VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentácia k projektu z predmetu IFJ a IAL
Implementácia prekladača jazyka IFJ20
Tým 085, varianta 1

Vedúci tímu:
Marián Zimmermann, xzimme03 – 33%
Ďalší členovia:
Vanessa Jóriová, xjorio00 – 34%
Magdaléna Bellayová, xbella01 – 33%
Matušík Adrián, xmatus35 – 0%

Úvod

Táto dokumentácia popisuje návrh a implementáciu prekladača jazyka IFJ20, ktorý je podmnožinou jazyka Go. Prekladač načítava zdrojový kód zapísaný jazykom IFJ20 a prekladá ho do cieľového jazyka IFJcode20 – medzikódu. Pre implementáciu sme si zvolili variantu 1, využívajúcu štruktúru binárneho stromu.

Riešenie projektu sa dá rozdeliť na 4 hlavné (neskôr popísané) úseky:

- Lexikálny analyzátor (scanner)
- Syntaktická a sémantická analýza metódou rekurzívneho zostupu (parser)
- Precedenčná analýza výrazov
- Generovanie medzikódu

Rozdelenie práce v tíme

Marián Zimmermann – lexikálna analýza, generovanie kódu
Magdaléna Bellayová – precedenčná analýza
Vanessa Jóriová – syntaktická a sémantická analýza, generovanie kódu
Adrián Matušík – člen sa do vypracovania projektu nezapojil

Práca v týme

Na projekte sme pracovali spoločne, každý člen tímu si vzal na starosti jednu podmnožinu projektu s tým, že mu ostatní v prípade potreby pomáhali. Situáciu nám skomplikovalo vypadnutie štvrtého člena, ktorý mal na starosti generovanie kódu, ktoré sa teda rozdelilo medzi ostatných.

Ako verzovacií systém sme využili Git a ako vzdialený repozitár Github.

Keďže nám aktuálna situácia nedovolila osobné stretnutia, väčšina komunikácie prebiehala písomne (cez skupinovú konverzáciu) alebo hlasovo na Discorde.

Lexikálny analyzátor

Lexikálny analyzátor (scanner) je implementovaný ako deterministický konečný automat. Pozostáva z cyklu while a switchu do ktorého sa postupne posielajú znaky (ignoruje biele znaky a komentáre) a na základe nich vyhodnocuje o aký token sa jedná po prípade ukladá jeho hodnotu do stringu strtmp.

Syntaktický a sémantický analyzátor

Syntaktický a sémantický analyzátor (neskôr len *parser*) je ústrednou a hlavnou časťou nášho programu, ktorá sa spúšťa hneď na začiatku a svoju činnosť ukončuje až po spracovaní vstupného kódu v jazyku IFJ20. Počas svojej činnosti úzko spolupracuje s ostatnými modulmi:

- Žiada scanner o tokeny, pomocou ktorých vyhodnocuje, či je zdrojový kód napísaný v jazyku popísanom vytvorenou gramatikou
- Pokiaľ narazí na výraz, predá riadenie precedenčnej analýze, ktorá výraz vyhodnotí

- Priamo z parseru sa generuje cieľový medzikód

Syntaktická a sémantická analýza prebiehajú súbežne. Zdrojový kód sa spracuje jedným behom, syntaktická analýza prijíma tokeny od lexikálneho analyzátoru a rekurzívnym zostupom vyhodnocuje, či je zdrojový súbor gramaticky správne. V prípade, že syntaktická analýza narazí na výraz, predá riadenie precedenčnej analýze, ktorá tokeny až po koniec výrazu spracuje.

Pri implementácii **syntaktickej analýzy** sme ako najzásadnejší problém vnímali implementáciu lexikálneho analyzátora, ktorý načíta token po tokene bez možnosti token „vrátiť“. To čiastočne vyriešil mód *Peek,* ktorý nahliadol na ďalší token a uložil ho, aby bol vrátený pri ďalšom zavolaní. V niektorých prípadoch sa ale z tohto dôvodu program implementačne odchyľuje od LL-gramatiky, popr. sú využité iné spôsoby, ako rozhodnúť o rozložení neterminálu (sémantická analýza), podobné odchýlky sú ale vykonávané s dôrazom na to, aby nenarušili gramatiku popísanú LL gramatikou a prijímali rovnaký jazyk ako ten LL gramatikou popísaný.

