Dokumentace k bakalářské práci

Srovnání efektivity různých programovacích jazyků při práci s automaty

Ondřej Polanský 25.05.2020

Obsah

Obsah paměťového média a návod k použití

Tato kapitola popisuje způsoby práce s programy a zdrojovými kódy k bakalářské práci Srovnání efektivity různých programovacích jazyků při práci s automaty. Paměťové médium obsahuje tyto položky:

- **Prelozene_programy** – obsahuje přeložené programy a skripty použité při tvorbě práce. Dále obsahuje množinu vybraných testovacích automatů pro ukázku činnosti.
- **Zdrojove_kody** – obsahuje zdrojové kódy programů.
- **Dokumentace** – tento soubor. Obsahuje návod k použití.
- **Ostatni_materialy** – obsahuje Excelové tabulky a grafy použité ke zpracování výsledků.
- **Bakalarka_tex** – obsahuje zdrojové soubory potřebné pro vytvoření výsledného pdf práce.
- **xpolan09-Automatove-algoritmy** – text práce ve formátu pdf.

Způsob spuštění:

V adresáři *Prelozene_programy* jsou umístěny spustitelné verze programů, konkrétně jsou v podadresářích *c++*, *csharp*, *csharp_mono*, *ocaml* a *python*. Struktura tohoto adresáře by se neměla měnit. Spustitelnost programů byla testována na školním serveru Merlin (http://merlin.fit.vutbr.cz/). Jazyk C++ funguje bez problémů. Jazyk python vyžaduje, aby byl spuštěn příkazem python3. Jazyk C# má dvě ekvivalentní verze. První z nich je verze přeložená pomocí programu Visual studio 2019 (podadresář *csharp*). Ta byla přeložena a publikována přímo pro unixový operační systém, neměl by tedy být problém s jejím spuštěním klasickým způsobem. Druhá verze (podadresář *csharp_mono*) je určena pro program mono a přeložena je pomocí programu mcs. Pro její spuštění je však vyžadována instalace programu mono, která na serveru Merlin není. V této práci byla pro výpočty použita jen první verze, mono-verze C# je zde navíc pro případ, že by se náhodou vyskytly problémy při jejím spouštění. Pokud to bude možné, doporučuji používat pouze verzi v podadresáři *csharp*. Jazyk OCaml je problematický, jelikož na Merlinovi není nainstalovaný. Při pokusu o spuštění bohužel hlásí chybějící knihovny. Spuštění této implementace je tak pravděpodobně možné jen na počítači s odpovídající instalací překladače tohoto jazyka.

Spustitelné programy jsou v C++, C# (verze Visual studio) a OCamlu pojmenovány „bakalarka“, v Pythonu je příslušný skript pojmenován „bakalarka.py“ a v C# verze mono je program pojmenován bakalarka_mono.exe. Programy očekávají automat nebo několik automatů (ve formátu tymbuk; ve složce automata jsou předpřipraveny jednoduché automaty pro spuštění) na standardním vstupu, výsledek vypisují na standardní výstup. Zde jsou uvedeny příklady spuštění jednotlivých programů pro každý jazyk (z příslušného podadresáře) z příkazové řádky unixového operačního systému:

C++

./bakalarka [-alg] <"path_to_automaton"

příklad: ./bakalarka –e <../automata/A1

Python (Je třeba spustit python 3, na Merlinovi je třeba použít příkaz "python3".)

python3 bakalarka.py [-alg] <"path_to_automaton"

příklad: python3 bakalarka.py –d <../automata/A2

C# (podadresář *csharp*)

./bakalarka [-alg] <"path_to_automaton"

příklad: ./bakalarka –m <../automata/A1

C# (podadresář *csharp_mono*)

mono bakalarka_mono.exe [-alg] <"path_to_automaton"

příklad: mono bakalarka_mono.exe –p <../automata/intersection1

OCaml

./bakalarka [-alg] <"path_to_automaton"

příklad: ./bakalarka –s <../automata/A2

Pro více automatů na vstupu lze použít následující příkaz:

cat "path_to_automaton1" "path_to_automaton2" | ./bakalarka [-alg]

nebo vybrat nějaký předpřipravený soubor s přesným počtem automatů, určený k testování daného algoritmu.

Pokud nastane problém s oprávněním, lze použít příkaz chmod, např.:

chmod 777 bakalarka

Možnosti –alg (přepíná jednotlivé algoritmy; povinný parametr):

- **-e** spustí algoritmus test prázdnosti (vyžaduje 1 automat)
- **-n** spustí algoritmus odstranění zbytečných stavů (vyžaduje 1 automat)
- **-p** spustí algoritmus pro výpočet produktu (vyžaduje 2 automaty se stejnými abecedami)
- **-d** spustí algoritmus pro determinizaci (vyžaduje 1 automat)
- **-m** spustí algoritmus pro minimalizaci (vyžaduje 1 automat)
- **-s** spustí algoritmus pro výpočet relace simulace (vyžaduje 1 automat)
- **-u** spustí algoritmus test univerzality s relací simulace (vyžaduje 1 automat)
- **-ui** spustí algoritmus test univerzality s relací identity (vyžaduje 1 automat)
- **-i** spustí algoritmus test inkluze s relací simulace (vyžaduje 2 automaty se stejnými abecedami)
- **-ii** spustí algoritmus test inkluze s relací identity (vyžaduje 2 automaty se stejnými abecedami)
- **-o** spustí algoritmus pro výpočet sjednocení (vyžaduje 2 automaty se stejnými abecedami) – tento algoritmus je zde uvedený nad rámec zadání této práce
- **-x** spustí sekvenci algoritmů (vyžaduje 3 automaty se stejnými abecedami)

