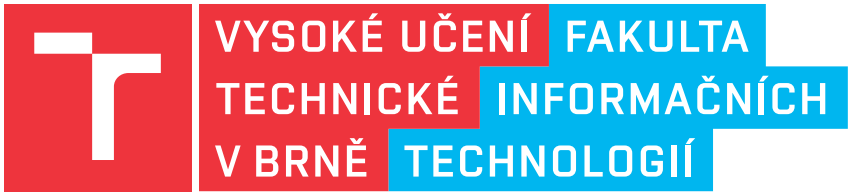


Projektová dokumentácia

Implementácia prekladača imperatívneho jazyka IFJ21

Tým 082, varianta II

Andrei Shchapaniak	**(xshcha00)**	25 %
Andrej Binovsky	(xbinov00)	25 %
Zdenek Lapes	(xlapes02)	25 %
Richard Gajdosik	(xgajdo33)	25 %

8. decembra 2021

Obsah

1 Úvod

2 Návrh a implementácia

2.1 Lexikálna analýza

Scanner slúži pre lexikálnu analýzu. Je implementovaný ako deterministický konečný automat, ktorý rozpoznáva všetky prichádzajúce tokeny. Uchováva informácie o tom či se jedná o komentár, identifikátor, textové bloky, relačné operátory alebo iné validné, poprípade nevalidné tokeny u ktorých nastane lexikálna chyba 1. V prípade validného tokenu na zaklade koncového stavu se vyplnia nasledujúce informácie v štruktúre token_t:

- type – typ načítaného tokenu
- keyword – keď typ je T_KW, do premennej keyword sa uloží odpovedajúca hodnota
- attr – hodnota tokenu
- attr.num_i – integer number
- attr.num_f – double number
- attr.id – ostatné tokeny

V prípade že sa narazí na komentár je celý textový blok, ktorý je podľa správnej lexikálnej štruktúry chápaný ako komentár zahodený.

2.2 Syntaktická analýza

Parser je hlavným modulom prekladača, protože komunikuje se všetkými ostatnými modulmi a riadi celú funkčnosť prekladača. Syntaktická analýza sa vykonáva zhora dolu metódou rekurzivného zostupu. Syntaktická analýza dostava od lexikálneho analyzátoru postupne tokeny, ktoré následne musia mať presnú syntaktickú štruktúru a postupnosť podla pravidiel LL-gramatiky.

2.2.1 Spracovanie výrazov pomocou precedenčnej syntaktickej analýzy

2.3 Sémantická analýza

- Sémantické chyby pre nekompatibilitu typu priradenia, návratových hodnôt a predaných argumentov do funkcií se detekujú pomocou dvoch polí (ľavá strana typu a pravá strana typu). Po spracovaní určitého pravidla se vykoná porovnanie podľa sémantiky jazyka IFJ21.
- Kompatibilita typu vo výraze sa detekuje tým istým spôsobom, ale je použitý dve premenné typu char.
- Sémantické chyby pre nedefiníciu, redefiníciu sa detekujú pomocou tabuliek symbolov. Globálna tabulka symbolov je určená pre názvy funkcií. Pre názvy premenných je vytvorený zásobník tabuliek symbolov. Dôvodom implementácie zásobníka je riešenie problému s totožnými názvami premenných v rôznych rámcoch.

2.4 Generovanie cielového kódu

Generovanie cieľového kódu IFJcode21 je implementovaný ako samostatný modul, ktorý je riadený syntaxou. Komponenty modulu su volané v parseri na základe pravidiel LL-gramatiky. Cieľový kód sa generuje priamo bez tvorby trojadresného kódu, ktorý sme nevytvárali na základe neoptimalizácie cieľového kódu.

2.4.1 Implementácia výpisu cieľového kódu

Na zaistenie výpisu cieľového kódu len za podmienky bezchybnej analýzy zapisujeme cieľový kód do dvoch textových blokov – definície funkcií a volanie funkcií. Tieto dva textové bloky po úspešnej analýze skonkatenujeme a vypíšeme na štandardný výstup.

