Vysoké učení technické v Brně
Fakulta informačních technologií

Prekladač imperatívneho jazyka IFJ17

Tím 086, varianta II

Dlabaja Drahomír,	**xdlaba02,**	**46%**
Ďurovič Róbert,	xdurov01,	27%
Kubík Michal,	xkubik33,	00%
Sabo Jozef,	xsaboj00,	27%

Brno, 6. prosince 2017

1 Úvod

Dokumentácia popisuje návrh a implementáciu prekladača jazyka IFJ17 4-členným tímom, ktorého zloženie je uvedené na úvodnej strane. Jazyk IFJ17 je zjednodušenou podmnožinou jazyka FreeBASIC. Výsledný program, napísaný v jazyku C, funguje ako konzolová aplikácia. Načítava riadiaci program v jazyku IFJ17 zo štandardného vstupu a generuje kód v jazyku IFJcode17 na štandardný výstup. Všetky chybové hlásenia a varovania vypisuje na štandardný chybový výstup, pričom vracia číslo, ktoré reprezentuje typ chyby. V prípade, že preklad prebehne bez problémov sa vracia nula. Cieľový jazyk IFJcode17 je medzikódom, ktorý zahrňuje inštrukcie triadresné a zásobníkové.

Požadovaný diagram konečného automatu, LL-gramatika, LL-tabuľka a precedenčná tabuľka sa nachádzajú v sekcii dokumentu 6. Prílohy.

2 Analýza problému

Po prvých prednáškach predmetu IFJ a po prečítaní zadania, sme zistili, že našou úlohou je navrhnúť prekladač zložený zo 4 základných častí, ktorých implementáciu by bolo vhodné si medzi sebou rozdeliť:

- Lexikálna analýza
- Syntaktická analýza
- Sémantická analýza
- Generovanie výsledného kódu

Pri našom úvodnom stretnutí sme analyzovali zadanie, diskutovali sme o tom aké skúsenosti máme s jazykom C, aký verzovací systém budeme používať, ako bude prebiehať naša ďalšia komunikácia a taktiež sme sa dohodli na tom, ako si jednotlivé časti medzi sebou rozdelíme.

3 Práca v tíme

3.1 Komunikácia

Komunikácia v tíme prebiehala aktívne najmä na sociálnej sieti Facebook, vďaka ktorej náš tím vlastne aj vznikol. Pre spriehľadnenie vývoja projektu sme použili verzovací systém Git.

3.2 Postup práce

Pri práci na projekte, sme postupovali priebežne s čerstvo naučenými znalosťami o jednotlivých častiach prekladača z prednášok. Sami sme si vybrali, čím chceme a vieme prispieť. Pri problémoch sme si navzájom pomáhali a spoločne navrhovali prepojenie modulov, ktoré sme implementovali.

3.3 Rozdelenie práce

Dlabaja Drahomír : vedúci tímu, git, Makefile, syntaktický analyzátor, sémantický analyzátor

Ďurovič Róbert : lexikálny analyzátor, prevod dokumentácie do texu

Kubík Michal : testovanie

Sabo Jozef : načítanie stringov, výpis chybových hlásení, vstavané funkcie, generovanie kódu, dokumentácia

Rozdělení bodů je dáno poměrným množstvím odvedené práce jednotlivých členů.

4 Popis riešenia

Popisy jednotlivých častí riešenia sú okomentované ich hlavnými riešiteľmi.

4.1 Lexikálny analyzátor

Prvým krokom k riešeniu projektu a lexikálnej analýzy bol návrh konečného automatu. Následná konzultácia s ostatnými členmi tímu viedla k niekoľkým úpravam a korekciám tohto návrhu až do finálnej podoby. V priebehu vývoja interpretu tak už nedochádzalo ku žiadnym radikálnym zmenám tejto časti interpretu, čím sme sa vyhli zbytočným problémom.

Ďalšia fáza pozostávala zo samotnej implementácie konečného automatu, ktorá vychádzala z dodaného demo intepretu. Kód však musel byť značne rozšírený a upravený pre lepšiu zrozumiteľnosť. Samotná implementácia stavov automatu do `switch` a `else if` konštrukcií podľa návrhu už nebola nijak obtiažna.