V podadresáři *automata* se nachází množina souborů, ve kterých jsou uloženy jednotlivé automaty v požadovaném formátu. Většina souborů je pojmenována tak, aby bylo poznat, k testování kterých algoritmů jsou určeny. Ke každému algoritmu jsou předpřipraveny přibližně čtyři soubory. Každý takový soubor obsahuje správný počet automatů, potřebných pro daný algoritmus. Tyto soubory je tedy možné dodat programům na standardní vstup. Algoritmy, které pracují s více automaty, obvykle vyžadují automaty se stejnou abecedou. Pokud automaty nebudou mít stejnou abecedu, neměly by výpočty končit chybou, ale zároveň není zaručena správnost výsledku.

Dále se v tomto adresáři pro úplnost nachází pomocné skripty, které byly použity pro automatizaci a zpracování výsledků:

measure-script - unixový skript pro automatizované měření výsledků. Spouští programy jednotlivých algoritmů a jazyků pro všechny automaty v podadresáři *automata*. Vytváří adresář *results*, ve kterém jsou jednotlivé výsledky uloženy. Tento skript by měl za předpokladu umístění v adresáři *Prelozene_programy* fungovat, avšak je nejspíš lepší spouštět jednotlivé experimenty ručně. Tento skript pro dané množství automatů obvykle trvá asi pět minut, zahltí však člověka velkým množstvím výsledků.

xml-script.py – skript v jazyce Python 3, který transformuje výsledky v adresáři *results* do formátu .xml a vytváří adresář *results_xml*. Navazuje na *measure-script*.

pdfcrop-script – unixový skript pro ořezání bílých okrajů obrázků (grafů), uložených ve formátu .pdf. Použit pro úpravu Excelovských grafů.

Formát *Timbuk*, ve kterém jsou uloženy automaty:

<file> : 'Ops' <label_list> <automaton> <automaton> ...
<label_list> : <label_decl> <label_decl> ... // seznam label_decl
<label_decl> : string ':' int // deklarace symbolu (jméno písmena abecedy a jeho arita)
<automaton> :
'Automaton' string 'States' <state_list> 'Final States' <state_list> 'Transitions' <transition_list>
<state_list> : <state> <state> ... // seznam stavů
<state> : string // jméno stavu
<transition_list>: <transition> <transition> ... // seznam přechodů
<transition> : <label> '(' <state> ',' <state> ',' ... ')' '->' <state> // přechod
<label> : string // jméno label

Příklad:
Ops a:0 b:1 c:1

Automaton A
States q0 q1 q2
Final States q2
Transitions
a -> q0
b(q0) -> q1
c(q1) -> q1
c(q1) -> q2

c(q2) -> q2

Způsob překladu:

Zdrojové kódy pro překlad jsou uloženy v adresáři *Zdrojove_kody*, pojmenování podadresářů je stejné, jako v předchozím případě. Pro jazyky C++, C# (mono-verze) a OCaml je připraven Makefile. Pro překlad C# (verze Visual studio) je potřeba zmíněný program. V podadresáři *csharp* je uložen celý projekt, který je možno zkopírovat do adresáře projektů příslušné instalace Visual studia. Skript jazyka Python se překládat nemusí. Problém může nastat u jazyků C# mono-verze (vyžaduje program mcs) a OCaml (vyžaduje program ocamlopt). Ani jeden z těchto programů bohužel není nainstalovaný na Merlinovi. C++ vyžaduje program g++, který na Merlinovi je.

C++

- Ve složce se nachází Makefile, který příkazem make přeloží zdrojový kód. Překlad byl testován na unixovém systému programem g++.

Python

- Nepřekládá se.

C# (podadresář *csharp*)

- Obsahuje C# projekt pro Visual studio.

C# (podadresář *csharp_mono*)

- Obsahuje zdrojový kód pro mono-verzi C# a Makefile, který jej přeloží pomocí programu mcs.

OCaml

- Ve složce se nachází Makefile, který příkazem make přeloží zdrojový kód. Překlad byl testován na unixovém systému programem ocamlopt. Na Merlinovi chybí.

Obecné

Celkem byly implementovány algoritmy ve čtyřech jazycích – C++, Pythonu, C# a OCamlu. Byla snaha z důvodu objektivního srovnání algoritmy ve všech jazycích implementovat co možná nejpodobněji. Použité verze jazyků byly C++17, Python 3.7.6, C#8.0 (.NET Core 3.1), OCaml 4.08.1. Byly použity pouze standardní knihovny jazyků. C++, Python a OCaml byly implementovány na operačním systému unix, na kterém poté byly i měřeny výsledky. C# byl implementován, přeložen a publikován na OS Windows v programu Visual studio 2019, na unixu byla poté měřena publikovaná verze specificky pro tento OS.

