Bab 1

Lexer Generator (Flex/re2c) dan Praktikum Lexer

1.1 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

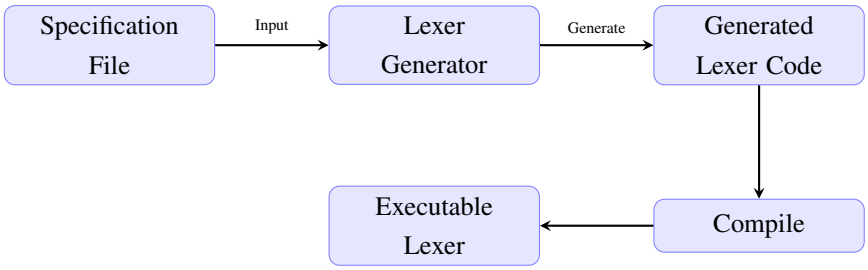
1.2 Pendahuluan

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."**?**

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 1.1 menunjukkan alur kerja lexer generator secara umum.



Gambar 1.1: 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

1.3 Flex (Fast Lexical Analyzer)

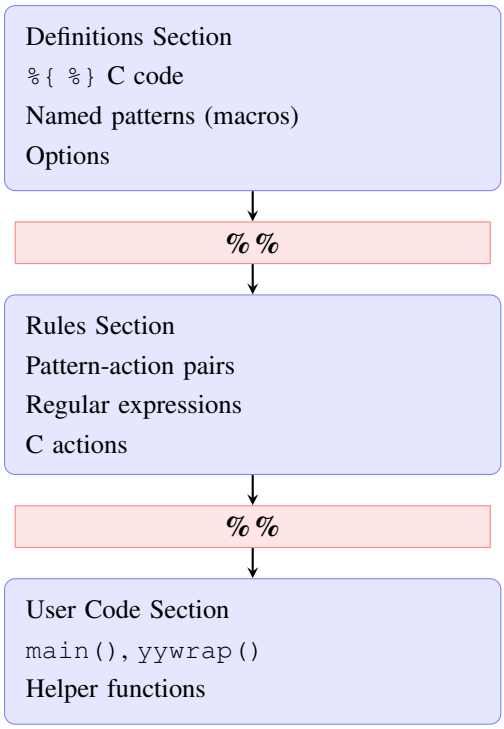
Flex adalah lexer generator yang paling banyak digunakan, terutama dalam kombinasi dengan Bison (parser generator). Flex membaca specification file dengan ekstensi `.l` dan menghasilkan kode C untuk lexer.

1.3.1 Struktur Flex Specification File

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

```
Definitions
%%
Rules
%%
User Code
```

Gambar 1.2 menunjukkan struktur file specification Flex secara visual.



Gambar 1.2: 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}+
```

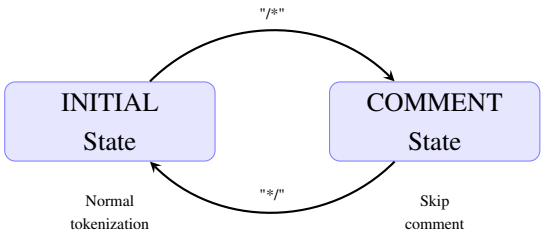
Rules Section

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

Contoh:

