



Semestrální práce z KIV/PC

# Generování Konečného Automatu

Pavel Čácha  
A19B0022P  
cache@students.zcu.cz

6. 12. 2020

# Obsah

<b>1</b>	<b>Úvod</b>	<b>3</b>
<b>2</b>	<b>Zadání</b>	<b>1</b>
2.1	Specifikace vstupu programu . . . . .	1
2.2	Specifikace výstupu programu . . . . .	1
2.3	Plné znění zadání . . . . .	2
<b>3</b>	<b>Analýza úlohy</b>	<b>3</b>
3.1	Reprezentace konečného automatu . . . . .	3
3.1.1	Seznamy sousednosti . . . . .	3
3.1.2	Matice sousednosti . . . . .	5
3.1.3	Zvolený způsob . . . . .	6
3.2	Načtení hrany grafu . . . . .	6
3.2.1	Uložení názvu a lineární prohledání . . . . .	6
3.2.2	Hashovací tabulka . . . . .	6
3.2.3	Zvolený způsob . . . . .	7
3.3	Generovaný kód zpracovávající regulární výraz . . . . .	7
3.3.1	Vytvoření polí . . . . .	7
3.3.2	Zanořované podmínky . . . . .	8
3.3.3	Zvolený způsob . . . . .	8
<b>4</b>	<b>Programátorská dokumentace</b>	<b>9</b>
4.1	Načtení grafu . . . . .	9
4.1.1	Načtení vrcholů . . . . .	9
4.1.2	Načtení hran . . . . .	9
4.2	Implementace hashovací tabulky . . . . .	10
4.2.1	Struktury hashovací tabulky . . . . .	11
4.2.2	Generování hashovacího kódu . . . . .	11
4.2.3	Přidávání a vyhledávání prvků . . . . .	12
4.3	Generování kódu . . . . .	12
4.4	Popis jednotlivých modulů . . . . .	12
4.4.1	<code>main.c</code> . . . . .	12
4.4.2	<code>loader.c</code> . . . . .	13
4.4.3	<code>hash_table.c</code> . . . . .	13
4.4.4	<code>memory_observer.c</code> . . . . .	13
4.4.5	<code>err_manager.c</code> . . . . .	13

<b>5</b>	<b>Uživatelská příručka</b>	<b>14</b>
5.1	Překlad zdrojových kódů . . . . .	14
5.2	Ovládání programu . . . . .	14
5.3	Popis vstupního souboru . . . . .	15
<b>6</b>	<b>Závěr</b>	<b>16</b>

# 1 Úvod

Cílem mé semestrální práce z předmětu Programování v jazyce C bylo vytvořit multiplatformní konzolovou aplikaci v jazyce C, která bude plně respektovat normu ANSI X3.159-1989. Jako téma své práce jsem zvolil zadání číslo 2 – generování konečného automatu.

Úkol programu spočívá v načtení definice konečného automatu a následném vygenerování přeložitelného souboru se zdrojovým kódem v jazyce C. Po přeložení vznikne samostatně spustitelná aplikace představující programovou implementaci daného automatu. Při spuštění je zadán jako argument regulární výraz, o kterém má být rozhodnuto, zda ho automat akceptuje, nebo ne.

Obsahem dokumentace je nejprve zkrácený popis zadání. Poté následuje analýza hlavních problémů, které se při implementaci vyskytly. Samotný popis programu se nachází v kapitole nazvané programátorská dokumentace. Po ní následuje návod k používání programu a na závěr je krátké zhodnocení celé práce.

## 2 Zadání

Naprogramujte v ANSI C přenositelnou **konzolovou aplikaci**, která načte ze souboru definici konečného automatu, který přijme textový řetězec popsaný regulárním výrazem. A vygeneruje validní zdrojový kód v ANSI C, který bude představovat implementaci daného konečného automatu.

Program se bude spouštět příkazem: `fsmgen.exe <file>`. Symbol `<file>` zastupuje jméno souboru s popisem konečného stavového automatu. Váš program tedy může být během testování spuštěn takto:

```
...\>fsmgen.exe graph.gv
```

Výstupem programu bude validní zdrojový kód v ANSI C, představující implementaci daného automatu. Pokud nebude uveden právě jeden argument, vypíše chybové hlášení a stručný návod k použití programu v angličtině podle běžných zvyklostí. **Vstupem programu jsou pouze argumenty na příkazové řádce – interakce s uživatelem pomocí klávesnice či myši v průběhu práce programem se neočekává.**

Práci odevzdejte v archivu ZIP. Archiv necht obsahovat zdrojové kódy, **makefile** pro Windows i Linux a dokumentaci ve formátu PDF.

