Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-5212-47810

Gabriel Duchoň

# Generovanie dokumentácie zo zdrojových kódov

Bakalárska práca

Študijný program: Informatika
Študijný odbor: 9.2.1 Informatika
Miesto vypracovania: Ústav informatiky a softvérového inžinierstva, FIIT STU Bratislave
Vedúci práce: Ing. Peter Kapec

máj, 2010

## ČESTNÉ PREHLÁSENIE

Čestne prehlasujem, že záverečnú prácu som vypracovával samostatne s použitím uvedenej literatúry.

Gabriel Duchoň

## POĎAKOVANIE

Týmto spôsobom by som sa chcel poďakovať najmä vedúcemu záverečnej práce, Ing. Petrovi Kapcovi, za užitočné rady, ochotu, pomoc, vynaložený čas a usmerňovanie pri vypracovávaní záverečnej práce. Ďakujem tiež svojim rodičom a blízkym, ktorý ma podporovali počas štúdia.

# Anotácia

Slovenská Technická Univerzita v Bratislave
Fakulta informatiky a informačných technológií

Študijný program: Informatika

Autor: Gabriel Duchoň
Názov bakalárskej práce: Generovanie dokumentácie zo zdrojových kódov
Vedúci bakalárskej práce: Ing. Peter Kapec
máj, 2010

Táto práca sa zaoberá problematikou generovania dokumentácií zo zdrojových kódov. Existuje niekoľko nástrojov a technológií používaných v tejto oblasti. Zameriavame sa tiež na problematiku analyzovania zdrojového kódu a jej hlavné typy. Práca analyzuje bežné techniky, ako je formátovanie a zvýrazňovanie syntaxe, ale aj menej rozšírené prístupy, ako je dokumentačné programovanie. Opísali sme niekoľko existujúcich nástrojov a porovnali. Hlavným cieľom tejto práce bolo vyvinúť nové funkcie existujúceho nástroja LuaDoc. Pridali sme funkcionalitu pre vytvorenie indexu všetkých funkcií, formátovanie a zvýrazňovanie syntaxe zdrojového kódu a rozšírenie generovanej dokumentácie o základné metriky zdrojového kódu. Všetky pridané funkcionality boli vyvinuté na základe PEG gramatík.

Kľúčové slová: *generovanie dokumentácií, parsing expression grammars, Lua, LuaDoc,*

# Annotation

The Slovak University of Technology Bratislava
Faculty of Informatics and Information Technology

Degree Course: Informatics

Author: Gabriel Duchoň
Title of the bachelor theses: Generating documentation from source code
Supervisor: Ing. Peter Kapec
2010, May

This work deals with the problematics of generating documentation from source code. There are several tools and technologies used in this area. We focus also on the problem of source code analysis and its main types. This work analyzes common techniques like, pretty printing and syntax highlighting, but also describes less common approaches like literate programming. Several existing tools are described and compared. The main goal of this thesis was to develop new features to the existing tool LuaDoc. We added functionality to generate global indexes of functions, added pretty printed source code and augmented the generated documentation with basic source code metrics. All added functionality was developed upon parsing expression grammars.

Keywords: *documentation generation, parsing expression grammars, Lua, LuaDoc,*

# Obsah

**1 ÚVOD ... 1**

**2 ANALÝZA ... 3**

2.1 Generovanie dokumentácie ... 3

2.2 Syntaktická analýza projektu ... 4

2.2.1 Parsovanie založené na gramatikách PEG ... 6

2.3 Formátovanie textu a zvýrazňovanie syntaxe ... 7

2.4 Existujúce nástroje na zvýrazňovanie syntaxe ... 8

2.5 Jazykovo-agnostické zobrazovanie zdrojového kódu ... 9

2.6 Dokumentačné programovanie ... 11

2.7 Meranie metrík zdrojového kódu ... 12

2.8 Skriptovací jazyk Lua ... 14

2.9 Existujúce nástroje na generovanie dokumentácií ... 14

2.9.1 LuaDoc ... 15

2.9.2 Doxygen ... 17

2.9.3 Javadoc ... 21

2.9.4 NaturalDocs ... 22

2.10 Sumarizácia analýzy nástrojov ... 23

2.11 Zhodnotenie analýzy ... 23

**3 OPIS RIEŠENIA ... 25**

3.1 Špecifikácia požiadaviek ... 25

3.2 Návrh ... 25

3.3 Architektúra systému ... 27

3.4 Implementácia ... 28

**4 OVERENIE RIEŠENIA ... 29**

**5 ZHODNOTENIE ... 34**

5.1 Známe obmedzenia ... 34

5.2 Možné smery ďalšieho rozvíjania ... 35

**ZOZNAM POUŽITEJ LITERATÚRY ... 36**

**PRÍLOHA A – TECHNICKÁ DOKUMENTÁCIA ..... 38**
A.1 Dokumentácia k špecifikácii ..... 38
A.2 Dokumentácia k návrhu ..... 38
A.3 Dokumentácia k implementácii ..... 40
A.3.1 Opis funkcie `processASTAsHtml` modulu `highlighter` ..... 40
A.3.2 Opis funkcie `processAST` modulu `functionLister` ..... 42
A.3.3 Zmenená funkcia `start` modulu `luadoc.doclet.html` ..... 43

**PRÍLOHA B – NÁVOD NA POUŽÍVANIE ..... 46**

**PRÍLOHA C – OBSAH ELEKTRONICKÉHO MÉDIA ..... 47**

**PRÍLOHA D – KONŠTRUKTORY JAZYKA LPEG ..... 48**

**PRÍLOHA E – SYNTAX JAZYKA LUA ..... 49**

**PRÍLOHA F – PEG PRAVIDLÁ OPISUJÚCE GRAMATIKU JAZYKA LUA (ŠTRUKTÚRA AST) ..... 50**

# 1 Úvod

Dokumentácia je textový dokument, ktorý je súčasťou projektu, v našom prípade softvérového projektu. Obsahuje informácie o danom projekte, všeobecné až podrobné. Podrobnosť informácií zváži autor/autori dokumentácie (zvyčajne autor/autori projektu).

Dokumentácia zhromažďuje informácie o vývoji softvérového projektu, alebo jeho časti. Môže obsahovať informácie o analýze, špecifikácii, návrhu riešeného problému, implementácie, testovania, ladení a o celom manažmente procesu vývoja. Je to slovný opis štruktúry, komponentov a jednotlivých parametrov navrhovaného systému, ktorý môže autor dopĺňať diagramami, grafmi, obrázkami a inými objektmi potrebnými pre vhodnú prezentáciu danej abstrakcie problému. Dokumentácia tiež môže opisovať už hotový projekt. Výhoda je v tom, že autor nemusí robiť revíziu jednotlivých kapitol pri patričných zmenách, ale opisuje už vytvorené dielo, ktoré má presnú funkcionalitu a štruktúru.

Dokumentácia obsahuje rôzne abstrakcie projektu. Postupnou konkretizáciou získavame postupne viac a viac detailov projektu. Najpresnejší opis projektu je zdrojový kód. Predmetom tejto bakalárskej práce sú dokumentácie, ktoré opisujú práve programový kód. Tieto dokumentácie opisujú vnútornú štruktúru, rozdelenie, jednotlivé časti projektu (triedy, moduly, balíky, komponenty), funkcie jednotlivých častí (tried, modulov) projektu, použité technológie, štatistiky, poznámky autora a iné. Na uľahčenie vytvorenia dokumentácií tohto typu slúžia programy na generovanie dokumentácií. Sú to nástroje používajúce rôzne technológie na spracovávanie zdrojových kódov a vytvárajú jeden dokument, alebo množinu súvislých dokumentov opisujúcich zdrojový kód.

Pri vytváraní dokumentácií, generátor získava informácie o kóde vyjadrujúci program, čiže obsah zdrojových kódov. V rámci zdrojového kódu je možné definovať funkcionalitu jednotlivých blokov pomocou komentárov. V tele komentárov je možné definovať aj informácie, inštrukcie, ktoré sú určené len pre softvér generovania dokumentácií. Tieto inštrukcie, tak ako ani iná časť komentárov, sa pri vykonávaní programu nepoužijú a funkcionalitu softvérového projektu neovplyvňujú. Takéto informácie sú napríklad opisy častí zdrojového kódu, poznámky autora, referencia na príbuzné časti zdrojového kódu, nastavenia formátovania výslednej dokumentácie.

V súčasnosti je programovací/skriptovací jazyk Lua len veľmi málo podporovaný v oblasti generovania dokumentácií. Hlavný nástroj na generovanie dokumentácií zo zdrojových kódov vytvorený pre jazyk Lua je LuaDoc. Prvá verzia (LuaDoc 1.0) sa objavila v roku 1999. Aktuálna verzia v čase písania práce je LuaDoc verzia 3.0.0, ktorá bola vydaná 13.08.2007. Predmetom tejto práce je analýza nástrojov na generovanie dokumentácií, ich opis a porovnanie, opis jednotlivých technológií, ktoré sa využívajú pri generovaní dokumentácie a naimplementovanie modulov, ktoré by rozširovali možnosti generovania dokumentácií zo zdrojových kódov zo skriptovacieho jazyka Lua.

# 2 Analýza

Kapitola sa zaoberá analýzou problémového prostredia. Poskytuje opis všeobecných informácií, opis techník a metód analýzy zdrojového kódu, charakteristiku PEG gramatík, komplexnejšie metódy a paradigmy v oblasti generovania dokumentácie. Ďalej opisuje vlastnosti skriptovacieho jazyka Lua a nástroje na generovanie dokumentácií z tohto a iných programovacích jazykov.

## 2.1 Generovanie dokumentácie

Nástroj na generovanie dokumentácií je užitočný, ako pre autora projektu, tak aj pre iných vývojárov, ktorí chcú použiť daný projekt. Kvalitná štruktúra, formát a výzor výslednej dokumentácie, bohaté funkcie a možnosť konfigurácie generovania pridávajú na kvalite softvéru tohto typu.

Dokumentovanie môžeme rozdeliť aj podľa toho, aký rozsiahly je zdrojový kód projektu, na *dokumentovanie v malom* a *dokumentovanie vo veľkom* [11]. Je značný rozdiel medzi dokumentovaním projektu, ktorý má zdrojový kód rozsiahly niekoľko stoviek, alebo tisícok riadkov a projektom, ktorý má niekoľko miliónov riadkov kódu. Zvyčajne sa dokumentácie (etapy implementovania) týkajú menších projektov, kde dokumentácia opisuje jednotlivé funkcie, algoritmy a štruktúru dát. Pri rozsiahlych dokumentáciách je skôr potrebné prezentovať architektúru systému, ako menšie komponenty, algoritmy. Dokumentáciu pre rozsiahly systém je manuálne vytvoriť obtiažné, vytvoriť viacero dokumentácií pre rôzne pohľady je tiež nesprávne. Pre tento účel je možné použiť reverzné inžinierstvo (*reverse engineering*). Je to proces extrahovania rôznych abstrakcií existujúceho systému. Štruktúra softvérového systému je potrebná k tomu, aby čitateľ dokumentácie vedel vytvoriť mentálny model systému, ktorý použije pri vyvíjaní, analyzovaní, integrovaní, implementovaní systému.

Prvoradý účel softvéru je vývoj. Pri vývoji nového softvéru sa zvyčajne použijú už existujúce systémy, alebo ich časti. Dokumentácia je potrebná pri vývoji, aby sa vývojár vedel rozhodnúť, či je vhodné použiť daný softvér, alebo nie. Pri rozsiahlych projektoch je tiež využívaná ako nástroj čitateľa, umožňujúci analýzu projektu, jeho architektúry a pochopenie jeho podstaty. Pomocou dokumentácie je možné sledovať aj vývoj (evolúciu) softvérového systému.

Pri vytváraní dokumentácií je potrebné brať ohľad na štyri kritériá [2]:

1. Dokumentácia by mala obsahovať opis systému na rôznych úrovniach abstrakcie.
2. Je potrebné, aby používateľ dokumentácie mohol meniť úroveň abstrakcie bez toho, aby stratil pozíciu v kontexte dokumentácie (*zoom in* a *zoom out*).
3. Rôzne úrovne abstrakcie systému musia byť zrozumiteľné pre určeného čitateľa.
4. Dokumentácia musí byť neustále konzistentná so zdrojovým kódom.

## 2.2 Syntaktická analýza projektu

Pri generovaní dokumentácie je potrebné analyzovať projekt. Ako najjednoduchšia možnosť je *lexikálna analýza* [2]. Je nutné, aby jazyk, v ktorom je projekt vytvorený, bol jednoducho rozpoznateľný. Dve výhody tejto analýzy sú, že nie je potrebné poznať celú syntax a je časovo nenáročná. Nevýhoda je, že nie je precízna, nie je možné rozoznať rôzne kontexty (napríklad, či je rozoznaný reťazec naozaj kód, alebo len časť komentára).

Ďalší typ analýzy je *syntaktická analýza* [2]. Vytvára abstraktný syntaktický strom, a pri tejto analýze je potrebné definovať celú syntax daného jazyka, čo je časovo náročná úloha. Táto metóda analýzy je opísaná v podkapitole 2.2. Ako najlepšia možnosť analýzy sa ukazuje analýza pomocou ostrovných gramatík (*island grammars*) [2]. Umožňuje definovať gramatiku na rozoznávanie syntaxe tak, že opisujeme rozoznávanie syntaxe len pre nás potrebných dát. Najväčšia výhoda je flexibilita techniky *ostrovných gramatík.* V princípe ich je možné použiť v kombinácii akéhokoľvek *parser generátora*. Táto gramatika používa SDF2[1], na ktorom je možné realizovať *bezskenové parsovanie* (lexikálne skenovanie a parsovanie kódu je jeden krok) [12]. Parsery pre SDF2 akceptujú bezkontextové gramatiky a LALR[2] gramatiky. SDF2 sú modulárne syntaktické gramatiky a vďaka tomu vieme špecifikovať *ostrovné gramatiky* v rôznych moduloch. Takto pri rôznych analýzach máme rôzne gramatiky, ktoré sú rozšíreniami hlavnej gramatiky. Je ich možné použiť tiež pri parsovaní viacerých jazykov (pri tzv. *hybridnom programovaní*) súčasne do jedného *parsovacieho stromu* [8].

[1] *S*yntax *D*efinition *F*ormalism, version 2

[2] *L*ook *A*head *L*eft *R*ight

*„Ostrovy (islands) môžu byť ľubovoľne zložité a preto nekladú žiadne obmedzenia voči jazyku, ktorý je parsovaný. Rozličné ostrovy sú voľne viazané, a preto zmeny jedného jazyka neovplyvňujú zmeny iného jazyka. Ostrovné gramatiky zabezpečujú vhniezdenie ostrovov do iných ostrovov, čo umožňuje spracovanie jazykov, ktoré sú ľubovoľne vhniezdené do ďalších jazykov.*“ [8]

Extrahovanie informácií zo zdrojového kódu môže prebiehať manuálne, alebo automaticky. Pri automatickom extrahovaní informácií sa môžu pridávať prídavné informácie aj manuálne. Informácie sa získavajú tiež: dekompozíciou softvérového systému (je rozdelený podľa úrovní dekompozície), agregáciou a použitím relácie *use*, rozdelením systému a podsystémov (podľa adresárov a súborov), opisom programov (komentármi opisujúcimi účel komponentu, užitočnými informáciami, príkazmi – *tagy* sú tiež umiestnené v komentároch, aby neovplyvňovali správnosť a syntax zdrojového kódu), opismi sekcií/procedúr (zvyčajne sú informatívne už aj ich názvy, opisujú účel segmentu a informácie o ňom), atď.

Vybrané vygenerované informácie sa zvyčajne prezentujú pomocou hypertextu, kde jedna stránka opisuje jeden komponent a pomocou *hypertextových odkazov* (ktoré umožňujú pohyb v dokumentácií) vyjadrujeme vzťahy a závislosti medzi komponentmi. Pre ľahšie vyhľadávanie potrebných informácií je možné uviesť index mien programov, vyhľadávač kľúčových slov v obsahu, grafy vzťahov častí systému.

