VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

KNIHOVNA PRO ZVÝRAZNĚNÍ SYNTAXE V TEXTOVÉM EDITORU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE PETR LEKEŠ
AUTHOR

BRNO 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

KNIHOVNA PRO ZVÝRAZNĚNÍ SYNTAXE V TEXTOVÉM EDITORU

LIBRARY FOR SYNTAX STYLING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE PETR LEKEŠ
AUTHOR

VEDOUCÍ PRÁCE Ing. JIŘÍ TECHET
SUPERVISOR

BRNO 2007

Abstrakt

Práce popisuje tvorbu knihovny pro zvýraznění syntaxe v textovém editoru. Knihovna umožňuje zvýraznění řádkových a blokových komentářů, klíčových slov, operátorů, párových symbolů apod. Definice částí textu, které chceme zvýrazňovat, jsou uloženy v externím textovém souboru. Uživatel vytvoří tento soubor a pomocí regulárních výrazů nadefinuje podobu jednotlivých zvýrazňovaných částí textu.

Klíčová slova

Zvýraznění syntaxe, lexikální analýza, lexém, token, regulární výraz, konečný automat.

Abstract

This thesis describes the creation of library for syntax styling at the text editor. Library is able to highlight line and block comments, keywords, operators, pair symbols, etc. The definitions of text sections which we want to highlight are saved in the extern text file. User creates this file and defines pattern for each highlighted text section by regular expressions.

Keywords

Syntax styling, lexical analysis, lexeme, token, regular expression, finite automaton.

Citace

Petr Lekeš: Knihovna pro zvýraznění syntaxe v textovém editoru, bakalářská práce, Brno, FIT VUT v Brně, 2007

Knihovna pro zvýraznění syntaxe v textovém editoru

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Techeta

. . . . . .
Petr Lekeš
29. května 2007

Zadaní

1. Seznamte se se základy lexikální analýzy a vhodným programovacím jazykem použitým při implementaci.
2. Proveďte funkční návrh vytvářené knihovny. Knihovna musí podporovat zvýraznění syntaxe jednotlivých slov ve vstupním souboru (klíčová slova), lite rálů, párových symbolů (závorky), řádkových a blokových komentářů apod.
3. Navrhněte vhodné rozhraní vámi navržené knihovny.
4. Implementujte tuto knihovnu.
5. Pomocí knihovny implementujte jednoduchý textový editor se zvýrazněním syntaxe.
6. Zhodnoťte vytvořenou knihovnu, uveďte její klady a nedostatky. Diskutujte další možný rozvoj projektu.

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologí Vysokého učení technického v Brně.

Obsah

Kapitola 1

Úvod

Zvýrazňování syntaxe patří k užitečným nástrojům textových editorů, které pomocí různých barev nebo písem zobrazují určité části textu. Slouží obvykle ke zvýraznění zdrojových kódů, ale používá se také pro zvýraznění například pravopisných chyb. Takto zvýrazněný text (zdrojový kód) je mnohem přehlednější a lépe se v něm orientuje. Navíc může již během psaní zdrojového kódu vizuálně upozornit uživatele na chybu.

Cílem mé bakalářské práce bylo vytvořit knihovnu umožnující zvýraznění syntaxe v textovém editoru. Následně pomocí této knihovny implementovat jednoduchý textový editor se zvýrazněním syntaxe. Knihovna musí podporovat zvýraznění klíčových slov, literálů, čísel, řádkových a blokových komentářů, párových symbolů apod. Definice těchto rozpoznávaných elementů jazyka musí být měnitelná. Díky této schopnosti může být textový editor využit pro zvýraznování syntaxe libovolného programovacího či skriptovacího jazyka. Důležitou roli hraje efektivita zvolené implementace.

Problémem, jak v textu rozpoznat jednotlivé lexémy se zabývá lexikální analýza, která je podrobněji popsána v následující kapitole. Dále je potřeba vhodným způsobem nadefinovat podobu lexémů, které mají být zvýrazněny. Pro tento účel je výhodné využít regulární výrazy, které si přiblížíme v další kapitole. Lexikální analýza je obvykle založena na simulaci konečného automatu, jehož principy a varianty jsou vysvětleny opět ve 2. kapitole. Dalším úkolem knihovny je zajistit komunikaci s textovým editorem. Analýzou a návrhem knihovny se zabývá kapitola 3. Uživatelské rozhraní textového editoru se stará jen o zpracování vstupních událostí od uživatele (např. stisk klávesy) a zobrazení editovaného textu na obrazovce. Konkrétní implementace knihovny je popsána v kapitole 4.

Kapitola 2

Teorie

V této kapitole jsou shrnuty teoretické znalosti potřebné pro tvorbu knihovny. Definice použité v této kapitole byly převzány z [2], [1] a [4].

2.1 Lexikální analýza

Lexikální analýza je proces, který převádí posloupnost znaků na posloupnost tokenů. Token je obvykle tvořen nedělitelnou posloupností znaků a svým typem. Tato posloupnost znaků se označuje pojmem lexém. Právě typ tokenu určuje, o který rozpoznaný lexém se jedná. Program provádějící tuto činnost se nazývá lexikální analyzátor. Obvykle bývá první fází překladačů. Překladač je program, který převádí soubor napsaný ve zdrojovém jazyce do jazyka cílového. Nejčastěji jde o převod zdrojového souboru napsaného v některém z vyšších programovacích jazyků do strojového kodu cílové architektury.

