# Programátorská dokumentace k ročníkovému projektu

## Vedoucí projektu: Barbora Vidová-Hladká

## Autor: Vladimír Rovenský

## Téma: Rozpoznání smysluplnosti české věty

**Platformy: Unix, Windows**

**Programovací jazyk: C++ (IDE MS Visual Studio)**

# Programátorská dokumentace

### Popis programu

Následuje popis struktury celého programu, tříd, metod, atd. Tato část dokumentace je z velké části obsažena i v komentářích přímo v programu, zde bude jen základní popis.

## Struktura programu

Základ programu tvoří třída *Sense* deklarovaná v souboru *sense.h*, která pokrývá veškerou funkčnost. Stará se o načtení a rozdělení věty, její uložení do připravených struktur a uchovávání globální konfigurace.

Třída *Tester* obstarává kontrolu platnosti podmínek a jejich načítání i uchovávání ve vhodných strukturách. Instance třídy *Tester* je obsažena v třídě *Sense*, která jí posílá dvojice slovních jednotek ke kontrole platnosti podmínek. Je deklarovaná v souboru *tester.h.*

Grafové operace provádí třída *Graph (*soubor *graph.h*), jedná se o graf realizovaný seznamem následníků, poskytuje metody pro kontrolu spojitosti.

Poslední z podstatných tříd je *Token* (soubor *token.h*), která reprezentuje jednu slovní jednotku věty spolu s jejími morfologickými údaji.

V souboru *constants.h* jsou uloženy globální konstanty programu,v *exceptions.h* jsou deklarovány použité typy výjimek a ve *functions.h* jsou deklarace různých globálních funkcí použitých v programu.

V souboru *args.h* se nachází struktura určená k udržování nastavení programu a správě parametrů.

## Popis důležitých funkcí, typů a rozhraní

## Třída Sense

## Privátní data

* **std::vector<Sentence\*> bareSentences\_**Seznam jednoduchých vět (naplněn po rozdělení souvětí).
* **Typ Sentence**Definován jako std::vector<Token\*> - jedna věta.
* **Tester tester\_**  
  Instance třídy *Tester,* poskytující veškerou kontrolu podmínek.
* **bool verbose\_**Uchovává informaci o tom, zda má být poskytnut podrobný výstup.
* Args args\_;  
  Struktura uchovávající nastavení aplikace

**Privátní metody**

* **void freeData()**Uvolní naalokovanou paměť, zatím pouze volá *freeSentences().*
* **void freeSentences()**Uvolní naalokované věty (tokeny).
* **void readSentence(std::ifstream& from,std::vector<Token\*>& to)**Načte větu z *from* do *to.*
* **bool endOfSentence(const std::string& line)const**Vrací true, pokud *line* je řádek CSTS výstupu nástroje *tool\_chain* indikující konec věty.
* **Token\* getToken(const std::string& line,int tokenNumber)const**Z řádky *line* CSTS výstupu nástroje *tool\_chain* vyrobí jeden token.
* **int checkMark(unsigned int from,const std::string& line,const std::string& mark)const**V řádce CSTS výstupu *line* hledá od pozice *from* značku *mark*, vrací index prvního nemezerového znaku po této značce, případně –1, pokud značku nenajde.
* **void splitSentence(std::vector<Token\*>& tokens)**Rozdělí větu na věty jednoduché.
* **bool checkSentence(int i)const**Otestuje, zda je jednoduchá věta číslo *num* smysluplná. Zkonstruuje graf, přidá do něj hrany pomocí *checkConjunctions()* a *objektu* tester\_*,* aplikuje podmínky (3) pomocí *setMaxConditions()* a zkontroluje souvislost grafu.
* **void checkConjunctions(Sentence\* sentence,Graph& g)const**Projde větu a pokud narazí na spojku nebo čárku, pokusí se najít dvě slovní jednotky (podle definovaných podmínek), které by mohla spojovat. Hledá tak, že od spojky postupuje nalevo a pro každou slovní jednotku se pokusí napravo od spojky najít další, která by vyhovovala některé z definovaných podmínek(2). První takto nalezený pár je zanesen do grafu a hledá se další spojka (čárka).
* **void setMaxConditions(Graph& g,const int num)const**Zanese do grafu věty číslo *num* informaci o podmínkách typu (3) – omezení počtu hran pro DFS. V podstatě označí hrany grafu podobným způsobem jako u ilustračního obrázku výše, akorát místo barev používá k odlišení skupinek hran čísla.
* **void transformSentence(Sentence\* sentence)**Projde větu a udělá různé drobné úpravy zjednodušující analýzu.
* bool readInput(std::istream& fin)Načte ze vstupního proudu *fin* větu a rozdělí ji do příslušných struktur.
* void clearUnknown()Pokud uživatel zadal volbu *–g,* provede vymazání nerozeznaných slovních jednotek ve větě (až do limitu, který stanovil uživatel).

