Penerapan *Pattern Matching* dan *Regular Expression* dalam Pembuatan *Linter* TypeScript Sederhana

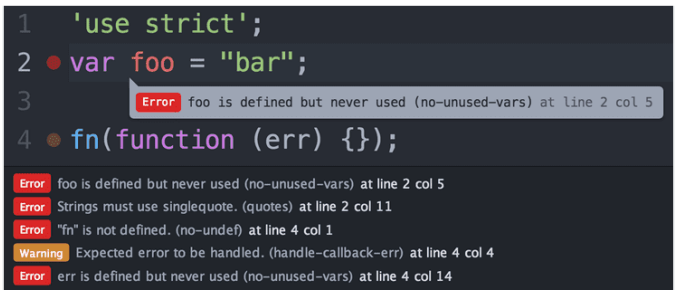
Jimly Firdaus - 13521102
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13521102@std.stei.itb.ac.id

***Abstract*—*Linter* merupakan alat yang sudah menjadi bagian penting dalam dunia pemrograman. *Linter* digunakan untuk membantu *developer* mendeteksi *bug* dan *error* dalam proses *development software* secara lokal saat sedang tahap *developing*. *Linter* dapat dibuat dengan menggunakan *string matching* atau *regular expression* untuk mendeteksi kode program yang umum yang dapat menyebabkan *bug* atau *error*.**

Keywords—string matching, regex, linter, typescript

I. Pendahuluan

Linter adalah alat yang biasa digunakan oleh *developer* untuk melakukan analisis *static* pada kode program yang dibuat dengan tujuan menemukan masalah atau potensi bagian kode program yang bisa menimbulkan *bug* maupun *error* dengan cepat. *Linter* digunakan oleh *developer* untuk meningkatkan ketepatan, kemananan, performa, serta kegunaan kode program yang dibuat.



Gambar 1.1 Contoh pemanfaatan *linter* dalam *code editor*

Sumber: https://blog.theodo.com/2017/05/linters/

Linter pada umumnya disediakan dalam bentuk *plugin* seperti *plugin* dalam IDE *Visual Studio Code*. Ada banyak sekali *plugin linter* yang tersedia untuk berbagai macam bahasa pemrograman seperti Java, C, TypeScript, dan masih banyak yang lain. Sebagai contoh, untuk bahasa pemrograman C++, tersedia *linter* yaitu *IntelliSense* yang dapat digunakan untuk membantu *developer* mencari dan menemukan masalah umum / *bug* yang umum terjadi pada bahasa pemrograman C++ tepat di dalam *Visual Studio*. Untuk bahasa pemrograman yang dibahas pada lingkup makalah ini yaitu TypeScript, terdapat *linter* yang sudah tersedia dan digunakan secara umum yaitu *ESlint* untuk membantu pendeteksian kode program secara *static* saat *development*.

Secara umum, *linter* dibuat dengan beberapa aturan-aturan dan program yang dapat membaca dan menganalisis kode sesuai dengan aturan-aturan yang telah ditentukan. Aturan-aturan tersebut mencakup pengecekan sintaksis, gaya penulisan kode, dan *best practices* dalam bahasa pemrograman yang ditujukan oleh *linter*. Pada makalah ini, pembuatan *linter* hanya akan berfokus pada penggunaan *string matching* dan *regex* dalam mendeteksi sintaksis yang *error* ataupun *bug* pada kode program.

II. Dasar Teori

A. *Linter*

Linter merupakan alat yang digunakan untuk menganalisis *source code* dan menandai *bug* atau *error* yang ditemukan. *Linter* dapat membantu *developer* menemukan dan memperbaiki masalah-masalah seperti *error* sintaksis, gaya penulisan kode yang tidak konsisten, dan *violation* terhadap *best practices* dalam bahasa pemrograman yang digunakan. Dengan menggunakan linter, *developer* dapat meningkatkan kualitas kode program yang mereka kembangkan dan terhindar dari kode program yang *buggy*.