Pri syntaktickej analýze sa vytvára abstraktný syntaktický strom, ktorý obsahuje entity reprezentujúce syntaktickú štruktúru zdrojového kódu. Jednotlivé uzly reprezentujú väčšie, alebo menšie časti kódu, podľa danej hĺbky stromu. Koreň abstraktného syntaktického stromu reprezentuje celý zdrojový kód a z neho vystupujú vetvy, ktoré vlastne rozoberajú zdrojový kód, jednotlivé funkcie, inštrukcie až na symboly, kľúčové slová, premenné, biele znaky, komentáre, atď. Pomocou tejto reprezentácie zdrojového kódu je jednoduchšie analyzovať danú časť projektu, keďže máme neustále prehľad o tom, v akej hĺbke sa nachádzame, aké sú vyššie vrstvy daného uzla.

Pri tejto analýze sa využíva rekurzívne prehľadávanie stromu (ktorý môže byť v špeciálnych prípadoch aj cyklický) a podľa hľadaných uzlov a ich predchodcov sa generuje správny formát výstupu (formátovací strom, viď. podkapitola 2.3)

### 2.2.1 Parsovanie založené na gramatikách PEG

**PEG** (*parsing expression grammar*) je typ analytickej formálnej gramatiky, ktorá slúži na rozoznávanie formálnej gramatiky [4].

Výhody gramatiky PEG sú: je ľahko rozšíriteľná o nové pravidlá, je založená teóriou, ktorá formálne popisuje správanie akéhokoľvek vzoru (*pattern*), majú implementované niekoľko *pattern-matching mechanizmov*, ako sú: *lačné opakovanie* (*greedy repetitions*), *nelačné opakovanie* (*non-greedy repetitions*), *pozitívny lookhead, negatívny lookhead*. Môžu vyjadriť každý deterministický jazyk (nie každú deterministickú gramatiku) a tiež niektoré bezkontextové jazyky. Umožňujú rýchlu a jednoduchú implementáciu založenú na *parsovacom mechanizme*. Najhoršiu časovú náročnosť majú $O(n^k)$ (a pamäťovú $O(k)$, čo je konštanta pre daný vzor).

```
A <- B C D / E F / ...
```

PEG je postupnosť pravidiel, kde *vzor* (*pattern*), pravá strana pravidla (vpravo od symbolu '<-'), je postupnosť alternatívnych možností oddelený so znakom '/'. Každý výraz môže mať operátor formou prípony: *E** znamená nula, alebo viac výskytov výrazu za sebou, *E+* znamená jeden, alebo viac výskytov výrazu za sebou. Regulárne výrazy musia byť uvedené úvodzovkami. Tiež je možné uviesť výraz ako množinu *x-y*, čo znamená množinu všetkých znakov medzi znakmi *x* a *y*.

Opakovanie môžu byť lačné (*greedy*), nelačné (*non-greedy*), slepé (*blind*) a neslepé (*non-blind*). Typy opakovania rozdeľujeme podľa ich kombinácie (*greedy*/*non-greedy* s *blind*/*non-blind*). Opakovania sú založené na *podmienenom backtrackingu*, čo znamená, že po zhodnotení prvého výrazu sa vykoná *backtracking* len v prípade ak výraz nevyhovie danej podmienke. Napríklad ak máme pravidlá:

```
S <- A B
A <- A1 / A2 / A3 / ... / An
```

aby vyhovel výraz A, musí sa skontrolovať výraz $A_1$, ak nevyhovie, prebehne *backtrack* a skontroluje sa výraz $A_2$. Toto sa opakuje až kým $A_n$ nevyhovie. Ak vyhovie aj len jeden výraz, už sa nevykoná žiadny *backtrack*, aj keď B zlyhá.

PEG tiež umožňuje uviesť syntaktické predikáty *not* a *and*. Predikát *not* je vyjadrený znakom '!'. Výraz *!E* vyhovie, len ak výraz *E* nevyhovie. Predikátom *not* sa tiež zisťuje koniec vstupu: „!.“. Predikát *and* je vyjadrený znakom '&'. Výraz *&E* je

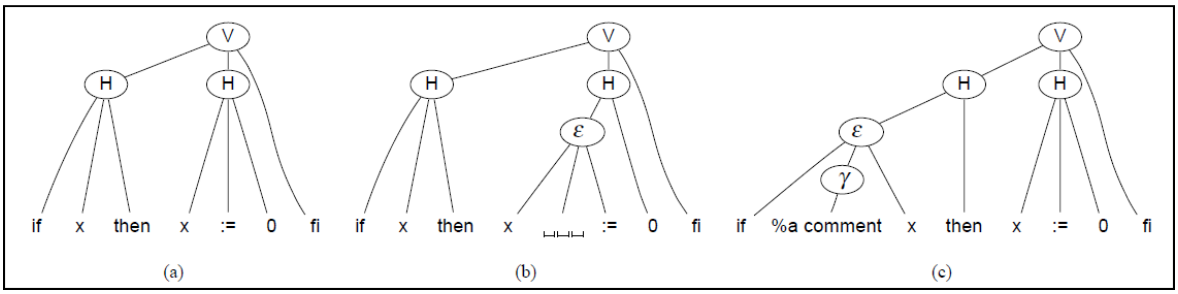
ekvivalentný výrazu *!!E*, vyhovie v prípade, ak aj výraz *E* vyhovie, ale nepotrebuje žiadny vstup.

Knižnica **LPeg** implementuje gramatiku PEG pre jazyk Lua a slúži na vyhľadávanie vzoriek – *pattern matching*. Zoznam konštruktorov na vytváranie vzoriek je uvedený v prílohe D. LPeg je *greedy*, t.j. „pažravý, lačný“ – pri vyhľadávaní zachytáva čo najväčší možný reťazec znakov.

Knižnica **Leg** používa knižnicu LPeg – napr. verziu 0.8.1, (resp. je kompatibilná len s verziami nižšími ako verzia 0.9). Ponúka kompletnú gramatiku jazyka Lua. Pozostáva z troch modulov *grammar*, *parser*, *scanner* a tak vytvárajú nástroj na rozoznávanie, parsovanie zdrojových kódov v jazyku Lua. Štruktúra gramatiky je zostavená pomocou pravidiel, ktoré vytvárajú vzorky (*patterns*) a tie sú ďalej vyhľadávané knižnicou LPeg.

## 2.3 Formátovanie textu a zvýrazňovanie syntaxe

Formátovanie zdrojového kódu, alebo *pretty printing* [9], je súčasťou spätného inžinierstva softvéru. V prvom rade sa využíva pri regenerovaní zdrojového kódu, transformuje abstraktnú reprezentáciu programu späť pre človeka zrozumiteľného textu. V druhom rade, sa využíva aj pri generovaní dokumentácie. Umožňuje formátovanie zdrojového kódu na krajší, čitateľnejší. Nástroj umožňujúci formátovanie zdrojového kódu by mal podporovať generovanie do rôznych formátov pomocou pravidiel, mal by tiež rešpektovať špecifické konvencie určené používateľom a úprava by mala byť minimálna (komentáre a rozloženie textu by mali byť zachované). Pri formátovaní zdrojového kódu je potrebné vytvoriť abstraktnú reprezentáciu programu. Je to možné realizovať pomocou *parsovacieho stromu*, alebo *abstraktného syntaktického stromu* [5]. *Parsovací strom* obsahuje všetky lexikálne informácie vrátane komentárov a celý obsah zdrojového kódu (aj biele znaky). *Abstraktný syntaktický strom* neobsahuje žiadne informácie o rozložení, avšak obsahuje všetky kľúčové slová programu. Ako prvý krok sa z abstraktnej reprezentácie vytvorí *pokročilý formát*, ktorý už obsahuje aj informácie o formátovaní. Na tento účel slúži *formátovací strom* [5]. Prvky tohto stromu korešpondujú operátorom formátovania (Obrázok 1) Ako posledný krok, sa pomocou informácií o formátovaní vytvorí očakávaný výstup. Výstupom môže byť jednoduchý text, alebo tiež iný formát (LaTeX, HTML alebo PDF).




**Obrázok 1 – (a) Formátovací strom, (b) formátovací strom, v ktorom je vzhľad zachovaný (medzery), (c) formátovací strom so zachovanými komentármi.**

Z pohľadu údržby rozdeľujeme *pretty printer* na [5]:

i) *spracovávateľ formátovacieho stromu*, ktorý vytvára jazykovo-špecifický formát reprezentovaný *formátovacím stromom;*

ii) *spotrebiteľ formátovacieho stromu*, ktorý transformuje *formátovací strom* na výstupný formát.

Zvýrazňovanie zdrojového kódu môže byť súčasťou formátovania zdrojového kódu. Najjednoduchšia forma zvýrazňovania zdrojového kódu je zvýrazňovanie kľúčových slov, štandardných typov, konštánt a komentárov zmenou ich farby. Komplexnejšia forma formátovania vyžaduje nástroj pre dokumentačné programovanie (*literate programming*) [6]. Tento výstup je oveľa komplexnejší a umožňuje zvýrazňovanie aj jednotlivých sémantických častí v závislosti od toho, či reprezentujú typy, triedy, premenné, alebo funkcie. Túto techniku používajú hlavne nástroje dokumentačného programovania, keďže je jazykovo veľmi závislá.

Ďalšia obľúbená a častá funkcia formátovania je používanie súborov symbolov a znakov, ktoré majú bohatší obsah symbolov ako štandardné (napr. v jazyku C nahradenie znakov '<=' symbolom '≤'). Táto funkcia slúži na zvyšovanie čitateľnosti naformátovaného textu.

## 2.4 Existujúce nástroje na zvýrazňovanie syntaxe

Existuje mnoho editorov, ktoré ponúkajú možnosť zvýrazňovania textov. V súčasnosti sa kladie dôraz na čoraz väčšiu všestrannosť editorov a s tým sa rozširujú aj možnosti využívania editora na programovanie v rôznych programovacích jazykoch. Editor programového kódu je atraktívny, ak čo najlepšie zobrazuje zdrojový kód a k tomu patrí formátovanie zdrojového kódu, zvýrazňovanie syntaxe, zvinutie/rozvinutie tela funkcií, cyklov a iných väčších, alebo menších blokov. Hlavné nástroje ponúkajúce možnosť

pracovať so skriptami Lua sú: SciTE (viď. vzorový kód zobrazený editorom SciTE na obrázku 2), Notepad++, Programmer's Notepad, LuaEclipse, GNU Emacs, atď.

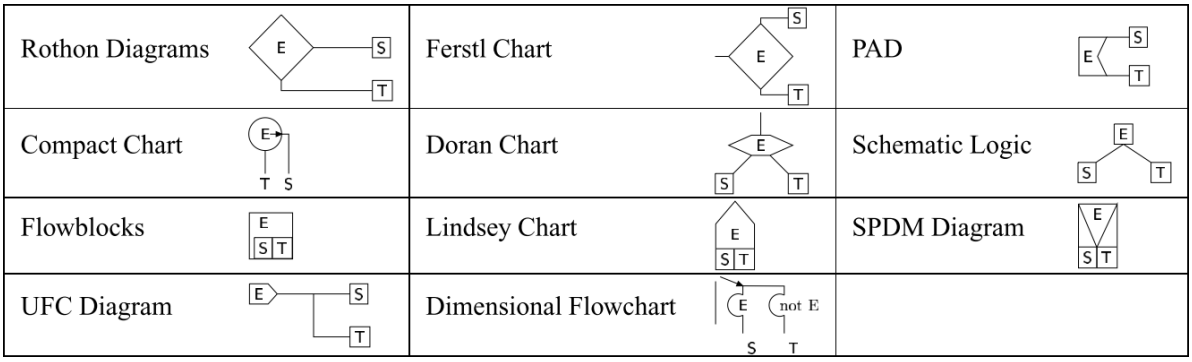
```
21  ----------------------------------------------------------------------------
22  -- Main function
23  -- @see luadoc.doclet.html, luadoc.doclet.formatter, luadoc.doclet.raw
24  -- @see luadoc.taglet.standard
25
26  function main (files, options)
27      logger = util.loadlogengine(options)
28
29      -- load config file
30      if options.config ~= nil then
31          -- load specified config file
32          dofile(options.config)
33      else
34          -- load default config file
35          require("luadoc.config")
36      end
37
38      local taglet = require(options.taglet)
39      local doclet = require(options.doclet)
40
41      -- analyze input
42      taglet.options = options
43      taglet.logger = logger
44      local doc = taglet.start(files)
45
46      -- generate output
47      doclet.options = options
48      doclet.logger = logger
49      doclet.start(doc)
50  end
```

**Obrázok 2 – Zobrazenie zdrojového kódu editorom SciTE**

## 2.5 Jazykovo-agnostické zobrazovanie zdrojového kódu

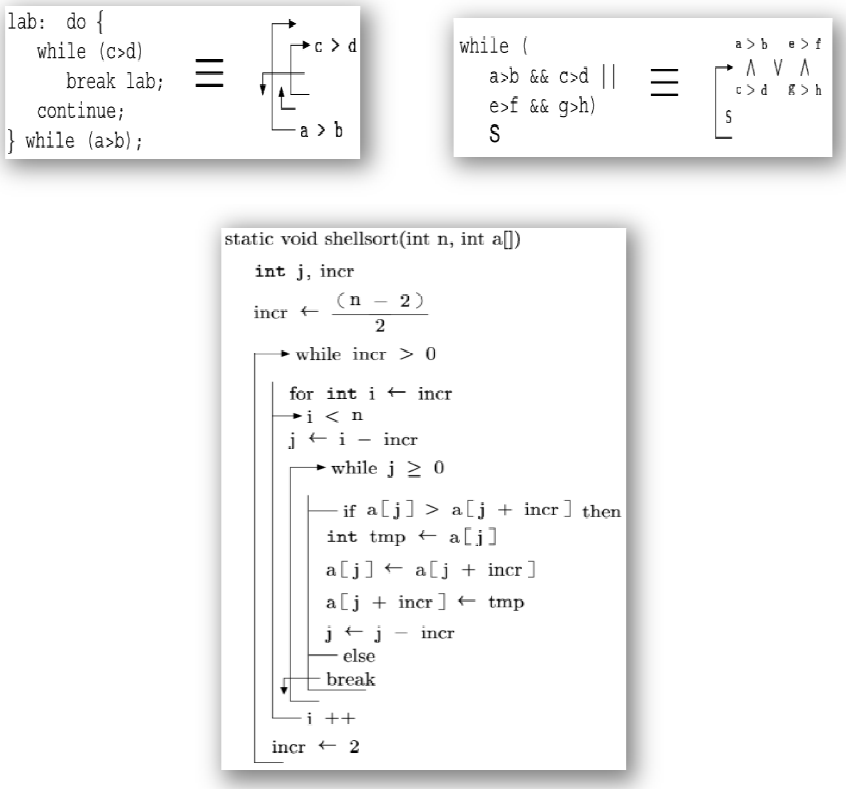
Agnosticizmus je myšlienka, podľa ktorej ľudská myseľ dokáže zachytiť podstatu vecí, len skúsenosti a skutočnosť. Napriek tomu, že existuje veľa návrhov grafickej notácie zdrojového kódu, ani jeden sa nepresadil v masovom využití v programátorskej komunite. Dôvody môžu byť nasledovné: vývojári uprednostňujú textový opis pred grafickým; vývojári môžu považovať grafickú notáciu za zbytočnú keďže neobsahuje žiadnu informáciu, ktorú by neobsahoval zdrojový kód; notácie založené na „krabičkách“ (Obrázok 3) vyzerajú prívetivo pre malé programy, ale reálne programy sú dlhé, zložité a ich zobrazenie by bolo často neprehľadné. ART[3] je jazykovo nezávislý nástroj na zobrazovanie zdrojových kódov [1]. Je to jeden z nástrojov pre „skrášľovanie“ zdrojového kódu, tak ako aj formátovanie textu (*pretty printing)*, alebo dokumentačné programovanie.

