Datastructuren 2023

Programmeeropdracht 3: Patroonherkenning

Deadlines. Zaterdag 18 november 23:59, resp. zaterdag 2 december 23:59.

Inleiding. Deze opdracht is gebaseerd op Hoofdstuk 13.1.7 in het boek van Drozdek, getiteld *Regular Expression Matching*, onderdeel van Hoofdstuk 13 *String Matching*.

Veel programmeertalen hebben tegenwoordig faciliteiten voor het matchen van strings aan reguliere expressies. In deze opdracht zullen we een eenvoudige vorm daarvan uitwerken. Reguliere expressies geven het patroon weer waaraan gematcht moet worden. Dat matchen doen we (meestal) niet rechtstreeks aan de expressie, maar we representeren de expressie met een eindige automaat. Dat is het doel van het eerste deel.

In het tweede deel werken we het matchen uit, door een eenvoudige graafwandeling toe te passen.

De uiteindelijke opdracht staat op de laatste bladzijde, lees dus helemaal door!

Introductie

We zetten reguliere expressies om naar eindige automaten, zie de vakken *Foundations of Computer Science* en *Automata Theory*. De grote lijnen zijn bekend, maar er zijn een aantal kleine wijzigingen in de notatie tov. deze vakken.

Reguliere Expressies. De patronen die we willen herkennen zijn de zogenaamde *reguliere expressies*, die de reguliere talen beschrijven. Deze talen worden opgebouwd vanuit de eindige talen met de operaties vereniging, concatenatie en ster. Een voorbeeld van zo'n expressie is `c(a|bb)*c`. Expressies bevatten letters en speciale hulpsymbolen: de haakjes `(` en `)`, en operatoren `|` en `*`.

- **Letters.** De basiselementen vanwaaruit de expressie opgebouwd gaat worden. De expressie `a` specificeert enkel de string `a`. In deze opgave zijn de letters de 'kleine' letters `a` tm. `z` ('onderkast').
- **Keuze, vereniging.** Operatie, weergegeven door het symbool `|`. De expressie `a|bb` specificeert de string `a` of de string `bb`. In de tekst mag één van deze voorkomen.
- **Herhaling, ster.** Unaire postfix operatie `*`, dus weergegeven *achter* de expressie. De expressie `b*` geeft een willekeurig aantal (nul, één of meer) `b`'s achter elkaar aan. De expressie `(a|bb)*` geeft aan dat we herhaald ofwel een `a` ofwel twee `b`'s mogen schrijven. Hieraan voldoen bijvoorbeeld `abbaabb` en `aaa` , maar niet `aba` en `bbb`. Let op het gebruik van de haakjes: `bbb` voldoet wél aan de expressie `a|bb*` (één `a` of tenminste één `b`).
- **Concatenatie.** Operatie, het achter elkaar schrijven van expressies; heeft geen expliciet symbool als notatie. Zo geeft `c(a|bb)*c` aan dat we eerst een `c` willen zien, daarna herhaald een `a` of twee `b`'s en tenslotte weer een `c`. Eigenlijk is de expressie `bb` zelf ook de concatenatie van twee losse `b`'s.

- Haakjes, om voorrangsregels te omzeilen. Normaal gaat herhaling vóór concatenatie vóór keuze. Hierboven zagen we al haakjes: `ca|bb*c` specificeert onder andere `ca` en `bbbc`, die niet door `c(a|bb)*c` beschreven worden.

Een beetje formeler zouden we de *semantiek* $L(E)$ van reguliere een expressie E recursief kunnen definiëren als een taal, dus een deelverzameling van $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}^*$, volgens $L(\mathtt{x}) = \{\mathtt{x}\}$, $L(\mathtt{E|F}) = L(\mathtt{E}) \cup L(\mathtt{F})$, $L(\mathtt{EF}) = L(\mathtt{E}) \cdot L(\mathtt{F})$, en $L(\mathtt{E*}) = L(\mathtt{E})^*$.

Een BNF-grammatica voor reguliere expressies staat hieronder, passend bij de voorrangsregels; tussen vierkante haken staan optionele delen, verticale streep scheidt alternatieven. Een expressie is dus een keuze uit een reeks termen, een term de concatenatie van factoren, en een factor een letter of een expressie tussen haakjes (eventueel gesterd).

