VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PARALELNÍ LEXIKÁLNÍ ANALYZÁTOR

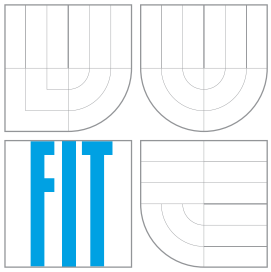
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ JEŽEK

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PARALELNÍ LEXIKÁLNÍ ANALYZÁTOR

PARALLEL LEXICAL ANALYZER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUKÁŠ JEŽEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. MARTIN ČERMÁK
SUPERVISOR

BRNO 2010

Abstrakt

Tato práce se zabývá generováním lexikálního analyzátoru, který analyzuje soubor paralelně, tj. několika vlákny. Analyzátor je vygenerován na základě regulárního jazyka a reprezentován konečným automatem s potřebnými funkcemi. Důraz je zde kladen právě na paralelní zpracování. Diskutuje možnost zpracování dopřednými a zpětnými vlákny. V implementaci byla zvolena metoda dopředných vláken. Rozebírá problémy, které se vyskytly při implemetaci a způsobily, že výsledný program nedosáhl téměř žádného zrychlení.

Abstract

This bachelor thesis deals with generating a lexical analyzer which analyzes a file in parallel, i.e. by several threads at a time. The analyzer is generated on the basis of regular language and is represented by finite automaton with the necessary functions. The emphasis is placed on the parallel processing. The possibility of forward and back threads processing is discussed in this thesis. The method of the forward threads was decided to be used for implementation. The problems during implementation that lead to almost none of the desired speed-up of the analysis are discussed.

Klíčová slova

Paralelní lexikální analyzátor, konečný automat, regulární výraz, vlákna.

Keywords

Parallel lexical analyzer, finite automaton, regular expression, threads.

Citace

Lukáš Ježek: Paralelní lexikální analyzátor, bakalářská práce, Brno, FIT VUT v Brně, 2010

Paralelní lexikální analyzátor

Prohlášení

Prohlašuji, že jsem předkládanou bakalářskou práci vypracoval samostatně pod vedením pana Ing. Martina Čermáka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Lukáš Ježek
14. května 2010

Poděkování

Chtěl bych poděkovat mému vedoucímu Ing. Martinu Čermákovi za jeho odbornou pomoc, rychlé odpovědi a časovou flexibilitu při zjednávání konzultací. Dále bych chtěl poděkovat spolužákům Petru Špačkovi a Miroslavu Sobotkovi za propůjčení testovacích počítačů.

Obsah

Kapitola 1

Úvod

Cílem této bakalářské práce je vytvořit program, který vygeneruje lexikální analyzátor v jazyce C. Konečný automat, tvořící základ lexikálního analyzátoru, je vygenerovaný na základě regulárního jazyka. Analýza souboru probíhá paralelně pomocí několika vláken, která čtou pouze část souboru. Tokeny, které lexikální analyzátor vrací, jsou vraceny v pořadí, v jakém se v souboru skutečně nacházejí.

Druhá kapitola obsahuje základní matematické definice pojmů, použítých v tomto textu.

V kapitole 3 je uvedeno několik informací o překladači a fázích zpracování, kterými prochází analyzovaný soubor. Dále je nastíněn i důvod tvorby paralelního lexikálního analyzátoru.

Kapitola 4 obsahuje vztah mezi regulárními výrazy, konečnými automaty a jejich využití při implementaci lexikálního analyzátoru. Popisuje důvod využití speciálních znaků a některé odlišnosti v různých implementacích regulárních výrazů. Protože se budou regulární výrazy převádět na konečný automat, obsahuje také dvě možné reprezentace automatů.

Pátá kapitola je zaměřena na problematiku paralelního zpracování. Popisuje, co jsou to vlákna a jejich rozdíl oproti procesům. Upozorňuje na mechanismy, řešící chyby souběhu a jejich důvod použití. V neposlední řadě se zabývá dvěmi možnostmi čtení souboru více vlákny — dopřednými a protichůdnými. Rozebírá i nové problémy, které toto zpracování přináší.

Kapitola 6 upřesňuje zadání. Především definuje vstup a výstup vytvořeného programu. Je zde také zmíňka o aplikaci *flex*, ze které byla přejata syntaxe souboru, popisujícího regulární jazyk.

V 7. kapitole je uvedena přesná syntaxe vstupního souboru. Způsob převodu regulárních výrazů do postfixové notace a následný převod na nedeterministický konečný automat, který je převeden na deterministický konečný automat. Ten je pak ve formě tabulky vygenerován do výstupního souboru, představujícího lexikální analyzátor.

Kapitola 8 obsahuje popis některých klíčových funkcí, které jsou použity v jednotlivých fázích převodu. Popisuje také struktury, reprezentující jednotlivé stavy automatů. U generovaného souboru popisuje hlavní funkce a některé jejich proměnné, zajišťující většinu funkčnosti lexikálního analyzátoru. Rozebírá různé způsoby implementace vláken a jejich front. Většina těchto způsobů očekávané zrychlení nepřinesla, proto je zde diskutovaná pravděpodobná příčina neúspěchu.

Závěr shrnuje problémy, na které se narazilo při implementaci paralelního zpracování. Jsou zhodnoceny naměřené výsledky. Diskutuje, zda má smysl se zabývat paralelní analýzou a snažit se o paralelismus při překladu aplikací. Nabízí cesty, které se jeví jako možné řešení, vedoucí k cíli. Zároveň upozorňuje na cesty, které k němu pravděpodobně nepovedou.

Kapitola 2

Základní definice a pojmy

Tato kapitola obsahuje základní definice pojmů, které jsou použité dále v textu. Pro podrobnější informace doporučuji nahlédnou například do [7].

2.1 Definice základních pojmů

Definice 2.1.1 (Abeceda a symboly) *Abeceda je konečná, neprázdná množina prvků, které se nazývají symboly.*

Prázdné slovo je označováno symbolem ε.

Definice 2.1.2 (Slovo) *Nechť Σ označuje abecedu. Pak:*

- *ε je slovo nad abecedou Σ,*
- *pokud je x slovo nad touto abecedou a $b \in \Sigma$, pak také xb je slovo nad abecedou Σ.*

Místo pojmu *slovo* se také můžeme setkat s výrazem *řetězec*, který má stejný význam.

Definice 2.1.3 (Jazyk) *Nechť Σ označuje abecedu a nechť $L \subseteq \Sigma^*$. Pak L se nazývá jazykem nad abecedou Σ.*

Σ^* označuje množinu všech slov nad abecedou Σ, včetně ε.

Definice 2.1.4 (Reverzní slovo) *Nechť x je slovo nad abecedou Σ. Reverzní slovo x, reverse(x), je definováno jako:*

- *pokud $x = \varepsilon$, pak $reversal(x) = \varepsilon$,*
- *když $x = b_1 \dots b_n$ pro $n \geq 1$ a $b_i \in \Sigma$, pro $i = 1 \dots n$, pak $reversal(b_1 \dots b_n) = b_n \dots b_1$.*

Definice 2.1.5 (Reverzní jazyk) *Nechť L je jazyk. Reverzní jazyk L, reversal(L), je definován jako:*

$$reversal(L) = \{reversal(x) : x \in L\}$$

2.2 Regulární výrazy

V následující definici 2.2.1 se uvažují operátory `*`, `.` a `+`. Operátory v tomto pořadí označují iteraci, konkatenaci a sjednocení.

Definice 2.2.1 (Regulární výrazy) *Nechť Σ označuje abecedu. Pak regulární výrazy nad abecenou Σ a jazyky, které tyto výrazy popisují, jsou definovány následovně:*

- *$\varnothing$ je regulární výraz, popisující prázdný jazyk (prázdnou množinu),*
- *ε je regulární výraz, popisující jazyk $\{\varepsilon\}$,*
- *a, kde $a \in \Sigma$, je regulární výraz, popisující jazyk $\{a\}$.*
- *A nechť x a y jsou regulární výrazy, popisující jazyky L_x a L_y, potom:*
  - *$(x.y)$ je regulární výraz, popisující jazyk $L = L_x L_y$,*
  - *$(x + y)$ je regulární výraz, popisující jazyk $L = L_x \cup L_y$,*
  - *(x^*) je regulární výraz, popisující jazyk $L = L^*$.*

Pro regulární výraz r, označuje $L(r)$ jazyk, popsaný právě výrazem r.

Definice 2.2.2 (Regulární jazyk) *Nechť L je jazyk. Pak L je regulárním jazykem nad abecedou Σ, když $L = L(r)$ pro regulární výraz r nad abecedou Σ.*

Definice 2.2.3 (Postfixová notace) *Nechť Σ označuje abecedu, jejíž symboly popisují operandy. Postfixová notace je definována rekurzivně následovně:*

- *když a je infixový výraz, $a \in \Sigma$, pak a je také postfixová notace,*
- *pokud U a V jsou infixové výrazy popsané postfixovým výrazem X a Y a $\circ$ je operátor $\circ \in \{+, -, *, /\}$, pak $\circ XY$ je postfixový výraz popisující $U \circ V$,*
- *když (U) je infixový výraz, kde U popisuje postfixový výraz X, pak X je postfixová notace popisující (U).*

Postfixová notace je někdy označována jako reverzní polská notace[1].

2.3 Konečný automat

Definice 2.3.1 (Nedeterministický konečný automat – NKA) *Konečný automat je pětice $M = (Q, \Sigma, R, s, F)$, kde:*

- *Q je konečná množina stavů,*
- *Σ je vstupní abeceda,*
- *R je konečná množina pravidel tvaru $pa \to q$, kde $p, q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$,*
- *s je počáteční stav,*

[1] Reverzní proto, že v prefixové notaci se operátor píše před operandy. Prefixovou notaci vymyslel polský matematik Jan Łukasiewicz. Více na [3].

- $F \subseteq Q$ *je konečná množina koncových stavů.*

Zápis $pa \rightarrow q$ se nazývá pravidlem.

Definice 2.3.2 (Konečný automat bez ε–přechodů) *Nechť $M = (Q, \Sigma, R, s, F)$ je nedeterministický konečný automat. Pokud pro každé pravidlo $pb \rightarrow q \in R$, kde $p, q \in Q$ a $b \in \Sigma$ platí, že $b \in \Sigma$ a $b \neq \varepsilon$. Pak automat M je konečný automat bez ε–přechodů.*

Definice 2.3.3 (Konfigurace) *Nechť $M = (Q, \Sigma, R, s, F)$ označuje konečný automat. Konfigurace automatu M je slovo χ definované jako:*

$$\chi = Q\Sigma^*$$

Definice 2.3.4 (Deterministický konečný automat – DKA) *Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat bez ε–přechodů. Pokud pro každé pravidlo $pb \rightarrow q \in R$, kde $p, q \in Q$ a $b \in \Sigma$ platí, že množina $R \setminus \{pa \rightarrow q\}$ neobsahuje žádné pravidlo s levou stranou pa. Pak automat M je deterministický konečný automat.*

Definice 2.3.5 (Přechod) *Nechť pbx a qx jsou dvě konfigurace konečného automatu M, kde $p, q \in Q$, $b \in \Sigma \cup \{\varepsilon\}$ a $x \in \Sigma^*$. A nechť $r = pb \rightarrow q \in R$ je pravidlo. Potom M může provést přechod z pbx do qx za použití pravidla r a přechod lze zapsat $pbx \vdash qx[r]$ nebo $pbx \vdash qx$.*

Definice 2.3.6 (Epsilon uzávěr) *Pro každý stav $p \in Q$ je ε–uzávěr, ε–closure, definován jako:*

$$\varepsilon\text{–}closure(p) = \{q : q \in Q, p \vdash^* q\}$$

Kapitola 3

Překladač

Překladač transformuje vstupní řetězec na významově ekvivalentní výstupní řetězec. Přitom vstupní řetězec, resp. soubor, ve kterém je uložen, prochází několika fázemi zpracování, které lze rozdělit do dvou skupin. Toto dělení ale není striktní, jak je uvedeno dále v kapitole. V té jsou popsány fáze zpracování (také viz. [8]), kterými může transformace vstupního řetězce procházet. S tím související rozdíl mezi kompilátorem, interpretem a hybridním překladačem. Dále uvádí, zda má smysl první fázi překladu urychlovat.

