FAKULTA INFORMAČNÝCH TECHNOLÓGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Projektová dokumentácia

Implementácia prekladača imperatívneho jazyka IFJ22

Tím xbukas00, varianta TRP

Jozef Michal Bukas	<**xbukas00**>	25 %
Samuel Stolárik	<xstola03>	25 %
Adam Bezák	<xbezak02>	25 %
Vojtěch Novotný	<xnovot1t>	25 %

7. decembra 2022

Obsah

1 Úvod

Cieľom tohto projektu bolo vytvoriť prekladač pre jazyk **IFJ22**, ktorý je podmnožinou jazyka **PHP**. Prekladač načíta zdrojový kód zo štandardného vstupu, ten následne preloží do medzikódu **IFJcode22** alebo v prípade chyby vráti chybový návratový kód.

2 Základná štruktúra prekladača

2.1 Lexikálny analyzátor (Scanner)

Cieľom tejto analýzy je načítať podmnožinu jazyka **IFJ22** a konvertovať tento kód na **tokeny**, ktoré sú posielané syntaktickej analýze. Ďalej táto analýza odhaľuje nepovolené znaky alebo reťazce nepodporované jazykom **IFJ22**.

Token je uložený ako štruktúra skladajúca sa z diskriminantu a `Union`. Diskriminant určuje akého typu je daný token a taktiež určuje ako daný `Union` interpretovať. `Union` ukladá informácie ako číselnú hodnotu, reťazec alebo desatinné číslo.

Scanner bol implementovaný ako **deterministický konečný automat**. Diskriminant sa určuje na základe koncového stavu daného automatu. Nasleduje využitie funkcie `switch` na určenie v ktorej hlavnej vetve sa automat aktuálne nachádza. Ďalej nasledujú jednoduché funkcie `if`, ktoré určujú ďalšie chovanie automatu.

Ak automat nie je schopný vykonať žiadny prvotný prechod (z počiatočného stavu start do vetvy) alebo počas čítania narazí na nevalídnu postupnosť znakov, tak vracia návratový chybový kód. Pri spracovaní vstupného súboru nastáva špeciálny prípad, kedy sa na začiatku súboru musí skontrolovať správnosť prologu. Tento stav sa ošetril pomocou špeciálneho módu funkcie `getToken`, ktorá pri prvom volaní kontroluje len prolog a po úspešnom dokončení kontroly sa pomocou statickej premennej prepne do štandardného módu a vráti token prolog.

2.2 Syntaktický analyzátor

Pre väčšinu syntaktickej analýzy využívame spôsob **rekurzívneho spádu**, ktorého implementácia priamo odráža jednotlivé pravidlá gramatiky. Toto však neplatí pre analýzu výrazov, kde využívame implementovanú pomocou algoritmu na prevod výrazu z **in-fixovej** notácie na **post-fixovú**, v ktorom ale navyše kontrolujeme, či dvojica aktuálny a predošlý terminál môžu po sebe syntakticky nasledovať.

Syntaktická analýza ďalej ukladá postupne analyzované terminály do **n-árneho stromu**, v ktorom simuluje štruktúru použitých gramatických pravidiel. Výrazy do tohto stromu vkladá už prevedené do **post-fixovej** notácie.

2.3 Sémantický analyzátor

Sémantická analýza pracuje so **stromom terminálov a neterminálov** vygenerovaný syntaktickou analýzou. Keďže jeho formát kopíruje pravidlá vytvorenej **LL gramatiky**, sémantická analýza postupne spracováva strom a vždy podľa prvého listu v rade sa rozhodne, ktorá funkcia pre kontrolu sémantický chýb sa zavolá.

Ak pri prechode stromu sémantická analýza narazí na pretypovanie, tak vloží do abstraktného syntaktického stromu neterminál značiaci pretypovanie, čo neskôr využije generátor kódu.

Problémom pri návrhu bola možnosť jazyka **IFJ22** volať funkciu predtým ako bola definovaná. Tento problém je vyriešený **dvojitým prechodom stromu**. V prvom prechode sa v strome nájdu všetky definície funkcií a pridajú sa do tabuľky symbolov. V druhom prechode sa vykoná zvyšok sémantických kontrol.

