Analisis Leksikal

TEKNIK KOMPILASI

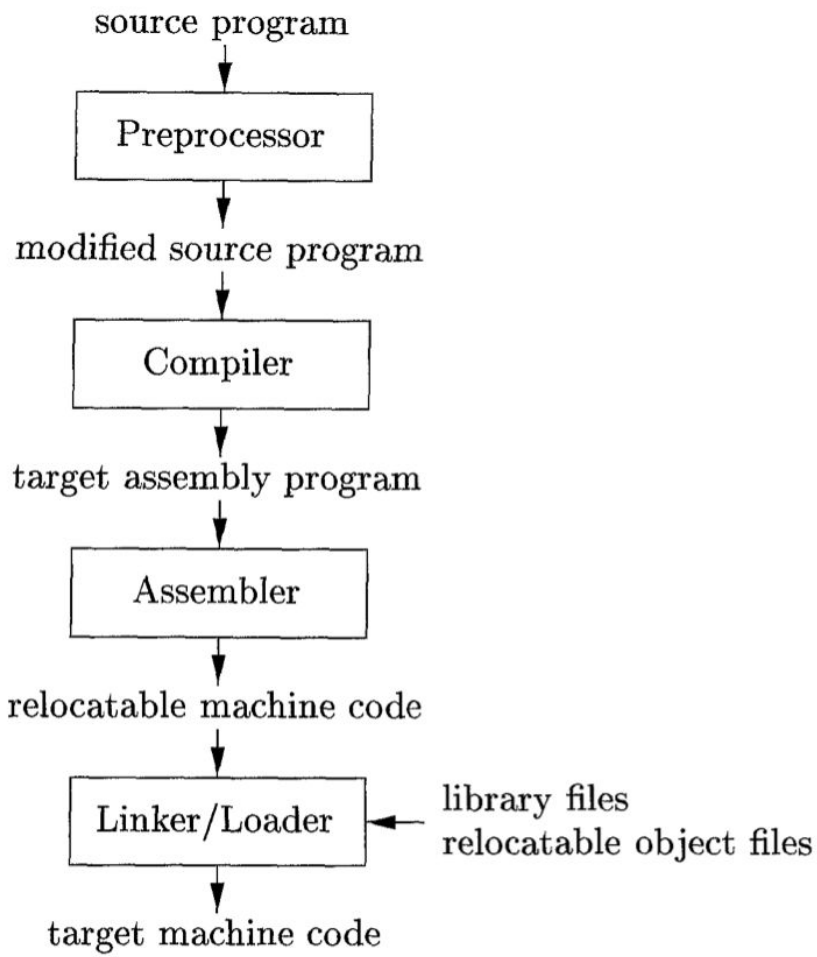
