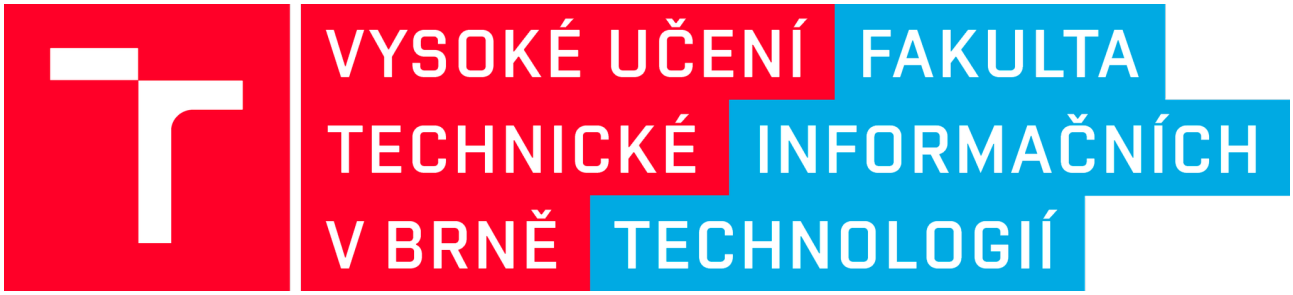


Dokumentácia projektu IFJ

Implementácia prekladača imperatívneho jazyka IFJ22

Tým xharma05 , varianta TRP

7. decembra 2022

Richard Harman	**xharma05**	**25%**
Jasmína Csalová	xcsalo00	25%
Terézia Hundáková	xhunda01	25%
Marek Špirka	xspirk01	25%

Obsah

1 Úvod

Cieľom tohto projektu bolo vytvoriť prekladač v jazyku C, ktorý má za úlohu premeniť vstupný kód zapísaný v zdrojovom jazyku IFJ22, ktorý je zjednodušená verzia jazyka PHP. Tento kód sa následne spracovával v niekoľkých častiach, ktoré neskôr popisujeme. Výstupom tohto prekladača je kód v jazyku IFJcode22.

2 Práca v tíme

2.1 Rozdelenie práce

- Richard Harman – Lexikálna analýza, Sémantická analýza, Generátor kódu
- Marek Špirka – Generátor kódu
- Jasmína Csalová – Syntaktická analýza, Dokumentácia
- Terézia Hundáková – Syntaktická analýza, Sémantická analýza

2.2 Postup práce

Nakoľko projekty tohto rozsahu sú časovo aj technicky náročné, dbali sme na častú komunikáciu medzi tým, ako sme tento prekladač programovali. Ako komunikačné kanály sme využívali Messenger a v niektorých prípadoch Discord. Dohodli sme sa na verzovacom systéme Git, ktorý sme používali prostredníctvom klienta GitKraken. Pri závažnejších problémov s implementáciou tohto projektu sme namiesto online komunikácií zvolili osobné stretnutie, kde sa prebrali všetky problémy a našli na nich riešenia.

3 Implementácia prekladača

Prácu na implementácií projektu sme si rozdelili do základných častí, ktoré sú:

- Lexikálna analýza
- Syntaktická analýza
- Sémantické analýza
- Generátor

3.1 Lexikálna analýza

Lexikálna analýza je jedna z prvých častí prekladaču, ktorá je implementovaná na základe lexikálnych pravidiel jazyka IFJ22 a funguje na základe nami navrhnutého konečného stavového automatu (Príloha 8.1). Ten prečíta znak zo súboru, vyhodnotí ho a určí nasledujúci krok v stavovom automate. Jej hlavné úlohy sú čítanie znakov zo vstupu a ich preklad na tokeny a nájdenie lexikálnych chýb. Naša lexikálna analýza taktiež vynecháva komentáre a biele znaky, avšak kontroluje ich, kde sú žiadané. Podľa stavového automatu rozdeľujeme jednotlivé postupnosti znakov na lexémy – menšie lexikálne časti. Rozpoznané lexémy vstupujú do nášho programu obalené v štruktúre – token. Táto štruktúra obsahuje informácie o dátovom type tokenu a o jeho dátach, ktoré neskôr napomáhajú v správnom vyhodnocovaní syntaktickej aj sémantickej analýzy. Jednotlivé tokeny odovzdávame do syntaktickej analýzy v prípade lexikálnej bezchybnosti. V prípade, ak by bola lexikálna bezchybnosť porušená,

znamená to, že s daným tokenom pokračovať nemôžeme a voláme lexikálnu chybu. Následne sa validné tokeny ukladajú do listu. Lexikálna analýza je implementovaná v súboroch `scanner.c` a `scanner.h`. List do ktorého tokeny vkladáme je implementovaný v súboroch `list.c` a `list.h`.

