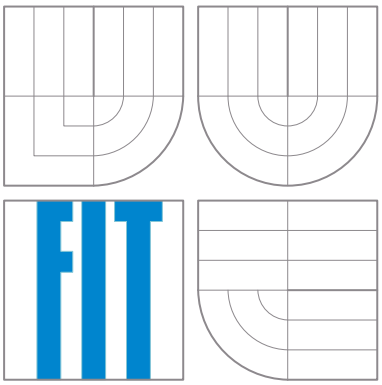


Dokumentace k projektu IFJ

# Interpret jazyka IFJ11

Implementace interpretu imperativního jazyka IFJ11
Tým 012, varianta b/2/II

Rozšíření: MUTREC, LOCALEXP, REPEAT, LENGTH, MODULO

11. prosince 2011

Autoři: Jan Bednařík, xbedna45, 25 % - vedoucí
Hynek Blaha, xblaha22, 25 %
Martin Janyš, xjanys00, 25 %
Ondřej Macháček, xmacha48, 25%

Fakulta Informačních Technologií
Vysoké Učení Technické v Brně

# Obsah

# 1 Úvod

Tento dokument popisuje práci čtyřčlenného týmu na implementaci interpretu jazyka IFJ11. Následující kapitoly jsou věnovány našemu přístupu k týmové spolupráci, komunikaci, řízení projektu a samotnému způsobu řešení klíčových fází, mezi něž patří lexikální, syntaktický a sémantický analyzátor a interpret. Dále je v dokumentu vysvětlen základní princip některých významných datových struktur, jako je tabulka symbolů či programový zásobník, a algoritmů z pohledu naší implementace.

Pro celistvost dokumentu jen stručně uveďme požadavky na výsledný program: Cílem projektu je vytvořit interpret, jenž ověří syntaktickou a sémantickou správnost libovolného programu zapsaného v jazyce IFJ11 a vykoná jej.

# 2 Řešitelský tým

## 2.1 Sestavení týmu

Při výběru členů týmu byl naší hlavní prioritou zodpovědný přístup k práci. Tým jsme se rozhdodli sestavit z pouhých čtyř členů, neboť jsme všichni prošli předmětem Jazyk C (IJC) a disponovali jsme tedy potřebnou programátorskou zkušeností. Dále nás pochopitelně motivoval i vyšší bodový zisk.

## 2.2 Komunikace a správa kódu

Před započetím vlastní práce je nezbytné jednoznačně stanovit komunikační prostředky, systém správy kódu a štábní kulturu programování, aby se tým mohl později plně soustředit pouze na vlastní řešení zadaného projektu.

Jako hlavní komunikační prostředek jsme zvolili chatovací službu Skype umožňující provozovat skupinové hlasové i video konference. Hlasovou online komunikaci považujeme za obrovskou výhodu, neboť jsme v době plného pracovního nasazení diskutovali prakticky denně a problémy jsme s využitím techniky brainstormingu řešili velmi efektivně a rychle. Nutno podotknout, že některé situace vedly k více než šestihodinovým skupinovým online hovorům. Videokonference potom nabízí možnost sdílení plochy počítače, čehož jsme využili pro vzájemnou grafickou demonstraci řešených problémů.

Při volbě nástroje pro verzování zdrojových kódů jsme vsadili na systém Subversion, jenž nám byl poskytnut na školním serveru Merlin. Díky systému SVN jsme byli schopni veškeré změny ihned distribuovat ostatním členům týmu a pohodlně spravovat případné kolize.

## 2.3 Spolupráce v týmu

Práci na jednotlivých částech projektu jsme se snažili rozdělit rovnoměrně a učinili jsme tak ihned po zveřejnění zadání. Jako nejrozumnější přístup se nám jevilo celý program rozčlenit na přibližně stejně náročné části[1], které jsme následně přibližně rovným dílem přerozdělili mezi všechny členy týmu. Každý řešitel však své nápady i kód pravidelně konzultoval se zbytkem týmu, kterýžto přístup nám pomohl odhalit množství chyb již v zárodku a především dobře posloužil k všeobecnému porozumění projektu jako celku.

