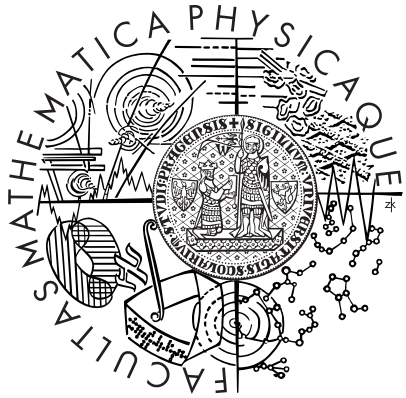
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Ondřej Černý

Nástroj na generování programů pro čtení textových vstupů.

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. David Kolovratník
Studijní program: Informatika, Programování

2008

Poděkování patří Mgr. Davidu Kolovratníkovi za vedení práce a cenné podněty při návrhu nástroje. Dále chci poděkovat svým rodičům za trpělivou podporu při studiu.

Prohlašuji, že jsem svou bakalářskou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 30.5.2008 Ondřej Černý

Obsah

Název práce: Nástroj na generování programů pro čtení textových vstupů
Autor: Ondřej Černý
Katedra (ústav): Ústav formální a aplikované lingvistiky
Vedoucí bakalářské práce: Mgr. David Kolovratník
e-mail vedoucího: David.Kolovratnik@mff.cuni.cz

Abstrakt: V předložené práci se zabýváme návrhem a implementací nástroje AutoComp na generování programů pro zpracování strukturovaných texotvých dat. Vstupem nástroje je popis textových dat vyžívající regulárních výrazů a stavových strojů. Výstupem je zdrojový kód v programovacím jazyce, který poskytuje rozhraní pro začlenění do programu. Ve vstupním popisu je možno definovat body volání specifikovaných funkcí a návrat řízení volajícímu kódu. Generované programy mohou sloužit pro tokenizaci textu nebo pro parsování jednodušších gramatik. Nástroj podporuje dva programovací jazyky a několik druhů kódování znaků, návrh byl zaměřen na rozšiřitelnost na další jazyky a kódování.

Klíčová slova: konečný automat, regulární výraz, lexkální analýza, generátor

Title: A tool for generating code supporting reading of text files
Author: Ondřej Černý
Department: Institue of Formal and Aplied Linguistics
Supervisor: Mgr. David Kolovratník
Supervisor's e-mail address: David.Kolovratnik@mff.cuni.cz

Abstract: In the present work we deal with the concept and implementation of tool AutoComp for generating programs for processing structured text files. The tool accepts description of the text whith making use of regular expressions and state machines. The output of the tool is a code in a programming language which gives interface for integrating code to a final program. It is possible to define points for calling specified functions and returning controll to the caller program in the input description. The use of generated code is to tokenize text or parse elementary gramatics. The tool provides two target programming languages and few character encodings. The concept was focused on extensibility to more languages and character encodings.

Keywords: finite automaton, regular expression, lexical analysis, lexer, scanner generator

Kapitola 1

Úvod

Načítání strukturovaných dat v textové formě je součástí velké skupiny počítačových programů. Práce se snaží analyzovat tuto problematiku a vyvynout nástroj, který na základě popisu struktury textových dat vygeneruje část programu pro zpracování dat. Vygenerovaný kód v zadaném programovacím jazyce poskytne rozhraní pro načtení a zpracování textu. Programátor pak začlení vygenerovaný kód do své aplikace aniž by musel sám psát funkce pro načtení textových dat. Vygnerovaný program může například posloužit jako součást překladače, zejména pro lexikální analýzu. Pro velmi jednoduché jazyky je sám dobře použitelný pro celé parsování dat.

1.1 Popis nástroje

Součástí bakalářské práce je program AutoComp (Automaton Compiler - překladač automatů), který na vstupní zdrojový kód přeloží do zdrojového kódu ve specifikovaném programovcím jazyce. Vstupní zdrojový kód, nazvěme ho jen zdrojový kód, se skládá z deklarací stavů, přechodových hran, popisu vstupního zařízení a dalších specifikací výstupního zdrojového kódu. Soustava stavů a přechodových hran tvoří systém podobný konečnému automatu, odtud pochází název nástroje. Výstupní zdrojový kód je tvořen sadou funkcí či tříd s metodami, které slouží k načítání textu ze vstupního zařízení a jeho dělení na řetězce, lexikální elementy. Zejména je podstatná hlavní funkce (metoda) *RUN*[1], která zastřešuje načítání a předzpracování vstupního textu, je volána z programového celku, do něhož je výstupní kód začleněn uživatelem. Hlavní funkce v určitých fázích výpočtu volá funkce programového celku, jejichž návratové hodnoty determinují další průběh výpočtu. V určitých bodech také ukončuje svůj běh s návratovou hodnotou a tím předává řízení programovému celku. Sémantiku načítání řetězců ze vstupního zařízení, stějně jako body volání funkcí programového celku a ukončení běhu definuje zdrojový kód, zejména stavy, přechodové hrany a jejich vlastnosti. Výstupní kód budeme označovat jako generovaný program, programový celek pak budeme nazývat aplikací. Až aplikace se stává samostatně spustitelným programem.

[1] symbolické jméno

Nástroj AutoComp podporuje programovací jazyky C (Standart C) a Perl a je rozšiřitelný na podporu dalších procedurálních programovacích jazyků. Dále podporuje různá kódování znaků zejména ASCII, UCS2, UCS4 a UTF-8 a je též rozšiřitelný o podporu příbuzných kódování, například ISO-8859-X. Nástroj AutoComp je napsán v jazyce C, generované zdrojové kódy i zdrojové kódy AutoCompu jsou přenositelné na platformy LINUX/UNIX, OSX a Windows (testováno na architekturách i686, Darwin, Solaris a Win32).

1.2 Definice základních pojmů

V následujícím textu se budeme často setkávat s pojmem regulární výraz a regulární jazyk.

Řekněme, že Σ je konečná abeceda, tzn. konečná neprázdná množina symbolů. Nechť prázdný symbol $\lambda \in \Sigma$. **Regulární jazyk**[2] nad abecedou Σ je definována rekurzivně body (1) až (5).

(1) $\emptyset$ je regulární jazyk nad Σ.

(2) $\{\lambda\}$ je regulární jazyk nad Σ.

(3) $\{a\}$ je regulární jazyk nad Σ, pokud $a \in \Sigma$.

(4) Jestliže P a Q jsou regulární jazyky nad Σ, pak

$P \bigcup Q$ je regulární jazyk nad Σ.

PQ (zřetězení) je regulární jazyk nad Σ.

P^* (libovolná iterace) je regulární jazyk nad Σ.

(5) Nic jiného není regulárním jazyk nad Σ.

Výsledkem operace zřetězení je množina obsahující právě slova vzniklá postavením nějakého prvku z druhé množiny za nějaký prvek první množiny. Výsledkem libovolné iterace je množina slov obsahující slova vzniklá umístěním libovolného počtu (včetně 0) nějakých slov z původní množiny.

Regulární výraz nad abecedou Σ a regulární jazyk, který označuje, jsou definovány následovně:

(1) $\emptyset$ je regulární výraz označující regulární jazyk $\emptyset$.

(2) λ je regulární výraz označující regulární jazyk $\{\lambda\}$.

(3) $a \in \Sigma$ je regulární výraz označující regulární jazyk $\{a\}$.

[2] Běžnější je definice všech regulráních jazyků nad abecedou Σ jako nejmenší množiny obsahující prázdný jazyk a všechny elementární jazyky nad Σ, která je uzavřené na operace zřetězení, $\bigcup$ a *. V [4] je dokázána ekvivalence mezi těmito dvěma strukturami.

(4) Nechť p a q jsou regulární výrazy označující regulární jazyky P a Q. Pak

$p|q$ je regulární výraz a označuje regulární jazyk $P \bigcup Q$.

pq je regulární výraz a označuje regulární jazyk PQ.

p^* je regulární výraz a označuje regulární jazyk P^*.

(p) je regulární výraz označující regulární jazyk P.

(5) Nic jiného není regulárním výrazem nad Σ.

V praxi se používají ještě následující zkrácené zápisy některých operací.

p^+ je zkratka za pp^*.

$p?$ je zkratka za $p|\lambda$.

$p\{n, m\}$ je zkratka za $ppp...p|ppp...pp|...|ppp...pp...p$, kde se v jednotlivých operandech operce | výraz p opakuje n až m krát.

$p\{n\}$ je zkratku za $p\{n, n\}$, tedy $ppp...p$, kde se p vyskytuje právě n-krát.

$p\{n, \}$ je zkratka za $p\{n, n\}p^*$.

Takto definované regulární výrazy ukazují jen jeden z více možných zápisů regulárních výrazů. V podstatě odpovídá syntaxi použité v programu AutoComp, kde je možné zadávat přechody mezi stavy pomocí regulárních výrazů. Kód programu podporuje uvedení rozsahu či výčtu znaků což je zkratka za operaci |, naopak nedovoluje zapsat výraz λ.

S pojmem regulárních jazyků úzce souvisí pojem konečný automat. **Nedeterministický konečný automat** je 5-tice $(Q, S, \delta, \Sigma, F)$, kde

Q je konečná neprázdná množina stavů.

Σ je abeceda, konečná neprázdná množina vstupních symbolů.

$\delta \subseteq (Q \times \Sigma \times Q)$ je množina přechodů. Někdy se také používá $\delta : Q \times \Sigma \to \mathcal{P}^Q$ jako přechodová funkce.

$S \subseteq Q$ je množina startovních (počátečních) stavů.

$F \subseteq Q$ je množina koncových (přijímacích) stavů.

Nedeterministický konečný automat pracuje tak, že provádí přechody mezi množinami stavů na základě vstupních symbolů. Vše, co potřebujeme znát k určení dalšího výpočtu, je množina stavů, ve které se výpočet nachází a řetězec symbolů na vstupu (vstupní slovo, vstupní páska). Krokem výpočtu je přechod z množiny stavů P do množiny stavů R, provede se, pokud řetězec na vstupu začíná symbolem v a $(\forall p, r \in Q) : (\exists (p, v, r) \in \delta) \iff$

$p \in P \wedge r \in R$). Při provádění kroku se posune vstupní páska o jeden symbol doleva, to znamená, že vstupní řetězec bude bez prvního symbolu. Výpočet automatu končí, pokud se dostal na konec vstupní pásky. Výpočet začíná z množiny startovních stavů. Říkáme, že přijímá vstupní slovo, pokud se na konci nachází v množině stavů, která obsahuje alespoň jeden koncový stav. Dále říkáme, že automat přijímá jazyk L pokud přijímá právě slova jazyka L.

Deterministický konečný automat je speciálním případem nedeterministického konečného automatu, pro který platí $|S| = 1$ a $(\forall p \in Q, v \in \Sigma) : (|\{(p, v, q) \in \delta : q \in Q\}| = 1)$. Je dokázána ekvivalence mezi nedeterministickými a deterministickými automaty v tom smyslu, že pro každý nedeterministický automat existuje deterministický který přijímá stejný jazyk a naopak. Dále je dokázáno, že konečné automaty přijímají právě regulární jazyky, tedy pro každý regulární jazyk lze sestrojit přijímající konečný automat a naopak. Oba tyto důkazy jsou konstruktivní, tzn. přinášejí algoritmus pro sestavení odpovídajících automatů či jazyků.

Mooreův stroj je 5-tice $(Q, \Sigma, Y, \delta, \mu)$, kde

Q je konečná neprázdná množina stavů.

Σ je abeceda, konečná neprázdná množina vstupních symbolů.

Y je množina výstupních symbolů.

$\delta \subseteq (Q \times \Sigma \times Q)$ je množina přechodů. Někdy se také používá $\delta : Q \times \Sigma \to \mathcal{P}^Q$ jako přechodová funkce.

$\mu : Q \to Y$ je značkovací funkce.

$S \subseteq Q$ je množina startovních (počátečních) stavů.

Princip výpočtu je stejný jako u konečných automatů s tím rozdílem, že značkovací funkce μ „vysílá“ v každém kroku výstupní symboly cílového stavu (na začátku výpočtu vyšle symboly stavů startovních). Chybějící koncové stavy může suplovat právě značkovací funkce vyčleněním některých výstupních symbolů jako indikátorů koncových stavů. Deterministický Mooreův stroj je opět speciální případ analogicky s deterministickým konečným automatem. Opět je znám způsob konstrukce ekvivalentního deterministického Mooreova stroje z nedeterministického Mooreova, musíme se ovšem spokojit s tím, že množina výstupních symbolů deterministického stroje bude tvořena podmnožinami výstupních symbolů nedeterministického stroje. Pokud řekneme, že vyslání symbolu indikujícího koncový stav na konci vstupní pásky znaméná přijetí slova strojem, je přímo vidět, že Mooreovy stroje jsou alespoň stejně silné jako konečné automaty. Ve skutečnosti jsou ekvivaletní.

Poznámky ke zdrojům

Definice byly převzaty z knihy [1], která obsahuje souhrn definic a výsledků teorie automatů a gramatik s ohledem na aplikaci v oblasti parsování. Tato kniha převypravuje definice a

výsledky z původních zdrojů. Regulární výrazy byly definovány v [5], důkaz ekvivalence nedeterministických a nedeterministických konečných automatů byl předveden v [6] a důkaz ekvivalence mezi regulárními jazyky, množinami a jazyky přijímanými konečnými automaty uveden v [4]. Mooreovy stroje byly uvedeny v [8].

1.3 Přehled kapitol

V kapitole 2 je proveden stručný rozbor problému, zařazení nástroje AutoComp do kategorie počítačových programů a porovnání s jinými existujícími implementacemi.

Kapitola 3 je věnována popisu nástroje AutoComp z uživatelského hlediska, je zde provedena abstrakce vstupního zdrojového kódu. Konkrétní popis syntaxe uvádí Příloha B.

Kapitola 4 se věnuje programátorskému pohledu a implementaci nástroje AutoComp.

V kapitole 5 je provedeno zhodnocení průběhu vývoje AutoCompu a návrhy možných rozšíření.

Příloha A popisuje obsah přiloženého CD a Příloha B obsahuje uživatelský manuál.

Kapitola 2

Analýza problému a existujících implementací

Nejprve shrneme podrobnější požadavky na vyvíjený nástroj. Vstupem nástroje je zdrojový kód, který popisuje strukturu podobnou konečnému automatu, říkejme ji krátce stroj (stroj M). Stroj umožňuje specifikovat přechody mezi stavy pomocí regulárních výrazů a určit akci, která se má provést před a při přechodu. Výsledek akce může zakázat daný přechod. Nástroj musí na základě stroje generovat zdrojový kód ve specifikovaném programovacím jazyce, který bude načítat data z nějakého vstupního zařízení a provádět odpovídající akce při přechodech. Je možno říci, že má simulovat zadaný stroj. Zmíněné akce na přechodových hranách omezíme na volání funkcí v daném programovacím jazyce, specifikace akce je tedy provedena jménem funkce. Abychom v textu odlišily tyto speciální funkce, budeme je označovat *procedury*, blíže *tázací procedura* bude představovat akci před přechodem a *přechodová procedura* akci při přechodu.

Nástroj se musí určitým způsobem vyrovnat s nedeterminismem zadaného stroje na úrovni regulárních výrazů, musí počítat s podporou více programovacích jazyků, vstupních zařízení a různých kódování znaků. Dále musí podporovat ladění výsledného programu ve formě výpisů a speciálních procedur. Podpora ladění musí být vypínatelná, tzn. generovaný kód se liší v ladícím a „ostrém“ režimu. Generovaný zdrojový kód bude „ušit na míru“ zadání, tzn. bude se specializovat na čtení ze zadaného vstupního zařízení a dle zadaného stroje.

2.1 Lexikální analýza

Lexikální analýza je převod sekvence znaků na sekvenci logických položek zvaných ***tokenů***[1]. Někdy se také používá označení *tokenizace*. Tokenem může být jakýkoliv symbol nebo objekt. Token je vnímán jako jakási kategorie řetězců. Pro specifikaci řetězců, které budou převedy na token, se nejčastěji používá regulárních výrazů. V souvislosti s lexikální analýzou se konkrétní řetězec znaků, který odpovídá nějakému tokenu nazývá **lexém**. Budeme-li v našem případě považovat proceduru volanou při přechodu mezi stavy zadaného stroje za

token, bude vygenerovaný zdrojový kód plnit úlohu lexikální analýzy. Procesům provádějícím převod znaků na tokeny se říká lexikální analyzátory nebo scannery.

Lexikální analyzátor vygenerovaný programem AutoComp bude provádět tzv. přímou lexikální analýzu, bude číst sekvenci znaků od začátku do konce a vyhodnocovat, jestli načtený řetězec netvoří token. Pokud řetězec určuje token, zavolá odpovídající přechodovou proceduru a předá lexém jako argument. Odmítnutí přechodu procedurou zapříčiní hledání dalšího řetězce, který bude určovat token, tedy odpovídat regulárnímu výrazu. Přechod do jiného stavu znamená určitou změnu kontextu, v novém stavu se mohou hledat jiné tokeny. Zapojení přechodových hran do stroje tedy do jisté míry určuje vzájemné pořadí tokenů, v jakých se mohou vyskytovat. Tento aspekt již spíše spadá do další oblasti teorie překladu, do tzv. syntaktické analýzy. Dá se tedy říci, že generovaný program provádí lexikální analýzu posilněnou o kontrolu jistých syntaktických závislostí.

V klasické přímé lexikální analýze založené na specifikaci tokenů pomocí regulárních výrazů se scanner modeluje pomocí paralelně běžících konečných automatů. Paralelismus se odstraňuje pomocí Mooreova stroje, který simuluje paralelní běh automatů a na koncových stavech obsahuje informaci o tom, kterým automatům odpovídá načtený lexém. V anglicky psané literatuře se pro takový stroj používá označení „transducer“. V našem analyzátoru odpovídá jeden klasický analyzátor jednomu stavu zadaného stroje. Přechodové hrany jsou napojeny na koncové stavy Mooreových strojů a vedou do dalších stavů zadaného stroje.