Problém rozdielnych kontextov v programe v ktorých sú definované premenné a funkcie je riešený pomocou štruktúry `VECTOR`, teda sémantická analýza pracuje s vektorom tabuliek symbolov, kedy najvrchnejšia tabuľka je vždy tá najaktuálnejšia.

2.4 Generátor kódu

Preklad výrazu do výsledného kódu je robený **post-order traverzovaním stromu** daného výrazu (výrazy sú uložené v syntaktickom strome v **post-fix** formáte).Všetky výrazy sú spracované na **zásobníku**, s výnimkou výrazu pre konkatenáciu, ktorý pošle obidva reťazce zo zásobníku do dvoch rezervovaných premenných (*RAX*, *RBX*) a výsledok sa následne pošle späť na zásobník.

Počas generácie sa objavili 2 problémy. Obidva nastali pri funkcií `while loop`. Prvý problém nastal, pretože `while loop` môže dynamicky zmeniť typ premennej. To sme vykonali tak, že po vyvolaní výrazu v podmienke sme dynamicky zisťovali typ výsledku a použili sme správnu funkciu na zistenie pravdivostnej hodnoty. Taký istý postup bol použitý aj v podmienkach pre funkcie `if`. Ďalší problém nastal tiež pri premenných, pretože tie musia byť definované mimo `while loop`. V opačnom prípade by interpret spadol nad každým `while loop` ktorý prebehne viac ako 1 krát. Vyriešili sme to tým, že generátor kódu urobí prechod funkcie, aby zistil, ktoré premenné v nej potenciálne budú definované, pričom ich definuje na začiatku funkcie. V hlavnom tele programu tieto premenné definuje pred `while loop`.

3 Súborová štruktúra prekladača

3.1 Lexikálna analýza

- *scanner.c/h*

3.2 Syntaktická analýza

- *syntactic_analysis.c/h*

3.3 Sémantická analýza

- *semantic_analysis.c/h*
- *semantic_expresion.c*

3.4 Generátor kódu

- *codegen.c/h*
- *codegen_context.c/h*

3.5 Dátové štruktúry

- *vector.c/h*
- *vector_ord.h*
- *symtable.c/h*
- *syntax_tree.c/h*
- *error.c/h*

4 Špeciálne dátové štruktúry

4.1 Tabuľka symbolov

Implementovaná ako **tabuľka s rozptýlenými položkami**. Na rozlišovanie kontextov je následné použitý vektor tabuliek symbolov. Štruktúra na uloženie jedného záznamu v tabuľke je pomerne komplikovaná, ale pomáha šetriť miesto v pamäti. Na dosiahnutie tohoto efektu bol použitý `Union`.

Štruktúra `htab_item` sa skladá z ukazateľa na ďalšie synonymum a zo štruktúry `htab_pair_t`, kde sú uložené všetky informácie o zázname (kľúč, typ symbolu, príznak redefinície, reprezentujúci `Union`)

4.2 N-árny strom

Pre simuláciu abstraktného syntaktického stromu sme sa rozhodli použiť dátovú štruktúru **n-árny strom**, ktorý sme implementovali s využitím konceptu takzvaného **Left child - Right sibling tree**[1]. Tento koncept využíva prevodu **n-árneho stromu na binárny strom** a to nasledovne. Všetky uzly na jednej úrovni sú previazané ukazateľom Right sibling a teda tvoria jednosmerne viazaný zoznam, ukazateľ na prvý uzol tohto zoznamu je uložený v spoločnom otcovi všetkých uzlov zoznamu, pod názvom **Left child**. Na prvý pohľad môže takto vytvorený strom vyzerať, kvôli svojej hĺbke, neoptimálne. Je ale dôležité poznamenať, že pre naše účely nie je potrebné v takomto strome vyhľadávať, teda nám jeho hĺbka neprekáža. Skôr nám vyhovuje, že v takto usporiadanom strome vieme jednoducho simulovať pravidlá gramatiky použité pri syntaktickej analýze.

