6 Reguljära uttryck

I unix-skal finns ange enkla mönster för filnamn med * och ?. En del program, t ex emacs, egrep och sed, erbjuder möjlighet att söka efter strängar som matchar mer komplicerade mönster som beskrivs med *reguljära uttryck*. I Java-paketet `java.util.regex` finns också klasser som tillhandahåller sådan funktionalitet.

Ett reguljärt uttryck beskriver ett språk och de nämnda programmen kan alltså avgöra om en given sträng tillhör språket. T ex så beskriver utrycket $0 \mid 1 \cdot (0 \mid 1)^\star$ det språk som innehåller alla binära tal utan onödiga inledande nollor, $\{0, 1, 10, 11, 100, \dots\}$. I detta uttryck är 0 och 1 symbolerna i alfabetet medan $|$, $\cdot$ och * är operatorer som kan förekomma i reguljära uttryck. Parenteser används som vanligt för att gruppera.

Definition. Mängden av reguljära uttryck på alfabetet Σ definieras av

$\boldsymbol{\emptyset}$ är ett reguljärt uttryck

$\boldsymbol{\epsilon}$ är ett reguljärt uttryck

om $\sigma \in \Sigma$ så är $\boldsymbol{\sigma}$ ett reguljärt uttryck

om α och β är reguljära uttryck så är $(\alpha \ \cdot \ \beta)$ ett reguljärt uttryck

om α och β är reguljära uttryck så är $(\alpha \ \mid \ \beta)$ ett reguljärt uttryck

om α är ett reguljärt uttryck så är $\alpha^\star$ ett reguljärt uttryck

$\triangle$

Vi har använt ett tjockare typsnitt i $\boldsymbol{\emptyset}$ och $\boldsymbol{\epsilon}$ för att göra det tydligt att de är reguljära uttryck och inte den tomma mängden och den tomma strängen. På samma sätt gör vi inledningsvis när en symbol i alfabetet Σ används i ett reguljärt uttryck.

När vi skriver reguljära uttryck exakt enligt denna definition säger vi att de är på *kanonisk* form.

Exempel.

$$\boldsymbol{\emptyset} \quad \mathbf{a} \qquad (\mathbf{a} \cdot \mathbf{b}) \quad (((\mathbf{a} \mid \mathbf{b}) \cdot \mathbf{a}) \cdot (\mathbf{b} \cdot \mathbf{a})^\star)$$

är reguljära uttryck på alfabetet $\{a, b\}$. $\Diamond$

Den normala konventionen är att * har högst precedens följt av $\cdot$ och lägst $|$ när man utelämnar parenteser och man inte skriver ut $\cdot$. Det sista uttrycket kan alltså skrivas $((a \mid b)a)(ba)^\star$. När vi har definierat hur man räknar ut värdet av ett reguljärt uttryck kommer vi att se att både $\cdot$ och $|$ är associativa så att vi kan undvara ytterligare några parenteser, $(a \mid b)a(ba)^\star$.

6.1 Semantik

Värdet av ett reguljärt uttryck är ett språk. Ordet *semantik* betyder 'mening' eller 'betydelse'. För att tydligt skilja på ett reguljärt uttryck och uttryckets värde kommer vi i denna kurs skriva $\mathcal{L}(()\alpha)$ när vi menar det språk som är värdet av uttrycket α. Definitionen av $\mathcal{L}()$ är induktiv med ett fall för varje sorts reguljärt uttryck.