2.4.2 Generovanie – Deklarácie premenných

Deklarácie premenných a ich možný konflikt názvov (na základe výskytu toho istého názvu v rôznych rámcoch) sme implementovali vďaka obojsmernému radu v ktorom sa ukladá adresa elementu tabulky symbolov s príslušným identifikátorom. Element tabulky obsahuje unikátne číslo premennej, ktorý zaisťuje jedinečnosť názvu premennej.

2.4.3 Generovanie – funkcie

Volanie funkcií je zaistené vygenerovaním kódu, ktorý predá funkcii argumenty pomocou dočasného rámcu, následne je vygenerovaný kód pre zavolanie funkcie. Pre správnu funkčnosť volanej funkcie je ihneď na začiatku generovaný kód, ktorý z dočasného rámca vytvôri lokálny rámec funkcie a pre všetky argumenty ktoré boli funkcii predané vytvori promenné s názvami podľa parametrov funkcie. Následne se generuje kód tela funkcie.

2.4.4 Generovanie – výrazy

Generovanie kódu pre výraz sa začne vykonávať ihned po jeho redukcii. V priebehu redukcii je výraz zapísaný do obojstranného radu v postfixovom formáte. Jednotlivé elementy v rade nesú všetky potrebné informácie na generovanie výrazu - typ operátora, názov premennej či hodnotu konštanty. Generátor generuje len inštrukcie kódu Ifjcode21 ktoré využívajú zásobník. To znamená že hodnoty, medzivýsledky a následne výsledok výrazu sú uložené na zásobník.

2.4.5 Generovanie – podmienky a cykly

Pre generovanie podmienok a cyklov využívame náveštia, ktoré sú taktiež reprezentované unikátnym číslom a názvom funkcie kde sa nachádzajú. Na zabránenie redeklarácie premenných sa telo cyklu zapisuje do dvoch rôznych textových blokov. Vyskytnuté deklarácie zapisujeme naďalej do bloku definícií funkcií. No zvyšný kód tela cyklu zapisujeme do tretieho pomocného textového bloku. Následne po vygenerovaní celého cyklu tieto dva bloky skonkatenujeme.

2.5 Prekladový systém

2.5.1 CMake

CMake je multiplatformný nástroj na preklad zdrojových kódov. Nástroj sme vybrali na základe preferencií všetkých členov tímu. CMake nám predovšetkým pomáhal kompilovať a testovať výsledný program. Pravidlá pre preklad sú napísané v súbore CMakeLists.txt a po spustení nastroja CMake je automaticky vygenerovaný súbor Makefile. Na testovanie sme používali google testy, ktoré sme prekladali výhradne pomocou CMaku.

2.5.2 Skripty

Pre účely testovania boli vytvorené shellovské skripty. Jeden rozsiahly script, ktorý uľahčoval a automatizoval testovanie všetkých častí projektu. Taktiež sme vytvorili script pre preklad projektu a čistenie prebytočných súborov vzniknutých v dôsledku kompilácie projektu či testovacích suborov.

2.5.3 GNU Make

Zo zadania bolo požadované aby odovzdaný projekt obsahoval Makefile, ktorý s príkazom "make" preloží zdrojové súbory projektu a s "make clean" zmazal prebytočné súbory vzniknuté v dôsledku kompilácie. Tento nástroj nám taktiež pomáhal zabaliť celý projekt do jedného archívu zip.

3 Špeciálne algoritmy a dátové štruktúry

3.1 Tabuľka s rozptýlenými položkami

Túto dátovú štruktúru sme si zvolili vďaka časovej zložitosti (je reprezentovaná medzi O(1) až O(n)) a so skúsenosti štruktúry z predmetov IAL a IJC. Veľkosť tabuľky jsme zvolili 101. Ako unikátny kľúč pre prístup k dátam v tabulke slúži názov identifikátoru a názvy funkcií. Každý záznam v tabuľke obsahuje informácie o identifikátore. U premenných je uchovávaná aj informácia o hĺbke (redefinícia v zanorenejšom rámci kódu). Modul je implementovaný v súboroch symtable.h a symtable.c.

