VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

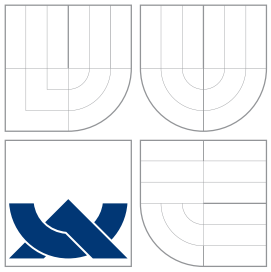
HLUBOKÝ SYNTAXÍ ŘÍZENÝ PŘEKLAD

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

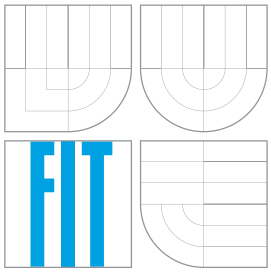
AUTOR PRÁCE
AUTHOR

Bc. JOZEF SENKO

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

HLUBOKÝ SYNTAXÍ ŘÍZENÝ PŘEKLAD

DEEP SYNTAX-DIRECTED TRANSLATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. JOZEF SENKO
AUTHOR

VEDOUCÍ PRÁCE — Prof. RNDr. ALEXANDER MEDUNA, CSc.
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce volně navazuje na moji bakalářskou práci, která byla věnována syntaktické analýze s použítím hlubokých zásobníkových. V teoretické části je této práce je definováno vše potřebné a základní pro tuto problematiku, jako například hlubokému syntaxí řízenému překladu, zásobníkovým automatům, hlubokým zásobníkovým automatům, konečným převodníkům a hlubokým zásobníkovým převodníkům.
Ve druhé části se věnuji programu, který je součástí této práce, kdy se jedná o program, který je určený pro předmět IFJ. V této části je rozebrán návrh programu, struktura a jednotlivé části programu jak z teoretické, tak i z praktické stránky.

Abstract

This thesis is a continuation of my bachelor thesis, which is dedicated to syntax analysis based on deep pushdown automata. In theorical part of this thesis is defined everything fundamental for this work, for example deep syntax-directed translation, pushdown automata, deep pushdown automata, finite transducer and deep pushdown transducer.
The second part of this thesis is dedicated to the educational program for students of IFJ. In this part is defined strucure of this program and its parts. All part of program are analyzed from a theoretical and practical point of view.

Klíčová slova

překlad, syntaktická analýza, zásobníkový automat, hluboký zásobníkový automat, bezkontextová gramatika, stavová gramatika, konečný převodník, hluboký zásobníkový převodník.

Keywords

translation, syntax analysis, pushdown automata, deep pushdown automata, context-free grammar, finite transducer, deep pushdown transducer

Citace

Jozef Senko: Hluboký syntaxí řízený překlad, diplomová práce, Brno, FIT VUT v Brně, 2015

Hluboký syntaxí řízený překlad

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Prof. RNDr. Alexandera Meduny CSc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Jozef Senko
27. května 2015

Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce, panu Prof. RNDr. Alexanderovi Medunovi CSc. za pomoc pří řešení této práce a vůbec tomu, že mne pro tento obor nadchnul již v předmětu IFJ.

Obsah

Kapitola 1

Úvod

Tato práce je věnována hlubokému syntaxí řízenýmu překladu, hlubokým zásobníkovým automatům a hlubokým zásobníkovým převodníkům. V této diplomové práci přímo navazuji na svoji bakalářskou práci [11] a práci vedoucího mé bakalářské práce Ing. Petera Solára [10].

Ve druhé kapitole [Kap 2] zavádím základní pojmy, které jsou potřebné pro další definice, nezbytné pro další pochopení textu a základní uvedení do problematiky.

Následující kapitola [Kap 3] pojednává o jazycích. Jazyky jsou zde definovány a popsány. Jde zde také zmíněna Chomského hierarchie s jejím popisem a rozdělením jednotlivých gramatik, které budou následně definovány a řádně zavedeny v následující kapitole.

Jak bylo zmíněno výše, další kapitola [Kap 4] je věnována gramatikám, jejich popisu, definici a také jejich rozdělení.

Další, tedy 5. kapitola [Kap 5] je věnována již automatům. A to dvěma typům, konečným a zásobníkovým. Každý automat je zde popsán, definován jak automat tak i jeho konfigurace a graficky zobrazen. Pro každý automat je zde také uveden příklad toho, jak funguje a pracuje.

Následující kapitola [Kap 6], která přimo navazuje na automaty se zaobírá hlubokým zásobníkovým automatem, který je rozšířením klasického zásobníkového automatu. Hluboký zásobníkový automat je zde opět definován, stejně jako jeho konfigurace. Je zde zmíněno téma determinismu hlubokých zásobníkových automatů a samozřejmě je i zde uveden příklad fungování tohoto hlubokého zásobníkového automatu.

Sedmá kapitola [Kap 7] je věnována překladu, rozdělení na jeho jednotlivé části, popisu vstupů a výstupů a samozřejmě i popisu fungování těchto části. Kapitola je uzavřena definicí formálního překladu a syntaxí řízeného překladového schématu.

V osmé kapitole [Kap 8] je neformálně popsán konečný převodník, následně formálně definován jak konečný převodník, tak i jeho konfigurace a přechod konečného převodníku.

Na šestou [Kap 6] a osmou [Kap 8] kapitolu navazuje kapitola devátá [Kap 9], která jej již věnována hlubokému zásobníkovýmu převodníku. Je opět neformálně popsán, následně formálně definován a stejně jako on je formálně definována jeho konfigurace.

Desátou kapitolou [Kap 10] začíná druhá část této práce, kdy tato kapitola je věnována již samotnému programu. Je zde popsáno využití a použití programu tohoto výukového

programu.

V kapitole s číslem 11 [Kap 11] je definován jazyk, který bude využit ve výukovém programu. Následující kapitola [Kap 12] obsahuje seznam všech pravidel dané gramatiky a také precedenční tabulku dané gramatiky pro vyhodnocování výrazů.

Další kapitola [Kap 13] je věnována návrhu a implementaci všech tří částí překladače - lexikální, syntaktické i precedenční analýzy. Jsou zde popsány všechny datové struktury, třídy a také podstatné a důležité metody.

Čtrnáctá kapitola [Kap 14] je již věnována uživatelskému rozhraní. Je rozdělena do tří základních částí - se zaměřením na lexikální analýzu, se zaměřením na syntaktickou analýzu a také se zaměřením na precedenční analýzu.

V poslední, tedy patnácté kapitole [Kap 15], jsou shrnuty získané závěry a uzavřena celá tato práce.

Kapitola 2

Základní pojmy

Definice pojmů 2.1 a 2.9 jsou převzaty z následujících zdrojů [9] a [7]. U dalších definicí jsem vycházel z [9], [7] a [4].

2.1 Abecedy a symboly

Abeceda je konečná, neprázdná množina elementů, kde tyto elementy jsou nazývány *symboly* dané abecedy.

2.2 Řetězec

Nechť Σ je abeceda.

1. ε je řetězec nad abecedou Σ.
2. Pokud x je řetězec nad Σ a $a \in \Sigma$, potom xa je řetězec nad abecedou Σ.

2.3 Délka řetězce

Nechť x je řetězec nad abecedou Σ. *Délka řetězce* x, $|x|$, je definována jako:

1. Pokud $x = \varepsilon$, pak $|x| = 0$.
2. Pokud $x = a_1 \dots a_n$, pak $|x| = n$
   pro $n \geq 1$ a $a_i \in \Sigma$ pro všechna $i = 1, \dots, n$.

2.4 Konkatenace řetězců

Nechť x a y jsou dva řetězce nad abecedou Σ. *Konkatenace* x a y je řetězec xy.

2.5 Mocnina řetězce

Nechť x je řetězec nad abecedou Σ. Pro $i \geq 0$, kde i-tá mocnina řetězce x, x^i, je definována jako:

1. $x^0 = \varepsilon$
2. Pro $i \geq 1$: $x^i = xx^{i-1}$, kde tedy využíváme operace konkatenace, která byla definována výše v této kapitole.

2.6 Prefix řetězce

Nechť x a y jsou dva řetězce nad abecedou Σ. x je prefixem y, pokud platí, že existuje takový řetězec z nad abecedou Σ, pro který platí $xz = y$. Zde je opět využito operace konkatenace, která byla definována výše v této kapitole.

2.7 Sufix řetězce

Nechť x a y jsou dva řetězce nad abecedou Σ. x je sufixem y, pokud platí, že existuje takový řetězec z nad abecedou Σ, pro který platí $zx = y$. Zde je opýt využito operace konkatenace, která byla definována výše v této kapitole.

2.8 Podřetězec

Nechť x a y jsou dva řetězce nad abecedou Σ. x je *podřetězec* y, pokud existují řetězce *z,z'* nad abecedou Σ, přičemž platí *zxz'=y*.

2.9 Card(Q)

$card(Q)$ je operace nazývána kardinalita množiny. Kardinalita množiny udává počet prvků dané množiny.

2.10 alph(w)

$alph(w)$ udává množinu všech symbolů, které se vyskytují v daném vstupním řetězci w.

2.11 occur(x,y)

$occur(x, y)$ je metoda, která udává počet výskytů všech symbolů, které se nacházejí ve vstupní množině y pro daný vstupní řetězec x.

2.12 Terminální symboly

Jedná se o elementární symboly jazyka (viz kapitola 3), které již nelze nahradit za jiné symboly, nebo řetězce symbolů na základě jakéhokoliv dostupného pravidla.

2.13 Neterminální symboly

Neterminální symboly jsou takové symboly, které lze dále nahrazovat. Ať už jednotlivými dalšími symboly, či řetězci. Tyto řetězce se mohou skládat z terminálních symbolů daného jazkya a zároveň také i z neterminálních symbolů daného jazyka.

Kapitola 3

Jazyky

Jazyk je v podstatě podmnožinu množiny všech možných řetězců nad danou abecedou.

Každý jazyk může být popsán dvěma ruznými způsoby - lze jej popsat buďto automaty nebo gramatikami. I přesto, že oba popisují stejný jazyk, slouží k dvěma různým účelům. Zatímco automaty slouží pro zjištění, zda daný řetězec na základě daných pravidel patří do určitého jazyka, který je tímto automatem popsán, tak na druhou stranu gramatiky jsou schopny na základě daných pravidel generovat řetězce toho určitého jazyka.

3.1 Definice

Nechť Σ^* značí množinu všech řetězců nad Σ, včetně prázdného řetězce. Každá podmnožina $L \subseteq \Sigma$ je *jazyk* nad Σ.

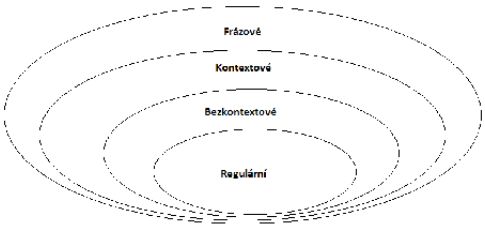
3.2 Chomského hierarchie

Jedná se o hierarchii tříd formálních gramatik, zavedenou Noamem Chomskym roku 1956. Tyto formální gramatiky jsou hierarchicky rozděleny do následujících 4 skupin, kdy každá gramatika generuje formální jazyk.

- Frázové gramatiky - gramatika typu 0
- Kontextové gramatiky - gramatika typu 1
- Bezkontextové gramatiky - gramatika typu 2
- Regulární gramatiky - gramatika typu 3

Přičemž zároveň platí, že všechny gramatiky, které se nacházejí hierarchicky níže jsou podmnožinami dané gramatiky, jak lze vidět na obrázku 3.1.

Hierarchičnost gramatik spočívá v tom, že každá regulární gramatika je bezkontextová, každá bezkontextová gramatika je poté kontextová a každá kontextová gramatika je gramatikou typu 0, také označovanou jako frázová gramatika.



Obrázek 3.1: Hierarchie gramatik

3.3 Regulární jazyky

Každý takovýto jazyk lze vždy popsat regulárním výrazem a je vytvořen ze základních symbolů abecedy. Při vytváření regulárního jazyka se využívá pouze základních operací sjednocení, zřetězení a iterace a takový to jazyk je rozpoznatelný konečným automatem.

3.3.1 Definice

Nechť L je jazyk. L je *regulární jazyk* právě tehdy, pokud existuje regulární výraz, který tento jazyk značí.