3.1 Analýza vstupního řetězce

Prvním krokem je analýza vstupního řetězce. Skládá se z lexikální, syntaktické a sémantické analýzy.

Lexikální analýza čte vstupní řetězec z daného jazyka a rozpoznává základní jednotky jazyka, tzv. lexémy. K rozpoznanému lexému se připojí typ a tak vznikne token. Pokud se například přečte číslo `5`, lexémem je hodnota *5* a typem *celé číslo*. V případě slova `paralelismus` je lexémem tento text a typ lze označit například jako *řetězec*. V dalších fázích se pracuje převážně s typem tokenu (tj. celé číslo, řetězec ...). Lexému v tokenu se říká atribut a zpravidla se používá v posledních fázích překladu.

Každý jazyk je definovaný gramatikou. Ta specifikuje strukturu vstupního řetězce, tj. posloupnost jednotlivých lexémů. Tuto činnost zajišťuje syntaktický analyzátor. Přijímá tokeny a kontroluje, zda jejich posloupnost odpovídá gramatice jazyka. Výsledkem syntaktické analýzy bývá zpravidla abstraktní syntaktický strom.

Ten v poslední fázi analýzy zpracovává sémantický analyzátor. Zajišťuje převážně kontrolu datových typů popř. zařídí implicitní přetypování.

Analýza vstupního řetězce však nemusí procházet striktně těmito třemi oddělenými kroky. Často se například provádí sémantická kontrola spolu se syntaktickou.

3.2 Fáze syntézy

Po analýze vstupního řetězce se provádí fáze syntézy. Ze syntaktického stromu se generuje mezikód. Obvykle se jedná o pomocný jazyk, který se dále snáze zpracovává. Ten může být optimalizován a verifikován. Z takto upraveného mezikódu se v posledním kroku generuje cílový kód.

Fáze syntézy se liší podle druhu překladače. Všechny fáze, uvedené v této podkapitole, jsou součástí syntézy většinou pouze v případě generického překladače (kompilátoru), kdy

je cílový kód vygenerován na konkrétní architekturu v binární podobě (např. jazyk C, Pascal...). Naopak u interpretačního překladače (interpretu) se cílový kód vůbec negeneruje, ale rovnou se zpracovává mezikód (např. Python, Perl...). Tím jsou omezené i možnosti optimalizace mezikódu. V případě hybridních překladačů se generuje kód, nezávislý na operačním systému. Tento kód je pak interpretován pomocí interpretu nainstalovaného na počítači (např. Java...).

3.3 Možnosti zrychlení lexikální analýzy

Lexikální analyzátor čte vstupní soubor znak po znaku a od začátku do konce. Rozpoznává lexémy a vytváří z nich tokeny, které dále zpracovává syntaktický analyzátor.

Na dnešních moderních procesorech s několika jádry je možné zpracovávaný program paralelizovat a tak rozložit zátěž na všechna jádra. Při překladu každého programu je nutné provést jeho analýzu. Ta v případě rozsáhlých aplikací může být časově náročná. Tento problém lze pravděpodobně vyřešit paralelizací analýzy a tím tento proces urychlit.

Paralelizací lexikální analýzy se zabývá právě tato práce. Uvažuje, že vygenerovaný lexikální analyzátor bude analyzovat paralelně vstupní soubor. Vytvářené tokeny bude vracet v pořadí, v jakém se v souboru skutečně nacházejí. Na výstupu tedy dojde opět k serializaci. Analyzátor je generovaný na základě regulárního jazyka, který ho popisuje.

Kapitola 4

Rozšíření nad regulárními výrazy a konečné automaty

Jedním možným prostředkem pro popis regulárního jazyka (viz. definice 2.2.2) je použití regulárních výrazů (viz. definice 2.2.1). Pomocí nich lze jednoduše definovat vlastnosti jazyka.

Pokud je například definován regulární jazyk, obsahující pouze slovo *lex*, regulární výraz, který tento jazyk popisuje je `lex`. Definování jazyka pouhým výčtem slov, které má obsahovat, by bylo velice náročné a v některých případech možná i nemožné. Z tohoto důvodu regulární výrazy zavádějí tzv. speciální znaky, které definici jazyka zjednodušují.

Konečný automat představuje nejběžnější model pro přijímání regulárních výrazů. Základem konečného automatu je vstupní páska obsahující symboly. Ty se čtou od začátku souboru až do jeho konce a to znak po znaku. Po přečtení symbolu se přejde z počátečního stavu do stavu následujícího. Pokud nelze do jiného stavu přejít a automat se nachází v konečném stavu, je přijatý příslušný lexém. V případě, že se však v konečném stavu automat nenachází, nastala při zpracování chyba. Může však nastat situace, kdy se sice automat v konečném stavu nachází, ale je možné s načteným symbolem přejít dále. V tomto případě se provede přechod do následujícího stavu a to i tehdy, jestliže následující stav oproti předchozímu není konečný.

Konečný automat lze reprezentovat buď graficky nebo pomocí tabulky. Každá z reprezentací má přesně definované formy zápisu, které jsou popsány v této kapitole.

4.1 Speciální znaky regulárních výrazů

Protože regulární výrazy jsou v informatice velice používané, množství speciálních znaků a jejich způsob zápisu se může v některých detailech lišit. Nejčastějším rozdílem je právě zápis speciálních znaků. V případě, že znak `+` je speciálním znakem, potom, pokud má být zapsán znak sčítání, musí být zapsán ve tvaru `\+`. Je ale možné, že v jiných nástrojích bude zápis opačný a `\` se použije pro zápis speciálního znaku. Některé základní speciální znaky používané v regulárních výrazech obsahuje tabulka 4.1. Pro více informací o konstrukcích a zvláštních případech regulárních výrazů je možné nahlédnou např. do literatury [4].

Kromě speciálních znaků, definují některé implementace regulárních výrazů také proměnné. Ty mají zpřehlednit a urychlit zápis regulárního výrazu. Ale i v tomto případě se zápis proměnných liší. Například posixová implmentace definuje pro čísla $0-9$ proměnnou `[[:digit:]]`. Naproti tomu implementace perlu pro čísla $0-9$ používá proměnnou `\d`.

Tabulka 4.1: Některé operátory a metaznaky regulárních výrazů

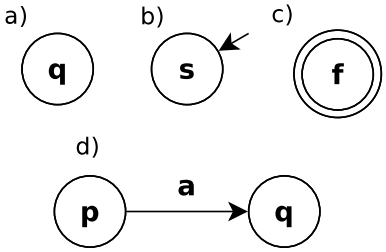
Znak	Funkce
`[ ]`	Výčet znaků. Zápisem `[0-9]`, je přijímaným jazykem jazyk $L = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
`|`	Volba několika možností. Zápisem `a|b`, je přijímaným jazykem jazyk $L_1 = \{a\}$ nebo $L_2 = \{b\}$.
`( )`	Používá se pro práci s blokem znaků. Výraz `(ab)*` znamená, že přijímaný jazyk je $L = \{(ab)^m : m \geq 0\}$.
`-`	Rozsah znaků. Používá se spolu s metaznaky `[ ]`. Pokud je přijímaný jazyk $L = \{a, b, c\}$, nejsnažším způsobem je použít `[a-c]`.
`?`	Opakování $0 - 1$. Výrazem `a?` je přijímán jazyk $L = \{a^m : m \in \{0, 1\}\}$.
`*`	Opakování $0 - n$. Výrazem `a*` je přijímán jazyk $L = \{a^m : m \geq 0\}$.
`+`	Opakování $1 - n$. Výrazem `a+` je přijímán jazyk $L = \{a^m : m \geq 1\}$.
`{m,n}`	Opakování $m - n$. Výrazem `a{2,5}` je přijímán jazyk $L = \{a^m : 2 \leq m \leq 5\}$.
`{m}`	Opakování m. Výrazem `a{5}` přijímán jazyk $L = \{a^m : m = 5\}$.
`{m, }`	Opakování $m - n$. Výrazem `a{3, }` přijímán jazyk $L = \{a^m : m \geq 3\}$.

Dalším rozšířením je zapamatování částí nalezeného řetězce podle regulárního výrazu. Pro příklad uveďme regulární výraz `([a-c])([0-3])\1\2`. Hodnota výrazu odpovídající `[a-c]` se zapamatuje. A tato hodnota se znovu vyvolá použitím `\1`. Pokud se použije řetězec *ab01ab01*, pak odpovídá regulárnímu výrazu. V prvním kroku se totiž nalezl podřetězec *ab* a *01*. Oba se zapamatovaly a později byly nalezeny dále v řetězci. Naopak v případě řetězce *ab01ac01*, by regulárnímu výrazu nevyhověl. Podřetězec *ac* sice odpovídá regulárnímu výrazu `[a-c]`, ale nevyhovuje předchozímu nalezenému řetězci, tj. *ab*.

4.2 Grafická reprezentace konečného automatu

Stav $q \in Q$ se zakresluje pomocí kruhu, v jehož středu je název stavu (viz obr. 4.1/a). Pokud je stav $s \in Q$ počáteční, směřuje do něj šipka, jak je znázorněno na obr. 4.1/b). K zakreslení konečného stavu $f \in F$ je použito dvojitého kruhu (viz obr. 4.1/c) K reprezentaci přechodu ze stavu $p \in Q$ do stavu $q \in Q$ po přečtení symbolu $a \in \Sigma$ se používá orientovaná šipka vedoucí ze stavu $p \in Q$ do stavu $q \in Q$. Symbol, po jehož přečtení lze přechod provést, se zapisuje k šipce tak, jak je uvedeno na obrázku 4.1/d.

Obrázek 4.1: Základní elementy grafické reprezentace konečného automatu.



4.3 Tabulková reprezentace konečného automatu

Dalším možným způsobem je reprezentace konečného automatu pomocí tabulky tak, jak je popsáno v [7]. Sloupce tabulky reprezentují přečtený symbol, příslušící do množiny $\Sigma \cup \{\varepsilon\}$. Každý řádek reprezentuje jeden stav z množiny stavů Q. Řádky tabulky obsahují stavy, ve kterých se automat nachází. Sloupce pak symboly, kterými lze do těchto stavů přejít. Pokud se automat nachází ve stavu s_2, může, podle tabulky 4.2, po přečtení symbolu a přejít pouze do stavu s_3. Ostatní přechody jsou nepřípustné. Bývá zvykem zapisovat počáteční stav do prvního řádku tabulky. Konečné stavy se podtrhávají.

Tabulka 4.2: Konečný automat ekvivalentní regulárnímu výrazu $a^*b|ba^*$.