3.2 Pole rozptýlených tabuliek

TODO Modul je implementovaný v súboroch symstack.c a symstack.h

3.3 Obojsmerný rad

Obojsmerný rad je kombinácia zásobníka a radu. Je možné do neho vkladať aj odoberať dáta z oboch strán. Implementovali sme ho ako samostatný modul pre viac častí projektu. Je využívaný v generovaní kódu, kde slúži na uchovávanie postfixového výrazu či identifikátorov. Taktiež obsahuje informáciách o parametroch, argumentoch a navratovych hodnatách funkcií. Modul je implementovaný v súboroch queue.c a queue.h.

3.4 Dynamický reťazec

Pre uchovanie vygenerovaného kódu počas prekladu a prácu s identifikátormi sme vytvorili štruktúru string_t. Pre obsluhu štruktúry sme vytvorili pomocné funkcie ako alokácia/dealokácia štruktúry, odstraňovanie, pridávanie a konkatenacia reťazcov. Modul je implementovaný v súboroch str.c a str.h.

4 Práca v týmu

Ihneď pri skladaní tímu sme všetci rozumeli že sa očakáva pravidelná a skorá práca na projekte čo sa nám nakoniec aj podarilo. Každý na projekte pracoval vždy s predstihom a darilo sa nam dodržiavať deadliny, ktoré sme si stanovili.

4.0.1 Komunikácia a spôsob práce v tíme

Pre komunikáciu sme používali výhradne komunikačnú platformu Discord, ktorý funguje na rovnakom princípe ako platforma Slack, ktorá sa používa profesionálne účely. Na danej platforme boli vytvorené komunikačné vlákna v ktorých boli založené "TODO" či error listy. Taktiež tam prebiehala bežná komunikácia či hlasové rozhovory s možnosťou zdieľania obrazovky vďaka čomu sme mohli vyriešiť mnoho problémov digitálne a tým aj veľmi rýchlo. Avšak aj napriek dobrej digitálnej komunikácii sme sa snažili mať čo najviac osobných stretnutí.

4.0.2 Verzovací systém a vývojové prostredie

Ako vývojové prostredie sme využili Clion a Vim. Vývoj prebiehal na platformách MacOs, Linux, Windows, no testovanie prebiehalo len na operačnom systému Linux. Ako verzovací systém sme použili git spolu s portálom GitHub.

4.1 Rozdelenie práce medzi členmi tímu

Andrei:

- Lexikálna, sémantická a obecná syntaktická analýza
- Organizácia a kontrola práce nad projektom
- Tabulka symbolov

Richard:

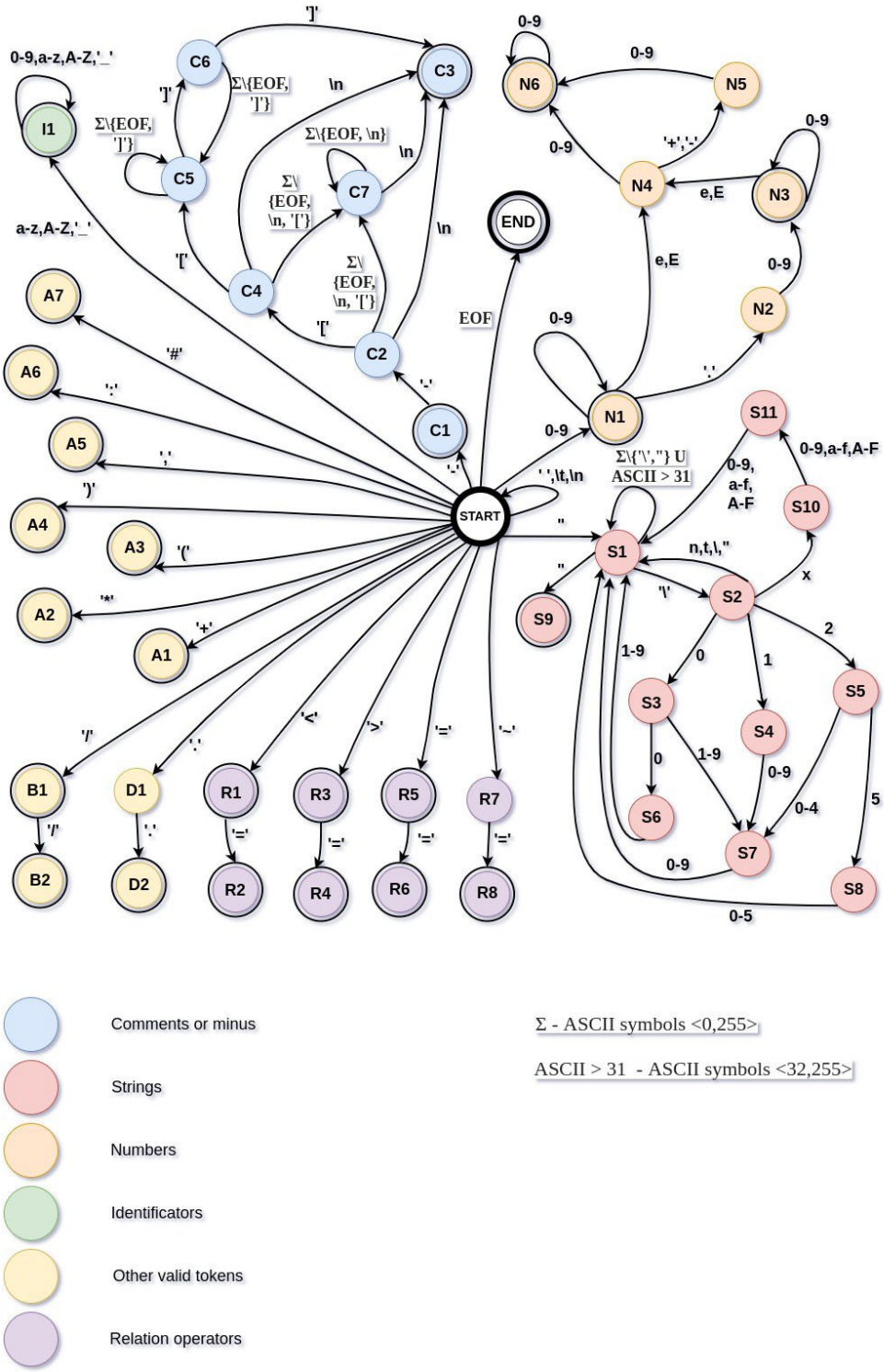
- Syntaktická a sémantická analýza pre výrazy
- Precedenčná tabulka
- Prezentácia

Zdenek a Andrej:

- Generovanie kódu
- Automatizácia testovania
- Google testy, tvorba testov
- Dokumentácia