Implementace Mooreových strojů je v generovaném kódu řešena pomocí tzv. tabulek, kdy jsou stavy a přechodová funkce stroje uloženy do datové struktury programu a funkce programu simulují výpočet stroje procházením datové struktury. Alternativou k tomuto přístupu je přímá implementace stroje pomocí programovacího jazyka konstrukcemi `goto`, `switch` či zanořenými `if-then-else`. Druhá varianta je považována za výkonnější, ovšem vzhledem k požadované podpoře a rozšiřitelnosti na více programovacích jazyků byla zvolena první možnost. Datová struktura tabulek bude pro všechny jazyky podobná, dá se tedy relativně snadno zajistit rozšiřitelnost na další jazyky. Naopak druhá varianta by činila tento úkol velmi složitým i generování kódu by bylo značně obtížné. Zůstává také otázkou, zda výhoda výkonosti přetrvá i při omezení vyjadřovacích možností programovacího jazyka na nějakou společnou podmnožinu. Některé programovací jazyky konstrukci `goto` již vůbec nepodporují.

Mezi nejznámější programy generující lexikální analyzátory patří např. **Flex**[1] nebo **Re2c**[2].

Flex je legendární nástroj svého druhu, vznikl kolem roku 1987 přepisem staršího **Lex**u[10] do jazyka C. Flex na základě zdrojového kódu, skládajícího se z regulárních výrazů a odpovídajících fragmentů kódu v jazyce C/C++ generuje kód analyzátoru. Nabízí řadu možností, z nichž zmiňme podporu přepínání kontextu[3] a odmítnutí regulárního výrazu, což je podobné stavům stroje a odmítnutí přechodu. Flex používá pro imlementaci konečných automatů tabulky, tedy stejný přístup jaký byl zvolen v AutoCompu. Nevýho-

[1] Domovská stránka: http://flex.sourceforge.net

[2] Domovská stránka: http://re2c.org

[3] Ve smyslu přepnutí transduceru.

dyou flexu je, že podporuje pouze základní znakovou sadu ASCII. Kód generovaný flexem často produkuje nečekaná varování při konečném překladu a používá Unixové API pro práci se soubory, čímž se stává nepřenositelným.

Program re2c funguje spíše jako preporcesor jazyka C/C++, nahrazuje specifické direktivy kódem generovaným na základě regulárních výrazů. Používá přímou implementaci konečných automatů, produkuje rychlejší programy. Program rovněž podporuje přepínání „kontextu“, neumožňuje však odmítnutí regulárního výrazu. Nevýhodou je opět omezení na základní znakovou sadu.

Obě popsaná řešení jsou špičkovými projekty s dlouholetým vývojem, jejich užitečnost je prokázána spoustou aplikací. Specializace na jazyk C/C++ jim dává jistou výhodu při implementaci různých variant a voleb, které je obtížné nabídnout za předpokladu podpory více jazyků. Na druhou stranu poskytují pouze hladové vyhodnocování regulárních výrazů, tzn. že je s regulárním výrazem asociován co nejdelší řetězec. Nástroj AutoComp umožňuje specifikovat každému regulárnímu výrazu, jestli se má chovat líně nebo hladově. Podpora líných regulárních výrazů může usnadnit návrh struktury zadaného stroje a formulaci regulárních výrazů.

2.2 Kódování znaků a znakové sady

Problém kódování znaků zahrnuje jak reprezentaci čísla znaku v podobě bytů tak přiřazení čísel jednotlivým znakům. Program AutoComp kromě základní sady ASCII podporuje znakovou sadu UCS (Univesal Character Set)[15] a její kódování UCS-2, UCS-4 a UTF-8. Znaková sada UCS pokrývá všechny známé národní abecedy a může dosáhnout velikosti až 2^{31}. Alternativou by mohla být sada Unicode, je do jisté míry kompatibilní s UCS, pro náš účel jsou obě sady téměř totožné. Unicode navíc definuje ekvivalenci mezi různými znaky a sekvencemi znaků a tzv. kompozici a dekompozici[14]. Implementace mechanismů kompozice a dekompozice by byla pouze zbytečným zkomplikováním problematiky bez užitečného přínosu.

Pro generovaný kód a jeho zavedení do aplikace je důležitý počet bytů nezbytný pro uložení jednoho znaku. V případě ASCII je to 1 byte, UCS-2 potřebuje 2 byty, UCS-4 používá 4 byty a UTF-8 od 1 do 6 bytů.

Generovaný kód bude obsahovat mechanismus pro převod sekvence bytů na čísla reprezentující znaky. V případě dvou a čtyř bytových kódování to zahrnuje detekci pořadí bytů na základě tzv. BOM (Byte Order Mark). Znaky jsou při rozeznávání řetězců pomocí Mooreových strojů porovnávány jako čísla, takže v Moorevých strojích je zakódováno vzájemné přiřazení znaku a čísla. Vstupní zdrojový kód je v kódování UTF-8, což umožňuje zadat libovolný znak sady UCS jako část regulárního výrazu.

Podpora znakových sad je řešena pomocí sdíleného objeku (dynamicky linkované knihovny) programu AutoComp, který při zpracování regulárních výrazů může zajistit případný převod čísel znaků (např. při podpoře ISO-8859-2) a zejména zpřostředkovává předdefinované rozsahy znaků.

Kapitola 3

Nástroj AutoComp

V této kapitole přistoupíme k podrobnostem týkajícím se významu zdrojového kódu a jeho implementace v podobě výstupního kódu.

Nebudeme se zde věnovat syntaxi ani podrobnostem spouštění programu AutocComp, informace tohoto charakteru obsahuje Příloha B. Nicméně postup při instalaci a spouštění programu nastíníme.

3.1 Instalace

V UNIXovém prostředí je instalace rozdělena do tří kroků; extrakce, kompilace a instalace. Prvním krokem je extrakce archivu `autocomp.tar` pomocí příkazu `tar`. Nejprve je nutné zkopírovat soubor `autocomp.tar` do nějakého adresáře kam má uživatel právo zapisovat. Extrakce se provádí spuštěním `tar -xf autocomp.tar` v adresáři. Vznikne adresář `autocomp/`, který obsahuje zdrojové kódy a další data potřebná pro instalaci. Spuštěním příkazu `make` v tomto adresáři se zkompilují zdrojové kódy a vznikne spustitelný soubor `autocomp` a sdílené objekty `autocomp.*.*.so`. Aby byl program spustitelný všemi uživateli systému, je třeba zkopírovat tyto soubory do odpovídajících adresářů v proměnných prostředí `PATH, LD_LIBRARY_PATH`. Manuálová stránka by měla být v podadresáři některého adresáře z proměnné prostředí `MANPATH`. Kopírování se provádí příkazem `make install`, který zkopíruje soubor `autocomp` do adresáře `/usr/localo/bin`, sdílené objekty do `/usr/local/lib` a manuálovou stránku `autocomp.1` do `/usr/local/share/man/cs/man1`. Pokud adresáře nevyhovují nastavení systému, lze použít *adresář* místo `/usr/local` spuštěním `make install prefix=`*`adresář`*. Rovněž lze nastavit adresáře zvlášť pro každou položku, provede se to příkazem `make install bindir=`*`adresář1`* `libdir=`*`adresář2`* `mancs1dir=`*`adresář3`*. Spustitelný soubor se zkopíruje do prvního adresáře, sdílené objekty do druhého a manuálová stránka do třetího adresáře.

Pro instalaci je doporučena GNU nebo BSD varianta utility make, testování bylo prováděno předně pro GNU Make verze 3.80 Kompilace se provádí překladačem gcc (testováno ve verzi 3.4.2 a vyšších). Více informací o projektu GNU lze nalézt v [13].

Instalační balíček pro systém Windows (Win32) je tvořen jedním zip archivem, který

obsahuje zkompilované binární soubory `autocomp.exe` a `autocomp.*.dll`. Rozbalením tohoto archivu do složky se stává program AutoComp funkčním, spustitelný je však pouze z dané složky nebo zadáním celé cesty k souboru `autocomp.exe`. Vyžaduje-li uživatel spustitelnost z jakéhokliv místa v souborovém systému bez celé cesty, nechť podnikne potřebné kroky (přesun souborů nebo přidání složky do `PATH`). Důležitou podmínkou plné funkčnosti je, aby všechny binární soubory z archivu byly umístěny ve stejném adresáři. Tento (zjednodušující) požadavek vyplývá z principu hlednání dynamicky linkovaných knihoven operačním systémem Windows.

3.2 Spouštění

Základní varianta spouštění programu autocomp má tvar `autocomp -o` ***`výstup vstup`***. *Vstup* představuje jméno souboru se zdrojovým kódem pro program, *výstup* představuje jméno výstupních souborů bez přípony. Přípony budou výstupním souborům přiřazeny na základě konvencí pro pojmenovávání zdrojových kódů konkrétního programovacího jazyka. Vstupní zdrojový kód obsahuje deklarace stavů, přechodových hran, popis zařízení pro čtení znaků, popis výsledného programového modulu a specifikaci programovacího jazyka. Podrobný popis syntaxe zdrojového kódu a všech možností a voleb spouštění obsahuje Příloha B.

3.3 Vstupní data

Jak již bylo řečeno v předchozím odstavci, zdrojový kód se z skládá deklarací stavů, přechodových hran a dalších položek. Následující text má přiblížit jednotlivé složky zdrojového kódu bez zbytečného zkoumání způsobu zápisu zdrojového kódu.

Stav (`state`) má unikátní jméno, je volitelně označen za startovní či koncový a obsahuje tázací proceduru. Dále může obsahovat ladící procedury, které jsou pouze informativního charakteru a neovlivňují interpretaci stavů.

Přechodová hrana (`trans`) spojuje dva stavy a je orientovaná. Obsahuje vzor řetězců, které umožňují přechod ze stavu na začátku hrany do stavu na kocni hrany. Vzor řetězců je určen regulárním výrazem a reprezentuje tedy regulární jazyk. Hrana je označena jako hladová nebo líná, označení ovlivňuje prioritu řetězců při náležení do regulárního jazyka. Hrana dále obsahuje přechodovou proceduru a ladící proceduru. Ladící procedura opět neovlivňuje intepretaci hrany.

Vstupní zařízení (`input`) je dáno kombinací typu zařízení a kódování znaků resp. znakové sady. Typy jsou podporovány dva, jde o **soubor** a **paměť**. Kodování znaků se vyznačuje počtem bytů potřebných na reprezentaci jednoho znaku, v této souvislosti označovaných jako kódová entita (code point), a způsobem převodu posloupnosti bytů na entity. Znaková sada pak určuje význam jednotlivých kódových entit, přiřazuje jim konkrétní znaky. Znaková sada je například Unicode a kódování této znakové sady je utf-8. Program AutoComp podporuje možnosti ASCII, Latin1 , UCS-2, UCS-4 a UTF-8. ASCII zastupuje

jednobytové kódování znakové sady ascii (omezeno na 7 bitů). Latin1 je též jednobytové kódování, rozšiřuje ASCII na 8 bitů. UCS-2 značí dvoubytové kódování podmonžiny znakové sady UCS, tzv. BMP - Basic Multilingual Plane. UCS-4 je čtyřbytové kódování, pokrývá celou znakovou sadu UCS. UTF-8 je kódování znakové sady UCS s proměnlivým počtem bytů.

Specifikace **výstup**u (**output**) umožňuje určit programovací jazyk ve kterém bude vygenerován zdrojový kód. Podporovány jsou jazyky C (dle standartu C89) a Perl (verze 5.8). Další položkou specifikace výstupu je určení, zda bude program generován v ladícím ostrém režimu. Program ve vygenerovaný v ladícím režimu volá zmíněné ladící procedury a umožňuje provádět automatické ladící výpisy. To se samozřejmě více či méně projeví na náročnosti práce programu. Cílem této možnosti je usnadnit vývoj aplikace tím způsobem, že uživatel - vývojář nejprve odladí aplikaci používající vygenerovaný kód v ladícím režimu. Poté vygeneruje kód bez ladících operací, který se chová naprosto stejně a je rychlejší. Aby bylo možné tohoto cíle dosáhnout, musí uživatel programovat ladící procedury s vědomím, že ve výsledné verzi nebudou, tzn. ladící procedury nesmí žádným způsobem měnit obsah jim svěřených datových struktur.

Poslední složkou zdrojového kódu je specifikace **modul**u (`module`), která určuje zapojení generovaného kódu do programového celku. Obsahuje název modulu, který bude ve výsledném kódu tvořit předpony jmen symbolů, tedy tříd nebo funkcí, globálních dat a definic. Cílem je určitá kontrola jmeného prostrou vygenerovaného programu. Další složkou je typ *uživatelských dat*. Uživatelská data tvoří určitý styčný bod, který umožňuje ovlivnit práci vygenerovaného programu prostřednictvím zmíněných procedur na základě „stavu“ celého programu. Specifikace modulu dále obsahuje jméno souboru se zdrojovými kódy tázacích a ladících procedur a datové struktury pro uživatelská data. V ladícím režimu je též nutné určit jméno souboru se zdrojovým kódem ladících procedur. Obsah těchto souborů se bude includovat až při překladu vygenerovaného zdrojového kódu překladačem programovacího jazyka.

3.4 Interpretace vstupních dat

Nyní navrhneme strukturu, kterou použijeme pro popis mechanismu, jakým vstupní kód určuje význam výstupního kódu. Instance této struktury bude jednoznačně určena zdrojovým kódem a bude jednoznačně určovat sémantiku výstupního kódu v obecné rovině. Některé části vstupního kódu se v tomto modelu neuplatní, protože se týkají konkrétní podoby výstupního kódu nebo vstupního zařízení.

Definujme si **stroj** M, jako 5-tici (Q, X, δ, S, F), kde

Q je množina stavů

X je množina vstupních symbolů

$\delta \subseteq (Q \times X \times Q)$ je přechodová relace, též přechodová funkce $(\delta : (Q \times X) \to \mathcal{P}^Q)$

$S \subseteq Q$ je množina stavů označených jako startovní.

$F \subseteq Q$ je množina stavů označených za koncové, platí $S \cup F \subseteq Q$

Definovaný stroj nápadně připomíná konečný automat. Našim cílem je však uchopit myšlenku regulárních výrazů na přechodových hranách stroje. V teorii automatů a gramatik je tato myšlenka zachycena pomocí substituce jazyků[9], v naší konstrukci se ovšem snažíme co nejvíce udržet návaznost na zdrojový kód. Vstupní symbol bude v našem případě regulární jazyk, tedy množina slov neboli řetězců. Výpočet stroje M bude zcela analogický výpočtu konečného automatu, budou-li na vstupní pásce regulární jazyky místo znaků. Výpočet deterministického konečného automatu je posloupnost přechodů mezi stavy a je jednoznačně určen posloupností vstupních symbolů a jedním startovním stavem.

Vstupním zdrojovým kódem máme jednoznačně určené množiny stavů. Přechodová hrana zdrojového kódu představuje trojici (p, R, q), kde $p, q \in Q_M$ a R je regulární jazyk. Máme tedy danou i relaci δ a množinu vstupních symbolů.

Těžiště našeho řešení spočívá v navržení mechanismu, který na základě vstupní pásky tvořené znaky předkládá automatu pásku tvořenou obecnějšími objekty, regulárními jazyky. Neformálně můžeme říci, že čte vstupní znakovou pásku a hledá shodu s regulárními jazyky. Podle toho, zda je jazyk na hladové či líné hraně pak rozhoduje, který jazyk předloží automatu. Líné hrany preferují nejkratší možné slovo z regulárního jazyka, hladové nejdelší. Dalším prostředkem, který používá k rozhodnutí, jsou tázací a přechodové procedury. Tázací procedura svou návratovou hodnotou umožní některé regulární výrazy vyloučit, přechodová procedura naopak „posvěcuje“ použití slpva pro konkrétní přechod, tedy interpretaci slova jako zástupce regulárního výrazu. Tázací a přechodové procedury lze tedy vnímat jako jakási rozhodnutí zvenčí, na základě informací, které sám stroj M nemůže uchopit.

Jakmile mechanismus vybere nějaký regulární jazyk a předloží jej automatu, automat přejde po odpovídající hraně do odpovídajícího stavu. Interakce mezi mechanismem převodu pásek a výpočtem automatu jsou ovšem oboustrané; stav výpočtu automatu určuje, z jakých regulárních výrazů lze vybírat. Výpočet stroje M pak budeme vnímat jako posloupnost přechodů mezi stavy a volání přechodových procedur.

Přistoupíme-li k formalizaci těchto myšlenek, záhy narazíme na nejednoznačnosti. Navrhneme tedy i jejich řešení. Upřesněme si pojem procedury. ***Procedura*** *je obecně konečná posloupnost instrukcí, která má určitý počet vstupů a výstupů.* [1]. Zmíněné tázací, přechodové a ladící procedury jsou reprezentovány funckemi v zadaném programovacím jazyce. Hranám a stavům se procedury přiřazují pomocí svých jmen. Tázací procedura bude mít vstupy $p \in Q_M, q \in Q_M$ a U, výstupem bude celé číslo. Přechodová procedura pak bude mít vstupy $p \in Q_M, q \in Q_M, s \in R \in X_M$ a U, výstup stejný jako tázací procedura. Symbol U není blíže specifikován, budeme jej nazývat uživatelská data. Očekává se, že to je ukazatel na datovou strukturu, která nějakým způsobem reprezentuje stav aplikace, do níž je výstup programu AutoComp začleněn. Datový typ symbolu U je též specifikován vstupním kódem. Formalizujme nyní přiřazení vlastností stavům a hranám:

Funkce $t: Q_M \rightarrow T$, kde T je množina tázacích procedur, přiřazuje stavu tázací proceduru.

Funkce $p: \delta_M \rightarrow P$, kde P je množina přechodových procedur, přiřazuje hraně přechodovou proceduru.

Funkce $h: \delta_M \rightarrow \{0, 1\}$ určuje, zda je hrana hladová či líná. Řekněme, že hrana je hladová, pokud $h\,(e) = 1$.

Funkce t,p a h jsou rovněž jednoznačně dány vstupním kódem.

Algoritmus 3.1 popisuje mechanismus, jakým se na základě vstupní pásky tvořené znaky určí jeden vstupní symbol stroje M. Jelikož jsme regulární výrazy jednoznačně očíslovali, existuje bijekce mezi regulárnímy výrazy (vstupními symboly) a přechodovými hranami. Protože algoritmus určí přímo přechodovou hranu, je zbytečné, aby automatu předkládal pouze její složku, tedy jazyk na hraně, a automat následně hledal odpovídající hranu.