Linter bekerja dengan membaca *source code* dan memeriksanya sesuai dengan aturan-aturan yang telah ditentukan. Aturan-aturan tersebut mencakup pengecekan sintaksis (proses memeriksa kode untuk memastikan bahwa kode tersebut mengikuti aturan sintaksis bahasa pemrograman yang digunakan), gaya penulisan kode, dan best practices dalam bahasa pemrograman yang digunakan. Jika *linter* menemukan pelanggaran terhadap aturan-aturan tersebut, maka *linter* akan menandai masalah tersebut dan juga dapat memberikan informasi tentang cara memperbaikinya.

Ada banyak *linter* yang tersedia untuk berbagai bahasa pemrograman dalam *development environment*. Beberapa linter populer antara lain *ESLint* untuk JavaScript dan TypeScript, *Pylint* untuk Python, dan *RuboCop* untuk Ruby. *Linter-linter* ini dapat di-*setting* / dikonfigurasi oleh pengguna untuk menyesuaikan aturan-aturan yang digunakan sesuai dengan kebutuhan mereka. Dengan menggunakan *linter*, *developer*

dapat meningkatkan produktivitas mereka dan menghasilkan kode program yang lebih berkualitas.

B. Bahasa Pemrograman TypeScript

Typescript adalah bahasa pemrograman yang dikembangkan oleh Microsoft dan merupakan *superset* dari JavaScript. Bahasa ini dibuat dan dirancang untuk mengatasi kelemahan utama JavaScript, yaitu pengelolaan *type* data yang lemah (*weakly-typed*). Dalam JavaScript, *type* data suatu variabel dapat berubah-ubah dengan mudah, sehingga dapat berpotensi menyebabkan *error* maupun *bug* saat menjalankan program. TypeScript memperkenalkan *type* data statis, yang memungkinkan *developer* untuk mendeklarasikan *type* data dari setiap variabel, parameter, dan fungsi dalam kode program mereka. Hal ini membantu mendeteksi kesalahan pada saat *compile time* dan meningkatkan kualitas kode secara keseluruhan.

TypeScript juga mendukung konsep pengembangan yang berorientasi objek yang kuat. *Developer* dapat membuat berbagai kelas, *interfaces*, dan *inheritance*, serta menggunakan konsep lain seperti *polimorphism* maupun *encapsulation*. Dengan adanya ini, TypeScript memungkinkan *developer* untuk mengorganisir dan mengelola kode program mereka dengan lebih baik, sehingga mempermudah *maintenance* dan *development* aplikasi yang lebih kompleks. Selain itu, TypeScript juga mendukung fitur-fitur terbaru JavaScript seperti *arrow functions*, *destructuring*, dan *async/await*, sehingga memperluas kemampuan dalam proses *development* dan memperkuat integrasi dengan ekosistem JavaScript yang sudah ada.

TypeScript juga dapat memberikan dukungan penuh dalam proses *development* aplikasi skala besar. Bahasa ini memungkinkan *developer* untuk memisahkan / memfaktor kode menjadi modul-modul yang terpisah, yang dapat dikelompokkan berdasarkan fungsionalitas atau per-komponen tertentu. Dengan modul-modul ini, *developer* dapat dengan mudah mengatur struktur *project*, membatasi *visibility* variabel dan fungsi, serta mengelola *dependency* antar modul. Selain itu, TypeScript juga memberikan *developer* kemampuan untuk mendefinisikan *type* khusus dan menyatukan beberapa *type* data yang berbeda. Hal ini membantu mencegah kesalahan pada proses *development*, sehingga mempercepat proses *debugging* dan meningkatkan kualitas aplikasi yang dihasilkan.

```typescript
export interface MessageInterface extends Message {
  responseStatusCode: number;
  responded: boolean;

  getId(): number;
  getStatus(): boolean;
  getText(): string;
  getResponseMsg(): string;
  getResponseCode(): number;
}

export class Message implements MessageInterface {
  id: number;
  sent: boolean;
  text: string;
  response: string;
  responseStatusCode: number;
  sentTime: string;
  responded: boolean;

  constructor(
    id: number,
    sent: boolean,
    text: string,
    sentTime: string,
  ) {
    this.id = id;
    this.sent = sent;
    this.text = text;
    this.response = "";
    this.responseStatusCode = 0;
    this.sentTime = sentTime;
    this.responded = false;
  }
```