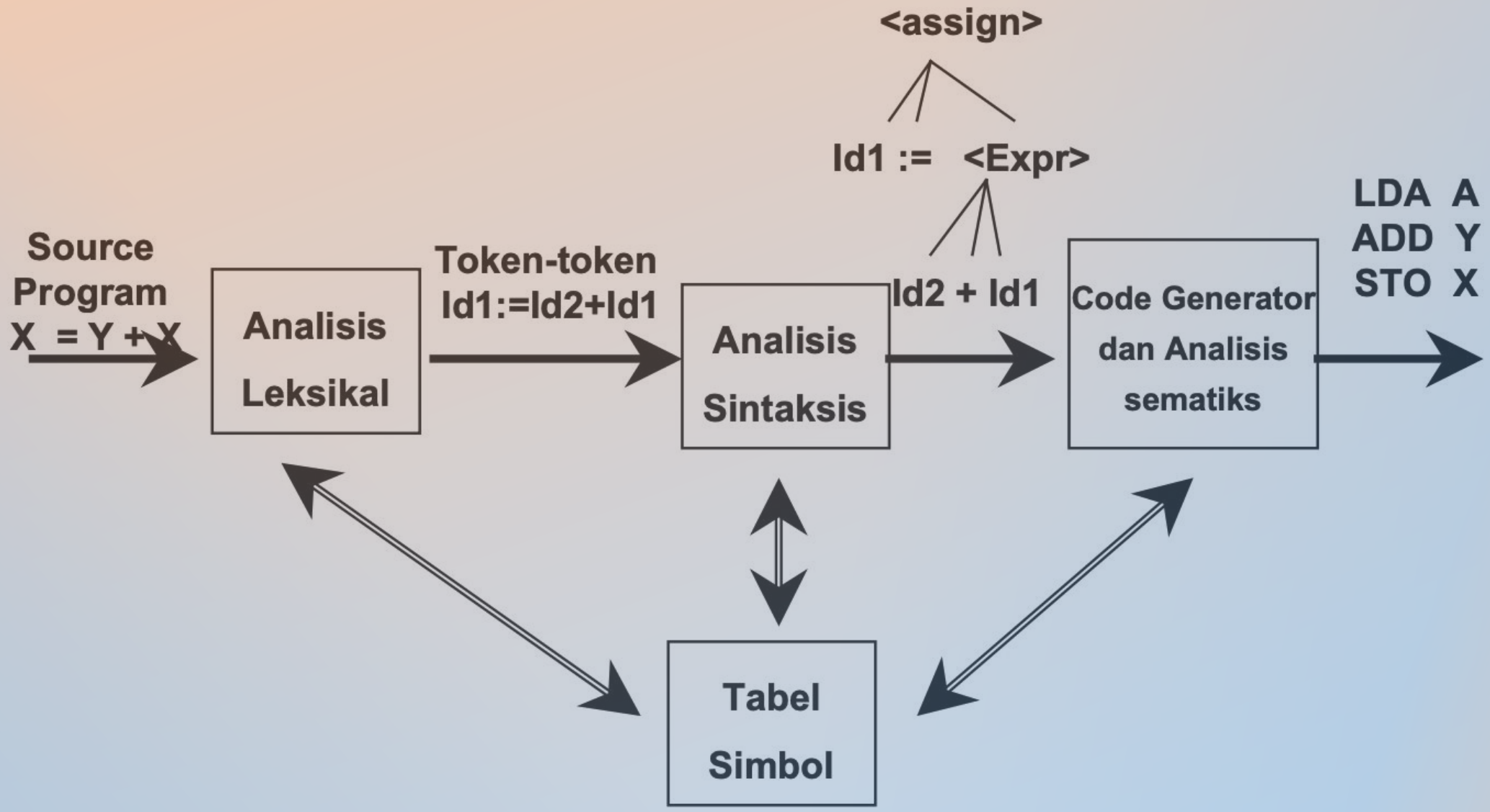
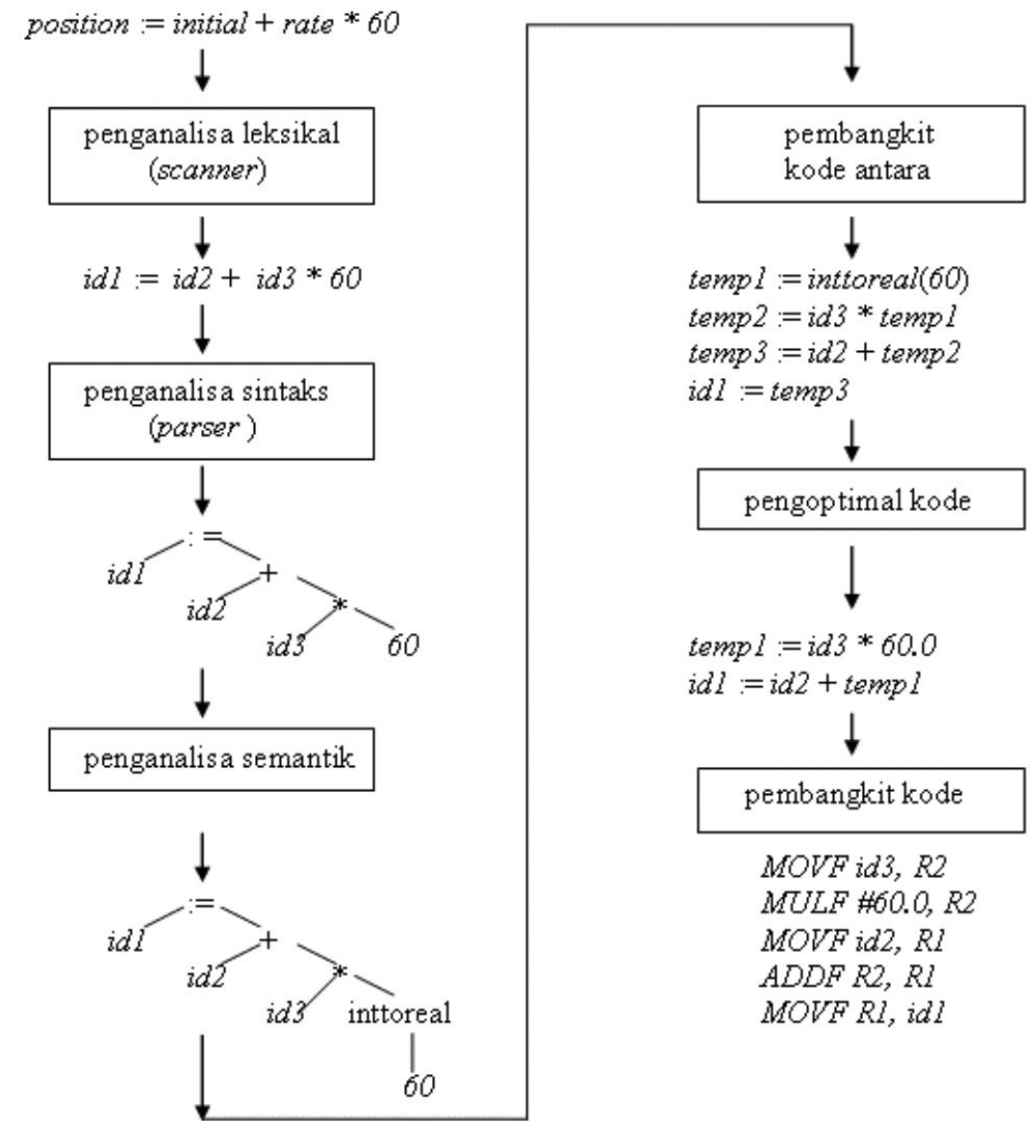