```
%%
"if"            { return IF; }
"else"          { return ELSE; }
"while"         { return WHILE; }
{ID}            { return IDENTIFIER; }
{NUMBER}        { yylval.intval = atoi(yytext); return NUMBER; }
"=="            { return EQ; }
"!="            { return NE; }
[ \t\n]+        { /* skip whitespace */ }
"//".*          { /* skip single-line comment */ }
"/*"            { BEGIN(COMMENT); }
<COMMENT>"*/"   { BEGIN(INITIAL); }
<COMMENT>.      { /* skip comment content */ }
.               { return yytext[0]; }  /* default: return character */
%%
```

Gambar 1.3 menunjukkan penggunaan start conditions dalam Flex untuk menangani komentar multi-line.

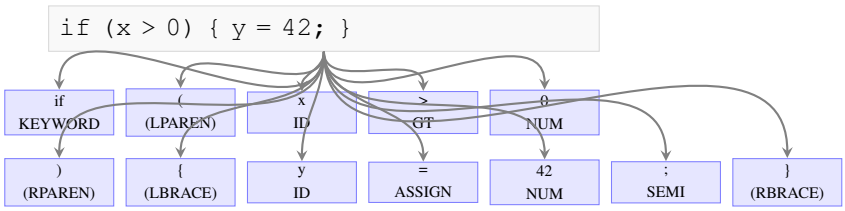


Gambar 1.3: Start conditions dalam Flex untuk handling komentar

User Code Section

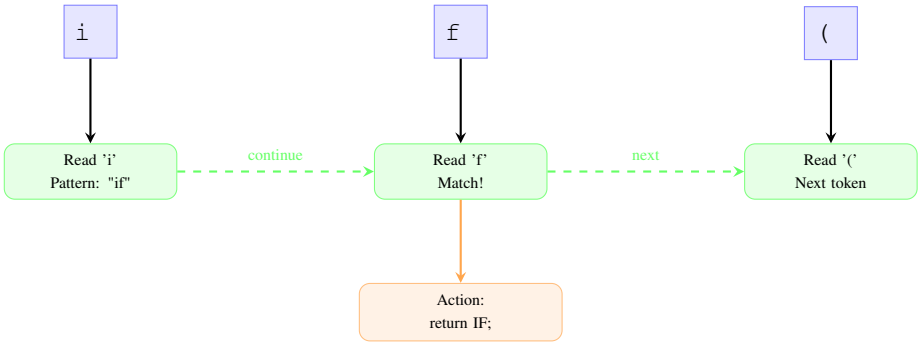
Bagian ini berisi fungsi-fungsi pendukung seperti ‘main()‘, ‘yywrap()‘, dan helper functions.