Všechny čtyři programy po spuštění se správnými argumenty načtou automat nebo více automatů (formátovaných ve stylu Timbuk – bližší popis na http://www.fit.vutbr.cz/research/groups/verifit/tools/sa/.cs) ze standardního vstupu (stdin) a následně na ně aplikují vybraný algoritmus, specifikovaný argumentem. Výsledný automat

poté vypíší na standardní výstup (stdout). Měří se jen doba provádění algoritmu. Podrobné informace o měření, principech algoritmů a odůvodnění použití datových struktur a postupů při implementaci jsou uvedeny v textu bakalářské práce. V této dokumentaci je popsán hlavně obsah zdrojových kódů a význam jednotlivých částí pro lepší orientaci.

Datové struktury

Tato kapitola byla převzata z hlavního textu práce.

Pro všechny algoritmy byly použity stejné datové struktury reprezentující konečné automaty, ne všechny algoritmy však využívají všechny její proměnné.

Pro uchování konečných automatů byly použity dvě základní struktury, moduly nebo třídy (podle jazyka). První z nich, nazvaná *FA*, reprezentuje samotný konečný automat a obsahuje proměnné:

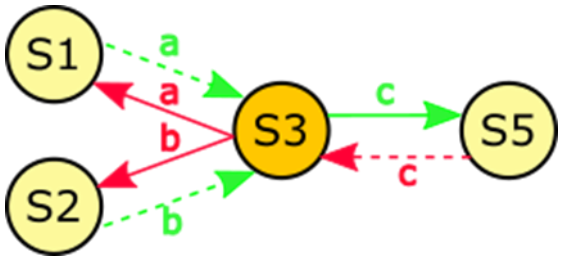
- **name** – Řetězec určující jméno KA.
- **states** – Neseřazená množina stavů KA (struktur *State*).
- **alphabet** – Seřazená množina určující abecedu KA. Za písmeno abecedy se považuje jakýkoli řetězec. Jelikož žádný z implementovaných algoritmů nemění abecedu KA, je množina seřazena pouze jednou při načítání KA a není tedy třeba vynakládat žádné další prostředky pro udržení seřazenosti množiny v průběhu provádění algoritmů. Seřazení této množiny umožňuje rychlé ověření rovnosti abeced dvou automatů, což je využíváno algoritmy pracujícími se dvěma KA. Jedinou výjimku představují algoritmy, které tvoří nový automat, a tedy i vytváří novou množinu abecedy. To je ovšem zajištěno pomocí mělké nebo hodnotové (C++) kopie celé množiny.
- **start_states** – Neseřazená množina počátečních stavů. Žádný z implementovaných algoritmů v této množině nevyhledává, pouze jí iteruje, není tedy třeba množinu udržovat seřazenou. Ke zjištění, zda je stav počáteční, slouží speciální proměnná ve stuktuře *State*.
- **final_states** – Neseřazená množina koncových stavů. Stejná funkce jako množina start_states.

Druhá z nich, nazvaná *State*, reprezentuje stav z množiny Q v KA a obsahuje proměnné:

- **name** – Řetězec určující jméno stavu. Jméno stavu je jeho jedinečný identifikátor v rámci daného KA.
- **transit_states** – Množina dvojic klíč-hodnota (písmeno abecedy - odkaz na stav) v C++ nebo množina dvojic klíč-seznam hodnot (písmeno abecedy – množina odkazů na stavy). Obsahuje odkazy na následnické stavy dosažitelné z daného stavu pomocí jednotlivých písmen abecedy. Musí být optimalizovaná pro velké množství vyhledávání, ovšem nepředpokládá se, že bude příliš velká, neboť obsahuje pouze přechody z jednoho stavu.
- **reversed_transit_states** – Obdoba množiny transit_states pro reverzní přechody.
- **start_st** – Booleovská proměnná indikující, že je stav počáteční.
- **final_st** – Booleovská proměnná indikující, že je stav koncový.
- **flag, card** – Pomocné proměnné, využívané některými algoritmy.

Struktura FA, na rozdíl od matematické definice KA, neobsahuje množinu přechodů mezi stavy δ. Důvodem je zjištění, že implementované algoritmy po zpracování aktuálního stavu vyhledávají jeho následníky, tedy stavy, do kterých z aktuálního stavu vede přechod. Není

tedy třeba procházet všechny přechody automatu, stačí projít jen přechody vycházející z daného stavu. Je oprávněné se domnívat, že takováto úprava výrazně zvýší efektivitu algoritmů při práci s KA s velkým množstvím přechodů a stavů, neboť urychlí dvě základní operace prováděné algoritmy – nalezení všech následníků daného stavu a nalezení následníků daného stavu dostupných daným písmenem abecedy. Množina přechodů δ tedy byla distribuována do jednotlivých stavů, kdy každý stav obsahuje množinu odkazů na následnické stavy a množinu odkazů na předchozí stavy (stejný princip pro množinu reverzních přechodů δ^r). Vznikla tak obousměrně propojená síť stavů (obr. 1). Tato modifikace sice mírně zvyšuje náročnost úprav KA (přidávání a odebírání stavů) pro některé algoritmy, konkrétně algoritmy odebírání zbytečných stavů a minimalizace KA, panuje však přesvědčení, že zisk převyšuje ztráty. Jako možné rozšíření této práce se nabízí ověření, zdali je tomu opravdu tak.