⟨expr⟩ := ⟨term⟩ [`|` ⟨expr⟩]
⟨term⟩ := ⟨fact⟩ [⟨term⟩]
⟨fact⟩ := ⟨lett⟩ [`*`] | `(` ⟨expr⟩ `)` [`*`]
⟨lett⟩ := `a` | `b` | ⋯ | `z`

Met deze grammatica is `a**` géén expressie, maar `(a*)*` wel. Bij Drozdek mogen er wel herhaalde `*`-en voorkomen. Als je dat natuurlijker vindt is dat geen probleem.

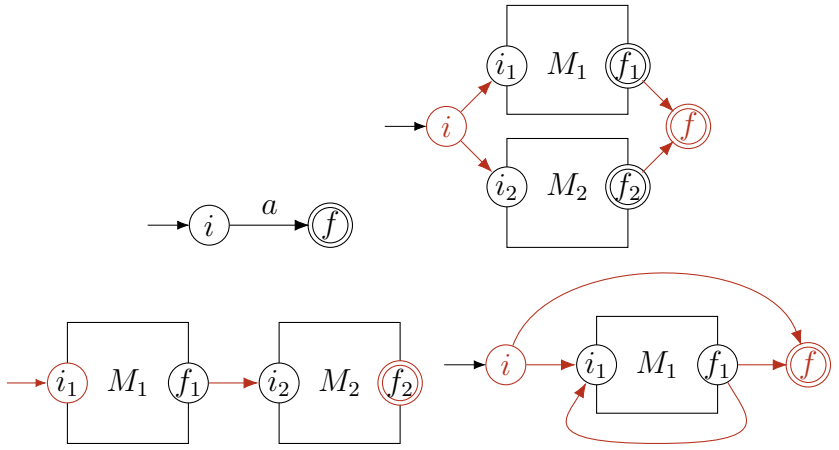
Eindige automaten. Reguliere expressies kunnen omgezet worden naar eindige automaten die dezelfde taal representeren met een constructie genoemd naar Ken Thompson. We zorgen er voor dat voor elk van de onderdelen van de expressie een equivalente operatie op eindige automaten is. In onze constructie gebruiken we automaten van een speciale vorm.

- Er is een unieke begin- en eindtoestand, hier aangegeven met i en f. De eindtoestand heeft geen uitgaande pijlen.
- We laten zgn. ε-transities toe, genoemd naar het lege woord ε. Dit zijn toestandsovergangen die geen letter lezen. Hier worden deze wergegeven als ongelabelde pijlen.
- Elke toestand (knoop) heeft ofwel precies één uitgaande pijl mét een letter, ofwel maximaal twee uitgaande pijlen zonder letter.

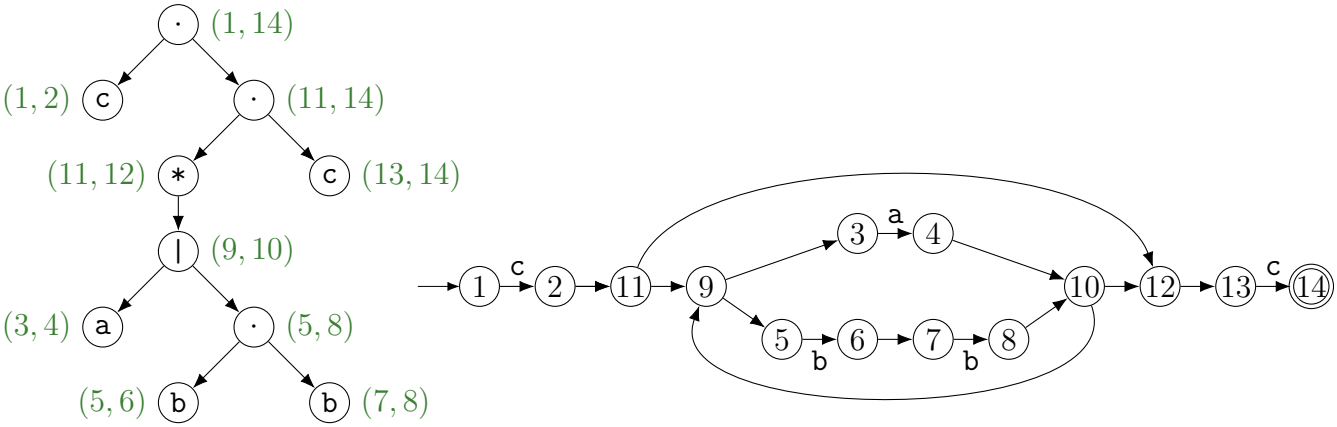
In Figuur 1 geven we de constructies van Thompson die passen bij de expressies `a`, `E|F`, `EF`, en `E*`, waarbij we aannemen dat de automaten voor `E` en `F` gevonden zijn als $\mathcal{A}_1$ en $\mathcal{A}_2$.