Gambar 1.4 menunjukkan contoh lengkap penggunaan Flex dari input hingga output token, mengikuti format yang konsisten dengan diagram tokenization di bab sebelumnya.



Gambar 1.4: Contoh lengkap tokenization dengan Flex: `if (x > 0) { y = 42; }`

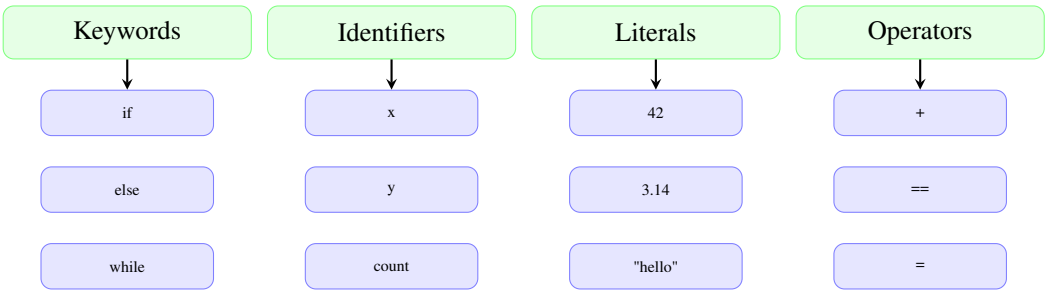
Gambar 1.5 menunjukkan proses pattern matching dalam Flex untuk keyword `"if"`.



Flex mencocokkan pola `"if"` dan mengeksekusi aksi leksikal

Gambar 1.5: Proses pattern matching dalam Flex: pencocokan keyword `"if"`

Gambar 1.6 menunjukkan berbagai jenis token yang dapat dikenali oleh Flex.



Gambar 1.6: Jenis-jenis token yang dikenali Flex

1.3.2 Contoh Lengkap: Flex Lexer untuk Bahasa Sederhana

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

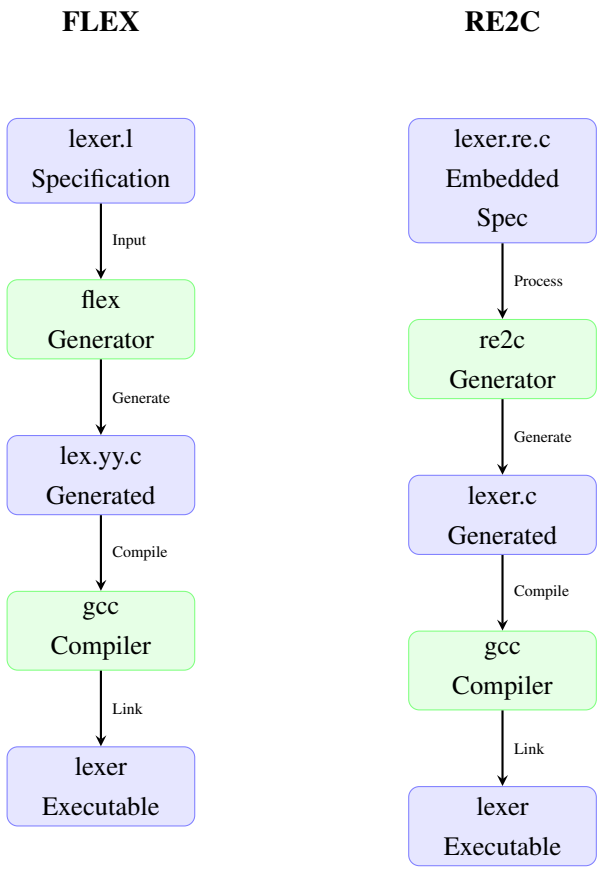
Listing 1.1: Contoh Flex specification file (calc.l)

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "parser.tab.h"  // Header dari Bison

int yylineno = 1;
%}

%option noyywrap
%option yylineno

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

