Vysoké učení technické v Brně

Fakulta informačních technologií

Dokumentácia k projektu do IFJ a IAL

Implementácia prekladaču imperatívneho jazyka IFJ18

Tým 076, varianta II

Členovia tímu:

Matej Janček (vedúci)	xjance00	25%
Peter Horňák	xhorna14	25%
Róbert Hubinák	xhubin03	25%
Nikolas Patrik	xpatri00	25%

Obsah

1. Úvod

Cieľom projektu bolo vytvoriť prekladač imperatívneho jazyka IFJ18, ktorý je založený na základoch jazyka Ruby 2.0, čo je dynamicky typovaný jazyk.

Táto dokumentácia popisuje implementáciu prekladača pre jazyk IFJ18. Rozoberá postup pri práci na projekte, spoluprácu v tíme a tak isto voľbu nášho riešenia.

Projekt sme si rozdelili na viaceré časti, pričom každý v tíme si zobral na starosť aspoň jednu časť. Podľa týchto častí je rozdelený aj tento dokument.

2. Spolupráca v tíme

2.1. Stretnutia a komunikácia

Zo začiatku práce na projektu sme sa častejšie stretávali, aby sme sa dohodli na určitých pravidlách pri tvorbe projektu a tak isto rozobratia projektu na menšie časti, ktoré sme dokázali jednoduchšie splniť. Využili sme možnosť pozrieť minuloročné záznamy z prednášok a naštudovať si potrebnú látku dopredu. Keď sme lepšie pochopili celému problému implementácie, stretnutia celého tímu sme obmedzili.

Celková komunikácia v tíme nebol problém, keďže takmer celý tím bývame na jednej izbe. Avšak aj napriek tomu sme si zvolili komunikačnú službu *Slack*, cez ktorú sme riešili projekt na diaľku.

2.2. Verzovanie kódu

Z dôvodu potreby pracovať na aktuálnom zdrojovom kóde sme sa zhodli na využitý verzovacieho systému *git* s repozitárom na *Github*-e. Postupom času sa vytvárali vetvy pre rôzne časti projektu, ktoré sa nakoniec zjednotili do jednej vetvi, master.

Pokúsili sme sa využiť aj *Issue tracking*, avšak po niekoľkých pokusoch sa od toho upustilo a problémy sa riešili zvyčajne osobne.

2.3. Rozdelenie práce

Matej Janček	Lexikálna analýza, dokumentácia
Peter Horňák	Syntaktická analýza, generovanie kódu, sémantická analýza
Róbert Hubinák	Lexikálna analýza, generovanie kódu
Nikolas Patrik	Precedenčná analýza, generovanie kódu, testovanie, sémantická analýza

3. Lexikálna analýza

Lexikálnu analýzu vykonáva tzv. skener, ktorý funguje na princípoch deterministického konečného automatu, ktorého vstupom je zdrojový kód IFJ18.

Vstupný kód spracováva po jednom znaku, kde biele znaky, ako medzera alebo tabulátor, preskočí. Skener sa riadi podľa implementovaného switchu s nekonečným cyklom. Ak znak nevyhovuje žiadnemu prechodu v konečnom automate, skener sa ukončí s lexikálnou chybou alebo ak sa nachádza v konečnom stave, vráti prečítaný nevyhovujúci znak do zdrojového kódu a predá syntaktickému analyzátoru svoj posledný stav.

Jednotlivé znaky spája do lexémov, ktoré následne predáva vo forme tokenov syntaktickému analyzátoru.

Token sme implementovali ako dvojicu (buffer, type). *Buffer* je pointer na miesto v pamäti, kde je uložená postupnosť znakov ukončená končiacou nulou. Táto postupnosť reprezentuje znakovú hodnotu typu. *Type* nám hovorí, akého typu je aktuálny lexém. Všetky možné typy sú uchované v štruktúre, pre jednoduchšiu implementáciu.

Pri spracovaní komentárov sme sa rozhodovali, či je potrebné odovzdávať syntaktickému analyzátoru typ zodpovedajúci komentáru. Rozhodli sme sa pre riešenie s ignoranciou celého textu v komentári a odovzdávaním typu komentára. Toto rozhodnutie sme spravili kvôli tomu, že programátor si píše komentáre len pre vizuálnu kontrolu a nie pre spracovanie prekladačom.