[3] *A*gnostic *R*endering *T*ool




**Obrázok 3 – Príklady vizuálnych notácií**

ART používa orientované šípky na zobrazenie prechodov a logiku vykonávania príkazov. Výstupom je pseudo kód (Obrázok 4), podobný pôvodnému zdrojovému kódu doplnený rôznou notáciou a šípok ktoré reprezentujú vykonávanie jednoduchých cyklov, algoritmov. Tiež sú nahradené kľúčové slová logických operátorov im ekvivalentnými symbolmi. Tento proces si tiež vyžaduje spracovanie syntaxe zdrojového kódu, z ktorého sa vytvorí správna štruktúra a zdrojového kódu a určité časti, kľúčové slová, sa nahradia grafickými značkami podľa určitých šablón.




**Obrázok 4 – Príklady pseudokódu získané s ART**

## 2.6 Dokumentačné programovanie

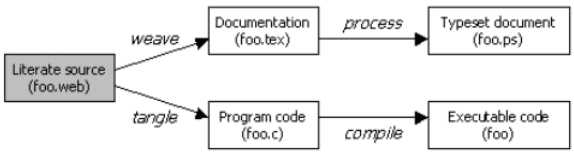
Paradigma dokumentačného programovania, bola zavedená v roku 1984 publikáciou *Literate programming* [6], ktorej autorom je *Donald Knuth* (tiež autor systému TeX). Je to paradigma tvorenia programu spôsobom, aby bol zdrojový súbor kompilovateľný aj čitateľný.

Základnou myšlienkou dokumentačného programovania bolo odkloniť sa od typického písania programov a namiesto toho umožniť programátorom vyvíjať program v poradí ľudskej logiky a nie v poradí kompilátora. Sú v ňom zahrnuté *makrá*, pomocou ktorých je skrytá abstrakcia tradičných zdrojových kódov a makrá vyjadrujú rôzne operácie. Programy vytvorené dokumentačným programovaním sú písané v ľudskom jazyku podobne ako literárne dielo, čo napomáha lepšiemu pochopeniu myšlienok a algoritmu programu. Táto paradigma tiež pridáva literárnu hodnotu programu.

Ďalšie možnosti v rámci dokumentačného programovania sú *formátovanie zdrojového kódu (pretty printing*, podkapitola 2.3*)*, *odkazovanie (cross-referencing)* a *indexovanie*. Spolu s *makrami* sú to prostriedky, ktoré umožňujú lepšiu čitateľnosť dokumentácie a zdrojového kódu. Pomocou *makier* sa nahrádzajú časti kódu, algoritmy. *Indexovanie* a *odkazovanie* slúži na vytváranie odkazov, referencií medzi rôznymi časťami dokumentu, alebo zdrojového kódu [10].

Existuje technika *Spätné dokumentačné programovanie*, ktorá je kombináciou výhod dokumentačného programovania a hypertextového prístupu. Nástroje tohto typu vytvárajú *dokumentačný objekt* vrátane dokumentácie, obrázkov a zdrojového kódu.

Ako prvý zástupca dokumentačného programovania bol jazyk WEB, z ktorého boli odvodené ostatné jazyky (CWEB, noweb, FunnelWeb). Jeho autor je *Donald Knuth*. Je to jazyk, pomocou ktorého sa vytvára dokument *.web* obsahujúci zdrojový kód aj dokumentáciu. Tento dokument slúži ako vstup systému. Formáty výstupu môžu byť rôzne v závislosti od systému. Prvý systém, WEB podporoval výstup TeX (dokumentácia) a Pascal (zdrojový kód). Ako to je naznačené aj na obrázku 5, systém obsahuje dve nástroje: *tangle*, ktorý je zodpovedný za vytvorenie zdrojového kódu, a *wave*, ktorý je zodpovedný za vytvorenie textovej dokumentácie.




**Obrázok 5 – Dva procesy dokumentačného programovania**

Na začiatku sa zameriavalo literárne programovanie na staršie procedurálne jazyky (Pascal, C, Fortran), ale dnes už existujú nástroje na aplikovanie tejto paradigmy na novšie, vyššie programovacie jazyky (Java, C++). Najznámejšie nástroje dokumentačného programovania sú CWEB, nuweb, noweb.

## 2.7 Meranie metrík zdrojového kódu

V tejto podkapitole si objasníme význam merania metrík zdrojového kódu, hlavné typy metrík, ktoré kvantifikujú jednotlivé časti analyzovaného projektu a ich výpočet.

Metriky zohrávajú dôležitú úlohu v každej etape životného cyklu projektu. Pri etape analýzy je vhodné skúmať početnosť logicky oddelených vstupov projektu, počet výstupov, počet dopytov, počet dátových zdrojov, počet rozhraní. Táto metrika sa nazýva *metrika funkčných bodov*. Druhá podobná metóda merania metrík v tejto etape je metóda *Feature Points***,** podľa ktorej sa analyzuje početnosť vstupov, výstupov, dopytov, dátových zdrojov, rozhraní a špeciálnych algoritmov riešiacich hlavné problémy. Pri etape návrhu sa meria zložitosť informačných tokov, cyklomatická zložitosť, úroveň zviazanosti a súdržnosti a tiež parametre používateľského rozhrania [12].

Najdôležitejšia etapa z nášho pohľadu merania metrík je etapa implementácie. Pri tomto meraní analyzujeme projekt na jeho najnižšej úrovni. Tieto metriky sú oveľa presnejšie ako vyššie spomínané metriky, keďže tieto metriky analyzujú projekt na jej najnižšej abstrakcii a to textové súbory obsahujúce zdrojový kód. Najzákladnejšia metrika tejto časti je počet riadkov (**LOC** – Lines of Code). Je to počet riadkov v jednom súbore zdrojového kódu. Z nej sú odvodené ďalšie metriky kvantifikujúce počet špeciálnych riadkov, napríklad:

- počet prázdnych riadkov (**BLANK**),
- počet riadkov s komentárom (**COMM**),
- počet riadkov len s komentárom (**COMM ONLY**),
- počet zdrojových riadkov kódu (**NCNB** – non-comment non-blank) – sú to také riadky zdrojového kódu, ktoré nie sú prázdne a tiež neobsahujú len komentár
- počet riadkov s nevykonávateľnými príkazmi,
- počet riadkov s vykonávateľnými príkazmi (**EXECSTMTS**),
- počet riadkov využívajúci preprocessor (**PREPROCESS**),
- percento komentárov v zdrojovom kóde (**CP**):

$$CP = COMM / (LOC - BLANK)$$

Ďalšie metriky zdrojového kódu sú:

- počet súborov projektu,
- počet tried/modulov projektu,
- počet všetkých funkcií (**NOF**) súboru/projektu,
- počet lokálnych funkcií (**NOLF**) súboru/projektu,
- počet globálnych funkcií (**NOGF**) súboru/projektu,
- cyklomatická zložitosť (**CC** – cyclomatic complexity),
- Halsteadove metriky.

**Cyklomatická zložitosť** – vychádza z reprezentácie štruktúry systému pomocou orientovaného grafu. Z toho dôvodu pre jej výpočet je nutné analyzovať štruktúru programu. Metrika je založená na meraní počtu lineárne nezávislých ciest v grafe. Výpočet je realizovaný funkciou [7] :

$$v(G) = počet\ hrán - počet\ uzlov + 2p$$

**Halsteadove metriky** – Podľa autora týchto metrík, Mauricea Halsteada, je zdrojový kód postupnosť symbolov, pričom symboly môžu byť operátory, alebo operandy. Operátory v rámci zdrojového kódu môžu byť kľúčové slová, volania funkcií, symboly aritmetických operácií, zátvorky, bodkočiarky, atď. Operandy sú zvyčajne premenné, konštanty, definície funkcií. Jednotlivé Halsteadove metriky sa vypočítavajú pomocou 4 premenných: $n_1$, $n_2$, $N_1$, $N_2$. Premenná $n_1$ vyjadruje počet typov operátorov, premenná $N_1$ vyjadruje celkový počet operátorov. Premenná $n_2$ vyjadruje počet typov operandov a premenná $N_2$ vyjadruje celkový počet operandov. Veľkosť programového slovníka označujeme symbolom $n$, ktorý je súčtom symbolov $n_1$ a $n_2$. Dĺžku programu môžeme vypočítať súčtom $N_1$ a $N_2$ a výsledok označujeme symbolom $N$. Ďalšie zložitejšie metriky sú:

- odhad dĺžky programu – $N' = n_1 log_2 n_1 + n_2 log_2 n_2$
- objem programu – $V = N log_2 n$ (minimálny počet bitov na zakódovanie sekvencie $N$ operátorov a operandov)

**Index udržovateľnosti** – je to vlastnosť softvérového systému, ktorá poukazuje na jej modifikovateľnosť, analyzovateľnosť, stabilitu, testovateľnosť. Práve náklady

udržovateľnosti predstavujú 40 až 60 % celkových nákladov počas vývoja softvérového systému. Pomocou nasledujúcej formuly sa dá vypočítať index udržovateľnosti pre jeden modul:

$$MI = 171 - 5.2 * \ln(HV) - 0.23 * CC - 16.2 * \ln(LOC) + 50 * \sin(\sqrt{2.4 * COM})$$

*MI* – index udržovateľnosti (maintainability index)
*HV* – objem programu (Halstead Volume),
*CC* – cyklomatická zložitosť jedného modulu,
*LOC* – počet riadkov kódu jedného modulu,
*COM* – počet komentárov v zdrojovom kóde.

## 2.8 Skriptovací jazyk Lua

Jazyk Lua [3] je skriptovací jazyk s tradičnou syntaxou (syntax jazyka Lua je uvedená v prílohe E). Je multiplatformový, prenosný, efektívny, jednoducho sa môže vkladať do vyšších programovacích jazykov (Java, C/C++, C#, Fortran…). Vykonáva sa na virtuálnom stroji, ktorý používa zásobníkový model. Kombinuje procedurálnu syntax s účinnou konštrukciou dátového reprezentovania, ktorá je založená na asociatívnych poliach a rozšíriteľnej sémantike. Asociatívne polia, fungujú ako jednoduché dátové štruktúry: každá hodnota je sprístupnená pomocou indexu (ktorá môže byť akoukoľvek hodnotou, akéhokoľvek typu). Je umožnené vytváranie modulov, objektov podobných prototypom a triedam, záznamov (*records*), polí, zoznamov a ďalších dátových štruktúr. Jazyk Lua má tiež automatickú správu pamäti, to znamená, že pamäť sa uvoľňuje automaticky pomocou *garbage collector*.

Tieto parametre robia jazyk Lua ideálnu na konfigurovanie, skriptovanie a rapídne prototypovanie [3].

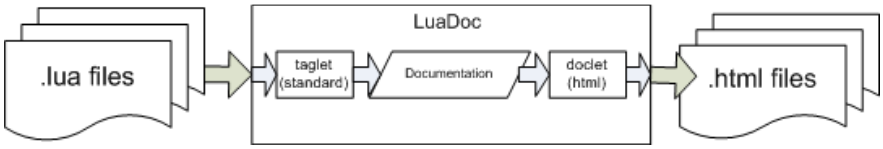
## 2.9 Existujúce nástroje na generovanie dokumentácií

Táto podkapitola obsahuje opis nástroja LuaDoc a tiež najznámejších nástrojov na generovanie dokumentácií ako je Doxygen, JavaDoc, NaturalDocs. Uvedieme hlavné črty uvedených nástrojov, ich vlastnosti, pozitíva, negatíva a príklady výstupných dokumentácií.

### 2.9.1 LuaDoc

LuaDoc[4] bol vyvinutý pre generovanie dokumentácií zo skriptovacieho jazyka Lua a jeho rozšírenie je predmetom tejto bakalárskej práce.

Nástroj LuaDoc sa skladá z dvoch častí. Prvá časť nástroja je zodpovedná za spustenie a druhá časť obsahuje knižnice, v ktorých sú definované funkcie zodpovedné za generovanie dokumentácie. LuaDoc nasleduje model rozloženia balíkov pre Lua 5.1, preto pre ľahší prístup by mal byť správne nainštalovaný v systéme Lua 5.1. Pre fungovanie nástroja je potrebné mať nainštalované knižnice *LuaFileSystem* a *LuaLogging*. Štandardne umožňuje generovanie zo zdrojových kódov napísaných v jazyku Lua a výstupná dokumentácia sa uloží v súboroch HTML. Zo súborov obsahujúce zdrojový kód, sa pomocou komponentu *Taglet* vygeneruje dokumentačný objekt, ktorý obsahuje všetky informácie potrebné pre vytvorenie výslednej dokumentácie. Tento objekt potom prevezme komponent *Doclet*, ktorý naformátuje informácie dokumentačného objektu do výsledného formátu a uloží ako súbory typu HTML (štandard). Celý proces je znázornený na obrázku 6.




**Obrázok 6 – Architektúra generovania dokumentácií pomocou LuaDoc**

Príkazy/*Tagy* nástroja LuaDoc:

`@param` – umožňuje opísať parametre funkcie

`@see` – umožňuje referenciu na iné tabuľky, funkcie

`@return` – umožňuje opísať hodnotu, alebo hodnoty, ktoré vracia funkcia

`@usage` – umožňuje opísať využitie premennej, alebo funkcie

`@description` – umožňuje opísať popis tabuľky, alebo funkcie. Zvyčajne sa odvodí automaticky.

`@name` – umožňuje zadať názov funkcie, alebo názov definície tabuľky. Zvyčajne sa odvodí pri analyzovaní kódu a programátor ho nepotrebuje definovať (ani sa neodporúča zmeniť názov funkcie, aby sa predišlo redundancii)

[4] Dostupné na internete: < http://luadoc.luaforge.net/ >

`@class` – LuaDoc nedokáže rozoznať typ objektu (funkcia, tabuľka, definícia modulu), preto sa to umožňuje zadávať explicitne

`@field` – umožňuje zadať definíciu tabuľky

`@release` – umožňuje zadať reťazec vyjadrujúci verziu modulu, alebo súboru

Príkazy na generovanie dokumentácie zo zdrojového kódu sa zadávajú do príkazového riadku napísaním príkazu *luadoc* a následne zadaním parametrov.

`-d <path>` – nastaví cieľový adresár pre výstup

`-h, --help` – zobrazí pomoc

`--noindexpage` – zruší generovanie súboru „*index*“

`--nofiles` – zruší generovanie dokumentácie zamerané na súbory,

`--nomodules` – zruší generovanie dokumentácie zamerané na moduly

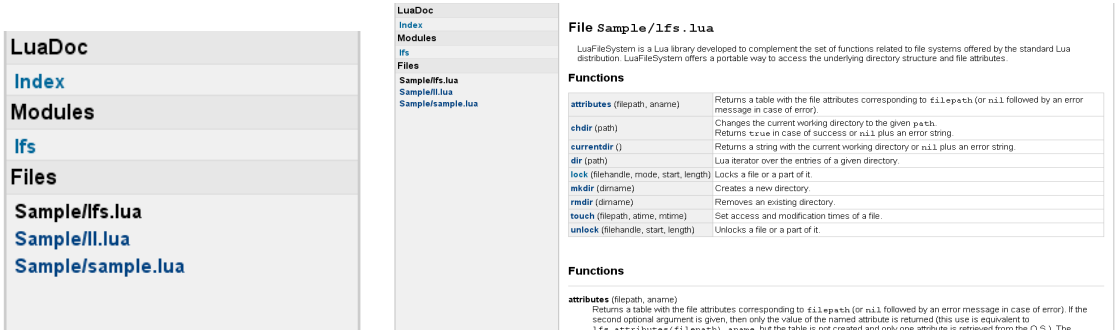
`--doclet <doclet_module>` – nastaví skript, ktorý sa použije ako *doclet*

`--taglet <taglet_module>` – nastaví skript, ktorý sa použije ako *taglet*

`-q, --quiet` – zruší informačné výpisy pri generovaní dokumentácie

`-v, --version` – zobrazí informácie o verzii

Výstupná dokumentácia LuaDoc sa skladá z bočného panelu (Obrázok 7, pravá strana) a z hlavného panelu (Obrázok 7, ľavá strana). Bočný panel zobrazuje hlavné stránky dokumentácie, ako sú index dokumentácie, zoznam jednotlivých modulov a zoznam súborov. Hlavný panel zobrazuje obsah jednotlivých častí dokumentu. Štruktúry dokumentov opisujúce moduly a súbory sú rovnaké. Ako prvý je uvedený opis súboru alebo modulu, ďalej je uvedený zoznam funkcií a tabuliek a nakoniec sú opísané uvedené funkcie, tabuľky.