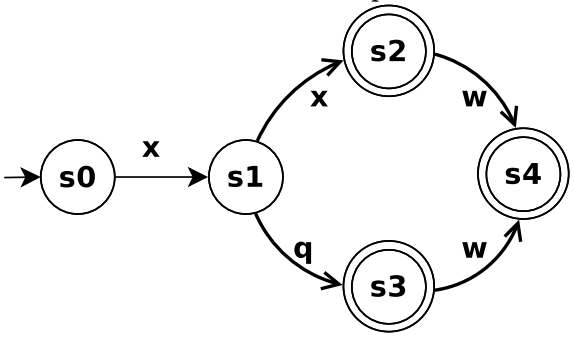
	a	b	ε
s_1	$\varnothing$	$\varnothing$	$\{s_2, s_4\}$
s_2	$\{s_3\}$	$\varnothing$	$\varnothing$
$\underline{s_3}$	$\varnothing$	$\{s_3\}$	$\varnothing$
s_4	$\{s_4\}$	$\{s_5\}$	$\varnothing$
$\underline{s_5}$	$\varnothing$	$\varnothing$	$\varnothing$

Pro lepší názornost vztahu mezi regulárním výrazem, konečným automatem a jeho různými reprezentacemi, uvažujme výraz `x(x|q)w?`. Odpovídající grafická reprezentace konečného automatu je na obrázku 4.2 a jeho tabulková reprezentace na obrázku 4.3. Počátečním stavem je s_0. Uvažujme na vstupní pásce řetězec xw. Po přečtení symbolu `x` automat přejde do stavu s_1. Se symbolem `w` nemůže automat provést žádný přechod a proto vstupní řetězec zamítne.

Nyní uvažujme na vstupní pásce řetězec xqw. Do stavu s_1 automat přejde stejně, jako v předchozím případě po přečtení symbolu `x`. Se symbolem `q` automat může provést přechod do stavu s_3. Svou činnost však neukončí, protože vstupní páska obsahuje ještě některé symboly. Přečte tedy symbol `w` a přejde tak do stavu s_4, kde také ukončí svou činnost. Automat tedy řetězec xqw přijme.

Konečný automat ekvivalentní regulárnímu výrazu $x(x|q)w?$.

Obrázek 4.2: Grafická reprezentace.



Obrázek 4.3: Reprezentace tabulkou.

	x	q	w
s_0	$\{s_1\}$	$\varnothing$	$\varnothing$
s_1	$\{s_2\}$	$\{s_3\}$	$\varnothing$
$\underline{s_2}$	$\varnothing$	$\varnothing$	$\{s_4\}$
$\underline{s_3}$	$\varnothing$	$\varnothing$	$\{s_4\}$
$\underline{s_4}$	$\varnothing$	$\varnothing$	$\varnothing$

4.4 Vztah regulárních výrazů, konečného automatu a lexikálního analyzátoru

Jak je uvedeno výše, regulárními výrazy se definuje regulární jazyk. Ty se převedou na ekvivalentní konečný automat, který přijímá stejný regulární jazyk.

Lexikální analyzátor pak využívá právě činnosti tohoto automatu. Vstupní páska konečného automatu je nahrazena souborem, ze kterého čte lexikální analyzátor jednotlivé symboly. S přečtenými symboly přechází automat postupně do definovaných stavů.

Pokud konečný automat řetězec nepříjme, pak se ukončí celá činnost. V praxi by však takové řešení nebylo vhodné. Je snaha, i v případě nepřijatého řetězce, v analýze pokračovat. Jestliže taková situace, kdy se řetězec nepříjme nastane, označí se místo v souboru jako chybé a v analýze se pokračuje. Na konci analýzy jsou potom všechny chyby vypsány najednou.

Pokud k chybě nedojde a konečný automat přejde do konečného stavu, pak příjme daný řetězec. Lexikální analyzátor poté z tohoto řetězce, lexému, vytvoří token. Podle konečného stavu rozezná typ lexému, který uloží do tokenu. Následně se může řetězec (lexém) uložit jako atribut tokenu.

Kapitola 5

Paralelní zpracování

Převážná většina programů čte soubor od začátku do konce. V dnešní době jsou procesory rychlé a při čtení z disku program zbytečně čeká a není tak efektivně využitý strojový čas procesoru. Sofistikovanější programy při čtení souboru provádějí i jiné operace, například výpočty nebo potřebnou režii programů. Tím efektivně využijí strojový čas, který mají přidělený od plánovače. Takové programy využívají vláken popř. procesů.

V této kapitole je nastíněna problematika vláken. Čím se liší od procesů, jejich výhody a nevýhody. Pro lepší pochopení problematiky vláken nebo procesů, doporučuji především [9, kapitola 2.2]. Dále jsou zde probrány dva možné způsoby čtení souboru více vlákny, výhody a nevýhody jednotlivých způsobů.

5.1 Vlákna

Každý proces je samostatný a nezávisí na ostatních běžících procesech. Pokud má být v rámci jednoho programu vykonáno více činností zároveň, nabízí se možnost vytvořit další procesy. Nevýhodou tohoto přístupu však je, že každý proces má svůj adresový prostor, globální proměnné, otevřené soubory atd. Tento problém řeší vlákna. Běžný proces lze považovat za program s jedním vláknem, jak je znázorněno na obrázku 5.1/a. Pokud operační systém, resp. jeho jádro, podporuje vlákna[1], může se v rámci jednoho procesu vykonávat několik činností, neboť každý proces může obsahovat více vláken, která mohou být spuštěna paralelně. Každý proces může mít tedy několik vláken, která jsou spuštěna paralelně. Na obrázku 5.1/b je příklad procesu s třemi vlákny.

Každé vlákno má svůj vlastní programový čítač, zásobník, stav a vlastní registry. Naopak vlákna mezi sebou sdílejí adresový prostor, globální proměnné, otevřené soubory, alarmy, signály a potomky procesů.

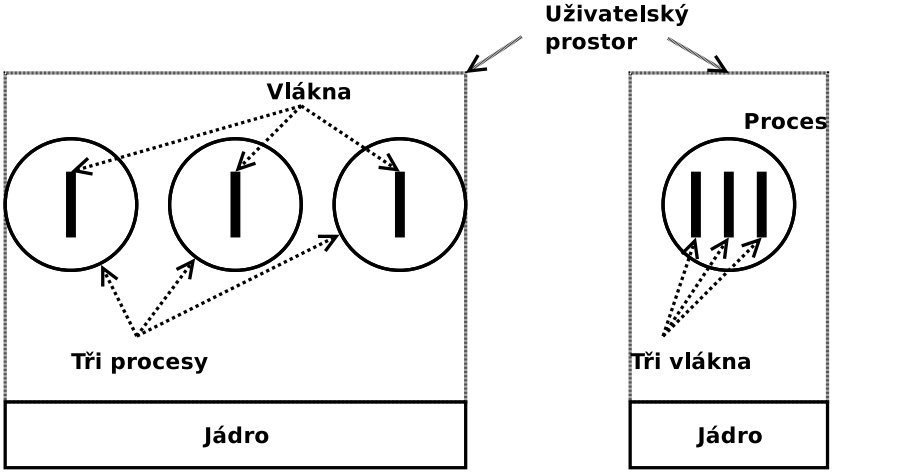
5.1.1 Správa vláken

Každé vlákno může mít svůj vlastní stav, podobně jako procesy. Prvním stavem, ve kterém se vlákno může nacházet, je stav *připravený* (angl. *ready*). To znamená, že vlákno je naplánováno plánovačem a po přidělení procesorového času začne vykonávat svou činnost. Pokud má vlákno přidělen procesorový čas, nachází se ve stavu *spuštěno* (angl. *running*). Do tohoto stavu může vlákno přejít pouze ze stavu *připravené*. V případě, že jedno vlákno

[1] Zda se o správu vláken stará přímo jádro či nikoliv, záleží na zvolené implementaci. Vlákna lze totiž impementovat v uživatelském prostoru a nebo v prostoru jádra popř. lze použít i hybridní implementaci.

provádí například výpočty a druhé čeká na vstup od uživatele, je druhé vlákno na dlouhou dobu blokováno. Dokud uživatel nezadá vstup a vlákno tak nepřejde do stavu *připravené*, nevykonává svou činnost. V tomto případě se nachází v *blokovaném* stavu (angl. *blocked*). Takto zablokované vlákno je spuštěno buď nějakou činností (jako je zadání vstupu) nebo na vyžádání jiným vláknem. Po ukončení vlákna se nachází ve stavu *ukončení* (angl. *terminated*). V tomto stavu již vlákno v podstatě neexistuje, ale pro jeho úplné zničení musí být vyzvednut jeho návratový kód.

Obrázek 5.1: Rozdíl mezi procesem a vláknem převzatý z [9].



a) Tři oddělené procesy b) Jediný proces s třemi vlákny

Protože jsou vlákna na sobě v podstatě nazávislá, existují mechanismy, jak je synchronizovat a řídit tak tok programu. Tímto nástrojem mohou být například signály. Pomocí nich lze vlákno v zablokovaném stavu dostat do stavu, kdy je připraveno na spuštení. Je však důležité, neplést tyto signály se signály jádra v unixových operačních systémech.

Vlákna mohou sdílet některá data, ať už proměnné nebo otevřené soubory. Protože není jasné, v jakém pořadí a kdy vláknům plánovač přiřadí procesorový čas, není ani jasné, v jakém pořadí a jak dlouho budou vlákna vykonávat svou činnost. Z toho důvodu je třeba tato sdílená data uzamknout, když s nimi jedno vlákno pracuje, a po dokončení práce je opět odemknout. Tak se zajistí, že s aktuálním zdrojem pracuje pouze jedno vlákno a ostatní čekají na jeho uvolnění. K tomu slouží tzv. mutexy nebo semafory.

5.2 Dopředná vlákna

První metodou, jak paralelně číst soubor, je použití dopředných vláken. Počet vláken, která analyzují vstupní soubor, není teoreticky omezený. Při použití velkého počtu však režie, potřebná k synchronizaci vláken, může vyžadovat více procesorového času než samotné zpracování souboru. Při čtení je nutné pro každé vlákno definovat začátek a konec bloku, který má zpracovávat. První vlákno čte soubor od jeho začátku. Zpracování ukončí na konci svého bloku, což může být např. v polovině souboru, kde zároveň začíná blok zpracovávaný dalším vláknem. To ukončí své čtení, až narazí na konec svého bloku. Pokud nad souborem operují pouze dvě vlákna, konec bloku druhého vlákna bude současně konec souboru.

Vzhledem k tomu, že obě vlákna pracují paralelně, kritická část je právě přechod mezi

bloky, které jsou zpracovávány rozdílnými vlákny. Druhé vlákno musí vědět odkud má číst mnohem dříve, než dokončí svou činnost první vlákno. Nejjednoduším způsobem je zařídit, aby konec prvního a začátek druhého vlákna byly uprostřed souboru. V tomto případě se musí uvažovat skutečnost, že druhé vlákno může začít číst v polovině lexému, což by způsobilo chybu, protože lexém by nebyl rozpoznán. Tomuto problému se lze vyhnout zavedením podmínky, že druhé vlákno musí svou činnost začínat v polovině souboru, ale až za prvním rozpoznaným lexémem.