Gambar 2.1 Penggunaan *Interface* dan Kelas dalam TypeScript

Sumber: https://github.com/Jimly-Firdaus/Tubes3_13521102

C. Pattern Matching

Pattern matching adalah teknik untuk mencocokkan pola tertentu dalam data dengan pola yang telah ditentukan sebelumnya. *Pattern matching* dapat digunakan dalam berbagai bidang seperti pemrosesan teks, pencarian data, dan analisis data. *Pattern matching* dapat digunakan untuk mencocokkan *string* dengan pola tertentu menggunakan *regular expression* atau algoritma pencocokan *string* seperti KMP (*Knuth-Morris-Pratt*) dan Boyer-Moore.

1. *Algoritma Brute Force*

Algoritma *Brute Force* bekerja dengan membandingkan setiap karakter dalam teks dengan setiap karakter dalam pola secara berurutan. Jika terdapat kecocokan, maka pencocokan dilanjutkan ke karakter selanjutnya dalam pola. Jika semua karakter dalam pola cocok dengan karakter dalam teks, maka pola ditemukan. Jika terdapat ketidakcocokan, maka pencocokan dimulai lagi dari awal pola pada karakter selanjutnya dalam teks.

Teks: NOBODY NOTICED HIM
Pattern: NOT

```
  NOBODY NOTICED HIM
1 NOT
2  NOT
3   NOT
4    NOT
5     NOT
6      NOT
7       NOT
8        NOT
```

Gambar 2.2 Ilustrasi pencocokan *string* dengan algoritma *Brute Force*
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

Brute Force in Java

Return index where pattern starts, or -1

```
public static int brute(String text,String pattern)
{ int n = text.length();     // n is length of text
  int m = pattern.length(); // m is length of pattern
  int j;
  for(int i=0; i <= (n-m); i++) {
    j = 0;
    while ((j < m) && (text.charAt(i+j)== pattern.charAt(j)))
    {
        j++;
    }
    if (j == m)
        return i;    // match at i
  }
  return -1;    // no match
 }// end of brute()
```

Gambar 2.3 Contoh kode program algoritma *Brute Force* dalam bahasa Java
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

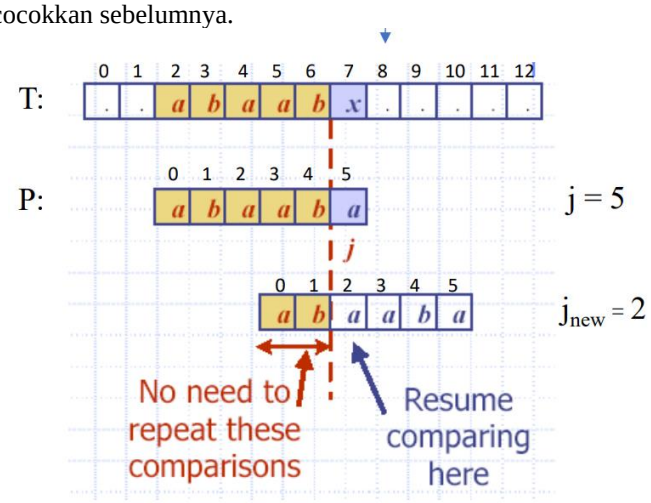
Kompleksitas algoritma *Brute Force* pada *worst case scenario* adalah dimana terjadi perbandingan sebanyak m kali (m adalah panjang pola) dan terjadi sebanyak n kali (n adalah panjang teks) yaitu sebesar O(m * n)

Pada *best case scenario*, besar kompleksitasnya adalah O(n) yang terjadi bila karakter pertama dari pola tidak pernah sama dengan karakter teks yang sedang dicocokan.