4.3 VECTOR

Zásobníky pre rôzne typy sú definovane cez makrá `DEFINE_VEC_*`. Zásobníky sú definovane originálne pre každý typ, aby sme mohli využiť typový systém C, ktorý síce nie je ten najefektívnejší (napríklad bez problémov pretypuje `unsigned long long` na typ `double` a späť, čím stratí všetky dáta nad prvými 53 bitmi), ale zachytáva viacej chýb ako keby sme mali len jeden druh zásobníku, ktorý drží `void*`.

4.4 VECTOR_ORD

Keďže pre niektoré dátové typy, ktoré môže držať vektor sa dá definovať **koncept poradia**, máme makrá, ktoré definujú funkcie, ktoré tento koncept potrebujú. Makra definujú 2 funkcie na triedenie zásobníku: jednu, ktorá používa stabilný triediaci algoritmus (*insertion sort* [2]) a druhú, ktorá používa nestabilný triediaci algoritmus (*quicksort* [3]). Makrá taktiež definujú 2 funkcie pre hľadanie v zásobníku: prvá funguje aj pre **netriedený zásobník**, zatiaľ čo druhá používa **binárne vyhľadávanie** a teda vyžaduje, aby bol vektor zoradený. Interne je použitý algoritmus na hľadanie prvku **najviac vľavo**. Táto verzia je použitá, lebo makro definuje aj funkciu na vloženie objektu do zásobníku na pozícií takej, aby zásobník zostal zoradený. Táto funkcia taktiež používa tuto modifikovanú verziu binárneho vyhľadávania.

4.5 ERROR

Makrá potrebné na definíciu a prácu s polymorfným typom error (tento typ je inšpirovaný typom `Result`[4] z programovacieho jazyka **Rust**). Typ error neimplementuje funkciu `bind/flatMap` a teda teoreticky není **monad**[5], ale v princípe sa tak správa. Error je implementovaný cez koncept **Tagged union**[6], kde error je buď nejaký typ alebo error. Výhoda tohto typu je že funkcia, ktorá vracia `error(T)`, môže vrátiť validnú hodnotu typu T alebo signalizovať chybový stav pre akýkoľvek typ T.

[1] `https://en.wikipedia.org/wiki/Left-child_right-sibling_binary_tree`
[2] `https://en.wikipedia.org/wiki/Insertion_sort.`
[3] `https://en.wikipedia.org/wiki/Quicksort`
[4] `https://doc.rust-lang.org/std/result/`
[5] `https://en.wikipedia.org/wiki/Monad_(functional_programming)`
[6] `https://en.wikipedia.org/wiki/Tagged_union)`

5 Práca v tíme

5.1 Rozdelenie práce

Jozef Michal Bukas: Lexikálna analýza, Tabuľka symbolov, Dokumentácia, Sémantická kontrola výrazov
Samuel Stolárik : Syntaktická analýza, testy
Vojtěch Novotný : Sémantická analýza
Adam Bezák : Generátor kódu, Špeciálne dátové štruktúry

5.2 Vývojový cyklus

Zloženie tímu bolo veľmi rýchle, keďže traja sme už boli dohodnutý v predstihu a nájsť štvrtého člena nebol problém.

Na prvom stretnutí si každý vybral čo chce robiť. Každý týždeň sme sa stretli, osobne alebo online cez discord, kde sme prebrali, ako kto od posledného stretnutia postúpil, a čo bude náplň práce do najbližšieho stretnutia a prebrali prípadné sme implementačné otázky alebo dotazy. Ak dáka otázka nevyžadovala prítomnosť celého tímu, tak sa riešila individuálne.

Snažili sme sa pracovať priebežne a nenechávať veci na poslednú chvíľu, čo sa nám až na menšie výnimky darilo.