Obrázek 1: Síť stavů. Zeleně vyznačeny přechody, červeně reverzní přechody. Plná čára – přechody stavu S3, přerušovaná čára – přechody okolních stavů.

Celková myšlenka této reprezentace KA spočívá v tom, že jednotlivé stavy se všemi svými proměnnými budou uloženy pohromadě na jednom místě v množině *states* a všechny ostatní proměnné struktur FA a State se budou do této množiny odkazovat. V závislosti na implementačním jazyce byly pro odkazy použity ukazatele (C++) nebo princip referenčního typu spojený s automatickou správou paměti (ostatní jazyky). Tento přístup nejenže šetří paměť a odstraňuje redundanci jednotlivých stavů, ale hlavně řeší problém se zacyklením v případě přechodů typu pa→p. Z tohoto důvodu musí být množina *states* neseřazená a pořadí prvků se nesmí měnit. Tento požadavek nemá žádný negativní dopad na efektivitu, protože v množině stavů žádný zadaný algoritmus nevyhledává.

Množiny start_states a final_states ve struktuře FA neslouží k ověřování, zdali je stav počáteční nebo koncový, k tomuto účelu jsou určené booleovské proměnné start_st a final_st, které obsahuje každý stav zvlášť. Tento přístup sice přidává menší redundanci, avšak zajišťuje jak rychlé ověření, jestli je stav počáteční nebo koncový, bez nutnosti prohledávání obou množin, tak také rychlou iteraci přes počáteční a koncové stavy bez nutnosti procházení všech stavů. Obě tyto operace jsou algoritmy hojně využívány.

V jazyce C++ byl pro neseřazené množiny použit vektor, jedinou výjimkou je množina stavů *states*, která v implementaci s ukazateli musí být lineárně zřetězeným seznamem kvůli nutnosti odebírat a přidávat stavy bez porušení platnosti odkazů ostatních stavů. Vektor má konstantní časové složitosti pro vyhledání (*O(1)*), vkládání (*O(1)*) a odstranění (*O(n)*). Lineárně zřetězený seznam má konstantní časové složitosti pro vkládání a odstraňování prvků. Jelikož se v množině stavů nikdy nevyhledává, nevadí ani lineární složitost vyhledávání. Množina abecedy je reprezentována seřazeným vektorem, vektor je řazen pouze jednou při načítání automatu ze souboru mimo měřené úseky. Množiny přechodů jsou implementovány pomocí multimapy, jelikož hashovací tabulka v C++ nezachovává pořadí vložených prvků a je složitější vyhledat konkrétní stav pro konkrétní písmeno abecedy.

Nabízela se i možnost vytvořit hashovací tabulku množin odkazů na stavy, ale nakonec byla vybrána jednodušší a specializovaná třída multimap. Vyhledání, vkládání a odstranění prvků má tedy logaritmickou časovou složitost. Předpokládá se, že množství přechodů každého stavu v drtivé většině případů nepřesáhne 50-100, rozdíl mezi konstantní složitostí hashovací tabulky, která ovšem musí vypočítat hash pro každý prvek a logaritmickou složitostí multimapy tedy bude zanedbatelný. Algoritmy použité v této práci pro měření výsledků měly průměrně méně než 10 přechodů na jeden stav. Pro reprezentaci množiny simulačního předuspořádání byla použita třída unordered_set, což je hashovaná množina klíčů. Tato množina bývá obrovská a musí být perfektně optimalizovaná pro vyhledávání, které má konstantní průměrnou časovou složitost, stejně jako vkládání. Dále jsou použity třídy queue a stack, které mají konstantní složitosti vkládání a odebírání prvků.

V jazyce Python 3 je využíván převážně list. Je snaha používat jen operace se složitostí *O(1),* tedy vložení na konec, odebrání z konce nebo vyhledání prvku pomocí indexu. Odebírání prvků z množiny stavů tedy probíhá tak, že je nejdříve odstraňovaný prvek vyměněn s posledním a poté je smazán z konce listu. Bohužel bylo nutné občas použít i neefektivní listové operace jako průnik ($O(n^2)$) a rozdíl množin ($O(n)$). List je efektivně použit i pro implementaci zásobníku. Fronta je implementována pomocí třídy queue, která nabízí konstantní *O(1)* složitosti pro všechny operace. Množiny přechodů jsou uloženy ve slovníku s formátem písmeno abecedy (klíč) – list stavů dostupných daným písmenem (hodnota). Vkládání a vyhledání množiny stavů jsou tedy průměrně konstantní, ovšem vyhledání konkrétního stavu v dané množině už je nejhůře lineární, tedy *O(n)*. To by opět nemělo příliš vadit, jelikož se jedná o přechody pouze jednoho stavu a ne celého automatu. Pro uchování množiny relace simulace je použit slovník s klíčem typu set dvou řetězců a hodnotami nastavenými na None.

V C# jsou použity datové struktury s velmi podobnými vlastnostmi jako v Pythonu. Jednou z nich je list – *O(1)* pro všechny použité operace. Dále třídy Queue a Stack, opět s průměrnou složitostí *O(1)* pro všechny operace. Přechody pro jednotlivé stavy jsou uloženy ve třídě Dictionary, množina relace simulace je uložena ve třídě HashSet, které mají průměrnou složitost *O(1)* pro všechny použité operace. Pro minimalizaci byl využit LinkedList z důvodu potřeby konstantního vkládání a odstraňování prvků ze všech míst množiny a nutnosti udržet množinu v neměnném pořadí.