Část knihovny, která se rozpoznáváním lexémů zabývá, je založená na principech činnosti lexikálního analyzátoru. V překladači může lexikální analyzátor provádět některé další úkony. Obvykle se ze zdrojového textu odstraňují komentáře a více oddělovacích znaků (např. mezera), které jsou obvykle využity jen pro zpřehlednění zdrojového kódu, se nahrazují jen jedním oddělovacím znakem. Provádět tyto činnosti v naší knihovně by nemělo smysl.

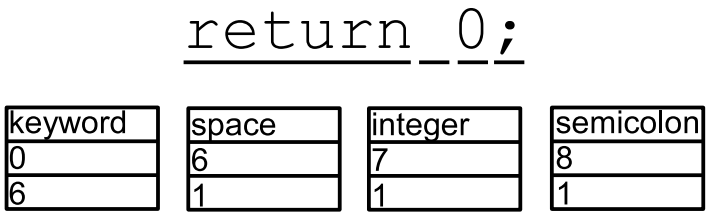
Také podoba tokenu je upravena jen pro účely knihovny. Jelikož se editovaný text ukládá do textového řetězce, je z pamětových důvodů výhodné ukládat do tokenu následující informace:

1. Typ – určuje, o který rozpoznaný lexém se jedná.
2. Pozice – udává startovní pozici (index) v textovém řetezci, na které se dáný lexém nachází.
3. Délka – odpovídá délce rozpoznaného lexému.

Výsledek lexikální analýzy vstupního řetězce v podobě tokenů je znázorněn na obrázku 2.1.

2.2 Regulární výrazy

Ve svých počátcích spadaly regulární výrazy do teorie automatů a formálních jazyků. V roce 1950 matematik Stephen Kleene pomocí své vlastní matematické notace popsal modely au-



Obrázek 2.1: Lexikální analýza

tomatů a způsoby, jak popsat a klasifikovat formální jazyky. Této notaci dal název regulární množiny.

2.2.1 Formální definice

Regulární výraz je řetězec znaků, který popisuje určitou množinu řetězců neboli regulární jazyk. Skládá se ze symbolů vstupní abecedy (Σ) nebo prázdného symbolu (ϵ) a operátorů iterace (`*`), konkatenace (`.`)a sjednocení (`|`). Vstupní abeceda Σ spolu s prázdným symbolem ϵ určuje množinu všech symbolů, ze které mohou regulární jazyk tvořit. Regulární výrazy nad abecedou Σ a jazyky, které značí, jsou definovány následovně:

1. ϕ je regulární výraz značící prázdnou množinu (prázdný jazyk)
2. ϵ je regulární výraz značící jazyk $\{\epsilon\}$
3. a, kde $a \in \Sigma$, je regulární výraz značící jazyk $\{a\}$
4. r a s jsou regulární výrazy značící jazyky L_r a L_s, platí:
   (a) $(r.s)$ je regulární výraz značící jazyk $L = L_r\, L_s$
   (b) $(r|s)$ je regulární výraz značící jazyk $L = L_r \cup L_s$
   (c) $(r*)$ je regulární výraz značící jazyk $L = L_r*$

Z těchto definic byly vytvořeny další operátory, které umožnují zapsat základní konstrukce jedušším způsobem. Zde jsou uvedeny jen tři nejčastěji používané a jejich ekvivalentní zápis pomocí základních regulárních výrazů.

1. `[]` alternativní operátor pro sjednocení
   - `[ace]` odpovídá regulárnímu výrazu `(a|c|e)`
   - `[a-e]` odpovídá regulárnímu výrazu `(a|b|c|d|e)`
2. `+` 1 až N opakování řetězce tvořeného daným regulárním výrazem
   - `((a)+)` odpovídá regulárnímu výrazu `((a).(a)*)`
3. `?` 0 nebo 1 výskyt řetězce tvořeného daným regulárním výrazem
   - `((a)?)` odpovídá regulárnímu výrazu `((a)|`ϵ`)`

2.3 Konečné automaty

Konečný automat je teoretický výpočetní model používaný v informatice k popisu jednoduchých systémů. Tento systém se může nacházet v jednom ze svých stavů, mezi kterými přechází jen na základě symbolů, které čte ze vstupu. Množina stavů je konečná. Tento výpočetní model je schopný rozpoznávat jen regulární jazyky.

2.3.1 Formální definice

Konečný automat je definován jako uspořádaná pětice (S, Σ, σ, s, F), kde:

1. S je konečná množina stavů.
2. Σ je konečná množina vstupních symbolů, nazývaná abeceda.
3. σ je přechodová funkce (tabulka), popisující pravidla přechodů mezi stavy. Může mít dvě podoby:
   - $S \times \Sigma \rightarrow S$ pro deterministický konečný automat.
   - $S \times \{\Sigma \cup \epsilon\} \rightarrow \mathbf{P}(S)$ pro nedeterministický konečný automat.
4. s je počáteční stav, $s \in S$.
5. F je množina koncových stavů, $F \subseteq S$.

2.3.2 Popis činnosti automatu

Na počátku se nachází konečný automat ve svém počátečním stavu. V každém dalším kroku přečte jeden symbol ze vstupu a přejde do stavu, který je dán hodnotou přechodové funkce. Tato hodnota se zjistí z přechodové tabulky podle přečteného symbolu a stavu, ve kterém se automat právě nachází. Tento postup se opakuje. Pokud automat skončí ve stavu, který patří do množiny koncových stavů, pak to znamená, že byl daný vstup přijat. Množina všech řetězců, které konečný automat přijme tvoří regulární jazyk.

