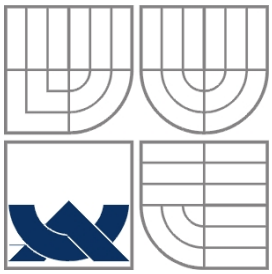
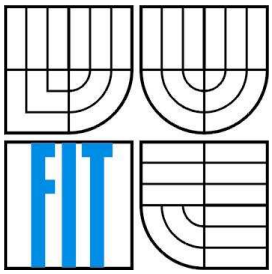


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYNTAKTICKÁ ANALÝZA ZALOŽENÁ NA MULTIGENEROVÁNÍ

PARSING BASED ON MULTIGENERATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. Linda Kyjovská
AUTHOR

VEDOUCÍ PRÁCE — Ing. Roman Lukáš, Ph.D.
SUPERVISOR

BRNO 2008

Abstrakt

Práce se zabývá problematikou syntaktické analýzy založené na multigenerování. Cílem je vytvořit program, který zadaný vstupní řetězec transformuje na *n*-1 výstupních řetězců. Vstupem programu je uživatelem vytvořený textový soubor obsahující pravidla *n* gramatik. Právě jedna gramatika je označena za vstupní a zbývajících *n*-1 gramatik se stává výstupními gramatikami. Na základě vstupní gramatiky se provede syntaktická analýza uživatelem zadaného řetězce, která nám určí použitá gramatická pravidla. Paralelně s touto analýzou vytváříme výstupní řetězce za použití zbývajících *n*-1 gramatik. Implementace bude provedena pomocí technologií C++ a Bison.

Klíčová slova

Syntaktická analýza, lexikální analýza, multigenerování, terminál, nonterminál, formální jazyk, gramatika, bezkontextová gramatika, zásobníkový automat, LR-syntaktická analýza, Bison.

Abstract

This work deals with syntax analysis problems based on multi-generation. The basic idea is to create computer program, which transforms one input string to *n*-1 output strings. An Input of this program is some plain text file created by user, which contains n grammar rules. Just one grammar from the input file is marked as an input grammar and others *n*-1 grammars are output grammars. This program creates list of used input grammar rules for an input string and uses corresponding output grammar rules for the creation of *n*-1 output strings. The program is written in C++ and Bison

Key words

Syntax analysis, lexical analysis, multi-generation, terminal, nonterminal, formal language, grammar, context-free grammar, pushdown automaton, LR-syntax analysis, Bison.

Citace

Linda Kyjovská: Syntaktická analýza založená na multigenerování, diplomová práce, Brno, FIT VUT v Brně, 2008

Syntaktická analýza založená na multigenerování

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením ing. Romana Lukáše,Ph.D.
Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

……
Datum a podpis

Poděkování

Na tomto místě bych ráda poděkovala ing. Romanu Lukášovi, Ph.D za odborné vedení, užitečné rady, připomínky, konzultace a recenzi projektu.

Obsah

1. Úvod

Ve své diplomové práci se zabývám metodou syntaktické analýzy založené na multigenerování. Navrhla jsem vlastní metodu multigenerování, která splňuje následující vlastnosti:

- je paralelní – ze vstupního řetězce se paralelně generují výstupní řetězce
- při implementaci byl použit zásobník
- využívá syntaktickou analýzu zdola nahoru
- právě jedna gramatika je vstupní
- *n*-1 gramatik je výstupních

Téma mé diplomové práce jsem si vybrala záměrně, protože mě velice nadchl povinný předmět Formální jazyky a překladače, který jsem v rámci výuky na VUT absolvovala. Proto jsem se chtěla blíže seznámit s principy, které se v tomto oboru využívají.

Praktické uplatnění vidím například pro převod z infixového zápisu zároveň na postfixový a prefixový. Dále se dá moje metoda využít při překladu z jednoho jazyka do libovolného množství jiných jazyků, jak také bude uvedeno později. Mým cílem je metodu zobecnit natolik, aby byla použitelná pro vzájemný překlad mezi vyššími programovacími jazyky.

Cílem mé diplomové práce je vytvořit program, který zadaný vstupní řetězec, respektive soubor obsahující zdrojový kód zapsaný v některém z programovacích jazyků, transformuje na *n*-1 výstupních řetězců, respektive *n*-1 výstupních souborů obsahujících zdrojové kódy zapsané v jiných programovacích jazycích. Vstupem programu bude uživatelem vytvořený textový soubor obsahující pravidla *n* gramatik a textový soubor obsahující klíčová slova a symboly komentářů. Pravidla gramatik i klíčová slova se do souborů musí zapisovat podle určitých zásad. Právě jedna gramatika je označena za vstupní a zbývajících *n*-1 gramatik se stává výstupními gramatikami. Na základě vstupní gramatiky se provede syntaktická analýza uživatelem zadaného řetězce, která určí použitá gramatická pravidla. Paralelně s touto analýzou se vytváří výstupní řetězce za použití zbývajících *n*-1 gramatik. Syntaktická analýza využívá práce lexikální analýzy, která prochází soubor obsahující zdrojový kód zapsaný v některém z programovacích jazyků, a vrací syntaktické analýze jednotlivé lexémy.

Implementace bude provedena pomocí technologií C++ a Bison. Při implementaci musím počítat s tím, že syntaktickou analýzu si provádí překladač Bison sám. Ten používá syntaktickou analýzu zdola nahoru, což může zkomplikovat zajištění paralelního generování výstupních řetězců z řetězce vstupního. Tento problém budu řešit využitím pole zásobníků, kde se každá položka zásobníku bude skládat ze struktury obsahující název nonterminálu a jemu odpovídající

vygenerovaný řetězec terminálů. Program bude možné využít nejen pro překlad mezi vyššími programovacími jazyky, ale pro jakýkoliv překlad podle uživatelem zadaných gramatik.

Diplomová práce je rozdělena do sedmi kapitol. Ve druhé kapitole se věnuji základním pojmům a jejich definicím. Tato kapitola je teoretickým úvodem, který zajišťuje seznámení se s problematikou formálních jazyků a gramatik.

Ve třetí kapitole popisuji princip mého řešení, který vysvětluji na názorných příkladech.

Ve čtvrté kapitole popisuji implementaci jednotlivých částí programu, použité technologie, implementaci jednotlivých metod. Princip metod je vysvětlen pomocí názorných obrázků.

V páté kapitole popisuji použití jednotlivých pravidel. Vysvětluji proč jsem zvolila určitá omezení pro vstupní zdrojový kód dodaný uživatelem, a proč jsem si vybrala jednotlivá řešení.

V šesté kapitole popisuji práci s programem a postup při jeho spouštění.

Diplomová práce navazuje na semestrální projekt. Z tohoto projektu přebírá teoretickou část, která je uvedena ve druhé kapitole a v první části třetí kapitoly. V semestrálním projektu byl zmíněn i princip metody, který je základem pro implementaci. V diplomovém projektu se teoretický úvod převádí do praxe a řeší se problémy s tím spojené.

2. Základní definice

Než se dostaneme k definici a popisu syntaktické analýzy založené na multigenerování, bylo by dobré si vysvětlit základní pojmy. Se syntaktickou analýzou souvisí bezkontextová gramatika a bezkontextový jazyk. Syntaktická analýza používá pravidla bezkontextové gramatiky pro vytvoření derivačního stromu.

Dále v této kapitole vysvětlím pojem lexikální analýzy a formálně popíši některé části týkající se principu mé metody.

Pro snadnější pochopení problematiky začneme definicemi obecnějších pojmů.

2.1 Základní pojmy

2.1.1 Definice abecedy

Abeceda je neprázdná konečná množina prvků, které nazýváme symboly.

2.1.2 Definice řetězce nad danou abecedou

Nechť Σ je abeceda. Potom:

- ε je řetězec nad abecedou Σ.
- Jestliže x je řetězec nad abecedou Σ, $a \in \Sigma$, potom xa je řetězec nad abecedou Σ.

Symbol ε značí *prázdný řetězec*. Prázdný řetězec je takový řetězec, který neobsahuje žádný symbol.

Symbolem Σ^* budeme značit množinu všech řetězců nad abecedou Σ.

Příkladem abecedy může být $\{a, b\}$, řetězcem nad touto abecedou je například *ababba*.

2.1.3 Definice délky řetězce

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

- Pokud $x = \varepsilon$, potom $|x| = 0$.
- Pokud $x = a_1 a_2 \ldots a_n$, kde $a_i \in \Sigma$ pro všechna $i = 1, 2, \ldots n$, potom $|x| = n$.

2.1.4 Formální jazyk

Nechť je dána abeceda Σ. Potom množinu *L*, pro kterou platí: $L \subseteq \Sigma^*$, nazveme *formálním jazykem* nad abecedou Σ.

V matematice, logice a informatice se pojmem **formální jazyk** označuje množina řetězců nad určitou abecedou. Definice pojmu *formální jazyk* se může měnit podle toho, v jakém kontextu a v jakém vědním oboru jej používáme.

Příklady formálních jazyků:

- množina všech řetězců nad abecedou *a, b*
- množina $\{a^n$: *n* je prvočíslo$\}$, kde a^n znamená, že *a* se vyskytuje *n*-krát za sebou
- množina všech programů v daném programovacím jazyce

Formální jazyk může být definován různými způsoby, například:

- množina řetězců generovaná nějakou formální gramatikou (viz. Chomského hierarchie);
- množina řetězců vyhovující nějakému regulárnímu výrazu;
- množina řetězců akceptovaná nějakým automatem, například konečným automatem

2.2 Definice gramatik

V informatice se pojmem **gramatika** označuje struktura, která popisuje formální jazyk. Pojmenování je zvoleno kvůli podobnosti s gramatikami používanými v přirozených jazycích.

Gramatika se skládá z množiny pravidel, pomocí kterých může být každé slovo předepsaným způsobem *vygenerováno* z předem daného počátečního symbolu. Generování probíhá tak, že vezmeme počáteční symbol, na něj aplikujeme kterékoli z pravidel, na získaný řetězec opět aplikujeme kterékoli z pravidel atd., dokud nevygenerujeme požadovaný řetězec. Pokud je pro každý řetězec nejvýše jeden postup generování, gramatika je **jednoznačná.**

2.3 Chomského hierarchie gramatik

Gramatiky jsou hierarchicky uspořádány. Toto uspořádání zavedl Noam Chomsky (*7. prosince 1928) - je americký lingvista, tvůrce Chomského hierarchické klasifikace formálních jazyků, profesor MIT, ale také levicově orientovaný aktivista, známý svým kritickým vztahem ke globalizaci a jejím dopadům, k USA a k jejich angažovanosti v mezinárodních konfliktech.

Chomského hierarchie je hierarchie tříd formálních gramatik generujících formální jazyky. Byla vytvořena v roce 1956.

Chomského hierarchie se skládá ze čtyř tříd.

2.3.1 Gramatiky typu 0 (neomezené gramatiky)

Neomezená gramatika je gramatika, která obsahuje nejobecnější tvar pravidel. Množinu všech jazyků, které jsou generovány nějakou neomezenou gramatikou, nazveme třídou rekurzivně vyčíslitelných jazyků nebo také třídou jazyků typu 0. Formální definice je následující:

2.3.1.1 Definice neomezené gramatiky

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

- N je konečná množina nonterminálních symbolů,
- T je konečná množina terminálních symbolů, přičemž $N \cap T = \emptyset$,
- P je konečná množina pravidel tvaru $x \rightarrow y$, kde $x \in (N \cup T)^* N (N \cup T)^*$ a $y \in (N \cup T)^*$,
- S je počáteční nonterminální symbol.

2.3.1.2 Definice přímé derivace u neomezené gramatiky

Nechť $G = (N, T, P, S)$ je neomezená gramatika, nechť $u, v \in (N \cup T)^*$ a $p = x \rightarrow y \in P$ je pravidlo. Pak říkáme, že *uxv přímo derivuje uyv* podle pravidla p a zapisujeme $uxv \Rightarrow uyv\ [p]$ nebo také zkráceně $uxv \Rightarrow uyv$.

2.3.1.3 Definice sekvence derivací u neomezené gramatiky

Nechť $G = (N, T, P, S)$ je neomezená gramatika.

- Nechť $u \in (N \cup T)^*$. Pak říkáme, že u derivuje *v 0-krocích* u a zapisujeme $u \Rightarrow^0 u\ [\varepsilon]$ nebo také zkráceně $u \Rightarrow^0 u$.
- Nechť $u_0, u_1, \ldots, u_n \in (N \cup T)^*$, nechť pro všechna $i = 1, ..., n$ platí: $u_{i-1} \Rightarrow u_i\ [p_i]$. Pak říkáme, že u_0 *derivuje v n-krocích* u_n a zapisujeme $u_0 \Rightarrow^n u_n\ [p_1 p_2 \ldots p_n]$ nebo také zkráceně $u_0 \Rightarrow^n u_n$.
- Nechť $u \Rightarrow^n v\ [\pi]$ pro nějaké $n \geq 1$, $u, v \in (N \cup T)^*$. Pak říkáme, že u *netriviálně derivuje* v a zapisujeme $u \Rightarrow^+ v\ [\pi]$ nebo také zkráceně $u \Rightarrow^+ v$.
- Nechť $u \Rightarrow^n v\ [\pi]$ pro nějaké $n \geq 0$, $u, v \in (N \cup T)^*$. Pak říkáme, že u *derivuje* v a zapisujeme $u \Rightarrow^* v\ [\pi]$ nebo také zkráceně $u \Rightarrow^* v$.

2.3.1.4 Definice jazyka generovaného neomezenou gramatikou

Nechť $G = (N, T, P, S)$ je neomezená gramatika. Jazyk generovaný neomezenou gramatikou G (budeme jej označovat $L(G)$) je definován následovně:

$$L(G) = \{w : w \in T^* \wedge S \Rightarrow^* w\}.$$

2.3.1.5 Definice rekurzivně vyčíslitelného jazyka

Jazyk je rekurzivně vyčíslitelný právě tehdy, když je generován nějakou neomezenou gramatikou.

2.3.2 Gramatiky typu 1 (kontextové gramatiky)

Kontextová gramatika je speciální neomezená gramatika obsahující pravidla tvaru $x \rightarrow y$, pro která navíc platí, že $|x| \leq |y|$. Množinu všech jazyků, které jsou generovány nějakou kontextovou gramatikou, nazveme třídou kontextových jazyků nebo také třídou jazyků typu 1. Formální definice je následující:

2.3.2.1 Definice kontextové gramatiky

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

- N je konečná množina nonterminálních symbolů,
- T je konečná množina terminálních symbolů, přičemž $N \cap T = \emptyset$,
- P je konečná množina pravidel tvaru $x \rightarrow y$, kde $x \in (N \cup T)^* N (N \cup T)^*$ a $y \in (N \cup T)^*$, přičemž $|x| \leq |y|$,
- S je počáteční nonterminální symbol.

2.3.2.2 Definice přímého derivačního kroku, sekvence derivačních kroků a jazyka definovaného kontextovou gramatikou

Tyto definice jsou shodné s příslušnými definicemi u neomezené gramatiky.

2.3.2.3 Definice kontextového jazyka

Jazyk je kontextový právě tehdy, když je generován nějakou kontextovou gramatikou.

2.3.3 Gramatiky typu 2 (bezkontextové gramatiky)

Bezkontextová gramatika je speciální kontextová gramatika obsahující pravidla tvaru $x \rightarrow y$, pro která navíc platí, že řetězec x je pouze jeden nonterminální symbol. Množinu všech jazyků, které

jsou generovány nějakou bezkontextovou gramatikou, nazveme třídou bezkontextových jazyků nebo také třídou jazyků typu 2. Formální definice je následující:

2.3.3.1 Definice bezkontextové gramatiky

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

- N je konečná množina nonterminálních symbolů,
- T je konečná množina terminálních symbolů, přičemž $N \cap T = \emptyset$,
- P je konečná množina pravidel tvaru $A \rightarrow x$, kde $A \in N$ a $x \in (N \cup T)^*$,
- S je počáteční nonterminální symbol.

Název „bezkontextová“ vychází ze skutečnosti, že nonterminál A se může přepsat na x bez ohledu na okolní *kontext*. Jazyky generované bezkontextovými gramatikami se nazývají bezkontextové.

2.3.3.2 Definice přímého derivačního kroku, sekvence derivačních kroků a jazyka definovaného bezkontextovou gramatikou

Tyto definice jsou shodné s příslušnými definicemi u neomezené gramatiky.

2.3.3.3 Definice bezkontextového jazyka

Jazyk je bezkontextový právě tehdy, když je generován nějakou bezkontextovou gramatikou.

2.3.3.4 Příklad

Uvažujme následující bezkontextovou gramatiku:

$G = (N, T, P, S)$, kde:

$N = \{S\}$

$T = \{x, y, z, +, -, *, /, (,)\}$,

$P = \{S \rightarrow x, S \rightarrow y, S \rightarrow z, S \rightarrow S + S, S \rightarrow S - S, S \rightarrow S * S, S \rightarrow S / S, S \rightarrow (S)\}$

Tato bezkontextová gramatika generuje jazyk všech aritmetických výrazů obsahující proměnné *x*, *y*, *z*. Může například vygenerovat řetězec “(*x*+*y*)**x*-*z***y*/(*x*+*x*)“

2.3.4 Gramatiky typu 3 (Pravé lineární gramatiky)

Pravá lineární gramatika je speciální bezkontextová gramatika obsahující pravidla tvaru $A \rightarrow x$, pro která navíc platí, že řetězec x obsahuje buď samé terminální symboly a nebo terminální symboly zakončené pouze jedním nonterminálním symbolem. Množinu všech jazyků, které jsou generovány

nějakou pravou lineární gramatikou, nazveme třídou regulárních jazyků nebo také třídou jazyků typu 3. Formální definice je následující:

2.3.4.1 Definice pravé lineární gramatiky

Pravá lineární gramatika G je čtveřice $G = (N, T, P, S)$, kde:

- N je konečná množina nonterminálních symbolů,
- T je konečná množina terminálních symbolů, přičemž $N \cap T = \emptyset$,
- P je konečná množina pravidel tvaru $A \rightarrow xB$ nebo $A \rightarrow y$, kde $A, B \in N$ a $x, y \in T^*$,
- S je počáteční nonterminální symbol.

2.3.4.2 Definice přímého derivačního kroku, sekvence derivačních kroků a jazyka definovaného lineární gramatikou

Tyto definice jsou shodné s příslušnými definicemi u bezkontextové gramatiky.

2.3.4.3 Příklad

Uvažujme následující bezkontextovou gramatiku:

$G = (N, T, P, S)$, kde:

$N = \{S, A\}$

$T = \{x, y\}$,

$P = \{S \rightarrow xS, S \rightarrow \varepsilon, S \rightarrow yA, A \rightarrow yS, A \rightarrow xA\}$

Tato bezkontextová gramatika generuje jazyk obsahující řetězce na abecedou $\{x, y\}$, které obsahují sudý počet „*y*“. Může například vygenerovat řetězec “*xyxyxx*“

2.3.4.4 Definice regulárního jazyka

Jazyk je regulární právě tehdy, když je generován nějakou pravou lineární gramatikou.

2.3.5 Věta o Chomského hierarchii jazyků

Označme symbolem L_{RE} třídu rekurzivně vyčíslitelných jazyků, symbolem L_{CS} třídu kontextových jazyků, symbolem L_{CF} třídu bezkontextových jazyků a symbolem L_{REG} třídu regulárních jazyků. Mezi těmito třídami jazyků platí vztah:

$$L_{REG} \subset L_{CF} \subset L_{CS} \subset L_{RE}$$

2.4 Další modely pro formální popisy jazyků

2.4.1 Konečný automat

Konečný automat (KA, též FSM z anglického *finite state machine*) je teoretický výpočetní model používaný v informatice pro studium vyčíslitelnosti a obecně formálních jazyků. Popisuje velice jednoduchý počítač, který může být v jednom z několika stavů, mezi kterými přechází na základě symbolů, které čte ze vstupu. Množina stavů je konečná (odtud název), konečný automat nemá žádnou další paměť kromě informace o aktuálním stavu. Konečný automat je velice jednoduchý výpočetní model, dokáže rozpoznávat pouze regulární jazyky. Konečné automaty se používají pro zpracování regulárních výrazů, např. jako součást lexikálního analyzátoru v překladačích .

2.4.1.1 Definice konečného automatu

Konečný automat M je pětice $M = (Q, T, R, q_0, F)$, kde:

Q je konečná množina stavů,

T je konečná vstupní abeceda,

R je konečná množina pravidel tvaru: $pa \rightarrow q$, kde $p, q \in Q, a \in T \cup \{\varepsilon\}$,

$q_0 \in Q$ je počáteční stav,

$F \subseteq Q$ je množina koncových stavů.

2.4.1.2 Definice konfigurace konečného automatu

Nechť $M = (Q, T, R, q_0, F)$ je konečný automat. Potom konfigurací χ konečného automatu M je řetězec: $\chi = qw$, kde $q \in Q, w \in T^*$.

2.4.1.3 Definice přechodu konečného automatu

Nechť $M = (Q, T, R, q_0, F)$ je konečný automat. Nechť nechť pax, qx jsou dvě konfigurace konečného automatu M, kde $p, q \in Q, x \in T^*, a \in T \cup \{\varepsilon\}$. Nechť $r = pa \rightarrow q \in R$ je pravidlo. Pak říkáme, že konečný automat M provede *přechod* z konfigurace pax do qx podle pravidla r a zapisujeme $pax \vdash qx\ [r]$ nebo také zkráceně $pax \vdash qx$.

2.4.1.4 Definice sekvence přechodů u konečného automatu

Nechť $M = (Q, T, R, q_0, F)$ je konečný automat.

- Nechť χ je konfigurace. *M* provede *nula přechodů* z χ do χ; zapisujeme $\chi \vdash^0 \chi$ [ε] nebo zjednodušeně $\chi \vdash^0 \chi$.
- Nechť $\chi_0, \chi_1, \ldots, \chi_n$ jsou konfigurace, nechť pro všechna $i = 1, ..., n$ platí: $\chi_{i-1} \vdash \chi_i$ $[r_i]$. Pak říkáme, že *M* provede *n* přechodů z χ_0 do χ_n. zapisujeme $\chi_0 \vdash^n \chi_n$ $[r_1 r_2 \ldots r_n]$ nebo také zkráceně $\chi_0 \vdash^n \chi_n$.
- Pokud $\chi_0 \vdash^n \chi_n$ pro nějaké $n \geq 1$, potom zapisujeme $\chi_0 \vdash^+ \chi_n$
- Pokud $\chi_0 \vdash^n \chi_n$ pro nějaké $n \geq 0$, potom zapisujeme $\chi_0 \vdash^* \chi_n$

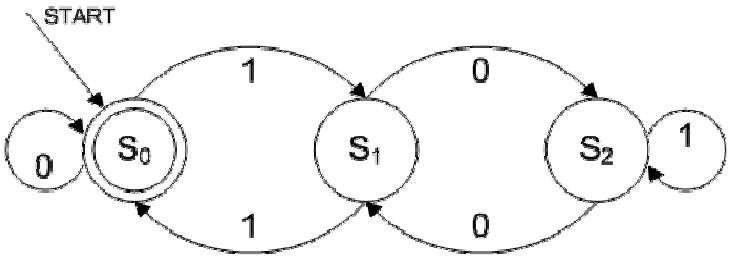
2.4.1.5 Definice jazyka přijímaným konečným automatem

Nechť $M = (Q, T, R, q_0, F)$ je konečný automat. Potom jazyk přijímaný konečným automatem *M*, *L*(*M*), je definován jako:

$$L(M) = \{w: w \in T^* \ q_0 w \vdash^* f \wedge f \in F\}$$

2.4.1.6 Znázornění konečného automatu

Místo relativně nepřehledného (zvláště pro větší automaty) popisu konečného automatu přímo pomocí definice se obvykle používá grafické znázornění, na kterém kolečka znázorňují jednotlivé stavy a šipky (s přidruženým vstupním symbolem) mezi těmito kolečky popisují jednotlivé přechody. Příklad znázornění:



Dvojité kolečko označuje přijímající stavy (v našem případě pouze jeden, S_0), počáteční stav je označen šipkou, někdy s připsaným textem, např. *START*. (Tato notace není jediná, jindy se např. koncové stavy označují tlustším orámováním a dvojité kolečko označuje počáteční stav apod.)

Pokud má daný automat zpracovat vstup 1011, bude to probíhat takto: Na počátku je automat ve stavu S_0. Na vstup přijde první symbol, jednička. Z grafu vyplývá, že na příchod jedničky ve stavu S_0 automat reaguje přechodem do stavu S_1. Dále přichází nula, ze stavu S_1 se příchodem nuly přechází do stavu S_2. Poté přichází jednička, ze stavu S_2 se příchodem jedničky přechází do stavu S_2 (tzn. zůstává se ve stejném stavu). Nakonec přichází další jednička, takže automat opět zůstává ve stavu S_2. Stav S_2 *nepatří* do množiny *A*, tudíž tento automat vstup 1011

nepřijal, řetězec 1011 *nepatří* do jazyka přijímaného tímto automatem. Pro úplnost: tento konečný automat přijímá regulární jazyk řetězců, které vyjadřují binární číslo dělitelné beze zbytku třemi. Číslo $1011_2 = 11_{10}$, číslo, které není dělitelné třemi (má zbytek 2, odpovídající výslednému stavu S_2).