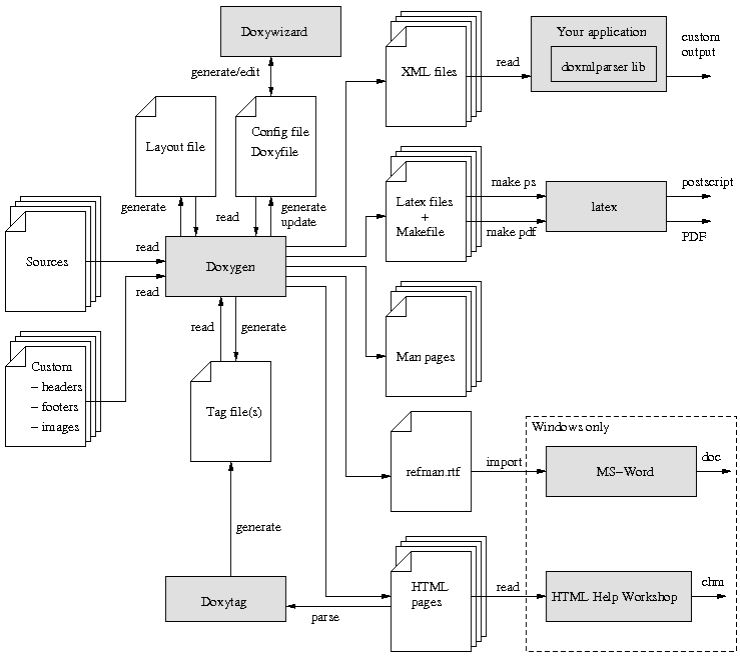

**Obrázok 7 – Dokumentácia generovaná s LuaDoc (naľavo bočný panel, napravo celá obrazovka)**

### 2.9.2 Doxygen

Je to nástroj na generovanie dokumentácií projektov napísaných v programovacom jazyku C, C++, Java, Objective-C, Python, IDL, Fortran, VHDL, PHP, C#, D. Podporuje tiež viacero formátov výstupu, ako napríklad HTML, LaTeX, RTF, MAN, XML a ďalšie. Je to multiplatformový systém (je určený pre Linux, Mac OS X, Windows).

Inštalácia systému na operačný systém môže prebehnúť rôzne. Na operačnom systéme Unix, ako aj na operačnom systéme Windows, môže používateľ nainštalovať binárne súbory danej distribúcie, alebo skompilovať softvér zo zdrojových kódov. Inštalácia nástroja na operačný systém Windows je jednoduchá a intuitívna. Pred inštaláciou nie je potrebné mať nainštalované špeciálne ovládače, knižnice, aplikácie. Po inštalácii je odporúčané[5] nainštalovať knižnicu GraphViz a z dôvodu možnej práce v budúcnosti s dokumentmi typu LaTeX je odporúčané si nainštalovať distribúciu LaTeX-u. Použiť nástroj Doxygen je možné z príkazového riadku programom *doxygen*, ale nástroj tiež obsahuje program *doxywizard* s grafickým používateľským rozhraním, ktorý pripraví konfiguračný súbor pre generovanie dokumentácie a spustí generovanie. Tretí program nástroja Doxygen je *doxytag*, ktorý sa použije len v prípade, ak chce používateľ do generovanej dokumentácie pridať referencie na externý dokument, alebo referenciu na časť obsahu externého dokumentu. Architektúru nástroja a priebeh generovania dokumentácie znázorňuje obrázok 8.

[5] Heesch, V. (2009). Doxygen installation. Dostupné na internete: <http://www.doxygen.nl/install.html>




**Obrázok 8 – Architektúra generovania dokumentácií pomocou Doxygen**

Doxygen ponúka rôzne príkazy, ktoré sa nazývajú „*tag*“. Pomocou nich sa do dokumentácie pridávajú rôzne informácie, ktoré ovplyvňujú výsledný dokument. Tieto príkazy sú umiestnené do zdrojového kódu ako komentáre, aby neovplyvňovali zdrojový kód. Aby systém spoznal jednotlivé príkazy, je treba presne dodržať ich syntax.

Jednotlivé príkazy/*tagy* podľa kategórií:

- **Špeciálne príkazy:** začínajú znakom ‚\‘, alebo ‚@‘.
    - **Indikátory štruktúry** – pridávajú informácie jednotlivým častiam projektu, opisujú jednotlivé členy, premenné, funkcie, súbory, adresáre. Typické *tagy* tejto kategórie sú (bez formátu atribútov): `@addtogroup`, `@callgraph`, `@class`, `@def`, `@dir`, `@enum`, `@example`, `@extends`, `@file`, `@fn`, `@headerfile`, `@implements`, `@interface`, `@mainpage`, `@memberof`, `@name`, `@namespace`, `@package`, `@page`, `@private`, `@protected`, `@public`, `@relates`, `@relatesalso`, `@typedef`, `@union`, `@var`, atď.
    - **Indikátory sekcií** – zodpovedajú za pridávanie informácií ako sú poznámka, meno autora, pridanie, informácie o známych chybách, informácie *TODO*, atď. Typické *tagy* tejto kategórie sú (bez formátu atribútov): `@author`, `@bug`, `@date`, `@deprecated`,

`@details`, `@else`, `@elseif`, `@endcond`, `@endif`, `@exception`, `@if`, `@ifnot`, `@note`, `@param`, `@post`, `@pre`, `@return`, `@retval`, `@see`, `@test`, `@todo`, `@version`, `@warning`, atď.

- **Príkazy na vytváranie vzťahov (linkov)** – umožňujú pridávať referencie na iné časti dokumentu, rozdeľovať dokumentáciu na rôzne sekcie. Typické *tagy* tejto kategórie sú (bez formátu atribútov): `@addindex`, `@anchor`, `@endlink`, `@link`, `@ref`, `@subpage`, `@section`, `@subsection`, `@subsubsection`, `@paragraph`, atď.
- **Príkazy na zobrazovanie príkladov** – umožňujú pridávať do dokumentácie časť zdrojového kódu ako príklad, ukážku. Typické *tagy* tejto kategórie sú (bez formátu atribútov): `@include`, `@includelineno`, `@line`, `@skip`, `@until`, `@verbinclude`, `@htmlinclude`, atď.
- **Príkazy na vizuálne zlepšenia** – umožňujú nastaviť formát niektorých textových častí dokumentácie, ponúkajú funkcie na zlepšenie vzhľadu dokumentácie. Typické *tagy* tejto kategórie sú (bez formátu atribútov): `@a`, `@arg`, `@b`, `@code`, `@copy-doc`, `@dot`, `@msc`, `@e`, `@htmlonly`, `@image`, `@n`, `@p`

- **HTML príkazy:**
  - **HTML príkazy:** `<body>`, `<a href="…">`, `<center>`, `<hr>`, `<ul>`, `<ol>`, `<li>`, `<table>`, `<tr>`, `<td>`, atď.
  - **Špeciálne entity znakov:** ` `, `&copy;`, `&tm;`, `&reg;`, `&lt;`, `&amp;`, `&apos;`, `&quot;`, `&?uml;`, atď.

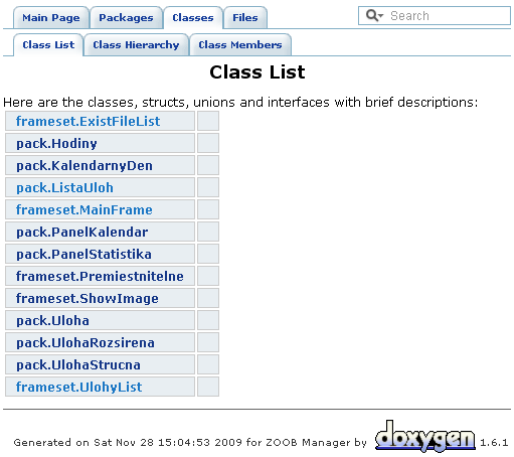
- **XML príkazy:** Doxygen podporuje väčšinu XML príkazov, ktoré sú typické pre jazyk C#.

Ďalšie funkcie systému Doxygen sú:

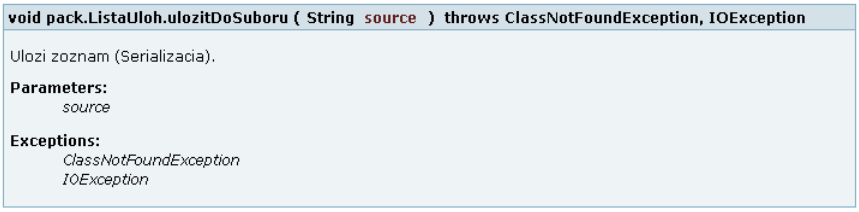
- podporuje dokumentáciu „*namespace*“-ov,
- je kompatibilný s *Javadoc*, *Qt-doc*, *ECMA-334*

- automaticky generuje diagram tried a diagram kolaborácií v dokumentáciách HTML aj LaTeX
- umožňuje pridávať dokumentáciu k hlavičke zdrojového kódu
- generuje zoznam všetkých členov triedy spolu s ich viditeľnosťou (*public, private, protected*)
- do dokumentácií typu HTML pridáva rýchly vyhľadávač pre vyhľadávanie reťazcov, alebo slov v popisoch tried, funkcií a iných členov
- umožňuje pridávanie referencií na rôzne časti dokumentácie, alebo dokumentácie iných projektov
- umožňuje do výslednej dokumentácie pridávať príklady zo zdrojového kódu
- podporuje *syntax highlighting* – zvýrazňovanie zdrojového kódu rôznymi farbami podľa štruktúry kódu
- všetky nastavenia generovania dokumentácie sa konfigurujú v ľahko vytvárateľnom konfiguračnom súbore

Výstupná dokumentácia je hierarchicky rozdelená na štyri časti: hlavná stránka, stránka balíkov, stránka tried a stránka súborov. Hlavná stránka štandardne obsahuje opis projektu, jej verziu a informácie o čase vyhotovenia generovanej dokumentácie. Stránka opisujúca balíky zobrazuje zoznam balíkov a ich opis. Po výbere požadovaného balíka sa zobrazia jednotlivé triedy. Stránka tried (Obrázok 9) zobrazuje všetky triedy projektu a ich členy. Používateľ má možnosť vybrať z troch zobrazení: zoznam tried a ich opisov, hierarchia tried a zoznam členov. Výberom triedy sa zobrazí opis triedy, zoznam jej členov a ich opisov (Obrázok 10). Posledná časť dokumentácie – stránka súborov – zobrazí všetky súbory a výberom súboru sa zobrazia triedy ktoré obsahuje a balík v ktorom sa nachádza.




**Obrázok 9 – Štruktúra dokumentácie generovaná s nástrojom Doxygen**




**Obrázok 10 – Opis funkcie v dokumentácií generovaná s nástrojom Doxygen**

### 2.9.3 Javadoc

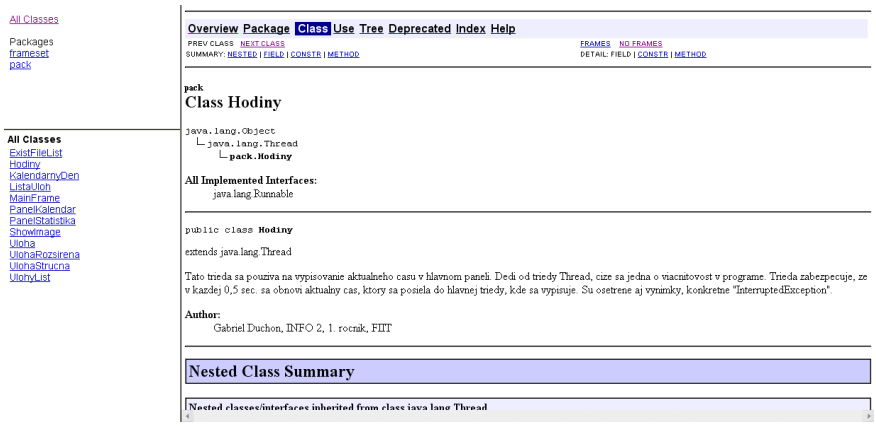
Nástroj Javadoc[6] umožňuje generovanie dokumentácie zo zdrojových kódov napísaných v jazyku Java. Tiež vytvára súbory HTML opisujúce triedy, rozhrania (*interfaces*), metódy a polia (*fields*). Javadoc rozdeľujeme na dve časti: *doclet* a *taglet*.

*Doclet* je program napísaný s *Doclet API*, ktorá špecifikuje obsah a formát generovaný nástrojom Javadoc. *Taglet* je program umožňujúci vytvárať a používať vlastné *tagy*. *Taglet* musí implementovať rozhranie *Taglet*. *Taglet* obsahuje informácie o mene *tag*u, či ide o *vnorený tag* alebo *blokový tag*, kde presne by sa mal nachádzať *tag* v rámci zdrojového kódu (pri konštruktore, pri poli, pri metóde, pri úvode, pri balíku, pri type) a opisuje ako sa má reprezentovať daný *tag* vo výstupnom dokumente. Je voľne dostupná, len ako časť balíku *Java 2 SDK*.

Dokumentácia generovaná s Javadoc je rozdelená na tri rámce (Obrázok 11). V bočnom rámci, ktorá je umiestnená v ľavom hornom rohu stránky, si môže používateľ vybrať či sa majú zobraziť v druhom rámci všetky triedy, alebo len triedy daného balíka. Druhý rámec zobrazuje triedy. Je umiestnený v ľavom dolnom rohu. Pri

[6] Hlavná stránka aplikácie Javadoc je: http://java.sun.com/j2se/javadoc/

výbere triedy z tohto zoznamu sa v hlavnom rámci zobrazí názov triedy, jej umiestnenie triedy v hierarchii tried, zoznam implementovaných rozhraní, textový opis triedy, zoznam jej členov (vnorené triedy, premenné, konštruktor, metódy) a detailný opis členov.



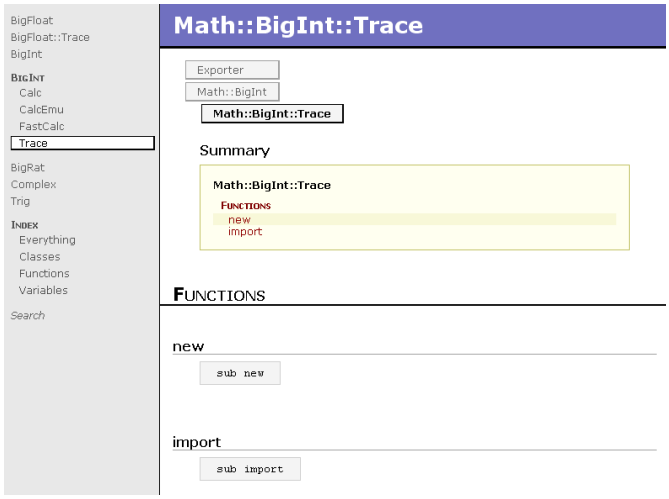

**Obrázok 11 – Dokumentácia generovaná nástrojom Javadoc**

### 2.9.4 NaturalDocs

NaturalDocs[7] je nástroj na generovanie dokumentácií a funguje na tom istom princípe ako predošlé nástroje. Jej hlavné funkcie sú prirodzená syntax v rámci zdrojového kódu, podporuje až deväťnásť programovacích jazykov (bohužiaľ naplno sú podporované len tri: C#, Perl a ActionScript), umožňuje nakonfigurovanie podpory ďalšieho/vlastného jazyka, výber podrobnosti dokumentácie, je kompatibilná so syntaxou Javadoc (podporuje spracovávanie väčšinu *tag*ov), podporuje vyše 200 *tag*ov/kľúčových slov (rozdelené do rôznych kategórií). Naďalej podporuje výber zobrazenia HTML dokumentácie rámcované/nerámcované, umožňuje vyhľadávanie v dokumentácií, čo umožňuje vstavaný vyhľadávač. Čo sa týka samotného programu je prenosný, má rozšíriteľnú architektúru, a je *open source* (s licenciou pod GPL).