**Veřejné metody**

* **bool check()const**Zkontroluje smysluplnost načtené věty.
* void run()  
  Hlavní metoda aplikace, načte věty ze vstupu a zkontroluje jejich smysluplnost.

## Třída Tester

## Privátní data

* **posCouple**Typ definován jako *std::pair<char,char>* - dvojice slovních druhů.
* **CondList**Typ pro uložení podmínek, definován jako *std::map<posCouple,std::vector<Condition> >,* jedná se o asociativní pole dvojice slovních druhů -> seznam podmínek, za kterých je tato dvojice smysluplná. Stačí, aby platila jediná z podmínek.
* **ConjCondList**Typ pro ukládání podmínek pro spojky a čárky, definován jako *std::map<posCouple,std::vector<ConjunctionCondition> >,* opět asociativní pole dvojice slovních druhů -> seznam podmínek.
* **MaxCondList**Typ pro uchovávání podmínek (3), definován jako *std::map<posCouple,int>,* asociativní pole dvojice slovních druhů -> maximální počet relací. V tomto případě je dvojice slovních druhů nesymetrická.
* **CheckingFunction**Typ pro funkci ověřující nějakou podmínku mezi dvěma slovními jednotkami. Definován jako *bool (Tester::\*checkingFunction)(const Condition\*,const Token&,const Token&)const.* Tedy ukazatel na funkci přijimající podmínku a dva kontrolované tokeny, vracející true pokud tokeny podmínce vyhovují, false pokud ne.
* **std::vector<checkingFunction>checkingFunctions\_**Seznam kontrolních funkcí, které budou použity při ověřování platnosti podmínek.