2.4.1.7 Věta

Daný jazyk je regulární, právě tehdy když je přijímán nějakým konečným automatem.

2.4.2 Zásobníkový automat

Zásobníkový automat (*PDA* z anglického *pushdown automaton*) je teoretický výpočetní model používaný v informatice pro studium vyčíslitelnosti a obecně formálních jazyků. Popisuje jednoduchý počítač, který má jako pracovní paměť k dispozici pouze zásobník. Konečný automat dokáže rozpoznávat bezkontextové jazyky.

2.4.2.1 Definice zásobníkového automatu

Zásobníkový automat M je sedmice $M = (Q, T, \Gamma, R, q_0, Z_0, F)$, kde:

Q je konečná množina vnitřních stavů,

T je konečná vstupní abeceda,

Γ je konečná zásobníková abeceda,

R je konečná množina pravidel tvaru: $Apa \rightarrow xq$, kde $p, q \in Q, a \in T \cup \{\varepsilon\}, A \in \Gamma, x \in \Gamma^*$

$q_0 \in Q$ je počáteční stav,

$Z_0 \in \Gamma$ je startovací symbol zásobníku,

$F \subseteq Q$ je množina koncových stavů.

2.4.2.2 Definice konfigurace zásobníkového automatu

Nechť $M = (Q, T, \Gamma, R, q_0, Z_0, F)$ je zásobníkový automat. Potom konfigurací χ konečného automatu M je řetězec: $\chi = uqw$, kde $u \in \Gamma^*, q \in Q, w \in T^*$.

2.4.2.3 Definice přechodu zásobníkového automatu

Nechť $M = (Q, T, \Gamma, R, q_0, Z_0, F)$ je zásobníkový automat. Nechť $uApav$, $uxqv$ jsou dvě konfigurace konečného automatu M, kde $p, q \in Q, v \in T^*, a \in T \cup \{\varepsilon\}, A \in \Gamma, u, x \in \Gamma^*$. Nechť $r = Apa \rightarrow xq \in R$ je pravidlo. Pak říkáme, že zásobníkový automat M provede *přechod*

z konfigurace *uApav* do *uxqv* podle pravidla *r* a zapisujeme $uApav \vdash uxqv$ [*r*] nebo také zkráceně $uApav \vdash uxqv$.

2.4.2.4 Definice sekvence přechodů u zásobníkového automatu

Nechť $M = (Q, T, \Gamma, R, q_0, Z_0, F)$ je zásobníkový automat.

- Nechť χ je konfigurace. *M* provede *nula přechodů* z χ do χ; zapisujeme $\chi \vdash^0 \chi$ [ε] nebo zjednodušeně $\chi \vdash^0 \chi$.
- Nechť $\chi_0, \chi_1, \ldots, \chi_n$ jsou konfigurace, nechť pro všechna $i = 1, ..., n$ platí: $\chi_{i-1} \vdash \chi_i$ [r_i]. Pak říkáme, že *M* provede *n* přechodů z χ_0 do χ_n. zapisujeme $\chi_0 \vdash^n \chi_n$ [$r_1 r_2 \ldots r_n$] nebo také zkráceně $\chi_0 \vdash^n \chi_n$.
- Pokud $\chi_0 \vdash^n \chi_n$ pro nějaké $n \geq 1$, potom zapisujeme $\chi_0 \vdash^+ \chi_n$
- Pokud $\chi_0 \vdash^n \chi_n$ pro nějaké $n \geq 0$, potom zapisujeme $\chi_0 \vdash^* \chi_n$

2.4.2.5 Definice jazyka přijímaným zásobníkovým automatem

Nechť $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je zásobníkový automat.
Jazyk přijímaný zásobníkovým automatem M přechodem do koncového stavu, značen jako $L(M)$, je definován:

$$L(M) = \{w: w \in T^*, Z_0 q_0 w \vdash^* zf, z \in \Gamma^*, f \in F\}$$

2.5 Operace nad jazyky

2.5.1 Definice sjednocení dvou jazyků

Nechť L_1, L_2 jsou formální jazyky nad abecedou Σ. Sjednocením jazyků L_1 a L_2 rozumíme jazyk $L_1 \cup L_2$, který je definován následovně:

$$L_1 \cup L_2 = \{x: x \in L_1 \vee x \in L_2\}$$

2.5.2 Definice průniku dvou jazyků

Nechť L_1, L_2 jsou formální jazyky nad abecedou Σ. Průnikem jazyků L_1 a L_2 rozumíme jazyk $L_1 \cap L_2$, který je definován následovně:

$$L_1 \cap L_2 = \{x: x \in L_1 \wedge x \in L_2\}$$

2.5.3 Definice konkatenace dvou jazyků

Nechť L_1, L_2 jsou formální jazyky nad abecedou Σ. Konkatenací jazyků L_1, L_2 rozumíme jazyk $L_1.L_2$, který je definován následovně:

$$L_1.L_2 = \{xy: x \in L_1 \wedge y \in L_2\}$$

2.5.4 Definice rozdílu dvou jazyků

Nechť L_1, L_2 jsou formální jazyky nad abecedou Σ. Rozdílem jazyků L_1, L_2 rozumíme jazyk $L_1 – L_2$, který je definován následovně:

$$L_1 – L_2 = \{x: x \in L_1 \wedge x \notin L_2\}$$

2.5.5 Definice doplňku jazyka

Nechť L je formální jazyk nad abecedou Σ. Doplňkem jazyka L rozumíme jazyk $\overline{L}$, který je definován následovně:

$$\overline{L} = \Sigma^* – L$$

2.5.6 Definice iterace a pozitivní iterace jazyka

Nechť L je formální jazyk nad abecedou Σ. Iterací jazyka L rozumíme jazyk L^* a pozitivní iterací jazyka L rozumíme jazyk L^+, které jsou definovány následovně:

$L^0 = \{\varepsilon\}$

$L^n = L.L^{n-1}$

$L^* = \bigcup_{n \geq 0} L^n$

$L^+ = \bigcup_{n \geq 1} L^n$

2.6 Syntaktická analýza

Úkolem syntaktické analýzy je určit závislosti mezi jednotlivými řetězci a frázemi ve větě. Výstupem syntaktické analýzy je syntaktická struktura daného řetězce v podobě derivačního stromu.

K popisu struktury řetězců se používají formální gramatiky. Je nutné zvolit gramatiky, které mají dostatečnou vyjadřovací sílu pro zachycení složité syntaxe přirozeného jazyka, na druhé straně musí pro zvolenou třídu gramatik existovat efektivní algoritmus pro analýzu promluv. Nejčastěji se používají nějakým způsobem rozšířené bezkontextové gramatiky, zvané gramatiky s omezujícími podmínkami.

2.7 Lexikální analýza

Než si vysvětlíme princip lexikální analýzy, nadefinujeme si základní pojmy.

2.7.1 Lexém

Lexém je lexikální jednotka, např. identifikátor, operátor, číslo atd.

2.7.2 Token

Token je výstupem lexikální analýzy a vstupem syntaktické analýzy (zde se nazývá terminál). Token je ve tvaru:

```
struct Token
{
        Jmeno: string; // jméno rozpoznaného symbolu
        Atribut: atr;    // atribut symbolu
}
```

2.7.3 Princip lexikální analýzy

Lexikální analýza je činnost, kterou provádí lexikální analyzátor, tzv. scanner, ten je součástí překladače. Lexikální analyzátor rozdělí vstupní posloupnost znaků na lexémy. Tyto lexémy jsou poté reprezentovány ve formě tokenů a ty jsou dále poskytnuty ke zpracování syntaktické analýze. Lexikální analýza dále odstraňuje ze zdrojového souboru komentáře a bílé znaky. V praxi je lexikální analyzátor realizován pomocí konečného automatu.

Lexikální analyzátor je vstupní a nejjednodušší částí překladače. Čte ze vstupního souboru znaky reprezentující zdrojový program a z těchto znaků vytváří symboly programu, což je forma vhodná pro zpracování syntaktickou analýzou. Provádí tedy překlad posloupnosti znaků vstupního textu do posloupnosti symbolů na výstupu.

3. Popis metody

3.1 Derivace

3.1.1 Definice nejlevější derivace u bezkontextové gramatiky

Nechť $G = (N, T, P, S)$ je bezkontextová gramatika, $u \in T^*$ a $v \in (N \cup T)^*$ a $p = A \rightarrow x \in P$ je pravidlo. Pak říkáme, že *uAv přímo derivuje v nejlevější derivaci uxv* podle pravidla *p*, a zapisujeme $uAv \Rightarrow_{lm} uxv[p]$ nebo také zkráceně $uAv \Rightarrow_{lm} uxv$.

Během nejlevější derivace je přepsán nejlevější nonterminál.

3.1.2 Definice nejpravější derivace u bezkontextové gramatiky

Nechť $G = (N, T, P, S)$ je bezkontextová gramatika, $u \in (N \cup T)^*$ a $v \in T^*$ a $p = A \rightarrow x \in P$ je pravidlo. Pak říkáme, že *uAv přímo derivuje v nejpravější derivaci uxv* podle pravidla *p*, a zapisujeme $uAv \Rightarrow_{rm} uxv[p]$ nebo také zkráceně $uAv \Rightarrow_{rm} uxv$.

Během nejpravější derivace je přepsán nejpravější nonterminál.

3.2 Typy syntaktické analýzy

Syntaktická analýza znamená vytvoření derivačního stromu k danému řetězci. Derivační strom se vytváří za použití pravidel konkrétní gramatiky. Každá gramatika má startující nonterminál, od kterého se začíná vytvářet derivační strom.

Na základě syntaktické analýzy můžeme například určit zda daný řetězec patří do daného jazyka a zda je generovaný danou gramatikou. Známe dva druhy syntaktické analýzy:

- shora dolů
- zdola nahoru

3.2.1. Syntaktická analýza shora dolů

Syntaktická analýza shora dolů spočívá v postupném generování, které začíná od startovního nonterminálu dané gramatiky. A končí ve chvíli, kdy jsou všechny nonterminály přepsány na terminály, které tvoří řetězec. Při generování se používají pravidla dané gramatiky, která určují

přepis nonterminálů na terminály a další nonterminály. Po ukončení syntaktické analýzy nám vznikne derivační strom. Posloupnost pravidel použitá v syntaktické analýze shora dolů je identická s posloupností pravidel použitých v nejlevější derivaci.

3.2.2 Syntaktická analýza zdola nahoru

U syntaktické analýzy zdola nahoru se postupuje přesně naopak než je tomu u syntaktické analýzy shora dolů. Vytváření derivačního stromu začíná u řetězce terminálů a končí u startovního nonterminálu. Při generování se používají pravidla dané gramatiky, která určují přepis nonterminálů na terminály a další nonterminály. Pravidla jsou předem zadaná. Po ukončení syntaktické analýzy nám vznikne derivační strom. Posloupnost pravidel použitá v syntaktické analýze zdola nahoru je identická s reverzovanou posloupností pravidel použitých v nejpravější derivaci.

Problém této metody může nastat při nejednoznačnosti gramatiky nebo pokud existují v gramatice dvě nebo více pravidel se shodnou pravou stranou.

3.3. Multigenerování

Syntaktická analýza obecně funguje tak, že se vytváří jeden derivační strom pro jeden vstupní řetězec. Moje diplomová práce je o syntaktické analýze založené na multigenerování. Jednoduše se dá říci, že se jedná o generování více výstupů. Já jsem toto ještě upravila tak, aby vše probíhalo paralelně. Současně tak tvořím derivační strom vstupního řetězce a derivační strom výstupních řetězců.

Syntaktický analyzátor určí pravidlo dané gramatiky, které má být použít, a na základě tohoto pravidla se paralelně generují dané části derivačních stromů, které vedou k vytvoření výstupních řetězců.

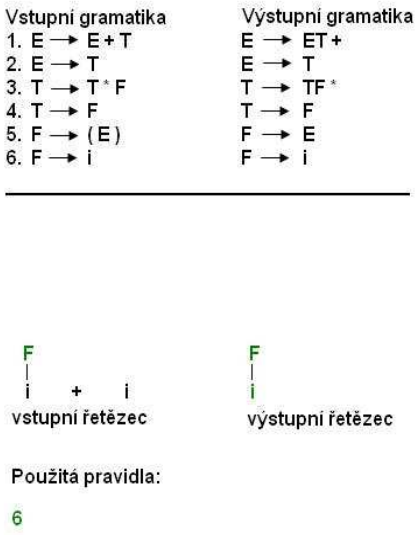
3.4 Překlad pomocí dvou gramatik

Abychom lépe pochopili princip multigenerování, ukážeme si příklad překladu pomocí dvou gramatik. Vstupní gramatika generuje zápis v infixové notaci, výstupní gramatika generuje zápis v postfixové notaci. Syntaktická analýza probíhá zdola nahoru a paralelně se vytváří derivační strom vstupního i výstupního řetězce.

3.4.1 Příklad překladu pomocí dvou gramatik

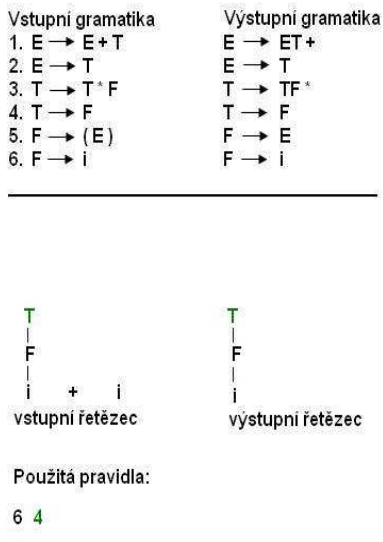
Na vstupu je dán řetězec v infixové notaci, který chceme převést na řetězec v postfixové notaci. Na začátku nadefinujeme pravidla obou gramatik, která nám určí, jak daný nonterminál přepsat na terminály a další nonterminály. Na základě syntaktické analýzy, která je použita na vstupní řetězec, se určí, které pravidlo se má použít. Toto pravidlo se poté paralelně použije na vytvoření dané části derivačního stromu u vstupního i výstupního řetězce.

Vzhledem k tomu, že se jedná o syntaktickou analýzu zdola nahoru, začíná se vytvářet derivační strom od řetězce terminálů. V prvním kroku generování se paralelně použije šesté pravidlo vstupní a výstupní gramatiky, které říká, že nonterminál *F* se přepíše na terminál *i.* Vytvoří se odpovídající část derivačního stromu (viz Obr 3.1).



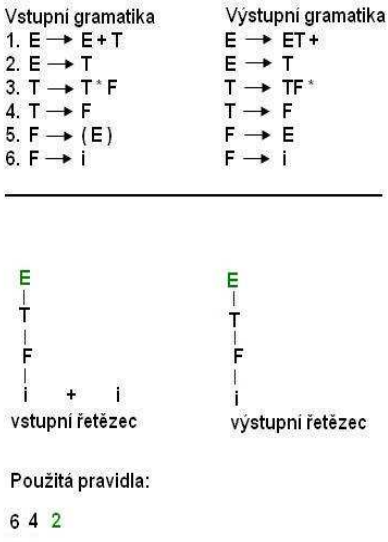
Obr 3.1

Ve druhém kroku syntaktický analyzátor určí pravidlo číslo čtyři, které nonterminál *T* přepíše na nonterminál *F* (viz Obr 3.2). Tímto se vytvořila další část derivačního stromu vstupního i výstupního řetězce.



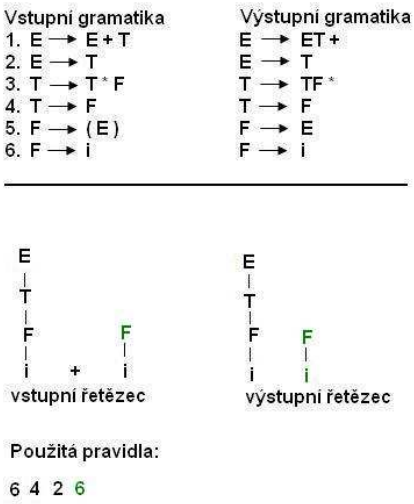
Obr 3.2

Dalším použitým pravidlem je pravidlo číslo dva, které je u obou gramatik stejné. Nonterminál *E* se přepíše na nonterminál *T* (viz Obr 3.3).



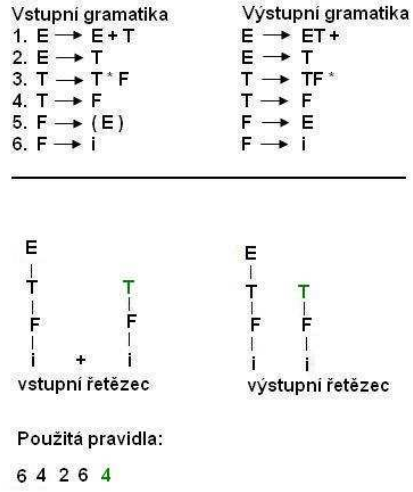
Obr -3.3

Čtvrtým použitým pravidlem je pravidlo číslo šest, které bylo už jednou použito. Toto pravidlo je opět u obou gramatik stejné. Nonterminál *F* se přepíše na terminál *i*. Tím se vytvoří další část výstupního řetězce (viz Obr 3.4).



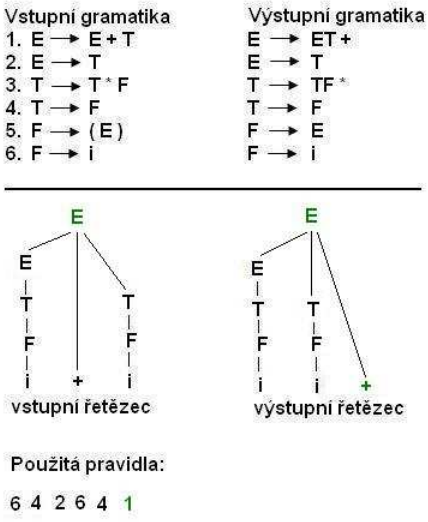
Obr 3.4

V pátém kroku se použije znovu pravidlo číslo čtyři a tím se přepíše nonterminál *T* na nonterminál *F* (viz Obr 3.5).



Obr 3.5

V posledním kroku se použije pravidlo číslo jedna, které je u každé gramatiky jiné. Pravidlo vstupní gramatiky přepíše nonterminál *E* na *E* + *T*. Pravidlo výstupní gramatiky přepíše nonterminál *E* na *ET*+. Tímto posledním krokem se vytvořil celý derivační strom a vstupní řetězec se přeložil na výstupní řetězec(viz Obr 3.6). Proces generování byl ukončen u startujícího nonterminálu.



Obr 3.6

3.5 Překlad pomocí n gramatik

Výše popsanou metodu překladu pomocí dvou gramatik jsem zobecnila na metodu překladu pomocí *n* gramatik, kde pomocí jedné (vstupní) gramatiky se provede syntaktická analýza pro vstupní řetězec, a pomocí zbývajících *n*-1 gramatik se paralelně vygeneruje *n*-1 výstupních řetězců. Každému pravidlu vstupní gramatiky jsou přiřazena odpovídající pravidla zbývajících *n*-1 výstupních gramatik. Tedy pravidlu číslo jedna vstupní gramatiky je přiřazeno pravidlo číslo jedna první gramatiky, pravidlo číslo jedna druhé gramatiky až pravidlo číslo jedna gramatiky s indexem *n*-1. Syntaktická analýza vstupního řetězce určí, které pravidlo se má použít a na základě toho se paralelně generuje *n*-1 výstupních řetězců.

Praktické využití této metody je například při paralelním generování výstupů v prefixové a postfixové notaci ze vstupu v infixové notaci. Nebo paralelní překlad do více jazyků současně.

3.6 Zavedení nových formálních modelů provádějících multipřeklad

3.6.1 Rozšířené překladové schéma

Rozšířené překladové schéma je obecně *n*-tice bezkontextových gramatik pracujících paralelním způsobem. Mezi pravidly jednotlivých gramatik existují vzájemné relace, které vytvářejí mezi

pravidly vazby. Podrobnější požadavky kladené na dané *n*-tice pravidel jsou popsány ve formální definici rozšířeného překladového schématu.

V každém derivačním kroku je vybrána právě jedna *n*-tice těchto pravidel, která je aplikována na vygenerovanou multiformu. Tímto způsobem je generován tzv. *n*-jazyk, tj. jazyk obsahující *n*-tice řetězců (ne jednoduché řetězce). Jednoduchý překlad je tedy speciální případ *n*-jazyka pro $n = 2$.

3.6.1.1 Definice rozšířeného překladového schématu

Rozšířené překladové schéma je n+3-tice

$\Gamma = (N, T_1, T_2, \ldots, T_n, P, S)$, kde:

- N je konečná množina nonterminálních symbolů,
- T_i je konečná množina terminálních symbolů pro všechna $i = 1, \ldots, n$,
- P je konečná množina pravidel tvaru $A \rightarrow x_1 \mid x_2 \mid \ldots \mid x_n$, kde $A \in N$, $x_i \in (N \cup T_i)^*$ pro všechna $i = 1, \ldots, n$, přičemž každý z řetězců $x_2, \ldots, x_n$ obsahuje právě ty nonterminální symboly, které jsou obsažené v řetězci x_1,
- $S \in N$ je počáteční nonterminální symbol.

Poznámka: Každé rozšířené překladové schéma $\Gamma = (N, T_1, T_2, \ldots, T_n, P, S)$ lze rozložit na n bezkontextových gramatik $G_1, G_2, \ldots, G_n$, přičemž pro všechna $i = 1, \ldots, n$ je i-tá gramatika tvaru $G_i = (N, T_i, P_i, S)$, kde $P_i = \{A \rightarrow x_i : A \rightarrow x_1 \mid x_2 \mid \ldots \mid x_i \mid \ldots \mid x_n \in P\}$.

3.6.1.2 Definice multiformy

Nechť $\Gamma = (N, T_1, T_2, \ldots, T_n, P, S)$ je rozšířené překladové schéma. Potom *multiforma* je n-tice $\chi = (x_1, x_2, \ldots, x_n)$, kde $x_i \in (T_i \cup N)^*$ pro všechna $i = 1, \ldots, n$.

3.6.1.3 Definice přímého derivačního kroku

Nechť $\Gamma = (N, T_1, T_2, \ldots, T_n, P, S)$ je rozšířené překladové schéma, nechť $\chi = (u_1Av_1, u_2Av_2, \ldots, u_nAv_n)$, $\overline{\chi} = (u_1x_1v_1, u_2x_2v_2, \ldots, u_nx_nv_n)$, jsou dvě multiformy, kde $A \in N$, $u_i, v_i, x_i \in (N \cup T_i)^*$ pro všechna $i = 1, \ldots, n$. Dále nechť $A \rightarrow x_1 \mid x_2 \mid \ldots \mid x_n \in P$. Pak říkáme, že χ *přímo derivuje* $\overline{\chi}$ a zapisujeme $\chi \Rightarrow \overline{\chi}$.

3.6.1.4 Definice přímého derivačního kroku v nejlevější derivaci

Nechť $\Gamma = (N, T_1, T_2, \ldots, T_n, P, S)$ je rozšířené překladové schéma, nechť $\chi = (u_1Av_1, u_2Av_2, \ldots, u_nAv_n)$, $\overline{\chi} = (u_1x_1v_1, u_2x_2v_2, \ldots, u_nx_nv_n)$, jsou dvě multiformy, kde $A \in N$, $u_i, v_i, x_i \in (N \cup T_i)^*$ pro všechna $i = 1, \ldots, n$, přičemž žádný z řetězců u_i neobsahuje symbol A. Dále nechť $A \rightarrow x_1 \mid x_2 \mid \ldots \mid x_n \in P$. Pak říkáme, že χ *přímo derivuje v nejlevější derivaci* $\overline{\chi}$ a zapisujeme $\chi \Rightarrow_{lm} \overline{\chi}$.

3.6.1.5 Definice přímého derivačního kroku v nejpravější derivaci

Nechť $\Gamma = (N, T_1, T_2, \ldots, T_n, P, S)$ je rozšířené překladové schéma, nechť $\chi = (u_1Av_1, u_2Av_2, \ldots, u_nAv_n)$, $\overline{\chi} = (u_1x_1v_1, u_2x_2v_2, \ldots, u_nx_nv_n)$, jsou dvě multiformy, kde $A \in N$, $u_i, v_i, x_i \in (N \cup T_i)^*$ pro všechna $i = 1, \ldots, n$, přičemž žádný z řetězců v_i neobsahuje symbol A. Dále nechť $A \rightarrow x_1 \mid x_2 \mid \ldots \mid x_n \in P$. Pak říkáme, že χ *přímo derivuje v nejpravější derivaci* $\overline{\chi}$ a zapisujeme $\chi \Rightarrow_{rm} \overline{\chi}$.