V ďalších krokoch však bolo potrebné vyriešiť identifikovanie kľúčových slov cez určenú funkciu, escape sekvencie a informácie o tokenoch v konečných stavoch, ktoré sa ukladali do hashovacej tabuľky. Na konci tejto etapy sme už získali základný lexikálny analyzátor. Na jeho finálne dokončenie bolo nevyhnutné ešte ošetriť chybové stavy. Najprv sme museli ale indetifikovať potenciálny chybový výstup, čo sa nie celkom optimálne darilo a nakoniec to viedlo k nízkemu percentuálnemu hodnoteniu lexikálneho analyzátoru. V tomto prípade bolo nutné ošetriť veľkosť písmen v stringoch, úprava však bola našťastie iba kozmetická a nebolo potrebné výrazne zasahovať do konštrukcie scanneru.

4.2 Syntaktický analyzátor

Syntaktický analyzátor je z velké části napsaný metodou rekurzivního sestupu, výrazy jsou dále řešeny pomocí precedenční analýzy.

Klíčovým krokem rekurzivního sestupu bylo navrhnout pravidla gramatiky a následně i LL tabulku. Pomocí gramatiky jsme poté napsali samotný kód, ve kterém platí, že pro každý neterminál existuje právě jedna funkce a tyto funkce se podle pravidel navzájem volají. Tohle všechno jsme stihli za jedno dopoledne. Znamená to, že tím byla polovina syntaktické analýzy hotová?

Ani omylem. Jednalo se o modul, který v průběhu vývoje zaznamenal největší změny a s jistotou se dá říct, že jak pravidla, tak kód, byl v průběhu vývoje minimálně dvakrát kompletně přepsán. Jeho finální verze uzřela světlo světa teprve pár dní před termínem pokusného odevzdání, naprosto nesrovnatelná se svým syntaktickým pradědečkem.

U precedenční analýzy byla situace velmi podobná. Precedenční tabulku jsme s kolegama sestrojili za pár hodin a měli jsme z ní dobrý pocit. Začali jsme psát kód, šlo to jako po másle a zaslepeni vidinou brzkého dokončení jsme ztratili pojem úplně o všem. Hlavně o přehlednosti kódu. Když jsme totiž začali s debugováním, najít chybový kus kódu byl nadlidský úkol, který přiváděl k zoufalství snad i našeho největšího pomocníka, GDB.

Podařilo se nám opravit pár chyb, ale bylo nám jasné, že takhle to dál nejde. Pomocí znalostí získaných při programování první verze kódu jsme zdrojový soubor s precedenční analýzou smazali a na jeho následníkovi jsme začali pracovat lépe a radostněji. Jak jinak taky.

A že to radost byla, když se nám původně téměr tisíc řádků kódu podařilo zkrátit na polovinu. Relační výrazy jsme přesunuli do rekurzivního sestupu, volání funkce také, takže nám zbyl krásně čistý kód, který se příjemně debugoval. Nakonec to s ním taky moc nedopadlo, ale o tom více v další kapitole.

4.3 Sémantický analyzátor a tabuľka symbolov

Tabulka symbolů je implementována dle zadání pomocí hashovací tabulky. S tím nebyl žádný problém, nic co bychom neznali z IAL, případně IJC. Dalším krokem bylo tuto tabulku nějak zapouzdřit.

Vytvořili jsme si tedy modul `semnatics.h`, který pracuje s globální a lokální hashovací tabulkou, přidává a kontroluje identifikátory a řeší datové typy. Další práce s tabulkou symbolů je řešena právě přes tyto funkce a metody.

Generování abstraktního syntaktického stromu jsme se rozhodli vynechat a v rámci úspory času jsme se rozhodli sémantické akce (vč. generování výstupního kódu) zasadit přímo do syntaktického analyzátoru, čímž jsme efektivně zavraždili zbývající naděje na čistý a přehledný kód.

