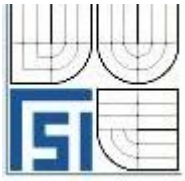
V Y S O K É U Č E N Í T E C H N I C K É V B R N Ě

Matematické základy informatiky



B R N O 2 0 0 6

Obsah :

1. Úvod, opakování základů Delphi

Struktura programu:

- hlavička(automaticky vytvářena Delphi, začíná slovem program, za kterým je uvedeno jméno programu)
- připojení programových hlaviček(klauzule uses)
- definice globálních konstant(vyhrazené slovo const)
- definice typů(slouží pro definice vlastních datových typů)
- deklarace proměnných(vyhrazené slovo var)
- deklarace procedur a funkcí
- vlastní program(seznam příkazů, které má program vykonat, začíná slovem begin a končí end)

Celý program zakončen tečkou.

Členění programu
Pro rozdělení rozsáhlých textů programů se používají tzv. UNITY(programové jednotky)

Oddělování příkazů
Příkaz nekončí dříve dokud nenapíšete středník, proto můžete psát například

```
...
prom:=10; pomocna:=5;
...
```

což znamená to samé jako

```
...
prom:=10;
pomocna:=5;
…
```

Komentáře
Existuji tři základní druhy :
a) { vlastní komentář }
b) (* stejný případ, ale použitím hvězdičky a závorky *)
c) // lomítka ohraničují komentář, platí pouze pro jeden řádek

Přetypovaní
Pascal je přísně typový jazyk, čili hlídá operace mezi jednotlivými proměnnými
Příklad:

```
var
  a:byte;
  c:char;

begin
  c:='AHOJ' ;
  a:=c; { hrubá chyba, překladač zahlásí chybu }
     kdežto
  a:=byte(c); { správný způsob přetypování }
end;
```

Některé typy nemusíme přetypovávat vůbec(Delphi to udělají za nás)

```
var
  i:integer;
  r:real;

begin
  i:=10;
  r:=i;  // tzv. Implicitní přetypování v proměnné r bude hodnota 2.000
  i:=r;  // chyba, nelze provést tento druh přetypování
  i:=integer(r); // také nelze provést, protože je rozdílná velikost dat. Typu Real a Integer
end;
```

Speciálním případem je univerzální datový typ Variant(zabírá místo v paměti a zvyšuje dobu zpracovaní kódu)

Řídící struktury

Podmíněné příkazy

If *podmínka* then *příkaz, který se provede pokud je podmínka splněna*
else *příkaz pokud podmínka není splněna*

Příklad:

```
if a<b then  ajemensi :=true
       else   ajemensi:=false;
```

Příkaz case(přepínač)

```
...
case promenna of
podminka1: příkazy;
podminka2: příkazy;
else
   kód , který bude proveden, pokud nevyhovuje žádná z podmínek
end;
```

Cykly

s pevným počtem opakování
s podmínkou na počátku
s podmínkou na konci

Cyklus for

nejjednodušší s pevným počtem opakováním
použití kde je zřejmý počet cyklů

Příklad:

```
for i:=0 to 5 do a:=a+1; // tento cyklus proběhne 6 krát
```

opakem je použití downto kde se hodnota čítače bude snižovat o jedničku v každém kroku

Cyklus while … do

použití v případech kdy není dopředu znám počet opakování

Příklad:

```
var   i,x:integer;
```

```
begin
...
i:=1;              // nastavení čítače na počáteční hodnotu
x:=0;              // vynulování proměnné
while i <=5 do          //cyklus bude probíhat tak dlouho dokud bude i menší nebo rovno 5
  begin
    x:=x+1;
    i:=i+1;        // zvýšení i o jedničku
end;
```

Cyklus repeat … until
první iterace proběhne vždy
Příklad:

```
var  i,x:integer;
…
x:=0;
i:=1;
repeat
  x:=x+i;   // pozor, nepíše se zde begin a end
  i:=i+1;
until i>5;
…
```

Procedury a funkce
struktura jazyka Delphi je založena na podprogramech, vychází z myšlenky rozdělit celý program na několik menších problémů
hlavní program pak jen volá ve vhodném pořadí jednotlivé podprogramy
v Delphi známe dva druhy podprogramů procedury a funkce

Příklad:

```
...
Procedure zprava;
begin
  showmessage('Ahoj svete');
end;


function dvakrat(hodnota:integer):integer;
begin
  dvakrat:=dvakrat*5;
end;
```

nebo můžeme použít tento zápis :

```
function dvakrat(hodnota:integer):integer;
begin
  result:=hodnota*5;
end;
```

Oba zápisy jsou shodné a lze je zaměnit. Pro použití v programu je musíme zavolat :

```
…
zprava;          // výsledkem bude dialogové okno s nápisem
x:=dvakrat(3);// do x se přiřadí výsledek procedury dvakrat  (15)
...
```

procedury a funkce se definují na začátku programu
musí být nadeklarovány dříve, než se poprvé zavolají

Procedury s parametry
pro zvýšení univerzálnosti se používají parametry, čili proměnné uvedené v definici procedury

Příklad:

```
procedure zprava(tisknitoto:string);
begin
  showmessage(tisknitoto);
end;
```

zavoláme například takto:

```
…
zprava('Dobry den');
zprava('ctenari této edice');
…
```

Parametry mohou být předávané odkazem nebo hodnotou.

Parametr předávaný odkazem :

```
procedure zprava( var tisknitoto:string );      // klíčové slovo var je zde nesmírně důležité
```

Parametr předávaný odkazem :

```
procedure zprava(tisknitoto:string);  // příklad uveden na další straně
```

Příklad:

```
procedure zprava(tisknitoto:string);
begin
  showmessage(tisknitoto);
  tisknitoto:='procedura zprava probehla uspesne';
end;
```

pokud ovšem zavoláme proceduru předávanou hodnotou takto dostaneme tyto dvakrát stejné výsledky

```
var parametr:string;
…
parametr:='Prave bezi procedura zprava';
zprava(parametr);
zprava(parametr);       // vypíše znovu stejnou zprávu
…
```

Musíme tedy použít klíčové slovo var v deklaraci procedury zprava viz. Výše.

Procedury s konstantními parametry
v deklaraci procedury musíme uvést klíčové slovo const

Příklad:
```
procedure konstatniparam(const param:integer);
…
param:=10;     // nelze! Protože param je konstantní parametr
…
```

Dopředná deklarace
Jazyk pascal má pravidlo, kde platí že každá procedura než se poprvé použije musí být poprvé nadeklarována

Lokální proměnné
deklarace stejná pomocí klíčového slova var hned za hlavičkou procedury

```
procedure spocti(param:integer):
var i:integer;   // i je lokální proměnná a lze ji použít pouze v proceduře dva
                 // pozor , nutno pamatovat na počáteční nastavení těchto hodnot, je totiž náhodné
```

Přetížené procedury
jedná se vlastně o použití procedur se stejným názvem které se od sebe liší počtem parametrů nebo typem
překladač sám určí která z procedur se má použít
klíčové slovo je zde overload

Příklad:

```
procedure zprava(stextem:string); overload;
begin
  showmessage(stextem);
end;

procedure zprava(scislem:integer); overload;
begin
  showmessage(inttostr(scislem));
end;

procedure zprava(stextem:string; realne:real ); overload;
begin
  showmessage(stextem + floattostr(realne));
end;
```

volání procedur bude vypadat takto :
```
zprava('Funguje to! ');
zprava(50);
zprava('Vysledek je ',3.14);
```

Přetěžované funkce :
klíčové slovo overload v deklaraci funkce
např. takto :
...
function ukaz(cislo:integer):integer;overload;
function ukaz(cislo:string):integer;overload;
…

zbytek programu by mohl vypadat takto :
...
procedure TForm1.Button1Click(Sender: TObject);
begin
ukaz(15);
ukaz('152.2');
end;

function TForm1.ukaz(cislo: integer): integer; // provede se výpis cisla 15 na obrazovku
begin
showmessage(inttostr(cislo));
end;

function TForm1.ukaz(cislo: string): integer; //// provede se výpis cisla 152.2 na obrazovku
begin
showmessage(cislo);
end;
...

Pole
dělení:
-jednorozměrné nebo vícerozměrné
-statické a dynamické
lze si je představit jako souvislý blok paměti, který je rozdělen na menší části do nichž se ukládají hodnoty
výhodou polí v pascalu je, že indexy pole mohou začít v libovolném rozsahu a ne nutně od 0

Příklad:
type
dny=array [1..7] of string;

var den:dny;
begin
den[1]:='pondeli';
…
den[7]:='nedele';
end;

Jediná operace , kterou je možné provádět s proměnnou typu pole, je přiřazování:
kopie:=den; // tento příkaz zkopíruje najednou celý obsah pole den do pole kopie

Pro vynulování pole můžeme použít tuto konstrukci:
for i:=1 to 7 do den[i]:='';

Vícerozměrné pole
Příklad
```
type
  dvourozmerne = array [1..10,1..20] of integer;
var
  i,j:integer;
  pole:dvourozmerne;
…
for i:=1 to 10 do
  for j:=1 to 20 do
     pole[i,j]:=0;          // vynulování dvourozměrného pole
…
```

Podobně se deklarují vícerozměrné pole.

Dynamické pole
-indexy dynamického pole vždy začínají od 0
-velikost pole, nejmenší a nejvyšší index se zajišťuje pomocí funkcí sizeof, low, high
low vrací 0, high vrací velikost pole-1

Příklad:
```
type
  dynamickepole = array of byte;


var
  a,b:dynamickepole;
  i:integer;
begin
 setlength(a,10);
for i:=0 to 9 do
  a[i]:=5;        // pole bude vyplněno pouze pětkami
  b:=a;           // do b se uloží odkaz na pole a!
  B[1]:=1;        // na pozici b[1] se uloží hodnota 1, ale tímto se změní i hodnota a[1]
  …
end;
```

<u>Programové jednotky (UNITY)</u>
členění programu do více částí tzv. programových jednotek(unit)
zvýšení čitelnosti programu
zvyšujeme univerzálnost procedur a funkcí
obsah jednotky je skryt před okolním světem, ostatní programy používají jen ty procedury a funkce , které sami povolíme

Struktura jednotky
hlavička-klíčové slovo unit
rozhraní (interface) jsou zde uvedeny hlavičky procedur a funkcí, deklarace proměnných a typů
vlastní kód uvozeno slovem implementation
inicializační část-initialization

úklidová část
konec slovem end.

Práce s jednotkami
pokud chceme ve svém programu použít nějakou jednotku
uses jednotka1, jednotka2,...

Kruhová reference
problém při nevhodné deklaraci a volání procedur kdy se používají dvě jednotky
lze použít ikdyž to není vhodné
příklad:
unit druha;

interface

procedure proced(spocti:integer);

implementation

uses první;

procedure proced(spocti:integer);

Základy objektově orientovaného programování (OOP)
-základem každého OOP jsou objekty a třídy (class)
-jedná se o datový typ třída, který se deklaruje pomocí klíčového slova class
-uvnitř třídy jsou definovány jednotlivé objekty (objects)
-objektem mohou být proměnné, procedury a funkce
-proměnné uvnitř třídy se nazývají atributy
-procedury a funkce, které jsou objekty třídy se souhrnně nazývají metody
-definice nové třídy :

```
type Tnovatrida=class
        sirka:integer;
        vyska:integer;
        procedure nastavsirku(x:integer);
        procedure nastavvysku(y:integer);
    end;
```

Nepsaným pravidlem je použití v definici nové třídy písmeno T.
Pro použití třídy je nutné vytvořit proměnnou typu třída
var mojetrida:Tnovatrida;

Proměnné typu třída se nazývají instance třídy. K objektům se přistupuje pomocí tečky:

```
…
mojetrida.vyska:=5;
mojetrida.nastavsirku(15);
…
```

Balíky metod pro práci s třídami:
1. Zapouzdření
2. Dědičnost
3. Polymorfizmus

Zapouzdření
důležitá vlastnost třídy, která říká že všechny objekty jsou uzavřeny uvnitř třídy a pro každou instanci jsou na sobě zcela nezávislé
private, public, protected,published:
slouží k ochraně objektů před neoprávněným přístupem, protože je nepřípustné, aby s objekty manipuloval někdo jiný než vlastní metody třídy
public-veřejné objekty
private-soukromé použití pouze v rámci jednotky
published-podobné chovaní jako veřejné objekty, navíc obsahují informace o běhu programu
portected-objekt bude neviditelný všem objektům mimo vlastní třídu a třídu potomků , ale není omezen pouze na jednu jednotku

Dědičnost
-je vlastně odvozování nových tříd od stávajících tak, aby všechny objekty předchozí třídy zůstaly zachoványa pouze se rozšířili o nové.

Prříklad:
```
type
Tbod = class(Tobject)rodič ale zároveň potomek třídy Tobject( nejvyšší ) ze které dědí vlastnosti a metody
        x:integer;
end;

Tkruznice =class(Tbod)        // potomek
….
end;
```

Delphi umožnuje i vicenásobnou dědičnost, kdy je možné vytvořit potomka dalšího potomka z již děděné třídy
Výsledná třída pak obsahuje metody i atributy všech svých předchůdců.

<u>Pozdní vazba, virtuální a statické metody(POLYMORFISMUS)</u>
pozdní vazba používá se pro virtuální metody, pokud jsme měli třídu, která obsahovala statickou metodu, a vytvořili jsme potomka této třídy, mohli jsme v této nové třídě zděděnou metodu upravit a tím jsme odstranili metodu původní. Pokud nadefinujeme ve třídě předka metodu jako virtuální , můžeme ji ve třídě potomka předefinovat. Použijeme li metodu jako virtuální , říkáme tím že neví dopředu jasné , zda se má volat metoda předka nebo potomka.To se vyhodnotí až za běhu programu, tento přístup se nazývá polymorfizmus.