Definition.

$$\mathcal{L}(\boldsymbol{\emptyset}) \triangleq \emptyset \qquad\qquad \mathcal{L}(\alpha\,\beta) \triangleq \mathcal{L}(\alpha)\mathcal{L}(\beta)$$

$$\mathcal{L}(\boldsymbol{\epsilon}) \triangleq \{\epsilon\} \qquad\qquad \mathcal{L}(\alpha \mid \beta) \triangleq \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$$

$$\mathcal{L}(\boldsymbol{\sigma}) \triangleq \{\sigma\}, \quad \sigma \in \Sigma \qquad\qquad \mathcal{L}(\alpha^\star) \triangleq (\mathcal{L}(\alpha))^\star$$

$\triangle$

I den första fallet, $\mathcal{L}(\boldsymbol{\emptyset}) \triangleq \emptyset$ är $\boldsymbol{\emptyset}$ i vänsterledet ett reguljärt uttryck medan $\emptyset$ i högerledet är den vanliga tomma mängden.

I det andra fallet är det tydligare att bokstaven epsilon betyder helt olika saker när den används som ett reguljärt uttryck och när den är ett namn på en tomma strängen. Det språk som $\boldsymbol{\epsilon}$ betecknar, $\{\epsilon\}$ är inte särskilt intressant, men $\boldsymbol{\epsilon}$ är praktiskt att kunna användas som en del i mer komplexa reguljära uttryck.

Uttrycket $\boldsymbol{\sigma}$ betyder också ett språk med bara ett element, men är nödvändigt som bas när man skall beskriva större språk.

Med hjälp av | och (den osynliga) $\cdot$ kan man beskriva språk med flera strängar och längre strängar.

Exempel.

$$\mathcal{L}(ab) = \mathcal{L}(a)\mathcal{L}(b) = \{a\}\{b\} = \{ab\}$$
$$\mathcal{L}(a \mid b) = \mathcal{L}(a) \cup \mathcal{L}(b) = \{a\} \cup \{b\} = \{a, b\}$$
$$\mathcal{L}((a \mid b)(a \mid b)) = \mathcal{L}(a \mid b)\mathcal{L}(a \mid b) = \{a, b\}\{a, b\} = \{aa, ab, ba, bb\}$$

$\Diamond$

Stjärnoperatorn ger oss möjlighet att beskriva språk med oändligt många element.

Exempel.

$$\mathcal{L}(a^\star) = (\mathcal{L}(a))^\star = \{a\}^\star = \{\epsilon, a, aa, aaa, \dots\}$$
$$\mathcal{L}(a(a \mid b)^\star) = \cdots = \{a, aa, ab, aaa, aab, aba, abb, \dots\}$$

$\Diamond$

En del författare använder $\cup$ i stället för | i reguljära uttryck. Det är lätt att förstå varför. I vårt sammanhang är det en fördel att ha olika operatorer i reguljära uttryck och mängduttryck.

I matematiken har man sällan anledning att skilja på ett uttryck och uttryckets värde. När man skriver ett uttryck menar man nästan alltid dess värde. I programsammanhang är det annorlunda. Om man i ett program skriver uttrycket `x*x+1` så är det dess värde som skall beräknas med användning av det värde `x` har vid tillfället. Om man matar in $x * x + 1$ i Matlab eller Maple så måste uttrycket själv sparas för att kunna beräknas eller t ex deriveras vid ett senare tillfälle.

Exempel. Det reguljära uttrycket $(0 \mid 1)^\star$ betecknar språket av alla binära strängar.
$\mathcal{L}((0 \mid 1)^\star) = (\mathcal{L}((0 \mid 1)))^\star = (\mathcal{L}(0) \cup \mathcal{L}(1))^\star = (\{0\} \cup \{1\})^\star = \{0, 1\}^\star = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$.
$\Diamond$

Exempel. $(0 \mid (1(0 \mid 1)^\star))$ beskriver språket av alla binära strängar utan extra inledande nollor, $\{0, 1, 10, 11, 100, \dots\}$ $\Diamond$

6.2 Utvidgad notation

Reguljära uttryck används ofta för att beskriva hur man får skriva numeriska konstanter och variabelnamn i programspråk. I sådana sammanhang inför man ofta ytterligare några konstruktioner. I stället för att skriva $0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$ tillåter vi oss $[0 - 9]$.

