1 Anotace

Hlavní projekt je dynamicky linkovaná knihovna pro Windows implementující parsování textu na lexikální tokeny definované pomocí regulárních výrazů. Vedlejší testovací projekt ukazuje možné použití knihovny.

2 Reprezentace regulárních výrazů

Regulární výraz je:

- znak
- backslashovaný speciální znak
- [...] - výčet hodnot(v závorce mohou být nespeciální znaky, backslashované speciální znaky, nebo výčty hodnot a-z, A-Z, 0-9)
- .

Pokud e_1, e_2 jsou regulární výrazy, potom je regulární výraz taky:

- e_1*
- e_1+
- (e_1)
- e_1e_2
- $e_1|e_2$

3 Testovací projekt

Testovací program testuje parsování regulárních výrazů, deterministické automaty vytvořené z regulárních výrazů a parsování souborů na tokeny. Testování je prováděno pomocí příkazů na konzoli. Knihovna je k projektu přilinkovaná staticky. Před spuštěním programu je třeba sestavit projekt LexicalAnalyzer. LexicalAnalyzer.dll je v post-build eventu testovacího projektu zkopírován z výstupu knihovny do výstupního adresáře testovacího projektu.

Všechny soubory berou jako první argument soubor s příponou .in, jako argument se zadává pouze název **bez přípony**.

Příkaz t vypíše syntaktické stromy regulárních výrazů ze souboru zadaného jako první argument. Na každém řádku souboru se nachází jeden regulární výraz. Soubor `good_reg.in` obsahuje správně vytvořené výrazy, `bad_reg.in` obsahuje špatně vytvořené výrazy.

Příkaz m má jako argument soubor, který má na lichých řádcích regulární výrazy, a na sudých řádcích text který má být otestován regulárním výrazem z předchozího řádku. Příkaz vypíše jestli text odpovídá reg. výrazu a deterministický automat odpovídající reg. výrazu. Testovací soubor je `matching.in`.

Příkaz a má dva argumenty, soubor který má na lichých řádcích názvy tokenů, a na sudých řádcích regulární výrazy odpovídající tokenu na předchozí řádce. Druhý argument je název souboru i **s příponou**, obsahující text k parsování. Příkaz vypíše tokeny a jejich hodnoty, nebo že text nešlo pomocí výrazů naparsovat. Výstup lze zapsat do souboru zadaného jako třetí argument. Dvojice testovacích souborů jsou:

- `tokens0.in, text0s.txt`
- `tokens0.in, text0f.txt` - parsování neuspěje
- `tokensSimpleCpp.in, cppCode.txt` - parsování některých tokenů c++
- `tokensSimpleXml.in, books.xml`[1] - parsování některých tokenů xml

4 Knihovna

Implementace

Knihovna se skládá z tříd reprezentujících stavové automaty - `dfa, nfa`; tříd vytvářejících stavové automaty z regulárních výrazů - `dfa_builder, nfa_builder`; tříd umožňujících spouštění stavových automatů - `dfa_runner, nfa_runner`; třídy vytvářející `dfa` z regulárního výrazu `regexp_parser`; třídy tokenizer, která pomocí regulární výrazů rozdělí text na tokeny.

`regex_parser` načítá regulární výrazy pomocí rekurzivního sestupu pro následující gramatiku:

$$E \to T|E \mid T$$
$$T \to FT \mid F$$
$$F \to B* \mid B+ \mid B$$
$$B \to . \mid P \mid [R] \mid (E)$$
$$R \to PR \mid P$$
$$P \to \backslash n \mid \backslash t \mid \backslash r \mid \backslash . \mid \backslash * \mid \backslash + \mid \backslash (\mid \backslash) \mid \backslash [\mid \backslash] \mid \backslash | \mid a$$

Během načítání, pomocí metody `create_machine` je rovnou rekurzivně konstruován nedeterministický stavový automat. Funkce vracející stavový automat pro neterminály a funkce skládající automaty jsou ve třídě `nfa_builder`, která se také stará o číslování stavů. Pokud je zadán neplatný regulární výraz, je vyhozena výjimka `invalid_argument`.

`nfa` obsahuje tři typy hran v přechodové funkci: hranu odpovídající neterminálu(libovolné hodnotě typu char), hranu odpovídající lambda přechodu a

[1] Soubor převzatý z `https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms762271(v%3Dvs.85)`

hranu odpovídající libovolnému terminálu(tu by šlo nahradit 256 hranami neterminálů, ale tato optimalizace ušetří čas a místo při převodu na deterministický automat). Běh `nfa` zajišťuje `nfa_runner`, ten je také použit při konstrukci deterministického automatu.

`dfa` je vytvořen pomocí třídy `dfa_factory`. `dfa_factory` simuluje průchod `nfa`, pro každý stav dosažitelný z počátečního stavu najde všechna písmena na vycházejících hranách. Pro každé nalezené písmeno najde lambda uzávěr přechodu pomocí tohoto písmena a přidá ho s příslušnou hranou do stavů `dfa` , pokud ještě nebyl přidán. Pokud některá písmena nejsou na hranách ven ze stavu použita, přidá se přechod do prázdného stavu, pomocí speciální hrany. Nakonec jsou ve vytvořeném `dfa` pomocí Floyd-Warshallowa algoritmu nalezeny stavy, ze kterých se nelze dostat do žádného koncového stavu.

Rozhraní

Knihovna exportuje metody tříd `regex_parser`, `dfa_runner`, `dfa_factory` a `tokenizer`.

Pomocí metody `create_machine` třídy `regex_parser` lze zkontrolovat jestli je zadaný regulární výraz syntakticky správně. Metoda `create_dfa` třídy `dfa_factory` vytváří instance `dfa_runner` odpovídající zadanému regulárnímu výrazu.

`dfa_runner` umožňuje testovat, jestli text odpovída regulárnímu výrazu. Text se zadává po znacích pomocí metody `move`. Testování jestli text odpovídá výrazu zajišťuje metoda `matches`. Třída dále obsahuje metody `longest_matching_prefix`, vracející nejdelší prefix odpovídající regulárnímu výrazu, `has_matching_prefix`, která říká jestli takový prefix existuje, a `failed`, říkající jestli automat neuspěl.

`tokenizer` umožňuje pomocí metod `add_token_type` a `remove_token_type` přidávat a odebírat regulární výrazy s názvy tokenů, kterým odpovídají. Po specifikování tokenů lze metodou `set_input` zadat text k naparsování. Metodou `next_token` je naparsován další token, metoda `get_token` tento token vrací. Pomocí metod `finished` a `correct_input` lze zjistit, jestli je parsování dokončeno a jestli při něm nastala chyba. Pokud při parsování dojde k chybě, je vypsána chybová hláška s číslem řádky a pozicí, kde k chybě došlo.