

# Bab 4

## Lexer Generator (Flex/re2c) dan Praktikum Lexer

### 4.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep dan keuntungan menggunakan lexer generator
2. Menggunakan Flex untuk membuat specification file (.l) dan generate lexer code
3. Menggunakan re2c untuk membuat lexer dengan embedded specification
4. Membuat specification file untuk lexer bahasa sederhana
5. Mengintegrasikan generated lexer dengan program utama
6. Membandingkan hand-written lexer dengan generator-based lexer
7. Mengevaluasi trade-off antara performa, kemudahan maintenance, dan fleksibilitas

### 4.2 Pendahuluan

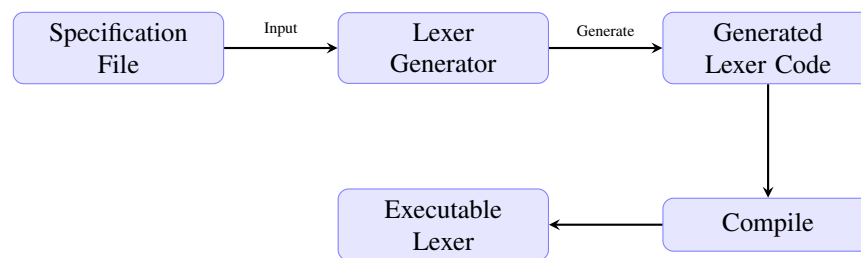
Pada bab ini kita menambah **lexer proyek compiler subset C** menggunakan Flex. Spesifikasi token proyek telah didefinisikan di Bab 1 (Bagian ??); file `simplec.l` yang dibangun di bab ini (dan dipakai bersama parser di Bab 8) mengimplementasikan token set tersebut. Contoh lengkap `simplec.l` terintegrasi dengan parser `simplec.y` di Bab 8; token yang dihasilkan mengikuti spesifikasi Bab 1. Kode hasil generate menjadi bagian dari codebase proyek di folder `proyek-compiler-subset-c/`.

Pada bab sebelumnya, kita telah mempelajari implementasi hand-written lexer. Meskipun pendekatan tersebut memberikan kontrol penuh dan pemahaman mendalam, dalam praktik industri, penggunaan **lexer generator** lebih umum karena efisiensi dan kemudahan maintenance. Menurut sumber terbuka:

“re2c is a high-performance lexer generator for C/C++ that takes regex specifications and builds deterministic finite automata. It’s used in real projects.”[1]

Lexer generator adalah tools yang menerima specification file (berisi pattern dan action) dan menghasilkan kode lexer yang siap digunakan. Dua generator populer untuk C/C++ adalah **Flex** (Fast Lexical Analyzer) dan **re2c** (Regular Expressions to Code).

Gambar 4.1 menunjukkan alur kerja lexer generator secara umum.



Gambar 4.1: Alur kerja lexer generator

Keuntungan menggunakan lexer generator:

- **Produktivitas:** Lebih cepat dalam development karena tidak perlu menulis state machine manual
- **Maintainability:** Specification file lebih mudah dibaca dan dimodifikasi dibanding kode state machine
- **Optimasi Otomatis:** Generator menghasilkan kode yang sudah dioptimasi (DFA minimization, dll.)
- **Konsistensi:** Mengurangi bug karena generator sudah teruji

### 4.3 Flex (Fast Lexical Analyzer)

Flex adalah lexer generator yang paling banyak digunakan, terutama dalam kombinasi dengan Bison (parser generator). Flex membaca specification file dengan ekstensi `.l` dan menghasilkan kode C untuk lexer.

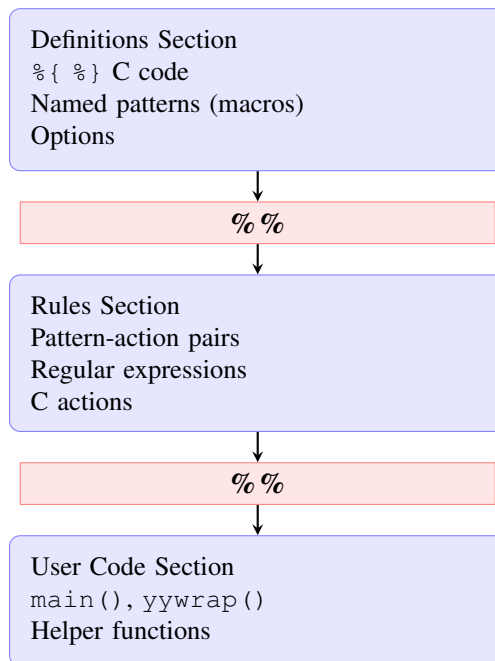
#### 4.3.1 Struktur Flex Specification File

File specification Flex (`.l`) terdiri dari tiga bagian yang dipisahkan oleh `%%`:

```
Definitions
%%
```

Rules  
%%  
User Code

Gambar 4.2 menunjukkan struktur file specification Flex secara visual.



