VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace k projektu s názvem:

Implementace překladače imperativního jazyka IFJ23

do předmětů: IFJ, IAL

4.12.2023

Tým xcuprm01, varianta vv-BVS

Čupr Marek	xcuprm01	34 %
Halva Jindřich	xhalva05	33 %
Hrabovský Vojtěch	xhrabo18	33 %
Červený Jindřich	xcerve31	0 %

Obsah

1 Úvod

Tento text slouží jako dokumentace k projektu, do předmětů IFJ a IAL, který byl vypracován týmem „xcuprm01“. V tomto projektu bylo cílem implementovat imperativní překladač pro jazyk IFJ23, jež je zjednodušenou podmnožinou jazyka Swift. Program načte zdrojový kód a přeloží jej do jazyka IFJcode23.

2 Začátky a práce v týmu

K našemu plus jsme tým složili velmi brzy po tom, co byl projekt zadán. Zanedlouho po individuálním nastudování zadání, jsme uspořádali osobní schůzku s hlavním cílem ujasnit si neshody v pochopení zadání, rozdělit si práci a také se lépe poznat.

Na počátku pro nás bylo obtížné se v projektu zorientovat, dívali jsme na něj jako na celek. O to těžší pro nás bylo přemýšlení nad spravedlivým rozdělením práce. V den první schůzky jsme si nakonec práci jednoznačně rozdělit nedokázali, a tak jsme se rozhodli řešit tento problém v průběhu.

K velkému úskalí jsme došli v moment, kdy nám jeden ze členů oznámil, že s námi spolupráci ruší. Zbyly jsme tedy tři, a tak jsme si museli práci přerozdělovat znovu.

2.1 Konečné rozdělení práce

xcuprm01	konečný automat, lexikální analýza, návrh gramatiky, syntaktická a sémantická analýza, testování
xhalva05	Precedenční tabulka, syntaktická a sémantická analýza pro výrazy, testování, organizace, dokumentace
xhrabo18	tabulka symbolů, návrh gramatiky, syntaktická analýza, testování, generování kódu

Pro rozdělení bodů po třetinách jsme se rozhodli, protože každý odvedl velké množství práce. Nevíme, zda se dá říct stejné množství, každá část totiž obnáší něco jiného, jiné dovednosti, znalosti a úsilí. Takovéto rozdělení se nám zdá ale spravedlivé. Kapitán dostane za práci s odevzdáváním archivů a reprezentaci týmu o procento více.

Člen týmu „xcerve31“ obdrží celkem 0 % bodů, a to kvůli neaktivitě.

3 Návrh a Implementace

V této časti budou stručně popsány dílčí části projektu. Podkapitoly jsou seřazeny v pořadí, v jakém jsme k jednotlivým částem přistupovali. U každé části je uvedeno, s jakými soubory souvisí.

3.1 Automat

Celá naše práce započala vytvořením konečného automatu, který přehledně zobrazí, jaké tokeny bude lexikální analyzátor generovat. První návrh automatu jsme museli v průběhu prací párkrát upravit, ale nešlo o velké změny. Náš konečný automat naleznete v příloze.

3.2 Tabulka Symbolů

Tabulku symbolů jsme implementovali díky znalostem z předmětu IAL. Vybrali jsme si variantu projektu, kde jsme měli za úkol využít výškově vyváženého binárního stromu. Implementaci tabulky symbolů naleznete v souborech *symtable.c* a *symtable.h.* Klíčem každého prvku je název identifikátoru. Každý uzel ve stromu obsahuje informaci o tom, zda se jedná o strom proměnných nebo funkcí, podle toho pak závisí, která ze struktur je vyplněna informacemi. Uzel totiž obsahuje strukturu pro proměnné a jinou strukturu pro funkce. U každé struktury nás zajímají jiné informace. Každý uzel také uchovává ukazatel na levý a pravý podstrom.