3.6.1.6 Definice sekvence derivačních kroků

Nechť $\Gamma = (N, T_1, T_2, \ldots, T_n, P, S)$ je rozšířené překladové schéma.

- Nechť χ je multiforma. Pak říkáme, že χ *derivuje v 0-krocích* χ a zapisujeme $\chi \Rightarrow^0 \chi$.
- Nechť $\chi_0, \chi_1, \ldots, \chi_k$ jsou multiformy, u kterých pro všechna $i = 1, ..., n$ platí: $\chi_{i-1} \Rightarrow \chi_i$. Pak říkáme, že χ_0 *derivuje v k-krocích* χ_n a zapisujeme $\chi_0 \Rightarrow^k \chi_n$.
- Nechť $\chi \Rightarrow^k \overline{\chi}$ pro nějaké $k \geq 1$, kde χ, $\overline{\chi}$ jsou multiformy. Pak říkáme, že χ *netriviálně derivuje* $\overline{\chi}$ a zapisujeme $\chi \Rightarrow^+ \overline{\chi}$.
- Nechť $\chi \Rightarrow^k \overline{\chi}$ pro nějaké $k \geq 0$, kde χ, $\overline{\chi}$ jsou multiformy. Pak říkáme, že χ *derivuje* $\overline{\chi}$ a zapisujeme $\chi \Rightarrow^* \overline{\chi}$.

Poznámka: Analogicky bychom mohli nadefinovat sekvence derivačních kroků v nejlevější resp. nejpravější derivaci.

3.6.1.7 Definice n-jazyka generovaného rozšířeným překladovým schématem

Nechť $\Gamma = (N, T_1, T_2, \ldots, T_n, P, S)$ je rozšířené překladové schéma. Potom *n-jazyk* generovaný Γ (budeme značit $n\text{-}L(\Gamma)$) je definován:

$n\text{-}L(\Gamma) = \{(w_1, w_2, \ldots, w_n): (S, S, \ldots, S) \Rightarrow^* (w_1, w_2, \ldots, w_n), w_i \in T_i^* \text{ pro všechna } i = 1, \ldots, n\}$

Poznámka: Analogicky bychom mohli nadefinovat *n-jazyk* generovaný Γ v nejlevější resp. nejpravější derivaci.

3.6.1.8 Příklad rozšířeného překladového schématu

$\Gamma = (N, T_1, T_2, T_3, P, S)$, kde:

- $N = \{S\}$,
- $T_1 = \{a, b\}$,
- $T_2 = \{c, d\}$,
- $T_3 = \{e, f\}$,
- $P = \{ S \rightarrow aSb \mid cSd \mid eSf, S \rightarrow ab \mid cd \mid ef \}$

je rozšířené překladové schéma.

V tomto rozšířeném překladovém schématu existují například následující sekvence derivačních kroků:

- $(S, S, S) \Rightarrow (ab, cd, ef)$
- $(S, S, S) \Rightarrow (aSb, cSd, eSf) \Rightarrow (aabb, ccdd, eeff)$
- $(S, S, S) \Rightarrow (aSb, cSd, eSf) \Rightarrow (aaSbb, ccSdd, eeSff) \Rightarrow (aaabbb, cccddd, eeefff)$
- …

Poznamenejme, že *n*-jazyk (konkrétně 3-jazyk) generovaný tímto překladovým schématem je $3\text{-}L(\Gamma) = \{(a^n b^n, c^n d^n, e^n f^n): n \geq 1\}$.

3.6.2 Deterministická syntaktická analýza pomocí rozšířeného překladového schématu

V dalších částech této diplomové práce se zaměříme na provádění syntaktické analýzy pomocí rozšířeného překladového schématu. Abychom mohli provádět syntaktickou analýzu deterministickým způsobem, omezíme se pouze na následující rozšířená překladová schémata, u nichž pro všechny jednotlivé gramatiky G_1, G_2, …, G_n platí, že jsou LR-gramatiky, tzn. že pro každou z těchto gramatik je možné provádět LR-syntaktickou analýzu. Jelikož LR-syntaktická analýza pracuje metodou zdola nahoru, výsledný *n*-jazyk bude generovaný pomocí nejpravějších derivací. Dále budeme vyžadovat, aby pravé strany všech pravidel neobsahovaly dva a více stejných nonterminálních symbolů. Toto bude povoleno jen ve speciálních případech, které budou popsány dále v této práci.

3.7 Princip metody

Princip metody spočívá v postupném vkládání a odebírání nonterminálů na a ze zásobníku v pořadí, které určí syntaktický analyzátor. Použití zásobníku umožní paralelní generování výstupních řetězců. Protože jsem svoji metodu použila pro překlad mezi třemi programovacími jazyky a to Pascalem, C a Modula-2, budu pro názorné vysvětlení používat zjednodušená gramatická pravidla popisující syntaxi těchto jazyků. Také zde uvedu jen zjednodušené struktury, které v programu používám, detailněji bude vše popsáno v následující kapitole.

Základním stavebním kamenem jsou objekty tříd, které obsahují jméno nonterminálu z levé strany pravidla a řetězec terminálů, který se na tyto nonterminály „navazuje“. Zjednodušená struktura je na obrázku Obr 3.7. Tyto objekty se vkládají na zásobník a dle potřeby se z něj vyjímají. Používám datový typ *string* a to z toho důvodu, aby bylo řešení obecné, tedy aby si mohl uživatel nadefinovat libovolně dlouhé názvy nonterminálů, a zároveň aby bylo možné

vygenerovat libovolně dlouhý řetězec terminálů. Díky použití datového typu *string*, respektive knihovny *string*, nemusím řešit alokování a uvolňování paměti.

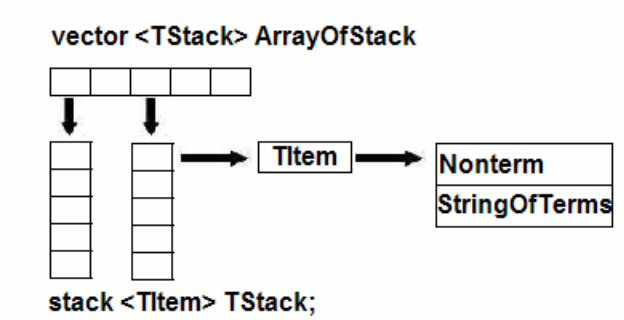
```
class TItem
{
        string Nonterm;
        string StringOfTerms;
}
```

Příklad:

Nonterm = def_identifikator
StringOfTerms = int i;

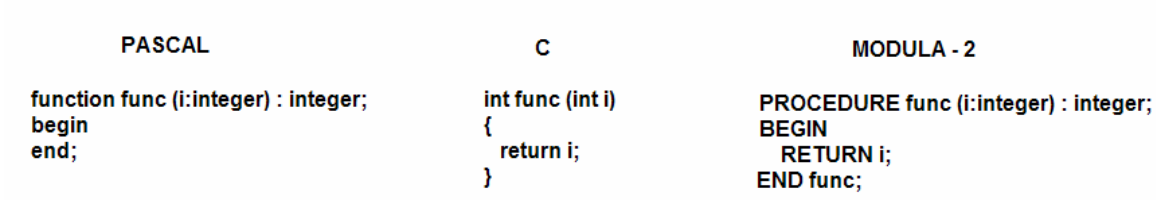
Obr 3.7

Další důležitou strukturou je vektor, použitý jako pole, zásobníků. Protože se jedná o multigenerování, musím mít pro každou gramatiku, a tudíž pro každý výstupní řetězec, vlastní zásobník. Pro lepší představu je toto ukázáno na obrázku Obr 3.8. Využila jsem knihoven *vector* a *stack*. Knihovna *vector* řeší problém vektorů, které jsou použity ve smyslu dynamického pole. Opět nemusím alokovat ani uvolňovat paměť. S vektorem pracuji prostřednictvím funkcí nadefinovaných v knihovně. Knihovna *stack* řeší práci se zásobníkem. Nemusím tedy definovat funkce pro práci se zásobníkem, opět využívám již nadefinovaných funkcí. Obě tyto knihovny mi umožnily se soustředit na podstatné části implementace a díky nim jsem nemusela řešit podpůrné funkce.



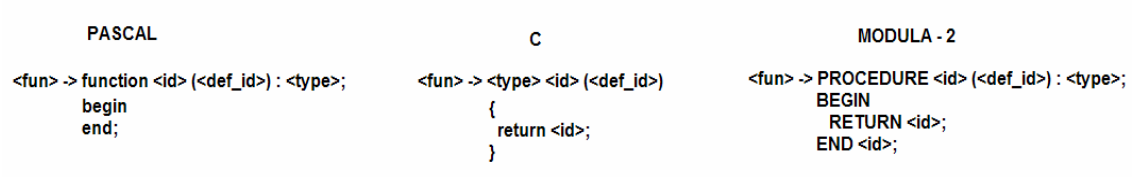
Obr 3.8

Nyní popíši samotný princip metody. Pro lepší pochopení bude vše ukázáno na příkladu. Uvažujme, že potřebujeme přeložit zdrojový kód zapsaný v jazyce Pascal do zdrojových kódů jazyků C a Modula-2. Protože překlad smysluplného kódu je zdlouhavý, vybrala jsem pouze jeden případ a to definici funkce. Na této ukázce bude dobře viditelný princip metody, protože všechny tři výše zmíněné jazyky mají jinou syntaxi zápisu definice funkce, což je vidět na obrázku Obr 3.9. Každý jazyk má jiné pořadí typů a identifikátorů, Pascal používá klíčové slovo *function*, Modula-2 zase klíčové slovo *procedure*. Zatímco C a Modula-2 vrací hodnotu přes *return*, Pascal nic takového nemá.



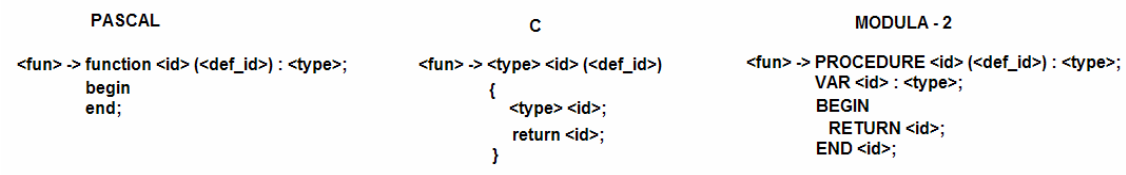
Obr 3.9

Syntaxi jednotlivých jazyků je potřeba přepsat pomocí gramatických pravidel. Zjednodušený zápis je na obrázku Obr 3.10.



Obr **3.10**

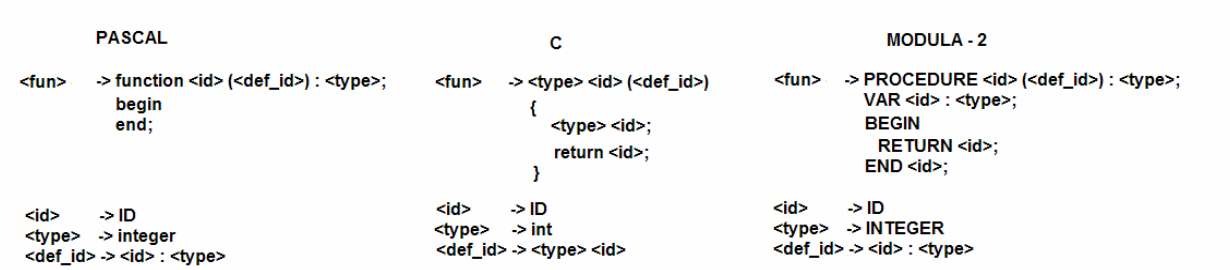
Na gramatických pravidlech je vidět, že různé jazyky mají různý počet nonterminálů na pravé straně pravidla. Zatímco Pascal má nonterminály na pravé straně pravidla pouze tři, jazyk C má nonterminály čtyři a jazyk Modula-2 dokonce pět. Jak již bylo na začátku zmíněno, budeme uvažovat překlad z jazyka Pascal do jazyků C a Modula-2. Úkolem je vytvořit z funkčního vstupního zdrojového kódu dva funkční, chápeme ve smyslu přeložitelné, výstupní zdrojové kódy. Při použití výše uvedených gramatických pravidel by ale funkčnost nebyla zajištěna a to hned z několika důvodů. Jazyk C musí výslednou hodnotu vracet prostřednictvím *return*. Otázkou ale zůstává, jak zjistíme, která hodnota se má vrátit, když jazyk Pascal nic takového nepoužívá. Musíme tedy pravidlo přepsat tak, aby vygenerovalo funkční kód. Výsledné řešení (viz Obr 3.11) sice není efektivní, ale efektivita není naším cílem – cílem je funkční výstupní kód.



Obr 3.11

Základním předpokladem je mít na levé straně odpovídajících si pravidel nonterminály se shodnými názvy. To stejné musí platit i o nonterminálech na pravých stranách pravidel. Pokud máme pravidlo u jazyka Pascal nazvané <*fun*>, musí mít odpovídající pravidla jazyků C a Modula-2 shodný název. Pokud se na pravé straně pravidla u jazyka Pascal vyskytuje nonterminál s názvem <*id*>, musí se nonterminál s tímto názvem vyskytovat i na pravých stranách jazyků C a Modula-2. A naopak, pokud se nonterminál s názvem např. <*id2*> na pravé straně pravidla jazyka Pascal nevyskytuje, nesmí se objevit ani na pravých stranách jazyků C a Modula-2.

Pro úplnost uvedu všechna zjednodušená pravidla, která při překladu použijeme (viz Obr 3.12). Ve skutečnosti jsou pravidla zapsána složitěji, aby postihla všechny možné případy, které mohou nastat.

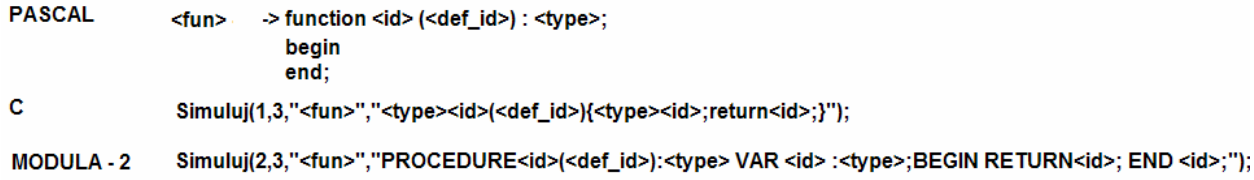


Obr 3.12

Na výše uvedených pravidlech je vyřešen problém výsledné funkčnosti výstupního kódu. Stále ale máme na pravých stranách pravidel rozdílný počet nonterminálů. Aby byl vygenerovaný kód skutečně funkční, tak se při psaní gramatik tomuto jevu nevyhneme. Dále toto tedy nebudeme brát jako problém, ale jako fakt, se kterým musíme počítat. Jediné omezení, které na pravidla gramatik klademe, je shoda názvů nonterminálů, jak bylo popsáno výše.

Hlavní výhoda metody tedy je, že řeší problém rozdílného počtu nonterminálů na pravých stranách pravidel. Zároveň můžeme vytvářet klony nonterminálů (jeden nonterminál z jednoho pravidla přepíšeme na *n* nonterminálů se shodným názvem druhého pravidla) nebo naopak více nonterminálů se shodným názvem můžeme přepsat na jeden nonterminál.

V prvním kroku překladu se pravidla gramatik načtou ze vstupního souboru a podle potřeby jsou upravena (detailněji bude popsáno ve čtvrté a šesté kapitole). K modifikaci pravidel patří i určení počtu nonterminálů na pravé straně vstupního pravidla. Výsledná hodnota se použije jako jeden z argumentů funkce *Simuluj()* (detailněji bude tato funkce popsána ve čtvrté kapitole). V našem případě bude modifikace vypadat následovně (viz obrázek Obr 3.13) :



Obr 3.13

Prvním argumentem funkce *Simuluj()* je pořadí gramatiky. Druhým argumentem je počet nonterminálů na pravé straně vstupního pravidla (čili pravidla popisujícího syntaxi jazyka Pascal). Pravidlo jazyka Pascal má na pravé straně tři nonterminály, proto má druhý argument hodnotu tři. Třetím argumentem je levá strana pravidla, respektive nonterminál. Čtvrtým argumentem je pravá strana pravidla, respektive posloupnost terminálů a nonterminálů.

V druhém kroku překladu určí syntaktický analyzátor pravidlo, které se má použít a volá se funkce *Simuluj()*. Tato funkce nejprve zpracuje nonterminál z levé strany pravidla, tedy třetí

argument funkce. Poté prochází pravou strana znak po znaku a zpracuje ji následujícím způsobem:

- samostatné znaky jako například středník, závorka, hvězdička atd. se vloží do pomocného řetězce,
- víceznakové terminály, např. klíčová slova, se nejprve celé načtou a poté vloží do pomocného řetězce,
- mezera se vloží do pomocného řetězce,
- nonterminál se celý načte, do pomocného řetězce se vloží „zarážka“, která označuje, kam je potřeba obsah nonterminálu po jeho zpracování vložit. Nonterminál se vloží na konec vektoru nonterminálů.

Po zpracování pravé strany pravidla je k dispozici vektor nonterminálů, které se vyskytovaly na pravé straně pravidla, a pomocný řetězec terminálů se „zarážkami“, které označují pozici nonterminálů. V případě, že na pravé straně pravidla nebyly žádné nonterminály, vytvoří se nový prvek obsahující název nonterminálu a řetěz terminálů a vloží se na vrchol zásobníku. Pokud pravá strana pravidla obsahuje nonterminály, pokračuje se v jejím zpracování.

Důležitou roli hraje nyní druhý argument funkce *Simuluj()* jehož hodnota udává počet nonterminálů z pravé strany pravidla vstupní gramatiky. Přesně tolik nonterminálů se postupně odebere z vrcholu zásobníku. Každý nonterminál z vrcholu zásobníku je porovnán s nonterminály uloženými ve vektoru nonterminálů. V případě shody jmen se k nonterminálu ve vektoru připojí řetězec terminálů navázaných na nonterminál z vrcholu zásobníku. V případě, že je ve vektoru několik nonterminálů se shodným názvem, připojí se řetězec terminálů na všechny tyto nonterminály – jde o klonování. Může nastat i případ, kdy je ze zásobníku odebráno více nonterminálů se shodným názvem a přitom ve vektoru nonterminálů je pouze jeden nonterminál s tímto názvem. Potom se tedy obsah všech nonterminálů ze zásobníku vloží do tohoto jediného nonterminálu ve vektoru nonterminálů. Musí ale platit, že obsah všech stejně pojmenovaných nonterminálů je totožný.

Po skončení této fáze je potřeba všechny nonterminály z vektoru vzít a jejich obsah (řetězec terminálů) vložit do dříve vytvořeného pomocného řetězce na místo určené zarážkou.

Nakonec je vytvořen nový prvek obsahující jméno nonterminálu z levé strany pravidla a řetězec terminálů uložený v pomocném řetězci a vložen na zásobník. Protože postup není triviální, ukážeme si jej opět na příkladě.

Předpokládáme, že již byla provedena modifikace pravidel a nyní nám syntaktická analýza určuje pravidla, která mají být použita. Postupně bude volána funkce *Simuluj()* s parametry uvedenými na obrázku Obr 3.14. Pro přehlednost jsou pravidla očíslovaná.

PASCAL

```
1  <fun>     -> function <id> (<def_id>) : <type>;
                begin
                end;

2  <id>      -> ID
3  <type>    -> integer
4  <def_id>  -> <id> : <type>
```

C

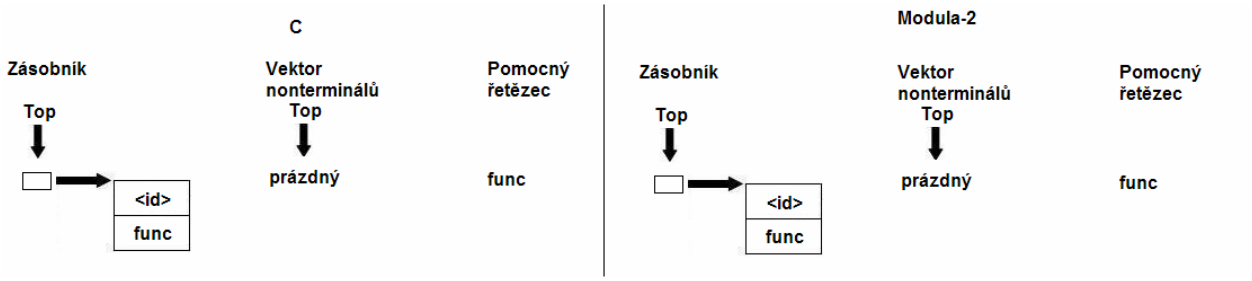
```
1  Simuluj(1,3,"<fun>","<type><id>(<def_id>){<type><id>;return<id>;}");

2  Simuluj(1,0,"<id>","ID");
3  Simuluj(1,0,"<type>","int");
4  Simuluj(1,2,"<def_id>","<type> <id>");
```

Modula-2

```
1  Simuluj(2,3,"<fun>","PROCEDURE<id>(<def_id>):<type> ;VAR<id>:<type>;
                        BEGIN RETURN<id>;END<id>;");
2  Simuluj(2,0,"<id>","ID");
3  Simuluj(2,0,"<type>","INTEGER");
4  Simuluj(2,2,"<def_id>","<id>:<type>");
```

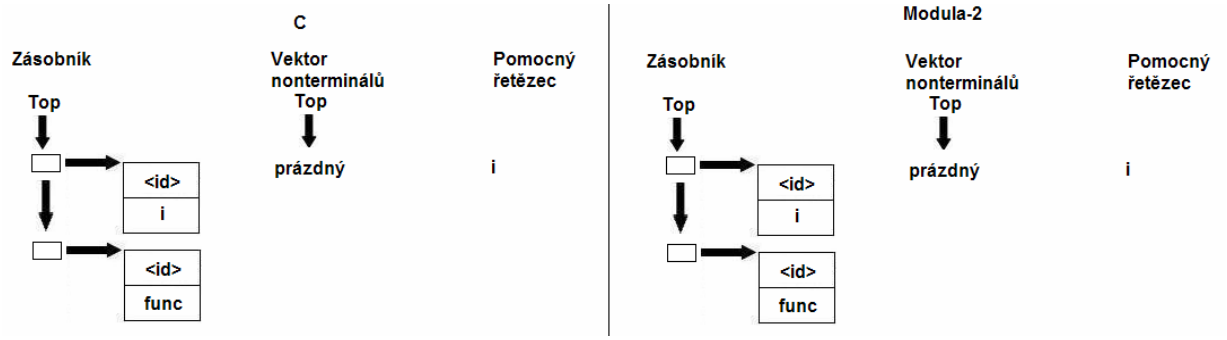
Obr 3.14

Pro názornost ukážeme stav na zásobníku, vektor nonterminálů a obsah pomocného řetězce. Uvažujme, že název funkce je *func*, proměnná je typu *integer* a její název je *i*. Nejprve se použije pravidlo číslo dva. Druhý argument funkce *Simuluj()* udává, že odpovídající pravidlo vstupní gramatiky (tedy pravidlo popisující syntaxi jazyka Pascal) neobsahuje na pravé straně žádný nonterminál. Vytvoří se nový prvek s odpovídajícím názvem nonterminálu a řetězcem terminálů a vloží se na vrchol zásobníku. Stav na zásobníku po aplikaci tohoto pravidla ukazuje obrázek Obr 3.15.



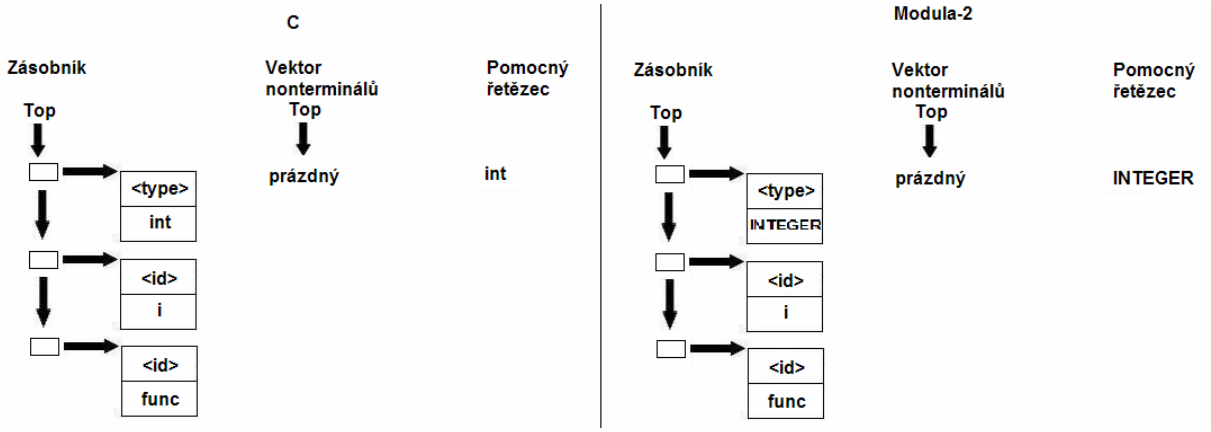
Obr 3.15

Pokračujeme aplikací pravidla číslo dva. Opět dojde pouze k vložení na zásobník, protože pravá strana vstupního pravidla neobsahuje žádný nonterminál. Stav na zásobníku po aplikaci tohoto pravidla je na obrázku Obr 3.16.