Příklad:
```
Tbod = class(Tobject)
…
procedure Nakresli:virtual; //  virtual klíčové slovo v definici rodiče
end;
```

```
Tkruznice = class(Tbod)
…
procedure nakresli:override; //  override  klíčové slovo v definici rodiče
end;
```

Abstraktní metody

klíčové slovo abstract
metoda která předpokládá, že ji budou používat až její potomci

Příklad:

```
type
Tbod = class(Tobject)
…
function obvod:real;virtual;abstract;  // tato metoda nebude nikdy použita protože počítáni obvodu
bodu
end;                                          // je nesmysl

Tkruznice = class(Tbod)        //potomek třídy Tbod
…
function obvod:real;override;
end;
```

2. Koncept výjimek a chráněných bloků

výjimky:
-nastávají při nesprávné funkci programu, jeho podprogramů, obvykle způsobeny chybou programátora, či uživatele

Příklad:

```
procedure Tform1.button1click(Sender:Tobject);
var a,b,vysledek:integer;
begin
  a:=10;
  b:=0;
  vysledek:=a div b; // tento řádek vyvolá výjimku, dělit nulou nelze
end;
```

-každá výjimka je potomkem třída výjimek exception, existují krom univerzálních tříd také odvozené, v jednotce sysutils lze nalézt přehled výjimek a chyby, které je způsobují

obsluha výjimek

a) try ... except

```
try
  // zde se nachází potencionálně nebezpečný kód, který může vyvolat výjimku
except
  // kód , který se provede, pouze když je vyvolána výjimka
end;
```

b) try ... finally

```
try
  // zde se nachází potencionálně nebezpečný kód, který může vyvolat výjimku
finally
  // kód , který se provede vždy, i když výjimka byla nebo nebyla vyvolána při průchodu blokem try
end;
```

Příklad:
```
try
  a:=5 div 0; //zde dojde k vyvolání výjimky
showmessage('Tento výpis na obrazovku se neprovede nikdy');
finally
  showmessage('Tento výpis na obrazovku se provede vždy');
end;
```
možnost použiti vnořených výjimek do sebe
opětovné vyvolání výjimky lze provést pomocí příkazu *raise*

vlastní výjimky:
můžeme si je nadefinovat sami, jsou potomkem třídy Exception
příklad:
```
type
mojevyjimka=class(exception)
end;

procedure velikost(cas:integer;max:integer);
begin
  if cas>=max then
          raise mojevyjimka.create('Prilis velka hodnota');
end;
...
velikost(5,1);  //zavoláme proceduru testující velikost, dojde k vyvolání výjimky
…
```
tichá výjimka
```
raise Eabort.create('Toto je tichá výjimka');
```
text, který je v parametru konstruktoru se nikdy neobjeví
chráněné bloky:
uvozeny typicky pomocí if ... then … else
příklad:
```
if podmínka splněna then
begin
...
// chráněná část bloku, provede se pouze při platnosti podminky
...
end;
```

Vyhledávání: přehled algoritmů, jejich princip, analýza a hodnocení
Dělení metod vyhledávání:
· statické - pracují nad datovou strukturou, která se v průběhu zpracování nemění
· dynamické - předpokládá se, že v datové struktuře mohou v průběhu zpracování vznikat nové a zanikat nepotřebné položky

Sekvenční vyhledávání
Sekvenční vyhledávání v seznamu

Prochází se seznam od prvního prvku a porovnává se s vyhledávaným klíčem. Pokud po průchodu je stále aktivní nějaký prvek (ukazatel na něj), byl klíč nalezen.

Časová analýza:
Čas při neúspěšném vyhledání: $T_f = cn$ (c je konstanta vyjadřující délku jednoho průchodu cyklu)
Čas při úspěšném vyhledání i-té položky: $T_{ti} = ci$
Průměrný čas pro úspěšné vyhledání:

$$T_t = \frac{n+1}{2} c$$

Nejrychleji budou vyhledány ty položky, které jsou zařazeny na začátek seznamu. Je-li známa pravděpodobnost vyhledávání jednotlivých položek, je účelné, aby byly položky seřazeny sestupně podle pravděpodobnosti svého vyhledávání.

Sekvenční vyhledávání v poli
Procházejí se položky od začátku a u každého prvku probíhá kontrola zda již není zpracováván poslední aktivní prvek a zda právě zpracovávaný prvek není tím hledaným.
Jednoduchou úpravou lze tento algoritmus zrychlit. Zrychlení spočívá v odstranění testování na poslední prvek při každém průchodu cyklu. Pokud vložíme před vyhledáváním zarážku (prvek s právě hledaným klíčem) na konec pole, bude prvek vždy nalezen a stačí jen zkontrolovat index - zda se jedná o zarážku nebo pole prvek skutečně obsahuje.
Časová analýza*:*
Úpravou dojde pouze ke snížení konstanty c (režie po každém průchodu cyklu).

Sekvenční vyhledávání v seřazeném seznamu
Je-li nad typem klíče definována *relace uspořádání*, lze seznam seřadit podle velikosti klíče. Sekvenční vyhledávání v seřazeném seznamu se pak zrychlí v případě neúspěšného vyhledávání, protože vyhledávání lze ukončit, když je klíč testované položky větší, než vyhledávaný klíč.

Vyhledávání v seřazeném poli se liší nepatrně. I při vyhledávání v seřazeném poli lze využít varianty se zarážkou, ale hodnota zarážky musí být větší než hodnoty všech možných vyhledávacích klíčů.

Nesekvenční vyhledávání v seřazeném poli

Binární vyhledávání
Vyhledávaný klíč se porovná s klíčem položky, která je umístěna v polovině prohledávaného pole. Dojde-li ke shodě, končí vyhledávání úspěšně. Je-li vyhledávaný klíč menší, postupuje se porovnáváním prostředního prvku v levé polovině původního pole, je-li větší v pravé polovině původního pole. Vyhledávání končí neúspěšně v případě, že prohledávaná část pole je prázdná.

Dijkstrova varianta binárního vyhledávání vychází z předpokladu, že pole může obsahovat více položek, jejichž klíče se navzájem rovnají. V případě úspěšného vyhledávání se nalezne nejlevější položka ze skupiny položek se stejnými klíči.

Časová analýza:
Zatímco u prvního algoritmu může úspěšné vyhledávání trvat kratší dobu než neúspěšné, Dijkstrova varianta má úspěšné i neúspěšné vyhledání stejně dlouhé.

Stromová reprezentace binárního vyhledávání
Mechanismus binárního vyhledávání lze znázornit jeho reprezentací binárním rozhodovacím stromem. Každý uzel je reprezentován trojicí čísel: indexem i a hranicemi části pole l a r. Jednotlivé cesty od kořene k listu představují postup polem při vyhledávání.

Uniformní binární vyhledávání
Místo tří proměnných i, l, a r lze použít pouze dvou: aktuální index i a odchylka m od aktuálního indexu i. Po každém neúspěšném porovnání ustavíme:

```
i:=i+m
m:=m div 2
```

Fibonacciho vyhledávání
Algoritmus pracuje podobně jako binární vyhledávání, ale daný interval v poli se nedělí na dvě poloviny, ale dělící bod se odvozuje z Fibonacciho posloupnosti a k jeho získání stačí *aditivní operace*, což zvýší rychlost tam, kde aditivní operace jsou výrazně rychlejší než celočíselné dělení číslem 2.

Binární vyhledávací stromy
Binární vyhledávací strom (BVS) je takový binární uspořádaný strom, pro jehož každý uzel platí, že jeho levý podstrom je buď prázdný, nebo sestává z uzlů, hodnoty jejichž klíčů jsou menší, než hodnota klíče daného uzlu a podobně jeho pravý podstrom je buď prázdný, nebo sestává z uzlů, hodnoty jejichž klíčů jsou větší, než hodnota klíče daného uzlu.
Průchodem typu INORDER binárním vyhledávacím stromem získáme seřazený lineární seznam.

Vyvážené binární stromy
Délka vyhledávání ve stromové struktuře záleží na uspořádání stromu. Nejhorší případ neúspěšného vyhledávání je dán nejdelší cestou od kořene k listu stromu. Ideálně uspořádaný strom má délky všech cest od kořene k listům stejně dlouhé.
Dokonale vyvážený binární strom je strom, pro jehož každý uzel platí, že *počet uzlů v jeho levém a pravém podstromu se liší maximálně o 1*. Tento stav je v dynamicky vytvářených stromech velice obtížné zachovat.

AVL stromy (výškově vyvážené stromy)
Výškově vyvážený strom, pro jehož každý uzel platí, že výška obou jeho podstromů se liší maximálně o 1.
Tvůrci AVL stromu dokázali, že AVL-strom není vyšší o více než o 45% než odpovídající dokonale vyvážený strom.

Tabulky s rozptýlenými položkami

Tabulky s přímým přístupem
Je-li známa množina všech klíčů, které se budou vkládat do vyhledávací tabulky, a je-li možné nalézt
jedno-jednoznačnou mapovací funkci pro všechny klíče, je možné vytvořit *tabulku s přímým přístupem*. Tuto tabulku tvoří pole, v němž položka s klíčem K_i bude uložena na indexu i daného pole.
Obtíž s využitím jinak vysoce účinné tabulky s přímým přístupem spočívá v obtížném nalezení vhodné mapovací funkce f. V praxi se tato potíž někdy obchází používáním numerických klíčů pro identifikaci položek. V řadě případů je to však neúčinné, když je nutné pracovat s textovým tvarem klíče. Typickým příkladem je manipulace překladače s identifikátory.

Princip tabulek s rozptýlenými položkami
Princip vyhledávání v tabulkách s rozptýlenými položkami (TRP) je velmi podobný principu vyhledávání v index-sekvenčním souboru. Pomocí rozptylovací funkce se získá index pole, na nějž se uloží (od něj se vyhledává) položka s daným klíčem. Obsahuje-li tabulka synonyma vzhledem k danému indexu (více různých klíčů mělo shodnou hodnotu rozptylovací), pak na daném indexu začíná *lineární seznam synonym*, v němž se vyhledává položka s daným klíčem. Vyhledávání v TRP

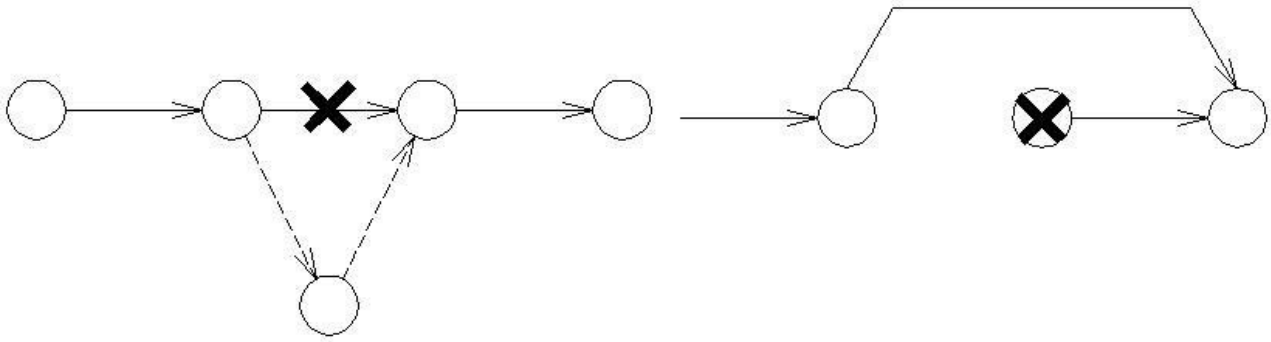
bude účinné tehdy, jestliže počet seznamů bude co největší a jejich délka bude co nejkratší.

3. Seznam, struktury LIFO, FIFO, kolekce, návrhy implementace

Seznam:

seznam je struktura umožnující uchovávat posloupnost prvků téhož typu, jejichž počet může být dynamicky se měnící.
Základní operace nad datovou strukturou:
- vkládaní prvků do seznamu
- mazání prvků ze seznamu

schéma:



uložení prvku mazání prvku

způsoby implementace mohou být:

- jednosměrný (definován následník)
- obousměrný (definován následník i předchůdce)
- lineární (definován první a poslední prvek)
- cyklický

seznamy aplikujeme především jako dynamické proměnné
alokace dynamické proměnné znamená rezervování místa (pomocí new(...)) odpovídající danému typu v určité oblasti paměti-adresa tohoto místa je ukazatelem identifikující tuto proměnnou
jakmile ji nepotřebujeme , uvolníme místo v paměti pro ní rezervovanou pomocí příkazů dispose(...)

způsob aplikace těchto operací aplikujeme jako speciální proměnné seznamů:

Zásobník

Zásobník je homogenní, lineární, dynamická struktura, která zpřístupňuje prvky lineární struktury výhradně na jednom konci, označovaném jako vrchol zásobníku.
Základní vlastností zásobníku je jeho účinek na pořadí lineární struktury do zásobníku vložené. Postupným vyjmutím se pořadí položek této struktury obrátí (invertuje). Proto se zásobníku také říká struktura typu LIFO, což je zkratka anglického názvu ”Last In First Out”.

Mezi základní operace nad abstraktní datový typ (ADT)zásobník patří:
1) Inicializace zásobníku
2) Vložení prvku na vrchol zásobníku
3) Dotaz na prázdnost zásobníku
4) Čtení hodnoty prvku na vrcholu zásobníku. Obsah zásobníku zůstává nezměněn. V případě, že zásobník
neobsahuje žádný prvek (je prázdný), dojde k chybě.
Při praktickém použití v programu je operace Top vždy předcházena testem na neprázdnost zásobníku.
5) Zrušení hodnoty prvku na vrcholu zásobníku. V případě, že zásobník je prázdný, má operace účinek prázdné operace.