5 Záver

Diagram konečného automatu specifikujúceho lexikálny analyzátor



Obr. 1: Diagram konečného automatu specifikující lexikální analyzátor

LL – gramatika

1. <**prolog**> → `require t_string` <**prog**>
2. <**prog**> → `global id :  function (` <**arg_T**> `)` <**ret_T**> <**prog**>
3. <**prog**> → `function id (` <**arg**> `)` <**ret_T**> <**stmt**> `end` <**prog**>
4. <**prog**> → `id (` <**param**> `)` <**prog**>
5. <**prog**> → `EOF`
6. <**arg_T**> → <**type**> <**next_arg_T**>
7. <**arg_T**> → ε
8. <**next_arg_T**> → `,` <**type**> <**next_arg_T**>
9. <**next_arg_T**> → ε
10. <**ret_T**> → `:` <**type**> <**next_ret_T**>
11. <**ret_T**> → ε
12. <**next_ret_T**> → `,` <**type**> <**next_ret_T**>
13. <**next_ret_T**> → ε
14. <**arg**> → `id :` <**type**> <**next_arg**>
15. <**arg**> → ε
16. <**next_arg**> → `, id :` <**type**> <**next_arg**>
17. <**next_arg**> → ε
18. <**type**> → `integer`
19. <**type**> → `number`
20. <**type**> → `string`
21. <**type**> → `nil`
22. <**stmt**> → `if` <**expr**> `then` <**stmt**> `else` <**stmt**> `end` <**stmt**>
23. <**stmt**> → `while` <**expr**> `do` <**stmt**> `end` <**stmt**>
24. <**stmt**> → `local id :` <**type**> <**def_var**> <**stmt**>
25. <**stmt**> → `return` <**expr**> <**next_expr**> <**stmt**>
26. <**stmt**> → `id` <**fork_id**> <**stmt**>
27. <**stmt**> → ε
28. <**def_var**> → `=` <**one_assign**>
29. <**def_var**> → ε
30. <**one_assign**> → `id (` <**param**> `)`

31. <one_assign> → <expr>
32. <param> → <param_val> <next_param>
33. <param> → ε
34. <param_val> → id
35. <param_val> → <term>
36. <term> → t_string
37. <term> → t_integer
38. <term> → t_number
39. <term> → nil
40. <next_param> → , <param_val> <next_param>
41. <next_param> → ε
42. <next_expr> → , <expr> <next_expr>
43. <next_expr> → ε
44. <fork_id> → (<param>)
45. <fork_id> → <next_id>
46. <next_id> → , id <next_id>
47. <next_id> → = <mult_assign>
48. <mult_assign> → id (<param>)
49. <mult_assign> → <expr> <next_expr>

LL – tabulka

	require	global	function	id	integer	string	number	nil	t_integer	t_number	t_string	if	while	local	return	:	=	(	,	EOF	$
<prolog>	1																				
<prog>		2	3	4																5	
<arg_T>					6	6	6	6													7
<next_arg_T>																			8		9
<ret_T>																10					11
<next_ret_T>																			12		13
<arg>				14																	15
<next_arg>																			16		17
<type>					18	20	19	21													
<stmt>				26								22	23	24	25						27
<def_var>																	28				29
<one_assign>				30																	31
<param>				32				32	32	32	32										33
<param_val>				34				35	35	35	35										
<term>								39	37	38	36										
<next_param>																			40		41
<next_expr>																			42		43
<fork_id>																	45	44	45		
<next_id>																	47		46		
<mult_assign>				48																	49

Precedenčná tabuľka

1. **E** → i
2. **E** → (**E**)
3. **E** → # **E**
4. **E** → **E** + **E**
5. **E** → **E** − **E**
6. **E** → **E** ∗ **E**
7. **E** → **E** / **E**
8. **E** → **E** // **E**
9. **E** → **E** .. **E**
10. **E** → **E** > **E**
11. **E** → **E** < **E**
12. **E** → **E** >= **E**
13. **E** → **E** <= **E**
14. **E** → **E** == **E**
15. **E** → **E** ~= **E**

	#	*	/	//	+	-	..	<	<=	>	>=	==	~=	(	)	i	$
#	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
*	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
//	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
+	<	<	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
-	<	<	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
..	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
==	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
~=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	e
)	>	>	>	>	>	>	>	>	>	>	>	>	>	e	>	s	>
i	e	>	>	>	>	>	>	>	>	>	>	>	>	e	>	s	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	e	<	e

LEGENDA:
< - insert to stack with shift
> - reduction
= - insert to stack
e - error
s - special case (end of expression)