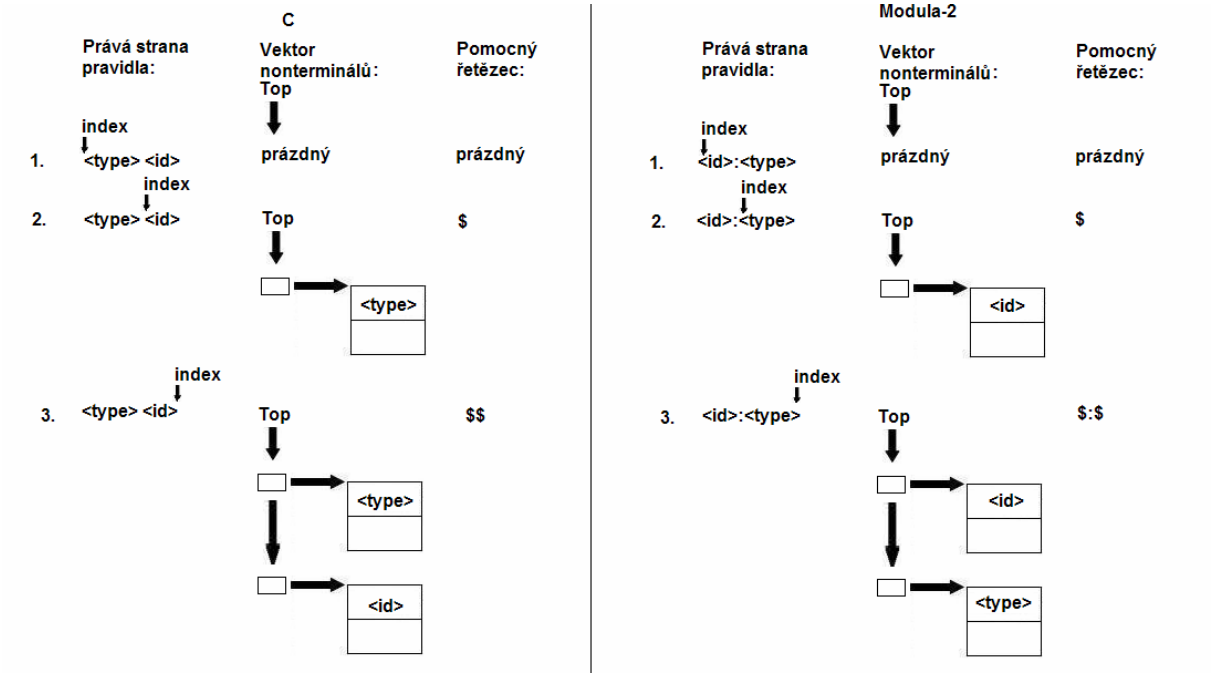
Obr 3.16

V dalším kroku syntaktický analyzátor určil pravidlo číslo tři. Opět je to pravidlo bez nonterminálu na pravé straně a proto opět pouze vložíme na zásobník nonterminál a odpovídající řetězec terminálů. Stav na zásobníku viz Obr 3.17.



Obr 3.17

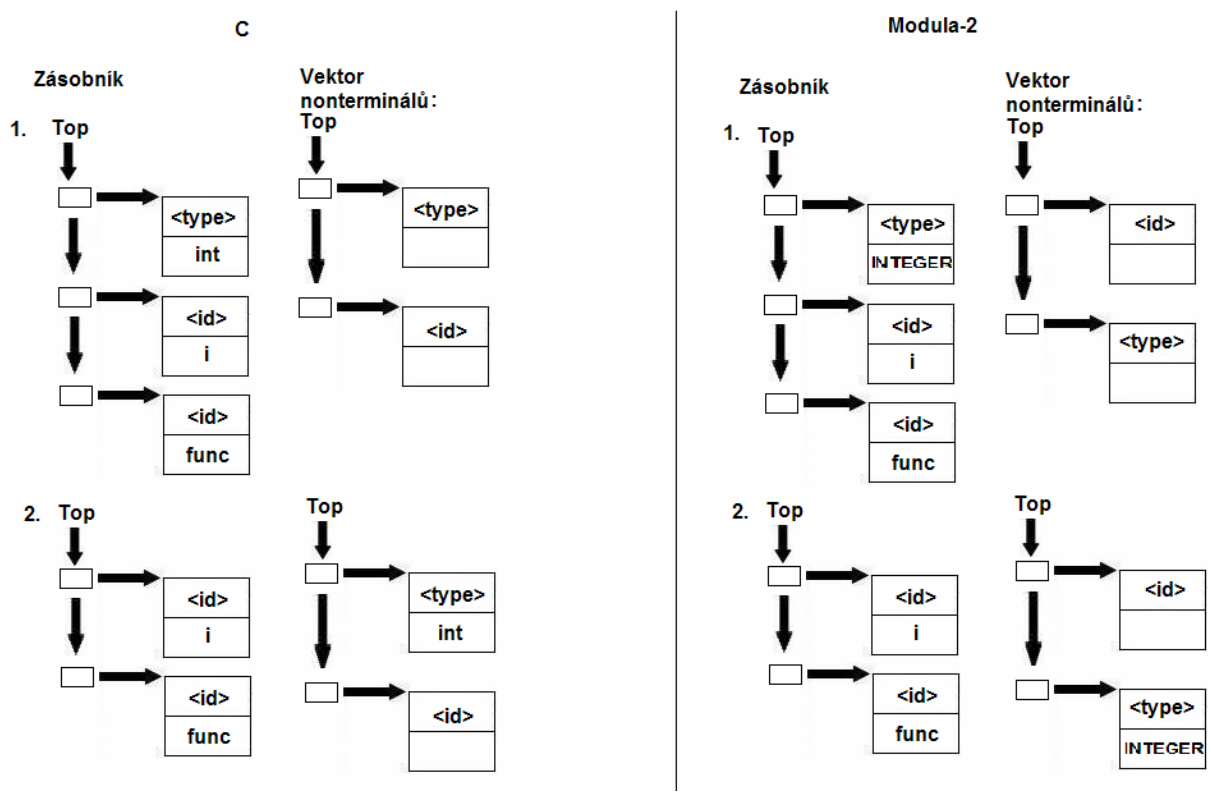
Nyní se poprvé použije redukční pravidlo a to pravidlo číslo čtyři. Postup si ukážeme krok po kroku. Nejprve je zpracována pravá strana pravidla jak ukazuje obrázek Obr 3.18. Pravá strana pravidla se prochází znak po znaku, terminály jsou přidávány do pomocného řetězce, nonterminály vkládány do vektoru nonterminálů a na jejich místo v pomocném řetězci se vloží zarážka.



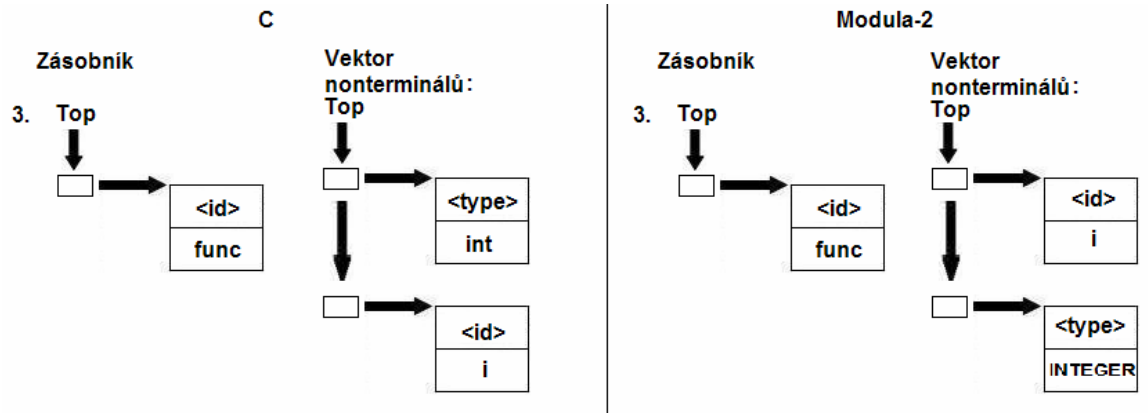
Obr 3.18

Znak *$* v pomocném řetězci označuje kam bude umístěn obsah nonterminálu. Po zpracování pravé strany pravidla můžeme začít pracovat se zásobníkem. Postupně se z vrcholu zásobníku vyjme tolik nonterminálů, kolik udává druhý argument funkce *Simuluj()* (hodnota

argumentu byla získána pomocí metody *CountNonterms*, více bude metoda popsána ve čtvrté kapitole). V našem případě se jedná o dva nonterminály. Každý nonterminál z vrcholu zásobníku je porovnán s nonterminálem z vektoru nonterminálů. Při shodě názvů je obsah (řetězec terminálů) nonterminálu ze zásobníku vložen do nonterminálu ve vektoru nonterminálů. Postup je vidět na obrázcích Obr 3.19a až Obr 3.19b.



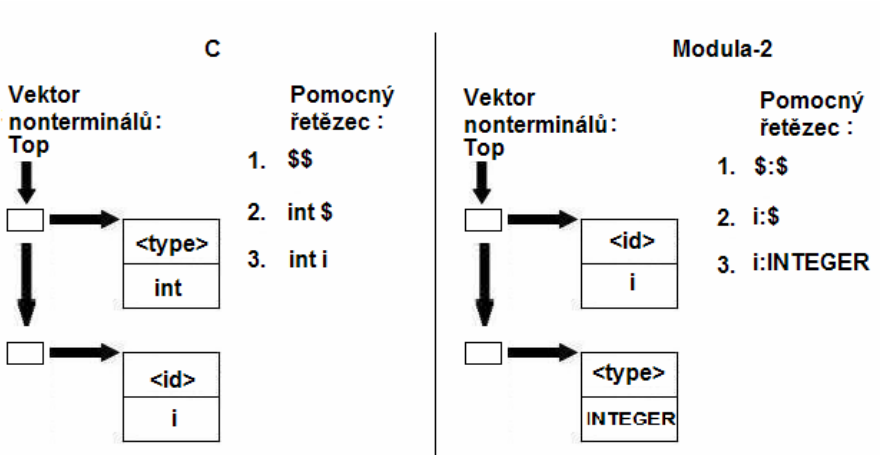
Obr 3.19a



Obr 3.19b

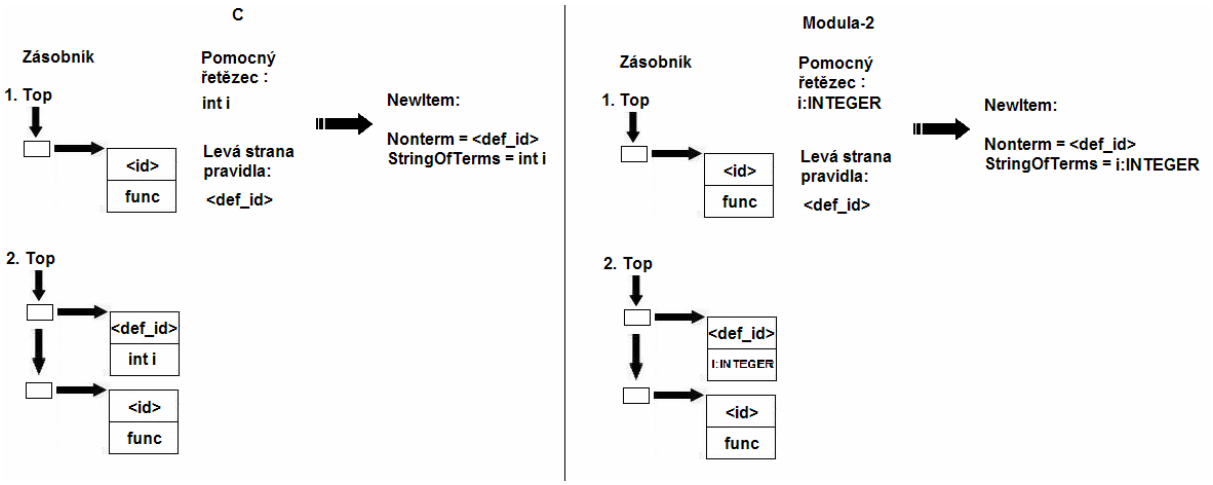
Dále je potřeba řetězce terminálů, které jsou „navázané“ na nonterminály ve vektoru vložit do pomocného řetězce (viz Obr 3.20). Pomocný řetězec se prochází od začátku znak po znaku a hledá se zarážka. Na místo zarážky se vloží obsah nonterminálu z vektoru nonterminálů.

Vektor nonterminálů se prochází od začátku, jednotlivé nonterminály jsou čteny v pořadí v jakém byly do vektoru vloženy, jedná se v podstatě o frontu.



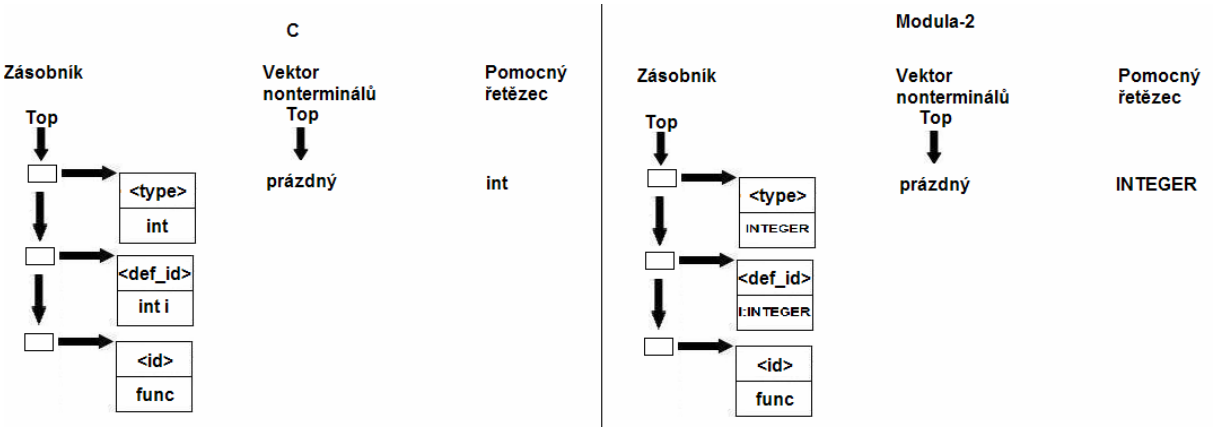
Obr 3. 20

Nakonec se vytvoří nový objekt s nonterminálem z levé strany pravidla a řetězcem terminálů uloženým v pomocném řetězci. Takto vytvořený objekt je vložen na vrchol zásobníku (viz Obr 3.21).



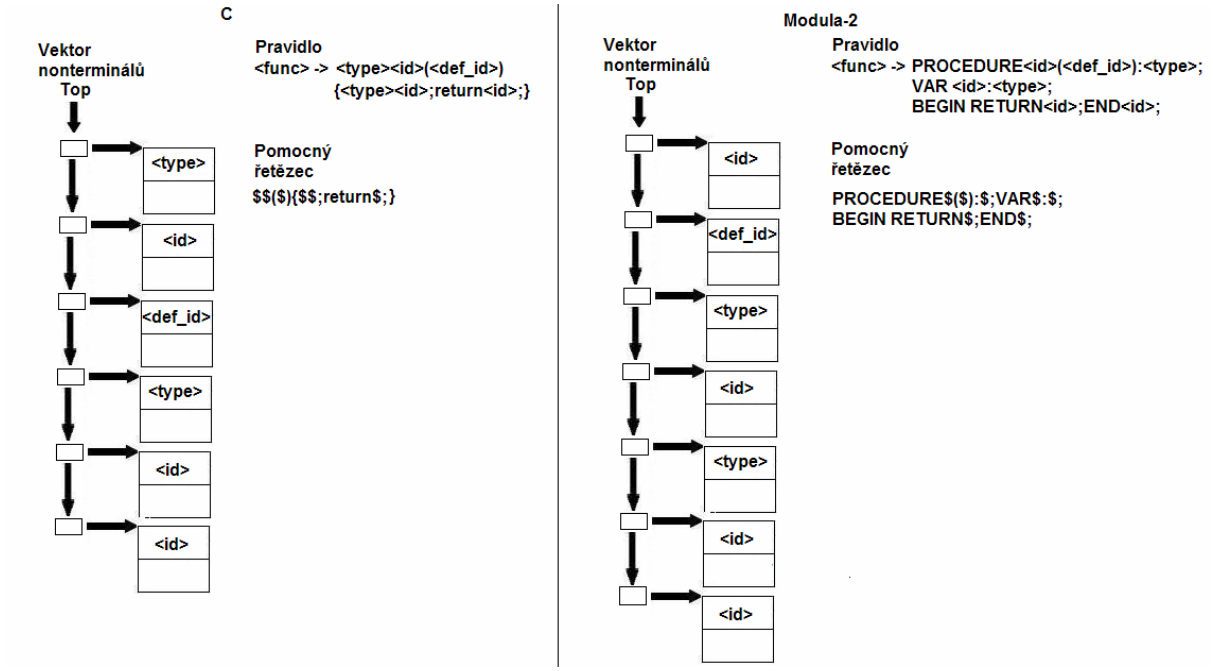
Obr 3.21

Pokračujeme výběrem a aplikací pravidla číslo tři. Na pravé straně není žádný nonterminál, proto se jen vloží na zásobník (viz Obr 3.22).

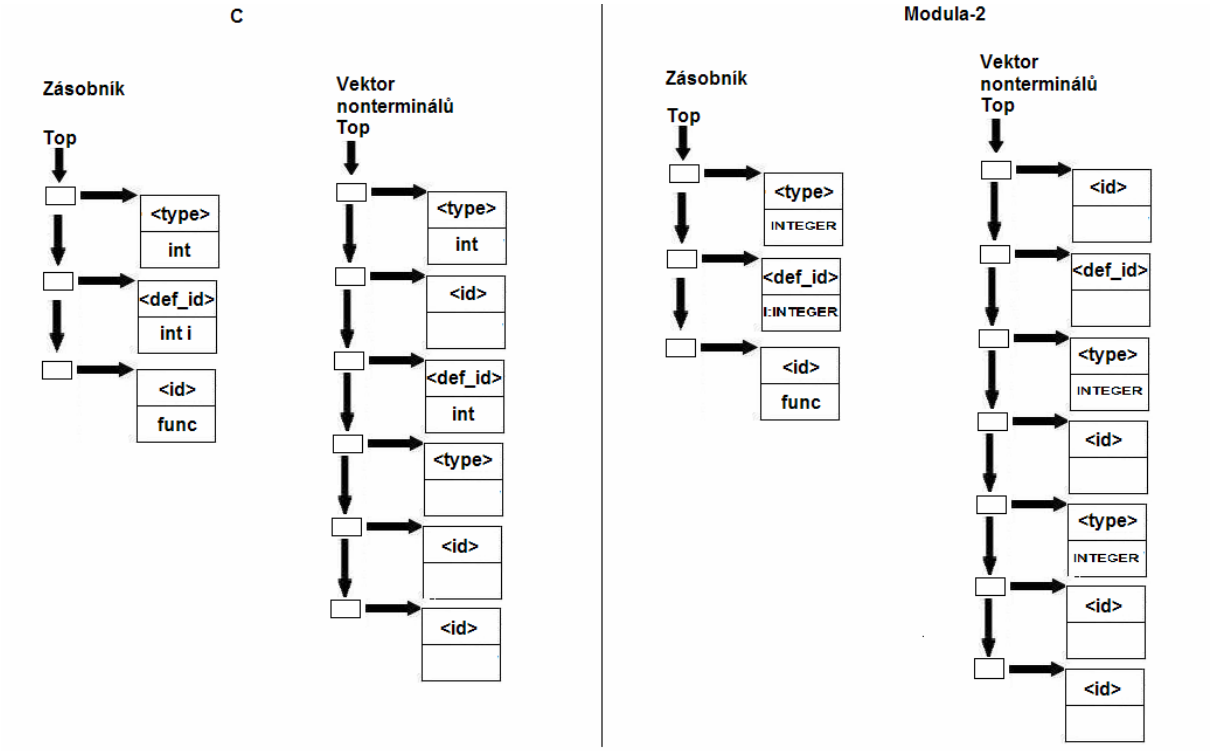


Obr 3. 22

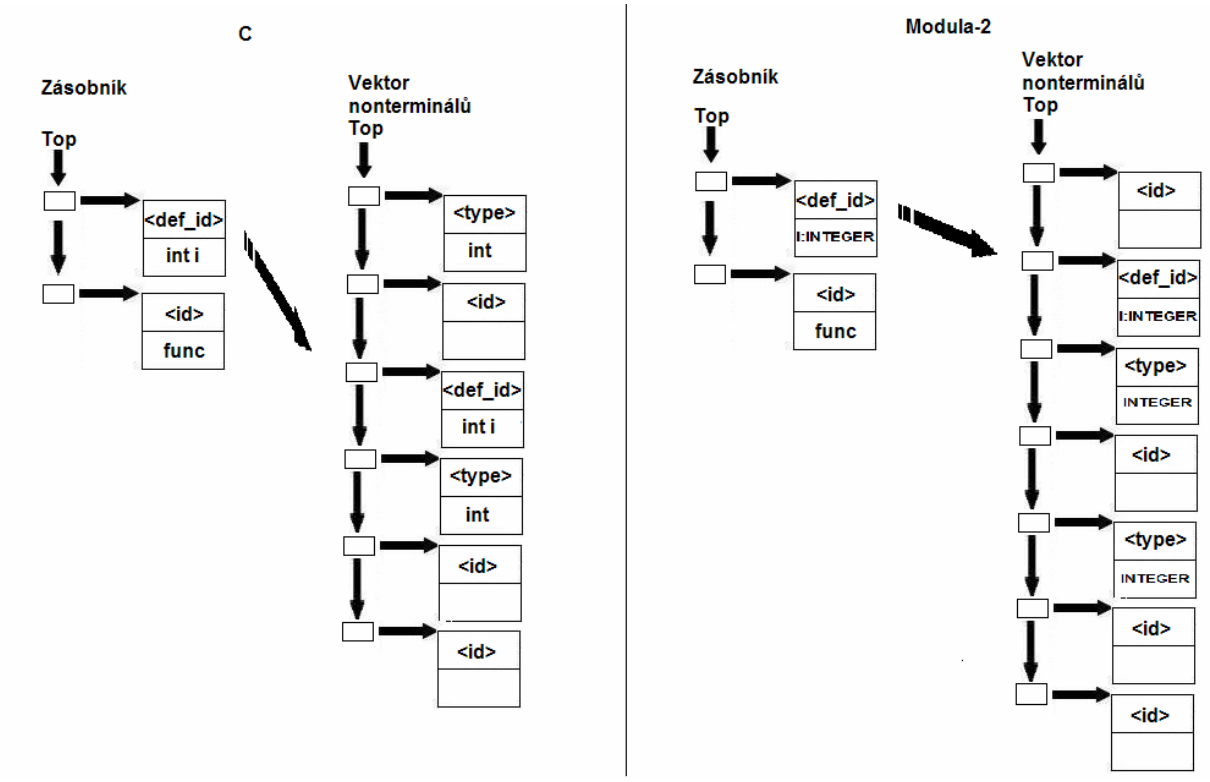
K dokončení překladu se použije pravidlo číslo jedna, které je opět redukční. Nejprve se projde pravá strana pravidla, vytvoří se vektor nonterminálů a pomocný řetězec se „zarážkami“. Poté se z vrcholu zásobníku odebere tolik nonterminálů, kolik určuje druhý parametr funkce *Simuluj()*. Tyto nonterminály se porovnají s nonterminály ve vektoru nonterminálů a pokud souhlasí jejich názvy, naváže se řetězec terminálů patřící nonterminálu z vrcholu zásobníku na nonterminál z vektoru nonterminálů. Dále se projde pomocný řetězec a na místo zarážek se vloží obsah nonterminálů z vektoru nonterminálů. Na závěr se vytvoří nový objekt osahující levou stranu pravidla (název nonterminálu) a řetězec terminálů (obsah pomocného řetězece). Celý postup je zkráceně vidět na obrázcích Obr 3.23a až Obr 3.23f.