Pri implementácii parseru sme sa pridŕžali tejto gramatiky (neterminály sú písané veľkým písmenom:

1. START → PROLOG PROG
2. PROLOG → package main
3. PROG → FUNCTION PROG
4. PROG → *eps*
5. FUNCTION → func id (ARGS) RETURN_F {S_BODY }
6. ARGS → ARG ARG_NEXT
7. ARG_NEXT → *eps*
8. ARG_NEXT → , ARG ARG_NEXT
9. ARG → id DATA_TYPE
10. RETURN_F → (RETURN_TYPES)
11. RETURN_F → *eps*
12. RETURN_TYPES → RETURN RETURN_NEXT
13. RETURN → DATA_TYPE
14. RETURN_NEXT → , RETURN RETURN_NEXT
15. RETURN_NEXT → *eps*
16. S_BODY → STATEMENT S_BODY
17. S_BODY → *eps*
18. STATEMENT → DEFINITION
19. STATEMENT → ASSIGN
20. STATEMENT → IF
21. STATEMENT → FOR
22. STATEMENT → F_CALL
23. STATEMENT → RETURN_S
24. DEFINITION → id := <expr>
25. ASSIGN → id ID_NEXT = FUNC_OR_EXPR
26. FUNC_OR_EXPR → <expr> EXPR_NEXT
27. FUNC_OR_EXPR → F_CALL
28. ID_NEXT → , id ID_NEXT
29. ID_NEXT → *eps*
30. EXPR_NEXT → , <expr> EXPR_NEXT
31. EXPR_NEXT → *eps*
32. IF → if <expr> { S_BODY } else { S_BODY }
33. FOR → for FOR_DEFINITION ; <expr> ; FOR_ASSING { S_BODY }
34. FOR_DEFINITION → DEFINITION
35. FOR_DEFINITION → *eps*

36. FOR_ASSING → F_ASSIGN
37. F_ASSIGN → id ID_NEXT = <expr> EXPR_NEXZ
38. FOR_ASSIGN → *eps*
39. F_CALL → id(PARAMS)
40. PARAMS → TERM TERM_NEXT
41. PARAMS → *eps*
42. TERM_NEXT → , TERM TERM_NEXT
43. TERM_NEXT → *eps*
44. RETURN_S → return RET_EXPR
45. RET_EXPR → <expr> EXPR_NEXT
46. RET_EXPR → *eps*
47. TERM → int_value
48. TERM → float_value
49. TERM → string_value
50. TERM → id
51. DATA_TYPE → int
52. DATA_TYPE → float64
53. DATA_TYPE → string

Väčšina **sémantickej analýzy** prebieha tiež v parseri, kedy parser sémanticky okontroluje takmer všetko, k čomu má prístup, čím sa minimalizuje nutnosť behových kontrôl.

IFJ20 je silne typovaný jazyk a to umožňuje statické priradenie typu novo definovanej premennej. Najprv sa vyhodnotí výraz za definíciou novej premennej, precedenčná analýza oboznámi parser o type novej premennej a premenná sa s informáciou o type a fakte, že sa jedná o premennú, nie o funkciu, uloží na vrchol zásobníka tabuliek symbolov.

Sémantické kontroly volania funkcie sú o niečo zložitejšie vďaka faktu, že funkcia môže byť zavolaná ešte pred definíciou. Implementácia lexikálneho analyzátora nepodporovala sémantickú kontrolu v dvoch behoch, ktorá by samotnú sémantickú analýzu zjednodušila, funkcie teda sémanticky kontrolujeme v jednom behu spôsobom:

- Pokiaľ sa narazí na definíciu funkcie ešte pred jej volaním, je názov funkcie uložený do tabuľky symbolov aj s informáciami o počte, type, názve parametrov a počte a type návratových hodnôt. Zaznačený je aj fakt, že bola definovaná. Definovanie funkcie s rovnakým menom vedie na chybu, rovnako ako aj následné volanie funkcie so zlými parametrami/návratovými hodnotami
- Pokiaľ sa narazí na volanie dosiaľ nedefinovanej funkcie, do tabuľky symbolov sú poznačené všetky informácie, ktoré z volania môžeme získať – typ parametru, návratových hodnôt a jej názov. Pokiaľ sú tieto informácie nekompletné (napr. _ v návratových hodnotách) a narazí sa na ďalšie volanie, ktoré informácie rozšíri (volanie so známym typom návratovej hodnoty), informácie sú aktualizované. Volanie s konfliktnými informáciami vedie na sémantickú chybu. Pokiaľ sa neskôr v programe nájde definícia funkcie odpovedajúca zaznačeným informáciám, tabuľka symbolov je aktualizovaná o fakt, že bola funkcia definovaná – definícia nekompatibilná s volaniami vedie na chybu. Na konci programu sa vyhodnotí, či boli všetky funkcie definované – nedefinované funkcie spôsobia chybu.

Parser využíva funkcie na generovanie kódu, ktorým predáva informácie získané syntaktickou a sémantickou analýzou.

Precedenčná analýza

Zavolá ju scanner, keď narazí na výraz. Tu sa berú tokeny, kým sa nenarazí na koniec výrazu, alebo nejaký chybu. Keď sa prečíta token tak sa podľa toho, čo je na vrchole zásobníka určí stav z precedenčnej tabuľky. Keď je v precedenčnej tabuľke stav LO tak sa vloží do zásobníku stav LO a potom až token. V prípade, že je stav HI tak sa ide skracovať výraz podľa pravidla. Do druhého zásobníku sa uložia znaky až po stav LO a súčasne sa vyberú z prvého zásobníka. Podľa počtu a typu operandov je následne vybrané pravidlo. Keď sa našlo pravidlo tak celá časť sa nahradí neterminálom, ktorý sa vloží do prvého zásobníka. Ak nastane stav EQ, tak sa len vloží token do zásobníka a ak nastane stav ER, tak sa vráti syntaktický chyba. Súčasný token sa porovnáva z najvrchnejším terminálom, kým ešte má, čo redukovať, ak nie a ešte súčasný token nebol vložený do zásobníka tak sa vloží. Ukončenie bez chyby nastane, keď príde koniec riadku a pomocou pravidiel sa bude vedieť na všetko na čo ešte neboli uplatnené pravidlá skrátiť tak, že zostane len jeden neterminál. Precedenčná analýza vracia TRUE ak prebehla v poriadku a nastavuje flag na typ výsledného neterminálu.

Využívajú sa tu aj funkcie generovania kódu, kde posiela generátoru potrebné informácie na vygenerovanie kódu pre výrazy.

Dátové štruktúry

Tabuľka symbolov

Pri sémantickej analýze sa využíva zásobník tabuliek symbolov. Jednotlivé tabuľky symbolov sú implementované pomocou binárneho stromu. Uzly binárneho stromu si štruktúry obsahujúce kľúč (názov funkcie/premennej) a dáta – skutočnosť, či sa jedná o premennú alebo funkciu, či je funkcia definovaná, upravený názov rozšírený o návestie scopu využívaný pri generácii kódu, počet a typy parametrov a návratových hodnôt...

Jednotlivé tabuľky symbolov sú zviazané do zásobníka. Vkladať nové uzly sa dá len na aktuálny vrchol zásobníka (mimo volania dosiaľ nedefinovanej funkcie, kedy sa informácie značia do globálnej tabuľky symbolov – spodok zásobníka), vyhľadáva sa v zásobníku celom a vracia najvyššie položenú zhodu s kľúčom – to nám umožňuje prácu so scopmi v tele programu a dodržiavať viditeľnosť a prekrývanie premenných.

String

Pre prácu s identifikátormi neobmedzenej dĺžky využívame knižnicu str.h, ktorú sme prevzali z IFJ stránok.

Zásobník

Pri práci s precedenčnou analýzou sa využíva zásobník, kde si pre každý element ukladá symbol, typ, hodnotu a nasledovníka. Samotný zásobník má v sebe len pointer na vrchol zásobníka.