3.4 Bezkontextové jazyky

Jedná se o formální jazyky akceptovatelné zásobníkovým automatem a mohou být generovány bezkontextovými gramatikami. Pro každý regulární jazyk platí, že je zároveň i bezkontextový, avšak každý bezkontextový jazyk není jazykem regulárním.

3.4.1 Definice

Nechť L je jazyk. L je *bezkontextový jazyk* právě tehdy, pokud existuje bezkontextová gramatika, která generuje tento jazyk L.

Kapitola 4

Gramatiky

Formální gramatiky označují strukturu, která popisuje formální jazyk.

Jedná se o model, který je schopný popsat různé jazyky. Gramatika je tvořena množinou gramatických pravidel, na základě kterých dochází k vytvoření určitého řetězce daného jazyka z počátečního symbolu. Samotné generování následně probíhá tak, že aplikujeme postupně ruzná pravidla dané gramatiky na počáteční symbol a dostáváme tak ruzné řetězce daného jazyka.

4.1 Jednoznačnost a nejednoznačnost(víceznačnost) gramatiky

Pokud je pro každé slovo gramatiky možný pouze jeden způsob vygenerování, pak takovou gramatiku nazýváme *jednoznačnou*. V opačném případě tedy dostáváme gramatiky *nejednoznačné*, tedy takové gramatiky, jejichž řetězce lze vygenerovat dvěma čí více různými způsoby.

4.2 Regulární gramatiky

Regulární gramatiky, tedy gramatiky typu 3 jsou v Chomského hierarchii na nejnižší úrovni a z čehož tedy plyne, že každá regulární gramatika je i bezkontextová, kontextová a frázová. S regulárními gramatikami jsou úzce spjaty konečné automaty, které budou definovány v následující kapitole [Kap 5]. Dále umožňují generovat regulární jazyky.

4.2.1 Definice

Regulární gramatika G je čtveřice $G = (N, T, P, S)$, kde

- N je neprázdná konečná *abeceda neterminálů*, tedy neterminálních symbolů
- T je konečná *abeceda terminálů*, přičemž platí $N \cap T = \varnothing$

- P je konečná *množina pravidel* tvaru: $A \rightarrow xB$, nebo $A \rightarrow x$, kde $A, B \in N, x \in T^*$, popřípadně speciální pravidlo $S \rightarrow \epsilon$, pokud se S nevyskytuje na pravé straně žádného pravidla.
- $S \in N$ je *počáteční neterminál*, neboli kořen gramatiky

4.3 Bezkontextové gramatiky

Bezkontextové gramatiky, tedy gramatiky typu 2 - pomocí těchto gramatik bývají definovány syntaxe programovacích jazyků a jsou spjaty se zásobníkovými automaty a jejich rozšířeními, s tím, že tyto automaty budou definovány v následující kapitole [Kap 5]. Dále umožňují generovat bezkontextové jazyky.

4.3.1 Definice

Bezkontextová gramatika G je čtveřice $G = (N, T, P, S)$, kde

- N je neprázdná konečná *abeceda neterminálů*, tedy neterminálních symbolů
- T je konečná *abeceda terminálů*, přičemž platí $N \cap T = \varnothing$
- P je konečná *množina pravidel* tvaru: $A \rightarrow x$, kde $A \in N, x \in (N \cup T)^*$
- $S \in N$ je *počáteční neterminál*, neboli kořen gramatiky

4.4 Kontextové gramatiky

Kontextové gramatiky, tedy gramatiky typu 1 jsou přijmány lineárně omezenými automaty. Lineárně omezené automaty jsou v podstatě nedeterministické turingovy stroje [viz [3]], které nikdy neopustí tu část pásky, na které je zapsaný jeho vstup.

4.4.1 Definice

Kontextová gramatika G je čtveřice $G = (N, T, P, S)$, kde

- N je neprázdná konečná *abeceda neterminálů*, tedy neterminálních symbolů
- T je konečná *abeceda terminálů*, přičemž platí $N \cap T = \varnothing$
- P je konečná *množina pravidel* tvaru: $\alpha A \beta \rightarrow \alpha \gamma \beta$, kde $A \in N, \alpha, \beta \in (N \cup T)^*, \gamma \in (N \cup T)^+$, popřípadně speciální pravidlo $S \rightarrow \epsilon$, pokud se S nevyskytuje na pravé straně žádného pravidla.
- $S \in N$ je *počáteční neterminál*, neboli kořen gramatiky

4.5 Stavové gramatiky

Stavové gramatiky jsou silnější, než vyše zmíněné, bezkontextové. Oproti bezkontextovým gramatikám obsahují navíc stavy, kdy je možnost, že ne vždy lze najít pravidlo přepisující nejlevější neterminální symbol, avšak může dojít k přepsání neterminálního symbolu nacházejícího se hlouběji ve větné formě. Při definích jsem vycházel ze dvou článků a to z [8] a [2], s tím, že tato definici je převzatá z mé bakalářské práce [11], kde jsem tyto stavové gramatiky definoval.

4.5.1 Definice

Stavová gramatika G je pěticě $G = (V, W, T, P, S)$, kde

- V je *úplná abeceda*
- W je konečná *množina stavů*
- $T \subseteq V$ je *abeceda terminálů*, přičemž platí $N \cap T = \varnothing$
- $S \in (V\ -T)$ je *počáteční symbol*
- $P \subseteq (W \times (V\ -T)) \times (W \times V^+)$ je *konečná relace*. Místo zapisování pravidel ve tvaru $(q, A, p, v) \in P$ je zapisujeme $(q, A) \rightarrow (p, v) \in P$.

Pro každý řetězec $z \in V^*$ ustanovme ${}_G states(z) = q|(q, B) \rightarrow (p, v) \in P$, kde $B \in (V - T) \cap alph(z), v\ \in V^+, q, p \in W$. Za předpokladu, že existuje pravidlo $(q, A) \rightarrow (p, v) \in P$ a máme řetězce $x, y \in V^*$, množinu ${}_G statex(x) \cap q =$, pak gramatika G udělá derivační krok z (q, xAy) do (p, xvy), symbolicky zapsáno jako $(q, xAy) \Rightarrow (p, xvy)[(q, A) \rightarrow (p, v)]$.

V případě, že přidáme kladné, celé číslo n splňující $occur(xA, V - T) \leq n$, říkáme, že $(q, xAy) \Rightarrow (p, xvy)[(q, A) \rightarrow (p, v)]$ je n-omezené, symbolicky zapsáno $(q, xAy)_n \Rightarrow (p, xvy)$
$[(q, A) \rightarrow (p, v)]$.

V případě, že nehrozí záměny, zjednodušeně zapíšeme $(q, xAy) \Rightarrow (p, xvy)[(q, A) \Rightarrow (p, v)]$ a $(q, xAy)_n \Rightarrow (p, xvy)[(q, A) \rightarrow (p, v)]$ na $(q, xAy) \Rightarrow (p, xvy)$ a $(q, xAy)_n \Rightarrow (p, xvy)$.

Také lze rozšířit derivační krok $\Rightarrow$ na $\Rightarrow^m$, kde $m \geq 0$. Po té na základě $\Rightarrow^m$ můžeme definovat $\Rightarrow^+$, značící provedení alespoň jednoho derivačního kroku a $\Rightarrow^*$, značící možnost neprovedení ani jednoho derivačního kroku.

Nechť $n \in I$ a $v, \varpi \in (W \times V^+)$. K vyjádření, že každý derivační krok $\Rightarrow^m$, $\Rightarrow^+$ a $\Rightarrow^m$ je n-omezený, zapisujeme ${}_n \Rightarrow^m$, ${}_n \Rightarrow^+$, ${}_n \Rightarrow^*$. Pomocí $string(v_n \Rightarrow^* \varpi)$ značíme množinu všech řetězců vyskytujících se v derivaci $v_n \Rightarrow^* \varpi$.

Jazyk $L(G)$ je definován jako $L(G) = w \in T^*|(q, S) \Rightarrow^* (p, w), q, p \in W$. Mimoto definujeme pro každé $n \geq 1, L(G, n) = w \in T^*|(q, S)_n \Rightarrow^* (p, w), q, p \in W$. Derivace tvaru $(q, S)_n \Rightarrow^* (p, w)$, kde $q, p \in W$ a $w \in T^*$, reprezentuje úspěšné n-omezené generování řetězce w v gramatice G.

Kapitola 5

Automaty

Tato kapitola je věnována dvěma základním automatům - konečnému automatu a automatu zásobníkovému. Oba tyto automaty slouží k rozhodování, zda daný vstupní řetězec je, či není řetězcem daného jazyka.

U definice konečného automatu jsem vycházel z [5] a při definici zásobnikového automatu jsem využil zdroje [6].

Jak je zmíněno v kapitole [Kap 4], konečné automaty jsou spojeny s regulárními jazyky, zatímco zásobníkové automaty jsou spojeny s bezkontextovými jazyky.

Oba typy automatů jsou základem pro další podobné typy automatů, které zlepšují některé vlastnosti, či rozšiřují možnost.

5.1 Konečné automaty

Prvním automatem, který bych rád zmínil je automat konečný. Jedná se o teoretický výpočetní model používáný nejen pro studium formálních jazyků. Popisuje jednoduchý počítač, který se může nacházet v jednom stavu z jeho množiny stavů. Přechody mezi těmito stavy jsou prováděny na základě symbolů, načtených se vstupní pásky. Rozhodování probíhá na základě aktuálního stavu a aktuálně načteného vstupního symbolu.

Jak jsem již několikrát zmínil, tyto automaty jsou schopny rozeznávat pouze regulární jazyky.

5.1.1 Definice

Konečný automat je pětice $M = (Q, \Sigma, R, s, F)$, kde

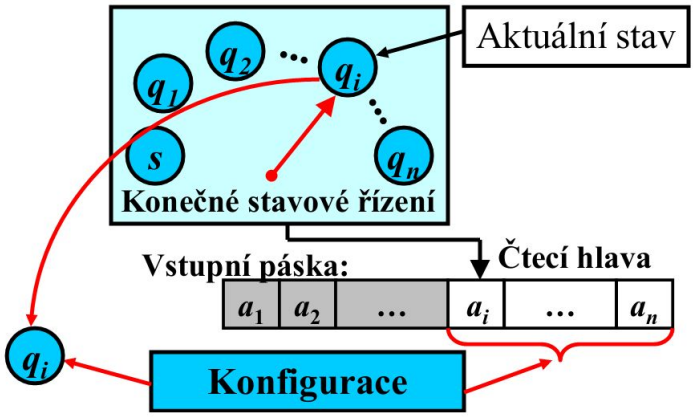
- Q je *konečná množina stavů*
- Σ je *vstupní abeceda*
- R je *konečná množina pravidel* tvaru: $pa \to q$, kde $p, q \in Q, a \in \Sigma \cup \{\varepsilon\}$

- $s \in Q$ je *počáteční stav*
- $F \subseteq Q$ je *množina koncových stavů*

5.1.2 Konfigurace

Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat.

Konfigurace konečného automatu M je řetězec $\chi \in Q\Sigma^*$.



Obrázek 5.1: Konfigurace konečného automatu [1]

5.1.3 Příklad

Uvažujme $M = (Q, \Sigma, R, s, F)$, kde:

- $Q = \{s, m, n, f\}$
- $\Sigma = \{a, b\}$
- $R = \{sa \to s, sb \to m, ma \to f, mb \to n, nb \to f\}$
- $F = \{f\}$

Při vstupním řetězci *abbb* bude automat M postupovat takto:

$$(s, abbb) \quad \Rightarrow \quad (s, bbb) \quad [sa \to s]$$

$$(s, bbb) \quad \Rightarrow \quad (m, bb) \quad [sb \to m]$$

$$(m, bb) \quad \Rightarrow \quad (n, b) \quad [mb \to n]$$

$$(n, b) \quad \Rightarrow \quad (f,) \quad [nb \to f]$$

5.1.4 Typy konečných automatů

Jak jsem již zmínil, klasický konečný automat má několik typů, kdy jich alespoň pár zmíním.

- Konečný automat bez ϵ přechodů
- Deterministický konečný automat
- Úplný konečný automat
- Dobře specifikovaný konečný automat
- Minimální konečný automat

5.2 Zásobníkové automaty

Druhým typem automatů, který v této kapitole zmíním a podrobně rozeberu bude automat zásobníkový. Tento automat je v podstatě rozšířením konečného automatu. Jedná se o teoretický výpočetní model používáný nejen pro studium formálních jazyků. Popisuje jednoduchý počítač, který již na rozdíl od konečného automatu využívá jednoduchou pamět - zásobník.

