VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

MNOHAÚROVŇOVÉ AUTOMATY A JEJICH APLIKACE

MULTI-LEVEL AUTOMATA AND THEIR APPLICATIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **KAMIL PŠENÁK**
AUTHOR

VEDOUCÍ PRÁCE **prof. RNDr. ALEXANDER MEDUNA, CSc.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Pšenák Kamil**
Program: Informační technologie
Název: **Mnohaúrovňové automaty a jejich aplikace**
Multi-Level Automata and Their Applications
Kategorie: Teoretická informatika

Zadání:

1. Dle instrukcí vedoucího se seznamte s různými automaty a jejich jazyky.
2. Zaveďte mnohaúrovňové automaty jako automaty, jejichž stavy jsou jiné automaty, které mohou být také mnohaúrovňové.
3. Studujte vlastnosti těchto automatů a jazyků. Porovnejte jejich sílu s jinými automaty.
4. Uvažujte proces lexikální analýzy prováděný překladači. Dle pokynů vedoucího jej formalizujte prostřednictvím mnohaúrovňových automatů.
5. Implementujte formalizaci navrženou v bodě 4.
6. Zhodnoťte dosažené výsledky. Diskutujte další vývoj projektu.

Literatura:

- Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1-3, Springer, 1997, ISBN 3-540-60649-1
- Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition), Pearson Education, 2006, ISBN 0-321-48681-1

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Meduna Alexander, prof. RNDr., CSc.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 31. října 2018

Abstrakt

V tejto práci rozšírime zastarané prístupy v teoretickej informatike. Ukážeme si, že je možný paralelizmus v konečných automatoch zavedením viacúrovňového konceptu. Priblížime si proces kompilácie a stavbu kompilátoru, aby sme mali reálny príklad pre viacúrovňové konečné automaty. Posunieme sa hlbšie do teoretickej informatiky a vysvetlíme si paralelné pravo-linárne gramatiky a jazyky. Následne si na príklade aj s návrhom implementácie dokážeme tvrdenie. Na záver si spomenieme ďalšie možné odvetvia, kde by sa tento koncept dal využiť.

Abstract

In this thesis, we will add to already known finite automata paradigm. We start with basic definitions used in theoretical informatics. Afterwards we define multi-level finite automata, which is the base of this thesis. Then we move on to the compilation process and construction of a compiler. With that we define lexical analysis as our example for multi-level finite automata implementation. Once we implement the concept, we compare the new and the old way. Then we dig deeper into theoretical informatics to define parallel right-linear grammars and languages. To prove a concept we create another concept with implementation strategy, using multi-level framework. Lastly, we mention some other areas in informatics, where this multi-level concept could be useful.

Kľúčové slová

konečné automaty, viacúrovňové konečné automaty, koncept viacúrovňovosti, paralelizmus v teoretickej informatike, paralelné pravo-lineárne gramatiky, kompilácia, kompilátor, lexikálna analýza

Keywords

finite automata, multi-level finite automata, multi-level concept, parallelism in theoretical informatics, parallel right-linear grammars, compilation, compiler, lexical analysis

Citácia

PŠENÁK, Kamil. *Mnohaúrovňové automaty a jejich aplikace*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. RNDr. Alexander Meduna, CSc.

Mnohaúrovňové automaty a jejich aplikace

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne za pomoci pána prof. RNDr. ALEXANDER MEDUNA, CSc.. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Kamil Pšenák
7. mája 2019

Poďakovanie

Chcel by som sa poďakovať pánovi prof. RNDr. ALEXANDER MEDUNA, CSc. za pomoc pri tvorbe tejto práce.

Obsah

Kapitola 1

Úvod

V dnešnej dobe, kde sa technika posúva dopredu neuveritelne rýchlo, sa stále stretávame so zastaralými metódami v informatike. Tak ako všetky oblasti informatiky aj teoretická informatika vyžaduje inovácie. Jednou z nich je návrh popísaný v tejto práci. Odrážajúca myšlienka vznikla vďaka otázke, prečo hľadať riešenie tam, kde sa problém vôbec nenachádza? Začneme krátkym úvodom do regulárnych jazykov, tie si definujeme. Preberieme si regulárne výrazy, ktoré popisujú regulárne jazyky. Naučíme sa skracovať regulárne výrazy, a to odstraňovaním zátvoriek a stanovením si priority operátorov. Následne zistíme, že regulárne jazyky nám najjednoduchšie popisujú konečné automaty, a preto si ich priblížime. Konečný automat je založený na konečnej množine stavov a prechodov medzi nimi. Na obrázkoch si ukážeme ako sa zakreslujú stavy a prechody medzi nimi. Následne zistíme, že si potrebujeme definovať čo je to vlastne prechod, a taktiež sekvencia prechodov. Akonáhle budeme vedieť čo sú konečné automaty, prichádzajú na rad viacúrovňové konečné automaty, ktoré sú hlavným dôvodom tejto práce. Zavedenie takzvaných viacúrovňových konečných automatov nám prinesie možnosť zabudnúť na staré prístupy modelovania, a programovania, a umožní používať elegantnejší a ľahšie modulovateľný prístup k práci. Jednoduché rozdelenie úloh v rámci tímu bol jeden z hlavných motivátorov pri vzniku tohoto konceptu. Taktiež veľkú rolu zahŕňa myšlienka paralelizmu, ktorá bola v pôvodnom princípe konečných automatov nemožná. Na začiatok si samozrejme definujeme viacúrovňové konečné automaty. Následne si ukážeme rozdiely v modelovaní pomocou týchto automatov. A to rozdiely v zakreslení stavov a hlavne všetky možnosti prechodov. Celý prístup k problematike je použiteľný ako rámec pri riešení témy gramatík, prekladačov, jazykov, tak ako je popísaný v tejto práci. Aby sme si mohli popísať rozdiel v implementácií konečných automatov, pod ktorými budeme rozumieť deterministické konečné automaty, a v implementácií viacúrovňových konečných automatov, si priblížime postupne proces kompilácie programu. Kompilátor číta zdrojový program napísaný v zdrojovom jazyku a prekladá tento program do cieľového programu napísaného v cieľovom jazyku, tak aby oba programy boli funkčne ekvivalentné, teda špecifikovali rovnakú výpočetnú úlohu. Kompilátor na začiatku prevedie lexikálnu, syntaktickú a sémantickú analýzu zdrojového programu. Potom na základe zozbieraných informácií vygeneruje medzikód, následne prevedie optimalizácie, a nakoniec vygeneruje cieľový kód. Jednotlivé fázy si hlbšie popíšeme a nakoľko vo fázy lexikálnej analýzy sa využívajú konečné automaty, na tejto fázy si ukážeme implementačné rozdiely. Samozrejme si celkový proces lexikálnej analýzy popíšeme viac dopodrobna, a určíme si model pre lexikálnu analýzu, ktorý sme už spomenuli vyššie, ktorým sú konečné automaty. Aby sme mohli porovnať dva modely, tak si definujeme aj model viacúrovňových konečných automatov na proces lexikálnej analýzy. Následne pre potrebu reálnej implementácie, bu-

deme potrebovať jazyk nad ktorým prevedieme analýzu. Postupne sa dostaneme od zadania jazyku IFJ17, na jeho model vytvorený ako konečnými automatmi, tak aj viacúrovňovými konečnými automatmi. Funkciu jednotlivých automatov si podrobne popíšeme, tak aby príjmali daný jazyk. Po tomto kroku, si ukážeme ako sa implementujú oba druhy automatov, a taktiež si opíšeme kde sú výhody a kde nevýhody viacúrovňového prístupu. Aby sme si naozaj overili, že sa tento prístup dá použiť ako rámec aj pri iných druhoch gramatík, priblížime si problematiku paralelne pravo-lineárnych gramatík. Tieto gramatiky si definujeme a presunieme sa do konceptu sebaregulujúcich automatov. Priblížime si čo sa skrýva pod pojmom sebaregulácie. Pre dokázanie významu viacúrovňového prístupu, si urobíme podrobný koncept teoretického príkladu implementácie sebaregulujúcich viacúrovňových konečných automatov. V danom koncepte si dokážeme jednoduchosť návrhu, a schopnosť zvýšenia výpočtovej sily automatu na reálnom príklade. Aby sme si na záver ešte viac potvrdili dôležitosť tohto rámcu, ukážeme si alternatívne využitie daného konceptu. Navrhneme si možné využitie tohto prístupu pri celulárnych automatoch, aby sme sa trošku oddialili od gramatík a jazykov, a poukázali na to, že je tento rámec možné použiť aj v iných sférach informatiky. A nakoniec sa posunieme do sféry informačných systémov, kde je daný koncept viacúrovnosti taktiež použiteľný.

Kapitola 2

Modely pre regulárne jazyky

Na to aby sme mohli spomínať modely pre regulárne jazyky, musíme najskôr určiť aké to sú jazyky. Na úvod si definujeme regulárne výrazy. Hlavná myšlienka: Regulárne výrazy sú výrazy s operátormi ., + a *, ktoré značia v tomto poradí konkatenáciu, zjednotenie a iteráciu.

Definícia 2.0.1 *Nech Σ je abeceda. Regulárne výrazy nad abecedou Σ a jazyky, ktoré značia, sú definované nasledovne:*

- *$\emptyset$ je regulárny výraz označujúci prázdnu množinu (prázdny jazyk)*
- *ϵ je regulárny výraz označujúci jazyk ϵ*
- *a, kde $a \in \Sigma$, je regulárny výraz označujúci jazyk a*
- *Nech r a s sú regulárne výrazy označujúce v rade jazyky L_r a L_s, potom*
  - *(r.s) je regulárny výraz označujúci jazyk $L=L_rL_s$*
  - *(r + s) je regulárny výraz označujúci jazyk $L=L_r \cup L_s$*
  - *(r*) je regulárny výraz označujúci jazyk $L=L_r*$*

Po definovaní regulárnych výrazov sa pozrime na príklad.
Otázka znie: Je ((c + (a.(b*))) + ϵ) regulárny výraz nad abecedou Σ = a, b, c? Postupné riešenie:

- c, a, b, ϵ sú regulárne výrazy nad Σ
- (b*) je regulárny výraz nad Σ
- (a.(b*)) je regulárny výraz nad Σ
- (c + (a.(b*))) je regulárny výraz nad Σ
- ((c + (a.(b*))) + ϵ) je regulárny výraz nad Σ

Nakoľko môžeme vidieť, že v regulárnych výrazoch sa nachádza veľa zátvoriek, môžeme ich zjednodušiť. Zátvorky môžeme redukovať zavedením priority operátorov, a to nasledovne: operátor * má najväčšiu prioritu, operátor . strednú a operátor + najmenšiu. Skrátene: * > . > +.
Ďalej skrátenie zápisu: regulárny výraz r.s môžeme zapísať ako rs a rr* alebo r*r môžeme zapísať ako r^+. Príklad: ((a.(a*)) + ((b*).b)) môžeme zapísať ako a.a* + b*.b, a a.a* + b*.b môžeme zapísať ako $a^+ + b^+$.

Týmto sa dostávame ku regulárnym jazykom. Základná myšlienka je, že každý regulárny výraz značí regulárny jazyk.

Definícia 2.0.2 *Nech L je jazyk. L je regulárny jazyk, pokiaľ existuje regulárny výraz, ktorý tento jazyk značí. Konvencia: L(r) označuje jazyk, ktorá značí regulárny výraz r.*

Príklady:

$r_1 = ab + ba$	značí $L_1 = \{ab, ba\}$
$r_2 = a_+b^*$	značí $L_2 = \{a^n b^m$: n $\geqq$ 1, m $\geqq$ 0$\}$
$r_3 = ab(a + b)^*$	značí $L_3 = \{$x: ab je prefix x$\}$
$r_4 = (a + b)^*ab(a + b)^*$	značí $L_4 = \{$x: ab je podreťazec x$\}$

L_1,L_2,L_3,L_4, sú regulárne jazyky nad Σ.

Ako sme sa dozvedeli, regulárne výrazy nám popisujú regulárne jazyky, avšak existuje ešte jeden základný model na popis regulárnych jazykov, a tým sú konečné automaty, ktoré si viac priblížime v nasledujúcej kapitole.

Kapitola 3

Konečné automaty

Konečný automat (viac v [3] a [4]) je najjednoduchší model založený na konečnej množine stavov a výpočetných pravidiel. Princíp automatu je, že na základe prechodových pravidiel, prechádzame medzi stavmi automatu. Automat vždy začína zo začiatočného stavu. Aplikáciou pravidiel na základe prečítaného vstupu prechádza automat medzi stavmi, stav v ktorom sa automat nachádza voláme aktuálny stav. Úspešne ukončí automat činnosť a prevedie nejakú koncovú akciu, ak sa nachádza v koncovom stave. Znak číta automat zo vstupnej pásky, čítacia hlava sa pri tom nachádza vždy na jednom znaku, a pohybuje sa z ľava do prava.

Definícia 3.0.1 *Konečný automat je pätica: $M = (Q, \Sigma, R, s, F)$, kde*

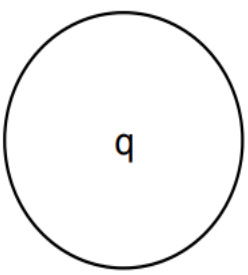
- *Q je konečná množina stavov*
- *Σ je vstupná abeceda*
- *R je konečná množina pravidiel tvaru: $pa \rightarrow q$, kde $p, q \in Q$, $a \in \Sigma \cup \{\epsilon\}$*
- *$s \in Q$ je začiatočný stav*
- *$F \subseteq Q$ je množina koncových stavov*

Matematická poznámka ku pravidlám: Čisto matematicky, R je relácia z $Q \times (\Sigma \cup \epsilon)$ do Q Namiesto relačného zápisu (pa, q) $\in$ R, zapisujeme: pa $\rightarrow$ q $\in$ R, pa $\rightarrow$ q znamená, že pri prečítaní znaku a, automat M urobí prechod zo stavu p do stavu q, ak sa a $= \epsilon$, zo vstupnej pásky nie je prečítaný žiadny symbol

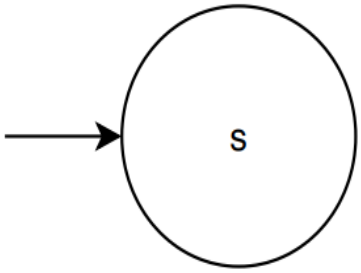
My sa v tejto práci zaoberáme iba grafickou reprezentáciou automatov, avšak existujú aj iné formy reprezentácie.

