Laporan Tugas II
IF2220 Teori Bahasa Formal dan Automata

Jonathan Christopher
(13515001, K-01)

Nicholas Thie
(13515079, K-01)

November 22, 2016

1 Deskripsi Persoalan

Dibutuhkan sebuah program untuk mengenali dan menghitung ekspresi aritmatika menggunakan tata bahasa bebas konteks (*context-free grammar*). Bila diberikan sebuah ekspresi aritmatika, maka program harus bisa mengenali apakah ekspresi tersebut valid atau tidak (*syntax error*). Jika ekspresi yang diberikan valid, maka program tersebut harus menghitung nilai dari ekspresi tersebut dengan mengubah terlebih dahulu setiap simbol terminal (angka) menjadi nilai numerik yang bersesuaian. Contoh ekspresi aritmatika yang valid adalah $(-457.01 + 1280) * (35.7 - 11.0233)/(-6.1450)$ (setelah dieksekusi akan menampilkan hasil perhitungan ekspresi tersebut yaitu -3304.91). Contoh ekspresi tidak valid adalah $3 * + - 12/(57)$ (setelah dieksekusi akan ditampilkan pesan *syntax error*).

1.1 Batasan Masalah

- Terminal terdiri dari karakter-karakter $\{+, -, *, /, (,), 0..9, .\}$.
- Operator terdiri dari $\{+, -, *, /, (,)\}$.
- Ekspresi di dalam kurung dievaluasi terlebih dahulu, kemudian operator perkalian dan pembagian, baru sesudahnya operator penambahan dan pengurangan.
- Operan terdiri dari bilangan bulat dan bilangan desimal positif atau negatif.
- Program merupakan implementasi dari tata bahasa dan *pushdown automata* yang dibuat terlebih dahulu (disain sendiri) menggunakan teori yang telah dipelajari.
- Implementasi program menggunakan bahasa pemrograman prosedural C atau Pascal.

2 Jawab Persoalan

Untuk validasi sintaks dan perhitungan ekspresi aritmatika, digunakan CFG (*context-free grammar*). Ekspresi aritmatika terlebih dahulu dimodelkan dalam context-free grammar tersebut. Karakter-karakter yang dierima dalam ekspresi, yaitu $\{+, -, *, /, (,), 0..9, .\}$, menjadi simbol terminal dalam CFG yang dibuat. Selain simbol terminal, CFG tersebut juga mengandung sejumlah variabel yang merepresentasikan fungsi dari potongan-potongan ekspresi tertentu. Salah satu variabel melambangkan keseluruhan ekspresi aritmatika itu sendiri dan menjadi *start symbol*. Eksekusi untuk validasi dan perhitungan akan selalu dimulai dengan variabel tersebut, lalu berlanjut ke sejumlah aturan produksi yang mendefinisikan tata bahasa ekspresi secara rekursif.

Implementasi *parser* yang dibuat menggunakan metode rekursif, sehingga program yang dihasilkan strukturnya tidak jauh berbeda dengan CFG yang digunakan. Sebuah variabel dalam CFG diubah menjadi fungsi dalam program, yang dapat memanggil fungsi-fungsi lainnya yang melambangkan variabel lainnya sesuai aturan produksi CFG. Karakter terminal dalam CFG berarti program harus membaca karakter tersebut dari *string* masukan. Akan tetapi, terdapat beberapa kriteria yang harus dipenuhi oleh CFG yang digunakan agar implementasi dengan metode rekursif ini dapat berjalan. Pertama, CFG tidak boleh ambigu, atau memiliki beberapa kemungkinan alur *parsing*. Kedua, tidak boleh ada aturan produksi dalam CFG dimana suatu variabel mengandung dirinya sendiri sebelum variabel lain atau simbol terminal.