Gambar 4.2: Struktur file specification Flex

### Definitions Section

Bagian ini berisi:

- **Named patterns (macros):** Definisi pattern yang dapat digunakan kembali
- **C code:** Kode C yang akan disalin langsung ke output (dalam ‘%{ %}’)
- **Options:** Konfigurasi Flex (misalnya ‘%option noyywrap’)

Contoh:

```

%{
#include <stdio.h>
#include "tokens.h" // Definisi token constants
%}

DIGIT    [0-9]
LETTER   [a-zA-Z]
ID       {LETTER} ({LETTER} | {DIGIT}) *
NUMBER   {DIGIT}+
  
```

## Rules Section

Bagian ini berisi pattern-action pairs. Pattern menggunakan regular expression, dan action adalah kode C yang dieksekusi ketika pattern match.

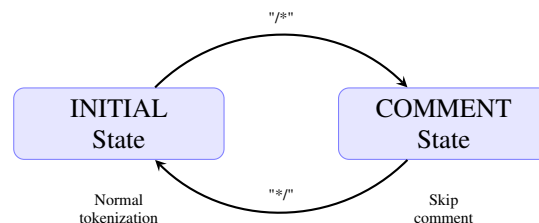
Contoh:

```

1 %%
2 "if"      { return IF; }
3 "else"    { return ELSE; }
4 "while"   { return WHILE; }
5 {ID}      { return IDENTIFIER; }
6 {NUMBER}  { yylval.intval = atoi(yytext); return NUMBER; }
7 "=="      { return EQ; }
8 "!="      { return NE; }
9 [ \t\n]+  { /* skip whitespace */ }
10 "//".*    { /* skip single-line comment */ }
11 "/*"      { BEGIN(COMMENT); }
12 <COMMENT>"*/" { BEGIN(INITIAL); }
13 <COMMENT>. { /* skip comment content */ }
14 .         { return yytext[0]; } /* default: return character */
15 %%

```

Gambar 4.3 menunjukkan penggunaan start conditions dalam Flex untuk menangani komentar multi-line.



Gambar 4.3: Start conditions dalam Flex untuk handling komentar

## User Code Section

Bagian ini berisi fungsi-fungsi pendukung seperti 'main()', 'yywrap()', dan helper functions.

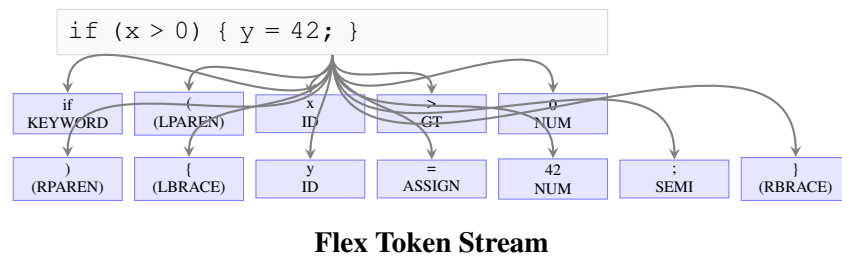
Gambar 4.4 menunjukkan contoh lengkap penggunaan Flex dari input hingga output token, mengikuti format yang konsisten dengan diagram tokenization di bab sebelumnya.

Gambar 4.5 menunjukkan proses pattern matching dalam Flex untuk keyword "if".

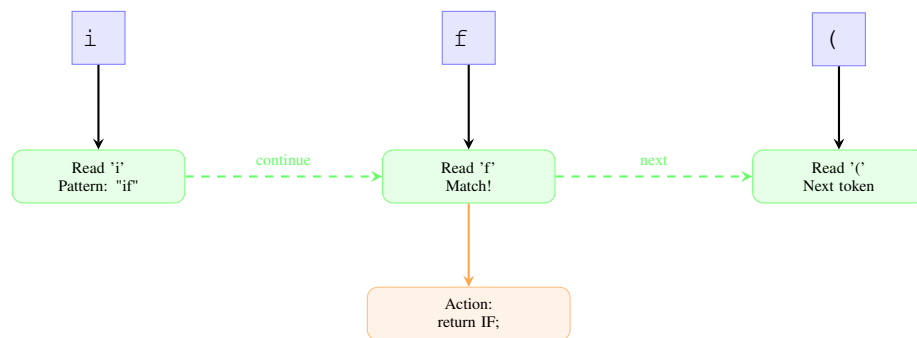
Gambar 4.6 menunjukkan berbagai jenis token yang dapat dikenali oleh Flex.

### 4.3.2 Contoh Lengkap: Flex Lexer untuk Bahasa Sederhana

Berikut adalah contoh specification file Flex untuk bahasa sederhana dengan token: identifier, number, keyword, dan operator:

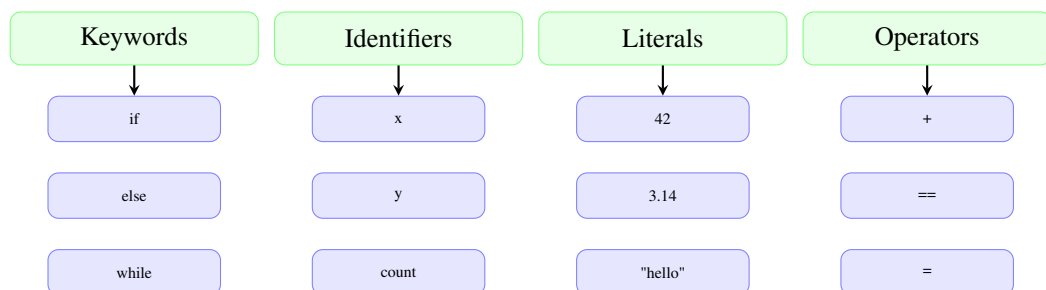


Gambar 4.4: Contoh lengkap tokenization dengan Flex: `if (x > 0) { y = 42; }`



Flex mencocokkan pola "if" dan mengeksekusi aksi leksikal

Gambar 4.5: Proses pattern matching dalam Flex: pencocokan keyword "if"