## Privátní metody

* **bool checkCondition(const Token& t1,const Token& t2)const**Projde všechny dostupné podmínky (1) pro tokeny *t1 a t2* a pokud je alespoň jedna z nich splněna, vrátí true, jinak false.
* **bool checkValues(string val1,string val2,std::vector<std::string>****cond1,vector<string> cond2,int tagNum=-1)const**Porovná hodnoty *val1,val2* (jedná se o hodnoty tagů / základní tvar atp) s hodnotami v podmínce (viz popis formátu souboru s podmínkami).
* **void readTags(std::istringstream& from,std::vector<char>& to)**Načte pomocí *getTags()* hodnoty tagů z výstupu nástroje *tool\_chain.*
* **void getTags(const std::string& from,std::vector<char>& to)**Parametr *from* je string obsahující hodnoty tagů z výstupu nástroje *tool\_chain*, tato funkce je načte do *to.*
* **void readLemmas(std::istringstream& from, std::vector<std::string>& to)**Stejná funkce jako *readTags()* pro základní tvar.
* **void getLemmas(const std::string& from,std::vector<std::string>& to)**Stejná funkce jako *getTags()* pro základní tvary.
* **bool checkTag(const std::string& c1,const std::string& c2,const****std::string& val1,const std::string& val2)const**Zkontroluje hodnoty jedné pozice tagu (obou kontrolovaných slovních jednotek) proti hodnotám v podmínce. V *c1,c2* jsou hodnoty dané pozice načtené z podmínek, ve *val1,val2* jsou hodnoty této pozice načtené ze slovních jednotek. Funkce tyto hodnoty porovná a vrátí true, pokud souhlasí, false pokud ne. Hodnoty z podmínek mohou obsahovat i znaky *-,+,(,)* (viz formát souboru podmínek).
* **bool checkConjunctionCondition(const ConjunctionCondition\* cond,const Token& t1,const Token& t2,const Token& conj)const**Zkontroluje zda tokeny *t1,t2* mohou být spojeny spojkou (čárkou) *conj.* V *cond* je jedna podmínka typu (2), kterou bude funkce kontrolovat.
* **void initCheckingFunctions()**Inicializace kontrolních funkcí - naplnění vektoru *checkingFunctions\_*. Při přidávání nových funkcí je třeba je ve této funkci do vektoru rovněž přiřadit.
* ***bool useCheckingFunctions(const Condition\* cond,const Token& t1,const Token& t2)const***Použije dostupné kontrolní funkce k ověření, zda tokeny *t1,t2* vyhovují podmínce *cond*. Vrací true, pokud všechny kontrolní funkce vrátí true, jinak false.
* **bool ccCheckTags(const Condition\* cond,const Token& t1,const Token&****t2)const**Kontrolní funkce, ověří, že hodnoty tagů v tokenech odpovídají hodnotám v podmínce.
* **bool ccCheckWordOrder(const Condition\* cond,const Token& t1,const****Token& t2)const**Kontrolní funkce, ověří, že slovosled tokenů odpovídá slovosledu vyžádanému v podmínce.
* **bool ccCheckLemmas(const Condition\* cond,const Token& t1,const Token& t2)const**Kontrolní funkce, ověří, že hodnoty základních tvarů v tokenech odpovídají základním tvarům v podmínce.
* **bool checkEq(const std::string& c,const std::string& val)const**Vrací true, pokud se parametry (znaky v pozicích tagů používaných nástrojem *tool\_chain*) buď rovnají, nebo pokud jeden je symbolem pro množinu obsahující druhý. Například Y,M atp. (viz dokumentace nástroje *tool\_chain)*

## 

## Veřejné metody

* **bool check(const Token& t1,const Token& t2)const**Podívá se na slovní druhy tokenů *t1,t2* a najde všechny odpovídající podmínky typu (1) pro tuto dvojici slovních druhů. Dále zkontroluje, zda alespoň jedna z nich pro tyto tokeny platí, pokud ano, vrátí true, pokud žádná neplatí (nebo pokud žádná podmínka pro tuto dvojici slovních druhů není definována), vrátí false. Implementováno pomocí privátní funkce *checkCondition()*
* **bool checkConjunction(const Token& t1,const Token& t2,const Token& conj)const**Podívá se na slovní druhy tokenů *t1,t2* a najde všechny odpovídající podmínky typu (2) pro tuto dvojici slovních druhů. Dále zkontroluje, zda alespoň jedna z nich pro tyto tokeny platí, pokud ano, vrátí true, pokud žádná neplatí (nebo pokud žádná podmínka pro tuto dvojici slovních druhů není definována), vrátí false.
* **bool getMaxCondition(posCouple c,int& max)const**Dostane dvojici slovních druhů a pokud je pro ně definována podmínka (3), vrátí true a do *max* uloží hodnotu z podmínky. Jinak vrací false.

## Struktura TagVal

Zastupuje jednu pozici jednoho tagu v podmínce – de facto jeden řádek v definici podmínky typu (1) určující přípustné hodnoty na dané pozici tagu.