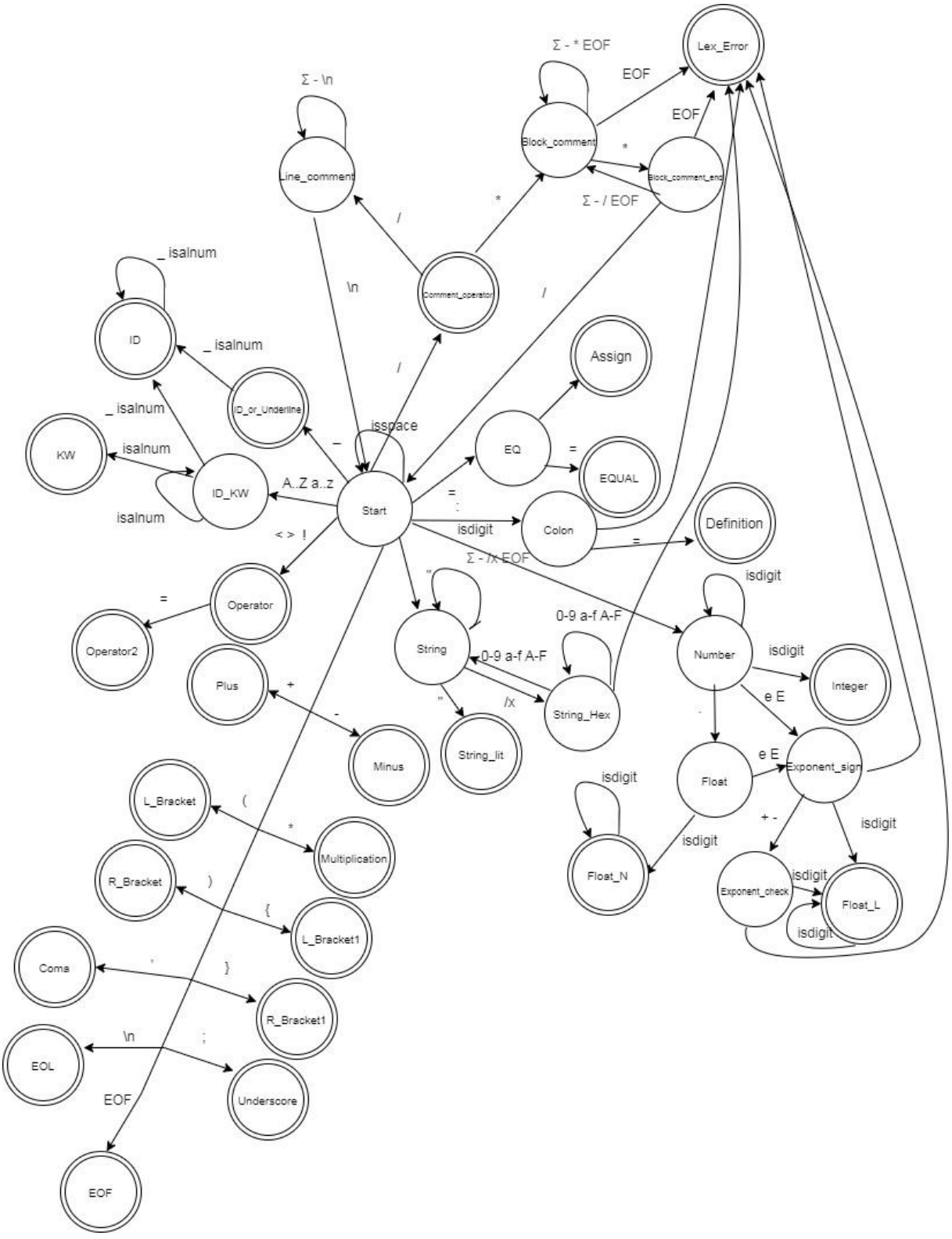
Vytvorili sme jednoduché funkcie na prácu so zásobníkom aby sme do neho vedeli vkladať aj vyberať hodnoty, zisťovať vrchol aj vrchný terminál, vyprázdňovať celý zásobník a zisťovať, či je prázdny.

Záver

Aj keď sme boli oboznámení s rozsiahlosťou a náročnosťou projektu, jeho rozsah nás predsa prekvapil. Ako tím sme s programom podobnej náročnosti skúsenosti nemalo. Zadanie bolo náročné a opieralo sa o teoretické znalosti z predmetov IFJ a IAL – oba predmety nám však tieto znalosti v plnej miere poskytli.