Gambar 4.6: Jenis-jenis token yang dikenali Flex

Listing 4.1: Contoh Flex specification file (calc.l)

```

1 %{
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "parser.tab.h" // Header dari Bison
5
6 int yylineno = 1;
7 %}
8
9 %option noyywrap
10 %option yylineno
11
12 DIGIT      [0-9]
13 LETTER     [a-zA-Z_]
14 ID         {LETTER} ({LETTER} | {DIGIT}) *
15 NUMBER     {DIGIT}+
16 FLOAT      {DIGIT}+\. {DIGIT}+
17
18 %%
19
20 "int"      { return INT; }
21 "float"    { return FLOAT_TYPE; }
22 "if"       { return IF; }
23 "else"     { return ELSE; }
24 "while"    { return WHILE; }
25 "return"   { return RETURN; }
26
27 {ID}       {
28             yylval.string = strdup(yytext);
29             return IDENTIFIER;
30         }
31
32 {NUMBER}   {
33             yylval.intval = atoi(yytext);
34             return NUMBER;
35         }
36
37 {FLOAT}    {
38             yylval.floatval = atof(yytext);
39             return FLOAT_LITERAL;
40         }
41
42 "+"        { return PLUS; }
43 "-"        { return MINUS; }
44 "*"        { return MULTIPLY; }
45 "/"        { return DIVIDE; }
46 "="        { return ASSIGN; }
47 "=="       { return EQ; }
48 "!="       { return NE; }
49 "<"        { return LT; }
50 ">"        { return GT; }
51 "<="       { return LE; }
52 ">="       { return GE; }
53

```

```

54 "("          { return LPAREN; }
55 ")"          { return RPAREN; }
56 "{"          { return LBRACE; }
57 "}"          { return RBRACE; }
58 ";"          { return SEMICOLON; }
59 ",",         { return COMMA; }
60
61 [ \t ]+      { /* skip whitespace */ }
62 \n           { yylineno++; }
63 "//" .*      { /* skip single-line comment */ }
64 "/*"         {
65             int c;
66             while ((c = input()) != EOF) {
67                 if (c == '\n') yylineno++;
68                 if (c == '*' && (c = input()) == '/') break;
69                 if (c != EOF) unput(c);
70             }
71         }
72
73 .           {
74             fprintf(stderr, "Error: unexpected character '%c' at line
↪ %d\n",
75             yytext[0], yylineno);
76             return ERROR;
77         }
78
79 %%
80
81 int yywrap(void) {
82     return 1;
83 }

```

Gambar 4.7 menunjukkan workflow kompilasi dan penggunaan Flex dan re2c secara perbandingan.

Gambar 4.8 menunjukkan bagaimana Flex menangani rule priority dan longest match.

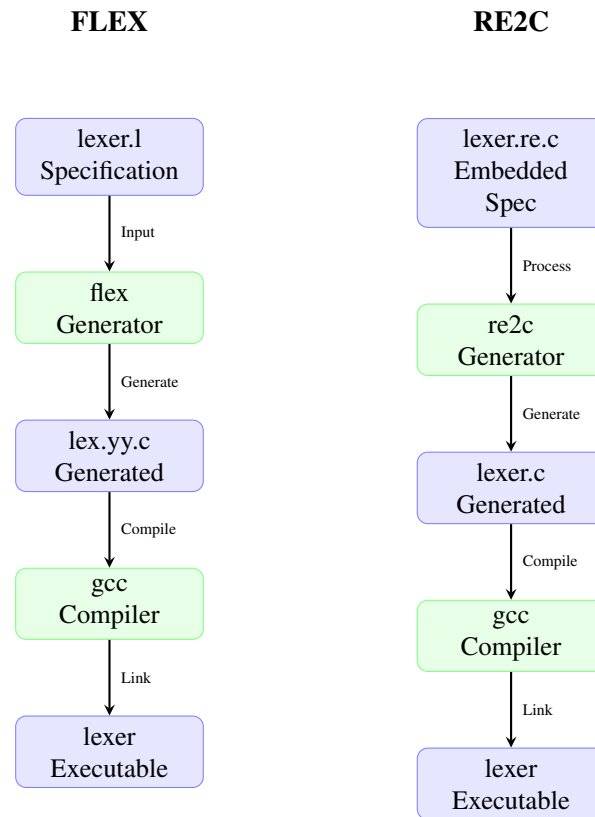
### 4.3.3 Kompilasi dan Penggunaan Flex

Untuk menggunakan Flex:

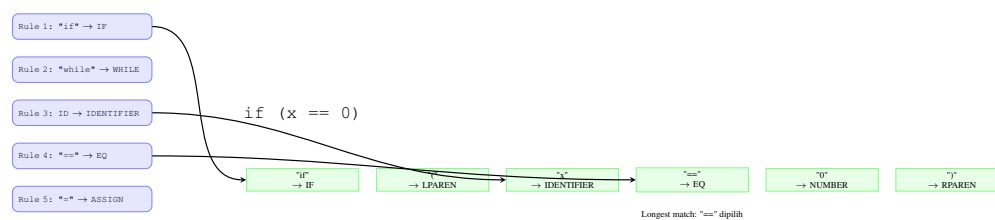
1. Buat file specification (misalnya `lexer.l`)
2. Generate lexer code: `flex lexer.l` (menghasilkan `lex.yy.c`)
3. Compile dengan compiler C: `gcc lex.yy.c -o lexer -lfl`
4. Atau link dengan program utama: `gcc main.c lex.yy.c -o program -lfl`

