VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

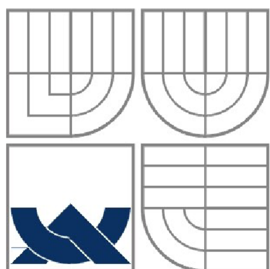
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DEMONSTRAČNÍ PROGRAM VÝPOČTU ŽIVÝCH A MRTVÝCH PROMĚNNÝCH

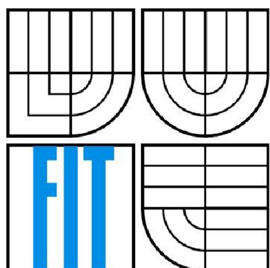
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE PETRA PAVLAČIČOVÁ
AUTHOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DEMONSTRAČNÍ PROGRAM VÝPOČTU ŽIVÝCH A MRTVÝCH PROMĚNNÝCH

DEMONSTRATION PROGRAM OF COMPUTATION OF LIVE AND DEAD VARIABLES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE PETRA PAVLAČIČOVÁ
AUTHOR

VEDOUCÍ PRÁCE Ing. LUKÁŠ ROMAN, Ph.D.
SUPERVISOR

BRNO 2009

Abstrakt

Demonstrační program výpočtu živých a mrtvých proměnných slouží jako vizuální pomůcka při výuce optimalimalizace cílového kódu označovaním živých a mrtvých promněnných ze vstupního řetezce, kterým je matematický výraz s použitím syntaktické analýzy, práce s registry a následné generace optimalizovaného kódu v asembleru.

Abstract

Demonstration program of computation of live and dead variables is visual utility for learning about optimalization of target code with tagging live and dead variables from input string, which is mathematic phrase with using syntactic analysis, work with registers and generation of target code in assembler.

Klíčová slova

živé a mrtvé promněnné, syntaktická analýza, precedenční tabulka, optimalizace, generace kódu, registry, asembler

Keywords

live and dead variables, syntactic analysis, precedence table, optimalization, generation code, registers, assembler

Citace

Petra Pavlačičová: Demonstrační program výpočtu živých a mrtvých proměnných, bakalářská práce, Brno, FIT VUT v Brně, 2009

Demonstrační program výpočtu živých a mrtvých preměnných

Prohlášení

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

……………………

Petra Pavlačičová

22.4.2009

Poděkování

Chtěla bych poděkovat svému vedoucímu Ing. Romanovi Lukášovi, Ph.D. za čas a pomoc při psaní bakalářské práce.

© Petra Pavlačičová, 2009

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah 1
1 Úvod 3
2 Základné definície 4
3 Prekladač a jeho časti 6
3.1 Lexikálna analýza 7
3.1.1 Základné pojmy 7
3.1.2 Činnosť lexikálneho analyzátora 8
3.2 Syntaktická analýza 10
3.2.1 Činnosť syntaktickej analýzy 11
3.2.2 Syntaktická analýza zdola nahor 11
3.3 Sémantická analýza 14
3.4 Generovanie intermediárneho kódu 14
3.5 Optimalizácia 14
3.5.1 Druhy optimalizácií 15
3.5.2 Optimalizačné metódy 15
3.6 Generovanie cieľového kódu 16
3.6.1 Typy generovania cieľového kódu 16
3.6.1.1 Slepé generovanie 16
3.6.1.2 Kontextové generovanie 17
4 Analýza požiadaviek a návrh aplikácie 26
4.1 Vývojové prostredie 26
4.1.1 Microsoft Visual C++ 26
4.1.2 MFC 26
4.2 Požiadavky na aplikáciu 27
4.2.1 Rozhranie 27
5 Implementácia programu 29
5.1 Trieda ParsingFunction 29
5.1.1 Funkcia Lex 29
5.1.2 Funkcia CreateTZB 29
5.2 Trieda LiveAndDeadVariablesDlg 30
5.2.1 Funkcia OnPaint 31
5.3 Trieda CliveAndDeadVariablesApp 31
5.4 Viacjazyčnosť 32
5.5 Proces prekladu 32

6 Návod na použitie **37**

6.1 Postup 38

7 Záver **39**

1 Úvod

Cieľom tejto práce bolo vytvoriť program, ktorý bude názorne ukazovať, ako to vyzerá v priebehu syntaktickej analýzy, práce s registrami a následného generovania optimalizovaného kódu.

Tento program by mohli využiť študenti nielen predmetu Formálne jazyky a prekladače, kde sa táto téma vyučuje a venuje sa jej celá jedna vyučovacia hodina. Na programe by si študenti mohli otestovať svoje vedomosti, či už ako kontrola správnych výsledkov, alebo pochopenie princípu.

V tejto správe som sa snažila zoradiť kapitoly tak, aby sa čitateľ dozvedel všetky potrebné informácie a pojmy k danej téme na začiatku práce, čiže ozrejmiť veci skôr ako o nich začnem písať.

V kapitole Základné definície uvádzam výber z definíc pojmov, ktoré budem v práci využívať. Tieto pojmy som sa snažila napísať v čo najjednoduchších formuláciách, pretože je to len na ozrejmenie, čo ktorý pojem znamená.

V nasledujúcej kapitole sa venujem téme prekladačov a tomu, z čoho sa skladajú. V mojom prípade nevyužívam všetky časti prekladača, ale vypísala a ozrejmila som každú časť. Pri tých, ktoré využívam menej alebo vôbec, som spomenula ako fungujú a na čo je ich potreba. Ostatným častiam som sa venovala obšírnejšie. Najviac rozpísaná je časť syntaktickej analýzy, optimalizácia a generovanie kódu, pretože tie sú pre danú problematiku najzaujímavejšie.

Ďalej nasleduje kapitola Analýza požiadaviek a návrh aplikácie, v ktorej sa zaoberám tým, v čom bol projekt implementovaný a aké boli moje kritériá pre vznik aplikácie a popis rozhrania.

V kapitole Implementácia sa venujem priamo jednotlivým triedam a funkciám tak, ako boli napísané a s akými problémami som sa stretla pri ich vytváraní.

Dôležitou kapitolou je aj kapitola Návod na použitie, kde popisujem, ako treba program používať a vysvetľujem čo je kde zobrazené.

2 Základné definície

Pre porozumenie ďalšieho textu, by sme si mali na začiatok ozrejmiť zopár základných definíc. *Abecedu* môžeme opísať ako konečnú a neprázdnu možnou obsahujúcu elementy a tieto elementy sa nazývajú *symboly*. Abecedu označujeme znakom Σ. Napríklad abeceda môjho mena by sa označovala takto: $\Sigma = \{p, e, t, r, a\}$ čiže pozostáva z množiny 5 symbolov.

Reťazec je konečná postupnosť symbolov nad abecedou. Prázdny reťazec ε, čiže ten, ktorý neobsahuje žiadny symbol je tiež reťazec nad danou abecedou.

Nasledujúcim dôležitým pojmom je pojem *jazyk*. Ak Σ^* značí množinu všetkých reťazcov nad abecedou Σ. Každá podmnožina množiny všetkých reťazcov nad abecedou je jazyk nad touto abecedou. Jazyky môžu byť konečné a nekonečné, kde konečné obsahujú konečný počet reťazcov a nekonečné majú nekonečný počet reťazcov.

Ďalej si ozrejmíme, čo sú to regulárne výrazy a jazyky.

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

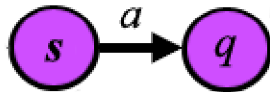
- Ø je Regulárny výraz značiaci prázdnu množinu (prázdny jazyk)
- ε je regulárny výraz značiaci jazyk {ε}
- *a*, kde $a \in \Sigma$ je regulárny výraz značiaci jazyk $\{a\}$
- Ak *r* a *s* sú regulárne jazyky značiace po rade jazyky L_r a L_s, potom
  - $(r.s)$ je Regulárny výraz značiaci jazyk $L=L_rL_s$
  - $(r+s)$ je Regulárny výraz značiaci jazyk $L= L_r \cup L_s$
  - (r^*) je Regulárny výraz značiaci jazyk $L=L_r^*$

Teraz už vieme čo je to regulárny výraz, tak si môžeme povedať čo je to *Regulárny jazyk*. Zjednodušene by sme mohli povedať, že každý regulárny výraz značí regulárny jazyk. Zadefinujeme si to teda tak, že *L* je jazyk a *L* je regulárny jazyk, ak k nemu existuje regulárny vraz r, ktorý vlastne tento jazyk značí.

Konečný automat je vlastne najjednoduchší model, ktorý obsahuje nejaké stavy, tieto stavy majú konečný počet to znamená že vieme koľko ich bude, odtiaľ aj názov Konečný automat. Taktiež má aj konečný počet výpočtových pravidiel.

Konečný automat je pätica, ktorá sa označuje ako $M=(Q, \Sigma, R, s, F)$, kde Q je konečná množina stavov, Σ značí vstupnú abecedu, R je nejaká konečná množina, ktorá značí že ak z nejakého stavu pri prečítaní symbolu z abecedy, urobí prechod do stavu u ďalšieho. Ak je tento symbol rovný ε to znamená že nebol prečítaný symbol na vstupe. Počiatočný stav označujeme

písmenom s, koncový stav písmenom f ktorý patri do množiny koncových stavov a q označuje ľubovolný iný stav patriaci do Q, čiže množine stavov. Na obrázku 2.1 môžeme vidieť aj grafickú reprezentáciu konečného automatu, alebo jeho časti. Na obrázku je znázornené pravidlo $pa \rightarrow q \in R$ čiže z počiatočného stavu sme načítali symbol a prešli do ďalšieho stavu.



2.1 Prechod do ďalšieho stavu

Deterministický konečný automat je konečný automat, ktorý má naproti konečnému automatu jednu vlastnosť a to že z každej konfigurácie sa dostane prijatím jedného symbolu práve do maximálne jednej ďalšej.

Bezkontextová gramatika

Definícia: Bezkontextová gramatika (BKG) je štvorica $G=(N, T, P, S)$, kde:

- N je abeceda nonterminálov
- T je abeceda terminálov, pričom $N \cap T = \emptyset$
- P je konečná množina pravidiel tvaru $A \rightarrow x$; kde $A \in N, x \in (N \cup T)^*$
- $S \in N$ je počiatočný nonterminál.

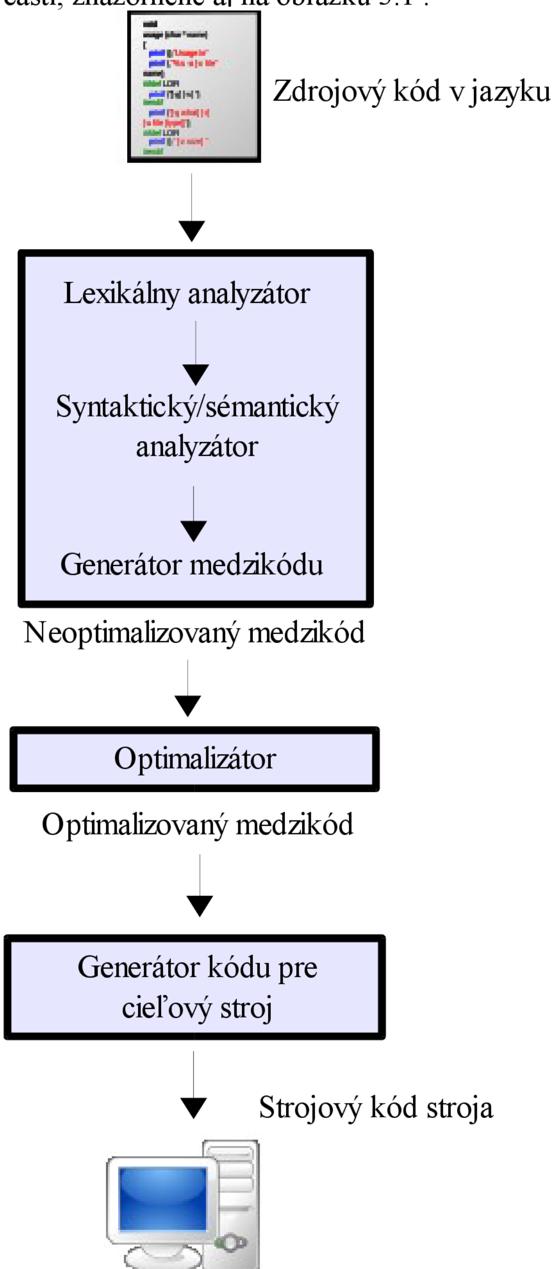
3 Prekladač a jeho časti

Prekladač[1] je program, ktorý vezme zdrojový program a prevedie ho do cieľového programu ktorý je ekvivalentom zdrojového programu.