Om man vill upprepa något en eller flera gånger skriver man ett plustecken i stället för stjärna, $[0 - 9]^+ = [0 - 9][0 - 9]^\star$.

Ett frågetecken efter ett reguljärt uttryck betyder upprepning ingen eller en gång, $\alpha? \triangleq \alpha \mid \epsilon$. Operatorn har samma precedens som $\star$ och $+$.

Det kan också vara bekvämt att sätta namn på reguljära uttryck och använda dessa namn i andra reguljära uttryck. Den utvidgade notationen innebär inte att man kan beskriva flera språk utan bara att en del beskrivningar blir kortare.

Exempel.

$$\text{DIGIT} \triangleq [0-9]$$
$$\text{NAT} \triangleq \text{DIGIT}^{+}$$
$$\text{INT} \triangleq (-)?\text{NAT}$$
$$\text{FLOAT} \triangleq \text{INT} \,.\, \text{NAT}$$

$\Diamond$

Det är emellertid inte tillåtet att använda sådana namn rekursivt. En sådan utvidgning behandlas i kapitlet om *grammatik*.

6.3 Reguljära uttryck med Java

I `java.util.regex` finns två klasser för effektiv matchning med reguljära uttryck. Om samma reguljära uttryck skall användas flera gånger ”kompileras” det först:

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

Kompileringen skapar en tillståndsmaskin som representeras av en tabell med tillståndsövergångar. ”Matcharen” innehåller ett tillstånd som beskriver hur långt matchningen kommit i strängen, som alltså kan utföras en bit i taget. Metoden `matches()` försöker matcha det reguljära uttrycket mot hela strängen.

Om det reguljära uttrycket bara skall användas en gång skriver man:

```
boolean b = Pattern.matches("a*b", "aaaaab");
```

7 Grammatik

De språk som kan beskrivas med reguljära uttryck är tämligen primitiva. Man har bara tre operationer: en för sammasättning, en för alternativ och en för upprepning.

Tidigare i kursen har vi haft anledning att diskutera den *externa* eller *konkreta* representationen av aritmetiska uttryck. Reguljära uttryck är för primitiva för att kunna beskriva mängden av aritmetiska uttryck.

> Ett *uttryck* består av en eller flera *termer* separerade av enkla plus- eller minustecken. En *term* består i sin tur av en eller flera *faktorer* separerade av enkla multiplikations- eller divisions- tecken. En *faktor* är ett *tal*, en *variabel* eller ett *uttryck* inom parenteser. Ett *tal* består av en eller flera siffror och får inledas med ett minustecken. En *variabel* består av en eller flera bokstäver bland a–z.

Vi använde en särskild formell notation, *Backus–Naur–form* (BNF), för detta:

```
expr ::= term (addop term)*
term ::= factor (mulop factor)*
factor ::= NUMBER | NAME | '(' expr ')'
addop ::= '+' | '-'
mulop ::= '*' | '/'
```

Detta kallas för en *grammatik* och den består av *produktioner*. Varje produktion har ett vänsterled som vi kallar *syntaxsymbol* (eng. nonterminal symbol). Syntaxsymbolen är ett namn. Eftersom grammatiken ibland skall översättas till ett program som kan analysera strängar skrivna enligt grammatiken använder vi gärna namn som duger som metodnamn i Java.

Högerledet i en produktion påminner om reguljära uttryck; vi använder samma operatorer, $|$, $*$ och sammansättningsoperatorn (som inte syns). Vi använder parenteser för gruppering. Det som skiljer är att man får använda syntaxsymboler i uttrycket och att symbolerna i alfabetet omges av citationstecken för att skilja dem från syntaxsymboler. Symbolerna i alfabetet kallas för *slutsymboler* (eng. terminal symbols). `NUMBER` och `NAME` är också slutsymboler och betecknar tal resp. variabelnamn. Normalt använder reguljära uttryck för att beskriva hur tal och variabelnman skall skrivas.