Fronta
Fronta je homogenní, lineární, dynamická struktura, která jeden konec lineární struktury zpřístupňuje pro vkládání nového prvku a druhý pro čtení hodnoty prvku a pro rušení prvku.
S ohledem na vlastnost fronty, se frontě také říká struktura typu FIFO, což je zkratka anglického názvu ”First In First Out”.

Mezi základní operace nad ADT fronta patří:
1) Inicializace prázdné fronty
2) Vložení nového prvku na konec fronty
3) Vrácení hodnoty prvku na začátku fronty. Fronta se čtením nemění. Čtení z prázdné fronty způsobuje zásadní chybu.
4) Zrušení prvku na začátku fronty. Pro prázdnou frontu operace prázdná.
5) Dotaz na prázdnost fronty.

Řetězec
Řetězec je homogenní, lineární, dynamická struktura, jejímiž prvky jsou znaky. Řetězec je spolu s textovým souborem základní datovou strukturou pro zpracování textu.

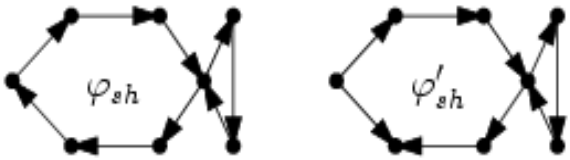
Mezi základní operace patří např.:
lexikografická relace uspořádání dvou řetězců
zjištění hodnoty a změna hodnoty pořadím zadaného znaku v řetězci
vložení a zrušení podřetězce o zadaném začátku a délce
zjištění délky řetězce
spojení dvou a více řetězců do jednoho řetězce

4. Grafy, implementace orientovaného grafu:
Definice:
graf G=(U,H) kde
U je konečná množina prvků n>=0, které se nazývají uzly grafu
H konečná množina neuspořádaných dvojic (u,v) kde u a v jsou dva různé prvky množiny U
prvky množiny H se nazývají hranami grafu
grafy rozlišujeme :
souvislé (možné se dostat z libovolného uzlu do libovolného jiného uzlu)
nesouvislé

orientované grafy:
Často potřebujeme, aby hrany byly pouze jednosměrné. Takovému grafu říkáme *orientovaný graf*. Hrany jsou nyní uspořádané dvojice vrcholů (x, y) a říkáme, že hrana vede z vrcholu x do vrcholu y. Hrany (x,y) a (y,x) jsou tedy dvě různé hrany. Orientovaný graf většinou zobrazujeme jako body spojené šipkami. Většina pojmů, které jsme definovali pro neorientované grafy, dává smysl i pro

grafy orientované, jen si musíme dát pozor na směr hran.

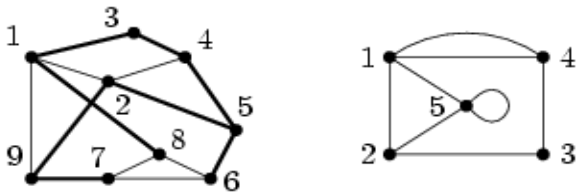
schéma:



Silně a slabě souvislý orientovaný graf

Neorientované grafy:

Jsou určeny množinou vrcholů V a množinou hran E, což jsou neuspořádané dvojice vrcholů. Hrana e = {x, y} spojuje vrcholy x a y. Většinou požadujeme, aby hrany nespojovaly vrchol se sebou samým (takovým hranám říkáme *smyčky*) a aby mezi dvěma vrcholy nevedla více než jedna hrana (pokud toto neplatí, mluvíme o *multigrafech*). Obvykle také předpokládáme, že vrcholů je konečně mnoho. Neorientovaný graf většinou zobrazujeme jako body pospojované čarami.



Neorientovaný graf a jeho kostra; multigraf

Ohodnocené grafy:
Další možností, jak si graf "vyzdobit", je ohodnotit jeho hrany čísly. Například v grafu silniční sítě (vrcholy jsou města, hrany silnice mezi nimi) je zcela přirozené ohodnotit hrany délkami silnic nebo třeba mýtným vybíraným za průjezd silnicí. Přiřazeným číslům se proto často říká *délky* hran nebo jejich *ceny.* Pojmy, které jsme si před chvílí nadefinovali pro obyčejné grafy, můžeme opět snadno rozšířit pro grafy ohodnocené – např. délku sledu budeme namísto počtu hran sledu počítat jako součet jejich ohodnocení. Neohodnocený graf pak odpovídá grafu, v němž mají všechny hrany jednotkovou délku.

5. Prohledávání grafu do šířky, do hloubky, smíšené prohledávání, intuitivní implementace, využití struktur LIFO, FIFO při implementaci

Metody prohledávání:
Vzhledem k velikosti stavového prostoru může být systematické prohledávání stavového prostoru velmi neefektivní (zbytečně se prohledává značná část stavového prostoru, která nevede k cíli). Prohledávání lze omezit znalostí o řešeném problému. Znalosti mají někdy charakter empirický, mohou to být neexaktní poznatky, které jsou často užitečné při řešení, ale často nezaručují, že povedou k řešení
(**heuristické znalosti**, **heuristiky**).
Heuristiky se používají tam, kde není k dispozici exaktní algoritmus. Ze dvou řešitelů

stejného problému je lepší ten, který je vybaven lepší množinou heuristik (prohledává menší část stavového prostoru, postupuje přímočařeji k cíli a jeho způsob řešení se jeví jako „inteligentnější“).
Podle využití znalostí o úloze je prohledávání
neinformované (nevyužívá znalostí o úloze)
informované (využívá znalostí o úloze)
Neinformované metody prohledávání
Metody neinformovaného prohledávání se dělí z hlediska pořadí, v jakém jsou uzly expandovány, na:

slepé prohledávání do šířky
o nejdříve se expanduje uzel s nejmenší hloubkou,
o nalezne nejmenší řešení (s nejmenším počtem operátorů – nejkratší cestu).

slepé prohledávání do hloubky
o nejdříve se expanduje uzel s největší hloubkou,
o má nižší nároky na paměť (uchovává pouze uzly na cestě od počátečního stavu ke stavu právě expandovanému),
o často spojeno s omezením maximální prohledávané hloubky, při jejím dosažení se používá mechanismu navracení.
Poznámky:
Hloubka uzlu = počet hran od počátečního uzlu k uzlu

Kompromis mezi prohledáváním do šířky a do hloubky:

Algoritmus DFID (algoritmus iterativního prohlubování)
Úplné prohledávání do hloubky tak, že se v každé iteraci zvyšuje povolená hloubka prohledávání o 1.
První nalezené řešení je optimální (ve smyslu nejkratší délky cesty).

Poznámky:
Neinformované metody jsou použitelné jen v nejjednodušších úlohách.
Nepoužívání znalostí o úloze vede na prohledávání příliš velkých částí stavového prostoru.
Představují metodologický podklad pro složitější, informované strategie.

6.Způsoby implementace ohodnocení grafu

Reprezentace grafů

graf lze reprezentovat v paměti počítače například tak, že vrcholy očíslujeme přirozenými čísly od 1 do N, hrany od 1 do M a odkud kam vedou hrany, popíšeme jedním z následujících tří způsobů:

- *matice sousednosti* – to je pole A velikosti N x N. Na pozici A[i, j] uložíme hodnotu 0 nebo 1 podle toho, zda z vrcholu i do vrcholu j vede hrana (1) nebo nevede (0). S maticí sousednosti se zachází velmi snadno, ale má tu nevýhodu, že je vždy kvadraticky velká bez ohledu na to, kolik je hran. Výhodou naopak je, že místo jedniček můžeme ukládat nějaké další informace o hranách, třeba jejich délky.
- řešení problému tzv. obchodního cestujícího
- princip:
- Problém obchodního cestujícího spočívá v tom, že chce navštívit n měst, které jsou mezi sebou různě propojena různě dlouhými silnicemi, přičemž by rád navštívil města v takovém pořadí, aby celková vzdálenost, kterou bude muset mezi nimi překonat, byla minimální.

Převzato do teorie grafů, za předpokladu, že města a jejich propojení reprezentujeme neorientovaným grafem s ohodnocenými hranami, lze tento problém formulovat jako problém nalezení co nejlevnější Hamiltonovy kružnice, tj. Hamiltonovy kružnice s co nejmenším součtem cen (ohodnocení) všech hran, které tuto kružnici tvoří. Připomeňme, že Hamiltonova kružnice je taková kružnice (taková uzavřená cesta), která prochází přes všechny uzly grafu. Z toho důvodu tuto kružnici tvoří právě tolik hran, kolik je v grafu uzlů (čili n).

- *seznam sousedů* je obvykle tvořen dvěma poli: polem sousedů S[1…M] obsahujícím postupně čísla všech vrcholů, do kterých vede hrana z vrcholu 1, pak z vrcholu 2 atd., a polem začátků Z[1…N], v němž se pro každý vrchol dozvíme začátek odpovídajícího úseku v poli S. Pokud navíc do Z[N+1] uložíme M+1, bude platit, že sousedé vrcholu i jsou uloženi v S[Z[i]], …, S[Z[i+1]-1]. Tato reprezentace má tu výhodu, že zabírá pouze prostor O(N+M) a sousedy každého vrcholu máme pěkně pohromadě a nemusíme je hledat.

- *Půl-hranami* – tato reprezentace se používá tehdy, pokud potřebujeme během výpočtu graf složitě upravovat. Je univerzální, ale dost pracná na naprogramování. Spočívá v tom, že si každou hranu uložíme jako dvě půl hrany (začátek a konec hrany), každý vrchol bude obsahovat spojové seznamy přicházejících a odcházejících půl hran a každá půl hrana bude ukazovat na svou druhou polovici.

```
  123456789
1 011000011
2 100110001
3 100100000
4 011010000
5 010101000
6 000010110
7 000001011
8 100001100
9 110000100
```

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
$E[i].a$	1	1	1	1	2	2	2	2	3	3	4	4	4	5	5	5	6	6	6	7	7	7	8	8	8	9	9	9
$E[i].b$	2	3	8	9	1	4	5	9	1	4	2	3	5	2	4	6	5	7	8	6	8	9	1	6	7	1	2	7

i	1	2	3	4	5	6	7	8	9	10
$V[i]$	1	5	9	11	14	17	20	23	26	29

Reprezentace grafu seznamem sousedů

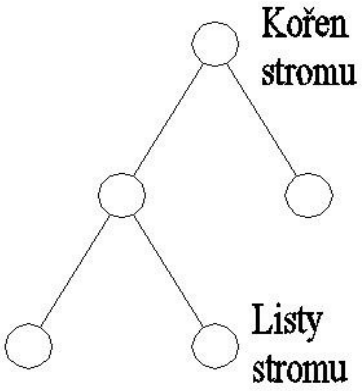
7. Speciální grafové topologie(binární stromy), implementace, AND/OR grafy

Strom:

speciální grafová struktura-acyklický graf (neobsahuje smyčky)
stromy obsahují tyto typy uzlů:

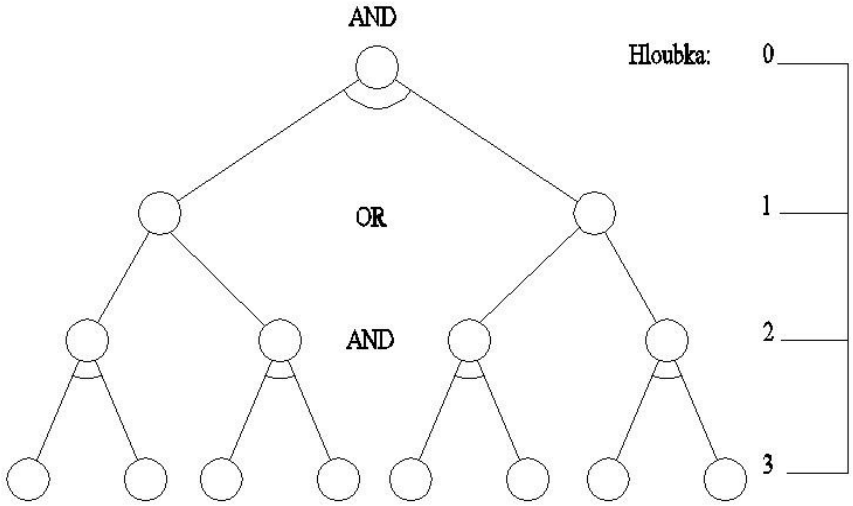
a) P(u)=0, N(u)>0 tzv. kořen stromu, nemá přímého předchůdce
b) P(u)=1, N(u)>0
c) P(u)=1, N(u)=0 tzv. list stromu, nemá přímého následníka

platí tedy : P(u) <=1 a N(u)>=0
kde P-předchůdce uzlu u,N-následník uzlu u
schéma binárního stromu(každý uzel má maximálně 2 následovníky):



And/or grafy

Tvořeny uzly výhradně typu AND nebo OR kde AND-představuje konjunkci pod úloh, k řešení úlohy pomocí AND/OR grafu je nezbytné, aby každá z pod úloh byla řešitelná
OR-představuje disjunkci pod úloh, k řešení úlohy je nezbytné, aby alespoň jedna z pod úloh byla řešitelná.



8.Výraz (formule) jako strom, manipulace s výrazy jako manipulace se stromy

9.Pojmy jazyka a gramatiky

Jazyky

Libovolná neprázdná konečná množina V je abecedou. Její prvky nazýváme znaky nebo symboly.