Pada *average case scenario*, besar kompleksitasnya adalah O(m + n). Algoritma *Brute Force* umumnya cepat ketika *range* alphabet dari teks besar (contohnya A-Z) dan sebaliknya jika *range* alphabet dari teks kecil (contohnya binary).

2. *Algoritma KMP* (*Knuth-Morris-Pratt*)

Algoritma KMP (Knuth-Morris-Pratt) adalah algoritma pencocokan *string* yang lebih efisien dibandingkan dengan algoritma *brute force*. Algoritma ini menggunakan teknik yang disebut "prefix function" atau "border function" untuk menghindari pencocokan ulang karakter yang sudah dicocokkan sebelumnya.



Gambar 2.4 Ilustrasi algoritma KMP
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

Dalam algoritma KMP, pertama-tama dihitung *border function* dari pola yang ingin dicocokkan. *Border function* ini menunjukkan panjang maksimum dari sub-pola yang sama antara awal dan akhir pola pada setiap posisi dalam pola. Kemudian, pencocokan dilakukan dengan membandingkan karakter dalam teks dengan karakter dalam pola secara berurutan. Jika terdapat kecocokan, maka pencocokan dilanjutkan ke karakter selanjutnya dalam pola. Jika terdapat ketidakcocokan (*mismatch*), maka pencocokan tidak dimulai lagi dari awal pola tetapi dari posisi yang ditunjukkan oleh *border function*.

(k = j-1)

j	0	1	2	3	4	5	6	7	8	9
P [j]	a	b	a	b	a	b	a	b	c	a
k	0	1	2	3	4	5	6	7	8	
b[k]	0	0	1	2	3	4	5	6	0	

Gambar 2.5 Ilustrasi *border function* (b[k]) pada algoritma KMP
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

Dengan menggunakan *border function*, algoritma KMP dapat menghindari pencocokan ulang karakter yang sudah dicocokkan sebelumnya dan meningkatkan efisiensi pencocokan pola.

Dalam algoritma KMP, terjadi perhitungan fungsi pinggiran sebesar O(m) dengan m adalah panjang pola, pencarian *string* sebesar O(n) dengan n adalah panjang teks sehingga kompleksitas algoritma KMP adalah O(m + n). Algoritma KMP sendiri tidak bagus seiring dengan

bertambahnya *range* dari alphabet pada teks karena *chance mismatch* menjadi lebih tinggi.

3. *Algoritma BoyerMoore*

Algoritma *BoyerMoore* adalah algoritma pencocokan *string* dengan dua teknik:

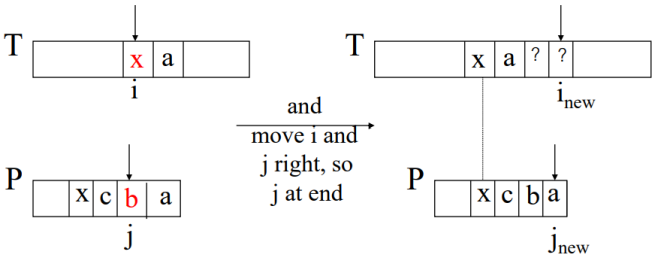
1. *Looking-glass technique*

   Pencocokan *string* dimulai dari akhir pola hingga mencapai awal pola.

2. *Character-jump technique*

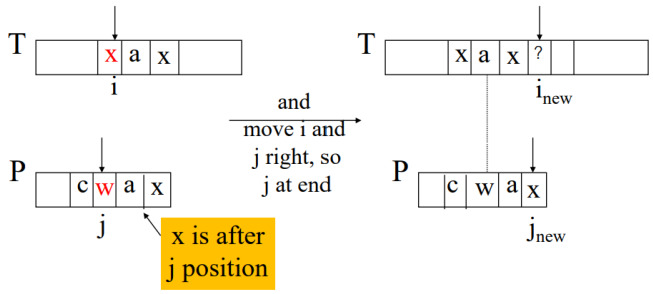
   Pada saat terdapat *mismatch* pada teks dengan indeks ke – i dan karakter pada pola yang ke – j tidak sama dengan karakter teks pada indeks ke ke – i. Terdapat 3 macam *mismatch cases:*

   1. Jika *mismatch* terjadi pada suatu karakter x pada teks dan x terdapat pada bagian kiri dari posisi perbandingan terakhir pada pola maka *shift right* pola hingga terjadi karakter x pada pola berada pada posisi perbandingan yang sama dengan karakter x pada teks.