```
NUMBER    {DIGIT}+
FLOAT     {DIGIT}+\.{DIGIT}+

%%

"int"         { return INT; }
"float"       { return FLOAT_TYPE; }
"if"          { return IF; }
"else"        { return ELSE; }
"while"       { return WHILE; }
"return"    { return RETURN; }

{ID}          {
                  yylval.string = strdup(yytext);
                  return IDENTIFIER;
              }

{NUMBER}      {
                  yylval.intval = atoi(yytext);
                  return NUMBER;
              }

{FLOAT}       {
                  yylval.floatval = atof(yytext);
                  return FLOAT_LITERAL;
              }

"+"           { return PLUS; }
"-"           { return MINUS; }
"*"           { return MULTIPLY; }
"/"           { return DIVIDE; }
"="           { return ASSIGN; }
"=="          { return EQ; }
"!="          { return NE; }
"<"           { return LT; }
">"           { return GT; }
"<="          { return LE; }
">="          { return GE; }

"("           { return LPAREN; }
")"           { return RPAREN; }
"{"           { return LBRACE; }
"}"           { return RBRACE; }
";"           { return SEMICOLON; }
","           { return COMMA; }

[ \t]+        { /* skip whitespace */ }
\n            { yylineno++; }
"//".*        { /* skip single-line comment */ }
"/*"          {
                  int c;
                  while ((c = input()) != EOF) {
                      if (c == '\n') yylineno++;
                      if (c == '*' && (c = input()) == '/') break;
```

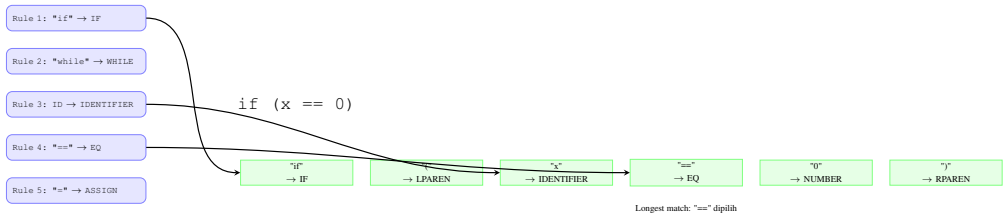
```
69                      if (c != EOF) unput(c);
70                  }
71              }
72
73  .           {
74                  fprintf(stderr, "Error: unexpected character '%c' at line
↪  %d\n",
75                          yytext[0], yylineno);
76                  return ERROR;
77              }
78
79  %%
80
81  int yywrap(void) {
82      return 1;
83  }
```

Gambar 1.7 menunjukkan workflow kompilasi dan penggunaan Flex dan re2c secara perbandingan.



Gambar 1.7: Workflow kompilasi dan penggunaan Flex dan re2c

Gambar 1.8 menunjukkan bagaimana Flex menangani rule priority dan longest match.



Gambar 1.8: Rule priority dan longest match dalam Flex

1.3.3 Kompilasi dan Penggunaan Flex

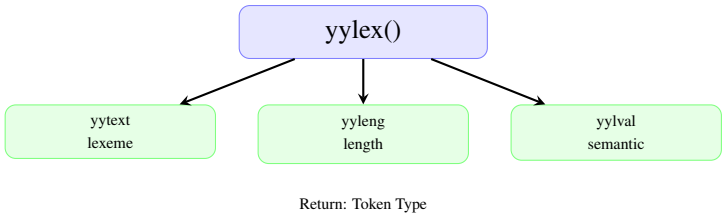
Untuk menggunakan Flex:

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

Gambar 1.9 menunjukkan penggunaan fungsi-fungsi utama Flex.



Gambar 1.9: 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 1.7.

1.4 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."[1]

1.4.1 Struktur re2c Specification

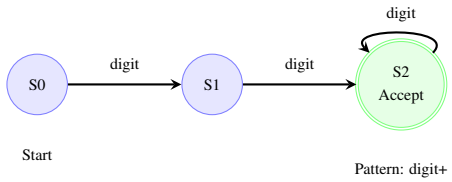
re2c specification ditulis dalam komentar khusus '/*!re2c ... */' yang disisipkan dalam kode C/C++:

Listing 1.2: Struktur dasar re2c

```c
#include <stdio.h>

static int lex(const char *YYCURSOR) {
    const char *YYMARKER;
    /*!re2c
        re2c:define:YYCTYPE = "char";
        re2c:yyfill:enable = 0;

        // Named patterns
        digit    = [0-9];
        letter   = [a-zA-Z_];
        id       = letter (letter | digit)*;
        number   = digit+;

        // Rules
        *        { return 0; }  // error
        number   { printf("Number: %.*s\n", (int)(YYCURSOR - YYMARKER),
↪ YYMARKER); return 1; }
        id       { printf("ID: %.*s\n", (int)(YYCURSOR - YYMARKER),
↪ YYMARKER); return 1; }
        [ \t\n]+ { continue; }  // skip whitespace
    */
}