I grammatiken ovan är $N \triangleq \{\texttt{expr}, \texttt{term}, \texttt{factor}, \texttt{addop}, \texttt{mulop}\}$ mängden av syntaxsymboler och $\Sigma \triangleq \{+, -, *, /, \texttt{NUMBER}, \texttt{NAME}\}$ slutsymboler.

Ibland gör man tvärt om och dekorerar syntaxsymbolerna i stället för slutsymbolerna.

```
<expr> ::= <term> (<addop> <term>)*
<term> ::= <factor> (<mulop> <factor>)*
<factor> ::= NUMBER | NAME | ( <expr> )
<addop> ::= + | -
<mulop> ::= * | /
```

Varje rad namnger en *syntaxsymbol* och räknar upp beståndsdelarna. Syntaxsymbolen står till vänster om "::=" och beståndsdelarna till höger. Det som står inom parentes och följs av en asterisk får upprepas noll eller flera gånger. När det finns alternativa beståndsdelar separeras dessa av `|`-tecken som utläses *eller*. Det som omges av apostrofer står för sig själv. Vi ser alltså att en `addop` är ett plus- eller minus-tecken och att ett `expr` består av en eller flera `term`er separerade av plus- eller minus-tecken. Symbolen för alternativ binder svagast så att `'(' expr ')'` är ett av tre alternativ för `factor`.

Vi har nu en *grammatik* för aritmetisk uttryck och kan göra en *härledning* för att visa hur en sträng kan konstrueras. En härledning börjar med grammatikens *startsymbol*, som enligt konvention brukar vara den symbol som finns i högerledet i den första raden. I varje härledningssteg

ersätter man en syntaxsymbol med motsvarande högerled. Härledningen är klar när det inte finns några syntaxsymboler kvar.

Följande är en härledning av strängen `1+x` eller egentligen av `NUMBER + NAME`. När det inte kan bli några missförstånd i en härledning utelämnar vi citationstecken kring slutsymbolerna .

```
expr ⇒ term addop term ⇒ factor addop term ⇒ NUMBER addop term ⇒
NUMBER addop term ⇒ NUMBER + term ⇒ NUMBER + factor ⇒ NUMBER + NAME
```

Vi utläser ⇒som 'härleder i ett steg'. Ett steg i en härledning innebär att man ersätter en syntaxsymbol med motsvarande högerled från någon produktion.

Om vi vill sammanfatta en sådan härledning i noll eller flera steg skriver vi

`expr` $\Rightarrow^*$ `NUMBER + NAME`

Definition. Låt G vara en grammatik med syntaxsymboler i N, slutsymboler i Σ, produktioner som definierar ⇒och startsymbol S. Mängden av alla strängar i Σ^* som kan härledas från S kallas det språk som *genereras* av grammatiken och betecknas L(G).

$$\mathcal{L}(G) \triangleq \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

△

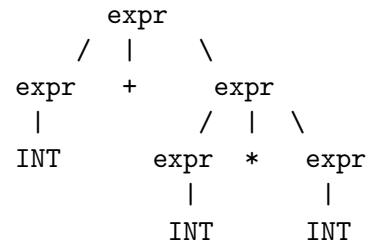
7.1 Härledningsträd

Följande grammatik beskriver enkla aritmetiska uttryck med addition och multiplikation.

expr ::= *expr* `"+"` *expr*
expr ::= *expr* `"*"` *expr*
expr ::= *INT*

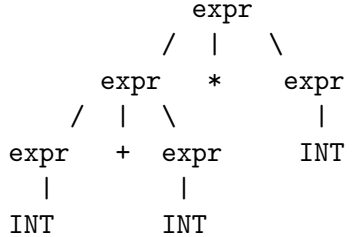
Härledning av `INT + INT * INT` tillsammans med ett träd som visar hur produktionerna har använts. Man kan notera likheten med härledningar i satslogiken.

```
expr  =>
expr + expr =>
expr + expr * expr =>
INT + expr * expr =>
INT + INT * expr =>
INT + INT * INT
```



Med samma grammatik går det att härleda samma sträng på ett annat vis. Härledningsträdet blir fundamentalt annorlunda.

```
expr  =>
expr * expr =>
expr + expr * expr =>
INT + expr * expr =>
INT + INT * expr =>
INT + INT * INT
```



Om vi bygger en abstrakt representation som svarar mot det senare härledningsträdet så blir den inte den avsedda.

