INDU: Satslogik

Anders Mörtberg

Innehåll

Introduktion

I det här projektet ska du skriva ett Python-program för att hantera satslogiska formler och deras sanningstabeller. Satslogik är ett logiskt system där man kan resonera om påståenden och deras sanningsvärde. Tre exempel på påståenden är:

p **:** Python är bäst

q **:** Programmering är roligt

r **:** Kommentarer ska skrivas på svenska

Här är p, q, och r namn/variabler vilka representerar påståendena. Genom att använda logiska operatorer kan man sedan skapa mer komplicerade sammansatta påståenden. De logiska operatorer som ska stödjas är:

$\wedge$ **:** *och*

$\vee$ **:** *eller*

$\neg$ **:** *icke*

Med dessa tre operatorer kan vi nu skapa formler som $p \wedge q$ eller $\neg p \vee (q \wedge r)$ vilka kan läsas som följande meningar:

$p \wedge q$**:** Python är bäst *och* programmering är roligt

$\neg p \vee (q \wedge r)$**:** Python är *inte* bäst *eller* programmering är roligt *och* kommentarer ska skrivas på svenska

För att avgöra när en satslogisk formel är sann eller inte används sanningsvärdestabeller. Dessa ser ut på följande sätt för operatorerna ovan:

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

p	$\neg p$
T	F
F	T

Här står T för "True" och F för "False". Med hjälp av dessa kan vi sedan beräkna sanningsvärdestabellen för mer komplexa formler med flera variabler genom att titta på alla kombinationer av T och F för variablerna:

p	q	r	$\neg p \vee (q \wedge r)$
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	T
F	T	F	T
F	F	T	T
F	F	F	T

Notera att satslogiska formler alltså beter sig som booleska uttryck. Om formeln har n variabler har sanningsvärdestabellen 2^n olika rader (en för varje möjlig kombination av tilldelningar av T och F till variablerna).

Med hjälp av satslogiska formler kan man modellera en massa problem och er uppgift är att skriva ett program som underlättar för användaren att analysera satslogiska formler och deras sanningsvärdestabeller.

Uppgift 1 (E-C)

Implementera ett program som stödjer följande valmöjligheter för användaren:

1 Skriv in en satslogisk formel och skriv ut dess sanningsvärdestabell.

q Avsluta.

För att underlätta detta har vi utökat koden för algebraiska uttryck från kapitel 11 i kompendiet med en parser (se Appendix i slutet av det här dokumentet). En "parser" är en funktion som tar in en sträng och översätter till den datarepresentation som vi vill arbeta med (i appendix algebraiska uttryck och i koden ni ska skriva satslogiska formler). Den parser vi har skrivit kan parsa algebraiska uttryck skrivna på *reverse polish notation* (RPN).[1] Detta är ett sätt att skriva olika typer av formler på där operatorn skrivs efter argumenten vilket gör parsning lättare. Så exempelvis skrivs uttrycket 1 + 2 istället 1 2 +, så först kommer argumenten (1 och 2) och sen kommer operatorn (+). Detta har fördelen att uttryck kan skrivas på ett otvetydigt sätt som gör det lättare för en dator att parsa uttryck, exempelvis skrivs formeln $(1+x)*y$ i RPN som $1\ x\ +\ y\ *$ medans $1+(x*y)$ skrivs som $1\ x\ y\ *\ +$ vilket gör det lättare att parsa då det inte finns några paranteser. Vidare, om man bara hade skrivit $1+x*y$ så hade det ju kunnat tolkas som antingen $1+(x*y)$ eller $(1+x)*y$ beroende på vilka konventioner vi har kring hur operatorerna binder. Så genom att istället skriva uttryck på RPN blir det mycket lättare att parsa och man behöver inte hantera några konventioner kring operatorer eller paranteser.

Innan ni börjar på den här uppgiften bör ni alltså kopia vår kod, förstå vad den gör, och skriva om så att ni istället kan parsa satslogiska formler. Ni bör även anpassa koden så att man kan konvertera satslogiska formler till strängar (dessa behöver *inte* vara i RPN!). För att göra allt detta enklare bör ni använda symbolerna `&`, `|`, och `~` för de satslogiska operatorerna $\wedge$, $\vee$, och $\neg$. När ni gjort detta ska man alltså kunna skriva saker som `p ~ q r & |` för `~ p | (q & r)`, vilket motsvarar formeln $\neg p \vee (q \wedge r)$.