5.2.1 Předbíhající se vlákna

Metodu dopředných vláken lze ještě upravit. Při použití dvou vláken nebude první vlákno zpracovávat první polovinu souboru a druhé zbylou polovinu. U této metody se zvolí velikost bloku, kterou má jedno vlákno zpracovat. Pokud příslušný blok zpracuje, zjistí, zda existuje ještě blok, který není analyzován. V případě, že takový blok existuje, začne ho zpracovávat. Jako příklad uvažujme dvě vlákna nad souborem o velikosti 26 B a s velikostí bloku zpracování 10 B. První vlákno analyzuje soubor od začátku do jeho 10. B, druhé vlákno od 10. do 20. B. Když první vlákno ukončí svou činnost, zjistí, že posledních 6 B nebude zpracováno a tak začne analyzovat soubor od 20. B do jeho konce. Naopak až ukončí svou činnost druhé vlákno, zjistí, že neexistuje blok, který by měl ještě analyzovat. Druhé vlákno tedy ukončí svou činnost.

Zároveň je vidět, že když první vlákno začalo zpracovávat druhý blok (tj. 20. – 26. B), zmenšilo velikost bloku na zpracování o 4 B. Původně by totiž končil až 30. B, což je však za koncem souboru. Díky definici velikosti bloku, který mají vlákna zpracovávat je tak umožněno jejich předbíhání a tím i střídání vláken v činnosti.

5.2.2 Víceřádkové lexémy

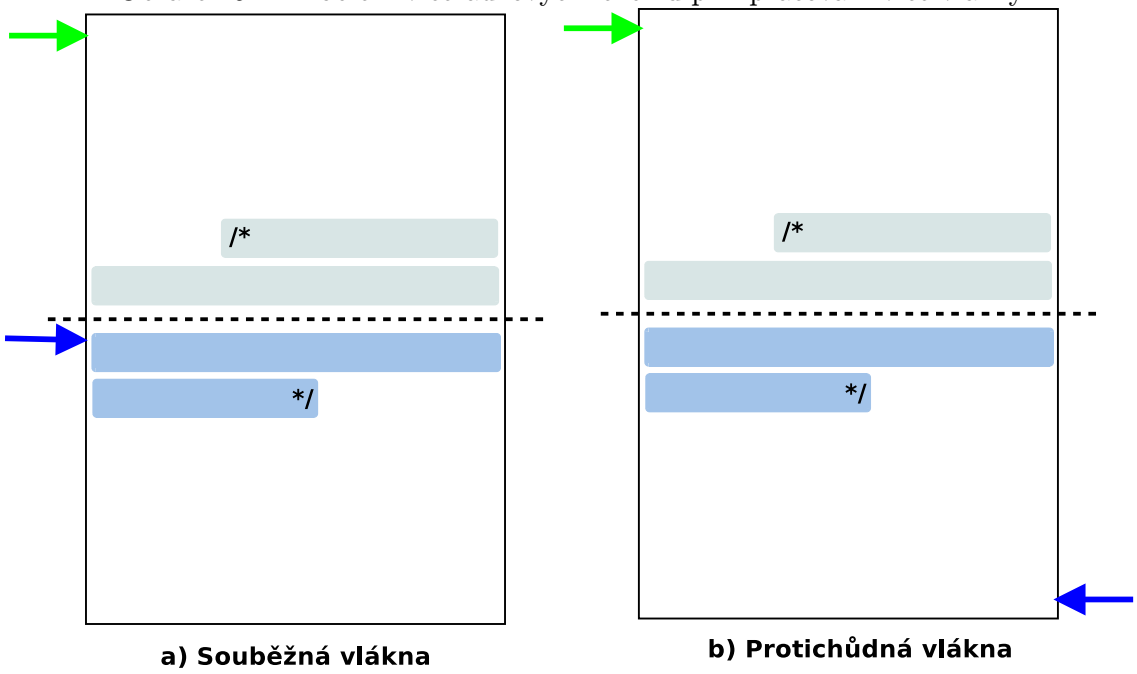
Mnohem závažnější problém se však projeví při zpracování víceřádkových lexémů, jako jsou například víceřádkové komentáře, řetězce nebo makra. Jak je vidět na obrázku 5.2/a, druhé vlákno svou činnost začíná v polovině víceřádkového komentáře.

Uvažujme, že obě vlákna začnou svou činnost současně. První vlákno pracuje dle očekávání od začátku souboru. Druhé vlákno, nacházející se uprostřed komentáře, může některé lexémy rozpoznat, ale může narazit i na chybné lexémy. Jestliže taková situace nastane, nesmí svou činnost ukončit s chybou, ale číst dále. Mezi tím první vlákno narazí na začátek komentáře /* a zahazuje všechny následující lexémy až do konce komentáře */. Může však nastat situace, kdy první vlákno narazí na konec bloku, který má zpracovávat, a ukončí svou činnost v polovině komentáře. Tento případ je znázorněn právě na obrázku 5.2/a. V takovém případě musí sdělit druhému vláknu, že lexémy, které od začátku svého bloku přečetl jsou součástí komentáře. Na základě této informace tak i druhé vlákno zahodí vše až do konce komentáře */. Teprve v případě, kdy narazí na konec bloku, který zpracovává, dříve než na konec komentáře, dává vědět tuto skutečnost dalšímu vláknu. Pokud je však konec bloku zároveň koncem souboru, jedná se o chybu, kdy nebyl ukončen řetězec.

5.3 Protichůdná vlákna

Další možností zpracování je použití protichůdných vláken. Oproti předchozí variantě, kde teoreticky na počtu vláken nezáleží, lze tuto variantu provést pouze se dvěma vlákny. První, dopředné, vlákno čte soubor od začátku a druhé, zpětné vlákno, od jeho konce.

Obrázek 5.2: Problém víceřádkových lexémů při zpracování více vlákny.



Obě vlákna mohou mít předem daný blok určený velikostí, který mají zpracovávat. Dopředné vlákno po přečtení svého bloku vyčká na dokončení zpětného vlákna a poté se ukončí. Zpětné vlákno ukončí svou činnost později, protože musí otáčet přečtené řetězce, proměnné apod.

Aby dopředné vlákno nečekalo na ukončení činnosti zpětného vlákna, lze výpočet velikosti bloku vynechat. V takovém případě se analýza ukončí, až se obě vlákna setkají. Musí se však hlídat pozice vláken, aby nebyla činnost ukončena uprostřed lexému. Pokud by se tak stalo, každé vlákno by hlásilo jinou chybu, což je další problém tohoto přístupu.

Zpětné vlákno musí mít vlastní konečný automat, který přijímá reverzní jazyk (definice 2.1.5) dopředného vlákna.

5.3.1 Víceřádkové lexémy

Bohužel, ani tento způsob zpracování souboru neřeší problém s víceřádkovými lexémy, popsaný v kapitole 5.2.2. Může nastat situace, kdy obě vlákna narazí na začátek víceřádkového lexému, jako je například blokový komentář, ale dříve, než narazí na konec komentáře, ukončí svou činnost. Příklad takového souboru je zázorněn na obrázku 5.2/b. I zde je tedy nutná vzájemná komunikace mezi vlákny a v případě, že první vlákno skončilo svou činnost v komentáři, musí oznámit tuto skutečnost druhému vláknu. Pokud se obě vlákna shodnou, že se skutečně jedná o komentář na rozhraní jednotlivých bloků, mohou rozpoznané i nerozpoznané lexémy zahodit.

Kapitola 6

Specifikace

V této kapitole je nastíněn způsob generování analyzátoru programem *flex*. Dále je popisován způsob práce s vytvářeným programem (resp. jeho vstup a výstup), jehož činností je generovat lexikální analyzátor. Od kapitoly 6.2 se tedy popisuje vlastní práce na generátoru paralelního lexikálního analyzátoru a na samotném paralelním analyzátoru. Přičemž se využívá teorie popisované v předchozích kapitolách.

6.1 Generovaný lexikální analyzátor

V dnešní době vývojáři překladačů nebo programů, využívajících lexikální analyzátor, mají v zásadě dvě cesty, kterými se mohou při vývoji vydat. V prvním případě si vytvoří celý návrh a následně ručně přepíšou konečný automat do některého z programovacích jazyků. V druhém případě nechají samotné přepsání konečného automatu do zdrojového kódu na aplikaci. Výhodou tohoto způsobu je, že při takto generovaném kódu se vývojáři vyhnou implementačním chybám, které by v další fázi museli řešit. Ušetřený čas tak můžou věnovat na části, které jsou pro program klíčové.

U této aplikace je pak ve vstupním souboru určitým způsobem popsaný konečný automat, který definuje přijímaný regulární jazyk. Výstupem je zdrojový kód v daném jazyce, reprezentující popsaný konečný automat. Aplikací, pracující tímto způsobem, je například program *flex*[1].

6.1.1 Flex

Flex je otevřenou implementaci unixového programu *lex*. Jedná se o nástroj pro generování lexikálního analyzátoru. Podrobnější informace o aplikaci *flex* lze najít v manuálových stránkách [10].

Program čte uživatelem definovaný vstupní soubor. V tomto souboru je popsán regulárními výrazy regulární jazyk přijímaný konečným automatem.

Soubor je rozdělen na tři části, které se oddělují symboly **%%**. V první části se převážně definují proměnné, které jsou charakterizovány regulárním výrazem. Tyto proměnné lze použít v definici dalších složitějších proměnných nebo v další sekci. Druhá sekce popisuje konečný automat pomocí regulárních výrazů. Ke každému lexému, popsanému regulárním výrazem, náleží zdrojový kód v jazyce C, který má být vykonán, pokud je lexém rozpoznán.

[1] Fast Lexical Analyzer Generator

Třetí část pak obsahuje uživatelem definované funkce. Tento soubor pak *flex* přeloží do zdrojového kódu v jazyce C.

6.2 Vygenerovaný soubor

Dle zadání má být vytvořen program, podobající se svou činností programu *flex*. Vstupem vytvořeného programu má být soubor, ve kterém je, pomocí regulárních výrazů, popsán jazyk, přijímaný konečným automatem. Tento vstupní soubor program transformuje a na základě údajů v něm vygeneruje odpovídající konečný automat v jazyce C, který popsaný jazyk přijímá.

Přeložený vygenerovaný program simuluje lexikální analyzátor. Analýzu provádí na vstupním souboru, předaném jako argument programu, přičemž analýza má být speciální v tom, že analyzovaný vstupní soubor se má číst několika vlákny. Jednotlivé bloky souboru jsou tak zpracovávány paralelně.

Kapitola 7

Návrh

V kapitole je souhrně uvedena struktura souboru, popisujícího generovaný lexikální analyzátor. Dále je v ní postupně rozebírán převod regulárních výrazů do postfixové notace a jejich převod na nedeterministický automat. Ten je pak převeden na deterministický konečný automat a vygenerován do výstupního souboru.

7.1 Vstupní soubor

Syntaxe a sémantika jazyka ve vstupním souboru je inspirovaná programem *flex*. Konečný automat je tedy popsán pomocí regulárních výrazů a soubor je rozdělen do tří sekcí oddělených `%%`. Speciální znaky regulárních výrazů, které je možné použít při popisu lexémů jsou uvedeny v teoretické části práce v tabulce 4.1.

7.1.1 Sekce definic

První sekcí, která je nepovinná, je sekce s definicemi. Pokud má být na začátek výsledného zdrojového souboru zkopírovaný nějaký kód, uzavře se mezi `%{` a `}%`. Tato konstrukce je vhodná pro definici proměnných a maker, použitých ve výsledném kódu. Jestliže nemá být do výsledného zdrojového kódu nic kopírováno, lze tuto část vynechat. Dále se zde mohou definovat vlastní proměnné pro zpřehlednění kódu, zapsaného v tomto souboru. Tím je možné vytvářet složité regulární výrazy, ale zároveň zachovat dobrou čitelnost zdrojového kódu. Pokud má být dříve definovaná proměnná použita, musí se uzavřít mezi složené závorky { a }.

