



FELIS

Implementasi CFG pada Pengenalan Sintaks Program
Sederhana

ABSTRAK

Felis adalah sebuah aplikasi yang ditulis dengan Java untuk mengenali sintaks suatu program dengan bahasa tertentu. Bahasa yang dikenali dapat berupa bahasa Felis, yang merupakan sebuah bahasa pemrograman sederhana dari program, atau dapat berupa bahasa lain yang telah disediakan CFG (Context-Free Grammar) dalam bentuk CNF (Chomsky Normal Form). Ketika terjadi kesalahan sintaks, Felis dapat mengenali salah satu posisi lokasi kesalahan yang mungkin.

Tito D. Kesumo Siregar (13511018)

Rifki Afina Putri (13511066)

IF-2052 Teori Bahasa dan Otomata

Daftar Isi

Daftar Isi	1
I. Pendahuluan.....	2
1. Deskripsi Permasalahan	2
2. Implementasi.....	3
3. Bonus	3
II. Solusi dan Analisis.....	4
1. Asumsi	4
2. Context-Free Grammar dari Bahasa Felis.....	4
3. Komentar.....	6
4. Pemroses	7
5. Algoritma CYK.....	7
6. Default Unit Production	8
III. Fitur-Fitur.....	9
1. Felis Editor.....	9
2. Grammar Editor	9
3. Parse Tree Viewer.....	10
4. Fitur Utama	10
a. Melakukan input kode pada teks editor	10
b. Memeriksa kode program	10
c. Menampilkan informasi baris yang menyebabkan kode salah.....	10
5. Fitur Bonus.....	11
a. Cats!	11
b. Parse Tree Viewer.....	11
c. Grammar Editor	12
6. Contoh Masukan dan Hasil.....	12
a. Contoh Program yang Benar.....	12
b. Contoh Program yang Salah	13
c. Contoh Program yang Benar dengan Parse Tree	13
IV. Potongan Kode Felis	14
1. Kode Algoritma CYK.....	14
2. Kode Prosesor File.....	16

I. Pendahuluan

1. Deskripsi Permasalahan

Terdapat sebuah kode program sederhana yang memiliki sintaks tertentu. Program tersebut hanya memiliki algoritma, tanpa pernyataan kamus atau judul program. Tipe variabel hanya *integer* dan operasi terhadap variabel hanya berupa operasi matematika untuk *integer*.

Berikut ini adalah fungsi-fungsi yang dimiliki oleh kode program:

Kode	Fungsi	Keterangan
begin..end	Batas awal dan akhir dari program.	Juga merupakan batas awal dan akhir suatu deretan perintah.
if <kondisi> then <perintah> if <kondisi> then <perintah-true> else <perintah-false>	Kondisional	Bagian else adalah opsional. Perlu menggunakan begin..end jika ada banyak perintah.
repeat <perintah> until <kondisi>	Pengulangan sampai kondisi tertentu	Tidak perlu menggunakan begin..end jika ada banyak perintah.
while <kondisi> do <perintah>	Pengulangan selagi tidak dalam kondisi tertentu	Perlu menggunakan begin..end jika ada banyak perintah.
a = b a <> b a > b a < b a <= b a >= b	Kondisi	Kondisi mirip dengan deretan perintah yang mengembalikan jenis data <i>boolean</i> di bahasa pemrograman lain. Kondisi dianggap dibedakan dengan ekspresi matematika.
V = X + Y V = X - Y V = X * Y	Assignment	Variabel hanya dapat di-assign berupa ekspresi atau operasi matematika, atau variabel lain.
input (V) output (E)	Input dan output	Input berupa variabel, output berupa ekspresi atau kondisi.
{ ... }	Komentar	Komentar diabaikan oleh program.