Slovo (řetězec) nad abecedou V je libovolná posloupnost konečné délky tvořená symboly z V.

Slova zapisujeme bez čárek, tedy posloupnost w = $\{a_i\}_{i=1}^{n}$ kde $a_i \in V$ zapíšeme jako slovo

w= $a_1 \ldots a_n$.

Délka slova w je délka w jako posloupnosti,značíme |w|=n.

Prázdné slovo, tedy posloupnost délky nula, značíme ε platí tedy délka $|\varepsilon| = 0$.

Symbolem V^* značíme množinu všech slov nad abecedou V, symbolem V^+ označujeme množinu $V^* \setminus \{\varepsilon\}$.

Jazyk L nad abecedou V je libovolná množina slov, $L \subseteq V^*$

platí : $\varepsilon * a = a * \varepsilon = a$

Příklad:
Nechť V={a,b} je abeceda. Jazyk nad touto abecedou může být definovaný takto:
L={w:w=$a^i b^i$, i>=0}
Potom slova patřící do jazyka L jsou : ε ,ab,aabb, nepatří tam ale slova b, ba.

Operace zřetězení
definujeme ji $V^* \times V^* \rightarrow V^*$ takto
$w_1=a_1 \ldots a_n$
$w_2=b_1 \ldots b_n$ => $w_1 w_2 = a_1 \ldots a_n b_1 \ldots b_n$

Zřetězení jazyků $L_1 L_2$ $L_1, L_2 \subseteq V^*: 2^{V^*} \times 2^{V^*} \rightarrow 2^{V^*}$
$L_1 L_2 = \{ w_1 w_2 : w_1 \in L_1, w_2 \in L_2 \}$

Tímto jsme definovali jazyky, které ale mohou být nekonečné, které můžeme dále se pokusit reprezentovat konečně. K tomuto účelu použijeme algoritmy a procedury.

Algoritmus:
program, který se vždy zastaví když narazí na slovo w na vstupu vrátí odpověď ano jestliže $w \in L$, a odpověď ne, jestliže $w \notin L$.

Procedura:
program, který pro vstupní slovo w se zastaví a vrátí odpověď ano a pro vstup $w \notin L$ se buď zastaví, a vrátí ne, nebo se vůbec nezastaví.

Nechť U a V jsou dva jazyky. Zřetězení jazyků značíme a definujeme: UV={uv:u$\in$U $\wedge$ v$\in$V}

Gramatiky
gramatika je čtveřice G=(N,∑,P,S) kde
N – množina neterminálů
∑ – množina terminálů, platí N∩∑= 0 a označíme V = $N \supset \sum$ kde V je celková abeceda gramatiky G

P – množina pravidel, P ⊆ $V^* N V^*$ x V^* jde tedy o dvojice slov, první z nich obsahuje alespoň jeden neterminální symbol
S – počáteční symbol (kořen) gramatiky G, je to neterminál (S∈ N)

Pravidla typu [α,β] gramatiky G budeme zapisovat ve tvaru $\alpha \rightarrow \beta$

Definice relací odvozených v gramatice G, tzv. relace na množině slov celkové abecedy G: $=>_G \subseteq V^{*2}$, tedy slovo y∈ V^* lze odvodit ze slova x∈ V^* v gramatice G=(N,∑,P,S), $x=>_G y$, jestliže existuje pravidlo (α →β)∈ P a slova γ∈ V^* a δ∈ V^* tak, že x= γαδ a y=γβδ.

Symbolické značení:
neterminály : A, B, ...Z
terminály : a, b, c, ...
řetězce terminálů : u, v, w, x, y, z
obecné řetězce : α, β, γ, …

Příklad:
Nechť G_1=({A,S},{a,b},P,S), kde P={S→aAb,aA→aaAb,A→ε}, je gramatika. Pak v ní existují např. tyto odvození:
aaAbbaabb
aAbb $\Rightarrow G_1$ aaaAbbb

definice:

Všem slovům α∈ V^* takovým, že $S \overset{*}{\Rightarrow} G1\,\alpha$ říkáme větné formy gramatiky G. Pokud je α∈ $\sum^*$ pak slovo α nazýváme větou gramatiky G.
Jazyk generovaný gramatikou G je množina všech vět gramatiky G, značíme
L(G)={w∈ $\sum^*$: $S \overset{*}{\Rightarrow} G1\,\alpha\,\omega$ }
Gramatiky G_1 a G_2 nazýváme ekvivalentní, pokud $L(G_1)= L(G_2)$.

V gramatice G_1 z příkladu je
S $\Rightarrow G_1$ aAb $\Rightarrow G_1$ ab
S $\Rightarrow G_1$ aAb $\Rightarrow G_1$ aaAbb $\Rightarrow G_1$ aabb
aAb , aaAbb jsou větné formy a ab, aabb její věty. Celkem $L(G_1)$={ $a^i b^i$:i∈ N }
Gramatika G_2=({S},{a,b},{S→aSb,S→ab},{S})je ekvivalentní gramatice G_1.

<u>Chomského hierarchie</u>
Chomského hierarchie je hierarchie tříd formálních gramatik generujících formální jazyky. Byla vytvořena Noamem Chomskym v roce 1956.
Některé ze speciálních jazyků lze generovat speciálními gramatikami s pravidly speciálního typu.

Gramatiky typu 0
Zahrnují v sobě všechny formální gramatiky, generují právě ty jazyky, které mohou být rozpoznané nějakým Turingovým strojem. Tyto jazyky se někdy nazývají rekursivně spočetné jazyky.
Gramatiky typu 1 (kontextové gramatiky)
Generují kontextové jazyky. Tyto gramatiky se skládají z pravidel $\alpha A\beta \rightarrow \alpha\gamma\beta$. , kde A je neterminál a α, β a γ řetězce terminálů a neterminálů. Řetězce α a β mohou být prázdné, ale γ musí být neprázdná. Pravidlo $S \rightarrow \varepsilon$ je povoleno, pokud se S nevyskytuje na pravé straně žádného pravidla. Tyto jazyky jsou právě jazyky rozpoznatelné lineárně ohraničeným Turingovým strojem.

Gramatiky typu 2 (bezkontextové gramatiky)
Generují bezkontextové jazyky. Skládají se z pravidel $A \rightarrow \gamma$ s neterminálem A a řetězcem terminálů a neterminálů γ. Pravidlo $S \rightarrow \varepsilon$ je povoleno, pokud se S nevyskytuje na pravé straně žádného pravidla. Tyto jazyky jsou právě jazyky rozpoznatelné nějakým nedeterministickým zásobníkovým automatem.

Gramatiky typu 3 (regulární gramatiky)
Generují regulární jazyky. Pravidla těchto gramatik jsou omezena na jeden neterminál na levé straně. Pravá strana se skládá z řetězce terminálů, který může být následován jedním neterminálem. Tyto gramatiky se také nazývají pravé lineární gramatiky. Obdobně se definují i levé lineární gramatiky, kde může být na pravé straně pravidel řetězec terminálů předcházen jedním neterminálem. Pravé lineární gramatiky a levé lineární gramatiky jsou ekvivalentní. Regulární gramatika je ve standardní formě, pokud je pravá strana tvořena jedním terminálem následovaným jedním neterminálem nebo pokud je pravá strana prázdné slovo. Tyto jazyky jsou právě jazyky rozpoznatelné konečným automatem.

10.Automaty a gramatiky. Konečné automaty, deterministické a nedeterministické, bez zásobníku, se zásobníkem

Konečný automat(KA)

Definice KA:

Nedeterministickým konečným automatem (NKA) rozumíme jednocestný iniciální automat M

specifikovaný 5-ticí $M = (Q, \Sigma, \delta, q_0, F)$ kde

(1) Q .... je konečná *množina stavů*

(2) Σ .....je konečná vstupní abeceda

(3) δ .....je přechodová funkce tvaru $\delta: Q \times \Sigma \rightarrow 2^Q$

(4) $q_0 \in Q$ je *počáteční stav*

(5) $F \subseteq Q$ je *množina koncových stavů*

Je-li $\delta: Q \times \Sigma \rightarrow Q \cup \{\text{nedefinovaný prvek}\}$, tj. $|\delta(q, a)| = 1$ pro všechny $q \in Q$ a $a \in \Sigma$, pak se M nazývá *deterministický konečný automat (DKA)*.

Jazyk přijímaný konečným automatem

• Řetězec w přijímaný NKA M je definován takto: $(q_0, w) \overset{*}{\underset{M}{|}} (q, e),\ q \in F$

• Jazyk L(M) přijímaný NKA M: $L(M) = \{w \mid (q_0, w) \mid \overset{*}{\underset{M}{-}} (q, e) \wedge q \in F\}$

Ekvivalence NKA a DKA

Každý NKA M lze převést na ekvivalentní DKA M' tak, že L(M) = L(M')

Důkaz: (1) nalezneme algoritmus převodu M → M'
(2) ukážeme, že L(M) = L(M') tj. ukážeme, že platí:
(a) $L(M) \subseteq L(M')$ a současně
(b) $L(M') \subseteq L(M)$

Algoritmus převodu NKA na ekvivalentní DKA:

(1) Polož $Q' = (2^Q - \{\emptyset\}) \cup \{\text{nedef.}\}$

(2) $q'_0 = \{q_0\}$

(3) $F' = \{S \mid S \in 2^Q \wedge S \cap F \neq \emptyset\}$

(4) Pro všechna $S \in 2^Q - \{\emptyset\}$ a pro všechna $a \in \Sigma$ polož $\delta'(S,a) = \bigcup_{q \in S} \delta(q,a)$.

Je-li $\delta'(S,a) = \emptyset$, polož $\delta'(S,a) = \text{nedef.}$

Vztah regulárních jazyků a jazyků přijímaných konečným automatem

Definice pravé lineární resp. levé lineární gramatiky:

Gramatika G=(N, Σ, P, S) s pravidly tvaru

$A \to xB$ $A,B \in N, x \in \Sigma^*$ nebo

$A \to x$ $x \in \Sigma^*$

resp. tvaru

$A \to Bx$ $A,B \in N$

$A \to x$ $x \in \Sigma^*$

se nazývá *pravá lineární* resp. *levá lineární* gramatika.

Gramatika G=(N, Σ, P, S) s pravidly tvaru

$A \to aB$ $A,B \in N, a \in \Sigma$

$A \to a$

případně $A \to e$, pokud $e \in L(G)$

resp. s pravidly tvaru

$A \to Ba$ $A,B \in N, a \in \Sigma$

$A \to a$

případně $A \to e$, pokud $e \in L(G)$

se nazývá *pravá regulární* resp. *levá regulární* gramatika.

Poznámka:

Gramatika G=(N, Σ, P, S) s pravidly tvaru $A \rightarrow xBy$ nebo $A \rightarrow x$, kde $A,B \in N$ a $x,y \in \Sigma^*$ se nazývá *lineární.*

Označme

L_{PL} všechny jazyky generované pravými lineárními gramatikami

L_{LL} všechny jazyky generované levými lineárními gramatikami

L_L všechny jazyky generované lineárními gramatikami

Platí

$L_{PL} = L_{LL}$ a $L_{PL} \subset L_L$

Každá pravá lin. gramatika G=(N, Σ, P, S) může být převedena na gramatiku G'=(N', Σ', P', S'), kde P' obsahuje pouze pravidla tvaru

$A \rightarrow aB$ nebo

$A \rightarrow e$ $A,B \in N'$, $a \in \Sigma$

tak, že L(G)=L(G')

Regulární množiny a výrazy, rovnice nad regulárními výrazy, převod regulárního výrazu na konečný automat, minimalizace konečného automatu, vlastnosti regulárních jazyků

Definice regulární množiny:

Nechť Σ je konečná abeceda. *Regulární množinu nad Σ* definujeme (rekurentně) takto:

(1) ∅ (prázdná množina) je regulární množina nad Σ

(2) {e} je regulární množina nad Σ

(3) {a} je regulární množina nad Σ pro všechna $a \in \Sigma$

(4) jsou-li P a Q regulární množiny nad Σ, pak také

(a) $P \cup Q$

(b) P.Q

(c) P^*

jsou regulární množiny nad Σ.

(5) Jiné regulární množiny nad Σ než ty, které lze získat aplikací (1)-(4) neexistují.

Definice regulárních výrazů:

Regulární výrazy nad Σ a regulární množiny, které označují jsou definovány takto:

(1) ∅ je regulární výraz označující regulární množinu ∅

(2) e je regulární výraz pro množinu {e}

(3) a je regulární výraz pro množinu {a}

(4) jsou-li p,q regulární výrazy označující regulární množiny P,Q, pak

(a) (p+q) je regulární výraz označující množinu P∪Q

(b) (pq) je regulární výraz označující množinu P.Q

(c) (p^*) je regulární výraz označující množinu P^*

(5) Jiné regulární výrazy nad Σ neexistují.