Na rozdíl od konečných automatů, které pracují pouze s aktuálním stavem a aktuálně načteným symbolem, zásobníkové automaty využívají navíc ještě symbolu umístěného na vrcholu tohoto zásobníku. Díky tomuto rozšíření o zásobník jsou tyto automaty již

[1] Meduna, A., Lukáš, R.:*Formální jazyky a překladače*, Kapitola III. Modely pro regulární jazyky, Brno, FIT VUT v Brně.

schopny rozpoznávat i bezkontextové jazyky.

S tímto zásobníkem mohou být prováděny dvě operace - vyjmutí a expanze. Operaci vyjmutí lze provést právě tehdy, když je na vrcholu zásobníku stejný symbol jako symbol aktuálně načtený ze vstupní pásky. Při tomto kroku dojde k odebrání symbolu na vrcholu zásobníku a zároveň dojde k posunu čtecí hlavy na vstupní pásce směrem doprava. Při operaci expanze dojde k nahrazení nevstupního symbolu na vrcholu tohoto zásobníku daným řetězcem symbolů na základě daného pravidla přechodu tohoto automatu.

5.2.1 Definice

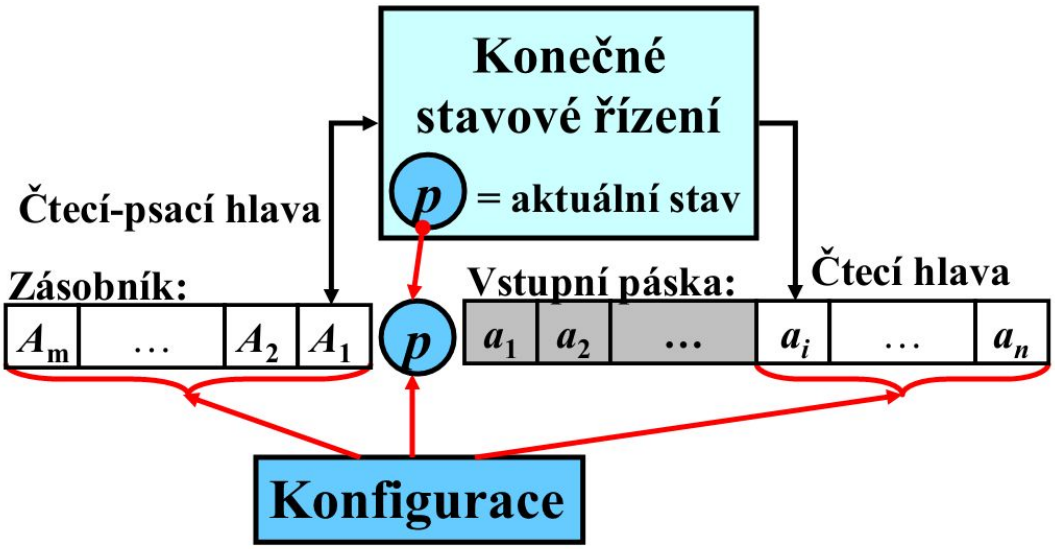
Zásobníkový automat je sedmice $M = (Q, \Sigma, \Gamma, R, s, S, F)$, kde

- Q je *konečná množina stavů*
- Σ je *vstupní abeceda*
- Γ je *zásobníková abeceda*
- R je *konečná množina pravidel* tvaru: $Apa \rightarrow wq$, kde $A \in \Gamma, p, q \in Q, a \in \Sigma \cup \{\varepsilon\}, w \in \Gamma^*$
- $s \in Q$ je *počáteční stav*
- $S \in \Gamma$ je *počáteční symbol na zásobníku*
- $F \subseteq Q$ je *množina koncových stavů*

5.2.2 Přijmaný jazyk

Při přijmání řetězce může zásobnikový automat přijmout daný řetězec třemi různými způsoby.

- Pokud zásobníkový automat M přijmá daný jazyk přechodem do koncového stavu, pak takto přijmaný jazyk je definovaný jako $L(M) = \{w : w \in \Sigma^*, Ssw \Rightarrow^* zf, z \in \Gamma^*, f \in F\}$
- Druhou možností je, že automat M přijmá daný jazyk vyprázdněním svého zásobníku a tudíž nezáleží, zda se nachází v koncovém stavu. Takto přijmaný jazyk je pak definován následovně $L(M) = \{w : w \in \Sigma^*, Ssw \Rightarrow^* zf, z = \varepsilon, f \in Q\}$
- Třetí a poslední možností je kombinace dvou již zmíněných přístupů. Tedy, že automat M přijmá daný jazyk přechodem do koncového stavu a vyprázdněním svého zásobníku. Takto přijmaný jazyk je pak definován následovně $L(M) = \{w : w \in \Sigma^*, Ssw \Rightarrow^* zf, z = \varepsilon, f \in F\}$



Obrázek 5.2: Konfigurace zásobníkového automatu [2]

5.2.3 Konfigurace

Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je zásobníkový automat.

Konfigurace takového zásobníkového automatu M je řetězec $\chi \in \Gamma^* Q \Sigma^*$.

[2] Meduna, A., Lukáš, R.: *Formální jazyky a překladače*, Kapitola VI. Modely pro bezkontextové jazyky, Brno, FIT VUT v Brně.

5.2.4 Příklad

Uvažujme $M = (Q, \Sigma, \Gamma, R, s, S, F)$, kde:

- $Q = \{s, m, n, f\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{a, S\}$
- $R = \{Ssa \rightarrow Sap, apa \rightarrow aap, apb \rightarrow q, aqb \rightarrow q, Sq \rightarrow f\}$
- $F = \{f\}$

Při vstupním řetězci *aabb* bude automat M postupovat takto:

$$
\begin{array}{llll}
(s, aabb, S) & \Rightarrow & (p, abb, Sa) & [Ssa \rightarrow Sap] \\
(p, abb, Sa) & \Rightarrow & (p, bb, Saa) & [apa \rightarrow aap] \\
(p, bb, Saa) & \Rightarrow & (q, b, Sa) & [apb \rightarrow q] \\
(q, b, Sa) & \Rightarrow & (q, , S) & [aqb \rightarrow q] \\
(q, , S) & \Rightarrow & (f, ,) & [Sq \rightarrow f]
\end{array}
$$

5.2.5 Typy zásobníkových automatů

Jak jsem již zmínil, klasický zásobníkový automat má několik typů, kdy jich opět alespoň pár zmíním.

- Deterministický zásobníkový automat
- Rozšířený zásobníkový automat
- Hluboký zásobníkový automat [Kap 6]
- Bezestavový hluboký zásobníkový automat
- Paralelní hluboký zásobníkový automat
- Paralelní bezestavový hluboký zásobníkový automat

Kapitola 6

Hluboký zásobníkový automat

Jak jsem již zmínil v předchozí kapitole, hluboký zásobníkový automat je mírně rozšířený zásobníkový automat a tedy využívá zásobník pro svou práci. Stejně tak i operace, které může automat provádět nad již zmíněným zásobníkem jsou v podstatě stejné, jako u klasického zásobnikového automatu.

Hlavním a tím podstatným rozdílem je fakt, že tento automat může pracovat i se symboly, které se nenechází přímo na vrcholu zásobníku, ale i s těmi, které se nacházejí hlouběji v zásobníku. Jak moc hluboko do zásobníku můžeme zasahovat je definováno typem daného hlubokého zásobníkového automatu. Díky tomuto odstranění omezení možnosti pracovat pouze se symbolem nacházejícím se na vrcholu zásobníku je dosaženo větší generativní síly a dokonce i možnosti tímto hlubokým zásobníkovým automatem generovat některé kontextové jazyky.

Následná definice tohoto hlubokého zásobníkového automatu pochází ze článku profesora Meduny *Deep pushdown automata* [8], který tento typ automatů zavedl.

6.1 Definice

Hluboký zásobníkový automat je sedmice ${}_nM = (Q, \Sigma, \Gamma, R, s, S, F)$, kde

- $n \in I$ je *maximální hloubka*, v niž může dojít k nahrazení,
- Q je *konečná množina stavů*
- Σ je *vstupní abeceda*
- Γ je *zásobníková abeceda*
- R je *konečná množina pravidel* tvaru: $mqA \to pv$, kde $A \in \Gamma, p, q \in Q, a \in \Sigma \cup \{\varepsilon\}, w \in \Gamma^*$
- $s \in Q$ je *počáteční stav*
- $S \in \Gamma$ je *počáteční symbol na zásobníku*

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

6.2 Konfigurace

Konfigurace hlubokého zásobníkového automatu je trojice údajů potřebných pro uložení aktualního stavu automatu. Těmito údaji jsou aktuální stav, nezpracovaná část vstupního řetězce a stav zásobníku.

6.2.1 Definice

Konfigurace hlubokého zásobníkového automatu ${}_nM$ je trojice $Q \times \Sigma^* \times (\Gamma - \{\#\})^*\{\#\}$.

6.3 Determinismus

Determinismus je vlastnost algoritmu, v tomto případě hlubokého zásobníkového automatu, kdy vždy za stejných výchozích, tedy vstupních, podmínek získáme stejný výstup. Z čehož tedy plyne, že tím je tento automat předvídatelný.

Až do teď jsme brali v potaz pouze nedeterministickou verzi hlubokého zásobnikového automatu, avšak případě hlubokého zásobníkového autmatu exitují 2 typy determinismu.

6.3.1 Striktní determinismus

V prvím, tedy tomto, případě platí, že hluboký zásobníkový automat může pro každý stav použít vždy pouze jedno pravidlo ve všech dostupných hloubkách.
${}_nM = (Q, \Sigma, \Gamma, R, s, S, F)$ je striktně deterministický právě tehdy, když platí $mqA \rightarrow pv \in R, card(\{mqA \rightarrow ow | mqA \rightarrow ow \in R, o \ \in Q, w \in \Gamma^+\} - \{mqA \rightarrow pv\}) = 0$.

6.3.2 Determinismus s ohledem na hloubku

Determinismus s ohledem na hloubku je již slabší formou determinismu narozdíl od striktního determinismu. I v tomto případě lze použít pro každý stav pouze jedno pravidlo, avšak s tím rozdílem, že u determinismu s ohledem na hloubku lze toto pravidlo použít pro každou možnou hloubku.

Pro každé $q \in Q, card(\{m | mqA \rightarrow pv \in R, A \ \in \Gamma, p \in Q, v \ \in \Gamma^+\}) \leq 1$

6.4 Příklad

Příklad byl převzat z mé bakalářské práce [11].
Uvažujme hluboký zásobníkový automat ${}_3M = (\{s, m, n, o, f\}, \{a, b, c, d\}, \{S, A, B, C, D, \#\}, R, s, S, \{f\})$, který má v R tato následující pravidla:

- $1sS \quad \rightarrow \quad mABCD$
- $3mC \quad \rightarrow \quad nc$

- $3nD \quad \rightarrow \quad odd$
- $1oA \quad \rightarrow \quad paa$
- $1pB \quad \rightarrow \quad fbb$

Při vstupním řetězci *aabbcdd* bude automat M postupovat takto:

$$
\begin{aligned}
(s, aabbcdd, S\#) &\ {}_{e}\Rightarrow\ (m, aabbcdd, ABCD\#) \quad [1sS \rightarrow mABCD] \\
(m, aabbcdd, ABCD\#) &\ {}_{e}\Rightarrow\ (n, aabbcdd, ABcD\#) \quad [3mC \rightarrow nc] \\
(n, aabbcdd, ABcD\#) &\ {}_{e}\Rightarrow\ (o, aabbcdd, ABcdd\#) \quad [3nD \rightarrow odd] \\
(o, aabbcdd, ABcdd\#) &\ {}_{e}\Rightarrow\ (p, aabbcdd, aaBcdd\#) \quad [1oA \rightarrow paa] \\
(p, aabbcdd, aaBcdd\#) &\ {}_{v}\Rightarrow\ (p, abbcdd, aBcdd\#) \\
(p, abbcdd, aBcdd\#) &\ {}_{v}\Rightarrow\ (p, bbcdd, Bcdd\#) \\
(p, bbcdd, Bcdd\#) &\ {}_{e}\Rightarrow\ (f, bbcdd, bbcdd\#) \quad [1pB \rightarrow fbb] \\
(f, bbcdd, bbcdd\#) &\ {}_{v}\Rightarrow\ (f, bcdd, bcdd\#) \\
(f, bcdd, bcdd\#) &\ {}_{v}\Rightarrow\ (f, cdd, cdd\#) \\
(f, cdd, cdd\#) &\ {}_{v}\Rightarrow\ (dd, dd\#) \\
(f, dd, dd\#) &\ {}_{v}\Rightarrow\ (f, d, d\#) \\
(f, d, d\#) &\ {}_{v}\Rightarrow\ (f, \varepsilon, \#)
\end{aligned}
\tag{6.1}
$$

Kapitola 7

Překlad

Překladač je program, který na zpracovává vstupní zdrojový program, který je napsán v nějakém zdrojovém jazyce a na výstup produkuje cílový program, uvedené v cílovém jazyce.