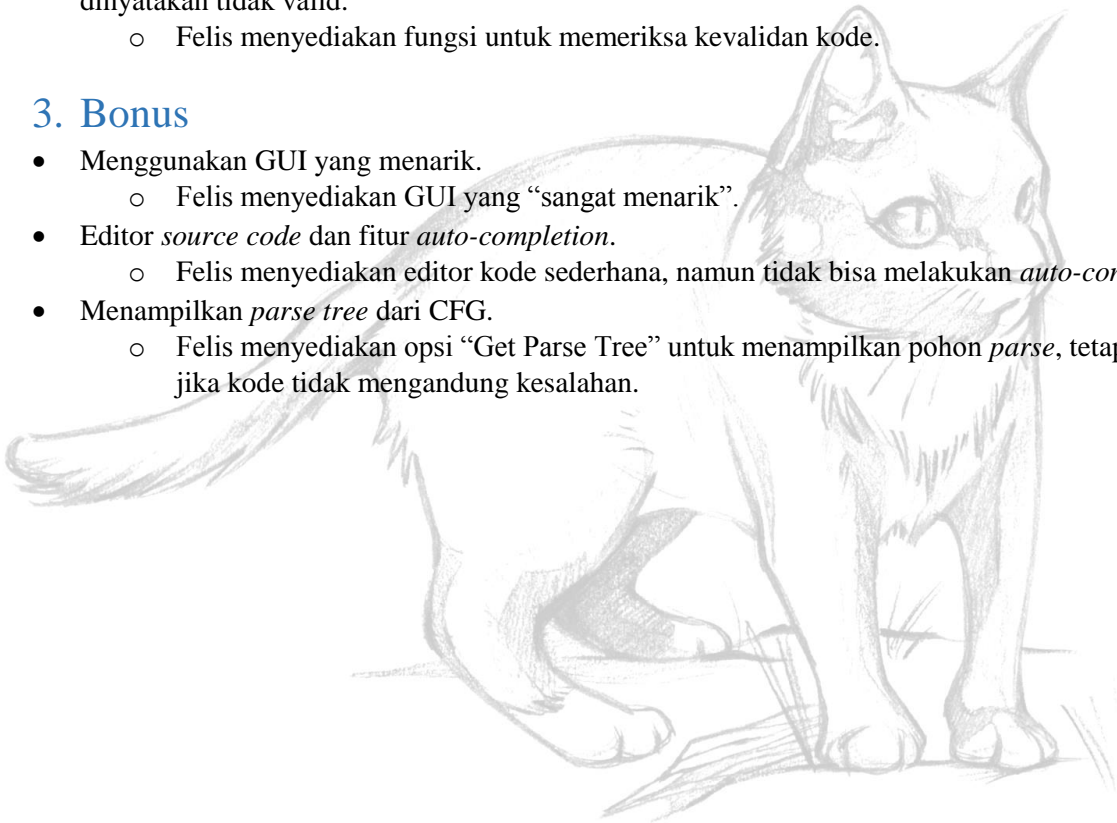
Berdasarkan deskripsi di atas, harus dibuat sebuah aplikasi untuk mengenali apakah kode sebuah program sesuai atau tidak. Aplikasi akan membuka file yang berisi sintaks program dan kemudian memeriksa apakah kode pada program tersebut valid atau tidak. Jika tidak valid maka aplikasi akan mengeluarkan informasi baris yang menyebabkan kode program tidak valid.

2. Implementasi

- Program dibuat dalam bahasa pemrograman prosedural atau *object-oriented* (C++, C atau Java).
 - Felis dibuat dalam bahasa Java dengan menggunakan IDE NetBeans.
- Aturan CFG (dalam bentuk CNF) disimpan di *file* terpisah menggunakan algoritma CYK.
 - Felis dapat membaca *file* tertentu menjadi tata bahasa, menyediakan editor tata bahasa, dan menyimpan tata bahasa ke dalam suatu *file* yang bernama `grammar.fls`.
- Masukan berupa teks (atau *file* teks) yang berisi program sederhana sesuai dengan aturan yang dimaksud.
 - Felis menyediakan editor teks untuk menuliskan kode, atau membuka file tertentu untuk dituliskan isinya ke editor teks.
- Keluaran berupa pernyataan valid atau tidak dan baris penyebab ketidakvalidan jika program dinyatakan tidak valid.
 - Felis menyediakan fungsi untuk memeriksa kevalidan kode.

3. Bonus

- Menggunakan GUI yang menarik.
 - Felis menyediakan GUI yang “sangat menarik”.
- Editor *source code* dan fitur *auto-completion*.
 - Felis menyediakan editor kode sederhana, namun tidak bisa melakukan *auto-completion*.
- Menampilkan *parse tree* dari CFG.
 - Felis menyediakan opsi “Get Parse Tree” untuk menampilkan pohon *parse*, tetapi hanya jika kode tidak mengandung kesalahan.



II. Solusi dan Analisis

1. Asumsi

Untuk melengkapi hal-hal yang belum dijelaskan di spesifikasi tugas besar tetapi perlu dijelaskan, dibuat beberapa asumsi. Berikut adalah daftar asumsi serta akibat dari asumsi tersebut:

Asumsi	Akibat
Terdapat ekspresi dan kondisi. Ekspresi adalah kumpulan variabel, angka, dan operator ekspresi yang memiliki nilai berupa bilangan. Kondisi adalah kumpulan variabel, angka, dan operator kondisi yang memiliki nilai berupa 'benar' atau 'salah' (nilai <i>boolean</i>).	Pada blok <code>if ... then</code> , bagian ... diisi oleh kondisi, bukan ekspresi. Hal ini mengakibatkan bentuk yang mirip dengan bahasa Pascal, yaitu <code>if x = y + 5 then</code> Bentuk seperti C tidak dimungkinkan, yaitu <code>if (1)</code> Begitu pula pada blok <code>repeat ... until ...</code> dan <code>while ... do</code> Selain itu, tidak dimungkinkan suatu variabel di- <i>assign</i> dengan sebuah kondisi.
Variabel dinyatakan sebagai deretan karakter alfanumerik (<code>a..z</code> , <code>A..Z</code> , atau <code>0..9</code>) yang diawali oleh karakter alfabet (<code>a..z</code> atau <code>A..Z</code>).	Deretan karakter seperti <code>inf0matika</code> dianggap variabel, sedangkan deretan karakter seperti <code>lnformatika</code> dan <code>123</code> tidak dianggap variabel.
Operator <i>assignment</i> dan operator kondisi kesamaan dibedakan.	Ditambahkan operator <i>assignment</i> yaitu <code>:=</code> , sedangkan operator kondisi kesamaan tetap <code>=</code> .

2. Context-Free Grammar dari Bahasa Felis

Untuk memeriksa sintaks dari bahasa Felis, digunakan Context-Free Grammar (CFG) yang telah diproses ke dalam bentuk Chomsky Normal Form (CNF). CFG tersebut kemudian diproses menggunakan algoritma Cocke-Younger-Kasami (CYK). Berikut adalah penjelasan dari aturan-aturan CFG yang digunakan untuk pemrosesan bahasa Felis:

Bagian Program yang Dikenali	Produksi-Produksi	Keterangan
Ekspresi	$OP-MATH \rightarrow + \mid - \mid *$	
	$LBRACE \rightarrow ($	
	$RBRACE \rightarrow)$	
	$EXP \rightarrow$ [NUMBER] [VARIABLE] $EXP \ EXP-TAIL$ $LBRACE \ EXP-BRACED-TAIL$	[NUMBER] dan [VARIABLE] adalah <i>unit production</i> yang dikenali di luar CFG.
	$EXP-TAIL \rightarrow OP-MATH \ EXP$	
	$EXP-BRACED-TAIL \rightarrow EXP \ RBRACE$	
Kondisi	$OP-COND \rightarrow$ $=== \mid =/= \mid < \mid <= \mid >= \mid >$	
	$LBRACE \rightarrow ($	Sama dengan yang digunakan untuk ekspresi.
	$RBRACE \rightarrow)$	Sama dengan yang digunakan untuk ekspresi.

	COND → EXP COND-TAIL LBRACE COND-BRACED-TAIL	Tidak dimungkinkan adanya bentuk seperti (x <= 5) > 2.
	COND-TAIL → OP-COND EXP	
	COND-BRACED-TAIL → COND RBRACE	
Input dan Output	L-IOBRACE → (Simbol dari IOBRACE dibedakan dengan BRACE untuk memudahkan jika ingin diganti.
	R-IOBRACE →)	Simbol dari IOBRACE dibedakan dengan BRACE untuk memudahkan jika ingin diganti.
	INPUT → input	
	OUTPUT → output	
	STATEMENT → INPUT INPUT-TAIL	
	INPUT-TAIL → L-IOBRACE INPUT-CONTENT	
	INPUT-CONTENT → [VARIABLE] R-IOBRACE	<i>Input</i> harus dimasukkan ke dalam sebuah variabel.
	STATEMENT → OUTPUT OUTPUT-TAIL	
	OUTPUT-TAIL → L-IOBRACE OUTPUT-CONTENT	
	OUTPUT-CONTENT → EXP R-IOBRACE	<i>Output</i> yang dikeluarkan dapat berupa ekspresi, misalnya output (5 + 3).
	OUTPUT-CONTENT → COND R-IOBRACE	<i>Output</i> yang dikeluarkan dapat berupa kondisi, misalnya output (z < 10).
Blok Program	BLOCK-BEGIN → begin	
	BLOCK-END → end	
	BLOCK → BLOCK-BEGIN BLOCK-END BLOCK-BEGIN BLOCK-REST	Satu blok bisa langsung berupa blok kosong (begin end).
	BLOCK-REST → STATEMENT BLOCK-REST STATEMENT BLOCK-END	
If-Then-Else	IF → if	
	THEN → then	
	ELSE → else	
	STATEMENT → IF IF-NEXT-1	
	IF-NEXT-1 → COND IF-NEXT-2	
	IF-NEXT-2 → THEN IF-NEXT-3 THEN STATEMENT	Bisa berupa if..then... atau if..then...else....
	IF-NEXT-3 → STATEMENT IF-NEXT-4	
	IF-NEXT-4 → ELSE STATEMENT	

