Od teorie formálních jazyků k jednoduchému překladači

HASHIM HABIBALLA - EVA VOLNÁ - ROSTISLAV FOJTÍK

Přírodovědecká fakulta OU, Ostrava

Ve starších ročnících MFI jsme se mohli setkat s články odhalující zajímavý svět teoretické informatiky, který může mnoha učitelům i studentům informatiky, ale i profesionálům - vývojářům a programátorům, připadat jako nutné zlo nebo zbytečná matematická teorie bez využití. V těchto článcích jsme se snažili na jednoduchých příkladech ukázat, že jde o zajímavou a potřebnou základnu informatiky, u které je spíše problém ve stylech a metodách výuky, které ji mnohdy předurčují k podobnému vidění v očích studujících. Ukázali jsme jednak logiku jako informatickou vědu o automatizaci usuzování [10], dále teorii vyčíslitelnosti a složitosti, která dává exaktní a univerzální nástroje pro popis algoritmů a zjišťování jejich efektivity [9] a také teorii formálních jazyků [8]. Právě k teorii formálních jazyků bychom se chtěli nyní vrátit a ukázat příklad, který více než co jiného blízký praxi informatika. Ukážeme jej samozřejmě na zjednodušeném modelu, který je ale principiálně univerzálně použitelný. Chceme ukázat, jak jednoduchou úlohu má informatik s dobrou znalostí teoretické informatiky oproti informatikovi, který tyto znalosti nepoužívá. Vše samozřejmě ukážeme na programátorské úrovni, včetně hotového kódu.

Úloha kterou přinášíme má hodně společného s typickou úlohou v informatické praxi a tou je překlad resp. interpretace nějakého zdrojového kódu (programu) do jiného jazyka (včetně optimalizací vzniklého kódu). S tímto se setkáváme prakticky každodenně (otázka je zda si to uvědomujeme). Samozřejmě nejprve informatika asi napadne úloha typu - programuji řešení zadaného úkolu v nějakém programovacím jazyce (např. Pascalu) a pak použiji překladač, který mi vyrobí kód v nějakém počítačově použitelném formátu (třeba exe soubor pro počítač typu PC pod platformou MS-DOS). Nemusí však jít jen o tento "programátorský" případ. Aniž si to mnohdy uvědomujeme, pokud prohlížíme libovolnou webovou stránku, tak náš webový prohlížeč získává z daného URL kód v jazyce HTML. Ten by samozřejmě pro člověka asi nebyl na čtení to nejlepší. Proto prohlížeč musí PŘELOŽIT a interpretovat tento kód, tak aby byl pro člověka příjemný. Jde o zobrazení různých velikostí písma, tabulek, obrázků, pozadí stránek atd. Jakkoliv to vypadá naprosto přirozeně, pokud budeme zkoumat, jak fungují programy, které tuto činnost dělají, není to až tak triviální činnost a souvisí neoddělitelně právě s teorií formálních jazyků. I zdánlivě jednoduchá součást tohoto procesu je bez dobré znalosti pojmů a postupů, které přináší teorie formálních jazyků a automatů, takřka neřešitelný úkol.

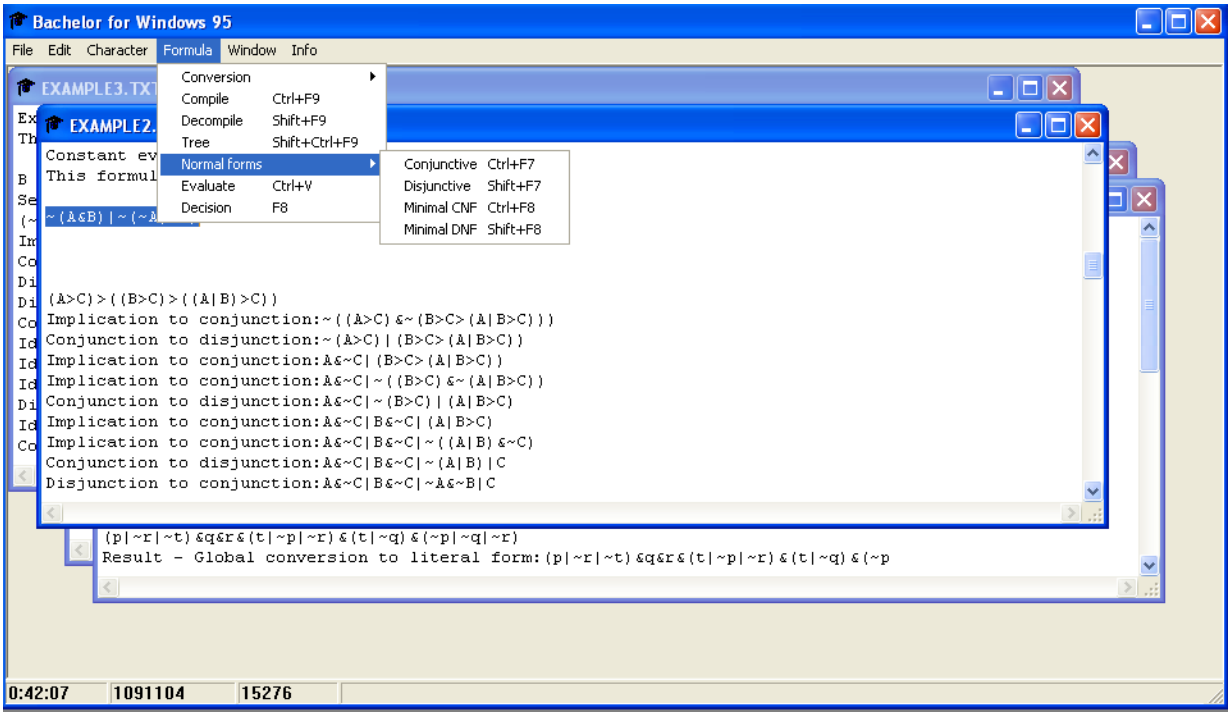
Touto jednoduchou úlohou, se kterou se nyní důkladně seznámíme a ukážeme si její

(obecně použitelné) řešení, je kontrola, překlad a optimalizované vyhodnocení logického výrazu. Úlohu si samozřejmě pro naše účely výkladu algoritmů zjednodušíme oproti klasickým programovacím jazykům, kde můžeme používat i relační operátory, číselné proměnné (to ale nijak nesnižuje obecnost principů, které se zde naučíme). Logickým výrazem (nebo chcete-li výrokovými formulemi) budeme myslet řetězce s jednopísmennými identifikátory, logickou konstantou 0 (pravda), 1 (nepravda), logickými operátory (symboly užívané v následujících příkladech budou první v pořadí uvedené - používáme symboly ASCII kódu do 127) $\sim$ (negace - $\neg$), & (konjunkce - $\wedge$), | (disjunkce - $\vee$), > (implikace - $\rightarrow$), = (ekvivalence - $\leftrightarrow$) a pomocným symbolem závorek. Např. $\neg(A \wedge B) \vee \neg(\neg A \vee \neg B)$ - je výraz podle standardních interpretací logických operátorů zjevně vždy pravdivý a tedy jeho opakovaného vyhodnocování je zbytečné mrhaní strojovým časem.

Takovéto výrazy jsou nedílnou každého vyššího procedurálního programovacího jazyka (řešení všech problémů obvykle stavíme na správně formulovaných podmínkách, které leckdy bývají docela složité a mnohdy ani sám programátor neodhalí, že daná podmínka buď nemůže být nikdy splněna, nebo je splněna vždy, či případně může být podstatně zjednodušena a tím i zvýšena efektivita výpočtu programu). Jelikož takovéto složitější podmínky bývají mnohdy vykonávány v cyklech a to třeba mnoha milionkrát, je jistě vhodné, aby moderní překladače programovacích jazyků provedli při vyhodnocení podmínky, co nejméně operací - tedy překladač by měl jistě odhalit například skutečnost v našem příkladě - tedy, že $\neg(A \wedge B) \vee \neg(\neg A \vee \neg B)$ je po formálních úpravách zjednodušitelné na 0 (logickou pravdu).

Uživatelé překladačů jazyka Pascal nebo C jistě dobře ví, že tyto překladače více či méně těchto optimalizací provádějí. Ale jak tuto činnost realizují? Samozřejmě, že uvažujeme-li jako lidé s inteligencí a schopností číst text nejen lineárně (po znacích), pak v našem mozku probíhají pochody přímo na počítači realizovatelné velmi těžko. Na našem výrazu si ihned uvědomíme, v jakém pořadí probíhá vyhodnocení jednotlivých operací (tedy nejvyšší prioritu mají logické proměnné a závorky, pak postupujeme ve vyhodnocování podvýrazů podle struktury formule a priorit operátorů (tedy nejdříve negace, pak konjunkce, implikace a nakonec ekvivalence). Bohužel dnešní počítače a klasické procedurální programovací jazyky takovou inteligencí nedisponují a uvědomme si například, že výraz musíme zadat jako nějaký řetězec znaků, který pak můžeme číst po znacích. Je také dobré si uvědomit, že aby program (překladač), který bychom si rádi udělali sami, měl pro praktickou práci smysl, musí být efektivní - mít dobrou časovou složitost. Určitě bychom nechtěli mít překladač, který řetězec (což v obecném případě může být program o desítkách nebo stovkách tisíc znaků) prochází opakovaně a stále se vrací k tomu, co už jednou přečetl (takový algoritmus byl představen již dříve v MFI, i přes jeho nespornou jednoduchost je však pro profesionální programy nepřijatelný). Sami si představte, že program v Pascalu vám překladač kompiluje hodiny, dny nebo měsíce, aby nakonec oznámil, že v něm máte na konci syntaktickou chybu. Takový přístup by vás asi napadl, budete-li snažit vyhodnocovat výraz nějakých triviálním algoritmem. Tedy postupně vyhodnocovat podvýrazy a stále znovu procházet řetězec. To je však z hlediska optimality zcela nepřijatelné - my se přece snažíme ušetřit každou operaci a tedy opakované procházení je zcela proti tomuto cíli. Budeme tedy muset zřejmě zapojit nějaké "know-how" a v našem případě to budou právě prostředky a algoritmy, které nám dává teorie formálních jazyků a překladačů.

Tento text si samozřejmě nečiní ambice vás seznámit s celou teorií tvorby překladačů (ta obsahuje mnoho sofistikovaných postupů, jak efektivně překládat sekvence příkazů, složité datové struktury atd.), ale spíše se zaměříme na analytickou část překladače a optimalizace ve výrazech (logických). K celému textu lze také velmi dobře využít již hotové aplikace hlavního autora [4]. Tento balíček přináší spustitelné aplikace (s názvem Bachelor) pro platformu MS-DOS, MS-Windows (32-bit), včetně zdrojových kódů v jazyce Pascal (Turbo Pascal 7.0 + Turbo Vision 2.0, resp. Object Pascal pod Borland Delphi), i ve verzi v C++ (Borland C++ 3.1 s Turbo Vision). Balíček umožňuje zapisovat logické výrazy se stejnou syntaxí jakou jsme již popsali a provádět jejich převod (kompilaci) do formy syntaktického stromu [14], převod do funkčně úplných množin spojek (např. jen na negaci a implikaci), převody na konjunktivní a disjunktivní normální formy, jejich minimalizaci a nakonec i rozhodnutí, zda formule není kontradikcí nebo tautologií [10]. To vše je možné také sledovat po jednotlivých krocích i s komentářem, jaké logické ekvivalence (zákony) jsme k tomu použili. Vše ilustruje obrázek 1, s otevřeným souborem obsahujícím výstupy formálních úprav provedených aplikací Bachelor. Pozn. Do aplikace WinBch95 byl dodatečně dopracován pro čtenáře i výpis označené formule do formy syntaktického stromu (textově) - DOS a Win16-bit verze již nebudou dále rozšiřovány ani opravovány.



