DD1361
Programmeringsparadigm

Formella Språk & Syntaxanalys
Föreläsning 1

Per Austrin

2016-10-31

Kursavsnittet syntax/formella språk

- **Teori om formella språk – verktygslåda för strängmatchning:**
  - Ändliga automater och reguljära uttryck
  - Olika kraftfulla språkklasser

- **Formella språk i praktiken:**
  - Regex
  - Lexikal analys
  - Att skriva en parser för ett språk

Idag

Reguljära uttryck

Ändliga automater

Formella språk

Några praktiska exempel

Idag

Reguljära uttryck

Ändliga automater

Formella språk

Några praktiska exempel

Reguljära uttryck

Reguljära uttryck är *mönster* för att beskriva hur en sträng ska se ut.

Man kan säga att det är ett språk för att beskriva strängar.

Reguljära uttryck

Reguljära uttryck är *mönster* för att beskriva hur en sträng ska se ut.

Man kan säga att det är ett språk för att beskriva strängar.

Exempel:
Vi är intresserade av namnen på de obligatoriska labbarna i kursen:
F1, F2, L1, L2, S1, S2, Inet

Reguljära uttryck

Reguljära uttryck är *mönster* för att beskriva hur en sträng ska se ut.

Man kan säga att det är ett språk för att beskriva strängar.

Exempel:
Vi är intresserade av namnen på de obligatoriska labbarna i kursen:
F1, F2, L1, L2, S1, S2, Inet

Vi kan beskriva dem med det reguljära uttrycket
`F1|F2|L1|L2|S1|S2|Inet`

Reguljära uttryck

Reguljära uttryck är *mönster* för att beskriva hur en sträng ska se ut.

Man kan säga att det är ett språk för att beskriva strängar.

Exempel:
Vi är intresserade av namnen på de obligatoriska labbarna i kursen:
F1, F2, L1, L2, S1, S2, Inet

Vi kan beskriva dem med det reguljära uttrycket
`F1|F2|L1|L2|S1|S2|Inet`

Vertikalstreck/pipetecken är en operator i reguljära uttryck som betyder "eller" – antingen vänstra eller högra operanden

Fortsättning på exempel

Namnen på labbarna i kursen följer ju egentligen ett tämligen enkelt mönster:

- Antingen är strängen “Inet”
- Eller så består den av något av tecknen ‘F’, ‘L’ eller ‘S’, följt av något av tecknen ‘1’ eller ‘2’

Fortsättning på exempel

Namnen på labbarna i kursen följer ju egentligen ett tämligen enkelt mönster:

- Antingen är strängen “Inet”
- Eller så består den av något av tecknen ‘F’, ‘L’ eller ‘S’, följt av något av tecknen ‘1’ eller ‘2’

Denna typ av beskrivning kan formuleras i reguljära uttryck:
`Inet|(F|L|S)(1|2)`

Fortsättning på exempel

Namnen på labbarna i kursen följer ju egentligen ett tämligen enkelt mönster:

- Antingen är strängen “Inet”
- Eller så består den av något av tecknen ‘F’, ‘L’ eller ‘S’, följt av något av tecknen ‘1’ eller ‘2’

Denna typ av beskrivning kan formuleras i reguljära uttryck:
`Inet|(F|L|S)(1|2)`

Parenteser används för att gruppera uttryck

Fortsättning på exempel

Namnen på labbarna i kursen följer ju egentligen ett tämligen enkelt mönster:

- Antingen är strängen “Inet”
- Eller så består den av något av tecknen ‘F’, ‘L’ eller ‘S’, följt av något av tecknen ‘1’ eller ‘2’

Denna typ av beskrivning kan formuleras i reguljära uttryck:
`Inet|(F|L|S)(1|2)`

Parenteser används för att gruppera uttryck

Antingen F eller L eller S

Fortsättning på exempel

Namnen på labbarna i kursen följer ju egentligen ett tämligen enkelt mönster:

- Antingen är strängen “Inet”
- Eller så består den av något av tecknen ‘F’, ‘L’ eller ‘S’, följt av något av tecknen ‘1’ eller ‘2’

Denna typ av beskrivning kan formuleras i reguljära uttryck:
`Inet|(F|L|S)(1|2)`

Parenteser används för att gruppera uttryck

Antingen F eller L eller S

Antingen 1 eller 2

Fortsättning på exempel

Namnen på labbarna i kursen följer ju egentligen ett tämligen enkelt mönster:

- Antingen är strängen “Inet”
- Eller så består den av något av tecknen ‘F’, ‘L’ eller ‘S’, följt av något av tecknen ‘1’ eller ‘2’

Denna typ av beskrivning kan formuleras i reguljära uttryck:
`Inet|(F|L|S)(1|2)`

Parenteser används för att gruppera uttryck

Antingen F eller L eller S

Antingen 1 eller 2

(Antingen F eller L eller S) följt av (antingen 1 eller 2)

Exempel 2 – upprepning

Exempel:
Vi är intresserade av alla binära strängar:
“”, “0”, “1”, “00”, “01”, “10”, “11”, . . .

Exempel 2 – upprepning

Exempel:
Vi är intresserade av alla binära strängar:
“”, “0”, “1”, “00”, “01”, “10”, “11”, . . .

Oändligt många strängar, vi kan inte skriva en lista med alla

Exempel 2 – upprepning

Exempel:
Vi är intresserade av alla binära strängar:
“”, “0”, “1”, “00”, “01”, “10”, “11”, . . .

Oändligt många strängar, vi kan inte skriva en lista med alla

Vi kan beskriva dem med det reguljära uttrycket
`(0|1)*`

Exempel 2 – upprepning

Exempel:
Vi är intresserade av alla binära strängar:
“”, “0”, “1”, “00”, “01”, “10”, “11”, . . .

Oändligt många strängar, vi kan inte skriva en lista med alla

Vi kan beskriva dem med det reguljära uttrycket
`(0|1)*`

Upprepningsoperatorn ‘*’ betyder:
“det som stod precis framför, ett valfritt antal gånger”

Exempel 2 – upprepning

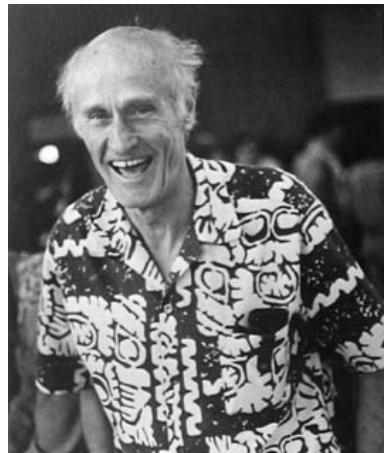
Exempel:
Vi är intresserade av alla binära strängar:
“”, “0”, “1”, “00”, “01”, “10”, “11”, . . .

Oändligt många strängar, vi kan inte skriva en lista med alla

Vi kan beskriva dem med det reguljära uttrycket
`(0|1)*`

Upprepningsoperatorn ‘*’ betyder:
“det som stod precis framför, ett valfritt antal gånger”

Den här operationen kallas även Kleene-stjärna
eller Kleene-slutning (Kleene closure)



Reguljära uttryck – sammanfattning av operationerna

Konkatenering: "`abc`" beskriver `a` följt av `b` följt av `c` (dvs strängen `abc`).

Reguljära uttryck – sammanfattning av operationerna

Konkatenering: "`abc`" beskriver `a` följt av `b` följt av `c` (dvs strängen `abc`).

Alternering: "`ab|cd`" beskriver någon av strängarna `ab` eller `cd`.

Reguljära uttryck – sammanfattning av operationerna

Konkatenering: "`abc`" beskriver `a` följt av `b` följt av `c` (dvs strängen `abc`).

Alternering: "`ab|cd`" beskriver någon av strängarna `ab` eller `cd`.

Gruppering: Kan använda parenteser för att ändra i vilken ordning konkatenering och alternering görs: "`a(b|c)d`" beskriver någon av strängarna `abd` eller `acd`

Reguljära uttryck – sammanfattning av operationerna

Konkatenering: "`abc`" beskriver `a` följt av `b` följt av `c` (dvs strängen `abc`).

Alternering: "`ab|cd`" beskriver någon av strängarna `ab` eller `cd`.

Gruppering: Kan använda parenteser för att ändra i vilken ordning konkatenering och alternering görs: "`a(b|c)d`" beskriver någon av strängarna `abd` eller `acd`

Upprepning: "`a(b|c)*d`" beskriver alla strängar som börjar på a, slutar på d, och däremellan har vilken sträng som helst bestående av tecknen b och c, t.ex. `acccccccccd`, `acbd`, `abbbcccbcbcd` och `ad`.

Syntaktiskt socker

I praktiken lägger man alltid till lite syntaktiskt socker till reguljära uttryck för att förenkla en del konstruktioner.

Syntaktiskt socker

I praktiken lägger man alltid till lite syntaktiskt socker till reguljära uttryck för att förenkla en del konstruktioner.

De vanligaste och mest oumbärliga exemplen:

- **Teckenmängd:** "`[abcd]`" betyder "`a|b|c|d`" något av tecknena `a`, `b` `c` eller `d`

Syntaktiskt socker

I praktiken lägger man alltid till lite syntaktiskt socker till reguljära uttryck för att förenkla en del konstruktioner.

De vanligaste och mest oumbärliga exemplen:

- **Teckenmängd:** "`[abcd]`" betyder "`a|b|c|d`" något av tecknena `a`, `b` `c` eller `d`
- **Teckenintervall:** "`[a-d]`" betyder samma som ovan – något av tecknen i intervallet

Syntaktiskt socker

I praktiken lägger man alltid till lite syntaktiskt socker till reguljära uttryck för att förenkla en del konstruktioner.

De vanligaste och mest oumbärliga exemplen:

- **Teckenmängd:** "`[abcd]`" betyder "`a|b|c|d`" något av tecknena `a`, `b` `c` eller `d`
- **Teckenintervall:** "`[a-d]`" betyder samma som ovan – något av tecknen i intervallet
- **Teckenkomplement:** "`[^a-d]`" beskriver alla tecken som *inte* är i intervallet

Syntaktiskt socker

I praktiken lägger man alltid till lite syntaktiskt socker till reguljära uttryck för att förenkla en del konstruktioner.

De vanligaste och mest oumbärliga exemplen:

- **Teckenmängd:** "`[abcd]`" betyder "`a|b|c|d`" något av tecknena `a`, `b` `c` eller `d`
- **Teckenintervall:** "`[a-d]`" betyder samma som ovan – något av tecknen i intervallet
- **Teckenkomplement:** "`[^a-d]`" beskriver alla tecken som *inte* är i intervallet
- **Noll eller en:** "`X?`" betyder "`(|X)`", dvs antingen en tom sträng eller `X`

Syntaktiskt socker

I praktiken lägger man alltid till lite syntaktiskt socker till reguljära uttryck för att förenkla en del konstruktioner.

De vanligaste och mest oumbärliga exemplen:

- **Teckenmängd:** "`[abcd]`" betyder "`a|b|c|d`" något av tecknena `a`, `b` `c` eller `d`
- **Teckenintervall:** "`[a-d]`" betyder samma som ovan – något av tecknen i intervallet
- **Teckenkomplement:** "`[^a-d]`" beskriver alla tecken som *inte* är i intervallet
- **Noll eller en:** "`X?`" betyder "`(|X)`", dvs antingen en tom sträng eller `X`
- **En eller flera:** "`X+`" betyder "`XX*`", dvs en eller flera upprepningar av `X`

Syntaktiskt socker

I praktiken lägger man alltid till lite syntaktiskt socker till reguljära uttryck för att förenkla en del konstruktioner.

De vanligaste och mest oumbärliga exemplen:

- **Teckenmängd:** "`[abcd]`" betyder "`a|b|c|d`" något av tecknena `a`, `b` `c` eller `d`
- **Teckenintervall:** "`[a-d]`" betyder samma som ovan – något av tecknen i intervallet
- **Teckenkomplement:** "`[^a-d]`" beskriver alla tecken som *inte* är i intervallet
- **Noll eller en:** "`X?`" betyder "`(|X)`", dvs antingen en tom sträng eller `X`
- **En eller flera:** "`X+`" betyder "`XX*`", dvs en eller flera upprepningar av `X`
- **Godtyckligt tecken:** "`.`" betyder "vilket tecken som helst"

Exempel: e-post-adresser

Exempel: skriv ett reguljärt uttryck som beskriver giltiga e-post-adress.

Exempel: e-post-adresser

Exempel: skriv ett reguljärt uttryck som beskriver giltiga e-post-adress.

I det här exemplet definierar vi giltig e-post-adress enligt följande regler:

1. Adressen består av en lokal del och ett domännamn, som separeras av @.
2. Den lokala delen får bestå av
   - stora, små bokstäver och siffror,
   - punkter, men inte som första eller sista tecken i lokala delen.
3. Domännamnet är en lista med minst två punktseparerade icke-tomma strängar med bara små bokstäver och siffror.

Reguljärt uttryck för lokal del

2. Den lokala delen får bestå av
   - stora, små bokstäver och siffror,
   - punkter, men inte som första eller sista tecken i lokala delen.

Reguljärt uttryck för lokal del

2. Den lokala delen får bestå av
   - stora, små bokstäver och siffror,
   - punkter, men inte som första eller sista tecken i lokala delen.

Reguljära uttryck för de tillåtna tecknen:

`[A-Za-z0-9.]` – inklusive punkt

`[A-Za-z0-9]` – exklusive punkt

Reguljärt uttryck för lokal del

2. Den lokala delen får bestå av
   - stora, små bokstäver och siffror,
   - punkter, men inte som första eller sista tecken i lokala delen.

Reguljära uttryck för de tillåtna tecknen:
`[A-Za-z0-9.]` – inklusive punkt
`[A-Za-z0-9]` – exklusive punkt

Reguljärt uttryck för lokal del:
`[A-Za-z0-9][A-Za-z0-9.]*[A-Za-z0-9]`

Reguljärt uttryck för lokal del

2. Den lokala delen får bestå av
   - stora, små bokstäver och siffror,
   - punkter, men inte som första eller sista tecken i lokala delen.