Repeat-Until	REPEAT → repeat	
	REPEAT → until	
	STATEMENT → REPEAT REPEAT-NEXT-1	
	REPEAT-NEXT-1 → STATEMENT REPEAT-NEXT-1 STATEMENT REPEAT-NEXT-2	
	REPEAT-NEXT-2 → UNTIL COND	
While-Do	WHILE → while	
	DO → do	
	STATEMENT → WHILE WHILE-NEXT-1	
	WHILE-NEXT-1 → COND WHILE-NEXT-2	
	WHILE-NEXT-2 → DO STATEMENT	
Assignment	OP-ASSIGN → :=	
	STATEMENT → [VARIABLE] ASSIGN-TAIL	
	ASSIGN-TAIL → OP-ASSIGN EXP	

Secara formal, tata bahasa Felis adalah $G = (V, T, P, S)$, dengan V adalah deretan karakter yang sesuai dengan deskripsi produksi pada tabel di atas (di sebelah kiri tanda panah), T adalah deretan karakter yang sesuai dengan deskripsi unit pada tabel di atas (sendirian di sebelah kanan tanda panah), P adalah aturan produksi sesuai tabel, dan S adalah STATEMENT.

3. Komentar

Komentar didefinisikan sebagai `{ ... }`, dengan ... berisi string apapun. Komentar tidak diproses menggunakan algoritma CYK, melainkan dihapus ketika sebuah *file* dibaca dan diproses. Hal ini mengurangi beban dari algoritma CYK sendiri (banyak terminal-terminal yang harus diproses berkurang) dan menjaga tata bahasa tetap “bersih” (karena komentar bisa berada di banyak tempat), misalnya:

```
begin
  input ( catname ) { menerima nama kucing }
  if ( catname = 1 { 1 = angora } ) then
    begin { proses hanya jika angora }
      output ( angora + angora )
    { proses selesai } end
  else { selain angora, buang }
    output ( hapus )
end { selesai }
{ komentar iseng }
{ komentar iseng lagi }
{ komentar yang lari dari kenyataan }
```

Efek sampingnya, jika komentar tidak ditutup dengan benar (ada simbol `{` dan `}` yang tidak memiliki pasangan), kode program yang dibaca pemroses dapat berubah. Efek samping ini tidak bermasalah karena kode program yang ditulis dengan benar tidak terpengaruh.

4. Pemroses

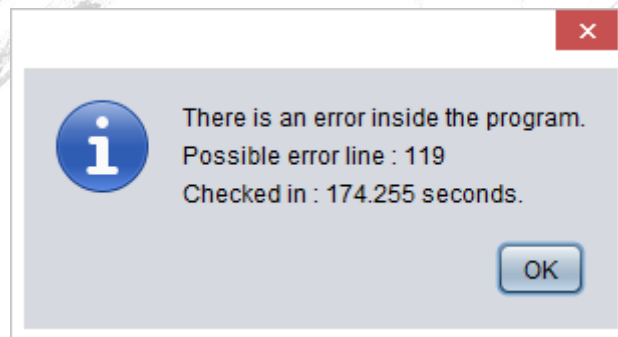
Sebelum file kode dimasukkan ke dalam algoritma CYK, terlebih dahulu file kode diproses menjadi *tuple* (S, T) di mana S adalah satu buah string pada kode dan T adalah angka baris kode di file kode. Untuk kode program berikut akan dihasilkan deretan *tuple* $(begin, 1), (input, 2), ((, 2), (hello, 2), (, 2), dan (end, 3)$:

```
begin
    input ( hello )
end
```

Pemroses yang disediakan tidak terlalu kuat sehingga tidak bisa memisahkan bentuk kode seperti `input (x)`. Akibatnya, setiap *string* pada kode program harus dipisahkan dengan *whitespace*.

5. Algoritma CYK

Untuk memproses masukan yang sudah dipecah-pecah, digunakan algoritma CYK, yang merupakan algoritma yang memanfaatkan teknik *dynamic programming*. Implementasi yang biasa digunakan adalah dengan menggunakan *array* tiga dimensi atau semacamnya untuk memoisasi, misalnya saja *pseudocode* CYK yang tersedia di Wikipedia. Namun, karena iseng, Felis menggunakan pohon biner seimbang. Akibatnya, karena program sering mengakses data di memo, Felis lebih lambat sebanyak $\log n$ kali. Hal ini akan terasa jika kode program terdiri dari banyak *token* (untuk program sebesar 130 baris, dibutuhkan waktu hingga 3 menit).