3.1 Grafická reprezentácia konečných automatov

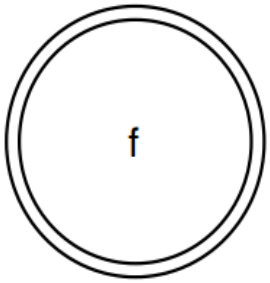
Konečné automaty reprezentujeme grafickými konvenciami, ktoré môžeme vidieť na nasledujúcich obrázkoch.



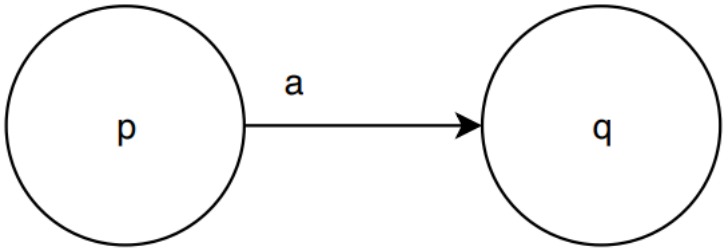
(a) Na tomto obrázku vidíme stav q ∈ Q



(b) Na tomto obrázku vidíme začiatočný stav s ∈ Q



(c) Na tomto obrázku vidíme koncový stav f ∈ F



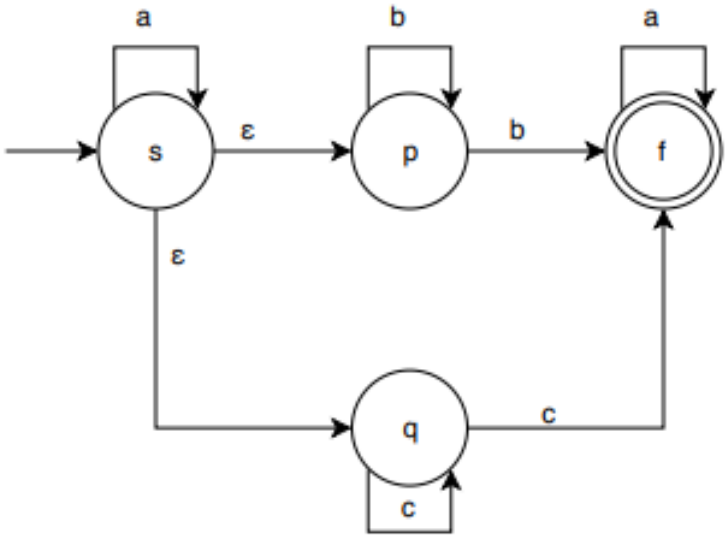
Obr. 3.2: Na tomto obrázku vidíme pravidlo pa → q ∈ R

Ukázali sme si základnú notáciu zakreslovania konečných automatov, teraz si ukážeme príklad.

Príklad:
M = (Q, Σ, R, s, F), kde:

- Q = {s, p, q, f}
- Σ = {a, b, c}
- R = {sa → s, s → p, pb → p, pb → f, s → q, qc → q, qc → f, fa → f}
- F = {f}

Grafickú reprezentáciu príkladu si ukážeme na nasledujúcom obrázku:



Obr. 3.3: Obrázok znázorňujúci príklad

3.2 Ostatné dôležité definície

Ďalej si musíme definovať prechod automatu. Myšlienka prechodu je, že prechod je jeden výpočetný krok automatu. Preto si musíme definovať čo je konfigurácia konečného automatu. Konfigurácia je inštancia popisu konečného automatu.

Definícia 3.2.1 *Nech $M = (Q, \Sigma, R, s, F)$ je konečný automat. Konfigurácia konečného automatu M je reťazec $\chi \in Q\Sigma^*$.*

Definícia 3.2.2 *Nech pax a qx sú dve konfigurácie konečného automatu M, kde $p, q \in Q$, $a \in \Sigma \cup \{\epsilon\}$ a $x \in \Sigma^*$. Nech $r = pa \to q \in R$ je pravidlo. Potom M môže previesť prechod z pax do qx použitím r, zapísané ako $pax \vdash qx$ $[r]$ alebo zjednodušene $pax \vdash qx$. Pokiaľ $a = \epsilon$, zo vstupnej pásky nie je prečítaný žiadny symbol.*

Následne si musíme definovať sekvenciu prechodov. Nosná myšlienka je zápis viacerých výpočetných krokov po sebe.

Definícia 3.2.3 *Nech χ je konfigurácia. M prevedie nula prechodov z χ do χ; zapisujeme: $\chi \vdash^0 \chi$ $[\epsilon]$ alebo jednoduchšie $\chi \vdash^0 \chi$.*

Definícia 3.2.4 *Nech $\chi_0, \chi_1, \ldots, \chi_n$ je sekvencia prechodov konfigurácií*
pre $n \geqq 1$ a $\chi_{i-1} \vdash \chi_i$ $[r_i]$, $r_i \in R$ pre všetky $i = 1, \ldots, n$, čo znamená:
$\chi_0 \vdash \chi_1$ $[r_1] \vdash \chi_2$ $[r_2] \ldots \vdash \chi_n$ $[r_n]$. Potom M prevedie n prechodov z χ_0 do χ_n; zapisujeme: $\chi_0 \vdash^n \chi_n$ $[r_1 \ldots r_n]$ alebo jednoduchšie $\chi_0 \vdash^n \chi_n$.
Pokiaľ $\chi_0 \vdash^n \chi_n$ $[\rho]$ pre nejaké $n \geqq 1$, potom $\chi_0 \vdash^+ \chi_n$ $[\rho]$.
Pokiaľ $\chi_0 \vdash^n \chi_n$ $[\rho]$ pre nejaké $n \geqq 0$, potom $\chi_0 \vdash^* \chi_n$ $[\rho]$.

Nakoniec si definujeme prijímaný jazyk. Myšlienka znie nasledovne: M prijíma reťazec w, pokiaľ je celý prečítaný pomocou sekvencie prechodov a skončí v nejakom koncovom stave.

Definícia 3.2.5 *Nech M = (Q, Σ, R, s, F) je konečný automat. Jazyk prijímaný konečným automatom M, L(M), je definovaný: L(M) = w: w* $\in \Sigma^*$*, sw* $\vdash^*$ *f, f* $\in$ *F.*

Rôznymi algoritmami sme schopný previesť regulárny výraz na konečný automat, a preto môžeme tvrdiť, že pre každý regulárny výraz R existuje konečný automat M, pre ktoré platí: L(R) = L(M).

Konečné automaty sú druhý základný model pre regulárne jazyky. Ich implementáciu budeme riešiť v kapitole lexikálna analýza spolu s porovnaním implementácií viacúrovňových konečných automatov.

Kapitola 4

Viacúrovňové konečné automaty

V predošlej kapitole sme sa zaoberali problematikou regulárnych jazykov a modelov používaných na ich reprezentáciu. Dozvedeli sme sa, že regulárne jazyky sú jednoznačne reprezentovateľné regulárnymi výrazmi a konečnými automatmi. Avšak, konečné automaty boli vymyslené ešte za čias keď neboli programovacie jazyky tak efektívne ako v dnešnej dobe. Konečné automaty nám neumožňujú reprezentovať paralelizmus v žiadnej forme, taktiež strácajú na sile po implementačnej stránke ako si ukážeme v ďalšej kapitole. Pre tieto dôvody si definujeme viacúrovňové konečné automaty, ktorých sila spočíva v jednoduchej modulácií práce a možnosti paralelného spracovania. Tento prístup je prehľadnejší, lepšie škálovateľný a implementačná časť sa dá jednoducho rozdeliť viacerým členom tímu ako je v dnešnej dobe zvykom. Oproti tomu nevýhodou by bolo, že viacúrovňové konečné automaty sú náročnejšie pre pochopenie a náročnejšie na zakreslenie.

4.1 Definícia

Na začiatok si musíme viacúrovňové konenčé automaty definovať, a to nasledovne.

Definícia 4.1.1 *Súbor viacúrovňových konečných automatov tvorí (n+1)-tica:*
$M = (M_1, M_2, \ldots, M_n, S)$, *kde*

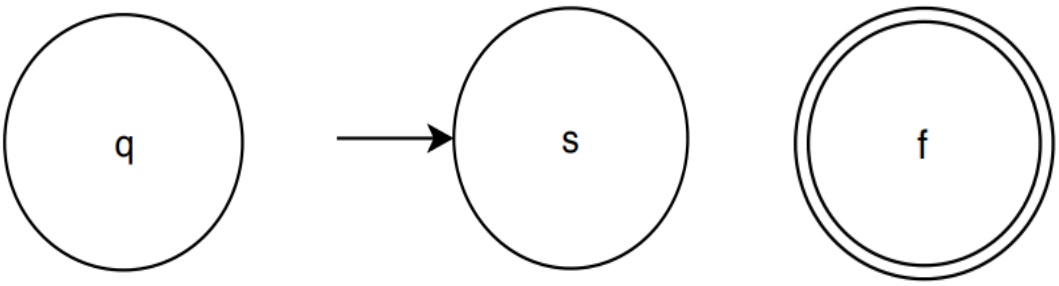
- S *je štartujúci automat*

a každý automat je definovaný štvoricou: $M_i = (Q_i, \Sigma_i, R_i, F_i)$, $i > 0$, *kde*

- Q_i *je konečná množina stavov*
- Σ_i, *je vstupná abeceda*
- $R_i = \{qa \rightarrow p \mid q \in Q_i, p \in Q_j, j \in \{1, \ldots, n\}; a \in \Sigma_i \cup \{\epsilon\}\}$
- $F_i \subseteq Q_i \cup \emptyset$ *je množina koncových stavov*

4.2 Grafická reprezentácia

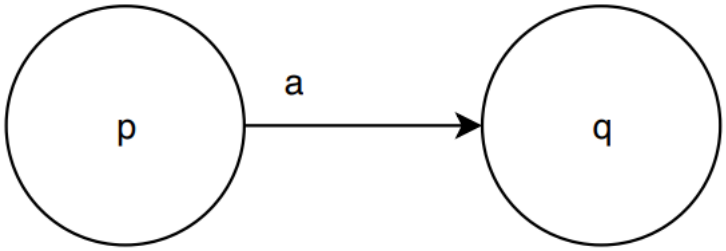
Praktické značenie zostáva rovnaké ako pri konečných automatoch. Avšak, s výnimkou značenia prechodov. Pri značení prechodov musíme myslieť na to, že v tomto prípade môžeme ísť nie len do iného stavu, ale aj do iného automatu, ktorý môže byť konečný alebo nie. Preto zavádzame konvenciu kde pred názov stavu do ktorého prechádzame, pridáme značku M_ aby bolo jasné, že sa jedná o nový automat. Grafickú reprezentáciu teda rozširujeme nasledovne:



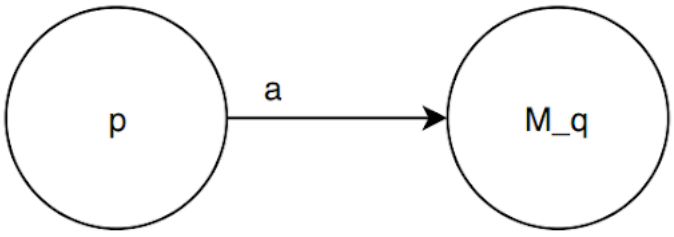
(a) Na tomto obrázku vidíme stav q ∈ Q

(b) Na tomto obrázku vidíme začiatočný stav s ∈ Q

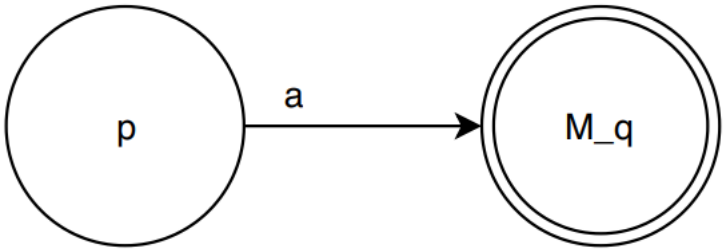
(c) Na tomto obrázku vidíme koncový stav f ∈ F



Obr. 4.2: Na tomto obrázku vidíme pravidlo pa → q ∈ R



Obr. 4.3: Na tomto obrázku vidíme pravidlo qa → p, kde q ∈ Q_i, p ∈ Q_j a i ≠ j, prechod do stavu iného automatu



Obr. 4.4: Na tomto obrázku vidíme pravidlo qa $\rightarrow$ p, kde q $\in Q_i$, p $\in F_j$ a i $\neq$ j, prechod do koncového stavu iného automatu

Bližšie si priblížime konkrétnu implementáciu a grafickú reprezentáciu na príklade v nasledujúcich kapitolách.

Kapitola 5

Kompilácia

V tejto kapitole si predstavíme proces kompilácie (viac v [3]), nakoľko v jednom kroku kompilácie sa využívajú konečné automaty, a preto sme si ho zvolili ako vzor pre túto prácu. Kompilátor číta zdrojový program napísaný v zdrojovom jazyku a prekladá tento program do cieľového programu napísaného v cieľovom jazyku, tak aby boli oba programy funkčne ekvivalentné, teda špecifikovali rovnakú výpočetnú úlohu. Zdrojový jazyk je z pravidla vysoko úrovňový jazyk, ako napríklad C, Pascal, Java, Python, zatiaľ čo cieľový jazyk je špecifický jazyk daného stroja, alebo assembler, ktorý je ľahko transformovateľný na strojový jazyk. Počas procesu prekladu kompilátor ako prvé analyzuje zdrojový súbor aby overil, že zdrojový program je korektne zapísaný v zdrojovom jazyku. Ak áno, kompilátor generuje cieľový program, inak kompilátor vráti chybové hlásenie.