Tabulky symbolů jsme ukládali do zásobníku, který je implementován v souborech *symtable_stack.c* a *symtable_stack.h*. Po každé, když program vstoupí do podmínky nebo cyklu, na vrcholu zásobníku vznikne nová lokální tabulka symbolů. Až podmínka nebo cyklus skončí, lokální tabulka je vymazána.

3.3 Lexikální analýza

Lexikální analýzu jsme vytvářeli podle navrhnutého konečného automatu. Tato část se promítne do souborů *scanner.c* a *scanner.h.*

Hlavním výstupem této části je struktura token, struktura předávána dál do programu, kterou vytváří funkce *get_token*. Tato struktura obsahuje informace jako například konkrétní typ tokenu (*TOKEN_INT, TOKEN_EOF*…), hodnotu aktuálního tokenu (například pokud je tokenem integer, v podstruktuře token_value, nalezneme konkrétní číslo, které tento token představuje, pro řetězce jsou jeho znaky uchovány ve struktuře *Dynamic_Str_T,* která je implementována v souborech *dynamic_str.c* a *dynamic_str.h*), údaj o tom, zda token může nabývat hodnoty nil (u *Int?* například), údaj o počtu zanoření v blokových komentářích, a pravdivostní hodnotu, která indikuje, zda je exponent u float záporný či ne.

V této části jsme se zdrželi s blokovými komentáři, víceřádkovými řetězci a také escape sekvencí *\u{}*.

3.4 Syntaktická analýza

Většinově se v projektu provádí syntaktická analýza pomocí rekurzivního sestupu, řídí se námi navrhnutou LL-Gramatikou, kterou naleznete v příloze. Tento způsob analýzy jsme využili v souborech *parser.c* s hlavičkovým souborem *parser.h*. Jinak jsme ale zpracovávali výrazy, na ty jsme využili metodu precedenční analýzy. Implementace analýzy pro výrazy se rozložila do souborů exp_parser.c a exp_stack.c (plus hlavičkové soubory). Tento způsob je řízen precedenční tabulkou, kterou také naleznete v příloze. Hlavní soubor syntaktické analýzy *parser.c* volá postupně tokeny z lexikálního analyzátoru a provádí různé kontroly. Pokud se ve zdrojovém kódu narazí na výraz, volá funkci *expression_parse.* Ta se snaží daný komplexní výraz zredukovat na jediný prostý výraz. Pracuje na základě precedenční tabulky, která udává kdy je vhodné výraz již redukovat nebo kdy se má naopak daný token vložit na zásobník (implementován v *exp_stack.c*). Pokud lze výraz bez problému zredukovat do jediného výrazu, syntaktická analýza výrazu proběhla bez chyby.

Zásadní součástí parseru je struktura *Parser_T*, která uchovává informace důležité pro správný chod překladače, jako je například aktuální token či aktuální pravidlo, ve kterém se překladač nachází. Zajímavou součásti je pravdivostní hodnota *EOL_skip*, při nastavení této hodnoty na true, program přehlíží konce řádků ve zdrojovém souboru, to je vhodné pro metodu rekurzivního sestupu. Pro výrazy se naopak v naší implementaci hodí i znak konce řádku, tudíž do funkce *expression_parse* vstupuje program s *EOL_skip = false*.

Tato část projektu nám zabrala nejvíce času.

3.5 Sémantická analýza

Sémantická analýza se provádí zároveň s analýzou syntaktickou, nebo přesněji, tyto analýzy se v podstatě prolínají. Soubory, ve kterých je tato analýza implementována, tedy odpovídají soborům z kapitoly výše. Kontrolují se například datové typy při definici proměnných a při přiřazeních, dále zda funkce vracejí požadovaný datový typ atd. Při této analýze i analýze syntaktické je velmi důležitá komunikace mezi soubory *parser.c* a *exp_parser.c*.

Například když ověřujeme návratový typ funkce, *parser.c* pošle informace přes strukturu *Parser_T* do funkce *expression_parse*, ta zpracuje výraz za klíčovým slovem *return*. Pokud je výraz validní a je stejného typu jako požadovaný typ uvedený v *Parser_T*, sémantická kontrola typů proběhla v pořádku.