Fungsi utama yang digunakan:

- `yylex()`: Fungsi yang dipanggil untuk mendapatkan token berikutnya



Gambar 4.7: Workflow kompilasi dan penggunaan Flex dan re2c

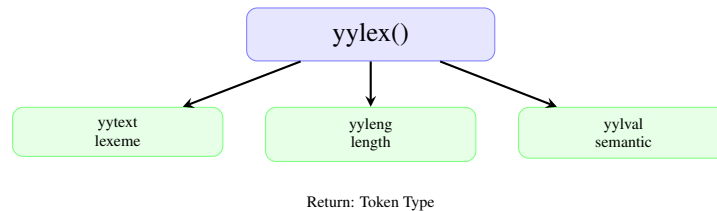


Gambar 4.8: Rule priority dan longest match dalam Flex



- `ytext`: String yang berisi lexeme yang baru saja di-match
- `yleng`: Panjang dari `ytext`
- `yylval`: Union untuk menyimpan nilai semantic (untuk integrasi dengan parser)

Gambar 4.9 menunjukkan penggunaan fungsi-fungsi utama Flex.



Gambar 4.9: Fungsi dan variabel utama dalam Flex

Perbedaan utama antara Flex dan re2c adalah Flex menggunakan **separate file** (`lexer.l`) sedangkan re2c menggunakan **embedded specification** dalam kode C/C++ (`lexer.re.c`). Detail workflow kompilasi keduanya dapat dilihat pada Gambar 4.7.

## 4.4 re2c (Regular Expressions to Code)

re2c adalah lexer generator modern yang menghasilkan kode C/C++ dengan performa tinggi. Berbeda dengan Flex yang menggunakan file terpisah, re2c menggunakan **embedded specification** dalam kode C/C++.

Menurut dokumentasi resmi re2c:

“re2c is a tool that generates fast lexers for C, C++ and Go. It compiles regular expressions to deterministic finite automata and encodes them as conditional jumps and comparisons. The generated code is highly optimized and does not use tables.”<sup>1</sup>

### 4.4.1 Struktur re2c Specification

re2c specification ditulis dalam komentar khusus `/*!re2c ... */` yang disisipkan dalam kode C/C++:

Listing 4.2: Struktur dasar re2c

```

1 #include <stdio.h>
2
3 static int lex(const char *YYCURSOR) {
4     const char *YYMARKER;
  
```

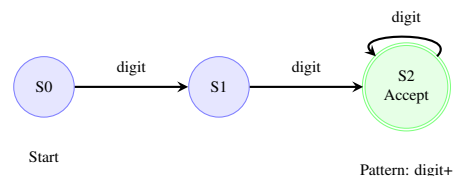
<sup>1</sup><https://re2c.org/>

```

5      /*!re2c
6          re2c:define:YYCTYPE = "char";
7          re2c:yyfill:enable = 0;
8
9          // Named patterns
10         digit    = [0-9];
11         letter   = [a-zA-Z_];
12         id       = letter (letter | digit)*;
13         number   = digit+;
14
15         // Rules
16         *        { return 0; } // error
17         number   { printf("Number: %.*s\n", (int)(YYCURSOR - YYMARKER),
18 → YYMARKER); return 1; }
19         id       { printf("ID: %.*s\n", (int)(YYCURSOR - YYMARKER),
20 → YYMARKER); return 1; }
21         [ \t\n]+ { continue; } // skip whitespace
22     */
23
24 int main(int argc, char *argv[]) {
25     for (int i = 1; i < argc; i++) {
26         lex(argv[i]);
27     }
28     return 0;
29 }

```

Gambar 4.10 menunjukkan bagaimana re2c menghasilkan state machine untuk pattern matching.



Gambar 4.10: State machine yang dihasilkan re2c untuk pattern `digit+`

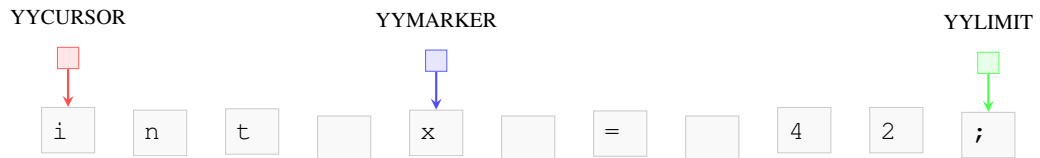
## 4.4.2 Key Concepts dalam re2c

Gambar 4.11 menunjukkan penggunaan variabel khusus dalam re2c.