Výsledná dokumentácia (Obrázok 12) generovaná s NaturalDocs je podobná dokumentácií generovanej s nástrojom LuaDoc. Bočný panel zobrazuje odkazy na opis jednotlivých súborov, odkaz na zoznam všetkých objektov, odkaz na zoznam všetkých tried, odkaz na zoznam všetkých funkcií, odkaz na zoznam všetkých premenných a vyhľadávač v texte dokumentácie. Dokument opisujúci triedu sa začína zoznamom premenných a funkcií a napokon opisu premenných a funkcií uvedených v zozname.

[7] Hlavná stránka aplikácie NaturalDocs je: http://www.naturaldocs.org/




**Obrázok 12 – Dokumentácia generovaná nástrojom NaturalDocs**

## 2.10 Sumarizácia analýzy nástrojov

Testovaním všetkých štyroch nástrojov som dospel k výsledku, že najprepracovanejší nástroj je Doxygen. Podporuje najviac programovacích jazykov, najviac výstupných formátov a najviac tagov pre opis zdrojového kódu. Javadoc a LuaDoc sú nástroje na generovanie dokumentácie jedného jazyka (Java, Lua), preto nepodporujú generovanie dokumentácií pre viaceré programovacie jazyky. NaturalDocs podporuje generovanie dokumentácie z množstva jazykov, ale len v rámci základnej podpory, čo je generovanie dokumentácie s informáciami, ktoré sú uvedené v komentároch danou syntaxou ktoré akceptuje NaturalDocs. Plná podpora je len pre tri jazyky.

Používateľské rozhranie nástrojov je zvyčajne konzolová, okrem Doxygenu, ktorý podporuje konzolové aj grafické rozhranie.

Jazyk Lua je podporovaná len nástrojom LuaDoc. Avšak NaturalDocs plánuje uviesť podporu pre tento jazyk. LuaDoc je voči ostatným nástrojom najmenej rozvinutý prídavnými funkciami. Nepodporuje vkladanie vzorového zdrojového kódu a z tohto vyplýva, že nepodporuje ani *pretty printing*. Nástroj LuaDoc tiež negeneruje zoznam všetkých funkcií a premenných, len zoznam súborov a modulov (pri generovaní dokumentácie neindexuje jednotlivé premenné a funkcie).

## 2.11 Zhodnotenie analýzy

Z analýzy problému sme sa dozvedeli, že existujú rôzne typy nástrojov, ktoré môžu byť užitočné pri získavaní informácií o zdrojovom kóde a štruktúre projektu. Generovaná

dokumentácia z komentárov zdrojového kódu môže zodpovedať na viaceré otázky „čitateľa“, ku ktorým by sa možno dopracoval len veľmi ťažko, analyzovaním kódu. Meraním metrík systému sa získavajú štatistické dáta o systéme, počet jednotlivých metód, funkcií, premenných, tried, modulov, tabuliek, odhadujú kľúčové body softvérového projektu, pomocou zložitejších formúl sa zisťujú metriky odhadujúce robustnosť algoritmov, udržovateľnosť, slabšie body systému. Samotný zdrojový kód so zvýraznenou syntaxou ponúka najjasnejší, ale aj najkomplikovanejší opis informácií o systéme. Poukázali sme aj na iné paradigmy tvorenia zdrojového kódu a obsahu dokumentácie súčasne – dokumentačné programovanie.

# 3 Opis riešenia

## 3.1 Špecifikácia požiadaviek

Projekt je zameraný na rozšírenie funkcionality nástroja LuaDoc, ktorý má viaceré nedostatky, popísané v podkapitole 2.9.1. Cieľom projektu bude implementovať nasledovné funkcionality pre nástroj LuaDoc, prípadne bude implementovaný samostatný nástroj doplňujúci generátor dokumentácie o nasledujúce funkcie:

- výpis zoznamu všetkých lokálnych a globálnych funkcií,
- výpis hierarchie objektov,
- automatické naformátovanie zdrojového kódu a zvýrazňovanie syntaxe,
- umožnenie používateľovi pridať vlastné tagy,
- vypísanie základných metrík.

Na vytvorenie týchto prídavných funkcií použijeme knižnicu Lpeg a Leg. Použijeme ich, na vytvorenie parserov, ktoré budú parsovať súbory so zdrojovým kódom. Týmto spôsobom vytvoríme *abstraktný syntaktický strom* zdrojového kódu, ktorý použijeme pri generovaní jednotlivých častí dokumentácie.

Cieľom pridania týchto funkcií je zjednodušiť prácu používateľa s vygenerovanou dokumentáciou a sprehľadniť dokument pomocou techník, ktoré sa využívajú v oblasti generovania dokumentácií. Výstup týchto funkcií by sa generoval tiež do výstupného dokumentu nástroja LuaDoc formou HTML kódu.

## 3.2 Návrh

Systém by mal byť rozdelený na viaceré menšie časti, ktoré by samostatne vytvárali funkčné moduly. Vytvorené moduly by mali byť oddelené od nástroja LuaDoc a mali by byť navzájom nezávislé z dôvodu modularizácie. Kombinovaním týchto modulov by sa dosiahla väčšia flexibilita pri generovaní dokumentácie a voľby obsahu dokumentácie. Jednotlivé moduly by mali zabezpečovať funkciu formátovania (zarovnávania), farebného zvýrazňovania, merania metrík a zisťovanie funkcií. Pre jednotlivé moduly by mal byť zabezpečený mechanizmus parsovania, ktorý by mal byť tiež samostatný pre každý hlavný modul. Rozhranie by mohlo byť konzolové, keďže aj samotný nástroj LuaDoc má konzolové rozhranie.

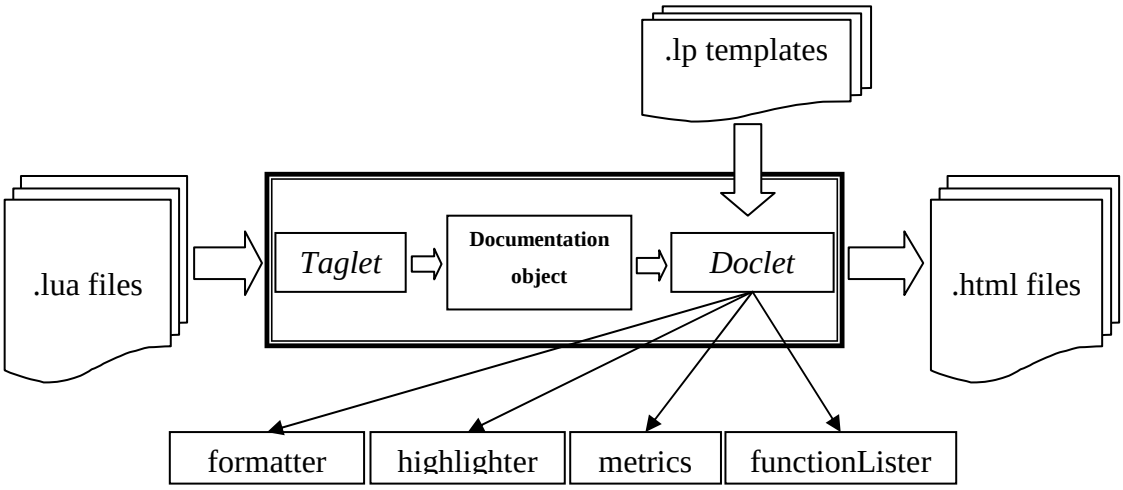
Algoritmus prehľadávania abstraktného syntaktického stromu by mal byť rekurzívny, keďže prehľadávame strom, iteračný cyklus by bol neprehľadný a neelegantný. Funkcia určite bude mať parameter vyjadrujúci strom alebo podstrom, ktorý sa má prehľadávať. Pri prehľadávaní sa musí ošetriť situácia, aby sa nevytvoril cyklus, keďže jednotlivé elementy by sa mohli teoreticky aj opakovať. Pre tieto elementy (ktoré môžu byť rôzneho typu: tabuľka, číslo, reťazec) treba vytvoriť funkciu, ktorá by element serializovala do textového reťazca a tá hodnota by sa ukladala do zoznamu už prehľadaných elementov. Jednotlivé elementy sú vždy reprezentované tabuľkou, ktorá má atribúty: *text*, *tag*, *position* a *data*. K atribútu *text* je priradený reťazec obsahujúci obsah elementu (napr.: celý zdrojový kód, telo funkcie, telo podmienky, reťazec, atď.). Atribút *tag* označuje typ (podľa syntaxe) načítaného elementu. Jednotlivé typy a pravidlá ich syntaxe sú uvedené v prílohe F. Atribút *position* má priradenú číselnú hodnotu na ktorom znaku sa daný element v zdrojovom kóde nachádza. Podľa týchto informácií vieme vytvoriť funkcionalitu, ktorá po rozoznaní jednotlivých typov vykoná nejaké inštrukcie (formátovanie, zvýraznenie). Atribút *data* má priradenú tabuľku všetkých podelementov.

Pseudokód rekurzívneho spracovania abstraktného syntaktického stromu:

```
local ast = parse(filetext)
local output = “”
local nodeText, nodeTag

function processAST(name, node, savedNodes)
  savedNodes = savedNodes or {}
  if type(node) == “table” then
    if not savedNode[node] then
      if nodeTag==“TAG” then
        ... -- main instructions
      end

      if node[“text”] then nodeText = node[“text”] end
      if node[“tag”] then nodeTag = node[“tag”] end
      for attib, value in pairs(node) do
        processAST(serializeName(attib),value,savedNodes)
      end
    end
  end
end
processAST('root',ast)
```

## 3.3 Architektúra systému

Nástroj LuaDoc pozostáva z dvoch častí: *Taglet* a *Doclet*. Keďže spracovanie vstupných súborov ostáva nezmenené, tak pôvodná časť *Taglet* ostane nezmenený. Funkcionalita nových modulov by sa mala zavolať z modulu *Doclet*. Návrh architektúry systému je znázornený na obrázku 13.




**Obrázok 13 – Návrh architektúry systému**

Opis navrhovaných modulov:

[formatter] – úloha tohto modulu je vytvoriť abstraktný syntaktický strom, ktorý by sa rekurzívne prehľadával, vytváral pôvodný zdrojový kód. Súčasne by sa kontrolovali jednotlivé členy syntaxe, a podľa toho by sa riadky zdrojového kódu odsadzovali. (napr.: začiatok/koniec cyklov, podmienok, funkcií, definície tabuliek, atď.). Funkcia modulu je vytvárať syntakticky správne odsadený zdrojový kód.

[functionLister] – úloha tohto modulu je vytvoriť abstraktný syntaktický strom, v ktorom by sa mohli rekurzívne vyhľadať tie uzly, ktoré obsahujú definície funkcií. Nájdené funkcie by sa vracali ako tabuľka, obsahujúca názvy funkcie, ich typ (global/local), parametre a poradové číslo funkcie v zdrojovom kóde.

[highlighter] – pomocou parsera by sa mal vytvoriť úplný abstraktný syntaktický strom, ktorý by tento modul prehľadával a podobným spôsobom ako modul *formatter*, by reprodukoval text zdrojového kódu vo formáte HTML. V prípadoch keby

prehľadávanie našlo člen, ktorý by sa mal farebne, alebo typograficky zvýrazniť, tak by sa aplikovalo zvýraznenie. Typy zvýraznení by mohli byť definované samostatne ako šablóny.

[metrics] – modul by mal zabezpečiť vypočítavanie základných metrík zdrojového kódu.

## 3.4 Implementácia

Projekt je implementovaný v programovacom jazyku Lua (verzia 5.1.4). Naimplementovali sa jednotlivé moduly na vykonávanie formátovania, zvýrazňovania, analyzovania funkcií a vypočítania hlavných metrík systému. Pre tieto hlavné moduly (okrem modulu na vypočítanie metrík) sú naimplementované podmoduly na vykonávanie parsovania.

Knižnicu Leg používame na vytvorenie abstraktného syntaktického stromu, ktorá používa knižnicu LPeg na parsovanie vstupu (obsah súboru).

Projekt by mohol mať grafické používateľské rozhranie, ktoré by ponúkalo možnosť intuitívne používať aplikáciu. Táto funkcia by sa mohla realizovať knižnicou wxLua, ktorý ponúka možnosť vytvárať grafické rozhranie pod rôznymi operačnými systémami (Windows, Linux, Mac OS, Nokia Maemo).

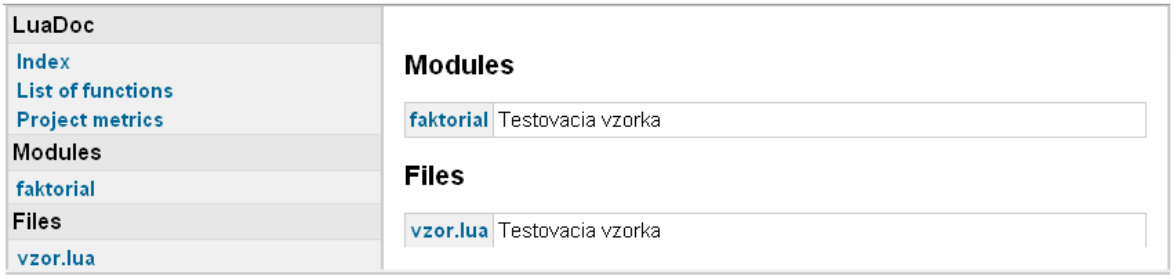
Tiež by sa bola alternatívna možnosť vytvoriť si nástroj na generovanie dokumentácií nanovo, respektíve implementovať knižnicu, ktorú by mohla využívať už existujúci univerzálny nástroj na generovanie dokumentácií (napr. Doxygen, alebo NaturalDocs).

# 4 Overenie riešenia

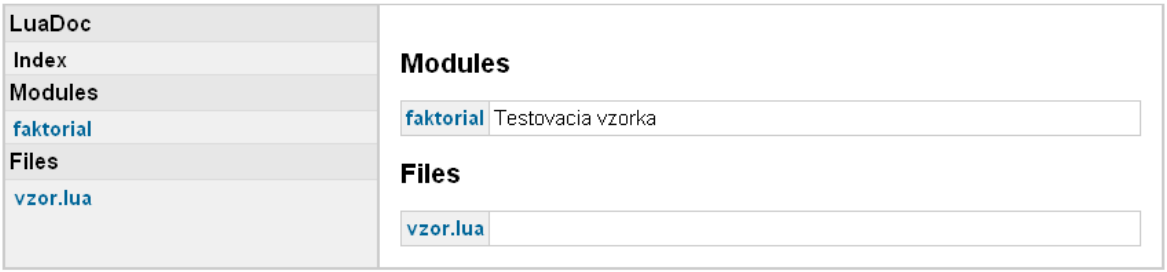
V tejto kapitole si ukážeme funkcionalitu vytvoreného nástroja a zhodnotíme výsledné výstupy nástroja. Ako vstup použijeme nasledovný zdrojový kód:

```
------------------------------
-- Testovacia vzorka
module('faktorial')

------------------------------
-- Funkcia faktorialu
-- @param n vstupna premenna
-- @return funkcna hodnota funkcie pre cislo n
function factorial(n)
if n == 0 then
return 1
else
return n * factorial(n - 1)
end
end

------------------------------
-- Dalsia vzorova funkcia
-- @param tab tabulka
local function localfunc (tab)
for i, textValue in pairs(tab) do --cyklus
textValue  = string.gsub(textValue , "a", "b")
print(i, textval)
end
end
```

Po generovaní dokumentácie s nami rozšírenou verziou nástroja LuaDoc a originálnou verziou dostávame nasledovný výsledok (Obrázok 14, Obrázok 15):

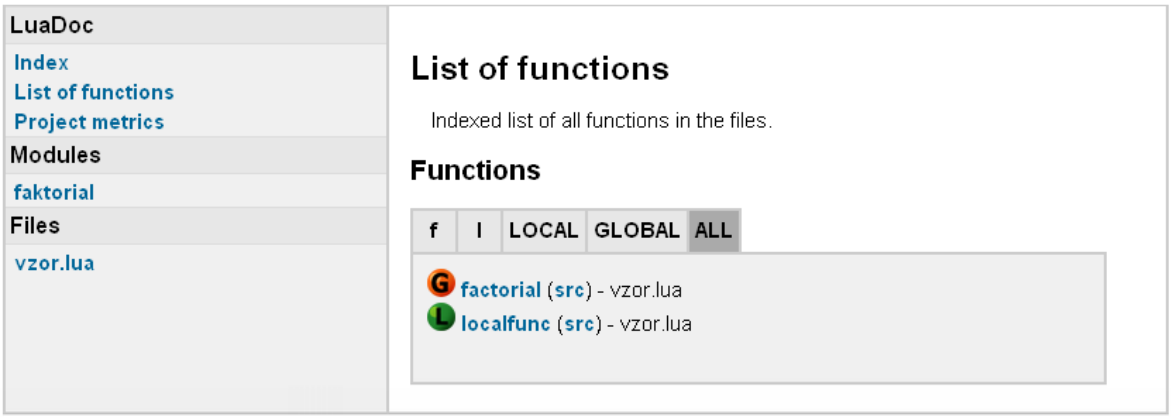



**Obrázok 14 – Index dokumentácie generovaný s rozšírenou verziou nástroja LuaDoc**




**Obrázok 15 – Index dokumentácie generovaný s originálnou verziou nástroja LuaDoc**

Ako vidíme na obidvoch obrázkoch, bočné menu obsahuje odkaz na hlavnú stránku (index), názvy modulov a názvy súborov projektu. Dokumentácia generovaná rozšírenou verziou nástroja LuaDoc obsahuje naviac odkazy na podstránku so zoznamom funkcií (List of functions) a odkaz na podstránku obsahujúcu hlavné metriky projektu (Project metrics).