Samotný proces překlad se skládá z několika na sebe navazujících částí

- Lexikální analýza
- Syntaktická analýzy
- Sémantická analýza
- Generování vnitřního kódu
- Optimalizace vygenerovaného vnitřního kódu
- Generování cílového programu v cílovém jazyze

7.1 Lexikální analýza

- Vstup: Zdrojový program
- Výstup: Posloupnost tokenů

Lexikální analýza je uvodní částí celého překladu. Většinou se však nevyskytuje jako samostatná část, ale většinou jako procedura, či funkce synktaktické analýzy, která je volána, když syntaktická analýza potřebuje nový token. A právě tokeny jsou výstupem lexikální analýzy, kdy vždy po zavolání tato analýza přečte první, ještě nepřečtený lexikální symbol, nastaví mu potřebné parametry a takto zpracovaný token vrací syntaktické analýze.

Dalším úkolem lexikální analýzy je také například přeskakování komentářů, které nejsou pro samotný překlad podstatné a tudíž je ani neposílá syntaktické analýze.

7.2 Syntaktická analýza

- Vstup: Posloupnost tokenů
- Výstup: Derivační strom

Jak již bylo zmíněno, syntaktická analýza v podstatě ovládá lexikální analýzu a na základě dostaných tokenů od lexikální analýzy kontroluje, zda je syntaktická stránka vstupního programu v pořádku - tedy zda pořadí, či posloupnost lexikálních tokenů na sebe navazují v pořádku, dle pravidel daného jazyka. Pokud je tedy nalezený derivační strom, tak je program po syntaktické stránce správný.

V případě syntaktické analýzy máme dva přístupy - *Shora dolů* a druhý přístup *Zdola nahoru.*

7.2.1 Shora dolů

V tomto případě vytváříme derivační strom od jeho kořene směrem dolů, až ke koncovým listům.

7.2.2 Shora dolů

Druhým přístupem je tedy, jak již název vypovídá, postup kdy začínáme vytvářet strom od koncových listů nahoru, až ke kořeni tohoto derivačního stromu.

7.3 Sémantická analýza

- Vstup: Derivační strom
- Výstup: Abstraktní syntaktický strom

Jak již název vypovídá, tato analýza kontroluje sémantickou správnost zdrojového programu. Kontroluje tedy typy, tedy zda do proměnné jednoho typu nepřiřazujeme proměnnou jiného typu a dále také například kontroluje deklaraci všech proměnných.

7.4 Generování vnitřního kódu

- Vstup: Abstraktní syntaktický strom
- Výstup: Vnitřní kód

Tato část generuje nějaký vnitřní kód, kdy často se používá 3-adresný kód

7.5 Optimalizace vygenerovaného vnitřního kódu

- Vstup: Vnitřní kód
- Výstup: Optimalizovaný vnitřní kód

Pomocí optimalizace vnitřního kódu se můžeme zbavit například zbytečných, či mrtvých částí kódu. Díky tomuto se zbavíme zbytečných instrukcí a kód zefektivníme.

7.6 Generování cílového programu

- Vstup: Optimalizovaný vnitřní kód
- Výstup: Cílový program

V této části již pouze strojově přeložíme vnitřní optimalizovaný kód do cílového programu. Kód jsme již optimalizovali v předchozí části, takže zde již nemusíme nic kontrolovat a pouze přeložit.

7.7 Formální překlad

U formálního překladu a hned po tomto následujícím, syntaxí řízeném překladovém schématu jsem vycházel z [1].

Předpokládejme, že Σ je vstupní abeceda a Γ je výstupní abeceda. Po té definujeme překlad z jazyka $L_1 \subseteq \Sigma^*$ do jazyka $L_2 \subseteq \Gamma^*$ jako relaci T z Σ^* do Γ^*. Jestliže tedy existuje $(x, y) \in T$, pak je věta y nazývána *výstupem, či překladem* pro x.

U takového překladu je však možnost, že pro jeden vstup lze dostat více různých výstupů, což ne vždy je žádoucí jev.

7.8 Syntaxí řízené překladové schéma

A právě pro specifikaci nekonečné množiny zmíněné u formálního překladu jsou zavedeny syntaxí řízené předkladové schémata.

7.8.1 Definice

Syntaxí řízené překladové schéma je pětice $T = (N, \Sigma, \Gamma, R, S)$, kde

- N je konečná množina neterminálních symbolů
- Σ je konečná vstupní abeceda
- Γ je konečná výstupní abeceda
- R je konečná množina pravidel ve tvaru $A \rightarrow \alpha, \beta$, kde platí, že $\alpha \in (N \cup \Sigma)^*, \beta \in (N \cup \Gamma)^*$ a neterminaly v β jsou permutací neterminálů v α.
- $S \in N$ je počáteční neterminál

7.8.2 Překladová forma

Překladová forma T je definována následovně:

- (1) (S, S) je překladová forma, kde S jsou přidruženy
- (2) Pokud je $(\alpha A\beta, \alpha^1 A\beta^1)$ překladová forma, ve které jsou výskyty neterminálu A přidruženy a pokud existuje pravidlo $A \to \delta, \delta^1$ v R, pak $(\alpha\delta\beta, \alpha^1\delta^1\beta^1)$ je překladová forma. Neterminální symboly z δ, δ^1 jsou přidruženy v překladové formě stejně jako jsou přidruženy v pravidle. Neterminální symboly z α, β jsou přidruženy s těmi z α^1, β^1 v nové překladové formě naprosto stejně jako v původní překladové formě.
  Pak říkáme, že překladová forma $(\alpha A\beta, \alpha^1 A\beta^1)$ derivuje překladovou formu $(\alpha\delta\beta, \alpha^1\delta^1\beta^1)$, zapsáno $(\alpha A\beta, \alpha^1 A\beta^1) \Rightarrow (\alpha\delta\beta, \alpha^1\delta^1\beta^1)$, kde $\Rightarrow$ je překladová derivace.

Překlad definovaný pomocí T je označen jako $P(T)$
$(x, y)|(S, S) \Rightarrow^* (x, y), x \in \Sigma^* ay \in \Gamma^*$.

Kapitola 8

Konečný převodník

Konečný převodník je ve své podstatě rozšířený konečný automat. Takový automat rozšíříme o výstupní abecedu a tudíž v podstatě o možnost generovat nějaké výstupní řetězce. Tyto řetězce jsou generovány opět na základě pravidel. Z tohoto tedy plyne, že kromě přídání výstupní abecedy musíme ještě upravit i pravidla přechodu takového konečného automatu.

8.0.3 Definice

Konečný převodník je tedy definován jako šestice $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, kde

- Q je konečná množina stavů
- Σ je konečná vstupní abeceda
- Γ je konečná výstupní abeceda
- δ je zobrazení z $Q \times (\Sigma \cup \varepsilon)$ do množiny konečných podmožin $Q \times \Gamma^*$
- $q_0 \in Q$ je počáteční stav
- $F \subseteq Q$ je konečná množina koncových stavů

8.0.4 Konfigurace

Konfigurace konečnéhé převodníku je opět podobná konfiguraci konečného automatu, pouze doplněná o řerězec symbolů, které byly vypsány.

Z toho tedy plyne, že konfigurace je trojice (q, x, y), kde

- $q \in Q$ je aktuální stav konečného převodníku
- $x \in \Sigma^*$ je řetězec, který ještě nebyl načtený konečným převodníkem a tedy zbývá ještě k načtení
- $y \in \Gamma^*$ je řetězec, který byl pro zatím vygenerovaný konečným převodníkem

8.0.5 Přechod

Mějme 2 konfigurace konečného převodníku. (q, ax, y) a (r, x, zy), kde $q, r \in Q, x \in \Sigma^*, y, z \in \Gamma^*, a \in (\Sigma \cup \varepsilon)$. Následně můžeme říci, že z konfigurace (q, ax, y) přejdeme do (r, x, zy), pouze pokud existuje $\delta(q, a)$ obsahující (r, z).

Dále konfiguraci (q, x, y) označíme jako *počáteční konfiguraci* pokud $q = q_0, x \in \Sigma^*$ a $y = \varepsilon$.

Konfiguraci (q, x, y) označíme jako *koncovou konfiguraci* pokud $q \in F, x = \varepsilon$ a $y = \in \Gamma^*$.

Kapitola 9

Hluboký zásobníkový převodník

Podobně jako konečný převodník, tak i hluboký zásobníkový převodník je ve své podstatě pouze rozšířením hlubokého zásobníkového automatu o výstupní abecedu a tedy i jako konečný převodník o možnost generování výstupních řetězců na základě pravidel.

9.0.6 Definice

Hluboký zásobníkový převodník je tedy definován jako osmice $i_M = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$, kde

- Q je konečná množina stavů
- Σ je konečná vstupní abeceda
- Γ je konečná zásobníková abeceda, $\Sigma \subset \Gamma, \# \in (\Gamma - \Sigma)$ a $\Delta \subset \Gamma, \# \in (\Gamma - \Delta)$, kde $\#$ je speciální symbol značící dno zásobníku
- Δ je konečná výstupní abeceda
- δ je zobrazení z $I \times Q \times (\Sigma \cup \varepsilon) \times \Gamma$ do množiny konečných podmožin $Q \times \Gamma^* \times \Delta^*$
- $q_0 \in Q$ je počáteční stav
- $Z_0 \in Z$ je počáteční symbol na zásobníku
- $F \subseteq Q$ je konečná množina koncových stavů
- $i \in I$ je maximální hloubka, ve které může dojít k operaci

9.0.7 Konfigurace

Konfigurace zásobníku převodníku je opět podobná konfiguraci hlubokého zásobníkového automatu, pouze doplněná o řerězec symbolů, které byly vypsány.

Z toho tedy plyne, že konfigurace je čtveřice (q, x, y, z), kde

- $q \in Q$ je aktuální stav zásobníkového převodníku

- $x \in \Sigma^*$ je řetězec, který ještě nebyl načtený zásobníkovým převodníkem a tedy zbývá ještě k načtení
- $y \in (\Gamma^* - \{\#\})\{\#\}$ je aktuální stav zásobníku
- $z \in \Delta^*$ je řetězec, který byl pro zatím vygenerovaný zásobníkovým převodníkem

Kapitola 10

Výukový program

Jak již bylo zmíněno v úvodu celé této práce, jedná se o výukový program se zaměřením na zobrazení průncipů fungování lexíkální a syntaktické analýzy, kdy součástí syntaktické analýzy je i precedenční analýza. Program by tedy měl být k dispozici studentům IFJ a pomoci jim při studiu překladačů. Program je navržen tak, aby zobrazil každý jednotlivý krok lexikální, syntaktické a precedenční analýzy včetně jejich pořadí.

Program zpracovává uživatelem zadaný vstupní soubor, kdy podporovanými formáty jsou ".txt" a ".c". Uživateli jsou zobrazeny veškeré potřebné informace, jako například stav zásobníku pro syntaktickou, či precedenční analýzu, vstupní soubor, aktuální pozice programu v tomto souboru a mnoho dalších informací, které jsou zmíněny v kapitole věnované uživatelskému rozhraní [Kap 14] a měli by uživateli usnadnit pochopení celého procesu těchto analýz.