Sebuah CFG pada dasarnya hanya dapat digunakan untuk mengecek apakah suatu *string* memenuhi aturan CFG tersebut atau tidak (dalam hal ini, validitas sintaks sebuah ekspresi aritmatika). Untuk dapat menghitung juga nilai hasil evaluasi suatu ekspresi aritmatika, dibutuhkan modifikasi pada program yang mengimplementasikannya. Selain dapat mem-*parse* simbol terminal dan variabel, program yang dibuat juga dapat mengartikan nilai numerik simbol terminal yang berupa digit serta mengaitkannya dengan urutan dan makna yang benar tergantung pada

simbol terminal operator yang ditemukan. Hasil evaluasi dikembalikan dalam *return value* tiap fungsi yang merepresentasikan variabel dalam CFG. Selain itu, untuk mempertahankan urutan evaluasi yang asosiatif-kiri (untuk operator yang prioritasnya sama), terpaksa digunakan iterasi untuk bagian tersebut, bukan rekursi.

Selain implementasi CFG dan evaluasi ekspresi, terdapat beberapa fitur tambahan yang dimuat dalam program ini. Terdapat empat tipe hasil evaluasi ekspresi yang dikenali, yaitu bilangan bulat, bilangan desimal, *syntax error*, serta *division error* (terjadi jika ada pembagian dengan nol). Bilangan bulat disimpan dalam tipe *integer* 64-bit, sedangkan bilangan desimal disimpan dalam tipe *double-precision floating point*. Program sedapat mungkin melakukan evaluasi dalam tipe bilangan bulat dan menggunakan tipe *floating-point* jika ada hasil yang memang tidak bulat. Hal ini dilakukan untuk sedapat mungkin mencegah ketidaktelitian perhitungan yang diakibatkan keterbatasan representasi *floating-point*. Selain itu, jika terdapat *syntax error*, program juga akan mencatat posisi karakter pertama yang mengakibatkan *syntax error* untuk mempermudah pencarian kesalahan.

Terdapat beberapa aturan ekspresi aritmatika tambahan yang diasumsikan berlaku:

- Operator uner positif tidak boleh digunakan (dapat diatur dengan mendefinisikan makro *ALLOW_UNARY_POSITIVE_OPERATOR* di file *evaluator.c*).
- Operator uner tidak boleh konsekutif tanpa dipisahkan kurung (dapat diatur dengan mendefinisikan makro *ALLOW_CONSECUTIVE_UNARY_OPERATOR* di file *evaluator.c*).
- *Leading zeroes* (digit 0 di depan suatu angka) diperbolehkan.
- Urutan operasi dari operator yang prioritasnya sama adalah asosiatif kiri (dari kiri ke kanan).

3 *Context-Free Grammar*

Berikut adalah tata bahasa bebas konteks (*context-free grammar*) yang digunakan untuk memodelkan ekspresi aritmatika yang akan divalidasi dan dievaluasi:

$$G = (\{E, T, F, W, N, F, I, D\}, \{+, -, *, /, (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}, P, E)$$

dengan aturan produksi P yang didefinisikan sebagai berikut:

$$\begin{array}{rcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T/F \mid F \\ F & \rightarrow & -F \mid +F \mid W \\ W & \rightarrow & (E) \mid N \\ N & \rightarrow & I.F \mid I \\ F & \rightarrow & D \mid DF \\ I & \rightarrow & D \mid DI \\ D & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

Setiap variabel dalam CFG tersebut merepresentasikan sebuah komponen ekspresi aritmatika:

Variabel	Nama fungsi dalam program	Deskripsi
E	expression	keseluruhan ekspresi aritmatika
T	term	ekspresi angka, hasil perkalian atau hasil pembagian
F	factor	ekspresi yang dapat menjadi bagian dari operasi biner
W	factorWithoutUnary	angka atau ekspresi aritmatika yang dibatasi kurung
N	number	bilangan bulat atau desimal tidak negatif
I	integer	bilangan bulat tidak negatif
F	fractional	urutan digit yang terletak di belakang koma
D	digit	sebuah karakter digit (0..9)