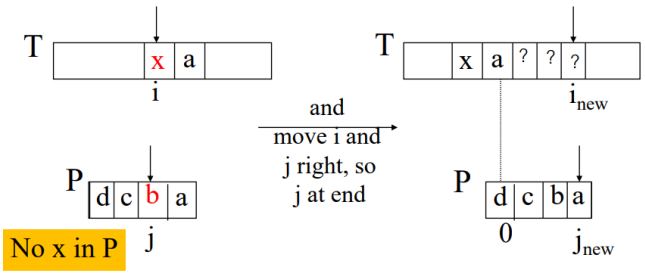
Gambar 2.6 Ilustrasi *mismatch* pada *case* 1
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

   2. Jika *mismatch* terjadi pada suatu karakter x pada teks dan x terdapat pada bagian kanan dari posisi perbandingan terakhir pada pola maka *shift right* pola sebanyak 1.



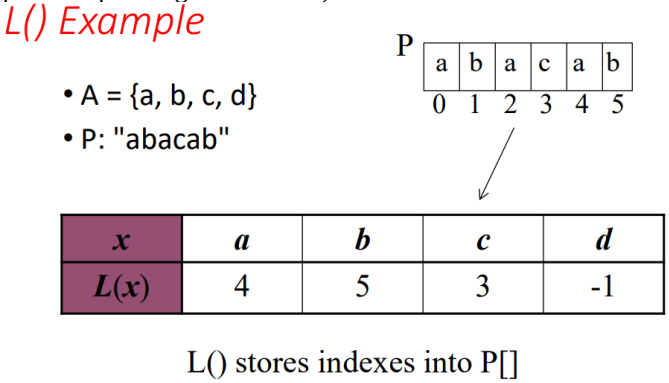
Gambar 2.7 Ilustrasi *mismatch* pada *case* 2
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

   3. Jika *mismatch* terjadi pada suatu karakter x pada teks dan x tidak terdapat pada pola maka *shift right* pola hingga indeks ke – 0 pola berada pada indeks ke – i + 1 pada teks. Ini disebut sebagai *character-jump*.



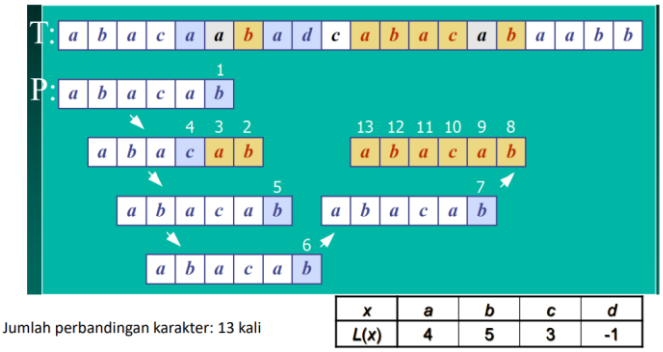
Gambar 2.8 Ilustrasi *mismatch* pada *case* 3
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

Dalam algoritma *BoyerMoore*, digunakan *last occurrence function* untuk menentukan *case-case* yang diperhadapkan jika terjadi *mismatch*. *Last occurrence function* dibuat dengan cara mencari indeks terakhir karakter tersebut ditemukan pada pola kemudian dibuat dalam bentuk tabel (seperti halnya *border function* pada algoritma KMP).



