VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta informačních technologií

FORMÁLNÍ JAZYKY A PŘEKLADAČE

2018/2019

Implementace překladače imperativního jazyka IFJ18

Tým 35, varianta I

Rozšírenia: BOOLOP

Vedúci:

Vorobiev Nikolaj (xvorob00) (28%)

Ostatní členovia:

Luhin Artsiom (xluhin00) (28%)

Koša Benjamín (xkosab00) (28%)

Cibák Michal (xcibak00) (16%)

Brno, 5. prosince 2018

Úvod

Cieľom tohto skupinového projektu je vytvoriť konzolovú aplikáciu v jazyku C, takzvaný prekladač, ktorý zo štandardného vstupu načíta program napísaný v jazyku IFJ18 a na štandardný výstup vypíše program preložený do jazyka IFJcode18.

IFJ18 je zjednodušená podmnožina jazyka Ruby 2.0.

IFJcode18 je jazyk podobný jazyku symbolických inštrukcií. V ňom napísaný výstupný program nazývame medzikód, ten je ďalej spracovávaný interpétom dostupným na školskom serveri.

Návrh

Zdrojový kód našej aplikácie sme rozdelili do troch logických celkov, v našom projekte označených ako scanner, parser a generátor cieľového kódu, ktoré sú na sebe závislé. Po dvoch prechodoch vstupného zdrojového kódu sme schopní vygenerovať celý výstupný kód.

Scanner – lexikálny analyzátor

Základom pre implementáciu lexikálneho analyzátoru je návrh konečného automatu. Jeho úlohou je rozdeliť zdrojový kód zo vstupu na jednotlivé lexémy jazyka, aby na požiadanie parseru mohol vždy jeden vrátiť vo forme tokenu.

Načítanie lexému prebieha tak, že zo vstupného kódu na štandardnom vstupe scanner po jednom číta znaky a zároveň sa z už načítanej časti snaží zistiť, či sa môže jednať o nejaký lexém jazyka a ak áno, o aký typ. Tento proces skončí, keď je prečítaný znak, ktorý už nepatrí do žiadneho možného lexému. Pokiaľ sa nejedná o biely znak, vieme, že sa nemôže jednať o platný lexém jazyka a lexikálny analyzátor vráti chybu. V opačnom prípade sa o lexém jedná a ten je prevedený do podoby tokenu, ktorý obsahuje jeho typ a prípadne aj nejakú jeho užitočnú hodnotu.

Výnimočný prípad tvoria porovnávacie operátory, ktoré je scanner schopný identifikovať aj bez nutnosti ich oddelenia od ostatných lexém bielymi znakmi.

Taktiež je dôležité poznamenať, že komentáre sú z pohľadu lexikálnej analýzy ignorované a najbližší platný znak je teda znak konca riadku. Ignorované sú aj nadbytočné biele znaky.

Parser – syntaktický a sémantický analyzátor

Veľmi dôležitou časťou prekladača je práve parser, ktorý zabezpečuje syntaktickú a sémantickú kontrolu vstupného kódu, ale môžeme o ňom tiež povedať, že riadi celú činnosť prekladača.

Prvý prechod

V prvom prechode kódu parser žiada tokeny od scanneru, nad nimi prebieha syntaktická analýza pomocou LL gramatiky, ktorá overí, či reťazec tokenov predstavuje syntakticky správne napísaný program, kontrolu výrazov však zabezpečuje precedenčná analýza s využitím pravidiel zapísaných v precedenčnej tabuľke.

Každý spracovaný token je uložený do jednosmerne viazaného zoznamu tokenov.

Zároveň prebieha aj prvá časť sémantickej analýzy s využitím tabuľky symbolov, ktorá je implementovaná pomocou binárneho vyhľadávacieho stromu a sú do nej ukladané definície funkcií a ich premenných. Každý uzol stromu predstavuje jednu funkciu vo vstupnom programe, či už vstavanú alebo užívateľom definovanú, pre nás užitočné informácie v nich uložené sú názov funkcie a ukazovateľ na zoznam jej premenných, pričom parametre tejto funkcie sú vždy na začiatku zoznamu, ostatné premenné nasledujú až za nimi. Aby sme mohli uložiť aj premenné hlavného tela programu, musíme ho tiež brať ako funkciu a teda mu priradiť meno, a to také, aké pri definícii užívateľskej funkcie vo vstupnom programe nie je možné použiť, aby nedošlo k prípadnej kolízii názvov, rozhodli sme sa preto ako názov použiť kľúčové slovo „def“ jazyka IFJ18.