2.3.3 Nedeterministický konečný automat

Tento automat má v každém poli přechodové tabulky celou množinu stavů. Znamená to, že může s jedním přečteným symbolem přejít do některého z těchto stavů. Obvykle se tento stav zapamatuje a postupně se projdou všechny stavy, do kterých můžeme s tímto symbolem přejít. Navíc nedeterministický konečný automat obsahuje tzv. ϵ-přechody. Mezi stavy, které jsou spojeny ϵ-přechody automat přechází, aniž by četl další vstupní symbol. V tomto případě se opět procházejí všechny možnosti.

Pro rozpoznávání lexémů pomocí nedeterministického konečného automatu potřebujeme vytvořit pro každý lexém samostatný automat. Jednotlivé automaty postupně procházejí vstupní řetězec a pokud některý z nich řetězec akceptuje, zapamatujeme si informaci o délce tohoto lexému. Po dokončení tohoto procesu se vybere lexém s největší délkou.

2.3.4 Deterministický konečný automat

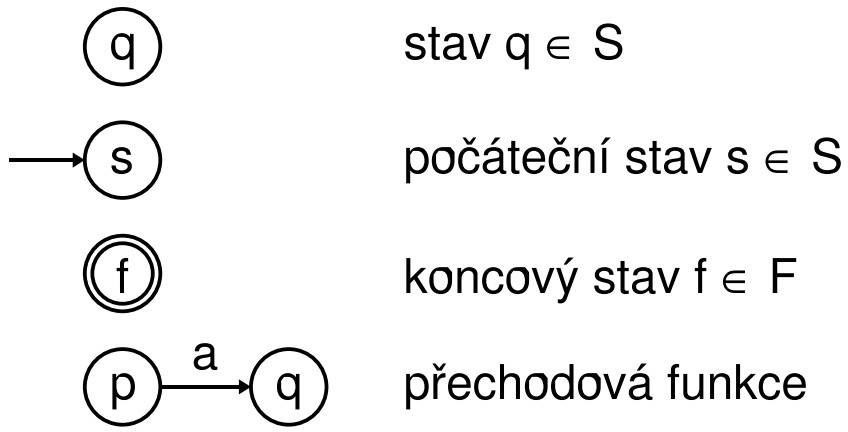
Přechodá tabulka udává jednoznačně každému stavu pro daný vstupní symbol, který stav bude po přečtení tohoto symbolu následovat. Deterministický konečný automat neobsahuje ϵ-přechody. Pokud automat přejde do koncového stavu, znamená to, že byl rozpoznán některý lexém.

V této knihovně se lexémy rozpoznávají pomocí deterministického konečného automatu, protože se určitá část editovaného textu bude rozpoznávat při každé změně. Složitost rozpoznávání pomocí deterministického konečného automatu je závislá jen na délce lexému ($O(|x|)$, kde $|x|$ představuje délku lexému). U nedeterministického konečného složitost závisí na celkovém počtu rozpoznávaných lexémů a délce lexému ($O(|r|\times|x|)$, kde $|r|$ udává celkový počet stavů a $|x|$ délku rozpoznávaného lexému).

Konstrukce deterministického konečného automatu je časově náročnější. Musíme z nedeterministického konečného automatu vytvořit deterministcký nebo použít algoritmus pro přímou konstrukci deterministického konečného automatu (lze nalézt v [1]). Je dokázáno, zě pro každý nedeterministický konečný automat je možné vytvořit ekvivalentní deterministický konečný automat.

2.3.5 Reprezentace konečného automatu pomocí grafu

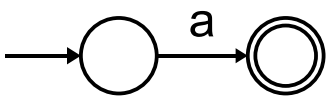
Konečný automat lze popsat pomocí orientovaného grafu, kde z každého uzlu vede hrana ohodnocená symbolem z Σ nebo prázdným symbolem ϵ. Každý uzel představuje stav automatu a každá hrana přechodovou funkci. Konečný automat obsahuje právě jeden počáteční uzel a množinu (i prázdnou) stavů koncových. Význam používaných grafických symbolů je popisuje obrázek 2.2.



Obrázek 2.2: Význam symbolů použitých pro grafickou reprezentaci

2.3.6 Konstrukce nedeterministického konečného automatu

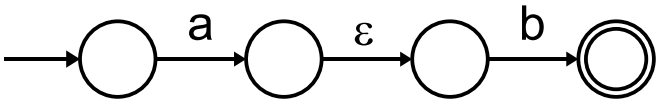
Pro každý symbol abecedy nebo pro prazdný symbol daného regulárního výrazu se vytvoří jednoduchý automat, který obsahuje dva stavy a přechod mezi nimi právě s tímto symbolem (viz. obrázek 2.3).



Obrázek 2.3: Vytvoření konečného automatu pro vstupní symbol `a`

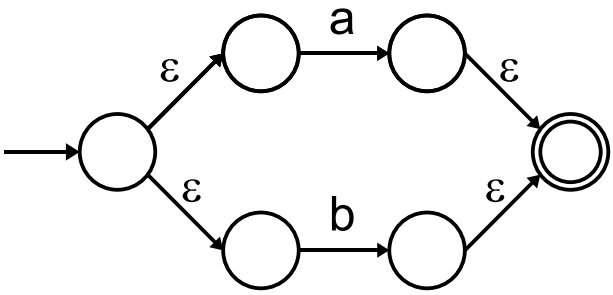
Dále pro každý operátor nahradíme některé z těchto automatů, novým automatem podle níže uvedených pravidel (Thompsonův algoritmus). Postupně nám tak vznikne jeden nedeterministický konečný automat. Jeho koncový stav je dán posledním stavem tohoto automatu.

U konkatenace potřebujeme přidat ϵ-přechod mezi posledním stavem prvního automatu a prvním stavem druhého autmatu, jak je znázorněno na obrázku 2.4.