int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        lex(argv[i]);
    }
    return 0;
}
```

[1] https://re2c.org/

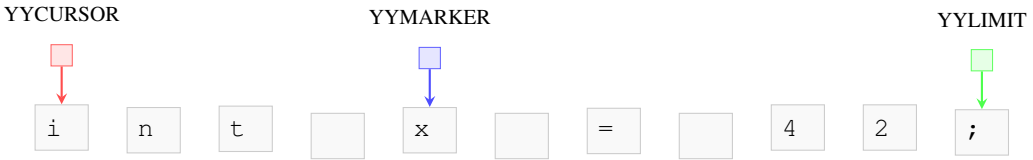
Gambar 1.10 menunjukkan bagaimana re2c menghasilkan state machine untuk pattern matching.



Gambar 1.10: State machine yang dihasilkan re2c untuk pattern `digit+`

1.4.2 Key Concepts dalam re2c

Gambar 1.11 menunjukkan penggunaan variabel khusus dalam re2c.



Gambar 1.11: Ilustrasi pointer internal `re2c` dalam buffer input

Variables Khusus

re2c menggunakan variabel khusus untuk tracking posisi input:

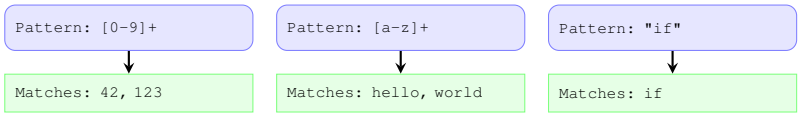
- `YYCURSOR`: Pointer ke posisi saat ini dalam input
- `YYMARKER`: Pointer untuk backtracking
- `YYLIMIT`: Pointer ke akhir buffer
- `YYCTYPE`: Tipe data untuk karakter (default: `unsigned char`)

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 1.12 menunjukkan berbagai pattern yang dapat digunakan dalam re2c.



Gambar 1.12: Contoh pattern dalam re2c

1.4.3 Contoh Lengkap: re2c Lexer untuk Identifier dan Number

Listing 1.3: Contoh re2c lexer (lexer.re.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef enum {
    TOKEN_EOF,
    TOKEN_IDENTIFIER,
    TOKEN_NUMBER,
    TOKEN_PLUS,
    TOKEN_MINUS,
    TOKEN_MULTIPLY,
    TOKEN_DIVIDE,
    TOKEN_ERROR
} TokenType;

typedef struct {
    TokenType type;
    char *value;
    int line;
} Token;

static Token tokenize(const char *input, int *line) {
    const char *YYCURSOR = input;
    const char *YYMARKER;
    const char *start;
    Token token = {TOKEN_EOF, NULL, *line};

    /*!re2c
        re2c:define:YYCTYPE = "char";
        re2c:yyfill:enable = 0;
        re2c:define:YYCURSOR = "YYCURSOR";

        digit    = [0-9];
        letter   = [a-zA-Z_];
        id       = letter (letter | digit)*;
        number   = digit+;
        ws       = [ \t]+;
        newline  = "\n";

        * {
```

```
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;
```

```
            token.line = *line;
            return token;
        }

        "/" {
            token.type = TOKEN_DIVIDE;
            token.line = *line;
            return token;
        }
    */
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <input>\n", argv[0]);
        return 1;
    }

    int line = 1;
    Token token;

    do {
        token = tokenize(argv[1], &line);
        printf("Token: %d, Value: %s, Line: %d\n",
               token.type, token.value ? token.value : "NULL", token.line
↪ );
        if (token.value) free(token.value);
    } while (token.type != TOKEN_EOF && token.type != TOKEN_ERROR);

    return 0;
}
```

Untuk mengkompilasi:

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

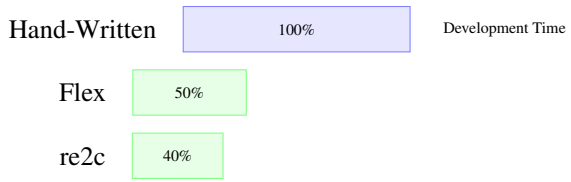
Gambar 1.13 menunjukkan perbandingan visual antara hand-written, Flex, dan re2c.

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

Gambar 1.13: Perbandingan hand-written, Flex, dan re2c

Gambar 1.14 menunjukkan perbandingan waktu development antara berbagai

pendekatan.