5.1 Fázy kompilácie

Kompilátor na začiatku prevedie lexikálnu, syntaktickú a sémantickú analýzu zdrojového programu. Potom na základe zozbieraných informácií vygeneruje medzikód, následne prevedie optimalizácie, a nakoniec vygeneruje cieľový kód.

5.1.1 Lexikálna analýza

Lexikálna analýza rozdeľuje zdrojový program na jednotlivé lexémy. Tie sú logicky súdržné lexikálne entity, ako napríklad identifikátory alebo čísla. Overuje, či sú tieto entity správne poskladané, vyprodukuje tokeny, ktoré jednoznačne reprezentujú dané lexémy vo fixnej veľkosti. Vygenerované tokeny posiela syntaktickému analyzátoru. Ak je za potreby, sú tieto tokeny asociované rôznymi atribútmi. Lexikálnu analýzu prevádza takzvaný scanner, ktorý číta sekvencie znakov a z nich vytvára lexémy. Scanner je implementovaný pomocou deterministických konečných automatov (ďalej budeme považovať konečné automaty za deterministické, nakoľko iba tie sa dajú implementovať, viac o prevode konečných automatov na deterministické konečné automaty v [4]), a preto sa mu budeme venovať ešte ďalej.

5.1.2 Syntaktická analýza

Syntaktická analýza rozhoduje, či je správna syntaktická štruktúra roztokenovaného zdrojového programu, ktorú prevezme od lexikálnej analýzy. Syntaktickú analýzu prevádza takzvaný parser.

5.1.3 Sémantická analýza

Sémantická analýza kontroluje, či zdrojový program napĺňa sémantické konvencie zdrojového jazyka. Asi najdôležitejšia čas je, že kontroluje typy aby overil, že každý operátor má iba operandy povolené zdrojovým jazykom.

5.1.4 Generovanie medzikódu

Generátor medzikódu mení tokenizovaný zdrojový program na funkčne ekvivalentný rovnaký medzi jazyk. Ako z názvu poznať, tento jazyk je niekde na medzi úrovni zdrojového a cieľového kódu, a je kompletne nezávislý na strojovom kóde.

5.1.5 Optimalizácia

Optimalizácia medzikódu, prefrázuje predom vytvorený medzikód aby sa vykonal efektívnejšie.

5.1.6 Generovanie cieľového kódu

Generovanie cieľového kódu namapuje optimalizovaný medzikód na jeho reprezentáciu v cieľovom jazyku. Napríklad preloží optimalizovaný medzikód na inštrukcie v assemblery, ktoré prevedú rovnakú úlohu.

Viac o jednotlivých fázach sa môžete dočítať v [3].

Kapitola 6

Kompilátor

Šesť základných fáz kompilácie: lexikálna analýza, syntaktická analýza, sémantická analýza, generovanie medzikódu, optimalizovanie medzikódu a generovanie cieľového kódu, je abstrahovaných z procesu, ktorý prevádza reálny kompilátor, ktorý nemusí tieto fázy previesť sktriktne za sebou. Reálny kompilátor radšej tieto fázy prekrýva, aby došlo ku prekladu čo najrýchlejšie. Keďže syntaktická štruktúra zdrojového programu reprezentuje pravdepodobne najdôležitejšie informácie pre analýzu ako celku, syntaktický analyzátor vedie proces analýzy, a taktiež aj proces generovania medzikódu. Vskutku lexikálny analyzátor je zavolaný iba vtedy, keď syntaktický analyzátor potrebuje nový token. Syntaktický analyzátor taktiež volá sémantický analyzátor vtedy, keď potrebuje previesť sématicky štruktúrované kontroly. Syntaktický analyzátor vedie proces generovania medzikódu, vždy preloží časť tokenizovaného zdrojového programu na funkčne ekvivalentný medzikód. Tento proces sa nazýva syntakticky riadený preklad programu. Konštrukciu takéhoto kompilátora si môžete pozrieť v B, a taktiež viac informácií sa môžete dočítať v [3].

Kapitola 7

Lexikálna analýza

Nakoľko je lexikálny analyzátor implementovaný pomocou konečného automatu, rozhodli sme sa, že to využijeme na ukázanie starej technológie a skúsime ju nahradiť novšou a to viacúrovňovými automatmi. Lexikálny analyzátor funguje tak, že keď dostane požiadavku od syntaktického analyzátora na vytvorenie nového tokenu, v tomto momente lexikálny analyzátor(ďalej scanner) začne čítať tok znakov, ktoré tvoria zdrojový program. Scanner číta zdrojový program z ľava do prava aby rozoznal lexémy. Ku lexémam sa ešte vrátime a vysvetlíme si modely na rozpoznávanie. Scanner po rozpoznaní nasledujúceho lexému a overení jeho lexikálnej korektnosti, vytvorí token, ktorý reprezentuje rozpoznanú lexému, v jednoduchej a jednoznačnej forme. Po dokončení jeho úlohy posiela token syntaktickému analyzátoru, a tak splní jeho požiadavku.

Mimo tejto základnej úlohy, scanner väčšinou spĺňa ďalšie menšie úlohy. Scanner zvyčajne úzko a často spolupracuje s tabuľkou symbolov(viac v [3]) kvôli ukladaniu alebo hľadaniu identifikátorov vždy keď je za potreby. Ďalej prevádza triviálne úlohy určené na simplifikáciu textu zdrojového programu, a to napríklad zmena znakov z malých na veľké alebo opačne, odstránenie zbytočných znakov, ako napríklad bielych znakov alebo komentárov. Ďalej si ukážeme základné jazykové modely pre lexikálnu analýzu. Následne si ukážeme metódy a techniky používané v tejto analýze.

7.1 Modely pre lexikálnu analýzu

Preto aby programátori mohli označiť lexikálne jednoty tak flexibilne ako je možné, každý vyšší programovací jazyk ponúka veľkú škálu lexém. Tieto lexémy sú zvyčajne špecifikované regulárnymi výrazmi, ktoré sme si definovali v samostatnej kapitole. Je dôležité uvedomiť si, že väčšina lexém v programovacích jazykoch je špecifikovaná práve regulárnymi výrazmi, a niektoré z nich môžu obsahovať viac rovnakých vnorených výrazov. Preto niektoré jednoduché regulárne výrazy zvykneme pomenovať, a potom vieme definovať zložitejšie regulárne výrazy za pomoci názvov jednoduchších. Príklady a viac sa môžete dočítať v [3].

7.2 Konečné automaty

Ako už vieme, konečné automaty sú základný model popisujúci regulárne výrazy. Konečné automaty sme si definovali v samostatnej kapitole. Presne preto si ich v nasledujúcej kapitole uvádzame ako nástroj na tvorbu a následne implementáciu lexikálnej analýzy. Viac

o špeciálnych typoch konečných automatoch a procese lexikálnej analýzy sa môžete dočítať v [3].

V nasledujúcej kapitole si ukážeme na príklade lexikálnu analýzu. Ako sme napísali predtým, v implementácií rátame, že konečný automat je deterministický. Navrhneme konečný automat, ktorý zodpovedá definovanému jazyku a následne ho pretransformujeme na viacúrovňový konečný automat. Lexikálna analýza bude ukrátená o tvorbu tokenov, a nebude riadená syntaktickou analýzou, nakoľko sa v syntaktickej analýze nenachádza hlavný dôvod tejto práce, ktorým sú viacúrovňové princípy.

Kapitola 8

Príklad využitia viacúrovňových automatov pri lexikálnej analýze

V predošlej kapitole sme si povedali, čo je to lexikálna analýza a prečo je dôležitá. Uviedli sme si, že lexikálna analýza funguje na báze konečných automatov. To je presne situácia, na ktorej si ukážeme rozdiel v zakreslení a programovaní konečných automatov a viacúrovňových konečných automatov. Preto aby sme vedeli vykonať lexikálnu analýzu budeme potrebovať jazyk. Zadanie pre lexikálnu analýzu preberieme z [2] a postupne sa dostaneme od dekompozície, po implementáciu a interpretáciu výsledkov. Pre našu prácu, nie je za potreby tvoriť celé "tokeny", a preto ich v implementácií iba lexikálnej analýzy nebudeme vytvárať. Naším cieľom je zamerať sa na čas implementácií automatov, a teda tvorba "tokenov"nebude implementovaná.

8.1 Zadanie

8.1.1 Popis programovacieho jazyka

Jazyk IFJ17 je zjednodušenou podmnožinou jazyka FreeBASIC, ktorý stavia na legendárnom jazyku BASIC, je staticky typovaný imperatívny jazyk.

8.1.2 Všeobecné vlastnosti a dátové typy

V programovacom jazyku IFJ17 nezáleží na veľkosti písmen pri identifikátoroch a kľúčových slovách (je takzvane case-insensitive).

- Identifikátor je definovaný ako neprázdna postupnosť číslic, písmen (malých aj veľkých) a znaku podtržítka ('_') začínajúci písmenom alebo číslicou.
- Jazyk IFJ17 obsahuje ešte nižšie uvedené kľúčové slová, ktoré majú špecifický význam, a preto sa nemôžu vyskytovať ako identifikátory:
  - As, Asc, Declare, Dim, Do, Double, Else, End, Chr, Function, If, Input, Integer, Length, Loop, Print, Return, Scope, String, SubStr, Then, While.
- Rezervované kľúčové slová sú:
  - And, Boolean, Continue, Elseif, Exit, False, For, Next, Not, Or, Shared, Static, True.

- Celočíselný literál (rozsah C-int) je tvorený neprázdnou postupnosťou číslic a vyjadruje hodnotu celého nezáporného čísla v desiatkovej sústave.
- Desatinný literál (rozsah C-double) taktiež vyjadruje nezáporné čísla v desiatkovej sústave, pričom literál je tvorený celou a desatinnou časťou, alebo celou časťou a exponentom, alebo celou a desatinnou časťou a exponentom. Celá aj desatinná časť je tvorená neprázdnou postupnosťou číslic. Exponent je celočíselný, začína znakom 'e', alebo 'E', následne nepovinné znamienko '+'(plus) alebo '-'(mínus) a poslednou časťou je neprázdna postupnosť číslic. Medzi jednotlivými časťami nemôže byť iná znak, celú a desatinnú časť oddeľuje znak '.'(bodka).
- Reťazcový literál začína výkričníkom (!, ASCII hodnota 33) a nasleduje samotný reťazec obojstranne ohraničený dvojitými úvodzovkami (", ASCII hodnota 34). Tvorí ho ľubovoľný počet znakov zapísaných na jednom riadku programu. Možný je aj prázdny reťazec (""). Znaky s ASCII hodnotou vyššou ako 31 (mimo ") ide zapisovať priamo. Niektoré ďalšie znaky ide zapisovať pomocou escape sekvencií:'"', '\n', '\t', '\\'. Ich význam sa zhoduje s odpovedajúcimi znakovými konštantami jazyka FreeBASIC. Znak v reťazci môže byť zadaný taktiež pomocou všeobecnej escape sekvencie '
ddd', kde ddd je trojmiestne dekadické číslo od 001 do 255. Dĺžka reťazca nie je obmedzená (resp. iba dostupnou pamäťou). Napríklad reťazcový literál: !"Ahoj\nSve'te\\\034"

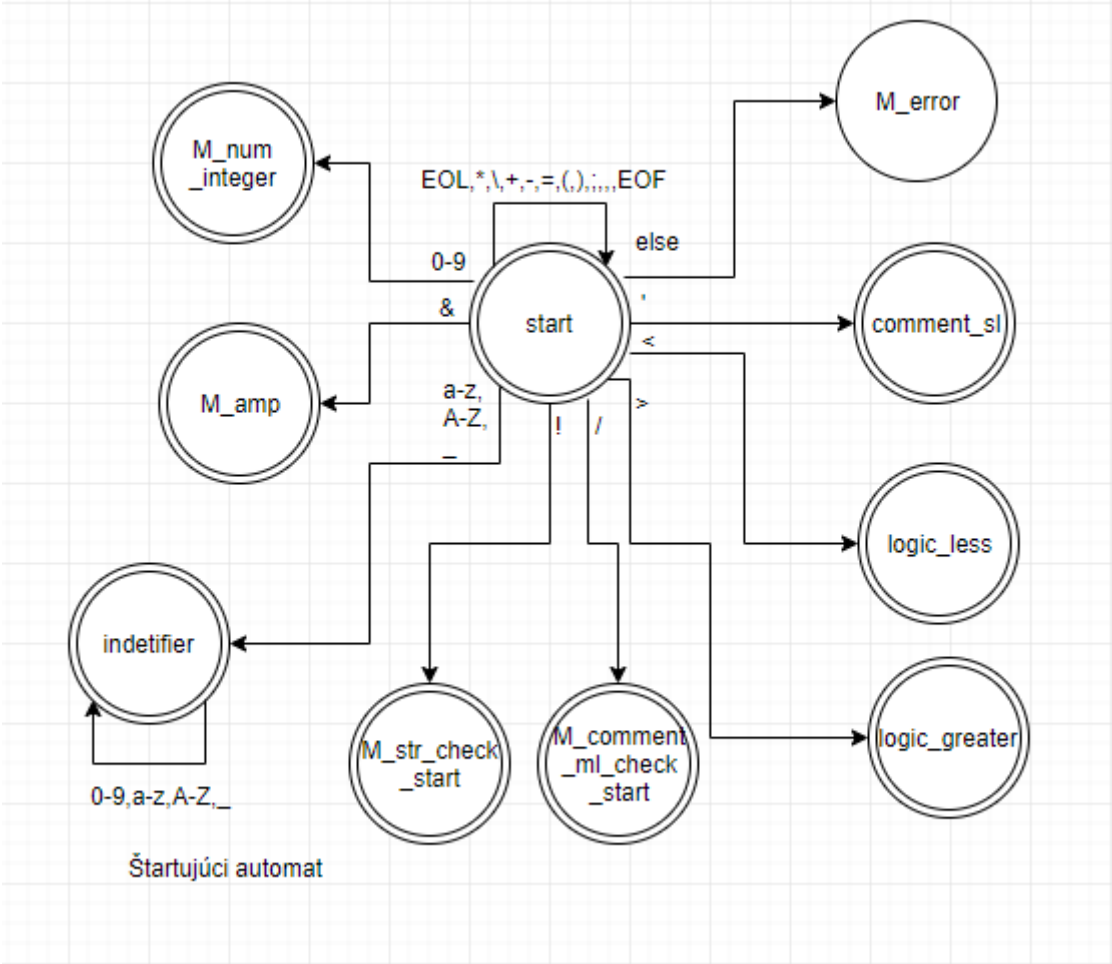
  reprezentuje reťazec Ahoj Sve'te\". Neuvažujeme reťazce, ktoré obsahujú viacbajtové znaky kódovania Unicode (napr. UTF-8).
- Dátové typy pre jednotlivé uvedené literály sú označené Integer, Double a String. typy sa používajú v definíciach premenných, a funkcií, a pri sémantických kontrolách.
- Term je ľubovoľný literál (celočíselný, desatinný či reťazcový) alebo identifikátor premennej.
- Jazyk IFJ17 podporuje riadkové a blokové komentáre tak isto ako FreeBASIC. Riadkový komentár začína znakom apostrof('", ASCII hodnota 39) a za komentár je považované všetko, čo nasleduje až do konca riadku. Blokový komentár začína dvojicou znakov '/"a je ukončený prvou nasledujúcou dvojicou znakov "'/', takže hierarchické vnorenie blokových komentárov nie je podporované.
- Celočíselné konštanty je možné zadávať aj v dvojkovej sústave (číslo začína znakmi '&b'), osmičkovej (číslo začína znakmi '&o') a v šestnáckovej (číslo začína znakmi '&h') sústave.
- Aritmetické, reťazcové a relačné operátory
  - Štandardné binárne operátory +,-,* značia sčítanie, odčítanie a násobenie.
  - Operátor / značí delenie.
  - Pre operátory <, >, <=, >=, =, <> platí, že výsledkom porovnávania je pravdivostná hodnota.