* **int tagNumber**Číslo pozice, které se podmínka týká.
* **std::vector<char> firstValue,secondValue**Seznamy přípustných hodnot na této pozici tagu pro první, respektive druhou kontrolovanou slovní jednotku.
* **bool neg**True pokud je tento řádek negován (symbol ! v definici podmínky, viz formát souboru podmínek).
* **void clear()**Vyprázdní strukturu (smaže oba vektory, *neg* nastaví na false)-

## Struktura ConditionSkeleton

Základní třída reprezentující klauzule podmínky soplečné pro typy podmínek (1) a (2).

## Struktura BasicCondition

Odvozena od *ConditionSkeleton*, reprezentuje podmínku typu (1).

## Struktura ConjunctionCondition

Odvozena od *BasicCondition*, určená k definici podmínky typu (2), rozšířená o možnost odkázat se na základní tvar spojovací slovní jednotky.

* **std::string conjunction\_**Základní tvar spojovací slovní jednotky (řádek *tok* v definici podmínky (2)). Prázdné pokud na něm nezáleží.

## Soubor constants.h

Tento soubor obsahuje deklarace globálních konstant.

* **const int PARTOFSPEECH\_NUMBER = 10**Počet slovních druhů.
* **const std::string CONDITIONS\_FILENAME = "conditions.txt"**Jméno souboru s podmínkami.
* **const int TAGS\_NUMBER = 15**Počet pozic tagu na výstupu nástroje *tool\_chain*.
* **const int MAX\_ERASED\_SPLITTERS = 3**Program pro zjednodušení odstraňuje ze začátků vět slovní jednotky oddělující věty, např. čárku, spojky atp. Tato konstanta udává maximální počet takových slovní jednotek, které mohou být odstraněny. Délka se vztahuje na souvislý úsek slovních jednotek, mazání končí první neoddělující slovní jednotkou.

## Soubor exceptions.h

Obsahuje definice tříd určených pro výjimky programu.

* **WrongConditionFileE**Výjimka vyvolaná když program narazí na chybu v souboru s podmínkami při jeho načítání. Typicky nedodržení formátu souboru.
* **WrongInputFileE**Výjimka vyvolaná při neúspěšném parsování vstupu, nejspíše neplatný soubor CSTS.

## Soubor functions.h

Obsahuje různé globální funkce potřebné v programu, sémanticky nenáležející žádné z tříd.

* **bool isSpace(char c)**Definice symbolů, jež jsou chápány jako bílé znaky (oddělovače řetězců).
* **std::vector<std::string> stringify(const std::vector<char>& vect)**Převede vektor znaků na vektor řetězců délky jedna. Používá se pro zjednodušení porovnávání hodnot atributů slovních jednotek s podmínkami.
* **int posCharToNum(char pos)**převede značku pro slovní druh vyprodukovanou nástrojem *tool\_chain (*viz příslušná dokumentace) na číslo tohoto slovního druhu (podstatné jméno – 1, citoslovce – 10).
* **bool isSplitter(const Token& t)**definice řetězců, které mohou dělit souvětí na jednoduché věty, například spojky nebo některé typy zájmen (který, jež…)

## Třída Graph

Zastupuje graf ve formátu seznamu vrcholů a následníků. Má možnost kontroly souvislosti, při které zohledňuje i podmínky typu (3). Ty jsou do grafu zaneseny (při jeho konstrukci) označením příslušných hran. Označení hrany má tvar dvojice čísel – číslo skupiny, do které patří, a maximálního počtu hran, které z této skupiny lze vybrat. Kontrola souvislosti potom probíhá v několika krocích:

1. DFS průchod grafu zcela bez použití označených hran. Tím se graf rozpadne na několik komponent. Pokud je jediná, graf je souvislý. Jinak vzniklo více komponent, mezi kterými mohou být pouze označené hrany.
2. Je třeba vybrat podmnožinu těchto hran tak, aby pospojovala všechny komponenty a zároveň byly uspokojeny všechny podmínky (3) – označení hran.
   1. Optimalizace – nejprve se vyberou ty hrany, které spojují komponenty, které žádná jiná hrana nespojuje. Ty zřejmě musíme vybrat, jinak graf souvislý být nemůže.
   2. Hledání podmnožiny. Probíhá generováním řešení – postupně jsou vybírány možné hrany, spojovány komponenty až do chvíle, kdy jsou buď všechny sloučeny do jedné a řešení existuje – graf je souvislý, nebo není další možnost výběru hrany – došlo by k porušení některé z podmínek. Graf tedy souvislý není.