### Variables Khusus

re2c menggunakan variabel khusus untuk tracking posisi input:

- `YYCURSOR`: Pointer ke posisi saat ini dalam input
- `YYMARKER`: Pointer untuk backtracking

Gambar 4.11: Ilustrasi pointer internal `re2c` dalam buffer input

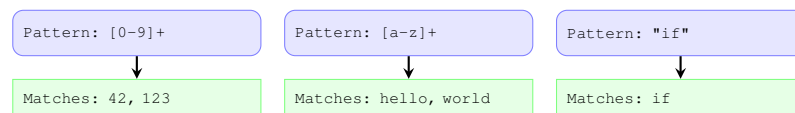
- `YYLIMIT`: Pointer ke akhir buffer
- `YYCTYPE`: Tipe data untuk karakter (default: `unsigned char`)

## Directives

Directives mengkonfigurasi behavior `re2c`:

- `re2c:define:YYCTYPE`: Mendefinisikan tipe karakter
- `re2c:yyfill:enable`: Enable/disable buffer filling
- `re2c:input`: Mendefinisikan cara membaca input
- `re2c:conditions`: Enable start conditions (seperti Flex)

Gambar 4.12 menunjukkan berbagai pattern yang dapat digunakan dalam `re2c`.

Gambar 4.12: Contoh pattern dalam `re2c`

### 4.4.3 Contoh Lengkap: `re2c` Lexer untuk Identifier dan Number

Listing 4.3: Contoh `re2c` lexer (`lexer.re.c`)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 typedef enum {
6     TOKEN_EOF,
7     TOKEN_IDENTIFIER,
8     TOKEN_NUMBER,
9     TOKEN_PLUS,
10    TOKEN_MINUS,
11    TOKEN_MULTIPLY,
12    TOKEN_DIVIDE,

```

```
13     TOKEN_ERROR
14 } TokenType;
15
16 typedef struct {
17     TokenType type;
18     char *value;
19     int line;
20 } Token;
21
22 static Token tokenize(const char *input, int *line) {
23     const char *YYCURSOR = input;
24     const char *YYMARKER;
25     const char *start;
26     Token token = {TOKEN_EOF, NULL, *line};
27
28     /*!re2c
29         re2c:define:YYCTYPE = "char";
30         re2c:yyfill:enable = 0;
31         re2c:define:YYCURSOR = "YYCURSOR";
32
33         digit    = [0-9];
34         letter   = [a-zA-Z_];
35         id       = letter (letter | digit)*;
36         number   = digit+;
37         ws       = [ \t]+;
38         newline  = "\n";
39
40         * {
41             token.type = TOKEN_ERROR;
42             return token;
43         }
44
45         "\x00" {
46             token.type = TOKEN_EOF;
47             return token;
48         }
49
50         ws {
51             continue;
52         }
53
54         newline {
55             (*line)++;
56             continue;
57         }
58
59         number {
60             start = YYMARKER;
61             int len = YYCURSOR - start;
62             token.value = (char*)malloc(len + 1);
63             strncpy(token.value, start, len);
64             token.value[len] = '\0';
65             token.type = TOKEN_NUMBER;
66             token.line = *line;
```

```

67         return token;
68     }
69
70     id {
71         start = YYMARKER;
72         int len = YYCURSOR - start;
73         token.value = (char*)malloc(len + 1);
74         strncpy(token.value, start, len);
75         token.value[len] = '\0';
76         token.type = TOKEN_IDENTIFIER;
77         token.line = *line;
78         return token;
79     }
80
81     "+" {
82         token.type = TOKEN_PLUS;
83         token.line = *line;
84         return token;
85     }
86
87     "-" {
88         token.type = TOKEN_MINUS;
89         token.line = *line;
90         return token;
91     }
92
93     "*" {
94         token.type = TOKEN_MULTIPLY;
95         token.line = *line;
96         return token;
97     }
98
99     "/" {
100         token.type = TOKEN_DIVIDE;
101         token.line = *line;
102         return token;
103     }
104     */
105 }
106
107 int main(int argc, char *argv[]) {
108     if (argc < 2) {
109         fprintf(stderr, "Usage: %s <input>\n", argv[0]);
110         return 1;
111     }
112
113     int line = 1;
114     Token token;
115
116     do {
117         token = tokenize(argv[1], &line);
118         printf("Token: %d, Value: %s, Line: %d\n",
119             token.type, token.value ? token.value : "NULL", token.line
120         );

```

```

120     if (token.value) free(token.value);
121 } while (token.type != TOKEN_EOF && token.type != TOKEN_ERROR);
122
123 return 0;
124 }

```

Untuk mengkompilasi:

```

re2c -o lexer.c lexer.re.c
gcc lexer.c -o lexer

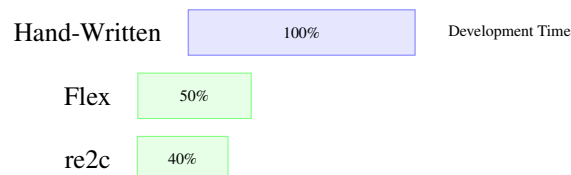
```

Gambar 4.13 menunjukkan perbandingan visual antara hand-written, Flex, dan re2c.

Aspek	Hand-Written	Flex	re2c
Produktivitas	Rendah	Tinggi	Tinggi
Maintainability	Sedang	Tinggi	Tinggi
Performa	Tinggi	Sedang	Sangat Tinggi
Fleksibilitas	Sangat Tinggi	Sedang	Sedang

Gambar 4.13: Perbandingan hand-written, Flex, dan re2c

Gambar 4.14 menunjukkan perbandingan waktu development antara berbagai pendekatan.



Gambar 4.14: Perbandingan waktu development (hand-written = baseline)

## 4.5 Perbandingan Hand-Written vs Generator-Based Lexer

Setelah mempelajari kedua pendekatan, mari kita bandingkan:

### 4.5.1 Hand-Written Lexer

**Keuntungan:**

- Kontrol penuh terhadap implementasi
- Tidak ada dependency eksternal

- Dapat dioptimasi secara spesifik untuk kebutuhan
- Pemahaman mendalam tentang proses tokenization

**Kekurangan:**

- Lebih banyak kode yang harus ditulis dan maintain
- Lebih mudah terjadi bug (edge cases)
- Perlu implementasi ulang untuk setiap bahasa
- Lebih sulit untuk modifikasi pattern

### 4.5.2 Generator-Based Lexer

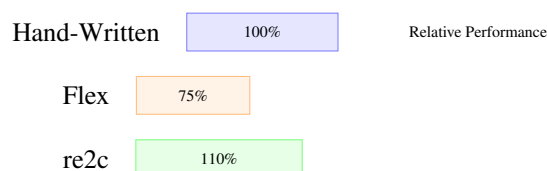
**Keuntungan:**

- Specification lebih ringkas dan mudah dibaca
- Generator menghasilkan kode yang sudah teroptimasi
- Lebih cepat dalam development
- Pattern mudah dimodifikasi tanpa mengubah banyak kode
- Sudah teruji dan digunakan di banyak project

**Kekurangan:**

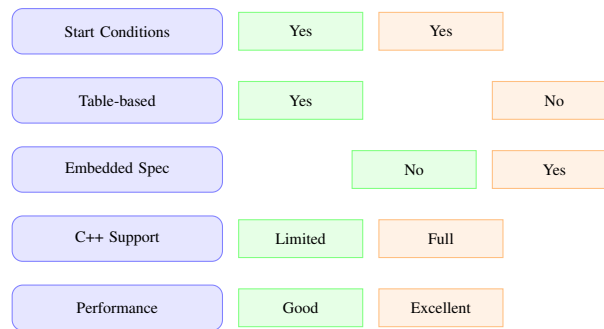
- Perlu mempelajari syntax generator
- Dependency pada tool eksternal
- Kurang fleksibel untuk kasus yang sangat spesifik
- Generated code mungkin lebih sulit di-debug

Gambar 4.15 menunjukkan perbandingan performa secara visual.



Gambar 4.15: Perbandingan performa relatif (hand-written = baseline)

Gambar 4.16 menunjukkan perbandingan fitur antara Flex dan re2c.



Gambar 4.16: Perbandingan fitur Flex vs re2c

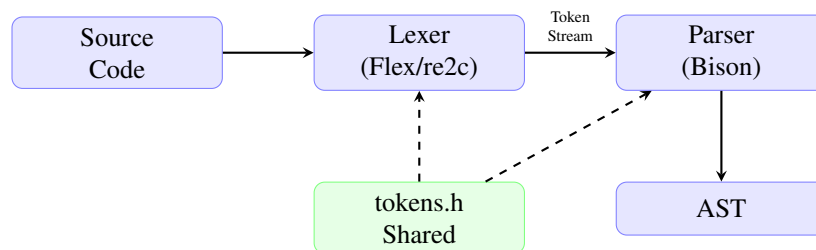
### 4.5.3 Perbandingan Performa

Secara umum, generator-based lexer (terutama re2c) memiliki performa yang sangat baik karena:

- Kode dihasilkan dengan optimasi DFA
- Tidak ada overhead dari table lookup (untuk re2c)
- Compiler dapat mengoptimasi generated code lebih baik

Hand-written lexer dapat lebih cepat hanya jika dioptimasi secara khusus untuk kasus tertentu, tetapi memerlukan effort yang lebih besar.

Gambar 4.17 menunjukkan alur integrasi lexer dengan parser.



Gambar 4.17: Integrasi lexer dengan parser

## 4.6 Integrasi dengan Parser

Lexer biasanya digunakan bersama dengan parser. Integrasi dilakukan melalui:

### 4.6.1 Token Definitions

Token constants didefinisikan dalam header file yang dibagi antara lexer dan parser:



Listing 4.4: File tokens.h

```

1 #ifndef TOKENS_H
2 #define TOKENS_H
3
4 typedef enum {
5     // Keywords
6     TOKEN_IF = 256,
7     TOKEN_ELSE,
8     TOKEN_WHILE,
9     TOKEN_RETURN,
10    TOKEN_INT,
11    TOKEN_FLOAT,
12
13    // Identifiers and literals
14    TOKEN_IDENTIFIER,
15    TOKEN_NUMBER,
16    TOKEN_FLOAT_LITERAL,
17
18    // Operators
19    TOKEN_PLUS,
20    TOKEN_MINUS,
21    TOKEN_MULTIPLY,
22    TOKEN_DIVIDE,
23    TOKEN_ASSIGN,
24    TOKEN_EQ,
25    TOKEN_NE,
26
27    // Punctuation
28    TOKEN_LPAREN,
29    TOKEN_RPAREN,
30    TOKEN_LBRACE,
31    TOKEN_RBRACE,
32    TOKEN_SEMICOLON,
33    TOKEN_COMMA,
34
35    TOKEN_EOF,
36    TOKEN_ERROR
37 } TokenType;
38
39 #endif

```

## 4.6.2 Semantic Values

Untuk mengirim nilai dari lexer ke parser, digunakan union `yylval`:

Listing 4.5: File YYSTYPE.h