Konvence:

1. reg. výraz p^+ je roven reg. výrazu pp^*
2. zavedení priority operátorů:
   (a) $^+,^*$ (iterace) - nejvyšší
   (b) .
   (c) + nejnižší
3. vynechávání redundantních závorek

Rovnice nad regulárními výrazy

Definice rovnice nad regulárními výrazy:

Rovnice, jejímiž složkami jsou koeficienty a neznámé, které reprezentují (dané a hledané) regulární výrazy

nazýváme *rovnicemi nad regulárními výrazy.*

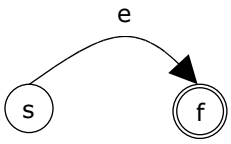
- rovnice $X = aX + b$ má řešení $X = a^*b$
- rovnice $X = pX + q$ má řešení $X = p^*(q+r)$
- obvykle hledáme „nejmenší řešení“ nazývané nejmenší pevný bod: $X=p^*q$
- soustava rovnic $X = a_1X + a_2Y + a_3$, $Y = b_1X + b_2Y + b_3$ má řešení
  $X = (a_1+a_2b_2^*b_1)^*(a_3+ a_2b_2^*b_3)$, $Y = (b_2+b_1a_1^*a_2)^*(b_3+ b_1a_1^*a_3)$

Převod regulárního výrazu na RKA

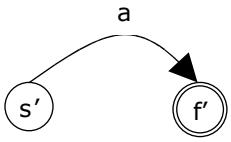
Metoda:

(a) rozložíme výraz *r* na jeho primitivní složky podle rekurentní definice regulární množiny

(b) 1. pro výraz e zkonstruujeme automat:

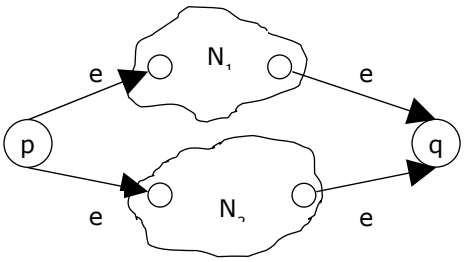


2. pro výraz a, $a \in \Sigma$ vytvoříme automat:

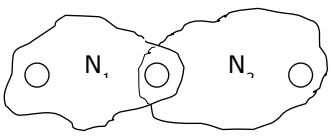


3. Nechť N_1 je automat přijímající jazyk specifikovaný výrazem r_1 a nechť N_2 je automat přijímající jazyk specifikovaný výrazem r_2.

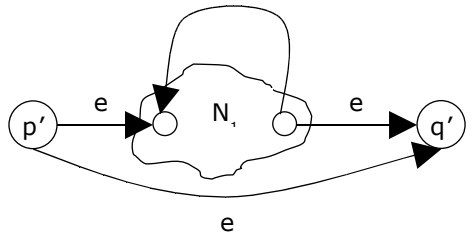
Pro výraz r_1+r_2 vytvoříme automat



Pro výraz r_1r_2 vytvoříme automat



Pro výraz r_1^* vytvoříme automat



Minimalizace konečného automatu

Definice rozlišování stavů:

Nechť M=(Q, Σ, δ, q_0, F) je deterministický, úplně definovaný konečný automat a nechť $q_1,q_2 \in Q$, $q_1 \neq q_2$. Říkáme, že řetězec $x \in \Sigma^*$ *rozlišuje* stav q_1 od stavu q_2, jestliže

$$(q_1, x) \vdash^* (q_3, e) \wedge (q_2, x) \vdash^* (q_4, e)$$

a právě jeden ze stavů q_3,q_4 je v množině F. Říkáme, že stavy q_1,q_2 jsou *k-nerozlišitelné*, píšeme

$q_1 \overset{k}{\equiv} q_2$ právě když neexistuje řetězec x, $|x| \leq k$, který rozlišuje stavy q_1,q_2.

Stavy q_1,q_2 jsou *nerozlišitelné*, jsou-li pro každé $k \geq 0$ k-nerozlišitelné. (≡)

Stav $q \in Q$ je *nedostupný*, jestliže neexistuje řetězec x, $x \in \Sigma^*$, pro který $(q_0, x) \vdash^* (q, e)$

Automat M nazýváme *redukovaný*, jestliže žádný stav z Q není nedostupný žádné dva stavy nejsou nerozlišitelné.

Převod na redukovaný konečný automat:

(1) Nalezni a odstraň nedostupné stavy

(2) Vytvoř relace $\overset{0}{\equiv}, \overset{1}{\equiv}, \ldots \overset{k}{\equiv} = \overset{k+1}{\equiv}$. Polož $\equiv = \overset{k}{\equiv}$.

(3) Vytvoř automat M'=(Q', Σ', δ', q'_0, F') takto:

(a) Q' = Q \ ≡ (faktorová množina podle ≡) ≡ je relace ekvivalence

Označme [p] třídu stavů ekvivalentních se stavem p.

(b) δ'([p],a)=[q], jestliže δ(p,a)=q

(c) $q'_0=[q_0]$

(d) F'={[q]|q∈F}

Vlastnosti regulárních jazyků

- Strukturální vlastnosti
  - Každý konečný jazyk je regulární.
  - Pumping Theorem:

    Nechť L je nekonečný regulární jazyk. Pak existuje celočíselná konstanta p taková, že platí:

    $w \in L \wedge |w| \geq p \Rightarrow w=xyz \wedge 0 \leq |y| \leq p \wedge xy^i z \in L$ pro $i \geq 0$

    Důkaz:

    Položme p=n, $w \in L$, $|w| \geq n$. M přijme w průchodem alespoň n+1 konfiguracemi a tudíž aspoň dvě z nich obsahují stejný stav. Pak ale existuje posloupnost konfigurací, které se mohou opakovat.
  - Jazyk $L=\{0^n 1^n \mid n \geq 1\}$ není regulárním jazykem
- Uzávěrové vlastnosti
  - Třída regulárních jazyků je uzavřena vzhledem k operacím: *sjednocení*, *konkatenace*, *iterace*
  - Třída regulárních jazyků tvoří množinovou Boolovu algebru ($1 = \Sigma^*$, $0 = \emptyset$)
- Rozhodnutelné problémy
  - Problém "neprázdnosti": $L \neq \emptyset$
  - Problém "náležitosti": $w \in L$
  - Problém "ekvivalence": $L(G_1)=L(G_2)$

Bezkontextové jazyky, derivační strom, víceznačnost gramatik, transformace bezkontextových gramatik, problém syntaktické analýzy bezkontextových jazyků

Bezkontextové jazyky

Definice bezkontextové gramatiky:

Gramatika $G=(N, \Sigma, P, S)$ se nazývá gramatikou *bezkontextovou*, jestliže všechna pravidla mají tvar $A \rightarrow \alpha$, $A \in N$, $\alpha \in (N \cup \Sigma)^*$

Každý regulární jazyk je bezkontextový.

Definice rekurzivnosti:

Nechť G=(N, Σ, P, S) je bezkontextová gramatika a $p \in P$ její přepisovací pravidlo. Pravidlo p nazveme *rekurzivní*, resp. *rekurzivní zleva*, resp. *rekurzivní zprava* má-li tvar

$A \to \alpha A \beta$, resp.

$A \to A\alpha$, resp.

$A \to \alpha A$,

kde $A \in N$, $\alpha, \beta \in (N \cup \Sigma)^*$.

Derivační strom

Definice levé resp. pravé derivace:

Nechť $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n \equiv \alpha$ je derivace větné formy α. Jestliže v každém řetězci α_i, i=1, ..., n-1 byl přepsán nejlevější resp. nejpravější nonterminál, pak tuto derivaci nazýváme *levou* resp. *pravou derivací* větné formy α.

Definice derivačního stromu:

Nechť δ je větná forma generovaná v gramatice G=(N, Σ, P, S) a nechť $S \equiv \nu_0 \Rightarrow \nu_1 \Rightarrow \ldots \Rightarrow \nu_n \equiv \delta$ je její derivace v G.

Derivační strom příslušející derivaci je kořenový (orientovaný) strom, vrcholy s těmito vlastnostmi:

(1) Vrcholy jsou ohodnoceny symboly z $(N \cup \Sigma)$, kořen je ohodnocen symbolem S

(2) Přímé derivaci $\nu_{i-1} \Rightarrow \nu_i$, i=1, ..., n, kde

$\nu_{i-1} = \mu A \lambda$ $\mu, \lambda \in (N \cup \Sigma)^*$, $A \in N$

$\nu_i = \mu X_1 X_2 \ldots X_k \lambda$ $X_i \in (N \cup \square)$

$A \to X_1 X_2 \ldots X_k$ je pravidlo z P

odpovídá *k* hran $(A,X_1),(A,X_2),\ldots,(A,X_k)$, uspořádaných (zleva do prava) v tomto pořadí.

(3) Konkatenace ohodnocení listů derivačního stromu zleva doprava tvoří odpovídající větnou formu δ.

Definice l-fráze větné formy:

Nechť G=(N, Σ, P, S) je gramatika a nechť řetězec $\lambda = \alpha\beta\chi$ je větná forma. Podřetězec β se nazývá *frází větné formy λ vzhledem k nonterminálu A*, jestliže platí:

$S \overset{*}{\Rightarrow} \alpha A \chi$ a současně

$A \overset{+}{\Rightarrow} \beta$

Jestliže navíc platí $A \Rightarrow \beta$, pak potom β se nazývá *jednoduchou frází větné formy λ vzhledem k A*.

Nechť $\lambda = \alpha_1\beta_1\alpha_2 \ldots \beta_n\alpha_{n+1}$, je větná forma a nechť podřetězce $\beta_1, \beta_2, \ldots, \beta_n$ jsou všechny

jednoduché fráze větné formy λ. Pak β_1 se nazývá *l-frází větné formy* λ.

Víceznačnost gramatik

Definice víceznačné gramatiky:

Nechť G je gramatika. Věta w generovaná gramatikou G se nazývá *víceznačnou větou*, jestliže k ní existuje alespoň dva různé derivační stromy. Gramatika G je *víceznačná*, pokud generuje víceznačnou větu. Jazyk, ke kterému neexistuje jednoznačná gramatika se nazývá jazykem s *inherentní víceznačností*.

Věta w je v gramatice G víceznačná právě tehdy, existují-li v G dvě různé levé (pravé) derivace věty w.

Definice gramatiky obsahující cyklus:

Nechť G=(N, Σ, P, S) je gramatika, A∈ N. Gramatika G obsahuje *cyklus*, jestliže $A \overset{+}{\Rightarrow} A$

Zdroje cyklu:

- jednoduchá pravidla např. A⇒B⇒C⇒A
- e-pravidla např. A⇒AB⇒A (v důsledku pravidla B→e)

Transformace bezkontextových gramatik

- odstranění zbytečných symbolů gramatiky
- výpočet množiny nonterminálů generujících terminální řetězce
- výpočet množiny dostupných symbolů
- odstranění zbytečných symbolů
- odstranění e-pravidel
- odstranění jednoduchých pravidel
- gramatika bez zbytečných pravidel, e-pravidel a bez cyklů se nazývá vlastní gramatikou
- odstranění přímé a nepřímé levé rekurze

Problém syntaktické analýzy bezkontextových jazyků

- Syntaktickou analýzou věty rozumíme nalezení její derivace nebo derivačního stromu
- Klasifikace podle způsobu konstrukce derivačního stromu
  - syntaktická analýza shora dolů
  - syntaktická analýza zdola nahoru
- Klasifikace podle pořadí aplikace přepisovacích pravidel

- nedeterministická (s návraty)
- deterministická

Zásobníkové automaty, vztah zásobníkových automatů a bezkontextových gramatik, deterministické bezkontextové jazyky, vlastnosti bezkontextových jazyků

Zásobníkové automaty

Definice zásobníkového automatu:

Zásobníkový automat je n-tice $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Q .... je konečná množina stavů

Σ ..... je konečná vstupní abeceda

Γ ..... je konečná zásobníková abeceda

δ ..... je přechodová funkce tvaru $\delta: Q\times(\Sigma\cup\{e\})\times\Gamma\rightarrow 2^{Q\times\Gamma^*}$

$q_0 \in Q$ je počáteční stav

$Z_0 \in \Gamma$ je startovací symbol zásobníku

$F \subseteq Q$ je množina koncových stavů

Pokud je $\delta: Q\times(\Sigma\cup\{e\})\times\Gamma^*\rightarrow 2^{Q\times\Gamma^*}$, potom je P rozšířený zásobníkový automat.

Ke každému RZA P existuje ZA P' takový, že L(P)=L(P').

Definice konfigurace zásobníkového automatu:

Nechť $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ je zásobníkový automat. *Konfigurací* automatu P nazveme trojici

$$(q, w, \alpha) \in Q\times\Sigma^*\times\Gamma^* \quad \text{kde}$$

(1) q je přítomný stav vnitřního řízení

(2) w je dosud nezpracovaná část vstupního řetězce

(3) α je obsah zásobníku (na začátku vlevo je vrchol zásobníku)

Definice přechodu zás. automatu:

Přechod zásobníkového automatu P je binární relace $|\overline{}_P$ definovaná na množině konfigurací:

$$(q, aw, Z\chi) \,|\overline{}_P (q', w, \beta\chi) \overset{def.}{\Leftrightarrow} (q', \beta) \in \delta(q, a, Z) \quad \text{kde } q,q' \in Q,\ a \in \Sigma\cup\{e\},\ w \in \Sigma^*,\ Z \in \Gamma^*.$$

Je-li a=e, pak odpovídající přechod nazýváme *e-přechodem*.

Relace $|\overline{}_P^{\,i}$, $|\overline{}_P^{\,*}$ $|\overline{}_P^{\,+}$ jsou definovány obvyklým způsobem.

Automat přijímající s vyprázdněním zásobníku:

Ke každému ZA P existuje ZA P', který přijímá s vyprázdněním zásobníku a L(P)=L(P').

Řetězec přijímaný zásobníkovým automatem:

Platí-li pro řetězec $w \in \Sigma^*$ relace $(q_0, w, Z_0) \vdash^* (q, e, \chi)$ $q \in F$, $\chi \in \Gamma^*$ pak říkáme, že w je přijímán ZA P.

(q_0,w,Z_0), resp. (q,e,γ) je počáteční resp. koncová konfigurace.