Viac o definícií jazyka IFJ17 nie je za potreby pre lexikálnu analýzu, avšak dočítať sa viac môžete v [2]. Následne si musíme vysvetliť čo takáto definícia jazyka pre nás znamená. Najlepšie sa nám bude problematika rozoberať pomocou konečných automatov, klasický konečný automat reprezentujúci tento jazyk, môžete vidieť v A. Avšak, my si v tejto kapitole

postupne rozoberieme jazykové konštrukcie a popíšeme si ich na koncepte viacúrovňového konečného automatu. Následne si ukážeme rozdieli v implementácií daných dvoch typov automatov.

8.2 Viacúrovňový automat popisujúci jazyk IFJ17

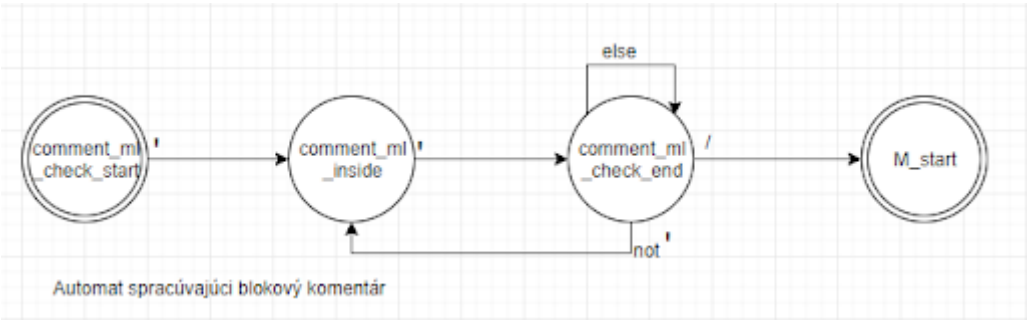
V tejto sekcií si postupne ukážeme, ako je graficky znázorenení a ako funguje, viacúrovňový konečný automat, ktorý prijíma jazyk IFJ17.



Obr. 8.1: Na tomto obrázku vidíme štartujúci automat.

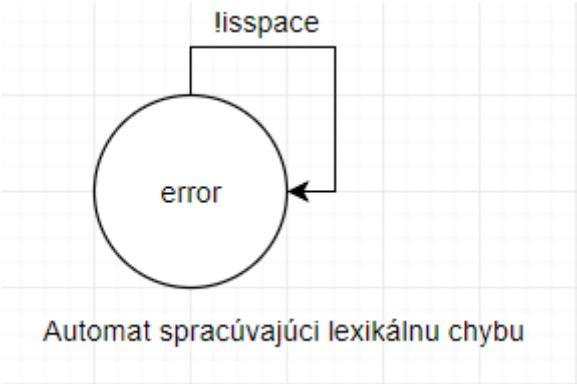
Na obrázku môžeme vidieť štartujúci automat. Zo zadania jazyku IFJ17 vieme, že podporujeme klasické operátory ako je naznačené prechodom pomocou znakov < a > do stavov logic_less a logic_greater, kde sa ďalej spracúvajú. Taktiež môžeme vidieť všetky znaky, ktoré sú pri prechode zo stavu start do stavu start. tieto znaky sa spracúvajú priamo v tomto stave, respektíve sa vracajú syntaktickému analyzátoru ako lexéma(ďalej nebudeme spomínať syntaktickú analýzu ani tvorbu tokenov, nakoľko to nie je predmetom tejto práce). Z definície jazyka môžeme taktiež vidieť, že pomocou číslice 0-9 sa dostávame zo stavu start

do automatu M_num_integer, kde sa ďalej spracúva číslo. Môžeme vidieť, že pomocou ampersandu (znak '&') sa dostávame do automatu M_amp, v ktorom sa ďalej spracúva číslo zadané v inej sústave. Identifikátor sa spracúva v stave identifier, do ktorého sa dostanem ak príjememe znaky a-z, alebo A-Z, alebo podtržník (znak '_'). V tomto stave ďalej spracujeme identifikátor, ktorý môže obsahovať v rámci jeho tela ešte aj číslice 0-9. Na zistenie, či sa jedná o jedno zo špeciálnych slov zo zadania sa implementuje pseudo binárny strom, kde ak sa nájde zhoda, tak vrátime číselný kód kľúčového slova, a ak sa nenájde zhoda, vraciame identifikátor. Môžeme vidieť, že do automatu, ktorý spracúva reťazce sa dostaneme pomocou znaku '!', ktorý je začiatočným znakom reťazcu. Aby sme mohli používať blokové komentáre ako je zo zadania zrejmé, že ich jazyk IFJ17 podporuje, musíme sa pomocou znaku '/', prejsť do automatu, ktorý spracúva blokové komentáre. Taktiež, aby sme mohli použiť jednoriadkové komentáre, musíme po prijatí znaku '"prejsť do stavu, ktorom sa komentár ďalej spracuje. Tým sa dostávame na posledný prechod, ktorý nám popisuje, že ak automat prečíta hocičo iné, tak nastáva lexikálna chyba a musíme prejsť do automatu, kde sa lexikálne chyby spracúvajú.



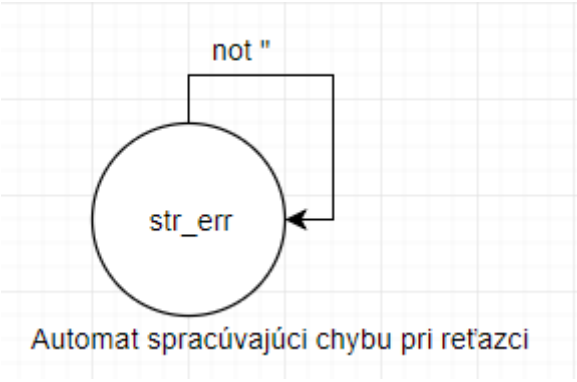
Obr. 8.2: Na tomto obrázku vidíme automat, ktorý spracúva blokový komentár.

Akonáhle sa dostaneme do automatu, ktorý spracúva blokový komentár vieme, že začal správnym znakom. Presne pre to, ďalej kontrolujeme či komentár pokračuje tak ako je definované pre náš jazyk. Teda automat musí prijať znak '"ihneď po znaku '/', inak končí lexikálnou chybou. Keď sa automat presunie do ďalšieho stavu, ostáva v ňom, až pokým neprečíta znak '", ktorým začína ukončujúca dvojica znakov, na ukončenie blokového komentáru. Avšak v ďalšom kroku musí hneď ako ďalší znak prečítať druhý znak z dvojice a to znak '/', ak prečíta iný znak, ostáva v danom stave. Ak prečíta znovu znak '"musí znovu skontrolovať ukončenie blokového komentáru, preto sme tam pridali prechod pre prečítanie niečoho iného. Po úspešnom prečítaní dvojice znakov ''/', vieme, že blokový komentár končí, a automat sa môže presunúť znovu do štartujúceho automatu.



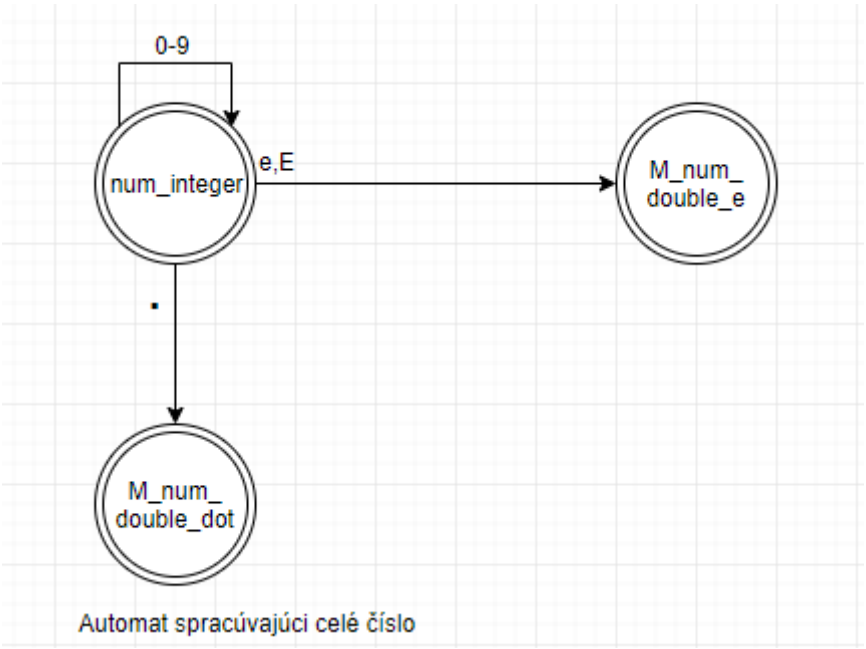
Obr. 8.3: Na tomto obrázku vidíme automat, ktorý spracúva lexikálnu chybu.

Automat na spracovanie lexikálnej chyby funguje veľmi jednoducho. Z definície jazyka je zrejmé, že lexéma končí až takzvaným bielym znakom. Presne preto ostávame v chybovom stave, až kým nenarazíme na taký znak, následne môžeme odoslať lexikálnu chybu. Tento proces robíme preto, aby sme mohli v analýze ďalej pokračovať a na jeden prechod zaznamenať viac ako jednu chybu. Keby hneď pri narazení na chybu ukončíme chod programu, tak nevieme odhaliť ostatné chyby v tom istom prechode.



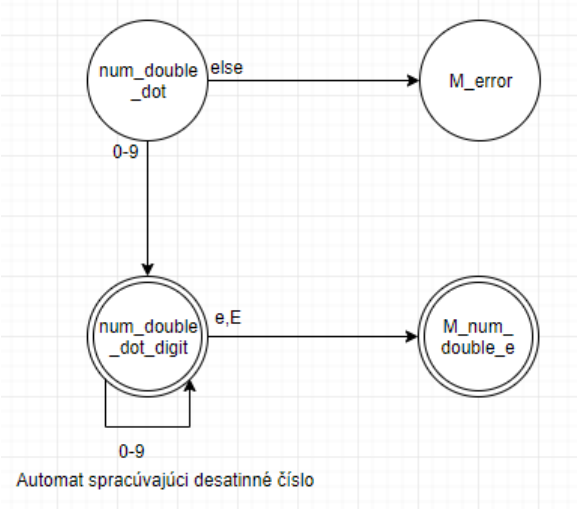
Obr. 8.4: Na tomto obrázku vidíme automat, ktorý spracúva chybu v reťazci.

Automat na spracovanie chyby, ktorá nastala pri spracovaní reťazcu funguje podobne. Ostáva v danom stane až kým nenarazí na znak ukončujúci reťazec. Dôvod, prečo to tak robíme, je totožný ako pri automate, ktorý spracúva ostatné lexikálne chyby.



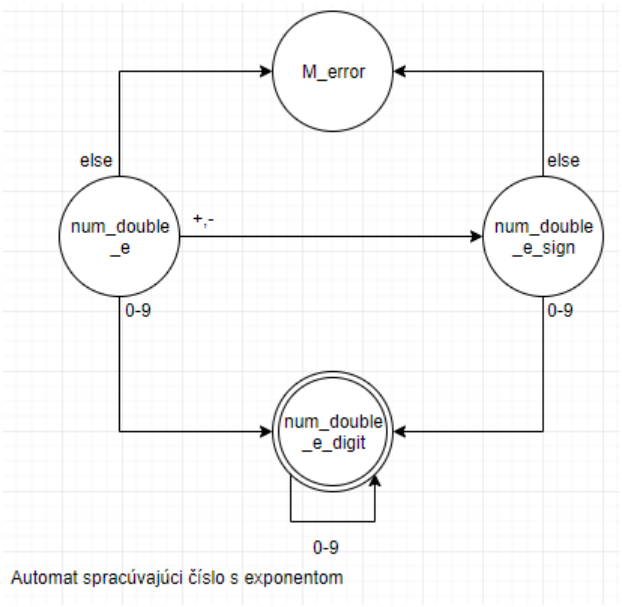
Obr. 8.5: Na tomto obrázku vidíme automat, ktorý spracúva celé číslo.