Obs: man måste inte börja från vår kod eller använda sig av RPN om man inte vill, men uppgiften kan bli rejält mycket svårare om man inte gör det.

[1] Se https://en.wikipedia.org/wiki/Reverse_Polish_notation#Explanation

Exempelkörning

Programmet förväntas ha följande ungefärliga interaktion:

```
Print truth table [1], exit [q].
Select an option: 1
Write a formula (in RPN): p q &

The truth table is:

p   q   p & q
-------------
T | T | T
T | F | F
F | T | F
F | F | F

Print truth table [1], exit [q].
Select an option: 1
Write a formula (in RPN): p ~ q r & |

The truth table is:

p   q   r   ~ p | (q & r)
-------------------------
T | T | T | T
T | T | F | F
T | F | T | F
T | F | F | F
F | T | T | T
F | T | F | T
F | F | T | T
F | F | F | T

Print truth table [1], exit [q].
Select an option: q

Thank you and goodbye!
```

Krav uppgift 1

- Korrekt implementation av de logiska uttrycken, parsern och beräkningen av sanningsvärdestabeller. För högre betyg måste man även uppfylla de betygskriterier som beskrivs i de generella instruktionerna i *INDU—Individuell Uppgift i Programmering*.
- **Obs:** lösningen måste fungera för komplexa formler med godtyckligt många variabler (d.v.s. formler med n variabler vilket då ger sanningsvärdestabeller med 2^n rader).
- Uppgifterna ska lösas utan att importera icke-standard bibliotek (d.v.s. såna som måste installeras). Fråga kursledaren om ni är osäkra om ni får använda ett specifikt bibliotek. Redogör för alla använda bibliotek i projektrapporten.
- Tänk på att ha lämplig felhantering i användarinteraktionen (t.ex. om användaren skriver in en formel som inte går att parsa så ska programmet inte krascha).

Tips

1. Utgå från vår kod och skriv om den. Lägga till en operator i taget och testa parsningen och strängkonverteringen med olika formler för varje operator ni lägger till.

2. Tänk på designen innan ni börjar programmera. Bör ni ha några specifika klasser? Vilka funktioner bör ni ha? Kom ihåg att separarera interaktion från beräkning då det verkligen underlättar debugging och leder till bättre kod vilken är lättare att testa.

Uppgift 2 (E-A)

Ni ska nu utöka programmet i uppgift 1. Programmet ska nu låta användaren lägga till sanna påståenden (eng. assertions). Dessa motsvarar att vi får mer information om vad som är sant och därför behöver vi inte skriva ut hela sanningsvärdestabellen. Exempelvis om vi vet att p är sann så behöver vi bara skriva ut de rader i sanningsvärdestabellen för $\neg p \vee (q \wedge r)$ där p är sann (de fyra första raderna i exemplet ovan). Om vi istället vet att $\neg p$ är sann så ska bara de fall där p är falsk skrivas ut (d.v.s. de fyra sista i exemplet ovan).

Om användaren lägger till flera påståenden ska bara de rader som uppfyller samtliga påståenden visas. Programmet måste nu alltså spara de påståenden som användaren lagt in. Om man lägger in $\neg r$ och $p \vee q$ som påståenden ska bara de rader i sanningsvärdestabellen för $\neg p \vee (q \wedge r)$ som uppfyller att r är falsk och, p eller q är sanna visas (d.v.s. rad 2, 4 och 6 i exemplet ovan).

Det kan hända att användaren försöker lägga till motsägelsefulla påståenden som gör att påståendena aldrig kan vara sanna på något sätt samtidigt, exempelvis p och $\neg p$. Programmet ska då rapportera detta och inte acceptera att man lägger till påståendet som leder till motsägelsen.

Alternativen i programmet ska nu vara:

1 Skriv in en satslogisk formel och skriv ut dess sanningsvärdestabell.

2 Lägg till ett påstående.

3 Ta bort ett påstående.

q Avsluta.

Funktionaliteten i alternativ 1 ska vara så att de påståenden som lagts in tas i beaktande. Om användaren inte lagt in några påståenden (eller tagit bort alla som den lagt in) ska programmet fungera som i uppgift 1.