Po spuštění programu je uživali nabídnuto vybrat si ze dvou způsobů, jakými lze provádět analýzu. První možností uživatele je manuální kontrola a druhou kontrola časová.

Dále si uživatel musí vybrat, jaký typ analýzy by měl program provádět. Vybírat si může ze samotné lexikální analýzy, syntaktické analýzy s precedenční anebo si samozřejmě může nechat zobrazovat všechny tři analýzy - tedy lexikální, syntaktickou a precedenční.

10.0.8 Manuální kontrola

Při tomto typu kontroly je zcela na uživateli, kdy bude proveden další krok programu. Jedná se o krokování, kdy uživatel kontroluje provádění jednotlivých kroků a pomocí tlačítka pro další krok může další krok provést.

10.0.9 Časová kontrola

Zde si uživatel ještě na úvodní obrazovce zvolí časový interval, po kterém dojde k provedení další kroku. V případě, že by při běhu programu potřeboval více času na pochopení daného kroku, je mu v uživatelském rozhrání místo tlačítka pro další krok nabídnuto tlačítko pro prodloužení časového intervalu pro daný krok. Tento časový interval si užitel může prodlužovat dle vlastní potřeby.

Kapitola 11

Definice použitého jazyka

Při návrhu tohoto jazyka jsem se snažil vycházet z jazyka určeného pro IFJ projekt a také z jazyka C, se kterým by všichni studenti měli být obeznámeni.

Výsledný jazyk, použitý v této práci, je tedy kombinací výše zmíněných dvou jazyků. Jazyk je case-sensitive, kdy tedy při porovnávání rezervovaných slov, či identifikátorů záleží na velikosti jednotlivých písmen.

Vzhledem k tomu, že tento program zpracovává pouze lexikální a syntaktickou analýzu, bude jazyk popisován bez semantických vlastností nebo vlasností, které by mohli být využity pro interpretovatelnost, protože na tuto práci a program nemají žádný vliv.

11.0.10 Identifikátor

Definovaný jako neprázdná posloupnost velkých a malých písmen, číslic a podtržítka, kdy prvním znakem musí být písmeno. Dále také jazyk obsahuje množinu rezervovaných slov.

11.0.11 Rezervovaná slova

and, bool, break, do, else, end, false, loop, fun, findstr, for, function, if, in, int, local, main, nil, not, or, read, return, string, double, substr, then, true, typeof, until, while, write

11.0.12 Datové typy

Tento jazyk podporuje 4 různé datové typy.

- int - celá čísla (tedy i záporná)
- double - desetinná čísla (i záporná)
- bool - logický typ, který nabývá hodnot true a false

- string - řetězcový typ, kdy proměnná tohoto typu obsahuje hodnotu ohraničenou z obou stran znaky “, s tím, že takto ohraničený řetězec může obsahovat escape sekvence \n \t \\ \„

11.0.13 Struktura jazyku

Vstupním bodem zde je hlavní funkce programu - funkce *main*. Tato funkce je povinná a musí být v každém programu. Každý program se skládá tedy z neprázdné množiny funkcí, s tím, že na pozici funkce *main* v kódě nezáleží.

Každý příkaz musí být ukončen znakem “;„.

11.0.14 Struktura funkce

Každá funkce musí být ve tvaru :

Algorithm 1 struktura funkce

function IDENTIFIKATOR (seznam parametru)
seznam deklaraci promennych
seznam prikazu
end;

Seznam parametrů je posloupnost identifikátorů, které jsou odděleny od sebe čárkou, s tím, že za posledním parametrem se čárka neuvádí.

Tělo funkce se tedy skládá ze dvou částí - ze seznamu deklarací proměnných a seznamu příkazů.

Seznam deklarací proměnných se tedy skládá z množiny příkazů ve tvaru *datový_ typ identifikátor;* nebo *datový_ typ identifikátor = prirazeni_ hodnoty;*. Po definici všech proměnných se dostáváme do druhé části, tedy do seznamu příkazů, kde se již nesmí vyskytovat žádná deklarace proměnné, ale mohou se zde vyskytovat následující příkazy:

- **id = prirazeni;** U tohoto příkazu může být na pravé straně obyčejná hodnota, výraz nebo volání funkce, která vrací hodnotu.
- **write(posloupnost_retezcu);** Funkce write vypisuje na standartní výstup řetězce, proměnné či hodnoty, které jsou uvedeny v *posloupnost_ retezcu*.
- **if(vyraz) {seznam prikazu}else{seznam prikazu}** U tohoto podmíněného příkazu je *else větev* nepovinná a nemusí být uvedena.
- **while(vyraz){seznam prikazu}**

- **do{seznam prikazu}while(vyraz)**
- **return vyraz;** Příkaz pro návrat z dané funkce. Může vracet hodnotu, nebo být pouze ve tvaru *return;*.
- **fun id(params);** Provedení funkce id, kde jsem kvůli přehlednosti při provádění syntaktické analýzy přidal rezervované slovo *fun*.

11.0.15 Výrazy

Výrazy mohou být tvořeny proměnnými, literály, operacemi definovanými v následující podkapitole nebo samozřejmě i závorkami. Všechny výrazy jsou v programu zpracovávany precedenční analýzou.

11.0.16 Relační a aritmetické operátory

Následující operátory se mohou vyskytovat ve výrazech zmíněných v předešlé kapitole.

- +
- **-**
- *****
- /
- %
- **and**
- **or**
- <
- >
- ==
- **!=**
- >=
- <=

11.0.17 Vestavěné funkce

Pro názornější a realističtější ukázku jazyka je definováno i několik vestavěných funkcí :

- **typeof(promenna)**
- **substr(string, number, number)**
- **findstr(string,string)**

Kapitola 12

Gramatická pravidla a precedenční tabulka

Tato kapitola obsahuje všechna gramtická pravidla, které jsou rozdělena do dvou tabulek kvůli lepší přehlednosti a dále obsahuje také precedenční tabulku se všemi pravidly.

Při rozlišování terminálních a neterminálních symbolů v tabulkách gramatických pravidel budeme využívat standartizovaného pravidla, kdy terminální a neterminální symboly jsou rozlišeny na základě velikosti písmen. Terminální symboly se skládají pouze z malých písmen a neterminální symboly se skládají pouze z velkých písmen.

12.1 Gramatická Pravidla

Tabulka 12.1: První část pravidel

0	START	function idf (PARAMS) STATEMENT START
1		ε
2	PARAMS	id PARAMS_NEXT
3		ε
4	PARAMS_NEXT	,id PARAMS_NEXT
5		ε
6	WRITE	string WRITE_NEXT
7		VYRAZEPS
8	WRITE_NEXT	, WRITE
9		ε
10	SEKVENCE_PROM	int id DEFINICE ; SEKVENCE_PROM
11		double id DEFINICE ; SEKVENCE_PROM
12		string id DEFINICE ; SEKVENCE_PROM
13		bool id DEFINICE ; SEKVENCE_PROM
14		ε
15	SEKVENCE_PRIK	ε
16		id = PRIRAZENI ; SEKVENCE_PRIK
17		write (WRITE) ; SEKVENCE_PRIK
18		if (VYRAZ) SEKVENCE_PRIK JINAK SEKVENCE_PRIK
19		while (VYRAZ) SEKVENCE_PRIK SEKVENCE_PRIK
20		return VYRAZeps ; SEKVENCE_PRIK
21		do SEKVENCE_PRIK while (VYRAZ) SEKVENCE_PRIK
22		fun idf (PARAMS) ; SEKVENCE_PRIK

Tabulka 12.2: Druhá část pravidel

23	DEFINICE	= PRIRAZENI
24		ε
25	STATEMENT	SEKVENCE_PROM SEKVENCE_PRIK end ;
26	PRIRAZENI	VYRAZ
27		read(TYPE)
28		id
29		fun idf(PARAMS)
30		Typeof(id)
31		substr(PARAMS)
32		findstr(PARAMS)
33	JINAK	} else { SEKVENCE_PRIK
34		ε
35	VYRAZ	id
36		int_value
37		string_value
38		double_value
39		bool_value
40		(VYRAZ)
41		VYRAZ OPERACE VYRAZ
42	VYRAZEPS	int_value
43		id
44		ε
45		string_value
46		double_value
47		bool_value
48		(VYRAZ)
49		VYRAZ OPERACE VYRAZ
50	TYPE	int
51		double
52		bool
53		string
54	OPERACE	+
55		-
56		*
57		/
58		%
59		and
60		or
61		!=
62		==
63		>
64		<
65		>=
66		<=

12.2 Precedenční tabulka

Tabulka obsahuje čtyři různé znaky, které určují následující krok. Samotné implementaci i podrobnějšímu rozebrání významu znaků z tabulky je věnován větší prostor v následující kapitole.

Tabulka 12.3: Precedenční tabulka

	num	**id**	+	-	*	/	%	==	!=	<	>	<=	>=	(	)	**and**	**or**	**bool**	**str**	**$**
num	-	-	>	>	>	>	>	>	>	>	>	>	>	-	>	-	-	-	-	>
id	-	-	>	>	>	>	>	>	>	>	>	>	>	-	>	-	-	-	-	>
+	<	<	>	>	<	<	<	>	>	>	>	>	>	<	>	-	-	-	-	>
-	<	<	>	>	<	<	<	>	>	>	>	>	>	<	>	-	-	-	-	>
*	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>	-	-	-	-	>
/	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>	-	-	-	-	>
%	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>	-	-	-	-	>
==	<	<	<	<	<	<	<	>	>	<	<	<	<	<	>	-	-	<	<	>
!=	<	<	<	<	<	<	<	>	>	<	<	<	<	<	>	-	-	<	<	>
<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	-	-	-	<	>
>	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	-	-	-	<	>
<=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	-	-	-	<	>
>=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	-	-	-	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	-	-	<	-	-
)	-	-	>	>	>	>	>	>	>	>	>	>	>	-	>	-	-	-	-	>
and	<	<	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	<	-	>
or	<	<	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	<	-	>
bool	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	>	>	-	-	-
str	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	-	<	<	<	<	-

Kapitola 13

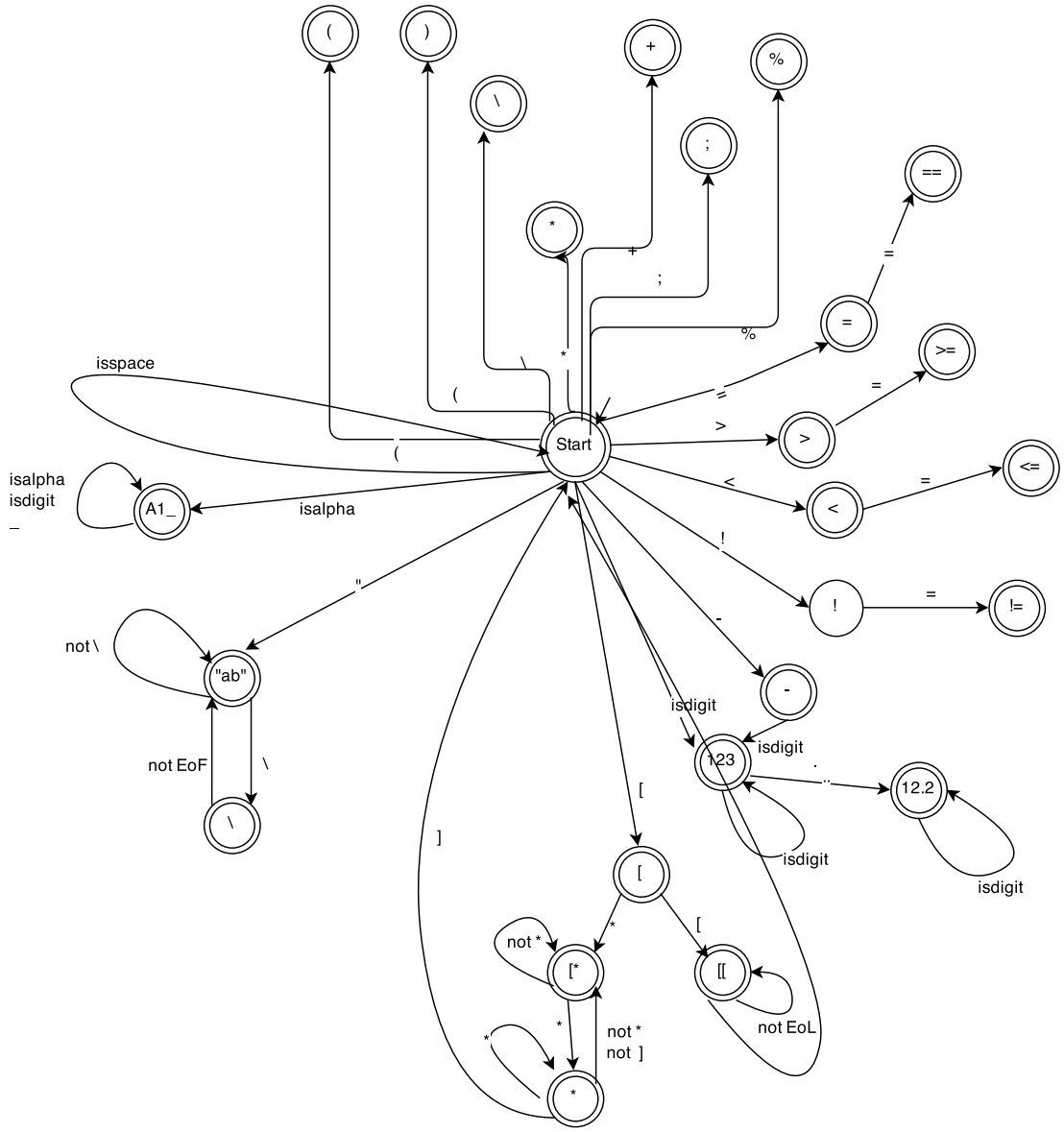
Návrh a implementace

Tato kapitola je věnována teoretickému návrhu a samotné implementaci jednotlivých částí. Pro každou část zde budou zmíněny datové struktury, čí třídy a jejich metody, kterých je využito při implementaci dané části.