Gambar 2.9 Ilustrasi *last occurrence function*
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

Pada algoritma *BoyerMoore*, kompleksitas waktu untuk *worst case scenario* adalah O(mn + A). Algoritma *BoyerMoore* cepat pada *range* alphabet yang besar dan sebaliknya.



x	a	b	c	d
L(x)	4	5	3	-1

Gambar 2.10 Ilustrasi algoritma *BoyerMoore*
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

III. Penerapan Pattern Matching pada Linter TypeScript

Pattern Matching yang akan digunakan adalah *string matching* dan *regular expression*. Untuk *syntax* yang akan di-*lint* pada makalah ini hanya sebatas pendefinisian dan deklarasi variabel dan fungsi karena hanya sebagai gambaran bagaimana aplikasi *string matching* dan *regular expression* dapat digunakan dalam pembuatan *linter* bahasa pemrograman typescript.

A. *Pattern pada String Matching dan Regex*

Pada makalah ini, pola yang akan digunakan untuk melakukan pencocokan terhadap *syntax* typescript dibatasi pada:

1. *Keyword* variabel (diikuti *type*-nya) : *const, let, var*
2. *Syntax* fungsi : *const (arrow function), function*

Untuk *regex*, akan digunakan untuk menyaring *syntax* typescript untuk diproses dengan *string matching algorithm* (KMP atau *BoyerMoore*).

B. *Penggunaan Algoritma BoyerMoore*

Algoritma *BoyerMoore* digunakan untuk mencocokan *syntax*, apakah pendeklarasian variabel atau fungsi. Algoritma *BoyerMoore* yang digunakan sama dengan yang sudah ada pada bagian dasar teori mengenai algoritma *BoyerMoore*.

Terdapat sedikit modifikasi pada bagian fungsi *Last Occurrence*, dari penggunaan *array* biasa menjadi pengunaan struktur data *Map* untuk mempercepat proses *look-up* saat terjadi *mismatch* pada proses pencocokan *string*.

```
const lastOccurrence = (pattern: string): Map<string, number> => {
    let last = new Map<string, number>();
    for (let i = 0; i < pattern.length; i++) {
        last.set(pattern[i], i);
    }
    return last;
}
```

Gambar 3.1 Kode program untuk *last occurrence function* dalam bahasa pemrograman TypeScript

Sumber: Dokumen pribadi penulis

Berikut *pseudocode* algoritma *BoyerMoore* yang digunakan:

```
function boyerMoore(text, pattern)
  last = lastOccurrence(pattern)
  n = length of text
  m = length of pattern
  i = m - 1
  if i > n - 1
    return -1
  end if
  j = m - 1
  while i <= n - 1
    if pattern[j] == text[i]
      if j == 0
        if n == m
          return -2
        else
          return i
        end if
      else
        i = i - 1
        j = j - 1
      end if
    else
      lo = last.get(text[i]) or -1
      i = i + m - min(j, 1 + lo)
      j = m - 1
    end if
  end while
  return -1
end function
```

C. *Penggunaan Regular Expression*

Regular Expression digunakan untuk mengecek kebenaran *syntax* dari *block* kode program apakah sudah sesuai dengan kaidah pemrograman bahasa TypeScript. *Regular Expression* dibatasi pada pengecekan untuk pendeklarasian variabel (*const, var, let*) dan pendefinisian fungsi (pendefinisian secara umum dan *arrow function*). Berikut adalah *pattern regular expression* yang digunakan:

1. Untuk pengecekan *keyword const* pada deklarasi variabel: const [a-zA-Z_$][a-zA-Z_$0-9]* *: *[a-zA-Z_$][a-zA-Z_$0-9]* *= *.+;?