Gambar 1.14: Perbandingan waktu development (hand-written = baseline)

1.5 Perbandingan Hand-Written vs Generator-Based Lexer

Setelah mempelajari kedua pendekatan, mari kita bandingkan:

1.5.1 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

1.5.2 Generator-Based Lexer

Keuntungan:

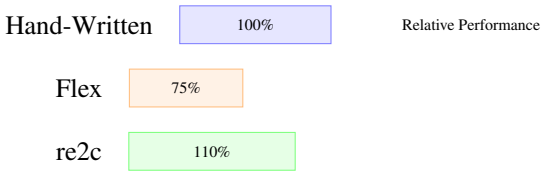
- Specification lebih ringkas dan mudah dibaca
- Generator menghasilkan kode yang sudah teroptimasi
- Lebih cepat dalam development
- Pattern mudah dimodifikasi tanpa mengubah banyak kode

- Sudah teruji dan digunakan di banyak project

Kekurangan:

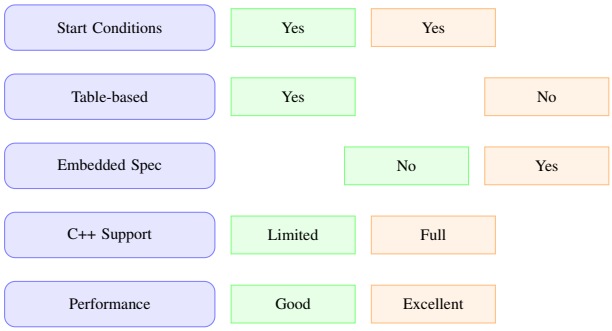
- Perlu mempelajari syntax generator
- Dependency pada tool eksternal
- Kurang fleksibel untuk kasus yang sangat spesifik
- Generated code mungkin lebih sulit di-debug

Gambar 1.15 menunjukkan perbandingan performa secara visual.



Gambar 1.15: Perbandingan performa relatif (hand-written = baseline)

Gambar 1.16 menunjukkan perbandingan fitur antara Flex dan re2c.



Gambar 1.16: Perbandingan fitur Flex vs re2c

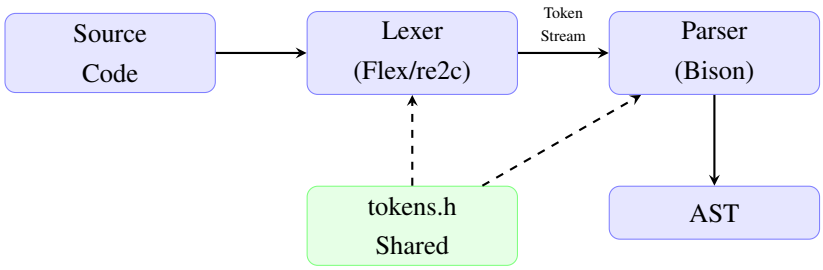
1.5.3 Perbandingan Performa

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 1.17 menunjukkan alur integrasi lexer dengan parser.



Gambar 1.17: Integrasi lexer dengan parser

1.6 Integrasi dengan Parser

Lexer biasanya digunakan bersama dengan parser. Integrasi dilakukan melalui:

1.6.1 Token Definitions

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

Listing 1.4: File tokens.h

```
#ifndef TOKENS_H
#define TOKENS_H

typedef enum {
    // Keywords
    TOKEN_IF = 256,
    TOKEN_ELSE,
    TOKEN_WHILE,
    TOKEN_RETURN,
    TOKEN_INT,
    TOKEN_FLOAT,

    // Identifiers and literals
    TOKEN_IDENTIFIER,
    TOKEN_NUMBER,
    TOKEN_FLOAT_LITERAL,

    // Operators
    TOKEN_PLUS,
    TOKEN_MINUS,
    TOKEN_MULTIPLY,
    TOKEN_DIVIDE,
    TOKEN_ASSIGN,
    TOKEN_EQ,
    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
```