Figure 1.5: A language-processing system

- Perjalanan Sebuah Instruksi



- Contoh Teknik Kompilasi:



Peran Lexical Analyzer

a) Lexical analyzer adalah fase pertama kompilasi.

b) Tugas utama

1. Membaca karakter input dari program sumber.
2. Mengelompokkan karakter menjadi lexeme.
3. Menghasilkan token untuk setiap lexeme.
4. Mengirim aliran token ke parser untuk analisis sintaksis

c) Tugas tambahan

1. Menghapus komentar dan whitespace (spasi, newline, tab).
2. Menambahkan informasi nomor baris untuk membantu pelaporan error.
3. Kadang membuat salinan program sumber dengan pesan error disisipkan pada posisi yang sesuai.
4. Bisa melakukan ekspansi makro bila bahasa mendukung preprocessor

Interaksi dengan Symbol Table

- Saat menemukan **identifier**, lexical analyzer akan mendaftarkannya ke **symbol table**.
- Jika identifier sudah ada, informasi dapat dibaca dari symbol table untuk menentukan token yang tepa

Pemrosesan Berlapis

a) Lexical analysis sering dibagi menjadi dua tahap:

b) Scanning: operasi sederhana seperti menghapus komentar atau mereduksi whitespace berturut-turut menjadi satu.

c) Lexical analysis proper: bagian kompleks di mana token benar-benar dikenali dan dihasilkan

Menurut Aho dkk., analisis leksikal dipisahkan dari parsing karena

1. Kesederhanaan desain – parser tidak perlu repot menangani komentar dan whitespace.
2. Efisiensi kompilator – lexical analyzer bisa menggunakan teknik buffering khusus untuk membaca karakter dengan cepat.
3. Portabilitas – detail spesifik perangkat input bisa dibatasi di tahap lexical

Error Handling

a) Lexical analyzer juga berperan dalam **penanganan error**.
b) Jika ada karakter/urutan karakter yang tidak cocok dengan pola token mana pun, strategi umum:
c) Menghapus karakter ilegal.
d) Mengganti dengan karakter valid.
e) Menyisipkan karakter yang hilang.
f) Memberitahu kesalahan sambil tetap melanjutkan proses