2. Untuk pengecekan *keyword let* pada deklarasi variabel: let [a-zA-Z_$][a-zA-Z_$0-9]* *: *[a-zA-Z_$][a-zA-Z_$0-9]* *= *.+;?
3. Untuk pengecekan *keyword var* pada deklarasi variabel: var [a-zA-Z_$][a-zA-Z_$0-9]* *: *[a-zA-Z_$][a-zA-Z_$0-9]* *= *.+;?
4. Untuk pengecekan *syntax* pendefinisian fungsi secara umum dalam bahasa pemrograman TypeScript: function [a-zA-Z_$][a-zA-Z_$0-9]*\([a-zA-Z_$][a-zA-Z_$0-9]*(*: *[a-zA-Z_$][a-zA-Z_$0-9]*)?(, *[a-zA-Z_$][a-zA-Z_$0-9]*(*: *[a-zA-Z_$][a-zA-Z_$0-9]*)?)*\) *: *[a-zA-Z_$][a-zA-Z_$0-9]* *{[^}]*}
5. Untuk pengecekan *syntax* pendefinisian fungsi dalam bentuk *arrow function*: [a-zA-Z_$][a-zA-Z_$0-9]* *= *\([a-zA-Z_$][a-zA-Z_$0-9]*(*: *[a-zA-Z_$][a-zA-Z_$0-9]*)?(, *[a-zA-Z_$][a-zA-Z_$0-9]*(*: *[a-zA-Z_$][a-zA-Z_$0-9]*)?)*\) *: *[a-zA-Z_$][a-zA-Z_$0-9]* *=> *(?:[^{].+|{[^}]*return[^}]*})

Kumpulan *regular expression* tersebut akan digunakan saat algoritma *BoyerMoore* sudah memvalidasi apakah *block* kode program memang merupakan *syntax* bukan *regular string*.

D. *Process Syntax*

Untuk membantu pemrosesan *syntax*, terdapat fungsi *processSyntax* untuk memudahkan pencocokan *string* serta pengecekan kebenaran *syntax*. Berikut adalah *pseudocode* untuk fungsi *processSyntax*:

```
function processSyntax(text)
  if boyerMoore(text, "const") ≠ -1
    return constPattern.test(text) or arrowFuncPattern.test(text)
  end if
  if boyerMoore(text, "var") ≠ -1
    return varPattern.test(text) or arrowFuncPattern.test(text)
  end if
  if boyerMoore(text, "let") ≠ -1
    return letPattern.test(text) or arrowFuncPattern.test(text)
  end if
  if boyerMoore(text, "function") ≠ -1
    return funcDefPattern.test(text)
  end if
  { regular string }
  return true
end function
```

Fungsi *processSyntax* akan menerima *string* yang berupa isi dari file TypeScript yang ingin dilakukan pengecekan. Kemudian dengan algoritma *BoyerMoore*, akan ditentukan jenis *keyword* yang digunakan dalam *string* tersebut dan jika terjadi *matched*, maka akan dilanjutkan dengan pengecekan *syntax* dengan menggunakan *regular expression* yang telah dibuat sebelumnya.

E. *Hasil Pengetesan Program*

Untuk memudahkan penggunaan program, terdapat sebuah *main* program yang membantu *splitting* bagian *block* kode dalam satu file TypeScript. Berikut merupakan *main* program yang digunakan untuk membaca file TypeScript dan memproses isinya:

```
fs.readFile('./test/test.ts', 'utf8', (err, data: string) => {
    if (err) {
        console.error(err);
        return;
    }
    let test: string[] = data.split(/\r?\n(?![^{]*})/).map(line => line.trim());
    let counteredError: number = 0;
    const red = '\x1b[31m';
    const green = '\x1b[32m';
    const reset = '\x1b[0m';
    test.forEach((ele, index) => {
        ele = ele.trim();
        if (ele.length !== 0) {
            if (!processSyntax(ele)) {
                counteredError++;
                console.log(`${red}Invalid syntax on:\n ${ele}${reset}`);
            }
        }
    })
    if (counteredError === 0) {
        console.log(`${green}No Invalid Syntax!${reset}`);
    }
});
```

Gambar 3.2 Kode program untuk *main* program dalam bahasa pemrograman TypeScript

Sumber: Dokumen pribadi penulis

Program akan membaca sebuah file TypeScript, kemudian dilakukan *splitting* berdasarkan *newline* (dan *carriage return* jika ada). Kemudian hasil *splitting* akan diproses satu persatu dengan fungsi *processSyntax* yang sudah dibuat.