### 2.1 Specifikace vstupu programu

Vstupem programu jsou pouze parametry na příkazové řádce. Předaný parametr představuje jméno vstupního souboru. Jedná se o textový soubor s definicí konečného automatu zapsanou pomocí jazyka DOT pro nástroj Graphviz.

### 2.2 Specifikace výstupu programu

Výstupem programu bude soubor `fsm.c`, jehož přeložením vznikne spustitelný soubor `fsm.exe`. Výsledný program bude poté spuštěn příkazem `fsm.exe <string>`. Symbol `<string>` představuje alfanumerický řetězec, který musí konečný automat zpracovat. U vstupního řetězce záleží na velikosti písmen.

V případě, že byl řetězec zpracován úspěšně, bude návratová hodnota programu 0, pokud se řetězec nepodařilo zpracovat, bude návratová hodnota programu 1. Při testování budou ověřovány obě možnosti.

## **2.3 Plné znění zadání**

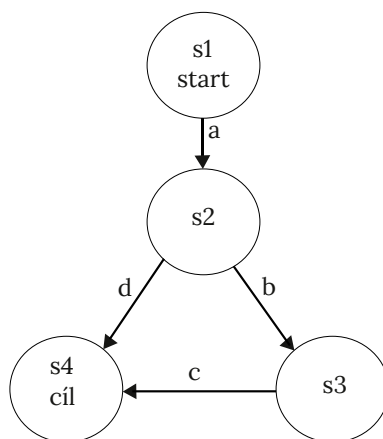
Původní zadání semestrální práce je k nalezení na adrese:

<https://www.kiv.zcu.cz/studies/predmety/pc/doc/work/sw2020-02.pdf>.

## 3 Analýza úlohy

### 3.1 Reprezentace konečného automatu

Konečný automat je ve vstupním textovém souboru popsán jako orientovaný graf, jehož hrany jsou ohodnoceny alfanumerickými znaky. Prvním krokem programu bude načtení takového grafu do paměti. Je tedy zapotřebí určit, jak budou získané vrcholy a hrany reprezentovány.

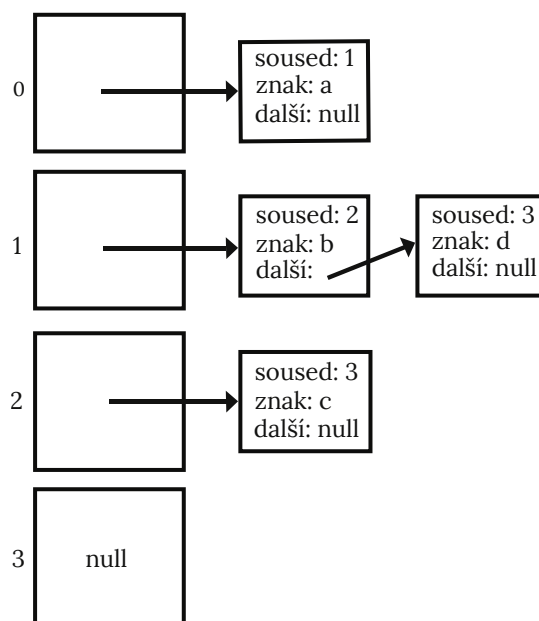


Obrázek 3.1: Příklad vstupního konečného automatu

Abstraktní datový typ graf se zabývá pouze formátem uložení hran. Případné informace o vrcholech jsou drženy v samostatném poli. Můžeme tedy vybírat ze dvou nejobvyklejších implementací tohoto abstraktního datového typu.

#### 3.1.1 Seznamy sousednosti

První možná programová reprezentace grafu je pomocí seznamů sousednosti, která najde využití převážně u řídkých grafů.