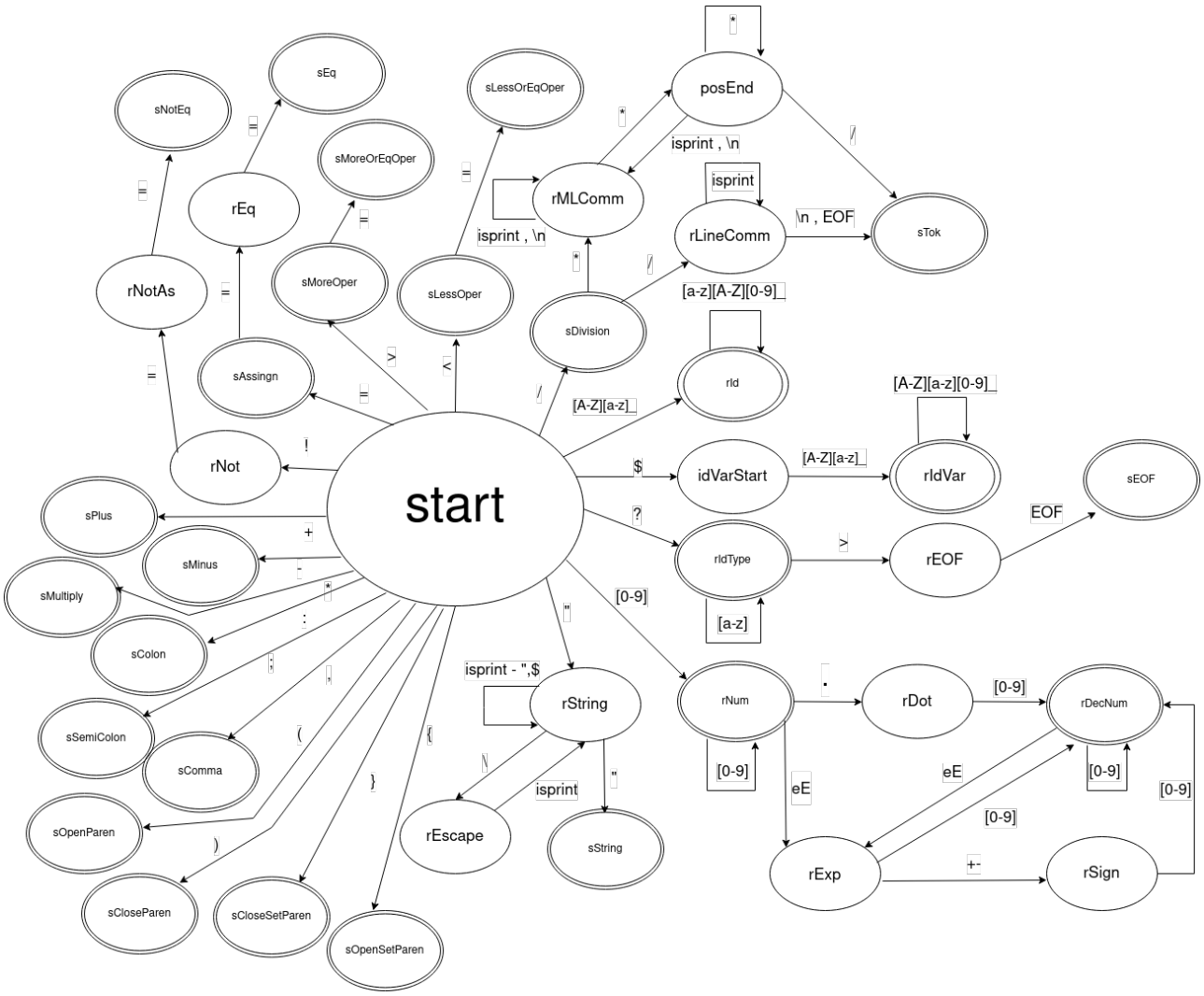
Na správu verzií sme používali git, kde sme pracovali tak, aby sme predchádzali merge konfliktom, čo urýchlilo samotné spájanie programu do celku.

6 Záver

Vďaka skorému započatiu práce na tomto projekte sme s jeho riešením nemali až taký problém. Na všetko sme mali dostatok času a preto nás ani zmena nášho návrhu vôbec neposunula vzad a v pokoji sme ju vykonali. Vďaka dostatku času sme si vyskúšali implementovať nové algoritmy, ktoré boli pre nás novinkou a toto bola dobrá príležitosť ich otestovať v praxi. Keďže sme sa už skoro všetci poznali, tak sme vedeli naše prednosti i slabé stránky, ale hlavne ako pracovať spolu v tíme.