Najrozšírenejšími zdrojovými jazykmi sú napríklad programovacie jazyky ako Modula-2, Pascal, alebo C, cieľovým jazykom je pre nás strojový kód alebo jazyk assembleru nejakého počítača.

Prekladač má z formálneho hľadiska tieto časti, znázornené aj na obrázku 3.1 :

- Lexikálna analýza
- Syntaktická analýza
- Sémantická analýza
- Generátor medzikódu
- Optimalizátor
- Generátor cieľového kódu



3.1 Obrázok znázorňuje jednotlivé časti prekladača

[1] Pri spracovaní bolo čerpané z týchto študijných materiálov [1]

3.1 Lexikálna analýza

Prvá fáza sa nazýva lexikálna analýza[2]. Jej úlohou je čítať zo zdrojového programy znaky, identifikuje ich ako lexikálne symboly a ak je postupnosť lexikálnych symbolov v nej každý symbol predstavuje logickú postupnosť ako napríklad identifikátor „premenná“ alebo operátor „+“ a iné. Postupnosť znakov tvoriacich symbol sa nazýva lexém.

Po lexikálnej analýze znakov napr. V tomto priraďovacom príkaze

```
a:=b*30-c
```

by sme mali tieto lexikálne jednotky:

1. identifikátor `a`
2. symbol priradenia `:=`
3. identifikátor `b`
4. operátor `*`
5. číslo `30`
6. operátor `-`
7. identifikátor `c`

Symboly sú reprezentované ako dvojice, ktoré obsahujú druh symbolu a hodnotu. Ako druh symbolu berieme lexikálne jednotky ako napríklad identifikátor, číslo, reťazec. V tejto dvojici môže byt druhá časť prázdna. Nasledujúca postupnosť by teda mohla byť výstupom lexikálnej analýzy:

```
<id, a> <:=> <id,b> <*> <num,30> <-> <id,c>
```

Prázdne riadky a medzery lexikálna analýza odstraňuje.

3.1.1 Základné pojmy

Lexém je postupnosť znakov zdrojového programu, ktorá odpovedá vzoru pre konkrétny symbol.

Napr. $x = 25$

x je lexém pre symbol identifikátora.

Symboly považujeme za terminálne symboly gramatiky zdrojového jazyka. Lexémy odpovedajúce vzorom pre symboly predstavujú reťazce znakov zdrojového programu, ktoré môžeme považovať za jedinú lexikálnu jednotku.

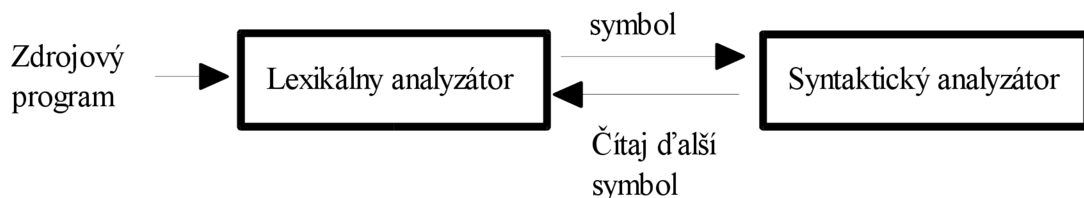
---

[2] Táto téma bola spracovaná podľa týchto materiálov [1]

Vzor je pravidlo popisujúce množinu lexémov, ktoré môžu predstavovať v zdrojovom programe konkrétny symbol.

3.1.2 Činnosť lexikálneho analyzátora

Hlavnou úlohou lexikálneho analyzátora[3] je čítať znaky zo vstupu a na svoj výstup dávať symboly, ktoré ďalej používa syntaktický analyzátor. Táto interakcia, graficky zhrnutá na obrázku 3.2, sa bežne implementuje tak, že lexikálny analyzátor vytvoríme ako podprogram alebo koprogram syntaktického analyzátoru. Po prijatí príkazu „daj ďalší symbol" od syntaktického analyzátoru číta lexikálny analyzátor vstupné znaky až dovtedy, kým môže identifikovať ďalší symbol.

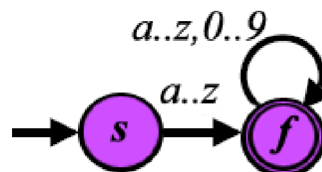


3.2 Vzťah medzi lexikálnym a syntaktickým analyzátorom

Ďalšou činnosťou lexikálneho analyzátora je odstraňovanie poznámok a odsadzovačov (medzier, tabulátorov a koncov riadkov) zo zdrojového programu.

Lexikálny analyzátor identifikuje jednotlivé lexémy pomocou deterministického konečného automatu.

Pre identifikátor vyzerá konečný automat ako na obrázku 3.3. Znamená to, že ak vstupný znak ktorý bol načítaný je písmeno, prejde do ďalšieho stavu, kde číta znaky ktoré sú buď čísla alebo písmená a to dovtedy, dokiaľ sa na vstupe neobjaví nejaký iný znak.

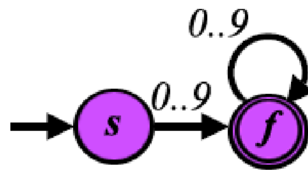


3.3 Konečný automat pre identifikátor

---

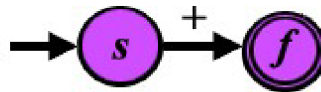
[3] Pri spracovaní bolo čerpané z týchto študijných materiálov [1]

To isté platí aj pre celé číslo, znázornené na obrázku 3.4. Tam je len podmienka, že ak počiatočný stav, čiže vstupný znak je číslo, musí byť číslom aj bezprostredne za vstupným znakom.

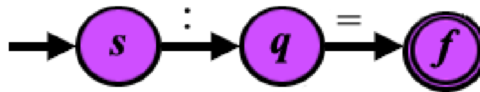


3.4 Konečný automat pre číslo

Obdobný prípad je aj pri rozpoznávaní operátorov (viď obrázok 3.5) a operátoru priradenia, kde za sebou musia nasledovať práve 2 znaky a to „:“ a „=“ ako na obrázku 3.6..



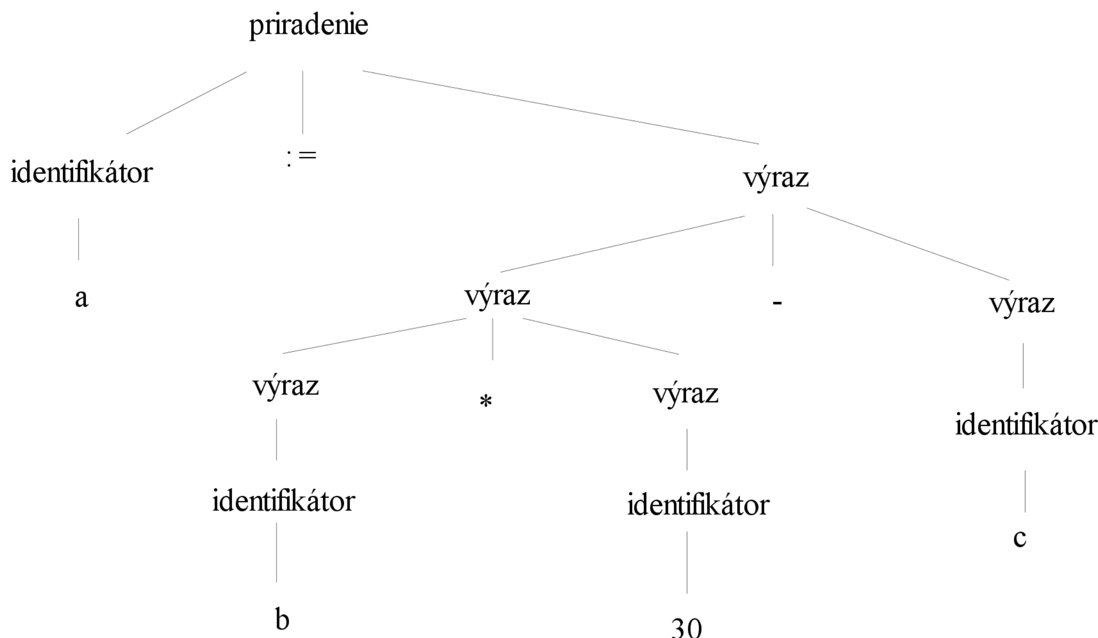
3.5 Konečný automat pre operátor plus



3.6 Konečný automat pre operátor priradenie

3.2 Syntaktická analýza

Syntaktická analýza[4] spočíva v zostavovaní lexikálnych jednotiek zo zdrojového programu do gramatických fráz, ktoré prekladač používa pre syntézu vstupu. Gramatické frázy zdrojového programu sa obvykle reprezentujú derivačným stromom obdobným stromu na obrázku 3.7.



3.7 Derivačný strom

Vo výraze `b*30-c` je fráza `b*30` logickou jednotkou, pretože podľa priority operátorov je násobenie silnejšie ako odčítanie, čiže sa musí vyhodnotiť skôr.

Hierarchická štruktúra programu sa obyčajne vyjadruje pomocou rekurzívnych pravidiel zapísaných vo forme bezkontextovej gramatiky. Napríklad pre definíciu časti výrazu môžeme mať nasledujúce pravidlá:

```
(1)  výraz    →   identifikátor
(2)  výraz    →   číslo
(3)  výraz    →   výraz + výraz
(4)  výraz    →   výraz * výraz
(5)  výraz    →   (výraz)
```

Pravidlá (1) a (2) sú (nerekurzívne) základné pravidlá, zatiaľ čo (3)-(5) definujú výraz pomocou operátorov aplikovaných na iné výrazy. Podľa pravidla (1) sú teda `b` a `c` výrazy. Podľa pravidla (2) je 30 výraz, zatiaľ čo z pravidla (4) môžeme najprv odvodiť, že `b*30` je výraz a konečne z pravidla (5) taktiež `b*30-c` je výraz.

[4] Pri spracovaní tejto témy bolo čerpané z týchto študijných materiálov [1]

3.2.1 Činnosť syntaktickej analýzy

Počas syntaktickej analýzy sa prekladač snaží zistiť, či zdrojový text tvorí vetu odpovedajúcu gramatike prekladaného jazyka. K tomu využíva postupnosť lexikálnych symbolov získanú ako výsledok lexikálnej analýzy. Pokiaľ text obsahuje nejaké chyby, prekladač ich nahlási a obvykle urobí určité zotavenie tak, aby pri výskyte chýb mohol pokračovať ďalej v činnosti a odhaliť prípadné ďalšie chyby.