Jazyk přijímaný zásobníkovým automatem:

Jazyk L(P) přijímaný ZA: $L(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, e, \chi) \wedge q \in F\}$

Vztah zásobníkových automatů a bezkontextových gramatik

- Ke každé bezkontextové gramatice existuje ZA P, který přijímá její jazyk s vyprázdněním zásobníku (bude vytvářet levou derivaci vstupního řetězce – tzn. modelovat syntaktickou analýzu shora dolů).
- $L_2 \subseteq L_P$

  Nechť P=(Q, Σ, Γ, δ, q_0, Z_0, F) je zásobníkový automat. Pak existuje gramatika G=(N, Σ, P, S) taková, že

  L(P)=L(G). Gramatiku G budeme definovat formálně takto:

  (1) $N=\{[qZr] \mid q,r \in Q, Z \in \Gamma\} \cup \{S\}$

  (2) Pokud $\delta(q, a, Z)$ obsahuje $(r, X_1X_2\ldots X_k)$, potom přidám pravidlo $[qZs_k] \rightarrow [rX_1s_1][s_1X_2s_2]\ldots[s_{k-1}X_ks_k]$

  pro každou posloupnost stavů $s_1, s_2, \ldots, s_k$ z množiny Q

  (3) Jestliže $(r,e) \in \delta(q,a,Z)$, pak k P přidám pravidlo $[qZr] \rightarrow a$

  (4) Pro každý stav $q \in Q$ přidám k P pravidlo $S \rightarrow [q_0Z_0q]$)
- $L_2 = L_P$

Deterministické bezkontextové jazyky

Definice deterministického zásobníkového automatu (DZA):

Rozšířený zásobníkový automat P=(Q, Σ, Γ, δ, q_0, Z_0, F) nazveme deterministickým zás. automatem pokud platí:

(1) $\forall q \in Q,\ \forall a \in \Sigma \cup \{e\},\ \forall \chi \in \Gamma^*: |\delta(q,a,\chi)| \leq 1$

(2) je-li $\delta(q,a,\alpha) \neq \emptyset$ a $\delta(q,a,\beta) \neq \emptyset$, $\alpha \neq \beta$, pak $\alpha \neq \beta\omega$, ani $\beta \neq \alpha\omega$ pro nějaké $\omega \in \Gamma^*$

(3) je-li $\delta(q,a,\alpha)|\neq\emptyset$ a $\delta(q,e,\beta)|\neq\emptyset$, $\alpha\neq\beta\omega$ ani $\beta\neq\alpha\omega$ pro nějaké $\omega\in\Gamma^*$

Bezkontextový jazyk $L=\{ww^R \mid w\in\Sigma^+\}$ nelze přijímat žádným DZA.

Definice deterministického bezkontextového jazyka:

Jazyk L je *deterministický bezkontextový jazyk*, pokud existuje deterministický zás. automat P takový, že L=L(P).

Označme L_P^D třídu deterministických bezkontextových jazyků. Pak $L_P^D \subset L_2$

NORMÁLNÍ TVARY BEZKONTEXTOVÝCH GRAMATIK

Vlastnosti bezkontextových jazyků

Definice bezkontextové gramatiky v Chomského normální formě:

Bezkontextová gramatika G=(N, Σ, P, S) je v Chomského normální formě, má-li každé pravidlo z P jeden z těchto tvarů:

(1) $A\rightarrow BC$ $A,B,C\in N$

(2) $A\rightarrow a$ $a\in\Sigma$

(3) je-li $e\in L(G)$, pak $S\rightarrow e$ je jediné e-pravidlo a S se nevyskytuje na pravé straně žádného přepisovacího pravidla

Nechť G je bezk. gramatika. Pak existuje gramatika G' v Chomského normální formě taková, že L(G')=L(G).

Definice bezkontextové gramatiky v Greibachové normální formě:

Bezkontextová gramatika G=(N, □, P, S) je v Greibachové normální formě, je-li G gramatikou bez e-pravidel a každé pravidlo z P (vyjma případného pravidla $S\rightarrow e$) má tvar: $A\rightarrow a\alpha$, kde $a\in\Sigma$, $\alpha\in N^*$

STRUKTURÁLNÍ VLASTNOSTI BEZKONTEXTOVÝCH GRAMATIK

Pumping theorem pro bezkontextové jazyky

Nechť L je bezkontextový jazyk. Pak existuje konstanta k taková, že je-li $z \in L$ a $|z| \geq k$, pak lze z napsat ve tvaru:

$z=uvwxy,\ vx \neq e,\ |vwx| \leq k$

a pro všechna $i \geq 0$ je

$uv^iwx^iy \in L$

např. $S \Rightarrow^* uAy \Rightarrow^+ uvAxy \Rightarrow^+ uvvAxxy \Rightarrow^+ uvvwxxy$

Jazyk $L=\{a^nb^nc^n \mid n \geq 1\}$ není bezkontextovým jazykem.

Důkaz: Platí, že jestliže $A \Rightarrow^+ \alpha$, pak $|\alpha| \leq 2^{m-1}$, kde m je počet vrcholů nejdelší cesty v odpovídajícím

derivačním stromu. Nechť $|N| = n$. Položme $k=2^{n+1}$ a uvažujme libovolnou větu z takovou, že $|z| \geq k$.

Pak nejdelší cesta v odpovídajícím derivačním stromu obsahuje alespoň n+2 vrcholů, z nichž jsou

nutně alespoň 2 označeny stejným nonterminálem.

UZÁVĚROVÉ VLASTNOSTI BEZKONTEXTOVÝCH GRAMATIK

Bezkontextové jazyky jsou uzavřeny vzhledem k:

- substituci
- sjednocení
- součinu
- iteraci
- pozitivní iteraci
- morfismu

Nejsou uzavřeny vzhledem k:

- průniku
- komplementu

ROZHODNUTELNÉ PROBLÉMY

- $L(G)=\emptyset$ jazyk generovaný gramatikou G je neprázdný
- $\exists k \in \mathbf{N}: |L(G)|=k$ jazyk L(G) je konečný

NEROZHODNUTELNÉ PROBLÉMY

- $L(G_1) \overset{?}{=} L(G_2)$
- G je jednoznačná?

Oba problémy lze dokázat např. předvedením na řešení tzv. *Postova problému přiřazení*, který je ekvivalentní *problémem zastavení Turingova stroje*, což jsou základní nerozhodnutelné problémy.

11) Implementace konečného automatu

Možná reprezentace konečného automatu:
Automat lze reprezentovat pomocí grafu. Stav je roven vrcholu, možný přechod mezi stavy je reprezentován spojnicí vrcholů. Ohodnocení přechodu jsou data spojnice. Aktuální stav je reprezentován proměnnou grafu, která určuje aktuální vrchol. Jestli je stav konečný je určeno pomocí booleovské proměnné vrcholu.

Algoritmus konečného automatu:

1) automat je v počátečním stavu (reprezentováno vnitřní proměnnou automatu – grafu, která určuje který stav – vrchol je aktuální)

2) automatu přijde znak z kontrolovaného slova (je zavolána metoda jejíž parametr je znak)
   - 2.1) automat zkontroluje zda existuje přechod (spojnice) ohodnocený příchozím znakem (z aktuálního stavu/vrcholu)
   - 2.1.1) pokud spojnice neexistuje → slovo není generováno gramatikou – není třeba dále kontrolovat
   - 2.1.2) spojnice existuje → přejít do dalšího stavu (vnitřní proměnnou, říkající který vrchol je aktuální, nastavit na vrchol daný přechodem/spojnicí)

3) krok 2 opakovat tak dlouho dokud se nevyčerpají všechny znaky kontrolovaného slova nebo nenastane případ 2.1.1

4) pokud jsou všechny znaky slova zkontrolována (tj. nenastal případ 2.1.1 – zkontrolovat zda je automat v konečném stavu
   - 4.1) pokud je automat v konečném stavu (aktuální stav/vrchol je označen jako konečný), je slovo generováno gramatikou
   - 4.2) pokud automat není v konečném stavu, slovo není generováno gramatikou

5) nastal případ 2.1.1 - slovo není generováno gramatikou

12.Turingův stroj, vyčíslitelnost, složitost algoritmu

Turingův stroj

Definice

Turingův stroj M je 6-tice tvaru $M=(S, \Sigma, \Gamma, \delta, q_0, q_F)$, kde

S .... je konečná množina vnitřních stavů

Σ .... je konečná množina (neblankových) symbolů nazývaná *strojová (vstupní)* abeceda

Γ .... je konečná množina, $\Sigma \subseteq \Gamma$, symbolů nazývaná pásková abeceda

δ .... je zobrazení:

$$\delta: (Q \setminus \{q_F\}) \times \Gamma \rightarrow Q \times (\Gamma \cup \{L,R\});\ L,R \notin \Gamma$$

q_0 .... je počáteční stav

q_F .... je koncový stav

Modulární stavba TS

Složitější stroje, jako akceptory jazyků, se vytváří ze základních bloků.

Spojování TS

Uvažujeme "sjednocení" a "konkatenaci" diagramů TS. Ačkoliv budeme mluvit o vytváření TS pomocí spojování jednodušších strojů, ve skutečnosti půjde o spojování přechodových diagramů TS, stejně jako se programové moduly kombinují při vytváření velkých softwarových systémů. Z tohoto důvodu bude vhodnější reprezentovat programy TS jako přechodové diagramy. Předpokládejme že TS M1 a M2 mají přechodové diagramy T1 a T2 a jejich páskové symboly jsou dány množinou Γ. Chceme-li vytvořit přechodový diagram jiného TS, který simuluje nejprve činnost M1 a potom činnost M2, odstraníme jednoduše příznak koncového stavu T1 a příznak počátečního stavu v T2 a potom pro každé x z Γ nakreslíme hranu označenou x/x z původního koncového stavu v T1 do původního počátečního stavu v T2.

Předpokládejme, že chceme spojit přechodové diagramy několika TS abychom získaly stroj, který simuluje nějakou kombinaci původních strojů. Postup je následující :

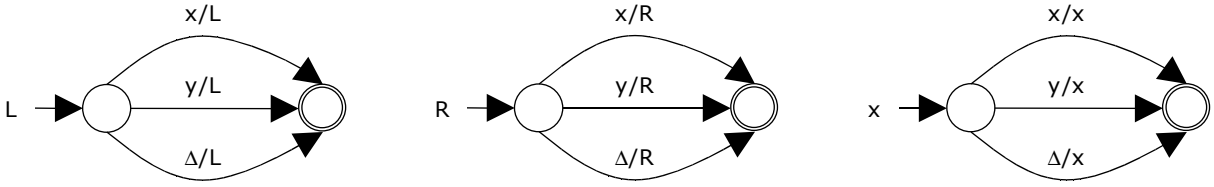
1) odstraň označení poč. stavu u všech strojů s výjimkou jednoho, zbylý počáteční stav bude poč. stavem,
   v němž bude začínat nově složený stroj.
2) odstraň koncové značení ze všech koncových stavů a vytvoř nový koncový stav, který není součástí

žádného původního přechodového diagramu.

3) z každého původního koncového stavu p a pro každé x z Γ nakresli hranu takto:
   a) Jestliže má složený stroj zastavit při dosažení stavu p a aktuálního symbolu x, nakresli hranu označenou x/x z p do nového kon. stavu.
   b) Jestliže má složený stroj předat řízení stroji M = (S, Σ, Γ, δ, q_0, q_k) při dosažení stavu p a aktuálního symbolu x, nakresli hranu označenou x/z z p do stavu q stroje M, kde δ (q_0, x)=(q, z).

Základní bloky:

Stroje L, R, x Γ={x,y,Δ}



Stroje L_x, $L_{\square x}$, R_x, $R_{\neg x}$

TS S_R a S_L pro posuv (Shift) obsahu pásky. Stroj S_R posune řetězec neblankových symbolů nacházejících se vlevo od aktuálního symbolu o jeden symbol doprava. Stroj S_L pracuje symetricky.

Vícepáskový TS

TS, které mají více než jednu pásku označujeme *k-páskové* TS, kde *k* je přirozené číslo, které udává počet pásek a odpovídajících hlav. Volba přechodu, který se v daném okamžiku provede, závisí na aktuálních symbolech všech pásek a na aktuálním stavu stroje. Akce, které se při přechodu provedou se týkají pouze jedné pásky. Testování, zda vícepáskový TS přijme daný řetězec symbolů, začíná z počátečního stavu, kdy je vstupní řetězec zapsán na první pásce. Ostatní pásky jsou prázdné a jsou umístěny nad nejlevějším místem. Řetězec je přijat, jestliže stroj přejde z počáteční konfigurace do koncového stavu.

Přechodová funkce k-páskového TS:

$$\boldsymbol{\delta}: (Q \setminus \{q_F\}) \times (\Gamma_1, \Gamma_{2,\ldots,} \Gamma_k) \rightarrow Q \times (\Gamma_i \cup \{L,R\}) \times \{1,..,k\}$$

Ke každému vícepáskovému TS M existuje jednopáskový TS M´ tak, že L(M)=L(M´).

Nedeterministický TS

Nedeterministický TS (dále už jen nTS) se liší od tradičního TS tím, že v jednom okamžiku může existovat více než jeden proveditelný přechod pro aktuální dvojici stav/symbol. Jestliže nTS přejde do stavu, v němž pro aktuální symbol není proveditelný žádný přechod, stroj zruší výpočet. Jestliže je proveditelných více než jeden, stroj nedeterministicky vybere jeden z nich a pokračuje ve výpočtu zvoleným přechodem.

Definice

Turingův stroj $M=(S, \Sigma, \Gamma, \delta, q_0, q_k)$ se nazývá nedeterministický, jestliže δ má tvar:

$$\boldsymbol{\delta}: (Q \setminus \{q_F\}) \times \Gamma \rightarrow 2^{Q \times (\Gamma \cup \{L,R\})}$$

nTS je zobecněním TS a proto každý jazyk přijímaný tradičním TS, je přijímán také nTS. nTS nejsou schopny přijímat více jazyků než deterministické .