Na jednom řádku může být definice jen jedné proměnné. Řádek začíná názvem proměnné a za mezerou nebo tabelátorem následuje vlastní regulární výraz, jak je ukázáno v příkladě.

```
%{
#include<stdlib.h>

#define DIGIT
#define VARIABLE
%}

digit  [0-9]
letter [a-zA-Z]
var    {letter}({letter}|{digit})*
```

7.1.2 Sekce popisující přijímaný jazyk

Tato sekce je jako jediná povinná. Pomocí regulárních výrazů popisuje konečný automat, který přijímá požadovaný jazyk. Na jednom řádku je definován jeden lexém. Řádek začíná regulárním výrazem popisujícím přijímaný lexém. Za ním následuje blok, který má být do výsledného souboru zkopírován, pokud je daný lexém rozpoznán. Regulární výraz a blok ke kopírování musí být odděleny mezerou. Blokem ke kopírování se myslí vše mezi symboly { a } a tento blok je do výsledného zdrojového souboru zkopírován bez jakýchkoliv kontrol a úprav.

Při definici víceřádkových lexémů, jako jsou například komentáře nebo řetězce, se do bloku ke kopírování musí přidat speciální makra. Zároveň je nutné definovat počáteční a konečný lexém. Rozliší se tak lexém /* a */. K bloku ke kopírování se přidá makro `PLA_INGNORE_BEG`, označující začátek vícečádkového lexému, a `PLA_IGNORE_END`, označující konec víceřádkového lexému. Všechny lexémy mezi /* a */ tak budou ignorovány. Pokud má být jejich hodnota vrácena jako atribut tokenu, použijí se makra `PLA_PROCESS_BEG` a `PLA_PROCESS_END`. Jestliže začátek i konec víceřádkového lexému označuje stejný symbol, jako např. “ u řetězců, označí se makrem `PLA_PROCESS`, resp. `PLA_IGNORE`. Není tak třeba definici začátku a konce lexému oddělovat.

V příkladě níže se používají proměnné `digit`, `var` a `letter`, které jsou definovány v první sekci. Pokud se tedy rozpozná číslo, činnost, která se vykoná je popsána kódem `return DIGIT;`.

```
{digit}  { return DIGIT; }
{var}    { return VARIABLE; }
{letter} { userFunction (); }
/\*      { PLA_IGNORE_BEG return BLOCK_COMMENT; }
\*/      { PLA_IGNORE_END return BLOCK_COMMENT; }
“       { PLA_PROCESS return STRING; }
%%
```

7.1.3 Sekce s uživatelským kódem

Do poslední sekce, která je do výsledného zdrojového souboru zkopírována bez úprav a neprovádí se žádná kontrola, lze zahrnout vlastní funkce. Tyto fynkce bývají většinou volány při rozeznání lexému. Je však důležité mít na paměti, že lexém může být rozpoznán dříve, než je zpracován uživatelem. Při rozpoznání se vykonná příslušný kód a lexém se zařadí do fronty. Z té se vyjme a zpracuje v uživatelském programu až později. V příkladě je definována funkce `userFunction`, použitá v předchozí sekci.

```
void userFunction (void)
{
  print (“We have a letter.”);
}
```

7.2 Převod regulárních výrazů do postfixové notace

Cílem je převést regulární výraz na konečný automat, který je s regulárním výrazem ekvivalentní. To znamená, že konečný automat bude přijímat jazyk, jaký je pomocí regulárních výrazů popsán.

Pro snažší zpacování je regulární výraz převáděn do postfixové notace (angl. postfix notation). Při převádění do postfixové notace se k tomuto zápisu přidává symbol konkatenace. Konkatenaci označuje symbol tečky (`.`). Je-li tedy regulárním výrazem sekvence symbolů `ab`, postfixová notace s přidanou konkatenací je ve tvaru `ab.`.

Při vyhodnocení regulárních výrazů se využívá zásobník a následující algoritmus.

- Program přečte znak.
  - Jestliže se jedná o symbol abecedy (operand), uloží ho na zásobník.
  - Jedná-li se o operátor, ze zásobníku vyzvedne potřebný počet operandů a výsledek uloží opět na zásobník.
  - Jestliže se jedná o operátor a na zásobníku není dostatek operandů, program ohlásí chybu.
- Pokud už není co číst a na zásobníku zůstala pouze jedna hodnota, tato hodnata je výsledkem.
- Není-li už co číst a na zásobníku je více hodnot, nastala chyba.

Pro lepší pochopení následuje příklad. Uvažujme tedy výraz $a(b|c)^*$, jehož postfixový zapis je ve tvaru $abc|^*.$. Vyhodnocení je uvedeno v tabulce 7.1.

Tabulka 7.1: Příklad vyhodnocení výrazu $abc|^*$.

Přečtený symbol	Provedená operace	Stav zásobníku	Poznámka (Výrazy jsou v infixovém zápisu.)
a	Uložení na zásobník.	a	
b	Uložení na zásobník.	b, a	
c	Uložení na zásobník.	c, b, a	
$\|$	Operace s 2 operandy.	d, a	Kde $d = b\|c$
*	Operace s 1 operandem.	e, a	Kde $e = d^*$
.	Přidaná konkatenace. Operace s 2 operandy.	*výsledek*	Kde *výsledek* $= a.e$

Pozn. Vrchol zásobníku je vlevo.

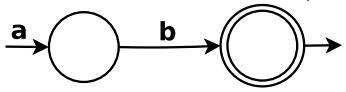
V příkladu lze vidět, že při vyhodnocení je pořadí operací zachováno i bez použití závorek. To je velká výhoda, která zjednoduší další zpracování. Zároveň, když na vstupu už není co číst, na zásobníku zůstal pouze jeden operand a to výsledek.

Při převodu se regulární výraz upravuje tak, aby obsahoval pouze operaci konkatenace (`.`), opakování $0-1$ (`?`), opakování $1-n$ (`+`), opakování $0-n$ (`*`) a alternaci (`|`). Tyto oparace stačí k vytvoření libovolného regulárního výrazu.

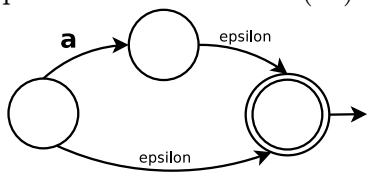
7.3 Převod regulárních výrazů na NKA

V dalším kroku zpracování je regulární výraz v postfixové notaci doplněný o symboly konkatenace převáděn na nedeterministický konečný automat (angl. nondeterministic finite automaton). Pro převod se vytvářejí malé automaty po vzoru [2], ze kterých se pak skládá automat, ekvivalentní regulárnímu výrazu.

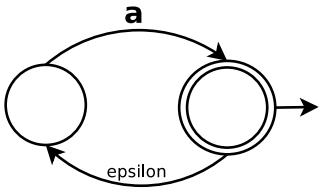
- Vytvoření konkatenace stavu a a b (`ab.`):



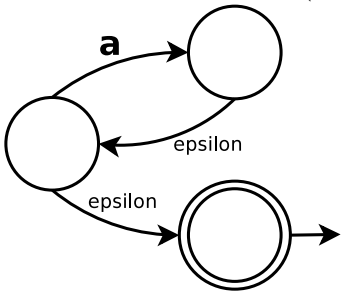
- Vytvoření opakování $0-1$ stavu a (`a?`):



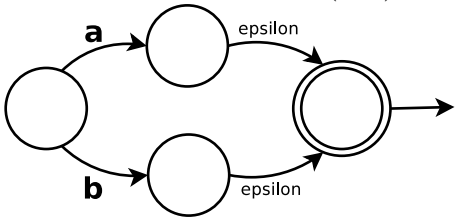
- Vytvoření opakování $1-n$ stavu a (`a+`):



- Vytvoření opakování $0-n$ stavu a (`a*`):



- Vytvoření alternace mezi stavy a a b (`ab|`):



7.4 Převod NKA na DKA

Nevýhodou NKA je, že po přečtení stejného symbolu, může automat přejít do různých stavů, do kterých může přejít i po přečtení prázdného symbolu ε. Matematicky jsou tyto

vlastnosti zcela korektní, ale špatně se algoritmizují. Při převodu na deterministický konečný automat se používá Thomsonův algoritmu. Detaily převodu lze najít například v [5].

Algoritmus převodu je následující:

1. Z počátečního stavu NKA automatu vytvoř ε–uzávěr. Tím vznikne stav DKA, který umísti na zásobník.
2. Vyjmi stav ze zásobníku.
3. Pro každý symbol, se kterým lze z tohoto stavu přejít:
   - zjisti, do kterých stavu NKA lze tímto symbolem přejít,
   - poté z těchto zjištěných stavů vytvoř ε–uzávěr. A nový stav DKA vlož na zásobník.
4. Pokud není zásobník prázdný, jdi na bod 2.

7.5 Vygenerovaný soubor

Při generování souboru lze měnit parametrem počet vláken, která mají zpracovávat soubor. Pokud tato hodnota není parametrem definována, použijí se implicitně dvě vlákna. Dalším parametrem je možno měnit název souboru, do kterého má být vygenerován program. V případě, že název souboru není zadaný, generuje se program implicitně do souboru *scanner.pla.c*. Dalšími parametry se nastavuje velikost bloku v bajtech, který má vlákno analyzovat a velikost front vláken.

Vygenerovaný program očekává při spuštění jeden parametr. Je jím název soubor, nad kterým se má provádět lexikální analýza. Pro použití paralelního lexikálního analyzátoru má uživatel k dispozici tři funkce. Jednou se zinicializují potřebné hodnoty, vypočítají začátky a konce bloků, které mají vlákna zpracovávat a následně vytvoří vlákna. Další funkce simuluje samotný analyzátor. Její návratovou hodnotou je rozpoznaný token. Poslední funkce počká na dokončení činnosti vláken, přečte si jejich návratový kód a regulérně je tím ukončí.

Aby uživatelem definované funkce neměli stejný název jako fuknce, potřebné pro běh analyzátoru, všechny proměnné, funkce a konstanty, viditelné z hlavní funkce `main`, používají prefix `pla_`.

Při zpracování souboru se použila metoda dopředných vláken. Každé vlákno má svou vlastní frontu, do které se řadí rozpoznané lexémy. Dále se využívá ještě výstupní fronty. I tuto frontu obsluhuje samostatné vlákno. Výstupní fronta zajišťuje správné pořadí lexémů, podle toho, jak jsou skutečně zapsány v souboru. Nejprve čte tedy tokeny z fronty prvního vlákna a zařazuje je do své vlastní fronty. Jakmile první vlákno ukončí svou činnost, výstupní vlákno čte z fronty dalšího vlákna. Fronty vláken i výstupní fronta se nedoplňují průběžně, ale pouze tehdy, je-li fronta ze tří čtvrtin prázdná. Pokud je tedy fronta téměř prázdá, začne se znovu naplňovat, přičemž množství tokenů ve frontě se kontroluje pokaždé, když se z fronty token odebere. Protože by bylo časově náročné tokeny, které fronta obsahuje, přesouvat na začátek fronty a nové tokeny zařazovat až za ně, používá se kruhová fronta. Tím je dosaženo správného pořadí tokenů a žádného zpoždění, ke kterému by docházelo při použití klasické fronty a přesouvání tokenů na její začátek.