Pri implementácii prekladača sa zvykne používať jeden z dvoch základných prístupov – prekladu zhora dolu alebo zdola hore. Tieto názvy odpovedajú postupu pri vytváraní derivačného stromu, pri preklade zhora dole vychádzame zo štartovacieho symbolu gramatiky a snažíme sa postupnou expanziou nonterminálnych symbolov dospieť až k terminálnym symbolom odpovedajúcim postupnosti lexikálnych symbolov na vstupe, pri preklade zdola nahor sa naopak snažíme postupnosť terminálnych symbolov zo vstupu redukovať až na štartovací nonterminál. Uvedeným dvom prístupom odpovedajú tiež 2 základné triedy gramatík, konkrétne LL a LR gramatiky, ktoré popisujú určité dostatočne veľké podmnožiny bezkonextových jazykov.

Výstupom analyzátoru bude určitá reprezentácia zdrojového textu, ktorá bude obsahovať len informácie podstatné pre ďalší priebeh prekladu. Touto reprezentáciou môže byť napríklad derivačný strom alebo v všeobecne určitá postupnosť akcií, ktoré vytvárajú vnútornú reprezentáciu štruktúry zdrojového programu a uchovávajú informácie o sémantike týchto štruktúr.

3.2.2 Syntaktická analýza zdola nahor

V tejto časti sa budem zaoberať len syntaktickou analýzou zdola nahor. Znamená to, že začína od celého výrazu a snaží sa prejsť s použitím pravidiel až k počiatočnému symbolu.

Na príklade by som tu chcela ozrejmiť a aj názorne ukázať, čo sa v priebehu syntaktickej analýzy zdola nahor deje a aké sú logické postupy pri práci s precedenčnou tabuľkou.

Ako prvé by sme si mali zadefinovať gramatiku. Takže máme gramatiku *G*=(*N, T, P, E*), kde:

- *N*={***E***} (N značí neterminály)
- *T*={+, -, *, /, **i**, (,)} (T značí terminály)
- *P*={**1:E→E+E, 2:E→E-E, 3:E→E*E, 4:E→E/E, 5:E→(E), 6:E→i**} (P sú pravidlá)

Precedenčná tabuľka je podľa zadanej gramatiky skonštruovaná tabuľka, ktorá nám ukazuje vzťah vždy medzi 2 terminálnymi symbolmi. Napríklad znamienko + má väčšiu prioritu ako znamienko *, to znamená že do tabuľky zapíšem znak "menšítko" (+<*).

Najväčšiu prioritu má násobenie a delenie, kde tieto dve operácie sú ľavoasociatívne, to znamená, že prednosť má medzi 2 operátormi s rovnakou prioritou vždy tá naľavo. Ďalej sú to operácie sčítanie a odčítanie, ktoré sú tiež ľavoasociatívne.

Precedenčnú tabuľku vytvoríme tak, že si riadky a stĺpce označíme terminálnymi symbolmi. Do záhlaví riadkov aj stĺpcov dáme vlastne tie isté terminálne symboly, ale v každom z nich budú značiť niečo iné. Stĺpce označujú vstupný reťazec a riadky označujú symboly na zásobníku. Okrem terminálnych symbolov obsahujú riadky a stĺpce ešte jeden symbol a to symbol $, ktorý vo vstupnom reťazci indikuje koniec reťazca a na zásobníku označuje dno zásobníka, čiže prvý znak ktorý je vložený na zásobník je práve znak $.

Máme vytvorené záhlavie tabuľky. Teraz ju treba vyplniť. Vypĺňovať ju budeme tak, že vezmeme symbol zo zásobníka a symbol zo vstupu a tam, kde sa pretínajú, vložíme znak <, >, = alebo prázdne políčko.

Vytvorená precedenčná tabuľka je na obrázku 3.8.

	+	-	*	/	(	I	)	$
+	>	>	<	<	<	<	>	>
-	>	>	<	<	<	<	>	>
*	>	>	>	>	<	<	>	>
/	>	>	>	>	<	<	>	>
(	<	<	<	<	<	<	=	
I	>	>	>	>			>	>
)	>	>	>	>			>	>
$	<	<	<	<	<	<		

3.8 Precedenčná tabuľka

Ak je precedenčná tabuľka hotová, môžeme pokračovať k samotnému algoritmu precedenčnej analýzy:

- vlož na zásobník symbol **$**
- Hlavný cyklus:
    - Nech **a** je aktuálny vstupný symbol, **b** je najvrchnejší terminálny symbol na zásobníku. Podľa obsahu políčka precedenčnej tabuľky na súradniciach [**b**,**a**] rozhodni:

      = :prečítaj symbol a zo vstupu a daj ho na vrchol zásobníku

      < : nájdi na zásobníku najvrchnejší terminálny symbol **b**. Hneď za tento symbol vlož do zásobníku symbol <. Prečítaj symbol **a** zo vstupu a daj ho na vrchol zásobníku.

> : nájdi najvrchnejší symbol <. Medzi týmto symbolom a vrcholom zásobníka nájdi pravú stranu istého pravidla r. Odstráň túto časť zo zásobníka vrátane symbolu <.Vlož na zásobník ľavú stranu pravidla **r**

prázdne políčko : značí syntaktickú chybu vo vstupnom reťazci.

- Ak sa **a**=$ a **b**=$ syntaktická analýza prebehla v poriadku

V tabuľke na obrázku 3.9 môžeme vidieť postup ako syntaktická analýza pracuje. Vidíme čo je na zásobníku, čo je na vstupe a aký operátor je medzi nimi.

Zásobník	Operátor	Vstup	Redukcia podľa pravidla
$	<	b+c*(d-5)/e$	
$<b	>	+c*(d-5)/e$	E=i
$E	<	+c*(d-5)/e$	
$<E+	<	c*(d-5)/e$	
$<E+<c	>	*(d-5)/e$	E=i
$<E+E	<	*(d-5)/e$	
$<E+<E*	<	(d-5)/e$	
$<E+<E*<(	<	d-5)/e$	
$<E+<E*<(<d	>	-5)/e$	E=i
$<E+<E*<(E	<	-5)/e$	
$<E+<E*<(E-	<	5)/e$	
$<E+<E*<(E-<5	>	)/e$	E=i
$<E+<E*<(E-E	>	)/e$	E=E-E
$<E+<E*<(E	=	)/e$	
$<E+<E*<(E)	>	/e$	E=(E)
$<E+<E*E	>	/e$	E=E*E
$<E+E	<	/e$	
$<E+<E/	<	e$	
$<E+<E/<e	>	$	E=i
$<E+<E/E	>	$	E=E/E
$<E+E	>	$	E=E+E
$E	>	$	Úspech :-)

3.9 Syntaktická analýza

3.3 Sémantická analýza

V priebehu sémantickej analýzy[5] sa vykonávajú niektoré kontroly, zaisťujúce správnosť programu z hľadiska väzieb, ktoré nejdú vykonávať v rámci syntaktickej analýzy. Táto fáza spracováva najmä informácie, ktoré sú uvedené v deklaráciách, ukladá ich do vnútorných dátových štruktúr a na ich základe vykonáva sémantickú kontrolu príkazov a výrazov v programe. Dôležitá zložka sémantickej analýzy je typová kontrola. Kompilátor tu kontroluje, či všetky operátory majú operandy povolené špecifikáciou zdrojového jazyka.

3.4 Generovanie intermediárneho kódu

Po ukončení syntaktickej a sémantickej analýzy, generujú niektoré prekladače explicitnú intermediárnu reprezentáciu zdrojového programu (medzikód)[6].

Intermediárnu reprezentáciu môžeme považovať za program pre nejaký abstraktný počítač. Táto reprezentácia by mala mať 2 dôležité vlastnosti: mala by byť jednoduchá pre vytváranie a jednoduchá pre preklad do tvaru cieľového programu.

Intermediárny kód slúži ako podklad pre optimalizácie a generovanie cieľového kódu. Môže však byť aj konečným produktom v interpretačnom prekladači, ktorý vygenerovaný medzikód priamo vykonáva.

Intermediárne reprezentácie môžu mať rôzne formy. Napríklad trojadresný kód sa podobá jazyku symbolických inštrukcií pre počítač, ktorého každé miesto v pamäti môže slúžiť ako register. Trojadresný kód sa skladá z postupnosti inštrukcií s najviac troma operandami.

3.5 Optimalizácia

Termín optimalizácia[7] sa v tejto súvislosti tradične používa pre určité vylepšenie cieľového programu, bez ktorých by bol výsledok pomalejší pri výpočte, alebo mal väčšie nároky na pamäť, prípadne oboje. Proces optimalizácie obvykle nezahŕňa transformáciu zo zdrojového programu, ktoré by menili programátorom stanovený algoritmus riešenia alebo jeho implementáciu v zdrojovom jazyku.

Základné kritéria podľa ktorých sa posudzuje úspešnosť optimalizácie sú:

- dĺžka generovania cieľového programu,

---

[5] Pri spracovaní bolo čerpané z týchto študijných materiálov [1]

[6] Na spracovanie tejto časti sú použité tieto materiály [1]

[7] Túto tému som spracovala pomocou týchto študijných materiálov [1]

- rýchlosť výpočtu,
- požadovaná pamäť pre údaje.

3.5.1 Druhy optimalizácií

Optimalizácia programu býva veľmi nákladnou činnosťou ako z hľadiska predĺženia celkovej doby prekladu programu, tak z hľadiska práce a úsilia pri realizácii optimalizujúceho prekladača.

Uskutočnením určitej optimalizácie umožňuje obvykle vykonať ďalšiu optimalizáciu toho istého alebo iného typu a celý proces má potom iteračný charakter. Nákladná je taktiež analýza toku údajov.

Niekedy sa preto odlišujú:

1.) *Lokálne optimalizácie* - v rámci základného bloku
- prebiehajú len nad sekvenciami operácií (bez skokov a vetvenia)

2.) *Globálne optimalizácie* - v rámci niekoľkých blokov
- využívajú kontext celého programu a vyžadujú globálnu analýzu toku údajov

Nie vždy je najdôležitejšia rýchlosť výpočtu. Na malých počítačoch môže byť pamäťová náročnosť programu rovnako dôležitá ako, ak nie dôležitejšia.

Delia sa teda na ďalšie dva druhy optimalizácie:

1.) Optimalizácia rýchlosti

2.) Optimalizácia veľkosti

3.5.2 Optimalizačné metódy

Máme niekoľko optimalizačných metód:

1.) Zabalenie konštanty

- je metóda, kde sa z viacerých výrazov, v ktorých sa nachádza priradenie konštanty (čísla) do premennej zredukujú na jeden príkaz v ktorom sa do výslednej premennej priradí výsledok tejto operácie s nahradenými premennými konštantou.

2.) Šírenie konštanty

- je metóda, ktorou sa eliminujú kroky v tom prípade, ak sa do nejakej premennej priradila konštanta, do ďalšej premennej sa priradila premenná s touto konštantou a v ďalšom kroku sa táto premenná priradila do ďalšej premennej. Sú tam teda

2 nadbytočné kroky, ktoré odstránime priradením konštanty až do výslednej premennej.

3.) *Kopírovanie premennej*

- táto metóda je principiálne zhodná s predchádzajúcou metódou, ale u tejto sa nepriraďuje konštanta, ale premenná. Čiže do výslednej premennej sa priradí premenná, ktorá sa mala priradiť v prvom kroku.

4.) *Výrazové invarianty v cykle*

- ak sa v nejakom cykle nachádza výraz ktorého obsah sa počas cyklu nemení, nahradí sa jeho obsah pred začiatkom cyklu premennou, čiže sa nebudú tie isté operácie vykonávať znova a znova, ale sa prevedú len raz pred začatím cyklu.

5.) *Rozbalenie cyklu*

- je metóda, kde sa považuje za optimálne zadať obsah cyklu ako pod sebou nasledujúce príkazy s pevne zadanou premennou, ktorá bola predtým dynamicky inkrementovaná alebo dekrementovaná.