4 *Source Code*

4.1 main.c

```
#include <stdio.h>
#include "stringutils.h"
#include "evaluator.h"

int main() {

    printf("\n");
    printf("EVALUATOR␣EKSPRESI␣ARITMATIKA\n");
    printf("============================\n");
    printf("Tugas␣2␣IF2220␣Teori␣Bahasa␣Formal␣dan␣Otomata,␣November␣2016\n");
    printf("13515001␣-␣Jonathan␣Christopher\n");
    printf("13515079␣-␣Nicholas␣Thie\n");
    printf("Masukkan␣ekspresi␣aritmatika␣atau␣[exit]␣untuk␣keluar.\n\n");

    while (1) {
        printf(">>␣");

        char *input;
        input = StringUtils_scan(stdin, '\n');

        if (StringUtils_strcmpi(input, "keluar") == 0 || StringUtils_strcmpi(input, "exit") ==
             0) break;

        EvalResult res = evaluate(input);
        if (res.resultType == INTEGRAL) {
            printf("%lld\n", res.integralValue);
        } else if (res.resultType == FRACTIONAL) {
            printf("%lf\n", res.fractionalValue);
        } else if (res.resultType == SYNTAX_ERROR) {
            printf("Syntax␣error␣at␣position␣%d.\n", res.errorPos+1);
        } else if (res.resultType == DIVISION_ERROR) {
            printf("Division␣by␣zero␣error.\n");
        }

        StringUtils_deallocate(input);
    }

    printf("\n");
    return 0;
}
```

4.2 evaluator.h

```
#ifndef EVALUATOR_H
#define EVALUATOR_H
```

```c
#include "evalresult.h"

EvalResult evaluate(const char *input);

#endif
```