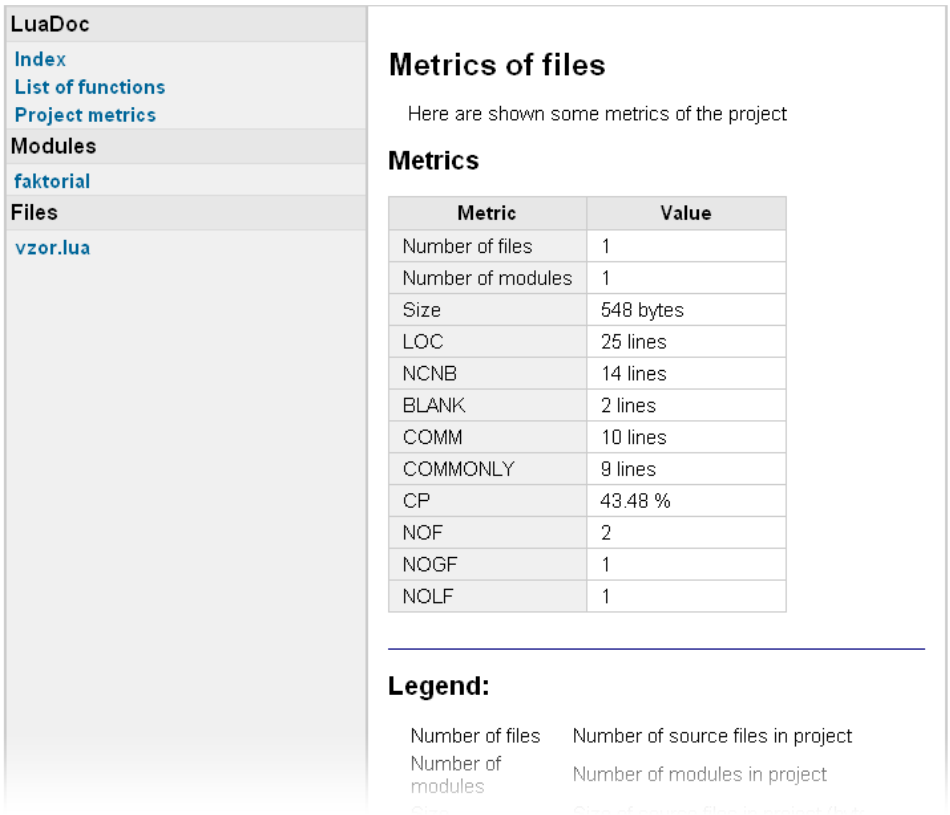



**Obrázok 16 – Podstránka zobrazujúca zoznam funkcií**

Zoznam funkcií (Obrázok 16) pozostáva z panelu[8], ktorý obsahuje zoznam funkcií. Funkcie sú rozdelené najprv podľa začiatočného písmena, ďalej sú uvedené lokálne a globálne funkcie a nakoniec všetky funkcie zoradené podľa abecedy. Ako prvá informácia je v zozname funkcií zobrazený typ funkcie (lokálna, alebo globálna), názov funkcie (je to vlastne odkaz na opis funkcie v rámci dokumentácie súboru), odkaz na telo funkcie v rámci zvýrazneného zdrojového kódu v dokumentácii a nakoniec cesta a názov súboru (zobrazuje sa pre prípad, ak by sa vyskytli duplicitné názvy funkcií).

Dokumentácia vytvorená rozšíreným nástrojom LuaDoc tiež obsahuje tabuľku metrík (Obrázok 17), ktorá zobrazuje hlavné metriky projektu.

[8] Záložkový panel je modifikáciou záložkového panelu, ktorej návod na implementáciu sa nachádza na stránke: http://www.switchonthecode.com/tutorials/javascript-and-css-tutorial-dynamic-tabbed-panels

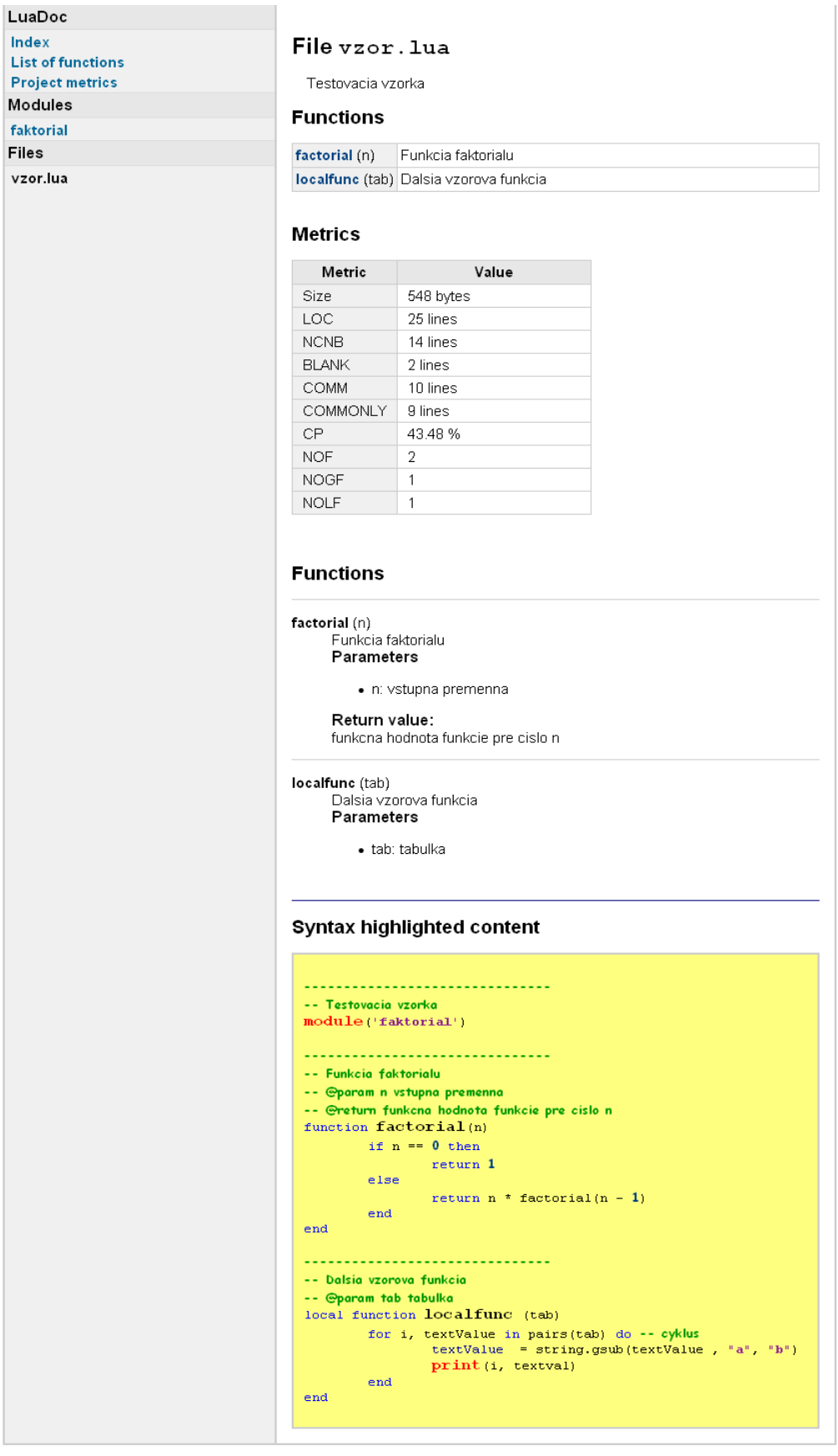



**Obrázok 17 – Podstránka zobrazujúca tabuľku metrík**

Tabuľka obsahuje zoznam názvov metrík a vypočítané metriky pre daný projekt. Tieto metriky sú sumarizáciou pre celý projekt, čiže sú to súčty hodnôt metrík každého súboru so zdrojovým kódom v danom projekte. Skratky metrík vysvetľuje časť *Legenda*, ktorá je umiestnená v dolnej časti podstránky (skratky legendy sú rozpísané v podkapitole 2.7).

Dokumentácia modulov obsahuje informácie tých súborov, ktoré sú definované ako moduly. Tieto výstupy sú ekvivalentné výstupom originálneho nástroja LuaDoc.

Dokumentácia súborov vytvorená originálnym nástrojom LuaDoc obsahuje tabuľkový zoznam funkcií/tabuliek zdrojového kódu a podrobný opis funkcií a tabuliek. Rozšírený LuaDoc vytvára do tejto časti dokumentácie tabuľku metrík analyzovaného súboru a syntakticky zvýraznený zdrojový kód (Obrázok 18). Ako vidíme výsledný zvýraznený zdrojový kód je zarovnaný, zvýraznené sú komentáre, kľúčové slová, definícia funkcií, definícia premenných, volania Lua funkcií, čísla, reťazce.



LuaDoc

Index
List of functions
Project metrics

Modules

faktorial

Files

vzor.lua

# File `vzor.lua`

Testovacia vzorka

## Functions

| | |
|---|---|
| **factorial** (n) | Funkcia faktorialu |
| **localfunc** (tab) | Dalsia vzorova funkcia |

## Metrics

| Metric | Value |
|---|---|
| Size | 548 bytes |
| LOC | 25 lines |
| NCNB | 14 lines |
| BLANK | 2 lines |
| COMM | 10 lines |
| COMMONLY | 9 lines |
| CP | 43.48 % |
| NOF | 2 |
| NOGF | 1 |
| NOLF | 1 |

## Functions

**factorial** (n)
Funkcia faktorialu
**Parameters**

- n: vstupna premenna

**Return value:**
funkcna hodnota funkcie pre cislo n

**localfunc** (tab)
Dalsia vzorova funkcia
**Parameters**

- tab: tabulka

## Syntax highlighted content

```
------------------------------
-- Testovacia vzorka
module('faktorial')

------------------------------
-- Funkcia faktorialu
-- @param n vstupna premenna
-- @return funkcna hodnota funkcie pre cislo n
function factorial(n)
        if n == 0 then
                return 1
        else
                return n * factorial(n - 1)
        end
end

------------------------------
-- Dalsia vzorova funkcia
-- @param tab tabulka
local function localfunc (tab)
        for i, textValue in pairs(tab) do -- cyklus
                textValue  = string.gsub(textValue , "a", "b")
                print(i, textval)
        end
end
```

XHTML 1.0

**Obrázok 18 – Dokumentácia vzorového súboru**

Podľa tabuľky metrík súbor má veľkosť 548 bajtov, čo môžeme ľahko skontrolovať pomocou operačného systému, ktorý umožňuje zobraziť hlavné vlastnosti súboru. Počet riadkov je 25, čo môžeme jednoducho skontrolovať manuálnym spočítaním riadkov, alebo použitím textového editora. Počet zdrojových riadkov kódu je 14. Sú to tie riadky, ktoré nie sú prázdne a neobsahujú len komentár. Ďalšie metriky sa týkajú komentárov. Zo zvýrazneného zdrojového kódu je zrejmé, že počet komentárov je 10 a z toho je jeden taký, ktorý má pred sebou inštrukciu, čiže nepatrí do kategórie komentárov **COMM ONLY**. Ďalej je ešte uvedený pomer komentárov k zdrojovým riadkom kódu podľa vzorca:

$$CP = COMM / (LOC – BLANK)$$

$$CP = 10 / (25 – 2) = 0,43478 \rightarrow 43,48\%$$

Nakoniec je ešte uvedený počet funkcií, počet globálnych funkcií a počet lokálnych funkcií. Ako vidíme zdrojový kód obsahuje dve funkcie, lokálnu funkciu *localfunc* a globálnu funkciu *factorial*.

# 5 Zhodnotenie

Špecifikovaná a navrhnutá funkcionalita bola implementovaná ako služby jednotlivých modulov. Pre tieto moduly bolo vytvorené rozhranie, ktoré zabezpečuje generovanie dokumentácie, respektíve textových súborov, ktoré obsahujú rôzne typy informácií o zdrojových kódoch. Tiež sme rozšírili existujúci nástroj na generovanie dokumentácií, LuaDoc, ktorého zdrojový kód bolo nutné zmeniť v minimálnom rozsahu, aby vygenerovaná dokumentácia obsahovala informácie získané našimi modulmi. Informácie získavame prehľadávaním abstraktného syntaktického stromu, ktorý vytvárame knižnicou Leg.

Hlavný výsledný nástroj, rozšírený LuaDoc, by sa mohol využiť na generovanie dokumentácií, keďže zobrazuje viac informácií o zdrojovom kóde ako pôvodný LuaDoc. Rozšírený LuaDoc má časovú zložitosť generovania dokumentácií horšiu ako pôvodný, keďže pri každom spracovaní (formátovanie, zvýrazňovanie, analýza funkcií) sa vytvorí abstraktný syntaktický strom, ktorý sa následne rekurzívne prehľadáva. Výhoda tohto riešenia je v jej modulárnosti; skúsenejší používateľ si vie ľahko odobrať nežiaduce moduly z rozšíreného LuaDoc bez toho, aby musel jednotlivé moduly meniť (keďže sú samostatne fungujúce).

## 5.1 Známe obmedzenia

Pri implementácií a testovaní som zistil nasledovné obmedzenia:

- knižnica Leg označuje syntax cyklickej inštrukcie *while* v nasledovnej forme

  **while** *a* **<=** *b* **do** … **end**

  ako chybnú (len pri použití logickej operácie „<=“ a „>=“)
- chýba ošetrenie nezvyčajných stavov (ako vstup projektu sa predpokladá správny zdrojový kód napísaný v jazyku Lua)
- výstup do formátu bol optimalizovaný len pre najpoužívanejšie prehliadače (Mozilla Firefox, Opera, Internet Explorer, Google Chrome), je možné, že pre staršie prehliadače bude výstup zobrazený nesprávne
- štruktúra dokumentácie vygenerovanej nástrojom LuaDoc je chybná v prípade, ak vstupné súbory majú dlhé názvy, veľa podpriečinkov. Ich názov je v pravom menu buď nie je absolútne zalomené (zoznam modulov), alebo pri zalomení (zoznam súborov) vyzerá menu neprehľadne.

- systémová Lua knižnica ***io***, ponúkajúca čítanie a zapisovanie do súborov a ktorú používa LuaDoc aj na kopírovanie súborov, ktoré nie sú zdrojové kódy (CSS súbory obsahujúce štýly HTML dokumentácie) nedokáže načítať súbor, ktorý obsahuje ASCII znak s hodnotou v šestnástkovej sústave `1A`. Pri načítavaní obsahu takéhoto súboru pri načítaní tohto znaku ukončí načítavanie predčasne. Kopírovanie obrázkov PNG do cieľového adresára som vyriešil spôsobom, že sa pomocou systémovej funkcie ***os.execute*** zavolá inštrukcia kopírovania operačného systému. Táto funkcia nemusí byť totožná v každom operačnom systéme.