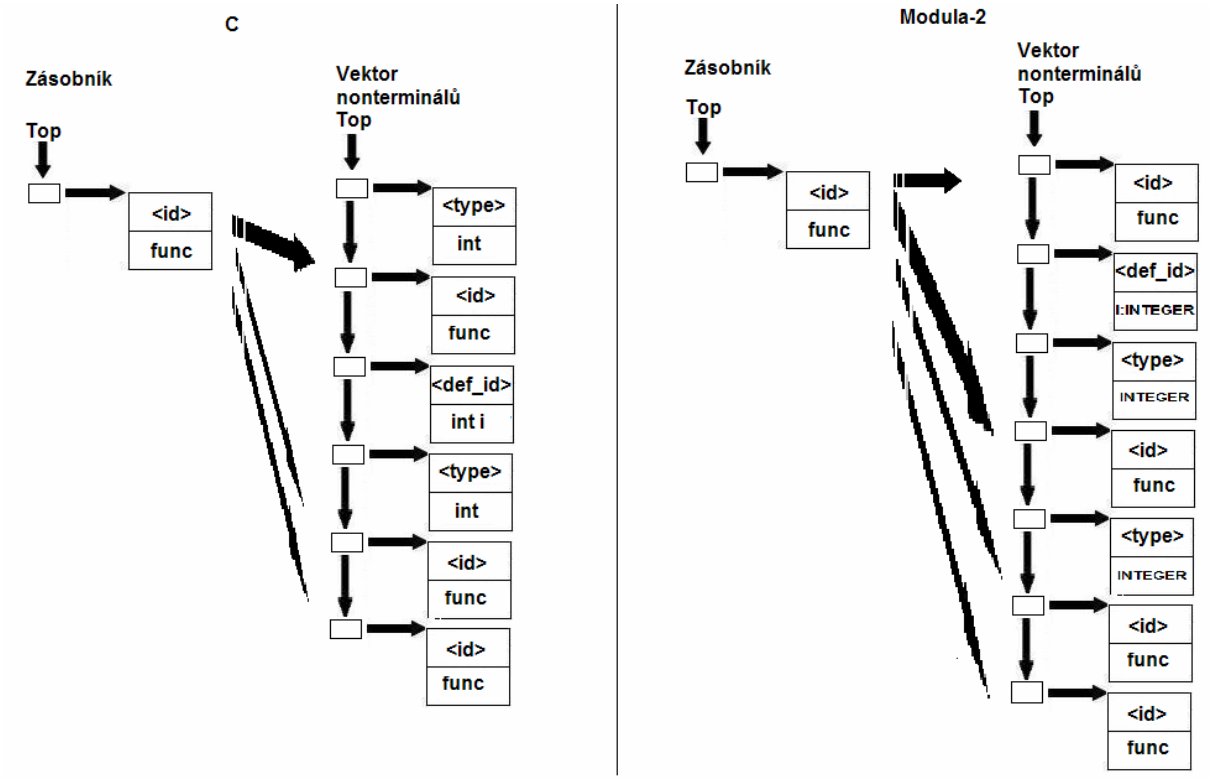
Obr 3.23a – vytvoření pomocného řetězce se „zarážkami“ a vektoru nonterminálů



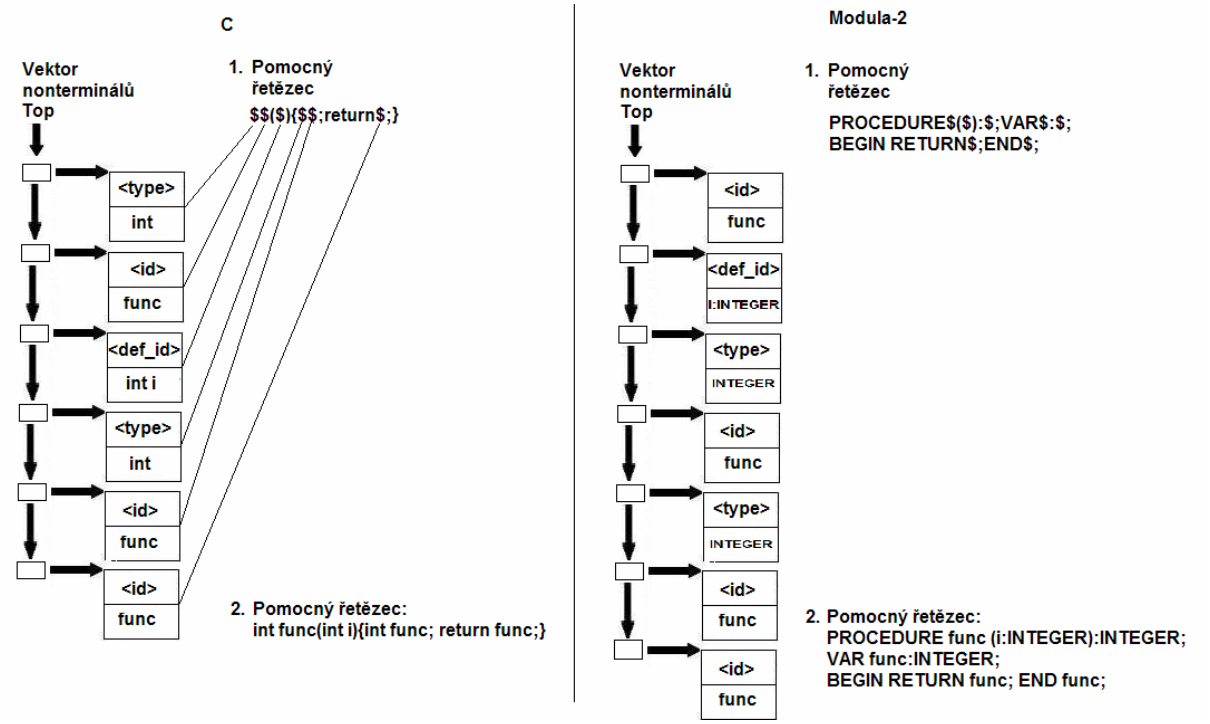
Obr 3.23b – přiřazení řetězce terminálů náležícího nonterminálu <type> všem nonterminálům z vektoru nonterminálů se shodným názvem



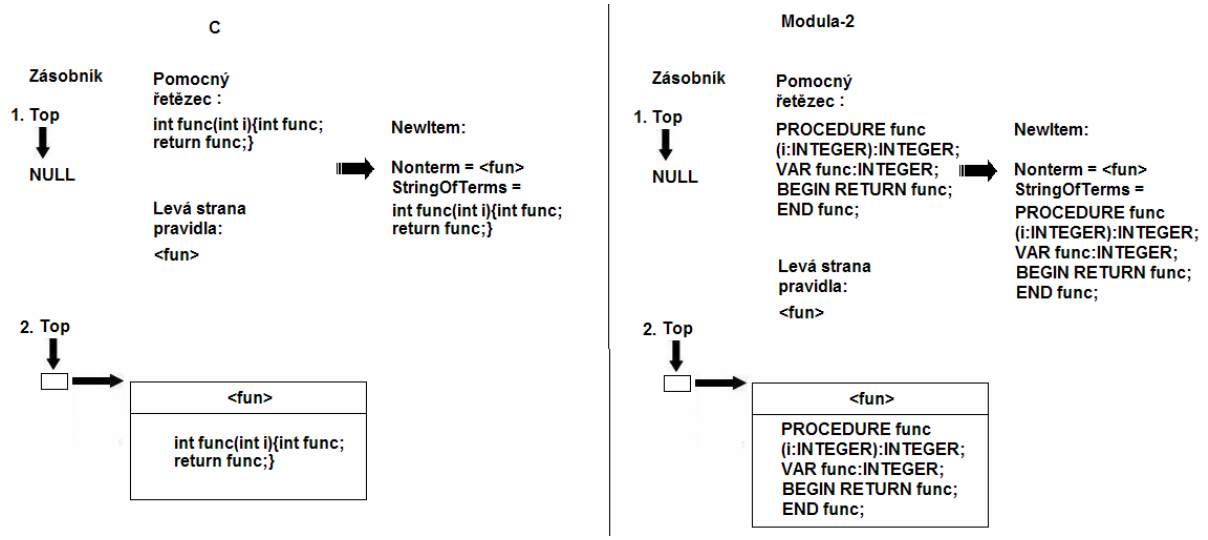
Obr 3.23c – přiřazení řetězce terminálů náležícího nonterminálu <def_id> všem nonterminálům z vektoru nonterminálů se shodným názvem



Obr 3. 23d– přiřazení řetězce terminálů náležícího nonterminálu <id> všem nonterminálům z vektoru nonterminálů se shodným názvem



Obr 3.23e – do pomocného řetězce se na místo určené zarážkou vloží obsah nonterminálů z vektoru nonterminálů. Nonterminály se berou z vektoru v pořadí, v jakém byly do vektoru vkládány.



Obr 3.23f – vytvoří se nový prvek, který se vloží na vrchol zásobníku

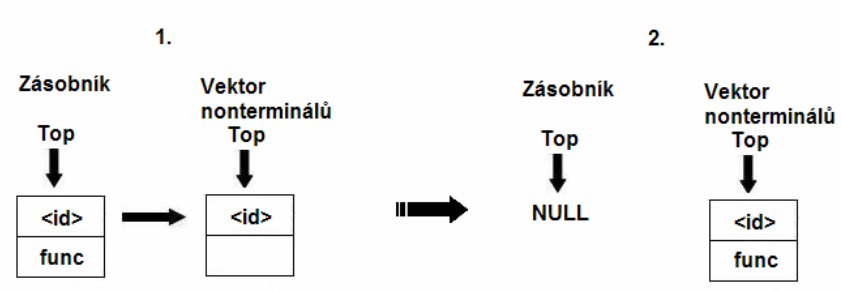
3.7.1 Redukce a klonování nonterminálů

Při přepisu obsahu nonterminálů ze zásobníku na obsah nonterminálů ve vektoru nonterminálů může dojít k redukci nebo naopak klonování těchto zásobníkových nonterminálů.

Redukcí nonterminálů rozumíme přepis *n* zásobníkových nonterminálů se shodným názvem na *m* vektorových nonterminálů se shodným názvem, kde $n > m$. Názvy nonterminálů na zásobníku jsou shodné s názvy nonterminálů ve vektoru.

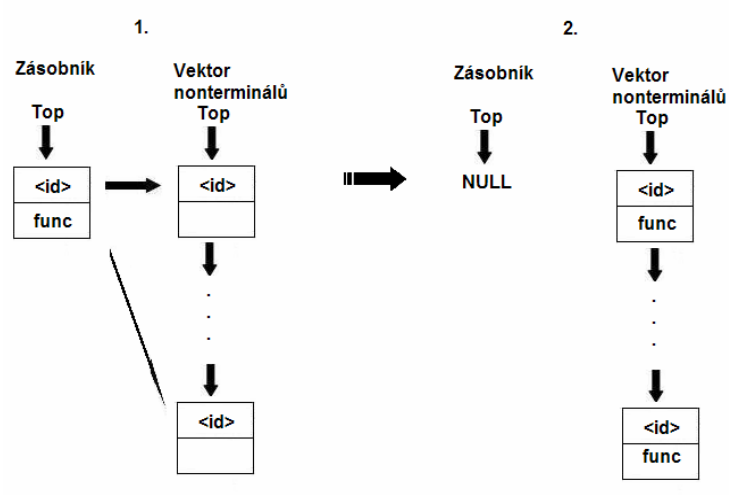
Klonováním nonterminálů rozumíme přepis *n* zásobníkových nonterminálů se shodným názvem na *m* vektorových nonterminálů se shodným názvem, kde $n < m$. Názvy nonterminálů na zásobníku jsou shodné s názvy nonterminálů ve vektoru.

Nejjednodušší případ přepisu *n* zásobníkových nonterminálů na *m* vektorových nonterminálů nastává, když $n=m=1$. V tomto případě nedochází ke klonování ani k redukci nonterminálů. Jednoduchý přepis je na obrázku Obr 3.24.



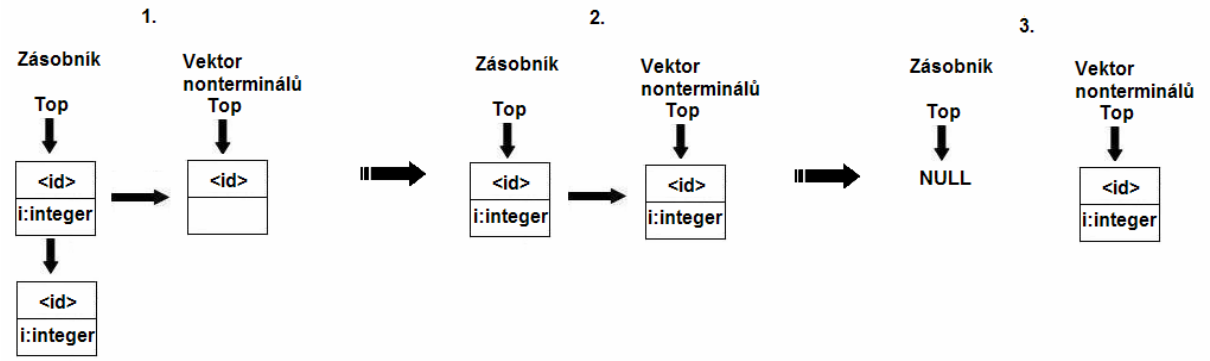
Obr 3.24

Druhou variantou je klonování. Pro jednoduchost si ukážeme klonování *n* zásobníkových nonterminálů na *m* vektorových nonterminálů, kde *n=1* a *m>n*. Postup při klonování je vidět na obrázku Obr 3.25.



Obr 3.25

Poslední variantou je redukce *n* zásobníkových nonterminálů na *m* vektorových nonterminálů, kde *m=1* a *n > m*. Předpokladem pro redukci je shodný obsah nonterminálů se shodným názvem. Máme-li dva nonterminály se shodným názvem <*id*>, pak i jejich obsah (řetězec terminálů) musí být shodný. Tedy pokud první nonterminál s názvem <*id*> obsahuje řetězec terminálů *i:integer*, pak i druhý nonterminál s názvem <*id*> musí obsahovat řetězec terminálů *i:integer*. Oba tyto zásobníkové nonterminály se přepíší na vektorový nonterminál se shodným názvem, čili <*id*>. Vektorový nonterminál obsahuje po přepisu řetězec terminálů *i:integer* (viz Obr 3.26).



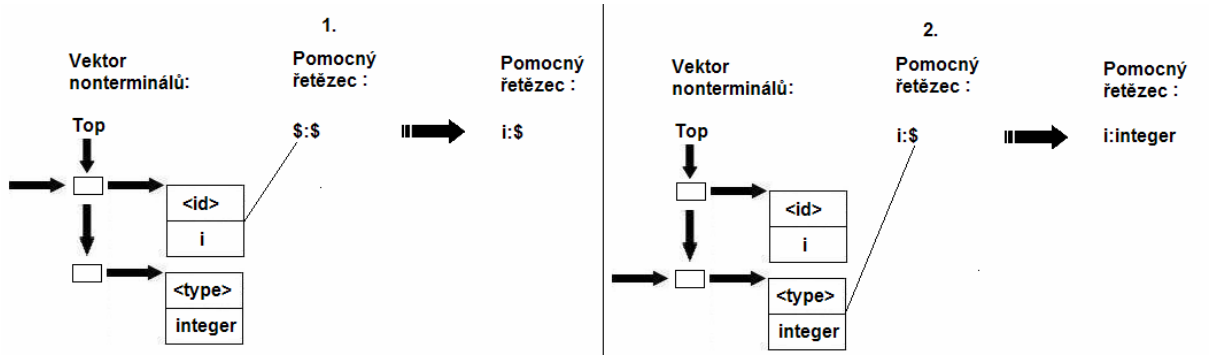
Obr 3.26

3.7.2 Vektor nonterminálů

Při zpracování pravé strany pravidla se všechny nalezené nonterminály vkládají do vektoru nonterminálů, respektive do vektoru, který obsahuje struktury skládající se z názvu nonterminálu a řetězce terminálů, které se na nonterminály navazují.

Nonterminály se do vektoru vkládají v pořadí, v jakém se nacházejí na pravé straně pravidla. Nový nonterminál se vloží vždy na konec vektoru. Do pomocného řetězce terminálů se místo nonterminálu vloží „zarážka“, která označuje místo budoucího vložení obsahu nonterminálu.

Po přepsání zásobníkových nonterminálů na nonterminály ve vektoru se provede vložení obsahu nonterminálů do pomocného řetězce terminálů. Obsah každého nonterminálu se vloží na místo určené „zarážkou“. Protože se nonterminály z vektoru čtou od začátku, v podstatě princip fronty, a i pomocný řetězec terminálů se prochází od začátku, je vkládání obsahu nonterminálů do řetězce terminálů triviální – v pomocném řetězci je nalezena „zarážka“, vezme se další nonterminál z vektoru, „zarážka“ se z pomocného řetězce odstraní a místo ní se vloží obsah nonterminálu. Postup je vidět na obrázku Obr 3.27.



Obr 3. 27

4. Implementace

Popis implementace jsem rozdělila na čtyři logické celky. První část je zaměřena na interakci s uživatelem, respektive získání potřebných vstupních údajů. Druhým celkem je převod souboru se vstupními gramatikami na zdrojový kód pro překladač Bison. Třetím logickým celkem je implementace lexikální analýzy. Čtvrtou část tvoří popis vlastní implementace metody, která byla vysvětlena ve třetí kapitole.

Hned v úvodu bych ráda zmínila, že používám knihovny *string*, *vector* a *stack*, které mi prostřednictvím svých funkcí usnadňují práci s řetězci, vektory a zásobníky. Využitím těchto knihoven jsem nemusela implementovat podpůrné funkce pro práci s výše zmíněnými strukturami. Zároveň jsem dosáhla obecnosti řešení, protože uživatel si může nadefinovat řetězce terminálů, jména nonterminálů i počet gramatik neomezeně, z mé strany není ničím limitován. Také jsem se tímto vyhnula případnému alokování a uvolňování paměti.

Program bude použit pro překlad z jednoho programovacího jazyka do *n* jiných programovacích jazyků, kde $n >= 1$.

4.1 Vstup od uživatele

Program s uživatelem během provádění jednotlivých fází nekomunikuje. Výjimkou je počáteční fáze před převodem vstupního souboru na zdrojový kód pro Bison. V této počáteční fázi je potřeba získat od uživatele informace o vstupní gramatice, klíčových slovech, znacích komentářů a pravidlech gramatik. Úvodní část je, co se týče kódu, umístěna jak v části převodu gramatik, tak v části lexikální analýzy. Používané třídy jsou nadeklarovány v hlavičkovém souboru *LexAnalys.h*, metody jsou definovány v souboru *LexAnalys.cpp* a volány z hlavní funkce souboru *Conversion.cpp*.

Prvotní komunikace s uživatelem spočívá ve vypsání požadavků na vstupní soubory. Jsou očekávány tři soubory, přičemž dva musí mít přesně stanovený název a strukturu. V prvním souboru jsou obsažena klíčová slova a znaky komentářů. Údaje jsou zapsány v požadovaném tvaru (bude vysvětleno v kapitole šesté). Soubor musí mít název *keywords.txt* a musí být umístěn ve stejném adresáři jako zdrojové soubory programu. Druhý vstupní soubor obsahuje pravidla gramatik, které musí být zapsány určeným způsobem (bude popsáno v kapitole šesté). Soubor se musí jmenovat *grammars.txt* a musí být umístěn ve stejném adresáři jako zdrojové soubory programu. Třetím souborem je zdrojový kód zapsaný v programovacím jazyce jehož syntaxi popisuje vstupní gramatika.

4.1.1 Vstupní gramatika

Po úvodních informacích je uživatel požádán o zadání hodnoty, která udává číslo gramatiky. Tato se dále bude považovat za gramatiku vstupní. Pod pojmem vstupní gramatika rozumíme pravidla gramatiky, která popisují syntaxi programovacího jazyka, který chceme přeložit. Vstupní parametr je instancí třídy a má své metody.

```
class TInputParameter
{
    private:
        int input_gram;

    public:
        TInputParameter();
        ~TInputParameter();
        void ReadInputGram();
        int GetInputGram();
        int ControlInputParameter(int);
};
```

4.1.1.1 Metoda ReadInputGram

Metoda nemá žádné vstupní ani výstupní parametry. Metoda zajistí načtení hodnoty určující číslo vstupní gramatiky. To udává, která gramatika je výchozí - na kterou gramatiku bude použita syntaktická analýza. Metoda vyžaduje od uživatele zadání platné hodnoty, respektive kladného celého čísla, které by nebylo mimo rozsah možných hodnot. Tento rozsah je dán počtem gramatik ve vstupním souboru *grammars.txt* a je kontrolován pomocí metody *ControlInputParameter*. Uvažujeme-li tři gramatiky ve vstupním souboru, pak uživatel může zadat pouze hodnoty v rozsahu jedna až tři.

4.1.1.2 Metoda ControlInputParameter

Vstupním parametrem hodnoty je proměnná typu *int* udávající hodnotu zadanou uživatelem. Výstupním parametrem je proměnná typu *int* fungující jako booleovská proměnná. Návratová hodnota je buď nula za předpokladu, že vstupní parametr nese hodnotu mimo rozsah, nebo jedna, pokud uživatel zadal hodnotu v rámci rozsahu.

Metoda načte první pravidla všech gramatik ze vstupního souboru *grammars.txt* a určí, kolik je pravidel. Pokud hodnota zadaná uživatelem je větší než počet gramatik, případně je nulová nebo záporná, je uživatel požádán o zadání nové hodnoty.

4.1.1.3 Metoda GetInputGram

Metoda nemá žádné vstupní ani výstupní parametry. Metoda pouze vrací hodnotu vstupní gramatiky.

4.1.2 Klíčová slova a znaky komentáře

Jak bylo zmíněno výše, jedním ze souborů, které se od uživatele očekávají, je i *keywords.txt* obsahující klíčová slova a znaky komentářů. Soubor má jasně danou strukturu, která bude popsána v šesté kapitole. Klíčová slova se ukládají do vektoru řetězců (typ *vector<string>*). Znaky komentáře se ukládají do speciální struktury a ta se ukládá do vektoru (typ *vector<TComment>*).

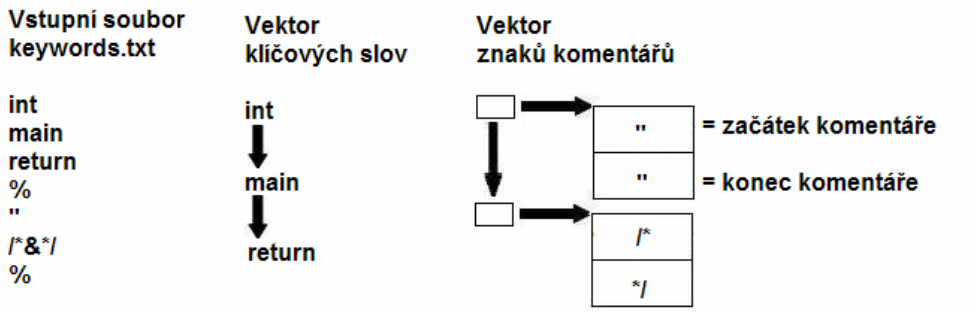
```
struct TComment
{
    string begin_comment;
    string end_commen
}

vector <string> KeyWords;
vector <TComment> CharComment;

Metody:
int ReadKeyWords();
int IsKeyWord(string);
void GetKeyWords();
void GetCharComment();
```

4.1.2.1 Metoda ReadKeyWords

Metoda nemá žádné vstupní parametry. Návratová hodnota je typu *int* a udává, zda načtení klíčových slov a znaků komentáře proběhlo úspěšně.

Metoda čte vstupní soubor po řádcích, co řádek, to jedno klíčové slovo. Každé klíčové slovo vloží do vektoru. Klíčová slova a znaky komentářů jsou od sebe odděleny znakem % (procento). Po ukončení načítání klíčových slov se přejde k načítání znaků komentáře. Těmito znaky jsou myšleny i uvozovky a apostrofy a např. dvojznak /* označující znak komentáře v jazyce C. Komentář může začínat jiným znakem a jiným znakem končit, např v jazyce C začíná komentář dvojznakem /*, ale končí dvojznakem */. Každá jedna definice symbolů komentáře je na samostatném řádku. Uživatel může nadefinovat komentář pomocí počátečního znaku a

koncového znaku, které jsou vzájemně oddělené znakem & (ampersant), nebo může zadat komentář jen jedním znakem (v případě, že komentář začíná i končí stejným znakem, jak je tomu například u uvozovek). Znaky komentáře jsou ukončeny znakem % (procento). Princip metody je vidět na obrázku Obr 4.1.



Obr 4.1

4.1.2.2 Metoda IsKeyWord

Vstupním parametrem je proměnná typu *string*, kterou chceme testovat. Návratovou hodnotou je proměnná typu *int*, která má hodnotu nula, pokud testovaný vstup není klíčovým slovem, nebo jedna v případě, že se jedná o klíčové slovo.