Obrázek 1: Aplikace Bachelor pro formální úpravy ve výrokové logice

Celý balíček lze získat na adrese: `http://www1.osu.cz/home/habibal/publ/bachelor.zip`

1 Definice jazyka logických výrazů pomocí gramatiky - Backusovy-Naurovy formy

Aby byl schopen počítač pracovat s logickým výrazem (resp. s jakýmkoliv jiným syntaktickým elementem v libovolném zdrojovém kódu) musíme nějakým vhodným formálním prostředkem zapsat definici tohoto výrazu. Tyto výrazy vlastně tvoří specifický jazyk a tedy teoretická informatika nám dává prostředek ve formálních gramatikách, o kterých jsme si již v MFI mohli přečíst článek (citace TFJA). Pro definice struktur programovacích jazyků se velmi dobře hodí takzvaná Backusova-Naurova forma. Je podobná bezkontextovým gramatikám (má terminální symboly, neterminální symboly, pravidla pro přepis neterminálů), ale navíc zjednodušuje zápis obvyklých technik jako je opakování nějakého řetězce (třeba příčítaní podvýrazů v aritmetických výrazech se může dělat opakovaně - u bezkontextových gramatik bychom toto museli řešit krkolomně pomocí jakési "rekurze"). Bezkontextová gramatika (BKG) je jedním z velmi vhodných způsobu zápisu syntaxe jazyků. Syntaxí zde rozumíme jejich jazykovou strukturu. Umožňuje totiž vyjádřit většinu technik, které například u programovacích jazyků používáme. Jde o alternativu několika možností, opakování stejného jazykového výrazu a hlavně vnořování celých rozvětvených struktur mezi sebou. Poslední zmiňovaná technika je právě onou technikou, kterou neumíme vyjádřit pomocí jazyků regulárních, ale teprve pomocí bezkontextových jazyků. Zkusme si představit velmi omezenou část nějakého programovacího jazyka - například strukturu aritmetického výrazu. Principiálně je většina jiných struktur velmi podobných (např. sekvence příkazů je analogická opakovanému sčítání podvýrazů!).

Příklad:

Sestrojme BKG pro jazyk složený z aritmetických výrazů s operandem x, operacemi +,* a umožňující vnořovat další podvýrazy stejného typu pomocí symbolů závorek (,). Kupříkladu se může jednat o výraz: $x * (x + x + x)$

Gramatiku sestrojíme hierarchicky - tedy aby byla rozlišena priorita operátorů a využijeme "rekurzivní" vlastnosti přepisu neterminálu, abychom docílili možnosti generovat opakovaně sčítání a násobení.

$G = (S, A, B, x, *, +, (,), S, P)$

$P : S \rightarrow^1 A + S, S \rightarrow^2 A$ (opakovaný přepis na S nám umožní generovat libovolně mnoho sčítání struktury A)

$A \rightarrow^3 B * A, A \rightarrow^4 B$ (opakovaný přepis na A nám umožní generovat libovolně mnoho násobení struktury B)

$B \rightarrow^5 (S), B \rightarrow^6 x$ (rekurzivním přepisem na S můžeme vnořit libovolně mnoho podvýrazů zcela stejné struktury jako výraz sám do závorek anebo ukončit generování operandem x) (Pozn.: index u symbolu určuje pomocné číslo pravidla)

V této gramatice pak lze snadno generovat například výše uvedený výraz $x*(x+x+x)$:

$S \Rightarrow^2 A \Rightarrow^3 B * A \Rightarrow^6 x * A \Rightarrow^4 x * B \Rightarrow^5 x * (S) \Rightarrow^1 x * (A + S) \Rightarrow^4 x * (B + S) \Rightarrow^6 x * (x + S) \Rightarrow^1 x * (x + A + S) \Rightarrow^4 x * (x + B + S) \Rightarrow^6 x * (x + x + S) \Rightarrow^2 x * (x + x + A) \Rightarrow^4 x * (x + x + B) \Rightarrow^6 x * (x + x + x)$

(Pozn.: index u symbolu $\Rightarrow^i$ určuje pomocné číslo pravidla)

Dalším přehledným a hlavně v praxi ještě více využívaným způsobem zápisu syntaxe jazyka je takzvaná Backusova-Naurova forma (BNF). Jde o zápis podobný bezkontextové gramatice, ale přitom bližší spíše programátorům, resp. praxi.

BNF obsahuje podobně jako BKG neterminály, které se uvádějí do úhlových závorek a přepisují skrze symbol := na řetězce terminálních a neterminálních symbolů. Jde tedy o pravidla tvaru:

$\langle X \rangle ::= \alpha_1 \dots \alpha_n$

Pro přehlednější zápis je však ještě lepší modifikace BNF zvaná EBNF (Extended BNF) - rozšířená BNF, která zjednodušuje zápis opakovaně používaných, příp. podmíněně vyskytujících se výrazů. Umožňuje následující zápisy:

$\{\alpha\}$ - znamená, že výraz se vyskytuje v libovolném počtu (ekvivalent operace iterace)
$\{\alpha\}_n^m$ - znamená, že výraz se vyskytuje v počtu nejméně n a nejvýše m (ekvivalent operace mocniny od n do m)
$[\alpha]$ - znamená, že výraz se může a nemusí na daném místě vyskytnout - je to ekvivalentní zápisu $\{\alpha\}_0^1$

Příklad:

Gramatika z předchozího příkladu by v BNF mohla být zapsána například takto:

$\langle aritmetickyvyraz+ \rangle ::= \langle aritmetickyvyraz* \rangle \{+ \langle aritmetickyvyraz* \rangle\}$
$\langle aritmetickyvyraz* \rangle ::= \langle operand/podvyraz \rangle \{* \langle operand/podvyraz \rangle\}$
$\langle operand/podvyraz \rangle ::= (\langle aritmetickyvyraz+ \rangle) | x$

BNF umožňuje přehledný zápis a navíc i jednoduchý přechod k některým typům SA, S pomocí BNF je zapsána například celá gramatika jazyka Pascal v učebnici [13]. Příkladem může být deklarace podmíněného příkazu:

$\langle podminenyprikaz \rangle ::= if \langle booleovskyvyraz \rangle then \langle prikaz \rangle [else \langle prikaz \rangle]$

Pro definici našeho logického výrazu použijeme následující gramatiku:

$\langle Exp1 \rangle ::= \langle Exp2 \rangle \{= \langle Exp2 \rangle\}$
$\langle Exp2 \rangle ::= \langle Exp3 \rangle \{> \langle Exp3 \rangle\}$
$\langle Exp3 \rangle ::= \langle Exp4 \rangle \{| \langle Exp4 \rangle\}$
$\langle Exp4 \rangle ::= \langle ICE \rangle \{\& \langle ICE \rangle\}$
$\langle ICE \rangle ::= \sim \langle ICE \rangle | (\langle Exp1 \rangle) | 0 | 1 | | A | \dots | Z | a | \dots | z$

Význam symbolů:

0 - nepravda , 1 - pravda , A ... Z, a ... z - atomy (logické proměnné), $\sim$ - negace , = - ekvivalence , > - implikace , | - disjunkce , & - konjunkce

2 Metoda rekurzivního sestupu pro syntaktickou analýzu

Prvním důležitým úkolem při překladu z nějakého zdrojového jazyka do cílového je především syntaktická analýza - tedy kontrola, zda je text ve zdrojovém jazyce správně zapsán. V nejjednodušším případě pouhé kontroly typu ANO/NE (program je správně/není správně syntakticky zapsán) se jedná o takzvanou syntaktickou analýzu (dále budeme zkracovat SA). V anglicky psaných zdrojích se setkáte spíše s jednoslovným označením "parsing". O daném postupu - algoritmu jak tuto SA provést, pak hovoříme jako syntaktickém analyzátoru (anglicky "parser").

Velice oblíbenou, jednoduchou a přehlednou metodou vedoucí přímo k hotovému a dobře čitelnému analyzátoru v libovolném strukturovaném programovacím jazyce je metoda rekurzivního sestupu (Recursive Descent Parsing) [2]. Metoda je založena na principu analýzy "shora dolů", tedy se snažíme hledat odvození slova podle dané gramatiky od počátečního neterminálu v hierarchii směrem "dolů".

Metoda rekurzivního sestupu spočívá v konstrukci procedur strukturovaného programovacího jazyka přesně dle Backusovy-Naurovy formy (BNF), kde je každému neterminálu přiřazena jedna procedura a je volána procedura GetChar (načítající vždy následující symbol slova) před každým terminálem. Výskyt neterminálu v pravidle je v proceduře nahrazen rekurzivně voláním příslušné procedury. Iterace a podmínka v Backus-Naurově formě je nahrazena jednoduše jejich programátorskými protějšky. Vyžaduje se, aby jazyk byl definován tzv. LL(k) gramatikou (detailní podmínky v [7], pro účely tohoto článku budou takovéto gramatiky vytvořeny). Z hlediska časové složitosti (obecně) jde opět o obecně neefektivní metodu s exponenciální časovou složitostí, nicméně pro jednoduché gramatiky z praxe je použitelná a zejména je její výhodnou vysoká čitelnost kódu ve vztahu k výchozí gramatice. Navíc existuje modifikace tzv. packrat parser, která pro omezenou třídu takzvaných parsing expression grammars pracuje v **lineární čase**. Také je tento postup flexibilní, protože umožňuje kdykoliv změnit a přidat syntaktický element bez nutnosti měnit celý kód, ale pouze dotčenou část gramatiky (například změna struktury číslo z celého čísla na reálné znamená pouze změnu procedury reprezentující tento element). Podívejme se nyní na příklad gramatiky pro generování aritmetických výrazů se sčítáním, násobením, číslicemi a vnořenými závorkovanými strukturami. Vyhodnocování takového výrazu strojově je pak velmi jednoduché pomocí tzv. postfixové notace (reverzní polská notace), kde vždy platí, že operátory se vyskytují až za jeho operandy. Pro vyhodnocení takové notace nám postačí jen datová struktura typu zásobník a nijak to neovlivní lineární časovou složitost celého procesu včetně syntaktické analýzy.