Program, včetně uživatelského rozhraní byl implementován v programu QT Creator 5.3.2 na operačním systému Windows 7 64 bit.

13.1 Lexikální analýza

Lexíkální analýza je založena na konečném automatu o 26ti stavech, který je uveden v [Obr 13.1]. Tento konečný automat ignoruje bíle znaky, tedy mezery, tabulátory či jiné obdobné znaky, které přeskakuje. Podobně je tomu i u komentářů, kdy komentáře sice neignoruje a daný komentář načtě, avšak jej nepovažuje za token a tudíž jej nevrací syntaktické či precedenční analýze.



Obrázek 13.1: Grafické znázornění konečného automatu lexikální analýzy

13.2 Třídy lexikální analýzy

Lexikální analýza je implementována pomocí dvou tříd, kdy první třídou je samotný konečný automat `Lex` a druhou třídou je třída samotného tokenu `Token`. Konečný automat po načtění uloží data to objektu `Token`, který následně předá syntaktické, či precedenční analýze.

13.2.1 Třída Token

Jak již bylo řečeno, tato třída [13.1]slouží pro uložení tokenu načteného lexikální analýzou, díky čemuž třída kromě metod typu `Get` a `Set` neobsahuje žádné další metody a skládá se pouze z proměnných určených pro uložení načtených dat.

```
class Token
{
public:
    Token();
    Getters && Setters;

private:
    QString Name;
    double DValue;
    bool BValue;
    int IValue;
    QString SValue;

};
```

Listing 13.1: Třída Token

13.2.2 Třída Lex

Jak již název vypovídá, tak se jedná o třídu [13.2] reprezentující lexikální analýzu. Tato třída již obsahuje nejen proměnné, ale i metody, které v této kapitole jsou popsány.

```
class Lex
{
public:
    Lex(QString);
    Token *GetNextToken();
    void SetActRow(int v);
    int GetActRow();
    void IncActRow();
    void DecActRow();

private:
    QString CheckReservedWord(QString);

    int state;
    QFile SourceFile;
    int ActRow;
};
```

Listing 13.2: Třída Lex

První metodou je `GetNextToken`, která implementuje již znázorněný konečný automat o 26ti stavech. Vrací objekt typu Token, který obsahuje uložené veškeré potřebné informace. Následující 4 metody `SetActRow(), GetActRow(), IncActRow(), DecActRow()` slouží pro změny hodnoty aktuálního řádku ve zdrojovém souboru. Poslední metodou této třídy je metoda `CheckReservedWord`, která slouží pouze pro interní účely lexikální analýzy. Má jediný parametr, ve kterém dostává aktuálně načtené slovo a vrací lexikální analýze informaci o tom, zda je dané slovo rezervované, nebo zda se jedná o identifikátor (ať už proměnné či funkce).

Kromě těchto metod třída `Lex` obsahuje také tři proměnné, kdy proměnná `state` slouží pro uchovávání posledního stavu, `SourceFile` obsahuje ukazatel na soubor, ze kterého probíhá čtení tokenů. A poslední proměnou je `ActRow`, která obsahuje číslo aktuálního řádku ve zdrojovém souboru.

13.3 Syntaktická analýza

Pro implementaci této analýzy jsem využil zásobníku, který je reprezentovaný datovou strukturou `QList`, a oproti implementacím v předmětech IFJ a VYPe, kde jsme implementovali tuto analýzu pomocí struktury různých příkazu typu `switch, if`, jsem se

rozhodl tuto analýzu implementovat efektivněji, kdy jsem všechna pravidla jsem uložil do datové struktury a tudíž s nevelkou změnou kodu by bylo možné tuto práci upravit tak, aby fungovala pro různé množiny gramatických pravidel. Zvažoval jsem i možnost tyto pravidla do paměti nenačítat a nechat si je uložená v externím souboru, ale došel jsem k závěru, že výukový program by měl být co nejpřenositelnější a nezávislý na externích souborech.

13.4 Datové struktury syntaktické analýzy

13.4.1 Datová struktura Terms

Tato struktura [13.3] slouží pro uložení informací o jednom symbolu pravidla, kdy jedno pravidlo je tvořeno seznamem těchto symbolů.

Hodnota `Type` udává, zda se jedná o symbol terminální, či neterminální.`RulNumber` je množina pravidel, které mohou být provedeny v případě, že daný symbol je neterminální a zároveň je na vrcholu zásobníku syntaktické analýzy. `TerminalString` může nabývat dvou hodnot v závislosti na typu symbolu. Pro neterminální symbol je zde uložen název neterminálního symbolu (například SEKVENCE_PROM) a pro terminální symboly je zde uvedena hodnota vracená lexikální analýzou. Poslední proměnná `Rulz` obsahuje všechny možná pravidla, která mohou byt provedena v případě, že daný symbol je neterminální a na vrcholu zásobníku syntaktické analýzy. Je tedy v podstatě duplicitní k `RulNumber`, avšak s tím rozdílem, že zde jsou tato pravidla uložena ve formě jednoho řetězce, aby nemusela být prováděna častá, náročná a zbytečná konverze.

```
typedef struct
{
    QString Type;
    QList<int> RulNumber;
    QString TerminalString;
    QString Rulz;
}Terms;
```

Listing 13.3: Struktura Terms

13.4.2 Datová struktura Idf

Tato struktura [13.4] je používaná pouze pro ukládání načtených proměnných, či funkcí, které jsou následně uživateli zobrazovány. V této struktuře proměnné od funkcí neodlišuji, rozlišeny jsou až tím, ve kterém seznamu jsou, kdy mám dva seznamy typu `Idf`.

`Def` udává, zda daná funkce, či proměnná byla definována a `Init` zda daná funkce byla deklarována, respektive zda daná proměnný byla inicializovaná. A jak již udává samotné pojmenování, tak `Name` obsahuje jméno dané proměnné či funkce.

```
typedef struct
{
    bool Def,Init;
    QString Name;
}Idf;
```

Listing 13.4: Struktura Idf

13.5 Třídy syntaktické analýzy

U syntaktické analýzy jsem využil pouze jedné třídy, kdy tato třída implementuje syntakticou analýzu, včetně analýzy precedenční, kdy precedenční analýza je v podstatě implementována pouze jako jedna funkce.

13.5.1 Třída SecondWindow

Jak lze vidět, tak tato třída [13.5] je oproti předešlým již obsáhlejší. Metoda `NextStepLexOrSyn` slouží k provedení samostatné lexikální nebo syntaktické analýzy, v zavislosti na tom, co si uživatel vybral. V případě lexikální analýzy je zavolána metoda `GetNextToken` a po načtení tokenu dojde ke kontrole, zda nedošlo k lexikální chybě a aktualizaci uživatelského rozhraní. Další metodou je `NextStepBoth`, která je volána v případě, že si uživatel přeje zobrazování obou analýz. Pro načtění tokenu se používá i tato metoda `GetNextToken` a pro syntaktickou analýzu je volána metoda `DoSynStep` [13.6]. Tato klíčová metoda je popsána v Metoda `SetRulz`, slouží k počátečnímu nastavení syntaktického zásobníku a `LoadAllRulz` načítá všechna gramatická pravidla do `ArrRulz`. Poslední inicializační metodou je `SetPrecTable`, která do `PrecTable` načte celou precedenční tabulku. Jak již název vypovídá, tak metoda `IsPossibleEnd` je volána při načtění konce souboru ke zjištění, zda je ukončení souboru syntakticky korektní (například zda je zásobník prázdný nebo jestli již byla definována hlavní funkce `main`). Další metodou, která tvoří mezikrok mezi syntaktickou a precedenční analýzou, je `IsVyraz`, která kontroluje, zda nebude zpracováván výraz. Poslední dvě metody se již váží k precedenční analýze výrazu, kde metoda `MyTop` vrací terminální symbol nejblíže vrcholu precedenčního zásobníku a `GetPrecRulz` vrací na základě vstupních parametrů symbol z precedenční tabulky.

`ActRulz` obsahuje aktuální pravidlo a `ArrRulz` je posloupnost všech gramatických pravidel. `ArrRulz` je syntaktický zásobník používaný k překladu. `Vars` je posloupost všech načtených proměnných, včetně informace o tom, zda proměnná byla definována, popřípadně inicializována. `Funs` je posloupnost všech funkcí a opět včetně informací o jejich definici či deklaraci. Poslední tři proměnné jsou již určeny opět pro precedenční analýzu, kde `Precedenc` je zásobník obsahující aktuální konfiguraci, `PrecTable` je precedenční tabulka, jak již bylo zmíněno, a `PrecedenRulz` jsou precedenční pravidla.

```
class SecondWindow : public QMainWindow{
private:
    void NextStepLexOrSyn();
    void NextStepBoth();
    void SetRulz();
    void LoadAllRulz();
    void SetPrecTable();
    bool IsPossibleEnd();
    bool IsVyraz(int);
    QString GetPrecRulz(QString, QString);
    QString DoSynStep(Token *);
    int MyTop();
    Ui::SecondWindow *ui;
    Lex *Lexi;
    QList<Terms> ActRulz;
    QList<QList<Terms>> ArrRulz;
    QList<Terms> MainAt;
    QList<Idf> Vars;
    QList<Idf> Funs;
    QList<Terms> Precedenc;
    QMap<QString,QString> PrecTable;
    QList<QString> PrecedenRulz;
}
```

Listing 13.5: Třída SecondWindow

13.6 Funkce DoSynStep syntaktické analýzy

13.6.1 Popis funkce

V případě, že na vrcholu syntaktického zásobníku je terminální symbol, dojde k porovnání tohoto symbolu a načteného tokenu. Pokud právě načtený token odpovídá symbolu na vrcholu zásobníku, je symbol z vrcholu odstraněn. Pokud je daný token jméno funkce či proměnné, tak je přidán do seznamu funkcí, resp proměnných. Pokud právě načtený token neodpovídá symbolu na vrcholu, dojde k syntaktické chybě. V případě, že je na vrcholu neterminální symbol, projdou se všechna možná následující pravidla s daným neterminálem na levé straně a jejich první symbol na pravé straně. Pokud je tento symbol terminální,tak se porovná s právě načteným symbolem. V případě, že prvním symbolem na pravé straně daného pravidla je neterminální symbol, dojde k rozgenerování tohoto neterminálního symbolu na všechna možná další pravidla a tyto symboly se opět porovnají s právě načteným symbolem.

V případě, že nedojde k nalezení vhodného pravidla, syntaktická analýza končí s chybou. V opařném případě se neterminální symbol na vrcholu zásobníku rozgeneruje dle vybraného pravidla.