4. Syntaktická analýza

Syntaktická analýza je vykonávaná tzv. parserom, ktorý riadi všetky ostatné moduly projektu. Je volaný takmer hneď po spustení programu a beží počas celej doby analýzy. Prijíma tokeny od lexikálneho analyzátoru, ktoré následne kontroluje podľa LL-gramatiky priloženej v prílohe. LL-gramatika je implementovaná pomocou rekurzívneho zostupu, kde každý neterminál predstavuje jednu funkciu. Avšak tento postup nevyužívame na vyhodnocovanie výrazov. Vyhodnocujeme ich pomocou precedenčnej analýzy, z dola hore, ktorá je implementovaná v osobitnom module *parseexp.c*.

Zároveň pri kontrole pravidiel vykonávame sémantické akcie, ktoré vedú ku kontrole správnej sémantiky vstupného kódu.

Informácie o funkciách ukladáme v globálnej tabuľke symbolov. Pričom každá funkcia spolu s hlavným telom programu má svoju lokálnu tabuľku symbolov, ktorá je využívaná na ukladanie premenných v danej funkcii.

Po úspešných syntaktických a sémantických kontrolách daných lexémov voláme modul generovania kódu.

5. Generovanie kódu

Generovanie kódu sme rozdelili ako samostatný modul *generate.c*, ktorého rozhranie sa nachádza v *generate.h*.

Generovaný kód vypisujeme na *stdout* v prípade, že všetky predošlé analýzy prebehli úspešne. Inštrukcie sa generujú počas syntaktickej analýzy a sú postupne ukladané do obojstranne viazaného zoznamu, implementovaného v *instrlist.c*. V prípade, že nastane chyba v niektorej analýze, zoznam je uvoľnený.

Pre využite obojstranne viazaného zoznamu sme sa rozhodli, pretože nám uľahčil riešenie určitých problémov. Medzi tieto problémy patrí redefinícia premennej v cykly alebo nedefinovnie premennej pri nevykonaní podmienky.

Základom generovania je funkcia *fillString*, ktorá do bufferu poslaného do funkcie cez argument, načíta výstup funkcie *printf* s premenným počtom parametrov.

5.1. Indexovanie

Na indexovanie pomocných premenných a návestí používame statické premenné typu *int*, ktoré sú počiatočne nastavené na hodnotu *1* a sú inkrementované pri každom zavolaní generovacej funkcie.

Rovnako využívame statický zásobník implementovaný v *stack.c*, ktorý slúži na indexovanie v prípade, že generujeme vnorenú podmienku alebo cyklus.

5.2. Generovanie výrazov

Generovanie výrazov prebieha počas precedenčnej analýzy, teda parsovanie výrazov. Generovanie nastáva vždy keď syntaktická analýza výrazov detekuje výskyt pravidla počas analýzy zásobníka. Po detekcii pravidla dochádza k volaniu pravidiel gramatiky syntaktickej analýzy. Počas tohto procesu sa môžu detekovať tri druhy pravidiel, čo zahŕňa prevod terminálu na neterminál na zásobníku. Pri tomto prevode dochádza k generovanie inštrukcie *push*, ktorá vkladá na zásobník nájdený terminál. V tomto prípade terminál môže byť jeden z nasledujúcich lexémov: indentifikátor, integer, float, exponenciálny integer, reťazec alebo hodnota nil.

Ďalší prípad detekcie pravidla môže byť výskyt dvoch neterminálov alebo jedného terminálu, pričom terminál reprezentuje určitý operátor. Neterminály reprezentujú už generované identifikátori. Pri generovaní inštrukcii tohto pravidla je najdôležitejšia typová kontrola pred tým, než sa

vykoná inštrukcia operácie, ktorá pri nevyhovujúcich typoch pre danú operáciu generuje inštrukciu *exit* s ukončovacím kódom 4, tj. chyba pri typovej kontrole za behu programu.

Zvláštnym prípadom je operácia delenia, pri ktorej dochádza za behu v interprete ku kontrole, či deliteľ nie je rovný nule. Ak áno, interpretácia končí chybou 9.

Posledným pravidlom, ktoré môže byť detekované je pravidlo zátvoriek, ktoré však negeneruje žiaden kód.

6. Testovanie projektu

Jednotlivé časti projektu si postupne každý testoval sám. Dohodli sme sa, že pred každým uverejnením kódu na *Github*-e, musí byť kód otestovaný a so správnou funkčnosťou. V neskoršej fáze projektu sme testovali prekladač ako jeden celok. Spísali sme si zdrojové kódy, kde sme testovali rôzne možné prípady vstupu a porovnávali výstup s referenčným výstupom.