## Privátní data

* **std::vector<int> vertices\_**Seznam vrcholů.
* **std::map<int,std::vector<int> > neighbours\_**Seznam následníků, klíčem je číslo vrcholu, hodnotou seznam následníků.
* **VerboseStruct**Typ pro předávání informací z grafu za účelem poskytnutí podrobného výstpu (volba –v). Definován jako *std::pair<std::vector<std::pair<int,int> >,std::vector<int> >.* První hodnotou je seznam hran použitých při kontrole souvislosti grafu, druhou je mapa komponent (indexem je vrchol, hodnotou na tomto indexu číslo komponenty vrcholu), po DFS průchodu bez použití hran označených podmínkou (3).
* **EdgeGroups**Struktura udržující označení hran (podmínky (3)). Typ je definován jako std::map<std::pair<int,int>,std::vector<std::pair<int,int> > > tedy asociativní pole, jehož klíčem je hrana a hodnotou seznam skupin, do nichž hrana náleží (typicky jednoprvkový). Skupina = číslo skupiny + počet hran, které z této skupiny mohu použít.
* **EdgeLine**Seznam skupin pro jednu hranu. Definován jako std::pair<std::pair<int,int>,std::vector<std::pair<int,int> > > tedy dvojice hrana a seznam skupin, skupina = číslo skupiny a počet hran, které z ní lze vybrat.

#### Privátní metody

* **bool isMarked(int v1,int v2)const**Vrátí true, pokud je hrana *v1v2* označena nějakou podmínkou (3).
* **std::vector<std::vector<int> > dfsUnmarked()const**Provede DFS na neoznačených hranách grafu, vrátí mapu komponent po tomto DFS.
* **void useEdge(int nextEdge,const std::vector<edgeLine>& list,const****std::vector<int>& cMap, std::vector<edgeLine>&****newList,std::vector<int>& newcMap)const**Použije hranu při generování vhodné podmnožiny označených hran. Použít znamená sloučit komponenty spojené touto hranou do jedné (nová mapa komponent) a odpovídajícím způsobem upravit označení hran (některé spojují jiné komponenty než předtím). Parametry: nextEdge – hrana k použití, index do *list*, *list* – seznam hran a skupin, v nichž jsou, *cMap* – mapa komponent, *newList,newCmap* – nový seznam hran a mapa komponent, po použití hrany.
* **void optimizeEdgeList(std::vector<edgeLine>& list,std::vector<int>&****cMap)const**Provádí optimalizaci, viz krok 2a) v popisu třídy *Graph*.
* **bool recurseEdgeList(std::vector<edgeLine> list,std::vector<int>****cMap)const**Funkce hledající podmnožinu označených hran spojující všechny komponenty a vyhovující všem podmínkám (3). Dostane seznam hran a jejich skupin a mapu komponent, použije některou hranu, upraví si tyto dvě struktury a rekurzivně se zavolá znovu, dokud není nalezeno řešení, nebo vyčerpány možnosti.

**Veřejné metody**

* **void addVertex(int v)**Přidá vrchol v.
* **void addEdge(int v1,int v2)**Přidá hranu mezi v1 a v2.
* **bool checkConnectivity()const**Vrátí true, pokud je graf souvislý.
* **void markEdge(int v1,int v2,std::pair<int,int> group)**Označí hranu *v1v2* – přiřadí jí další skupinu. Skupinou se rozumí číslo skupiny a počet hran, které z ní můžu použít. Volá se při aplikování podmínky typu (3).  
  Příklad. Mějme slovní jednotku T a podmínku typu (3) říkající, že T se může navázat nejvýše na K sloves. Program najde všechna slovesa S1…Sn ve stejné větě s T a každou z n hran TSi označí stejným číslem skupiny a číslem K.