V jazyce OCaml byl pro neseřazené množiny použit List se stejnými vlastnostmi jako v ostatních jazycích. Fronta a zásobník byly implementovány pomocí rekurze v rámci implementací jednotlivých algoritmů. Pro uložení abecedy byl použit Set řetězců, prvky se pouze iteruje. Přechody pro každý stav a množina relace simulace byly uloženy v modulu Hashtbl, který má stejné vlastnosti, jako hashovací tabulky v jiných jazycích. Jedinou změnou oproti ostatním jazykům bylo to, že z důvodu snahy o funkcionální řešení algoritmů nebyly použity pomocné proměnné card a flag v jednotlivých stavech, ale místo toho byly obě proměnné uloženy do jedné hashovací tabulky.

Dále byly v jednotlivých algoritmech použity tyto specifické struktury:

Inters_help – použitá v algoritmu pro výpočet produktu, ukládá se do hlavního work-setu.

- **first** – první stav
- **second** – druhý stav
- **source** – stav vytvořený spojením first a second

Determin_help – použitá v algoritmu pro výpočet determinizace, ukládá se do hlavního work-setu.

- **states** – neseřazená množina stavů
- **source** – stav vytvořený spojením stavů ve states

Macro_state – použitá v algoritmu test univerzality, ukládá se do hlavního work-setu.

- **states** – množina odkazů na stavy makro stavu
- **rejecting** – proměnná určující, zda makro stav přijímá nebo nepřijímá

Product_state – použitá v algoritmu test inkluze, ukládá se do hlavního work-setu.

- **a1_st** – odkaz na stav product-state
- **macro_st** – makro stav v product-state
- **rejecting** – proměnná určující, zda product-state přijímá nebo nepřijímá

Tyto struktury jsou definovány pouze v C++, v ostatních jazycích jsou implementovány pomocí nepojmenovaných listů (Python, OCaml) nebo třídy tuple (C#).

C++

Struktury

Inters_help – použitá v algoritmu pro výpočet produktu

- State * first – odkaz na první stav
- State * second – odkaz na druhý stav
- State * source – odkaz na stav vytvořený spojením first a second

Determin_help – použitá v algoritmu pro výpočet determinizace

- vector<State *> states – množina odkazů na stavy
- State * source – odkaz na stav vytvořený spojením stavů ve states

Macro_state – použitá v algoritmu test univerzality

- vector<State *> states – množina odkazů na stavy makro stavu
- bool rejecting – proměnná určující, zda makro stav přijímá nebo nepřijímá

Product_state – použitá v algoritmu test inkluze

- State * a1_st – odkaz na stav product-state
- Macro_state macro_st – makro stav v product-state
- bool rejecting – proměnná určující, zda product-state přijímá nebo nepřijímá

Funkce

void **parse_FA_stdin**(std::vector<FA> &Automatons);

- Funkce načítá posloupnost automatů ze stdin do datových struktur FA a State. Načtené automaty uloží do vektoru Automatons. Je spouštěna na začátku programu.
- vstup:
  - Automatons – reference na vector<FA>.
- výstup: Funkce nic nevrací.

void **print_FA**(std::vector<FA> &Automatons);

- Funkce vypíše všechny automaty v Automatons na stdout. Slouží pro účely debugování – vypisuje více informací.
- vstup:
  - Automatons – reference na vector<FA>.
- výstup: Funkce nic nevrací.

void **Print_result_FA**(FA &automaton, bool alg_flag = false);

- Funkce vypíše automat na stdout. Jedná se o hlavní výpisovou funkci, používanou pro výpis výsledných automatů na stdout.
- vstup:
  - automaton – reference na FA.
  - alg_flag – bool – určuje mód výpisu. Všechny funkce použité pro měření jej mají nastavený do false, jen jedna experimentální jej nastavuje do true – nevypisuje stavy s flagem nastaveným do -1.
- výstup: Funkce nic nevrací.

void **Print_state_queue**(std::queue<State *> q);

- Funkce vypíše stavy ve stavové frontě na stdout (test prázdnosti, zbytečné stavy). Slouží pro účely debugování.
- vstup:
  - q – queue<State *>.
- výstup: Funkce nic nevrací.

void **Print_state_stack**(std::stack<State *> q);

- Funkce vypíše stavy ve stavovém zásobníku na stdout (test prázdnosti, zbytečné stavy). Slouží pro účely debugování.
- vstup:
  - q – stack<State *>.
- výstup: Funkce nic nevrací.

void **Partition_print**(std::list<std::vector<State *>> &Partition_lan);

- Funkce vypíše jazykový rozklad na stdout (minimalizace). Slouží pro účely debugování.
- vstup:
  - Partition_lan – list<vector<State *>>.
- výstup: Funkce nic nevrací.

void **Minim_queue_print**(std::list<std::pair<std::string, std::vector<State *> *>> &W);

- Funkce vypíše stavy ve frontě v algoritmu minimalizace na stdout. Slouží pro účely debugování.
- vstup:
    - W – reference na list<pair<string, vector<State *> *>>.
- výstup: Funkce nic nevrací.