Obrázek 2.4: Konkatenace

Při sjednocení vytvoříme nový stav, ze kterého vedou dva ϵ-přechody do prvních stavů obou automatů jejichž sjednocení provádíme. Dále vytvoříme nový stav, do kterého povedou ϵ-přechody posledních stavů těchto automatů. Přehledněji je situace vyjádřena na obrázku 2.5.

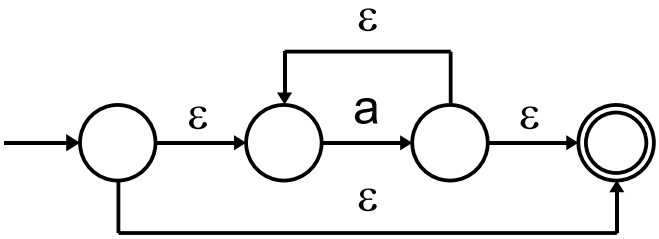


Obrázek 2.5: Sjednocení

Pro iteraci potřebujeme opět vytvořit dva stavy. Mezi nimi vytvoříme ϵ-přechod. Dále přidáme ϵ-přechod z prvního nově vytvořeného stavu do prvního stavu automatu, poté z posledního stavu automatu do druhého nově vytvořeného stavu. Na závěr potřebujeme zajistit opakování řetězce, proto přidáme ϵ-přechod z posledního stavu do prvního stavu automatu. Vše je jasné z obrázku 2.6.

2.3.7 Převod nedeterministického konečného automatu na deterministický

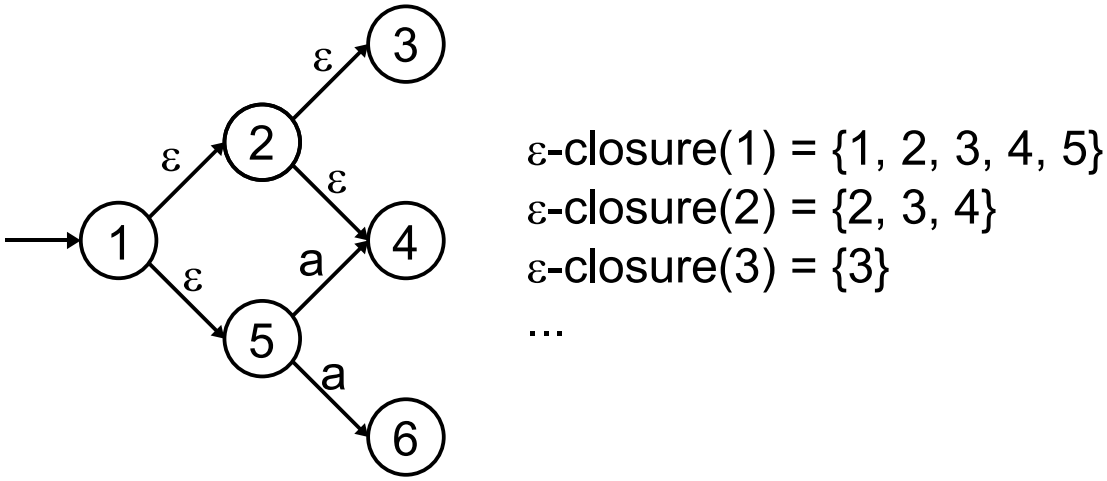
Základní myšlenka spočívá v odstranění ϵ-přechodů a odstranění nedeterminismu. Každý stav deterministického konečného automatu se vytvoří z množiny stavů nedeterministického



Obrázek 2.6: Iterace

konečného automatu. Pokud je některý stav z této množiny koncový, pak je i takto vytvořený stav deterministického konečného automatu koncový.

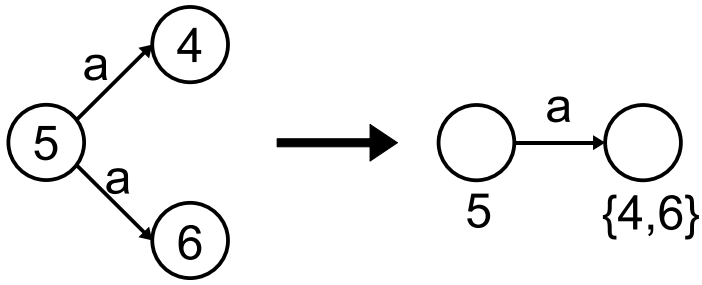
K odstranění ϵ-přechodů se využívá algoritmus nazývaný ϵ-uzávěr (anglicky ϵ-closure). Algoritmus pro vstupní stav nebo množinu stavů určí všechny stavy, kterých lze dosáhnout pomocí jednoho nebo více ϵ-přechodů. Tento algoritmus je znázorněn na obrázku 2.7.



Obrázek 2.7: Algoritmus ϵ-closure.

Při odstraňování nedeterminismu potřebujeme vytvořit množinu všech stavů, do kterých je automat schopen přejít přečtením daného vstupního symbolu. Situaci ilustruje obrázek 2.8.