Reguljära uttryck för de tillåtna tecknen:

`[A-Za-z0-9.]` – inklusive punkt

`[A-Za-z0-9]` – exklusive punkt

Reguljärt uttryck för lokal del:

`[A-Za-z0-9][A-Za-z0-9.]*[A-Za-z0-9]`

Första tecknet ej punkt

Tecken däremellan får även vara punkt

Sista tecken ej punkt

Reguljärt uttryck för lokal del

2. Den lokala delen får bestå av
   - stora, små bokstäver och siffror,
   - punkter, men inte som första eller sista tecken i lokala delen.

Reguljära uttryck för de tillåtna tecknen:
`[A-Za-z0-9.]` – inklusive punkt
`[A-Za-z0-9]` – exklusive punkt

Reguljärt uttryck för lokal del:

```
[A-Za-z0-9][A-Za-z0-9.]*[A-Za-z0-9]
```

Eller...?

Reguljärt uttryck för lokal del

2. Den lokala delen får bestå av
   - stora, små bokstäver och siffror,
   - punkter, men inte som första eller sista tecken i lokala delen.

Reguljära uttryck för de tillåtna tecknen:
`[A-Za-z0-9.]` – inklusive punkt
`[A-Za-z0-9]` – exklusive punkt

Reguljärt uttryck för lokal del:
`[A-Za-z0-9][A-Za-z0-9.]*[A-Za-z0-9]`

Eller...?

Uttrycket antar att lokala delen består av minst två tecken – men det ska vara tillåtet för den lokala delen att bestå av ett enda tecken – fel!

Reguljärt uttryck för lokal del

2. Den lokala delen får bestå av
   - stora, små bokstäver och siffror,
   - punkter, men inte som första eller sista tecken i lokala delen.

Reguljära uttryck för de tillåtna tecknen:
`[A-Za-z0-9.]` – inklusive punkt
`[A-Za-z0-9]` – exklusive punkt

Reguljärt uttryck för lokal del – försök 2:
`[A-Za-z0-9]([A-Za-z0-9.]*[A-Za-z0-9])?`

Reguljärt uttryck för domännamn

3. Domännamnet är en lista med minst två punktseparerade icke-tomma strängar med bara små bokstäver och siffror.

Reguljärt uttryck för domännamn

3. Domännamnet är en lista med minst två punktseparerade icke-tomma strängar med bara små bokstäver och siffror.

Reguljärt uttryck för "icke-tom sträng med bara små bokstäver och siffror":

```
[a-z0-9]+
```

Reguljärt uttryck för domännamn

3. Domännamnet är en lista med minst två punktseparerade icke-tomma strängar med bara små bokstäver och siffror.

Reguljärt uttryck för "icke-tom sträng med bara små bokstäver och siffror":

```
[a-z0-9]+
```

Reguljärt uttryck för domändelen:

```
[a-z0-9]+(\.[a-z0-9]+)*
```

Reguljärt uttryck för domännamn

3. Domännamnet är en lista med minst två punktseparerade icke-tomma strängar med bara små bokstäver och siffror.

Reguljärt uttryck för "icke-tom sträng med bara små bokstäver och siffror":

```
[a-z0-9]+
```

Reguljärt uttryck för domändelen:

```
[a-z0-9]+(\.[a-z0-9]+)*
```

Icke-tom sträng med små bokstäver och siffror

Punkt följd av icke-tom sträng med små bokstäver och siffror

Upprepat 0 eller fler gånger

Reguljärt uttryck för domännamn

3. Domännamnet är en lista med minst två punktseparerade icke-tomma strängar med bara små bokstäver och siffror.

Reguljärt uttryck för "icke-tom sträng med bara små bokstäver och siffror":

```
[a-z0-9]+
```

Reguljärt uttryck för domändelen:

```
[a-z0-9]+(\.[a-z0-9]+)*
```

Fel igen! Tillåter domändel med bara en del, ska vara minst två!

Reguljärt uttryck för domännamn

3. Domännamnet är en lista med minst två punktseparerade icke-tomma strängar med bara små bokstäver och siffror.

Reguljärt uttryck för "icke-tom sträng med bara små bokstäver och siffror":

```
[a-z0-9]+
```

Reguljärt uttryck för domändelen:

```
[a-z0-9]+(\.[a-z0-9]+)+
```

Sätta ihop lösningen

1. Adressen består av en lokal del och ett domännamn, som separeras av @.

Sätta ihop lösningen

1. Adressen består av en lokal del och ett domännamn, som separeras av @.

Reguljärt uttryck för lokal del:

```
[A-Za-z0-9]([A-Za-z0-9.]*[A-Za-z0-9])?
```

Reguljärt uttryck för domändelen:

```
[a-z0-9]+(\.[a-z0-9]+)+
```

Sätta ihop lösningen

1. Adressen består av en lokal del och ett domännamn, som separeras av @.

Reguljärt uttryck för lokal del:

```
[A-Za-z0-9]([A-Za-z0-9.]*[A-Za-z0-9])?
```

Reguljärt uttryck för domändelen:

```
[a-z0-9]+(\.[a-z0-9]+)+
```

Kombinerat till ett reguljärt uttryck för e-post-adresser:

```
[A-Za-z0-9]([A-Za-z0-9.]*[A-Za-z0-9])?@[a-z0-9]+(\.[a-z0-9]+)+
```

Sätta ihop lösningen

1. Adressen består av en lokal del och ett domännamn, som separeras av @.

Reguljärt uttryck för lokal del:
`[A-Za-z0-9]([A-Za-z0-9.]*[A-Za-z0-9])?`

Reguljärt uttryck för domändelen:
`[a-z0-9]+(\.[a-z0-9]+)+`

Kombinerat till ett reguljärt uttryck för e-post-adresser:
`[A-Za-z0-9]([A-Za-z0-9.]*[A-Za-z0-9])?@[a-z0-9]+(\.[a-z0-9]+)+`

(De riktiga reglerna för hur en giltig e-post-adress ser ut är mycket mer komplicerade, det här var bara ett litet leksaks-exempel)

Reguljärt uttryck för e-post (första halvan)