Algoritmus dává posloupnost volání přechodových procedur. Shrňme si pořadí těchto volání formou dvou pravidel:

> Nejprve jsou volány přechodové procedury pro řetězce odpovídající líným přechodovým hranám v pořadí od nejkratšího řetězce po nejdelší.
>
> Následně jsou přechodové procedury volány pro řetězce odpovídající hladovým přechodovým hranám v pořadí od nejdelšího po nekratší.

Všimněme si též, že jakmile přechodová procedura vrátí nenulovou hodnotu, přechod je uskutečněn aniž by se testovaly další možnosti. Jelikož nemusí být přechodová procedura definována pro každou přechodovou hranu, plyne z toho nebezpečí nejednoznačnosti výpočtu.

Vezměme stroj, ve kterém existují alespoň dvě různé hrany $(p, R_1, q_1), (p, R_2, q_2) \in \delta_M$ takové, že

$$h((p, R_1, q_1)) = h((p, R_2, q_2)) \wedge R_1 \bigcap R_2 \neq \emptyset \wedge \{(p, R_1, q_1), (p, R_2, q_2)\} \bigcap \mathcal{D}(t) = \emptyset.$$

$\mathcal{D}(t)$ značí definiční obor funkce t. Předpokládejme, že algoritmus 3.1 dospěl do fáze, kdy poprvé zkouší volat přechodovou proceduru jedné z hran. Jelikož přechodová procedura není definována, krok je ukončen a přechod se provede, aniž by byla vzata v úvahu další možnost či byl přechod *posvěcen zvenčí*. Při tom z algoritmu není možné určit, po které z hran se přejde. Korektním řešením by byl přechod po obou hranách, výpočet by pak pokračoval paralelně ze stavů q_1 a q_2. AutoComp takové chování simuluje přidáním speciálního stavu, který zastřešuje stavy q_1 a q_2. Obecněji se tímto tématem zabývá kap. 3.5, nyní budeme předpokládat stroj bez zmíněné nejednoznačnosti, tzn. takový, že

$$\forall((p_1, R_1, q_1), (p_2, R_2, q_2)) \in (\delta_M \times \delta_M) \text{ platí}$$
$$(p1 \neq p2 \vee h(p_1, R_1, q_1) \neq h(p_2, R_2, q_2) \vee R_1 \bigcap R_2 = \emptyset.$$

Hlavní částí výstupního zdrojového kódu programu AutoComp, generovaného programu, je právě efektivní simulace výpočtu stroje (odd. 3.6), který vzejde ze vstupního zdrojového kódu. Simulace je reprezentována hlavní funkcí *RUN*. Mimo ni je vygenerována řada dalších funkcí, které slouží k obsluze a inicializaci stroje a vstupního zařízení (odd. 3.7). Funkce *RUN* představuje však pouze část výpočtu, držíme-li se analogie s konečnými automaty. Jejich výpočet končí až s koncem vstupní pásky, naproti tomu výpočet

Vstup: stav $s \in S_M$, vstupní zařízení I (páska znaků), uživatelská data U
Výstup: přechodová hrana
begin
  $z \Leftarrow$ prázdný zásobník;
  $r \Leftarrow$ prázdný řetězec;
  while $(\exists c)$ *takové, že* $\delta_M^{s,(r.c)} \neq \emptyset$ **do**
    foreach $e \in \delta_M^{s,r}$ **do**
      if $Call(t(p_e), p_e, q_e, U) \neq 0$ **then**
        if $h(e) = 0$ **then**
          if $Call(p(e), p_e, q_e, r, U) \neq 0$ **then**
            return e;
        else
          $Push(z, (e, r))$;
    $r \Leftarrow r + GetChar(I)$;
  while $z \neq$ *prázdný zásobník* **do**
    $(e, u) \Leftarrow Pop(z)$;
    $Unget(I, v$ takové, že $u.v = r)$;
    $r \Leftarrow u$;
    if $Call(p(e), p_e, q_e, u, U) \neq 0$ **then**
      return e;
  return *neexistující hrana*;
end

- $\delta_M^{a,b} = \{(p, R, q) \in \delta_M$ takové, že $p, q \in Q_M \wedge R \in X_M \wedge p = a \wedge b \in R\}$
- p_e je stav na začátku hrany e, q_e je stav na konci hrany e
- $GetChar(I)$ načte znak ze vstupního zařízení I.
- $Unget(I, r)$ naopak „vrátí“ řetězec r zpět na vstup.
- $Call(p, a_1, a_2, ..., a_n)$ provede proceduru p se vstupy $a_1, a_2, ..., a_n$ a vrátí její výstup. V případě že procedura není definována vrací 1.

Algoritmus 3.1: Krok výpočtu stroje M.

```
Vstup: stav s ∈ M → S , vstupní zařízení I, uživatelská data U
Výstup: stav s
begin
    hrana e;
    k ⇐ 0;
    while (s je definován ) ∧ (k = 0 ∨ s není koncový ) ∧ (I není na konci ) do
        e ⇐ Krok(s, I, U);
        if e ∈ δ_M then
            s ⇐ p_e;
        else
            /* nedefinovaná hrana                                    */
            return nedefinovaný stav;
        k++;
    return s;
end
```

Algoritmus 3.2: Výpočet stroje M

zprostředkovaný funkcí *RUN* končí, jakmile stroj přejde do koncového stavu. Efektu celého výpočtu lze dosáhnout opakovaným voláním funkce *RUN*, dokud nebyl zpracován celý vstup. Algoritmus 3.2 ilustruje výpočet funkce *RUN*, operace $Krok$ zde představuje volání algoritmu 3.1.

3.5 Determinisace stroje

Jak bylo již řečeno, stroj využívá procedur k určení regulárního výrazu jako zástupce řetězce ze vstupu. Absence procedur může ve speciálním případě znemožnit správné určení tohoto výrazu, což je ovšem řešitelné pomocí úprav zadaného stroje. Výpočet stroje je zvenčí, tedy z aplikace, zachytitelný jen v určitých bodech. Jsou jimi volání procedur a návratová hodonota hlavní funkce. AutoComp při překladu zdrojového kódu provádí úpravy stroje tak, aby byly odstraněny nejednoznačnosti. Tím, že v kritických bodech chybí procedury, se tedy otevírá manévrovací prostor pro úpravy stroje. Cílem úprav stroje je dosáhnout požadovaného chování při minimální změně v projevech výpočtu.

Nejednoznačnost ve výpočtu stroje vzniká, pokud

$$(\exists s \in Q_M)(\exists e_1 = (p_1, R_1, q_1), e_2 = (p_2, R_2, q_2) \in \delta_M) \text{ takové, že}$$
$$(p_1 = p_2 \wedge q_1 \neq q_2 \wedge e_1, e_2 \notin \mathcal{D}(p) \wedge R_1 \bigcap R_2 \neq \emptyset \wedge h(e_1) = h(e_2)).$$

Stroj, ve kterém nejednoznačnost existuje, nazveme nedeterministický, jinak jej označíme přívlastkem deterministický. Algoritmus 3.3 ukazuje, jakým způsobem funguje převod řetězce ze vstupu na hrany v rámci nedeterministického stroje. Pokud tedy dojde k situaci, že stroj nemůže odpovědně rozhodnout, kterou hranu zvolit, použije všechny kandidáty. Provede se přechod po množině hran do množiny stavů, analogicky s výpočtem nedeterministického konečného automatu.

Vstup: množina stavů $S \subseteq S_N$, vstupní zařízení I, uživatelská data U
Výstup: množina přechodových hran
begin
    $z \Leftarrow$ prázdný zásobník;
    $r \Leftarrow$ prázdný řetězec;
    $v \Leftarrow$ prázdný řetězec;
    $T \Leftarrow \emptyset$;
    while $(\exists c)$ *takové, že* $\delta_N^{S,(r.c)} \neq \emptyset$ **do**
        foreach $e \in \delta_N^{S,r}$ **do**
            if $Call(t(p_e), p_e, q_e, U) \neq 0$ **then**
                if $h(e) = 0$ **then**
                    if $e \in \mathcal{D}(p)$ **then**
                        if $Call(p(e), p_e, q_e, r, U) \neq 0$ **then**
                            return $\{e\}$;
                    else
                        $T \Leftarrow T \bigcup \{e\}$;
                else
                    $Push(z, (e, r))$;
        if $T \neq \emptyset$ **then**
            return T;
        $r \Leftarrow r + GetChar(I)$;
    while $z \neq$ *prázdný zásobník* **do**
        $(e, u) \Leftarrow Pop(z)$;
        $Unget(I, v$ takové, že $u.v = r)$;
        $r \Leftarrow u$;
        if $e \in \mathcal{D}(p)$ **then**
            if $Call(p(e), p_e, q_e, u, U) \neq 0$ **then**
                return $\{e\}$;
        else
            if $T \neq \emptyset \wedge v \neq u$ **then**
                return T;
            $T \Leftarrow T \bigcup \{e\}$;
            $v \Leftarrow u$;
    return *neexistující stav*;
end

$\star$ $\delta_N^{A,b} = \{(p, R, q) \in \delta_M$ takové, že $p, q \in Q_M \wedge p \in A \wedge b \in R\}$

Algoritmus 3.3: Krok výpočtu nedeterministického stroje N.

Cílem úprav stroje je vytvořit deterministický stroj, při jehož výpočtu bude posloupnost volání přechodových procedur (včetně parametrů) shodná s posloupností generovanou výpočtem nedeterministického stroje. Princip úprav je podobný principu determinisace konečného automatui Vytvoří se stavy a přechody, které představují množiny stavů při paralelním výpočtu stroje. Požadavek zachování posloupnosti přechodových procedur však s sebou nese potřebu mapovat nově vzniklé přechodové hrany na původní. Ideu konstrukce deterministického stroje M na základě nedeterministického protějšku N je znázorněn v algoritmu 3.4. Součástí je i konstrukce funkce $\mu : \delta_N \rightarrow \delta_M$, která mapuje nové hrany na původní.

O stavu $\{s_1, ..., s_n\}$ takto zkonstruovaného stroje budeme říkat, že je **sloučený** a že **zastřešuje** stavy $s_1, ..., s_n$ pokud $n \geq 2$. Stavy zkonstruovaného stroje označíme (očíslujeme) tak, že stav $\{s\}$ ponese označení stejné jako stav s v původním automatu. Použijeme-li pro výpočet stroje M lehkou modifikaci původního algoritmu, která je předvedena v algoritmu 3.5, dosáhli jsme stanoveného cíle.

Podíváme-li se však na posloupnost volání tázacích funkcí při výpočtu, může se oproti původnímu stroji změnit. Tázací funkce dostane jako druhý argument sloučený stav, který se v původním stroji vůbec nevyskytoval. Z toho důvodu jsou během kompilace zdrojového kódu AutoCompem ohlášeny sloučené stavy formou varování na chybový výstup. Je-li to nutné, může programátor aplikace upravit chování tázacích procedur v případě, že dostanou sloučený stav. V takovém případě však stojí za zvážení, jestli nebylo možné navrhnout zdrojový kód pro AutoComp lépe.

Ještě zbývá vyřešit případ, kdy nějaký sloučený stav zastřešuje alespoň jeden koncový stav. V takovém případě bude i sloučený stav koncovým. V případě, že se výpočet stroje dostane do sloučeného a navíc koncového stavu, výpočet končí a vrací označení sloučeného stavu. Aplikace tak dostává prostor pokračovat z libovolného nesloučeného stavu. Koncový sloučená stav učiníme také startovním. Tím dáme aplikaci navíc možnost spustit výpočet ze stavu, ve kterém skončil. Opět ovšem stojí za zvážení, zda je vhodný návrh stroje, kde se lze „nejednoznačným“ přechodem dostat do koncového stavu.

3.6 Simulace stroje

Hlavním účelem vygenerovaného progarmu je simulovat výpočet stroje zadaného formou zdrojového kódu, pomocí přechodových a tázacích procedur komunikovat se zbytkem aplikace a v ladícím režimu poskytovat informace o průběhu simulace pomocí ladících funkcí. Simulace zachovává sled kroků a pořadí volání procedur dané v algoritmech 3.2 a 3.5. Využívá speciálních technik pro porovnávání vstupu se vzory (regulárními jazyky) s pomocí bufferování vstupních dat. Program obsahuje stroj M v datových strukturách v podobě tabulek. Značnou část tabulek tvoří Mooreovy stroje, které přijímají regulární jazyky na hranách zadaného stroje. Každému stavu je přiřazen jeden Mooreův stroj, který přijímá sjednocení jazyků na hranách vycházejících z tohoto stavu. Výstupem Mooreova stroje na koncovém stavu je identifikace přechodové hrany ve formě indexu do tabulek. Za pomocí indexu získává simulátor přechodovou proceduru, výchozí a cílový stav. Stav je určen

Vstup: nedeterministický stroj N
Výstup: deterministický stroj M spolu s mapovací funkcí
$\mu : (Q_N \times X_N \times Q_N) \to (Q_M \times X_N \times Q_M)$

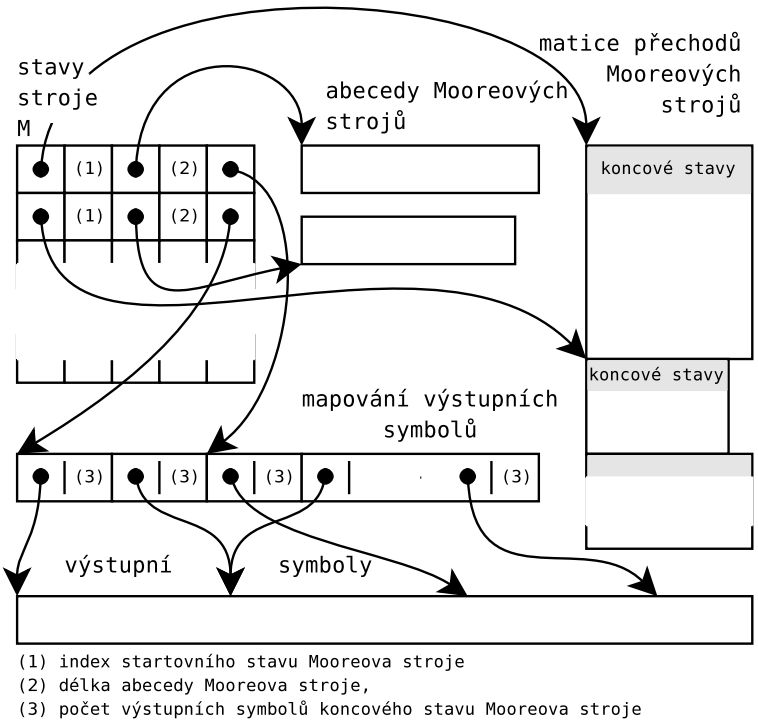
begin
- $\delta_M \Leftarrow \emptyset$;
- $X_M \Leftarrow X_N$;
- **foreach** $s \in S_N$ **do**
  - **if** $s \in F_N$ **then**
    - $F_M \Leftarrow F_M \bigcup \{\{s\}\}$;
  - $S_M \Leftarrow S_M \bigcup \{\{s\}\}$;
- $Q_M \Leftarrow S_M$;
- **while** $\exists T \in Q_M$ *takové, že* T *je dosud nenavštívený stav* **do**
  - **foreach** $r \in X_M$ **do**
    - $S_H \Leftarrow \emptyset$; $S_L \Leftarrow \emptyset$; $e_H \Leftarrow \emptyset$; $e_L \Leftarrow \emptyset$;
    - **foreach** $e \in \delta_N^{T,r}$ **do**
      - **if** $e \in \mathcal{D}(p)$ **then**
        - $\delta_M \Leftarrow \delta_M \bigcup (T, R_e, p_e)$;
        - $\mu(T, R_e, p_e) \Leftarrow e$;
        - $E_M \Leftarrow E_M \bigcup \{e\}$; $Q_M \Leftarrow Q_M \bigcup \{\{p_e\}\}$;
      - **else**
        - **if** $h(e) = 1$ **then**
          - $S_H \Leftarrow S_H \bigcup \{p_e\}$;
          - $e_H \Leftarrow e$;
        - **else**
          - $S_L \Leftarrow S_H \bigcup \{p_e\}$;
          - $e_L \Leftarrow e$;
    - **if** $S_H \neq \emptyset$ **then**
      - $\delta_M \Leftarrow \delta_M \bigcup (T, \{r\}, S_H)$;
      - $m(T, r, S_H) \Leftarrow m(T, r, S_H) \bigcup e_h$;
      - $E_M \Leftarrow E_M \bigcup \{e\}$; $Q_M \Leftarrow Q_M \bigcup \{S_H\}$;
    - **if** $S_L \neq \emptyset$ **then**
      - $\delta_M \Leftarrow \delta_M \bigcup (T, r, S_L)$;
      - $m(T, r, S_L) \Leftarrow m(T, r, S_L) \bigcup e_l$;
      - $E_M \Leftarrow E_M \bigcup \{e\}$; $Q_M \Leftarrow Q_M \bigcup \{S_L\}$;

end

Algoritmus 3.4: Konstrukce determinstického stroje M.

Vstup: stav $s \in S_M$, vstupní zařízení I (páska znaků), uživatelská data U
Výstup: přechodová hrana
begin
  $z \Leftarrow$ prázdný zásobník;
  $r \Leftarrow$ prázdný řetězec;
  while $(\exists c)$ *takové, že* $\delta_M^{s,(r.c)} \neq \emptyset$ **do**
    $E \Leftarrow \delta_M^{s,r}$;
    $f \Leftarrow$ nedefinovaná hodnota;
    foreach $e \in \delta_M^{s,r}$ *takové, že* $\mu(e) \notin \mathcal{D}(p)$ **do**
      if $Call(t(p(\mu(e)), p_{\mu(e)}, q_{\mu(e)}, U) \neq 0$ **then**
        if $h(e) = 0$ **then**
          $f \Leftarrow e$;
        else
          $Push(z, (f, r))$;
    foreach $e \in \delta_M^{s,r}$ *takové, že* $\mu(e) \in \mathcal{D}(p)$ **do**
      if $Call(t(p(\mu(e)), p_{\mu(e)}, q_{\mu(e)}, U) \neq 0$ **then**
        if $h(e) = 0$ **then**
          if $Call(p(\mu(e)), p_{\mu(e)}, q_{\mu(e)}, r, U) \neq 0$ **then**
            return e;
        else
          $Push(z, (e, r))$;
    if f *je definovaná* **then**
      return f;
    $r \Leftarrow r + GetChar(I)$;
  while $z \neq$ *prázdný zásobník* **do**
    $(e, u) \Leftarrow Pop(z)$;
    $Unget(I, v$ takové, že $u.v = r)$;
    $r \Leftarrow u$;
    if $Call(p(\mu(e)), p_{\mu(e)}, q_{\mu(e)}, u, U) \neq 0$ **then**
      return e;
  return *neexistující hrana*;
end

Algoritmus 3.5: Krok výpočtu stroje M.