Metoda projde celý vektor klíčových slov a porovnává jednotlivá klíčová slova se vstupním řetězcem. V případě shody má návratová proměnná hodnotu jedna.

4.1.2.3 Metoda GetKeyWords

Metoda nemá žádné vstupní ani výstupní parametry. Metoda vypíše všechna klíčová slova na výstup.

4.1.2.4 Metoda GetCharComment

Metoda nemá žádné vstupní ani výstupní parametry. Metoda vypíše všechny znaky komentáře na výstup.

4.2 Převod vstupních gramatik

Druhým logickým celkem celého programu je samostatný podprogram pro převod gramatických pravidel uložených v souboru *grammars.txt* na zdrojový kód pro překladač Bison. Kód tohoto podprogramu je uložen v souboru *Conversion.cpp*, hlavičkový soubor má název *Conversion.h.*

Soubor s gramatickými pravidly musí mít název *grammars.txt* a musí být uložen ve stejném adresáři jako jsou zdrojové soubory programu. Tento soubor musí mít předepsaný formát

(více bude uvedeno v kapitole šesté), protože program, který tento soubor zpracovává, očekává jisté pomocné znaky podle nichž se řídí. Podprogram si vyžádá zadání vstupní hodnoty, očekává se zadání celého čísla.

Podprogram je napsán v jazyce C++ a řídí se zásadami objektového programování. Jednotlivé třídy a jejich metody jsou nadeklarovány v hlavičkovém souboru. Definice funkcí je v *Conversion.cpp*. Postupně popíši činnost jednotlivých metod, které používám.

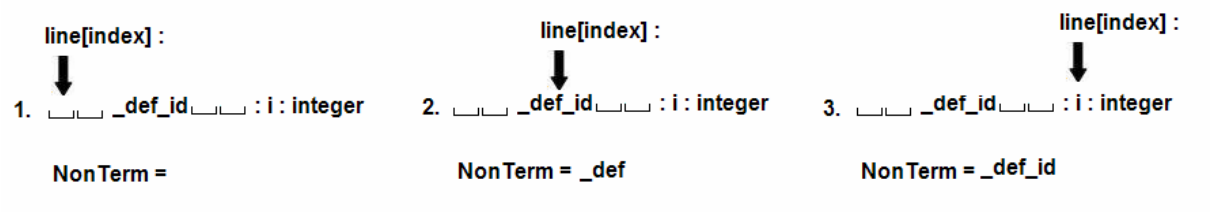
```
class TConvert
{       public:
                int index;
                string line;
                fstream fin;
                fstream fout;

                TConvert();
                ~TConvert();
                void MainFunction(int,TKeyWords);
                string ReadLeftSide();
                string ReadRightSide();
                int CountNonterms(string);
                string ConvertRightSide(string,TKeyWords);
};
```

4.2.1 Metoda ReadLeftSide

Tato metoda nemá žádné vstupní parametry, protože využívá proměnou *line*, která je deklarována jako proměnná třídy *TConvert.* Výstupním parametrem je řetězec typu *string*

Tato metoda, jak je zřejmé z jejího názvu, načte levou stranu pravidla, tedy nonterminál, který je zároveň návratovou hodnotou. Postupně se prochází řetězec *line* (prochází se znak po znaku), případné mezery se ignorují a do pomocné proměnné *NonTerm* typu *string se* uloží název nonterminálu. Gramatická pravidla mají daný tvar (více v kapitole šesté), metoda tedy ví, že nonterminál začíná znakem ‘_’ (podtržítko), a ví že má ukončit načítání ve chvíli, kdy najde znak ‘:’ (dvojtečka). Znak dvojtečka nahrazuje dvojznak ‘->’, který se při definici přepisovacích pravidel většinou používá, a označuje, že nonterminál z levé strany pravidla má být přepsán na posloupnost terminálů a nonterminálů z pravé strany pravidla. Znak dvojtečka se samozřejmě

může objevit i na pravé straně pravidla, ale my víme, že je to pravá strana pravidla, a že tento znak má být pouze načten jako znak a není tudíž chápán jako oddělovač mezi levou a pravou stranou pravidla. Princip metody je názorně vysvětlen na obrázku Obr 4.2.

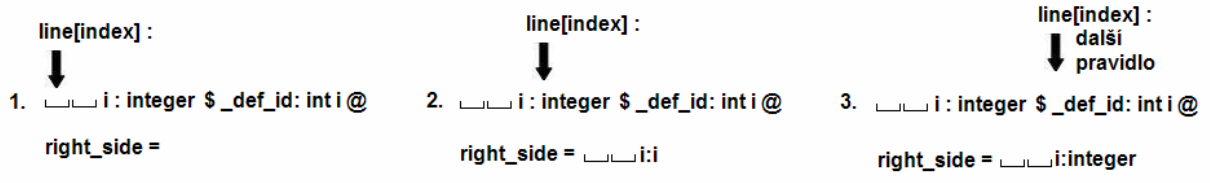


Obr 4.2

4.2.2 Metoda ReadRightSide

Metoda nemá žádné vstupní parametry, protože využívá proměnou *line*, která je deklarována jako proměnná třídy *TConvert*. Výstupním parametrem je řetězec typu *string*

Jak je opět z názvu patrné, načte metoda pravou stranu pravidla, tedy posloupnost terminálních a neterminálních symbolů. Postupně se prochází řetězec *line* (prochází se znak po znaku) a všechny znaky (včetně mezer) se kopírují do pomocného řetězce *right_side*, který je zároveň návratovou hodnotou metody. Metoda opět pracuje se znalostí tvaru gramatických pravidel. Jednotlivá pravidla jsou oddělena znakem *'$'* (dolar) a poslední pravidlo v řadě je ukončeno znakem *'@'* (zavináč). Načítání pravé strany pravidla končí ve chvíli, kdy je objeven jeden ze zmíněných znaků. Postup je názorně ukázán na obrázku Obr 4.3.

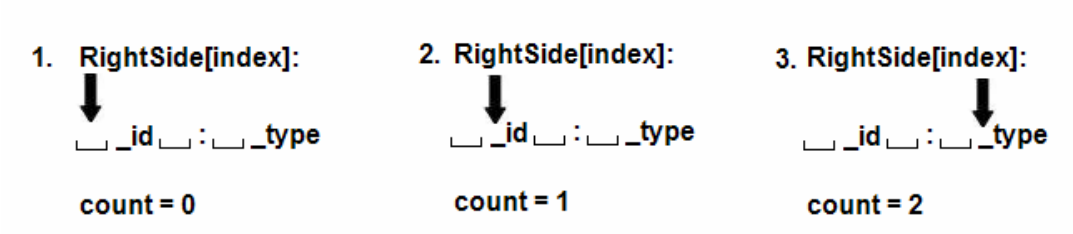


Obr 4.3

4.2.3 Metoda CountNonterms

Vstupním parametrem metody je řetězec terminálů a nonterminálů z pravé strany pravidla (proměnná typu *string*), který byl načten pomocí metody *ReadRightSide*(). Výstupním parametrem je proměnná typu *int*, která udává počet nonterminálů v řetězci.

Postupně se prochází řetězec terminálů a nonterminálů, který byl metodě předán jako vstupní parametr. Řetězec se prochází znak po znaku a hledají se nonterminály, které jsou označeny počátečním znakem podtržítko. Návratovou hodnotou je počet nonterminálů v řetězci. Princip metody je ukázán na obrázku Obr 4.4.

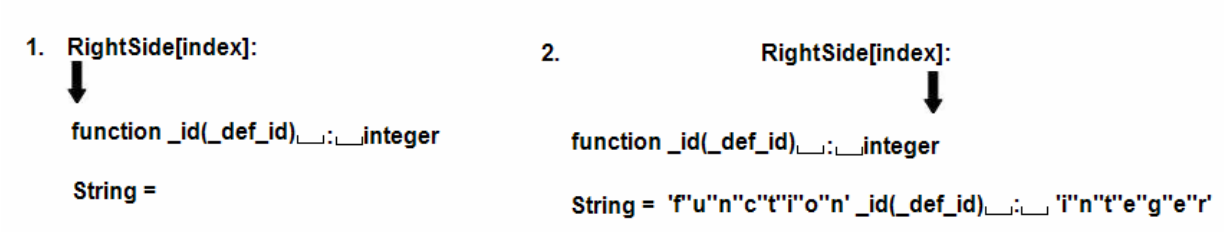


Obr 4.4

4.2.4 Metoda ConvertRightSide

Metoda má dva vstupní parametry. Prvním parametrem metody je řetězec terminálů a nonterminálů z pravé strany pravidla (proměnná typu *string*), který byl načten pomocí metody *ReadRightSide*(). Druhým vstupním parametrem je vektor klíčových slov, který je typu *TKeyWords.* Výstupním parametrem je upravený řetězec terminálních a neterminálních symbolů, parametr je typu *string.*

Postupně se prochází řetězec terminálů a nonterminálů. Při načtení víceznakového terminálu se nejprve zkontroluje, zda se nejedná o klíčové slovo. To musí být přepsáno do požadovaného tvaru – každé písmeno klíčového slova je odděleno apostrofy, tedy ve výsledku je klíčové slovo rozděleno na jednotlivé znaky (respektive písmena). Víceznakové terminály, které nejsou klíčovými slovy se beze změny překopírují do výstupního řetězce. Mezery, jednotlivé znaky a nonterminály jsou vloženy do výstupního řetězce beze změny. Princip metody je ukázán na obrázku Obr 4.5.



Obr 4.5

Tento přepis klíčových slov je nutný kvůli práci překladače Bison. Ten dostává od lexikální analýzy pouze ordinální čísla jednotlivých znaků, tedy nemůže získat celé slovo. Možností, jak nadefinovat celá slova, je definice tokenů, kterou překladač Bison umožňuje. Bohužel syntaxe překladače vyžaduje, aby všechny nadefinované tokeny byly zapsány pouze velkými písmeny. Toto omezení neumožňuje např. definovat klíčová slova jazyka C, který naopak vyžaduje zapsání klíčových slov malými písmeny - zde dochází ke kolizi syntaxe jazyka C a syntaxe překladače Bison. Proto jsem zvolila řešení přepisu klíčových slov po jednotlivých znacích. Syntaktický analyzátor tedy nepředloží překladači celé klíčové slovo např. *function*, ale

bude mu jej postupně „hláskovat“ (*f-u-n-c-t-i-o-n*), tedy předkládat ordinální čísla jednotlivých znaků (respektive písmen).

4.2.5 MainFunction

Metoda má dva vstupní parametry. Prvním parametrem je proměnná typu *int*, která udává číslo vstupní gramatiky. Druhým vstupním parametrem je vektor klíčových slov, který je typu *TKeyWords*. Metoda nemá žádné výstupní parametry.

Metoda vytvoří soubor s názvem *output.y++*, který je zdrojovým souborem pro překladač Bison. Vygenerovaný soubor musí mít všechny náležitosti, které překladač Bison vyžaduje. Soubor je v následujícím tvaru:

```
%{
deklarace v jazyce C++
%}
deklarace překladače Bison (definice tokenů)
%%
pravidla gramatika  sémantické akce
%%
kód v jazyce C++ (lexikální analýza, výpis chyb)
```

Nejprve se do výstupního souboru zapíší všechny úvodní deklarace jazyka C++ a deklarace překladače Bison (definice tří pomocných tokenů). Následuje zpracování gramatik, které jsou očekávány v předepsaném tvaru (detailní informace budou uvedeny v šesté kapitole) v souboru *grammars.txt*, který musí být umístěn ve stejném adresáři jako zdrojové soubory programu. Soubor s gramatickými pravidly se čte po řádcích. Pravidla mohou být i na více řádcích, proto se všechna odpovídající si pravidla načtou do jednoho pomocného řetězce typu *string*. Odpovídající mi si pravidly jsou myšlena pravidla, která popisují stejný syntaktický jev (např. popis funkce, definice konstant).

Jako první se zpracuje vstupní gramatika, která je určena vstupním parametrem metody. Projde se pomocný řetězec s načtenými odpovídajícími si pravidly a najde se pravidlo vstupní gramatiky. Pravidlo je načteno pomocí metod *ReadLeftSide*, *ReadRightSide* a modifikováno pomocí metody *ConvertRightSide*. V načtené pravé straně pravidla se navíc určí počet nonterminálů pomocí metody *CountNonterms*. Vstupní pravidlo je vloženo do výstupního souboru. Následuje zpracování zbývajících pravidel. Ta se už pouze načtou metodami *ReadLeftSide* a *ReadRightSide* a bez dalších úprav se vloží v požadovaném tvaru do výstupního souboru. Požadovaným tvarem je myšlen zápis sémantické akce, která spočívá ve volání metody *Simuluj* s jejími parametry. Zápis je ve tvaru:

Simuluj(pořadí_gramatiky,count=CountNonterms,“LeftSide“,“RightSide“).

Výše popsaným způsobem se projde celý soubor s gramatikami. Nakonec se do výstupního souboru zapíše kód v jazyce C++, který volá lexikální analýzu a předává překladači Bison jednotlivé tokeny.

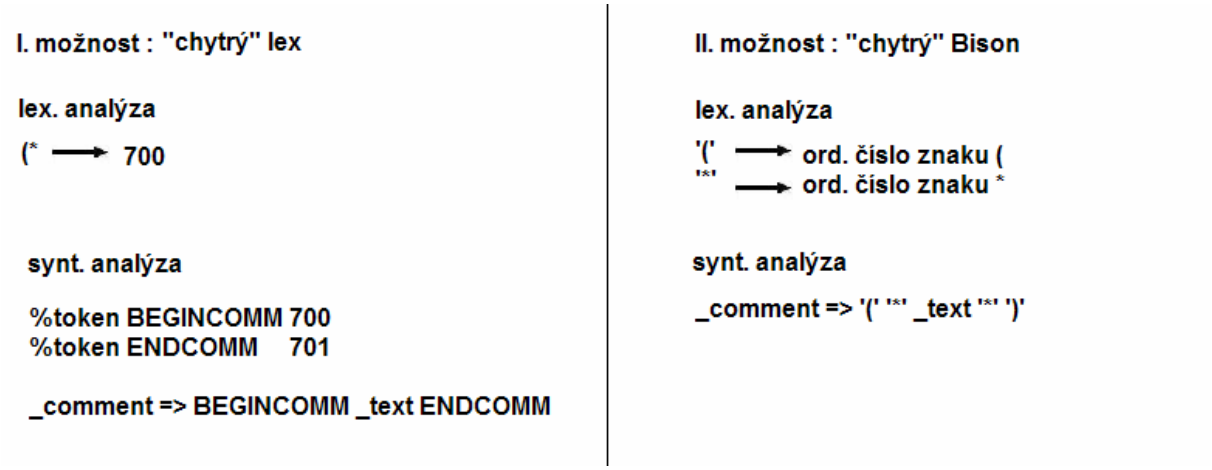
4.3 Lexikální analýza

Třetím logickým celkem je lexikální analýza. Syntaktický analyzátor volá lexikální analyzátor pokaždé, když potřebuje získat další token. Syntaktický analyzátor potřebuje tokeny v podobě ordinálních čísel jednotlivých načtených znaků, proto lexikální analyzátor vrací vždy jen jeden znak, respektive jeho ordinální číslo. Jsou zde ale tři výjimky - čísla, identifikátory a text v komentářích – těm je přidělena speciální hodnota, podle které syntaktický analyzátor pozná, že se jedná o jednu z těchto výjimek. Čísla, identifikátory a texty jsou předávány vcelku, čili mohou mít více znaků. Toto je zajištěno pomocí speciálních tokenů NUM, ID a TEXT, které jsou definovány ve zdrojovém kódu *output.y++* předávaném překladači Bison. Používání identifikátorů je omezeno - musí začínat písmenem a mohou obsahovat čísla a znak podtržítko.

Při vzájemné komunikaci mezi syntaktickým a lexikálním analyzátorem je potřeba si pamatovat tři základní údaje – stav, ve kterém je lexikální analyzátor (např. je ve stavu „komentář“), typ komentáře (musí vědět, jakým znakem byl komentář uveden, aby věděl, jakým znakem musí skončit) a pomocný řetězec (pokud se vyskytne víceznakový terminál, respektive klíčové slovo, nebo víceznakový symbol komentáře, musí jej předat po jednom znaku).

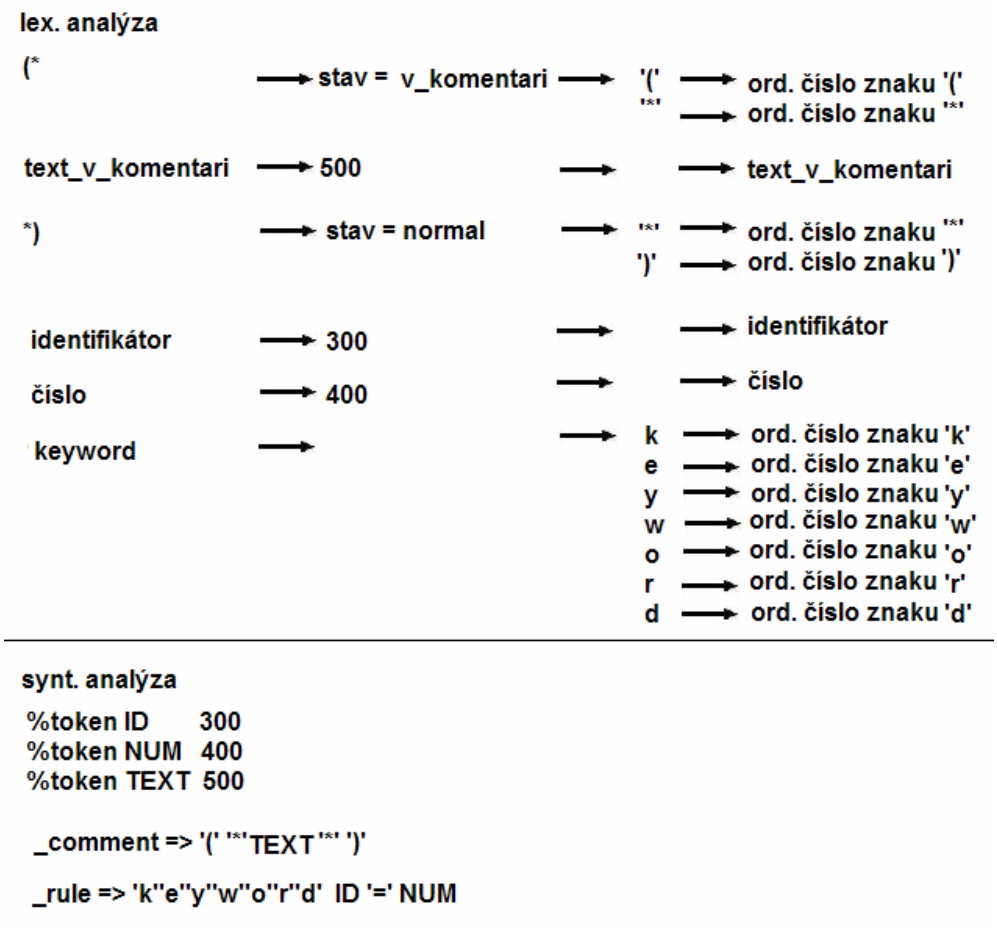
S nutností pamatovat si tři údaje souvisí i návrh lexikální analýzy. Jak již bylo zmíněno, syntaktický analyzátor očekává od lexikálního analyzátoru pouze jedno ordinální číslo, které určuje načtený znak. Problém nastává v případě víceznakových symbolů pro komentář a u klíčových slov. Možná řešení jsou dvě. V prvním případě bychom měli „chytrý“ lexikální analyzátor, který by nám načetl celé slovo a syntaktickému analyzátoru by předal hodnotu, která by toto slovo označovala. Bylo by ale potřeba nadefinovat všechna klíčová slova a všechny víceznakové symboly komentáře jako tokeny. Ty ale mohou být zapsány pouze písmeny a navíc pouze velkými písmeny, což v případě komentáře ani v případě většiny klíčových slov není splněno. Navíc v případě více možností komentářů bychom nevěděli, jakými znaky komentář začal a jakými znaky jej tedy ukončit. Druhým možným řešením je mít „chytrý“ syntaktický analyzátor. V rámci pravidel by se nadefinovaly všechny symboly komentářů a klíčová slova po jednotlivých znacích a syntaktický analyzátor by mohl vybrat správné pravidlo. Syntaktický analyzátor by ale nevěděl, co je textem v komentáři a kdy komentář končí, navíc by nebyla

rozlišena klíčová slova od identifikátorů (název identifikátoru by nemohl být načtený jako celek, byl by předáván po znacích). Obě varianty jsou vysvětleny na obrázku Obr 4.6.



Obr 4.6

Ani jedna z výše popsaných možností nebyla vyhovující. Proto jsem zvolila kombinaci obou těchto variant. Mám „chytrý“ lexikální analyzátor a zároveň „chytrý“ syntaktický analyzátor, respektive z první možnosti jsem si vzala lexikální analýzu a z druhé možnosti syntaktickou analýzu. Díky lexikální analýze jsou symboly komentáře i klíčová slova načtena vcelku, čili načte se celé klíčové slovo a celý víceznakový symbol komentáře. Protože překladač Bison neumí pracovat s celým slovem, je tento celek rozdělen po jednotlivých znacích. Z tohoto důvodu potřebuji uchovávat stav lexikálního analyzátoru a pomocný řetězec, který si pamatuje vždy zbytek slova, které je potřeba rozdělit na jednotlivé znaky. Stav analyzátoru slouží pouze k informaci, zda se nacházíme v komentáři, tedy načítáme text uvnitř komentáře aniž bychom zjišťovali význam jednotlivých znaků. „Chytrý“ syntaktický analyzátor určuje pravidla, která se použijí a jednotlivá klíčová slova a symboly komentářů má rozděleny po znacích. Toto řešení je vidět na obrázku Obr 4.7.



Obr 4.7

Při implementaci používám dvě třídy, jednu popisující strukturu tokenu předávaného syntaktickému analýtoru a druhou obsahující metody pro rozpoznávání jednotlivých tokenů.

```
class TToken
{
        string TokenName;
        int TokenType; // 300 = identificator, 400 = number, 500 = text
        TToken();
        ~TToken();
};
```

```
enum States {normal, comment_establish, comment, comment_saved};
class TParser
{
    private:
          States state;
          int typeOfcomment;
          string AuxString;

    public:
         string NewWord;
         char c;
         TKeyWords KeyWords;
         TToken NewToken;

         TParser();
         ~TParser();
         TToken MainFunction(int, States, string);
         void ReadDigit();
         void ReadWord();
         void ReadChar();
         States GetState();
         int GetTypeComment();
         string GetAuxString();
};
```

4.3.1 Metoda ReadDigit

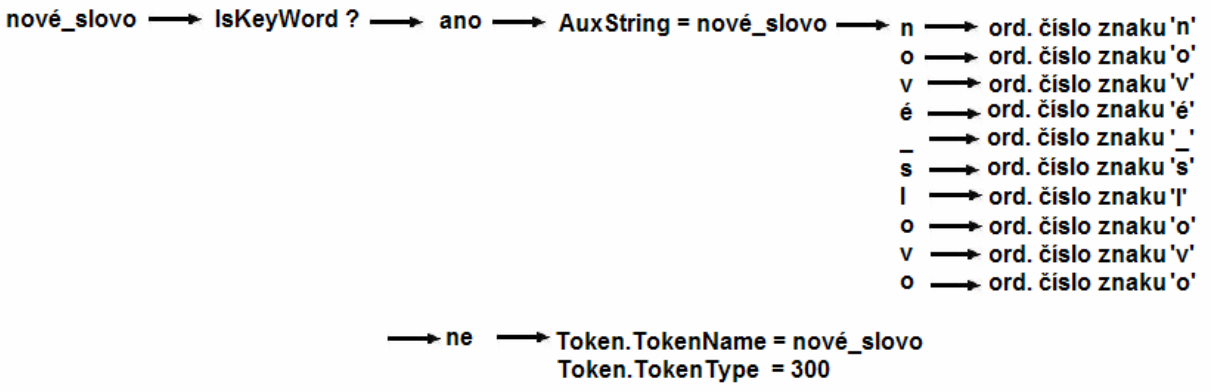
Metoda nemá žádné vstupní ani výstupní parametry. Výsledkem je nový token, který je proměnou třídy *TParser*.

Metoda je volána za předpokladu, že se na vstupu objevila číslice. Metoda načte celé číslo i desetinné číslo - předpokladem je použití desetinné tečky (nikoliv čárky). Jméno nového tokenu je tvořeno číselnou hodnotou, typ tokenu je nastaven na hodnotu čtyři sta – je to hodnota označující, že se jedná o číslo. Překladač Bison má nadefinovaný speciální token *NUM 400*, díky němuž může být číslo vráceno jako celek, nikoliv po jednotlivých znacích.