Ke každému nTS M existuje deterministický TS D tak, že L(M)=L(D).

Rozpoznávací schopnosti TS nelze zvýšit ani přidáním dalších pásek ani zavedením nedeterm. chování. Tento závěr podporuje Turingovu tezi, že třída jazyků přijímaných TS přestavuje vrchol hierarchie strojově rozpoznatelných jazyků.

Univerzální TS

Je to programovatelný TS, který na základě svého programu může simulovat jiný TS. Je tedy abstraktním předchůdcem dnešních programovatelných počítačů, které přijímají a vykonávají program, uložený v jejich paměti.

uTS umožňuje ve tvaru vstupního řetězce specifikovat konkrétní Turingův stroj i data, nad nimiž má tento konkrétní stroj pracovat.

Univerzální Turingův stroj, který zpracuje toto zadání může být navržen jako 3-páskový TS takto:

1. páska - program + data (+ výstupní data)
2. páska - pracovní páska
3. páska - registruje stav stroje, který je simulován (stav programu)

K takovému stroji můžeme sestrojit ekvivalentní 1-páskový TS.

Problém zastavení

Problém zastavení je klasický nerozhodnutelný problém Turingových strojů.

Předpokládejme, že abecedy Turingova stroje jsou binární tj. $\Sigma=\{0,1\}$, $\Gamma=\{0,1,\Delta\}$ a označme kódovanou verzi Turingova stroje M symbolem $\sigma(M)$. $\sigma(M)$ je binární řetězec, na který může být aplikován Turingův stroj M.

Nyní def. jazyk L_F nad abecedou $\Sigma=\{0,1\}$ takto:

$$L_F=\{\sigma(M) \mid M \text{ je selfterminating}\}$$

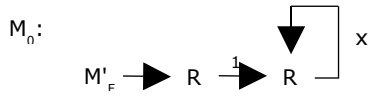
Problém zastavení je takto formulován jako problém rozhodnutelnosti jazyka L_F.

Jazyk L_F je nerozhodnutelný - důkaz sporem:

Předpokládejme tedy, že existuje Turingův stroj, označme ho jako M_F, který rozhoduje jazyk L_F.

Nyní modifikujeme stroj M_F tak, že modifikovaný Turingův stroj M'_F dává na výstup 0 resp. 1, právě když stroj M_F dává zprávu N resp. Y.

M'_F má tuto páskovou abecedu $\{0,1,\Delta\}$ a může být použit k vytvoření stroje M_0,



který se zastaví právě když M'_F dosáhne koncový stav s výstupem 0.

Nyní si položme otázku, zda je stroj M_0 selfterminating. Pokud je L_F rozhodnutelný jazyk, pak existuje odpověď ANO, nebo NE a žádná jiná možnost.

- **a)** Předpokládejme že M_0 je self-terminating. Pak pro vstup $\sigma(M)$ se stroj M'_F zastaví s výstupem 1 a stroj M_0 přejde po hraně R -1-> R a nezastaví se. Tedy "M_0 je selfterminating => M_0 je nonselfterminating".
- **b)** Předpokládejme že M_0 je nonselfterminating. Pak pro vstup $\sigma(M)$ se stroj M'_F zastaví s výstupem 0 a stroj M_0 se zastaví. Tedy "M_0 je nonselfterminating => M_0 je selfterminating".

Dospěli jsme ke sporu, z čehož vyplývá, že jazyk L_F a problém zastavení Turingova stroje je nerozhodnutelný.

Turing-Churchova teze

Turingova teze: výpočetní mocnost TS zahrnuje výpočetní mocnost jakéhokoliv výpočetního systému.

Churchova teze: parciálně rekurzivní funkce obsahují všechny vyčíslitelné parciální funkce.

Souvislost TT a CHT

Obě teze jsou stejné. Nikdo nenalezl parciální funkci, která je vyčíslitelná a zároveň není parciálně rekurzivní. *Nástin důkazu:* musíme ukázat, že výpočetní síla TS je omezena na možnost vyčíslovat parciálně rekurzivní funkce (každý proces prováděný TS je procesem vyčíslení nějaké parciálně rekurzivní funkce)

TS jako akceptor jazyka, rozhodnutelnost jazyka

Definice jazyka přijímaného Turingovým strojem M

Řetězec $w \in \Sigma^*$ je přijat Turingovým strojem $M=(S, \Sigma, \Gamma, \delta, q_0, q_F)$, jestliže se při aktivaci M z počáteční konfigurace pásky: $\underline{\Delta}w\Delta\Delta...$ a počátečního stavu q_0 stroj M zastaví (přejde do stavu q_F).

Množinu $L(M)=\{w \mid w$ je přijat T. strojem M$\}$ nazýváme jazyk přijímaný T. strojem M.

Zatím jsme definovali přijímání řetězce TS způsobem obvyklým i pro ostatní automaty. Někdy je vhodné, aby TS na pásku zapsal informaci o tom, že řetězec přijal, předtím než zastaví. Můžeme například přijmout konvenci, kdy stroj oznámí přijetí vstupního řetězce právě tím, že zastaví s páskovou konfigurací $\Delta\_Y\Delta\Delta\Delta$ .., kde symbol Y reprezentuje kladnou odpověď. Podle této konvergence však zastavení s jiným obsahem pásky značí nepřijetí řetězce.

Jazyky přijímané TS versus rozhodnutelné jazyky :

Schopnost Turingových strojů přijmout jazyk není symetrická se schopností nepřijmout jeho komplement. S pomocí univerzálního Turingova stroje lze ukázat tento fakt na příkladě jazyka L_0. Zkonstruujeme Turingův stroj, který přijímá komplement jazyka L_0:

Vytvoříme stroj M'_w který pro každý řetězec $w \in \Sigma^*$:

1. Vytvoří binární číslo $\approx^{-1}(\rho(w))$, tj. kód automatu M_w
2. Zřetězí tento kód s kódem řetězce w.

Pak vytvoříme složení T.strojů$\rightarrow M'_w \rightarrow T_\cup$. Výsledný stroj přijímá jazyk L_1

$$L_1=\{w \mid M_w \text{ přijímá řetězec } w\}=\text{neg } L_0.$$

To tedy znamená, že existují jazyky přijímané Turingovým strojem, pro které nelze sestrojit Turingův stroj, který končí zprávou „Y“ pro lib. $w \in L$ a zprávou „N“ pro vš. $w \notin L$. Tyto jazyky se nazývají jazyky přijatelné Turingovým strojem. Jazyky pro které lze takový Turingův stroj sestrojit se nazývají jazyky rozhodnutelné.

Příklady:

1. Jazyk L_1 je přijatelný Turingovým strojem, ne však rozhodnutelný
2. Bezkontextové jazyky jsou rozhodnutelné jazyky
3. Jazyk $L=\{x^n y^n z^n | n \in \mathbf{N}\}$ je rozhodnutelný

Jazyky přijímané TS jsou totožné s jazyky generovanými neomezenými gramatikami. V jiném kontextu jsou často jazyky přijímané TS označované jako rekurzívně vyčíslitelné jazyky. (nebo rekurzívní jazyky).

Vyčíslitelnost, primitivní rekurzivní funkce, parciální rekurzivní funkce, vztah funkce - TS

Vyčíslitelnost

Z praktického hlediska je nutné vymezit třídu všech funkcí, která obsahuje všechny vyčíslitelné funkce, tj. funkce, které lze vypočítat podle nějakého algoritmu bez ohledu na to, jak je tento algoritmus vyjádřen či implementován. Všechny tyto funkce lze vypočítat pomocí Turingova stroje.

Primitivní rekurzivní funkce

Vytvářejí se z jednoduchých funkcí a to 3-mi způsoby:

1) kombinace

Kombinací dvou funkcí $f: N^k \to N^m$ a $g: N^k \to N^n$ získáme funkci $f \times g : N^k \to N^{m+n}$ pro kterou $f \times g(\bar{x}) = (f(\bar{x}), g(\bar{x}))$

2) kompozice

Kompozicí dvou funkcí $f: N^k \to N^m$ a $g: N^k \to N^n$ je funkce $g \circ f : N^k \to N^n$ pro kterou

$$g \circ f(\bar{x}) = g(f(\bar{x}))$$

3) primitivní rekurze

Umožňuje vytvořit funkci $f: N^{k+1} \to N^m$ na základě jiných dvou funkcí g a h; $g: N^k \to N^m$, $h: N^{k+m+1} \to N^m$

rovnicemi
$$f(\bar{x},0) = g(\bar{x})$$
$$f(\bar{x}, y+1) = h(\bar{x}, y, f(\bar{x}, y))$$

Parciální rekurzivní funkce

K rozšíření třídy vyčíslitelných funkcí za totální vyčíslitelné funkce zavedeme techniku známou pod názvem *minimalizace*. Tato technika umožňuje vytvořit funkci f: $N^n \rightarrow N$ z jiné funkce g: $N^{k+1} \rightarrow N$ předpisem, v němž $f(\bar{x})$ je nejmenší y takové, že

(1) $g(\bar{x}, y) = 0$

(2) $g(\bar{x}, z)$ je definována pro všechny z<y, z∈**N**

Tuto konstrukci zapisujeme notací:

$$f(\bar{x}) = \mu y[g(\bar{x}, y) = 0]$$

Funkce definovaná minimalizací je skutečně vyčíslitelná.

Definice

Třída parciálně rekurzivních funkcí je třída parciálních funkcí, které mohou být vytvořeny z počátečních funkcí aplikací, kombinací, kompozicí, primitivní rekurzí a minimalizací.

Vztah funkce - TS

Turingův stroj M=(S, Σ, Γ, δ, q_0, q_F) vyčísluje parciální funkci $f: \Sigma^{*m} \rightarrow \Sigma_1^{*n}$ ($\Sigma_1 \subseteq \Gamma$, $\Delta \notin \Sigma_1$), jestliže pro každé $(w_1,w_2,...,w_m) \in \Sigma^{*m}$ a odpovídající počáteční konfiguraci $\Delta w_1 \Delta w_2 \Delta ... \Delta w_m \Delta\Delta\Delta$ stroj M

1. v případě, že $f(w_1,\ldots,w_m)$ je definována, pak M zastaví a páska obsahuje $\Delta v_1 \Delta v_2 \Delta \ldots \Delta v_n \Delta\Delta\Delta$, kde

   $(v_1,\ldots,v_n)=f(w_1,\ldots,w_m)$.

2. v případě, že $f(w_1,\ldots,w_m)$ není definována, M cykluje (nikdy nezastaví) nebo zastaví abnormálně.

Parciální funkce, kterou může počítat nějaký Turingův stroj se nazývá funkcí Turingovsky vyčíslitelnou.

Věta : Každá parciálně rekurzívní funkce je Turingovsky vyčíslitelná.

Věta : Každý výpočetní proces prováděný Turingovým strojem je procesem vyčíslení nějaké parciálně rekurzivní funkce.

Funkce nevyčíslitelná TS

f: {0, 1}* definovaná:

f(w)={1 jestliže w=ρ(M) pro samoukončující stroj M; 0 v ostatních případech}.

Protože její výpočet odpovídá vyřešení problému zastavení, je tato funkce nevyčíslitelná.

Algoritmická složitost, složitost problému, hierarchie tříd složitosti, P a NP problémy

Měření složitosti:

Jedním z prostředků, které se v tomto kontextu často používají je čas. Množství času potřebné k provedení výpočtu označujeme jako *časovou složitost výpočtu*. Množství paměťového prostoru se označuje jako *paměťová složitost*.

Složitost výpočtů na TS:

Provedení jednoho přechodu TS budeme považovat za jeden krok výpočtu a časovou složitost výpočtu TS definujeme jako počet kroků provedených během výpočtu.

Složitost algoritmů:

Každý TS můžeme považovat za implementaci jednoduchého algoritmu, který je reprezentovaný ve formě přechodového diagramu stroje. Při určování složitosti algoritmu existují různé možnosti v intervalu od nejhoršího k nejlepšímu případu. Pro účely formální definice se obvykle používá pesimistický odhad a časová složitost algoritmu se definuje jako nejhorší možný případ činnosti algoritmu.

Průměrná složitost: se vypočítá jako součet násobků složitostí všech možných výpočtů a pravděpodobností, že k těmto výpočtům dojde:

$$\sum_{i=1}^{n} p_i c_i$$

Složitost problému:

Nelze problém označit za složitý, když má složité řešení. Složité řešení lze v zásadě najít pro každý problém. Problém označíme jako složitý tehdy, když nemá žádné jednoduché řešení. Vzhledem k množství řešení jediného problému je nalezení nejjednoduššího řešení velmi obtížné. Ve skutečnosti je mnoho problémů, které nemají nejjednodušší řešení.

Problém srovnávání řetězců:

Hledání nejlepšího řešení se stává nekonečným procesem vytváření stále lepších řešení. Zkoumejme tento jev na příkladu dvou řetězců z {x, y, z} stejné délky pomocí TS.
Vidíme, že pokus najít nejlepší řešení problému porovnání dvou řetězců vede k nekonečné

posloupnosti řešení, v níž je následující řešení vždy lepší než předchozí. *Blumova věta* říká, že některé problémy vůbec nemají nejjednodušší řešení.
V jiných případech se nalezené řešení může jevit jako příliš složité. Potom musíme nalézt lepší řešení nebo dokázat, že žádné lepší řešení neexistuje. Je možné dokázat, že každé řešení problému dvou řetězců pomocí TS má nejméně kvadratickou složitost vzhledem k délce srovnávání řetězců. Ukázalo se, že rozlišování klasifikace podle tříd funkcí je užitečným prostředkem určování složitosti problémů.