[1] lexikální analyzátor, syntaktický analyzátor jazykových konstrukcí, syntaktický analyzátor výrazů, sémantické akce, interpret a algoritmy pro řazení a vyhledávání podřetězce v řetězci

# 3 Rozbor

Projekt je složen z několika logických celků, které vyžadují odlišný přístup řešení a které lze s předem daným rozhraním implementovat nezávisle. Poněvadž se jedná o obecně známá fakta, shrneme zde popis jednotlivých celků velmi stručně.

Lexikální analyzátor rozpoznává jednotlivé lexémy ve zdrojovém souboru a je implementován deterministickým konečným automatem.

Syntaktický analyzátor kontroluje, zda je vstupní program reprezentovaný řetězcem tokenů korektně syntakticky zapsán, volá sémantické akce a generuje tříadresný kód.

Sémantický analyzátor poskytuje sadu funkcí kontrolujících významovou správnost vstupního programu. Jedná se především o kontrolu deklarací proměnných a definic funkcí.

Generátor vnitřního kódu se obecně stará o tvorbu vnitřní reprezenatce programu ve formě instrukcí reprezentovaných tříadresným kódem.

Interpret prochází instrukční pásku a provádí vstupní program.

# 4 Návrh a popis řešení

## 4.1 Model konečného automatu lexikální analýzy

**Identifikátory v automatu**

| | |
|---|---|
| `INIT` | počáteční stav |
| `PNT` | point – stav řadové čárky |
| `FLT` | float – stav čísla s řadovou čárkou |
| `SING` | signum – stav znaménka před exponentem |
| `EXP` | exponent – stav čísla s exponentem |
| `ESC` | escape – stav, který čeká na escapovanou sekvenci |
| `ddd` | escaped number – stav, který přijímá pouze trojici čísel |
| `ID` | identificator – stav identifikátoru |
| `ostatní stavy` – jsou pojmenovány podle znaků, které si analyzátor rozpoznává, případně i vrací | |

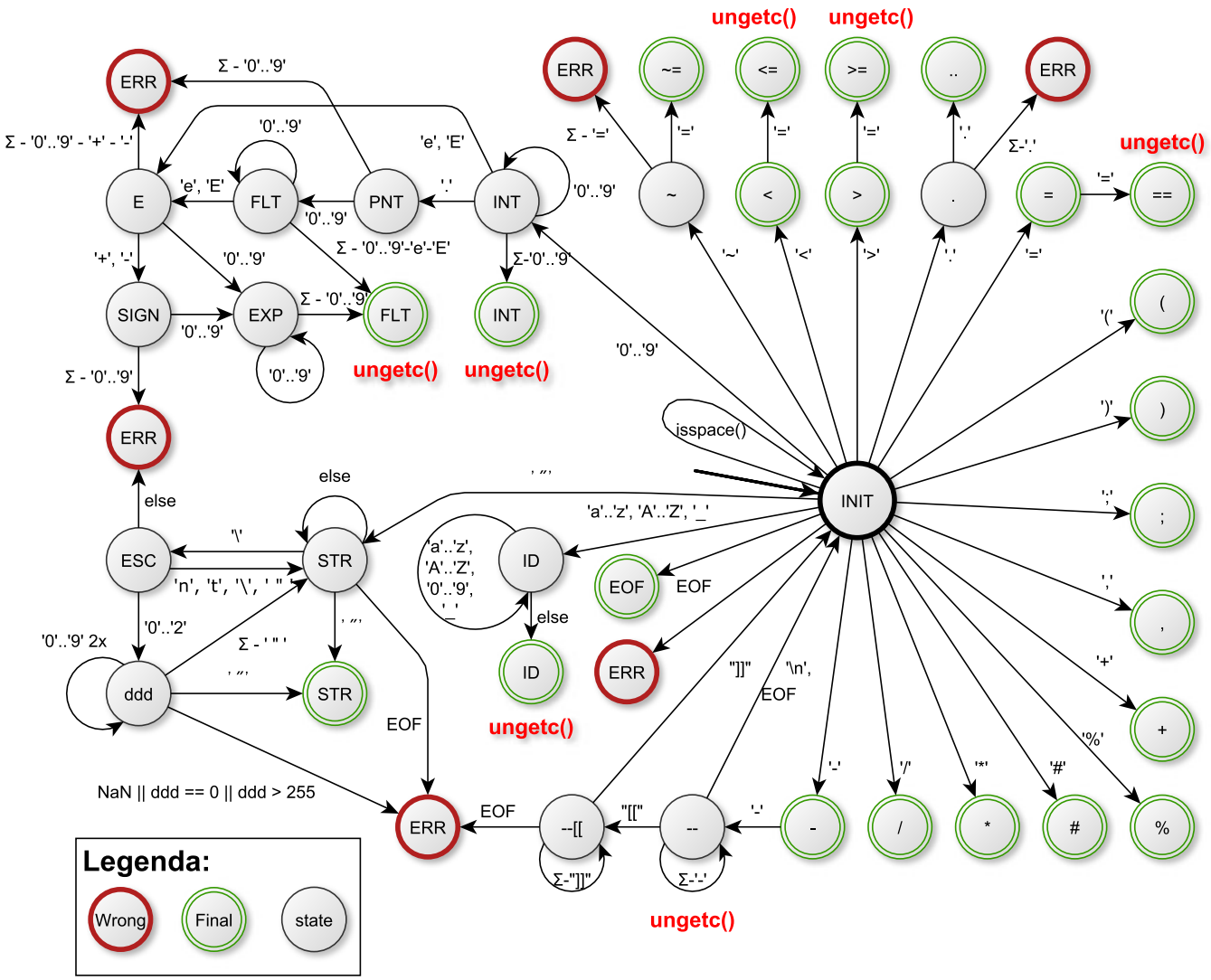
Stavy s označením `ungetc()` jsou konečné, avšak jejich typ lze určit až podle následujícího znaku. Pokud se nejedná o bílý znak, je znak navrácen.