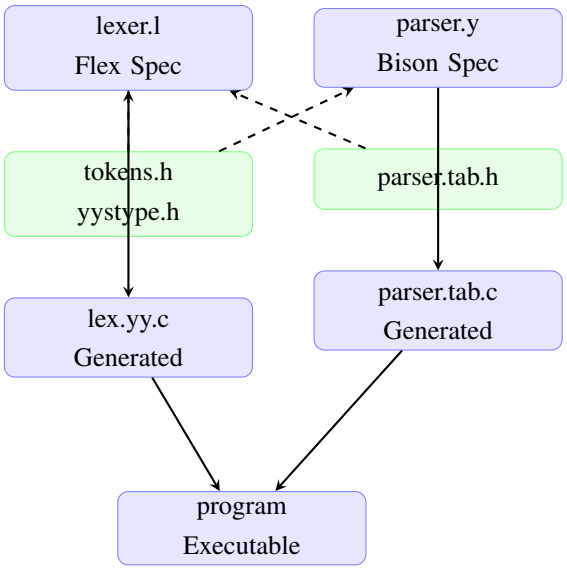
1.6.2 Semantic Values

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

Listing 1.5: File yystype.h

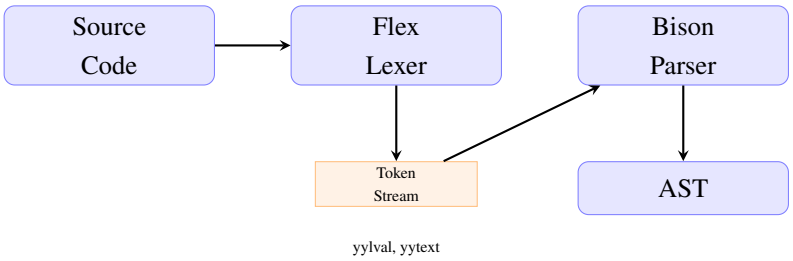
```
1  #ifndef YYSTYPE_H
2  #define YYSTYPE_H
3
4  #include "tokens.h"
5
6  typedef union {
7      int intval;
8      double floatval;
9      char *string;
10 } YYSTYPE;
11
12 extern YYSTYPE yylval;
13
14 #endif
```

Gambar 1.18 menunjukkan contoh integrasi Flex dengan Bison secara detail.



Gambar 1.18: Integrasi Flex dengan Bison: file dan dependencies

Gambar 1.19 menunjukkan alur data dalam integrasi Flex-Bison.



Gambar 1.19: Alur data dalam integrasi Flex-Bison

1.6.3 Contoh Integrasi Flex dengan Bison

File Flex (`lexer.l`):

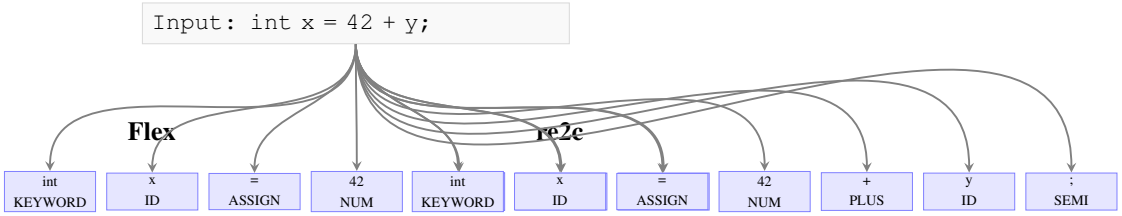
```
%{
#include "parser.tab.h"
#include "yystype.h"
%}

%%
{NUMBER} { yylval.intval = atoi(yytext); return NUMBER; }
{ID}     { yylval.string = strdup(yytext); return IDENTIFIER; }
%%
```

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



Gambar 1.20: Perbandingan tokenization: Flex dan re2c menghasilkan token stream yang konsisten untuk input `int x = 42 + y;`

1.7 Praktikum: Membuat Lexer dengan Flex

1.7.1 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 (`/* */`)

1.7.2 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)

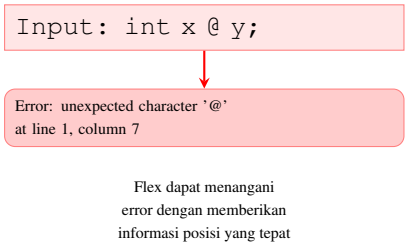
5. Dokumentasikan token types dan behavior lexer