V tomto automate spracúvame číslo, číslo môže byť buď celé, desatinné alebo s exponentom. Ak sa jedná o celé číslo neobsahujúce ‘.’(bodku) alebo ‘e’/’E’(exponent), tak pokiaľ prečíta automat iba číslice 0-9, skladá z nich celé číslo, ak prečíta iný znak, tak zložené číslo našiel. Tú situáciu nám značí prechod zo stavu num_integer do seba samého. Ďalší prechod do automatu M_num_double_e nastane ak automat hneď po čísle prečíta znak ‘e’alebo ‘E’, v tomto prípade vieme, že sa bude jednať o číslo s exponentom a na jeho spracovanie máme vytvorený ďalší automat, ktorý si popíšeme neskôr. Nakoniec ak sa bezprostredne za číslom nachádza znak ‘.’, tak vieme, že sa bude jednať o desatinné číslo, ktoré budeme ďalej spracovávať v samostatnom automate M_num_double_dot. Ako sme písali vyššie, automat je koncový nakoľko za číslom nemusí byť žiadny biely znak, z toho vyplýva, že v tej situácií máme zostavené výsledné číslo, a ďalším znakom bude začínať nová lexéma.



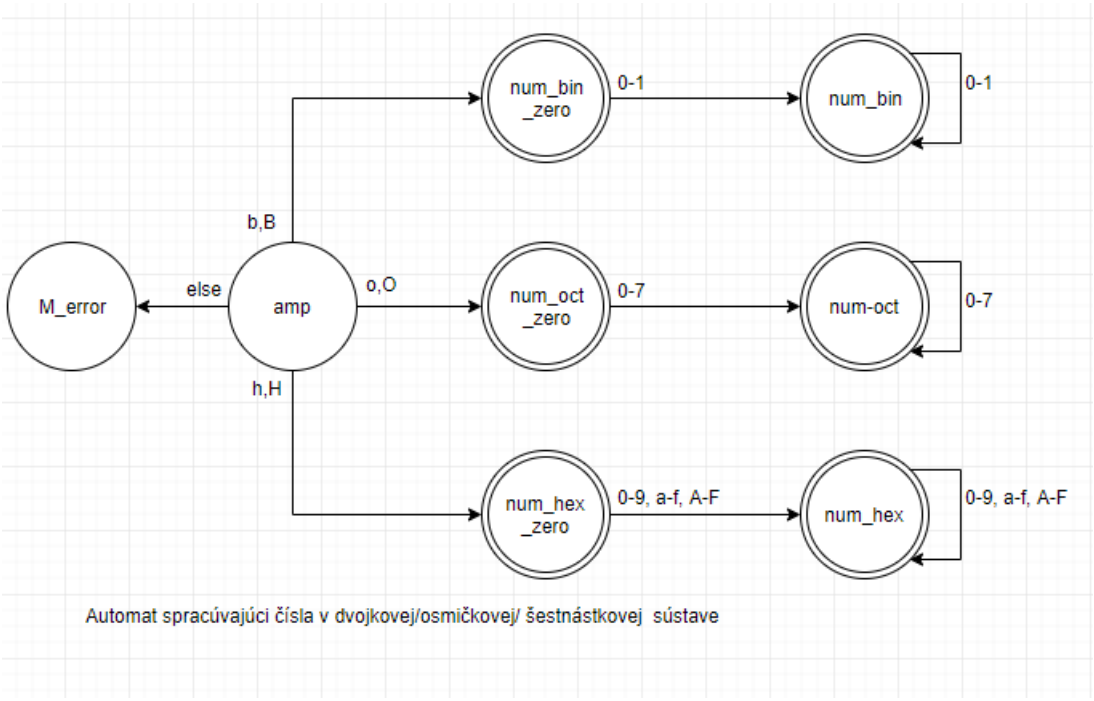
Obr. 8.6: Na tomto obrázku vidíme automat, ktorý spracúva desatinné číslo.

Z definície jazyka vieme, že pri desatinnom čísle, musí bezprostredne po desatinnej bodke nasledovať ďalšia číslica. Presne preto ak automat prečíta niečo iné ako číslicu, prechádza do automatu M_error, ktorého práca je popísaná vyššie. Z definície jazyka taktiež vieme, že ak automat prečíta po bodke číslo, môže nasledovať buď exponent, alebo ďalšie číslice. Tieto dve situácie sú popísané prechodmi zo stavu num_double_dot_digit do seba samého alebo do automatu M_num_double_e, ktorý spracúva čísla zapísané exponentom. Ako môžeme vidieť tak stav num_double_dot_digit je koncový, a to z rovnakého dôvodu aký bol opísaný vyššie. Za číslom nemusí nasledovať žiadny biely znak.



Obr. 8.7: Na tomto obrázku vidíme automat, ktorý spracúva desatinné číslo s exponentom.

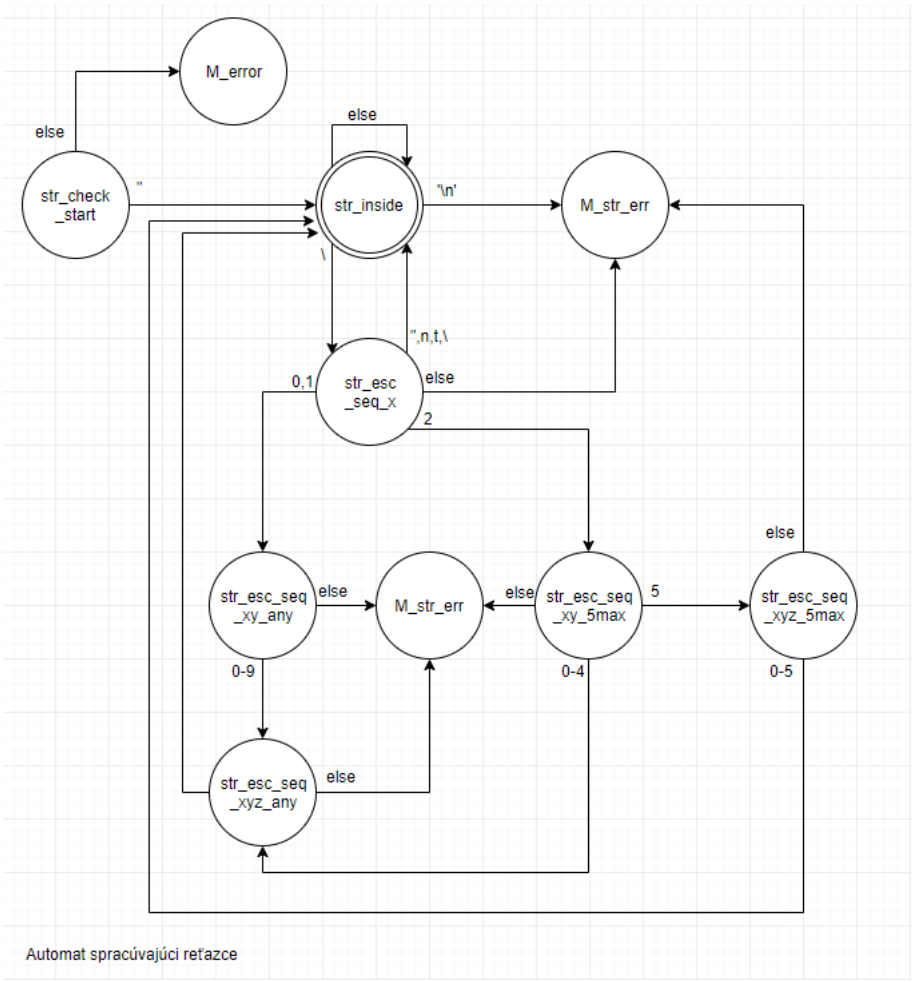
Z definície jazyka vieme, že bezprostredne za znakom exponentu sa môže, ale nemusí nachádzať znak + alebo -. Toto nám znázorňuje prechod zo stavu num_double_e do stavu num_double_e_sign za pomoci znaku + alebo -. Taktiež vieme, že ak sa po znaku nenachádza znamienko, ale nasleduje číslica 0-9, tak je to validne zapísané exponenciálne číslo, túto situáciu nám znázorňuje prechod zo stavu num_double_e do stavu num_double_e_digit, za prečítania číslice 0-9. Ak by v čísle po znaku 'e'/'E'bezprostredne nenasledovala ani jedna situáciu jedná sa o lexikálnu chybu, ktorú spracúva automat M_error. Môžeme vidieť, že stav num_double_e_digit je koncový, a to pre to isté, ako v predošlých automatoch, za korektne zapísaným číslom nemusí nasledovať biely znak.



Obr. 8.8: Na tomto obrázku vidíme automat, ktorý spracúva číslo v 2/8/16-kovej sústave.

Z definície jazyka vieme, že jazyk IFJ17 prijíma zápis čísla nie len v desiatkovej sústave, ale aj v dvojkovej, osmičkovej a šestnástkovej sústave. Pre túto situáciu sme si urobili zvlášť automat. Dostaneme sa do neho po prečítaní znaku '&'(apostrof). Vieme, že bezprostredne po apostrofe musí nasledovať buď, jeden z dvojice znakov 'b'/'B', alebo 'o'/'O', alebo 'h'/'H'. Ak automat prečíta niečo iné, jedná sa o lexikálnu chybu, čo nám značí prechod do automatu M_error. Pri správnom zápise čísla vieme, že dvojková sústava môže obsahovať iba číslice 0 a 1, osmičková sústava obsahuje iba číslice 0-7, a šestnástková sústava obsahuje číslice 0-9, a písmená a-f. Do dvojkovej vetvy automatu, horná vetva, sa dostaneme vtedy, ak automat prečíta znak 'b'alebo 'B'bezprostredne po ampersande. Môžeme vidieť, že ak automat následne prečíta niečo iné ako číslice 0 alebo 1, tak sa ukončuje korektne a značí nám to číslo 0. Avšak, ak automat prečíta korektnú číslicu z dvojkovej abecedy, tak sa presúva do stavu, kde sa dané dvojkové číslo môže zväčšovať. Taktiež končí v momente keď prečíta iný znak. Stredná, osmičková, vetva funguje na rovnakom princípe s rozdielom, že neprijíma

iba znaky dvojkovej abecedy, ale aj znaky osmičkovej abecedy. Spodná, šestnástková vetva funguje principiálne rovnako, avšak prijíma znaky šestnástkovej abecedy.



Obr. 8.9: Na tomto obrázku vidíme automat, ktorý spracúva reťazec.

Ako poslednú časť z definície jazyka, musíme vyriešiť zápis reťazcov. Vieme, že keď sa dostanem do tohoto automatu, musí pokračovať reťazec nakoľko sme prečítali znak '!'. Reťazec pokračuje znakom '"'čo nám značí prechod zo stavu str_check_start do stavu str_inside. Ak by automat prečítal iný znak, prechádza do automatu M_error, ktorý spracuje zvyšok chyby. Akonáhle sa nachádzame v tele reťazca vieme nasledovné. Ak prečítame znak nového riadku prechádzame do automatu M_str_err, nakoľko to nie je korektný zápis reťazca, tento prechod nám značí prechod zo stavu str_inside do automatu M_str_err vtedy, ak automat prečíta znak nového riadku. Ak automat prečíta znak '\'očakávame, že sa bude jednať o escape sekvenciu, túto situáciu nám značí prechod zo stavu str_inside do stavu str_esc_seq_x. Ak v rámci tela reťazcu prečíta automat iný znak, ostáva v danom stave. Korektne zapísaný reťazec je ukončený v stave str_inside prečítaním znaku '"'. Ďalej musíme vyriešiť zápis escape sekvencií. Akonáhle sa nachádzame v stave str_esc_seq_x očakávame buď číslicu 0,1 alebo 2, avšak prijímame aj znaky '"', 'n', 't', '\', nakoľko z definície jazyka vyplýva, že dané znaky môžeme zapisovať priamo a nie je za potreby písať ich

ASCII kód. V tomto stave nemôžeme už prijať nič iné, takže ak by automat prečítal iný znak, prechádza do automatu M_str_err, ktorý spracuje chybu ďalej. Z definície vieme, že escape sekvencie môžu byť zapísané priamo číslom a to od 001 do 255, tento fakt nám rozdeľuje automat ďalej, nakoľko po prijatí prvej číslice 0 alebo 1 môže nasledovať hocijaká číslica, ale po prijatí číslice 2 môže nasledovať iba číslica z rozmedzia 0-5. Začneme situáciou, v ktorej automat prečíta ako prvú číslicu escape sekvencie číslicu 0 alebo 1. Takto sa automat dostáva do stavu str_esc_seq_xy_any, v ktorom môže prečítať číslicu od 0-9, čo naznačuje korektný zápis a automat sa nasledovne posúva do stavu str_esc_seq_xyz_any. V tomto stave môže automat prečítať buď korektne jednu z číslic v rozmedzí 0-9, alebo niečo iné, kedy by musel prejsť do automatu M_str_err na spracovanie chyby. Ak by automat prečítal korektnú číslicu, tak by následne prešiel do stavu str_inside s tým, že by zapísal korektnú escape sekvenciu. Ak by automat ako prvú číslicu escape sekvencie prečítal číslo 2, dostáva sa do stavu str_esq_seq_xy_5max, v ktorom môžu nastať 3 situácie. V prvej situácií automat prečíta číslicu 0-4, nakoľko v tomto prípade môže byť posledná číslica ľubovoľná z 0-9, prechádza do stavu str_esc_seq_xyz_any, ktorý funguje ako popísaný vyššie. Ak by však prečítal číslicu 5, tak posledná číslica môže byť iba z rozmedzia 0-5, preto musí prejsť do samostatného stavu pomenovaného str_esc_seq_xyz_5max, v ktorom by ďalej buď prečítal 0-5 a vrátil by sa do stavu str_inside po korektnom spracovaní escape sekvencie, alebo by prešiel do automatu M_str_err, ktorý by vyriešil chybovú situáciu. Ešte existuje varianta, že automat po prijatí prvej číslice neprijme ďalšiu, v tom prípade prechádza do stavu M_str_err na spracovanie chyby.