6.) *Eliminácia mŕtveho kódu*

- odstránenie častí kódu, ktoré boli onačené ako mŕtvy kód:
  a) Taký ktorý sa nikdy nevykoná (napr. Nezmyselná podmienka)
  b) Ktorý nerobí nič užitočné (napr. Priradenie premennej do seba samej)

3.6 Generovanie cieľového kódu

Vstupom generátoru cieľového kódu[8] je postupnosť príkazov vo vnútornom jazyku prekladača, výstupom je potom výsledný produkt prekladu – cieľový program ekvivalentný k zdrojovému programu.

3.6.1 Typy generovania cieľového kódu

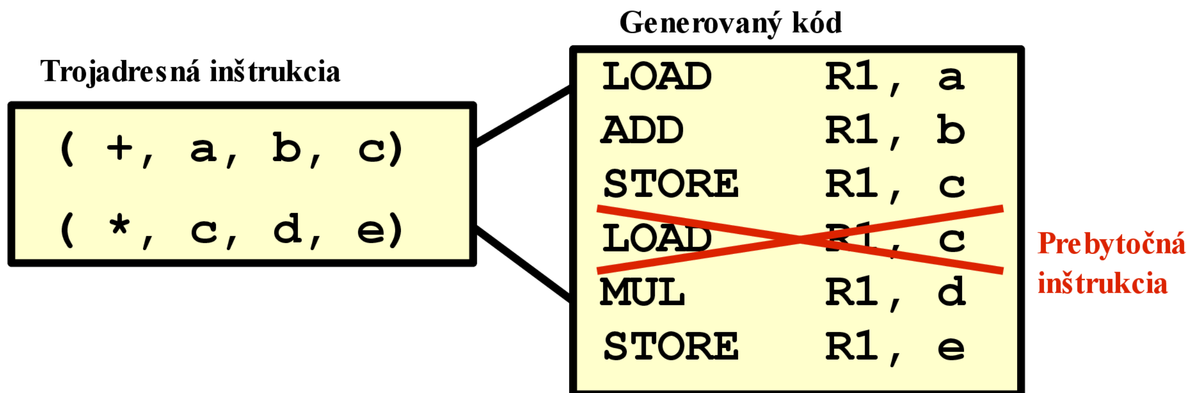
3.6.1.1 Slepé generovanie

Pre každú trojadresnú inštrukciu existuje procedúra, ktorá generuje príslušný cieľový kód

[8] Pri spracovaní bolo čerpané z týchto študijných materiálov [1]

Hlavná nevýhoda slepého generovania je tá, že každá trojadresná inštrukcia je mimo kontext ostatných inštrukcií bloku, dochádza teda k prebytočnému načítaniu a ukladaniu premenných.

Čiže vždy sú pre každú inštrukciu vygenerované 3 inštrukcie v cieľovom kóde. Kód potom nieje optimalizovaný a nachádzajú sa v ňom zbytočné úkony.



3.10 Príklad slepého generovania

Na obrázku 3.10 môžeme vidieť, aký kód sa z jednotlivých inštrukcií generuje, a tiež, sa nám vždy vygeneruje kód ktorý má dĺžku trikrát počet trojadresných inštrukcií.

3.6.1.2 Kontextové generovanie

Kontextové generovanie je založené na zredukovaní nepotrebných načítaní a ukladaní premenných, čiže minimalizácia počtu načítania a ukladania medzi registrami a pamäťou.

Obecne platí, že ak je hodnota premennej v registre a bude skoro použitá, ponecháme ju v registri.

Základné pojmy:

Tabuľka základného bloku (TZB) – udržiava informácie, ktoré premenné sú neskor potrebné a kde

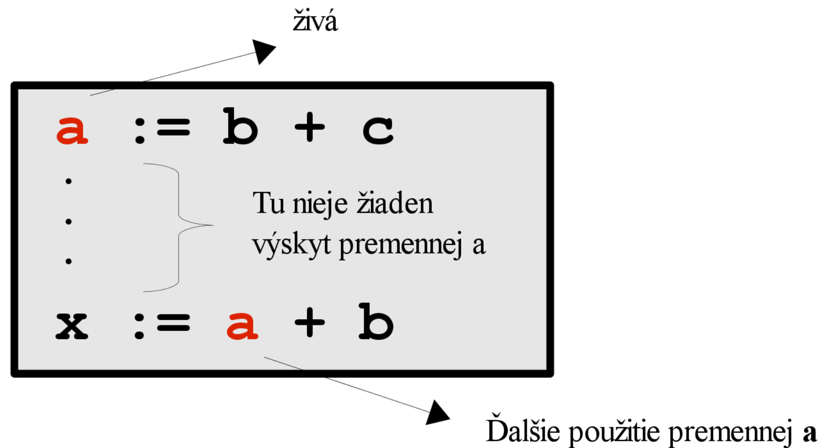
Tabuľka registrov (TR) – udržiava informácie, ktoré registry sú použité a čo v nich je

Tabuľka adries (TA) – udržiava informácie o tom, kde je uložená aktuálna hodnota danej premennej

Tabuľka symbolov (TS) – udržiava informácie o pomenovaných objektoch (napr. Užívateľské typy, procedúry, funkcie, premenné)

Analýza v základnom bloku:

V rámci základného bloku by sme potrebovali vedieť, či je premenná živá, to znamená že je použitá v bloku o niečo neskôr.



3.11 Znázornenie živej premennej

Na príklade 3.11 je znázornené, ako vyzerá v bloku živá premenná.

Živé premenné môžeme efektívne detektovať aplikáciou spätného algoritmu, tzn. že inštrukcie sa čítajú od konca bloku smerom k začiatku.

Tabuľka symbolov

Tabuľka symbolov obsahuje všetky objekty, ktoré boli v programe deklarované, udržiava informácie o týchto objektoch, ako sú napríklad ich typy, hodnoty, veľkosť a iné. Tieto hodnoty využíva prekladač k ďalším účelom, ako sú napríklad typové kontroly, generovanie kódu..

V našom prípade ale tabuľka symbolov obsahuje iba premenné, programátorské aj pomocné, ich stav a ich ďalšie použitie v základnom bloku. Programátorské premenné získame už po lexikálnej analýze a v priebehu syntaktickej analýzy doplníme zvyšné pomocné premenné.

Tabuľku na začiatku vyplníme tak, že programátorské premenné, to znamená tie ktoré sú zadané v programe inicializujeme na začiatok na živé (live), čiže L a pomocné premenné, ktoré sme si vytvorili na pomoc označíme ako mŕtve(dead), čiže D. Všetkým premenným potom priradíme ďalšie použitie na „none“ , to znamená, že nemajú výskyt nikde.

V priebehu spätného prechádzania postupne meníme tabuľku, pretože niektoré premenné, menia svoje vlastnosti. Tak napríklad vo výraze `a:=b+c,` je premenná a mŕtva, pretože sa do nej zapíše výsledok operácie, v tomto prípade sa jej obsah prepíše a tým sa táto premenná „usmrtí“. Jej

ďalšie použitie teda už nieje nikde a do tabuľky sa zapíše „none“.Naopak premenné b a c sú živé a do tabuľky sa napíše číslo riadku v TZB na ktorom sa práve nachádzajú.

Tabuľka symbolov by mohla vyzerať ako na obrázku 3.12.

Premenná	Stav	Ďalšie použitie
a	L	3
b	L	6
t1	D	none

3.12 Tabuľka symbolov

Tabuľka základného bloku

Tabuľka základného bloku obsahuje údaje o premenných a to kde sa nachádzajú a aký je ich stav.

Tabuľku vypĺňame trojadresnými inštrukciami počas syntaktickej analýzy, vždy ak sa redukuje podľa nejakého pravidla, ktoré obsahuje operátor, tak sa táto inštrukcia skopíruje so tabuľky základného bloku a je priradená do vygenerovanej pomocnej premennej. Ako príklad si ukážeme vstupný výraz `a:=b+c*(d-5)/e`. Ak máme TZB vyplnenú inštrukciami ako na obrázku 3.13 môžeme ísť na samotné vypĺňanie zvyšku tabuľky.

Riadok	Inštrukcia	Stav	Ďalšie použitie
1	`t1:=d-5`		
2	`t2:=c*t1`		
3	`t3:=t2/e`		
4	`a:=b+t3`		

3.13 Tabuľka základného bloku

Tabuľka symbolov vyzerá na začiatku ako na obrázku 3.14.

Premenná	Stav	Ďalšie použitie
a	L	none
b	L	none
c	L	none
d	L	none
e	L	none
t1	D	none
t2	D	none
t3	D	none

3.14 Tabuľka symbolov

Postup pri vypĺňovaní tabuľky základného bloku je nasledujúci:

- Začíname odspodu tabuľky, čiže posledným riadkom. Do stavu v TZB opíšeme stav jednotlivých premenných z TS. Potom do kolonky ďalšie použitie opíšeme taktiež z TS ďalšie použitie jednotlivých premenných. Výsledok tomto kroku by mal vyzerať ako na obrázku 3.15.

Riadok	Inštrukcia	Stav	Ďalšie použitie
1	`t1:=d-5`		
2	`t2:=c*t1`		
3	`t3:=t2/e`		
4	`a:=b+t3`	a: L, b:L, t3:D	a: N, b:N, t3:N

3.15 Časť vyplnenej tabuľky základného bloku

- Ak sme vyplnili TZB, môžeme pomeniť premenné v TS podľa aktuálneho stavu. To znamená, že musíme určiť, či je premenná živá alebo mŕtva a opísať riadok na ktorom sa nachádzajú. V našom prípade je vo výraze `a:=b+t3` a mŕtve a `b` a `t3` sú živé. Opravená tabuľka symbolov po prvom kroku vyzerá ako na obrázku 3.16.

Premenná	Stav	Ďalšie použitie
a	**D**	**none**
b	**L**	**4**
c	L	none
d	L	none
e	L	none
t1	D	none
t2	D	none
t3	**L**	**4**

3.16 Časť vyplnenej tabuľky symbolov

- Tento postup opakujeme, až kým nebude tabuľka základného bloku kompletná. Výsledná tabuľka základného bloku môže vyzerať ako na obrázku 3.17 a tabuľka symbolov ako na obrázku 3.18.

Riadok	Inštrukcia	Stav	Ďalšie použitie
1	`t1:=d-5`	t1:L, d:L	t1:2, d:L
2	`t2:=c*t1`	t2:L, c:L, t1:D	t2:3, c:N, t1:D
3	`t3:=t2/e`	t3:L, t2:D ,e:D	t3:4, t2:D, e:D
4	`a:=b+t3`	a: L, b:L, t3:D	a: N, b:N, t3:N

3.17 Vyplnená tabuľka základného bloku

Premenná	Stav	Ďalšie použitie
a	D	none
b	L	4
c	L	2
d	L	1
e	L	3
t1	D	none
t2	D	none
t3	L	none

3.18 Vyplnená a pozmenená tabuľka symbolov

Tabuľka registrov

Pre rýchlejší a efektívnejší chod programu sa využíva ukladanie premenných do registrov. Prácu s týmito registrami uchováva tabuľka registrom. Obsahuje názvy dostupných registrov a ich obsah.

Na začiatku sú registry prázdne, až za behu programu sa ich obsah mení. Je nutné po každej operácii kde sa pracuje s registrami obnovovať tabuľku.

Niekedy sa do tabuľky registrov zapisuje aj že nejaký register je už obsadený pre iné účely, to som však v mojom projekte neriešila, čiže mám dostupné len voľné registry.

Tabuľka registrov môže vyzerať ako na obrázku 3.19.