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+
(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)
?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
 \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+
|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t
]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031
]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](
?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?
:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?
:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?
:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|
\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>
@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"
(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?
:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[
\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-
```

Idag

Reguljära uttryck

Ändliga automater

Formella språk

Några praktiska exempel

Ändliga automater

Ändliga automater är, precis som reguljära uttryck, ett verktyg för att beskriva en uppsättning strängar

Den enklaste typen av automat kallas för
DFA, Deterministic Finite Automaton.

Lite informellt är en DFA en "maskin" som kan befinna sig i en given uppsättning olika tillstånd, som läser en sträng tecken för tecken och byter tillstånd baserat på de tecken som läses, enligt en given uppslagstabell.

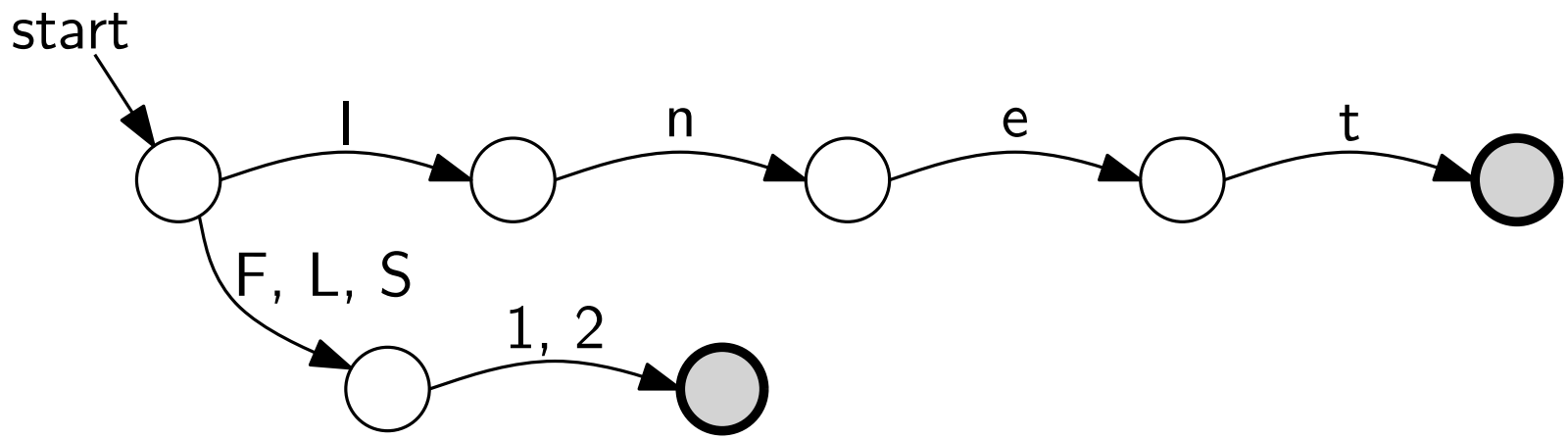
Progp-labbarna igen

De obligatoriska progp-labbarna:
F1, F2, L1, L2, S1, S2, Inet

Progp-labbarna igen

De obligatoriska progp-labbarna:
F1, F2, L1, L2, S1, S2, Inet

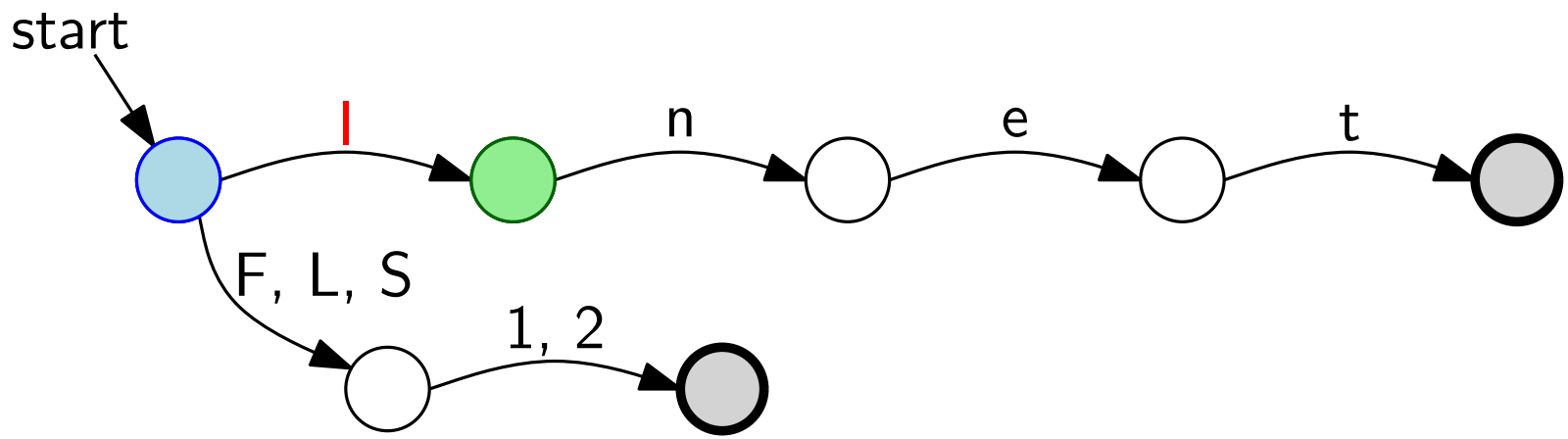
Dessa kan beskrivas med följande ändliga automat



Progp-labbarna igen

De obligatoriska progp-labbarna:
F1, F2, L1, L2, S1, S2, Inet

Dessa kan beskrivas med följande ändliga automat

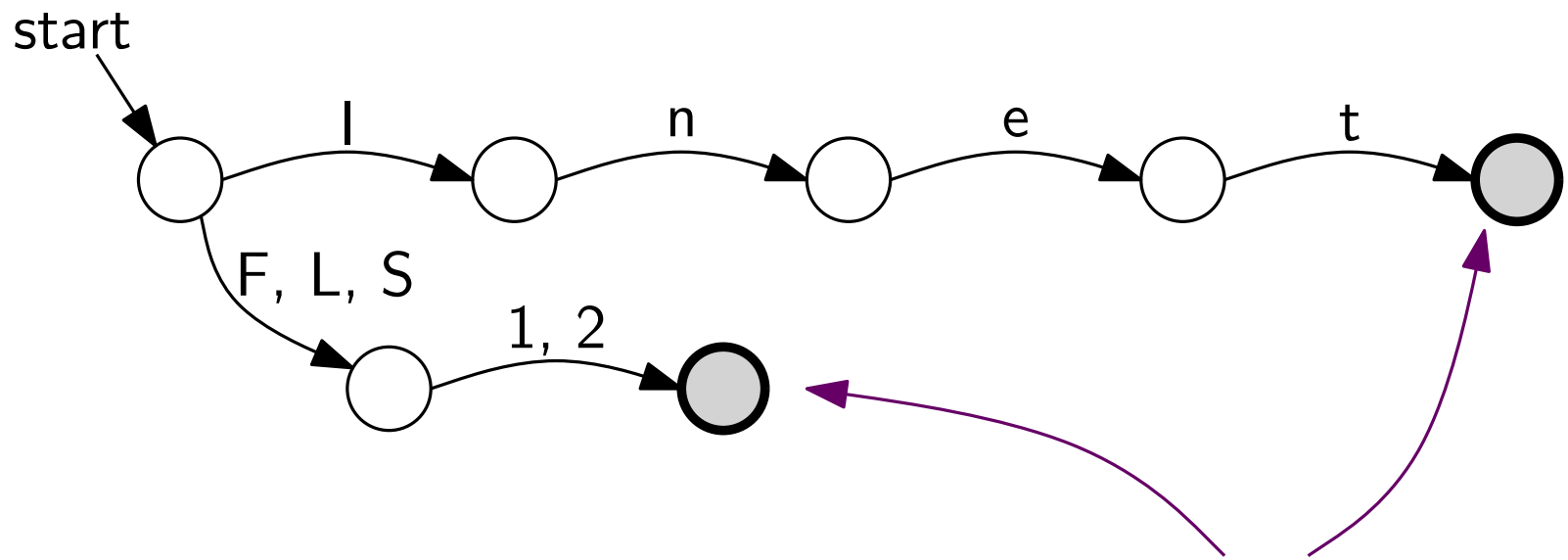


Om vi befinner oss här och nästa tecken är ett ‘I’, hoppa hit

Progp-labbarna igen

De obligatoriska progp-labbarna:
F1, F2, L1, L2, S1, S2, Inet

Dessa kan beskrivas med följande ändliga automat



Om vi befinner oss i något av dessa tillstånd när strängen är slut så är det en korrekt sträng

Lite mer formellt

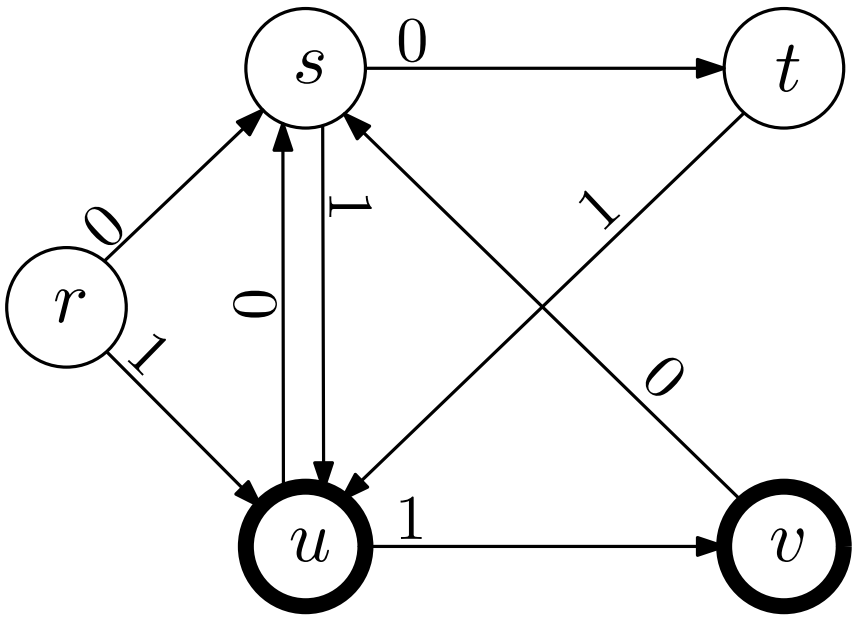
En ändlig automat består av:

- ett antal *tillstånd*
- *övergångar* mellan tillstånden, märkta med något tecken
- ett tillstånd är *starttillstånd*
- något/några tillstånd är *accepterande tillstånd*

Lite mer formellt

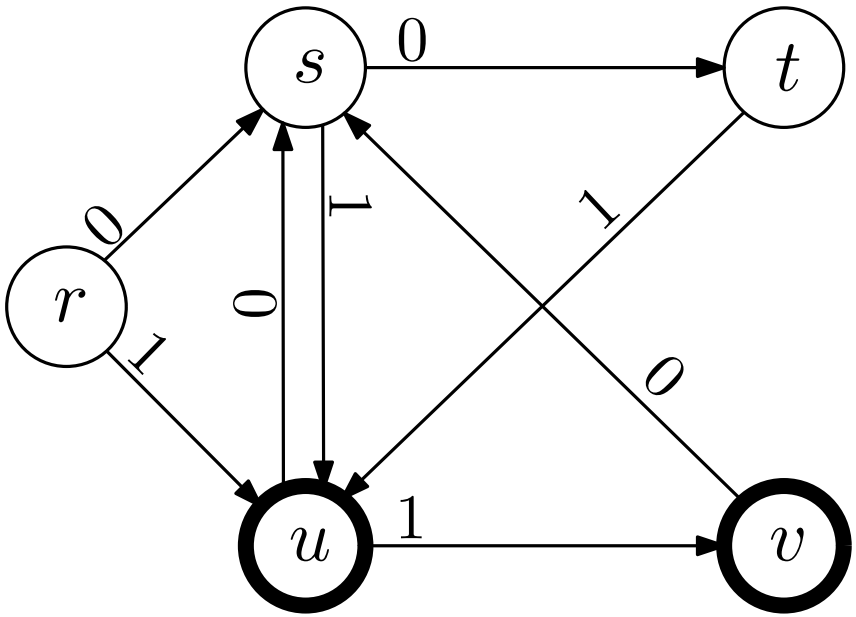
En ändlig automat består av:

- ett antal *tillstånd*