Počáteční stav deterministického konečného automatu vytvoříme z množiny stavů dané ϵ-uzávěrem počátečního stavu nedeterministického konečného automatu. Tento stav uložíme do fronty stavů, které ještě nebyly zpracovány. Pro první nezpracovaný stav ve frontě, zjistíme množinu stavů NFA, ze kterých byl vytvořen a provedeme přechod s prvním vstupním symbolem ze všech stavů obsažených v této množině. Tím dostaneme novou množinu stavů, na kterou aplikujeme ϵ-uzávěr. Výsledkem je opět množina stavů NFA, která je použita pro konstrukci nového stavu DFA, pokud z této množiny již nebyl některý ze stavů ve frontě vytvořen. Při vytvoření nového stavu přidáme tento stav do fronty k dalšímu zpracování. Nyní potřebujeme přidat do zpracovávaného stavu přechod pro aktuální symbol buď do nově vytvořeného stavu, pokud byl vytvořen, nebo do stavu, který již fronta obsahuje. Tento postup provedeme pro všechny vstupní symboly. Nyní máme zpracován

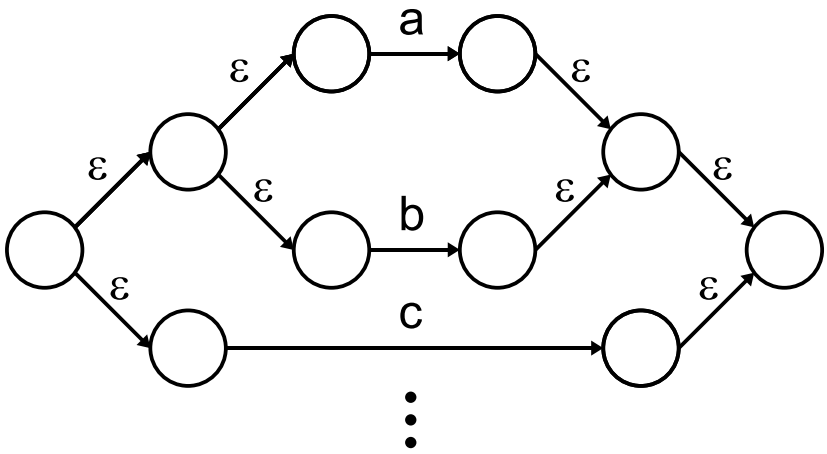


Obrázek 2.8: Odstraňování nedeterminismu

počáteční stav DFA. Tento stav označíme jako zpracovaný a začneme zpracovávat další doposud nezpracovaný stav ve frontě. Tento postup opakujeme dokud nejsou zpracovány všechny nově vytvořené stavy.

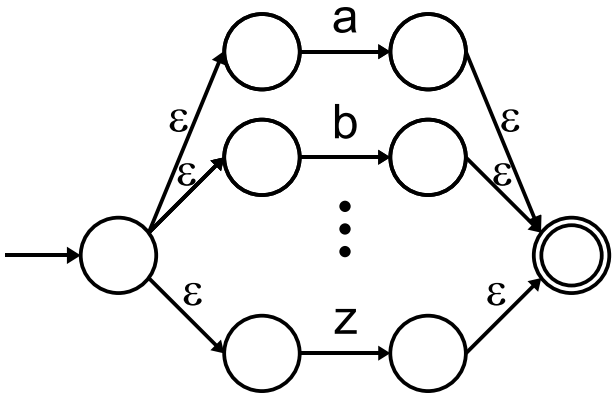
2.3.8 Složitost Thompsonova algoritmu

Nevýhodou Thopsonova algritmu je vytváření velkého množství stavů s ϵ-přechody při vyhodnocování regulárního výrazu s dlouhou sekvencí operátorů sjednocení `|`. Tato dlouhá sekvence vzniká například v regulárním výrazu `[a-z]`, který odpovídá regulárnímu výrazu `(a|b|...|y|z)`. Při sjednocení n symbolů vznikne kvůli tomuto algoritmu $2(n-1)$ stavů a $4(n-1)$ ϵ-přechodů. Situace je přehledně viditelná na obrázku 2.9.



Obrázek 2.9: Situace při vyhodnocování sekvence operátrů sjednocení

Tento problém značně zpomaluje převod NFA na DFA. Situaci by vyřešilo použití algoritmu, který by při sjednocení n symbolů vytvořil jen 2 stavy a $2n$ ϵ-přechodů. Automat vytvořený tímto způsobem je ukázán na obrázku 2.10.



Obrázek 2.10: Možné řešení pomocí jiného algoritmu

Kapitola 3

Analýza a návrh knihovny

Před samotnou implementací je potřeba analyzovat požadavky na knihovnu a vytvořit její návrh.

3.1 Požadavky na knihovnu

Knihovna musí umožňovat zvýraznění jednotlivých lexémů editovaného textu, jako jsou klíčová slova, párové symboly, literály, řádkové a blokové komentáře apod. Definice jednotlivých lexémů by měla být měnitelná. Knihovna by měla být použitelná nezávisle na zvolené platformě.

3.2 Funkční návrh knihovny

Z uvedených požadavků na funkce knihovny vyplývá, že potřebujeme zajistit rozpoznání lexémů. Již bylo zmíněno, že toto rozpoznání zajišťuje lexikální analýza. Tento proces by měl být co nejefektivnější, proto by bylo vhodné rozpoznávat co nejmenší možnou část zkoumaného textu. Rozpoznávání tedy omezíme jen na aktuálně viditelný text v okně textového editoru. Při editaci textu stačí vyhodnotit jen řádek, na kterém ke změně došlo. Pokud by změna probíhala uvnitř blokového komentáře, je třeba zahájit rozpoznávaní od začátku tohoto komentáře.

Definice lexémů budou uloženy v samostatném konfiguračním souboru mimo knihovnu. Do tohoto souboru uživatel nadefinuje podobu rozpoznávaných lexémů. Knihovna soubor zpracuje a vytvoří konečný automat pro účely lexikální analýzy.