3.2 Syntaktická analýza

Naša syntaktická analýza je implementovaná rekurzívne podľa pravidiel nami navrhnutej LL gramatiky (Príloha 8.2) a aplikuje sa na každý prijatý token až po úspešnej lexikálnej analýze. Tokeny prijíma z listu tokenov po jednom. V prípade, ak narazíme na výraz, spúšťame jeho kontrolu formou modulu `expr`, ktorý vykoná kontrolu výrazu a vyhodnotí, či operácie sú vo validnom poradí. Syntaktiku spúšťame v main pomocou prologu. Implementované v `parser.c`, `parser.h`.

3.3 Sémantická analýza

Vykonáva sa po úspešnej syntaktickej analýze. Spúšťa sa v `main` pomocou antilogu. Vzostupným spôsobom sa vykonávajú potrebné sémantické kontroly a naplňujeme tabuľku rozpýtelných prvkov validnými elementami. Nami vytvorená štruktúra element obsahuje časti programu, potrebné informácie o nich a skladá sa z tokenov. Kontrolujú sa volania nedefinovaných funkcií, použitia neexistujúcich premenných a tiež volanie funkcie so zlým počtom parametrov. V prípade, ak narazíme na výraz, opäť spúšťame kontrolu výrazu v module `expr.c`, ktorý vykoná sémantickú kontrolu výrazu a daný výraz vyhodnotí podľa jeho dátových typov, uloží si potrebný výsledný dátový typ z výrazu a vráti ho späť na kontrolu do `parser.c` . Sémantická analýza je implentovaná v súboroch `parser.c`, `parser.h`, `expr.c` a `expr.h`.

3.3.1 Tabuľka rozptýlených prvkov

Tabuľka je napĺňaná na začiatku sémantickej analýzy a rozdeľuje program na časti, takzvané elementy, implementované našou štruktúrov element. Elementom môže byť napríklad definícia funkcie, volanie funkcie, premenná, if, while cyklus, a tak ďalej. Tabuľka rozpýtelných prvkov je implementovaná v `symtable.c` a `symtable.h`.

3.4 Generátor

Po úspešnom prejdení predošlých troch časti nášho prekladača sa začína generovať kód. Kód sa generuje z vytvorenej tabuľky symbolov, ktorá má v sebe uložené jednotlivé elementy programu. Generátor programu sme si rozdelili na 3 menšie podčasti. Generátor programu je implementovaný v súboroch `generator.c` a `generator.h`

3.4.1 Generovanie Built-in funkcií

Ako prvé sme si v generátore implementovali built-in funkcie, ktorých kód sa následne vypíše pri ich použití. Pri vytvárani týchto funckií sme sa bližšie oboznámili ako funguje kód IFJcode22 a ako by mala vyzerať jeho štruktúra.

3.4.2 Generovanie definovaných funkcií

Druhou najťažšou úlohou bolo pre nás vytvoriť definície nových funkcií z IFJ22 a generovať dané funckie do výsledného kódu. Túto časť sme si rozdelili na menšie podčasti, ktoré nám uľahčili prácu a zjednodušili výsledné riešenie. Na začiatku sa vygeneruje tzv. `header` funckie, ktorý definuje danú funkciu a do lokálnych premmených uloží premenné, ktoré sú do funkcie posielané cez dátový zásobník. Následne sa vygeneruje telo danej funkcie, v tele sa volajú funkcié, ktoré toto delo vytvárajú a to je napríklad: volanie inej funkcie, priradenie hodnoty do premmenej, if a while cykly. Všekty

tieto časti berieme z tabuľky symbolov, ktorá sa nachádza v `symtable.c` a `symtable.h`. Na koniec funckie sa vytlačí dátový zásobník hodnota, ktorú funkcia vracia a funkciu ukončíme príkazom `return`.