```

1 #ifndef YYSTYPE_H
2 #define YYSTYPE_H
3
4 #include "tokens.h"
5
6 typedef union {
7     int intval;

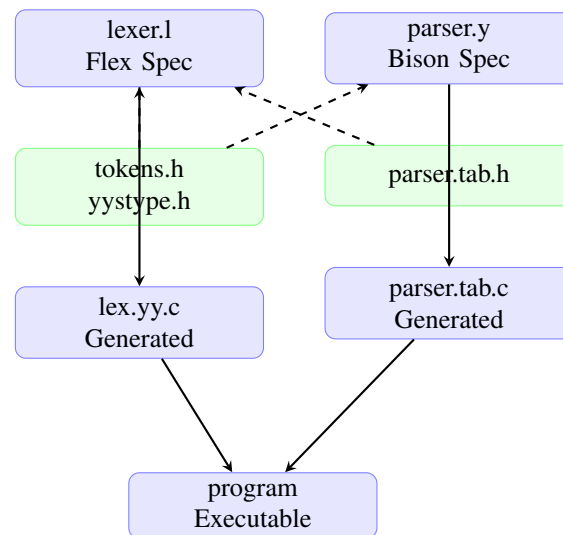
```

```

8      double floatval;
9      char *string;
10 } YYSTYPE;
11
12 extern YYSTYPE yylval;
13
14 #endif

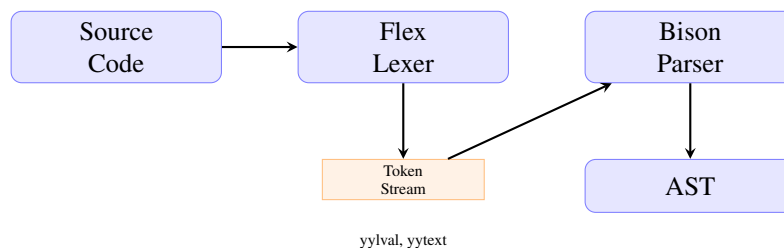
```

Gambar 4.18 menunjukkan contoh integrasi Flex dengan Bison secara detail.



Gambar 4.18: Integrasi Flex dengan Bison: file dan dependencies

Gambar 4.19 menunjukkan alur data dalam integrasi Flex-Bison.



Gambar 4.19: Alur data dalam integrasi Flex-Bison

### 4.6.3 Contoh Integrasi Flex dengan Bison

File Flex (lexer.l):

```

1 %{
2 #include "parser.tab.h"
3 #include "yytype.h"
4 %}
5

```

```

6 %%
7 {NUMBER} { yylval.intval = atoi(yytext); return NUMBER; }
8 {ID}      { yylval.string = strdup(yytext); return IDENTIFIER; }
9 %%

```

File Bison (parser.y):

```

%{
#include "yystype.h"
%}

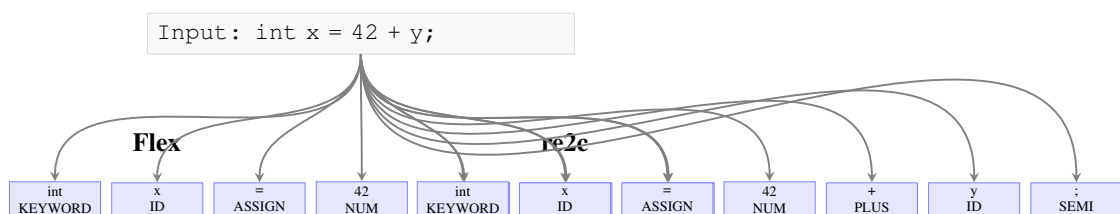
%union {
    int intval;
    char *string;
}