4.3.2 Metoda ReadWord

Metoda nemá žádné vstupní ani výstupní parametry. Výsledkem je nový token, který je proměnnou třídy *Tparser*.

Metoda je volána, pokud se na vstupu objeví písmeno. Metoda načte celé slovo, které začíná písmenem a může obsahovat číslice a znak podtržítko – identifikátory musí začínat písmenem a mohou mít v názvu číslice i podtržítka. Po načtení slova je volána metoda *IsKeyWord* jejímž parametrem je právě načtené slovo. Zjistí se, zda slovo je klíčovým slovem nebo identifikátorem. Nový token je vytvořen pouze v případě, že se jedná o identifikátor. Jméno tokenu je tvořeno názvem identifikátoru, typ tokenu je nastaven na hodnotu tři sta - je to hodnota označující, že se jedná o identifikátor. Překladač Bison má nadefinovaný speciální token *ID 300*, díky němuž může být identifikátor vrácen jako celek, nikoliv po jednotlivých znacích. V případě klíčového slova je toto vloženo do pomocného řetězce *AuxString* a syntaktickému analyzátoru je předáno po jednotlivých znacích. Princip metody je ukázán na obrázku Obr 4.8.



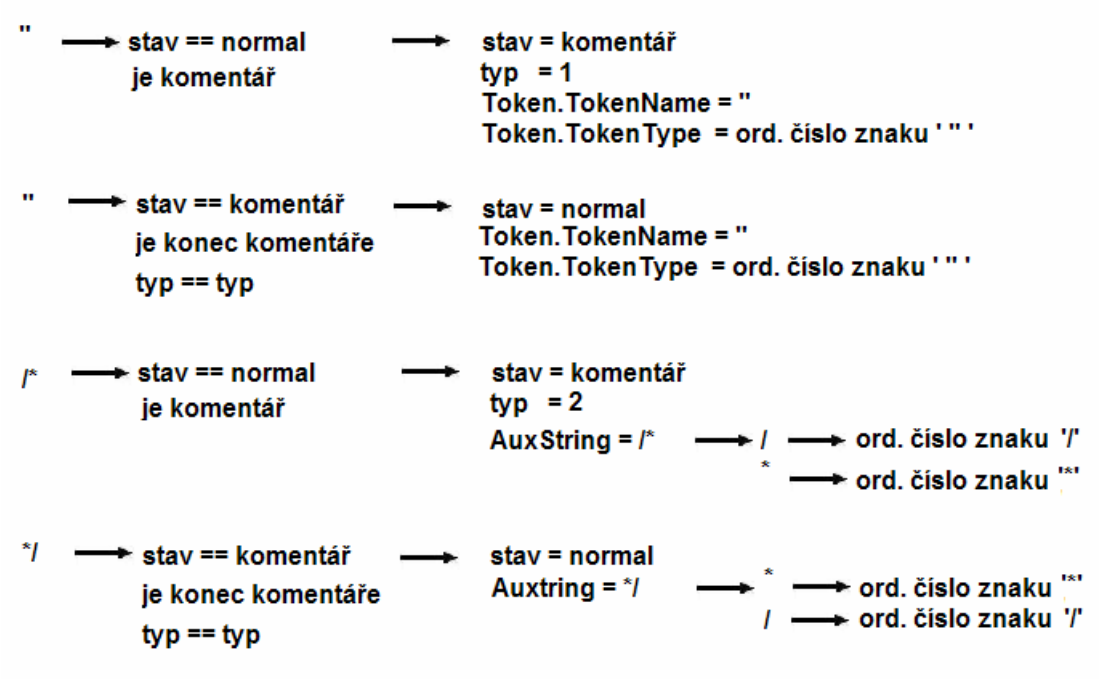
Obr 4.8

4.3.3 Metoda ReadChar

Metoda nemá žádný vstupní ani žádný výstupní parametr. Výsledkem je nový token, který je proměnnou třídy *Tparser*.

Metoda je volána pokud se na vstupu objeví libovolný znak (jakýkoliv znak mimo písmena a číslice). Metoda kontroluje, zda načtený znak není symbolem komentáře. Prochází se vektor komentářů a hledá se případná shoda. Symbol komentáře může být tvořen více znaky, v tom případě se načte ze vstupu více znaků, aby se ověřilo, zda se opravdu jedná o komentář. Pokud byl opravdu načten symbol komentáře, je stav lexikálního analyzátoru změněn na stav označující, že je v komentáři, a analyzátor si zapamatuje typ komentáře (např. byla načtena uvozovka, nebo byly načteny znaky /*). Typ komentáře si musí pamatovat z důvodu, aby věděl jaký symbol ukončení komentáře má hledat. Pokud je symbol komentáře tvořen více znaky, je

vložen do pomocného řetězce *AuxString*, v opačném případě se vytvoří nový token obsahující pouze jeden znak. Stejně tak pokud se nejedná o symbol komentáře, vytvoří se nový token jehož jméno je tvořeno jediným znakem a jeho typ je tvořen ordinálním číslem tohoto znaku. Pokud už lexikální analyzátor je ve stavu komentář, testuje se, zda načtený znak není symbolem ukončení komentáře. Musí se zkontrolovat, zda se jedná i o správný typ ukončení komentáře (např. komentář začínající uvozovkou může být ukončen pouze uvozovkou, komentář začínající znaky /* muže být ukončen pouze znaky */). Princip je ukázán na obrázku Obr 4.9.

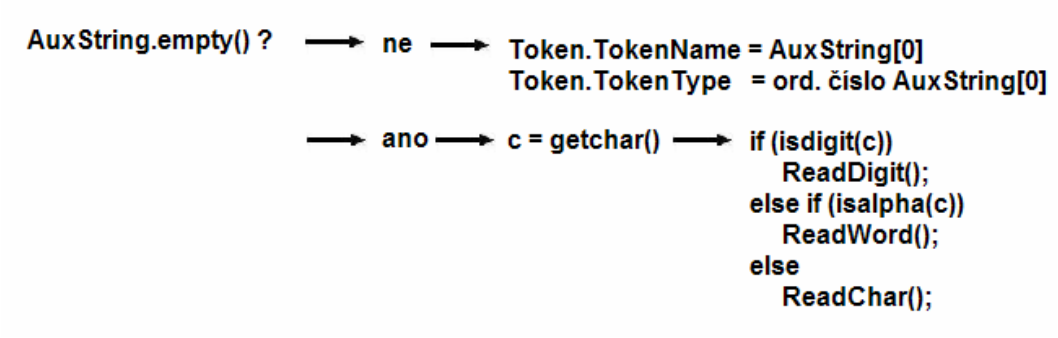


Obr 4.9

4.3.3 Metoda MainFunction

Metoda má tři vstupní parametry a to typu *int* udávající typ komentáře, typu *States* udávající stav lexikálního analyzátoru a typu *string* udávající obsah pomocného řetězce. Výstupní parametr je typu *TToken* a udává nový token.

Metoda načítá jednotlivé znaky ze vstupu, kterým je soubor obsahující zdrojový kód zapsaný v nějakém programovacím jazyce, jehož syntaxe je popsána v souboru *grammars.txt*, a který je dán hodnotou vstupní gramatiky. Podle načteného znaku volá jednu z metod *ReadDigit*, *ReadWord* nebo *ReadChar*. Pokud byl ve vstupním parametru předán neprázdný pomocný řetězec, je novým tokenem další znak tohoto řetězce, v opačném případě je novým tokenem to, co vznikne aplikací výše zmíněných metod. Princip metody je vidět na obrázku Obr 4.10.



Obr 4.10

4.4 Implementace metody

Čtvrtým logickým celkem je implementace vlastní metody. Výše uvedené tři celky jsou v podstatě pouze podporou pro využití této metody. Implementace je v souboru *Simulation.cpp* a její hlavní a nejdůležitější částí je funkce *Simuluj()*. Při implementaci používám tyto struktury:

```
class TItem
{
    public:
        string Nonterm;
        string StringOfTerms;

        TItem();
        ~TItem();
};

typedef stack <TItem> TStack;
vector <TStack> ArrayOfStack;
```

Třída *TItem* je strukturou pro uložení nonterminálu z levé strany pravidla a odpovídajícího řetězce terminálů, který se na nonterminál váže. Typ *TStack* je zásobník, do kterého se vkládají objekty třídy *TItem*. *ArrayOfStack* je vektor zásobníků, respektive dynamické pole zásobníků. Pro snazší implementaci používám knihovny *stack*, *string* a *vector*, které mají definované funkce pro práci se zásobníkem, řetězcem a vektorem. Díky těmto knihovnám nemusím implementovat podpůrné funkce a ani alokovat a uvolňovat paměť.

Nyní vysvětlím princip pomocných funkcí, které jsou volány z funkce *Simuluj()*.

4.4.1 Funkce PrintResult

Funkce nemá žádné vstupní ani výstupní parametry. Funkce vypíše výsledek překladu do výstupních souborů. Postupně projde celý vektor zásobníků *ArrayOfStack*, vezme obsah vrcholů

všech zásobníků, ty odstraní ze zásobníků a zkontroluje, zda jsou zásobníky prázdné. Obsahy zásobníků vypíše do výstupních souborů, které jsou pojmenovány *out_X.txt,* kde *X* je písmeno abecedy. První výstupní soubor se jmenuje *out_a.txt,* druhý výstupní soubor má název *out_b.txt* atd. Výstupních souborů je o jeden méně než je gramatik v souboru *grammars.txt.*

4.4.2 Funkce Init

Funkce má jeden vstupní parametr typu *int,* který udává požadovaný počet zásobníků ve vektoru. Funkce nemá žádný výstupní parametr.

Vstupní parametr udává požadovaný počet zásobníků uložených ve vektoru. Pokud je skutečný počet menší než požadovaný počet, přidá se do vektoru zásobník na jehož dně je symbol dna zásobníku (znak *$*).

4.4.2 Funkce InsertToVector

Funkce má dva vstupní parametry. Prvním parametrem je zásobník, který je potřeba vložit do vektoru a druhý je typu *int* udávající pozici ve vektoru, na kterou je potřeba zásobník vložit. Funkce nemá žádný výstupní parametr.

Funkce aktualizuje vektor zásobníků. Prochází celý vektor zásobníků a jednotlivé zásobníky, které nejsou na pozici zadané vstupním parametrem beze změny překopíruje do pomocného vektoru. Zásobník na hledané pozici se aktualizuje – na danou pozici se do vektoru vloží zásobník, který je vstupním parametrem.

4.4.3 Funkce InsertToNonterms

Funkce má tři vstupní parametry. Prvním je vektor nonterminálů, druhý je aktuální položka, kterou chceme do vektoru vložit, a třetím parametrem je pozice ve vektoru, na kterou chceme vstupní položku vložit. Funkce má jeden výstupní parametr, kterým je aktualizovaný vektor nonterminálů.

Funkce aktualizuje vektor nonterminálů. Prochází celý vektor nonterminálů a jednotlivé nonterminály i s obsahem, které nejsou na pozici zadané vstupním parametrem beze změny překopíruje do pomocného vektoru. Položka na hledané pozici se aktualizuje – na danou pozici se do vektoru vloží položka (nonterminál s řetězcem terminálů), který je vstupním parametrem.

4.4.4 Funkce Simuluj

Funkce má čtyři vstupní parametry. První parametr je typu *int* a udává pozici ve vektoru zásobníků. Druhý parametr je také typu *int* a udává počet nonterminálů na pravé straně vstupního pravidla. Třetí a čtvrtý parametr jsou typu *string* a obsahují levou a pravou stranu gramatického pravidla.

Funkce je rozdělena na tři části. V první části se zpracuje pravá strana pravidla, vytvoří se pomocný řetězec a vektor nonterminálů. Ve druhé části se vektor nonterminálů naplní obsahem nonterminálů ze zásobníku. Ve třetí fázi se obsah nonterminálů z vektoru nonterminálů překopíruje do upravené pravé strany pravidla reprezentované pomocným řetězcem. Celý princip metody je popsaný ve třetí kapitole.

Pravá strana pravidla se zpracuje následujícím způsobem:

- samostatné znaky jako například středník, závorka, hvězdička atd. se vloží do pomocného řetězce ,
- víceznakové terminály, např. klíčová slova, se nejprve celé načtou a poté vloží do pomocného řetězce,
- mezera se vloží do pomocného řetězce,
- nonterminál se celý načte, do pomocného řetězce se vloží „zarážka“, která označuje, kam je potřeba obsah nonterminálu po jeho zpracování vložit. Nonterminál se vloží na konec vektoru nonterminálů.

Princip je ukázán ve třetí kapitole na obrázku Obr 3.18.

V druhém kroku se ze zásobníku vyjme tolik nonterminálů, kolik určuje druhý parametr funkce *Simuluj()*. Názvy nonterminálů ze zásobníku se porovnají se všemi nonterminály z vektoru nonterminálů. Při shodě jmen je obsah zásobníkového nonterminálu vložen do vektorového nonterminálu. Zde se uplatňuje redukce a klonování nonterminálu, které je popsáno ve třetí kapitole. Princip této části je pak ukázán na obrázcích ze třetí kapitoly Obr 3.19a a Obr 3.19b.

V závěrečném třetím kroku se obsah nonterminálů z vektoru nonterminálů vloží do pomocného řetězce na místo určené zarážkou. Vznikne tak řetězec terminálů, který se vloží na vrchol zásobníku, viz obrázky Obr 3.20 a Obr 3.21 ve třetí kapitole.

5. Gramatická pravidla

V této kapitole popíši některá vybraná gramatická pravidla, která jsem vytvořila, a zdůvodním jejich použití a tvar. Na základě mých gramatických pravidel funguje obousměrný překlad z jazyka Pascal do jazyka C a naopak. Oba jazyky jsou zjednodušené a při zápisu zdrojového kódu je třeba dbát na omezení vyplývající z gramatických pravidel. Na základě pravidel se vygeneruje zdrojový kód v příslušném jazyce, který je funkční a spustitelný. Zdrojové kódy nejsou efektivní – vzhledem ke specifikům jednotlivých jazyků nelze navrhnout gramatiky, které by generovaly efektivní kód. Efektivita vygenerovaného kódu ale nebyla požadavkem. Třetí gramatika pravidla popisuje syntaxi jazyka Modula-2. Tento jazyk jsem si vybrala pouze na doplnění - jako ukázku funkčního multigenerování. Zdrojový text v jazyce Modula-2 vygenerovaný na základě těchto pravidel nemusí být plně funkční a přeložitelný. Stejně tak pravidla pro Modula-2 nemusí tento jazyk plně generovat.

Mezi jednotlivými jazyky je rozdíl v syntaxi, v některých případech je tento rozdíl značný. Výsledná gramatická pravidla tyto rozdíly řeší tak, že je uživatel omezen nutností zapsat do vstupního zdrojového kódu některé části, které se běžně v kódu nevyskytují, nebo není zvykem je používat. Některá omezení se mohou zdát být příliš přísná, ale jejich význam bude vysvětlen níže. Všechna gramatická pravidla jsou uvedena v příloze číslo dva.

5.1 Bílé znaky

Bílé znaky jsou značným problémem, který se nepodařilo zcela vyřešit. Nelze do gramatiky zadat pravidlo, které by zajistilo přepis nonterminálu na bílý znak (odřádkování, mezera, tabulátor atd.), protože bílý znak se může vyskytovat prakticky kdekoliv a není reálné do gramatik na jakékoliv místo vložit nonterminál generující bílý znak. Proto jsem zvolila variantu, kdy se z gramatiky bere znak mezery (kolik uživatel vloží mezer do gramatických pravidel, tolik jich bude ve vygenerovaném kódu) a odřádkování se provádí za každým středníkem. Tato varianta bohužel způsobí ztrátu uživatelova formátování – mezery ve zdrojovém kódu jsou ignorovány a odřádkování nemusí odpovídat uživatelovu odřádkování. Zároveň se například odřádkují i podmínky v příkazu *for*, neboť definice tohoto příkazu obsahuje dva středníky:

for(i=1;i<10;i++)

vygenerovaný výsledek bude vypadat následovně:

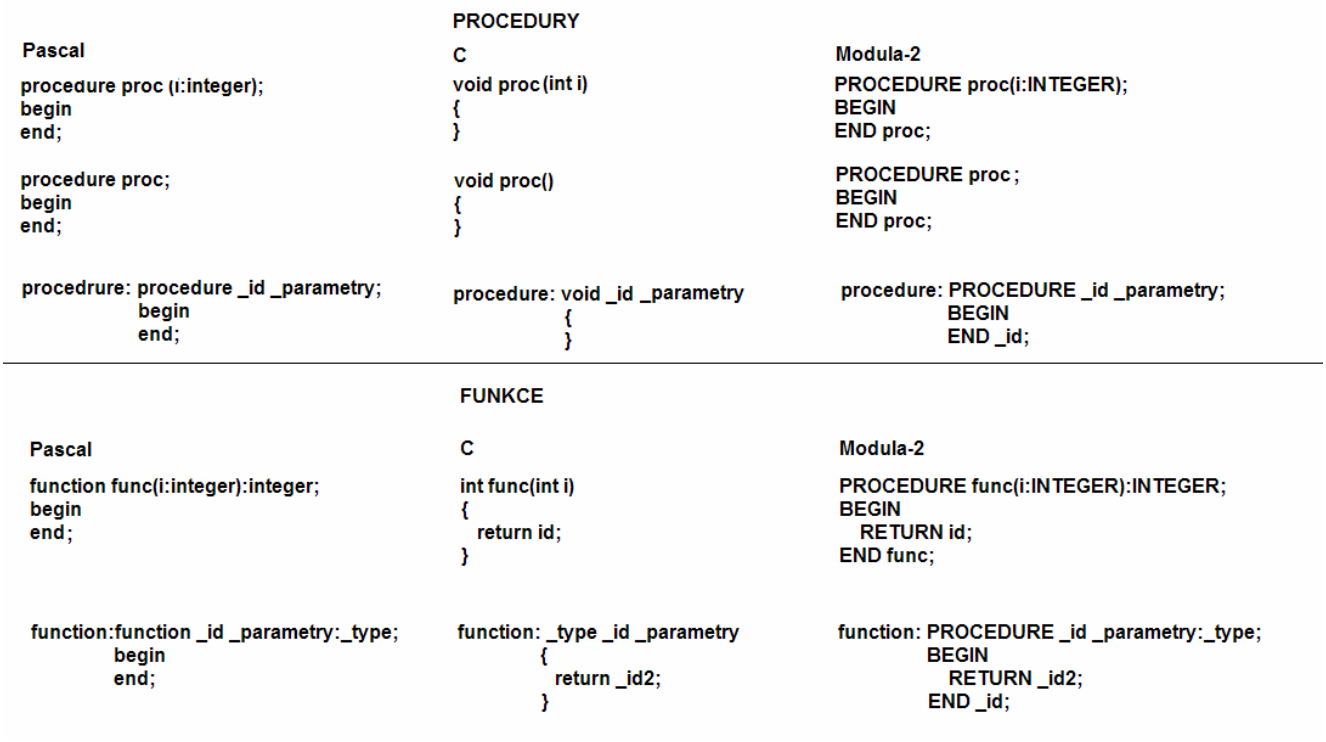
for(i=1;
i<10;
i++)

Avšak tento zápis je přeložitelný a nebrání funkčnosti vygenerovaného kódu.

Problém ale zůstává u definice konstant a knihoven. Za těmito definicemi nejsou středníky, a proto se neodřádkují. Uživatel si musí tuto úvodní část odřádkovat manuálně, aby byl kód přeložitelný.

5.2 Procedury a funkce

Definice procedur a funkcí má u každého ze tří jazyků jinou syntaxi. Na obrázku Obr 5.1 je tato syntaxe vidět. Stejně tak její zápis pomocí gramatik.



Obr 5.1

Výše uvedený zápis gramatických pravidel by nefungoval. U funkcí v jazyce C a Modula-2 se pro návrat hodnoty používá *return* a vrací se proměnná jejíž hodnota je funkcí vypočítána. U těchto dvou jazyků by gramatika obsahovala oproti jazyku Pascal jeden nonterminál navíc, což je nepřípustné. Neplatila by podmínka, která říká, pokud se na pravé straně pravidla u jazyka Pascal vyskytuje nonterminál s názvem <*id*>, musí se nonterminál s tímto názvem vyskytovat i na pravých stranách jazyků C a Modula-2. A naopak, pokud se nonterminál s názvem např. <*id2*> na pravé straně pravidla jazyka Pascal nevyskytuje, nesmí se objevit ani na pravých stranách jazyků C a Modula-2. Navíc při překladu z jazyka Pascal do jazyků C a Modula-2 bychom nevěděli, která proměnná má být návratová. Pro zachování funkčnosti a přeložitelnosti je potřeba upravit pravidla pro generování funkcí do tvaru, který je uveden na obrázku Obr 5.2.

FUNKCE

Pascal	C	Modula-2

function func(i:integer):integer;
begin
end;
	int func(int i)
	{
	int func;
	return func;
	}
		PROCEDURE func(i:INTEGER):INTEGER;
		VAR func:INTEGER;
		BEGIN
		RETURN func;
		END func;

function:function _id _parametry:_type;
begin
end;
	function: _type _id _parametry
	{
	_type _id;
	return _id;
	}
		function: PROCEDURE _id _parametry:_type;
		VAR _id:_type;
		BEGIN
		RETURN _id;
		END _id;

Obr 5.2

Upravená pravidla zajistí definici nové proměnná, která má stejný název a je stejného typu jako je název a typ funkce. Tato proměnná je zároveň návratovou hodnotou. Při generování z jazyka Pascal do jazyků C a Modula-2 tento fakt uživatele nijak neomezí, ale při zpětném překladu z C do Pascalu toto omezení vyžaduje, aby uživatel kód v jazyce C napsal přesně podle gramatického pravidla, čili aby návratová proměnná měla stejné jméno a typ jako má funkce. Takovýto zápis není pro jazyk C přirozený, ale bohužel není jiné vhodné řešení, které by splňovalo podmínky kladené na tvar gramatických pravidel. Ideálním řešením by bylo použití sémantické analýzy, kterou jsem ale neimplementovala.

5.3 Načítání a výpis

Dalším rozdílem v syntaxi je řešení načítání ze standardního vstupu a výpis na standardní výstup. Jazyk Pascal používá dvě univerzální funkce *read* a *write*, které zajistí načtení a výpis proměnné libovolného jednoduchého typu. Jazyk C používá funkce *scanf* a *printf*, které prostřednictvím parametrů načítají nebo vypisují proměnnou určeného typu. Jazyk Modula-2 má funkcí pro načítání a výpis několik, typ načítané nebo vypisované proměnné je dán názvem použité funkce. Na obrázku Obr 5.3 je vidět rozdíl v jednotlivých funkcích.

Pascal	C	Modula-2

i:integer :
read(i);
write(i);
	int i :
	scanf("%d",i);
	printf("%d",i);
		i:INTEGER :
		ReadInt(i);
		WriteInt(i,num);

i:real :
read(i);
write(i);
	float i :
	scanf("%f",i);
	printf("%f",i);
		i:REAL :
		ReadReal(i);
		WriteReal(i,num);

i: char :
read(i);
write(i);
	char i :
	scanf("%c",i);
	printf("%c",i);
		i: CHAR :
		ReadChar(i);
		WriteChar(i);

Obr 5.3

Kdyby se takto nadefinovala gramatická pravidla, docházelo by při překladu k nejednoznačnostem. Syntaktický analyzátor by při překladu z jazyka Pascal do jazyků C a Modula-2 nevěděl, které pravidlo má použít, protože by měl nadefinovaná tři stejná pravidla pro čtení a tři stejná pravidla pro zápis. Tento nedeterminismus je potřeba odstranit, ale jen pomocí gramatických pravidel bychom toho nedosáhli. Proto jsem musela vytvořit zvláštní knihovnu pro jazyk Pascal a pro jazyk C, kde se jednotlivé funkce pro načtení a výpis přejmenovaly podle vzoru jazyka Modula-2. Uživatel při zápisu zdrojového kódu v jazyce Pascal a C musí používat určené názvy funkcí pro čtení a výpis podle typu čtené nebo vypisované proměnné. Nadefinované uživatelské knihovny jsou v příloze číslo jedna. Tyto knihovny je při zápisu kódu nutné zadat, u jazyka Pascal je při definici knihoven nutné zapsat *uses crt, uses_lib*, u jazyka C navíc používáme knihovnu *#include "user_lib"*. Výsledná gramatická pravidla jsou vidět na obrázku Obr 5.4.

```
_write:writeln$ _write:WriteLn('')$ _write:WriteLn@
_write:WriteString('\"_text\"')$ _write:WriteString('\"'_text'\"')$ _write:WriteString('\"'_text'\"')@
_write:WriteInt('_expr','_num')$ _write:WriteInt('_expr','_num')$ _write:WriteInt('_expr','_num')@
_write:WriteReal('_expr','_num')$ _write:WriteReal('_expr','_num')$ _write:WriteReal('_expr','_num')@
_write:WriteChar('_expr')$ _write:WriteChar('_expr')$ _write:WriteChar('_expr')@


_read:ReadString('_expr')$_read:_expr'='ReadString('')$ _read:ReadString('_expr')@
_read:ReadInt('_expr')$_read:_expr'='ReadInt('')$ _read:ReadInt('_expr')@
_read:ReadReal('_expr')$_read:_expr'='ReadReal('')$_read:ReadReal('_expr')@
_read:ReadChar('_expr')$_read:_expr'='ReadChar('')$_read:ReadChar('_expr')@
```

Obr 5.4

5.4 Deklarace pole

Jedná se opravdu jen o deklaraci pole nikoliv o definici nového typu. Rozdíl v syntaxi mezi jednotlivými jazyky je vidět na obrázku Obr 5.5.

```
Pascal      i : array[0..10] : integer;
            _pole : _id : array[zero.._num]:_type

C           int i [10];
            _pole : _type _id [_num]

Modula-2    i :ARRAY[0..10]:INTEGER;
            _pole : _id:ARRAY[zero.._num]:_type
```

Obr 5.5

V jazyce C se definuje rozsah pole pomocí jediné číselné hodnoty *n*. Znamená to, že pole se vždy čísluje od nuly po *n*-1. U jazyků Pascal a Modula-2 se pole čísluje od *n* do *m* a jeho velikost je dána rozdílem *m*-*n*. Opět tu nastává problém v počtu nonterminálů na pravé straně pravidla a v počátku číslování. Směrodatný byl v tomto případě jazyk C, neboť u něj vždy platí, že pole je číslováno od nuly a proto i v jazycích Pascal a Modula-2 musí uživatel číslovat pole od nuly. Aby byl dodržen počet nonterminálů na pravé straně pravidla, musela být do jazyků Pascal a Modula-2 přidána konstanta *zero*. Tato konstanta se při použití pole musí v jazyce Modula-2 nadefinovat v částí úvodních deklarací, u jazyka Pascal je tato konstanta nadeklarovaná v uživatelské knihovně, která musí být automaticky použita (viz kapitola 5.3 Načítání a výpis).