Obrázek 3.2: Seznamy sousednosti

Princip implementace pomocí seznamů sousednosti spočívá ve vytvoření pole spojových seznamů o velikosti počtu vrcholů. Index v tomto poli značí počáteční vrchol hrany. V poli jsou uloženy ukazatele na příslušné první položky seznamů. Seznamy jsou tvořeny strukturami obsahujícími informace o hraně a referenci na strukturu stejného typu, tedy další položku seznamu. Pokud z vrcholu nevede žádná hrana, na jeho indexu se v poli nachází nulový ukazatel. Pokud má být přidána nová hrana, zařadí se na začátek seznamu a jako její následník se označí položka, která byla na tomto místě původně. Přidání hrany tedy proběhne s časovou složitostí  $O(1)$ .

Pro zjištění existence orientované hrany mezi dvěma vrcholy je zapotřebí projít celý spojový seznam nacházející se v poli na indexu počátečního vrcholu. V případě hustých grafů může být velikost spojového seznamu rovna počtu všech vrcholů, operace tedy proběhne v  $O(V)$ . U řídkých je v průměru konstantní, a proto můžeme prohlásit, že operace bude mít v průměrném čase složitost  $O(1)$ .

Suma počtu položek všech spojových seznamů je dána počtem hran grafu. Paměťová náročnost tedy lineárně roste s počtem hran grafu.



### 3.1.2 Matice sousednosti

Druhý způsob se hodí spíše pro husté grafy a implementuje se pomocí matice sousednosti.

		koncový vrchol			
		0	1	2	3
počáteční vrchol	0	null	a	null	null
	1	null	null	b	d
	2	null	null	null	c
	3	null	null	null	null

Obrázek 3.3: Matice sousednosti

Na obrázku 3.3 můžeme vidět matici sousednosti pro graf znázorněný na obrázku 3.1. Matice má počet řádků i sloupců roven počtu vrcholů grafu. Indexy řádků označují vstupní vrcholy a indexy sloupců výstupní. Pokud tedy chceme zaznamenat výskyt hrany mezi dvěma vrcholy, uložíme na příslušné místo v matici ukazatel na znak ohodnocující hranu. Pokud hrana neexistuje, nachází se zde nulový ukazatel. V matici může být obecně uložena reference na jakoukoliv strukturu obsahující libovolné informace o hraně.

Hranu můžeme do matice přidat v konstantním čase přidáním ukazatele do příslušného místa ve dvourozměrném poli. Test existence hrany mezi dvěma vrcholy probíhá též v  $O(1)$ .

Paměťová náročnost je podstatně horší než u implementace seznamů sousednosti. Spotřebovaná paměť kvadraticky roste s počtem vrcholů a je tedy přesně  $O(V^2)$ . Struktura vymezuje paměť i pro neexistující hrany, a proto se hodí především u grafů s velkým množstvím hran

### 3.1.3 Zvolený způsob

V rámci své semestrální práce jsem implementoval graf pomocí seznamů sousednosti. Vedlo mě k tomu především to, že očekávám na vstupu programu spíše řídké grafy, ale i v případě hustých grafů je složitost operací poměrně přívětivá.

## 3.2 Načtení hrany grafu

Hrany jsou v textovém souboru definovány v formátu `<počáteční vrchol>-><koncový vrchol>[label=<znak>]`. Potřebujeme tedy podle jména vrcholu určit jeho index. Pro jeho zjištění máme následující možnosti.

### 3.2.1 Uložení názvu a lineární prohledání

První možností pro získání odpovídajícího indexu vrcholu na základě jeho jména je uložit si do pole vrcholů dvě hodnoty v rámci nějaké struktury. První z nich je textový řetězec s příslušným jménem a druhá je číselný kód určující, zda je vrchol výstupní nebo ne.

Požadovaný index pak lze získat v lineárním čase iterací přes všechny vrcholy grafu. Pokud se bude hledané jméno shodovat se jménem vrcholu, můžeme nalezený index použít při vkládání hrany.

### 3.2.2 Hashovací tabulka

Druhé a o poznání sofistikovanější řešení je využití hashovací tabulky. Vynikající vlastností takové tabulky je přidání a nalezení záznamu v průměrně konstantním čase. Hodnoty do ní vkládáme v párech. Jedna z hodnot bude považována za klíč a druhá bude hodnota uchovávaná pod tímto klíčem. V našem případě potřebujeme hledat podle názvu vrcholu, a proto bude klíč typu textový řetězec. Hodnota uchovávaná pod tímto klíčem bude typu integer a bude označovat index vrcholu s daným jménem v poli vrcholů.