Register	Obsah
R1	free
R2	a
R3	b
R4	free

3.19 Tabuľka registrov

Tabuľka adries

Ak už máme v tabuľke registrov nejaké premenné, potrebovali by sme nejak vedieť, ktoré premenné sú v registroch, ktoré v pamäti a ktoré už nemáme nikde. Na to nám slúži tabuľka adries.

Sú v nej uložené všetky premenné a miesto, odkiaľ môžeme načítať ich hodnotu. Tabuľka adries sa taktiež mení v behu programu a pri každej zmene tabuľky registrov musíme obnoviť aj tabuľku adries. Tabuľka adries je znázornená na obrázku 3.20.

Premenná	Adresa
a	R2
b	R3
c	memory
t1	nowhere
t2	nowhere

3.20 Tabuľka adries

Tabuľky už máme vytvorené, teraz si povieme ako s nimi pracovať. Aby sme do tabuľky registrov mohli niečo zapísať, musíme vedieť kam to zapísať. Čiže potrebujeme nejakú funkciu, ktorá nám vráti register, ktorý je podla určitých kritérií najvhodnejší na uchovanie hodnoty premennej. K tomu nám slúži funkcia *GetReg.*

Vo výraze `a:=b+c` nám funkcia *GetReg* vráti optimálny register pre načítanie premennej `b` a postup je vysvetlený v nasledujúcom pseudo kóde.

Begin

if b je v registri R and `b` je „dead" **and** `b` má ďalšie použitie nastavené na „none"

then return R

else

if existuje voľný register R **then** return R

else

begin

- vyber register R obsahujúci premennú, ktorá je použitá čo najneskôr
- ulož obsah R do pamäti a modifikuj Tabuľku registrov a tabuľku adries
- return R

end;

end;

Výsledný výstup programu bude optimálny kód v jazyku asembler. Pre generovanie tohto kódu sme si vytvorili ďalšiu funkciu s názvom *GenCode.* Logický postup tejto funkcie pre výraz `a:=b+c` je v nasledujúcom pseudokóde:

begin

- Zavolaj funkciu *GetReg* pre výber registru R pre premennú `b`
- **if** `b` nieje v R **then** generuj („`load R,b`")

- **if** `c` je v S then generuj („`add R,S`“)

  else generuj („`add R,c`“)
- Modifikuj TR a TA tak, aby určovali, že súčasná hodnota premennej a je v registri R
- **if** `c` je v S and `c` je „dead“ and `c` má ďalšie použitie nastavené na none, potom nastav stav premennej S v tabuľke registrov na „free“

end;

Teraz už vieme, ako všetky súvislosti s generovaním optimalizovaného kódu fungujú, tak si ukážeme aký priebeh má generovanie vstupného výrazu `a:=b+c*(d-5)/e.`

Riadok	Inštrukcia	Stav	Ďalšie použitie
1	`t1:=d-5`	t1:L, d:L	t1:2, d:L
2	`t2:=c*t1`	t2:L, c:L, t1:D	t2:3, c:N, t1:D
3	`t3:=t2/e`	t3:L, t2:D ,e:D	t3:4, t2:D, e:D
4	`a:=b+t3`	a: L, b:L, t3:D	a: N, b:N, t3:N

3.21 Stav tabuľky základného bloku po vyplnení

Register	Obsah
R1	free
R2	free
R3	free
R4	free

3.22 Počiatočný stav tabuľky registrov

Premenná	Adresa
a	memory
b	memory
c	memory
d	memory
e	memory
t1	nowhere
t2	nowhere
t3	nowhere

3.23 Počiatočný stav tabuľky adries

Na obrázkoch 3.21, 3.22, 3.23 vidíme počiatočný stav tabuliek. Druhý priechod tabuľkou základného bloku už nerobíme spätným prechádzaním, ale začneme od inštrukcie na prvom riadku.

Na prvom mieste v TZB sa nachádza inštrukcia `t1:=d-5`. Každá premenná v tomto riadku je živá. Po zavolaní funkcie GetReg nám vráti register R0. A vidíme, že premenná d je v pamäti, čiže si ju musíme najprv načítať do registru R1 a potom vykonať inštrukciu. Vo výsledku by generovaný kód pre túto inštrukciu vyzeral nasledovne :

```
LOAD R1,d
SUB  R1,#5
```

Teraz je potreba pomeniť tabuľku registrov a tabuľku adries. V R0 už máme výsledok prvého výrazu, čiže hodnotu premennej t1. A v tabuľke adries zmeníme adresu T1 na register R0. Tabuľky budú teda vyzerať ako na obrázku 3.24.

Register	Obsah
R1	t1
R2	free
R3	free
R4	free

Premenná	Adresa
a	memory
b	memory
c	memory
d	memory
e	memory
t1	R1
t2	nowhere
t3	nowhere

3.24 Tabuľka registrov a tabuľka adries po prvom kroku

Na druhom mieste v TZB je inštrukcia `t2:=c*t1.` Po zavolaní funkcie GetReg sme dostali register R1. Generovaný kód pre túto inštrukciu bude

```
LOAD R2, c
MUL  R2, R1
```

Pretože premennú t1 máme v registri R1, nemusíme jeho hodnotu načítať z pamäte, ale z registru. A keďže podľa TZB je v tomto momente stav premennej t1 udaný ako „dead“, môžeme jeho obsah z registru uvoľniť, lebo ho už nebudeme potrebovať. Znova aktualizujeme TR a TA ako na obrázku 3.25.

Register	Obsah
R1	free
R2	t2
R3	free
R4	free

Premenná	Adresa
a	memory
b	memory
c	memory
d	memory
e	memory
t1	nowhere
t2	R2
t3	nowhere

3.25 Tabuľka registrov a tabuľka adries po druhom kroku

V tabuľke základného bloku sa posunieme na 3 riadok, v ktorom sa nachádza inštrukcia `t3:=t2/e.`

Funkcia *GetReg* nám vrátila register R2, pretože hodnotu premennej t2, ktorá bola v tomto registry bude použitá v tejto inštrukcii ale podľa TZB už jej hodnotu ďalej potrebovať nebudeme. Generovaný kód je nasledujúci: `DIV R2,e.`

Aktualizujeme TR a TA, v R2 máme teraz hodnotu t3 a hodnota t2 sa už nenachádza nikde, viď obrázok 3.26.

Register	Obsah
R1	free
R2	t3
R3	free
R4	free

Premenná	Adresa
a	memory
b	memory
c	memory
d	memory
e	memory
t1	nowhere
t2	nowhere
t3	R2

3.26 Tabuľka registrov a tabuľka adries po treťom kroku

Poslednou inštrukciou v TZB je `a:=b+t3.` Funkcia *GetReg* nám vráti register R0 a generovaný kód je:

```
LOAD R1,b
ADD  R2,R1
```

A znova aktualizujeme tabuľky TR a TA, ktoré by mali potom vyzerať ako na obrázku 3.27.

Register	Obsah
R1	a
R2	free
R3	free
R4	free

Premenná	Adresa
a	R1
b	memory
c	memory
d	memory
e	memory
t1	nowhere
t2	nowhere
t3	nowhere

3.27 Výsledný tvar tabuľky registrov a tabuľky adries

A výsledný generovaný kód pre vstupný reťazec `a:=b+c*(d-5)/e` je :

```
LOAD R1, d
SUB  R1, #5
LOAD R2, c
MUL  R2, R1
DIV  R2, e
LOAD R1, b
ADD  R2, R1
STORE R2,a
```

Posledná inštrukcia sa vygenerovala preto, lebo všetky žive premenné uložíme do pamäti. A premenná a živá je, čiže si ju nahráme do pamäti.

4 Analýza požiadaviek a návrh aplikácie

V tejto časti sa zameriavam práve na vývojové prostredie ktoré som si vybrala pre písanie aplikácie a návrhu, ako by mala táto aplikácia vyzerať a čo by všetko mala demonštrovať.

4.1 Vývojové prostredie

4.1.1 Microsoft Visual C++

Microsoft Visual C++[9] je komerčné integrované vývojové prostredie (IDE) výrobok inžinierstva, ktoré Microsoft pre C, C++ a C++ / CLI programovacie jazyky. Má nástroje pre rozvoj a ladenie C++ kódu, predovšetkým kód písaný pre Microsoft Windows API, DirectX API a Microsoft .NET Framework.

4.1.2 MFC

Microsoft Foundation Class Library[10] (Microsoft Foundation Classes alebo MFC) je knižnica, ktorá zabaluje časti Windows API do ucelených C++ tried, ktoré zaisťujú použitie a plnú kompatibilitu s väčšinou platforiem OS Windows. Triedy spravujú väčšinu objektov vo Windows a teda preddefinovanú napríklad okná alebo kontrolné panely.

Knižnica MFC je jednou z najrozšírenejších a najznámejších knižníc pre vytváranie grafického rozhrania v OS Windows.
Písaná je jazykom C++ a implementuje veľa funkcií Windows 32 API.

Keď sa na trhu objavila knižnica MFC, Microsoft tým rozšíril syntax C++ o veľa makier (napr. Message Maps) pre správu správ, pomocou ktorých OS Windows komunikuje s programami (aplikáciami). Je i mnoho výnimiek, pri ktorých je potreba uchýliť sa k funkciám Win32 API. Makrá sú navrhnuté pre maximalizáciu výkonu práce s pamäťou a pre poskytnutie čitateľných štruktúr, ktoré sú následne používané aj sprievodcami pre vytváranie základných partií programu. Práca s makrami správ nahrádza virtuálne funkcie, ktoré poskytuje C++. Avšak určite nastane situácia, kde použitie virtuálnych funkcií bude nevyhnutné a kde MFC neposkytuje prijateľný ekvivalent.

---

[9] V tejto téme bolo čerpané z týchto materiálov [6]

[10] Táto časť bola spracovaná pomocou týchto materiálov[2]

MFC triedy

Sú tu uvedené iba niektoré triedy. Hlavným rysom je začiatočné písmenko C, za ním nasleduje explicitné pomenovanie triedy.

- CDC
- CFile
- CRect

4.2 Požiadavky na aplikáciu

Aplikácia by mala byť demonštračným programom, ktorý by využívali študenti, učitelia alebo tí, ktorí by sa o túto tému zaujímali, čiže by mala čo najlepšie a najzrozumiteľnejšie ukázať, ako výpočet funguje a čo sa kde deje. Na tieto požiadavky som sa teda zamerala.

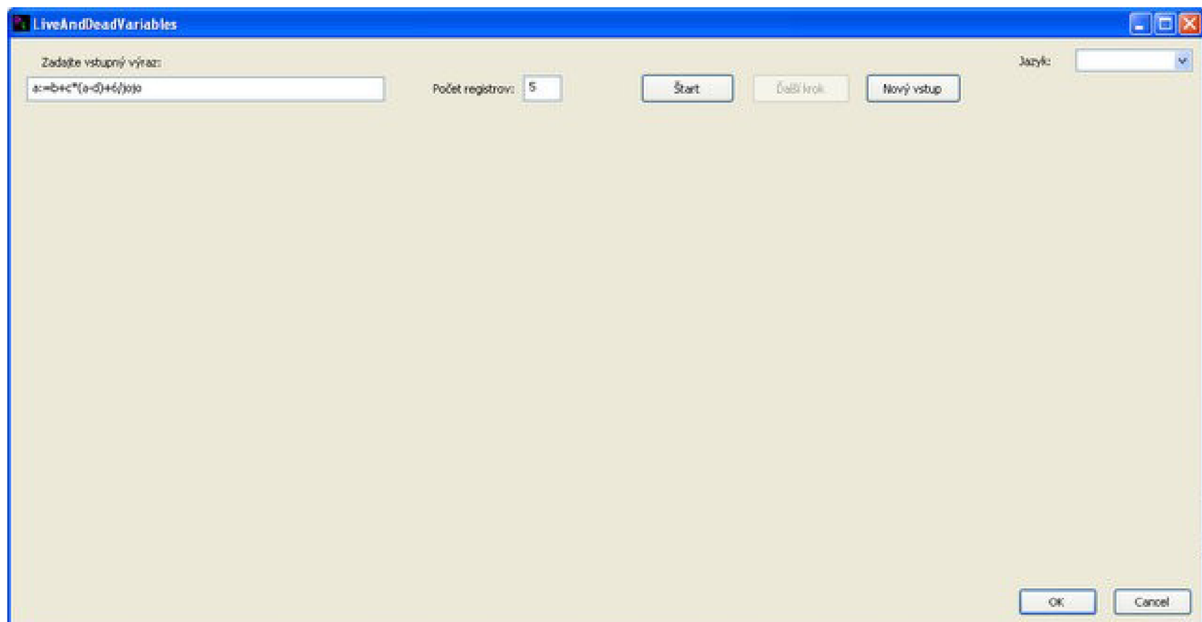
4.2.1 Rozhranie

Užívateľské rozhranie som zvolila v podobe dialógového okna, ktoré má vo vrchnej časti priestor pre vkladanie vstupných údajov, tlačidlá pre spustenie simulácie, krokovanie a vymazanie doterajších úkonov.