Příklad:

Nejprve sestrojme mírně modifikovanou gramatiku (oproti příkladu z minulé kapitoly) v Backusově-Naurově formě:

$\langle Vyraz \rangle ::= \langle Term \rangle \{+\langle Term \rangle\}$
$\langle Term \rangle ::= \langle Faktor \rangle \{*\langle Faktor \rangle\}$
$\langle Faktor \rangle ::= (\langle Vyraz \rangle)|0|1|2||9$

Nyní se schématicky pokusíme ukázat (nejde o zcela hotový kód, ale o jeho fragmenty po částech, které byly dohromady úspěšně testovány), jak bychom sestrojili SA metodou rekurzivního sestupu pro tuto gramatiku v jazyce Pascal. Tento kód, pak umožňuje nejen SA, ale i detekci možných chyb. Nejprve sestrojíme proceduru, která zapouzdřuje celou činnost SA. Její hlavička může vypadat například takto:

```
program Preklad;
var ch:char;              {aktualni zpracovavany znak}

    infixProg, postfixProg:string;     {globalni prom. unitu pro uchovani vyrazu}
    errProg, posProg, infixpos:word;                   {globalni prom. cisla chyby}

procedure SyntaktickaAnalyza(infix:string;var err,pos:word; var postfix:string);
          {procedura analyzuje aritmetický výraz infix, postfix obsahuje
           postfixovou notaci vhodnou pro vyhodnoceni zasobnikem
           err obsahuje číslo chyby, pos obsahuje pozici ,kde analýza skončila}

procedure Term;forward;
procedure Faktor;forward;
```

Používají se proměnné infixpos (pozice aktuálně čteného znaku ze vstupu), ch (aktuální znak). Analyzátor dále obsahuje nezbytný lexikální analyzátor pro načítaní jednotlivých symbolů (v našem zjednodušeném případě jde o jednoznakové symboly). Lexikální analýza je realizována procedurou GetChar, která ukládá znak do proměnné ch a případně provede detekci chybové situace err=2, pokud načteme zcela nepřípustný znak. V rámci SA je pak otázka jen přidat několik vhodně umístěných přiřazení do výstupní proměnné postfix, kde je přeložený výraz do postfixové notace, kterou lze velmi jednoduše algoritmicky vyhodnotit.

```
procedure Getchar;       {čte znak z infixu do proměnné ch}
begin
  if err=0 then
    begin
      Inc(infixpos);
      if infixpos<=Length(infix) then ch:=infix[infixpos] else ch:=#0;
      ch:=Upcase(ch);
      if not((ch in ['0'..'9'])or(ch in ['(',')','*','+',#0])) then err:=2;
         {ošetření nežádoucích znaků}
    end;
end;
```

Jádrem analyzátoru jsou jednotlivé rekurzivní procedury Výraz (sčítaní), Term (násobení), Faktor (číslice, vnořený závorkovaný výraz). Výraz přesně podle BNF buď volá podřízený Faktor nebo čte terminální symboly.

```
procedure Vyraz;                {výraz s nižší prioritou}
begin
  if err=0 then
    begin
      Term;
      while (ch='+') do
        begin
                Getchar;        {sčítání}
                Term;
         postfix := postfix + '+';
        end;
    end;
end;

procedure Term;                {výraz s vyšší prioritou}
begin
  if err=0 then
    begin
      Faktor;
      while (ch='*') do
        begin
                Getchar;            {násobení}
                Faktor;
         postfix := postfix + '*';
        end;
    end;
end;
```

A dále musíme sestrojit proceduru pro Faktor, která bude mít vzhledem k jinému charakteru přepisovaného řetězce i jiný kód.

```
procedure Faktor;   {synt. analýza operandu}
begin
  if err=0 then
    begin
      case ch of
      '0'..'9':
          begin
            postfix:=postfix+ch;                  {anal. číslic}
            Getchar;
          end;
      '(':begin
            Getchar;
```

```
                    {analýza výrazu se závorkou}
          Vyraz;
          if (ch<>')')and(err=0) then err:=4   {chyba- není ukončen závorkou}
          else if err=0 then
            begin
              Getchar;
            end;
        end;
    else if err=0 then err:=5;    {nebyl detekován ani vyraz,ani číslice}
    end;
  end;
end;
```

Faktor tedy rozlišuje dvě situace - buď jde o číslici nebo jde o výraz začínající závorkou a ukončený opačnou závorkou. Logicky tedy můžeme odhalit další dvě chyby (err=4, když chybí závorka, err=5, když není detekován ani výraz ani číslice). Postfixová notace se generuje postupně - každá číslice se vloží okamžitě po načtení, znak operátoru se přidá, až po zpracování podstromu. Pozn. Nebyl by problém se zcela vyhnout tvorbě postfixu a stejným rekurzivním principem přímo vyčíslovat postupně hodnotu výrazu (procedury Faktor, Term a Výraz by pak měly návratovou hodnotu rovnou výsledku po aplikaci všech operací na dané úrovni podvýrazu).

Pozn. Samozřejmě, že chybové detekce by mohly odhalit ještě další problematické konstrukce - např. skončení nejvyššího volání procedury Výraz před přečtením posledního znaku apod. Vlastní tělo procedury pro SA, pak provede volání počátečního neterminálu a odhalí některé chyby dodatečně po přečtení výrazu např. že sice skončila nejvyšší procedura Vyraz, ale ještě ve vstupu jsou nějaké znaky.

```
begin
  err:=0;           {inicializace prom.}
  infixpos:=0;
  postfix:='';
  Getchar;
  Vyraz;            {zavolani syntakticke analyzy vyrazu}
  if (ch=')') and (err<>5) then
    begin
      err:=6;
      pos:=0;
    end;     {osetreni prebytecne prave zavorky}
  if (ch<>#0) and (err=0) then err:=1;  {osetreni konce kompilace-neni konec
                                          infixu}
  pos:=infixpos;                {nastaveni navratovych hodnot}
end;
```

Nakonec můžeme v hlavním programu tuto proceduru použít.

```
begin
  infixProg := '5+3*2';
  SyntaktickaAnalyza(infixProg, errProg, posProg, postfixProg);
  Writeln('Doslo k chybe:', errProg);
  if err<>0 then exit;
  Writeln('Postfixova notace:', postfixProg);
  readln;
end;
```

Ilustrujme nyní průběh výpočtu procedury Výraz na výrazu $5 + 3 * 2$.

Infixová notace	**Aktuální znak**	**Aktuální procedura**	**Návrat do procedury**
5 + 3 * 2	5	Výraz	
5 + 3 * 2	5	Term	
+ 3 * 2	+	Faktor	Term
+ 3 * 2	+	Term	Výraz
3 * 2	3	Výraz (+)	
3 * 2	3	Term	
* 2	*	Faktor	Term
2	2	Term (*)	
		Faktor	Term (*)
		Term (*)	Výraz (+)
		Vyraz (+)	

Obrázek 2: Průběh rekurzivního sestupu

Rozeberme nyní jádro vyhodnocení výrazu v infixní formě. Toto jádro provádí kompilaci aritmetického výrazu v infixové (tedy přirozené) notaci do notace postfixové. Ta je vhodná pro zpracování pomocí počítače např. vyhodnocení pomocí zásobníku. Aritmetický výraz v infixové (přirozené) notaci 5 + 3 * 2 lze pomocí této procedury přeložit na výraz v postfixové notaci 5 3 2 * +. Ta je konstruována tak, že místo tvaru, kde je operátor mezi operandy, má operátor až za oběma operandy.

Tento výraz lze pak pomocí zásobníku vyhodnotit tak, že čteme jednotlivé symboly a provádíme s nimi tyto dvě operace:

1. Je-li čtený symbol operandem, pak ulož operand na zásobník.

2. Je-li čtený symbol operátorem, pak vyber ze zásobníku posledních n operandů (kde n je arita operátoru; např. pro + je n = 2). Proveď operaci dle operátoru s vybranými operandy a výsledek ulož na zásobník.

Pro výraz 5 + 3 * 2 vezměme jeho postfixovou notaci 5 3 2 * + a vyhodnoťme jej s pomocí zásobníku.

Nepřečtená část	Aktuální znak	Zásobník	Vybírané symboly	Operace
5 3 2 * +	5			
3 2 * +	3	5		
2 * +	2	3 5		
* +	*	2 3 5	2 3	2 * 3 = 6
+	+	6 5	6 5	6 + 5 = 11
		11		

Obrázek 3: Průběh rekurzivního sestupu

Obsah zásobníku po přečtení slova je roven hodnotě výrazu. Postup by samozřejmě bylo možno zobecnit na složitější čísla nebo proměnné, ale vyžadovalo by to složitější struktury zásobníku.

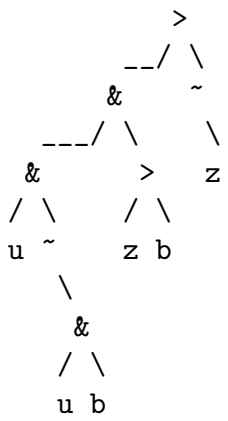
3 Grafová (stromová) reprezentace logických výrazů

Jak jsme již pochopili v předchozí kapitole, není zápis v klasickém infixním formátu pro počítač příliš výhodný, i když my lidé jsme na něj zvyklí a umíme s ním díky výuce matematiky dobře pracovat. To platí o výrazech i strukturovaných kódech všeho druhu - programy, texty, aritmetické a logické výrazy. Ukázali jsme si způsob, jak lze jednoduše vytvořit jinou notaci (v podstatě jde jen o pořadí operátorů a operandů), kterou už počítač dokáže relativně lehce zpracovat (s pomocí zásobníku). Existuje však mnohem univerzálnější a pro algoritmizaci složitých procesů velmi vhodná metoda a to je reprezentace pomocí syntaktických stromů. Syntaktický strom je v podstatě orientovaným grafem s ohodnocenými uzly neobsahujícím cykly a u kterého, lze určit kořen stromu (jakýsi nejvyšší prvek). Každý uzel, pak může mít své potomky k nimž vedou hrany a ty uzly, které potomky nemají se nazývají listy stromu (graf tedy opravdu připomíná živý strom). Pro jednoduchost budeme stromy reprezentovat textově - jde o výstupy z již zmiňovaného doplňku vytvořeného pro

čtenáře v rámci aplikace Bachelor [4]. Ovšem pozor, tento doplněk je obsažen v nabídce Formula, Tree jen ve verzi pro Windows 32-bit, tedy **WinBch95.exe**.

Příklad:

Mějme logickou formuli - `u&~(u&b)&(z>b)>~z`.

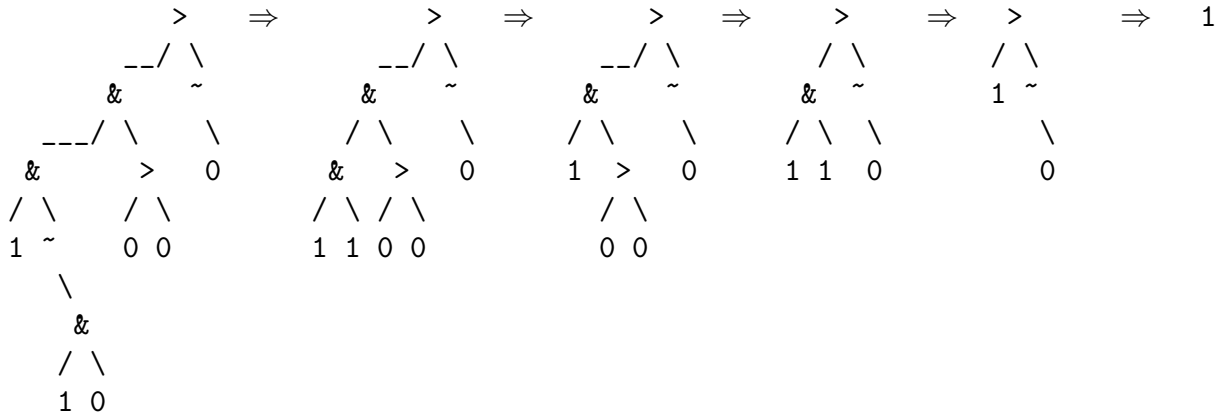


Tento strom výstižně a hlavně pro počítač dostatečně jednoduše postihuje celou hierarchii formule. V kořeni stromu je spojka implikace, což by počítač z infixní formy určoval velmi složitě (závorky, priority operátorů), zatímco pracuje-li se stromem, vše je z hierarchické struktury stromu jasné. Implikace má jako podstromy (spojené znaky / a \ a pomocnými znaky _) konjunkci složené formule a negace atomu z. Strom vůbec nemusí obsahovat závorky, neboť priorita operátorů je obsažena už v samotném stromu a není důvod ji narušovat nějakým speciálním symbolem jako jsou závorky. Algoritmy pro manipulace, změny a výpisy stromu mohou být navíc velmi chytře popsány rekurzivními algoritmy, které jsou pak velmi jednoduché (krátké).