13.6.2 Ukázka funkce

```
DoSynStep(Token)
{
if (MainAt.Vrchol == Terminal){
  if (MainAt.Vrchol == Token.GetName)&&(IsFunction) {
    AddFunction(Token.GetSValue);
    MainAt.pop_back();
    ActualizeUI();
  } else
  if (MainAt.Vrchol == Token.GetName)&&(IsIdentifier) {
    AddIdentifierToken.GetSValue);
    MainAt.pop_back();
    ActualizeUI();
  } else
  if (MainAt.Vrchol == Token.GetName) {
    MainAt.pop_back();
    ActualizeUI();
  } else "SYN_ERROR"
}else{
  // Na vrcholu neterminal
  Projdi všechna pravidla:
    pro každé pravidlo vyber první symbol:
      pokud je to terminál a je stejný, jako právě načtený,
        tak vyber toho pravidlo
      pokud se jedná o neterminál, tak opět projdi všechny
        možná následující pravidla
  Return nalezene pravidlo, jinak vrat "SYN_ERROR"
}
}
```

Listing 13.6: DoSynStep

13.7 Precedenční analýza

Precedenční analýza je součástí třídy syntaktické analýzy, kde je implementována pouze jako několik funkcí. Zde budou zmíněny dvě základní metody této analýzy.

13.8 Funkce precedenční analýzy

13.8.1 MyTop

Tato metoda vrací pozici terminálního symbolu, který se nachází nejblíže vrcholu precedenčního zásobníku.

13.8.2 PrecStep

`PrecStep` je hlavní metodou precedenční analýzy, kdy tato metoda přebírá parametrem načtený lexikální token a na základě dvojice [Precedenc[MyTop()], Nacteny Token] je získaný symbol z PrecTable. V závislosti na symbolu mohou být provedeny 4 různé akce.

- = push_back(Nacteny Token)
- < Před terminál nejblíže vrcholu precedenčního zásobníku se vloží znak '<' a push_back(Nacteny Token)
- > V precedenčních pravidlech najdi na pravé straně <X a nahraď <X v precedenčním zásobníku levou stranou nalezeného pravidla. Pokud se nepodaří najít žádné pravidlo, tak "PREC_ERROR„
- **-** "PREC_ERROR„

13.9 Tabulka precedenčních pravidel

Tabulka 13.1: Precedenční pravidla

0	E	identifier
1	E	bool
2	E	num
3	E	E + E
4	E	E - E
5	E	E * E
6	E	E / E
7	E	E % E
8	E	E == E
9	E	E != E
10	E	E > E
11	E	E < E
12	E	E >= E
13	E	E <=E
14	E	E and E
15	E	E or E
16	E	(E)

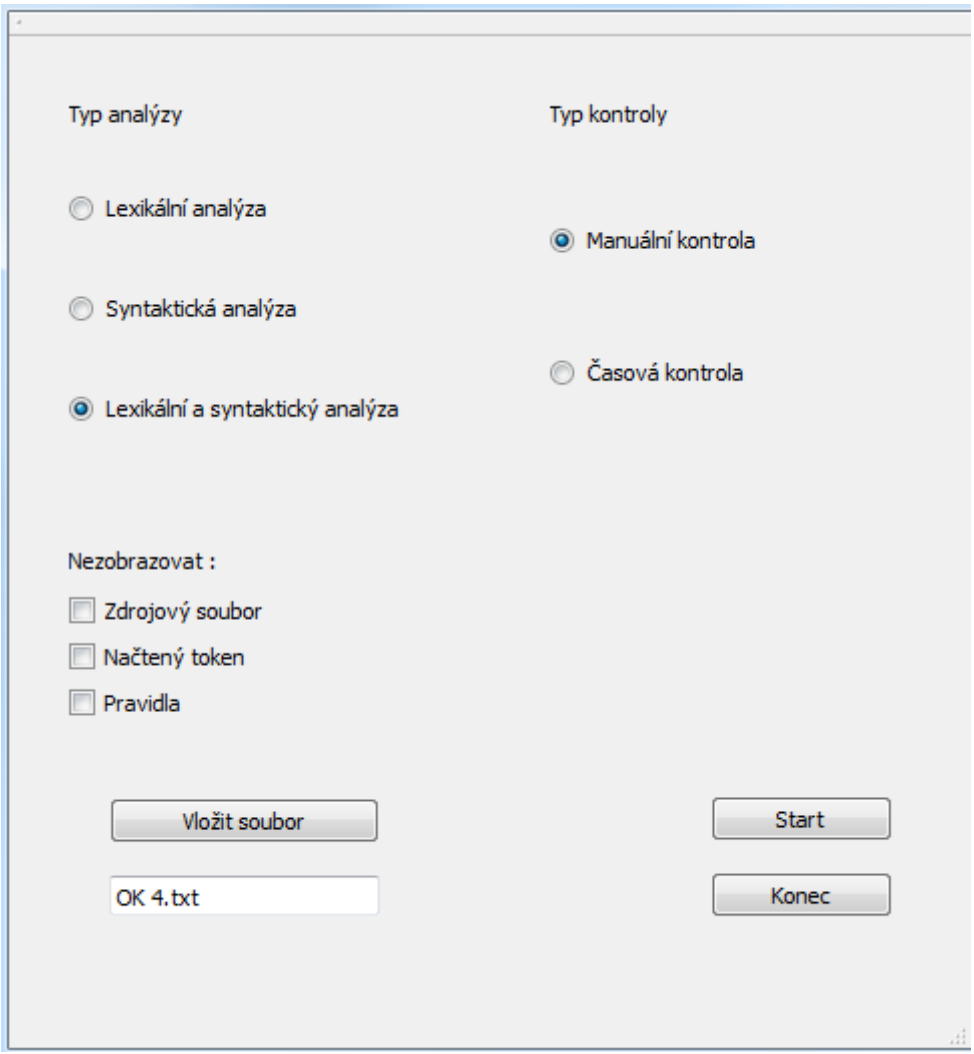
Kapitola 14

Uživatelské rozhraní

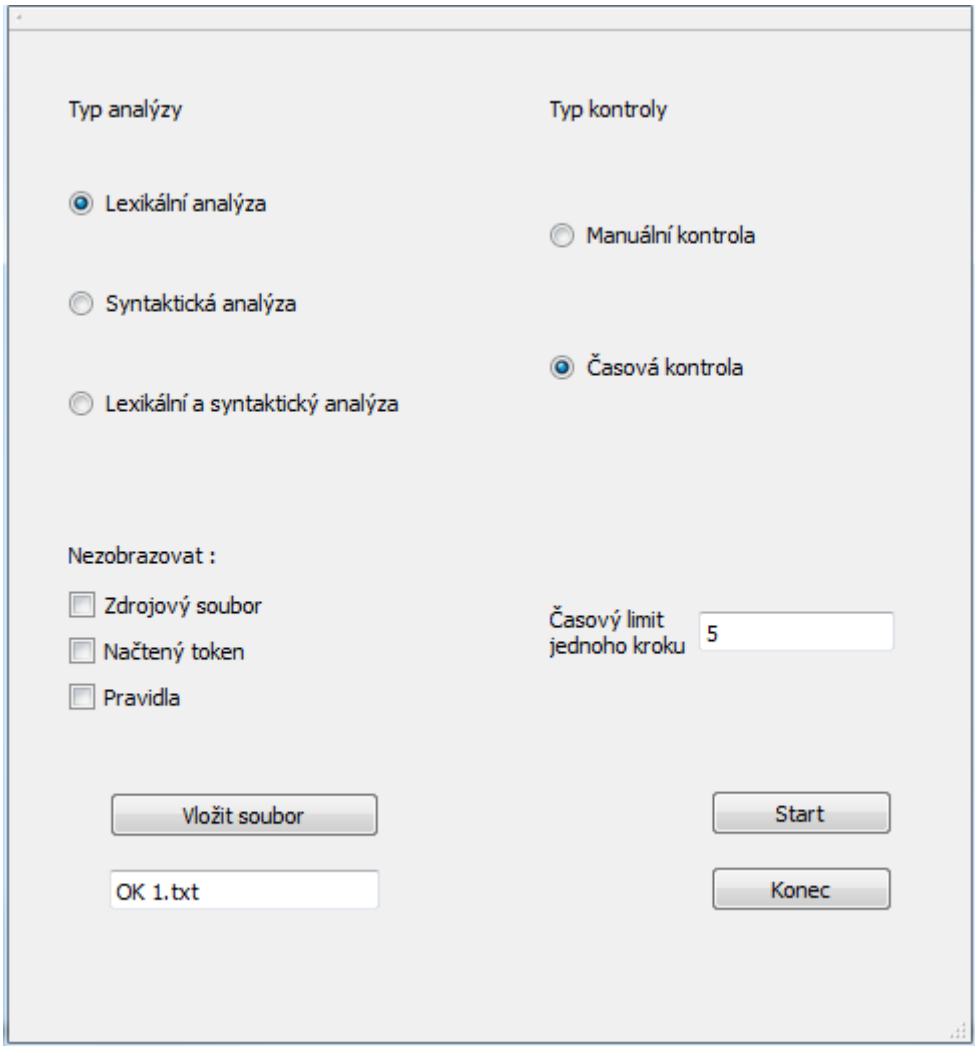
Vzhledem k fatku, že se jedná o výukový program, je grafické uživatelské rozhraní nedílnou součástí tohoto projektu, jehož hlavním cílem je vizualizace za účelem vysvětlení a pochopení principů lexikální, syntaktické a precedenční analýzy.

14.1 Úvodní obrazovka

Po spuštění programu je uživateli zobrazena úvodní obrazovka [Obr 14.1], kde si uživatel může načíst soubor se zdrojovým kódem, může si vybrat jaký typ analýzy bude zobrazen a také jaký typ kontroly chce uživatel použít. V případě, že si uživatel vybere typ kontroly časový, je mu umožněno určit si i časový limit pro jeden krok analýzy, jak lze vidět na [Obr 14.2].



Obrázek 14.1: Úvodní obrazovka uživatelského rozhraní



Obrázek 14.2: Úvodní obrazovka uživatelského rozhraní s časovým typem kontroly

14.2 Obrazovka analýzy

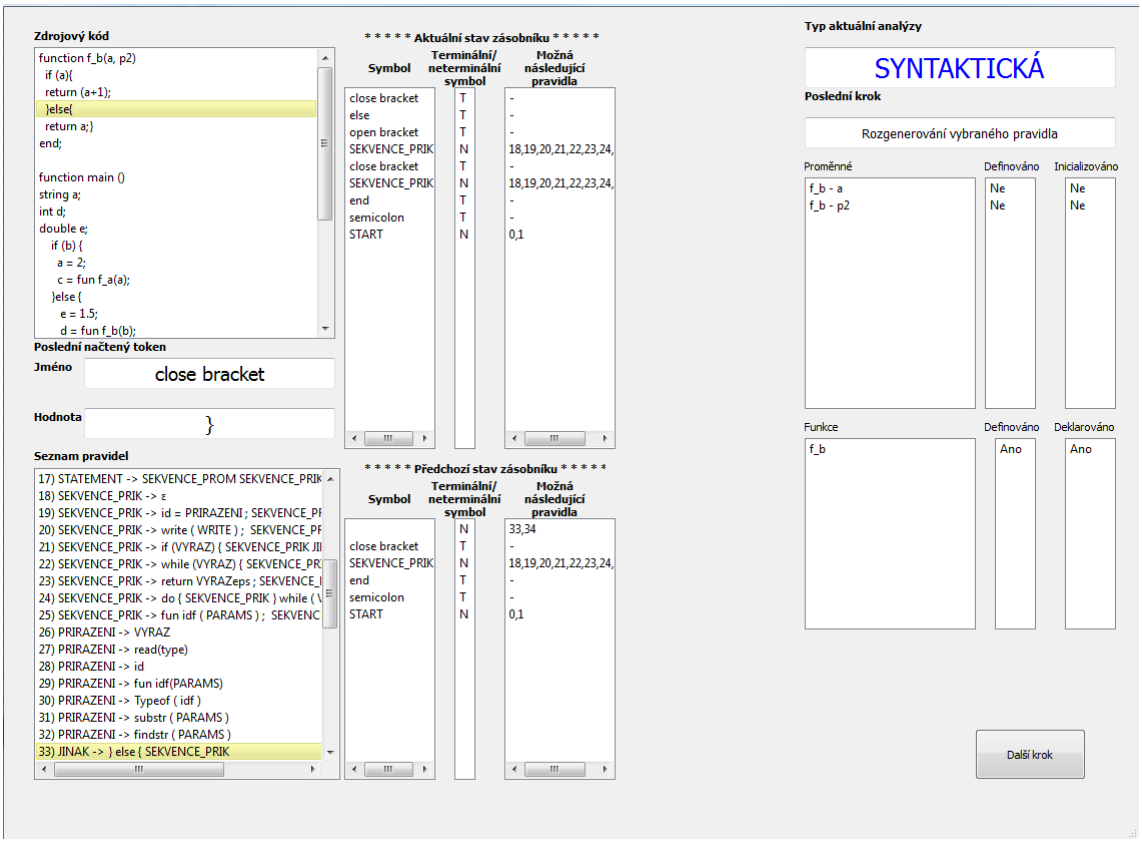
Tato obrazovka je již samotným srdcem celého programu, díky které by studenti měli pochopit samotné principy lexikální, syntaktické a precedenční analýzy.