Před začátkem načítání vrcholů si tedy vytvoříme a zinicilizujeme hashovací tabulku a v průběhu načítání do ní budeme ukládat jména a indexy vrcholů. Při načítání hran se tabulky budeme dotazovat na indexy uložené pod aktuálními jmény koncového a počátečního vrcholu hrany.

### 3.2.3 Zvolený způsob

V rámci svého programu jsem zvolil použití hashovací tabulky, protože je zcela zřejmě časově výhodnější, než lineární prohledávání. Na druhou stranu její implementace byla značně náročnější než jednoduchá iterace přes pole vrcholů.

## 3.3 Generovaný kód zpracovávající regulární výraz

Regulární výraz se zpracuje postupným procházením grafu v závislosti na aktuálně načteném znaku a aktuálně dostupných výstupních hran vrcholu. Načtený graf tedy potřebujeme nějakým způsobem předat do nově vygenerovaného zdrojového kódu. Můžeme použít jednu z následujících možností.

### 3.3.1 Vytvoření polí

Graf do vygenerovaného souboru předáme v podobné formě, jako ho máme načtený. Vygenerujeme tedy čtyři pole, kterými bude reprezentován.

První pole typu integer bude reprezentovat všechny vrcholy. Jednička bude označovat výstupní vrcholy automatu a nula všechny ostatní. Zbylá pole definují hrany a to tak, že jedno z nich bude obsahovat index počátečního vrcholu, další index koncového vrcholu a poslední bude pole znaků reprezentující ohodnocení hrany. Tyto tři pole samozřejmě musí mít synchronizované indexy, aby se tak daly propojit jednotlivé části hrany. Mimo to si budeme držet index vstupního vrcholu automatu. Pro automat z obrázku 3.1 by kód vypadal takto:

---

```
int vrcholy[4] = {0, 0, 0, 1};
int vaktualni = 0;
int hstart[4] = {0, 1, 1, 2};
int hcil[4] = {1, 2, 3, 3};
char hznak[4] = {'a', 'b', 'd', 'c'};
```

---

Pro zpracování regulárního výrazu poté budeme iterovat přes všechny jeho znaky. Pro každý znak budeme prohledávat pole hran a přitom se snažit nalézt hranu, která vychází z aktuálního vrcholu a její příslušný znak se rovná načtenému. Pokud ji najdeme, změníme aktuální vrchol na koncový vrchol hrany. Pokud nebude taková hrana nalezena, regulární výraz nebude

přijet. Po dokončení iterace přes všechny znaky výrazu můžeme pomocí pole s vrcholy určit, zda jsme skončili ve výstupním vrcholu, a rozhodnout o přijetí či nepřijetí regulárního výrazu.

### 3.3.2 Zanořované podmínky

Ve druhém způsobu budeme graf reprezentující automat procházet pomocí systému zanořovaných podmínek. Kód vytvoříme tak, že pro každý vrchol vygenerujeme podmínky pro každou jeho hranu. Cyklus iterující přes všechny znaky regulárního výrazu bude mít o jeden speciální průběh navíc. Ten bude zjišťovat akceptaci řetězce. Postup ilustruje následující příklad pro vrchol s indexem 1 z obrázku 3.1 pro obě jeho hrany:

---

```
if(vaktualni == 1) {
    if(i == stringlen) {
        return EXIT_FAILURE;
    }

    if(regex[i] == 'b') {
        vaktualni = 2;
        continue;
    }
    if(regex[i] == 'd') {
        vaktualni = 3;
        continue;
    }

    return EXIT_FAILURE;
}
```

---

### 3.3.3 Zvolený způsob

V semestrální práci jsem naimplementoval způsob 3.3.2. To zejména proto, že se mi zdá přehlednější. U delších polí není zapotřebí složitě dopočítávat všechny indexy při kontrole výsledného zdrojového kódu.

## 4 Programátorská dokumentace

### 4.1 Načtení grafu

Po ověření validity uživatelem zadaného souboru s definicí grafu je zavolána funkce `load_graph(...)` z modulu `loader.c`. Ta se zaměřuje na načtení grafu a jsou jí předány ukazatele na příslušné proměnné, které má naplnit.