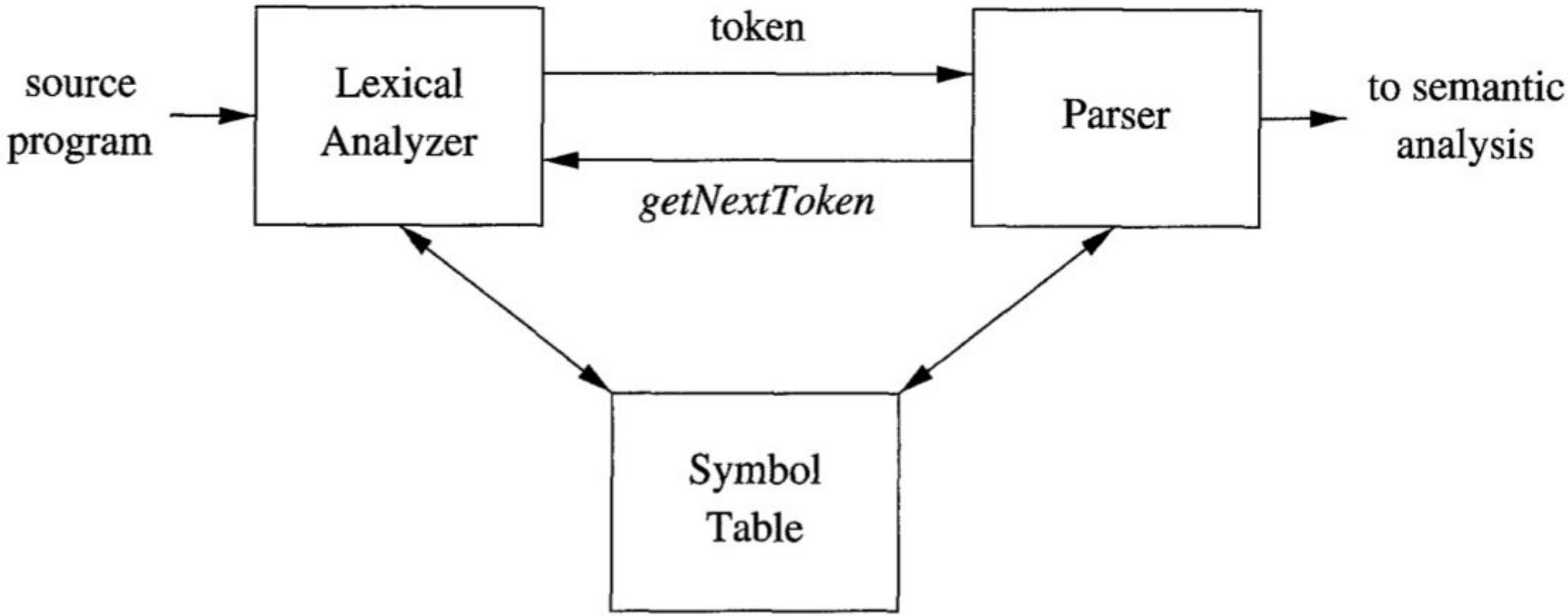


Figure 3.1: Interactions between the lexical analyzer and the parser

Konsep Dasar

- **Token**: pasangan nama token dan atribut opsional (misalnya keyword, identifier, number).
- **Pattern**: deskripsi bentuk lexeme yang sesuai dengan token (misalnya ekspresi reguler).
- **Lexeme**: urutan karakter aktual dalam program yang cocok dengan pola sebuah token .

Token

- **Definisi**: Pasangan yang terdiri dari **token name** dan **atribut opsional**.
- **Token name** → simbol abstrak yang mewakili jenis unit leksikal (misalnya *identifier*, *keyword*, *number*, *operator*).
- Parser menggunakan nama token ini sebagai **input symbol**.
- Contoh:
  - if → token (IF)
  - score → token (id, pointer_ke_symbol_table)

Pattern

a) Definisi: Deskripsi bentuk yang dapat diambil oleh lexeme suatu token.

b) Contoh:

c) Untuk keyword if → pattern adalah string literal "if".

d) Untuk *identifier* → pattern berupa *regular expression*, misalnya:

```
letter (letter | digit)*
```

Jadi **pattern** merupakan aturan formal (sering dengan *regular expressions*) untuk mengenali lexeme.