Například pokud bychom znali hodnoty logických proměnných u, z a b, pak spočtení hodnoty výrazu se dá popsat jednoduchým rekurzivním pravidlem P:

1. Pokud je uzel listem vrať hodnotu proměnné,
2. Pokud uzel není listem, spočítej hodnoty všech potomků uzlu pravidlem P a pak tyto hodnoty zpracuj operací, která je obsažena v uzlu, a vrať výsledek (negaci logické konstanty spočítej přímo).

Podívejme se na to ilustrativně na samotném stromu. Vezměme stejný strom, pouze dosaďme hodnoty proměnných do listů $u = 1, z = 0, b = 0$.



4 Konstrukce stromu

Viděli jsme, že práce se stromem je pro počítač velmi příjemná a dá se formulovat takřka triviálními algoritmy (a bude platit i u mnohem složitějších operacích, jak uvidíme dále). Otázkou zůstává, jak s pomocí již známého rekurzivního sestupu a postfixového způsobu zápisu výrazu sestrojit syntaktický strom. Principiálně to není příliš složité, pouze technické zpracování vyžaduje dovednosti při programování dynamických datových struktur a práci s ukazateli. Využijeme stejného algoritmu se zásobníkem, jakým bychom vyhodnocovali aritmetický výraz, pouze změníme vykonávané akce. Místo vložení číslice do zásobníku vytvoříme dynamickou proměnnou typu záznam, kterou vložíme do zásobníku. Tento záznam reprezentuje list stromu. Pokud v nějakém místě programu bychom podle starého postupu vkládali operátor, pak namísto toho vytvoříme opět dynamický záznam. Ten obsahuje odkaz na své nejvýše dva potomky (podvýrazy spojené logickou spojkou) - těmito odkazy ukážeme na poslední dva prvky v zásobníku, které předtím vybereme. Místo nich pak vložíme hotový záznam se symbolem daného operátoru. Na konci budeme mít ze správně utvořeného logického výrazu mít provázaný syntaktický strom a jeho kořen bude k dispozici na vrcholu zásobníku. Následující fragment kódu aplikace Bachelor ukazuje, jak vypadá datová struktura pro uložení jednoho uzlu stromu TSTreeNode a dále jak vypadá zapouzdřovací záznam pro dočasné uložení do zásobníku (jakýsi "obal", který potřebujeme pro uchování v zásobníkové paměti).

```
type

    PSTreeNode=^TSTreeNode;   {uzel stromu}
    TSTreeNode=record
     character:char;          {symbol : logická spojka,
                               proměnná nebo logická konstanta}
     prior:byte;              {priorita symbolu - potřebujeme ji pouze
               ke zpětnému výpisu do infixní podoby - tvorba závorek}
     neg:boolean;             {příznak zda je uzel negován
               - nevytváříme zbytečně v programu další větev}
     left:PSTreeNode;         {ukazatel na levý podstrom (podvýraz) - potomka}
     right:PSTreeNode;        {ukazatel na pravý podstrom (podvýraz) - potomka}
    end;


    PSStackNode=^TSStackNode; {obal pro uchování v zásobníku}
    TSStackNode=record        {během překladu do stromu}
     node:PSTreeNode;         {ukazatel na vložený uzel}
     next:PSStackNode;        {ukazatel na následující obal v zásobníku}
    end;
```

Princip tvorby stromu během rekurzivního sestupu si ukážeme na fragmentech analyzá-

toru. Popsat celý analyzátor by zabralo mnoho stran a principiálně nejde o odlišný případ, jaký jsme již předvedli na aritmetickém výrazu. Čtenář má navíc možnost se podívat do zdrojových kódů aplikace Bachelor [4]. Ke všem neterminálům jsou v souladu s definicí BNF našeho jazyka sestrojeny procedury rekurzivního sestupu.

```
...
procedure Exp3;
 begin
  if Error=OK then
    begin
      Exp4;
      while (ch='|')and(error=OK) do
        begin
         Getchar;
         Exp4;

         if error=OK then Action('|',3);

        end;
    end;
 end;
...
procedure ICE;
 begin
 if error=OK then
  begin
   case ch of
    '~':begin
          Getchar;
          ICE;
          VPSStack^.node^.neg:=not(VPSStack^.node^.neg);
        end;
    'a'..'z','A'..'Z','0'..'1':
        begin
          Put(ch,6);
          Getchar;
        end;
    '(':begin
          Getchar;
          Exp1;
          if (ch<>')')and(error=OK) then Error:=missbra;
          Getchar;
        end
```

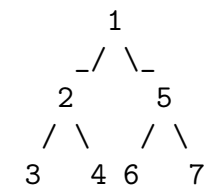
```
    else Error:=missexp;
    end;
  end;
 end;
```

Vidíme, že vložení listu do zásobníku nám realizuje procedura Put, která vytvoří daný uzel a jeho obal s příslušným znakem a prioritou. Pokud nám přijde znak negace, změníme u posledního vloženého atomu negaci na opačnou (teoreticky může obsahovat libovolné, množství negací za sebou). V programu je definováno mnoho chyb, které může výraz obsahovat - zde například chyba missexp vyjadřuje chybějící podvýraz a missexp chybějící uzavírací párovou závorku. Navázání vytvoření a navázání potomků uzlu, který není list (operátor), nám zajistí procedura Action.

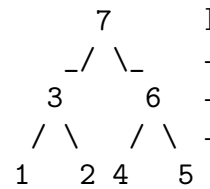
```
procedure Put(var chp:char;pr:byte);  {vytvoření a vložení atomu}
    begin
      new(pom); {alokace nového dynamického záznamu - obalu}
      pom^.next:=VPSStack; {původní vrchol zásobníku musíme navázat na nový}
      VPSStack:=pom; {nový obal je první v zásobníku - vrchol}
      new(pom^.node); {v obalu vytvoříme uzel stromu}
      VPSStack^.node^.character:=chp; {uzel má požadovaný znak}
      pom^.node^.left:=nil;
      pom^.node^.right:=nil; {levý i pravý podstrom atom - proměnná
                    nebo konstanta - nemá! tedy ukazuje na nil}
      pom^.node^.prior:=pr; {nastavíme prioritu podle požadavku}
      pom^.node^.neg:=false; {prvotní vytvoření nedělá negaci}

    end;
...
procedure Action(chr:char;pr:byte);   {vlož operátor do zásobníku}
 begin
   Put(chr,pr); {vlož do zásobníku nový obal s uzlem
                o požadovaném znaku operace a prioritě}
   pom:=Cut; {vyber operátor ze zásobníku - budeme s ním pracovat}
   pom^.node^.right:=VPSStack^.node; {navaž první uzel na zásobníku
                    jako pravý podstrom - pozor v zásobníku je pořadí obrácené}
   garbage:=Cut; {vyber tento uzel ze zásobníku a znič jeho obal}
   dispose(garbage);
   pom^.node^.left:=VPSStack^.node; {navaž nyní první uzel
                             na zásobníku jako levý podstrom}
   garbage:=Cut; {Vyjmi jej ze zásobníku}
   dispose(garbage); {znič obal}
   SInsert(pom); {vlož obal a uzel operátoru do zásobníku}
 end;
```

5 Optimalizace výrazu, logické ekvivalence a normální formy

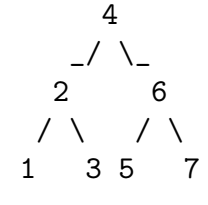
Vytvořený syntaktický strom nám dává možnost pracovat algoritmicky se strukturou formule. Strom můžeme procházet principiálně různými způsoby podle toho, co je cílem algoritmu. Existují 4 základní možnosti průchodu pre-order, in-order, post-order a level-order. Anglické slovíčko "order" jasně říká, co se myslí průchodem - jde o pořadí v jakém pracuje s uzly. Na následujících schématech lze pozorovat v jakém pořadí budou danou metodou zpracovány uzly (číslo je pořadí). Pravidlo se vždy aplikuje rekurzivně (s mírnou výjimkou u Level-order) - tedy používá samo sebe na potomky uzlu. Na uzel aplikujeme vybranou operaci (třeba výpis znaku uzlu). Trochu výjimečná je procedura průchodu Level-order, kde musíme implementovat frontu s ještě nezpracovanými uzly a ty postupně obsluhovat.



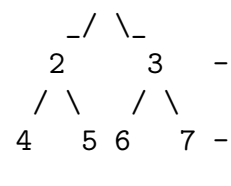
Pre-order
- operace s uzlem
- Pre-order na levý uzel
- Pre-order na pravý uzel



Post-order
- Post-order na levý uzel
- Post-order na pravý uzel
- operace s uzlem



In-order
- Inorder na levý uzel
- operace s uzlem
- In-order na pravý uzel



Level-order
- operace s uzlem (n-úroveň)
- vygeneruj uzly nižší $n+1$ úrovně a dej do fronty

Začni od úrovně 0 - kořene stromu
Aplikuj iterativně na všechny uzly ve frontě Level-order

Využití různých způsobu průchodu se dá ilustrovat na následujících příkladech:

- Výpis výrazu do infixní formy - použijeme In-order, protože chceme mít vždy binární spojky v pořadí operand operátor operand.
- Vyhodnocení výrazu - použijeme Post-order, neboť výraz se vyhodnocuje "zdola", tj. než začneme vyhodnocovat spojku, musíme oba podvýrazy již mít vyčíslené
- Odstranění negace směrem na proměnné - použijeme Pre-order, jedná se o často prováděnou operaci v logice, chceme aby negace byla jen u logických proměnných, musíme tedy negaci postupně shora od kořene tlačit směrem dolů pomocí logických zákonů jako jsou De-Morganovy zákony, např. $\neg(A \wedge B) \Leftrightarrow (\neg A \vee \neg B)$.