Uvedený návrh však v případě této bakalářské práce nefungoval podle očekávání, jak je popsáno v kapitole 8.4. Proto konečné úpravy programu skončili u metody předbíhajících

se vláken. Zároveň byla odstraněna výstupní fronta. Tokeny se čtou rovnou z front vláken. O správné pořadí se stará funkce, která nad frontami vláken operuje a uživateli vrací jednotlivé tokeny.

7.5.1 Vlákna

Pro práci s vlákny se používá posixovou knihovnou *pthread* poskytované API[1]. Tato knihovna je nativně vyvinuta pro Unixové operační systémy, a proto se tato aplikace implementuje na operačním systému GNU/Linux. Vzhledem k tomu, že je knihovna pthread rozšířena pouze na unixových systémech, je problém, aplikaci, využívající tuto knihovnu, portovat na operační systém Windows. Existuje však projekt, snažící se tuto knihovnu přepsat i pro prostředí Windows[2]. Více informací je dostupných na domovské adrese projektu [6].

Protože vlákna pracují nezávisle na sobě a jediným sdíleným prostředkem, kam může přistupovat více vláken, jsou jejich fronty, je třeba zajistit výlučný přístup. Pro tento případ výlučného přístupu postačí použití mutexů, které také nabízí tato knihovna.

Kromě toho se po odebrání tokenu z fronty kontroluje také počet tokenů ve frontě. Je nevhodné používat jen smyčku, testující podmínku, protože pak by program zbytečně zatěžoval procesor a spotřebovával tak drahý výpočetní čas. Pro tyto učely se využívá podmínek z knihovny *pthread*. Pokud podmínka není splněna, vlákno přeruší svou činnost a započne ji až po zaslání signálu, kdy opět otestuje podmínku. Příklad použití podmínek a signálu je dostupný například v [1, kapitola Condition Variables].

[1] Application Programming Interface
[2] Knihovna je určena pouze pro 32 bitové verze MS Windows.

Kapitola 8

Implementace

Tato kapitola obsahuje popis některých klíčových částí kódu. Popisuje také problémy, které se vyskytly při implementaci a jejich řešení.

8.1 Nedeterministický konečný automat

Každý stav reprezentuje struktura `TNFAstate` obsahující:

- unikátní identifikátor stavu `id` v rámci všech vytvořených automatů,
- symbol `symbol`, kterým lze provést přechod do tohoto stavu,
- typ stavu `type`, definující, zda se jedná o konečný, normální nebo pomocný stav, který vzniká při vytváření alternace (|). Typy stavů definuje výčet `TypeOfState`,
- ukazatele `out1` a `out2` ukazují na další stav kam lze přejít,
- ukazatel `last` ukazující na konec automatu.

Pokud není některý z ukazatelů `out1` nebo `out2` použit, je nastaven na hodnotu `NULL`. Ukazatel `last` ukazuje při vytvoření stavu sám na sebe, neboť se také jedná o konec automatu (s jedním stavem). Uplatní se, pokud se na konec vytvořeného automatu přidá nový stav. V takovém případě je znám pouze ukazatel na první stav v automatu a pomocí `last` se v tomto stavu odkážeme na poslední stav v automatu. Stavy mezi prvním a posledním stavem v automatu nejsou při tvorbě potřebné, a proto jsou jejich ukazatelé `last` nastaveny na NULL. Jediným možným způsobem, jak se do těchto stavů dostat, je projít celý automat. Při jeho tvorbě se musí měnit typy `type` přislušných stavů. Rozlišuje se mezi typem alternace, konečným a normálním, tj. stav, do kterého lze přejít po přečtení symbolu. Pokud se například vytvoří dva stavy, jsou oba považovány za konečné. Po spojení (konkatenaci) těchto stavů, zůstává poslední stav konečným a první se změní na normální (viz. automaty v kapitole 7.3).

Regulární výraz v postfixové notaci převádí na nedeterministický konečný automat funkce `post2nfa`. Parametrem se předává ukazatel na regulární výraz, který má být převeden na konečný automat. Funkce pak vrací ukazatel na první stav automatu.

Zároveň se u každého regulárního výrazu ukládá číslo konečného stavu. K tomu slouží globální seznam `finStatesNFA`. Jedná se o jednosměrný seznam struktur `TNFAfiniteStates`. Jedna struktura má následující položky:

- unikátní identifikátor stavu `id`,
- offset `fptr` od začátku souboru. Ukazuje na začátek bloku s kódem, který se má vykonat, pokud je daný regulární výraz rozpoznán,
- ukazatel `next`, ukazující na další konečný stav.

8.2 Deterministický konečný automat

Stav deterministického automatu reprezentuje struktura `TDFAState` s následujícmi položkami:

- unikátní identifikátor stavu `id` v rámci automatu,
- symbol `symbols`, se kterým lze to daného stavu přejít,
- seznam stavů nedeterministického konečného automatu `states`, ze kterých byl tento stav vytvořen,
- seznam stavů deterministického konečného automatu `next`, kam lze přejít,
- typ stavu `type`.

Nepoužité položky ve struktuře jsou nastaveny na `NULL`.

Z nedeterministického konečného automatu vytváří deterministický funkce `nfa2dfa`. Během převodu se zaznamenávají všechny znaky, se kterými lze přejít do jiného stavu, do pole `usedSymbols`. Každý typ symbolu je zaznamenán jen jednou. Dále se zaznamenávají i konečné stavy do seznamu `listOfDFS`. Položku v tomto seznamu popisuje zvláštní struktura `TDFAfiniteStates` s položkami:

- unikátní identifikátor stavu `id` v rámci automatu,
- offset `file` od začátku souboru. Ukazuje na začátek bloku s kódem, který se má vykonant, pokud automat ukončí činnost v tomto stavu,
- ukazatel `next` na další položku v seznamu.

8.3 Vygenerovaný soubor

Tabulka popisující konečný automat se jmenuje `pla_finiteAutomat`. Řádky představují jednotlivé stavy, sloupce pak symboly, kterými lze do daného stavu přejít. Po přečtení symbolu se mapuje symbol na sloupec v tabulce `pla_finiteAutomat`. K tomu slouží pole `pla_asciiToColumn` o velikosti 128 symbolů, což představuje spodní část ASCII[1] tabulky. Konečný stav je vyhledán funkcí `pla_idAccept` průchodem polem `pla_finiteState`.

Při manipulaci s tokenem se využívá struktura `pla_TstructToken`, která ho reprezentuje. Ta má následující položky:

- typ tokenu `tok`. Jedná se o hodnotu, s kterou pak v hlavní funkci manipuluje uživatel,

[1] American Standard Code for Information Interchange – tabulka mapuje znaky anglické abecedy na čísla. Existuje i rozšířená ASCII tabulka větší o dalších 128 položek. V ní jsou uloženy národní znaky. Protože nestačí pro všechny národní znaky, existují systémy mapování jako Latin2, Windows–1250, Unicode... Ty mapují některé národní znaky do horní části ascii tabulky.

- atribut tokenu `attr`. Pokud token atribut nemá, je tato hodnota nastavena na `NULL`
- atribut tokenu je ukazatel na řetězec. Položka `maxLen` definuje alokovanou velikost řetězce,
- položka `len` pak počet znaků atributu.

Všechny fronty, se kterými se manipuluje, reprezentuje struktura `pla_TstructQueue` obsahující následující položky:

- fronta tokenu `pla_queue`,
- protože se jedná o kruhovou frontu, `beg` značí její začátek `end` její konec.
- Pro zrychlení některých funkcí a podmíněných výrazů se velikost fronty nepočítá jako rozdíl `end - beg`, ale položka `items` uchovává aktuální počet položek. Tato položka byla přidána dotatečně pro zvýšení efektivity kódu.

Funkce, jejíž návratovou hodnotou je token, se jmenuje `pla_nextToken`. Čte tokeny z výstupní fronty. Pokud je token platný, nastaví jeho atribut globální proměnné `pla_value` a vrátí hodnotu tokenu. Funkce přeskakuje speciální token `PLA_EOB`, označující konec bloku.

K vytvoření vláken slouží funkce `pla_initAnalyzer`. Ta vypočítá začátek a konec bloku pro jednotlivá vlákna, inicializuje jejich globální fronty a mutexy a vlákna vytvoří. Vlákna, analyzující soubor, obsluhuje funkce `pla_threadRoutine`. Funkce `pla_threadOutputRoutine` obsluhuje výstupní frontu.

Funkce `pla_threadRoutine` nejprve zkontroluje, zda má začít generovat tokeny a tím naplňovat frontu `pla_queue[i]` vlákna `i`. Pokud ne, přeruší svou činnost. K přerušení činnosti slouží testování podmínky `pla_queue_cond[i]` a zámek `pla_queue_mutex[i]`, který se zamyká při manipulací s ní. Další testování proběhne po zaslání signálu z funkce `pla_nextToken`. Podmínky, zámek i fronta vlákna jsou definovány polem, kde `i` je pořadové číslo vlákna. Po načtení tokenu funkcí `pla_getNextToken` se token do fronty vkládá až po zamknutí zámku fronty `pla_queue_mutex[i]`.

Podobně se chová i funkce `pla_threadOutputRoutine`. S tím rozdílem, že testuje podmínku `pla_outputQueue_cond` a vlastní frontu `pla_outputQueue` zamyká zámkem `pla_outputQueue_cond_mutex`. Token však nečte ze souboru, ale z fronty vlákna, která musí být při manipulací s ní také zamknuta.

8.4 Testování a problémy při implementaci

Každá tabulka obsahuje název počítače, na kterém byly hodnoty naměřeny. Parametry testovacích počítačů jsou uvedeny v příloze A.

8.4.1 Způsob testování

Při testování rychlosti analýzy souboru, se provedlo měření času běhu programu s jedním a poté se dvěma vlákny. Časy se pak porovnali a teoretické zrychlení, kterého se mohlo dosáhnout, mělo být téměř 50 %.

Hlavní funkce programu při testování vypadala následovně:

```
pla_initAnalyzer (argc, argv);

while (pla_nextToken () != EOF)
{
  pla_printError ();
}

pla_destroyAnalyzer ();
```

Měření času se provedlo pomocí funkce `pla_printTime`. Tato funkce se přidala před a za volání funkce `pla_initAnalyzer` a `pla_destroyAnalyzer`. Podobným způsobem se změřil i čas, potřebný k vykonávání cyklu `while` s funkcí `pla_nextToken`. Předběžné měření odhalilo, že dobu strávenou ve funkcích `pla_initAnalyzer` a `pla_destroyAnalyzer` lze u dalších měření zanedbat (viz. tabulka 8.1).

Tabulka 8.1: Čas strávený v jednotlivých funkcích.

Eva			
Velikost fronty	Počet vláken	Název funkce	Čas strávený ve funkci [ms]
1000	2	initAnal	313
		nextToken	25076
	1	initAnal	203
		nextToken	21711
6	2	initAnal	222
		nextToken	23937
	1	initAnal	163
		nextToken	28980
Merlin			
Velikost fronty	Počet vláken	Název funkce	Čas strávený ve funkci [ms]
1000	2	initAnal	918
		nextToken	17435
	1	initAnal	624
		nextToken	23350
6	2	initAnal	788
		nextToken	24369
	1	initAnal	603
		nextToken	31613

pozn.: Testováno na souboru *test1* o velikosti cca 35 MB (viz. příloha B)

8.4.2 Dopředná vlákna a výstupní fronta

Použití dopředných vláken spolu s výstupní frontou nevedlo k žádnému zrychlení. Průběžné testování se zpočátku provádělo na malých souborech (např. souboru s cca 3600 tokeny). Program se s tímto souborem pustil 1000krát a dále se pracovalo s průměrem z naměřených časů. Výsledky jsou v tabulce 8.2. Nedosáhlo se žádného zrychlení. Protože naměřené hodnoty mohly být ve značné míře ovlivněny plánovačem operačního systému, režií při častém

otevírání a zavírání souboru, provedlo se i několik testů na větším souboru. Ani na souboru o velikosti cca 35 MB se však výsledky nezměnily.