Gambar 1. Waktu untuk memeriksa sebuah program yang cukup besar (130 baris)

Hasil dari implementasi pada Felis adalah sebuah *parse tree* jika tidak ada kesalahan, atau *null* jika ada kesalahan. Perkiraan lokasi kesalahan disimpan dalam sebuah variabel terpisah yang bisa diakses. Disebut perkiraan lokasi karena yang disimpan adalah baris ketika algoritma CYK tidak dapat menemukan pasangan yang cocok untuk suatu blok tertentu. Untuk kode berikut, algoritma menemukan kesalahan justru pada baris ke-7, padahal kesalahan terletak di baris ke-4:

```
begin
    input ( x )
    if x = 5 then
        outfoot
        [
            5
        ]
end
```


6. Default Unit Production

Felis menyediakan beberapa *unit production* yang bisa digunakan dengan menggunakan tombol “Add Default Unit Production”. *Unit production* yang disediakan adalah *unit production* spesial yang menggunakan ekspresi reguler yang disediakan di API Java (`java.util.regex`). Beberapa nama *unit production* yang disediakan adalah:

Nama Unit Production	Keterangan
[NUMBER]	Deretan digit 0 sampai 9. Contoh: 010, 252, 0000163.
[WORD]	Deretan karakter alfabet, baik huruf besar maupun huruf kecil. Contoh: informatika, TeknikInformatika
[VARIABLE]	Deretan karakter alfanumerik dengan karakter pertama adalah alfabet, baik huruf besar maupun huruf kecil. Contoh: inf0rmatika, yyl, YzX2.
[ANY]	Deretan sembarang karakter. Contoh: G4!Bk3r3N beudh.



III. Fitur-Fitur

1. Felis Editor

Felis Editor merupakan editor teks untuk memasukkan kode yang akan divalidasi. Kode dapat dibuka dari *file* eksternal atau diketikkan sendiri. Kemudian, pengecekan sintaks dapat dilakukan dengan perintah Edit > Check atau Edit > Get Parse Tree. Jika kode program valid, maka akan muncul pesan sukses, sedangkan jika kode program tidak valid, maka akan muncul pesan false. Perintah Get Parse Tree juga akan menampilkan *parse tree* pada Parse Tree Viewer jika kode program sukses diperiksa.

Tata bahasa yang digunakan Felis Editor bersumber dari Grammar Editor.

2. Grammar Editor

Grammar Editor merupakan tempat untuk mengatur tata bahasa yang digunakan untuk melakukan validasi kode program. User dapat memasukkan tata bahasa Felis (tata bahasa sesuai spesifikasi) yang telah didefinisikan pada kode program dengan menekan tombol “Set Grammar To Felis Language”. Jika diinginkan, tata bahasa dapat diketikkan sendiri dengan menggunakan fitur “Add Unit Productions” dan “Add Non-Unit Productions”, atau tata bahasa dibaca dari *file* eksternal.

Bagian Productions menampilkan produksi-produksi yang sudah dimasukkan ke dalam tata bahasa.

Tombol Save dan Load dapat digunakan untuk menyimpan tata bahasa yang sudah ada ke dalam suatu *file* bernama `grammar.flis`, yang disimpan di tempat program. Tombol Reset dapat digunakan untuk mengosongkan tata bahasa.

Letak dari bagian untuk memasang *start symbol* atau *root* terdapat di bagian “Other”, bersama dengan tombol “Add Default Unit Productions” untuk menambahkan produksi-produksi spesial dari Felis.

Untuk menggunakan *file* eksternal sebagai tata bahasa (menggunakan “Load Grammar From File”), perlu diikuti aturan-aturan berikut:

- Produksi unit dalam bentuk:
`<Nama Simbol> -> <Nama Terminal>`.
- Produksi non-unit dalam bentuk:
`<Nama Simbol> -> <Non Terminal 1> <Non Terminal 2>`
- Setiap aturan produksi harus berada dalam satu baris sendiri.
- Karakter spasi digunakan sebagai pembatas antara simbol dan tanda panah, tanda panah dan simbol, serta simbol dan simbol.
- Tidak ada fungsi untuk menambahkan *start symbol*; fungsi tersebut harus dilakukan melalui GUI Editor.

Berikut adalah contoh sebuah *file* eksternal yang dapat dibaca oleh Grammar Editor:

```
X -> 0
X -> X X
Y -> 1
Y -> Y Y
S -> X Y
S -> Y X
```

3. Parse Tree Viewer

Parse Tree Viewer merupakan tempat untuk menampilkan Parse Tree dari suatu kode yang sukses divalidasi jika digunakan fitur “Get Parse Tree” di Felis Editor. Bentuk Parse Tree mirip seperti tampilan pohon *file* yang biasa dijumpai di program *file manager* seperti Total Commander atau Windows Explorer.