Zatiaľ je pomocou nej overované iba to, či práve definície premenných a funkcií boli v kóde lexikálne správne umiestnené. To znamená, že pri použití premennej alebo volaní funkcie je táto vyhľadaná v tabuľke symbolov a ak sa ju nepodarí nájsť, znamená to, že ešte nebola definovaná, čo je chyba. Výnimkou je prípad, kde sa na volanie funkcie narazí v tele inej funkcie. Ak táto volaná funkcia ešte nebola definovaná, jej meno a počet parametrov sa uloží do pomocného zoznamu a ten je po skončení prvého prechodu rovnakým spôsobom celý skontrolovaný.

Druhý prechod

V druhom prechode sa už tokeny čítajú zo zoznamu tokenov, scanner teda nie je vôbec volaný. Pomocou generátoru cieľového kódu sa prevádzajú reťazce tokenov na výsledný kód, ktorý je uložený do tabuľky inštrukcií. V prípade, že niektorý reťazec tokenov predstavuje výraz, sú postupne kontrolované typy jeho operandov, čím sa dokončí sémantická kontrola. Následne je tento výraz s využitím pomocného zásobníku prevedený na postfixový tvar a dočasne uložený do pomocného zoznamu, čo značne uľahčuje jeho následný prevod do cieľového kódu, ktorý je opäť uložený do tabuľky inštrukcií.

Po úspešnom druhom prechode už nasleduje iba výpis obsahu tabuľky inštrukcií, teda programu prepísaného do cieľového jazyka, na štandardný výstup a aplikácia môže skončiť.

Generátor cieľového kódu

Pri každom zavolaní generátora je reťazec tokenov prevedený podľa pravidiel, ktoré sme si určili na základe špecifikácie vstupného a výstupného jazyka v zadaní, na inštrukcie v cieľovom kóde. Tie sa skladajú z názvu inštrukcie a jej parametrov, ktorých počet vychádza

zo špecifikácie jazyka IFJcode18. Jednotlivé parametre sa skladajú z hodnoty uloženej ako reťazec a pravdivostnej hodnoty, ktorá určuje, či bude daný parameter inštrukcie použitý pri jej výslednom výpise. Tie sú ukladané do zoznamu inštrukcií k príslušnej funkcii v tabuľke inštrukcií, čo je binárny vyhľadávací strom podobný tabuľke symbolov, rozdiel je len v tom, že sú z neho vynechané vstavané funkcie a ukazovateľ v uzloch ukazuje na zoznam vygenerovaných inštrukcií príslušnej funkcie. Vynechané sú z toho dôvodu, že ich priamy prepis je v cieľovom kóde veľmi jednoduchý, zatiaľ čo prepis do podoby samostatnej funkcie by výsledný kód zbytočne skomplikoval.

V cieľovom kóde sa vôbec nepracuje s globálnym rámcom. Lokálny rámec s premennými hlavného tela programu je na spodku zásobníka rámcov, na začiatku každej užívateľskej funkcie sa vytvorí dočasný rámec a presunie sa na vrchol tohto zásobníka, na jej konci sa na dátový zásobník nahrá jej návratová hodnota a tento rámec sa odstráni. Tento prístup jednoduchým spôsobom rieši problém oblasti viditeľnosti premenných a vrátenie návratovej hodnoty funkcie.

Všetky operácie s výrazmi sú v cieľovom kóde riešené výhradne pomocou zásobníkových inštrukcií, pretože sa jedná a časovo menej náročné operácie a zároveň nie je nutné vytvárať množstvo pomocných premenných.