Knihovna musí mít k disopzici text, jehož lexikální anlýzu bude provádět. Ten bude v knihovně uložen pomocí textového řetězce. Dále potřebuje knihovna komunikovat s textovým editorem a při změnách editovaného textu reflektovat tyto změny v textovém řetězci. Po těchto změnách je potřeba znovu analyzovat část textu. Textovému editoru potřebujeme dodávat informace o tom, jaké části textu má zvýrazňovat a jakým způsobem. Tyto informace budou předávány pomocí tokenů.

K uvedeným činnostem knihovna potřebuje uchovávat následující informace:

- aktuální pozice – určuje pozici kurzoru v editovaném textu
- počet znaků na řádku – určuje maximální délku řádku v textovém editoru
- počet řádků – udává počet zobrazených řádků v textovém editoru

- pozice prvního řádku – udává pozici prvního viditelného řádku v textovém editoru
- textový řetezec – obsahuje editovaný text
- pole tokenů – obsahuje tokeny vytvořené lexikální analýzou

3.3 Syntaxe regulárních výrazů

V praxi se obvykle používají rozšířené definice regulárních výrazů. Knihovna pracuje se základními regulárními výrazy a navíc umožnuje použít alternativní operátor pro sjednocení (`[]`). Zbylé rozšířené regulární výrazy lze jednoduše nahradit ekvivalentními základními regulárními výrazy, jak je uvedeno výše. Dále se v praxi nepoužívá operátor konkatenace (`.`). Zápis `a.b` je ekvivalentní zápisu `ab`.

Pokud potřebujeme zapsat prázdný symbol ϵ, je nutné jej zadat pomocí sekvence znaků \0. Všechny nezobrazitelné znaky se zapisují pomocí sekvence znaků \`ASCII`, kde `ASCII` přestavuje dekadickou hodnotu ASCII kódu daného znaku. Tyto sekvence se nesmějí nacházet uvnitř operátoru `[]`. V tomto případě je nutné použít konstrukci s operatátorem `|`. V tabulce 3.1 jsou uvedeny všechny znaky, které mají zvláštní význam a je nutné zadat pomocí těchto znakových sekvecí.

Znak	**Popis**	**ASCII**
ϵ	prázdný symbol	\0
`CR`	carrige return	\10
`LF`	line feed	\13
`(`	levá závorka	\40
`)`	pravá závorka	\41
`*`	operátor iterace	\42
`-`	mínus	\45
`[`	levá hranatá závorka	\91
\	zpětné lomítko	\92
`]`	pravá hranatá závorka	\93
`|`	operátor sjednocení	\124

Tabulka 3.1: Znaky se zvláštním významem a jejich hodnota ASCII kódu

3.4 Konfigurační soubor

Jak již bylo zmíněno, tento soubor slouží pro nadefinování podoby rozpoznávaných lexémů. Definice lexémů musejí být uloženy v následující podobě:

```
název_lexému
     regulární_výraz
     další_regulární_výraz
     ...
další_název_lexému
```

```
...
```

Zde je uvedena konkrétní ukázka:

```
keyword
     if
     else
identifier
     [_a-zA-Z][_a-zA-Z0-9]*
startblock
     {
endblock
     }
```

Na prvním řádku je název rozpoznávaného lexému. Název musí odpovídat jednomu z řetězců uvedených v tabulce 3.2. Následuje řádek začínající znakem *tabulátor*, za kterým následuje

Název	**Popis**
`space`	mezera
`tab`	tabulátor
`linefeed`	konec řádku
`datatype`	datový typ
`keyword`	klíčové slovo
`identifier`	identifikátor
`integer`	přirozené číslo
`real`	reálné číslo
`multiline`	blokový komentář
`singleline`	řádkový komentář
`oper`	operátor
`textlit`	textový literál
`charlit`	znakový literál
`beginblock`	začátek bloku
`endblock`	konec bloku
`lpar`	levá závorka
`rpar`	pravá závorka
`lbra`	levá hranatá závorka
`rbra`	pravá hranatá závorka
`semicolon`	středník
`backslash`	zpětné lomítko
`error`	nerozpoznaný lexém

Tabulka 3.2: Seznam názvů rozpoznávaných lexémů

regulární výraz. Těchto řádků může být pro jeden rozpoznávaný lexém více. Výsledkem je pak sjednocení všech regulárních výrazů uvedených na samostatných řádcích pro tento lexém. Výše uvedenou ukázku zápisu lexému `keyword` můžeme zapsat také tímto ekvivalentním zápisem:

```
keyword
     if|else
```

Pokud některou z definic vynecháme, pak tento lexém nebude rozpoznáván. V tabulce 3.2 jsou uvedeny všechny předdefinované názvy pro rozpoznávané lexémy. Uvedené názvy nemusejí vyjadřovat význam zadaného regulárního výrazu. Záleží na uživateli, co bude chtít ve skutečnosti zvýraznovat. Předpokládá se však, že budou využity ke svému skutečnému účelu. Navíc k vyznačení párových symbolů slouží lexémy `beginblock`–`endblock`, `lpar`–`rpar` a `lbra`–`rbra`. Tyto mohou být použity ke zvýraznění například těchto párových symbolů v programovacím jazyce C { }, `()` a `[]`.

Kapitola 4

Implementace knihovny

Knihovna byla napsána v programovacím jazyce *C++* a vyvíjena pod operačním systémem *Linux* v textovém editoru *vi*.