Celkovo hodnotíme prácu na tomto projekte ako pozitívnu skúsenosť, ktorá nás v profesionálnom živote určite posunula ďalej a prakticky nám ukázala nielen fungovanie prekladača v praxi ale aj v čom sa ešte môžeme zdokonaľovať ako jednotlivci ale i členovia tímu.

7 Prílohy



Obr. 1: Konečný automat

	$*, /$	$+, -, .$	Rel op	$===, !===$	$($	$)$	i	\$
$*, /$	$>$	$>$	$>$	$>$	$<$	$>$	$<$	$>$
$+, -, .$	$<$	$>$	$>$	$>$	$<$	$>$	$<$	$>$
Rel op	$<$	$<$	$>$	$>$	$<$	$>$	$<$	$>$
$===, !==$	$<$	$<$	$<$	$>$	$<$	$>$	$<$	$>$
$($	$<$	$<$	$<$	$<$	$<$	$=$	$<$	
$)$	$>$	$>$	$>$	$>$		$>$		$>$
i	$>$	$>$	$>$	$>$		$>$		$>$
\$	$<$	$<$	$<$	$<$	$<$		$<$	

Tabuľka 1: Precedenčná tabuľka

```
1. <PROLOG> -> prolog <PROG>

2. <PROG> -> endOfFile
3. <PROG> -> function identOfFunc ( <PARAMS> ) : <TYPE> { <BODY> } <PROG>
4. <PROG> -> if <EXPR> { <BODY> } else { <BODY> } <PROG>
5. <PROG> -> while <EXPR> { <BODY> } <PROG>
6. <PROG> -> <EXPR> ; <PROG>
7. <PROG> -> identOfFunc ( <ARGS> ) ; <PROG>
8. <PROG> -> identOfVar = <RVAL> ; <PROG>
9. <PROG> -> return <RET_VAL> ; <PROG>

10. <BODY> -> <STATEMENT> <BODY>
11. <BODY> -> ''

12. <STATEMENT> -> if <EXPR> { <BODY> } else { <BODY> }
13. <STATEMENT> -> while <EXPR> { <BODY> }
14. <STATEMENT> -> <EXPR> ;
15. <STATEMENT> -> identOfFunc ( <ARGS> ) ;
16. <STATEMENT> -> identOfVar = <RVAL> ;
17. <STATEMENT> -> return <RET_VAL> ;

18. <RVAL> -> <EXPR>
19. <RVAL> -> identOfFunc ( <ARGS> )

20. <PARAMS> -> <TYPE> identOfVar <PARAMS_NEXT>
21. <PARAMS> ->''

22. <PARAMS_NEXT> ->, <TYPE> identOfVar <PARAMS_NEXT>
23. <PARAMS_NEXT> ->''

24. <ARGS> -> <ARG_TYPE> <ARGS_NEXT>
25. <ARGS> ->''

26. <ARG_TYPE> -> identOfVar
27. <ARG_TYPE> -> <TERM>

28. <ARGS_NEXT> ->, <ARG_TYPE> <ARGS_NEXT>
29. <ARGS_NEXT> ->''

30. <RET_VAL> -> <EXPR>
31. <RET_VAL> ->''

32. <TERM> -> null
33. <TERM> -> float_lit
34. <TERM> -> int_lit
35. <TERM> -> string_lit

36. <TYPE> -> identOfType
37. <TYPE> -> identOfTypeN
```

Obr. 2: LL gramatika

	prolog	EOF	function	identOfFunc	(	)	:	{	}	if	else	=	while	;	,
prolog	1														
prog		2	3	7						4			5		
body				10					11	10			10		
statement				15						12			13		
rval				19											
params						21									
params_next						23									22
args						25									
arg_type															
args_next						29									28
ret_val														31	
term															
type															

	identOfVar	expr	return	null	float_lit	int_lit	string_lit	identOfType	identOfTypeN
prolog									
prog	8	6	9						
body	10	10	10						
statement	16	14	17						
rval		18							
params								20	20
params_next									
args	24			24	24	24	24		
arg_type	26			27	27	27	27		
args_next									
ret_val		30							
term				32	33	34	35		
type								36	37

Tabuľka 2: LL tabuľka