Podívejme se blíže na algoritmus pro výpis formule v infixním tvaru čitelném pro uživatele. Rekurzivní procedura typu In-order Dive provede zejména volání sama sebe pro levý podstrom pro logickou spojku, pak se vždy vypíše znak atomu a nakonec se zavolá Dive pro pravý podstrom u spojky. Samozřejmě je třeba vyřešit závorkování (infixní forma, na kterou jsou lidé zvyklí, se bez závorek neobejde narozdíl od postfixu a stromu). Detaily jsou popsány v komentářích zdrojového kódu.

```
procedure TEditForm.Dive; {vypisuje rekurzivně formuli v infixním tvaru}
   begin
         if (node^.neg=true) then
           begin  {pokud je uzel negován musíme vložit znak negace}
             InsertString('~');
             if node^.prior<>6 then
             {ale pokud to není atom jsou nutné závorky}
               begin InsertString('('); end;
           end;
         if (node^.prior <> 6) then
         {není-li atom budeme vypisovat
                        levý podstrom}
           if (node^.prior>node^.left^.prior)and(not(node^.left^.neg)) then

           {pokud má operátor levého podvýrazu nižší prioritu musíme
           vložit závorku, jinak by se provedl nejprve uzel, který právě řešíme}

              begin
                InsertString('(');

                {volá se Dive pro levý podstrom}
                Dive(node^.left); InsertString(')');
              end else Dive(node^.left); {volání bez závorek}
         InsertString(node^.character);              {vložíme znak spojky uzlu}
         if (node^.prior <> 6) then {není-li atom vypíšeme pravý podstrom}
           if ((node^.prior>node^.right^.prior)or((node^.prior=2)
              and(node^.right^.prior=2)))and(not(node^.right^.neg)) then

{pokud má operátor pravého podvýrazu nižší prioritu musíme vložit závorku, jinak
by se provedl nejprve uzel, který právě řešíme Pozor! Pro pravý podstrom je třeba
závorka, pokud je uzel i jeho pravý podvýraz implikace - není asociativní}

              begin
                InsertString('(');

                {volá se Dive pro pravý podstrom}
```

```
              Dive(node^.right); InsertString(')');
            end
          else Dive(node^.right); {volání bez závorek}
        if (node^.prior<>6)and(node^.neg=true) then
          begin
            InsertString(')');{ukončení závorky pro počáteční negaci}
          end;
  end;
```

Ve výuce se logika na SŠ většinou vyučuje jen v rámci matematiky a to způsobem, který odpovídá jejímu využití v matematice - tedy především jako formální aparát pro vyjadřování matematických vztahů. Většinou se spokojíme s výukou sémantiky logických spojek, případně s pojmem tautologie, ale existují také jednoduché a čistě formální metody na zjednodušování formulí a případně i určování platnosti, splnitelnosti a ověřování správnosti dedukce. Důležité je, že daná pravidla jsou poměrně jednoduchá a pracují výlučně se syntaxí formule, kterou nyní máme zapsánu velmi vhodně formou stromu. Formální pravidla známe i z jiných oblastí, například z teorie množin, kde třeba známe a často používáme vztah mezi dvěmi množinami a operacemi rozdílu a průniku:

$$A - B = A \cap \overline{B}$$

Podobně lze upravovat nejen výrazy s rovnicemi, ale i logické výrazy. Nejprve se podívejme na vyhodnocení logických konstant.

Vyhodnocení logických konstant

Pro vyhodnocení logických konstant může využít následující zákony (ekvivalence):

$$A \wedge B \Leftrightarrow B \wedge A, A \vee B \Leftrightarrow B \vee A, A \leftrightarrow B \Leftrightarrow B \leftrightarrow A$$

$$\neg 0 \Leftrightarrow 1, \neg 1 \Leftrightarrow 0, A \wedge 0 \Leftrightarrow 0, A \wedge 1 \Leftrightarrow A, A \vee 0 \Leftrightarrow A, A \vee 1 \Leftrightarrow 1, A \leftrightarrow 0 \Leftrightarrow \neg A, A \leftrightarrow 1 \Leftrightarrow A$$

$$A \rightarrow 1 \Leftrightarrow 1, A \rightarrow 0 \Leftrightarrow \neg A, 1 \rightarrow A \Leftrightarrow A, 0 \rightarrow A \Leftrightarrow 1$$

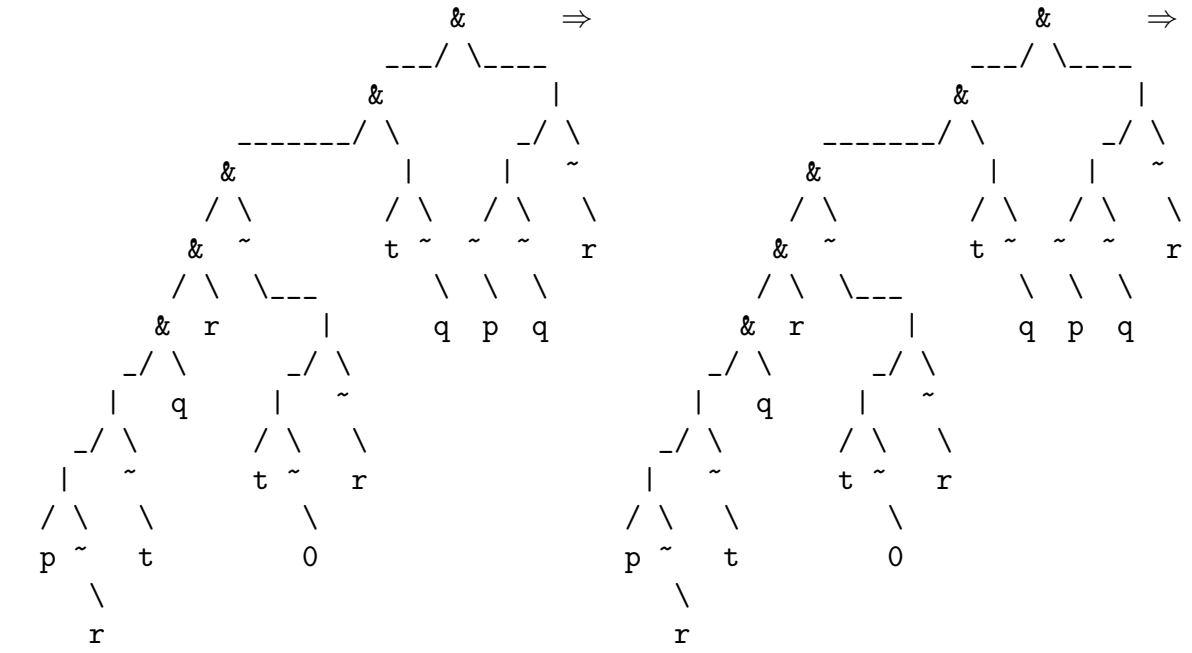
Tyto ekvivalence může zcela přirozeně naimplementovat do systému, který pracuje se syntaktickým stromem a pak už jen stačí napsat algoritmus, který vhodně prochází strom a aplikuje je. Vlastní procházení typu Post-order realizuje procedure RConstEval (uvádíme jen fragment) a každý uzel ja pak testován procedurou , zda se v něm díky logické konstantě nemůže zjednodušovat podle výše uvedených zákonů.

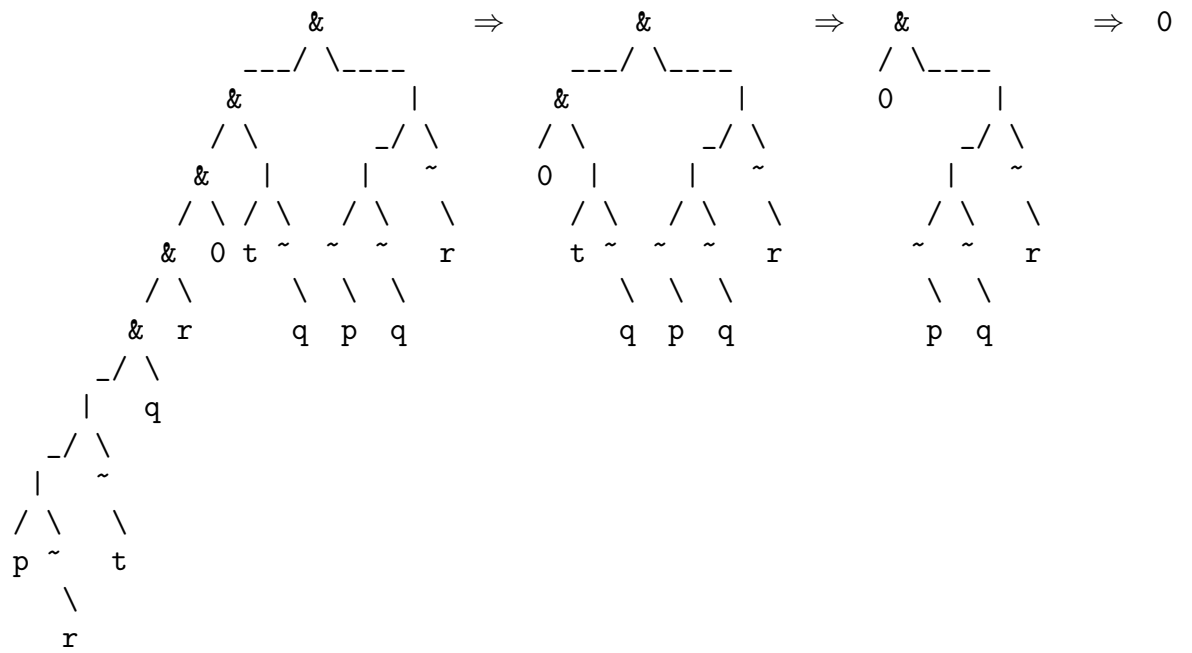
```
procedure TEditForm.RConstEval;
    begin
      if  (tr^.prior <> 6) then
        begin
```

```
          RConstEval(tr^.left); RConstEval(tr^.right); EvlConst1(tr);
        end; {levý podstrom, pravý podstrom, pak teprve uzel}
    end;
...
procedure TEditForm.EvlConst1;
 {pokus o vyhodnocení logické pravdy nebo nepravdy}
 ...
 begin
 ...
 case pchar of
      '=':begin
            if pompt2^.character='0' then pompt1^.neg:=not(pompt1^.neg);
            p:=pompt1;RdisposeTree(pompt2);
          end;{ekvivalence se vyhodnotí přesně dle ekvivalencí}
              { ( A = 1 ) = A
                ( A = 0 ) = ~A }
 ...
 end;
```

Podívejme se také na příklad takového zjednodušení, které může být velmi účinnou optimalizací a ušetřit nám zbytečné vyhodnocování velké části podmínky. Mějme poměrně složitou podmínku -logický výraz - který obsahuje jednu konstantu 0 (třeba bychom ji dostali po vyhodnocení jednoho porovnání v nějakém programovacím jazyce a nějakém programu v něm vytvořeném).

```
(p|~r|~t)&q&r&~(t|~0|~r)&(t|~q)&(~p|~q|~r)
```





Další zajímavou transformací logického výrazu je převod na tzv. funkčně úplnou množinu spojek. V logice existují kombinace spojek, pomocí kterých můžeme vyjádřit všechny formule jako ekvivalentní. To by mohlo být zajímavé v případě, že stroj, který pro zpracování našich logických výrazů použijeme, umí velmi rychle oproti jiným spojkám, řešit nějakou konkrétní spojku (procesory počítačů i jiné logické obvody mohou tuto vlastnost mít). Druhou možností jsou například některé důkazové metody (například metoda "reductio ad absurdum", nepřímý důkaz, vyžaduje formule ve formě implikací). Funkčně úplné množiny spojek jsou například:

- $\{\neg, \rightarrow\}, \{\rightarrow\}$ - negace a implikace, či dokonce pouze implikace (ovšem s pomocí logických konstant)!
- $\{\neg, \wedge, \vee\}, \{\neg, \wedge\}, \{\neg, \vee\}$ - negace, konjunkce, disjunkce či dokonce negace pouze s jednou těchto spojek; u první množiny lze dostat tvar tzv. negační normální formy - kdy negace se vyskytuje pouze u výrokových proměnný (lze ji vytlačit směrem dolů ve stromě až na listy)

Při převodu výrazu na tyto množiny spojek můžeme využít například tyto zákony (přepis mezi implikací a disjunkcí a konjunkcí, odstranění ekvivalence, vytlačení negace - De Morganovy zákony):

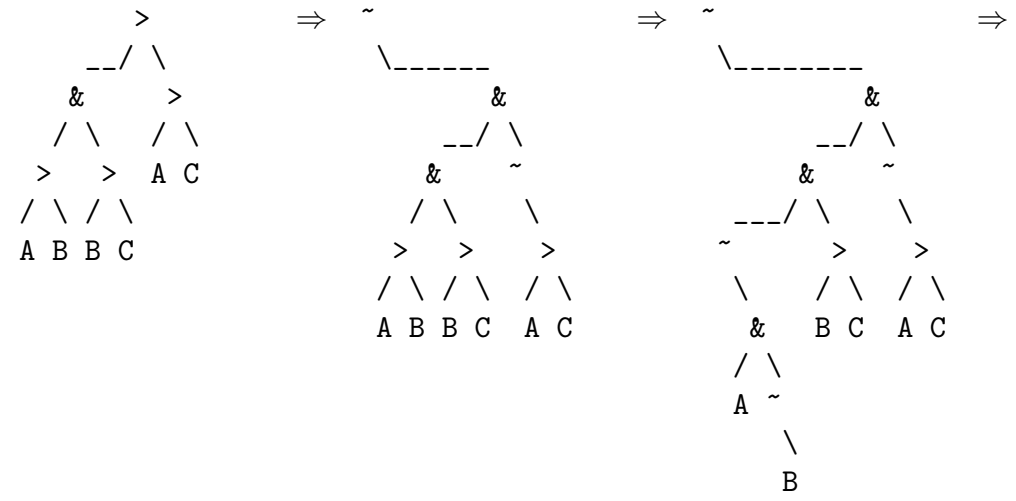
$$A \wedge B \Leftrightarrow \neg(A \rightarrow \neg B), A \vee B \Leftrightarrow \neg A \rightarrow B, A \leftrightarrow B \Leftrightarrow (A \rightarrow B) \wedge (B \rightarrow A), \neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B, \neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B, \neg\neg A \Leftrightarrow A$$

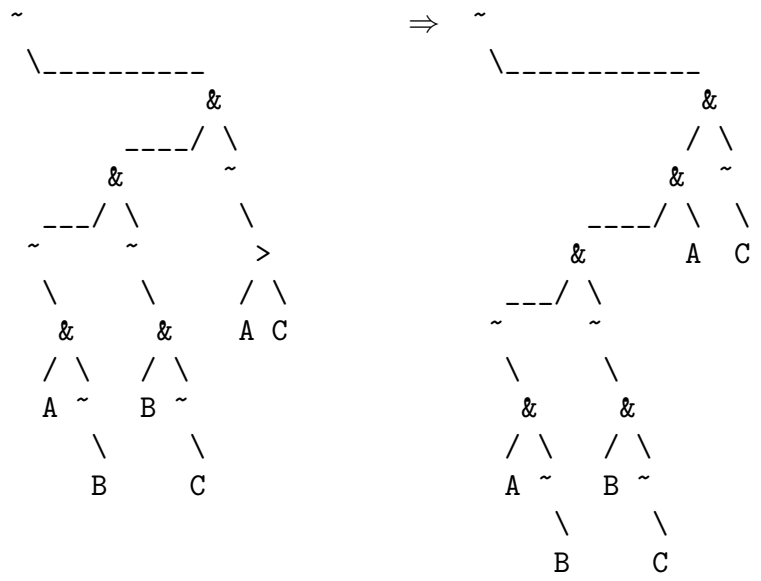
Tyto zákony lze implementovat průchodem typu Pre-order (nejprve se musí změnit spojka na vyšší úrovni a pak se může pokračovat dále na nižší). Podívejme se pro příklad na fragment algoritmu na převod na konjunkce a negace. Procedura Conjunction provede rekurzivně změny spojek a příslušné změny na negované formule dle ekvivalencí.

```
procedure TEditForm.Conjunction;{konverze na konjukce a negace}
...
     else if p^.character = '>' then
         begin {implikaci změníme na konjunkci}
           ChangeOper(p,pchar);p^.neg:=not(p^.neg);
           {pozor nová konjukce je negovaná}
           p^.right^.neg:=not(p^.right^.neg);
           {pozor pravý podstrom je také negovaný}
           ...
         end;
         { ( A > B ) =  ~ ( A & ~ B )  }
     end;
  end;
```

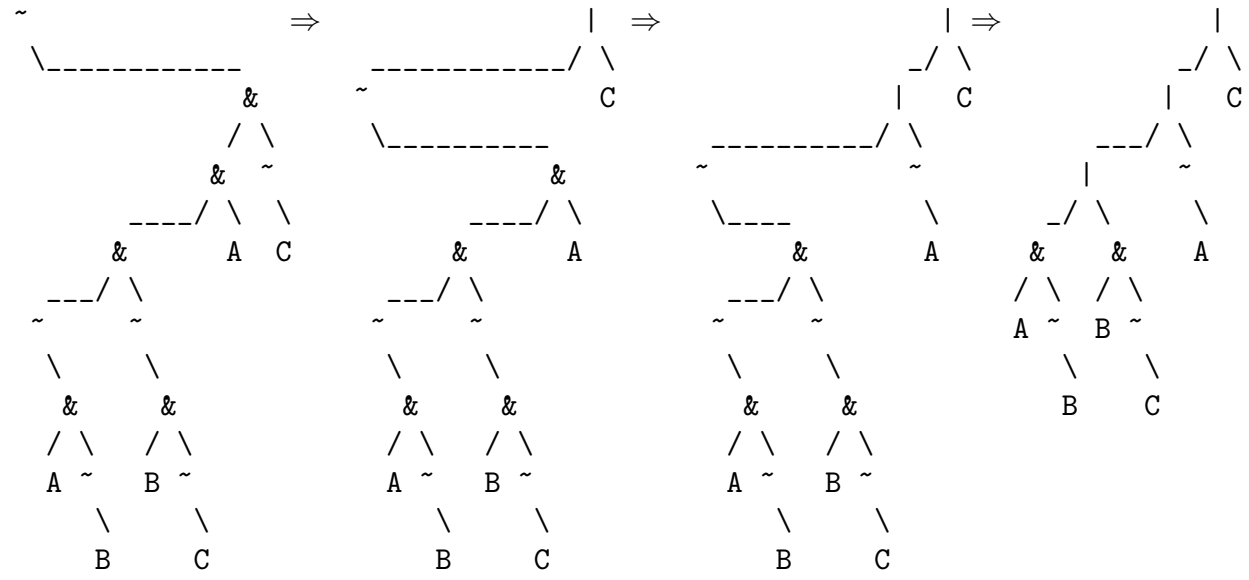
A nyní příklad, jak lze nejprve formuli převést pouze na konjunkce, negace a pak na konjunkce, disjunkce a negace, kde negace budou jen na proměnných. Výchozí formule je následující (vlastně jde o známé pravidlo tranzitivity):

```
(((A>B)&(B>C))>(A>C))
```





A nyní výsledné pravidlo převedeme do negační normální formy.



Nyní si oba převody ještě okomentujeme v infixní formě:

```
(((A>B)&(B>C))>(A>C))
Implikace na konjunkci:~((A>B)&(B>C)&~(A>C))
Implikace na konjunkci:~(~(A&~B)&(B>C)&~(A>C))
Implikace na konjunkci:~(~(A&~B)&~(B&~C)&~(A>C))
Implikace na konjunkci:~(~(A&~B)&~(B&~C)&A&~C)
~(~(A&~B)&~(B&~C)&A&~C)

Conjunction to disjunction:~(~(A&~B)&~(B&~C)&A)|C
```

```
Konjunkce na disjunkci:~(~(A&~B)&~(B&~C))|~A|C
Konjunkce na disjunkci:A&~B|B&~C|~A|C
A&~B|B&~C|~A|C
```

Nakonec se seznámíme s nejdůležitějším pojmem tzv. konjunktivní normální formy výrazu. Ten nám také umožní dokonce formálně prověřovat dedukci [10].

Konjunktivní normální forma (KNF) výrazu se skládá z tzv. konjunktů spojených spojkou konjunkce (konečný počet). Konjunkt je tvořen z výrokových proměnných bez nebo s negací spojených výhradně spojkou disjunkce. Tento tvar tedy z hlediska stromu obsahuje směrem od kořene dolů konjunkce a platí, že od první disjunkce se už směrem dolů vyskytují jen disjunkce nebo atomy s negací a bez.

Disjunktivní normální forma (DNF) výrazu je duální ke KNF a skládá se z tzv. disjunktů spojených spojkou disjunkce (konečný počet). Disjunkt je tvořen z výrokových proměnných bez nebo s negací spojených výhradně spojkou konjunkce. Tento tvar tedy z hlediska stromu obsahuje směrem od kořene dolů disjunkce a platí, že od první konjunkce se už směrem dolů vyskytují jen konjunkce nebo atomy s negací a bez. Příkladem DNF je vygenerovaná formule z minulého příkladu.

```
A&~B|B&~C|~A|C
```

KNF se používá především v oblasti automatizovaného dokazování při použití klauzulární rezoluční metody a DNF se dá využít pro zjištění platnosti formule (zda je tautologie) - DNF tautologie by po zjednodušení měla degradovat na 1 (pravdu). DNF je navíc velmi expresivní, pokud chceme určit, za jakých pravdivostních hodnot výchozích proměnných je výraz pravdivý. Druhá stránka věci je také pro praxi velice užitečná, neboť jednoduchými pravidly lze DNF i KNF **minimalizovat**, což může významně zjednodušit a zmenšit formuli (samozřejmě menší formule, znamená mnohem **méně vyhodnocování při výpočtu výrazu**). Nejprve si projděme pravidla pro přepis do normálních forem pomocí ekvivalencí (existují samozřejmě i sémantické metody, kdy z tabulky můžeme vytvořit tzv. úplné normální formy). Ekvivalence, kromě již výše použitých pravidel, zahrnují především dvě pravidla, která umožňují změnit hierarchii negace mezi spojkami konjunkce a disjunkce (jde o distributivní zákon), dále zákony absorpce, absorpce negace, idempotence a komplementarita, a nakonec rozšíření.

$$A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C), A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C),$$
$$A \vee (A \wedge B) \Leftrightarrow A, A \wedge (A \vee B) \Leftrightarrow A, A \vee (\neg A \wedge B) \Leftrightarrow A \vee B, \neg A \vee (A \wedge B) \Leftrightarrow$$
$$\neg A \vee B, A \wedge (\neg A \vee B) \Leftrightarrow A \wedge B, \neg A \wedge (A \vee B) \Leftrightarrow \neg A \wedge B,$$
$$A \vee A \Leftrightarrow A, A \wedge A \Leftrightarrow A, A \vee \neg A \Leftrightarrow 1, A \wedge \neg A \Leftrightarrow 0,$$
$$(A \wedge \neg B) \vee (A \vee B) \Leftrightarrow A$$

Pozn. V automaticky vyhodnocených příkladech se používá v algoritmech někdy pro jednodušší manipulaci (např. u zákona rozšíření) přiřazení logických konstant za A a B, tak aby výsledek po vyhodnocení odpovídal ekvivalenci.