1.7.3 Expected Output

Lexer harus dapat:

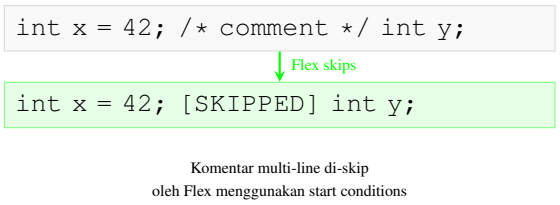
- Mengenali semua token types yang didefinisikan
- Menangani whitespace dan comments dengan benar
- Memberikan error message yang informatif untuk invalid input
- Melacak line number untuk error reporting

Gambar 1.21 menunjukkan contoh error handling dalam Flex.



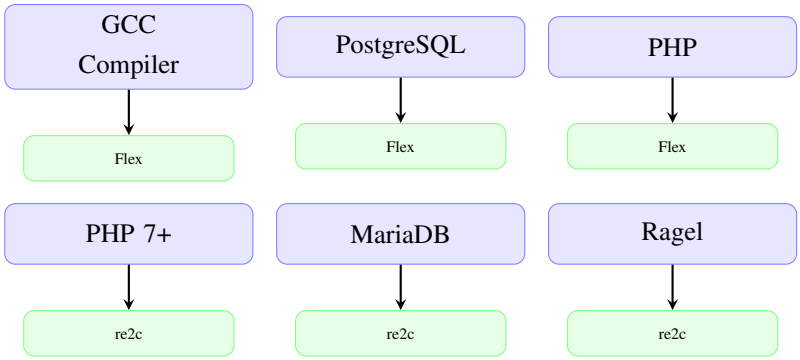
Gambar 1.21: Error handling dalam Flex

Gambar 1.22 menunjukkan bagaimana Flex menangani komentar multi-line.



Gambar 1.22: Handling komentar multi-line dalam Flex

Gambar 1.23 menunjukkan penggunaan Flex dan re2c dalam project nyata.



Gambar 1.23: Contoh penggunaan Flex dan re2c dalam project nyata

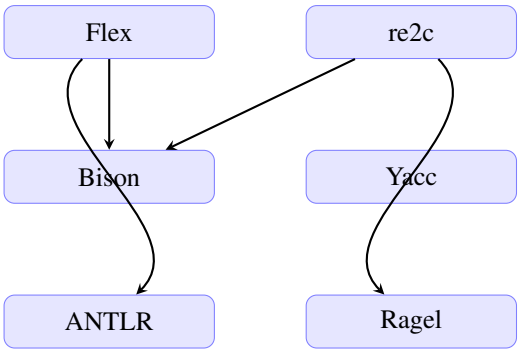
1.8 Kesimpulan

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 1.24 menunjukkan ekosistem tools yang terkait dengan lexer generator.



Gambar 1.24: Ekosistem tools lexer dan parser generator

1.9 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang lexer generator, mahasiswa disarankan membaca:

- **Flex Manual**: Dokumentasi resmi Flex [2]
- **re2c Documentation**: Dokumentasi dan tutorial re2c [3]
- **flex & bison**: Levine, J. R. (2009). *flex & bison: Text Processing Tools* **?** - Bab 2: Using Flex
- **Engineering a Compiler**: Cooper & Torczon (2011) **?** - Bab 2: Lexical Analysis (bagian tentang lexer generators)
- **IT Trip - C Parser Flex Bison**: Tutorial tentang integrasi Flex dan Bison **?**
- **Wikipedia - re2c**: Artikel tentang re2c **?**
- **Wikipedia - RE/flex**: Artikel tentang RE/flex (modern C++ lexer generator) [4]

[2] https://www.gnu.org/software/flex/manual/
[3] https://re2c.org/
[4] https://en.wikipedia.org/wiki/Draft:RE/flex