Taktiež má užívateľ možnosť viacjazyčného rozhrania, môže si zvoliť buď anglický alebo slovenský jazyk.

Ako vidíme na obrázku 4.1, užívateľ si môže zadať aj počet registrov, do ktorých sa neskôr budú ukladať premenné, pre lepšie znázornenie situácií, keď nemáme k dispozícii dostatočné množstvo registrov, ako i pre situácie keď ich máme priveľa. Obmedzenie kvôli veľkosti okna som nastavila na maximálne množstvo 20 registrov, keďže si myslím, že úlohou demonštračného programu je znázorniť, ako to funguje, čiže počet registrov čo má užívateľ k dispozícii je adekvátny.

Taktiež určité obmedzenia sú aj pre zadávanie vstupného reťazca, ktorý by nemal byť zbytočne dlhý alebo nemal by mať príliš dlhé názvy premenných. Optimálne množstvo počtu znakov v premennej je maximálne 6 znakov.



4.1 Dialógové okno tohto programu

5 Implementácia programu

Program bol implementovaný v jazyku C/C++ a využívala som funkcie z knižnice MFC. V tejto kapitole by som chcela opísať najdôležitejšie funkcie ktoré boli naprogramované, ich hlavný princíp a ich funkcia. Ďalej by som rada spomenula aké som mala v určitých častiach problémy a ako som ich riešila. Na koniec popíšem celý proces prekladu.

5.1 Trieda ParsingFunction

Takto je pomenovaná trieda, v ktorej sú naimplementované všetky funkcie, ktoré majú čo dočinenia so samotným projektom, to znamená výpočty a analýzy sú implementované tu.

5.1.1 Funkcia Lex

V tejto funkcii je implementovaná lexikálna analýza. Vstupné parametre má táto funkcia dve a to: vstupný reťazec a dĺžka tohto reťazca. Dĺžku potrebujeme na to, aby sme vedeli, koľko krát sa má požadovaný cyklus na kontrolu tokenov uskutočniť.

Táto funkcia nám vytvára zoznam tokenov, ktoré máju 2 atribúty a to názov a typ. Typ môže byť buď: číslo, identifikátor, znamienko, zátvorky, priradenie. Postupným priechodom tokenov kontroluje, aj či už nenastala zmena tokenu na iný typ a ak áno, doteraz načítané tokeny spojí do reťazca a uloží do štruktúry s názvom *BigToken*, priradí mu typ a následne vloží na koniec zoznamu lexémov s názvom *m_TokenList* ktorá udržuje všetky lexémy.

Táto funkcia taktiež detekuje chyby a to ak sú nesprávne znaky na vstupe, alebo sa na vstupe nenachádza symbol priradenia. Samozrejme ešte nieje schopná detektovať chyby syntaktického typu, ktoré vykonáva neskôr syntaktická analýza.

5.1.2 Funkcia CreateTZB

Toto je funkcia, ktorá má v sebe zahrnutú syntaktickú analýzu a vytváranie tabuľky základného bloku. Funkcia pracuje už s vytvoreným zoznamom všetkých za sebou idúcich lexémov. Tu už použijeme aj vopred vytvorenú precedenčnú tabuľku a tabuľku všetkých pravidiel.

Postup, ako funguje precedenčná syntaktická analýza som opisovala už v kapitole 3.2.2.K tomu potrebujeme zásobník, ktorý je implementovaný ako zoznam štruktúr.

Tu som natrafila na prvú prekážku a to, že ako budem vedieť, že ak redukujem podľa nejakého pravidla, aký má ten lexém tvar, tak aby sa mi zhodoval s tvarom v tabuľke pravidiel. Čiže ako môžem napríklad vedieť, že identifikátor xy symbolizuje znak E v tabuľke pravidiel. Tento problém som vyriešila tak, že som do štruktúry pre zásobník pridala aj položku, ktorá obsahuje podľa toho aký je to typ lexému buď to „E" pre identifikátor a číslo, alebo samotné znamienko, alebo zátvorky. Vďaka tomu budeme vždy vedieť aj aký je to typ lexému aj podľa čoho redukovať v tabuľke pravidiel.

V tejto funkcii vytváram teda tabuľku základného bloku, ktorá je implementovaná ako zoznam týchto štruktúr:

```
struct TZBrow
	{
		int row;
		Var dest;
		Var src1;
		Var src2;
		CString sign;
	};
```

```
struct Var
	{
		int otherUse;
		CString name;
		CString status;
	};
```

Štruktúra *TZBrow* je vlastne jeden riadok v tabuľke základného bloku. Obsahuje číslo riadku a inštrukciu v tvare `dest:=src1 sign src2`. Pre položky `src1,src2` a `dest` je zvlášť štruktúra, ktorá obsahuje ich názov, stav a ďalšie použitie.

V priebehu syntaktickej analýzy sa dajú odhaliť ďalšie chyby, vďaka precedenčnej tabuľke. Ak je v precedenčnej tabuľke prázdne políčko, značí to syntaktickú chybu.

5.2 Trieda LiveAndDeadVariablesDlg

V tejto triede je implementované vytvorenie dialógového okna, čiže rozhrania aplikácie. Tu sú vytvorené aj všetky komponenty viditeľné v dialógovom okne ako sú napríklad tlačidlá, výberové okná, polia pre písanie textu..

Pre tieto komponenty sú tu implementované funkcie, ktoré sa zavolajú pri zmene alebo použití nejakej komponenty. MFC má funkciu zasielania správ pri nejakej udalosti a následné odchytávanie správ. Napríklad pri stlačení tlačidla sa vyšle správa `ON_BN_CLICKED`, ktorá má ako parametre ID tlačidla ktoré ju vyvolalo a funkciu, ktorá sa vykoná pri stlačení tohto tlačidla.

V našom prípade sa správy posielajú pri stlačení tlačidiel: Štart, Ďalší krok, Nový vstup a pri zmene jazyka. Taktiež sa zasiela správa aj pri zatvorení okna, ktorá všetko zruší.

Na začiatok treba zadať všetky potrebné informácie do políčok pre vstup textu a stlačiť tlačidlo Štart. Toto tlačidlo je jediné prístupné na stlačenie, ostatné sú zatiaľ deaktivované. V okamihu stlačenia sa zavolá funkcia *OnBnClickedButtonStart* ktorá spustí lexikálnu aj syntaktickú analýzu. V tomto kroku sa zistí, či nastala chyba, ak áno, chyba sa vypíše a aplikácia sa znova dostane do pôvodného začiatočného stavu, s tým že všetky polia sú vyčistené a zoznamy vynulované.

Ak všetko po stlačení Štart prebehlo správne, zobrazí sa nám tabuľka základného bloku a tabuľka symbolov. Teraz sa tlačidlo Štart deaktivovalo a aktivovalo a tlačidlo Ďalší krok. Týmto tlačidlom si budeme aplikáciu krokovať až do konca. Pri každom stlačení sa vykoná nejaká grafická akcia, to znamená že sa vyznačí niečo v tabuľkách alebo sa zmení obsah tabuliek.

Po vyplnení tabuľky základného bloku sa nám vykreslia ďalšie 3 tabuľky: Tabuľka registrov, tabuľka adries a generovaný kód. Taktiež sa ich obsah mení a zvýrazňuje po stlačení tlačidla Ďalší krok.

Tretie tlačidlo je tam ak by sme chceli vymazať všetko čo sme doteraz vypočítali a chceli zadať nový vstup.

5.2.1 Funkcia OnPaint

Funkcia *OnPaint* sa zavolá vždy, ak je poslaná správa `ON_WM_PAINT`. A tá je poslaná vždy, ak je potreba niečo na aplikácii prekresliť.

Pre náš program je toto asi najdôležitejšia správa ktorá sa zasiela, pretože vďaka nej vidíme všetko čo vidíme. Ak chceme niečo nakresliť a sme napríklad mimo funkcie *OnPaint*, stačí zavolať funkciu *Invalidate*, ktorá prekreslí celé okno nanovo, alebo funkciu *InvalidateRect*, ktorá prekreslí iba určitú časť okna, udanú ako 2 súradnice.

Po každom stlačení tlačidla Ďalší krok i tlačidla Štart sa volá funkcia *Invalidate*, aby sa mohol vykresliť jeden krok pri demonštrovaní nášho výpočtu. Taktiež Pri stlačení tlačidla Nový vstup sa nám volá funkcia *Invalidate*, ale s tým, že už nepovolíme žiadne vykresľovanie tabuliek.

5.3 Trieda CliveAndDeadVariablesApp

Táto trieda je hlavná trieda pre celú aplikáciu. V nej sú všetky premenné, ktoré používam globálne, takisto všetky zoznamy a štruktúry používané vo všetkých triedach sú implementované tu.

5.4 Viacjazyčnosť

Aplikácia má na výber z 2 jazykov: slovenčina a angličtina. Vybrať jazyk je možné po štarte aplikácie v pravom hornom rohu ako výberové okno. Je to komponent ComboBox a na udalosť o zmene výberu sa vyšle správa a následne sa zavolá funkcia *OnCbnSelchangeComboLanguage* a podľa toho, aký jazyk je vybraný, podľa toho sa nastaví cesta k súboru, z ktorého sa majú jazykové reťazce čítať.

Zmena jazyku je možná iba pred spustením demonštrácie tlačidlom Štart, kde by to síce bolo možné zmeniť všetky reťazce, ale pri prekreslení plochy by sa stratili vyznačené riadky, ktoré sú vo funkcii *OnPaint* v priebehu programu.

Pri načítavaní zo súboru som narazila na problém kódovania. Funkcia ktorá patrí do triedy MFC, ktorá otvára súbor, nepodporovala kódovanie UTF-8, ktoré som potrebovala. Po hľadaní na internete som natrafila na funkciu, ktorá otvára súbor a priamo v nej je ako parameter, v akom kódovaní je súbor a tým sa problém vyriešil.

5.5 Proces prekladu

Teraz by som sa chcela venovať viac implementácii, ako som čo spravila a aký je postup pri preklade, čo sa kedy zavolá krok po kroku.