7. Dátové štruktúry

7.1. Tabuľka s rozptýlenými položkami

Keďže sme si vybrali variantu projektu II, ktorá obsahovala implementáciu tabuľky symbolov pomocou tabuľky s rozptýlenými položkami.

Po vzájomnej dohode v tíme sme použili už nami vytvorenú tabuľku a hashovaciu funkciu *smdbhash* na predmete *IJC*, ktorý absolvoval každý člen tímu. Tabuľka je implementovaná podľa návrhového vzoru obalový typ. Preto štruktúra položky tabuľky obsahuje ukazovateľ na *void* , ktorý sa pretypuje na štruktúru, s ktorou momentálne potrebujeme pracovať. Potom funkcie pre prácu s tabuľkou príjmajú ukazovatele na prácu so štruktúrami, ktorá je obsiahnutá v tomto obaľovacom type.

Veľkosť tabuľky sme si zvolili na hodnotu 512. Toto číslo sme si zvolili

7.2. Garbage collector

V tomto projekte sme naprogramovali obdobu garbage collectoru (ďalej už len GB) s obmedzenou funkcionalitou. GB využíva jednosmerný viazaný lineárny zoznam. Do tohto zoznamu sa ukladajú ukazatele na pamäť, ktorá bola alokovaná pomocou rozhrania GB.

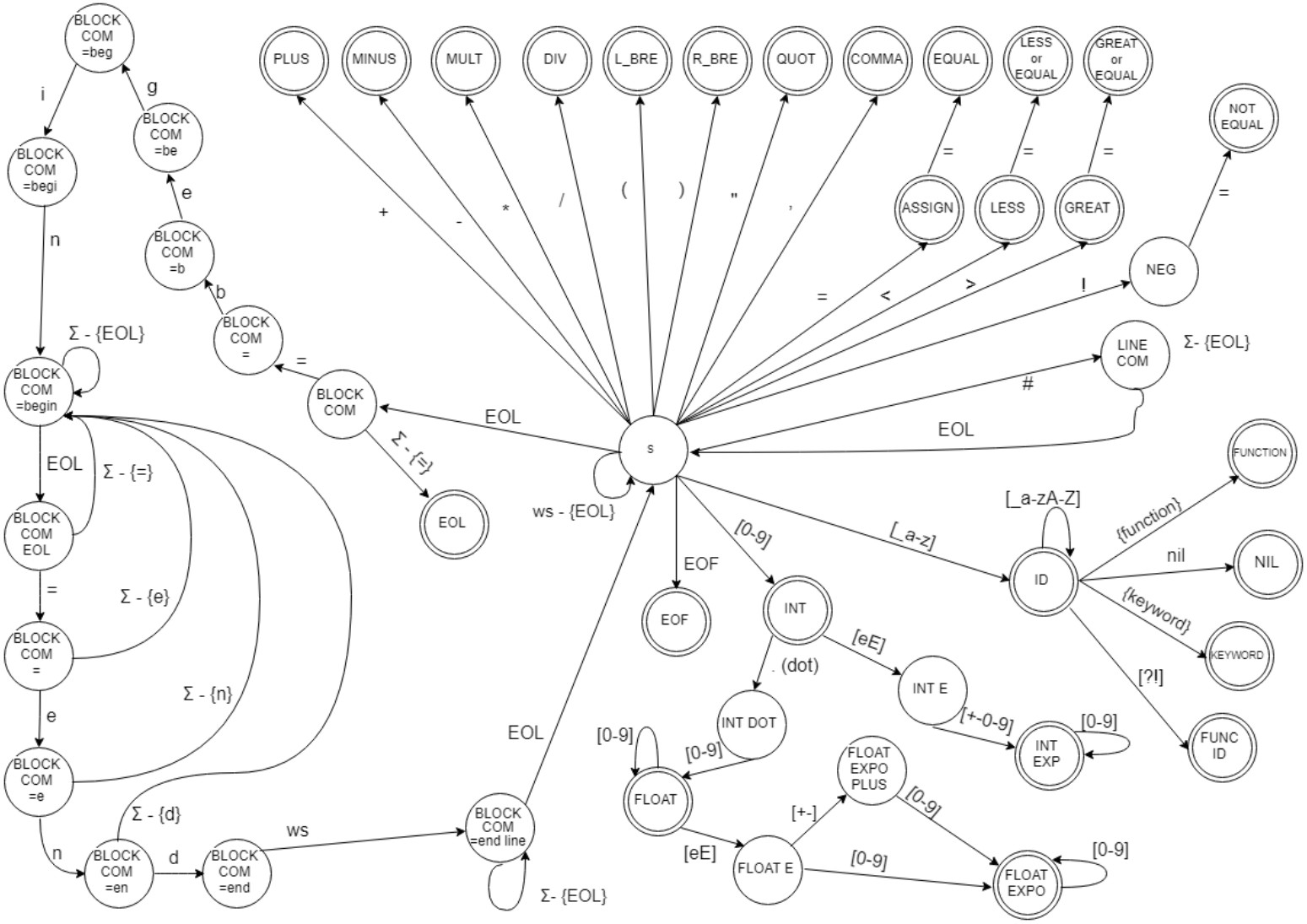
Rozhranie GB zahŕňa funkcie pre prácu s pamäťou a lineárnym zoznamom zapuzdrené do funkcií *gb_malloc*, *gb_realloc*, *gb_free* alebo funkcie na uvoľnenie všetkých registrovaných ukazateľov na pamäť, *gb_release_resources*. Pred ukončením programu sa volá funkcia *gb_exit_process,* ktorá uvoľní alokovanú pamäť a ukončí program s exit kódom z argumentu.