Lexikální analyzátor při načtení identifikátoru (posloupnost čísli, písmen a znaku podtržítka začínající písmenem nebo podtržítkem) musí získaný řetězec porovnat s klíčovými slovy.

Pro snížení počtu porovnání řetězců je v implementaci použit algoritmus vyhledávání v poli s adaptivní rekonfigurací prvků. V poli jsou frekventovaně hledané prvky přeřazovány na nižší indexy a jsou tedy porovnávány dříve.

- **Token** je struktura, kterou vrací lexikální analyzátor. Představuje reprezentaci lexému, jenž syntaktický analyzátor získal ze zdrojového souboru. Token obsahuje typ, nebo značí chybu při lexikální analýze či neúspěch při vykonávání programu (např. chyba při alokaci paměťi). Některé tokeny nesou také datový obsah, v případě jazyka IFJ11 textový řetězec nebo číslo. V implementaci se objevuje označení typů tokenů, které lexikální analýza nepoužívá. Slouží jako pomocné typy pro zpracování výrazů.

Zdrojové kódy lexikálního analyzátoru jsou v souborech `scaner.c` a `scaner.h`.




Obrázek 1: Konečný automat lexikální analýzy

## 4.2 Model syntaktické a sémantické analýzy

### 4.2.1 Implementace syntaktického analyzátoru

Přestože syntaktická analýza představuje jeden funkční celek, návrh programovacího jazyka IFJ11 umožňuje rozdělit programové řešení na dvě nezávislé části, syntaktickou analýzu jazykových konstrukcí a syntaktickou analýzu výrazů. Po navržení vhodného rozhraní jsme tedy byli schopni práci na implementaci rozdělit mezi dva členy týmu.

Programovací jazyk IFJ11 lze popsat LL gramatikou, tudíž jsme pro zpracování syntaktické analýzy jazykových konstrukcí využili přístup shora dolů pracující s LL gramatikou. LL gramatika představuje srdce celého syntaktického analyzátoru výrazů a v našem podání skýtá 43 přepisujících pravidel (viz 4.2.4). Přestože je syntaktická analýza všeobecně proklamovaná jako nejnáročnější část projektu [2], shledali jsme, že po precizním navržení LL gramatiky již implementace rekurzívního sestupu představuje poměrně přímočarý postup, který by se dal charakterizovat jako algoritmický a nepříliš náročný.