Obrázek 3.1: Schema struktury tabulek simulátoru.

indexem do tabulek, který zpřístupňuje Mooreův stroj, ukazatel na tázací proceduru a informaci o tom, zda je stav koncový. Mooreův stroj je tvořen abecedou a maticí jeho stavů, která určuje i přechodovou funkci Mooreova stroje. Struktura tabulek je znázorněna na obrázku 3.1.

Mooreův stroj slouží v tomto případě k určení, kterým regulárním jazykům náleží řetězec na vstupu. Simulátor mění stav Mooreova stroje na základě znaku získaného ze vstupu a dostane-li se do koncového stavu, provádí akce s odpovídající přechodovou hranou. Snadno lze také rozpoznat řetězec, který již nelze prodloužit tak, aby náležel do odpovídajících regulárních jazyků. Indikuje to prázdná (nedefinovaná) přechodové funkce Mooreova stroje pro aktuální stav a znak. Samotné přijímání řetězců Mooreovými stroji má lineární časovou složitost vzhledem k délce řetězce. Nečte se tedy žádný znak více než jednou. Ovšem možnost odmítnutí nejdelšího načteného slova přechodovou procedurou způsobuje, že může být jeden znak opakovaně zpracován jinými Mooreovy stroji. To s sebou nese potřebu možnosti vrátit se zpět při načítání vstupu ($Unget()$ v algoritmu 3.5). Pokud to vůbec lze provádět přímo se vstupním zařízením, může to být velmi náročné na prostředky a zpomalovat běh programu. Proto je použita mezipaměť vstupního zařízení, která poskytuje požadovanou funkcionalitu a téměř nezpomaluje výpočet, protože nevrací znaky do skutečného vstupního zařízení.

Implementace mezipaměti se liší podle konkrétního vstupního zařízení. Pro soubor načítá ze zařízení bloky *bytů*, ukládá je do paměti, kde je teprve reprezentuje jako znaky a

zprostředkovává čtení a vracení. V případě čtení z paměti není mezipaměť potřeba. Nečiní tedy nic kromě toho, že si pamatuje pozice v textu a upravuje je podle prováděných operací. Ve skutečnosti vstupní mezipaměť úzce spolupracuje se zásobníkem a při vyzvednutí vrcholu zásobníku mezipaměť automaticky provede nenáročné operace související s návratem zpět. Další součástí svázanou s mezipamětí jsou čítače načtených znaků a řádek. Ty udržují informaci o pozici v načteném textu srozumitelnou pro člověka, který se nejlépe orientuje podle čísla řádky a znaku na řádce. Čísla řádky a znaku jsou aplikaci dostupná skrze rozhraní a mohou dobře posloužit k informativním, varovným či chybovým výpisům aplikace. Buffer, zásobník a čítače tvoří společně datovou strukturu, která v programu reprezentuje vstupní zařízení. K této struktuře aplikace přiřadí skutečné vstupní zařízení, ve formě *handleru* souboru nebo oblasti paměti.

Logicky lze rozdělit práci simulátoru do tří spolupracujících jednotek; simulátor, vstupní struktura a Mooreovy stroje. Simulátor je jakousi řídící složkou, spouští porovnávání vzorů pomocí Mooreových strojů a na základě jejich výstupních symbolů vyzvedává řetězce ze vstpuní struktury a volá odpovídající procedury. Mooreovy stroje čtou ze vstupní struktury znaky a mění odpovídajícím způsobem stav. Dosáhnou-li koncového stavu, předají výstupní symboly zpět simulátoru. Vstupní struktura udržuje počátek a konec právě načítaného řetězce. Díky tomu je schopna poskytnout řetězec simulátoru či jej efektivně uložit na zásobník (uloží se pouze index počátku a konce řetězce v mezipaměti a hodnoty čítačů řádek a znaků).

3.7 Zavedení výstupu do aplikace

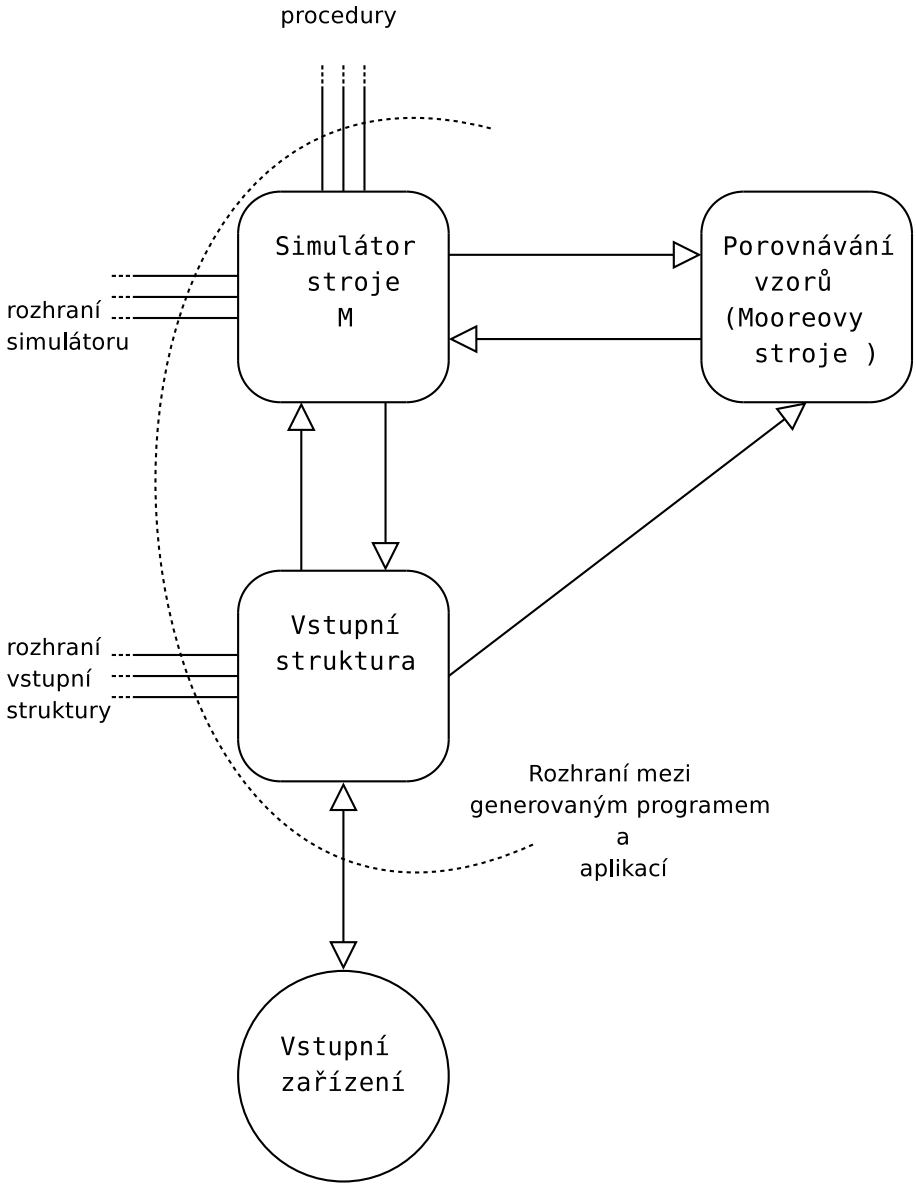
Funkcionalita vygenerovaného kódu, simulátoru stroje M, je přístupna skrze rozhraní, které je do jisté míry proměnlivé dle specifikovaného vstupního zařízení a programovacího jazyka. V objektově orientovaných programovacích jazycích je generovaný kód tvořen několika třídami a rozhraní je tvořeno jejich metodami. Objektový model je použit v jazyce Perl. V jazyce C je kód tvořen několika globálními proměnnými, definicí datových struktur a rozhraní je tvořeno sadou funkcí. Dále se liší argumenty některých funkcí (metod) v závislosti na vstupním zařízení. Popis rozhraní proto bude nejdříve proveden pomocí symbolických názvů volání a později upřesněny konkrétní typy parametrů a jména funkcí (metod).

Přidržíme-li se rozdělení simulátoru na části obsluhující vstupní zařízení a simulátor stroje M z odd. 3.6, budeme mluvit o rozhraní vstupní struktury a rozhraní simulátoru. Schematicky je struktura generovaného kódu a jeho rozhraní znázorněna na obrázku 3.2.

Nejprve popíšeme rozhraní vstupní struktury. Datovou strukturu sloužící k ovládání vstupního zařízení tvořenou mezipamětí, zásobníkem, čítači a vstupním zařízením označíme symbolem I. Nad touto strukturou jsou definovány následující operace.

$I \rightarrow$ *ASSIGN* (*device*) inicializuje datovou strukturu a přiřadí jí vstupní zařízení *device*, které je reprezentováno *handlerem* souboru nebo referencí na oblast paměti.

$I \rightarrow$ *END*() vrací nenulovou hodnotu, pokud byl již načten konec vstupního zařízení



Obrázek 3.2: Generovaný program a jeho rozhraní.

a zpracována všechna data v mezipaměti.

$I \rightarrow$ *ERROR*() indikuje chybu vstupního zařízení nenulovou návratovou hodnotou. Vrací číselný kód chyby, který rozlišuje mezi chybami struktury, zařízení a blíže nespecifikovatelnými chybami (chybové kódy viz Příloha B).

$I \rightarrow$ *POSITION*($\uparrow$ *line*, $\uparrow$ *column*) zjišťuje stav čítačů obsahujících informace o pozici v čteném textu. Značka $\uparrow$ určuje tzv. výstupní argumenty, které jsou předávány referencí nebo ukazatelem. Argumenty *line* a *column* jsou číselné proměnné, do kterých bude nastaven stav odpovídajících čítačů. Stav čítačů vždy odpovídá začátku načteného řetězce.

$I \rightarrow$ *BUFFER*() vrací dosud nezpracovaný obsah mezipaměti ve formě reference na řetězec v daném kódování.

$I \rightarrow$ *FREE*() „odpojí“ strukturu I od vstupního zařízení a uvolní alokované prostředky pro mezipaměť a zásobník. Neuvolňuje paměť obsazenou samotnou strukturou I. Uvádí strukturu do neinicializovaného stavu, je možné ji znovu inicializovat voláním *ASSIGN*.

Pro korektní funkci struktury I není doporučeno manipulovat s obsahem přiřazeného vstupního zařízení dokud není struktura uvolněna (*FREE*). Operace *END*, *ERROR*, *POSITION*, *BUFFER* je možné provádět pouze na inicializované struktuře I, tzn. mezi operacemi *ASSIGN* a *FREE*.

Symbolem S budeme označovat simulátor stroje. Rozhraní simulátoru je tvořeno následujícími operacemi.

$S \rightarrow$ *RUN*($\uparrow I, s, U$) provede simulaci výpočtu stroje M nad inicializovanou vstupní strukturou I. Stav, odkud výpočet začíná, je dán číslem s. U jsou uživatelská data. Tato operace vrací číslo stavu, ve kterém výpočet skončil.

$S \rightarrow$ *FINAL*(s) vrací nenulové číslo, pokud číslo s identifikuje koncový stav.

Další složkou rozhraní simulátoru jsou tázací, přechodové a ladící procedury. Ve skutečnosti jsou to funkce (metody) poskytované aplikací. Přesto si jej zde popíšeme abychom mohli specifikovat typy argumentů.

$ASK_o(p, q, U)$ je tázací procedura stavu o. Argumenty p a q jsou čísla stavů, argument U obsahuje uživatelská data.

$CBACK_e(p, q, r, U)$ je přechodová procedura hrany e. Argumnty p a q jsou čísla stavů, r je řetězec a U zastupuje uživatelská data.

$IN_o(p, q, U)$ je ladící procedura volaná při vstupu do stavu o (argumenty jako ASK_o).

$OUT_o(p, q, U)$ je ladící procedura volaná při opuštění stavu o (argumenty jako ASK_o).

$MOVE_e(p, q, r, U)$ je ladící procedura volaná při přechodu po hraně e (argumenty jako $CBACK_o$).

Aby procedury nenarušily korektnost simulace, není v nich v žádném případě dovoleno použití operací *RUN* (rekurze), *ASSIGN* a *FREE* nad aktuálně používanou vstupní strukturou. Naopak užitečné může být použití operace *POSITION* v tělech procedur, zejména přechodových. Poznamenejme, že je však nutné zpřístupnit vstupní strukturu I v těle procedury, nejlépe pomocí reference (ukazetele) v uživatelských datech.

Rozhraní pracuje s čísly stavů, zatímco uživatel zná jména stavů. Vygenerovaný program proto obsahuje mechanismus, který umožní získat číslo stavu na základě jeho jména. V jazyce C je to zprostředkováno definicemi maker obsahujících jméno stavu a nesoucích číslo stavu jako hodnotu. Nedefinovaný stav má speciální číslo a jemu odpovídající makro. V jazyce Perl je deklarováno asociativní pole, jehož klíči jsou jména stavů a hodnotami jsou čísla stavů. Nedefinovaný stav je zde reprezentován nedefinovanou hodnotou (funkce `undef()` a `defined()`).

Skutečná jména funkcí, tříd, metod a definic ve vygenerovaném kódu jsou tvořena na základě jména modulu (viz kap. 3.3). Důvodem je umožnit určitou kontrolu jmenného prostoru uživateli. V následujícím textu bude symbol <*modul*> zastupovat jméno modulu v originální podobě a symbol <*MODUL*> zastupovat jméno modulu velkými písmeny. Vezměme si například jméno modulu `Stroj007XYZ`. Řekneme-li, že jméno nějaké funkce je <*modul*>`_fnc`, bude skutečné jméno `Stroj007XYZ_fnc`. Podobně makro <*MODUL*>`_FOO`, bude ve skutečnosti `STROJ007XYZ_FOO`. Je zřejmé, že pokud je originální jméno velkými písmeny, symboly <*modul*> a <*MODUL*> zastupují totéž.

V jazyce Perl je zdrojový kód vygenerován do jednoho souboru s příponou `.pm`. Struktura vstupu I je představována třídou <*modul*>`Input`. Simulátor stroje pak třídou <*modul*>. Rozhraní je tvořeno metodami těchto tříd. Třída <*modul*>`States` představuje asociativní pole čísel stavů podle jmen. Nedefinovaný stav je reprezentován nedefinovanou hodnotou.

V jazyce C jsou vygenerovány dva soubory. Hlavičkový soubor s příponou `.h` obsahuje hlavičky funkcí, definice veřejných datových typů a maker. Zdrojový soubor s příponou `.c` obsahuje těla funkcí, datové struktury a proměnné. Předpokládá se separátní kompilace a inkluze hlavičkového souboru. V jazyce C je struktura vstupu zastoupena typem <*modul*>`_input_t`. Rozhraní simulátoru a vstupní struktury je tvořeno globálními funkcemi. Čísla stavů jsou přístupná skrze definice tvaru <*MODUL*>`_S_`$< jmeno >$, kde $< jmeno >$ značí jméno stavu. Řekněme, že jméno modulu je například `stroj` a stav jménem `Stav007` bude mít čílso například 24. Hlavičkový soubor bude obsahovat definici `#define STROJ_S_Stav007 24`. Makro <*MODUL*>`_STATE_NONE` nese číslo nedefinovaného stavu.

Tabulka 3.1 obsahuje přehled funkcí či metod odpovídajících symbolickým prvkům rozhraní v závislosti na programovacím jazyce. Funkce a metody jsou v tabulce uvedeny s typy

	C	Perl
operace	funkce	metoda

ASSIGN	`int`
	<*modul*>`_input_assign(`
	  <*modul*>`_input_t*,`
	  zařízení)
		`INT`
		<*modul*>`Input::assign`
		  zařízení

END	`int` <*modul*>`_input_end(` <*modul*>`_input_t*)`	`INT` <*modul*>`Input::at_end`
ERROR	`int` <*modul*>`_input_error(` <*modul*>`_input_t*)`	`INT` <*modul*>`Input::was_error`

POSITION	`void`
	<*modul*>`_input_get_position(`
	  <*modul*>`_input_t*,`
	  `int*,`
	  `int*)`
		`VOID`
		<*modul*>`Input::`
		`get_position`
		  `\INT,`
		  `\INT`

BUFFER	*řetězec*
	< *modul* >$\longleftrightarrow$
	`_input_get_buffer_content(`
	  <*modul*>`_input_t*)`
		`STRING`
		<*modul*>`Input`
		`::get_buffer_content`

FREE	`void` <*modul*>`input_free(` <*modul*>`_input_t*)`	`VOID` <*modul*>`Input::free`

RUN	`unsigned int`
	<*modul*>`run(`
	  <*modul*>`_input_t*,`
	  `unsigned int,`
	  uživatelská data)
		`INT`
		<*modul*>`::run`
		  `\` < *modul* >`Input,`
		  `INT,`
		  `\`*uživatelská data*

FINAL	`unsigned int`
	<*modul*>`_is_state_final(`
	  `unsigned int)`
		`INT`
		<*modul*>`::is_state_final`
		  `INT`

Tabulka 3.1: Rozrhaní generovaného kódu v závislosti na programovacím jazyce.

symb.	C	Perl

ASK_o	`int`
	procedura`(`
	`unsigned int,unsigned int,`
	uživatelská data`)`
		`INT`
		uživatelská data`::`*procedura*
		`INT, INT`

$CBACK_o$	`int`
	procedura`(`
	`unsigned int,unsigned int,`
	řetězec`,`
	uživatelská data`)`
		`INT`
		uživatelská data`::`*procedura*
		`INT, INT, STRING`

IN_o	`void`
	procedura`(`
	`unsigned int,unsigned int,`
	uživatelská data`)`
		`VOID`
		uživatelská data`::`*procedura*
		`INT, INT`

OUT_o	`void`
	procedura`(`
	`unsigned int,unsigned int,`
	uživatelská data`)`
		`VOID`
		uživatelská data`::`*procedura*
		`INT, INT`

$CHNG_o$	`void`
	procedura`(`
	`unsigned int,unsigned int,`
	řetězec`,`
	uživatelská data`)`
		`VOID`
		uživatelská data`::`*procedura*
		`INT, INT, STRING`

Tabulka 3.2: Typy procedur v závislosti na programovacím jazyce.