4.4 Načítanie stringov

Modul pre načítanie „nekonečných“ stringov vznikol úpravou funkcií zo súboru `str.c` v riešení jednoduchého interpretu uverejneného na stránke IFJ.

4.5 Výpis chybových hlásení

Súbor `error.c` obsahuje všetky možné typy fprintf hlásení, ktoré sa vypisujú na štandardný chybový výstup pri konkrétnych chybách v prebiehu prekladu. Je v ňom definovaná aj globálna premenná error_value (jej hodnota je na začiatku nula), do ktorej sa ukladá návratová hodnota programu.

4.6 Generovanie kódu

Klíčový moment při řešení generování kódu přišel ve chvíli, kdy jsme si uvědomili, že jazyk IFJcode17 podporuje zásobníkové instrukce. V ten moment odpadla potřeba drtivé většiny pomocných proměnných. Kvůli (Díky?) tomu jsme se rozhodli nevyužít možnosti dočasných rámců a argumenty i návratovou hodnotu předáváme přes zásobník. Počet pomocných proměnných se nám podařilo zredukovat na dvě pro každý rámec. Kdybychom měli více času, případně kdyby existovala zásobníková verze instrukce CONCAT, podařilo by se nám tento počet zredukovat na polovinu. Kdyby případně existovala zásobníková verze instrukce WRITE, pevně věříme, že pomocné proměnné by nebyly potřeba vůbec.

Pre generovanie výstupného kódu IFJcode17 slúži funkcia `generate`, ktorá má až 7 parametrov. Parametre funkcie definujú: typ inštrukcie, parametre inštrukcie a prefixy jej parametrov. Ak je prefix string, funkcia tento string prevádza do formy s escape sekvenciami potrebnými pre znaky s ASCII kódom 000-032, 035 a 092. Volanie funkcie generate má na starosti syntaktický analyzátor.

4.7 Vstavané funkcie

Vstavané funkcie boli vytvorené pomocou inštrukčnej sady jazyka IFJcode17. Generujú sa pri každom preklade spolu s výstupom prekladača. Ich výpis má na starosti funkcia `generate_builtin_functions`, ktorá sa nachádza v súbore `semantics.c`.

4.7.1 funkcia `Length`

Funkcia vracia dĺžku reťazca zadaného parametrom.

4.7.2 funkcia `SubStr`

Funkcia vracia podreťazec zadaného reťazca. Začiatok a dĺžka podreťazca sú dané parametrami.

4.7.3 funkcia `Asc`

Funkcia vracia ASCII hodnotu znaku na parametrom zadanej pozícii v reťazci.

4.7.4 funkcia `Chr`

Funkcia vracia jednoznakový reťazec so znakom, ktorého ASCII kód je zadaný parametrom.

4.8 Testovanie a ladenie

Drtivá většina ladění probíhala pomocí nástroje `GDB`, bez kterého by náš program byl jen pseudonáhodným generátorem znaků.

Svou zásluhu ovšem má i `Valgrind`, která nám aktivně pomáhal pochytat veškerou prchající paměť.

5 Záver

Tento projekt bol pre nás najrozsiahlejším tímovým projektom na akom sme dorez pracovali. Spoznali sme fázy, ktorými prechádza tvorba aplikácie od návrhu až po jej testovanie. Párkrát sme si aj ponadávali, lámali si hlavu, prečo nám to nefunguje. Nebola to prechádzka ružovou záhradou. No zároveň musíme uznať, že keby sme sa o prekladačoch len učili a nenavrhovali ich, nikdy by sme im tak dobre neporozumeli a s istotou zabudli ako fungujú do nasledujúceho semestra. Ďalším veľkým plusom sú získané skúsenosti na takomto rozsiahlom projekte s prácou v tíme, ktoré určite po škole využijeme vo firmách, kde sa zväčša pracuje v tímoch. Využili sme aj pokusné odovzdanie, ktoré nám pomohlo odhaliť chybu v lexikálnej analýze, kde sme získali málo percent, pretože scanner prevádzal na malé písmená aj stringy, čo by nemal a celkovo sme získali predstavu o stave jednotlivých častí prekladača, ktorá nás upokojila, že na tom nie sme až tak zle.