Daný viacúrovňový automat nám popisuje vyššie špecifikovaný jazyk IFJ17. Ďalej si ukážeme spôsob implementácie lexikálnej analýzy špecifikovanej konečným automatom z A, a viacúrovňovým konečným automatom popísaným vyššie.

8.3 Implementácia konečného automatu

V tejto sekcií sa pozrieme na princíp implementácie konečného automatu, ktorý nám popisuje jazyk IFJ17 a je zobrazený v A. Princíp je nasledovný, syntaktický analyzátor zavolá funkciu get_next_token(), v tomto momente začína pracovať lexikálny analyzátor na hľadní nasledujúceho tokenu zo zdrojového programu. Celý konečný automat je založený na jednom veľkom switch-e. Implementácia konečného automatu v pseudokóde:

```
procedure get_next_token(){
      #definicie
      int state = #stav v ktorom sa automat nachadza
      opakuj pokial neprecita EOF #koniec suboru
            .
            . #prevediem operacie, napriklad kontrola,
            . #ci mame znak v bufferi z predosleho
            . #citania, alebo zmena znakov na jednotne,
            . #ak sa nejedna o retazec


            switch(state){ #v tomto kroku musime vyriesit v jednom switch-e
                           #vsetky stavy
                  case def:
```

```
                        .
                        .
                        .
                case error:
                        .
                        .
                        .
                case num_integer:
                        .
                        .
                        .  #symbolicky naznacujeme viac stavov
                        .
                case logic greater:
        }
}

int is_keyword(){
        #pseudo binarny strom na vyhladavanie klucovych slov
}
```

Ako môžeme vidieť, takýto switch, ktorý má obsahovať všetky stavy je veľmi rozsiahly. Taktiež, v každom stave musíme previesť porovnania a operácie, ktoré značia prechody do ďalších stavov. Pre podrobný popis, ako prepísať grafické znázornenie konečného automatu do programovacieho jazyka, si môžete prečítať v [3] alebo v [4]. Následne si v skratke ukážeme, ako je implementovaný viacúrovňový konečný automat, a potom urobíme porovnanie.

8.4 Implementácia viacúrovňového konečného automatu

V tejto sekcií si opíšeme princíp implementácie viacúrovňového konečného automatu, ktorý je znázornení vyššie. Princíp kedy sú jednotlivé automaty implementované pomocou konštrukcie switch, zostáva rovnaký. Hlavná zmena však je, že jednotlivé automaty sú implementované pomocou funkcií. Máme dve možnosti ako začať, buď si deklarujeme pár globálnych premenných ako napríklad state, alebo si budeme posielať state medzi automatmi. My zvolíme jednoduchšiu variantu kde si na začiatku deklarujeme globálne premenné. Následne pokračujeme definíciou funkcií, ktoré nám budu popisovať jednotlivé automaty. Pseudo kód celkovo by vyzeral nasledovne:

```
#deklaracia globalnych premennych
#definicia funkcii, ktore popisuju automaty
.
.
.
int M_start(){ #funkcia popisujuca startovaci automat
        # zakladne operacie ako: kontrola bufferu
        switch(state){ #jednotlive stavy automatu
                case start:

                case comment_sl:
```

```
                .
                .
                .
                case identifier:
        }
}
```

Takáto jednoduchá funkcia nám popisuje jeden automat. ďalší automat nám popisuje ďalšia funkcia.

```
int M_err(){
        pokial neprecitam EOF
                ak precitam biely znak
                        vrat lexikalnu chybu
}
```

V tomto prípade môžeme vidieť, že ak sa jedná o triviálny automat, konštrukcia switch-u nie je za potreby. Prechod z automatu do automatu funguje na báze volania funkcie, ktorá daný automat popisuje. Takto sa nám konečný výsledok vynorí z poslednej úrovne úplne hore, a tvorcov jednotlivých automatov nezaujíma ako je ďalší automat implementovaný. Jednoducho ho zavolá, a počká kým dostane výsledok. Princíp sa opakuje, pre jednotlivé automaty vytvárame jednotlivé funkcie. V ďalšej sekcií si ukážeme prečo je ten prístup lepší.

8.5 Porovnanie implementácií

Ako sme mohli vidieť v predošlej sekcii, viacúrovňový prístup, aj keď môže byť náročnejší na pochopenie, tak nám umožňuje program jednoducho modulovať a rozdeliť prácu medzi viac ľudí. Človek implementujúci jeden automat, nepotrebuje vedieť ako je implementovaný druhý, jednoducho zavolá funkciu, ktorá mu vráti hodnotu. Týmto spôsobom je práca rozdelená a zároveň môže byť skrytá voči ostatným a nikomu to neprekáža. Jednotlivé automaty, môžu byť dokonca definované v iných súboroch a iba sa pripoja. Taktiež môžeme vidieť, že programátor nemusí tvoriť dlhé nepriehľadné konštrukcie. Kód je celkovo prehľadnejší. Nevýhodou je, že zakreslenie automatu a pochopenie, môže byť o niečo náročnejšie, nakoľko je to nový prístup, na ktorý si je treba zvyknúť. Rýchlosť lexikálnej analýzy sme neporovnávali, nakoľko rýchlosť kompilácie je viac obmedzovaná syntaktickou analýzou, ktorá nie je témou tejto práce. Ďalšie veľké plus majú viacúrovňové automaty, že sú jednoducho rozšíriteľné. Jednoducho pridáte ďalší automat, popíšete ho nezávislou funckiu a automat zvyšuje na sile. Presne tento princíp si ukážeme v nasledujúcich sekciách.

Kapitola 9

Paralelné pravo-lineárne gramatiky

V tejto sekcií si priblížime problematiku paralelizmu a prepisujúcich systémov. Viac k tejto téme, sa môžete dočítať v [7]. Berieme v úvahu n-paralelné pravo-lineárne jazyky z [6]. Viac informácií v [5].

Paralelné pravo-lineárne gramatiky dostaneme ak skombinujeme 3 rysy.

1. Každý derivačný krok nahradzuje neterminály, teda derivačné kroky prebiehajú paralelne.
2. V každej skupine derivačných krokov je nahradený vopred vydedukovaný počet neterminálov.
3. Každé pravidlo je lineárne.

Základná definícia:

Definícia 9.0.1 *Abeceda je neprázdna konečná množina symbolov. Pravo-lineárna gramatika(PRLG) je štvorica G = (N, T, S, P), kde:*

- *N je abeceda neterminálov*
- *T je abeceda terminálov*
- $S \in N$ *je symbol vety*
- $P \subseteq N \times (T^* N T^* \cup T^*)$ *je konečná množina lineárnych pravidiel*

  $P \subseteq N \times (T^* N \cup T^*)$ *je konečná množina pravo-lineárnych pravidiel*

Pravidlá môžu nadobúdať jednu z troch foriem:

1. $[S \to X_1 \ldots X_n]$, $X_i \in N_i$, $1 \leqq i \leqq n$
2. $[X_i \to a_1, \ldots, X_n \to a_n]$, $X_i \in N_i$, $a_i \in T^*$, $1 \leqq i \leqq n$
3. $[X_i \to a_1 Y_1, \ldots, X_n \to a_n Y_n]$, $X_i Y_i \in N_i$, $a_i \in T^*$, $1 \leqq i \leqq n$

Definícia 9.0.2 *Pre n > 0, je n-paralelná pravo-lineárna gramatika(n-PRLG) (n+3)-tica, G = (N1, ..., Nn, T, S, P), kde:*

- N_i *sú vzájomne nesúvisiace abecedy neterminálov,* $1 \leqq i \leqq n$

- *T je abeceda terminálov*
- $S \notin N_1 \cup \cdot \cdot \cdot \cup N_n$
- *P obsahuje možnosti pravidiel:*
  1. $S \to X_1 \ldots X_n$, $X_i \in N_i$, $1 \leqq i \leqq n$
  2. $X \to wY$, $X, Y \in N_i$ *for some i,* $1 \leqq i \leqq n$, $w \in T^*$
  3. $X \to w$, $X \in N$, $w \in T^*$

Z definície vyplýva, že sa neterminál môže nachádzať buď samostatne, alebo napravo od terminálu. Tým dostávame pravo rozširujúce sa vety.

Derivačný krok pre x, y $\in$ (N $\cup$ T $\cup$ S)*, x $\Rightarrow$ y, vtedy a len vtedy ak

1. x = S a S $\to$ y $\in$ P
2. $x = y_1X_1y_2X_2 \ldots y_nX_n$

   $\downarrow \quad \downarrow \ldots \downarrow$

   $y = y_1x_1y_2x_2 \ldots y_nx_n$

   $y_i \in T^*$, $x_i \in T^*N \cup T^*$, $X_i \in N_i$, $X_i \to x_i \in P$, $1 \leqq i \leqq n$

L (G) = w $\in$ T* : S $\Rightarrow^+$ w, $\Rightarrow^+$ definované ako zvyčajne. R_n = L (G) : G je n-PRLG.

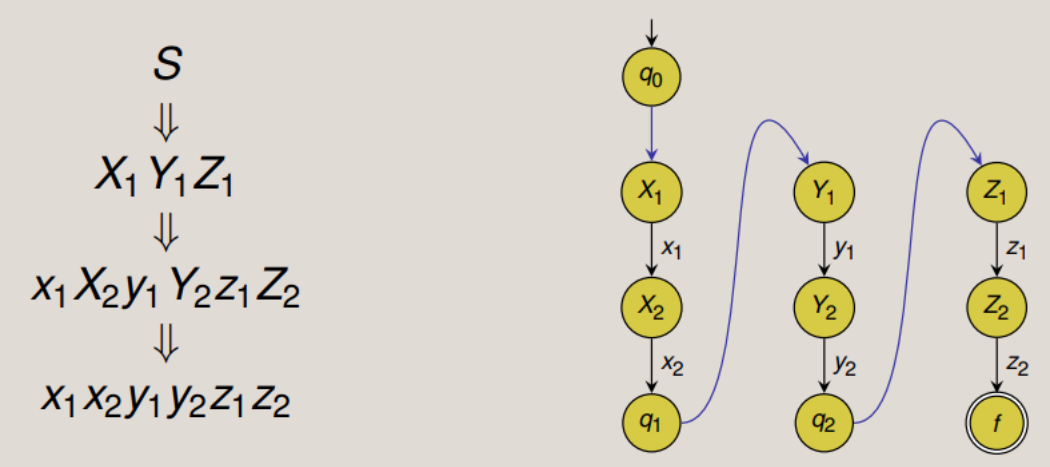
9.1 Príklad

Majme G = (A, B, a, b, S, P) je 2-PRLG, kde P obsahuje nasledovné pravidlá:

- S $\to$ AB
- A $\to$ aA
- A $\to$ a
- B $\to$ bB
- B $\to$ b

Berieme v úvahu nasledovný deriváciu G:
S $\Rightarrow$ AB $\Rightarrow$ aAbB $\Rightarrow$ aaAbbB $\Rightarrow$ a^3b^3
L (G) = a^nb^n : n $\geqq$ 1

Majme G je 3-PRLG. Existuje sebaregulujúci automat M kde L (G) = L (M). Myšlienka ako to dokázať je znázornená na nasledujúcom obrázku.



Obr. 9.1: Obrázok prevzatý z [5].

Pre pochopenie si musíme ujasniť sebaregulujúce automaty. Tými sa budeme zaoberať v nasledujúcej kapitole. Ku príkladu ako je na obrázku sa vrátime novo navrhnutým modelom, viacúrovňových sebaregulujúcich automatov, pre ktorý máme popísaný koncept vo vlastnej kapitole.

Kapitola 10

Sebaregulujúce konečné automaty

Myšlienka sebaregulácie (viac v [5]) spočíva v tom, že pri prevedení nejakého pravidla automatu, sa nám vytvorí nové pravidlo, takzvané obrácajúce pravidlo. Toto pravidlo sa musí vykonať, keď sa dostaneme, takzvaného obrácajúceho stavu. Majme príklad kde máme nasledovné pravidlá:

1. sa → s, {4}
2. sb → s, {5}
3. s → t, {6}
4. ta → t, ∅
5. tb → t, ∅
6. t → f , ∅

Pravidlá 1-3 sú zapísané a je ich možné vykonať od začiatku. Pravidlá 4-6 sú vygenerované aplikovaním pravidla, ktoré ich generuje. Stav f je konečný stav. Stav t je obracajúci stav automatu. V tomto prípade nám automat prijíma jazyk L (M) = ww : w ∈ a, b*.

Sebaregulujúci konečný automat je definovaný nasledovne:

Definícia 10.0.1 *Majme $N = (Q, \Sigma, \delta, q_0, F)$ je konečný automat. Sebaregulujúci konečný automat je teda trojica $M = (N, q_t, R)$, kde:*

- $q_t \in Q$ *je obracajúci stav*
- $R \subseteq \psi \times \psi$ *je konečný vzťah nad abecedou pravidiel*

Hlbšie si popíšeme celý koncept s pridaním viac úrovňosti konečných automatov v nasledujúcej kapitole. Ukážeme si vytvorený koncept založený na príklad z minulej kapitoly a taktiež si navrhneme možnú implementáciu daného príkladu.

Kapitola 11

Koncept viacúrovňových sebaregulújucich konečných automatov

Hlavná myšlienka znie:

Nahradiť zreťazené spracovanie vstupu pomocou viacerých úrovní, paralelným spracovaním, a tým zvýšiť silu automatov.