	C	Perl
velikost kódování	*řetězec*	
`1b` *(ascii, utf-8)*	`unsigned char*`	`STRING`
`2b` *(ucs-2)*	`uint16_t*`	`STRING`
`4b` *(ucs-4)*	`uint32_t*`	`STRING`
typ zařízení	*zařízení*	
`file` *(při **1b** znacích)*	`FILE*`	`HANDLER`
`file` *(při **2b** a **4b** znacích)*	`FILE*,int`	`HANDLER [,INT]`
`mem`	*řetězec*`, unsigned int`	`STRING`

Tabulka 3.3: Typy závislé na zařízení a kódování znaků.

Příklad 3.1 Jednoduché použití rozhraní v symbolické podobě.

```
/* název modulu je "prog" */
state = PROG_S_begin; /* stav jménem "begin" */
fopen(file,"soubor.dat","r");
if (!file)
{
  printf("Bad file.\n");
  halt();
}
I → ASSING(file);

while ( I → END() == 0 && I → ERROR() == 0 )
{
  printf("Simulace začne ve stavu č.%i\n",state);
  state = S → RUN(&I,state,dataptr);
  if (state == PROG_STATE_NONE) /* nedefinovaný stav */
  {
    printf("Špatná syntaxe vstupu.\n");
    break;
  }
  printf("Simulace ukončena ve stavu č.%i\n",state);
}

if ( I → END() != 0 )
  printf("Konec vstupu.\n");
if ( I → ERROR() != 0 )
  printf("Chyba vstupu.\n");
```

argumentů a návratových hodnot v konvencích daného jazyka. Vyjímkou jsou typy *řetězec*, *zařízení* a *uživatelská data*. Typ *uživatelských dat* je dán vstupním zdrojovým kódem a typy *řetězce* a *zařízení* jsou závislé na vstupním zařízení a kódování znakůi Jsou uvedeny v tabulce 3.3.

Všimněme si různosti symbolu pro zařízení typu soubor v případě jedno a více bytové reprezentace znaků. Souvisí to s pořadím bytů při ukládání znaků na diski. Část argumentu je číslo určující způsob, jakým se bude pořadí bytů vyhodnocovat. Číslo menší než 0 značí reprezentaci little-endian[1] a číslo větší než 0 indikuje reprezentaci big-endian[2]. Je-li číslo rovné 0 (nebo nedefinované), bude se na začátku vstupu hledat tzv. Byte Order Mark (BOM)[15]. BOM je znak nedělitelné mezery nulové délky, který se nikdy nesmí vyskytovat

[1]První byte při čtení zleva doprava má nejnižší hodnotu.

[2]První byte při čtená zleva do prava má nejvyšší hodnotu.

na začátku vstupu jako normální textový znak. Na základě tvaru BOM se automaticky určí pořadí bytů v zařízení. V případě, že na začátku vstupu BOM není, budou se byty vyhodnocovat v pořadí přirozeném pro danou architekturu (tzn. tak, jak jsou z pohledu operačního systému).

V objektovém modelu musí být typ uživatelských dat třída nebo reference na třídu. Důvodem je volání procedur jako metod objektu představujícího uživatelská data. V jazyce C jsou procedury reprezentovány globálními funkcemi. Tabulka 3.2 ukazuje specifikaci funkcí resp. metod, které zastupují tázací, přechodové a ladící procedury.

Kapitola 4

Implementace programu AutoComp

V této kapitole se budeme zabývat programovým řešením nástroje, provedeme krátký přehled struktury programu a dále se budeme věnovat zajímavým nebo důležitým částem.

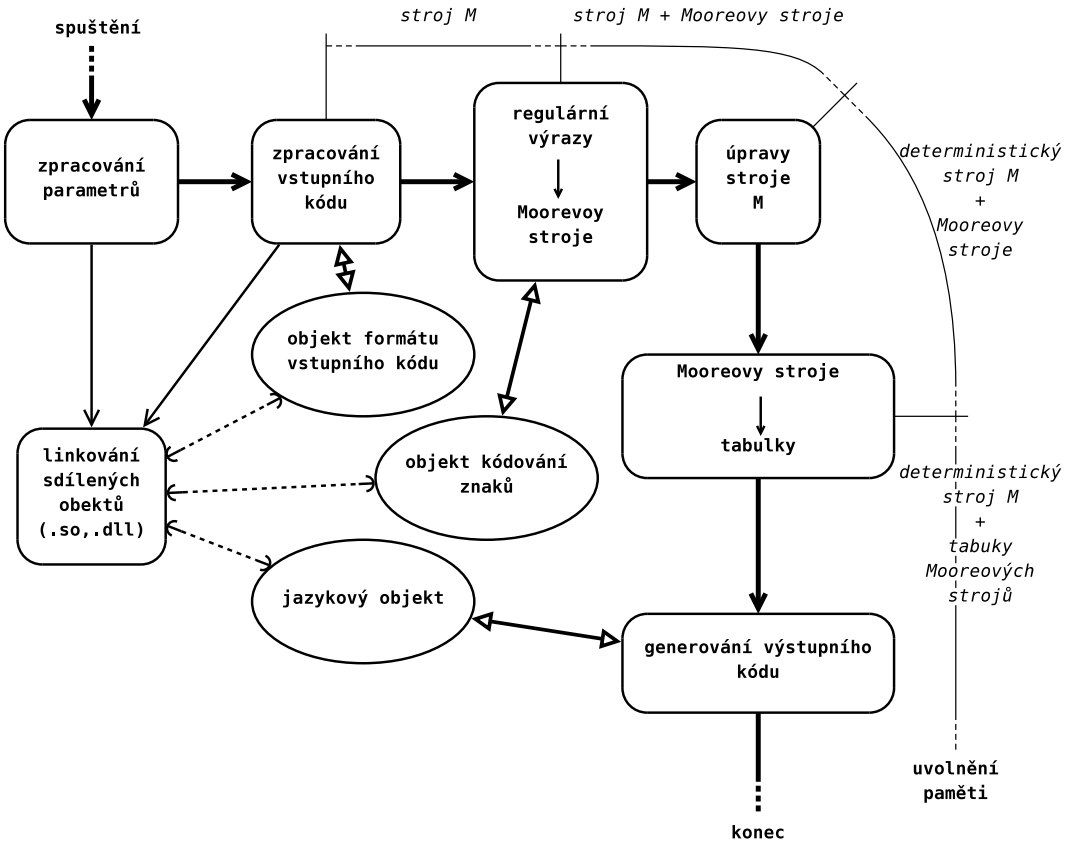
4.1 Přehled modulů a sestavení

Návrh programu AutoComp počítá s velkou variabilitou formátů vstupu i výstupu. Je proto rozdělen do čtyř částí, které jsou tvořeny hlavním programem, objekty pro načítání zdrojového kódu, objekty pro znakové sady a jazykové objekty. Objekty pro načítání zdrojového kódu jsou v instalaci přítomny dva. Jeden načítá vstupní data ve vnitřním formátu[1] a je přilinkován k hlavnímu programu v době kompilace. Druhý je linkován za běhu a umožňuje načítání dat v základním formátu. Objekty reprezentující znakové sady jsou linkovány za běhu programu AutoComp. Jedné znakové sadě odpovídá jeden objekt. Pro spolupráci s dalšími objekty je podstatné, kolik bytů je použito pro uložení jednoho znaku. Z tohoto hlediska program podporuje čtyři druhy: kódování 1-bytové (ascii), 2-bytové (ucs2), 4-bytové (ucs4) a s proměnlivou šířkou znaku (utf8). Jazykové objekty jsou též linkovány za běhu a poskytují data a funkce důležité pro generování programu v konkrétním programovacím jazyce a při konkrétním vstupním zařízení. Jednomu programovacímu jazyku odpovídá jeden jazykový objekt.

Hlavní program je tvořen řadou modulů, které v zásadě korespondují s jednotlivými úkony programu, ty jsou znázorněny na obrázku 4.1. Hlavní program zpracuje parametry příkazové řádky pomocí modulu `commandline`. Modul `commandline` používá volání *getopt()*[2]. Volby programu dovolují určit formát zadání. Podle toho je nalinkován sdílený objekt pro načítání zadání (případně použit staticky linkovaný objekt pro vnitřní formát). Rozhraní objektu je tvořeno jednou funkcí, která načte vstupní soubor a naplní datové struktury. Na základě načtených dat se nalinkuje objekt reprezentující znakovou sadu,

[1] Vnitřní formát není zmiňován v uživatelské dokumentaci jelikož primárně neslouží uživateli. Jeho popis lze nalézt v generované referenční příručne na CD (viz Příloha A).

[2] Součást GNU C library; URL:http://www.gnu.org/software/libtool/manual/libc/Getopt.html



Obrázek 4.1: Schema běhu programu AutoComp.

který bude použit při převodu regulárních výrazů na Mooreovy stroje modulem `regexs`. Po té se provedou úpravy zadaného stroje spojené s jeho determinisací a minimalizací, následně jsou minimalizovány i Mooreovy stroje. Před generováním kódu je třeba převést Mooreovy stroje do tvaru tzv. tabulek, úspornější reprezentace Mooreových strojů, která logickou strukturou odpovídá jejich výsledné podobě v kódu. V poslední fázi je generován výstupní kód modulem `generator`. Kód je generován ve spolupráci s jazykovým objektem.

Linkování sdílených objektů za běhu programu zprostředkovává modul `dlinking`, který využívá odpovídající systémová volání. Modul `commandline` kromě zpracování parametrů příkazové řádky též obsluhuje otevírání a zavírání souborů, do kterých bude generován výstupní kód. Modul `reading` načítá vstupní kód pomocí funkce z odpovídajícího sdíleného objektu nebo sám o sobě v případě vnitřního formátu. Modul `generator` zastřešuje další moduly jak pro práci s regulárními výrazy (`regexs`), tabulkami Mooreových strojů (`gen_tables`), úpravu zadaného kódu (`ua_mods`), tak pro generování výsledného kódu (`gen_format`). Modul `structures` obsahuje funkce pro alokaci, inicializaci a uvolnění hlavních datových struktur. Koordinaci všech modulů řídí `main.c`, obasahuje hlavní funkci *main()*.

4.2 Datové struktury

Základní datová struktura, kterou program používá, je binární strom (`typedef tree`[3]). Je to polymorfní datová struktura, obsahuje ukazatel na funkci pro porovnání prvků, funkci, která sjednotí prvky v případě duplicity, a funkci, která uvolní paměť alokovanou prvkem. Prvky jsou reprezentovány ukazateli. Tato struktura umožňuje vložit unikátní prvek, odebrat ekvivalentní prvek a najít ekvivalentní prvek. Uspořádání je dáno právě funkcí pro porovnání prvků. Vložení duplikátního prvku není možné. V případě duplicity se volá funkce pro sjedocení prvků. Funkce sjednocující prvky nesmí ovlivnit uspořádání prvku již přítomného ve stromu. Strom je použit pro reprezentaci široké škály objektů, od množin po zásobníky.

Datová struktura pro Mooreův stroj (`typedef regex_automata`[4]) je tvořena množinou stavů, počátečních stavů, koncových stavů, přechodových hran a prvků abecedy. Množiny jsou reprezentovány pomocí stromů. Stav obsahuje strom ukazatelů na hrany, které do něj vstupují, a strom ukazatelů na hrany, které z něj vystupují, dále množinu výstupních symbolů, což jsou ukazatele na přechodové hrany zadaného stroje. Hrana je tvořena ukazatelem na svůj počáteční a cílový stav a prvkem abecedy. Aabeceda Mooreova stroje je určena znakovou sadou. Za účelem snížení počtu hran je abeceda rozdělena na skupiny znaků. Stavy, přechodové hrany i prvky abecedy obsahují navíc celočíselnou položku `self_index`, která slouží k očíslování pro potřeby převodu do tabulek.

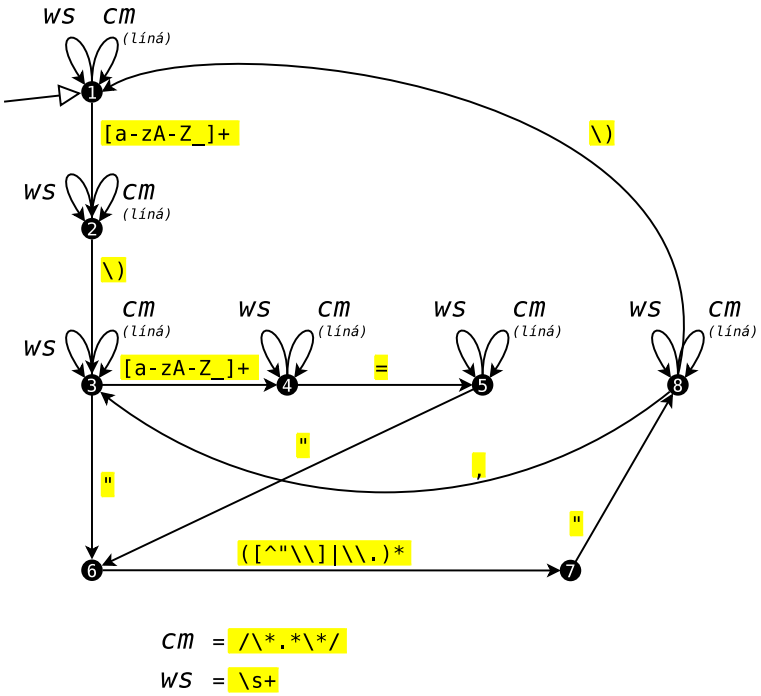
Tabulky (`typedef regex_automata_table`[5]) jsou tvořeny stromem Mooreových strojů, abeced, výstupů a podmnožin výstupních symbolů odpovídajících jednotlivým stavům. Výstupy mapují koncové stavy Mooreových strojů v závislosti na stavu zadaného stroje na množiny výstupních symbolů. Datová struktura pro tabulky dále udržuje velikosti jednotlivých množin s ohledem na jejich pozdější výpis do podoby polí. Pole vytvořená na základě tabulek jsou pak součástí vygenerovaného kódu.

Reprezentace zadaného stroje, stroje M, je logicky podobná reprezentaci Mooreova stroje. Stavy však navíc obsahují ukazatel na Mooreův stroj a jeho startovní stav, dále ukazatel na abecedu Mooreova stroje v neposlední řadě jména tázací a ladících procedur. Přechodové hrany mimo ukazatale na počáteční a cílový stav obsahují textovou reprezentaci regulárního výrazu. Struktura `automaton` obsahuje kromě reprezentace stroje M ve formě stromů stavů a přechodových hran také položky týkající se specifikace výstupního modulu, např. jméno modulu a souborů s procedurami.

4.3 Zpracování zdrojového kódu

Nástroj AutoComp podporuje dva formáty zdrojového kódu; základní a vnitřní. Vnitřní formát je velmi pozičně orientován, jeho části jsou odděleny prázdnými řádkami a neposkytuje žádny prostor pro komentáře, mohl být použit pro strojové generování vstupního

[3] soubor `shared/tree_typedefs.h`
[4] soubor `regexs/regex_automata.h`
[5] soubor `genrator/regex_tables.h`



Obrázek 4.2: Schema stroje pro parser základního formátu kódu.

kódu. Vnitřní formát hrál však důležitou roli během vývoje nástroje. Předně v úplném počátku se s ním počítalo jako s jediným formátem, ovšem záhy ustoupil pro člověka čitelnějšímu a přehlednějšímu základnímu formátu. Později pak posloužil při tvorbě parseru základního formátu. Tento parser byl totiž vytvořen pomocí AutoCompu a jeho zdrojový kód byl napsán ve vnitřním formátu. Zdrojový kód parseru základního formátu ve vnitřním formátu, ale i jeho ekvivalent v základním formátu, jsou k nahlédnutí na přiloženém CD (Příloha A). Poznamenejme, že obsahové možnosti obou formátů jsou ekvivalentní, tedy to, co lze napsat v jednom formátu, lze napsat „jinými slovy“ i v druhém formátu.

Schematické vyobrazení stroje pro parser vstupního zadání je na obrázku 4.2. Přechodová procedura na hraně $1 \rightarrow 2$ určí číslo příkazu podle přečteného řetězce, případně alokuje a inicializuje proměnnou reprezentující stav nebo přechodovou hranu nového stroje M. Číslo pojmenovaného parametru nastaví na -1 a vynuluje bitovou masku již přiřazených parametrů. Při načtení jména parametrů nastaví přechodová procedura hrany $3 \rightarrow 4$ číslo pojmenovaného parametru na hodnotu odpovídající jménu parametru. Přechodová procedura na hraně $6 \rightarrow 7$ zpracovává načtený text jako hodnotu položky stroje, k určení konkrétní položky použije číslo parametru nebo první neinicializovaný parametr a číslo příkazu. Nejčastěji se vytvoří kopie přečteného řetězce a ta je přiřazena položce stroje, příkladem buď jméno stavu. Dále upraví bitovou masku přiřazených parametrů, případně vypíše varování, že byl parametr přiřazen opakovaně. Při přechodu po hraně $8 \rightarrow 3$ od-

povídající přechodová procedura pouze nastaví číslo pojmenovaného parametru na -1. Přechodová procedura na hraně $8 \rightarrow 1$ začlení hranu nebo stav do stroje, vloží objekt do odpovídajícího stromu. V případě, že objekt je již přítomen ve stromu, upraví odlišné položky a vypíše varování.