#### 4.1.1 Načtení vrcholů

Jako první jsou v souboru definovány vrcholy. Před začátkem jejich načítání je tedy potřeba vytvořit a nastavit novou hashovací tabulku a alokovat místo pro pole vrcholů. Vrcholy jsou reprezentovány celočíselnou hodnotou, kde nula znamená, že vrchol není výstupní, a jednička, že vrchol je výstupní. Poté procházíme definice vrcholů v souboru a získané hodnoty ukládáme do dočasných proměnných.

Po rozebrání řádky s informacemi je potřeba udělat několik věcí. Pokud je vrchol vstupní, jeho index v poli nastavíme proměnné `starting_vertex`. Dále musíme označit v poli `vertices`, zda je vrchol výstupní. U tohoto pole ale musí být předem zkontrolována jeho velikost, pokud nedostačuje, zavolá se funkce `expand_array(...)`, která zvětší pole na dvojnásobnou velikost. V poslední řadě uložíme název vrcholu (klíč) a jeho index v poli (hodnota) do hashovací tabulky.

#### 4.1.2 Načtení hran

Reprezentace hran grafu je implementována pomocí seznamů sousednosti. Seznamy sousednosti jsou drženy jako pole ukazatelů na strukturu `edge`. Ta je v kódu popsána takto:

---

```
typedef struct the_edge {  
    int dest_index;  
    char edge_char;  
    struct the_edge *next;  
} edge;
```

---

Proměnná `dest_index` označuje index koncového vrcholu, `edge_char` je znak ohodnocující tuto hranu a `next` je další hrana v seznamu sousednosti. Index počátečního vrcholu je dán indexem seznamu sousednosti, ve kterém se hrana nachází.

Před načítáním hran je zapotřebí alokovat místo pro pole seznamů sousednosti. Jeho velikost se na rozdíl od pole vrcholů již měnit nebude a bude stejná jako je počet vrcholů. Poté můžeme procházet všechny hrany v souboru do té doby, než nalezneme ukončující závorku (}') výčtu hran.

Z řádky s definicí hrany získáme názvy počátečního a koncového vrcholu spolu se znakem aktivujícím přechod po hraně. Index vrcholů hrany pak získáme v konstantním čase voláním funkce `get_value(...)` z modulu `hash_table.c`. Poté alokujeme místo pro strukturu typu `edge` a po naplnění daty ji přidáme na první místo seznamu sousednosti. Toto je ilustrováno v následujícím úseku kódu:

---

```
new_edge = (edge *) malloc(sizeof(edge));
if(new_edge == NULL) {
    free_table_and_terminate();
}
increase_block_count();

new_edge->dest_index = end_vertex_index;
new_edge->edge_char = edge_char;
new_edge->next = NULL;

if((*edges)[start_vertex_index] != NULL) {
    new_edge->next = (*edges)[start_vertex_index];
}
(*edges)[start_vertex_index] = new_edge;
```

---

## 4.2 Implementace hashovací tabulky

Jednotlivé prvky hashovací tabulky jsou uloženy velmi obdobně jako hrany v rámci grafu. Tabulka obsahuje pole spojových seznamů s jednotlivými prvky. Index spojového seznamu, do kterého prvek spadá, se určí na základě vygenerovaného hashovacího kódu. Když průměrný počet položek v jednom seznamu překročí stanovenou maximální hranici, dojde k rozšíření pole se seznamy. To zaručí konstantní časy pro operace přidávání a vyhledávání.

### 4.2.1 Struktury hashovací tabulky

Struktura `hash_table` reprezentující samotnou hashovací tabulku.

---

```
typedef struct the_hash_table {  
    int items_count;  
    int array_lenght;  
    table_item **values;  
} hash_table;
```

---

- `items_count` značí celkový počet položek uložených v hashovací tabulce
- `array_lenght` značí velikost pole spojových seznamů
- `values` je pole ukazatelů na počáteční prvek spojového seznamu

Struktura `table_item` definuje jednu položku hashovací tabulky uchovávající klíč, hodnotu a odkaz na následovníka.