Exempelkörning

Vi visar här ett exempel på användning av alternativ **2** och **3**.

```
Print truth table [1], add assertion [2], remove assertion [3], exit [q].
Select an option: 2
Write a formula (in RPN): p q |

Current assertion(s) are:
1. p | q

Print truth table [1], add assertion [2], remove assertion [3], exit [q].
Select an option: 1
Write a formula (in RPN): p q &

Current assertion(s) are:
1. p | q

The truth table is:

p   q   p & q
-------------
T | T | T
T | F | F
F | T | F
```

```
Print truth table [1], add assertion [2], remove assertion [3], exit [q].
Select an option: 2
Write a formula (in RPN): p

Current assertion(s) are:
1. p | q
2. p

Print truth table [1], add assertion [2], remove assertion [3], exit [q].
Select an option: 1
Write a formula (in RPN): p q &

Current assertion(s) are:
1. p | q
2. p

The truth table is:

p   q   p & q
-------------
T | T | T
T | F | F

Print truth table [1], add assertion [2], remove assertion [3], exit [q].
Select an option: 2
Write a formula (in RPN): p ~

Inconsistent assertion! Please try again


Print truth table [1], add assertion [2], remove assertion [3], exit [q].
Select an option: 3

Current assertion(s) are:
1. p | q
2. p

Which assertion do you want to remove? 2

Current assertion(s) are:
1. p | q


Print truth table [1], add assertion [2], remove assertion [3], exit [q].
Select an option: 2
Write a formula (in RPN): p ~

Current assertion(s) are:
1. p | q
2. ~ p

Print truth table [1], add assertion [2], remove assertion [3], exit [q].
Select an option: 1
Write a formula (in RPN): p q &

Current assertion(s) are:
```

```
1. p | q
2. ~ p

The truth table is:

p   q   p & q
-------------
F | T | F

Print truth table [1], add assertion [2], remove assertion [3], exit [q].
Select an option: q

Thank you and goodbye!
```

Krav uppgift 2

- Man måste ha löst uppgift 1 och alla krav som gäller för uppgift 1 gäller även uppgift 2.
- I övrigt gäller även att om du gör båda uppgifterna ska du lämna in endast ett program. D.v.s. du ska utöka funktionaliteten i ditt program från uppgift 1 med uppgift 2. Dock är det bra att spara en kopia av uppgift 1 när du är klar med den innan du försöker implementera uppgift 2.

Betygssättning

Detaljer finns i de generella instruktionerna i *INDU—Individuell Uppgift i Programmering*.

Appendix

Kod för algebraiska uttryck från föreläsning 11 (utan konstanter) tillsammans med en parser:

```
def parens(p1,p2,s):
    if p1 < p2:
        return "(" + s + ")"
    else:
        return s

class Expr:
    prec = 1000

class Var(Expr):

    def __init__(self, name):
        self._name = name

    def __str__(self):
        return self._name

class BinOp(Expr):

    def __init__(self, left, right):
        self._left = left
        self._right = right

class Plus(BinOp):

    prec = 1

    def __str__(self):
        s1 = parens(self._left.prec,self.prec,str(self._left))
        s2 = parens(self._right.prec,self.prec,str(self._right))
        return s1 + " + " + s2

class Times(BinOp):

    prec = 2

    def __str__(self):
        s1 = parens(self._left.prec,self.prec,str(self._left))
        s2 = parens(self._right.prec,self.prec,str(self._right))
        return s1 + " * " + s2


def parse(expr_string):
    """Parse a string in RPN into an expression"""

    # Mapping from strings to operators. Each entry is a
    # tuple of the operator constructor and its arity (number
    # of arguments it takes)
    operators = {"+" : (Plus, 2), "*" : (Times, 2)}

    stack = []
```

```
for token in expr_string.split():
    if token in operators:
        op, arity = operators[token]
        if len(stack) < arity:
            raise ValueError("Not enough arguments to " + token)
        elif arity == 0:
            stack.append(op())
        else:
            args = stack[-arity:]
            stack = stack[:-arity]
            # apply op to the arguments in the list args, "unpacked"
         # https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-lists
            stack.append(op(*args))
    else:
        # All other strings are treated as variables
        stack.append(Var(token))

if len(stack) == 1:
    return stack[0]
else:
    raise ValueError("Not a valid expression: " + expr_string)
```