Zdrojový kód přechodových procedur parseru spolu s vygenerovaným kódem a dalšími funkcemi rozhraní pro použití parseru v nástroji tvoří modul `generated_reading`. Tento modul je zkompilován do sdíleného objektu `autocomp.rdr.impl.so` (nebo `.dll` v OS Windows). Objekt je v případě potřeby přilinkován za běhu a použit modulem `reading`, který zastřešuje načtení zdrojového kódu nezávisle na formátu. Parser vnitřního formátu byl naprogramován ručně a je součástí modulu `reading`.

Rozhraní mezi hlavním programem a separátním objektem pro načítání vstupního kódu je tvořeno jednou funkcí. Parametrem této funkce je mimo jiné struktura, která obsahuje jméno souboru se zdrojovým kódem a ukazatel na inicializovanou proměnnou reprezentující stroj M. Jelikož parser bude potřebovat vytvořit předem neznámé množství stavů a přechodových hran stroje, jsou součástí parametrů i ukazatele na funkce, které alokují paměť a inicializují stav či hranu. Odpovědnost za uvolnění alokované paměti, která je využita pro uložení částí stroje, přebírá hlavní program. Z toho důvodu je nutné aby pro alokaci byly použity výhradně funkce předané jako parametry.[6]

4.4 Objekty pro podporu kódování znaků

Program AutoComp na základě specifikace znakové sady a kódování znaků ve vstupním zdrojovém kódu přilinkuje sdílený objekt, který podporuje danou konfiguraci. Objekt se linkuje na základě řetězce, jenž ve zdrojovém kódu identifikuje kódování znaků a sadu. Jedná se o parametr `enc` příkazu `input`. Je-li tento řetězec např. `"ascii"`, hledá se objekt jménem `autocomp.enc.ascii.so` (v OS Windows `.dll`). Základní instalace obsahuje několik objektů sloužících k tomuto účelu. Přidáním dalšího objektu jménem např. `autocomp.enc.latin2.so` se tedy automaticky otevře možnost použít kódování `"latin2"`.

Přilinkovaný objekt poskytuje programu nástroj pro převod znaků a předdefinované rozsahy znaků (viz Příloha B), potřebné pro interpretaci regulárních výrazů. Předdefinovanému rozsahu je v objektu přiřazen znak, který zastupuje v regulárním výrazu tento rozsah, následuje-li za znakem `'\'`. Některé předdefinované rozsahy je nutné poskytovat (`\n`), jiné lze doporučit (`\w,\s,\W,\r` apod.). Zásadně by však měly tyto předdefinované rozsahy a jejich jména být v souladu s běžně používanými rozsahy ve známých aplikacích (např. Perl).

Vstupním bodem objektu je funkce `get_encoding_table( void)`. Návratovou hodnotou je ukazatel na konstantní strukturu `encoding_table`. Následující seznam popisuje jednotlivé položky `encoding_table`.

`convert_fnc` je ukazatel na funkci typu `uint32_t` *convert*`(uint32_t ucs_codepoint)`,

[6] Pamět musí být alokována i uvolněna jedním objektem, funkce pro alokaci jsou součástí objektu hlavího programu.

identifkátor	šířka znaku	příklady
ENCODING_TYPE_1BYTE	1b (7-8 bitů)	ascii, iso8859-2
ENCODING_TYPE_2BYTE	2b (16 bitů)	ucs2, utf-16
ENCODING_TYPE_VARBYTE	1-6 oktetů	utf-8
ENCODING_TYPE_4BYTE	4b (32 bitů)	ucs4

Tabulka 4.1: Druhy kódování.

která převede kódový bod ze sady UCS do podporovaného sady. Vrací číslo znaku v požadovaném kódování a znakové tabulce.

`table` je ukazatel na pole konstantních množin znaků (`typedef char_union`), které reprezentují předdefinované rozsahy.

`size` udává počet prvků pole `table`. Velikosti následujících polí jsou odvislá od tohoto čísla.

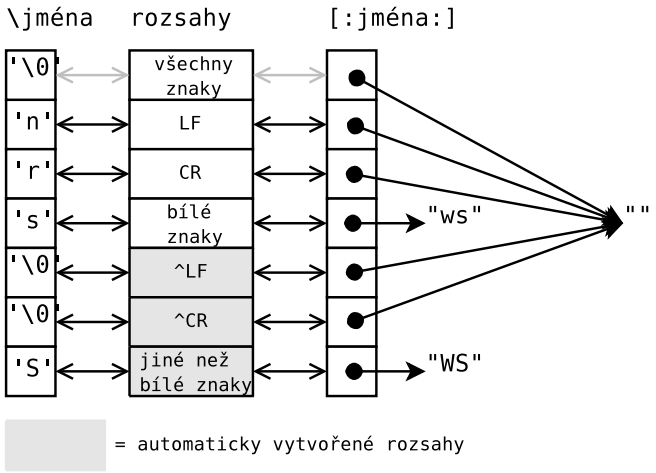
`backslash_rules` je ukazatel na pole konstantních 4-bytových znaků, které určují zástupce předdefinovaných rozsahů v regulárním výrazu ve tvaru `\X`. Je zde vhodné používat znaky anglické abecedy. V žadném případě nelze doporučit používání metaznaků.

`colon_rules` ukazuje na pole konstantních řetězců v UCS4, které určují zástupce předdefinovaných rozsahů v regulárním výrazu ve tvaru `:jméno:`. Tato forma použití předdefinovaných rozsahů není zatím AutoCompem podporována. Vzhledem k tomu, že se ovšem s podporou do budocna počítá, je vhodné ji zde již zahrnout.

`newline_regex` je regulární výraz, který nahradí výsktyt `\n` mimo `[ ]`. Doporučeno je `\r?\n`, pokud je objektem definován rozsah `\r`. Výskyt `\n` v regulárním výrazu vzniklém touto substitucí nebo v `[ ]` se již reprezentuje jako běžný předdefinovaný rozsah.

`encoding_class` identifikuje druh kódování. Druhy se vzájemně liší zejména šířkou znaku. Tabulka 4.1 ukazuje definované druhy. Tato položka je podstatná pro jazykový objekt a generátor kódu, které podle ní generují části kódu specifické pro interpretaci bytů ze vstupního zařízení.

První položka pole rozsahů (`table`) musí být rozsah všech znaků, je automaticky zastupován metaznakem `'.'`. První položka pole `backslash_rules` (resp. `colon_rules`) by tedy měl být nulový znak (resp. prázdný řetězec). Pro velikosti polí musí platit, že velikosti polí jmen (`backslash_rules` a `colon_rules`) jsou o jedna menší než je dvojnásobek velikosti pole rozsahů (`table`). Důvodem je, že AutoComp virtuálně „prodlouží“ pole rozsahů a nově vzniklé položky naplní doplňky daných rozsahů (s vyjímkou rozsahu všech znaků). Vzdálenost mezi rozsahem a jeho doplňkem je vždy `size`-1. Vztah mezi položkami polí je



Obrázek 4.3: Schema systému indexace polí rozsahů a jmen.

dán tím, že rozsah znaků v i-té položce pole rozsahů je zastupován znakem v i-té položce pole znaků a jménem i-té položce pole řetězců. Obrázek 4.3 ilustruje tuto závislost.

Jen poznamenejme, že sdílený objekt pro kódování by neměl alokovat žádnou paměť a každé volání musí vracet ukazatel na naprosto stejnou strukturu. Jakmile je již jednou struktura inicializována, její obsah nesmí být změněn po dobu existence objektu v paměti. Tohoto lze dosáhnout pomocí statické inicializace konstantí struktury, tak je provedeno v implementvaných objektech. Jinou možností je nastavení statické proměnné, která indikuje, že již byla struktura inicializována. V takovém případě může být nezbytné použití jednoduchého synchronizačního primitiva nebo vyloučení konfliktu pomocí atomické operace.

4.5 Převod regulárních výrazů na Mooreovy stroje

Cílem je převést regulární výrazy odpovídající přechodovým hranám vedoucích z jednoho stavu stroje na jeden Mooreův stroj, který bude přijímat sjednocení regulárních jazyků odpovídajících regulárním výrazům. Na koncových stavech Mooreova stroje musí být symboly identifikující regulární jazyk, do kterého patří vstupní slovo. Protože regulární jazyk je přiřazen přechodové hraně, budeme regulární jazyk identifikovat prostřednícvím přechodové hrany. Je zřejmé, že jeden koncový stav může nést více symbolů. Z toho důvodu je za množinu výstupních symbolů zvolen systém podmnožin množiny všech přechodových hran zadaného stroje. Regulární výrazy jsou v textové podobě a výsledný Mooreův stroj je reprezentován již popsanou datovou strukturou. Prvním krokem je převedení regulárních výrazů z textové podoby do podoby binárního stromu reprezentujícího výraz (dekompozice). Následuje určení abecedy Mooreova stroje, nakonec se vytvoří Mooreoův stroj.

Dekompozice regulárních výrazů je provedena pro každý výraz zvlášť, standartními

technikami a v lineární časové složitosti vůči délce výrazu. Stromy odpovídající jednotlivým regulárním výrazům jsou následně spojeny operací „or“ do jednoho stromu odpovídajícího jednomu stavu stroje „M“. List stromu obsahuje množinu znaků, ostatní uzly obsahují informaci o odpovídajícím operátoru. Uzel stromu, který je kořenem stromu představujícího dekompozici jednoho regulárního výrazu nese navíc identifikaci přechodové hrany. Ta poslouží při přiřazování výstupních symbolů stavům Mooreova stroje.

Rozsah znaků je podmnožina celé znakové sady, která může být velmi rozsáhlá, může dosahovat velikosti 2^{31} znaků. Vytvářet Mooreův stroj, kde každá přechodová hrana bude odpovídat jednomu znaku by bylo značně náročné na velikost automatu. Generováné tabulky takového stroje by dosahovaly neúnosných rozměrů. Efektivnější je vytvořit disjunktní pokrytí množin znaků v listech. Je ovšem nutné si v listech udržet informaci o tom, jaké prvky pokrývají množinu znaků, kterou list reprezentuje. Prvky pokrytí budou tvořit abecedu Mooreova stroje, čímž se výrazně sníží počet jeho hran.

Konstrukce nedeterministického Mooreova stroje z binárního stromu je pak provedena průchodem stromu v pořadí *postorder*. V listu stromu se vytvoří jednoduchý Mooreův stroj, který obsahuje pouze počáteční a koncový stav. Hrany, které stavy spujují, odpovídají pokrytí množiny znaků reprezentovanou listem. V ostatních uzlech se nejdříve vytvoří Mooreovy stroje pro podstromy. Následně se použijí jako operandy regulární operace dané vrcholem. Způsob aplikace regulárních operací na konečné automaty lze nalézt například v důkazu uzavřenosti jazyků konečných automatů na regulární operace [1]. Pokud uzel obsahuje výstupní symbol (uzel je kořen dekompozice jednoho regulárního výrazu), přiřadí se tento výstupní symbol všem koncovým stavům stroje, který vznikl v tomto uzlu. Popsaný postup principiálně odpovídá tzv. Thompsonově konstrukci konečného automatu[11].

Pro konstrukci deterministického Mooreova stroje ekvivalentního s nedeterministickým Mooreovým strojem je použit mechanismus *podmnožinové konstukrce* [2]. Je ovšem nutné speciálně ošetřit výstupní symboly stavů. Stav t zkonstruovaný z množiny stavů T bude mít výstupní symbol $Sym(t) := \bigcup_{s \in T} Sym(s)$.

4.6 Úpravy zadaného stroje

V oddíle 3.5 byla popsána idea determinisace zadaného stroje (algoritmus 3.4). Nyní se na věc podíváme z pohledu implemetnace. Připomeňme si nejprve datovou strukturu `automaton`, která reprezentuje stroj a další součásti kódu. Její položky podstatné pro reprezentaci stroje popisuje následující seznam.

`states` reprezentuje množinu stavů pomocí stromu.

`starting states` představuje množinu startovních stavů.

`edges` reprezentuje přechodovou funkci stroje formou stromu přechodových hran.

Jednotlivé stavy obsahují mimo jiné ukazatel na Mooreův stroj, který tyto hrany reprezentuje. Koncové stavy Mooreova stroje obsahují ukazatele na přechodové hrany Ty nesou mimo jiné informace o procedurách, hladovosti a ukazatel na stavy, jež hrana spojuje.

Klíčovým bodem algorimtu je určení sjednocených stavů na základě Mooreova stroje. Provádí se to průchodem přes všechny koncové stavy Mooreova stroje, přičemž v každém stavu se prochází všechny výstupní symboly. Výstupní symboly jsou ukazatele na přechodové hrany. Přechodové hrany bez přechodové procedury jsou po průchodu rozděleny se do dvou skupin, hladových hran a líných hran. Obsahuje-li skupina více než jednu hranu, znamená to, že se pro ni vytvoří sjednocený stav. Ten bude identifikován množinou všech cílových stavů dané skupiny hran. Na základě této identifikace se pak unikátně vkládá do kontejnerů, pokud by byl duplikátem, je stav zahozen.

Celý průběh determinisace je implementován pomocí dvou kontejnerů[7] , kde první obsahuje nově vytvořené stavy, které čekají na předzpracování, druhý sjednocené stavy, které jsou již připraveny k dokončení.

Prvním krokem algoritmu je průchod všech stavů stroje, vytvoření torz sjednocených stavů a jejich unikátní vložení do prvního kontejneru.

Ve druhém kroku se předzpracují torza sjednocených stavů; jsou jim vytvořeny deterministické Mooreovy stroje a jsou korektně připojeny do stroje pomocí hran. Mooreovy stroje jsou vytvořeny pomocí operace „or“ na Mooreových strojích zastřešených stavů. Výstupní symboly jsou zachovány, čímž se v podstatě vytvoří mapování přechodových hran na původní. Předzpracované stavy se přesunou z prvního kontejneru do druhého.

Třetí krok je podobný prvnímu s tím rozdílem, že se prochází druhý kontejner a sjednocené stavy jsou brány v potaz jen v případě, že v druhém kontejneru neexistuje stav, který zastřešuje stejné stavy zadaného stroje. Nové sjednocené stavy jsou pak opět vloženy do prvního kontejneru. Druhý a třetí krok jsou prováděny v cyklu, dokud není po třetím kroku první kontejner prázdný.

V závěrečné fázi se pojmenují nově vytvořené stavy a jsou vypsána varovná hlášení na chybový výstup. Sjednocené jsou označeny za startovní a koncové podle následujících pravidel. Sjednocený stav je označen za koncový, pokud je koncový alespoň jeden ze stavů, které jsou jím zastřešeny. Sjednocený stav je pak označen za startovní, pokud je koncový. Dále se minimalizují Mooreovy stroje na nově vytvořených stavech.

Po determisaci se provádějí operace za cílem minimalizace zadaného stroje. Minimalizace je založena na odstranění nedosažitelných stavů a přebytečných hran, není možné vylučovat ekvivalentní stavy. Předně ekvivalence mezi stavy je jen těžko definovatelná, neboť význam stavů je spolutvořen celou výslednou aplikací. Následně uživatel může vyžadovat spuštění simulace stroje i ze stavů, které by byly odstraněny v rámci nějaké ekvivalence. Nedosažitelné stavy mohou být okrajově způsobeny poněkud nepromyšleným zadáním, pokud v něm existuje stav, do kterého nevede hrana. Hlavním zdrojem nedosažitelných stavů je však provedená determinisace stroje. Nedosažitelný stav tady vznikne úplným nahrazením stavu sjednocenými stavy, tedy pokud byly všechny přechody do něj „nedeterministické“. Vyhledání nedosažitelných stavů se provádí průchodem stroje do šířky ze všech stavů označených jako startovní. Průchod se provádí analogicky s průchodem orientovaného grafu do šířky.

[7]Kontejnery jsou implementované binárními stromy, typ `tree`.

Odstranění přebytečných hran se provádí po odstranění nedosažitelných stavů. Jsou jednoduše odstraněny ty hrany, které skutečně nemohou být použity, tedy ty, které se nevyskytují jako výstupní symbol žádného z koncových stavů všech Mooreových strojů.

4.7 Generování zdrojového kódu

Pro generování zdrojových kódů byla navržena technika, ve které se využívá šablon čili předloh kódu, v níž se vyskytují značky ve tvaru `%%z#%%`[8]. Šablona je kopírována na výstup s vyjímkou značek, které jsou nahrazeny jinými částmi kódu. Pointa tohoto řešení spočívá v obsažení neměnných částí kódu, jichž je většina, textem šablon. Naproti tomu části závislé na zadaném stroji či parametrech jsou generovány pomocí značek. V původním plánu bylo zamýšleno generovat zdrojový kód po lexikálních elemetnech, což se však ukázalo jako nesmírně obtížný úkol vzhledem k cíli obsáhnout více programovacích jazyků. Přestože mají programovací jazyky často velmi podobnou strukturu, mohla by být využita jen společná podmnožina výrazových prostředků nabízených jazyky. Tím by se velmi omezila možnost generovat kód využívající specifik jazyka a čitelný pro programátora. Naopak v systému šablon by se dalo uvažovat i o podpoře neprocedurálních jazyků, kteréžto mívají výrazně odlišnou strukturu od procedurálních. Je však nutné poznamenat, že šablony i samotný princip reprezentace stroje přinášejí též svá omezení, která jsou patrná zejména v jazyce Perl, kde nebylo možné využít nativních prostředků pro práci s regulárními výrazy.

Značky podle svého druhu mohou plnit funkci zástupnce nějakého symbolu, ať už jde o klíčové slovo, typ či hodnotu proměnné. Nabízejí též možnost zanoření, tedy substituci jiné šablony za značku. Podstatnou složku tvoří tzv. iterační značky, které zastupují sekvenční výpis prvků nějaké struktury. Pro představu uveďme jako příklad výpis přechodové funkce Mooreova stroje jako matice. Ve výsledném kódu tento výpis představuje statickou inicializaci pole. Výpis obsahu iterační značky je prováděn pomocí funkce, která je volána iterativně přes odpovídající prvky. Pro implemntaci volitelného ladícího režimu byly použity párová značka, která umožňuje vynechat kód mezi jejím lichým a sudým výskytem.