- *övergångar* mellan tillstånden, märkta med något tecken
- ett tillstånd är *starttillstånd*
- något/några tillstånd är *accepterande tillstånd*



Starttillstånd r
Accepterande tillstånd u och v

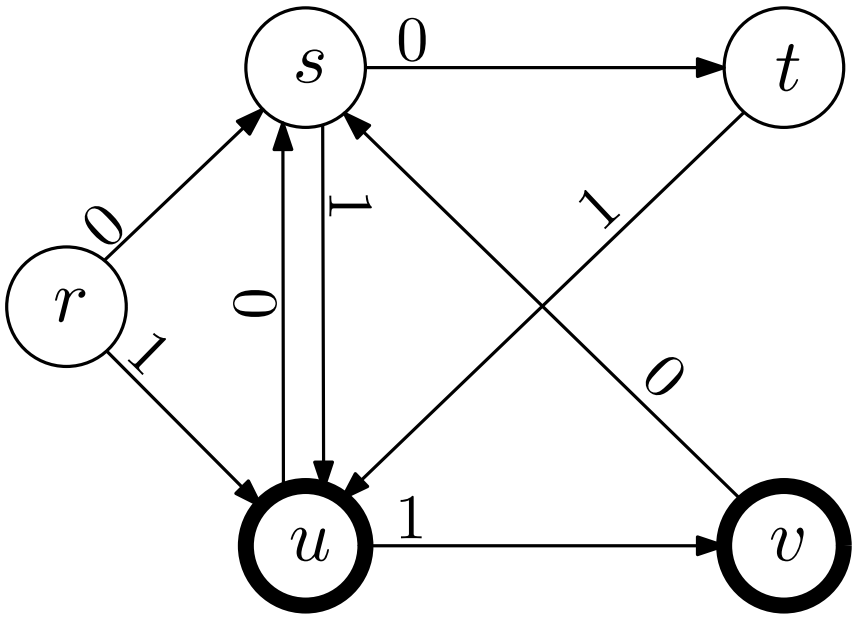
Exempel 1



Starttillstånd r

Accepterande tillstånd u och v

Exempel 1

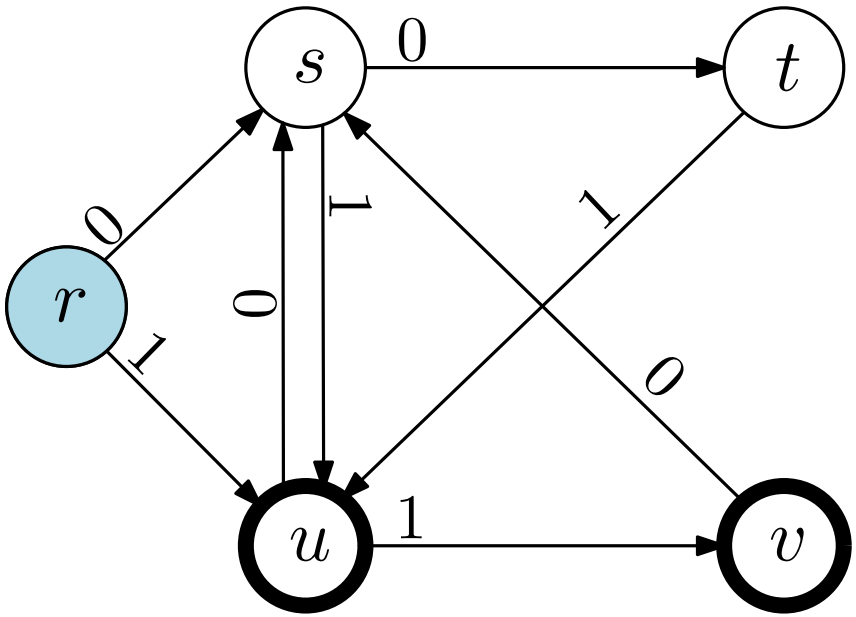


Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101

Exempel 1

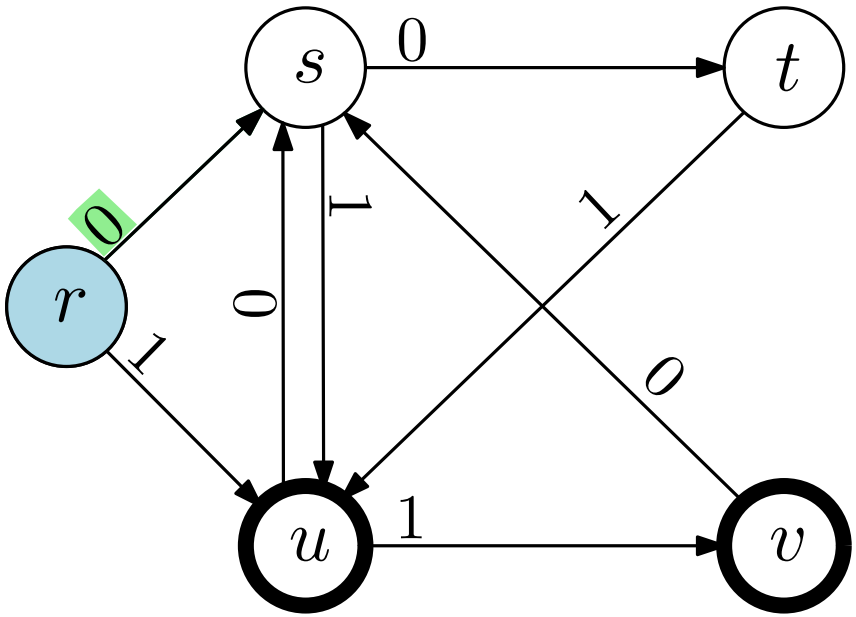


Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101

Exempel 1

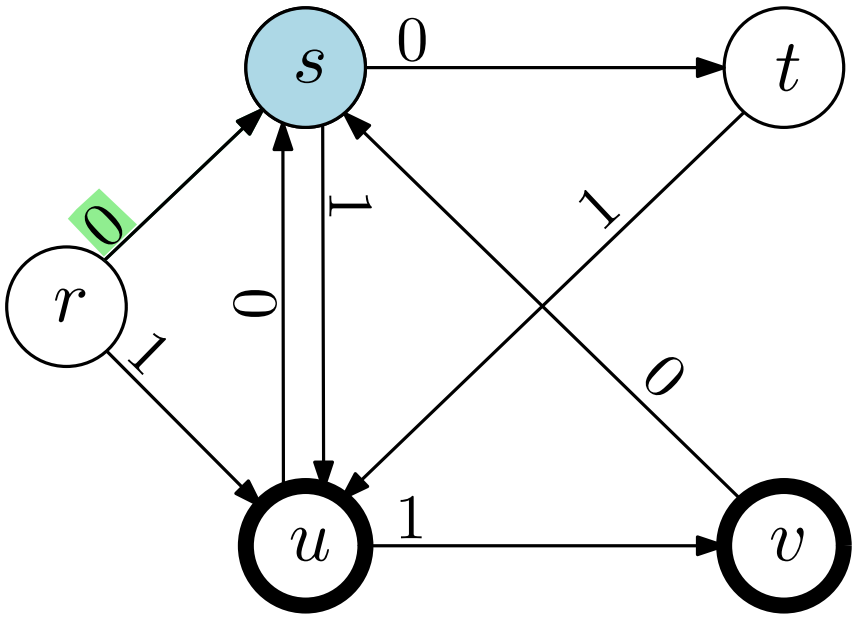


Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101

Exempel 1

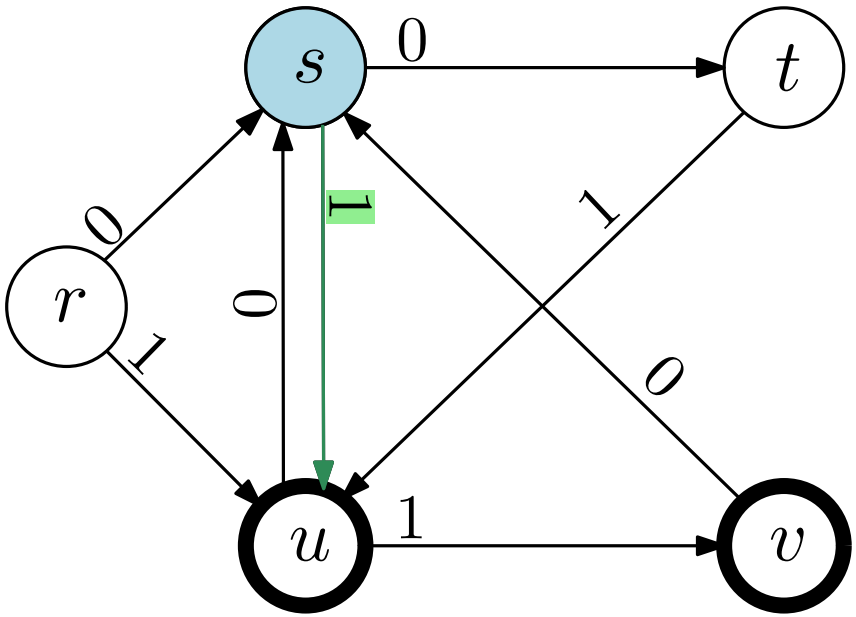


Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101

Exempel 1

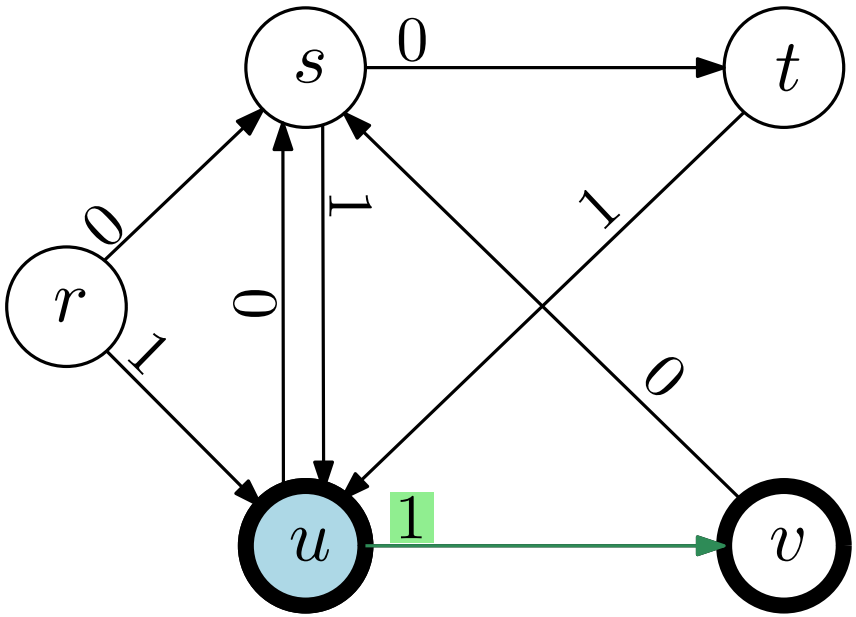


Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101

Exempel 1

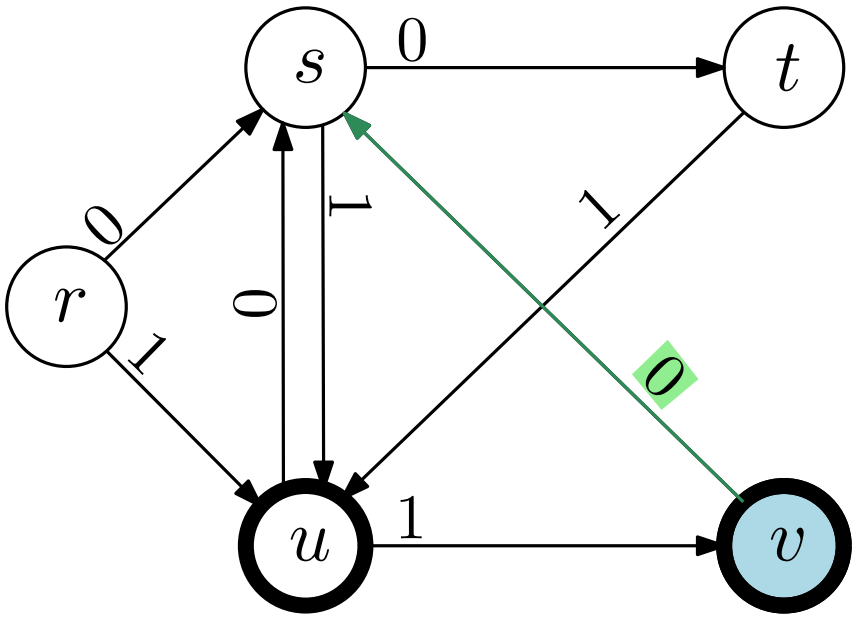


Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101

Exempel 1

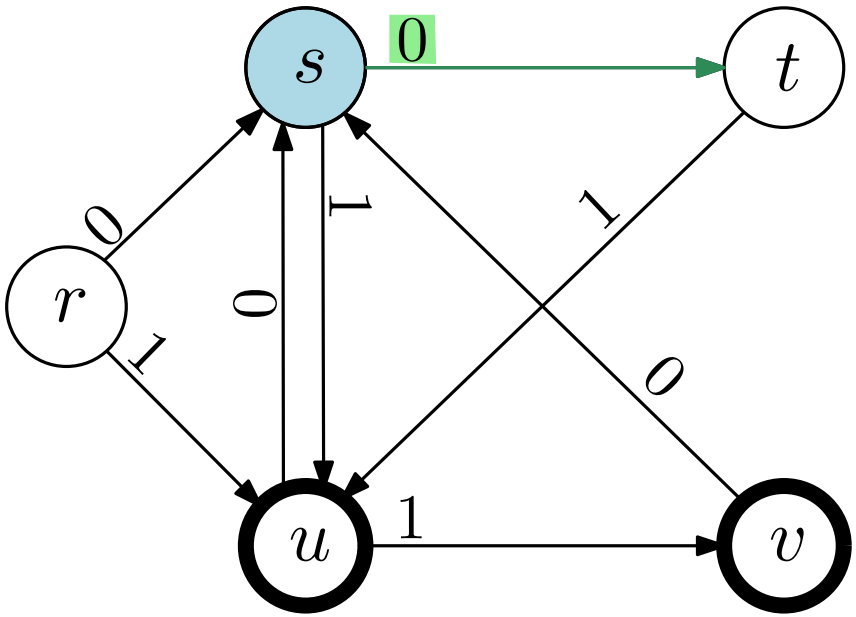


Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101

Exempel 1

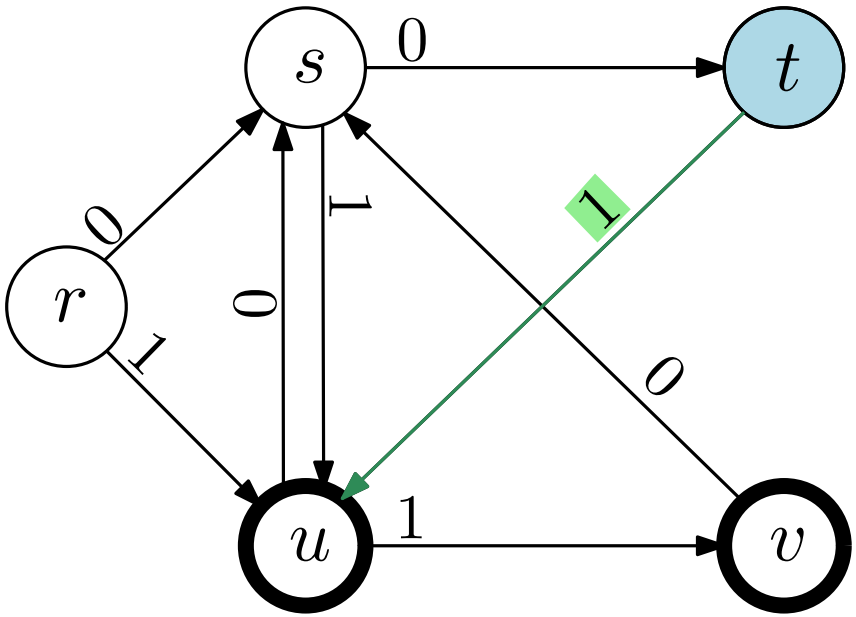


Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101

Exempel 1

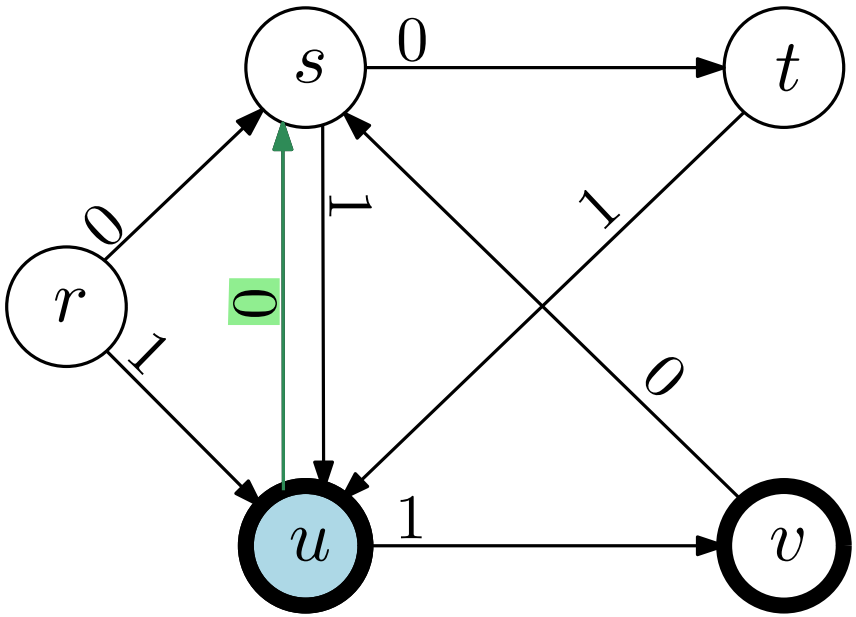


Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101

Exempel 1

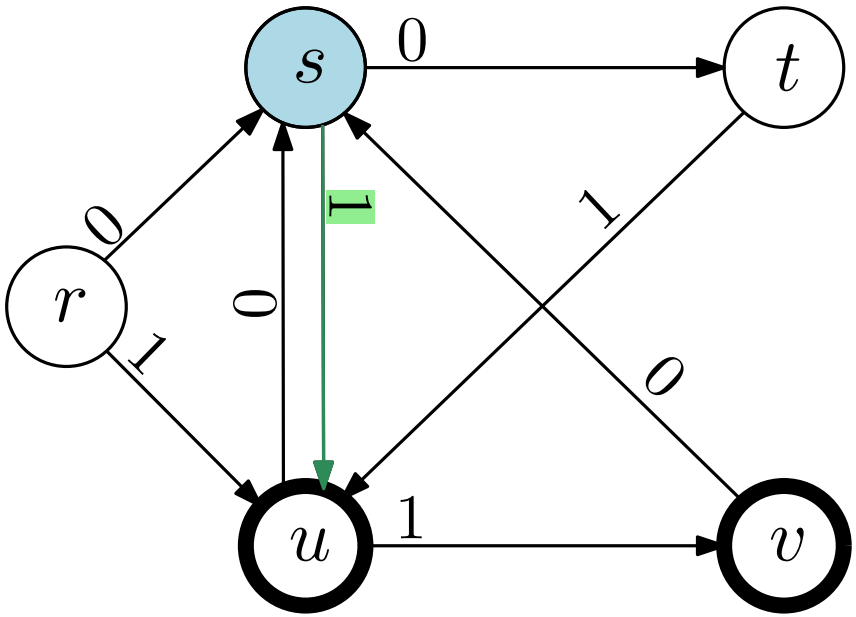


Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101

Exempel 1

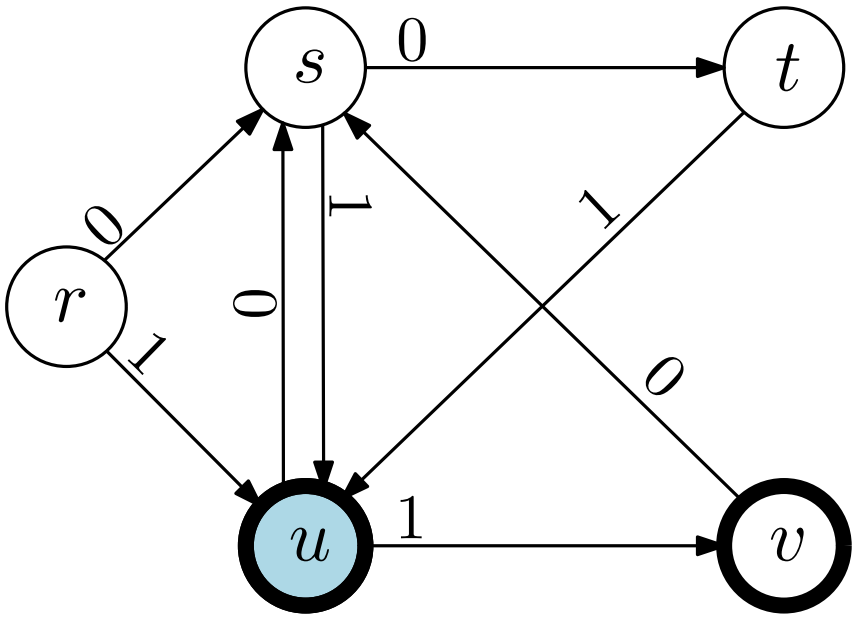


Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101

Exempel 1

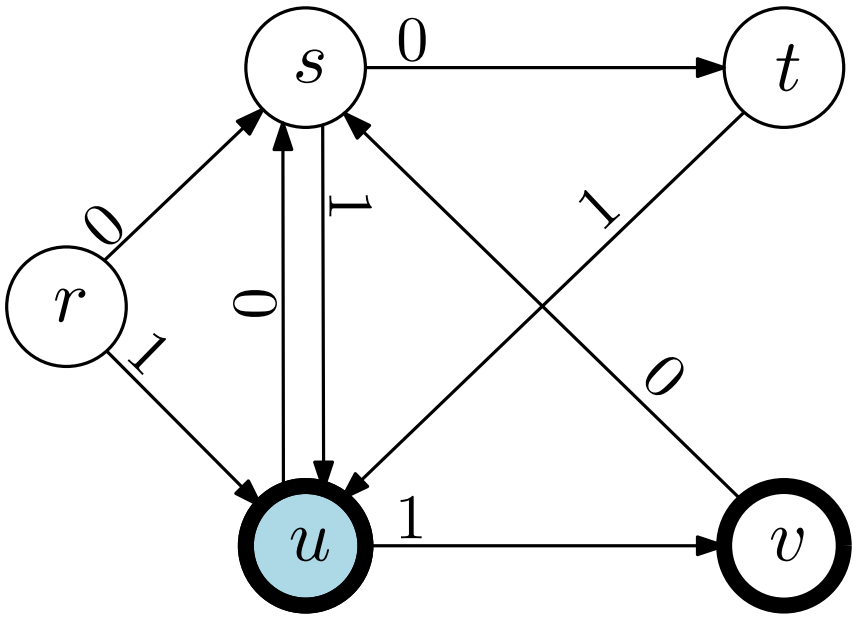


Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101

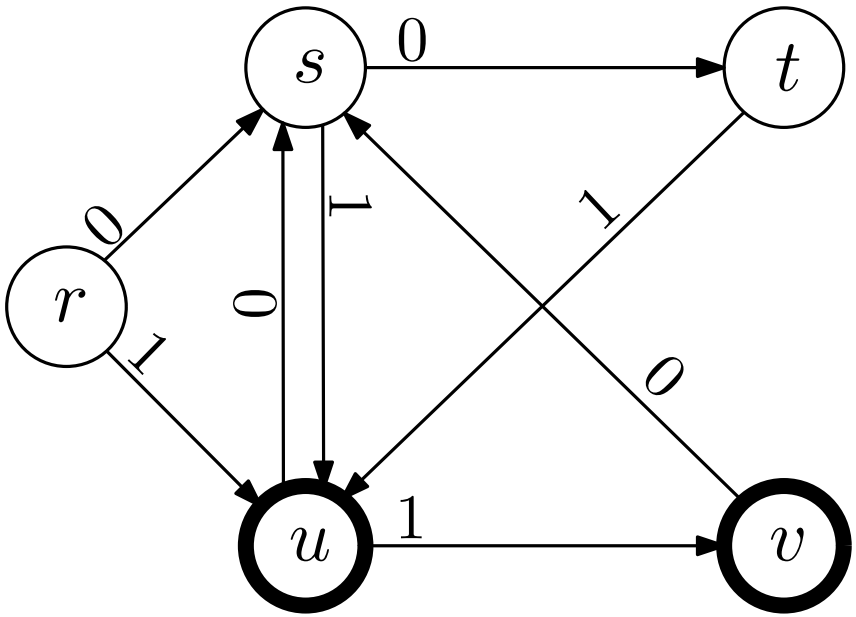
Exempel 1



Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Exempel 1

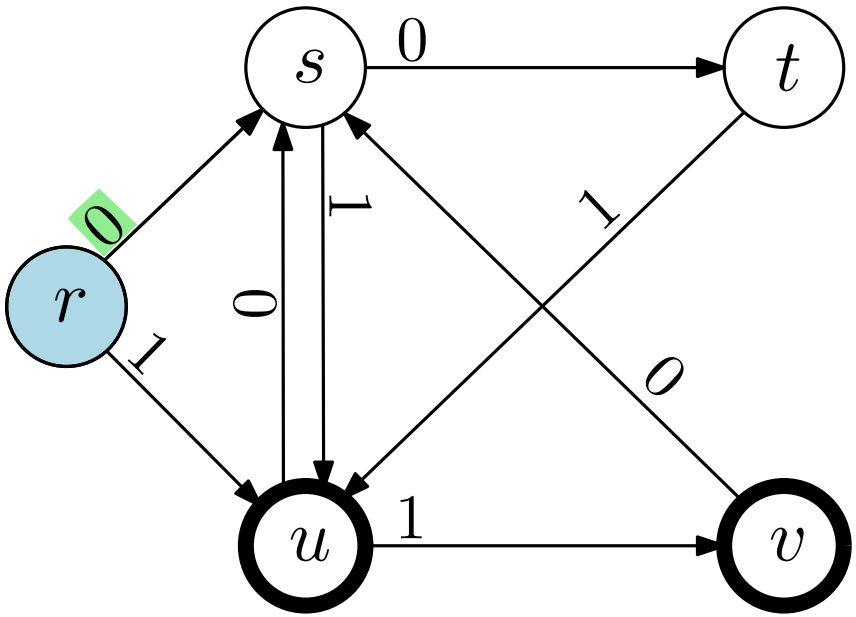


Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101

Exempel 1

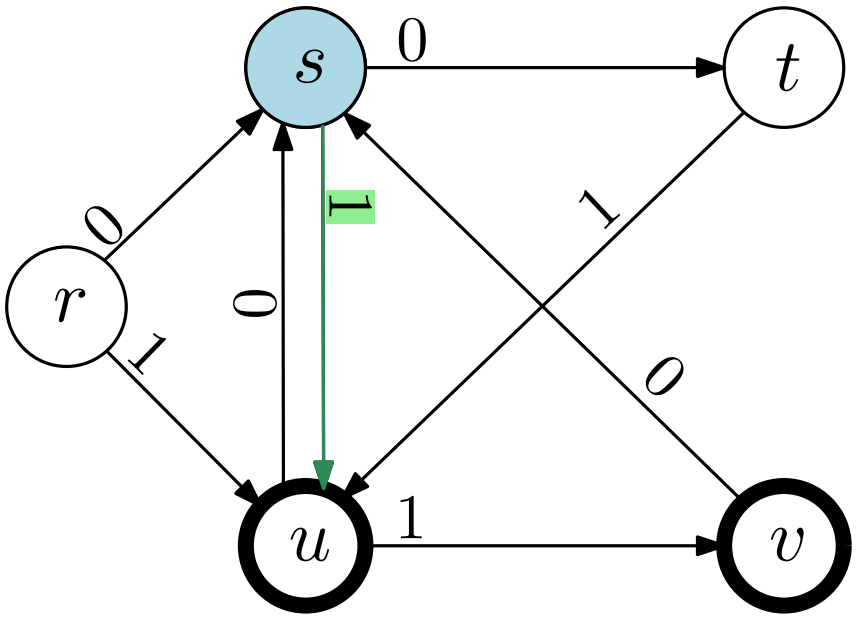


Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101

Exempel 1

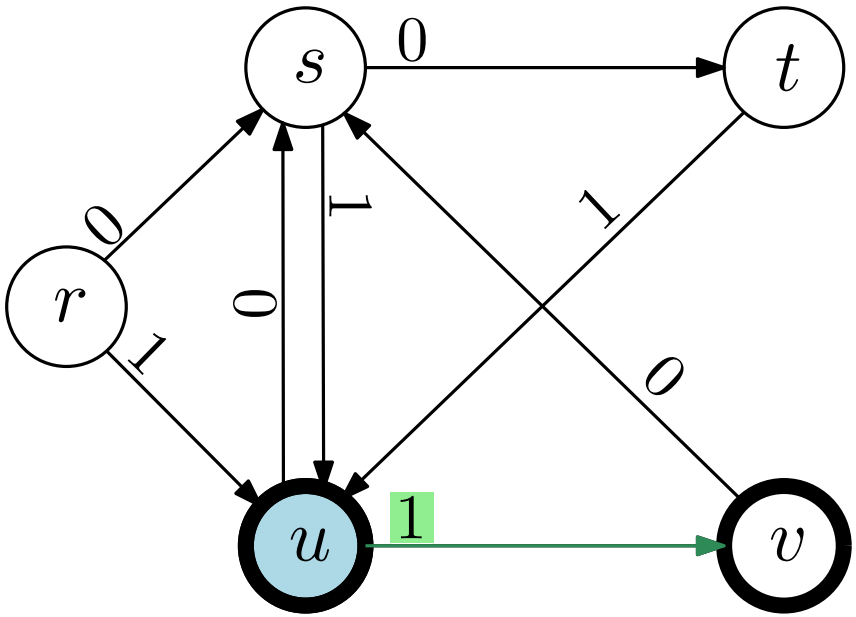


Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101

Exempel 1

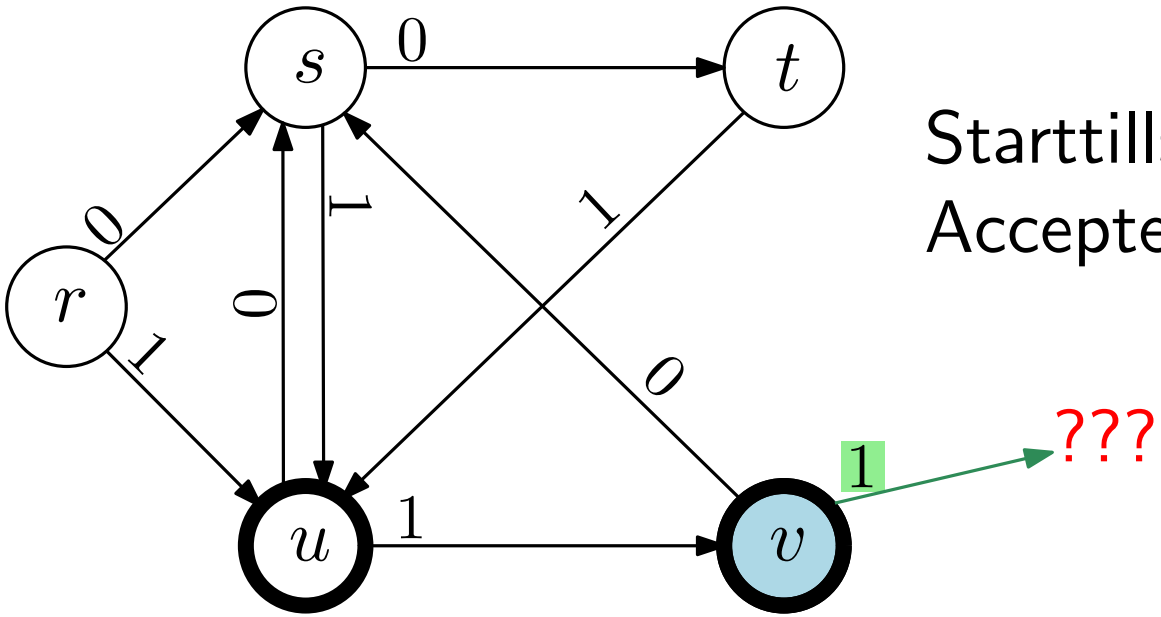


Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101

Exempel 1



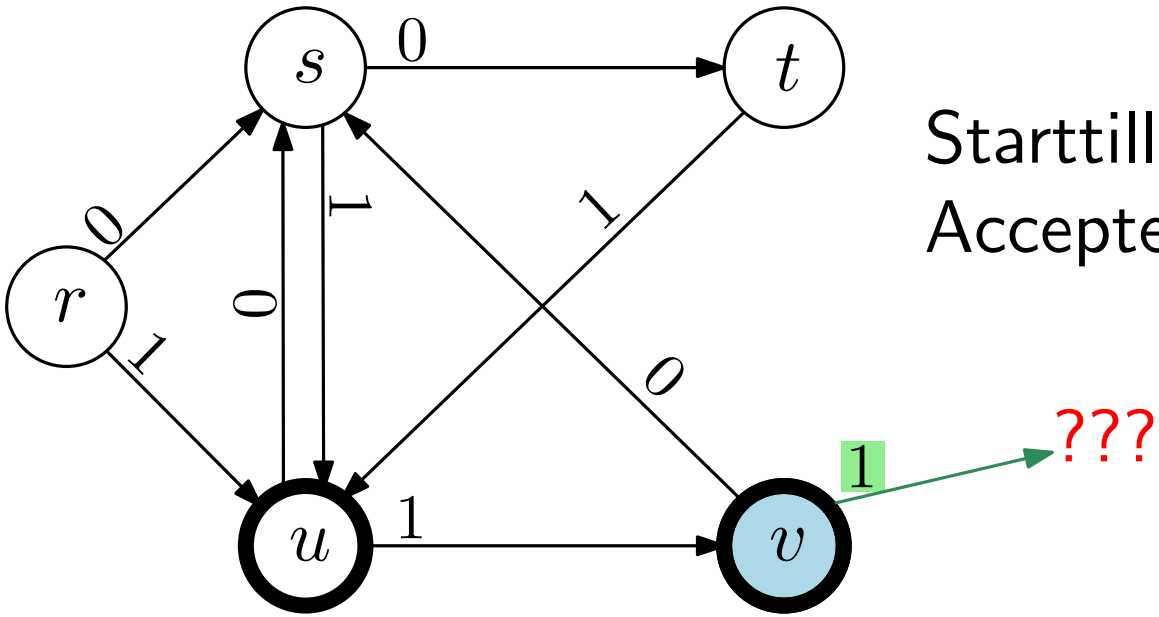
Starttillstånd r

Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101

Exempel 1

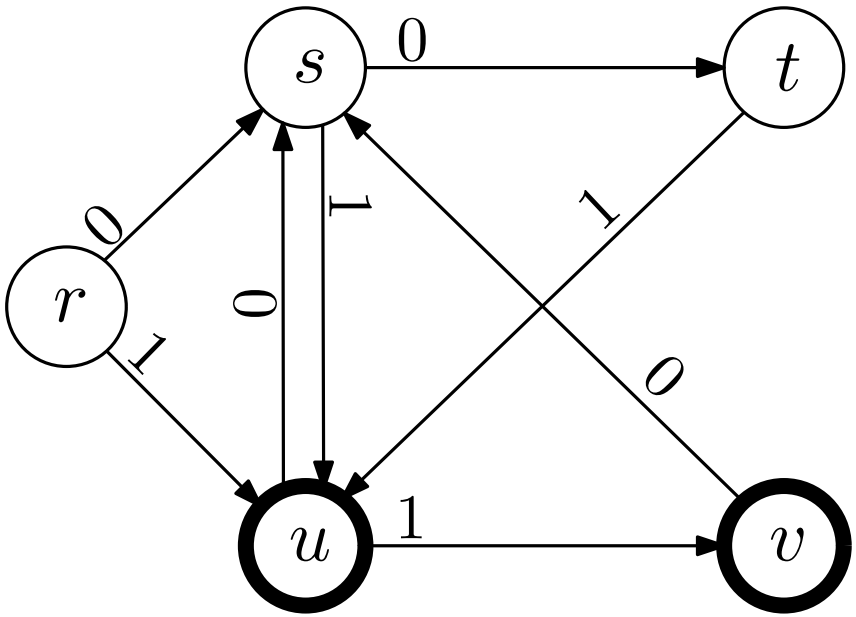


Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101 Övergång saknas – strängen accepteras ej