6 Prílohy

6.1 Precedenčná tabuľka

	±	*/	\	(	)	term	$
±	>	<	<	<	>	<	>
*/	>	>	>	<	>	<	>
\	>	<	>	<	>	<	>
(	<	<	<	<	=	<	
)	>	>	>		>		>
term	>	>	>		>		>
$	<	<	<	<		<	

6.2 LL - gramatika

```
<prog> -> declare <fdecl> <prog>
<prog> -> function <fdef> <prog>
<prog> -> scope <scope>
<scope> -> EOL <scopelist> scope EOL
<scopelist> -> input <input> <scopelist>
<scopelist> -> print <print> <scopelist>
<scopelist> -> if <if> <scopelist>
<scopelist> -> do <do> <scopelist>
<scopelist> -> dim <dim> <scopelist>
<scopelist> -> <assign> <scopelist>
<scopelist> -> end
<funclist> -> input <input> <funclist>
<funclist> -> print <print> <funclist>
<funclist> -> if <if> <funclist>
<funclist> -> do <do> <funclist>
<funclist> -> dim <dim> <funclist>
<funclist> -> <assign> <funclist>
```

```
<funclist> -> return <return> <funclist>
<funclist> -> end
<iflist> -> input <input> <iflist>
<iflist> -> print <print> <iflist>
<iflist> -> if <if> <iflist>
<iflist> -> do <do> <iflist>
<iflist> -> <assign> <iflist>
<iflist> -> else <else> <iflist>
<iflist> -> end
<elselist> -> input <input> <elselist>
<elselist> -> print <print> <elselist>
<elselist> -> if <if> <elselist>
<elselist> -> do <do> <elselist>
<elselist> -> <assign> <elselist>
<elselist> -> end
<looplist> -> input <input> <looplist>
<looplist> -> print <print> <looplist>
<looplist> -> if <if> <looplist>
<looplist> -> do <do> <looplist>
<looplist> -> <assign> <looplist>
<looplist> -> loop
<params> -> ID as <type> <parlist>
<params> -> )
<parlist> -> , ID as <type> <parlist>
<parlist> -> )
<fdecl> -> function ID ( <params> as <type> EOL
<fdef> -> ID ( <params> as <type> EOL <funclist> function EOL
<rel> -> ( <rel> )
<rel> -> <term> OPERATOR <term>
<dim> -> ID as <type> <dim2>
<dim2> -> EOL
<dim2> -> = <expr> EOL
<input> -> ID EOL
<print> -> <expr> ; <print>
<print> -> EOL
<if> -> <rel> then EOL <iflist> if EOL
<else> -> EOL <elselist>
<do> -> while <rel> EOL <looplist> EOL
<return> -> <expr> EOL
<assign> -> ID = <expr> EOL
<term> -> ID
<term> -> CONST
<type> -> integer
<type> -> double
<type> -> string
<expr> -> <expr> OPERATOR <expr>
<expr> -> ( <expr> )
<expr> -> CONST
<expr> -> ID
<expr> -> <call>
```

```
<call> -> ID ( <args>
<args> -> <term> <arglist>
<args> -> )
<arglist> -> , <term> <arglist>
<arglist> -> )
```

6.3 LL - tabuľka

	declare	function	scope	EOL	input	print	if	do	dim	ID	end	return	else	loop	)	,	(	CNST	=	EXPR	while	integer	double	string
<prog>	1	2	3																					
<scope>				4																				
<scopelist>					5	6	7	8	9	10	11													
<funclist>					12	13	14	15	16	17	19	18												
<iflist>					20	21	22	23		24	26		25											
<elselist>					17	28	29	30		31	32													
<looplist>					33	34	35	36		37				38										
<params>										39					40									
<parlist>															42	41								
<fdecl>		43																						
<fdef>										44														
<rel>										46							45	46						
<dim>										47														
<dim2>				48															49					
<input>										50														
<print>				52																51				
<if>										53							53	53						
<else>				54																				
<do>																					55			
<return>																				56				
<assign>										57														
<term>										58								59				60	61	62

6.4 Diagram konečného automatu