5.5 Název programu

U jazyků Pascal a Modula-2 se používá úvodní pojmenování programu. V jazyce Pascal je použit zápis *Program Name*, v jazyce Modula-2 pak zápis *MODULE Name*. Jazyk C žádné pojmenování programu nepoužívá. Tímto opět nastává problém v počtu nonterminálů na pravé straně pravidla. Při překladu z jazyka Pascal do jazyka C by přebýval nonterminál obsahující název programu. Při překladu z jazyka C do jazyka Pascal by nonterminál s názvem programu naopak chyběl a nevědělo by se, jak program pojmenovat. Proto je v gramatice napevno dán název *Program Pascal* a *MODELU MODULA*, který musí být uživatelem dodržen. Slova *Pascal* a *MODULA* jsou definována jako klíčová – musí být uvedena v souboru *keywords.txt*.

5.6 Komentáře

Uživatel si může nadefinovat komentáře dle potřeby. Může používat libovolné symboly komentáře, které si nadefinuje jednak v souboru *keywords.txt* (podrobnosti viz kapitola šestá) a zároveň je nadefinuje jako gramatické pravidlo. V mých gramatikách je použití komentáře omezeno na místa, kde by se dal napsat příkaz. Čili nemůže být mezi jednotlivými deklaracemi, názvy funkcí v úvodních definicích atd. Ačkoliv mám implementovanou lexikální analýzu, která by mimo jiné měla odstraňovat komentáře, moje lexikální analýza toto nedělá. Myslím si, že když si uživatel napíše zdrojový kód včetně komentářů, měly by být tyto komentáře při překladu do jiného jazyka zachovány.

5.7 Identifikátory, čísla a text

V každém zdrojovém kódu se objevují identifikátory, čísla a texty umístěné v komentáři nebo mezi uvozovkami či apostrofy (např. výpis textu na standardní výstup). Při definici gramatik musí být na místě, kde chce uživatel mít identifikátor , umístěn nonterminál *_id*, na místě čísla nonterminál *_num* a na místě textu nonterminál *_text*. V gramatikách pak musí být navíc pravidlo přepisu těchto nonterminálů na tokeny, díky nimž bude brát syntaktický analyzátor takovýto token jako celek (celé slovo, celé číslo) nikoliv po jednotlivých znacích. Gramatická pravidla jsou uvedena na obrázku Obr 5.6.

```
_id :ID$  _id :ID$  _id :ID@
_num :NUM$  _num :NUM$ _num :NUM@
_text :TEXT$ _text:TEXT$ _text:TEXT@
```

Obr 5.6

5.8 Odstranění dalších nedeterminismů

Výše jsem popsala některá řešení, která odstraňují nederminismus, i když za cenu některých omezení uživatele. Mezi další nedeterminismy patří například definice typů. Jazyky Pascal a Modula-2 používají typ *Boolean*, který se ale v jazyce C nepoužívá, respektive není standardně používaným typem. Funkci typu *boolean* může v jazyce C zastoupit typ *int* – proměnná tohoto typu by nabývala hodnot pouze jedna nebo nula. Pokud bychom toto použili, vytvořili bychom nedeterminismum ukázaný na obrázku Obr 5.7. Při překladu z jazyka C do jazyků Pascal a Modula-2 by syntaktická analýza nevěděla, které pravidlo použít. Z tohoto důvodu není typ *Boolean* přípustný.

```
_type :int$ _type :integer$ _type :INTEGER@
_type :int$ _type :boolean$ _type :BOOLEAN@
```

Obr 5.7

Podobný problém nastává i při používání operátorů *DIV* a *MOD*. Zatímco pro *MOD* má jazyk C symbol *%*, tak pro *DIV* nic podobného nemá. Proto bylo použití operátoru *DIV* zakázáno.

5.9 Posloupnost deklarací a příkazů

Při zápisu zdrojového kódu je očekávána následující posloupnost deklarací a definicí:

- název programu – *Program Pascal* pro Pascal, *MODULE MODULA* pro jazyk Modula-2
- knihovny – ve tvaru přesně dle pravidel gramatik uvedených v příloze číslo dva

- definice konstant – při použití pole musí být v jazyce Modula-2 nadefinována konstanta *zero.*
- deklarace proměnných
- definice funkcí a procedur– uvnitř nejdříve deklarace proměnných
- hlavní část - u jazyka Pascal mezi *begin end.*, u jazyka Modula-2 mezi *BEGIN END MODULA.*, u jazyka C v *int main(){ return 0}*

6. Práce s programem

V této kapitole bude popsána práce s programem, požadované vstupní soubory, tvary těchto souborů, a výstupní soubory, které program vytvoří.

6.1 Vstupy programu

Požadované vstupy se dají rozdělit do dvou kategorií: vstupy, které si program sám vyhledá v adresáři a načte si je. A vstupy, které jsou zadány uživatelem jako vstupní parametry při spouštění části programu. První typ vstupu budeme nazývat *automatické vstupy* a druhé *vstupní parametry.*

6.1.1 Automatické vstupy

Název automatické vstupy je použit z toho důvodu, že uživatel je nikam nezadává jako vstupní parametr při spouštění programu, ale jsou programem očekávány – program si je sám načte. Automatické vstupy jsou požadovány dva. Jedním je soubor obsahující pravidla gramatik s názvem *grammars.txt* a druhým je soubor obsahující klíčová slova a symboly komentářů s názvem *keywords.txt.* Oba tyto soubory musí mít předepsaný tvar a oba musí být umístěny v adresáři společně se zdrojovými kódy programu.

6.1.1.1 Vstupní soubor grammars.txt

Vstupní soubor *grammars.txt* musí být umístěn v adresáři společně se zdrojovými kódy programu. Soubor obsahuje pravidla jednotlivých gramatik. Požadavky na tvar souboru jsou následující:

- jednotlivá odpovídající si pravidla jsou oddělena znakem *$* (dolar), za posledním pravidlem je znak @ (zavináč).
- na prvním řádku jsou startovací pravidla všech *n* gramatik, na dalších řádcích jsou vzájemně si odpovídající pravidla všech *n* gramatik,
- nonterminály mohou mít libovolně dlouhý název, který musí začínat znakem ‘_’ (podtržítko). V názvu mohou být písmena, číslice a podtržítka,
- základním předpokladem je mít na levé straně odpovídajících si pravidel nonterminály se shodnými názvy. To stejné musí platit i o nonterminálech na pravých stranách pravidel. Pokud máme pravidlo u jazyka Pascal nazvané <*fun*>, musí mít odpovídající pravidla jazyků C a Modula-2 shodný název. Pokud se na pravé straně pravidla u jazyka Pascal vyskytuje nonterminál s názvem <*id*>, musí se nonterminál s tímto názvem vyskytovat i

na pravých stranách jazyků C a Modula-2. A naopak, pokud se nonterminál s názvem např. <*id_id*> na pravé straně pravidla jazyka Pascal nevyskytuje, nesmí se objevit ani na pravých stranách jazyků C a Modula-2.

- klíčová slova neboli víceznakové terminály mohou mít libovolnou délku. Jejich název musí začínat písmenem a může obsahovat písmena, číslice a podtržítka,
- pokud klíčové slovo začíná nebo končí znakem, pak tento znak musí být v gramatice zapsán jako samostatný znak a do souboru klíčových slov se zapíše pouze slovo bez znaků (např. *#define* musí být v gramatice zapsáno jako *'#'define* a do souboru klíčových slov ze zapíše pouze *define*)
- samostatné znaky se musí vložit do apostrofů (např. *'='*), toto platí i v případě, že máme přímo v gramatice nadefinované číslo (např. *_id'='1'* nebo *_id'='1'.'2'* - přiřazení desetinného čísla),
- levá strana pravidla je od pravé strany pravidla oddělena znakem *':'* (dvojtečka),
- na místě, kam má být vložen případný text, číslo nebo identifikátor použijeme nonterminály s názvem *_text*, *_num* nebo *_id* přesně v tomto tvaru. Do gramatiky musí být navíc připsána pravidla *_text:TEXT*, *_num:NUM* a *_id:ID*,
- kolik bude v pravidlech mezer, tolik bude mezer ve vygenerovaném výstupu. Proto vkládáme mezery pouze tam, kde jsou nutné, nebo kde je opravdu chceme mít (např. mezera mezi názvy nonterminálů),

Uživatel si sám musí zkontrolovat, zda si jednotlivá pravidla všech gramatik na daném řádku odpovídají. Dále si musí uživatel zkontrolovat, zda zadal pravidla správně a zda daná gramatika přijme uživatelem zadaný vstupní řetězec. Příklad vstupního souboru je na obrázku Obr 6.1.

```
_Expr:_Expr '+' _id$_Expr:'+' _Expr _ id $_Expr:_Expr _ id '+'@
_Expr:_num$_Expr:_num$_Expr:_ num @
_comment:'/''*' _text '*''/' $_comment:'{' _text '}' $_comment:'(''*' _text '*'')' @
_lib:'#'include _id$_lib: uses_id$_lib:FROM T IMPORT _id@
_num:NUM$_num:NUM$_num:NUM@
_id:ID$_id:ID$_id:ID@
_text:TEXT$_text:TEXT$_text:TEXT@
```

Obr 6.1

6.1.1.2 Vstupní soubor keywords.txt

Vstupní soubor *keywords.txt* musí být umístěn v adresáři společně se zdrojovými kódy programu. Soubor obsahuje klíčová slova a symboly komentářů vstupní gramatiky. Pod pojmem vstupní gramatika se myslí pravidla popisující syntaxi jazyka, ve kterém je zapsán vstupní soubor – zadává jej uživatel jako vstupní parametr podprogramu (např. vstupním souborem je zdrojový kód zapsaný v jazyce Pascal a proto *keywords.txt* bude obsahovat klíčová slova a symboly komentářů jazyka Pascal). Požadavky na tvar souboru jsou následující:

- nejprve jsou nadefinována klíčová slova,
- každé slovo je na samostatném řádku,
- klíčové slovo začíná písmenem, může obsahovat i čísla a podtržítka a má neomezenou délku,
- v případě, že klíčové slovo začíná nebo končí znakem, pak do souboru je zapsána pouze část klíčového slova, která obsahuje písmena, čísla nebo podržítka, „přebývající“ znak je nadefinován v pravidlech gramatiky (např. *#include* -> do klíčových slov se vloží pouze *include* a do pravidel gramatiky se zapíše např. *_nonterm:'#'include*),
- klíčová slova jsou od symbolů gramatiky oddělena znakem % (procento), který je umístěn na samostatném řádku,
- symboly komentářů označují, kterým znakem začíná a kterým znakem končí komentář. Za komentář se považují i znaky uvozovky a apostrof, neboť vše, co je mezi těmito znaky se bere jako celek bez zjišťování významu jednotlivých znaků,
- každá dvojice nebo každý symbol komentáře je na samostatném řádku
- symboly začátku a konce komentáře jsou odděleny znakem & (ampersant, např. /*&*/)
- pokud je symbol pro začátek i konec komentáře totožný, nemusíme jej udávat jako párový (např. znak uvozovka může být zadán dvěma způsoby: "&" nebo pouze ")
- za posledním znakem je na samostatném řádku znak % (procento)
- některá část může být vynechána. Pokud chceme nadefinovat pouze symboly komentáře bez klíčových slov, musíme nejprve zadat znak % (procento). V případě, že nechceme definovat ani klíčová slova a ani symboly komentáře, necháme soubor prázdný, soubor ale musí existovat.

Na obrázku Obr 6.2 jsou vidět čtyři varianty vstupního souboru.

řádný vstup:	bez klíčových slov:	bez symbolů komentáře:	prázdný soubor:
main include % “ ‘&’ /*&*/ %	% “ ‘&’ /*&*/ %	main include	

Obr 6.2

6.1.2 Vstupní parametry

Vstupní parametry jsou dva. Prvním je uživatelem definovaný soubor a druhým je soubor *output.y++*, který je vytvořen podprogramem a je dále použit jako vstup pro překladač Bison.

6.1.2.1 Vstupní řetězec

V libovolně pojmenovaném souboru uživatel zadává vstupní řetězec, který chce pomocí syntaktické analýzy založené na multigenerování převést na *n*-1 výstupních řetězců. Vstupním řetězcem se myslí zdrojový kód zapsaný v některém z programovacích jazyků. Kód musí být zapsán v souladu se vstupní gramatikou, protože tento řetězec se zpracuje použitím pravidel gramatik, které jsme zadali v souboru *grammars.txt*. Příklad vstupu je uveden na obrázku Obr 6.3

```
Program Pascal;
uses crt,user_lib;
var n,i,max:integer;
   a:array[zero..9] of integer;
begin
 max:=0;
 WriteString('Zadejte pocet prvku, maximum je 10:');
 writeln;
 ReadInt(n);
end.
```

Obr 6.3

Při zadávání pravidel gramatik si musí uživatel zkontrolovat, zda pravidla vstupní gramatiky generují vstupní řetězec. Pokud tomu tak není, pak dojde během provádění programu

k chybě a program bude ukončen. Důvod ukončení programu se objeví jako chybové hlášení na standardním výstupu.

6.1.2.2 Soubor output.y++

Obsah souboru *output.y++* nemusí uživatele zajímat. Jedná se o soubor, který je vygenerovaný podprogramem, a který je použit jako zdrojový soubor předkládaný překladači Bison. Do souboru *output.y++* se vygenerují potřebné deklarace pro překladač Bison, upravená pravidla gramatik zadaných ve vstupním souboru *grammars.txt* a další potřebné funkce. Vygenerování souboru *output.y++* zajistí podprogram *Conversion.cpp*. Tento soubor můžeme považovat jak za vstupní, tak i za výstupní (je výstupem podprogramu *Conversion.cpp* a zároveň vstupem pro překladač Bison). Příklad souboru *output.y++* je na obrázku Obr 6.4

```
%{
#include "LexAnalys.cpp"
#include "Simulation.cpp"
TToken Token;
%}
%token ID 300
%token NUM 400
%token TEXT 500
%%
_id : ID  { Simuluj(1,0,"_id",Token.TokenName);  Simuluj(2,0,"_id",Token.TokenName);  }
%%
int yylex ()
{
    Token = Parser.MainFunction(typeOfcomment, state, Word);
     return Token.TokenType;
}

void yyerror (const char *error)
{
cout << error << endl;
}
```

Obr 6.4

6.2 Výstupy programu

Výstupy programu lze rozdělit na soubor *output.y++* a na soubory obsahující řetězce vygenerované na základě práce syntaktické analýzy využívající gramatická pravidla nadefinovaná v souboru *grammars.txt.* Soubor *output.y++* může být považován za výstup, protože je výstupem podprogramu *Conversion.cpp*, ale zároveň také za vstup, protože se jako parametr předkládá překladači Bison, viz kapitola „6.1.2.2. Soubor output.y++“.

6.2.1 Výstupní soubory out_X.txt

Výstupních souborů je několik. Celkový počet souborů je dán počtem gramatik, které jsou nadefinovány v souboru *grammars.txt*. Máme-li *n* gramatik, pak bude vygenerováno *n*-1 výstupních souborů. Název souborů bude *out_X.txt*, kde *X* udává písmeno abecedy. První soubor bude mít tedy název *out_a.txt*, druhý soubor bude mít název *out_b.txt* atd. Ve výstupních souborech je zapsán výsledek syntaktické analýzy založené na multigenerování po provedení programu. Jedná se o vygenerované zdrojové kódy zapsané v programovacích jazycích, jejichž syntaxi gramatiky popisují.

6.3 Spuštění programu

Nejprve všechny zdrojové soubory (*Conversion.cpp, Conversion.h, ErrorMsgs.h, LexAnalys.cpp, LexAnalys.h, Msgs.h, Simulation.cpp, main.cpp, makefile*) a požadované vstupní soubory (*keywords.txt, grammars.txt*) umístíme do jednoho adresáře.

Podprogram *Conversion* zajišťuje převod vstupního souboru *grammars.txt* obsahujícího pravidla gramatik na výstupní soubor *output.y++* obsahující deklarace pro překladač Bison a upravená pravidla gramatik. Postup je vysvětlen v kapitole „4.2 Převod vstupních gramatik“. Podprogram *LexAnalys* provádí lexikální analýzu a je využíván syntaktickou analýzou, viz kapitola „4.3 Lexikální analýza“. Podprogram *Simulation* obsahuje nejdůležitější funkci celého programu a tou je funkce *Simuluj()*, která pracuje s pravidly gramatik a generuje výstupní řetězce. Podrobně je funkce popsána v kapitole „4.4 Implementace metody“. Hlavní program *main* sjednocuje podprogram lexikální analýzy a podprogram generování výstupů.

Program potřebuje mít k dispozici překladač Bison, proto jej spouštíme na serveru, který má tento překladač nainstalovaný. V rámci FIT VUT je to server *merlin.fit.vutbr.cz*. Program spouštíme z příkazové řádky. Spuštění programu probíhá následovně:

- nejprve použijeme makefile pro soubor *Conversion.cpp*, do příkazové řádky zapíšeme příkaz *make*. tímto se zdrojový kód zkompiluje a vytvoří se program *conversion*,
- pokračujeme zadáním příkazu *./conversion*, zde je uživatel požádán o zadání vstupní hodnoty, očekává se celé číslo. V našem případě se zadáním čísla jedna vstupní gramatikou stává ta, která popisuje syntaxi jazyka Pascal. Zadáním hodnoty dva se vstupní gramatikou stává ta, která popisuje syntaxi jazyka C. Zadáním hodnoty tři jsou vstupními pravidly ta, která popisují syntaxi jazyka Modula-2 (tato pravidla jsou pouze pro doplnění za účelem ukázky multigenerování, nemusí tedy generovat přeložitelný

zdrojový kód v jazyce Modula-2, stejně tak nemusí tento jazyk správně popisovat). Výstupem podprogramu je vygenerovaný soubor *output.y++*,

- spustíme překladač Bison a jako vstupní parametr mu předložíme právě vygenerovaný soubor *output.y++*. Do příkazové řádky zadáme příkaz *bison output.y++*, tímto se vytvoří soubor *output.tab.c++*, který je použit v *main.cpp*,
- zkompilujeme *main.cpp* zadáním příkazu *g++ main.cpp*, tímto se vytvoří program *a.out*,
- nakonec tento program spustíme zadáním příkazu *./a.out <soubor_pro_preklad*, kde *soubor_pro_preklad* obsahuje zdrojový kód napsaný v programovacím jazyce jehož syntaxi určuje vstupní gramatika definovaná v souboru *grammars.txt*

Pro jednoduchost uvedu ještě jednou a zkráceně posloupnost příkazů:

- make
- ./conversion
- bison output.y++
- g++ main.cpp
- ./a.out <soubor_pro_preklad

7. Závěr

Ve své diplomové práci jsem popsala novou metodu multigenerování, která je zcela paralelní. Metoda jeden vstupní řetězec generovaný vstupní gramatikou převede paralelně na *n*-1 výstupních řetězců.

Snažila jsem se dosáhnout obecnosti použití programu. Ten nemusí sloužit pouze k překladu mezi vyššími programovacími jazyky, ale obecně k překladu libovolného řetězce, který je generován uživatelem zadanou gramatikou.

Aby byla navržená metoda obecně použitelná, rozšířila jsem ji o lexikální analýzu, která umožňuje práci s čísly, identifikátory a textovými řetězci. Díky lexikální analýze nemusí být vstup čten po jednotlivých znacích, ale může být čten po slovech, respektive znakových řetězcích.
Metoda je určena pro jednu vstupní gramatiku a *n*-1 výstupních gramatik. Délka terminálních ani neterminálních znaků není omezena a to díky použité knihovně *string* a díky lexikální analýze. Ta rozdělí vstupní posloupnost znaků na lexémy - lexikální jednotky (např. identifikátory, čísla, klíčová slova, atd). Tyto lexémy jsou reprezentovány ve formě tokenů, které jsou poskytnuty syntaktickému analyzátoru. Metoda je obecně použitelná pro libovolný shora neomezený počet výstupních gramatik.

Jediným omezením je tvar gramatických pravidel, které si uživatel nadefinuje. Vstupní řetězec musí být gramatickými pravidly generovatelný. Mnou zvolená gramatická pravidla vygenerují spustitelný program v jazyce C a Pascal, vygenerovaný kód v jazyce Modula-2 je doplňkem pro ukázku multigenerování, tudíž nemusí být po vygenerování plně funkční. Gramatiky zajistí vygenerování funkčního kódu, nikoliv efektivního kódu. Efektivnost nelze zaručit, neboť každý programovací jazyk je něčím specifický a při vzájemném překladu jazyků se tato specifika musí částečně potlačit, respektive musí být nějakým způsobem převoditelná do jiného jazyka. Bylo nutné přesně stanovit způsob používání nonterminálů na pravých stranách pravidel - na levé straně odpovídajících si pravidel musí být nonterminály se shodnými názvy. To stejné musí platit i o nonterminálech na pravých stranách pravidel. Pokud máme např. pravidlo u jazyka Pascal nazvané <*fun*>, musí mít odpovídající pravidla jazyků C a Modula-2 shodný název. Pokud se na pravé straně pravidla u jazyka Pascal vyskytuje nonterminál s názvem <*id*>, musí se nonterminál s tímto názvem vyskytovat i na pravých stranách jazyků C a Modula-2. A naopak, pokud se nonterminál s názvem např. <*id_id*> na pravé straně pravidla jazyka Pascal nevyskytuje, nesmí se objevit ani na pravých stranách jazyků C a Modula-2.

Možné vylepšení své práce vidím v implementaci sémantického analyzátoru. Díky němu by některá gramatická omezení mohla být zjednodušena nebo zcela zrušena.

Použitá literatura

[1] Meduna, A.: Automata and Languages: Theory and Applications. London, Springer 2000

[2] Meduna, A.: Two-Way Metalinear PC Grammar Systems and Their Descriptional Komplexity. Acta Cybernetica 2003

[3] Salomaa, A.: Formal Languages. New York, Academic Press 1973.

[4] http://cs.wikipedia.org/

[5] Modula-2 Tutorial, 13.5.2008. Dokument dostupný na URL http://www.modula2.org/tutor/

[6] Parent, J.: Learning Modula2 through exercises, 13.5.2008. Dokument dostupný na URL http://parallel.vub.ac.be/~johan/Modula-2/New2002_2003/

[7] Donnelly, C., Stallman, R.: Bison - The YACC-compatible Parser Generátor, November 1995, 13.5.2008. Dokument dostupný na URL http://dinosaur.compilertools.net/bison/index.html

[8] Herout, P.: Učebnice jazyka C. Vydání III., České Budějovice, KOPP 2001

[9] Kvoch, M.: Programování v Turbo Pascalu 7.0. Výdání I., České Budějovice, KOPP 2000

[10] Honzík, J.: Technologie programování. Vydání I., VUT fakulta elektrotechnická, Nakladatelství VUT v Brně 1991

Seznam příloh