4.3 evaluator.c

```c
#include <string.h>
#include "evaluator.h"

/* Pengaturan kompilasi: menentukan karakter koma (pembatas desimal) */
#define DECIMAL_POINT '.'

/* Pengaturan kompilasi: jika terdefinisi, maka operator '+' uner diperbolehkan */
// #define ALLOW_UNARY_POSITIVE_OPERATOR

/* Pengaturan kompilasi: jika terdefinisi, maka operator uner '+' dan/atau '-' yang tepat
    berurutan diperbolehkan */
// #define ALLOW_CONSECUTIVE_UNARY_OPERATORS

EvalResult expression(const char *input, size_t *pos, size_t end);

/* Terminal digit */
EvalResult digit(const char *input, size_t *pos, size_t end) {
    if (*pos <= end && '0' <= input[*pos] && input[*pos] <= '9') {
        return integralResult(input[(*pos)++] - '0');
    } else {
        return syntaxErrorResult(*pos);
    }
}

/* Fractional -> Digit | Digit_Fractional */
EvalResult fractional(const char *input, size_t *pos, size_t end) {
    EvalResult res1 = digit(input, pos, end);
    if (isError(res1)) return res1;

    EvalResult res2 = fractional(input, pos, end);
    if (isError(res2)) {
        return fractionalResult(0.1 * (double) res1.integralValue); // Digit
    }

    return fractionalResult(0.1 * ((double) res1.integralValue + res2.fractionalValue)); //
        Digit_Fractional
}

/* Integer -> Digit | Digit_Integer */
EvalResult integer(const char *input, size_t *pos, size_t end) {
    size_t initialPos = *pos;

    EvalResult res1 = digit(input, pos, end);
    if (isError(res1)) return res1;

    EvalResult res2 = integer(input, pos, end);
    if (isError(res2)) {
        return integralResult(res1.integralValue); // Digit
    }

    long long exp = 1;
    int i;
    for (i = 0; i < (*pos)-initialPos-1; i++) exp *= 10;

    return integralResult(res1.integralValue * exp + res2.integralValue); // Digit_Integer
}

/* Number -> Integer.Fractional | Integer */
EvalResult number(const char *input, size_t *pos, size_t end) {
    EvalResult res1 = integer(input, pos, end);
```

```
    if (isError(res1)) return res1;

    if (*pos <= end && input[*pos] == DECIMAL_POINT) {
        (*pos)++;
        EvalResult res2 = fractional(input, pos, end);
        if (isError(res2)) return res2;
        return fractionalResult((double) res1.integralValue + res2.fractionalValue); // Integer.
            Fractional
    }

    return integralResult(res1.integralValue); // Integer
}

/* FactorWithoutUnary -> (Expression) | Number */
EvalResult factorWithoutUnary(const char *input, size_t *pos, size_t end) {
    if (*pos <= end && input[*pos] == '(') {
        (*pos)++;
        EvalResult res = expression(input, pos, end);
        if (*pos <= end && input[*pos] == ')') {
            (*pos)++;
            return res; // (Expression)
        } else {
            return syntaxErrorResult(*pos); // Tidak ada kurung tutup yang cocok
        }
    } else {
        return number(input, pos, end); // number
    }
}

/* Factor -> -Factor | +Factor | FactorWithoutUnary */
EvalResult factor(const char *input, size_t *pos, size_t end) {
    if (*pos <= end && input[*pos] == '-') { // Operator uner negatif
        (*pos)++;
        return multiply(factor(input, pos, end), integralResult(-1));
    }

#ifdef ALLOW_UNARY_POSITIVE_OPERATOR
    else if (*pos <= end && input[*pos] == '+') { // Operator uner positif
        (*pos)++;
        return factor(input, pos, end);
    }
#endif

    return factorWithoutUnary(input, pos, end); // FactorWithoutUnary
}

/* Term -> Term*Factor | Term/Factor | Factor
   Dimodifikasi menjadi Factor_((*|/)_Factor)* untuk mencegah rekursi tanpa batas dan
       mempertahankan asosiativitas-kiri */
EvalResult term(const char *input, size_t *pos, size_t end) {
    EvalResult res1 = factor(input, pos, end);
    if (isError(res1)) return res1;

    EvalResult resAccumulator = res1;
    while(1) {
        if (*pos <= end && input[*pos] == '*') {
            (*pos)++;

            EvalResult res2 = factor(input, pos, end);
            if (isError(res2)) resAccumulator = res2;
            resAccumulator = multiply(resAccumulator, res2);

        } else if (*pos <= end && input[*pos] == '/') {
            (*pos)++;

            EvalResult res2 = factor(input, pos, end);
            if (isError(res2)) resAccumulator = res2;
            resAccumulator = divide(resAccumulator, res2);

        } else {
            break;
```

```
        }
    }
    return resAccumulator;
}

/* Expression -> Expression+Term | Expression-Term | Term
   Dimodifikasi menjadi Term_((+|-)_Term)* untuk mencegah rekursi tanpa batas dan mempertahankan
        asosiativitas-kiri */
EvalResult expression(const char *input, size_t *pos, size_t end) {
    EvalResult res1 = term(input, pos, end);
    if (isError(res1)) return res1;

    EvalResult resAccumulator = res1;
    while(1) {
        if (*pos <= end && input[*pos] == '+') {
            (*pos)++;

            EvalResult res2 = term(input, pos, end);
            if (isError(res2)) resAccumulator = res2;
            resAccumulator = add(resAccumulator, res2);

        } else if (*pos <= end && input[*pos] == '-') {
            (*pos)++;

            EvalResult res2 = term(input, pos, end);
            if (isError(res2)) resAccumulator = res2;
            resAccumulator = subtract(resAccumulator, res2);

        } else {
            break;
        }
    }
    return resAccumulator;
}

/* Fungsi pembungkus untuk mempermudah pemanggilan; menambahkan beberapa pengecekan awal dan
    akhir */
EvalResult evaluate(const char *input) {
    size_t pos = 0;
    size_t length = strlen(input);

    #ifndef ALLOW_CONSECUTIVE_UNARY_OPERATORS
        // Menggagalkan evaluasi apabila terdapat operator uner '+' dan/atau '-' yang tepat
            bersebelahan
        if (length > 1) {
            size_t i;
            for (i = 1; i < length; i++) {
                if ((input[i] == '+' || input[i] == '-') && (input[i-1] == '+' || input[i-1] ==
                    '-')) {
                    return syntaxErrorResult(i);
                }
            }
        }
    #endif

    EvalResult res = expression(input, &pos, length);

    // Menggagalkan evaluasi apabila masih ada karakter yang belum terproses atau berlebih
    if (pos != length) {
        return syntaxErrorResult(pos);
    }
    return res;
}
```

4.4 evaluator.h

```
#ifndef EVALRESULT_H
#define EVALRESULT_H
```

```
#include <stdlib.h>

typedef enum {
    INTEGRAL, FRACTIONAL, SYNTAX_ERROR, DIVISION_ERROR
} EvalResultType;

typedef struct {
    EvalResultType resultType;
    double fractionalValue;
    long long integralValue;
    int errorPos;
} EvalResult;

#include "evalresult.h"

inline unsigned char isError(EvalResult res);

inline EvalResult integralResult(long long value);

inline EvalResult fractionalResult(double value);

inline EvalResult syntaxErrorResult(size_t pos);

inline EvalResult divisionErrorResult();

EvalResult add(EvalResult op1, EvalResult op2);

EvalResult subtract(EvalResult op1, EvalResult op2);

EvalResult multiply(EvalResult op1, EvalResult op2);

EvalResult divide(EvalResult op1, EvalResult op2);

#endif
```