## 5.2 Možné smery ďalšieho rozvíjania

Vytvorený projekt by sa ešte mohol ďalej rozširovať o nové funkcie ako sú:

- analyzovanie ďalších metrík,
- pridanie ďalších tagov na definovanie špeciálnych dát v rámci komentárov,
- pridanie možnosti formátovania zdrojového kódu spôsobom, aby sa mohol presne parametrizovať formát jednotlivých blokov (napríklad formát definovania premenných, alebo formát if-then-else blokov, atď.)
- vylepšenie rozvrhu jednotlivých častí výsledného HTML dokumentu,
- generovanie grafov a diagramov do výslednej dokumentácie,
- možnosť generovať dokumentáciu aj do iných formátov, napr. PDF, RTF, atď.

Ohľadne metrík by bolo vhodné doplniť metriky napríklad cyklomatickej zložitosti, a metriky týkajúce sa premenných. Modifikovať by sa mohla štruktúra výstupného dokumentu, aby bola prehľadná, normalizovaná aj pre dlhé názvy súborov a balíkov.

# Zoznam použitej literatúry


[1] COLLBERG, Christian S., PROEBSTING, Todd. Language-Agnostic Program Rendering for Presentation, debugging and visualization. In *Proceeding* 2000 IEEE International Symposium on Visual Languages. IEEE Computer Society, 2000. Dostupné na internete: <http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergDaveyProebsting2000b/A4.pdf >

[2] DEURSEN, Arie Van, KUIPERS, Tobias. Building documentation generators. In *Proceedings*; IEEE International Conference on Software Maintenance. IEEE Computer Society, 1999. Dostupné na internete: <http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R9916.pdf>

[3] IERUSALIMSCHY, Roberto, DE FIGUEIREDO, Luiz Henrique, CELES, Waldemar. Lua 5.0 Reference Manual. Tecgraf, PUC-Rio, 2003. Dostupné na internete: <http://www.lua.org/ftp/refman-5.0.pdf>

[4] IERUSALIMSCHY, Roberto. A Text Pattern-Matching Tool based on Parsing Expression Grammars. In *Software – Practice and Experience*. 2008. Dostupné na internete: <http://www.inf.puc-rio.br/~roberto/docs/peg.pdf>

[5] JONGE, Merijn De. Pretty-Printing for Software Reengineering. In *Proceedings*: International Conference on Software Maintenance. IEEE Computer Society Press, 2002. ISBN: 0-7695-1819-2. Dostupné v digitálnej knižnici ACM: <http://portal.acm.org/>

[6] KNUTH, Donald E. Literate programming. In *The Computer Journal*. 1984, vol 27. Dostupné na internete: <http://www.literateprogramming.com/knuthweb.pdf>

[7] MILLS, E. Software Metrics – SEI Curriculum Module SEI-CM-12-1.1. Seattle University, 1988. Dostupné na internete: <http://www.sei.cmu.edu/reports/88cm012.pdf>

[8] NIKITA SYNYTSKYY, James, CORDY, James R., DEAN, Thomas. Robust Multilingual Parsing Using Island Grammars. In *Conference of the Centre for*

*Advanced Studies on Collaborative Research*, 2003. Dostupné na internete: <http://research.cs.queensu.ca/home/cordy/Papers/CASCON03_MultiParsing.pdf>

[9] OPPEN, Derek C. Pretty Printing. Stanford University Stanford, CA, USA, 1979. Dostupné na internete: <ftp://db.stanford.edu/pub/cstr/reports/cs/tr/79/770/CS-TR-79-770.pdf>

[10] SMITH, Matthew. Towards Modern Literate Programming. In *Honours Project Report*. 2001. Dostupné na internete: <http://www.cosc.canterbury.ac.nz/research/reports/HonsReps/2001/hons_0110.pdf>

[11] TILLEY, SCOTT R. Documenting-in-the-large vs. Documenting-in-the-small [online]. IEEE Transactions on Software Engineering. 1993. Dostupné v digitálnej knižnici ACM: <http://portal.acm.org/>

[12] TRGO, V. Meranie v softvérovom inžinierstve a eseje o manažmente softvérových projektov: Meranie v etape implementácie. Bratislava: Slovenská technická univerzita, 2001. s. 111-117. Dostupné na internete: <http://www2.fiit.stuba.sk/~bielik/courses/msi-slov/kniha/2001/meranie.pdf>

[13] VISSER, E. Syntax Definition for Language Prototyping: PhD thesis. Academisch Proefschrift, University of Amsterdam, 1997. ISBN 90-74795-75-7. Dostupné v digitálnej knižnici CiteSeerX: <http://citeseerx.ist.psu.edu/>

# Príloha A – Technická dokumentácia

## A.1 Dokumentácia k špecifikácii

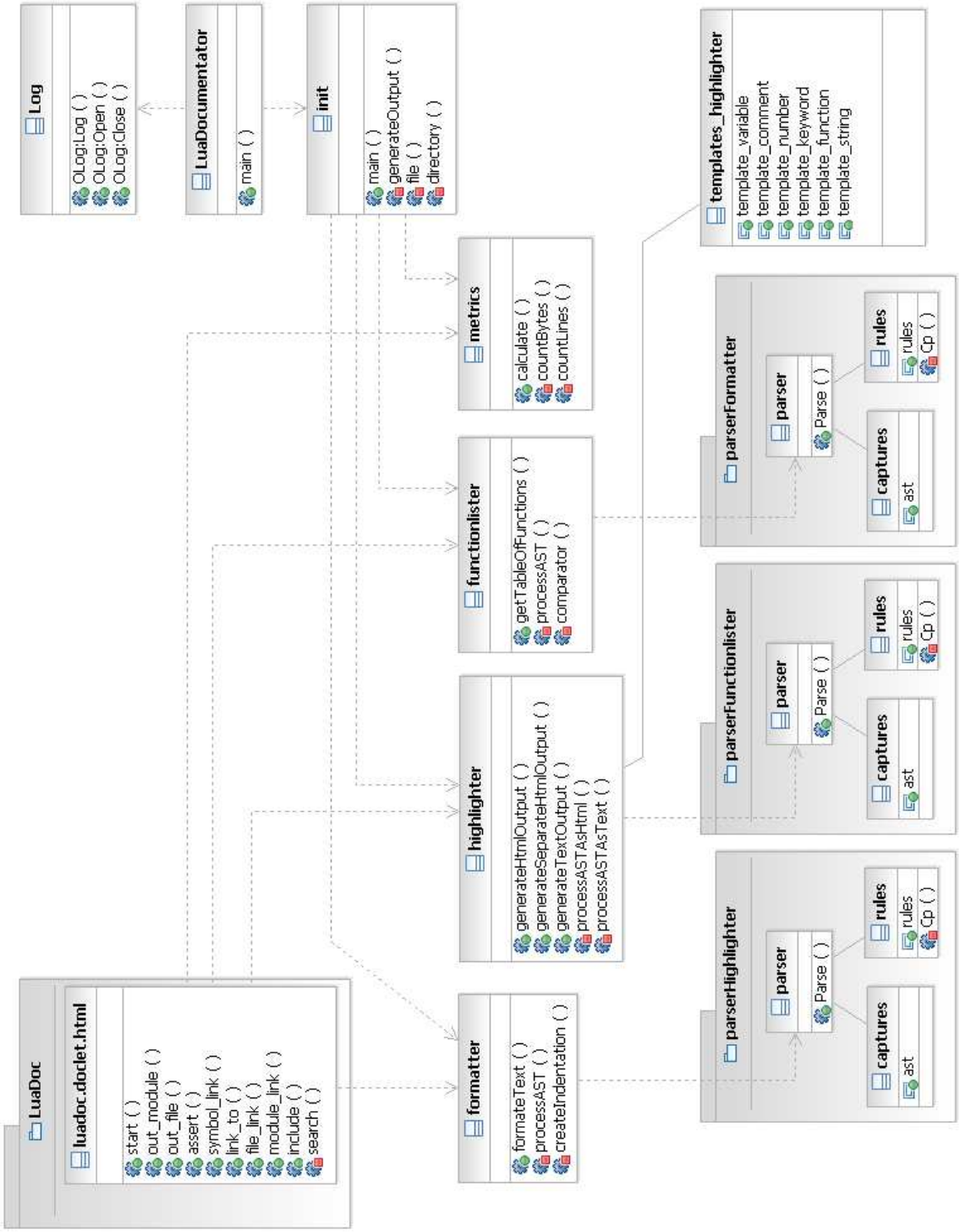
Presná špecifikácia požiadaviek:

- vytvorenie abstraktného syntaktického stromu zo zdrojového kódu,
- analýza a výpis indexu globálnych a lokálnych funkcií zdrojového kódu,
- odkazovanie zo zoznamu funkcií na tie časti zvýrazneného zdrojového kódu, kde sú funkcie definované,
- zvýrazňovanie syntaxe zdrojového kódu,
- formátovanie (zarovnávanie) zdrojového kódu,
- dosadenie naimplementovaných funkcií do nástroja LuaDoc,
- umožnenie parametrizovania formátu výstupu,
- meranie základných metrík projektu a jednotlivých súborov zdrojového kódu.

## A.2 Dokumentácia k návrhu

Projekt pozostáva zo štyroch hlavných modulov, ktoré vykonávajú hlavnú funkcionalitu. Sú to moduly *formatter*, *highlighter*, *functionLister*, *metrics*. Moduly *formatter*, *highlighter* a *functionLister* používajú parsovanie na vytvorenie AST, preto je pre každý z nich navrhnutý balík obsahujúci tri moduly: *parser*, *captures*, *rules*. Modul *captures* obsahuje tabuľku definícií vzoriek a modul *rules* obsahuje tabuľku definícií pravidiel. Pri parsovaní definície v týchto tabuľkách prekonávajú pôvodné definície knižnice Leg.

Tieto štyri moduly rozširujú funkcionalitu pôvodného nástroja LuaDoc a používajú sa z modulu *luadoc.doclet.html*. Navrhnutý je tiež aj prototyp rozhrania (*LuaDocumentator*), ktorý umožňuje vytvárať dokumenty informácií, ktoré sú výstupmi jednotlivých modulov nezávisle od nástroja LuaDoc. Architektúra projektu je znázornená diagramom tried na obrázku 19.




**Obrázok 19 – Diagram tried navrhovaného projektu**

## A.3 Dokumentácia k implementácii

Vytvorený nástroj potrebuje nasledovné knižnice na vykonávanie funkcií:

- lua 5.1.4
- luafilesystem 1.4.2
- luasocket 2.0.2
- lpeg 0.8.1
- leg 0.1.2
- cosmo 9.09.22
- strict

Pre správne zobrazenie zoznamu funkcií je nutné mať v prehliadači nainštalovaný a povolený JavaScript, keďže zoznam funkcií vo vygenerovanej dokumentácií je panel vytvorený týmto skriptovacím jazykom.

### A.3.1 Opis funkcie `processASTAsHtml` modulu `highlighter`

Funkcia `processASTAsHtml` vykonáva spracovanie abstraktného syntaktického stromu, ktorá je tabuľka reprezentujúca koreňový uzol AST a obsahuje zoznam uzlov potomkov, ktoré sú tiež vyjadrené tabuľkami. Týmto dostávame systém tabuliek, ktorú prehľadáva nasledujúca rekurzívna funkcia a podľa atribútov *tag* aplikuje rôzne zvýrazňovanie (CSS štýly) definované v module `templates_highlighter`.

```
local function processASTAsHtml(name, node, alreadySaved)
  alreadySaved = alreadySaved or {} -- initial value of saved tables
  if type(node) == "table" then -- we care only of tables
    if not alreadySaved[node] then -- pretention against cyklics

      alreadySaved[node] = name -- save name for next time

      if node['position'] and type(node['position'])=='number' then
        -- if the offset of text position was changed
        if offset < node['position'] then
          -- conversion of problematic char to they HTML equivalent
          nodeText = string.gsub(nodeText,"&", "&amp;");
          nodeText = string.gsub(nodeText,"<", "&lt;");
          nodeText = string.gsub(nodeText,">", "&gt;");
          output = output .. cosmo.fill(htmltemplate,{content =
            nodeText, fcnID = functionID}) -- appenging new output
          htmltemplate = "$content" -- reset formatting template
          offset = node['position'] -- reset new offset
        end
      end
```

```
if node['text'] then nodeText = node['text'] end
if node['tag'] and node['text'] then nodeTag = node['tag']
  if nodeTag == 'LocalFunction' then prepareForFunction=true end
  if nodeTag == 'GlobalFunction' then prepareForFunction=true end
  if nodeTag == 'Var') then
    htmltemplate = templates_highlighter.template_variable end
  if nodeTag == 'COMMENT' then
    htmltemplate = templates_highlighter.template_comment end
  if nodeTag == 'NUMBER' then
    htmltemplate = templates_highlighter.template_number end
  if nodeTag == 'keyword' then
    htmltemplate = templates_highlighter.template_keyword end
  if nodeTag == 'STRING' then
    htmltemplate = templates_highlighter.template_string end
  if nodeTag == 'FuncName' then
    htmltemplate = templates_highlighter.template_functionCall end
  if nodeTag == 'FuncName' and prepareForFunction then
    htmltemplate = templates_highlighter.template_function;
    functionID = nodeText;
    prepareForFunction = false;
  end
  if (nodeTag == 'Name')and prepareForFunction then
    htmltemplate = templates_highlighter.template_function;
    functionID = nodeText; prepareForFunction = false;
  end
  if (nodeTag == 'FunctionCall') then
    local added = false
    for k,val in pairs(luafcn) do
      -- check if function is Lua function
      local foundfcn = 0;
      if string.find(nodeText,val..'[(]')==1 then foundfcn=1; end
      if string.find(nodeText,val..' ')==1 then foundfcn=1; end
      if string.find(nodeText,val..'["]')==1 then foundfcn=1; end
      if string.find(nodeText,val..'\n')==1 then foundfcn=1; end
      if string.find(nodeText,val..'\t')==1 then foundfcn=1; end
      if string.find(nodeText,val.."[']")==1 then foundfcn=1; end
      if foundfcn==1 then
        htmltemplate = templates_highlighter.template_LuaFunction
        output = output .. cosmo.fill(htmltemplate,{content=val})
        offset = offset + string.len(val)
        htmltemplate = "$content"
        added = true
        break
      end
    end
    if not added then
      htmltemplate = templates_highlighter.template_functionCall
    end
  end
end

for k,val in pairs(node) do -- recursion
  local fieldname = string.format("[%s]", basicserialize(k))
```

```
            processASTAsHtml(fieldname, val, alreadySaved)
          end
        end
      end
    end
```

### A.3.2 Opis funkcie `processAST` modulu `functionLister`

Funkcia `processAST` vykonáva tiež spracovanie abstraktného syntaktického stromu. Rekurzívne prehľadáva systém tabuliek reprezentujúci AST a pri nájdení atribútu *tag* s hodnotou *LocalFunction*, alebo *GlobalFunction*, zistí dáta funkcie a uloží do tabuľkového zoznamu `fcntable`, ktorú na konci funkcie vracia.

```
local function processAST(name, node, fcntable, alreadySaved)
  if type(node) == "table" then -- we care only of tables
    alreadySaved = alreadySaved or {}
    if not alreadySaved[node] then -- pretention against cyklics
      alreadySaved[node] = name -- save name for next time
      if node['text'] then
        nodeText = node['text']
      end
      if node['tag'] then
        nodeTag = node['tag'] -- save tag to variable tag
        if (nodeTag == 'LocalFunction') then
          functionStart = true
          local i = table.getn(fcntable)
          fcntable[i+1] = {}
          fcntable[i+1].fcntype = "local"
          fcntable[i+1].name = " "
          fcntable[i+1].attrib = " "
          fcntable[i+1].ID = i+1       -- ID = <1, 2, 3, ...>
        end
        if (nodeTag == 'GlobalFunction') then
          functionStart = true
          local i = table.getn(fcntable)
          fcntable[i+1] = {}
          fcntable[i+1].fcntype = "global"
          fcntable[i+1].name = " "
          fcntable[i+1].attrib = " "
          fcntable[i+1].ID = i+1       -- ID = <1, 2, 3, ...>
        end
        if (nodeTag == 'Name') then
          if functionStart then
            local i = table.getn(fcntable)
            fcntable[i]['name'] = nodeText
          end
        end
        if (nodeTag == 'FuncBody') then
          if functionStart then
            local i = table.getn(fcntable)
            local x = string.find(nodeText,")")
            fcntable[i]['attrib'] = string.sub(nodeText, 1, x)
            functionStart = false
          end
        end
```

```
      end
      for k,val in pairs(node) do      -- recursion
        processAST(tostring(k), val, fcntable, alreadySaved)
      end
    end
  end
  return fcntable
end
```

### A.3.3 Zmenená funkcia `start` modulu `luadoc.doclet.html`

Funkcia `start` v rámci modulu `luadoc.doclet.html` zabezpečuje generovanie výstupných súborov formátu HTML. Ako parameter dostáva premennú `doc`, ktorá obsahuje dokumentačný objekt obsahujúci všetky informácie dokumentácie. V prvej časti sa vytvorí index, v druhej dokumentácie modulov, tretia časť obsahuje modifikácie, ktorými sa pridávajú výstupy modulov k informáciám potrebným k vytvoreniu dokumentácie súborov. Ďalej sa ešte vytvorí index funkcií a stránka s tabuľkou metrík projektu. V poslednom kroku sa skopírujú súbory do vybraných priečinkov (obrázky, CSS súbory, JS súbor).