(underförstått: implicit 1-övergång från v till ett "fail"-tillstånd där vi ligger och snurrar och aldrig kan komma tillbaka till ett accepterande tillstånd)

Exempel 1



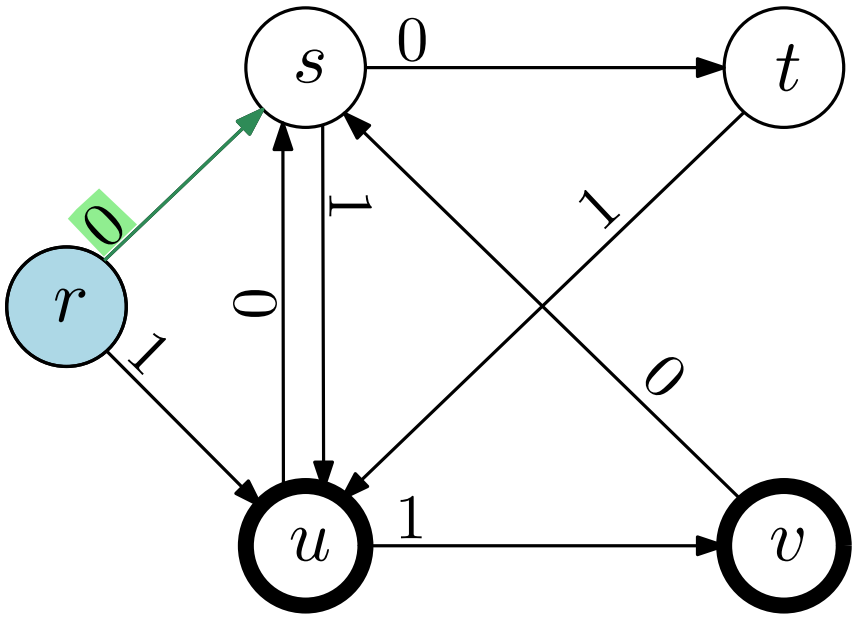
Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101 Övergång saknas – strängen accepteras ej

Indata: 01100100

Exempel 1



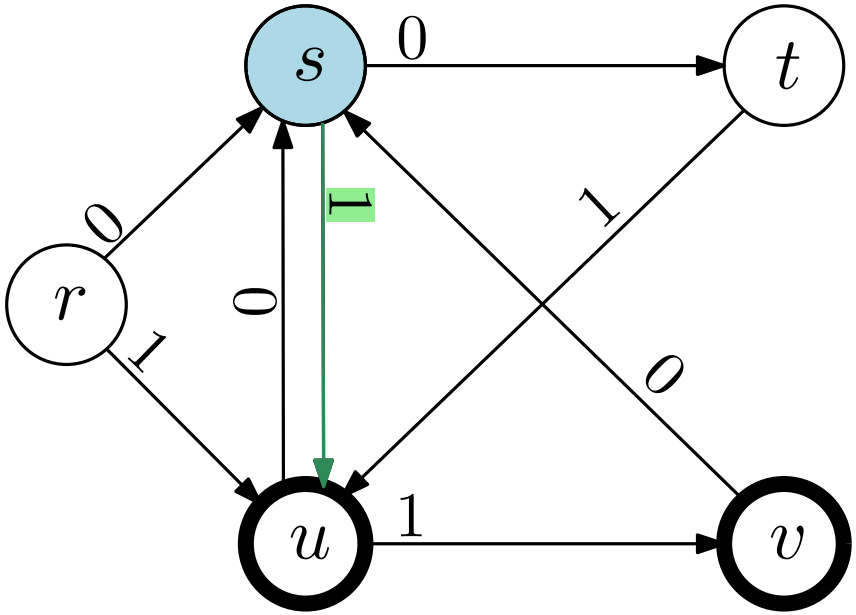
Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101 Övergång saknas – strängen accepteras ej

Indata: 01100100

Exempel 1



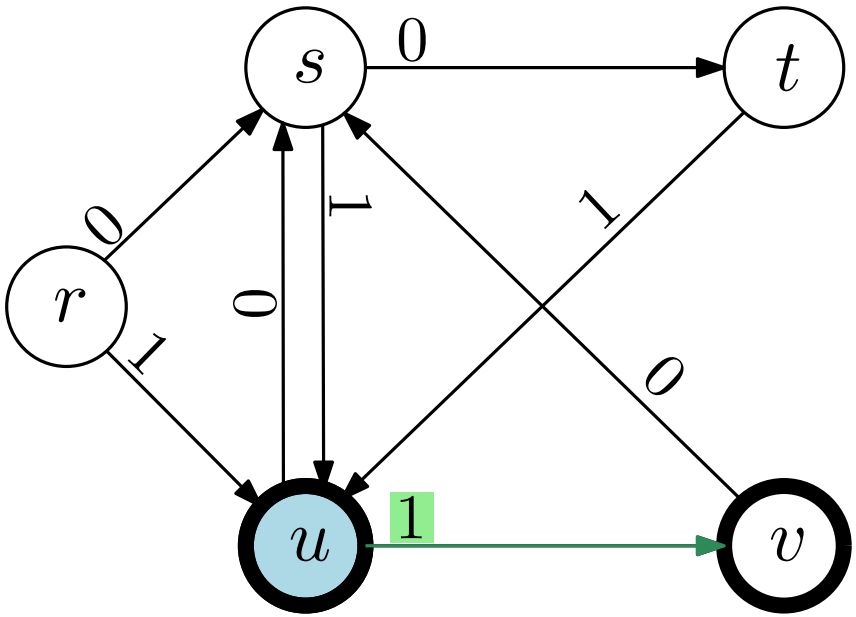
Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101 Övergång saknas – strängen accepteras ej

Indata: 01100100

Exempel 1



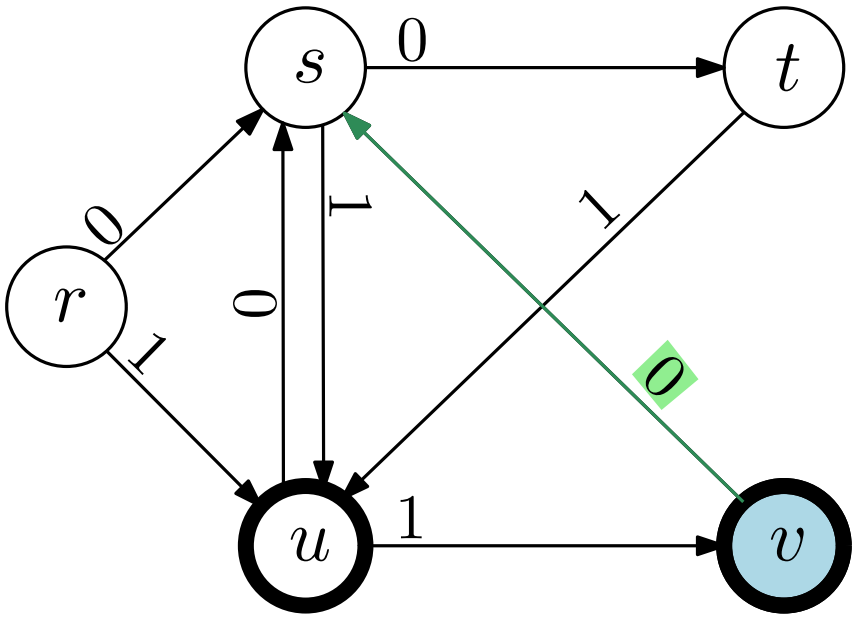
Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101 Övergång saknas – strängen accepteras ej

Indata: 01100100

Exempel 1



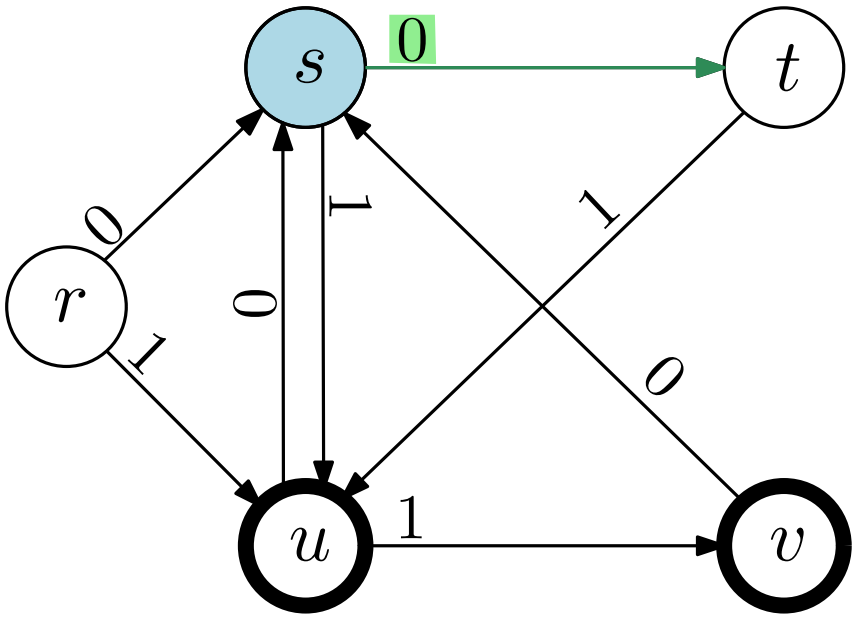
Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101 Övergång saknas – strängen accepteras ej

Indata: 01100100

Exempel 1



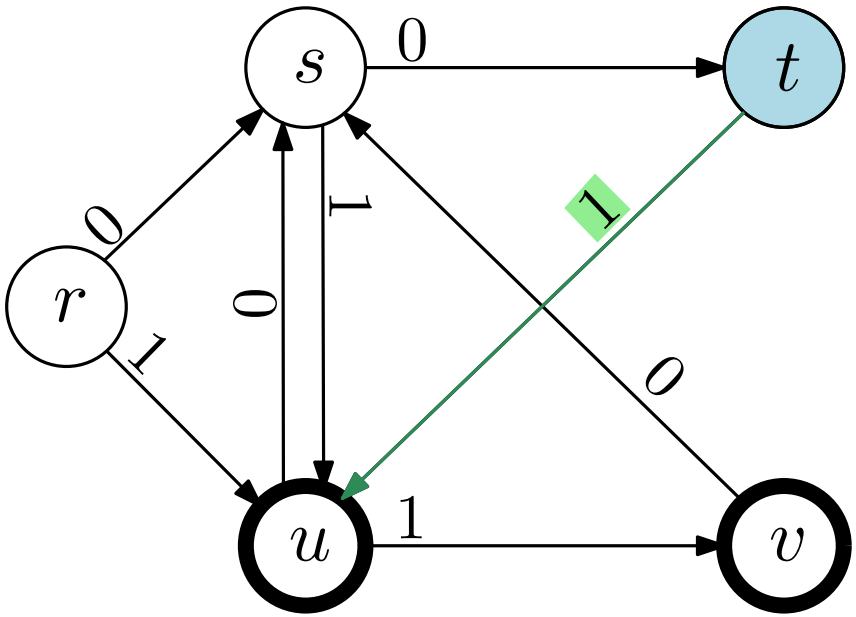
Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101 Övergång saknas – strängen accepteras ej

Indata: 01100100

Exempel 1



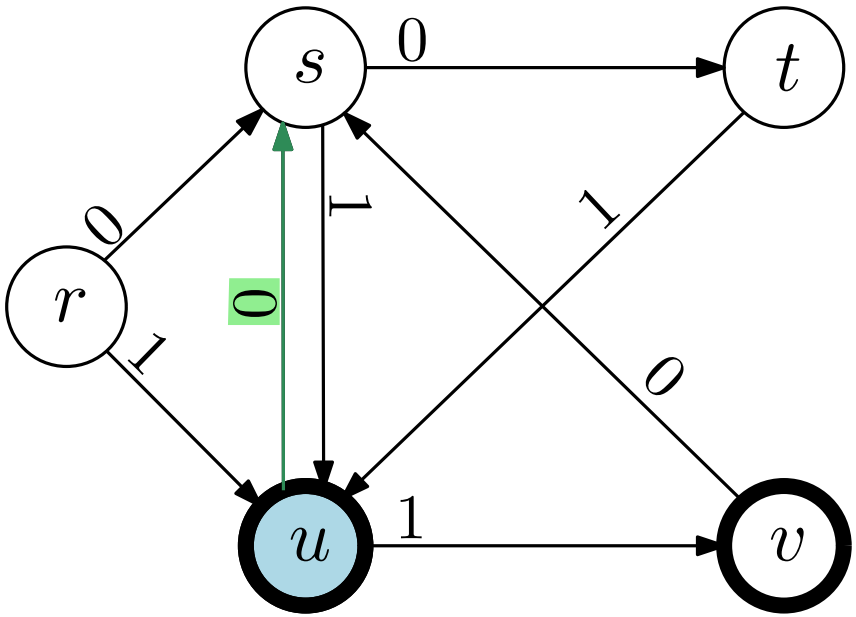
Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101 Övergång saknas – strängen accepteras ej

Indata: 01100100

Exempel 1



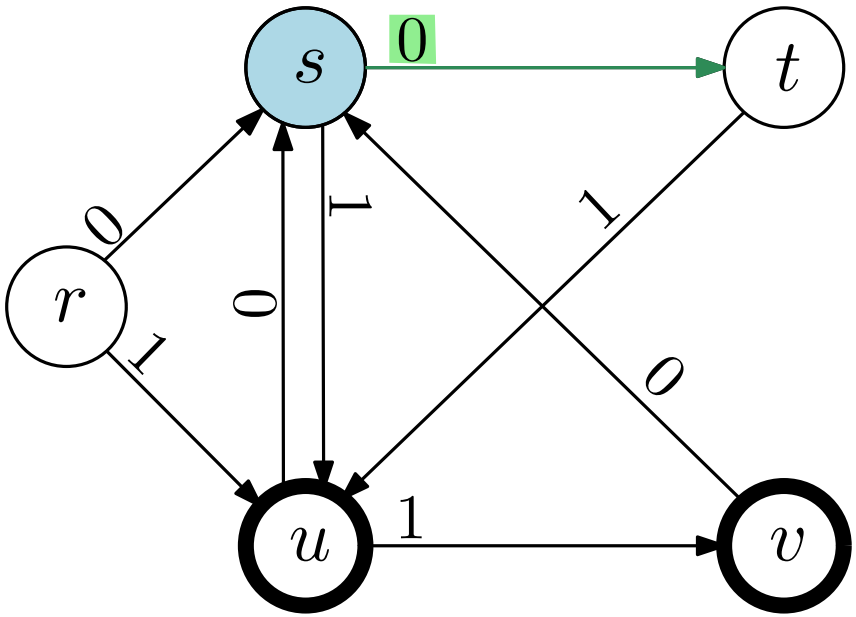
Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101 Övergång saknas – strängen accepteras ej

Indata: 01100100

Exempel 1



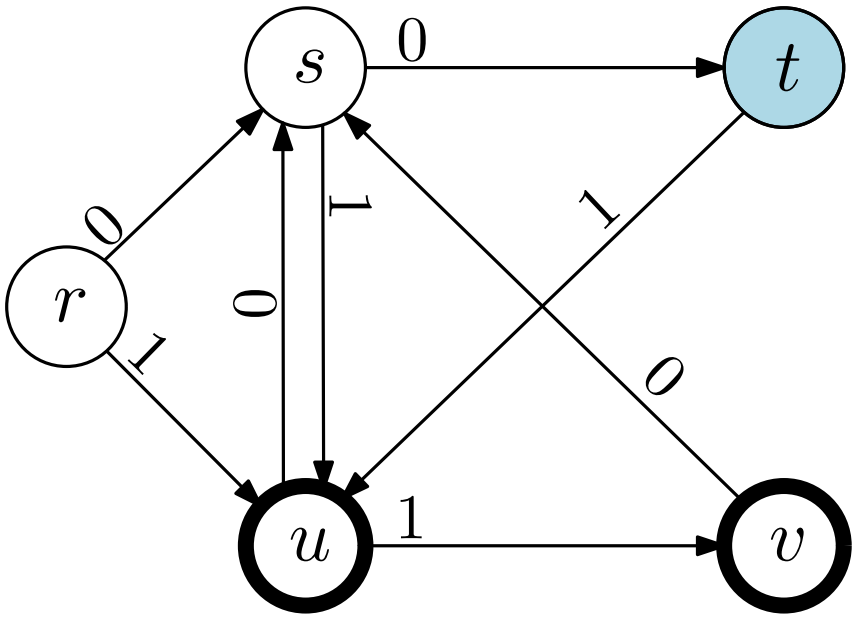
Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101 Övergång saknas – strängen accepteras ej

Indata: 01100100

Exempel 1



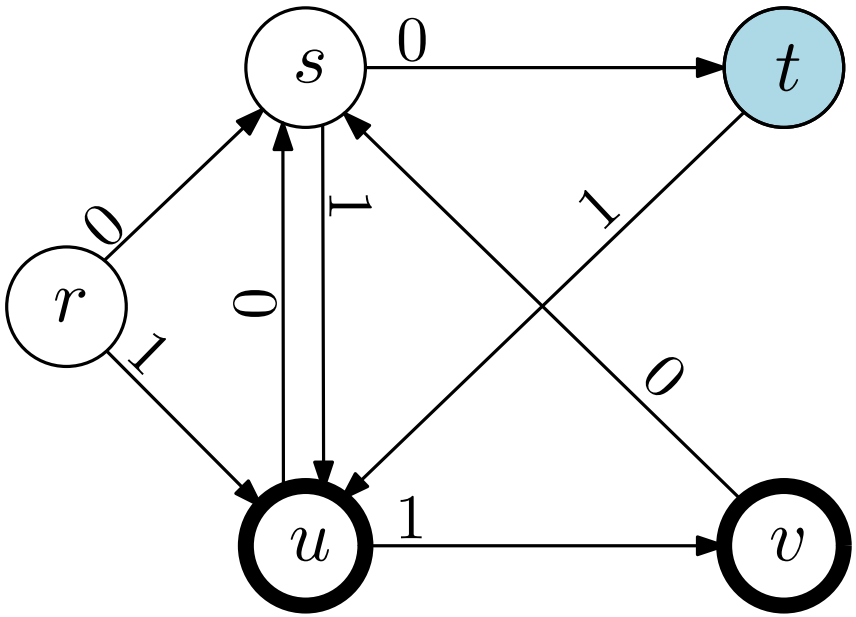
Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101 Övergång saknas – strängen accepteras ej

Indata: 01100100

Exempel 1



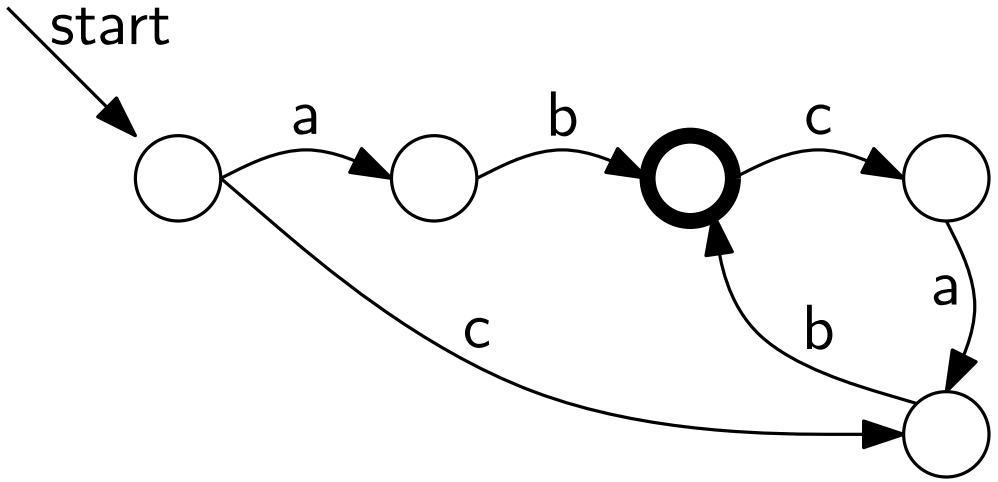
Starttillstånd r
Accepterande tillstånd u och v

Indata: 01100101 Slutar i tillstånd u – strängen accepteras

Indata: 01110101 Övergång saknas – strängen accepteras ej

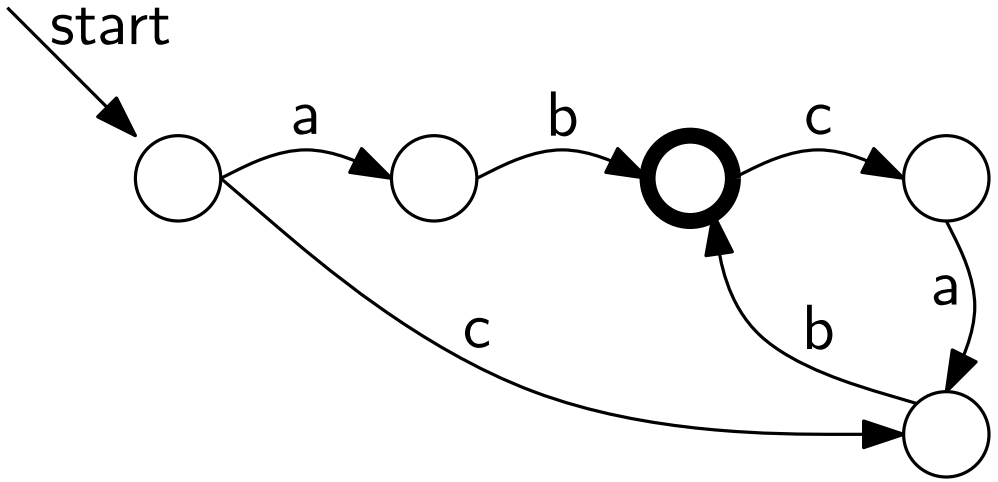
Indata: 01100100 Slutar i tillstånd t – strängen accepteras ej

Exempel 2



Hur hitta några exempel på strängar som automaten accepterar?

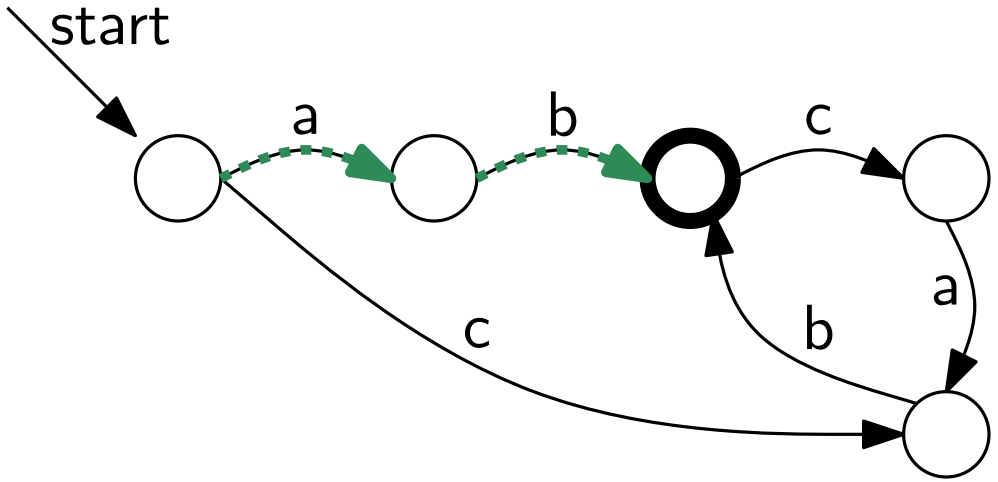
Exempel 2



Hur hitta några exempel på strängar som automaten accepterar?

Följ pilarna och hitta en väg till ett accepterande tillstånd!

Exempel 2

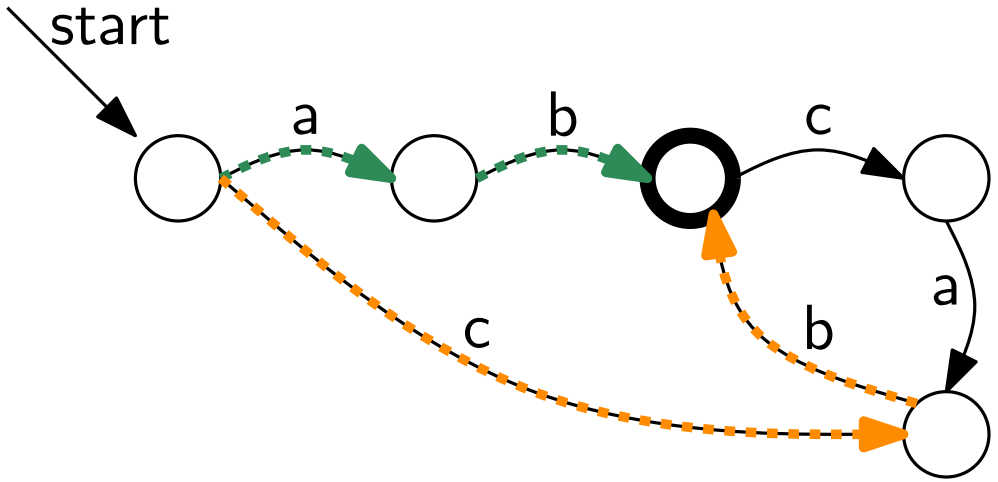


Hur hitta några exempel på strängar som automaten accepterar?

Följ pilarna och hitta en väg till ett accepterande tillstånd!