3.4.3 Generovanie mainu

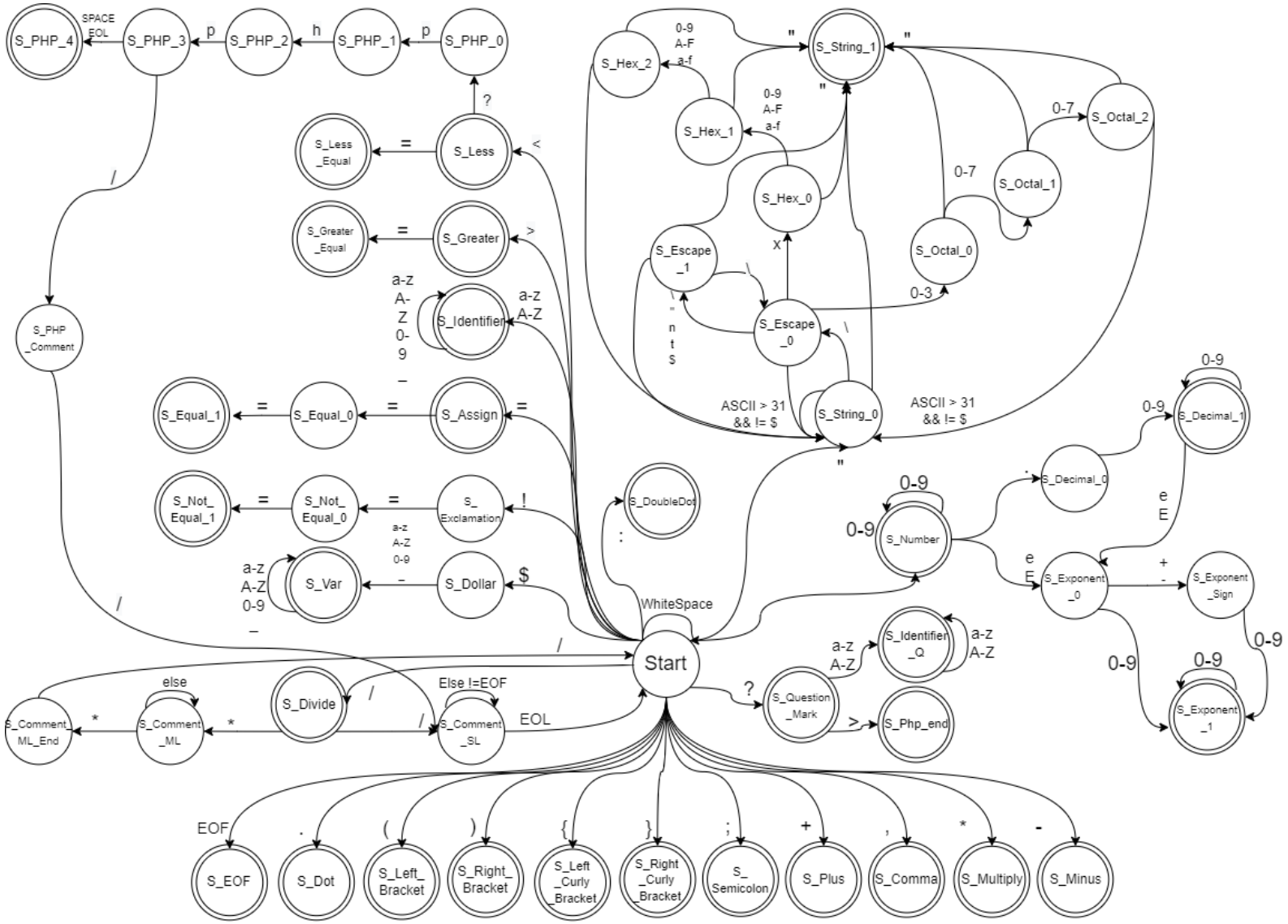
Posledná časť bola generovanie main funckie, na ktorú program skočí hneď po zapnutí programu pomocou príkazu `jump`. Je to pre nás hlavná funkcia, ktorá následne riadi čo sa bude generovať. Telo funkcie obsahuje rovnaké časti ako pri generované definovanch funckií.