En grammatik är *tvetydig* om det finns mer än ett härledningsträd för någon sträng i språket.

Om en grammatik är tvetydig måste man försöka hitta en grammatik som inte är det och som genererar samma språk. I det aktuella fallet vill man ha en grammatik som respekterar gängse precedens för operatorerna.

Hur man konstruerar grammatiker som gör det lätt att bygga rätt abstrakt representation studeras i kursen i Kompilatorteknik.

7.2 Syntaxanlys

Det är enkelt att skriva en metod som avgör om en sträng innehåller ett variabelnamn eller en följd av siffror och returnerar motsvarande `Variable`- eller `Int`-objekt. I kompilatortekniken kallas detta för en *lexikalanalysator*. I `java.io` finns en klass, `StreamTokenizer`, som gör jobbet. I följande kodavsnitt skapas ett sådant objekt som kommer att läsa från en sträng via en `StringReader`.

```
public class ExprParser extends StreamTokenizer {
    private int token;
    public ExprParser(String string) throws IOException  {
        super(new StringReader(string));
        ordinaryChar('-');
        ordinaryChar('/');
        token = nextToken();
    }
```

`StreamTokenizer` initieras så att minus- och divisionstecken behandlas som vanliga tecken. De är annars initierade för att passa analys av programspråk där man har negativa tal och kommentarer som inleds med `'/'`. `nextToken` returnerar ett heltal som anger vad för slags tecken den hittat.

I klassen finns metoder med samma namn som syntaxsymbolerna i grammatiken för uttryck. Metoden `expr` skall analysera ett helt uttryck enligt grammatiken

```
expr ::= term (addop term)*
```

Metoden `term` skall göra sammalunda med

```
term ::= factor (mulop factor)*
```

och `factor` skall klara av

```
factor ::= number | name | '(' expr ')'
```

Vi börjar med `factor`. Om nästa tecken (`token`) är en vänsterparentes så bygger vi ett helt uttryck genom att anropa `expr` rekursivt. Om `token` anger att den hittat ett tal `StreamTT_NUMBER` så finns detta att hämta i attributet `nval`.

```
    private Expr factor() throws IOException {
        Expr e;
        switch (token) {
            case '(' :
                token = nextToken();
                e = expr();
                token = nextToken();
                return e;
```

```
            case TT_NUMBER :
                double x = nval;
                token = nextToken();
                return new Int((int) x);
            case TT_WORD :
                String s = sval;
                token = nextToken();
                return new Variable(s);
        }
    }
```

Om `token` i stället markerar att den hitta ett `StreamTT_WORD` finns strängen att hämta i `sval`.

Metoden `term` skall analysera en eller flera faktorer med hjälp av `factor`.

```
    private Expr term() throws IOException {
        Expr result, factor;
        result = factor();
        while (token == '*' || token == '/') {
            int op = token;
            token = nextToken();
            factor = factor();
            switch (op) {
                case '*' :
                    result = new Mul(result, factor);
                    break;
                case '/' :
                    result = new Div(result, factor);
                    break;
            }
        }
        return result;
    }
```

Slutligen skall `expr` ta hand om termerna. Detta är helt analogt med metoden `factor`. Detta lämnas som övning.

Slutligen definieras `build` som en synonym till `expr`.

```
    public Expr build() throws IOException {
        return expr();
    }
```