Automaten accepterar:
`ab`

Exempel 2



Hur hitta några exempel på strängar som automaten accepterar?

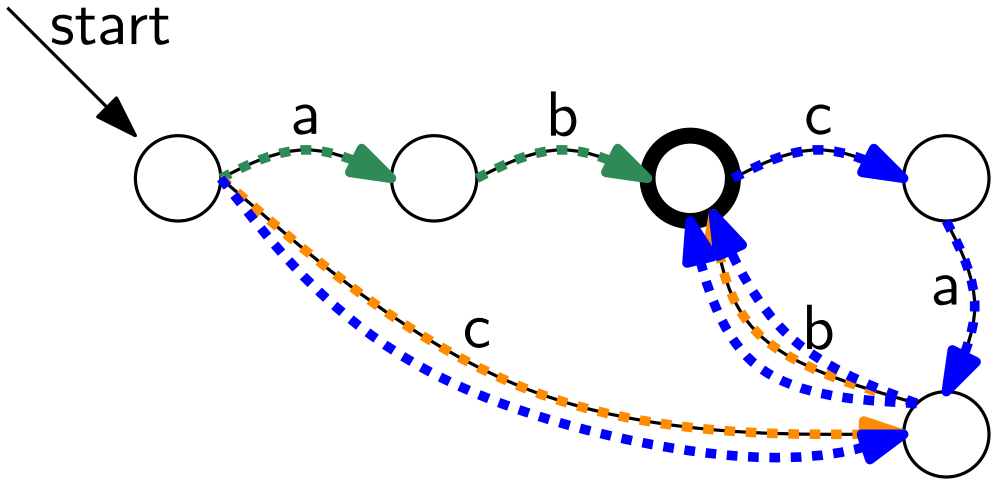
Följ pilarna och hitta en väg till ett accepterande tillstånd!

Automaten accepterar:

`ab`

`cb`

Exempel 2



Hur hitta några exempel på strängar som automaten accepterar?

Följ pilarna och hitta en väg till ett accepterande tillstånd!

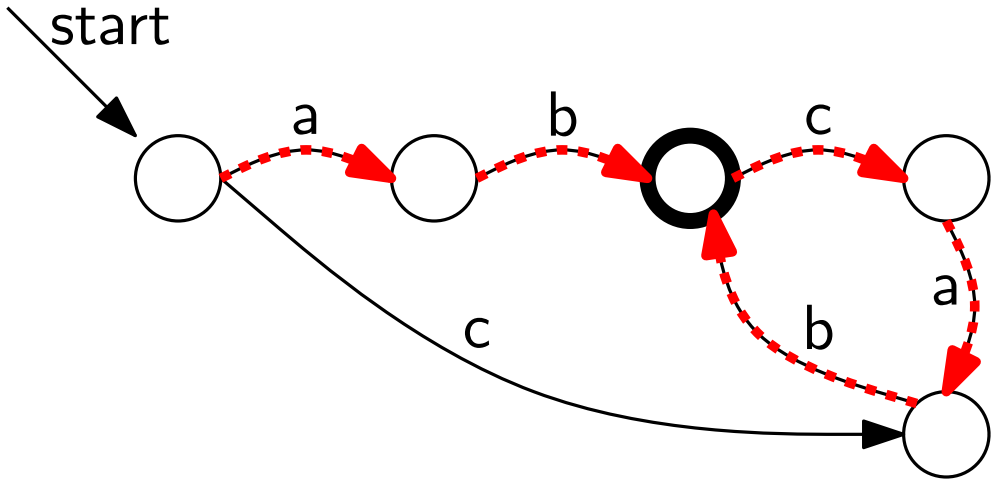
Automaten accepterar:

`ab`

`cb`

`cbcab`

Exempel 2



Hur hitta några exempel på strängar som automaten accepterar?

Följ pilarna och hitta en väg till ett accepterande tillstånd!

Automaten accepterar:

ab

cb

cbcab

abcab

abcabcabcabcabcab

...

Idag

Reguljära uttryck

Ändliga automater

Formella språk

Några praktiska exempel

Formella språk

Ett *formellt språk* L är en mängd strängar.

- $L = \{\text{java, haskell, prolog}\}$
- 8-bitars binära tal:
  $L = \{0, 1\}^8$
- Alla binära strängar med udda paritet:
  $L = \{w \in \{0, 1\}^* \mid \text{ udda antal ettor i } w\}$
- Alla primtal skrivna som siffersträngar i bas 10,
  $L = \{2, 3, 5, 7, 11, \ldots\}$
- Satisfierbara Booleska formler
- Syntaktiskt korrekta Javaprogram

Att beskriva ett formellt språk

Ett formellt språk kan definieras / specificeras / beskrivas på många olika sätt:

Att beskriva ett formellt språk

Ett formellt språk kan definieras / specificeras / beskrivas på många olika sätt:

- Räkna upp strängarna i språket (om språket är ändligt)
  $L = \{\mathsf{Inet}, \mathsf{F1}, \mathsf{F2}, \mathsf{L1}, \mathsf{L2}, \mathsf{S1}, \mathsf{S2}\}$

Att beskriva ett formellt språk

Ett formellt språk kan definieras / specificeras / beskrivas på många olika sätt:

- Räkna upp strängarna i språket (om språket är ändligt)

  $L = \{\mathsf{Inet}, \mathsf{F1}, \mathsf{F2}, \mathsf{L1}, \mathsf{L2}, \mathsf{S1}, \mathsf{S2}\}$
- Ge regler för hur strängar som tillhör språket måste se ut

  `Inet | [FLS] [12]`

  `[A-Za-z0-9]([A-Za-z0-9.]*[A-Za-z0-9])?@[a-z0-9]+(\.[a-z0-9]+)+`

Att beskriva ett formellt språk

Ett formellt språk kan definieras / specificeras / beskrivas på många olika sätt:

- Räkna upp strängarna i språket (om språket är ändligt)

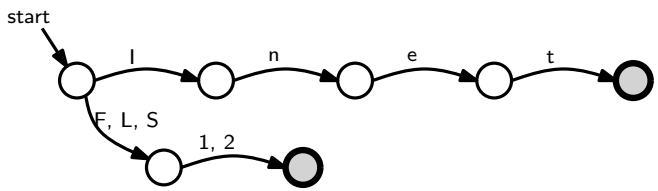
  $L = \{\text{Inet}, \text{F1}, \text{F2}, \text{L1}, \text{L2}, \text{S1}, \text{S2}\}$

- Ge regler för hur strängar som tillhör språket måste se ut

  `Inet | [FLS] [12]`

  `[A-Za-z0-9]([A-Za-z0-9.]*[A-Za-z0-9])?@[a-z0-9]+(\.[a-z0-9]+)+`

- Konstruera en algoritm som avgör huruvida en sträng tillhör språket eller inte (dvs "känner igen" språket)