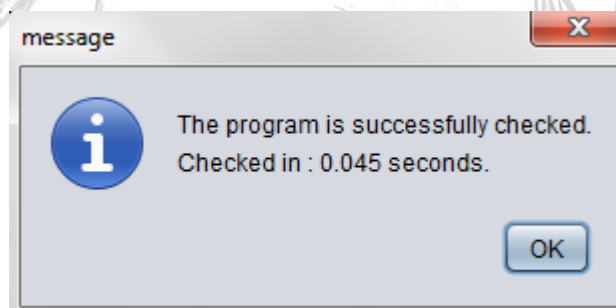
4. Fitur Utama

a. Melakukan input kode pada teks editor

Kode dimasukkan secara manual pada Felis Editor atau menggunakan File > Open.

b. Memeriksa kode program

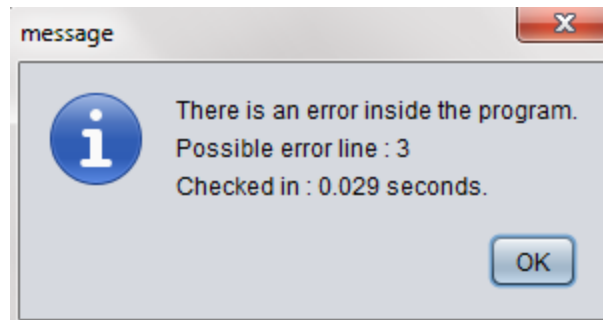
Program akan memeriksa apakah kode program valid atau tidak. Hasil pemeriksaan kemudian akan muncul sebagai suatu pesan. Selain itu, program juga menampilkan lama waktu pemeriksaan.



Gambar 2. Pesan jika program sukses diperiksa.

c. Menampilkan informasi baris yang menyebabkan kode salah

Apabila sintaks tidak valid, maka program akan menampilkan informasi baris kode yang mungkin salah. Informasi baris ini tidak terlalu akurat karena batasan dari *parse tree* dan akibat dari penggunaan algoritma CYK.

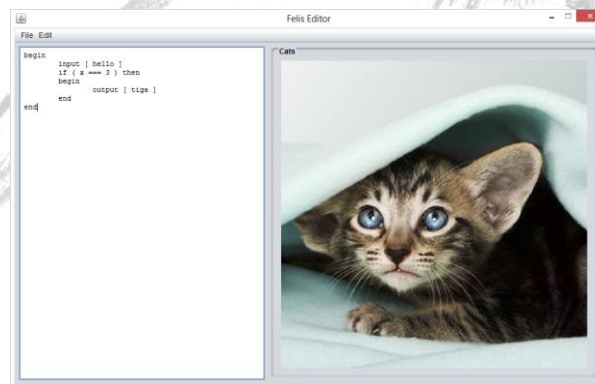


Gambar 3. Pesan jika program gagal diperiksa.

5. Fitur Bonus

a. Cats!

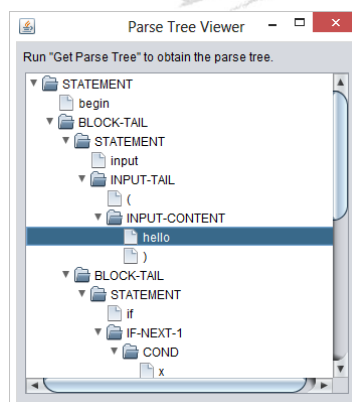
Siapa yang tidak tahan melihat kucing-kucing nan imut? Felis Editor menyediakannya tepat di samping kode, sehingga ketika pengguna mengetikkan kodenya di editor, kucing-kucing tersebut dapat menyejukkan pandangan dan membuat GUI menjadi menarik. Ditambah lagi, ada banyak gambar kucing yang disediakan (hingga 10 buah), sehingga pengguna tidak akan merasa bosan.



Gambar 4. Tampilan Felis Editor. Perhatikan kucing yang lucu.

b. Parse Tree Viewer

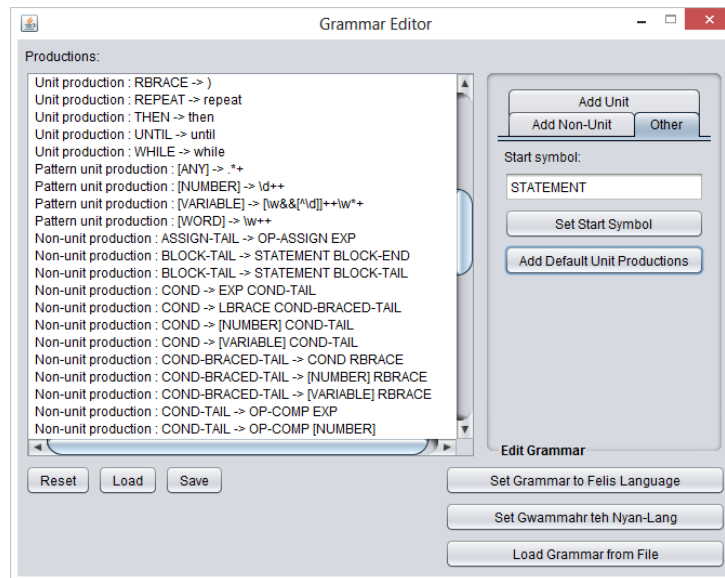
Parse Tree Viewer adalah spesifikasi bonus yang diimplementasikan pada Felis. Bentuk dari Parse Tree Viewer ini menggunakan fitur Java API “Tree”.