Jak je vidět na [Obr 14.2], v levé horní části obrazovky je uživateli zobrazen zdrojový kód, který byl načten z uživatelem zadaného souboru, a dále je také zvýrazněn i aktuální řádek v souboru, ze kterého probíhá načítání tokenů. Pod zobrazením zdrojového souboru je vždy zobrazen poslední načtený token. V levé dolní částí obrazovky jsou uživateli zobrazeny všechna syntaktická pravidla a v případě, že je jako typ analýzy vybraná syntaktická, nebo syntaktická a lexikální, je uživateli opět zvýrazněno poslední použité pravidlo.

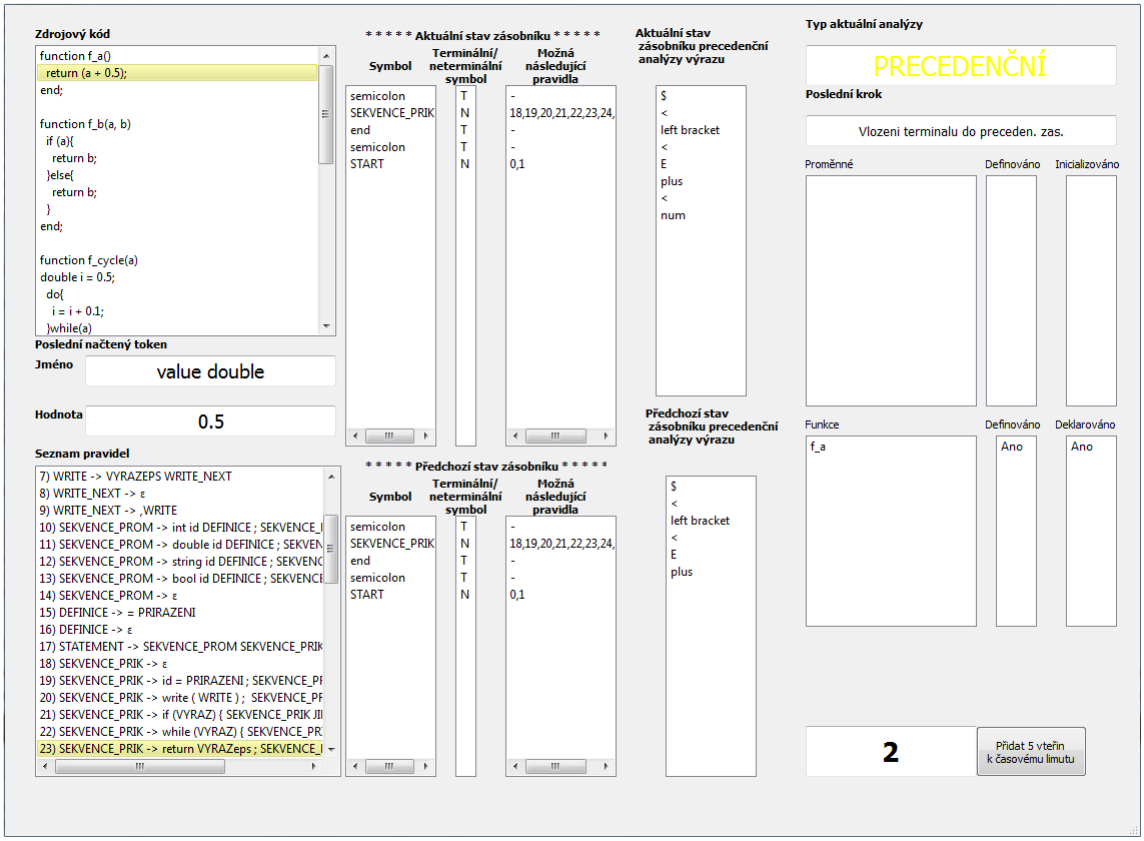
Prostřední části obrazovky dominuje zobrazení syntaktického zásobníku, kde je uživateli zobrazen vždy jak zásobník před vykonáním aktuálního kroku, tak i po vykonání daného kroku. Jsou zde zobrazeny veškeré potřebné informace, jako již zmíněný obsah zásobníku, informace o tom, zda se jedná o terminální, či neterminální symboly, a také pro každý neterminální symbol jsou zobrazena všechná možná pravidla pro rozgenerování daného neterminálního symbolu.

V případě, že se program dostal k výrazu a probíhá precedenční analýza, je uživateli zobrazen ještě stav precedenčního zásobníku, jak lze vidět na [Obr 14.2]. Opět je zobrazen jak stav precedenčního zásobníku před provedením daného kroku, tak i po provedení daného kroku.

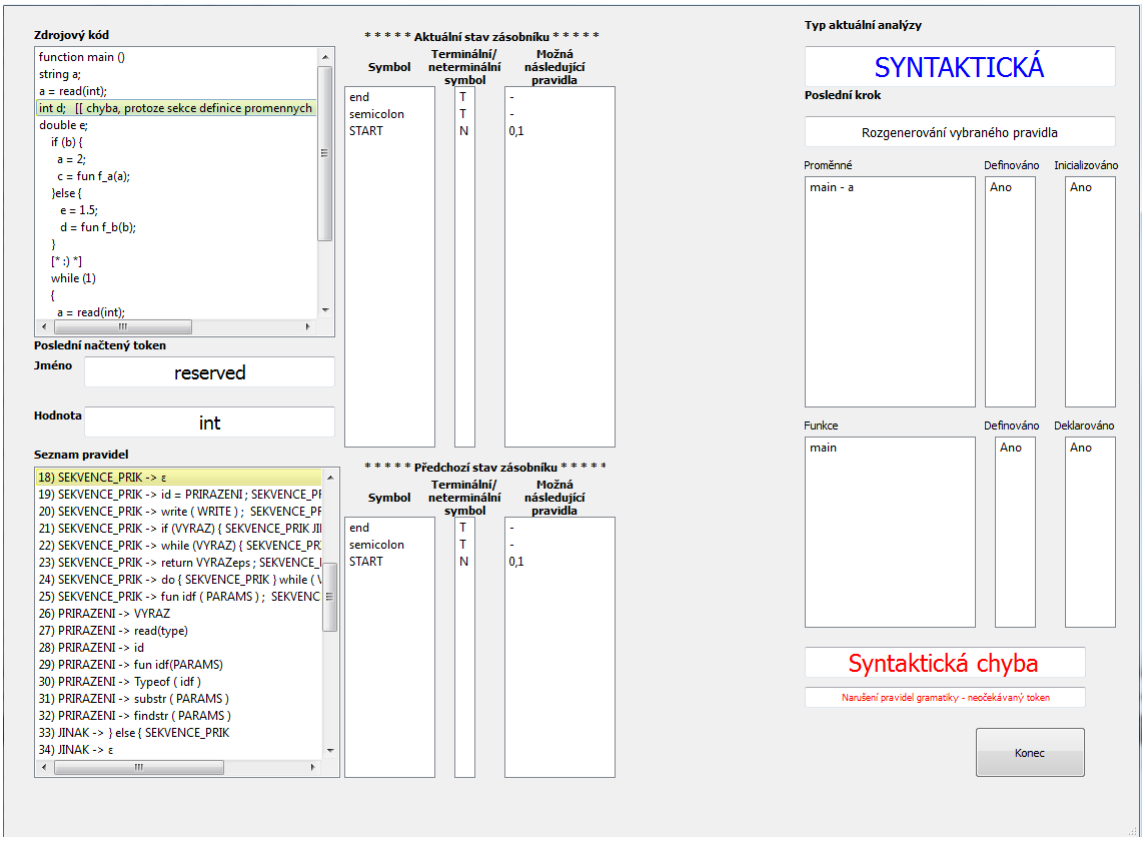
V pravém horním rohu obrazovky je uživateli vždy zobrazena aktuální analýza a pod ní i krok, který byl právě v rámci dané analýzy proveden. Dále je v této části obrazovky také seznam všech proměnných i s informacemi o jejich definici a inicializaci a samozřejmě i seznam všech funkcí, kdy ani zde opět nechybí informace o definici a deklaraci daných funkcí. Pod tímto seznam funkcí se nachází ještě místo [Obr 14.2], kde je uživatel informován o případné chybě, která vznikla při analýze.



Obrázek 14.3: Obrazovka analýzy se syntaktickou analýzou



Obrázek 14.4: Obrazovka analýzy s precedenční analýzou analýzou a časovým typem kontroly



Obrázek 14.5: Obrazovka analýzy se syntaktickou chybou

Kapitola 15

Závěr

Tato diplomová práce se zabývala hlubokým, syntaxí řízeným překladem a prostředy potřebnými pro něj. Na začátku práce byly definovány veškeré potřebné teoretické pojmy, kdy jsem čerpal zejména z následujících zdrojů [9],[7] a [4]. Dále na začátku této práce byly také definovány různé typy jazyků, gramatik a chomského hierarchie.

Po teoretickém úvodu byly zavedeny různé typy automatů, od konečných až po hluboké zásobníkové automaty, při kterých jsem vycházel ze své bakalářské práce. Dále jsem také definoval překlad, všechny části překladu a pro tuhle práci důležité převodníky - jak konečné, tak i zásobníkové. Po zavedení pojmů převodníků jsem s využítím hlubokých zasobníkových automatů zavedl hluboké zásobníkové převodníky.

Poslední část práce byla věnována implementaci samotného vyukového programu. V této části jsem definoval jazyk, který je používán pro analýzu, definoval jsem všechny důležité datové struktury, třídy a funkce. U těch nejdůležitějších funkcí jsem se věnoval i jejich implementaci. Poslední kapitola této práce již byla věnována grafickému uživatelskému rozhraní, vysvětlení jeho rozvržení a toho co může uživateli poskytnout.

Literatura

[1] Alfred V. Aho, J. D. U.: *The theory of parsing, translation, and compiling, Volume I: Parsing.* Prentice Hall, 1972, iSBN 0139145567.

[2] Kasai, T.: An hierarchy between context-free and context-sensitive languages. *Joural of computer and system sciences*, ročník 4, 1970: s. 492–508.

[3] Češka M., S. A., Vojnar T.: Teoretická informatika TIN - studijní opora.

[4] Meduda Alexander, L. R.: Formální jazyky a překladače, Kapitola I. Abecedy, řetězce a jazyky.

[5] Meduda Alexander, L. R.: Formální jazyky a překladače, Kapitola III. Modely pro regulární jazyky.

[6] Meduda Alexander, L. R.: Formální jazyky a překladače, Kapitola VI. Modely pro bezkontextové jazyky.

[7] Meduna, A.: *Automata and Languages.* Springer, 2000, iSBN 8181283333.

[8] Meduna, A.: Deep pushdown automata. *Acta informatica*, ročník 98, 2006: s. 114–124.

[9] Meduna, A.: *Elements of Compiler Design.* Auerbach Publications, 2007, iSBN 1420063235.

[10] Peter, S.: Syntaxí řízený překlad založený na hlubokých zásobníkových automatech. 2009.

[11] Senko, J.: Syntaktická analýza s použitím hlubokých zásobníkových automatů. 2013.

Dodatek A

Obsah CD