3.6 Generování kódu

Generace kódu probíhá zároveň se syntaktickou a sémantickou analýzou. Generace je tedy rozdělena mezi příslušné soubory *parser.c, exp_parser.c.* Část IFJcode23 je generována samostatně v souboru *code_gen.c* s hlavičkovým souborem *code_gen.h*. Tento soubor se stará primárně o vypisování kódu pro podmíněný výraz *if* a příkaz cyklu *while.* Dále také zajišťuje převod řetězce znaků do formy podporované IFJcode23 a o správné rozhodování, zda použít proměnnou z lokálního nebo globálního rámce.

V *exp_parser.c* se zpracovávají aritmetické výrazy, v *parser.c* zejména generování návěští a skoků, definice funkci, jejich volání, předávání parametrů a kontrola rámců. Parametry se předávají pomocí dočasného rámce a návratová hodnota je předána přes vnitřní zásobník interpretu. Soubor *parser.c* také generuje kód pro vestavěné funkce jazyka IFJ23.

4 Verzování

K verzování našeho projektu jsme využívali webovou aplikaci GitHub, která nám vždy umožnila přístup k aktuální verzi. Pokud někdo pracoval na projektu a následně chtěl uložit svoji práci, dal vědět ostatním členům týmu. Nikdy se nám nestalo, že bychom si nějakou verzi přepsali jiným commitem, tudíž jsme nepraktikovali verzování pomocí jiných větví (branches). Snažili jsme se, aby více lidí nepracovalo v jeden moment na stejném souboru.

Zálohování starších verzí jsme prováděli manuálně. Každý si před vložením novější verze stáhl tu starší, aby se v případě nouze mohl projekt vrátit zpět.