11.1 Príklad

Chceme prijať jazyk L(G) = $a^x b^y c^z$: x/y/z $\geqq$ 1, klasickým zreťazeným spracovaním, by sme postupne spracovali prijaté neterminály, a vytvorením takzvaného obracajúceho stavu by sme použili predtým vytvorené pravidlo. Tento prístup je neefektívny a nie veľmi modulovateľný. Návrh znie, použiť spracovanie vo viacerých úrovniach pre zefektívnenie a možné rozdelenie práce pre viacerých ľudí. Pre tento príklad musíme vyriešiť 4 problémy.

1. Vytvorenie obracajúcich pravidiel a obracajúcich stavov, v tomto stave sa použije predtým vytvorené pravidlo, a tým sa ukončí spracovanie neterminálu.
2. Rozdelenie vstupu na potrebný počet jednotlivých vstupov. Tento krok sa bude diať na prvej úrovni (L1).
3. Skladanie výstupu z výstupov každej vetvy. Tento krok sa bude diať na poslednej úrovni (L3).
4. Samotné škálovateľné spracovanie oddelených vstupov. Tento krok budeme riešiť na strednej úrovni (L2).

Riešenie prvého problému:

Majme pravidlá:

1. A $\rightarrow$ aA{4}
2. B $\rightarrow$ bB{5}
3. C $\rightarrow$ cC{6}

4. A → a
5. B → b
6. C → c

Pravidlá 4,5,6 sú obracajúce pravidlá, ktoré sa musia previesť v danom pracujúcom stave. Tieto pravidlá nemáme vopred vytvorené, vytvorania sa po prevedení pravidla, ktoré ich tvorí. V našom prípade sa vytvoria pri prvom prechode strednou úrovňou automatov.

Riešenie druhého problému:

Vytvoríme si zástupný oddeľovač, ktorý sa nebude nachádzať v prijímanej abecede. Teda modifikujeme pôvodnú definíciu, kde : $\notin \Sigma$. Na úrovni L1 nastane táto postupnosť: S $\Rightarrow$ A:B:C $\Rightarrow$ xA0yB1zC2 x/y/z-počítadlo pre každý neterminál x/y/z$\geqq$1. V tomto prípade dostaneme na úrovni L1 3 reťazce, každý reťazec dostaneme nájdením rozdeľovača a pridaním postupne inkrementovaného čísla na koniec znakov, a nakoniec pridáme počet prechodov automatov ako prvé číslo na začiatku reťazca. Celkový počet reťazcov si uložíme aby posledný, konkatenujúci automat vedel, koľko reťazcov musí prijať, pred začatím konkatenácie, nakoľko je celý proces možné spustiť paralelne.

Riešenie tretieho problému:

Na poslednej, tretej, úrovni čakáme, kým dostanem toľko reťazcov, na koľko bol začiatočný rozdelený. Akonáhle máme všetky nasleduje proces kde najskôr z každého odstránime prvý znak, ktorý nám naznačuje, že sa počítadlo dostalo na minimum a máme teda finálny reťazec bez neterminálov. Nakoľko pri paralelnom spracovaní nevieme v akom poradí reťazce dostaneme, museli sme si ich na začiatku očíslovať. Výsledný reťazec začneme nájdením reťazca, ktorý končí číslom 0, číslo odstránime a zvyšok reťazca nám udáva začiatok výsledného reťazca. Postupne hľadáme väčšie čísla, číslo odstránime a reťazec pridáme na koniec finálneho reťazca. tento proces opakujeme, kým neprejdeme všetky menšie reťazce.

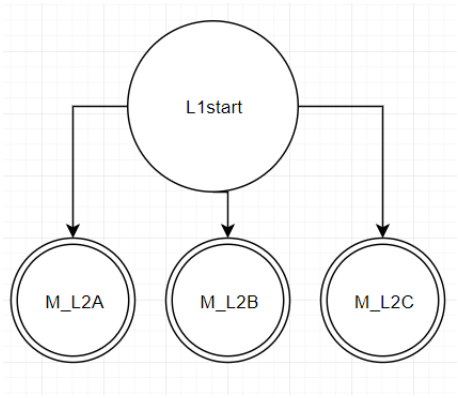
Riešenie posledného problému:

Vďaka rozdeleniu a pridaniu počítadla na prvej úrovni máme tento krok jednoduchší. Počítadlo nám jednoducho zvýši silu automatu. Na začiatku má každá vetva iba jedno pravidlo a to buď 1,2 alebo 3. Po prevedení tohto pravidla sa vytvorí obracajúce pravidlo, ktoré sa bude musieť aplikovať, keď sa dostaneme do obracajúceho stavu. Toto pravidlo vytvoríme iba raz, nakoľko by sa nám potom iba prepísalo.

11.2 Implementácia

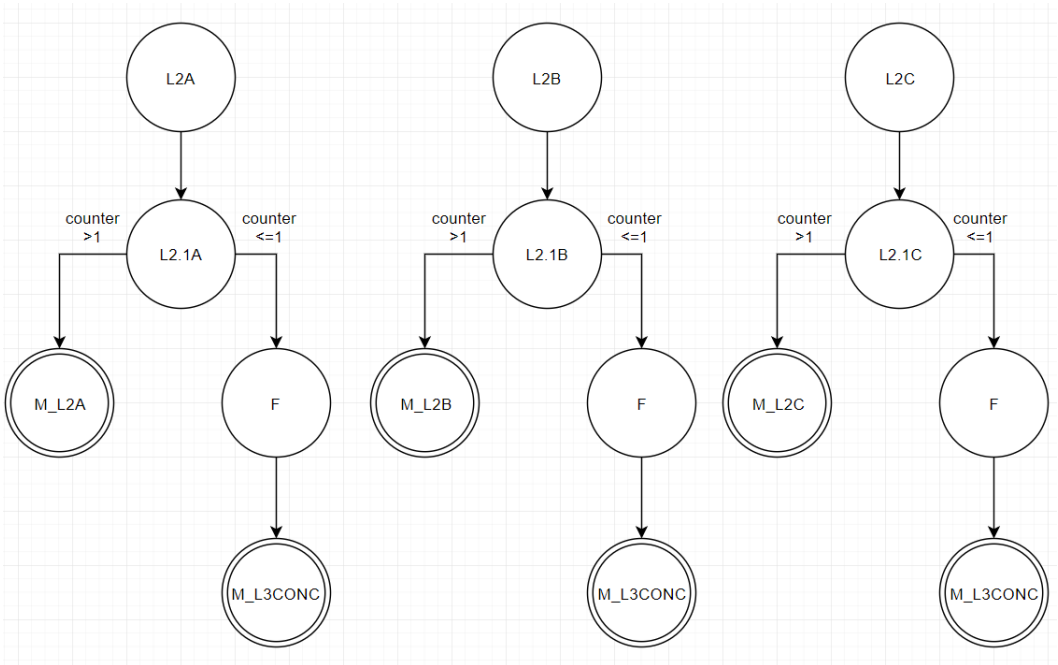
Na začiatku druhej úrovni decrementujeme počítadlo a prevedieme prvé možné pravidlo. V ďalšom kroku druhej úrovni na základe počítadla rozhodneme, či proces opakujeme, alebo ideme dokončiť jednotlivé reťazce. V stave F sa nachádzame v obracajúcom stave, kde musíme použiť novovytvorené pravidlo, finalizujeme jednotlivé reťazce aplikovaním pravidiel, ktoré boli vytvorené aplikáciou prvých pravidiel. Po vytvorení jednotlivých reťazcov prechádzame do finálnej úrovni, kde posledný automat vytvorí výsledok. Ak by sme pridali rovnaký automat na druhú úroveň a pridali ku nemu nové pravidlo, dostávame celkovo silnejší automat kde jeho výsledok bude nadmnožina predošlého výsledku.

Poznámky k značeniu. Ak sa na za začiatku názvu nachádza ‘M_’jedná sa o prechod do automatu, a to konkrétne do stavu pomenované ako zvyšok názvu automatu. Rozhodnutie o opakovaní značíme dvoma vetvami pomocou counter $>$ 1 alebo counter $\leqq$ 1. Dvojitá kružnica značí, že daný stav alebo automat je konečný, teda ak sa jedná o automat, tak sa v ňom musí nachádzať cesta do koncového stavu.



Obr. 11.1: Automat prvej úrovni.

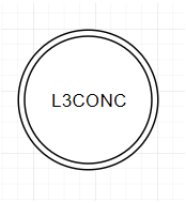
Môžeme vidieť, že sa zo štartujúceho stavu vetvíme do 3 vetiev paralelne, v štartujúcom stave prevedieme všetky potrebné operácie popísané vyššie.



Obr. 11.2: Automaty druhej úrovni.

Na tomto obrázku môžeme vidieť, jednotlivé vetvy spracovania. Symbolicky vidíme naznačenie počítadla, a tým pádom vieme, kedy proces ukončiť. Taktiež môžeme podotknúť,

že schéma jednotlivých vetiev vyzerá skoro totožne, teda je veľmi jednoducho škálovateľná, a v podstate nám stačí implementovať jednu vetvy, a s malými úpravami z nej vieme vytvoriť aj ostatné.



Obr. 11.3: Automat tretej úrovni.

Zakreslenie koncového automatu a zároveň aj stavu je veľmi jednoduché, implementujeme operácie popísané vyššie a dostávame výsledok.

Konkrétny príklad môže vyzerať nasledovne.
Majme pravidlá ako zmienené na začiatku príkladu, a x=y=z=3. Potom bude postupnosť prechodov vyzerať nasledovne.

- L1: S ⇒ A:B:C ⇒ 3A03B13C2 (Zaznamenáme rozdelenie na 3 menšie reťazce)
- L2A: 3A0 ⇒ 2aA0 ⇒ 1aaA0 ⇒ 1aaa0
- L2B: 3B1 ⇒ 2bB1 ⇒ 1bbB1 ⇒ 1bbb1
- L2C: 3C2 ⇒ 2cC2 ⇒ 1ccC2 ⇒ 1ccc2
- L3: aaabbbccc

Ak by sme chceli zvýšiť silu, môžeme tak dosiahnuť dvoma spôsobmi, buď pridáme ďalší automat na úroveň L2, a adekvátne pravidlá pre prevedenie požadovaného, a to napríklad pravidlá 7. D → dD 8, 8. D → d.
Týmto by sme príjmali jazyk L(G) = $a^x b^y c^z d^w$: x/y/z/w≧1, alebo môžeme jednoducho zvýšiť počítadlo.

Kapitola 12

Alternatívy využitia viacúrovňového prístupu

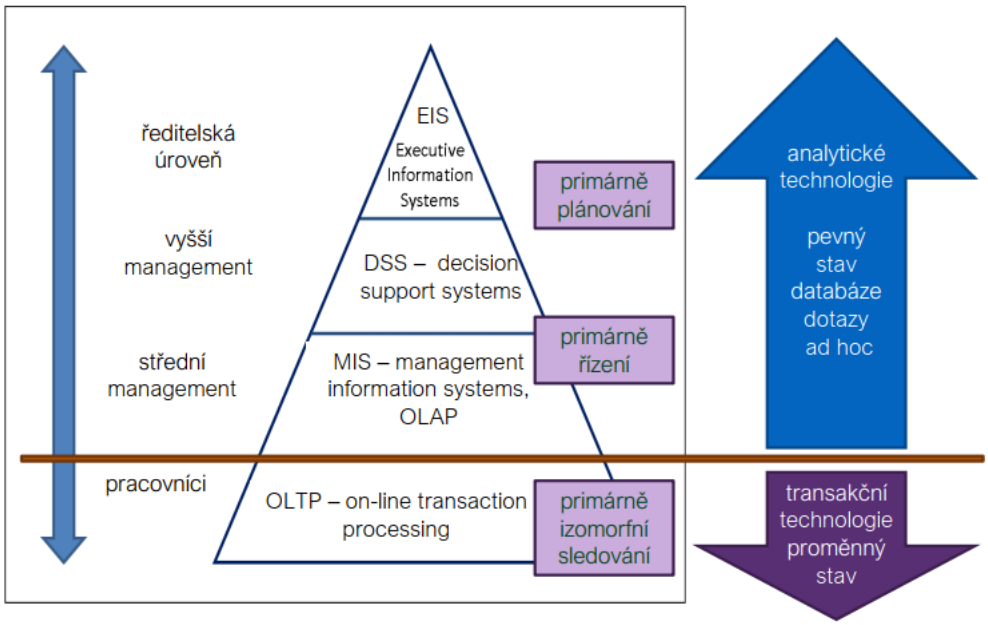
V tejto sekcií si ukážeme, že sa náš viacúrovňový prístup dá posunúť aj do iných oblastí informatiky. Začneme celulárnymi automatmi, kde si stručne ukážeme možnosť využitia celulárnych automatov. Následne sa presunieme do sféry dnes čoraz viac populárnych informačných systémov.

12.1 Viacúrovňové celulárne automaty

Celulárny automat(CA) je označenie pre určitý typ fyzikálneho modelu, či už v podobe reálneho prístroja alebo počítačového algoritmu. Slúži k časovej a priestorovej diskrétnej idealizácií fyzikálnych systémov, kde hodnoty veličin nadobúdajú iba diskrétnych hodnôt. Celulárny automat je dynamický systém, diskrétny v hodnotách, priestoru a času. Tvorí ho pravidelná štruktúra buniek v N-rozmernom priestore. Každá bunka môže nadobúdať jeden z K možných stavov, často ide iba o 2 stavy a to 0 - mŕtva bunka, 1 - živá bunka. základný princíp je, že hodnoty stavu v ďalšom časovom kroku, sa počítajú na základe hodnoty aktuálnej, a hodnôt okolia bunky. Okolie si môžeme stanoviť ľubovoľné, avšak existujú už pomenované okolia. Teda hodnota celej mriežky buniek, sa mení skupinovo v jednom časovom okamihu, na základe predchádzajúcich hodnôt. Avšak, tento fakt nám prináša aj problémy, CA nemá pamäť, CA reprezentuje iba jednu možnosť nakoľko má napevno nastavené pravidlá, podľa ktorých sa chová. Viacúrovňový koncept by nám mohol priniesť presne tieto veci. Nakoľko by sme mohli bunky rozdeliť na viac úrovní hlbšie, vedeli by sme si zapamätať ich stav, tým pádom by nám zaviedol pamäť. Takto by sme sa vedeli vrátiť ku situácií, kde nám nastala napríklad chyba pri testovaní CA. Taktiež by sme vedeli rozdeliť bunky do viacerých úrovní a držať si tým napríklad iné pravidlá, ktoré by sme mohli použiť napríklad po 1000 časových jednotkách a tak simulovať inú situáciu.