---

```
typedef struct the_table_item {  
    char key[MAX_VERTEX_NAME_LEN];  
    int value;  
    struct the_table_item *next;  
} table_item;
```

---

- `key` je textový řetězec označující klíč položky
- `value` je celočíselná hodnota uložená pod daným klíčem
- `next` označuje ukazatel na další položku ve spojovém seznamu

### 4.2.2 Generování hashovacího kódu

Hashovací kód je stanovován na základě klíče, tedy textového řetězce. Při vytváření bere v potaz všechny znaky a na základě pozice jim přiděluje váhu. Hodnota kódu se počítá v cyklu následovně:

---

```
for(i = 0; i < (int) strlen(key); i++)
    code += (i + 1) * key[i];
}
```

---

### 4.2.3 Přidávání a vyhledávání prvků

Přidávání položek do tabulky probíhá následovně. Nejprve se vygeneruje hashovací kód pro daný klíč. Následně se alokuje paměť pro strukturu typu `table_item` a naplní se daty. Na základě hashovacího kódu se určí index spojového seznamu, kam má být položka zařazena. Zařadí se tedy na začátek seznamu a případně si uloží odkaz na původní hlavičku spojového seznamu. Na konci funkce dojde ke kontrole průměrného počtu prvků na seznam. Pokud došlo k překročení maximálního počtu, dojde k rozšíření počtu spojových seznamů na dvojnásobek a vložení všech prvků do nové struktury pomocí funkce `expand_hash_table(...)`.

Při vyhledávání se opět zjistí index spojového seznamu podle vygenerovaného hashovacího kódu. Následně se iteruje přes všechny prvky seznamu a kontroluje se shodnost klíčů. Pokud se klíče shodují, hledaný prvek byl nalezen a můžeme vrátit příslušnou hodnotu.

## 4.3 Generování kódu

Generování kódu do výstupního souboru `fsm.c` probíhá ve funkci `make_output()`, která se nachází v modulu `main.c`. Generování se realizuje pomocí textových řetězců předpřipravených v konstantách. Nejprve se vygeneruje kód, který je nezávislý na vstupním automatu. Následně ve dvou vnořených cyklech proběhne generování podmínek pro zpracování regulárního výrazu. Vnější cyklus slouží pro přechod mezi vrcholy grafu a vnitřní pro nalezení příslušných hran.

## 4.4 Popis jednotlivých modulů

### 4.4.1 `main.c`

Modul se stará o ošetření uživatelského vstupu z příkazové řádky. A to zejména tak, že ověří přítomnost argumentu a existenci uváděného souboru. Obsahuje proměnné reprezentující načtený graf. Hned první instrukce ve výchozí funkci `main(...)` zaregistruje funkci volanou při ukončení programu. Je



to funkce `shutdown()` a stará se o vyčištění alokované paměti. V neposlední řadě obsahuje funkci pro generování zdrojového kódu `make_output_file()`.

#### **4.4.2 loader.c**

Tento modul se zaměřuje na načtení grafu z textového souboru. Obsahuje funkcionalitu pro získávání hodnot z definic hran i vrcholů. Vytváří si hashovací tabulku, kterou dále využívá pro rychlejší získávání indexů vrcholů na základě jejich názvů. Stará se o zvětšování pole vrcholů v případě, že velikost přestane stačit.

#### **4.4.3 hash\_table.c**

Zde se vyskytuje veškerá logika pro práci s hashovací tabulkou. Funkce `initialize_hash_table(...)` slouží pro nastavení výchozích hodnot tabulky. Dále se zde nachází funkce pro přidávání položek do tabulky a jejich vyhledávání. O generování hashovacího kódu se stará funkce `get_hash_code(...)`. Funkcionalita pro uvolňování paměti spotřebované tabulkou se nachází také zde.

#### **4.4.4 memory\_observer.c**

Tento modul je velmi prostý. Obsahuje proměnnou držící počet alokovaných bloků paměti. Jsou zde přítomny také funkce pro zvyšování, snižování a získávání počtu alokovaných bloků. Modul slouží pro ověření správnosti práce s pamětí.

#### **4.4.5 err\_manager.c**

Tento modul slouží pro reagování na chyby a neočekávané stavy. Obsahuje funkce pro ukončení běhu programu a případné vypsání krátké nápovědy.