Tabulka 8.2: Testování na malém souboru.

Merlin		
Velikost fronty	Počet vláken	Doba měření [s]
1000	2	25, 4
		21, 2
	1	25, 2
		23, 9
6	2	30, 1
		35, 7
	1	30, 4
		28, 6

Eva		
Velikost fronty	vláken vláken	Doba měření [s]
1000	2	34
		30
	1	33
		32
6	2	68
		74
	1	74
		63

Evna-bagauio		
Velikost frony	Počet vláken	Doba měření [s]
1000	2	48, 9
		53, 4
	1	76, 1
		77.4
6	2	560, 3
		346, 4
	1	350, 8
		313, 8

pozn.: Testováno na souboru s cca 3600 tokenama. Pro jedno měření bylo provedeno 1000 opakování.

V kódu se pak odhalila pomocí nástroje *oprofile* nejpomalejší místa v programu a opravila se. Ani opravení těchto chyb však nevedlo k významně rychlejší analýze při použití dvou vláken.

Po konzultaci se snaha zaměřila na ovlivnění programu výstupní frontou. Výsledky naznačovaly, že by výstupní fronta mohla použití dvou vláken degradovat na klasickou jednovláknovou analýzu. Zrychlení, kterého se tak mohlo dosáhnou, teoreticky záviselo pouze na velikosti fronty druhého vlákna. Ta je totiž po přečtení tokenů z fronty prvního vlákna naplněna. Postupné odebrání všech tokenů, načtených ve frontě, by tak mohlo způsobit částečné zrychlení. Tuto teorii se však nepovedlo ani potvrdit, ani vyvrátit. Nebylo to ani smyslem této práce.

V této době se zároveň změnili i testovací počítače. Na školních serverech *eva* a *merlin* často docházelo k maximálnímu vytížení procesoru, což by mohlo ovlivnit výsledky. Testování se tak přesunulo na neškolní nevytížený server s dvoujádrovým procesorem a téměř nevyužívaný server s čtyřjádrovým procesorem.

8.4.3 Čtení přímo z front vláken

V dalším pokusu se měla potvrdit dobrá implementace vláken a jejich práce s frontou. Z tohoto důvodu se v hlavní funkci programu odstranila z činnosti funkce `pla_nextToken` spolu s cyklem. Zároveň se vypnula obsluha výstupní fronty, protože bez funkce `pla_nextToken` již neměla smysl. Pokud vlákno mělo naplňovat frontu, zamklo ji, vložilo token, odemklo, znovu zamklo, vyjmulo token a odemklo. Zamykání a odemykání front zde zůstalo, aby se při měření alespoň částečně projevila režie práce s frontami. Testování probíhalo už jen na souboru o velikosti cca 35 MB. Tím se předešlo ovlivnění výsledků přílišnou režií operačního systému při použití malých souborů. Popsaná je v kapilole 8.4.2. Z výsledků v tabulce 8.3 se usoudilo, že je práce vláken implementována správně. Tím se i potvrdily předpoklady, že zpomalení pravděpodobně způsobila výstupní fronta. Aby se tato teorie potvrdila, provedlo se ještě jedno měření, popsané v následující kapitole.

Tabulka 8.3: Časy běhů při čtení přímo z fronty.

Evna-bagauio	
Čas běhu s jedním vláknem [s]	Čas běhu s dvěmi vláknami [s]
23, 421	14, 037
23, 251	13, 854
23, 175	13, 844
23, 116	13, 931
23, 160	13, 963
23, 243	13, 825
23, 175	13, 944

Průměrný čas běhu [s]	
23, 201	13, 907

Průměrné zrychlení:	40, 057 %

Callisto	
Čas běhu s jedním vláknem [s]	Čas běhu s dvěmi vláknami [s]
177, 45	103, 38
177, 44	104, 31
176, 76	103, 00
176, 83	103, 09
176, 97	103, 23
177, 00	103, 48
176, 90	103, 91

Průměrný čas běhu [s]	
177, 03	103, 42

Průměrné zrychlení:	41, 581 %

pozn.: Testováno na souboru *test1* o velikosti cca 35 MB (viz. příloha B)

8.4.4 Čtení bez výstupní fronty

Toto měření mělo potvrdit nebo vyvrátit, že zpomalení způsobuje výstupní fronta. Zároveň mělo ukázat, jakým způsobem ovlivňuje rychlost programu zbylá režie. Při čtení z fronty funkcí `pla_nextToken` je totiž třeba frontu uzamknout a až poté odebrat token. Zpomalení tedy mohla způsobovat i snaha vlákna a funkce `pla_nextToken` zamykat stejnou frontu. Naměřné hodnoty jsou v tabulce 8.4.

Na počítači se čtyřmi jádry se dosáhlo téměř poloviního zrychlení. Zrychlení bylo dokonce větší, než při čtení přímo z front vláken. Pravděpodobně je to kvůli zvýšené režii při použití jednoho vlákna. Pokud se čte přímo z fronty vlákna, režie není téměř žádná. Fronta se pouze zamyká a odemyká, ale nečeká se na zámek. Pokud se však funkce `pla_nextToken` snaží z fronty vlákna číst a zároveň se funkce, obsluhující frontu vlákna, do ní snaží vložit token, dojde k malému zpoždění.

Tabulka 8.4: Časy běhů bez výstupní fronty.

Evna-bagauio	
Čas běhu s jedním vláknem [s]	Čas běhu s dvěmi vláknami [s]
34, 605	50, 641
34, 721	51, 110
34, 104	52, 535
34, 484	53, 418
34, 202	52, 933
33, 164	50, 077

Průměrný čas běhu [s]	
34, 378	51, 805

Průměrné zrychlení:	$-50,693\,\%$

Callisto	
Čas běhu s jedním vláknem [s]	Čas běhu s dvěmi vláknami [s]
357, 49	182, 66
417, 48	191, 26
358, 34	171, 36
394, 14	164, 36
378, 51	174, 36
370, 98	181, 25

Průměrný čas běhu [s]	
375, 49	177, 41

Průměrné zrychlení:	$52,753\,\%$

pozn.: Testováno na souboru *test1* o velikosti cca 35 MB (viz. příloha B)

Dále byla pozorována skutečnost, že při spuštění programu s jedním vláknem není využíváno jen jedno jádro. V hlavní funkci `main` se neustále volá funkce `pla_nextToken`, která čte tokeny z fronty vlákna. Zároveň v hlavní funci není vykonáván mezi jednotlivým čtením tokenů žádný kód. Proto je vytížení dalšího jádra přisuzováno právě vykonávání kódu v hlavní funkci programu. Obecně lze tedy říci, že program vytěžuje $n + 1$ jader, kde n je počet vláken.

To je také příčina, že na počítači s dvěma jádry nedošlo k žádnému zrychlení. Jedno vlákno je obsluhováno na jednom jádře, na druhém je obsluhováno druhé vlákno a hlavní funkce.

8.5 Výsledky paralelního zpracování

Ze zkušenostech popsaných v kapitole 8.4.4, se v konečné podobě programu neimplementovala výstupní fronta. Funkce `pla_nextToken` čte tokeny přímo z front vláken. Zároveň byla dopředná vlákna předělána na předbíhající se vlákna. Tím se odstranily problémy, kdy se zpracování s dvěma vlákny degradovalo na jednovláknové zpracování (viz. kapitola 8.4.2). Naměřené hodnoty pro jednojádrový a dvoujádrový počítač shrnuje tabulka 8.5 a pro čtyřjádrový počítač tabulka 8.6. Měření bylo prováděno na třech souborech. Jejich velikost a obsah jsou popsány v příloze B. Při měření s dvěma vlákny se program spustil s každým souborem 10krát, stejně tak i při měření s jedním vláknem. Při počítání průměru se pak největší a nejmenší hodnota z naměřených časů zanedbala a průměr se tak spočítal z osmi hodnot.

Podrobnosti o velikostech jednotlivých souborů a jejich obsahu jsou uvedeny v příloze B. Na počítači *evna-bagauio* se měření provádělo jak na obou jádrech, tak i na jednom. Jak je vidět v tabulce 8.5, při běhu programu za použití pouze jednoho jádra, se dosáhlo tak malého zrychlení, že ho lze zanedbat.

Při běhu na obou jádrech se dosáhlo zrychlení mezi 25 – 30 %. Dle očekávaní se tak

Tabulka 8.5: Čas paralelního zpracování na počítači Evna-bagauio.

Evna-bagauio s oběma jádry			
Počet vláken	Doba běhu pro jednotlivé soubory [s]		
	test1	test2	test3
1	50,172	31,957	40,509
	49,118	32,853	40,306
	50,026	32,874	41,120
	49,479	32,387	40,619
	48,876	31,633	40,607
	49,056	32,791	41,459
	49,546	32,053	40,317
	48,060	31,880	41,399
	49,665	30,245	40,842
	48,764	32,380	40,671
Průměrná doba běhu	49,316	32,242	40,761
Doba běhu (medián $\tilde{x}$)	49,299	32,216	40,645
2	34,920	23,292	29,786
	35,401	24,583	30,279
	33,842	25,073	30,956
	34,012	23,648	30,637
	33,836	24,657	30,359
	34,320	23,881	29,925
	33,757	24,577	30,453
	33,611	23,735	29,730
	35,439	23,531	30,640
	34,023	24,599	29,823
Průměrná doba běhu	34,264	24,151	30,238
Doba běhu (medián $\tilde{x}$)	34,017	24,229	30,319
Zrychlení [%]	30,522	25,093	25,816
Zrychlení (z $\tilde{x}$)[%]	30,999	24,792	25,405

Evna-bagauio s jedním jádrem			
Počet vláken	Doba běhu pro jednotlivé soubory [s]		
	test1	test2	test3
1	93,509	69,352	86,693
	95,230	70,548	85,020
	93,100	68,660	84,787
	94,518	68,782	85,659
	96,084	68,916	85,057
	94,068	68,788	85,018
	93,860	69,764	86,441
	94,276	68,533	86,949
	94,253	69,731	85,266
	93,379	68,793	84,825
Průměrná doba běhu	94,137	69,098	85,497
Doba běhu (medián $\tilde{x}$)	94,160	68,855	85,162
2	92,351	67,912	84,808
	93,407	66,670	84,353
	95,400	66,783	85,105
	92,587	68,113	83,565
	92,942	68,553	84,225
	93,944	66,453	85,862
	92,637	66,751	84,463
	93,043	67,456	83,739
	92,694	67,085	84,648
	94,574	68,281	84,277
Průměrná doba běhu	93,228	67,381	84,452
Doba běhu (medián $\tilde{x}$)	92,993	67,270	84,408
Zrychlení [%]	0,965	2,485	1,222
Zrychlení (z $\tilde{x}$)[%]	1,239	2,302	0,885

pozn.: Blok, který zpracovávají jednotlivá vlákna, měl velikost 10 000 B.

nedosáhlo zrychlení, jaké bylo naměřeno při čtení přímo z front (viz. kapitola 8.4.3). Je nutné brát v úvahu, že při běhu s dvěma vlákny je v ideálním případě jedno vlákno obsluhováno na jednom jádře. O druhé jádro se dělí druhé vlákno spolu s obsluhou hlavního programu (tzn. funkce `main`). Naopak při této implementaci, kdy je nutné brát v úvahu i režii ve funkci `pla_nextToken`, se paralelní zpracování nedegradovalo na klasické sériové zpracování, jako v případě střídavého čtení z front vláken, popsaného v kapitole 8.4.4.