Gambar 5. Tampilan Parse Tree Viewer.

c. Grammar Editor

Grammar Editor memungkinkan pengguna untuk membuat tata bahasa, mengambil tata bahasa dari *file* eksternal, dan menyimpannya ke dalam *file* internal program (*grammar.flis*).

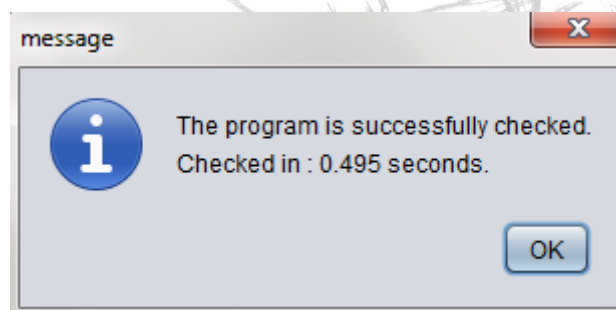


Gambar 6. Tampilan Grammar Editor

6. Contoh Masukan dan Hasil

a. Contoh Program yang Benar

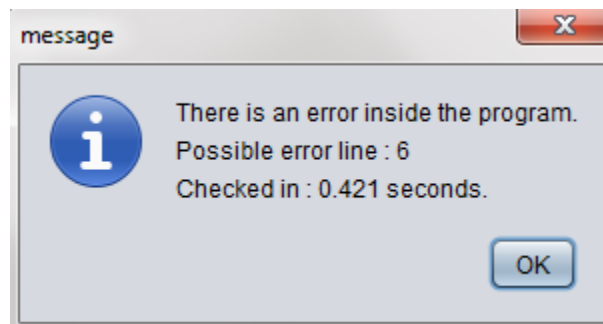
```
begin
  input ( hello )
  if ( x === 3 ) then
    begin
      output ( tiga )
    end
  end
end
```



Gambar 7. Hasil dari eksekusi Check pada program yang benar

b. Contoh Program yang Salah

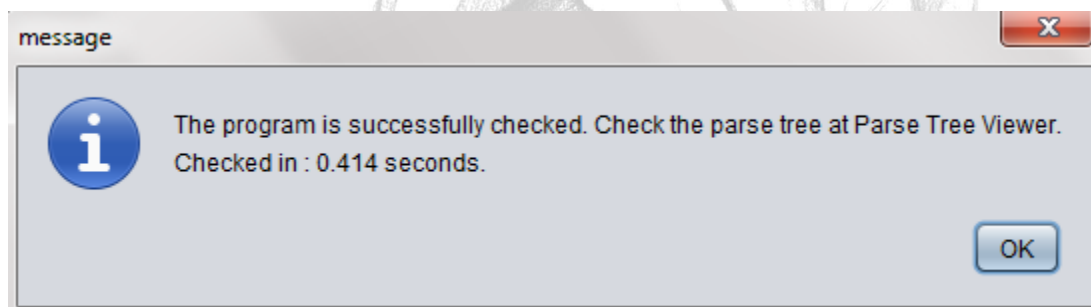
```
begin
  input ( hello )
  if ( x === 3 ) then
    begin
      output 9 tiga )
    end
  end
end
```



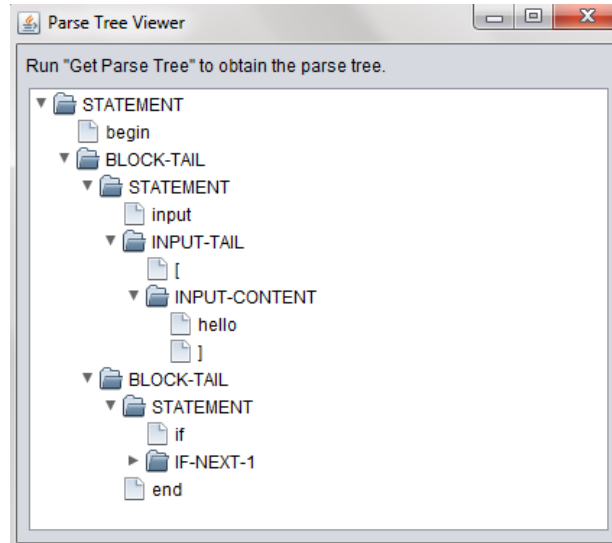
Gambar 8. Hasil eksekusi Check pada program yang salah

c. Contoh Program yang Benar dengan Parse Tree

```
begin
  input ( hello )
  if ( x === 3 ) then
    begin
      output ( tiga )
    end
  end
end
```