5 Závěr

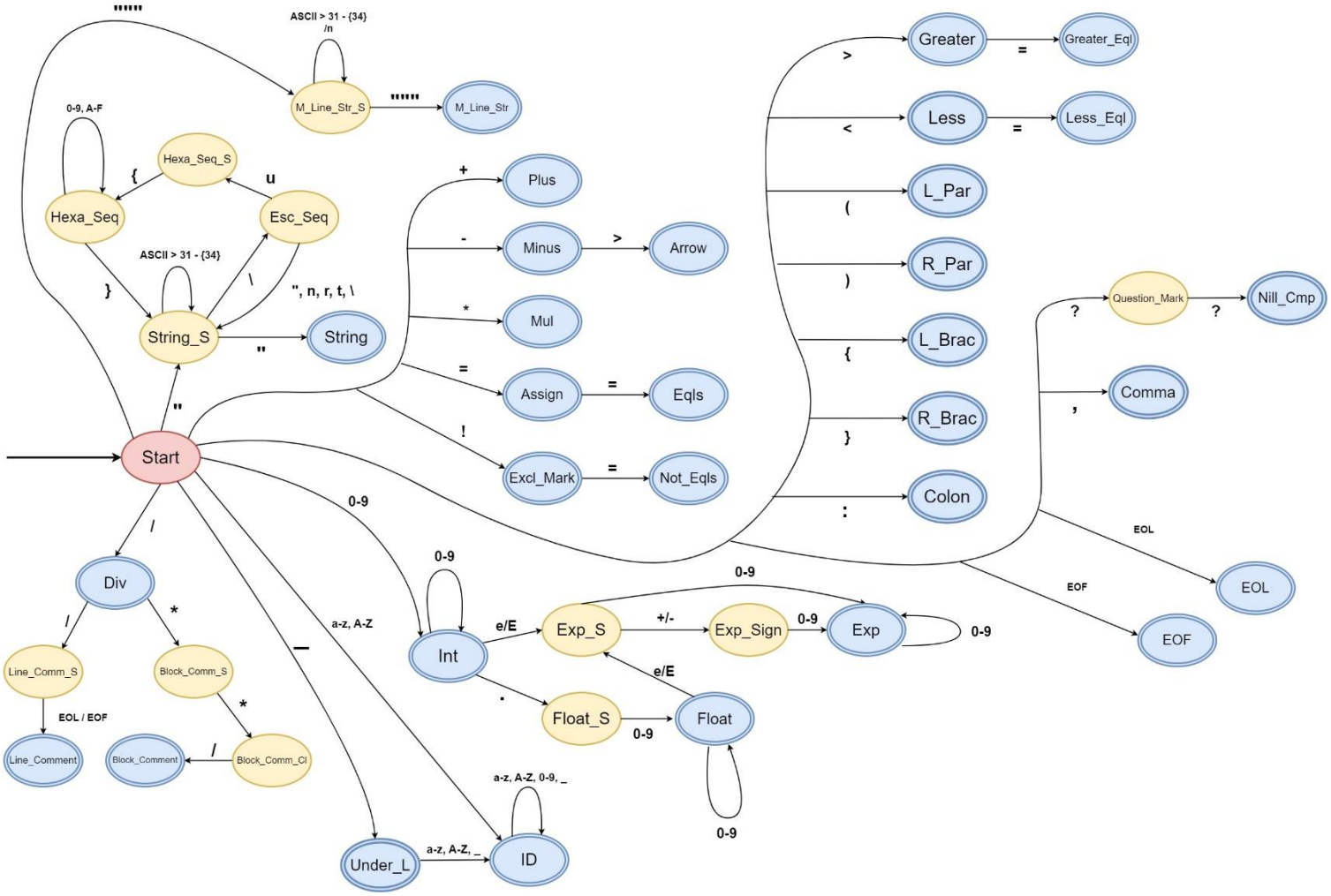
Na začátku jsme s týmem diskutovali o zadání projektu a všichni jsme byli velmi zaskočeni z jeho rozsahu. Postupem času, když jsme si rozdělili práci a každý se soustředil především na to svoje, jsme konečně viděli, že je reálné projekt zhotovit. Bohužel jsme se k řešení a samotnému programování projektu dostali později, než jsme plánovali. To nám samozřejmě lehce uškodilo. Největší zvrat však přišel v moment, kdy se od nás odpojil jeden ze členů týmu. Na každého se tak nahromadilo více práce, než jsme zprvu počítali. Se situací jsme se nakonec ale vypořádali vcelku dobře.

Velmi nám pomohlo pokusné odevzdání, na které jsme odeslali projekt v podstatě bez generace výsledného kódu, na tuto část jsme neměli již před prvním odevzdáním čas. Avšak důležité pro nás bylo především hodnocení lexikální, syntaktické a sémantické analýzy. Na těchto částech jsme totiž strávili spoustu času, a ověřili jsme si, že postupujeme správným směrem. To nám dodalo optimistické myšlení. Do dalšího pokusného odevzdání jsme zvládli už i část generátoru kódu, a poopravili jsme nějaké chyby v analýzách.

Nakonec jsme projekt vypracovali celý až na LL-tabulku, na kterou nám bohužel nezbyl čas. Víme také o tom, že generátor kódu je u nás slabší část a rozhodně za ni nečekáme sto procent. Na druhou stranu podle pokusných odevzdání se zdá, že analýzy se nám vcelku vydařili a hodiny práce jež jsme jim věnovali se vyplatili. Na závěr je potřeba říct, že nám projekt přinesl nové zkušenosti, které můžeme uplatnit do budoucna (například už jen práce s gitem pro nás byla přínosná, do této doby jsme verzování takovýmto způsobem nepraktikovali), což bylo jistě hlavním cílem.