Zpracovávání a vyhodnocování výrazů je řešeno s pomocí precedenční analýzy, kdy jsou jednotlivé tokeny postupně čteny a kopírovány na zásobník. Prováděné akce v průběhu zpracování

výrazu jsou založeny na tabulce priorit, jež obsahuje kombinace všech přípustných tokenů vyskytnuvších se ve výrazu. Příchozí tokeny se porovnávají s vrcholem zásobníku a na základě tabulky pravidel jsou generovány příslušné instrukce, či je vstup vyhodnocen jako chybný.

Implementace syntaktického analyzátoru je zpracována v modulech `parser.c`, `parser.h`, `stackexp.c` a `stackexp.h`.

### 4.2.2 Implementace sémantického analyzátoru

Pro zpracování celého interpretu jsme se rozhodli využít metodu syntaxí řízeného překladu, jež spojuje syntaktický a sémantický analyzátor i generátor vnitřního kódu do jednoho funkčního celku. Sémantická analýza tedy nepředstavuje autonomní část, nýbrž je plně integrována do syntaktického analyzátoru, který v případě potřeby volá její akce.

Mezi sémantické akce jsme zařadili kontrolu deklarací proměnných a definic funkcí, zpracování formátovacího řetězce příkazu `read()` a kontrolu kompatibility přímých hodnot vůči operátoru ve výrazech.

Obecně je jedním z častých úkolů sémantického analyzátoru kontrola typů proměnných. Tu však stejně jako implicitní přetypování neprovádíme, neboť jazyk IFJ11 (potažmo Lua) spadá do rodiny dynamicky typovaných jazyků, jež tolerují změnu typu proměnné v průběhu programu. Na úrovni syntaktické a sémantické analýzy tedy není možné určit, jakého typu daná proměnná aktuálně nabývá.

### 4.2.3 Implementace generátoru vnitřního kódu

Generátor vnitřního kódu je stejně jako sémanticky analyzátor integrován do syntaktického analyzátoru, který jej volá po dokončení zpracování elementárního funkčního celku.

V našem podání je generátor vnitřního kódu implementován jako jediná funkce, jež při každém volání vytvoří položku seznamu instrukcí a do seznamu ji vloží. Jakmile syntaktický analyzátor narazí na hlavičku hlavní funkce programu `main()`, vygeneruje instrukci návěští funkce a do seznamu zaznačí, že od této pozice bude později startovat interpretace.

### 4.2.4 LL Gramatika

| | | | |
|---|---|---|---|
| 01) | <prog> | ⟶ | function <restOfFuncs> main ( <params> ) <body> ; |
| 02) | <restOfFuncs> | ⟶ | ID ( <formParams> ) <body> function <restOfFuncs> |
| 03) | <restOfFuncs> | ⟶ | $\epsilon$ |
| 04) | <formParams> | ⟶ | ID <paramsNext> |
| 05) | <formParams> | ⟶ | $\epsilon$ |
| 06) | <formParamsNext> | ⟶ | , ID <paramsNext> |
| 07) | <formParamsNext> | ⟶ | $\epsilon$ |
| 08) | <body> | ⟶ | <varDecl> <stats> end |
| 09) | <varDecl> | ⟶ | local ID <assign> ; <varDecl> |
| 10) | <varDecl> | ⟶ | $\epsilon$ |
| 11) | <assign> | ⟶ | = <liter> |
| 12) | <assign> | ⟶ | $\epsilon$ |
| 13) | <liter> | ⟶ | NUMBER |
| 14) | <liter> | ⟶ | STRING |
| 15) | <liter> | ⟶ | true |
| 16) | <liter> | ⟶ | false |
| 17) | <liter> | ⟶ | nil |
| 18) | <stats> | ⟶ | write ( <exprWrite> ) ; <stats> |
| 19) | <stats> | ⟶ | if <expr> then <stats> else <stats> end ; <stats> |
| 20) | <stats> | ⟶ | while <expr> do <stats> end ; <stats> |
| 21) | <stats> | ⟶ | repeat <stats> until <expr> ; <stats> |
| 22) | <stats> | ⟶ | return <expr> ; |
| 23) | <stats> | ⟶ | ID = <funcOrExpr> ; <stats> |
| 24) | <stats> | ⟶ | $\epsilon$ |
| 25) | <funcOrExpr> | ⟶ | read ( <liter> ) |
| 26) | <funcOrExpr> | ⟶ | type ( <actParams> ) |
| 27) | <funcOrExpr> | ⟶ | substr ( <actParams> ) |
| 28) | <funcOrExpr> | ⟶ | find ( <actParams> ) |
| 29) | <funcOrExpr> | ⟶ | sort ( <actParams> ) |
| 30) | <funcOrExpr> | ⟶ | ID ( <actParams> ) |
| 31) | <funcOrExpr> | ⟶ | <expr> |
| 32) | <actParams> | ⟶ | <IDorLiter> <actParamsNext> |
| 33) | <actParams> | ⟶ | $\epsilon$ |
| 34) | <actParamsNext> | ⟶ | , <IDorLiter> <actParamsNext> |
| 35) | <actParamsNext> | ⟶ | $\epsilon$ |
| 36) | <IDorLiter> | ⟶ | ID |
| 37) | <IDorLiter> | ⟶ | <Liter> |
| 38) | <paramsWrite> | ⟶ | <expr> <exprNext> |
| 39) | <paramsWrite> | ⟶ | <expr> <paramsWriteNext> |
| 40) | <paramsWrite> | ⟶ | $\epsilon$ |
| 41) | <paramsWriteNext> | ⟶ | , <expr> <exprNext> |
| 42) | <paramsWriteNext> | ⟶ | , <expr> <paramsWriteNext> |
| 43) | <paramsWriteNext> | ⟶ | $\epsilon$ |