Příklad:

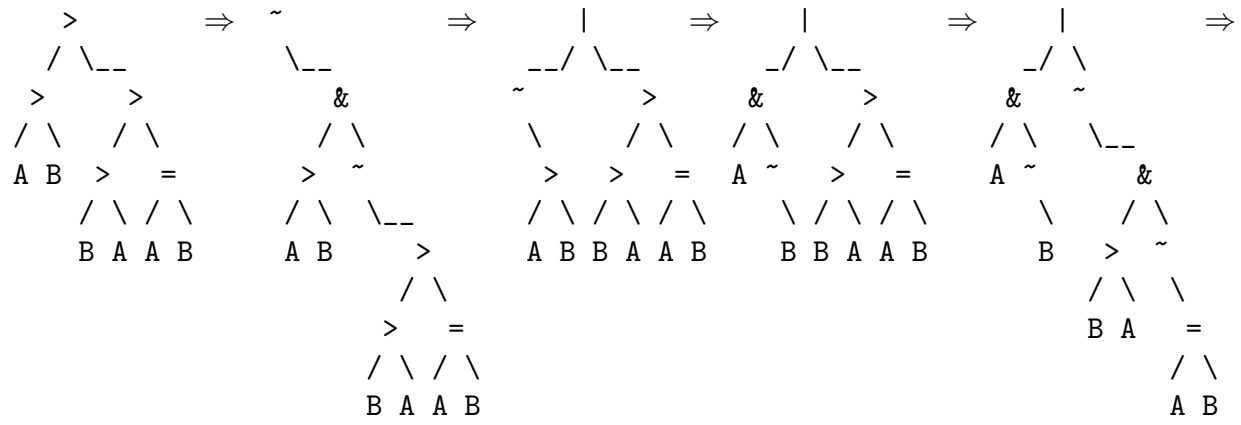
Mějme formuli vyjadřující, že existuje k implikaci i její obrácená implikace, pak lze odvodit i ekvivalenci obou formulí, které jsou argumenty implikace.

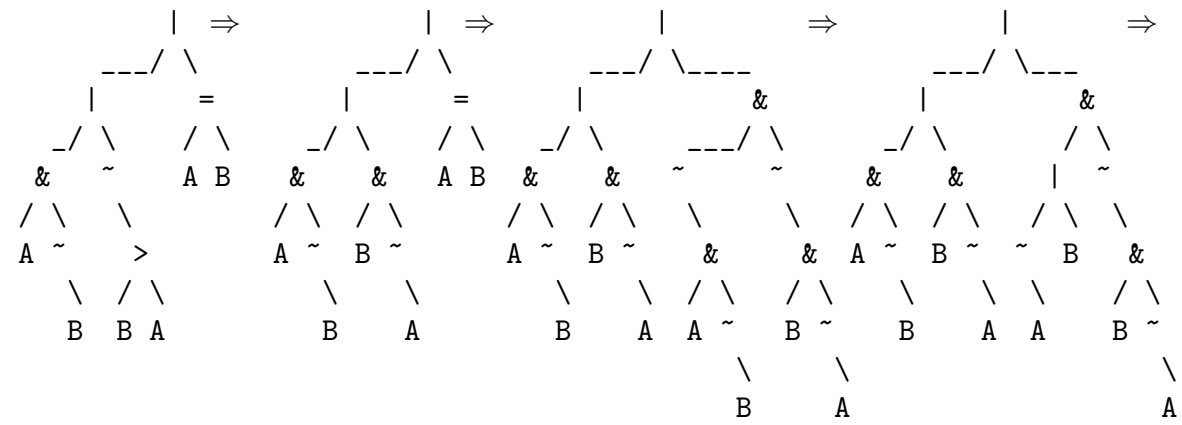
```
(A>B)>((B>A)>(A=B))
```

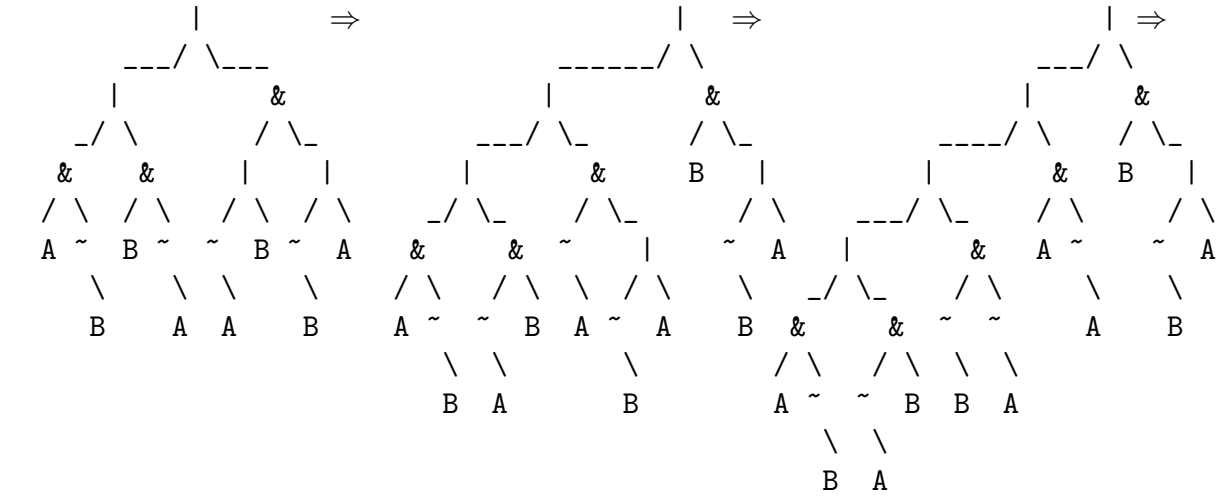
Nejprve se podívejme, které operace musíme udělat, abychom dospěli k DNF. Na této DNF je vidět, za jakých podmínek je platná. Obsahuje 4 disjunkty a můžeme z nich i vyčíst, při jakých hodnotách dosazených za logické proměnné bude výraz pravdivý. Disjunkt 1 říká, že výraz platí, pokud A platí (tedy $A = 1$) a zároveň platí $\neg B$ (tedy $B = 0$), disjunkt 2 - $A = 0$, $B = 1$, disjunkt 3 - $A = 0, B = 0$, disjunkt 4 - $A = 1, B = 1$. V tomto případě jednoduchou úvahou vidíme, že v podstatě to jsou všechny možnosti, jaké hodnoty mohou být A a B dosazeny a tudíž formule je tautologie. Minimalizace nám však toto zajistí algoritmicky.

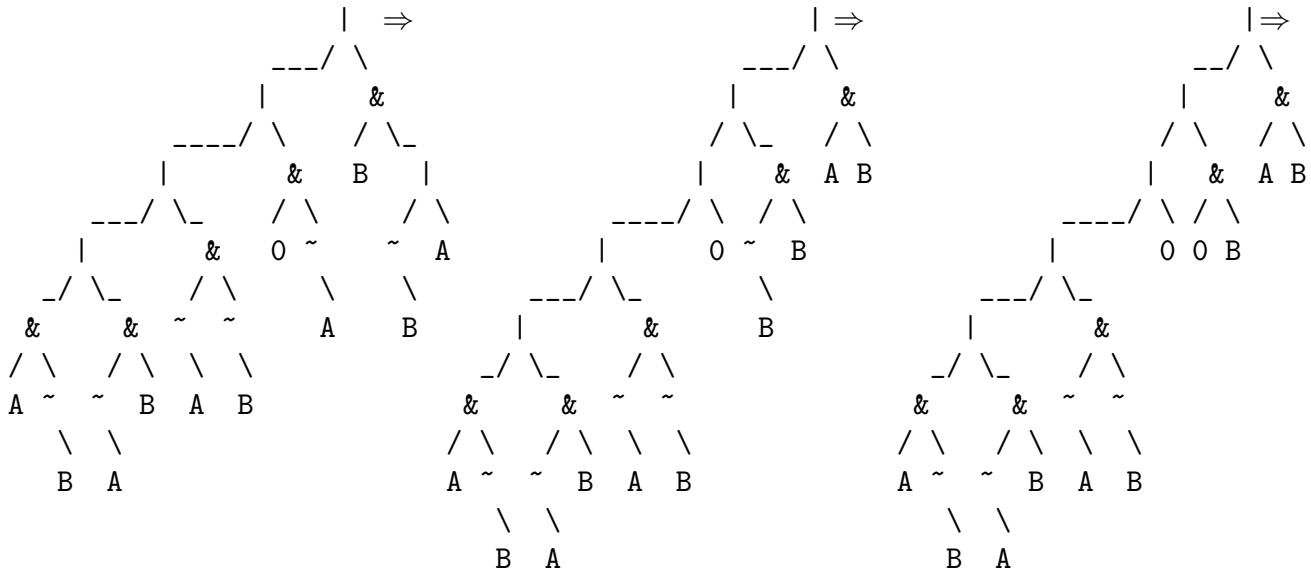
```
(A>B)>((B>A)>(A=B))
Implikace na konjunkci:~((A>B)&~(B>A>(A=B)))
Konjunkce na disjunkci:~(A>B)|(B>A>(A=B))
Implikace na konjunkci:A&~B|(B>A>(A=B))
Implikace na konjunkci:A&~B|~((B>A)&~(A=B))
Konjunkce na disjunkci:A&~B|~(B>A)|(A=B)
Implikace na konjunkci:A&~B|B&~A|(A=B)
Ekvivalence na konjunkci:A&~B|B&~A|~(A&~B)&~(B&~A)
Konjunkce na disjunkci:A&~B|B&~A|(~A|B)&~(B&~A)
Konjunkce na disjunkci:A&~B|B&~A|(~A|B)&(~B|A)
Distribuce:A&~B|~A&B|~A&(~B|A)|B&(~B|A)
Distribuce:A&~B|~A&B|~B&~A|A&~A|B&(~B|A)
Komplementarita:A&~B|~A&B|~A&~B|0&~A|B&(~B|A)
Distribuce:A&~B|~A&B|~A&~B|0|~B&B|A&B
Komplementarita:A&~B|~A&B|~A&~B|0|0&B|A&B

DNF: A&~B|~A&B|~A&~B|A&B
```

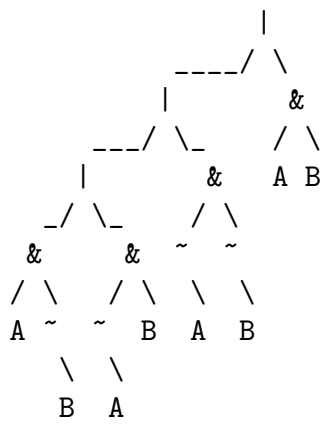
Nebo výstižněji pomocí stromu.