## Třída Token

Reprezentuje jednu slovní jednotku věty.

## Privátní data

* **std::string token\_**Vlastní slovní jednotka.
* **std::vector<char> tags\_**Seznam hodnot všech pozic tagů z výstupu nástroje *tool\_chain*.
* **int number\_**(Pořadové) číslo slovní jednotky ve větě.
* **std::string lemma\_**Základní slovní tvar (lemma) slovní jednotky.

##### Veřejné metody

* **int getPartOfSpeechNum()const**Vrací číslo slovního druhu slovní jednotky (1 – podstatné jméno, 10 – citoslovce).
* **char getPartOfSpeech()const**Vrací značku (jako tool\_chain) slovního druhu slovní jednotky.

## Graf struktury programu

Hrubý graf použití tříd definovaných v programu. Hlavní třídou je *Sense*, ten ke kontrole smysluplnosti *Tokenu* používá třídu *Tester.* Při kontrole je vytvořena instance třídy *Graph.*

Graph

Check()

Sense

Tester

Seznam Tokenů – věta

### Příklad spuštění programu

Mějme vstupní větu „Před domem stojí auto a v domě štěká pes.“ Tuto necháme zpracovat nástrojem *tool\_chain* a výstup uložíme do souboru *in.txt*. Dále spustíme program příkazem  
“*sense.exe –i in.txt*.“ Výstupem bude řetězec „OK“ signalizující smysluplnost věty. Pokud přidáme volbu *–v*, výstup bude následující (vysvětlivky modrou barvou):

Checking sentence: Kontrolovaná věta.

v domě štěká pes

Setting conjunction conditions: Aplikace podmínek (2) – zde nejsou

Setting relation conditions: Aplikace podmínek (1) – seznam smysluplných dvojic.

Relation: v---domě

Relation: v---štěká

Relation: domě---štěká

Relation: štěká---pes

Setting max number of relations conditions: Aplikace podmínek (3) – seznam označených hran.

Marking edge: v---štěká group: 1 max: 1

Marking edge: domě---v group: 2 max: 1

Marking edge: domě---štěká group: 3 max: 1

Marking edge: pes---štěká group: 4 max: 1

Checking connectivity: Kontrola souvislosti:

Optimization procedure found 4 edges. Kolik hran nalezla optimalizační procedura  
(Graph::**optimizeEdgeList())**.

Components after dfs not using marked edges: Komponenty po provedení DFS bez označených hran:

v---0

domě---1

štěká---2

pes---3

Marked edges used: Vygenerované řešení (vybrané označené hrany):

Used edge: domě---v

Used edge: v---štěká

Used edge: domě---štěká

Used edge: pes---štěká

Sentence OK Vyhodnocení jednoduché věty.

Checking sentence: Další kontrolovaná věta.

před domem stojí auto

Setting conjunction conditions:

Setting relation conditions:

Relation: před---stojí

Relation: domem---stojí

Relation: stojí---auto

Setting max number of relations conditions:

Marking edge: před---stojí group: 1 max: 1

Marking edge: domem---stojí group: 2 max: 1

Marking edge: auto---stojí group: 3 max: 1

Checking connectivity:

Optimization procedure found 3 edges.

Components after dfs not using marked edges:

před---0

domem---1

stojí---2

auto---3

Marked edges used:

Used edge: před---stojí

Used edge: domem---stojí

Used edge: auto---stojí

Sentence OK

OK Celkový výsledek (všechny jednoduché věty)

### Literatura

* Popis nástroje *tool\_chain*:  
   Barbora Vidová Hladká a kol., Český akademický korpus 2.0, v tisku <http://ufal.mff.cuni.cz/rest/CAC/cac_20.html>
* Popis morfologických značek:  
   http://ufal.mff.cuni.cz/rest/CAC/doc-cac20/cac-guide/cz/html/ch13.html