## 4.3 Interpret

Interpretace probíhá nad tříadresným kódem, který jsme definovali. Veškeré operace interpret provádí nad hlavním programovým zásobníkem implementovaným formou pole, jež roste při požadavcích na další paměť.

Veškeré syntaktické a sémantické prostředky jsou před interpretací zahozeny (identifikátory apod.) a interpretace probíhá pouze podle indexů v poli nad programovým zásobníkem.

Zásobník využívá dva pomocné indexy, které značí aktuální prvek a vrchol zásobníku. Od aktuálního prvku se získává relativní adresa na zásobník. Vrchol vždy ukazuje na první prázdný prvek.

- **Třiadresný kód** je implemtován pro potřeby indexování hodnot na zasobníku a práci interpretu. Ve struktuře tříadresného kódu se uchovávají obecné ukazelete. Tyto ukazelete se potom podle potřeby použijí jako adresa nebo číslo udávající relativní adresu na zásobníku. Náš interpret využívá celkem 32 instrukcí. Zajímavě jsou vymyšleny především instrukce pro práci se zásobníkem.

Implementace zásobníku se nachází v modulech `stack.h`, `stack.c`, instrukční sada tříadresného kódu v modulech `ilist.h`, `ilist.c` a interpret v modulech `interpret.h` a `interpret.c`.

## 4.4 Algoritmy pro práci s řetězci

Hlavním kritériem pro výběr všech algoritmů byla zvolena rychlost a k této vlastnosti jsme pochopitelně přihlíželi nejvíce.

Zdrojový kód je umístěny v modulech `ial.h` a `ial.c`.

### 4.4.1 Řadící algoritmus Heap Sort

Algoritmus provádí řazení in situ. Dokáže tedy řadit větší počet prvků než jiné algoritmy, které při své funkci potřebují dodatečné paměťové zdroje (např. rekurzivní algoritmy).

K vlastnostem tohoto rychlého algoritmu patří časová složitost $O(n * log(n))$ a již zmíněná paměťová náročnost. Vlastnosti jako stabilita a přirozenost byly pro výběr sekundární. Na základě těchto faktů jsme si zvolili Heap Sort jako vhodný a efektivní.

Algoritmus jsme implementovali ve verzi, jejíž předloha je dostupná na [3]. Ke zvýšení efektivnosti je použita řada bitových operací pro aritmetiku algoritmu.

### 4.4.2 Boyer-Mooerův algoritmus

Boyer-Moore algorimus porovnává vyhledávaný podřetězec v textu zprava doleva. Pokud nenalezneme znak, který je nejvíce vpravo ve vzoru v textu, můžeme posunout celý text o délku vzoru za znak, který se neshoduje.

Jako typ heuristiky jsme zvolili *Turbo Boyer-Moore* [4]. TBM vyžaduje navíc dodatečný paměťový prostor. V nejhorším případě lze nalézt řetězec během $2n$ porovnání (kde $n$ je počet znaků textu, v němž vyhledáváme).