4.1 Objektový návrh knihovny

Po analýze problému vytvoříme objektový návrh knihovny. V programovacím jazyce C++ jsou objekty popsány pomocí tříd, které definují vnitřní datové struktury, metody objektu a další vlastnosti (více například v [3]). Do značné míry jsou využity šablony ze standardní knihovny C++. Pro přehlednost nebudou uvedeny všechny pomocné proměnné a algoritmy, které knihovna využívá. Budou zde popsány jen nejdůležitější části knihovny a jejich princip činnosti. Knihovna se skládá z několika tříd. Samotnou knihovnu představuje třída `synlib`. Ta ke své činnosti využívá další pomocné třídy. Ve třídě `token` je nadefinovaná podoba tokenu (viz kapitola 2). Další třída `rex` vytváří z konfiguračního souboru konečný automat. Třída `rex` používá třídu `nfa_state` a `dfa_state`, které představují stavy DFA a NFA.

4.2 Třída `synlib`

Tato třída definuje rozhraní knihovny. Obsahuje textový řetězec (`string`), který uchovává editovaný text. Dále je tvořena objektem třídy `rex`, který slouží k rozpoznávání lexémů. Lexémy jsou pomocí tokenů uloženy do pole (šablona `vector`). Kvůli efektivitě se rozpoznává jen viditelný text. K těmto účelům potřebuje knihovna promněnné pro uchování aktuální pozice, maximálního počtu znaků zobrazitelných na jednom řádku, počtu zobrazených řádků a pozici prvního zobrazovaného řádku. K nastavení těchto proměnných slouží metody této třídy, které budou volány z textového editoru. Rozpoznávání lexémů, se děje vždy při editaci textu. K těmto účelům slouží metody `insert` a `remove`. Tyto metody rovněž volá textový editor.

4.2.1 Metoda `insert`

Textový editor nastaví pozici, na kterou chceme nově vkládaný text nebo znak vložit. Následně zavolá tuto metodu, které předá jako vstupní parametr textový řetězec obsahující vkládaný text. Tato funkce sama zajistí rozpoznání potřebné části textu. Před vložením textu se prozkoumá pole s tokeny. Pokud obsahuje nějaký nerozpoznaný lexém, máme pozici, od které začneme editovaný text rozpoznávat. V opačném případě se pokusí nalézt

začátek a konec řádku, na kterém se právě nacházíme. Pokud není konec řádku nalezen, rozpoznáváme jen zbylou viditelnou část textu. Před samotným rozpoznáváním je potřeba z pole odstranit tokeny, které budou nově rozpoznány, a nastavit pozici, kam budou vloženy nové tokeny. Pokud po tomto procesu obsahuje pole za vkládací pozicí další tokeny, je potřeba jejich startovní pozici inkrementovat o délku vkládaného textu. Následně můžeme vložit text do textového řetězce a rozpoznat určenou oblast.

4.2.2 Metoda `remove`

Podobně jako u metody `insert` nastavíme pozici, od které chceme smazat daný počet znaků. Tento počet znaků předáme metodě jako vstupní parametr. Opět potřebujeme najít pozice, mezi kterými potřebujeme modifikovaný text znovu rozpoznat. Tento postup je obdobný jako při vkládání.

4.3 Třída `rex`

Hlavní činností je zpracování souboru s definicemi rozpoznávaných lexémů a následné vytvoření nedeterministického konečného automatu. O kontrukci NFA se stará metoda `create_nfa`. Ten je převeden metodou `nfa_to_dfa` na deterministický konečný automat, pomocí kterého probíhá lexikální analýza. NFA a DFA jsou uloženy pomocí šablony `vector`, která obsahuje ukazatele na jednotlivé stavy. Při implementaci konverze NFA na DFA byl zjištěn problém s časovou náročností tohoto převodu. Proto byly vyvinuty metody, které vnitřní podobu těchto automatů uloží do binárních souborů. NFA a DFA jsou pak načteny z těchto binárních souborů a nemusejí se znova dlouze vytvářet. Stavy NFA a DFA popisují třídy `nfa_state` a `dfa_state`, jak bude uvedeno v následující podkapitole.

4.3.1 Metoda `create_nfa`

Tato metoda vytvoří počáteční stav NFA. Každý regulární výraz je převeden do vnitřní podoby. V první fázi jsou rozšířené definice regulárního výrazu nahrazeny základními. Dále jsou všechny symboly, které představují operátory, nahrazeny jinými symboly. Tyto symboly již nekolidují s žádným symbolem v regulárním výrazu. V další fázi se hodnoty zadané pomocí ASCII kódu nahradí jejich skutečnou podobou. Nakonec se doplní speciální symbol pro konkatenaci. Z každého regulárního výrazu je vytvořen nedeterministický konečný automat pomocí Thompsonova algoritmu. Koncové stavy a odpovídající názvy lexémů jsou uloženy do asociativního pole (šablona `map`). Do počátečního stavu celkového NFA přidáme ϵ-přechod pro počateční stav všech automatů vytvořených z regulárních výrazů. Tím máme vytvořen NFA a pro každý jeho koncový stav víme, který rozpoznaný lexém představuje.

4.3.2 Metoda `nfa_to_dfa`

Převod NFA na DFA zajištuje tato metoda. Nejprve si popíšeme dvě základní funkce, které algoritmus pro tento převod využívá:

- ϵ-`closure` – vstupem je množina stavů NFA. Výstupem je opět množina stavů, kterých je automat schopen dosáhnout z dané vstupní množiny bez čtení vstupních symbolů.

- `move` – vstupem je množina stavů a vstupní symbol. Výstupem je množina všech stavů, do kterých je automat schopen přejít se vstupním symbolem pro danou vstupní množinu.