Nyní můžeme zkusit aplikovat pravidla pro minimalizaci formule a dostaneme logickou konstantu 1. Tudíž formule je opravdu logicky platná (tautologie - zákon). Takovou podmínku bychom bez problému mohli zcela vynechat a považovat ji za vždy splněnou.

```
A&~B|~A&B|~A&~B|A&B

Rozšíření:0&~B|~A&B|1&~B|A&B
Rozšíření:0&~B|0&B|1&~B|1&B
Rozšíření:0|1
Minimální DNF: 1
```

Příklad:

Na dalším příkladě můžeme vidět, jak drasticky se zjednoduší pomocí minimalizace formule - tedy ušetříme mnoho potenciálních vyhodnocení. Mějme následující formuli:

```
(~p>(q&~r))>(~q|r)
```

Nejprve ji po krocích převedeme na DNF.

```
Implikace na konjunkci:~((~p>q&~r)&~(~q|r))
Konjunkce na disjunkci:~(~p>q&~r)|~q|r
Implikace na konjunkci:~p&~(q&~r)|~q|r
Konjunkce na disjunkci:~p&(~q|r)|~q|r
Distribuce:~q&~p|r&~p|~q|r
DNF: ~p&~q|~p&r|~q|r
```

Minimalizací dostaneme překvapivě jednoduchou formuli a to pouze aplikací dvou absorpcí.

```
Absorpce:0&~q|~p&r|~q|r
Absorpce:0&~q|0&r|~q|r
Minimální DNF: ~q|r
```

Posledním použitím, které sice úzce nesouvisí s cílem našeho článku, je možnost minimalizací DNF zjišťovat, zda závěr vyplývá z daných předpokladů (již bylo probíráno v článku [10]). Ukažme si to pouze na příkladu.

Příklad:

Nechť jsou daná tři tvrzení - předpoklady:
1. Jan je učitel.
2. Neplatí, že Jan je učitel a zároveň je bohatý.
3. Je-li Jan rockový zpěvák, pak je bohatý.
Chtěli bychom prověřit závěr - Z. Jan není rockový zpěvák.
Nejprve musíme zvolit logické proměnné.
u - Jan je učitel.
b - Jan je bohatý.
z - Jan je rockový zpěvák.
Nyní musíme sestrojit výrazy pomocí spojek pro tvrzení 1., 2., 3. a spojit je všechny konjunkcí (platí současně), tuto konjunkci pak dáme jako první podvýraz implikace a druhým bude závěr Z. (Implikace vyjadřuje, že závěr vyplývá z předpokladu). Formuli následně převedeme na minimální DNF a vyjde-li 1, pak se jedná o správnou dedukci (systém Bachelor toto přímo umožňuje).

```
u&~(u&b)&(z>b)>~z

Implikace na konjunkci:~(u&~(u&b)&(z>b)&z)
Konjunkce na disjunkci:~(u&~(u&b)&(z>b))|~z
Konjunkce na disjunkci:~(u&~(u&b))|~(z>b)|~z
Konjunkce na disjunkci:~u|u&b|~(z>b)|~z
Implikace na konjunkci:~u|u&b|z&~b|~z
Absorpce negace:~u|b&1|~b&z|~z
Absorpce negace:~u|b&1|~b&1|~z
Rozšíření:~u|0|1|~z
Výraz je platný. {Dedukce je správná}
```

Druhý zajímavý příklad umožňuje pomocí DNF zjistit, kdy dává sada výrazů smysl - v tomto případě tak chceme odhalit viníka trestného činu (předpokládáme, že mluví pravdu).

Příklad:

Brown, Jones a Smith jsou podezřelí z podvodu. Svědčili pod přísahou takto:
1. Brown: Jones je vinen a Smith je nevinen.
2. Jones: Je-li vinen Brown, pak je vinen i Smith.
3. Smith: Já jsem nevinen, ale alespoň jeden ze zbývajících obviněných je vinen.
Nejprve musíme zvolit logické proměnné.
u - Brown je vinen.
b - Jones je vinen.
z - Smith je vinen.
Nyní musíme sestrojit výrazy pomocí spojek pro tvrzení 1., 2., 3. a spojit je všechny

konjunkcí (platí současně). Výraz následně převedeme na minimální DNF a disjunkty nám ukáží informace, kdy je konjunkce tvrzení pravdivá - dává smysl a také z nich přímo vyčteme, jaká musí vina a nevina jednotlivých obviněných.

```
(J&~S)&(B>S)&(~S&(J|B))
Implikace na konjunkci:J&~S&~(B&~S)&~S&(J|B)
Konjunkce na disjunkci:J&~S&(~B|S)&~S&(J|B)
Distribuce:~B&J&~S&~S&(J|B)|S&J&~S&~S&(J|B)
Distribuce:J&J&~S&~S&~B|B&J&~S&~S&~B|S&J&~S&~S&(J|B)
Idempotence:~B&1&J&~S&~S|B&J&~S&~S&~B|S&J&~S&~S&(J|B)
Idempotence:~B&1&J&1&~S|B&J&~S&~S&~B|S&J&~S&~S&(J|B)
Komplementarita:~B&1&1&J&~S|0&~B&J&~S&~S|S&J&~S&~S&(J|B)
Idempotence:~B&1&1&J&~S|0&~B&J&1&~S|S&J&~S&~S&(J|B)
Distribuce:~B&J&~S|0|J&J&~S&~S&S|B&J&~S&~S&S
Idempotence:~B&J&~S|0|1&J&~S&~S&S|B&J&~S&~S&S
Idempotence:~B&J&~S|0|1&J&1&~S&S|B&J&~S&~S&S
Komplementarita:~B&J&~S|0|1&J&1&0&S|B&J&~S&~S&S
Idempotence:~B&J&~S|0|1&0&1&J&S|B&J&1&~S&S
Komplementarita:~B&J&~S|0|1&0&1&J&S|B&J&1&0&S
Formule je konzistentní.Její modely vyjadřuje DNF:
~B&J&~S {Tedy vinen je Jones a ostatní jsou nevinní.}
```

6 Překladače všeho druhu a závěr

Na poměrně rozsáhlém prostoru jsme si ukázali, jak lze zapisovat syntaxi jazyka (výrazů), analyzovat je, překládat do jiných forem a také vyhodnocovat. Vše jsme doplnili také hotovou aplikací a ukázkou kódů (jednak pro jednoduchý příklad zcela funkční a jednak fragmenty z kódu, který je součástí odkazovaného balíčku). Chtěli jsme ukázat, že zdánlivě složité úkoly lze dělat přehledně, ilustrativně a hlavně efektivně z hlediska časové a prostorové složitosti. Náš článek má vlastně dvě roviny - jednak jsme ukázali, že teorie formálních jazyků a automatů je nám blíž než bychom čekali. Úloha sestrojit jednoduchý analyzátor nebo překladač bezkontextového jazyka může potkat každého informatika - vždyť strukturované vstupy jsou součástí mnoha informačních systémů včetně různých databázových aplikací. Lze to hezky ilustrovat na příkladu ze života. Jeden ze studentů autorů, který rozhodně nebyl nadšenec do teoretické informatiky nás asi rok po státnicích potkal a uvedl, že nejdůležitější znalostí, kterou využil z naší školy v softwarové firmě, nebyla žádná moderní technologie, ale znalost tvorby analyzátorů bezkontextových jazyků! Jeho úkolem bylo přijímat na vstupu jistý strukturovaný vstup informací zaslaných podle síťového protokolu a překládat jej do jiného formátu. Metoda rekurzivního sestupu není

sice tou nejefektivnější (spíše se používají přímo automatizované nástroje na tvorbu kódu analyzátoru), ale má výhodu v přehlednosti, jednoduchosti, kód lze lehce upravit (pokud bychom chtěli třeba místo číslic používat v aritmetických výrazech nějakou složitější strukturu, není problém změnit pouze tuto malou část gramatiky a kódu a zbytek se nezmění) a hlavně si jej programujete zcela sami - tudíž máte nad kódem plnou kontrolu. Také bylo užitečné si připomenout práci se stromy a jejich výhody a různé způsoby procházení. Druhá rovina článku je pak zajímavá z hlediska matematické logiky, naučili jsme se výrazně zjednodušovat logické výrazy a navíc jsme viděli, že to není nijak algoritmicky složité. Doufáme, že tento exkurz do světa překladačů je pro čtenáře poučný nejen teoreticky, ale i programátorsky. Tvorba složitějších překladačů samozřejmě obsahuje další důležité prvky, které zde nezmiňujeme (např. tabulky symbolů), ale vcelku netěžší je právě získat vhodnou reprezentaci hierarchicky strukturovaných kódů a stromy jsou k tomuto účelu vynikající.

Reference

[1] ČEŠKA, M., RÁBOVÁ, Z. Gramatiky a jazyky. VUT Brno, 1992
http://www.fit.vutbr.cz/study/courses/TI1/public/gj-1.3.pdf (2003).

[2] DVOŘÁK, S. Dekompozice a rekurzivní algoritmy. Grada Praha, 1992.

[3] HOPCROFT, J.E., ULLMAN, J. D. Introduction to Automata theory, Languages and Computation. Addison-Wesley, Reading (Mass.), 1979.

[4] HABIBALLA, H. Formální úpravy ve výrokové logice (aplikace Bachelor). Bakalářská práce : Ostravská Univerzita, PřF. 1997.
http://www1.osu.cz/home/habibal/publ/bachelor.zip

[5] HABIBALLA, H. Regulární a bezkontextové jazyky I. Ostravská univerzita, Ostrava, 2003.
http://www1.osu.cz/home/habibal/kurzy/xrab1.pdf

[6] HABIBALLA, H. Regulární a bezkontextové jazyky II. Ostravská univerzita, Ostrava, 2005.
http://www1.osu.cz/home/habibal/kurzy/rabj2.pdf

[7] HABIBALLA, H. Překladače. Ostravská univerzita, Ostrava, 2006.
http://www1.osu.cz/home/habibal/kurzy/xprek.pdf

[8] HABIBALLA, H. Formální jazyky a automaty. In Matematika-fyzika-informatika. 05-06/2004, roč.13.

[9] HABIBALLA, H.; KMEŤ, T. Vyčíslitelnost a složitost. In Matematika-fyzika-informatika. 02-03/2005-06(15).

[10] HABIBALLA, H.; PAVLISKOVÁ, L.; KMEŤ, T. Logika. In Matematika-fyzika-informatika. 07-09/2005-06(15).

[11] CHYTIL, M. Automaty a gramatiky. Matematický seminář SNTL Praha, 1984.

[12] JANČAR, P. Teorie jazyků a automatů. VŠB TU Ostrava, http://www.cs.vsb.cz/jancar/ (2003).

[13] JINOCH, J., MULLER, K., VOGEL, J. Programování v jazyce Pascal. SNTL 1988, Praha.

[14] LUKASOVÁ, A. Formální logika v umělé inteligenci. Computer Press, Brno, 2003.

Kontaktní adresa:

Hashim Habiballa
katedra informatiky a počítačů
Přírodovědecká fakulta Ostravské Univerzity
30.dubna 22, 701 03 Ostrava 1,
tel.: +420596160241,
e-mail: habiballa@volny.cz, www: http://www.volny.cz/habiballa/