Er bestaan vele kleine varianten van deze constructies. Zo worden in Wikipedia voor concatenatie de twee toestanden f_1 en i_2 geïdentificeerd. Het gaat er echter niet om dat er altijd het minimale aantal toestanden wordt gevonden. Als het je uitkomt mag je de constructies aanpassen, zolang ze blijven doen wat ze beloven. Zie ook bij het vak Automata Theory, lecture 6, waar meerdere eindtoestanden gebruikt worden.

Als we deze constructies toepassen op onze voorbeeld-expressie krijgen we de eindige automaat als in Figuur 2. We geven eerst de boom die de structuur weergeeft van de expressie en vormen daaruit de automaat met een postorde evaluatie. De boom hier is gelabeld met de begin- en eindtoestanden van de bij de deelexpressies behorende deelautomaten.



Figuur 1: Constructies op eindige automaten voor de reguliere operaties: enkele letter, keuze, concatenatie en ster.



Figuur 2: Expressie `c(a|bb)*c` weergegeven als boom, en de resulterende eindige automaat. De cijfers bij de knopen in de boom geven de begin- en eindtoestand van de corresponderende deelautomaat weer.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
c	-	a	-	b	-	b	-	-	-	-	-	c	-
2	11	4	10	6	7	8	10	3	9	9	13	14	-
-	-	-	-	-	-	-	-	5	12	12	-	-	-

Figuur 3: De informatie benodigd voor de structuur van de automaat past in een tabel!

Deel Eén

1 Ontleder

De reguliere expressie kan omgezet worden in een eindige automaat door een zogenaamde *recursive descent parser*. Dat is een techniek waar de structuur van de expressie wordt omgezet in een boom, die vervolgens *bottom-up* wordt geëvalueerd tot automaat. Speciaal hierbij is dat die expressie-boom nooit expliciet wordt gebouwd, maar slechts bestaat als de recursieve aanroepen van functies die de corresponderen met de onderdelen van de BNF-grammatica, dus met ⟨expr⟩, ⟨term⟩, en ⟨fact⟩.

Een schets van het recursieve programma wordt gegeven in Figuur 4. De aanroepen die de expressie ontleden, en de automaat construeren, staan in Figuur 5.

Bedenk zelf hoe je de eindige automaat wilt representeren. De automaat is een graaf, maar het is niet vanzelfsprekend om de standaard *adjacency lists* of *adjacency matrix* te nemen. Omdat de knopen maximaal twee uitgaande takken hebben kun je kiezen voor een speciale 'compacte' representatie, zie Figuur 3

Tips. Maak de automaat een private data element binnen de klasse die de reguliere expressies ontleedt. Die kan dan als 'globale variabele' gebruikt worden.

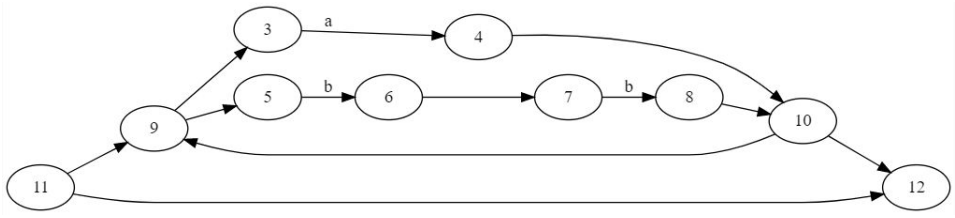
Het is niet handig is om een graaf met pointers te maken!

2 Weergeven

Om eenvoudig te kunnen controleren hoe de geconstrueerde automaat precies gevormd is willen we die ook kunnen uitvoeren. En weer op een grafische manier, in de DOT-notatie die ook in de vorige opgave werd gebruikt. Als de toestanden bij de constructie al een nummer hebben gekregen hoeven nu geen labels voor de knopen te worden gegenereerd.