## 5 Uživatelská příručka

### 5.1 Překlad zdrojových kódů

Spolu se zdrojovými kódy byl distribuován i soubor `makefile` pro Linux a `makefile.win` pro Windows. Tyto soubory slouží k přeložení a sestavení programu, jejich výstupem je spustitelný soubor.

V závislosti na platformě je potřeba si zvolit jednu z následujících možností. V rámci operačního systému Linux stačí spustit příkazovou řádku a zadat příkaz `make`. Ve Windows je potřeba otevřít vývojářskou příkazovou řádku a zadat příkaz: `nmake -f makefile.win`. V případě přítomnosti překladače GCC může být využit opět příkaz `make`. Tím se vytvořil spustitelný soubor `fsmgen.exe`.

```
gcc -c -Wall -pedantic -ansi err_manager.c -o err_manager.o
gcc -c -Wall -pedantic -ansi memory_observer.c -o memory_observer.o
gcc -c -Wall -pedantic -ansi hash_table.c -o hash_table.o
gcc -c -Wall -pedantic -ansi loader.c -o loader.o
gcc -c -Wall -pedantic -ansi main.c -o main.o
gcc err_manager.o memory_observer.o hash_table.o loader.o main.o -o fsmgen.exe
```

Obrázek 5.1: Výpis v konzoli po překladu a sestavení pomocí GCC

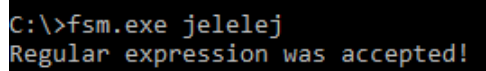
### 5.2 Ovládání programu

Aplikace se spustí příkazem `fsmgen.exe <graph.gv>`, kde `graph.gv` je textový soubor s definicí automatu. Pokud nebude parametr s názvem souboru přítomný, program vypíše nápovědu a skončí.

Pokud vše proběhlo v pořádku, program vygeneroval soubor `fsm.c` a do konzole bylo vypsáno: `fsm.c was successfully created!`.

Tím byl vygenerován zdrojový kód programu s implementací daného konečného automatu. Tento soubor je opět potřeba přeložit. Použijeme příkaz `gcc fsm.c -o fsm.exe`. Vytvořili jsme tak spustitelný soubor.

Soubor spustíme s požadovaným regulárním výrazem a automat vypíše, zda ho akceptuje.



```
C:\>fsm.exe jelelej
Regular expression was accepted!
```

Obrázek 5.2: Spuštění vygenerované aplikace

## 5.3 Popis vstupního souboru

Konečný automat musí být popsán pomocí jazyka DOT pro nástroj Graphviz. Definice grafu je prováděna v následujícím formátu.

Jako první řádek je deklarace grafu. Poté následuje otevírací složená závorka a hned za ní komentář označující začátek výčtu vrcholů. Vrcholy jsou zapisovány ve formátu `<název>[label=<popisek>];`, kde název může mít maximálně 10 znaků a volitelný popisek nabývá hodnot `start` nebo `end` v závislosti na tom, zda je vrchol vstupní nebo výstupní. Automat může mít jeden vstupní a více výstupních vrcholů.

Na uvození začátku výčtu hran se používá komentář `//edges`. Poté následuje výčet hran ve formátu:

`<počáteční vrchol>-><koncový vrchol>[label=<znak>];`.

Hrana je tedy definována názvem počátečního a koncového vrcholu a pomocí znaku aktivujícího přechod po hraně. Vstupní soubor pro automat z obrázku 3.1 vypadá takto:

---

```
digraph G
{
    //nodes
    s1[label=start];
    s2;
    s3;
    s4[label=end];
    //edges
    s1->s2[label=a];
    s2->s3[label=b];
    s2->s4[label=d];
    s3->s4[label=c];
}
```

---

## 6 Závěr

Vzhledem k tomu, že jsem se v rámci tohoto předmětu vůbec poprvé seznámil s jazykem C, bylo pro mě naprogramování semestrální práce výzvou. Ale i přesto se mi podařilo stvořit funkční aplikaci. Její zdrojové kódy jsou poměrně dobře strukturované, rozšiřitelné a komentované.

Největší problém mi dělala správa paměti. A to zejména čištění již nepotřebných bloků. Při implemetaci hashovací tabulky jsem mnohdy váhal, jestli nezvolím snazší řešení s lineární časovou náročností, ale nakonec jsem se zdárně dopracoval k dobrému výsledku.