template <class T>
void **Print_iter**(T &data_struct);

- Funkce vypíše list nebo vektor stavů na stdout. Slouží pro účely debugování.
- vstup:
    - data_struct – reference na T.
- výstup: Funkce nic nevrací.

template <class Tmpl>
void **Print_reduct**(Tmpl &data_struct);

- Funkce vypíše dvojice stavů v relaci simulace na stdout. Slouží pro účely debugování.
- vstup:
    - data_struct – reference na Tmpl.
- výstup: Funkce nic nevrací.

void **Print_MacroState**(Macro_state &m);

- Funkce vypíše makro stav na stdout. Slouží pro účely debugování.
- vstup:
    - m – reference na makro stav.
- výstup: Funkce nic nevrací.

bool **Emptiness_test**(FA &automaton);

- Funkce zjišťuje, jestli je jazyk automatu prázdný (automat neobsahuje konečný stav).
- vstup:
    - automaton – reference na KA.
- výstup: True – jazyk KA je prázdný, false – jazyk KA není prázdný.

inline std::vector<State *>::iterator **find_in_start_states**(std::vector<State *> &start_states, std::string name);

- Funkce vrací iterátor do vektoru stavů na stav se specifickým jménem.
- vstup:
    - start_states – reference na vektor stavů.
    - name – jméno stavu
- výstup: Iterátor do vektoru stavů.

inline void **Remove_relations**(std::list<State>::iterator state);

- Funkce odstraňuje přechody z a do daného stavu state.
- vstup:
    - state – iterátor do listu stavů.
- výstup: Funkce nevrací nic.

void **Restore_FA**(FA &automaton, short int alg, std::unordered_set<std::string> &visited_htab, std::unordered_set<std::string> &flag_htab);

- Funkce přijme KA se stavy, které mají být odstraněné, a odstraní je.

- vstup:
    - automaton – reference na KA.
    - alg – id algoritmu volajícího tuto funkci (0 – zbytečné stavy, 1 - redukce).
    - visited_htab – hashovací tabulka navštívených stavů (od počátečních stavů)
    - flag_htab – hashovací tabulka navštívených stavů (od konečných stavů)
- výstup: Funkce nevrací nic.

void **Remove_useless_states**(FA &automaton);

- Funkce implementuje algoritmus odstranění zbytečných stavů. Odstraňuje všechny neukončující a nedostupné stavy. Má tři části – prohledávání do šířky z počátečních stavů (ukládá do visited_htab), prohledávání do šířky z konečných stavů (ukládá do flag_htab), odstranění stavů, které nejsou v obou tabulkách.
- vstup:
    - automaton – reference na KA.
- výstup: Funkce nevrací nic.

void **Intersection_FA**(FA &automaton1, FA &automaton2, FA &result_automaton);

- Funkce implementuje algoritmus průniku. Vypočítá průnik dvou automatů vytvářením dvojic stavů.
- vstup:
    - automaton1 – reference na KA1, první operand výpočtu průniku.
    - automaton2 – reference na KA2, druhý operand výpočtu průniku.
    - result_automaton – reference na výsledný KA, slouží k uložení výsledku.
- výstup: Funkce nevrací nic.

void **Determinization_FA**(FA &automaton1, FA &result_automaton);

- Funkce implementuje algoritmus determinizace. Vytvoří deterministickou verzi zdrojového automatu.
- vstup:
    - automaton1 – reference na zdrojový KA, pro výpočet deterministického KA.
    - result_automaton – reference na výsledný KA, slouží k uložení výsledku.
- výstup: Funkce nevrací nic.

void **Hopcroft**(FA &automaton1, std::list<std::vector<State *>> &Partition_lan);

- Funkce implementuje Hopcroftův algoritmus. Vytvoří jazykový rozklad zdrojového automatu, což je zřetězený seznam bloků stavů.
- vstup:
    - automaton1 – reference na zdrojový KA, pro výpočet jazykového rozkladu.
    - Partition_lan – reference na jazykový rozklad.
- výstup: Funkce nevrací nic.

void **Minimalization_FA**(FA &automaton1, FA &result_automaton);

- Funkce implementuje algoritmus minimalizace. Vytvoří minimální verzi zdrojového automatu.
- vstup:
    - automaton1 – reference na zdrojový KA, pro výpočet minimální verze.
    - result_automaton – reference na výsledný KA.
- výstup: Funkce nevrací nic.

void **Preorder**(FA &automaton1, std::set<std::pair<std::string,std::string>> &preorder);

- Funkce implementuje algoritmus pro výpočet relace simulace. Vytvoří relaci simulace zdrojového automatu.
- vstup:
  - automaton1 – reference na zdrojový KA, pro výpočet relace simulace.
  - preorder – reference na výslednou relaci simulace.
- výstup: Funkce nevrací nic.