Untuk pengujian, akan digunakan file TypeScript seperti berikut:

```
const num: number = 9;

var num2:: number = 10;

let num3 : number = 11;


function kuadrat(x: number): number {
    return x * x;
}

function kuadrat+(x: number): number {
    return x * x;
}

const pangkat3 = (x:number): number => {
    return x * x * x
};

const pangkat_3(x:=number): number => {
    return x * x * x
};
```

Gambar 3.3 Kode program untuk pengujian

Sumber: Dokumen pribadi penulis

Hasil dari program yang dibuat adalah:

```
PS D:\Programming\Stima\Makalah> node dist/main.js
Invalid syntax on:
 var num2:: number = 10;
Invalid syntax on:
 function kuadrat+(x: number): number {
    return x * x;
}
Invalid syntax on:
 const pangkat_3(x:=number): number => {
    return x * x * x
};
```

Gambar 3.4 Hasil program

Sumber: Dokumen pribadi penulis

Dari hasil program, terdapat 3 *syntax error* pada file TypeScript yang digunakan. *Syntax error* yang pertama yaitu pada "*var num2::number = 10*" dimana terdapat dua tanda ":" pada penentuan *type* dari variabel tersebut. *Syntax error* yang kedua yaitu pada pendefinisian fungsi "kuadrat+" dimana terdapat "+" pada penamaan fungsi yang tidak diperbolehkan dalam bahasa pemrograman TypeScript. *Syntax error* ketiga terdapat pada pendefinisian fungsi "pangkat_3" dimana terdapat "=" setelah ":" untuk menentukan *type* dari argumen fungsi yang tidak diperbolehkan dalam bahasa pemrograman TypeScript dan juga diperlukan adanya "=" setelah nama fungsi sebagai untuk pendefinisian *arrow function*.

IV. KESIMPULAN

Algoritma *Pattern Matching*, khususnya algoritma *string matching* dan *regular expression* dapat digunakan untuk membuat suatu *linter* sederhana untuk bahasa pemrograman TypeScript. Pada dasarnya semua *linter* memiliki konsep yang sama namun hanya diimplementasikan untuk memenuhi kebutuhan bahasa pemrograman yang berbeda-beda. Menambahkan penggunaan algoritma lain selain *string matching* dan *regular expression* dalam implementasi *linter* juga akan memperkaya fungsionalitas dari *linter* yang dibuat.

UCAPAN TERIMA KASIH

Dengan selesainya makalah ini dibuat, saya mengucapkan rasa syukur sebesar-besarnya kepada Tuhan, atas hikmat yang diberikan-Nya untuk menyelesaikan makalah ini. Saya juga berterima kasih kepada ibu Dr. Nur Ulfa Maulidevi, S.T, M.Sc, bapak Dr. Ir. Rinaldi Munir, M.T, dan bapak Ir. Rila Mandala, M.Eng., Ph.D. selaku dosen yang membimbing saya dalam mata kuliah Strategi Algoritma.

REFERENCES

[1] https://blog.theodo.com/2017/05/linters/ diakses pada tanggal 20 Mei 2023.

[2] https://ichi.pro/id/linting-yang-lebih-baik-dengan-typecript-dan-eslint-kustom-86889192766547 diakses pada tanggal 20 Mei 2023.

[3] https://www.typescriptlang.org/ diakses pada tanggal 20 Mei 2023.

[4] https://www.microsoft.com/en-us/typescript diakses pada tanggal 20 Mei 2023.

[5] https://github.com/Jimly-Firdaus/Tubes3_13521102 diakses pada tanggal 20 Mei 2023.

[6] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf diakses pada tanggal 20 Mei 2023.

[7] https://github.com/Jimly-Firdaus/Simple-TypeScript-Linter diakses pada tanggal 21 Mei 2023

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 21 Mei 2023

Jimly Firdaus, 13521102