4.5 evalresult.c

```
#include "evalresult.h"

inline unsigned char isError(EvalResult res) {
    return res.resultType == SYNTAX_ERROR || res.resultType == DIVISION_ERROR;
}

inline EvalResult integralResult(long long value) {
    EvalResult res;
    res.resultType = INTEGRAL;
    res.integralValue = value;
    return res;
}

inline EvalResult fractionalResult(double value) {
    EvalResult res;
    res.resultType = FRACTIONAL;
    res.fractionalValue = value;
    return res;
}

inline EvalResult syntaxErrorResult(size_t pos) {
    EvalResult res;
    res.resultType = SYNTAX_ERROR;
    res.errorPos = pos;
    return res;
}

inline EvalResult divisionErrorResult() {
    EvalResult res;
    res.resultType = DIVISION_ERROR;
```

```
    return res;
}

EvalResult add(EvalResult op1, EvalResult op2) {
    if (isError(op1)) return op1;
    if (isError(op2)) return op2;

    if (op1.resultType == INTEGRAL && op2.resultType == INTEGRAL) {
        return integralResult(op1.integralValue + op2.integralValue);
    } else if (op1.resultType == FRACTIONAL && op2.resultType == INTEGRAL) {
        return fractionalResult(op1.fractionalValue + (double) op2.integralValue);
    } else if (op1.resultType == INTEGRAL && op2.resultType == FRACTIONAL) {
        return fractionalResult((double) op1.integralValue + op2.fractionalValue);
    } else {
        return fractionalResult(op1.fractionalValue - op2.fractionalValue);
    }
}

EvalResult subtract(EvalResult op1, EvalResult op2) {
    if (isError(op1)) return op1;
    if (isError(op2)) return op2;

    if (op1.resultType == INTEGRAL && op2.resultType == INTEGRAL) {
        return integralResult(op1.integralValue - op2.integralValue);
    } else if (op1.resultType == FRACTIONAL && op2.resultType == INTEGRAL) {
        return fractionalResult(op1.fractionalValue - (double) op2.integralValue);
    } else if (op1.resultType == INTEGRAL && op2.resultType == FRACTIONAL) {
        return fractionalResult((double) op1.integralValue - op2.fractionalValue);
    } else {
        return fractionalResult(op1.fractionalValue - op2.fractionalValue);
    }
}

EvalResult multiply(EvalResult op1, EvalResult op2) {
    if (isError(op1)) return op1;
    if (isError(op2)) return op2;

    if (op1.resultType == INTEGRAL && op2.resultType == INTEGRAL) {
        return integralResult(op1.integralValue * op2.integralValue);
    } else if (op1.resultType == FRACTIONAL && op2.resultType == INTEGRAL) {
        return fractionalResult(op1.fractionalValue * (double) op2.integralValue);
    } else if (op1.resultType == INTEGRAL && op2.resultType == FRACTIONAL) {
        return fractionalResult((double) op1.integralValue * op2.fractionalValue);
    } else {
        return fractionalResult(op1.fractionalValue * op2.fractionalValue);
    }
}

EvalResult divide(EvalResult op1, EvalResult op2) {
    if (isError(op1)) return op1;
    if (isError(op2)) return op2;
    if (op2.resultType == INTEGRAL && op2.integralValue == 0) return divisionErrorResult();
    if (op2.resultType == FRACTIONAL && (op2.fractionalValue == 0.0 || op2.fractionalValue ==
         -0.0)) return divisionErrorResult();

    if (op1.resultType == INTEGRAL && op2.resultType == INTEGRAL) {
        if (op1.integralValue % op2.integralValue == 0) {
            return integralResult(op1.integralValue / op2.integralValue);
        } else {
            return fractionalResult((double) op1.integralValue / (double) op2.integralValue);
        }
    } else if (op1.resultType == FRACTIONAL && op2.resultType == INTEGRAL) {
        return fractionalResult(op1.fractionalValue / (double) op2.integralValue);
    } else if (op1.resultType == INTEGRAL && op2.resultType == FRACTIONAL) {
        return fractionalResult((double) op1.integralValue / op2.fractionalValue);
    } else {
        return fractionalResult(op1.fractionalValue / op2.fractionalValue);
    }
}
```