Struktura značky je velmi jednoduchá. Je tvořena úvodní sekvencí `%%` následovanou jedním znakem, který identifikuje druh značky. Iterační značky zastupuje znak `i`, pro značky zastupující šablonu je to znak `t` a pro značky nesoucí jednoduchý symbol znaky `a`, `b`, `e`, `n` a `s`. Za znakem následuje číslo, které identifikuje konkrétní symbol, šablonu či iteraci. Symboly, šablony a ukazatele na iterační funkce jsou uloženy v polích. Zjednodušeně lze tedy říci, že znak určuje pole a číslo udává index do pole. Za číslem značky následuje libovolný text sloužící jako komentář, který se neobjeví ve výsledném kódu. Značku ukončuje sekvence `%%`. Párová značka se poněkud liší, má totiž pevný tvar `%%*%%`.

Hlavní program přilinkuje podle specifikovaného programovacího jazyka sdílený objekt, který poskytuje soustavu šablon a některá pole symbolů. Princip výběru objektu je podobný jako u objektů kódování; je-li příkladně specifikován jayzk `"perl"`, linkuje se objekt `autocomo.lan.perl.so`. Vstupní bod jazykového objektu tvoří funkce

[8] `z` zasupuje znak, `#` zastupuje číslo

`get_language_code_table( )`. Její argumenty mimo jiné určují typ zařízení a druh kódování. Funkce naplní strukturu reprezentující šablony a pole symbolů v závislosti na argumentech a vrátí ukazatel na ni. Součástí struktury je i specifikace počtu generovaných souborů a jejich přípon. Každý soubor má odpovídající šablonu. Hlavní program tedy otevře odpovídající soubory. Samotné generování kódu pak spočívá pouze v substituci symbolů a částí kódu za značky. Při tom část symbolů je poskytována strukturou získanou z jazykového objektu, část hlavním programem. Blíže jde o symboly přímo závislé na stroji nebo specifikaci modulu ze zdrojového kódu. U iteračních značek se použije odpovídající funkce daná jazykovým objektem volaná postupně na prvky datové struktury dané hlavním programem, zejména jde o tabulky `regex_automata_table`. Generování probíhá v modulu `generator/gen_format.c` včetně výběru pole symbolů na základě značek. Tento modul také zprostředkovává cyklická volání iteračních funkcí.

4.8 Jazykové objekty

Podobně jako objekty pro kódování znaků, jsou i jazykové objekty linkovány za běhu programu AutoComp a určují podporované programovací jazyky. Vstupním bodem jazykového objektu je následující funkce.

```
code_table*
get_language_code_table(
    const unsigned char* itype
   ,int encode
   ,int* errcode)
```

Argument `itype` je řetězec identifikátoru vstupního zařízení. `encode` je identifikátor druhu kódování (viz 4.1). V případě, že funkce neproběhla správně, nastavuje kód chyby do `errcode`.

Datová struktura typu `code_table` obsahuje pole šablon, symbolů kódu, ukazatelů na iterační funkce a přípon generovaných souborů. Popis jednotlivých položek stejně jako interakce mezi nimi lze nalézt v generovaném referenčním manuálu na přiloženém CD (Příloha A). Zde si pouze objasníme některé zajímavosti reprezentace a implementace. Pro případného implementátora dalších sdílených objektů může být inspirativní náhled do souborů `shared/code_table_type.h` a `dlibs/language_c.c`. První obsahuje definici datového typu a důležitých konstant, druhý jmenovaný soubor pak implementaci sdíleného objektu pro jazyk C.

Jedna šablona je reprezentována polem řetězců místo jedním řetězcem. Šablona může být poměrně dlouhý řetězec (řádově stovky řádků kódu), ale norma jazyka C specifikuje, že překladač musí být schopen korektně zpracovat řetězec o něco málo delší než 500 znaků. Odtud plyne důvod rozdělení šablony do pole řetězců. Při tvorbě sdílených objektů, které jsou součástí instalace, byla každá šablona napsána jako samostatný soubor a před kompilací byla pomocí jednoduchého skriptu a nástoje *gawk*[7] převedena do tvaru statické

inicializace pole řetězcových konstant v jazyce C. Zdrojové kódy na CD (Příloha A) obsahují soubory se šablonami v původní podobě v adresáři `dlibs/templates/source/`. Jejich převod na zdrojové kódy v jazyce C byl však předem proveden.

Před uvolněním nalinkovaného jazykového objektu je volána funkce `void free_language_code_table(code_table* t)`, která má uvolnit případnou alokovanou paměť. Funkce je rovněž součástí sdíleného objektu.

Kapitola 5

Závěr

Závěrem shrneme nejzajímavější problémy řešené v práci. Zmíníme také nejdůležitější momenty, které ovlivnily vývoj programu AutoComp a navrhneme některá další rozšíření.

5.1 Řešené problémy

Zajímavým problémem byl samotný návrh rozšiřitelnosti programu o další programovací jazyky a znakové sady. Bylo zde využito dynamicky linkovaných objektů (knihoven), které reprezentují jednotlivé jazyky a znakové sady. Rozšíření se pak provede implementací sdíleného objektu. Jeho linkování za běhu je řízeno zadaným jménem kódování, jazyka nebo vstupního zařízení.

Pro generování výstupního zdrojového kódu byl vytvořen systém šablon a značek, které se zastupují na variabilní části kódu a tabulky automatů. Systém by měl vyhovovat generování kódu pro většinu procedurálních programovacích jazyků, je možné uvažovat i o neprocedurálních jazycích. Je však orientován na implementaci konečných automatů v podobě tabulek a bez úprav by se dal těžko použít pro přímou reprezentaci.

Dalším zajímavým bodem byl také problém determinisace stroje popsaného vstupním kódem. Zde bylo potřeba vytvořit transparentní přechod mezi deterministickým a nedeterministickým strojem. Toho bylo dosaženo mapováním přechodových hran na nově vzniklé stavy a přechodové hrany.

Obohacující pro autora byla také implementace algoritmů pro převod regulárních výrazů na konečné automaty. Je to sice dobře známá oblast, ale přinesla řadu cenných zkušeností a představu o náročnosti implementace známých algoritmů v kontextu již rozvržených datových struktur. Zejména část týkající se vytvoření rozumné reprezentace abecedy pro automat nad rozsáhlou znakovou sadou se ukázala zajímavou. Část implementace týkající se regulárních výrazů je poměrně ucelená a nezávislá. Po několika úpravách a oddělení od zbytku nástroje může posloužit jako samostatná knihovna pro práci s regulárními výrazy a konečnými automaty.

5.2 Shrnutí vývoje nástroje

Během implementace nástroje prošel návrh celou řadou změn. Z tohoto hlediska nejdynamičtější byla oblast zpracování regulárních výrazů.

Zpočátku se předpokládalo použití knihovny, která by posloužila k vyhodnocování shody řetězce s regulárním výrazem. Tato knihovna by byla využívána vygenerovaným programem. Mezi základní požadavky na takovou knihovnu patřila možnost přenositelného uložení regulárního výrazu ve zkompilované podobě; bylo by neúnosné kompilovat regulární výrazy až při startu vygenerovaného programu. Dalším požadavkem byl určitý stupeň kontroly a přerušní průběhu vyhodnocování. Připočteme-li požadavek na více podporovaných jazyků, nebude snadné takovu knihovnu nalézt. Mezi nejznámější zkoumané kandidáty patří např. **PCRElib**[1] a modul **regex** GNU C knihovny [13]. Žádnou vhodnou knihovnu se však nepodařilo se nepodařilo nalézt.

V další fázi byl návrh změněn tak, že vyhodnocování shody řetězce s regulárním výrazem bude implementovat generovaný kód sám nad konečným automatem. Automat bude vytvořen během překladu AutoCompem a pro jeho tvorbu bude použita vhodná knihovna pro převod regulárních výrazů na konečný automat. Zde se ukázal problematickým požadavek, aby jeden automat mohl reprezentovat více regulárních výrazů a jeho koncový stav určoval, kterému regulárnímu výrazu řetězec náleží (tzv. transducer). Dalším požadavkem byla nezávislost automatu na kódování znaků nebo alespoň podpora *Unicode* nebo *UCS*. V tomto případě knihovny, které by se alespoň daly upravit pro zmíněné rozlišení koncových stavů, nepodporovaly *Unicode* a naopak. Ze zkoumaných knihoven zmiňme alespoň **hsre**[2], **xfa-lib**[3] a **libAMoRE**[4].

Posledním řešením tedy byla samostatná implementace převodu regulárních výrazů na konečné automaty, přesněji řečeno deterministické Mooreovy stroje. Autor musí připustit, že efektivita převodu a minimalizace nedosahuje nejlepších možných výsledků v tomto oboru. Byly použity poměrně letité algorimty spíše teoretického charakteru. Na omluvu ovšem podotkněme, že težiště práce neleží v perfektní implementaci nejlepších algoritmů pro zpracování regulárních výrazů.

Vzhledem k uvedenému problematičtějšímu vývoji bylo ustoupeno od podpory více různých programovacích jazyků a vstupních zařízení. V původním, řekněme velmi optimistickém plánu, bylo implementovat podporu jazyků C, C++, Java, PHP a Perl. V pozdější verzi specifikace se již počítalo pouze s podporou jazyků C,PHP a Perl. Ve výsledku došlo na podporu pouhých dvou jazyků: C a Perl a dvou typů vstupního zařízení: soubor a paměť. Byl implementován mechanismus přidání podpory dalších jazyků i s podporou dalších vstupních zařízení formou sdíleného obektu (dynamicky linkované knihovny). Jazyky C a Perl sice vycházejí ze stejného základu, ale programy v nich se velmi liší. V jazyce C byl implementován klasický způsob programování, kdežto v jazyce Perl se využívá objektově

[1] http://www.pcre.org
[2] http://www.straatinternet.com/opensource/hsre/
[3] http://www.xmailserver.org/xfa-lib.html
[4] http://amore.sourceforge.net

orientovaný přístup. Doufáme proto, že tyto dva jazyky postačí pro ilustraci rozšiřitelnosti programu AutoComp pro další jazyky, minimálně z rodiny procedurálních programovacích jazyků.

5.3 Možnosti dalšího rozšiřování

Modularita návrhu programu AutoComp dovoluje určitou úroveň rozšiřování programu pomocí implementace dalších dynamicky linkovaných objektů. Jde tedy o úpravy bez nutného zásahu do stávajícího kódu programu.

Definován je mechanismus rozšíření či úpravy podporovaných kódování. Zde by bylo optimální nejprve rozšířit podporu předdefinovaných rozsahů v již podporovaných kódováních a následně přidávat nová kódování. Je ovšem třeba dbát na to, aby nově definované rozsahy přinejmenším nekolidovaly s obecně zavedenými rozsahy z jazyka Perl či některých hojně používaných nástrojů pracujících s regulárními výrazy. Naopak ideální by bylo pokrytí takových obecně používaných rozsahů.

Rozšíření skupiny podporovaných jazyků lze provést přidáním jazykového objektu. Je možné nejen zavádět nové jazyky, ale i nová vstupní zařízení. Na místě je však větší obezřetnost, protože jazykový objekt může zásadně změnit chování generovaného kódu. Při implementaci sdíleného objektu pro nový programovací jazyk by měla být zahrnuta alespoň již podporovaná vstupní zařízení. V rozšiřování podporovaných vstupních zařízení se nabízejí dva možné směry, cesta specializace zařízení a nebo jejich zobecnění. Při zobecnění získává generovaný program možnost měnit za běhu i charakter zdroje dat, naopak při specializaci je možné dosahovat vyššího výkonu.

Poslední kategorií při rozšiřování bez zásahu do stávajícího kódu je přidání dalších formátů vstupu. Zavedení zcela nového formátu by mělo být velmi dobře odůvodněno, například napojením nástroje na další programy, které mu budou generovat zdrojový kód. V tomto ohledu by mohlo připadat v úvahu zavedení nějakého obecně známého jazyka, např. XML. Další myšlenkou je zavedení nových možností do základního formátu, příkladem může být hromadná deklarace více hran. Ty by se definovaly jedním příkazem, který by specifikoval jejich společné vlastnosti a dvojice stavů, jež hrany spojují. Zmiňme ještě myšlenku podpory maker v základním formátu vstupních dat.

Z dlouhodobějších a komplexnějších rozšíření bychom mohli uvést myšlenku napojení nástroje na grafické rozhraní s podporou sledování výpočtu generovaného programu. Idea řešení by spočívala v úpravě generovaného kódu tak, aby v určitých bodech výpočtu komunikoval s procesem grafického rozhraní např. pomocí socketů nebo roury. Takový „grafický režim“ vygenerovaného programu by pak měl být vypínatelný, podobně jako ladící režim.

Zajímavým námětem je také myšlenka implementace tzv. přímé reprezentace konečných automatů v generovaném kódu, která experimentálně vykázala větší výkonnost, než reprezentace tabulkou [3].

Literatura

[1] Aho, A.V. a Ullman J.D.: *The Theory of Parsing, Translation and Compiling; Volume 1: Parsing* (103–124), Prentice-Hall, USA, 1972

[2] Aho, A.V., Sethi R. a Ullman J.D.: *Compilers, Principles, Techniques and Tools* (117—121),Addison-Wesley, USA, 1986

[3] Brouwer, K., Gellrich, W. a Ploedereder, E.: *Myth and Facts about Efficient Implementation of Finite Automata and Lexical Analysis*, University of Stuttgart, Stuttgart, 1998

[4] Chomsky, N. a Miller G.A.: *Finite state languages.* Information and Controll 1:2 (91–112), 1958

[5] Kleene, S.C.: *Representation of event in nerve nets.* vydáno v Shannon, C.E a McCarthy, J.: *Automata Studies*, Princeton University Press, Princeton, 1956.

[6] Rabin, M.O. a Scott, *Finite automata and their decision problems.* IBM J. of Research and Development 3 (114–125), 1959

[7] Robbins, A.D.: *GAWK: Efective AWK Programming*, Free Software Foundation, Boston, 2004

[8] Moore, E.F.: *Sequential machines: Selected papers.*, Addison-Wesley, Reading, 1964

[9] Chytil M.: *Automaty a gramatiky*, SNTL, Praha, 1984.

[10] The Open Group: *The Single UNIX Specification, Version 3*,The Open Goup Base Specifications Issue 6, IEEE Std 1003.2, 2001

[11] Thopson, K.: *Regular expression search algorithm.* Communications of the ACM 11:(419–422), 1968

[12] Doxygen, www.doxygen.org

[13] The GNU project, http://www.gnu.org

[14] Unicode, http://www.unicode.org

[15] Universal Character Set, http://anubis.dkuug.dk/JTC1/SC2/WG2/

Příloha A

Obsah CD

Adresářová struktura přiloženého CD:

`install/`
: instalační balíčky pro Win32 (zkompilované binární soubory) a UNIX (zdrojové kódy a instalační makefile)

`doc/`
: manuálová stránka v troff a html formátu

`ref/`
: referenční příručka v html

`autocomp/`
: zdrojové kódy programu, makefile a příklad

`bak_cerny_ondrej.pdf`
: dokument bakalářské prace

Příloha B

Manuálová stránka

Následují vybrané části manuálové stránky, celou dokumentaci lze nalézt na CD (viz Příloha A). Uvádíme informace o syntaxi zdrojového kódu, regulárních výrazů a další poznámky, které nebyly plně rozvedeny v práci. Vodorovné linky označují vynechané části manuálové stránky.

NÁZEV

AutoComp - Automaton Compiler

PŘEHLED

autocomp [-bn] [-v *soubor*] [-i *typ*] [-s *adresář*] [-o *výstup*] *vstup*

ÚVOD

Autcomp je generátor programů pro práci s textovými daty, jejich načítání a analýzu. Na základě zdrojového kódu generuje *autocomp* modul programu v jazyce C nebo Perl dle zadání. Generovaný modul poskytuje rozhraní pro načítání a lexikální analýzu textu.

VOLBY

-b

Zdrojový kód *autocompu* je ve vnitřním formátu. Vnitřní formát není primárně určen pro uživatelské použití.

-n

Potlačí generování ladících částí programu.

-i *typ*

Vstupní soubor ve formátu *typ*, kde *typ* je název nestandartního vstupního formátu. Pro zpracování vstupního souboru a převedení do datových struktur se použije sdílený objekt `autocomp.rdr.`*`typ`*`.so`. Pokud není přitomen parametr **-b** ani **-i**, použije se objekt `autocomp.rdr.impl.so`, který je standartní součástí instalace.

-s *adresář*

Program bude hledat v adresáři *adresář* sdílené objekty, které se linkují za běhu. Sdílené objekty `autocomp.lan.*.so` poskytují prostředky pro generovnání zdrojových kódů v jednotlivých programovacích jazycích. Objekty `autocomp.enc.*.so` poskytují prostředky pro zpracování a převod znaků v konkrétním kódování znaků. Objekty `autocomp.rdr.*.so` umožňují načítání vstupu v různých formátech. Pokud není volba uvedena, objekty jsou hledány v pracovním adresáři a dle konvencí operačního systému pro hlednání sdílených objektů.

-o *výstup*

Jméno výstupního souboru bez přípony. Přípona bude přiřazena podle konvencí pro pojmenovávání zdrojových kódů v daném programovacím jazyce.

-v *soubor*

Zapne výpis doplňujících informací do souboru *soubor*.

vstup

Jméno vstupního souboru se zdrojovým kódem *autocomp*u.

NÁVRATOVÁ HODNOTA

Návratová hodnota programu *autocomp* bude 0 v případě bezchybného průběhu a ukončení. V případě chyby vrací nenulovou hodnotu a generuje chybový výpis na **stderr**.