Funkce find_in_states a Reduction_NFA byly pouze pokusné a nad rámec zadání. Zároveň nefungují vždy správně, je tedy dobré je nepoužívat.

void **Minimize**(Macro_state &macro_R, std::set<std::pair<std::string,std::string>> &preorder);

- Funkce iteruje přes stavy makro stavu (iterátory i, j) a pokud se dvojice (i, j) nachází v preorder, je z makro stavu smazán stav i. Implementuje druhou optimalizaci algoritmu test univerzality.
- vstup:
  - macro_R – reference na makro stav.
  - preorder – reference na relaci simulace.
- výstup: Funkce nevrací nic.

bool **Is_subset**(Macro_state &macroSubs, Macro_state &macroSuper, std::set<std::pair<std::string,std::string>> &preorder);

- Funkce ověřuje, že první makro stav (macroSubs) je podmnožinou druhého makro stavu (macroSuper).
- vstup:
  - macroSubs – reference na první makro stav.
  - macroSuper – reference na druhý makro stav.
  - preorder – reference na relaci simulace.
- výstup: Funkce nevrací nic.

bool **Universality_NFA**(FA &automaton1, std::set<std::pair<std::string,std::string>> &preorder);

- Funkce implementuje algoritmus pro test univerzality.
- vstup:
  - automaton1 – reference na zdrojový KA.
  - preorder – reference na relaci simulace.
- výstup: True pokud je KA univerzální, jinak false.

bool **Inclusion_NFA**(FA &automaton1, FA &automaton2, std::set<std::pair<std::string,std::string>> &preorder);

- Funkce implementuje algoritmus pro test inkluze. Ověřuje, zda L(automaton1) je podmnožinou L(automaton2).
- vstup:
  - automaton1 – reference na první KA.
  - automaton2 – reference na druhý KA.
  - preorder – reference na relaci simulace.

- výstup: True pokud je L(automaton1) podmnožinou L(automaton2), jinak false.

void **Get_identity_relation**(FA &automaton, std::set<std::pair<std::string,std::string>> &preorder);

- Funkce generuje relaci identity z množiny stavů automatu. Může být použita pro výpočet speciálních verzí algoritmů testu univerzality a inkluze.
- vstup:
  - automaton – reference na zdrojový KA.
  - preorder – reference na relaci simulace.
- výstup: Funkce nevrací nic.

void **Union_FA**(FA &automaton1, FA &automaton2, FA &result_automaton);

- Funkce implementuje algoritmus pro výpočet sjednocení automatů. Vytváří sjednocení KA1 a KA2.
- vstup:
  - automaton1 – reference na první KA.
  - automaton2 – reference na druhý KA.
  - result_automaton – reference na výsledný KA.
- výstup: Funkce nevrací nic.

FA **Copy_FA**(FA &automaton1);

- Funkce vytváří kopii zdrojového KA.
- vstup:
  - automaton1 – reference na zdrojový KA.
- výstup: Kopie zdrojového KA.

void **Complement_FA**(FA &automaton1);

- Funkce vytváří komplement zdrojového KA. Nevytváří však nový automat, ale upravuje zdrojový.
- vstup:
  - automaton1 – reference na zdrojový KA.
- výstup: Funkce nevrací nic.

Ve funkci main jsou zpracovány argumenty programu a následně jsou volány korespondující funkce.

Python, OCaml a C#

Python a C# jsou si strukturou kódu velmi podobné, téměř identické, a tudíž jsou zde uvedeny jejich funkce dohromady. Jsou uvedeny hlavičky funkce v obou jazycích. OCaml, jakožto funkcionální jazyk, je stavěný trochu jinak. Jsou v něm implementovány stejné funkce, mají stejné jméno a parametry, ale většinou se skládají z mnoha dalších podfunkcí, implementujících například cykly. Všechny níže uvedené funkce jsou v OCamlu implementovány pod stejným jménem, mají stejný efekt a níže uvedená dokumentace takových funkcí platí i pro OCaml. Podfunkce nejsou popsány, jelikož je jich obrovské množství.

Funkce

def **ParseFA**():
public void **Parse_FA**()

- Funkce načítá posloupnost automatů ze stdin do datových struktur FA a State. Je spouštěna na začátku programu.
- vstup: Žádný vstup.
- výstup: List tříd automatů.

Výpisové funkce jsou řešeny pomocí metod třídy FA.

def **Emptiness_test**(automaton):
public bool **Emptiness_test**(FA automaton1)

- Funkce zjišťuje, jestli je jazyk automatu prázdný (automat neobsahuje konečný stav).
- vstup:
  - automaton – třída KA.
- výstup: True – jazyk KA je prázdný, false – jazyk KA není prázdný.

def **Remove_relations**(state):
public void **Remove_relations**(State state)

- Funkce odstraňuje přechody z a do daného stavu state.
- vstup:
  - state – stav, jehož přechody by měly být odstraněny.
- výstup: Funkce nevrací nic.

def **Restore_FA**(automaton, alg, visited_htab, flag_htab):
public void **Restore_FA**(FA automaton, int alg, HashSet<string> visited_htab, HashSet<string> flag_htab)

- Funkce přijme KA se stavy, které mají být odstraněné, a odstraní je.
- vstup:
  - automaton – třída KA.
  - alg – id algoritmu volajícího tuto funkci (0 – zbytečné stavy, 1 - redukce).
  - visited_htab – hashovací tabulka navštívených stavů (od počátečních stavů).
  - flag_htab – hashovací tabulka navštívených stavů (od konečných stavů).
- výstup: Funkce nevrací nic.

def **Remove_useless_states**(automaton):
public void **Remove_useless_states**(FA automaton1)