Lexeme

a) Definisi: Urutan karakter aktual dalam program sumber yang mencocokkan pola (pattern) suatu token.

b) Lexical analyzer mengidentifikasi lexeme sebagai instance dari token.

Contoh (kode C):

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

- `printf` dan `score` → lexeme dari token **id**.
- `"Total = %d\n"` → lexeme dari token **literal**.

Klasifikasi Token Umum

Menurut Aho dkk., banyak bahasa pemrograman memiliki kelas token berikut.

a) Keyword → if, else, while.

b) Operator → bisa individual (+, -, *) atau kelompok (misalnya token comparison untuk ==, !=, <, >=).

c) Identifier → nama variabel/fungsi.

d) Constant → angka, string literal.

e) Punctuation → tanda baca seperti (,), ,, ;.

Attributes for Tokens

a) Kadang token membutuhkan **atribut tambahan** agar compiler fase berikutnya bisa bekerja dengan benar.
   - ✔ Misalnya, token number bisa mencocokkan 0, 1, 3.14 → tapi perlu disimpan nilai aktualnya.
   - ✔ Token id biasanya membawa pointer ke **symbol table entry**, yang menyimpan informasi nama, tipe, dan lokasi identifier.

b) Jadi **token name** memengaruhi *parsing*, sedangkan **attribute value** memengaruhi *translation* (kode selanjutnya).

Token

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or !=	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Token

1. **One token for each keyword**
   a) Setiap *keyword* dalam bahasa pemrograman (misalnya if, while, for, return) punya token khusus.
   b) Polanya adalah sama persis dengan kata kuncinya (jadi regex-nya hanya kata itu sendiri).
2. **Tokens for the operators**
   a) Operator juga dianggap sebagai token.
   b) Bisa direpresentasikan satu per satu (contoh: +, -, *, /) atau dalam kelas yang lebih umum (misalnya *comparison operators* untuk ==, !=, <, >).
3. **One token representing all identifiers**
   a) Semua **identifier** (nama variabel, fungsi, kelas, dll.) digabungkan dalam satu jenis token saja, misalnya ID.
   b) Nanti atribut token ini biasanya berisi pointer ke **symbol table** untuk tahu detail nama identifier yang sebenarnya.
4. **One or more tokens representing constants**
   a) Konstanta seperti angka (123, 3.14) atau string literal ("Hello", 'A') dianggap sebagai token.
   b) Bisa dibuat lebih dari satu token jika ingin membedakan (misalnya INT_CONST, FLOAT_CONST, STRING_CONST).
5. **Tokens for each punctuation symbol**
   Simbol tanda baca yang dipakai dalam sintaks, seperti (,), ,, ;, {, }, juga dipetakan jadi token.

Intinya: lexical analyzer mengubah **stream karakter** program menjadi **token stream** dengan kategori: **keywords, operators, identifiers, constants, punctuation**.

Mengapa Token Butuh Atribut ?

a) Token name saja sering tidak cukup.

b) Misalnya: token number dapat merepresentasikan banyak lexeme berbeda (0, 1, 42, 3.14), tapi code generator butuh tahu nilai spesifik mana yang muncul.

c) Karena itu, lexical analyzer mengirim token name + attribute value:

d) Token name → digunakan parser untuk keputusan sintaksis.

e) Attribute value → digunakan fase selanjutnya untuk translasi/semantik.

Bentuk Token

- **token-name**: simbol abstrak (misalnya id, number, if).
- **attribute-value**: informasi tambahan yang spesifik untuk lexeme tersebut.

```
(token-name, attribute-value)
```

Contoh Bentuk Token

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

Fortran statement akan diubah menjadi pasangan token + atribut berikut.

```
<id, pointer ke symbol table entry untuk E>
<assign-op>
<id, pointer ke symbol table entry untuk M>
<mult-op>
<id, pointer ke symbol table entry untuk C>
<exp-op>
<number, integer value 2>
```

Aho - Compilers - Principles, T...

Struktur Atribut

- **Satu atribut per token**, tetapi atribut itu bisa berbentuk **struktur data** yang menyimpan beberapa informasi sekaligus (misalnya nama, tipe, scope).
- **Token name** → dipakai parser (struktur sintaks).
- **Attribute value** → dipakai semantic analyzer & code generator (detail spesifik lexeme).
- Hubungan erat dengan **symbol table**, karena hampir semua identifier membawa pointer ke tabel simbol.