```
for i = 2 to n-1
   if n mod i == 0
      return false
return true
```

Att beskriva ett formellt språk

Ett formellt språk kan definieras / specificeras / beskrivas på många olika sätt:

- Räkna upp strängarna i språket (om språket är ändligt)

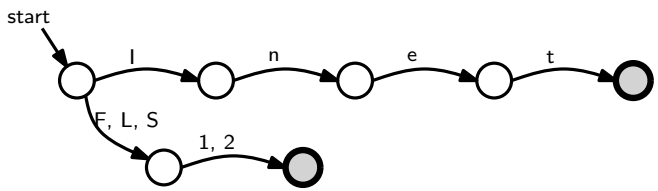
  $L = \{\text{Inet}, \text{F1}, \text{F2}, \text{L1}, \text{L2}, \text{S1}, \text{S2}\}$

- Ge regler för hur strängar som tillhör språket måste se ut

  `Inet | [FLS] [12]`

  `[A-Za-z0-9]([A-Za-z0-9.]*[A-Za-z0-9])?@[a-z0-9]+(\.[a-z0-9]+)+`

- Konstruera en algoritm som avgör huruvida en sträng tillhör språket eller inte (dvs "känner igen" språket)




```
for i = 2 to n-1
   if n mod i == 0
      return false
return true
```

- Ge en informell definition

  "mängden av alla Java-program som har en oändlig loop"

Den teoretiska delen om formella språk

Hur går man från en informell eller icke-konstruktiv definition till regler eller algoritmer för att känna igen språket?

Den teoretiska delen om formella språk

Hur går man från en informell eller icke-konstruktiv definition till regler eller algoritmer för att känna igen språket?

T.ex.:

- Vilka typer av språk kan man beskriva med ett reguljärt uttryck?
- Vad är den effektivaste algoritmen för att avgöra om ett tal är ett primtal?

Den teoretiska delen om formella språk

Hur går man från en informell eller icke-konstruktiv definition till regler eller algoritmer för att känna igen språket?

T.ex.:

- Vilka typer av språk kan man beskriva med ett reguljärt uttryck?
- Vad är den effektivaste algoritmen för att avgöra om ett tal är ett primtal?

Detta är grunden till forskningsområdet "teoretisk datalogi"

(Den typ av frågor vi kommer titta på i den här kursen är grundläggande frågor som man löst för 50+ år sedan och inte så forskningsnära.)

Idag

Reguljära uttryck

Ändliga automater

Formella språk

Reguljära uttryck i praktiken

Reguljära uttryck i praktiken – Regex

Det finns många olika mindre variationer på syntax för reguljära uttryck mellan olika implementationer

Reguljära uttryck i praktiken – Regex

Det finns många olika mindre variationer på syntax för reguljära uttryck mellan olika implementationer

Minsta gemensamma nämnaren är *POSIX-standarden* för reguljära uttryck.

Reguljära uttryck i praktiken – Regex

Det finns många olika mindre variationer på syntax för reguljära uttryck mellan olika implementationer

Minsta gemensamma nämnaren är *POSIX-standarden* för reguljära uttryck.

Det är också vanligt med extra funktionalitet som inte kan simuleras med “riktiga” reguljära uttryck

Reguljära uttryck i praktiken – Regex

Det finns många olika mindre variationer på syntax för reguljära uttryck mellan olika implementationer

Minsta gemensamma nämnaren är *POSIX-standarden* för reguljära uttryck.

Det är också vanligt med extra funktionalitet som inte kan simuleras med "riktiga" reguljära uttryck

Typ-exempel: *bakåtreferenser* `(.*)(.*)\2\1`

Reguljära uttryck i praktiken – Regex

Det finns många olika mindre variationer på syntax för reguljära uttryck mellan olika implementationer

Minsta gemensamma nämnaren är *POSIX-standarden* för reguljära uttryck.

Det är också vanligt med extra funktionalitet som inte kan simuleras med "riktiga" reguljära uttryck

Typ-exempel: *bakåtreferenser* `(.*)(.*)\2\1`

$\backslash n$ betyder "samma sträng som vi matchade med den n:te parentes-gruppen"

Reguljära uttryck i praktiken – Regex

Det finns många olika mindre variationer på syntax för reguljära uttryck mellan olika implementationer

Minsta gemensamma nämnaren är *POSIX-standarden* för reguljära uttryck.

Det är också vanligt med extra funktionalitet som inte kan simuleras med "riktiga" reguljära uttryck

Typ-exempel: *bakåtreferenser* `(.*)(.*)\2\1`

$\backslash n$ betyder "samma sträng som vi matchade med den n:te parentes-gruppen"

Matchar t.ex. `abba`, `erbananerb`

Reguljära uttryck i praktiken – Regex

Det finns många olika mindre variationer på syntax för reguljära uttryck mellan olika implementationer

Minsta gemensamma nämnaren är *POSIX-standarden* för reguljära uttryck.

Det är också vanligt med extra funktionalitet som inte kan simuleras med "riktiga" reguljära uttryck

Typ-exempel: *bakåtreferenser* `(.*)(.*)\2\1`

Tyvärr ganska stor begreppsförvirring: "reguljärt uttryck" kan betyda olika beroende på om man åsyftar teorin för formella språk, eller det praktiska verktyget för strängmatchning

Terminologi i kursen:
Det vi beskrev tidigare = "reguljärt uttryck"
Med utökningar som bakåtreferenser = "regex"

Regex i Unix