Všetci sme sa zhodli na tom, že nás projekt aj napriek neľahkej implementácii a problémov, ktoré sme museli riešiť, veľa naučil a poskytol nám cenné skúsenosti do študijného aj profesionálneho života.

Konečný automat



Operator = < ; > Operator2 = >= ; <= ; !=

LL-tabuľka

	package	main	FUNC	ID	(	)	{	}	,	:=	Expr	=	if	else	for	;	return	int_value	float_value	string_value	int	float64	string	ε
START	1																							
PROLOG	2																							
PROG			3																					4
FUNCTION			5																					
ARGS				6																				
RETURN_F					10		11																	
S_BODY				16				17					16		16		16							
ARG				9																				
ARG_NEXT						7			8															
RETURN_TYPES																					12	12	12	
STATEMENT				18 19 22									20		21		23							
DATA_TYPE																					51	52	53	
RETURN																					13	13	13	
RETURN_NEXT						15			14															
DEFINITION				24																				
ASSIGN				25																				
IF													32											
FOR															33									
F_CALL				39																				
RETURN_S																	44							
DATA_TYPE																					51	52	53	
ID_NEXT									28			29												
FUNC_OR_EXPR				27							26													
FOR_DEFINITION				34												ε, 35								
FOR_ASSIGN				36			38																	
PARAMS				40		41												40	40	40				
RET_EXPT				46				46			45		46		46		46							
EXPR_NEXT				31				31	30				31		31		31							
DEFINITION				24																				
F_ASSIGN				37																				
TERM				50														47	48	49				
TERM_NEXT						43			42															
EXPR_NEXT							31		30															

1. START → PROLOG PROG
2. PROLOG → package main
3. PROG → FUNCTION PROG
4. PROG → *eps*
5. FUNCTION → func id (ARGS) RETURN_F {S_BODY }
6. ARGS → ARG ARG_NEXT
7. ARG_NEXT → *eps*
8. ARG_NEXT → , ARG ARG_NEXT
9. ARG → id DATA_TYPE
10. RETURN_F → (RETURN_TYPES)
11. RETURN_F → *eps*
12. RETURN_TYPES → RETURN RETURN_NEXT
13. RETURN → DATA_TYPE
14. RETURN_NEXT → , RETURN RETURN_NEXT
15. RETURN_NEXT → *eps*
16. S_BODY → STATEMENT S_BODY
17. S_BODY → *eps*
18. STATEMENT → DEFINITION
19. STATEMENT → ASSIGN
20. STATEMENT → IF
21. STATEMENT → FOR
22. STATEMENT → F_CALL
23. STATEMENT → RETURN_S
24. DEFINITION → id := <expr>
25. ASSIGN → id ID_NEXT = FUNC_OR_EXPR
26. FUNC_OR_EXPR → <expr> EXPR_NEXT
27. FUNC_OR_EXPR → F_CALL
28. ID_NEXT → , id ID_NEXT
29. ID_NEXT → *eps*
30. EXPR_NEXT → , <expr> EXPR_NEXT
31. EXPR_NEXT → *eps*
32. IF → if <expr> { S_BODY } else { S_BODY }
33. FOR → for FOR_DEFINITION ; <expr> ; FOR_ASSING { S_BODY }
34. FOR_DEFINITION → DEFINITION
35. FOR_DEFINITION → *eps*
36. FOR_ASSING → F_ASSIGN
37. F_ASSIGN → id ID_NEXT = <expr> EXPR_NEXZ
38. FOR_ASSIGN → *eps*
39. F_CALL → id(PARAMS)
40. PARAMS → TERM TERM_NEXT
41. PARAMS → *eps*
42. TERM_NEXT → , TERM TERM_NEXT
43. TERM_NEXT → *eps*
44. RETURN_S → return RET_EXPR
45. RET_EXPR → <expr> EXPR_NEXT
46. RET_EXPR → *eps*
47. TERM → int_value
48. TERM → float_value
49. TERM → string_value
50. TERM → id
51. DATA_TYPE → int
52. DATA_TYPE → float64
53. DATA_TYPE → string

Precedenčná tabuľka

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