⋮

VSTUP

Vstupní zdrojový kód obsahuje deklarace *stavů*, *přechodových hran*, popisu *vstupního zařízení*, informací o *modulu* a *specifikací výstupu* generátoru. Deklarace má následující tvar:

```
název(argument1,argument2,argument3,...,argumentN)
```

Název určuje, zda se jedná o deklaraci *stavu*,*přechodové hrany* atd. *Argumenty* mají dané pořadí a jména, některé mají implicitní hodnotu. Argumenty bez implicitní hodnoty jsou povinné. *Autocomp* dovoluje přiřazovat argumenty pomocí jména nebo pořadí a vzájemné kombinace těchto dvou forem. Pojmenované přiřazení má tvar: jméno=hodnota Jednotlivé argumenty musí být odděleny čárkami. Při kombinaci pojmenovaných a nepojmenovaných hodnot se nepojmenovaná hodnota přiřadí prvnímu dosud neobsazenému argumentu. Hodnoty argumentů se zadávají uzavřené do dvojitých uvozovek (”). Zdrojový kód je case-sensitivní a podporuje komentáře dle normy jazyka C89. Jednotlivé elementy kódu mohou být proloženy libovolným počtem bílých znaků a komentářů. Zdrojový kód je přijímán v kódování UTF-8, což umožňuje zadávat libovolné znaky do regulárních výrazů. Všechny ostatní položky však musí být pouze ASCII znaky, kvůli použitelnosti v generovaném kódu.

Stav

Stavy určují kontext generovaného programu. Jsou navzájem pospojovány přechodovými hranami, které umožňují změnu stavu. Před opuštěním stavu se volá *tázací funkce*, jejíž argumenty jsou výchozí stav, cílový stav a uživatelská data. Návratová hodnota *tázací funkce* rozhoduje o tom zda se vůbec může uskutečnit přechod z výchozího stavu do cílového. V ladícím režimu se před vstupem do stavu volá *vstupní ladící funkce* a před opuštěním stavu *výstupní ladící funkce*, jejich argumenty jsou výchozí, cílový stav a uživatelská data. Stav se deklaruje pomocí příkazu:

state(name,start,final,ask,d_in,d_out)

name

je jméno stavu, řetězec skládající se z alfanumerických znaků sady ascii a podtržítka.

start

určuje, zda stav může být použit jako startovní. Je-li hodnotou libovolný řetězec začínající na znaky "1","y" nebo "Y", stav bude označen jako startovní. Implicitní hodnota je "0".

final

určuje, zda je stav koncový. Interpretace hodnoty a implicitní hodnota je stejná jako v předchozím případě.

ask

určuje jméno *tázací funkce* v daném programovacím jazyce. Implicitně je hodnota prázdná, což znamená že se v tomto stavu tázací funkce nepoužije.

d_in

jméno *vstupní ladící funkce* v daném programovacím jazyce. Implicitně je hodnota prázdná.

d_out

jméno *výstupní ladící funkce* v daném programovacím jazyce. Implicitně je hodnota prázdná.

Ve vygenerovaném programu jsou stavy očíslovány. Pro čísla stavů jsou v jazyce C definována makra <*MODUL*>**_S_***name*. V jazyce Perl jsou čísla přístupná přes třídu <*modul*>**States**.
Instance třídy <*modul*>**States** je referencí na asocitativní pole (hash), kde klíče jsou názvy stavů a hodnoty jejich čísla (viz perlobj(1), www.perl.com).

<*MODUL*> zastupuje název modulu velkými písmeny (viz toupper(3)) a <*modul*> název modulu v původní podobě, toto značení platí i v dalším textu.

Přechodová hrana

Přechodová hrana propojuje *výchozí* a *cílový* stav. Přechod z *výchozího* do *cílového* stavu lze uskutečnit, pokud se načtený řetezec shoduje se vzorem daným *regulárním výrazem*. Je-li *hladová*, preferuje přechod nejdelším možným řetězcem. Je-li *líná*, upřednostňuje shodu s nejkratším řetězcem. Při přechodu po přechodové hraně je volána *přechodová funkce*, jejíž argumenty jsou *výchozí* a *cílový* stav, načtený řetězec a uživatelská data. Návratová hodnota *přechodové funkce* může přechod odmítnout, vráti-li 0. V ladícím režimu se při přechodu volá také *ladící přechodová funkce*, jež dostává stejné argumenty. Přechodová hrana je deklarována příkazem:

trans(from,to,over,lazy,cback,d_chng)

from

je jméno *výchozího stavu*, platí stejná pravidla jako pro jméno stavu (**name**).

to

určuje jméno *cílového stavu*.

over

vzor daný *regulárním výrazem*, viz sekce REGULÁRNÍ VÝRAZY.

lazy

určuje, zda je přechodová hrana *líná* nebo *hladová*. Pokud je hodnotou řetězec, začínající na znaky "1", "y" nebo "Y", je hrana *líná*. Implicitně je přechodová hrana *hladová*.

cback

jméno *přechodové funkce* v daném programovacím jazyce. Implicitně je hodnota prázdná.

d_chng

jméno *ladící přechodové funkce* v daném programovacím jazyce. Implicitně je hodnota prázdná.

Vstupní zařízení

Popis vstupního zařízení určuje jeho typ a znaková sada pro kódování znaků. Podporované typy vstupních zařízení jsou *soubor* a *paměť*. Podporované znakové sady jsou *ASCII* (ansi), *ISO-8859-1*, *UTF-8*, *UCS-2* a *UCS-4*. (Poznamka: v ASCII projdou skutecne jen ASCII znaky tedy 0-127.) Vstupní zařízení se specifikuje příkazem:

input(type,enc)

type

je typ vstupního zařízení, hodnota "file" určuje *soubor* a hodnota "mem" určuje *paměť*.

enc

určuje znakovou sadu ("ascii" pro *ASCII*, "latin1" pro *ISO-8859-1*, "ucs2" pro *UCS-2*, "ucs4" pro *UCS-4* a "utf8" pro *UTF-8*).

Modul

Informace o modulu se skládají ze *jména modulu*, *typu uživatelských dat*, a jmen souborů s deklaracemi funkcí. Zadávají se příkazem:

module(name,dtype,cbacks,debugs)

name

je *jméno modulu*, řídí se stejnými pravidly jako jméno stavu.

dtype

je *typ uživatelských dat*, řídí se syntaxí daného programovacího jazyka.

cbacks

je jméno souboru s deklaracemi přechodových a tázacích funkcí v daném programovacím jazyce.

debugs

je jméno souboru s deklaravemi ladících funkcí v daném programovacím jazyce.

Soubory budou *includovány* do generovaných kódů, což znamená, že deklarace funkcí budou zapotřebí až při kompilování vygenerovaného programu.

Specifikace výstupu

Specifikace výstupu dovoluje určit *programovací jazyk* generovaného programu a zda generovat v *ladícím režimu*. Provádí se příkazem:

output(lang,debug)

lang

je *programovací jazyk*, dovolené jsou hodnoty "c" a "perl".

debug

určuje, zda generovat v *ladícím režimu*. V případě hodnot začínajících na znaky "1" nebo "y" nebo "Y" je program generován v *ladícím režimu*.

REGULÁRNÍ VÝRAZY

Regulární výraz, neformálně řečeno, je vzor, který zastupuje množinu řetězců. Výraz se skládá z konstant a operátorů. Syntaxe regulárních výrazů podporovaných *autocompem* je stejná jako základní syntaxe regulárních výrazů běžně používaných v Unixovém prostředí.

Operátory a další znaky se speciálním významem jsou tzv. metaznaky. Chceme-li použít metaznak jako normální znak, musíme před něj uvést znak \. Uvnitř hranatých závorek jsou za metaznaky považovány **^-[]** a mimo hranaté závorky **.+?*|{}()[]**.

Konstantou v regulárním výrazu je znak nebo skupina znaků a je sama o sobě regulárním vyrazem. Znak zastupuje množinu řetězců, která obsahuje právě daný znak. Skupina znaků se zapisuje do hranatých závorek [] ve tvaru [výčet1^výčet2]. Ta zastupuje znaky, které se vyskytují v 1. výčtu a nevyskytují se v 2. výčtu. Je-li 1. výčet prázdný, je vnímán jako výčet všech znaků. Je-li 2. výčet prázdný, metaznak ^ nemá smysl, není proto povoleno jej uvést. Výčet znaků může být dán skutečným výčtem všech znaků, které do něj patří nebo rozsahem. Ten je dán prvním znakem, pomlčkou a posledním znakem rozsahu. Rozsah je výčtem všech znaků mezi prvním a posledním znakem včeteně, pořadí znaků je dáno znakovou sadou. Existují předdefinované skupiny znaků:

.
: zastupuje libovolný znak.

\w
: zastupuje libovolný alfanumerický znak nebo podtržítko. Za alfanumerické znaky jsou považovány znaky ze skupiny [a-zA-Z0-9] v kódování ascii (tzn. nejsou zde zahrnuty znaky s diakritikou či jiné znaky specifické pro lokální abecedy).

\W
: zastupuje libovolný znak kromě znaků ve skupině **\w**.

\s
: zastupuje libovolný netisknutelný bílý znak včetně mezer, tabulátorů a oddělovačů řádek. Je třeba dát pozor na platformách, kde jsou oddělovače řádek tvořeny dvěma znaky (např. Windows).

\S
: zastupuje libovolný znak kromě znaků ve skupině **\s**.

\n
: zastupuje oddělovač řádek. Není zcela korektí uvádět tento regulární výraz jako skupinu, viz níže.

\t
: zastupuje tabulátor.

Skupiny **\w**, **\W**, **\s**, **\S**, **\n** a **\t** lze použít i jako výčty znaků. Použije-li se skupina **\n** jako výčet, je jejím obsahem skuetěčně jen znak LF. Je-li ovšem použit jako samostaná skupina, zastupuje množinu řetězců LF,CRLF. Zmíněné předdefinované skupiny jsou deklarovány ve sdíleném objektu (knihovně) odpovídající zvolené znakové sadě, lze je s jistotou použít při znakových sadách standartně podporovaných *autocompem*. Při implementaci sdíleného objektu pro jiné znakové sady je povinná deklarace pouze skupin **.** a **\n**. Naopak je možné deklarovat nové předdefinované skupiny.

Operátor zřetězení regulárních výrazů se zapisuje pouhým postavením regulárních výrazů za sebe. Výsledný regulární výraz zastupuje množinu řetězců, které jsou tvořeny řetezcem z prvního regulárního výrazu následovaným řetězcem z druhého regulárního výrazu.

Iterační operátory jsou unární a zapisují se za regulární výraz. Výsledek se vyznačuje jistým opakováním původního regulárního výrazu. Přehled iteračních operátorů:

*****

Výsledek zastupuje množinu řetězců, které jsou tvořeny libovolným (i žádným) opakováním řetězců z původního regulárního várazu.

+

Výsledek zastupuje množinu řetězců, které jsou tvořeny alespoň jedním opakováním řetězců z původního regulárního várazu.

?

Výsledek zastupuje množinu řetězců původního regulárního výrazu sjednocenou s množinou obsahující pouze prázdný řetězec.

{*n*}

Výsledek zastupuje množinu řetězců, které jsou tvořeny zřetězením právě n řetězců z původního regulárního výrazu.

{*n*,*m*}

Výsledek zastupuje množinu řetězců, které jsou tvořeny zřetězením alespoň n, nejvíce však m řetězců z původního regulárního výrazu.

{*n*,}

Výsledek zastupuje množinu řetězců, které jsou tvořeny zřetězením nejméně n řetězců z původního regulárního výrazu.

Posledním operátorem je | (svislítko), je to binární operátor a zapisuje se infixně. Výsledkem je sjednocení množin zastoupených regulárními výrazy.

Operátor | má nižší prioritu než operátor zřetězení a před ním mají přednost iterační operátory.

Pro hledání náležitosti řetězce do množiny zastoupené regulárním výrazem se v *autocompu* používá Mooreův stroj. Protože se zároveň testuje více množin, obsahuje na koncových stavech identifikaci prechodové hrany, která obsahuje daný regulární výraz. Samotný průchod Mooreovým strojem se vyznačuje lineární složitostí vůči délce vstpního řetězce, nečte žádný znak vícekrát.

Poznámka k regulárním výrazům

Při formulaci regulárních výrazů pro program AutoComp je třeba brát v úvahu princip funkce vygenerovaného programu. Vygenerovaný program striktně dodržuje pravidla uvedená v kapitole POPIS, což může vést při neopatrné formulaci výrazu k poněkud kontraproduktivnímu chování. Použije-li se například regulární výraz .* na hladové hraně, nutně to vede k načtení celého vstupu. Naopak stejný výraz na líné hraně se nejprve pokusí přejít prázdným slovem. Další upozornění je třeba uvést v souvislosti s omezenými iteracemi **{n,m}**. Zde je nutné mít na paměti, že velikost datových polí vygenerovaného programu roste s horní mezí iterace. V případě nutnosti omezené iterace s velkou horní mezí (> 5) je vhodné uvažovat o řešení pomocí konstrukce v rámci stroje. Možným řešením je cyklická hrana (ze stavu S do stavu S), jejíž přechodová funkce bude zaznamenávat počet iterací.

PŘÍKLAD VSTUPU

```
input("file","ascii")

state("operator","1")
        trans("operator","number","[0-9]+",cback="store_number")
                /* store_number() uloží načtené
                 * číslo do kontextu
```

```
                 */
        trans("operator","identifier","[a-zA-Z_]\w+"
                ,cback="handle_identifier")
                /* handle_identifier() se postará
                 * o~zjištění hodnoty identifikátoru
                 */
        trans("operator","operator","\-"
                ,cback="handle_unary_operator")
                /* handle_unary_operator() kontorluje
                 * korektnost výskytu unárního mínus
                 */
state("number")
        trans("number","operator","[\+\-\*/]"
                ,cback="handle_operator")
                /* handle_operator() se postará
                 * o~zařazení operátoru do zásobníku
                 */
        trans("number","end","=")
state("identifier")
        trans("identifier","operator","[\+\-\*/]"
                ,cback="handle_operator")
                /* handle_operator() se postará
                 * o~zařazení operátoru do zásobníku
                 */
        trans("identifier","end","=")
state("end",final="1")

trans("operator","operator","\(",cback="push_down")
                /* push_down() se postará o~zanoření
                 * do podvýrazu
                 */
trans("operator","operator","\)",cback="pop_up")
                /* push_down() se postará o~vynoření
                 * z~podvýrazu
                 * samozřejmě kontroluje, zda se
                 * předtím provedlo zanoření
                 */

module("simple_exps","struct context*","callbacks.h")

output("c")
        /* jazyk C, bez ladících operací */
```

Tento jednoduchý příklad může sloužit pro tvorbu jednoduchého programu na vyhodnocování výrazu s proměnnými. Součástí balíčku AutoComp je příklad založený na tomto zdrojovém kódu.

⋮

Ladící výpisy

Program generovaný v ladícím režimu umožňuje provádět ladící výpisy. Vypisovat lze informace na dvou úrovních, první obsahuje údaje o provedených přechodech, druhá údaje o stavu bufferu, dalších vnitřních struktur a Mealyho strojů. Obě úrovně ladících výpisů je třeba explicitně vyžádat.

V jazyce C je nutné vyžádat ladící výpisy definováním speciálních maker před kompilací vygenerovaného modulu. Makra tedy musí být definována v nějakém souboru zahnutém do modulu preprocesorem, nejlépe v souboru zadaném jako `debugs` v příkazu `module()`. Definováním makra <*MODUL*>_DEBUG_PRINT se zapnou ladící výpisy první úrovně, makrem <*MODUL*>_DEBUG_PRINT_REGEX výpisy druhé úrovně. Pokud makra nejsou definována, neprovádí se za běhu žádné operace spojené s výpisy.

V jazyce Perl se zapínání a vypínání provádí za běhu. Při vypnutých výpisech se provádí jeden test spojený s jedním výpisem. Pro ovládání výpisů jsou připraveny dvě metody třídy <*modul*>, které přijímají jeden argument. Pokud je argument vyhodnocen jako pravdivá hodnota, odpovídající výpisy se zapnou, jinak se vypnou. Metoda **do_debug_printing( )** slouží k ovládání výpisů první úrovně, metoda **do_debug_more_printing( )** ovládá výpisy druhé úrovně.

Pokud je program generován v normálním režimu (`output( debug="0")`), nejsou funkce a operace spojené s ladícími výpisy v žádném případě součástí vygenerovaného kódu.

⋮

Odstranění nedosažitelných stavů

V automatu se mohou vyskytovat nedosažitelné stavy. Mohou vznikout determinisací, nebo mohou být součástí nepříliš dobře navrženého zadání. Nedosažitelné stavy jsou odstraněny spolu s jejich hranami, vstupujícími i vystupujícími. Hledání nedosažitelných stavů se provádí prohledáváním automatu do šířky ze startovních stavů. Stavy, které se staly nedosažitelnými v důsledku "zastřešení" jiným dosažitelným stavem, je nutné zachovat jejich čísla (kvůli volání *přechodové funkce*).

Zmenšování tabulek

Každému dosažitelnému stavu je přiřazen jeden Mooreův stroj, který reprezentuje vzory na přechodových hranách. Mooreův stroj je uložen v polích vygenerovaného kódu a je representován tabulkou přechodů, abecedou (rozdělení znakové sady na intervaly specifické pro dané vzory) a množinou výstupních symbolů (čísla přechodových hran). Paměťový prostor, který je ve výsledném kódu spotřebován právě uložením těchto tabulek, se *autocomp* snaží snížit na minimum následujícími technikami:

přetěžování abeced

Pokud pracuje více Mooreových strojů nad stejnou abecedou, uloží se abeceda pouze jednou.

zmenšování Mooreových strojů

Pro zmenšování Mooreových strojů se používá mechanismus pro minimalizaci konečného automatu. Dále se neukládají stavy Mooreova stroje, pro které je přechodová funkce (Mooreova stroje) prázdná. Na tyto stavy je pak odkazováno speciálním číslem. Pomocí lineární transformace tohoto čísla se pak získá skutečné číslo stavu, které je použito pro indexování do pole výstupních symbolů. Stavy Mooreových strojů jsou seřazeny tak, že první jsou stavy s neprázdnou množinou výstupních symbolů, nemusí se tudíž indikovat prázdná množina pro každý stav, stačí číslo posledního stavu s neprázdnou množinou.

další odstraňování stavů automatu

Pokud nějaký stav má prázdný Mooreův stroj, tomuto stavu se přiřadí pouze číslo větší než čísla ostatních stavů, které pouze indikuje, zda je stav koncový či nikoliv (paritou). Takovému stavu není přiřazen žádný záznam v tabulkách Mooreových strojů, výstupních symbolů ani abeced.

AUTOR

Ondřej Černý, ondrej.cerny@email.cz

DALŠÍ INFORMACE

http://www.ms.mff.cuni.cz/~cerno4am/AutoComp/