)
- kekuranglengkapan operasi aritmatik ((tanpa), dll)
- kesalahan penulisan assignment (:= ditulis =, dll)

Syntax Errors & Handling

Terdapat beberapa pendekatan untuk melakukan error-recovery dalam tahap analisis sintaksis ini, yaitu :

- **Panic Mode**
  Teknik termudah. Parser akan 'membuang' input symbol yang tidak dikenali, sampai ditemukan token yang sesuai (synchronizing token). synchronizing token in biasanya berupa semicolon atau end.

- **Phrase Level**
  Model ini berupaya untuk melakukan perbaikan minor pada rangkaian input yang diterima (local correction). Misalnya mengganti koma dengan semicolon, menghapus kelebihan titik, menambahkan titik dua pada tempat yang semestinya.

- **Error Production**
  mengembangkan grammar dengan menambahkan production-production yang dapat memanipulasi kesalahan. Dan mencangkokkan production tersebut pada proses parsing yang semestinya.

- **Global Correction**
  membuat sebuah algoritma yang dapat memunculkan semua alternatif string yang mungkin benar, dari sebuah string inputan yang salah. Dan kemudian memilih salah satu string alternatif yang paling sedikit perubahannya terhadap string inputan.

Parsing Methods

Secara umum terdapat 3 metode untuk parsing, yaitu :

1. **Cocke-Younger-Kasami Algorithm**
   metode ini diyakini dapat mem-parsing semua bentuk grammar. Tetapi algoritma ini sangat tidak efisien untuk digunakan dalam kompilator. Dan sayangnya, dari semua referensi yang kita punyai, tidak ada yang membahas lebih jauh tentang algoritma ini.

2. **Earley's Algorithm**
   sama persis dengan algoritma di atas.

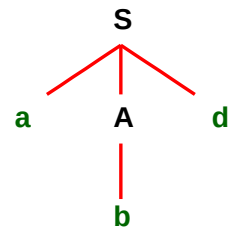
3. **Top-Down Parsing dan Bottom-Up Parsing**
   Top-Down Parsing akan membangun parse tree dari start symbol (sebagai *root*-nya), dan mengembangkannya menjadi rangkaian terminal (sebagai *leaves*-nya).
   Sedangkan Bottom-Up Parsing kebalikannya, berangkat dari fakta (rangkaian input string) dan mengembangkan parse tree untuk mendapatkan start symbol.

Brute-Force Approach

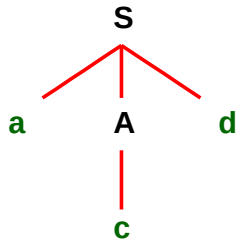
Langkah-langkah parsing *ala* Brute-Force adalah :

1. Parsing diawali dari start symbol;
2. Penggantian non-terminal diupayakan dari non-terminal terkiri;
3. Langkah kedua dapat diulangi jika rangkaian terminal yang terbentuk tidak sesuai. Sekali lagi, pilih non-terminal terkiri.

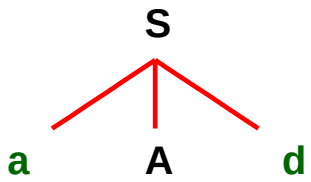
contoh :
Pelacakan string "**accd**" pada Grammar di bawah :
S ➔ a A d | a B
A ➔ b | c
B ➔ c c d | d d c