4.6 stringutils.h

```
#ifndef STRINGUTILS_H
#define STRINGUTILS_H

#include <stdio.h>

/* Operasi yang mengakibatkan string membesar secara otomatis akan mengalokasikan string
   dalam blok-blok dengan kapasitas ini. */
#define STRING_ALLOC_BLOCK_SIZE 16

/* Baca file dari posisi saat ini hingga terbaca sebuah karakter delimiter.
   String yang dihasilkan dialokasikan secara dinamis. Mengembalikan NULL jika alokasi gagal. */
char* StringUtils_scan(FILE *fin, const char delim);

/* Dealokasi string */
void StringUtils_deallocate(char *str);

/* Perbandingan string yang tidak memperhatikan huruf besar/kecil,
   mengembalikan nilai < 0 jika karakter pertama yang tidak cocok di str1 < str2,
   0 jika kedua string sama, atau > 0 jika tidak keduanya. */
int StringUtils_strcmpi(const char *str1, const char *str2);

#endif
```

4.7 stringutils.c

```
#include "stringutils.h"
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* Baca file dari posisi saat ini hingga terbaca sebuah karakter delimiter.
   String yang dihasilkan dialokasikan secara dinamis. Mengembalikan NULL jika alokasi gagal. */
char* StringUtils_scan(FILE *fin, const char delim) {
    char inp;
    size_t len = 0;
    size_t capacity = STRING_ALLOC_BLOCK_SIZE;
    char *str = malloc(sizeof(char)*capacity);
    if (!str) return str;

    while (1) {
        inp = fgetc(fin);
        if (inp == EOF || inp == delim) break;
        str[len] = inp;
        len++;

        // Perbesar kapasitas string jika ukuran akan melebihi kapasitas
        if (len == capacity) {
            str = realloc(str, sizeof(char) * (capacity + STRING_ALLOC_BLOCK_SIZE));
            if (!str) return str;
            capacity += STRING_ALLOC_BLOCK_SIZE;
        }
    }

    // Set karakter kosong di akhir
    str[len] = 0;

    // Ubah kapasitas string menjadi sesuai dengan ukurannya
    return realloc(str, sizeof(char) * (len + 1));
}

/* Dealokasi string */
void StringUtils_deallocate(char *str) {
    free(str);
}

/* Perbandingan string yang tidak memperhatikan huruf besar/kecil,
```

```
   mengembalikan nilai < 0 jika karakter pertama yang tidak cocok di str1 < str2,
   0 jika kedua string sama, atau > 0 jika tidak keduanya. */
int StringUtils_strcmpi(const char *str1, const char *str2) {
    const char *p1 = str1;
    const char *p2 = str2;
    char c1, c2;
    do {
        c1 = (char) tolower((int) *p1);
        c2 = (char) tolower((int) *p2);
        p1++;
        p2++;
    } while (c1 == c2 && c1 != 0);
    return c1-c2;
}
```

5 Contoh Interaksi dengan Program

Berikut adalah contoh interaksi pengguna dengan program. Input pengguna digarisbawahi.

```
EVALUATOR EKSPRESI ARITMATIKA
=============================
Tugas 2 IF2220 Teori Bahasa Formal dan Otomata, November 2016
13515001 - Jonathan Christopher
13515079 - Nicholas Thie
Masukkan ekspresi aritmatika atau [exit] untuk keluar.

>> (-457.01+1280)*(35.7-11.0233)/(-6.1450)
-3304.910876
>> 3*+-12/(57)
Syntax error at position 4.
>> (5)
5
>> 3/0
Division by zero error.
>> (-3*5)+2.033*5.21
-4.408070
>> ((12+23)
Syntax error at position 9.
>> 5/3
1.666667
>> 6/3
2
>> 6/3.0
2.000000
```

Referensi

[1] Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2013). *Introduction to Automata Theory, Languages, and Computation (3rd ed.)*. Pearson.