Buffering dan Efisiensi

- Lexical analyzer sering perlu **membaca maju (lookahead)** untuk menentukan batas lexeme (mis. membedakan = vs ==, atau membentuk identifier sampai karakter non-letter/digit). Mengambil satu karakter-per-system-call sangat mahal; membaca blok besar sekaligus jauh lebih efisien.

Skema *double-buffer* (Buffer Pairs) - Struktur & Invariants

1. Dua buffer ukuran sama NNN (biasanya ukuran blok disk, contoh 4096 byte). Saat satu buffer sedang diproses, buffer lain dapat di-reload dari file input dengan satu system read — mengurangi overhead I/O. Aho - Compilers - Principles, T…
2. Dua pointer utama: lexemeBegin (tanda awal lexeme yang sedang diuji) dan forward (pengeksplorasi maju untuk menemukan akhir lexeme). Setelah token ditentukan, forward menunjuk ke karakter setelah akhir lexeme dan kemudian lexemeBegin diset ke posisi itu. Prinsip: selama (panjang lexeme) + (jarak lookahead) ≤ N, kita tidak akan menimpa lexeme yang sedang dibentuk.

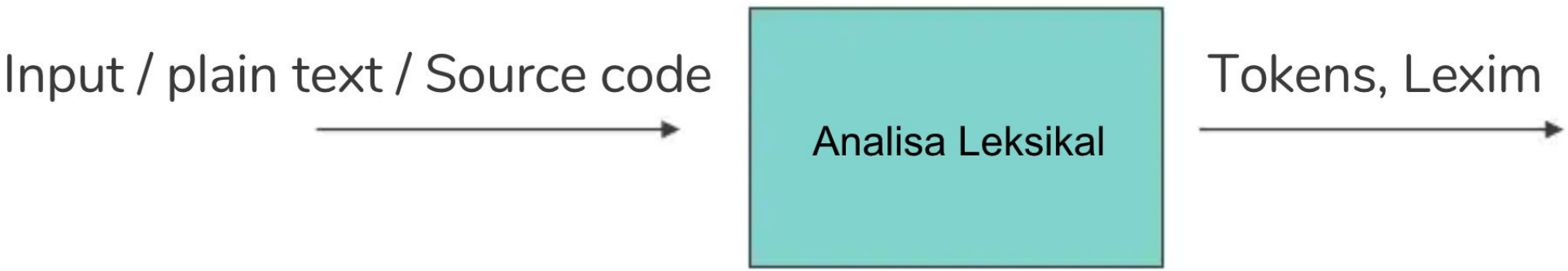
Optimasi *Sentinel* — Mengurangi Tes per Karakter

1. Tanpa sentinel, setiap kali forward maju kita harus mengecek: (a) apakah kita sampai akhir buffer? dan (b) apa karakter sekarang? — jadi dua tes per karakter. Dengan **sentinel** kita meletakkan sebuah karakter khusus (eof) di akhir tiap buffer (karakter ini tidak mungkin muncul dalam source). Maka ketika forward menunjuk ke sentinel kita tahu buffer habis; bila bukan sentinel, langsung proses karakter itu. Dengan demikian mayoritas langkah membaca hanya memerlukan **satu tes** (uji karakter), dan penanganan akhir buffer hanya terjadi jarang. Aho - Compilers - Principles, T…
2. Buku menampilkan algoritma ringkas (pseudocode switch) yang menunjukkan logika *forward++ dengan kasus eof untuk reload buffer/akhiri analisis; untuk kasus lain langsung masuk multiway branch (jump table) untuk menangani karakter. (Ilustrasi see Fig. 3.5). Aho - Compilers - Principles, T…

Manfaat efisiensi secara praktis

1. Mengurangi jumlah system calls: baca blok N byte sekali (daripada byte-per-byte), signifikan mempercepat pemrosesan file besar. Aho - Compilers - Principles, T…
2. Mengurangi pengecekan batas buffer pada tiap karakter (karena sentinel), sehingga loop scanning lebih cepat. Aho - Compilers - Principles, T…
3. Multiway branch via jump table: implementasi switch/case untuk karakter input biasanya di-implementasikan sebagai jump table sehingga pemilihan tindakan berdasarkan karakter menjadi O(1). Aho - Compilers - Principles, T…

Analisa Leksikal?