8. Záver

Počas práce na tomto projekte sme sa mohli lepšie oboznámiť s rôznymi fázami tvorby projektu, od rozloženia celého projektu na menšie časti, až po testovanie jednotlivých častí a výsledného projektu. Každý z tímu si odnáša nové poznatky v programátorských zručnostiach, ale aj v tímovej práce.

9. Prílohy

9.1. Diagram konečného automatu



9.2. LL-gramatika

```
(1)  ST_LIST → STAT ST_LIST
(2)  ST_LIST → EOF
(3)  STAT → id ID_ITEM
(4)  STAT → l_bracket EXPR
(5)  STAT → value EXPR
(6)  STAT → fid FUNC
(7)  STAT → if EXPR then EOL ELSE_ST_LIST EOL END_ST_LIST EOL
(8)  STAT → while EXPR do EOL END_ST_LIST EOL
(9)  STAT → def l_bracket BRACKET EOL END_ST_LIST EOL
(10) ELSE_ST_LIST → STAT ELSE_ST_LIST
(11) ELSE_ST_LIST → else
(12) END_ST_LIST → STAT END_ST_LIST
(13) END_ST_LIST → end
(14) ID_ITEM → equals ASSIGN
(15) ID_ITEM → EOL
(16) ID_ITEM → sign EXPR
(17) ID_ITEM → FUNC
(18) ASSIGN → id NEXT
(19) ASSIGN → fid FUNC
(20) ASSIGN → l_bracket EXPR
(21) ASSIGN → value EXPR
(22) NEXT → EOL
(23) NEXT → sign EXPR
(24) NEXT → id NEXT_PARAM
(25) NEXT → value NEXT_PARAM
(26) NEXT → l_bracket BRACKET EOL
(27) FUNC → l_bracket BRACKET EOL
(28) FUNC → EOL
(29) FUNC → PARAM
(30) PARAM → id NEXT_PARAM
(31) PARAM → value NEXT_PARAM
(32) NEXT_PARAM → EOL
(33) NEXT_PARAM → comma PARAM
(34) BRACKET → r_bracket
(35) BRACKET → BRC_PARAM
(36) BRC_PARAM → id NEXT_BRC_PARAM
(37) BRC_PARAM → value NEXT_BRC_PARAM
(38) NEXT_BRC_PARAM → r_bracket
(39) NEXT_BRC_PARAM → comma BRC_PARAM
(40) EXPR → ε
```

9.3. Precedenčná tabuľka

	+	-	*	/	(	)	<	<=	>	>=	==	!=	id	$
+	>	>	<	<	<	>	>	>	>	>	>	>	<	>
-	>	>	<	<	<	>	>	>	>	>	>	>	<	>
*****	>	>	>	>	<	>	>	>	>	>	>	>	<	>
/	>	>	>	>	<	>	>	>	>	>	>	>	<	>
(	<	<	<	<	<	=	<	<	<	<	<	<	>	
)	>	>	>	>		>	>	>	>	>	>	>		>
<	>	>	>	>	>	>					>	>	<	>
<=	<	<	<	<	<	>					>	>	<	>
>	<	<	<	<	<	>					>	>	<	>
>=	<	<	<	<	<	>					>	>	<	>
==	<	<	<	<	<	>	<	<	<	<			<	>
!=	<	<	<	<	<	>	<	<	<	<			<	>
Id	>	>	>	>		>	>	>	>	>	>	>		>
$	<	<	<	<	<		<	<	<	<	<	<	<	