Algoritmus pro převod NFA na DFA vypadá následovně:

1. Počáteční stav DFA se vytvoří pomocí funkce ϵ-`closure` z počátečního stavu NFA.
2. Pro každý nový stav DFA provedeme následující úkony pro všechny vstupní symboly:
   (a) Voláme funkci `move`.
   (b) Na výsledek bodu (a) zavoláme funkci ϵ-`closure`. Výsledkem může být množina stavů, která již je v našem DFA obsažena. V tomto případě, jen přidáme přechod pro aktuální symbol z právě zpracovávaného stavu DFA do tohoto již existujícího stavu. V opačném případě vytvoříme nový stav DFA z výsledku tohoto bodu a přidáme přechod do nově vzniklého stavu DFA.
3. Pro každý nově vytvořený stav provedeme bod 2.
4. Koncové stavy DFA jsou ty, které byly vytvořeny z alespoň jednoho koncového stavu NFA.

4.4 Třída `nfa_state` a `dfa_state`

Tyto třídy přestavují stav konečného automatu. Každý stav obsahuje svůj jednoznačný identifikátor. Dále je tvořen tabulkou (šablona `multimap` udávající stav nebo množinu stavů, kterých lze z tohoto stavu dosáhnout pro daný vstupní symbol. Třída `dfa_state` navíc obsahuje množinu stavů NFA, ze kterých byl tento stav DFA vytvořen. Dvě hlavní metody této třídy jsou `add_trans` a `reach`. První slouží k přidání přechodu s daným symbolem do dalšího stavu. Druhá metoda vrací pro daný vstupní symbol množinu stavů, kterých lze s tímto symbolem dosáhnout. Třída `dfa_state` obsahuje navíc metodu `get_nfa`, která vrací množinu stavů NFA.

4.5 Pomocné nástroje

Rozhraní textového editoru bylo vytvořeno pomocí toolkitu FLTK. Knihovna byla pomocí tohoto editoru testována pod operačním systém Linux.

Dále byla implementována jednoduchá konzolová aplikace sloužící pro vytvoření konečných automatů z konfiguračního souboru. Po vytvoření se automaty uloží do binárních souborů. Při spuštění textového editoru knihovna tyto binární soubory načte a opět vytvoří automaty. Tento proces je však mnohokrát rychlejší než tvorba automatů z konfiguračního souboru.

Kapitola 5

Závěr

Vytvořená knihovna podporuje zvýraznění syntaxe v textovém editoru. Je nezávislá na cílové platfomě a napsaná v programovacím jazyce C++. Umožňuje definovat podobu zvýrazňovaných lexémů pro více programovacích jazyků. Tyto definice jsou ukládány do konfiguračního souboru knihovny. Z něj se pomocí jednoduché aplikace vytváří konečné automaty sloužící pro lexikální analýzu. Tato činnost je časově náročná, proto aplikace vytvořené automaty uloží do binárních souborů. Tento proces je nutné opakovat, pokud modifikujeme konfigurační soubor nebo vytváříme nový. Při spuštění textového editoru knihovna vytvoří automaty z těchto binárních souborů. V textovém editoru je možné pomocí menu nastavit, který binarní soubor má být pro zvýraznování lexémů použit.

Zásadní nevýhodou této knihovny je časová náročnost převodu nedeterministického konečného automatu na deterministický. Tento problém by bylo možné vyřešit použitím jiných algoritmů pro tvorbu deterministického konečného automatu. Dalším možným řešením by bylo provádět lexikální analýzu pomocí nedeterministického konečného automatu.

Značným přínosem pro knihovnu by byla možnost zadávát regulární výrazy pomocí rozšířených regulárních výrazů. Definiční soubor by byl pak pro komplikovanější definice přehlednější.

Literatura

[1] Aho, A. V., Lam, M. S., Sethi R., Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006. ISBN 0-321-48681-1.

[2] Meduna, A. *Automata and Languages: Theory and Applications*. Springer, 2000. ISBN 1-85233-074-0.

[3] Bruce, E. *Thinking in C++, Volume One: Introduction to Standard C++* [online]. URL `http://www.mindviewinc.com/downloads/TICPP-2nd-ed-Vol-one.zip`, 2000. ISBN 0-13-979809-9.

[4] WWW stránky. Wikipedia [online]. URL `http://wikipedia.org`.

Dodatek A

Návod na použití knihovny

Pokud chceme využít tuto knihovnu v jiném programu, potřebujeme vložit hlavičkový soubor s rozhraním knihovny do zdrojového kódu tohoto programu.

```
#include ``synlib.h''
```

Nyní můžeme vytvořit objekt představují knihovnu. Pokud chceme z konfiguračního souboru vytvořit binární soubory s konečnými automaty, potřebujeme do konstruktoru zadat jméno konfiguračního souboru a zavolat metodu `load_conf`. Vytvořené soubory se budou jmenovat stejně jako konfigurační soubor, navíc budou mít přípony `.nfa` a `.dfa`.

```
synlib *lib = new synlib(``pascal.def'');
lib->load_conf();
```

K načtení binárních souborů slouží metody `load_nfa` a `load_dfa`. Těmto metodám předáme jako vstupní parametr název souboru s příslušným automatem.

```
lib->load_nfa(``pascal.nfa'');
lib->load_dfa(``pascal.dfa'');
```

Pro nastavení stavových proměnných slouží metody `set_start` (nastavení aktuální pozice), `set_col` (počet znaků na řádku), `set_fl` (pozice prvního řádku) a `set_nol` (počet viditelných řádků). Před vkládáním textu je potřeba nastavit pozici, na kterou chceme text vložit, pomocí metody `set_start`. Následně voláme metody `insert` a `delete`, které zajišťují vkládání a mazaní textu. Tyto metody také volají metodu `scan`, která zajistí rozpoznání lexémů. Lexémy následně získáme pomocí metody `get_next_token`, která postupně vrací všechny lexémy.