Proses Analisa Leksikal

Secara sederhana, prosesnya seperti ini

Source code -> pemisahan huruf -> leksim -> Validasi patern -> token

Leksim adalah sebarisan karakter program sumber yang sesuai dengan pola suatu token. Token adalah himpunan karakter yang mempunyai arti khusus. Pola Pattern adalah aturan pembentukan suatu token

Proses Analisa Leksikal

Source code

```
/* Program penjumlahan */

int a = b + 7;
cout << a;
```

Proses Analisa Leksikal

Pemisahan huruf

Komentar dihilangkan, new line kosong dihilangkan

```
int a = b + 7;cout << a;
```

Proses Analisa Leksikal

Pemisahan huruf menjadi leksim

i	n	t		a		=		b		+		7		;	c	o	u	t	

<	<		a	;

Proses Analisa Leksikal

Leksim

i	n	t		a		=		b		+		7		;	c	o	u	t	

<	<		a	;

Pattern

int float char num ...	cout cin ...	assign_op op id ...

Proses Analisa Leksikal

Token

int	a	=	b	+	7	;	cout <<	a	;
int	id	assign_op	id	op	num	special	cout	id	special

<int><id, a><assign_op, =><id, b><op, +><num,7><special, ;><cout><id, a><special, ;>

Proses Analisa Leksikal

Token dapat berupa

Identifier: a,b,c, dan lain-lain

Keyword: cout, cin, printf, echo, print, dan lain-lain

Operators: aritmatika, logical, comparison, dan lain-lain

Special Char: {,},(,),/, dan lain-lain

Conts/Num: 1,2,3,4,5,6,7,8,9,0

Peranan Analisa Leksikal

1. membaca karakter input dan menghasilkan output berupa token.Token akan akan dipakai oleh pengurai parser sebagai input untuk analisa sintak
2. membuang komentar, spasi,tab, newline dan karakter lain yang ‘tak berguna’,
3. menghubungkan pesan kesalahan kompilator dengan program sumbernya. Contoh : baris dari program

Peranan Analisa Leksikal

Membaca karakter input dan menghasilkan output berupa token.Token akan dipakai oleh pengurai parser sebagai input untuk analisa sintak

Peranan Analisa Leksikal

Membuang komentar, spasi,tab, newline dan karakter lain yang ‘tak berguna’,

Peranan Analisa Leksikal

Menghubungkan pesan kesalahan kompilator dengan program sumbernya. Contoh : baris dari program

Latihan 1 - Identifikasi Token Dasar (Manual)

Tujuan: Mengenali token, lexeme, dan pattern secara manual.

```
if (count <= 10) then sum = sum + count;
```

1. Berdasarkan sintaks di atas, buatlah tabel berisi:
    - Token name (mis. `IF`, `ID`, `RELOP`, `NUMBER`, `THEN`, `ID`, `ASSIGN`, `ID`, `PLUS`, `ID`).
    - Pattern (mis. `if`, `[a-zA-Z][a-zA-Z0-9]*`, `<=`).
    - Lexeme aktual (mis. `if`, `count`, `<=`, `10`, …).
2. Diskusikan peran atribut token (mis. `count` → pointer ke symbol tae) !

Latihan 2 - Regex untuk Token

Tujuan: Menuliskan ekspresi reguler untuk tiap kelas token.

Instruksi:

- Buat regex untuk:
  1. Identifier (huruf diikuti huruf/angka/underscore).
  2. Angka bulat & angka dengan desimal.
  3. Operator relasi (`<`, `<=`, `>`, `>=`, `==`, `!=`).
  4. String literal `" . . ."` (boleh berisi escape).

- Uji regex ini dengan beberapa contoh input (bisa pakai Python/grep untuk cek regex).

Latihan 3 — Buffering & Lookahead

Tujuan: Memahami bagaimana lexer membaca input dengan efisien.

Instruksi:

1. Jelaskan bagaimana lexer membedakan `=` dan `==` dengan 1 karakter lookahead.
2. Simulasikan pointer `lexemeBegin` dan `forward` pada input `a==b`. Tunjukkan posisi pointer pada setiap langkah !
3. Diskusikan mengapa *sentinel* (karakter EOF di akhir buffer) penting.