Gambar 9. Hasil dari eksekusi Get Parse Tree untuk program yang benar



Gambar 10. Tampilan Parse Tree Viewer untuk program yang benar

IV. Potongan Kode Felis

1. Kode Algoritma CYK

```

private static TreeSet<CYKParseTree> findDP(
    ArrayList<PairStringInt> inputs,
    TreeSet<ProductionUnit> units,
    TreeSet<ProductionNonunit> nonunits
)
{
    TreeSet<CYKParseTree> results = new TreeSet<>();

    String key = "";
    for (PairStringInt si : inputs) {
        key += si.toString() + si.getIndex();
    }

    if (dpMap.containsKey(key)) {
        //System.out.println(key);
        results = dpMap.get(key);
    } else {
        if (inputs.size() == 1) {
            for (ProductionUnit p : units) {
                if (p.validTerminal(inputs.get(0))) {
                    results.add(new CYKParseTree(p.getSymbol(), inputs.get(0)));
                }
            }
        } else {
            for (int i = 0; i < inputs.size() - 1; i++) {
                ArrayList<PairStringInt> leftInputs = new ArrayList<>();
                ArrayList<PairStringInt> rightInputs = new ArrayList<>();
                for (int j = 0; j <= i; j++) {
                    leftInputs.add(inputs.get(j));
                }
                for (int j = i + 1; j < inputs.size(); j++) {
                    rightInputs.add(inputs.get(j));
                }
                for (ProductionNonunit p : nonunits) {
                    TreeSet<CYKParseTree> leftTrees = findDP(leftInputs, units, nonunits);
                    TreeSet<CYKParseTree> rightTrees = findDP(rightInputs, units, nonunits);
                    for (CYKParseTree leftTree : leftTrees) {

```


```

        for (Iterator<CYKParseTree> it = rightTrees.iterator(); it.hasNext();) {
            CYKParseTree rightTree = it.next();
            if (leftTree.getNode().toString().equals(p.getLeft()) &&
                rightTree.getNode().toString().equals(p.getRight())) {
                results.add(new CYKParseTree(
                    p.getSymbol(),
                    leftTree,
                    rightTree));
            }
        }
    }
}
dpMap.put(key, results);
}
}
if (results.isEmpty()) {
    if (errorSize > inputs.size()) {
        errorSize = inputs.size();
        errorLine = inputs.get(0).getIndex();
    }
}
return results;
}

public static boolean check(
    ArrayList<PairStringInt> inputs,
    String start,
    TreeSet<ProductionUnit> units,
    TreeSet<ProductionNonunit> nonunits
)
{
    dpMap = new TreeMap<>();
    errorSize = inputs.size();
    TreeSet<CYKParseTree> resultTrees = findDP(inputs, units, nonunits);
    for (CYKParseTree resultTree : resultTrees) {
        if (resultTree.getNode().toString().equals(start)) {
            return true;
        }
    }
    errorAtLastRun = true;
    return false;
}

public static CYKParseTree getParseTree(
    ArrayList<PairStringInt> inputs,
    String start,
    TreeSet<ProductionUnit> units,
    TreeSet<ProductionNonunit> nonunits
)
{
    dpMap = new TreeMap<>();
    errorSize = inputs.size();
    TreeSet<CYKParseTree> resultTrees = findDP(inputs, units, nonunits);
    for (CYKParseTree resultTree : resultTrees) {
        if (resultTree.getNode().toString().equals(start)) {
            return resultTree;
        }
    }
    errorAtLastRun = true;
    return null;
}
}

```



2. Kode Prosesor File

```
public static ArrayList<PairStringInt> splitInputText(String t) {
    ArrayList<PairStringInt> result = new ArrayList<>();

    Scanner input = new Scanner(t);

    int comment_depth = 0;
    int line = 1;
    while (input.hasNextLine()) {
        Scanner subinput = new Scanner(input.nextLine());
        while (subinput.hasNext()) {
            String s = subinput.next();
            switch (s) {
                case "{":
                    comment_depth++;
                    break;
                case "}":
                    comment_depth--;
                    break;
                default:
                    if (comment_depth == 0) {
                        result.add(new PairStringInt(s, line));
                    }
                    break;
            }
            line++;
        }
    }
    return result;
}
```