Vybrané informácie dokumentačného objektu a vlastné informácie získané vlastnými modulmi sa predávajú šablónam, ktoré sú formátu HTML, ale tiež obsahujú vykonateľný Lua kód, ktorá pri generovaní dokumentácie generuje dynamický obsah dokumentu.

```
-- additional modules
local functionlister = require ('ldoc.functionlister')
local highlighter = require ('ldoc.highlighter')
local formatter = require ('ldoc.formatter')
local metricslister = require ('ldoc.metrics')
local pkio = require ('pk.io')

-----------------------------------------------------------------
-- Generate the output.
-- @param doc Table with the structured documentation.

function start (doc)
  -- Generate index file
  -- < uchanged source code >

  -- Process modules
  -- < uchanged source code >

  local tableOfFunctions = {};
  local tableOfMetrics = {};

  -- Process files
  if not options.nofiles then
    for _, filepath in ipairs(doc.files) do
```

```
      local file_doc = doc.files[filepath]
      local text = pkio.ReadFile(filepath)
      local indented_text = formatter.formateText(text)
      -- set prettyprinted text of file_doc
      file_doc.prettyprint =
        highlighter.generateHtmlOutput(indented_text)
      -- calculate main metrics
      local metricinfo = metricslister.calculate(filepath)
      -- appending function informations to the tableOfFunctions
      for _, funcinfo in
        pairs(functionlister.getTableOfFunctions(text,true)) do
        funcinfo.path = filepath -- set path
        -- set metrics about functions
        metricinfo.NOF = metricinfo.NOF + 1
        if funcinfo.fcntype == "global" then
          metricinfo.NOGF = metricinfo.NOGF + 1 end
        if funcinfo.fcntype == "local" then
          metricinfo.NOLF = metricinfo.NOLF + 1 end
        table.insert(tableOfFunctions,funcinfo)
      end
      table.insert(tableOfMetrics,metricinfo)
      print("processing: "..filepath)
      -- assembly the filename
      local filename = out_file(file_doc.name)
      logger:info(string.format("generating file `%s'", filename))
      local f = lfs.open(filename, "w")
      assert(f, string.format("could not open `%s' for writing",
        filename))
      io.output(f)
      include("file.lp",{
        doc = doc,
        file_doc = file_doc,
        metricinfo = metricinfo })
      f:close()
    end
end

-- FUNCTIONS
table.sort(tableOfFunctions,functionlister.comparator)
local functions = { name = "index.html" }
local f = lfs.open(options.output_dir ..
  "functionlist/index.html", "w")
assert(f, string.format("could not open functionlist/index.html
  for writing"))
io.output(f)
include("indexOfFunctions.lp", {
  doc = doc,
  functions = functions,
  tableOfFunctions = tableOfFunctions })
f:close()

-- METRICS
local metrics = { name = "index.html" }
local f = lfs.open(options.output_dir.."metrics/index.html","w")
assert(f, string.format("could not open metrics/index.html for
  writing"))
io.output(f)
include("indexOfMetrics.lp", {
  doc = doc,
  metrics = metrics,
  tableOfMetrics = tableOfMetrics,
```

```
      modulenum = #doc.modules,
      filenum = #doc.files })
  f:close()

  -- copy extra files
  -- < uchanged source code >
  -- copy file tabbedpanel.css to functionlist/
  -- copy file tabbedpanel.js to functionlist/
  -- copy file global.png to functionlist/
  -- copy file local.png to functionlist/

end
```

# Príloha B – Návod na používanie

**Inštalácia:**

Na spustenie aplikácie je nutné mať nainštalované prostredie jazyka Lua, resp. na elektronickom médiu je verzia spúšťacieho programu Lua skriptov pre operačný systém Windows XP/Vista/7. Inštalácia aplikácie zahŕňa len kopírovanie aplikácie na používateľom vybrané miesto.

**Spustenie:**

Doplnená verzia nástroja LuaDoc sa používa tým istým spôsobom ako originálna verzia. Opis používania a ponúkané prepínače sú nasledovné:

```
Usage: luadoc.lua [options|files]
Generate documentation from files. Available options are:
  -d path                       output directory path
  -t path                       template directory path
  -h, --help                    print this help and exit
      --noindexpage             do not generate global index page
      --nofiles                 do not generate documentation for files
      --nomodules               do not generate documentation for
modules
      --doclet doclet_module    doclet module to generate output
      --taglet taglet_module    taglet module to parse input code
  -q, --quiet                   suppress all normal output
  -v, --version                 print version information
```

Príklad na použitie:

```
luadoc –d outputdir –q luaprojekt
```

Vytvorený nástroj LuaDocumentator má nasledovný opis používania:

```
Usage: LuaDocumentator.lua [options|files]
Documentation generator tools. Available options are:
  -dest {path}          existing output directory path (default is
"\.")
  -indent               generate indented source code
  -html                 generate highlighted HTML output
  -html_indent          generate indented highlighted HTML output
  -ast                  generate abstract syntactic tree (time-
consuming)
  -functionlist         add list of functions to the HTML output
  -quiet                suppress all log messages to the console
  -verbose              allow all log messages to the console
  -version              print version information
  -help                 print this help and exit
```

Príklad na použitie:

```
luadocumentator –dest outputdir –indent –html_indent
–functionlist -verbose
```

# Príloha C – Obsah elektronického média

Zoznam adresárov a súborov a ich opis:

| | |
|---|---|
| `Aplikacia/` | adresár obsahujúci kópiu programu a potrebné knižnice na spustenie celej aplikácie, |
| `Dokumentacia/` | adresár obsahujúci dokumentáciu bakalárskej práce, |
| `Priklady_pouzitia/` | dokumentácia projektu vytvorená rozšíreným nástrojom, |
| `Zdrojove_kody/` | obsahuje zdrojové kódy projektu, |
| `readme.txt` | súbor popisujúci obsah elektronického média a návod na inštalovanie a spustenie aplikácie. |

# Príloha D – Konštruktory jazyka LPeg

LPeg poskytuje konštruktory na vytváranie vzoriek:

```
letter = lpeg.R(“az”)
digit = lpeg.R(“09”)
alphanum = letter + digit
```

Táto technika je prebraná z programovacieho jazyka SNOBOL. Hlavné konštruktory sú:

- `lpeg.R(“az”)` – rozpätie znakov
- `lpeg.S(“xyz”)` – množina
- `lpeg.P(“slovo“)` – slovo
- `lpeg.P(číslo)` – slovo, ktoré obsahuje presný počet znakov
- `lpeg.V(...)` – vytvorenie neterminálov
- `P1 + P2` – spojenie výrazov v danom poradí
- `P1 * P2` – zreťazenie
- `-P` – negácia
- `P1 – P2` – P1, ak nevyhovuje P2
- `P^n` – najmenej n opakovaní
- `P^-n` – najviac n opakovaní

LPeg nemá implicitné vyhľadávanie. Vyhľadávanie je vyjadrené v rámci vzoriek. Vzorky vytvárajú hodnoty založené na zhodách.

- `lpeg.C(vzorka)` – zachytáva zhodu
- `lpeg.P(vzorka)` – zachytáva pozíciu zhody
- `lpeg.Cc(hodnoty)` – zachytáva hodnoty
- `lpeg.Ct(vzorka)` – vytvára zoznam vhniezdených zhôd
- `lpeg.Ca(vzorka)` – zhromažďuje vhniezdené zhody

Metódy substituovania:

- `lpeg.Cs(vzorka)` – zachytáva zhody s vhniezdenými zhodami nahradenými ich hodnotami
- `vzorka / reťazec` – vracia *reťazec*, v ktorom sú nahradené zhody danou vzorkou
- `vzorka / tabuľka` – vracia hodnotu *tabuľka[c]*, kde *c* je prvá zhoda vzorky
- `vzorka / funkcia` – aplikuje funkciu na zhody

# Príloha E – Syntax jazyka Lua

Syntax jazyka Lua vyjadrená pomocou EBNF[9] notácie [3]:

*chunk* → { *stat* [ ‘;’ ] }
*block* → *chunk*
*stat* → *varlist1* ‘=’ *explist1*
| *functioncall*
| **do** *block* **end**
| **while** *exp* **do** *block* **end**
| **repeat** block **until** *exp*
| **if** *exp* **then** *block* { **elseif** *exp* **then** *block* } [ **else** *block* ] **end**
| **return** [ *explist1* ]
| **break**
| **for** Name ‘=’ *exp* ‘,’ *exp* [ ‘,’ *exp* ] **do** *block* **end**
| **for** Name { ‘,’ Name } **in** *explist1* **do** *block* **end**
| **function** *funcname funcbody*
| **local function** Name *funcbody*
| **local** *namelist* [ *init* ]
funcname → Name { ‘.’ Name } [ ‘:’ Name ]
varlist1 → *var* { ‘,’ *var* }
var → Name | *prefixexp* ‘[’ *exp* ‘]’ | *prefixexp* ‘.’ Name
namelist → Name { ‘,’ Name }
init → ‘=’ *explist1*
explist1 → { *exp* ‘,’ } *exp*
exp → **nil** | **false** | **true** | Number | Literal
| *function* | *prefixexp* | *tableconstructor* | *exp binop exp* | *unop exp*
prefixexp → *var* | *functioncall* | ‘(’ *exp* ‘)’
functioncall → *prefixexp args* | *prefixexp* ‘:’ Name *args*
args → ‘(’ [ *explist1* ] ‘)’ | *tableconstructor* | Literal
function → **function** *funcbody*
funcbody → ‘(’ [ *parlist1* ] ‘)’ *block* **end**
parlist1 → Name { ‘,’ Name } [ ‘,’ ‘...’ ] | ‘...’
tableconstructor → ‘{’ [ *fieldlist* ] ‘}’
fieldlist → *field* { *fieldsep field* } [ *fieldsep* ]
field → ‘[’ *exp* ‘]’ ‘=’ *exp* | *name* ‘=’ *exp* | *exp*
fieldsep → ‘,’ | ‘;’
binop → ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’ | ‘..’
| ‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘==’ | ‘~=’
| **and** | **or**
unop → ‘-’ | **not**

[9] EBNF – Extended Backus Naur Form

# Príloha F – PEG pravidlá opisujúce gramatiku jazyka Lua (štruktúra AST[10])

Tabuľka pravidiel, podľa ktorých sa zachytávajú zhody počas parsovania a vytvára sa abstraktný syntaktický strom obsahujúci uzly reprezentujúce časti zdrojového kódu. Nasledujúci kód je prebraný z oficiálnej stránky[11] nástroja Leg.

```
rules = {
   -- See peculiarities below
   IGNORED  = scanner.IGNORED  -- used as spacing, not depicted below
   EPSILON = lpeg.P(true)
   EOF     = scanner.EOF  -- end of file
   BOF     = scanner.BOF  -- beginning of file
   Name    = ID

   -- Default initial rule
   [1]     = CHUNK
   CHUNK   = scanner.BANG^-1 * Block

   Chunk   = (Stat * ';'^-1)^0 * (LastStat * ';'^-1)^-1
   Block   = Chunk

   -- STATEMENTS
   Stat          = Assign + FunctionCall + Do + While + Repeat + If
                 + NumericFor + GenericFor + GlobalFunction + LocalFunction
                 + LocalAssign
   Assign        = VarList * '=' * ExpList
   Do            = 'do' * Block * 'end'
   While         = 'while' * Exp * 'do' * Block * 'end'
   Repeat        = 'repeat' * Block * 'until' * Exp
   If            = 'if' * Exp * 'then' * Block
                     * ('elseif' * Exp * 'then' * Block)^0
                     * (('else' * Block) + EPSILON)
                     * 'end'
   NumericFor    = 'for' * Name * '='
                     * Exp * ',' * Exp * ((',' * Exp) + EPSILON)
                     * 'do' * Block * 'end'
   GenericFor    = 'for' * NameList * 'in' * ExpList * 'do' * Block * 'end'
   GlobalFunction = 'function' * FuncName * FuncBody
   LocalFunction = 'local' * 'function' * Name * FuncBody
   LocalAssign   = 'local' * NameList * ('=' * ExpList)^-1
   LastStat      = 'return' * ExpList^-1
                 + 'break'

   -- LISTS
   VarList  = Var * (',' * Var)^0
   NameList = Name * (',' * Name)^0
   ExpList  = Exp * (',' * Exp)^0

   -- EXPRESSIONS
   Exp          = _SimpleExp * (BinOp * _SimpleExp)^0
   _SimpleExp   = 'nil' + 'false' + 'true' + Number + String + '...' + Function
                + _PrefixExp + TableConstructor + (UnOp * _SimpleExp)
   _PrefixExp   = ( Name                  a Var
                  + _PrefixExpParens      only an expression
                  ) * (
                      _PrefixExpSquare    a Var
                    + _PrefixExpDot       a Var
                    + _PrefixExpArgs      a FunctionCall
                    + _PrefixExpColon     a FunctionCall
                  ) ^ 0
```

[10] AST – Abstract syntactic tree (Abstraktný syntaktický strom)

[11] Dostupné na internete: < http://leg.luaforge.net/parser.html#section_The_Grammar >

```
    -- Extra rules for semantic actions:
    _PrefixExpParens = '(' * Exp * ')'
    _PrefixExpSquare = '[' * Exp * ']'
    _PrefixExpDot    = '.' * ID
    _PrefixExpArgs   = Args
    _PrefixExpColon  = ':' * ID * _PrefixExpArgs

    -- These rules use an internal trick to be distingished from _PrefixExp
    Var              = _PrefixExp
    FunctionCall     = _PrefixExp

    -- FUNCTIONS
    Function     = 'function' * FuncBody
    FuncBody     = '(' * (ParList+EPSILON) * ')' * Block * 'end'
    FuncName     = Name * _PrefixExpDot^0 * ((':' * ID)+EPSILON)
    Args         = '(' * (ExpList+EPSILON) * ')'
                 + TableConstructor + String
    ParList      = NameList * (',' * '...')^-1
                 + '...'

    -- TABLES
    TableConstructor = '{' * (FieldList+EPSILON) * '}'
    FieldList        = Field * (FieldSep * Field)^0 * FieldSep^-1
    FieldSep         = ',' + ';'

    -- Extra rules for semantic actions:
    _FieldSquare     = '[' * Exp * ']' * '=' * Exp
    _FieldID         = ID * '=' * Exp
    _FieldExp        = Exp

    -- OPERATORS
    BinOp    = '+' + '-' + '*' + '/' + '^' + '%' + '..'
             + '<' + '<=' + '>' + '>=' + '==' + '~='
             + 'and' + 'or'
    UnOp     = '-' + 'not' + '#'

    -- ...plus scanner's keywords and symbols
}
```