Po vygenerovaní inštrukcií pre všetky funkcie a celé hlavné telo programu nastáva výpis celého cieľového kódu na štandardný výstup. Postupuje sa tak, že sa najprv vypíše úvodný riadok cieľového kódu, následne sa prechádza tabuľkou inštrukcií a vypisujú sa všetky užívateľom definované funkcie, nakoniec sa vypíše hlavné telo programu.

Aplikácia končí.

Tímová práca

Plánovanie procesu vývoja prekladača sme začali až po prebratí niekoľkých učebných látok, keď sme mali dostatočnú predstavu o tom, ako má výsledný produkt vyzerať. Zároveň sme už vedeli dosť na to, aby sme si mohli časť práce rozdeliť v skupine.

Na prvom stretnutí sme riešili hneď niekoľko dôležitých prvkov spoločnej práce na tomto projekte, a to ako spolu budeme komunikovať, akým spôsobom budeme spolu zdieľať priebežný pokrok na programe a ako si rozdelíme prácu.

Spoločnú komunikáciu sme chceli riešiť buď prostredníctvom sociálnej siete Facebook, alebo v dnešnej dobe veľmi populárnej aplikácie Discord, po návrhu jedného člena tímu sme sa však rozhodli pre aplikáciu Telegram, ktorá je veľmi vhodným komunikačným prostriedkom práve na prácu v skupine.

Zdieľanie zdrojových kódov v dnešnej dobe nie je žiaden problém, existujú rôzne webové aplikácie, ktoré umožňujú spoluprácu viacerých osôb na projekte, poskytujú zabezpečenie proti vzájomnému prepisovaniu kódu a správu jeho verzií. Rozhodli sme sa preto využiť službu „GitLab“.

Pri rozdeľovaní práce sme sa zamerali iba na prvé 2 časti projektu, na ktorých bolo možné pracovať nezávisle na sebe. Konkrétne sa jednalo o lexikálny a syntaktický analyzátor. Spísali sme si svoje nápady, ako by sme mohli postupovať pri ich implementácii, spoločne overili správnosť týchto návrhov, ujasnili sme si, ako budú tieto časti spolu prepojené a nakoniec sa dohodli na rozdelení práce do dvojíc.

Termín, ktorý sme si určili na vypracovanie týchto častí, sa nám ale nepodarilo dodržať, výsledok sa značne oneskoril nie len z dôvodu prípravy na skúšky, ale aj kvôli práci na projektoch z ďalších predmetov.

Po zdanlivom dokončení týchto 2 analyzátorov sme zorganizovali ďalšie stretnutie, na ktorom sme prebrali doterajšie výsledky a pristúpili k návrhu práce na sémantickom analyzátore. Dohodli sme sa, kto navrhne precedenčnú tabuľku a kto LL-gramatiku, ostatní mali dolaďovať už hotový kód a začať pracovať na kóde zabezpečujúcom sémantickú kontrolu. Spomínané návrhy boli uskutočnené pomerne rýchlo a teda sa aj ostatní členovia opäť vrátili k programovaniu. Dalo by sa povedať, že každý sa pustil do toho, čo bolo potrebné dokončiť najskôr, pokiaľ už na tom nepracoval iný člen tímu.

V tomto období sa komunikácia na Telegrame značne rozmohla. Jednotlivé funkcie spolu museli správne komunikovať, takže sme sa navzájom dohadovali, „kto“ pošle aké parametre „komu“ a podobne. Zároveň sme začali uskutočňovať prvé testy, pomocou ktorých sme odhalili niekoľko chýb, o ktorých sme taktiež dosť diskutovali.

Následne sme uskutočnili posledné osobné stretnutie, kde sme opäť prebrali doterajšie výsledky a diskutovali sme o nájdených chybách, čo ich mohlo spôsobiť a ako ich odstránime. Popri tom sme odhalili aj niekoľko chýb spôsobených rôznou interpretáciou zadania jednotlivými členmi tímu, o ktorých sme nemali šancu vedieť, kým sme danú problematiku spoločne neprediskutovali. Na záver stretnutia sme si ešte rozdelili úlohy, a to pokračovanie v implementácii, opravu chýb, navrhnutie prepisu vstavaných funkcií do cieľového kódu a vytvorenie návrhu, ako bude celé generovanie výsledného kódu prebiehať.