Hieronder een gedeeltelijk voorbeeld, gevisualiseerd via dreampuf.github.io/GraphvizOnline.

```
digraph G {
  rankdir="LR"
  11 -> 9 -> 3
  3 -> 4 [label="a"]
  4 -> 10 -> 9 -> 5
  5 -> 6 [label="b"]
  6 -> 7
  7 -> 8 [label="b"]
  8 -> 10 -> 12
  11 -> 12
}
```



3 Documentatie

Aan het einde van de gehele opdracht verwachten we een kort verslag. Beschrijf daarin duidelijk welke representatie voor de automaat is gekozen en beargumenteer ook waarom. Dit hoef je nu nog niet in te leveren, maar bereid alvast een voorbeeld voor: expressie en resulterende automaat.

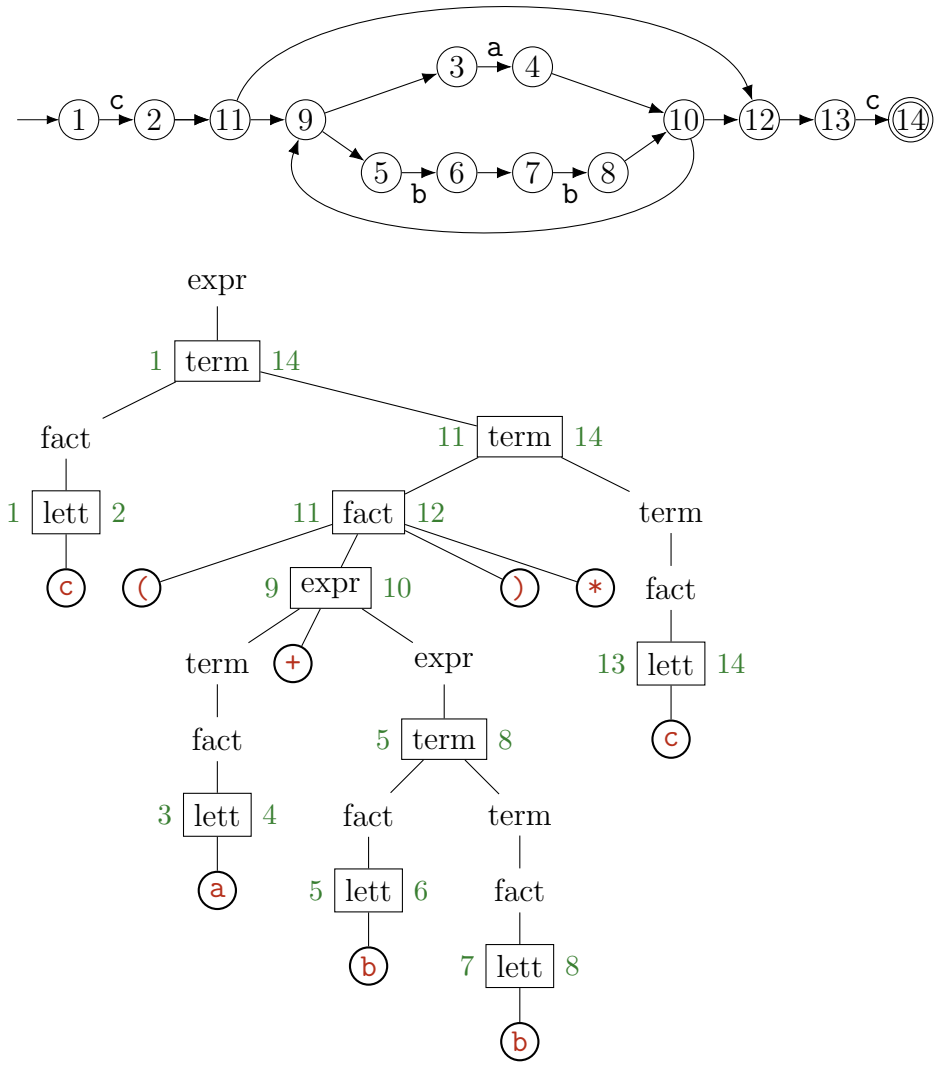
Voordat je stopt met lezen: er staan belangrijke instructies op bladzijde 8!

```
Automaat Expr() {
   Automaat Aut1 = Term() ;
   if ( huidige letter == '|' )
   { lees volgende letter;
     Automaat Aut2 = Expr();     // recursie!
   }
   return plus van Aut1 en Aut2;
} // Expr

Automaat Term () {
   Automaat Aut = Fact;
   if ( huidige letter in {'(','a',...,'z'}  )
      Automaat Aut2 = Term();        // recursie!
   return concatenatie van Aut1 en Aut2;
} // Term

Automaat Fact () {
   if ( huidige letter == '(')
   {  lees volgende letter;
      Automaat Aut = Expr() ;
      if ( huidige letter == ')')
         lees volgende letter;
      else
         Error! ;
   }
   elseif ( huidige letter in {'a',...,'z'} )
   {  Automaat Aut = tak met huidige letter;
      lees volgende letter;
   }
   else
      Error!
   if (HuidigeLetter == '*')  // eventueel while?
   {  pas ster toe op Aut;
      lees volgende letter;
   }
   return Aut;
} // Fact
```

Figuur 4: Programmaschets voor de recursieve ontleding van reguliere expressies.