### 4.4.3 Práce s řetězci

Práce s nekonečnými řetězci probíhá nad dynamickými poli. Funkce k tomu určené se nacházejí v modulech `str.h` a `str.c`. Jsou zde použity funkce, které byly k dispozici na stránkách předmětu `IFJ`.

### 4.5 Tabulka symbolů

Tabulka symbolů je implementována hašovací tabulkou (Hast Table) a inicializujeme ji na velikost 499 (pro efektivní chování hašovací tabulky je vhodné použít velikost danou prvočíslem). Vlastností této datové struktury je rychlé vyhledávání prvku, v ideálním případě se složitostí $O(1)$.

K sémantickým kontrolám bylo potřeba zavést dvě tabulky. První z nich je použita jako tabulka globalní s klíčem identifikujícím jméno funkce, daty určujícími počet parametrů a ukazatelem na první instrukci dané funkce. Druhá tabulka je lokální, je použita v celém programu pouze jednou a před zpracováním každé následující funkce je vyčištěna. Uchovává číslo, které je spolu s počtem aktualně uchovávaný položek v tabulce potřebné k výpočtu relativní adresy na programový zásobník.

V implementaci jsou použity materiály získané řešením projektu v předmětu IJC. Kód je tvořen výběrem ze zdrojových kódů všech čtyř řešitelů projektu IFJ, proto vytvoření tabulky nepředstavovalo problém.

## 5 Rozšíření

### 5.1 LOCALEXPR

Rozšíření LOCALEXPR jsme si paradoxně vybrali proto, že nám usnadnilo implementaci řešení deklarací proměnných. Pakliže se syntaktický analyzátor nachází v deklarační části funkce, volá funkci pro zpracování výrazu nezávisle na obsahu pravé strany přiřazovacího operátoru.

### 5.2 MUTREC

Pro implementaci rozšíření MUTREC jsme využili pomocnou strukturu typu seznam, do níž se v průběhu syntaktického analyzátoru ukládají identifikátory volaných funkcí a instrukce pro volání funkce. Před voláním intepretu se potom v rámci jednoho průchodu seznamem prověří, zda byly všechny funkce daných identifikátorů definované, a do dříve vygenerovaných instrukcí pro volání funkce se doplní návěští začátku volané funkce.

### 5.3 REPEAT

Rozšíření REPEAT považujeme za jedno z nejjednodušších, poněvadž se od povinného cyklu while-do liší pouze uspořádáním návěští podmíněného skoku a vyhodnocením podmínky ukončení.

### 5.4 MODULO a LENGTH

Tato rozšíření rovněž nepovažujeme za nikterak obtížné. Pouze bylo nutné doplnit lexikální analyzátor a syntaktický analyzátor výrazů o příslušné symboly a doplnit dvojici aritmetických instrukcí.

## 6 Závěr

Jelikož žádný ze členů týmu neměl dřívější zkušenost s týmovou programátorskou prací, získali jsme při řešení množství cenných zkušeností. Pochopili jsme, že rozsáhlejší projekty zkrátka nemůže řešit jeden člověk, avšak komunikace mezi více řešiteli přináší mnohé problémy a

značnou časovou režii. Závěrem snad můžeme s lehkou nadsázkou pouze dodat: Projekt se nám líbil a těšíme se na další.

# Reference


[1] HONZÍK J.: *Studijní opora předmětu Algoritmy.*

[2] MEDUNA, A.; LUKÁŠ, R.: *Přednášky k předmětu Formální jazyky a překladače.*

[3] *Programming Contest Central.* [online] [cit. 2011-12-08] Heap Sort Part 1: Learn. Dostupné z WWW: <https://www-927.ibm.com/ibm/cas/hspc/student/algorithms/HeapSort.html>.

[4] *Exact String Matching Algorithms.* [online] [cit. 2011-12-08]. Turbo-BM algorithm. Dostupné z WWW: <http://igm.univ-mlv.fr/~lecroq/string/node15.html>.


# A Metriky kódu

**Počet souborů:** 19 souborů

**Počet řádků zdrojového textu:** 4803 řádků

**Velikost spustitelného souboru:** 52 995 B (systém Linux, 32bitová architektura, při překladu bez ladicích informací)