Väčšina naplánovaných prác na kóde prekladača bola hotová do niekoľkých dní, zároveň sme prišli s dvoma návrhmi na spôsob, akým by mohol pracovať generátoru výsledného kódu. Zhodli sme sa však na tom, že lepšie ako spoločne rozoberať oba návrhy bude, keď jeden z nich implementuje ten, kto ho navrhol, pretože svojmu návrhu určite rozumie najlepšie. Týmto rozhodnutím sme sa snažili predísť vzniku chýb z dôvodu vzájomného nepochopenia.

Ako postupne končili hlavné práce na kóde a bolo možné uskutočniť prvé testy výstupu prekladača, komunikácia na Telegrame sa zamerala výhradne na tému opráv chybných častí. Na základe spätnej väzby zo skúšobného odovzdania sme však zistili, že naša implementácia má omnoho viac nedostatkov, ako sme odhadovali na základe výsledkov vlastných testov, a preto sme ešte intenzívnejšie pracovali na odhaľovaní a opravách chýb. Vždy s opravou jednej sme však odhalili ďalšiu a takto to pokračovalo až takmer do termínu odovzdania. Najzávažnejšie problémy sme však rýchlo vyriešili a nasledovalo už len dolaďovanie.

Nakoniec sa nám podarilo odovzdať program v stave, s ktorým sme boli celkom spokojní.

Výsledky práce:

Nikolaj: tabuľka symbolov, tabuľka inštrukcií, sémantická kontrola, generácia inštrukcií, testy

Artsiom: syntaktická kontrola tela programu, prevod na postfixový tvar, generácia inštrukcií, zoznam tokenov

Benjamín: lexikálny analyzátor, syntaktická kontrola výrazu, generácia inštrukcií pre výrazy, testy

Michal: dokumentácia

Literatúra:

Stručný prehľad jazyka RUBY:

http://www.newthinktank.com/2015/02/ruby-programming-tutorial/

Naprogramovanie jednoduchého kompilátoru:

https://www.destroyallsoftware.com/screencasts/catalog/a-compiler-from-scratch/

Prílohy:

LL-gramatika

```
1. <prog>              --> <stat-list> <fn-def-list> <prog>
2. <prog>              --> $
3. <stat-list>         --> <stat> EOL <stat-list>
5. <stat-list>         --> e
4. <fn-def-list>       --> <fn-def> EOL <fn-def-list>
6. <fn-def-list>       --> e

7. <fn-def>            --> DEF <fn-head> EOL <stat-list> END
8. <fn-def>            --> e
9. <fn-head>           --> ID ( <fn-def-pars> )
10. <fn-def-pars>      --> <fn-def-par> <fn-def-pars1>
11. <fn-def-pars1>     --> , <fn-def-par> <fn-def-pars1>
12. <fn-def-pars>      --> e
13. <fn-def-pars1>     --> e
14. <fn-def-par>       --> ID

15. <stat>             --> IF <EXP> THEN EOL <stat-list> ELSE EOL <stat-list> END
16. <stat>             --> WHILE <EXP> DO EOL <stat-list> END
17. <stat>             --> ID = <exp-call>
18. <stat>             --> <exp-call>
19. <stat>             --> e

20. <exp-call>         --> ID ( <fn-pars> )
21. <exp-call>         --> ID <fn-pars>
22. <exp-call>         --> <EXP>

23. <fn-pars>          --> <fn-par> <fn-pars1>
24. <fn-pars1>         --> , <fn-par> <fn-pars1>
25. <fn-pars>          --> e
26. <fn-pars1>         --> e
27. <fn-par>           --> ID
28. <fn-par>           --> INTEGER
29. <fn-par>           --> FLOAT
30. <fn-par>           --> STRING

e - epsilon pravidlo
$ - konec souboru
<EXP> - vyraz, vyhodnoceny precedencni syntaktickou analyzou
```

Precedenčná tabuľka

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

Diagram konečného automatu lexikálnej analýzy (nedokončený)