Na počítači *callisto* je zrychlení oproti výsledkům z počítače *evna-bagauio*, znatelně menší, i přes to, že disponuje čtyřmi jádry (viz. tabulka 8.6). To může mít několik příčin. Jednou je přítomnost staršího jádra operačního systému. V kombinaci režie ve funkci `pla_nextToken` spolu se starší implementací vláken v jádře operačního systému, se může program zpomalit. Dále se zjistilo, že tento počítač ve skutečnosti disponuje pouze dvěma jádry. Zbylá dvě jádra jsou simulována pomocí technologie *hyper–threading*. To mohlo mít vliv i na rychlost testů. Časy totiž oproti měření testů na počítači *evna-bagauio*, rapidně vzrostly. Nebyla však provedena žádná další měření, která by potvrdila některou z domněnek. Je to také z toho důvodu, že nebylo možné na počítači *callisto* zkompilovat nové jádro nebo nainstalovat profilovací program *oprofile*. Pomocí něho by bylo možné zjistit nejpomalejší části programu na tomto počítači.

Během měření se nadále zjistilo, že velikost bloku, který má vlákno zpracovávat, má veliký význam na výsledné zrychlení. Na všech počítačích byla nastavena vždy stejná velikost bloku, aby bylo možné výsledky porovnávat.

Tabulka 8.6: Čas paralelního zpracování na počítači Callisto.

Callisto			
Počet vláken	Doba běhu pro jednotlivé soubory [s]		
	test1	test2	test3
1	332, 67	231, 60	283, 52
	305, 46	232, 52	281, 10
	338, 06	233, 65	273, 76
	343, 94	233, 14	276, 85
	344, 27	226, 60	268, 21
	333, 73	237, 39	300, 20
	320, 71	239, 29	304, 03
	309, 29	239, 21	277, 25
	307, 05	235, 01	300, 38
	322, 15	236, 88	284, 73
Průměrná doba běhu	325, 95	234, 93	284, 72
Doba běhu (medián $\tilde{x}$)	327, 94	234, 33	282, 31
2	263, 26	199, 96	238, 39
	263, 99	199, 81	234, 84
	262, 19	200, 13	234, 97
	266, 58	200, 57	237, 49
	267, 06	200, 64	238, 04
	263, 97	199, 85	234, 43
	264, 95	201, 14	235, 99
	265, 11	201, 06	236, 43
	263, 72	200, 87	235, 92
	266, 02	200, 27	234, 87
Průměrná doba běhu	264, 70	200, 42	236, 07
Doba běhu (medián $\tilde{x}$)	264, 47	200, 42	235, 95

Zrychlení [%]	18, 79	14, 69	17, 09
Zrychlení (z $\tilde{x}$)[%]	19, 35	14, 47	16, 42

pozn.: Blok, který zpracovávají jednotlivá vlákna, měl velikost 10 000 B.

Kapitola 9

Závěr

Při implementaci programu se vyskytlo několik chyb. Ty byly postupně testovány a odstraňovány, dokud se nedosáhlo alespoň částečného zrychlení analýzy. Nejpomalejší části programu byly odladěny programem *oprofile*, což také přispělo ke zrychlení při paralelním zpracování souboru. Ani to však nevedlo k výraznému zrychlení analýzy. Nakonec se přišlo na to, že analýzu zpomaluje výstupní fronta a velké bloky na zpracování. Každé ze dvou vláken totiž zpracovávalo polovinu souboru. Druhé vlákno tak naplnilo svou frontu tokenama a čekalo na ukončení činnosti prvního vlákna a uvolnění místa ve frontě, aby ji mohla zaplňovat dalšími tokeny. Z těchto důvodů se odstranila výstupní fronta a dopředná vlákna se předělala tak, aby se předbíhala, čímž se zmenšila velikost zpracovávaného bloku.

Pro měření byly vytvořeny tři soubory (viz. příloha B). Program byl spuštěn a změřen s každým souborem 10krát za použití dvou vláken a 10krát za použití jednoho vlákna. U naměřených hodnot se pak zanedbala největší a nejmenší hodnota a ze zbylých osmi hodnot se spočítal průměrný čas běhu. Ten byl použitý při výpočtu procentuálního zrychlení. Na dvoujádrovém počítači *evna–babagauio* (viz. příloha A) se tak dosáhlo až 30 % zrychlení. Když bylo jedno jádro vypnuto, a zůstalo tak aktivní pouze jedno, naměřené časy byly podobné. Nedosáhlo se tedy žádného zrychlení. Na počítači se čtyřmi jádry (*callisto*) se dosáhlo zrychlení pouze cca 18 %. Jednalo se však o starší počítač s dvěma jádry. Další dvě jsou simulována pomocí technologie hyper–threading, což mohlo způsobit menší zrychlení než u dvoujádrového počítače.

Výsledky nasvědčují tomu, že i v případě serializace výstupu tokenů, má smysl implementovat lexikální analyzátor paralelně. Mnohem lepší výsledky by mohly být v případě, kdy by se paralelizovali i další fáze analýzy. Při čtení přímo z front vláken, je totiž možné dosáhnout zrychlení i více než 40 %. Pravděpodobně by však při paralelizaci ostatních fází analýzy vstupního souboru došlo k dalšímu zpomalení. Způsobovala by to potřebná režie při práci s vlákny a jejich sdílenými prostředky. Paralelizace zbylých fází tak otevírá spoustu nových otázek.

Literatura

[1] Blaise Barney: POSIX Threads Programming [online]. https://computing.llnl.gov/tutorials/pthreads/, 2010-01-13 [cit. 2010-04-27].

[2] Cox, R.: Regular Expression Matching Can Be Simple And Fast [online]. http://swtch.com/~rsc/regexp/regexp1.html, 2007 [cit. 2010].

[3] Flow Simulation Ltd.: Jan Lukasiewicz (1878 - 1956) [online]. http://www.calculator.org/Lukasiewicz.aspx, 2009 [cit. 2010].

[4] Friedl, J. E. F.: *Mastering regular expressions*. O'Reilly, 2006, ISBN 0-596-52812-4, 515 s.

[5] Gerzic, A.: Writing own regular expression parser [online]. http://www.codeproject.com/KB/recipes/OwnRegExpressionsParser.aspx, 2003 [cit. 2010].

[6] Johnson, R.: Open Source POSIX Threads for Win32 [online]. http://sourceware.org/pthreads-win32/, 2006 [cit. 2010].

[7] Meduna, A.: *Automata and Languages: theory and applications*. Springer, 2000, ISBN 81-8128-333-3, 916 s.

[8] Meduna, A.: *Elements of compiler design*. Auerbach Publications, 2008, ISBN 1-4200-6323-5, 286 s.

[9] Tanenbaum, A. S.: *Modern operating systems*. Alan Apt, druhé vydání, 2001, ISBN 0-13-031358-0.

[10] The Flex Project: Flex manual [online]. http://flex.sourceforge.net/manual/, 2008 [cit. 2010].

Příloha A

Parametry testovacích počítačů

Eva

- Procesor: Intel® Xeon® CPU 5160 @ 3 GHz
- Počet jader: 4
- Architektura: i386
- Operační paměť: 4 GB
- Operační systém: FreeBSD 8.0-STABLE
- Verze překladače gcc: 4.2.1 20070719 [FreeBSD]

Merlin

- Procesor: Quad-Core AMD Opteron^{TM} Processor 2387 2,8 GHz
- Počet procesorů: 2
- Celkem jader: 8
- Architektura: x86-64
- Operační paměť: 16 GB
- Operační systém (jádro): GNU/Linux (2.6.32.11)
- Verze překladače gcc: 4.3.4

Callisto

- Procesor: Intel® XEON^{TM} CPU 1,8 GHz
- Počet jader: 4
- Architektura: i386

- Operační paměť: 4 GB
- Operační systém (jádro): GNU/Linux (2.6.11.11)
- Verze překladače gcc: 3.4.4 20050721 (Red Hat 3.4.4-2)

Evna–bagauio

- Procesor: Intel® Core™2 Duo CPU T7500 @ 2,2 GHz
- Počet jader: 2
- Architektura: x86-64
- Operační paměť: 2 GB
- Operační systém (jádro): Ubuntu GNU/Linux (2.6.28-11-generic)
- Verze překladače gcc: 4.3.3 (Ubuntu 4.3.3-5ubuntu4)

Příloha B

Obsah CD

- `bin` ..... Přeložený lexikální nalyzátor s testovacím souborem.
- `doc` ..... Zdrojové kódy technické zprávy.
- `doc-src` ..... Programová dokumentace.
  - `pla` ..... Dokumentace ke generátoru paralelního lexikálního analyzátoru.
    - `html` ..... Hypertextová dokumentace.
    - `latex` ..... Dokumentace v latexu i se zdrojovými kódy.
  - `pla_scanner` ..... Dokumentace k paralelnímu lexikálnímu analyzátoru.
    - `html` ..... Hypertextová dokumentace.
    - `latex` ..... Dokumentace v latexu i se zdrojovými kódy.
  - `README, Makefile`
  - `doxygen-pla.conf` ... Konfigurační soubor pro *doxygen*. Dokumentace ke generátoru lexikálního analyzátoru.
  - `doxygen-pla_scanner.conf` ... Konfigurační soubor pro *doxygen*. Dokumentace k lexikálnímu analyzátoru.
- `src` ..... Zdrojové soubory generátoru paralelního lexikálního analyzátoru.
  - `dest` . Vytvoří se při spuštění `./pla` a obsahuje vygenerovaný lexikální analyzátor.
- `tests` ..... Obsahuje soubory pro testování.
  - `dest1` ..... Přeložený lexikální analyzátor popsaný v souboru `input1.pla`.
  - `dest2` ..... Přeložený lexikální analyzátor popsaný v souboru `input2.pla`.
  - `dest3` ..... Přeložený lexikální analyzátor popsaný v souboru `input3.pla`.
  - `input1.pla, input2.pla, input3.pla` ..... Popisují tři různé regulární jazyky.
  - `test1, test2, test3` ..... Testovací soubory s rozdílnými jazyky.
  - `README, Makefile`
- `README, Makefile`

Soubor `test1` o velikosti 35, 53 MB přijímá jazyk obsahující čísla, slova, jednopísmenné předložky a bíle znaky. Měl by tak být dostačující pro běžný psaný text. Soubor `test2` o velikosti 29 MB přijímá podmnožinu jazyka C. Přijímá pouze konstrukce, jaké jsou použité ve zdrojovém souboru paraleního lexikálního analyzátoru `pla_scanner.pla.c`. Soubor `test3` o velikosti 49, 45 MB přijímá jazyk sloužící k analýze manuálové stránky funkce `pthread_cond_signal`. Kromě běžných slov a bílých znaků obsahuje i některá klíčová slova používané v manuálovách stránkách (např. `DESCRIPTION`, `SYNOPSIS` ...). Obsah jednotlivých souborů byl několikrát zkopírován tak, aby se dosáhlo požadované velikosti souboru.

Podrobnější informace o parametrech programů přiložených na CD, jejich příklady spuštění se vzorovými soubory a popis práce s `Makefile` jsou popsané v příslušných adresářích v souboru `README`.