6 Příloha

6.1 Konečný automat



Obrázek 1: Konečný automat

6.2 LL-Gramatika

1. <program> ➜ <statement_list> <program_eof>
2. <statement_list> ➜ <function_definition> <statement_list>
3. <statement_list> ➜ <statement> <statement_list>
4. <statement_list> ➜ ε
5. <function_definition> ➜ func function_ID (<params_list>) <return_type> { <statement_list> }
6. <params_list> ➜ <param_name> param_ID : <type> <params_list_n>
7. <params_list> ➜ ε
8. <param_name> ➜ param_NAME
9. <param_name> ➜ _
10. <type> ➜ String
11. <type> ➜ String?
12. <type> ➜ Int
13. <type> ➜ Int?
14. <type> ➜ Double
15. <type> ➜ Double?
16. <params_list_n> ➜ , <param_name> param_ID : <type> <params_list_n>
17. <params_list_n> ➜ ε
18. <return_type> ➜ → <type>
19. <return_type> ➜ ε
20. <statement> ➜ <variable_definition>
21. <statement> ➜ <assignment>
22. <statement> ➜ <if_statement>
23. <statement> ➜ <while_statement>
24. <statement> ➜ <function_call>
25. <statement> ➜ <function_return>
26. <variable_definition> ➜ <var_def_type> var_ID <var_type> <var_assign>
27. <var_def_type> ➜ let
28. <var_def_type> ➜ var
29. <var_type> ➜ : <type>
30. <var_type> ➜ ε
31. <var_assign> ➜ = expression
32. <var_assign> ➜ ε
33. <assignment> ➜ var_ID = expression
34. <assignment> ➜ var_ID = <function_call>
35. <function_call> ➜ function_ID (<input_params_list>)
36. <input_params_list> ➜ <input_param_name> term <input_params_list_n>
37. <input_params_list> ➜ ε
38. <input_param_name> ➜ param_NAME :
39. <input_param_name> ➜ ε
40. <input_params_list_n> ➜ , <input_param_name> term <input_params_list_n>
41. <input_params_list_n> ➜ ε
42. <if_statement> ➜ if expression { statement_list } else { statement_list }
43. <while_statement> ➜ while expression { statement_list }
44. <function_return> ➜ return <return_option>
45. <function_return> ➜ ε
46. <return_option> ➜ expression
47. <return_option> ➜ ε
48. <program_eof> ➜ EOF

6.3 Precedenční tabulka

	TOKEN																
STACK		!	*	/	+	−	==	!=	<	>	<=	>=	??	(	)	id	$
	!		>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
	*	<	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
	/	<	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
	+	<	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
	−	<	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
	==	<	<	<	<	<							>	<	>	<	>
	!=	<	<	<	<	<							>	<	>	<	>
	<	<	<	<	<	<							>	<	>	<	>
	>	<	<	<	<	<							>	<	>	<	>
	<=	<	<	<	<	<							>	<	>	<	>
	>=	<	<	<	<	<							>	<	>	<	>
	??	<	<	<	<	<							<	<	>	<	>
	(	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
	)	>	>	>	>	>	>	>	>	>	>	>	>		>		>
	id	>	>	>	>	>	>	>	>	>	>	>	>		>		>
	$	<	<	<	<	<	<	<	<	<	<	<	<	<		<	

Tabulka 1: Precedenční tabulka

6.4 Členění implementačního řešení

1. Lexikální analyzátor
    - scanner.c, scanner.h
    - utils.c, utils.h – struktury pro stavy automatu a klíčová slova
    - dynamic_string.c, dynamic_string.h – nafukovací pole pro řetězce
2. Tabulka symbolů
    - symtable.c, symtable.h – implementace tabulky symbolů
    - symtable_stack.c, symtable_stack.h – zásobník tabulek symbolů
3. Syntaktická a sémantická analýza
    - parser.c, parser.h
    - exp_parser.c, exp_parser.h
    - exp_stack.c, exp_stack.h – zásobník pro výrazy
4. Generátor kódu
    - parser.c – generace kódu
    - exp_parser.c – generace kódu pro aritmetické operace
    - code_gen.c, code_gen.h – generace kódu pro if a while
5. Ostatní
    - error.c, error.h – zpracování chyb
    - ifj2023.c – začátek programu (main)
    - Makefile – překlad