```
> grep '<regex>' <fil>
```

visa alla rader i `<fil>` där något som matchar `<regex>` förekommer (variant: `egrep`, använder annan regex-dialekt)

Regex i Unix

`> grep '<regex>' <fil>`

visa alla rader i `<fil>` där något som matchar `<regex>` förekommer (variant: `egrep`, använder annan regex-dialekt)

Exempel:

```
> grep "a..str" pride_and_prejudice.txt
united, with great strength of feeling, a composure of temper and a
commission of the peace of the county, she was a most active magistrate
of his mind was visible in every feature.  He was struggling for the
am strongly inclined to hope the best.  Could he expect that her friends
```

Regex i Unix

`> grep '<regex>' <fil>`

visa alla rader i `<fil>` där något som matchar `<regex>` förekommer (variant: `egrep`, använder annan regex-dialekt)

Exempel:

```
> grep "a..str" pride_and_prejudice.txt
united, with great strength of feeling, a composure of temper and a
commission of the peace of the county, she was a most active magistrate
of his mind was visible in every feature.  He was struggling for the
am strongly inclined to hope the best.  Could he expect that her friends
```

`> sed 's/<regex>/<replacement>/g' < <fil>`

byter ut alla förekomster av `<regex>` mot `<replacement>`

Regex i Unix

`> grep '<regex>' <fil>`

visa alla rader i `<fil>` där något som matchar `<regex>` förekommer (variant: `egrep`, använder annan regex-dialekt)

Exempel:

```
> grep "a..str" pride_and_prejudice.txt
united, with great strength of feeling, a composure of temper and a
commission of the peace of the county, she was a most active magistrate
of his mind was visible in every feature.  He was struggling for the
am strongly inclined to hope the best.  Could he expect that her friends
```

`> sed 's/<regex>/<replacement>/g' < <fil>`

byter ut alla förekomster av `<regex>` mot `<replacement>`

Exempel:

```
sed 's/Darcy/Austrin/g' < pride_and_prejudice.txt > my_novel.txt
```

Regex i Unix

`> grep '<regex>' <fil>`

visa alla rader i `<fil>` där något som matchar `<regex>` förekommer (variant: `egrep`, använder annan regex-dialekt)

Exempel:

```
> grep "a..str" pride_and_prejudice.txt
united, with great strength of feeling, a composure of temper and a
commission of the peace of the county, she was a most active magistrate
of his mind was visible in every feature.  He was struggling for the
am strongly inclined to hope the best.  Could he expect that her friends
```

`> sed 's/<regex>/<replacement>/g' < <fil>`

byter ut alla förekomster av `<regex>` mot `<replacement>`

Exempel:

```
sed 's/Darcy/Austrin/g' < pride_and_prejudice.txt > my_novel.txt
```

...och många fler Unix-verktyg!

Regex i Java

Finns i paketet `java.util.regex`

Regex i Java

Finns i paketet `java.util.regex`

Kolla om en sträng matchar ett uttryck:

```
import java.util.regex.Pattern;

public boolean isValidEmail(String address) {
   return Pattern.matches(regex_for_email, address);
}
```

Regex i Java

Finns i paketet `java.util.regex`

Kolla om en sträng matchar ett uttryck:

```
import java.util.regex.Pattern;

public boolean isValidEmail(String address) {
   return Pattern.matches(regex_for_email, address);
}
```

Ersätta hundar med katter:

```
import java.util.regex.*;

Pattern p = Pattern.compile("cat");
Matcher m = p.matcher("one cat, two cats in the yard");
String s = m.replaceAll("dog");
// --> s = "one dog, two dogs in the yard"
```

Regex i Haskell

Ersätta hundar med katter

```
module Replace where
import Text.Regex

replacement = subRegex re s1 "cat"
    where re = mkRegex "dog"
          s1 = "one dog, two dogs in the yard"
```

Regex i Haskell

Ersätta hundar med katter

```
module Replace where
import Text.Regex

replacement = subRegex re s1 "cat"
   where re = mkRegex "dog"
         s1 = "one dog, two dogs in the yard"
```

Provkörning

```
> ghci Replace.hs
...
*Replace> replacement
...
"one cat, two cats in the yard"
```

Regex i Haskell, forts

Dela upp en text med avseende på ett regex

```
module Split where
import Text.Regex

split s = splitRegex nonletters s
   where nonletters = mkRegex "[^a-zA-Z]+"
```

Regex i Haskell, forts

Dela upp en text med avseende på ett regex

```
module Split where
import Text.Regex

split s = splitRegex nonletters s
   where nonletters = mkRegex "[^a-zA-Z]+"
```

Provkörning

```
> ghci Split.hs
...
*Split> split "One, two, three, and 4 balloons!"
...
["One","two","three","and","balloons", ""]
```

Regex i andra språk

Vi skulle kunna hålla på och gå igenom hur man använder regex i olika språk i flera timmar, men det blir snabbt tråkigt.

När man vill använda regex i sitt favorit-språk kollar man dokumentationen för språket hur regex fungerar där, exakt vilken notation som används osv

(typiskt brukar det gå bra att googla på "`<programspråk> regular expressions`"...)

På torsdag

Mer om reguljära uttryck och ändliga automater:
ekvivalens och begränsningar

Mer kraftfulla sätt att beskriva språk:
grammatiker och stackautomater