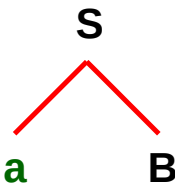
generated : **ab**
input : **ac | cd**



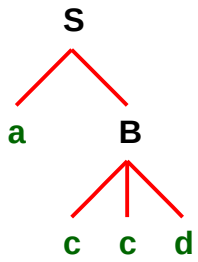
generated : **ac | acd**
input : **ac | cd | acc | d**



generated : **a**
input : **a | ccd**



generated : **a**
input : **a | ccd**

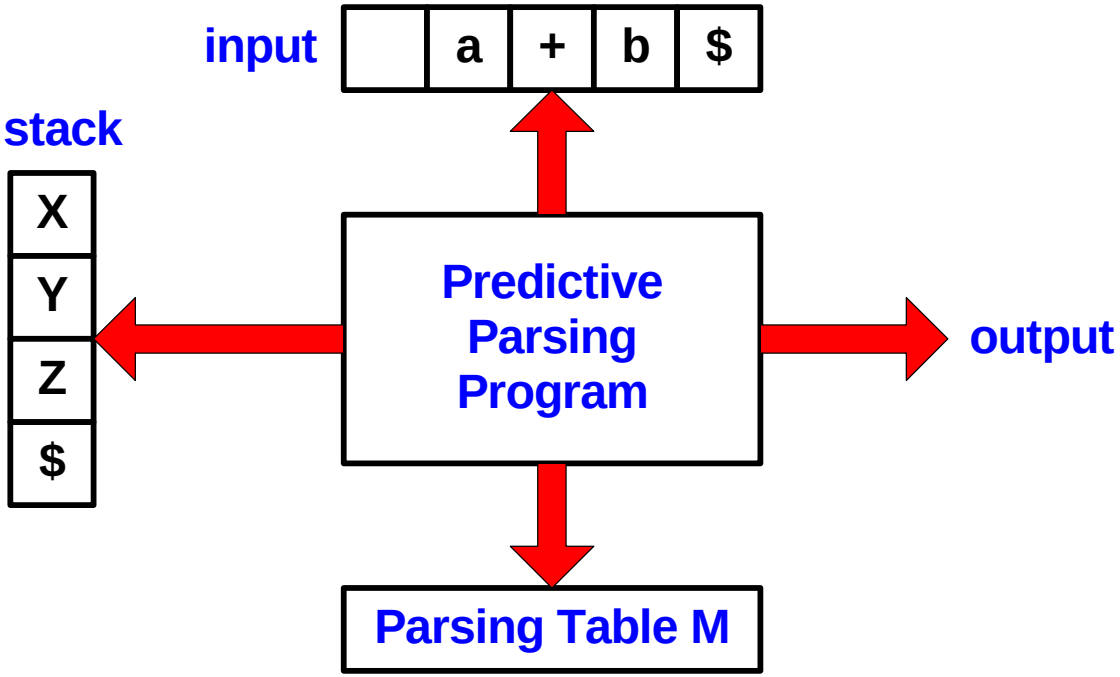


generated : **ac | acd | accd**
input : **ac | cd | acc | d | accd**

Recursive Descent (1)

Terdapat 2 metode untuk mengimplementasikan Recursive Descent Parser, yaitu :

1. Predictive Parsing
2. Non-Recursive Predictive Parsing



Recursive Descent (2)

1. **Input Buffer**
   Tempat untuk menampung rangkaian input yang akan diuraikan oleh parser. Selalu diakhiri simbol $ (untuk menyatakan akhir input);

2. **Stack**
   rangkaian simbol grammar (sesuai dengan uruta productionnya). Selalu diakhiri simbol $ (untuk menyatakan akhir input). Saat inisialisasi, stack hanya berisi start symbol dan simbol $.

3. **Parsing Table M**
   Sebuah array berdimensi 2 M[A, a]. Dimana A adalah non-terminal, dan a adalah terminal atau simbol $.

Recursive Descent (3)

Predictive Parsing Program

adalah sebuah program yang memiliki karakteristik sbb :

Jika X adalah simbol awal stack dan a adalah input terdepan, maka 3 kemungkinan yang ada adalah :

Jika X = a = $, maka parser akan berhenti dan proses parsing telah selesai;

Jika X = a ≠ $, parser mem-pop X dari stack dan memajukan pointer input ke simbol input berikutnya;

Jika X adalah non-terminal, program akan melihat entry M[X, a] dari tabel parsing M. entry ini dapat berupa production X atau suatu error entry.