Pri spustení aplikácie sa v triede *CliveAndDeadVariablesApp* inicializujú globálne premenné, zavolá sa vytvorenie dialógového okna, ktoré pri inicializácii vytvorí všetky potrebné komponenty, ako sú tlačidlá, výberové boxy atď. Čiže, keď už vidíme samotné okno aplikácie, sú na ňom všetky potrebné komponenty. Každá komponenta má svoje ID a premennú ktorá patrí tejto komponente. S touto premennou potom môžem volať funkcie ktoré prislúchajú danej komponente. Takisto sa načítajú zo súboru všetky reťazce pre jazykové rozhranie, implicitne je zadaná slovenčina. Zo súboru čítam po riadkoch a reťazce postupne priraďujem určitým premenným, pričom je dôležité poradie v akom sú reťazce zoradené, aby správny reťazec prislúchal správnej premennej.

Ak už je všetko nainicializované a všetko prebehlo v poriadku, prvým krokom je správne zadanie vstupných informácií, to znamená do prvého editovacieho okienka zadať vstupný výraz, do druhého počet registrov. Po stlačení tlačidla štart sa vyšle správa

```
ON_BN_CLICKED(IDC_BUTTON_START,&CliveAndDeadVariablesDlg::
OnBnClickedButtonStart)
```

ktorá má ako parametre ID tlačidla a funkciu ktorá sa zavolá.

Po zavolaní funkcie *OnBnClickedButtonStart* sa vezme čo je zadané v prvom políčku, vložíme to do premennej. Ďalej skontrolujeme či je počet registrov správne zadaný, či nemá nepovolené znaky,

alebo či nieje zadané príliš veľké číslo alebo nula. Ak je počet registrov v poriadku, vrátime sa k premennej v ktorej je uložený reťazec z prvého políčka. Táto premenná bude ako parameter funkcie *Lex(premenná s reťazcom, dĺžka reťazca)*, ktorá sa v tomto momente zavolá.

Sme vo funkcii *Lex*. Funkcia *Lex* robí lexikálnu analýzu a detekuje časť chýb. Funkcia začína cyklom `for`, ktorý sa opakuje toľko krát, aký je dlhý reťazec na vstupe. V každom priechode cyklom kontrolujem jeden znak zo vstupného reťazca. Tento znak najprv testujem, že či je to číslo, znak z abecedy (a-z), povolená operácia (+, -, *, /), priradenie (:=), alebo zátvorky. Na čísla a písmená je už vytvorená funkcia v knižnici c, ale pre ostatné 3 kontroly som si spravila vlastné funkcie. Každá funkcia kontroluje, či je to ten znak ktorý definuje, ak áno, vráti hodnotu `true`, inak `false`. Ak sa teda testovaný znak nachádza v jednej z týchto skupín, vloží sa do pomocného poľa štruktúr s názvom *m_token*, kde štruktúra obsahuje znak a typ toho znaku. Typ znaku je číslovaný od 1 do 6.

Ak sa znak nenachádza ani v jednej z týchto skupín, funkcia *Lex* vráti „`false`" a nastaví sa premenná v ktorej uchovávam chybové výpisy s názvom *m_Error*. Po návrate do funkcie *OnBnClickedButtonStart* sa zavolá funkcia, ktorá vypisuje chybové hlásenia a zároveň aj funkcia, ktorá všetky potrebné zoznamy, premenné vynuluje.

Ak znak patril do niektorej zo skupín, testuje sa ďalej podľa jeho typu pomocou príkazu `switch`, ktorý má ako parameter typ premennej. Podľa toho sa rozhodne, že či sa bude k znaku chovať ako k číslu, písmenu, atď. V tejto časti sa vlastne kontroluje, či je to, povedzme si príklad, identifikátor. To znamená že jeho typ bol 1. Tu sa kontroluje či je už nejaký znak v poli štruktúr *m_token*, alebo či je to prvý znak. Ak je to prvý znak, vložíme ho do štruktúry s názvom *m_BigToken*, ktorá má položky reťazec a číslo. Tento reťazec bude obsahovať jeden lexém, v tomto prípade názov identifikátoru. Teraz je v tomto reťazci máme vlastne prvý znak, ktorého typ je 1, akože identifikátor. Ale ak by to náhodou nebol prvý znak v poli štruktúr *m_token*, tak sa testuje, aký je predchádzajúci znak v tomto poli, teda najprv sa kontroluje jeho typ. Ak by bol predchádzajúci znak iný, ako je typ aktuálneho znaku, tak sa obsah štruktúry *m_BigToken* ,v ktorej sú všetky znaky jedného typu, vloží na začiatok zoznamu všetkých lexémov. Ak by bol znak predchádzajúci toho istého typu, zase by sa pripísal znak na koniec reťazca v štruktúre *m_BigToken*. Ešte jedna kontrola sa tam nachádza, ak je prvý znak v poli štruktúr *m_token* operátor alebo zátvorka, automaticky to vloží na koniec zoznamu všetkých lexémov.

Vždy ako niečo vložím zo zoznamu, vynulujem si pole *m_token* a počet znakov ktoré sme zatiaľ sem uložili. To znamená že sme našli jeden lexém a môžeme hľadať a rozpoznávať ďalšie. U identifikátoru je tam ešte jedna výnimka, že prvý musí byť načítaný vždy znak od a do z. Ak je tomu tak, nasledujúce môžu byť aj čísla. Tým pádom ak máme typ znaku číslo, musím testovať ešte či je predchádzajúci znak typu identifikátor, ak áno, zmeniť aj jeho typ na identifikátor. Pre priradenie je tam iná situácia. Vždy musí byť prvá „:" a a ňou musí nasledovať „=". A vždy sa to musí objaviť

práve raz v poradí priamo za sebou. Čiže ak narazím na rovná sa, predchádzajúci symbol musí byť dvojbodka, inak je to chyba vstupu. Tak isto aj keď bude iba samotná dvojbodka, je to chyba. To že sa mi priraďovací príkaz objaví, si uchovám v jednej premennej ktorá má ty boolean a keď sa mi to vyskytne ešte raz, vypíše to opäť chybu.
Ak nám lexikálna analýza prebehla v poriadku, vrátili sme sa opäť do funkcie *OnBnClickedButtonStart*. Tu sa zavolá funkcia *CreateTS*, ktorá vytvorí tabuľku symbolov, do ktorej sa uložia všetky identifikátory. Zatiaľ sa všetkým priradia hodnoty L ako živá, pretože sú to programátorské premenné a ďalšie použitie je none čiže žiadne. Takisto tu vytvoríme časť tabuľky symbolov, ktorá obsahuje zatiaľ iba programátorské premenné, ktoré majú nastavenú adresu v pamäti , teda v tabuľke je „memory".

Opäť sa vrátime do funkcie *OnBnClickedButtonStart* kde sa zavolá funkcia *CreateTZB*, ktorá v sebe obsahuje syntaktickú analýzu a tvorbu tabuľky základného bloku.
Sme vo funkcii *CreateTZB*. Na začiatku sa nám inicializuje zoznam ktorým je implementovaný zásobník a zoznam ktorým je implementovaný vstup. Ako prvá položka zoznam bude znak $, ktorý bude indikovať dno zásobníka. Ďalej je tam cyklus `for`, ktorý sa bude konať tak dlho ako je počet všetkých lexémov v zozname *m_TokenList* bez 2 lexémov. Tieto 2 lexémy sú identifikátor do ktorého výsledok priraďujeme a operátor priradenie, ktoré zatiaľ nepotrebujeme. Teraz testujeme lexém kvôli precedenčnej tabuľke. Čiže testujem či je ten konkrétny lexém už konkrétna operácia, zátvorka, alebo identifikátor, ktorý má teraz rovnaké číslo typu ako číslo. V precedenčnej tabuľke berieme ako identifikátor aj číslo. Ak už je typ priradený, vložíme to celé do zoznamu s názvom input, ktorý bude obsahovať vstupný reťazec. Toto spravíme pre všetky ostatné lexémy. Ak už máme zoznam input naplnený, vstupujeme do ďalšieho cyklu `while`, ktorý sa koná toľko krát, dokiaľ nieje premenná ktorá detekuje koniec nastavená na true. Táto premenná sa nastaví vtedy, ak je na vstupe posledný znak, znak $ a na zásobníku tiež tento znak. To znamená že sme prišli na dno zásobníka, čiže sme zredukovali podľa pravidiel vstup. A na vstupe už nebude nič, pretože sme to všetko prečítali a zredukovali.

Ale najprv začíname vykonávať syntaktickú analýzu. Vezmeme vždy znak zo vstupu a znak zo zásobníka, podľa ich typov pristúpime k súradniciam v precedenčnej tabuľke. Ďalej postupujeme podľa postupu popísaného v kapitole o syntaktickej analýze. V zozname si udržiavam názov lexému a vždy tvar, podľa akého ho porovnávam so zadanými pravidlami. Pre identifikátor je to napríklad znak E ktorým mám označené identifikátory v precedenčnej tabuľke. Na zásobníku to značí nonterminály. Pre operátory je to samotný operátor. Podľa toho lepšie určím, o aké pravidlo išlo a nestratím hodnotu lexému, ktoré budem potrebovať pri vytváraní tabuľky základného bloku. Postupne si ukladám tieto znaky do reťazca a keď sa zadaný reťazec rovná nejakému pravidlu, zredukujem výraz pomocou zadaného pravidla a na zásobník vložím ľavú stranu pravidla. Pravú stranu pravidla nahradíme

skutočnými hodnotami a to priradíme do pomocnej premennej, ktoré máme predom pripravené v rozsahu 20 pomocných premenných. Tu dotvárame tabuľku symbolov, do ktorej tento krát ukladáme pomocné premenné, ktoré sú tým pádom nastavené na D ako „dead“ a ďalšie použitie znova na „none“. Túto trojadresnú inštrukciu vložíme do štruktúry pre tabuľku základného bloku a nakoniec ju vložíme na začiatok zoznamu štruktúr pre tabuľku základného bloku. Tiež tu dotvárame aj tabuľku adries, kde pomocné premenné majú nastavenú adresu na „nowhere“, čiže nikde.

Ak pri porovnávaní v precedenčnej tabuľke narazíme na súradnice ktoré majú prázdne políčko, znamená to syntaktickú chybu, čiže zle zadaný vstupný reťazec. Opäť sa nastaví chybová premenná a neskôr sa zavolá funkcia pre výpis chyby.

Ak však všetko prebehlo v poriadku, máme naplnenú tabuľku základného bloku trojadresnými inštrukciami, tabuľku symbolov a i tabuľku adries. Nastaví sa premenná ktorá indikuje, že prvá časť prebehla v poriadku.

V tomto momente sa zavolá funkcia na prekreslenie okna, ktorá vykreslí aj tabuľku základného bloku a tabuľku symbolov. Teraz bude ukázané, ako sa tabuľka základného bloku napĺňa a ako sa mení tabuľka symbolov po každom stlačení tlačidla Ďalší krok. Toto tlačidlo odošle správu

```
ON_BN_CLICKED(IDC_BUTTON_NEXT,
&CliveAndDeadVariablesDlg::OnBnClickedButtonNext).
```

Táto funkcia bude zaznamenávať počet kliknutí na toto tlačidlo, podľa ktorého potom určíme jednotlivé fázy. Napríklad prvá fáza sa koná toľko krát, koľko je riadkov v tabuľke základného bloku krát dva. Dva krát je to preto, lebo táto fáza má 2 etapy. To znamená že až po 2 kliknutiach sa posunieme na ďalší riadok v tabuľke základného bloku.

Túto činnosť teda robím vždy v 2 etapách. V prvej vždy vyznačím riadok ktorá práve napĺňam a doplním zvyšnú časť tabuľky podľa tabuľky symbolov tj. Stav a Ďalšie použitie každej premennej a v tabuľke symbolov vyznačím všetky identifikátory ktoré sú v danom riadku tabuľky základného bloku. V druhej etape pomením tabuľku symbolov, tak, že identifikátor ktorý bol na ľavej strane inštrukcie je označený ako mŕtvy a ďalšie použitie nieje a identifikátory ktoré sú na pravej strane sú živé a ďalšie použitie bude riadok na ktorom sa nachádzajú.