Figuur 5: Recursieve functie-aanroepen voor ons voorbeeld `c(a|bb)*c`. Bij de speciaal aangegeven functies worden nieuwe takken en eventueel knopen van de automaat geconstrueerd (volgens Thompson). De bijbehorende begin- en eindknoop zijn dan vermeld.

Deel Twee

4 Matchen

Nu eenmaal een eindige automaat (met ε-transities) bepaald is kunen we een string matchen aan de expressie door te kijken of de automaat de string accepteert. Dat betekent dat er een wandeling is in de automaat van begin- naar eindtoestand waar de labels langs de takken de string vormen.

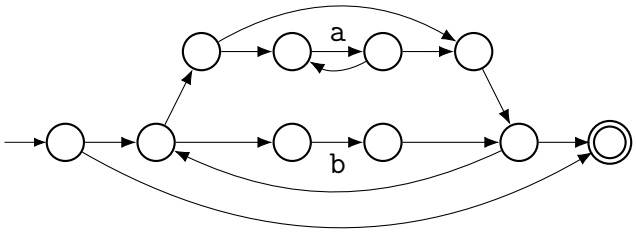
Het zoeken naar zo'n wandeling doen we door te beginnen in de begintoestand en afwisselend de volgende letter te volgen en de ε-afsluiting te bepalen van de toestanden waar we geëindigd zijn. De ε-afsluiting is de verzameling toestanden die we vanuit andere toestanden kunnen vinden door alleen ε-pijlen te volgen.

Voorbeeld. De ε-afsluiting van de verzameling $\{2\}$ in de automaat van Figuur 2 bestaat uit $\{2, 11, 9, 3, 5, 12, 13\}$.

Het is belangrijk om steeds letter-voor-letter te werken! Dat wil zeggen: na elke letter van de string die we matchen aan de expressie weten we precies in welke toestanden de autoimaat zich zou kunnen bevinden. Op die manier zoeken we min of meer *breadth-first* en niet *depth-first* in de automaat.

Terzijde. De automaat wordt bij dit proces dus niet expliciet deterministisch gemaakt. Als je wilt kun je wel verbanden trekken. Bij een deterministische automaat houden we de verzameling toestanden bij die het niet-deterministische origineel allemaal kan bereiken. Dat gebeurt hier *on the fly*.

Let op. De eindige automaat is een graaf. Er kunnen cykels van ε-takken in voorkomen: `(a*|b)*`.



En wat doet het programma dan precies?

We verwachten in `main` een "interactief" programma dat de volgende instructies accepteert.

- `exp <expressie>` – leest de reguliere expressie en bepaalt een passende eindige automaat. Als een nieuwe expressie gegeven wordt, hoeft de oude niet onthouden te worden.
- `dot <bestandsnaam>` – slaat de eindige automaat op naar de gegeven file, in DOT notatie.
- `end` – om het programma af te sluiten.

Voor de tweede deadline komt daar dan nog bij:

- `mat <string>` – checkt of de string wordt gaccepteerd door de automaat. De uitvoer is de string `match` of `geen match`.

Een voorbeeld van een simpele invoer voor het menu is uiteindelijk:

```
exp a(b|cd)*ef
dot test1.dot
mat abcdcdbbef
exp (a*|b)*a
mat baab
mat $
end
```

Omdat het niet altijd makkelijk is om de lege string in te lezen kan die aangegeven worden door `$` wanneer we die willen matchen aan de expressie.

Instructies. Zie vorige opdracht. Met `./main < file.txt` kan de inhoud van een bestand met vooraf bepaalde instructies naar standaard `in` worden gestuurd, dat maakt het testen makkelijker. *Zorg dat dit werkt!* (want wij gaan dat zo ook toepassen).

Werk in tweetallen. Maak nette code en documentatie (in leveren met de volledige opdracht). Geef in `readme.txt` alle informatie die we eigenlijk nog zouden moeten weten.

Veel succes.