- Funkce implementuje algoritmus odstranění zbytečných stavů. Odstraňuje všechny neukončující a nedostupné stavy. Má tři části – prohledávání do šířky z počátečních stavů (ukládá do visited_htab), prohledávání do šířky z konečných stavů (ukládá do flag_htab), odstranění stavů, které nejsou v obou tabulkách.
- vstup:
  - automaton – třída KA.
- výstup: Funkce nevrací nic.

def **Intersection_FA**(automaton1, automaton2):
public FA **Intersection_FA**(FA automaton1, FA automaton2)

- Funkce implementuje algoritmus průniku. Vypočítá průnik dvou automatů vytvářením dvojic stavů.
- vstup:
  - automaton1 – KA1, první operand výpočtu průniku.
  - automaton2 – KA2, druhý operand výpočtu průniku.
- výstup: Výsledný KA.

def **Determinization_FA**(automaton1):
public FA **Determinization_FA**(FA automaton1)

- Funkce implementuje algoritmus determinizace. Vytvoří deterministickou verzi zdrojového automatu.
- vstup:
  - automaton1 – reference na zdrojový KA, pro výpočet deterministického KA.
- výstup: Výsledný KA.

def **Hopcroft**(automaton1):
public LinkedList<IList<State>> **Hopcroft**(FA automaton1)

- Funkce implementuje Hopcroftův algoritmus. Vytvoří jazykový rozklad zdrojového automatu, což je zřetězený seznam bloků stavů.
- vstup:
  - automaton1 – reference na zdrojový KA, pro výpočet jazykového rozkladu.
- výstup: Jazykový rozklad – list bloků stavů.

def **Minimalization_FA**(automaton1):
public FA **Minimalization_FA**(FA automaton1)

- Funkce implementuje algoritmus minimalizace. Vytvoří minimální verzi zdrojového automatu.
- vstup:
  - automaton1 – zdrojový KA, pro výpočet minimální verze.
- výstup: Výsledný KA.

def **Preorder**(automaton1):
public HashSet<Tuple<string, string>> **Preorder**(FA automaton)

- Funkce implementuje algoritmus pro výpočet relace simulace. Vytvoří relaci simulace zdrojového automatu.
- vstup:
  - automaton1 – zdrojový KA, pro výpočet relace simulace.
- výstup: Výsledná relace simulace (slovník {(jméno_stavu, jméno_stavu):None}).

def **Minimize**(macro_R, preorder):
public Macro_state **Minimize**(Macro_state macro_R, HashSet<Tuple<string, string>> preorder)

- Funkce iteruje přes stavy makro stavu (iterátory i, j) a pokud se dvojice (i, j) nachází v preorder, je z makro stavu smazán stav i. Implementuje druhou optimalizaci algoritmu test univerzality.
- vstup:
  - macro_R – makro stav.
  - preorder – relace simulace.
- výstup: Minimalizovaný makro stav.

def **Is_subset**(macroSubs, macroSuper, preorder):
public bool **Is_subset**(Macro_state macroSubs, Macro_state macroSuper, HashSet<Tuple<string, string>> preorder)

- Funkce ověřuje, že první makro stav (macroSubs) je podmnožinou druhého makro stavu (macroSuper).
- vstup:
  - macroSubs – první makro stav.
  - macroSuper – druhý makro stav.
  - preorder – relace simulace.
- výstup: True pokud macroSubs je podmnožinou macroSuper., jinak false

def **Universality_NFA**(automaton1, preorder):
public bool **Universality_NFA**(FA automaton1, HashSet<Tuple<string, string>> preorder)

- Funkce implementuje algoritmus pro test univerzality.
- vstup:
  - automaton1 – zdrojový KA.
  - preorder – relaci simulace.
- výstup: True pokud je KA univerzální, jinak false.

def **Inclusion_NFA**(automaton1, automaton2, preorder):
public bool **Inclusion_NFA**(FA automaton1, FA automaton2, HashSet<Tuple<string, string>> preorder)

- Funkce implementuje algoritmus pro test inkluze. Ověřuje, zda L(automaton1) je podmnožinou L(automaton2).
- vstup:
  - automaton1 – první KA.
  - automaton2 – druhý KA.
  - preorder – relace simulace.
- výstup: True pokud je L(automaton1) podmnožinou L(automaton2), jinak false.

def **Create_identity_relation**(automaton1):
public HashSet<Tuple<string, string>> **Get_identity_relation**(FA automaton)

- Funkce generuje relaci identity z množiny stavů automatu. Může být použita pro výpočet speciálních verzí algoritmů testu univerzality a inkluze.
- vstup:
  - automaton1 – reference na zdrojový KA.
- výstup: Výsledná relace identity.

def **Union_FA**(automaton1, automaton2):
public FA **Union_FA**(FA autom1, FA autom2)

- Funkce implementuje algoritmus pro výpočet sjednocení automatů. Vytváří sjednocení KA1 a KA2.
- vstup:
  - automaton1 – první KA.
  - automaton2 – druhý KA.
- výstup: Výsledný automat.

def **Complement_FA**(automaton1):
public FA **Complement_FA**(FA automaton1)

- Funkce vytváří komplement zdrojového KA.
- vstup:
    - automaton1 – zdrojový KA.
- výstup: Výsledný automat.

Ve funkci main jsou zpracovány argumenty programu a následně jsou volány korespondující funkce.