Ak máme tabuľku zaplnenú, to znamená, že aj počet krokov sa dostal do stavu, kedy je koniec prvej fázy. Nastupuje fáza druhá. Vtedy sa vykreslia ďalšie tabuľky a to: tabuľka registrov, tabuľka adries a generovaný kód. Opäť počet krokov je udaný počtom riadkov v tabuľke základného bloku krát dva. V prvej etape vyznačím riadok v tabuľke základného bloku, tentokrát začíname od vrchu. Vyznačím si tieto premenné aj v tabuľke adries a vygenerujem prvú časť cieľového kódu. Kód sa generuje až v tejto časti a to po zavolaní funkcie *GenCode* v ktorej sa volá funkcia *GetReg*. Tieto funkcie boli podrobnejšie popísané v kapitole o Generovaní cieľového kódu. Funkcia *GetReg* je implementovaná ako 3 za sebou nasledujúce cykly `for`, nie vnorené, prechádzam v nich tabuľku

registrov a hľadám register, ktorý odpovedá podmienkam. Ak nájdem vyhovujúci register vrátim ho príkazom `return`. Ak ho nenájdem, vojdem do druhého cyklu až do posledného. V poslednom sú podmienky nastavené tak, že nám to vždy nejaký register vráti, s tým že ak v ňom bola nejaká hodnota, vloží sa do pamäti. Ak nám funkcia vrátila register, pracujeme s týmto registrom ďalej vo funkcii *GenCode*. Kde podľa toho, aké máme inštrukcie v tabuľke základného bloku a podľa toho kde sú jednotlivé identifikátory uložené, generujeme cieľový kód. Napríklad ak máme inštrukciu `a:=b+c` a `b` je v registri R1 a `c` je v pamäti, vygenerujeme kód `ADD R1,c` , čím nám vlastne v registri R1 ostal výsledok, čiže obsah premennej `a`. Ak nieje premenná `b` v registri, treba si ju do registru najprv načítať, generovaný kód potom vyzerá nasledovne: `STORE R1,b` . Takto postupne prejdeme všetky riadky tabuľky základného bloku, až dokým nevygenerujem celý požadovaný cieľový kód.

Teraz by som sa chcela ešte vrátiť k samotnému vykresľovaniu. Všetko sa vykresľuje vo funkcii *OnPaint,* ktorá sa volá pomocou správy `ON_WM_PAINT()` . Táto správa sa volá buď automaticky ak je potreba vykresliť okno nanovo, napríklad pri minimalizácii okna, alebo „umelo“ a to volaním funkcie *Invalidate()*, poprípade *InvalidateRect()*, pre časť okna. V našom prípade voláme na vykreslenie ďalších krokov funkciu *Invalidate()*, pretože po každom prekreslení, sa vrátime naspäť do funkcie, keďže je ďalší krok nastavený a znova volaný z tej konkrétnej funkcie.

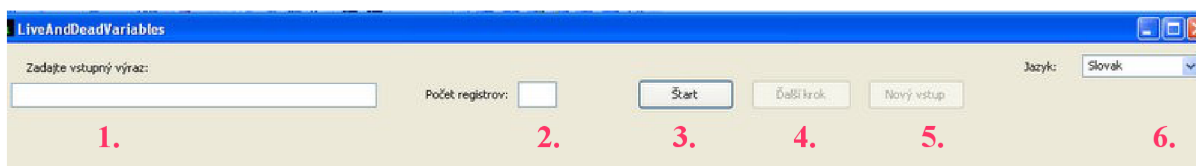
Vykresľovanie tabuliek je robené tak, že sú napevno zadané súradnice vrcholu každej tabuľky. Každý riadok je vrchná súradnica plus 15 bodov, to znamená, že riadok bude vysoký 15 bodov. Ak má teda tabuľka 10 riadkov, jej veľkosť bude 10x15. Najprv sa vykreslí podklad tabuliek, to je biely štvorec, potom sa vykreslí farebné „podsvietenie“ riadku s ktorým momentálne pracujeme , za tým grafika tabuľky a nakoniec názvy stĺpcov v tabuľke. To aký riadok vysvietim sa rozhodujem na základe toho, ak sa nachádzam na nejakom riadku tabuľky základného bloku, hľadám zadané identifikátory v tabuľke symbolov, ak ich nájdem, viem ich pozíciu a vykreslím podsvietenie vďaka veľkostiam okna a veľkosti riadku. Toto vykresľovanie je v cykle aby sme spravili pri jednom vykreslení všetky potrebné riadky. Ak máme grafiku tabuliek a vysvietenia hotovú, v cykloch sa vypíšu obsahy jednotlivých tabuliek. Tie sú až na konci,aby sme ich náhodou neprekreslili grafikou tabuliek.

Ak máme všetky kroky za sebou, deaktivuje sa nám tlačidlo Ďalší krok a zostanú vykreslené tabuľky vo výslednej podobe. Ďalším možným krokom je zatvorenie aplikácie alebo pokúsiť sa zadať nový vstup a začať proces odznova pomocou tlačidla Nový vstup.

6 Návod na použitie

Na priloženom CD je pod adresárom s názvom Aplikácia adresár s názvom LiveAndDeadVariables, ktorý obsahuje spustiteľný súbor a adresár so súbormi na načítanie počas behu programu. Adresár LiveAndDeadVariables je možné si skopírovať na ľubovolné miesto vo svojom počítači.

Po spustení súboru LiveAndDeadVariables.exe sa nám otvorí dialógové okno.



6.1 Vrchná časť dialógového okna

Na obrázku 6.1 je znázornená vrchná časť dialógového okna. Jednotlivé čísla označujú funkčné komponenty, ktoré podrobnejšie popíšem v nasledujúcej časti:

1. Je pole pre zadanie *vstupného reťazca*. Reťazec by mal byť v tvare `a:=b+c`, to znamená na ľavej strane je premenná, nie číslo ani iný operátor a na strane pravej môžu byť čísla, identifikátory, operátory (+,-,*,/) a zátvorky. Celý výraz by mal byť korektne matematicky zadaný.
2. Pred štartom programu treba vyplniť aj pole *Počet registrov.* Tento počet udáva, s koľkými registrami sa bude pri generovaní cieľového kódu počítať. Počet nesmie byť 0, ani číslo väčšie ako 20, kvôli veľkosti okna.
3. Tlačidlom *Štart* sa spúšťa program, ak sú obe polia vyplnené a správne. Po prvom stlačení sa deaktivuje, pretože je prístupné len na prvopočiatočné spustenie.
4. Tlačidlo *Ďalší krok* sa aktivuje po Stlačení tlačidla Štart ak je všetko zadané v poriadku. Toto tlačidlo bude vlastne „krokovač“ programu, každý ďalší krok sa vykoná po stlačení tohto tlačidla. Ak už sme prešli všetky kroky, tlačidlo sa samé deaktivuje.
5. Tlačidlom *Nový vstup* sa zruší všetko doteraz vytvorené, znova sa všetko nastaví ako na začiatku spustenia aplikácie a je možné zadať nový vstup.
6. Pred spustením programu, pred stlačením tlačidla Štart máme možnosť vo výberovom okne vybrať *typ jazyka* v ktorom aplikácia pobeží. Na výber je anglický a slovenský jazyk ako na obrázku 6.2

6.2 Výber jazyka

6.1 Postup

1. Spustiť aplikáciu dvojklikom na súbor LiveAndDeadVariables.exe
2. Do políčka vyznačeného na obrázku 6.1 s číslom 1 zadáme vstupný výraz
3. Do políčka číslo 2., Počet registrov zadáme počet registrov v rozmedzí 1-20
4. Stlačíme tlačidlo Štart ktoré je na obrázku pod číslom 3.
5. Ak sme zadali dobrý vstup, objavia sa nám 2 tabuľku ako na obrázku 6.3

6.3 Stav po stlačení tlačidla Štart

Ak sme zadali zlý vstup, vypíše sa chybové hlásenie a aká chyba nastala, ako napríklad na obrázku 6.4. Potom sa pokračuje znovu od bodu 2.

6.4 Chybové hlásenie

6. Po každom stlačení tlačidla Ďalší krok, ktoré je na obrázku 6.1 pod číslom 4., sa posúvame o krok vpred. Teraz vlastne prebieha celá demonštrácia, počítanie, vypĺňanie tabuliek, až dokým sa tlačidlo nedeaktivuje.
7. Výsledok demonštrácie je potom 5 tabuliek, kde v poslednej je generovaný cieľový kód.
8. Ak by sme chceli ísť odznova, nemusíme vypínať celý program, stačí stlačiť tlačidlo Nový vstup (na obrázku 6.1 pod číslom 5.) .
9. Program sa vypína tlačidlom Koniec, alebo krížikom vpravo hore.

7 Záver

V priebehu vytvárania projektu som sa zoznámila s knižnicou MFC, s ktorou som pracovala v prostredí Visual Studia 2005. Táto knižnica má výhodu, že už sú v nej predvytvorené triedy a funkcie napríklad pre tvorbu zoznamov, čiže keď som toto všetko zistila, dalo sa stým pekne pracovať. Taktiež výhodou bolo, že sa dalo dobre navrhnúť grafické prostredie.

Keďže je to demonštračný program, to aby to bolo graficky prívetivé a jednoduché na ovládanie bolo mojou prioritou. Program je určený pre študentov ale i pre vyučujúcich, ktorý by chceli vidieť celý proces generovania kódu a optimalizácie krok za krokom.

Výhodou je aj viacjazyčnosť programu, čiže nie sú obmedzovaní študenti, ktorý nevedia po slovensky, respektíve po česky.

Tento projekt by sa dal ešte vylepšiť takým spôsobom, že by mohol mať automatický „krokovač", čiže nejakú funkciu, ktorá bude automaticky prepínať na ďalší krok. Taktiež by bola dobrá možnosť návratu o krok späť, keby študent alebo vyučújuci nestihli zaznamenať čo sa dialo krok predtým.

Vylepšenie by mohlo prísť aj zo strany grafiky, kde by animácie neboli statické, ale mohli by sa nejak graficky posúvať a názornejšie ukázať spracovávané miesta.

Literatúra

[1] Češka,M.,Beneš,M.,Hruška,T.: Překladače, skriptum VUT Brno, VUT Brno 1993,
262 str., ISBN 80-214-0491-4

[2] Wikipédia: Microsoft Foundation Class Library. [online], posledná aktualizácia 24. 3. 2009 ,
[cit. 2009-04-28].
URL http://cs.wikipedia.org/wiki/Microsoft_Foundation_Class_Library

[3] Kurz Formální jazyky a překladače, FIT VUT v Brně. [online], študijné materiály,
[cit. 2009-04-20].
URL http://www.fit.vutbr.cz/study/courses/IFJ/.

[4] Morrison,M.: Sams Teach Yourself MFC, Sams Publishing, 1999, 510 str.,
ISBN 0-672-31553-X

[5] MFC Reference [online]
URL http://msdn.microsoft.com/en-us/library/d06h2x6e%28VS.80%29.aspx

[6] Wikipédia: Visual C++ . [online], posledná aktualizácia 7.5.2009, [cit. 2009-05-10].
URL http://en.wikipedia.org/wiki/Visual_C%2B%2B

[7] Meduna, A.: Automata and Languages, Springer, London 2000, 886 str., ISBN 1-85233-07-0

[8] Meduna, A.: Podklady pro výuku překladačů, [online], študijné materiály
URL http://www.fit.vutbr.cz/~meduna/zap/ZAP_SLAJDY.pdf

Zoznam príloh

Príloha 1. CD so zdrojovými textami, aplikáciou a programovou dokumenáciou