4 Záver

Zo začiatku sa tento projekt javil celkom náročný, avšak postupom času sme prišli na riešenie problémov. Tento projekt nás naučil lepšie pracovať v tíme a zlepšil nám komunikáciu medzi kolegami. Taktiež sme si upevnili poznatky s verzovacím systémom Git. Fungovanie a implementácia prekladača je delikátna záležitosť, ktorá si rozhodne žiada ďalšie štúdium. Z tohto projektu si berieme veľa nových poznatkov, ktoré určite využijeme v ďalších projektov, alebo v práci.

5 Prílohy

5.1 Diagram konečného automatu lexikálnej analýzy



5.2 LL-gramatika

1. <program> -> <?php EOL declare(strict_types=1); < body >
2. < body > -> < stmt > < body >
3. < body > -> <end>
4. <end> -> ?> EOF
5. <end> -> EOF
6. < func > -> function func_ID (<args>) <ret_type> { <stmt_list> }
7. < stmt _list > -> < stmt > <stmt_list >
8. <stmt_list > -> ε
9. < stmt > -> < func >
10. < stmt > -> [expresion];
11. < stmt > -> $var_ID = [expresion];
12. < stmt > -> while ([expresion]) { <stmt_list> }
13. < stmt > -> if ([expresion]) { stmt_list } <else_stmt>
14. < stmt > -> return [expresion];
15. < args > -> <data_type> $var_ID <arg_def> <arg_list>
16. < args > -> ε
17. < arg_list > -> , <data_type> $var_ID> <arg_list>
18. <arg_list > -> ε
19. < ret_type > -> : <data_type>
20. < ret_type > -> ε
21. < else_stmt > -> else { <stmt_list > }
22. < else_stmt > -> ε
23. < data_type > -> null
24. < data_type > -> int
25. < data_type > -> float
26. < data_type > -> string
27. < arg_def > -> = literal
28. < arg_def > -> ε

5.3 Gramatika výrazov

EXPR → EXPR + EXPR

EXPR → EXPR * EXPR

EXPR → EXPR – EXPR

EXPR → EXPR / EXPR

EXPR → EXPR . EXPR

EXPR → EXPR < EXPR

EXPR → EXPR > EXPR

EXPR → EXPR <= EXPR

EXPR → EXPR >= EXPR

EXPR → EXPR === EXPR

EXPR → EXPR !== EXPR

EXPR → ! EXPR

EXPR → (EXPR)

EXPR → id ( )

EXPR → id (EXPR)

EXPR → id (EXPR , EXPR)

EXPR → id (EXPR , EXPR , EXPR)

EXPR → literal

5.4 Tabuľka pre overenie správnosti výrazov

	value	+	-	*	/	<	>	<=	>=	===	!==	,	(	)	.	FUNC	END
value	X	<	<	<	<	<	<	<	<	<	<	<	<	<	<	X	<
+	>	=	<	<	<	<	<	<	<	<	<	X	<	<	>	>	<
-	>	<	=	<	<	<	<	<	<	<	<	X	<	<	>	>	<
*	>	<	<	=	<	<	<	<	<	<	<	X	<	<	>	>	<
/	>	<	<	<	=	<	<	<	<	<	<	X	<	<	>	>	<
<	>	<	<	<	<	>	>	>	>	>	>	X	<	>	<	>	<
>	>	<	<	<	<	>	>	>	>	>	>	X	<	>	<	>	<
<=	>	<	<	<	<	>	>	>	>	>	>	X	<	>	<	>	<
>=	>	<	<	<	<	>	>	>	>	>	>	X	<	>	<	>	<
===	>	<	<	<	<	>	>	>	>	>	>	X	<	>	<	>	<
!==	>	<	<	<	<	>	>	>	>	>	>	X	<	>	<	>	<
,	>	X	X	X	X	X	X	X	X	X	X	=	<	<	<	>	<
(	>	<	<	<	<	<	<	<	<	<	<	X	=	X	<	>	<
)	>	<	<	<	<	<	<	<	<	<	<	<	<	=	>	>	<
.	>	<	<	<	<	>	>	>	>	>	>	>	<	>	=	>	<
FUNC	X	<	<	<	<	<	<	<	<	<	<	<	<	<	<	X	<
END	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	X