12.2 Viacúrovňový informačný systém

Informačné systémy sú v dnešnej dobe veľmi rozšírený nástroj ako zvýšiť produktivitu vo firmách. Je zrejmé, že sa v organizačná štruktúra v každej firme delí na rôzne úrovne. Taktiež je zrejmé, že na každej úrovni sú za potreby iné informácie. Na nasledujúcom obrázku si ukážeme pyramidovú schému informačného systému.



Obr. 12.1: Obrázok prevzaný z [1]

Môžeme vidieť, že manažment a pracovníci sa nachádzajú na iných úrovniach systému. To je dôvod prečo potrebujú zobraziť iné informácie. Presne preto sa dá využiť viacúrovňový koncept, ktorý by nám pomohol spracovať rovnaké dáta, do rôznych výstupov. Nakoľko je za potreby možnosť z každej vyššej úrovne schopnosť dostať sa na informácie nižšej úrovne, by nám viacúrovňový koncept pomohol aj v tomto, mohli by sme si zaznamenať cestu akými automatmi sme transformovali dáta, a potom spätnou cestou by sme vedeli vykonať takzvanú operáciu "Drill down".

Ukázali sme si, že viacúrovňový koncept je možné aplikovať aj v iných odvetviach informatiky a nie len v teoretickej informatike. Dané koncepty boli len v skratke spomenuté, ako možnosti aplikácie, pre dokázanie, že tento prístup je efektívny a oplatí sa mu venovať viac pozornosti.

Kapitola 13

Zhrnutie a záver

Dokázali sme, že zastarané praktiky sú nepostačujúce v dnešnej dobe. Dnešná informatika vyžaduje nové praktiky, ako sme dokázali, dajú sa vytvoriť nové koncepty aj v odvetvý teoretickej informatiky a poprípade ich spojiť s praktickým programovaním. Ukázali sme, že regulárne výrazy nám popisujú regulárne jazyky. Avšak, sme si ukázali ešte jeden základný model, ktorý popisuje regulárne jazyky, a tým sú konečné automaty. Definovali sme si konečné automaty a ukázali sme si ich grafickú reprezentáciu. Následne sme sa posunuli do nového konceptu, viacúrovňových konečných automatov. Ukázali sme si, ako sa jednoduchou úvahou a pár zmenami dá zaviesť paralelizmus do konečných automatov, a tým zvýšiť ich celkovú silu, podporiť prácu viacerých ľudí na jednom projekte, nakoľko sa implementácia jednoduchšie rozdeľuje. Taktiež nám viac úrovňovosť umožňuje problémy lepšie škálovať, a rozširovať, a tým uľahčuje prácu v udržiavacej fázy projektov. Tieto automaty sme si definovali, a ukázali sme si rozdiely v grafickej reprezentácií klasických konečných automatov, a viacúrovňových konečných automatov. Následne sme si ujasnili proces kompílacie, a fázy kompilácie. Zistili sme, že vo fázy lexikálnej analýzy sa využívajú konečné automaty. To je dôvod, prečo sme si zvolili lexikálnu analýzu ako príklad implementácie viacúrovňových konečných automatov. Aby sme vedeli čo vlastne implementujeme, ukázali sme si čo je vlastne kompilátor. Následne sme si opísali čo je lexikálna analýza podrobnejšie, a prečo je pre nás v tejto práci poštacujúca iba daná analýza. Na to aby sme si mohli naimplementovať lexikálnu analýzu, sme potrebovali zadanie jazyku, ktorý automat reprezentujúci jeho lexikálnu analýzu prijímal. Daný jazyk sme prevzali z [2]. Postupne sme podľa zadania zostrojili viacúrovňový konečný automat, ktorý prijíma daný jazyk. Taktiež sme zostrojili klasický konečný automat, ktorý prijíma daný jazyk, aby sme mali s čím porovnať grafickú reprezentáciu. Uviedli sme si, že oba automaty musia byť deterministické, aby boli implementovateľné, a tak sme sa ku nim aj správali. Postupne sme si rozpísali jednotlivé úrovne automatu, a odôvodnili sme si prečo sú tak zakreslené. Následne sme v krátkosti porovnali implementáciu daných automatov. Z toho sme zistili, že nový koncept je viac efektívny, jednoduchšie škálovatelný, taktiež jednoducho modulovatelný. Na rozdiel od starého prístupu, kde bol celý automat reprezentovaný jednou programovou konštrukciou. Aj keď, sa na prvý pohľad vzdá nový koncept ako zložitejší, nakoľko rozdelujeme jeden automat na viac menších automatov, je to iba otázka zvyku. Potom, ako sme dokázali užitočnosť viacúrovňového prístupu pri lexikálnej analýze, posunuli sme sa hlbšie v teoretickej informatike. Definovali sme si rôzne paralelné pravo-lineárne gramatiky a sebaregulujúce konečné automaty. Pre dôkaz myšlienky sme navrhli koncept, ktorým sme znovu potvrdili dôležitosť inovácie v teoretickej informatike. Na príklade sme navrhli grafickú reprezentáciu spolu s návrhom implementácie príkladu na spracovanie paralelne pravo-lineárneho jazyka.

Na danom príklade sme si ukázali okrem základných výhod viacúrovňových automatov, aj to, že je nimi možné reprezentovať paralelizmus, ktorý sa v informatike využíva čoraz viac. Na koniec sme sa posunuli do iných odvetví informatiky. Najskôr sme si v krátkosti navrhli využitie viacúrovňového prístupu pre celulárne automaty. Následne sme si dokázali, že aj v čoraz viac rozrastajúcom sa odvetví informačných systémov, vieme nájsť miesto pre viacúrovňový koncept. Na záver je potrebné dodať, že daný koncept sa oplatí skúmať ďalej, a boli by sme radi, ak by sa mu začal venovať aj niekto iný. To je dôvod prečo uvádzame ďalšie možnosti využitia tohoto konceptu. Ako sme už spomenuli bolo by možné príklad paralelne pravo-lineárnych jazykov aj naimplementovať a otestovať. Ďalej je možné daný koncept abstrahovať a využiť ho na celý proces kompilácie zdrojového kódu na cieľový kód. Taktiež ako už bolo zmienené, je tu možnosť využiť tento koncept pri celulárnych automatoch. Vyššie zmienené využitie v informačných systémoch, by bolo tiež za potreby spracovať do hĺbky. Nakoniec je možné viacúrovňový rámec využiť aj v počítačovej grafike, a to napríklad na vykreslovanie 3D scén, alebo skladanie obrázkov po úrovniach.

Literatúra

[1] Burget, R.: *Pojem informačního systému, data, informace.* [Prednáška: Posledná zmena: 2017-09-19].

[2] Křivka, Z.; Kocman, R.; Milkovic, M.: *ZADÁNÍ PROJEKTU Z PREDMĚTŮ IFJ A IAL.* 2017, [Zadanie: Stiahnuté: 2018-10-19].

[3] Meduna, A.: *Elements of Compiler Design.* Auerbach Publications, 2007, ISBN 978-1-4200-6323-3.

[4] Meduna, A.; Lukáš, R.: *Formální jazyky a překladače.* [Prednášky: Posledná zmena: 2017-09-19].

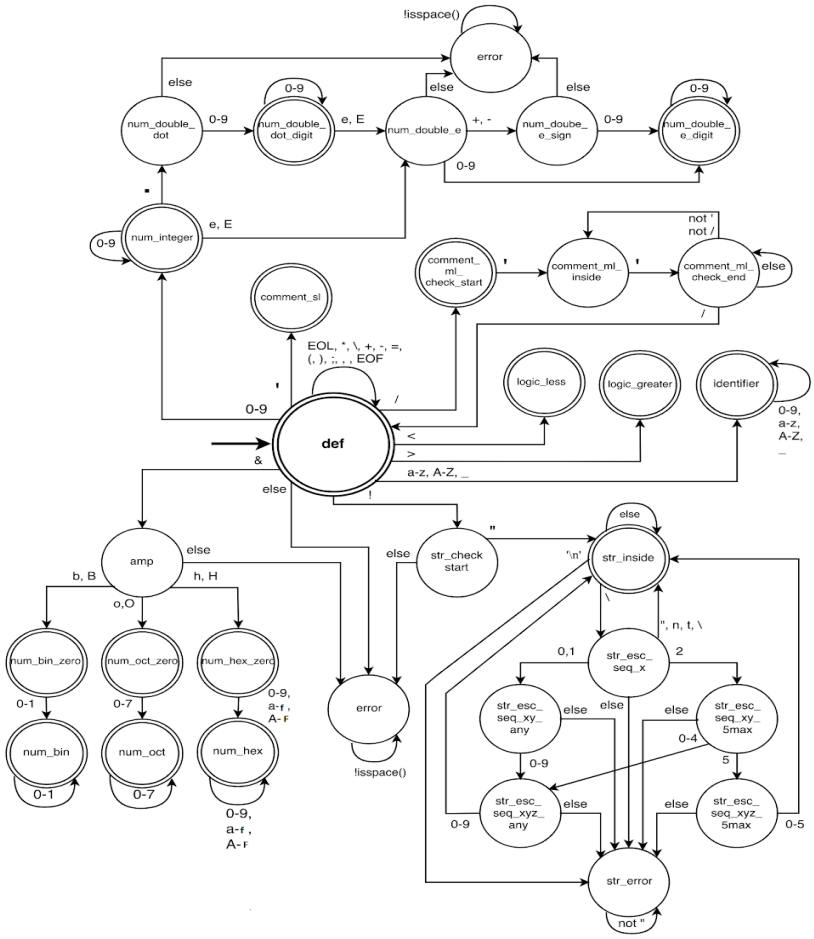
[5] Meduna, A.; Masopust, T.: *Self-Regulating Finite Automata.* [Prednáška: Stiahnutá: 2018-09-20].

[6] Wood, D.: *Properties of n-parallel finite state languages.* In *Utilitas Mathematica*, ročník 4, unknown, 1973, s. 103–113.

[7] Wood, D.: *n-Linear simple matrix languages and n-parallel linear languages.* In *Rev. Roumaine Math. Pures Appl.*, Romanian Academy, 1977, s. 403–412.

Príloha A

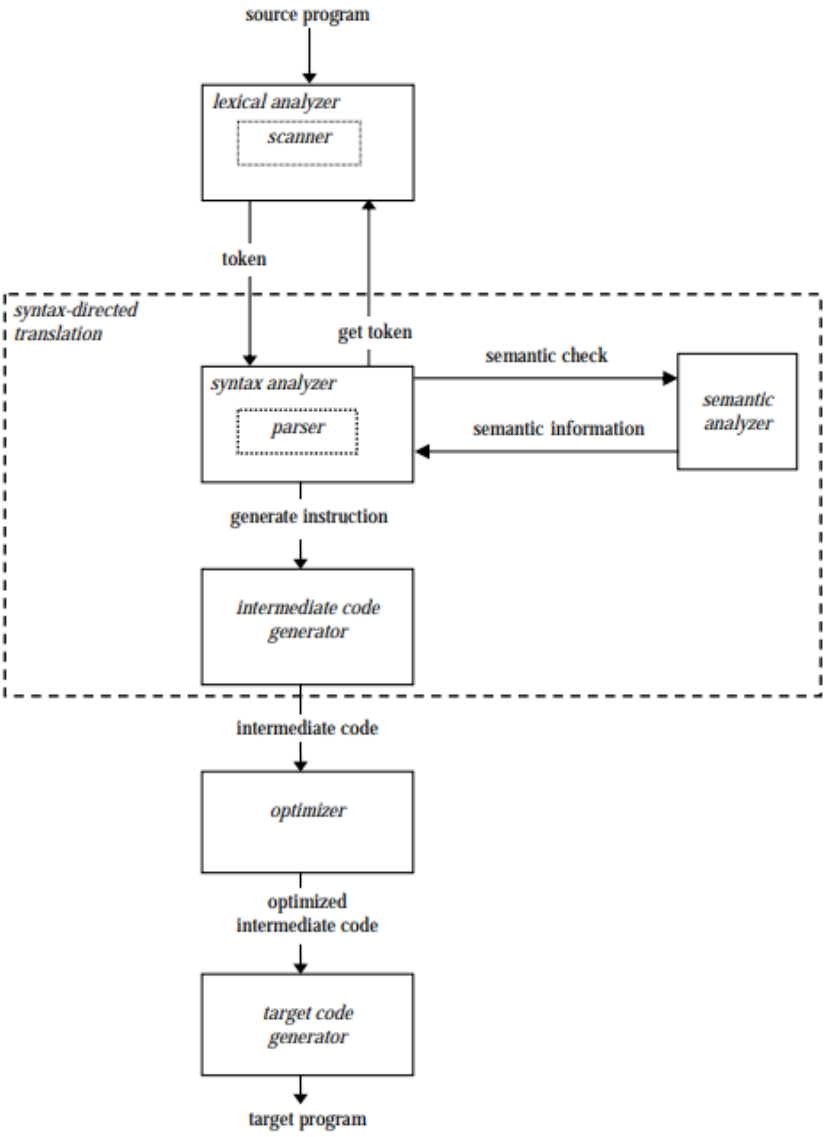
Schéma konečného automatu



Obr. A.1: Tento automat príjma jazyk IFJ17.

Príloha B

Konštrukcia kompilátora



Obr. B.1: Tento obrázok znázorňuje konštrukciu kompilátora (prevzaté z [3]).