Hierarchie tříd složitosti

Nechť **F** je množina funkcí z N do N. Pro danou Funkci f z **F** definujeme množiny funkcí $O(f)$, $\Theta(f)$ a $\Omega(f)$ takto:

- $O(f(n))=\{g(n)$: existují kladné konstanty c a n_0 takové že $0 \leq g(n) \leq c.f(n)$ pro všechna $n \geq n_0\}$
- $\Theta(f(n))=\{g(n)$: existují kladné konstanty c_1, c_2 a n_0 takové že $0 \leq c_1.f(n) \leq g(n) \leq c_2 .f(n)$ pro všechna $n \geq n_0\}$
- $\Omega(f(n))=\{g(n)$: existují kladné konstanty c a n_0 takové že $0 \leq c.f(n) \leq g(n)$ pro všechna $n \geq n_0\}$

kde g(n) je funkce z **F**.

Množinu $O(f(n))$ nazýváme asymptotickým horním ohraničením funkce f(n).
Množinu $\Theta(f(n))$ nazýváme asymptotickým (oboustranným) ohraničením funkce f(n).
Množinu $\Omega(f(n))$ nazýváme asymptotickým dolním ohraničením funkce f(n).

Věta: Pro libovolné dvě funkce f(n) a g(n) z **F** je $f(n) \in \Theta(g(n))$ právě když $f(n) \in O(g(n))$ a $f(n) \in \Omega(g(n))$.
Věta: Nechť p(n) je polynom stupně d. Pak $|p(n)| \in \Theta(n^d)$.

Řády složitosti poskytují klasifikační schéma, které je dostatečně hrubé, aby překrylo nepřesnosti, kterých se dopouštíme při určování složitosti problému. Časovou složitost problému zařadíme do třídy $\Theta(f)$, jestliže lze problém řešit algoritmem se složitostí f a každé lepší řešení má také časovou složitost ve třídě $\Theta(f)$.

13. Základní pojmy teorie fuzzy množin

Pojem fuzzy množin byl popsán A.L. Zadehem jako práce s daty a informací vykazujících nestatistickou neurčitost.
Fuzzy množiny jsou zobecněním pojmů množina a náležení do množiny. Vyjadřují neurčitost a mnohoznačnost prostředí a jevů kolem nás a rozdílnost subjektivního nahlížení či hodnocení tohoto prostředí a jevů. Tato neurčitost kolem nás může být popsána jako statistická neurčitost nebo nestatistická tj. fuzzy neurčitost. Svou podstatou je fuzzy způsob vhodným pro popis atributů objektů i míry náležení do některé ze tříd při řešení problémů klasifikace. Některé z atributů nelze popsat jasně danými numerickými hodnotami, ale pouze neurčitými a víceznačnými jazykovými obraty používanými v každodenním životě (např. asi tak, velmi mnoho, více méně, přibližně apod.).

Konvenční množina (např. $H=\{x;6\leq x\leq 8\} \subset R$) a náležení do ní mohou být popsány charakteristickou funkcí m_H

$m_H : R\rightarrow\{0,1\}$

která svými hodnotami pro libovolné $x\in R$ určí jednoznačně, zda x náleží do H (m_H=1) nebo nenáleží (m_H=0).

V případě jiné podmnožiny $H_f \subset R$, která obsahuje všechna čísla blízká číslu 7 už nelze použít předchozí charakteristickou funkci. Náležení x do množiny H_f není v tomto případě tak jednoznačné. Míra náležení tohoto čísla do množiny H_f je dána mírou vzdálenosti čísla x od čísla 7. Čím dále bude číslo x od čísla 7, tím menší bude jeho míra náležení do množiny H_f.
Charakteristická funkce m_H je v tomto případě nahrazena tzv. membership funkcí μ_{Hf}, neboli funkcí náležení do dané fuzzy množiny.

$\mu_{Hf} : R\rightarrow\{0,1\}$

Pro daný příklad může mít funkce μ_{Hf} např. následující tvar:

$$\mu_{Hf}(x)\equiv\begin{Bmatrix}\frac{x-5}{2}\ pro & 5\leq x\leq 7\\ \frac{9-x}{2}\ pro & 7\leq x\leq 9...\\ 0\end{Bmatrix}\mu_{Hf}(x)\equiv\begin{Bmatrix}\frac{x-5}{2}\ pro & 5\leq x\leq 7\\ \frac{9-x}{2}\ pro & 7\leq x\leq 9...\\ 0\end{Bmatrix}$$

Náležení objektu do dané fuzzy množiny je tedy dáno hodnotou funkce náležení dané množiny pro daný objekt. Tato hodnota **neurčuje** pravděpodobnost, náležení objektu do množiny (např. pravděpodobnost, že x je číslo 7), ale míru podobnosti s ostatními objekty, které nabývají vlastnosti dané neurčitě zkoumanou fuzzy množinou.
Funkce náležení získáváme z dat, která rozdělujeme do fuzzy podmnožin oboru hodnot pro daný atribut. Vycházíme často z funkce pravděpodobnostního rozložení (hustoty) pro danou vlastnost.

Obor hodnot každého vstupního atributu lze rozdělit do několik překrývajících se fuzzy podmnožin (dále jen fuzzy množin) daného oboru a dotazovanou hodnotu vstupního atributu vyjádřit ve formě číselných hodnot funkcí náležení jednotlivých fuzzy množin (dále jen hodnoty náležení), které určují míru náležení hodnoty atributu do jednotlivých fuzzy množin. Hodnoty vstupních atributů, které mají různý význam a interpretaci je tak možné převést na vektor reálných hodnot stejně interpretovatelných pro všechny různorodé atributy. Vstupní atributy, které popisují danou profesi, rozděluji podle typu fuzzy, které nad nimi definuji, do dvou skupin:

1. Fuzzy daná podobností
2. Fuzzy daná kontextem

ad 1) Fuzzy daná podobností
V první skupině jsou atributy, jejichž hodnoty náležení, představující subjektivně danou míru náležení zkoumaného objektu (zpovídaného reprezentanta, nebo uživatele) jsou určeny na základě podobnosti typického představitele konceptu, který představuje atribut, a zkoumaného objektu. Tento přístup vysvětlují teorie prototypů nebo teorie příkladů. První ukazuje, že lidé zkoumají, který z abstraktních subjektivních příkladů daného konceptu je lepší představitel než ostatní a podle tohoto prototypu, sumarizujícího v sobě všechny významné vlastnosti konceptu, potom určují svou míru náležení k danému konceptu. Druhá teorie popisuje určení náležení jako proces přímého srovnávání s více příklady daného konceptu.

Příkladem mohou být otázky:

- Má mít Vaše práce organizační charakter ?
- Chcete pracovat na vedoucím místě ?
- Má mít vaše práce tvůrčí charakter ?

V těchto případech dotazovaný určuje svou míru příslušnosti do konceptu lidí, kteří něco organizují, pracují jako vedoucí popř. něco vytvářejí. Hledané prototypy popř. mohou být abstraktně vymyšlené nebo konkrétní známí představitele. Pro lepší komunikaci nejsou odpovědi na tyto otázky požadovány ve formě čísel vyjadřující míru náležení do daného konceptu (fuzzy množiny), ale mohou být opět fuzzy podmnožinami (lingvistickými hodnotami) s pevně daným tvarem, které tak s dané fuzzy množiny (např. lidí, kteří něco organizují) vytvářejí fuzzy množinu vyššího stupně.

Jelikož výstupem musí být nakonec číselné hodnoty náležení z intervalu [0,1], je nutno převést každou odpověď z lingvistické hodnoty na číselnou pro každou určující míru náležení dané odpovědi k ostatním odpovědím. Následují příklady převodu pro odpovědi *Určitě ne* a *Možná ano*.

$$Urit\ ne \equiv \left\{ \frac{0.95}{\mu_{Urit\ ne}}, \frac{0.0}{\mu_{Spe\ ne}}, \frac{0.0}{\mu_{Mon\ ne}}, \frac{0.0}{\mu_{Mon\ ano}}, \right.$$

$$Urit\ ne \equiv \left\{ \frac{0.95}{\mu_{Urit\ ne}}, \frac{0.0}{\mu_{Spe\ ne}}, \frac{0.0}{\mu_{Mon\ ne}}, \frac{0.0}{\mu_{Mon\ ano}}, \frac{0.0}{\mu_{Spe\ ano}}, \frac{0.0}{\mu_{Urit\ ano}} \right\} \Big| \{\ Urit\ ne,\ Spe\ ne,\ Mon\ ne,\ Mon\ ano,\ Spe\ ano,\ Urit\ ano\ \}$$

$$Mon\ ano \equiv \left\{ \frac{0.0}{\mu_{Urit\ ne}}, \frac{0.0}{\mu_{Spe\ ne}}, \frac{0.3}{\mu_{Mon\ ne}}, \frac{0.9}{\mu_{Mon\ ano}}, \right.$$

$$Mon\ ano \equiv \left\{ \frac{0.0}{\mu_{Urit\ ne}}, \frac{0.0}{\mu_{Spe\ ne}}, \frac{0.3}{\mu_{Mon\ ne}}, \frac{0.9}{\mu_{Mon\ ano}}, \frac{0.3}{\mu_{Spe\ ano}}, \frac{0.0}{\mu_{Urit\ ano}} \right\} \Big| \{\ Urit\ ne,\ Spe\ ne,\ Mon\ ne,\ Mon\ ano,\ Spe\ ano,\ Urit\ ano\ \}$$

Zvláštní místo zaujímá možná odpověď **nevím**, která též musí být převedena na číselné vyjádření. Tato odpověď ponechává stejně velké šance na "všech stranách" a proto přiřazuje pro všechny fuzzy podmnožiny odpovědí hodnotu 0.5 jako poloviční v intervalu hodnot libovolné funkce náležení [0,1]. Tato odpověď však musí být v systému i nadále vedena jako neznámá a v případě většího výskytu těchto odpovědí pracovat s koeficientem důvěry v odpovědi na otázku v daném atributu.

ad 2) Fuzzy daná kontextem
Právě u fuzzy tohoto typu je důležitý kontext, ve kterém je prováděno hodnocení a přiřazení míry náležení do daného konceptu. K tomuto typu můžeme přiřadit atributy typu délka pracovní doby, flexibilita pracovní doby, výše platu apod. U těchto atributů je třeba zajistit vhodnou formulací otázky správné pochopení kontextu daného atributu učiteli systému i uživateli. Další, v čem se tyto atributy liší od prvního typu, je nutnost určení jejich fuzzy struktury ve "škálovacím" preprocesu s pomocí učitelů - reprezentantů profesí. Metoda zvolená pro konstrukci fuzzy množin určí i případné zapojení vzorku uživatelů do tohoto "škálovacího" procesu. Metoda je v tomto případě dána interpretací fuzzy, která je brána buď pouze jako výsledek překrývajících se uvažovaných fuzzy množin nebo i se zapojením nejednoznačnosti vyplývající z různých pohledů pozorovatelů (učitelů a uživatelů). Atributy tohoto typu mohou obsahovat i neurčité číselné informace typu *asi 5, více než 3000, něco mezi 8000 a 9500*.

Při "škálovacím" procesu je třeba určit strukturu neboli fuzzy množinu daných atributů vhodnými otázkami pro reprezentanty profesí. Příkladem mohou být následující otázky:

- Od kolika hodin práce denně je podle Vás dlouhá pracovní doba ?
- Kolik hodin práce denně podle Vás je krátká pracovní doba ?
- Od jaké částky uvažujete o vysokém ročním platu ?
- Kolik procent Vaší práce podle Vás zabírá práce s počítačem, jestliže se domníváte, že pracujete málo s počítačem ?

Odpověď na tyto otázky umožní konstrukci fuzzy množin představující základní lingvistické hodnoty pro odpovědi na příslušné otázky.

- Jak dlouhá by měla být Vaše pracovní doba ?
- Jaký by měl být Váš roční plat ?
- Jakou část Vaší práce by měla zaujímat práce s počítačem ?

Odpovědí na tyto otázky mohou být opět lingvistické hodnoty, které jsou obecně shodné pro všechny tyto atributy a liší se pouze ve formulaci pro každý jednotlivý atribut.
Pomocí aplikace vhodných matematických operátoru je dosaženo vyjádření hodnoty náležení pro stupně jednotlivých lingvistických hodnot, číselné neurčitosti i případné negace. Toto vyjadřují spojení *více méně malý, velmi velký, asi tak 2600, více než 5*

Na základě takto obecně definovaných jazykových hodnot sestavím možné odpovědi pro konkrétní atributy.

- Jak dlouhá by měla být Vaše pracovní doba ? (*asi tak 8.5 hodiny, mezi 8 a 10 hodinami, spíše kratší*)
- Jaký by měl být Váš roční plat ? (*od 120 tis., mezi 180 a 250 tis., ne menší než 150 tis., hodně vysoký*)
- Jakou část Vaší práce by měla zaujímat práce s počítačem (*spíše menší, vůbec žádnou, velkou*)

Všechny tyto odpovědi je možné s použitím sestavených fuzzy množin převést na číselné hodnoty náležení a dále s nimi pracovat.

Výsledkem zodpovězených dotazů reprezentanty nebo uživateli jsou vektory hodnot, určujících míry náležení do příslušných 5 (popř. 3) fuzzy množin v případě atributů 1. typu (popř. 2. typu) pro každý atribut. Tyto vektory jsou vstupními daty do systému pro účely učení v případě učící fáze, ve které dané vektory vyplnili reprezentanti jednotlivých profesí, a pro účely klasifikace v případě použití systému uživateli.