%token <intval> NUMBER
%token <string> IDENTIFIER

%%
expression: NUMBER { printf("Number: %d\n", $1); }
          | IDENTIFIER { printf("ID: %s\n", $1); }
          ;
%%

```

Gambar 4.20 menunjukkan contoh tokenization menggunakan Flex dan re2c untuk input yang sama, menunjukkan bahwa kedua tool menghasilkan hasil yang konsisten.



Gambar 4.20: Perbandingan tokenization: Flex dan re2c menghasilkan token stream yang konsisten untuk input `int x = 42 + y;`

## 4.7 Praktikum: Membuat Lexer dengan Flex

### 4.7.1 Tugas Praktikum

Buatlah lexer menggunakan Flex untuk bahasa mini dengan minimal 10 token types:

1. **Keywords:** `if, else, while, int, float, return`
2. **Identifiers:** Nama variabel dan fungsi
3. **Literals:** Integer dan float numbers
4. **Operators:** `+, -, *, /, =, ==, !=, <, >`
5. **Punctuation:** `(, ), {, }, ;, ,`
6. **Comments:** Single-line (`//`) dan multi-line (`/* */`)

### 4.7.2 Langkah-langkah

1. Buat file `lexer.l` dengan specification sesuai requirement
2. Generate lexer: `flex lexer.l`
3. Buat program test sederhana yang menggunakan `yylex()`
4. Test dengan berbagai input (valid dan invalid)
5. Dokumentasikan token types dan behavior lexer

### 4.7.3 Expected Output

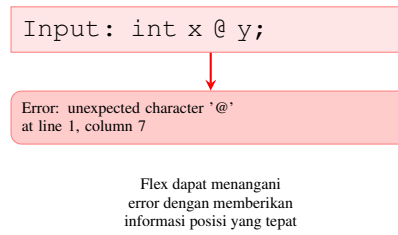
Lexer harus dapat:

- Mengenali semua token types yang didefinisikan
- Menangani whitespace dan comments dengan benar
- Memberikan error message yang informatif untuk invalid input
- Melacak line number untuk error reporting

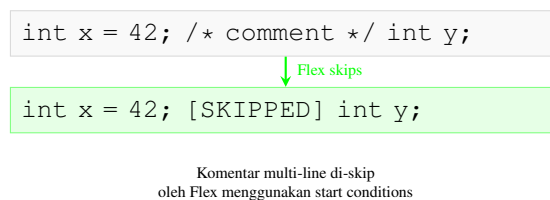
Gambar 4.21 menunjukkan contoh error handling dalam Flex.

Gambar 4.22 menunjukkan bagaimana Flex menangani komentar multi-line.

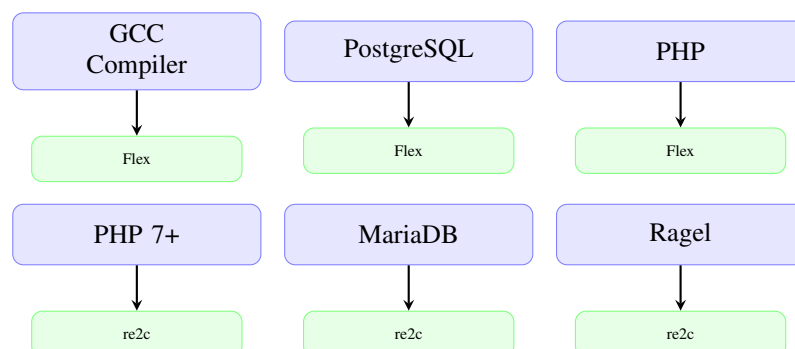
Gambar 4.23 menunjukkan penggunaan Flex dan re2c dalam project nyata.



Gambar 4.21: Error handling dalam Flex



Gambar 4.22: Handling komentar multi-line dalam Flex



Gambar 4.23: Contoh penggunaan Flex dan re2c dalam project nyata

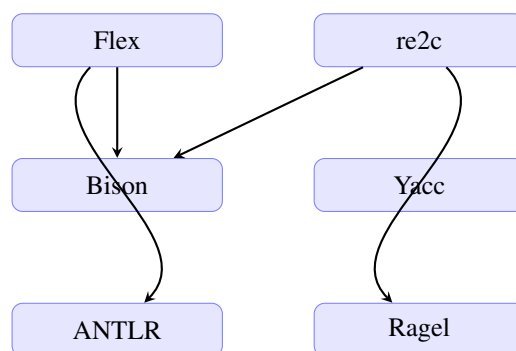
## 4.8 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Keuntungan menggunakan lexer generator dibanding hand-written lexer
2. Cara menggunakan Flex untuk membuat specification file dan generate lexer
3. Cara menggunakan re2c dengan embedded specification
4. Perbandingan antara hand-written dan generator-based lexer
5. Integrasi lexer dengan parser menggunakan token definitions dan semantic values

Generator-based lexer adalah pilihan yang tepat untuk sebagian besar kasus karena memberikan keseimbangan yang baik antara produktivitas, maintainability, dan performa. Namun, pemahaman tentang hand-written lexer (seperti yang dipelajari di bab sebelumnya) tetap penting untuk memahami proses tokenization secara mendalam.

Gambar 4.24 menunjukkan ekosistem tools yang terkait dengan lexer generator.



Gambar 4.24: Ekosistem tools lexer dan parser generator

## 4.9 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang lexer generator, mahasiswa disarankan membaca:

- **Flex Manual:** Dokumentasi resmi Flex <sup>2</sup>
- **re2c Documentation:** Dokumentasi dan tutorial re2c <sup>3</sup>
- **flex & bison:** Levine, J. R. (2009). *flex & bison: Text Processing Tools* [2] - Bab 2: Using Flex

---

<sup>2</sup><https://www.gnu.org/software/flex/manual/>

<sup>3</sup><https://re2c.org/>

- **Engineering a Compiler:** Cooper & Torczon (2011) [3] - Bab 2: Lexical Analysis (bagian tentang lexer generators)
- **IT Trip - C Parser Flex Bison:** Tutorial tentang integrasi Flex dan Bison [4]
- **Wikipedia - re2c:** Artikel tentang re2c [1]
- **Wikipedia - RE/flex:** Artikel tentang RE/flex (modern C++ lexer generator) <sup>4</sup>

---

<sup>4</sup><https://en.wikipedia.org/wiki/Draft:RE/flex>





# Daftar Pustaka

- [1] Wikipedia. *re2c*. Encyclopedia entry. 2024. URL: <https://en.wikipedia.org/wiki/Re2c>.
- [2] John R. Levine. *flex & bison: Text Processing Tools*. O'Reilly Media, 2009.
- [3] Keith D. Cooper **and** Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2011.
- [4] IT Trip. *C Parser Flex Bison*. Tutorial. 2024. URL: <https://en.ittrip.xyz/c-language/c-parser-flex-bison>.