Dokumentácia k projektu

Implementace překladače imperativního jazyka IFJ21

Tým 16, varianta II

Medvedev Anton	**(xmedve04)**	25%
Verevkin Aleksandr	(xverev00)	25%
Tsiareshkin Ivan	(xtsiar00)	25%
Matej Alexej Helc	(xhelcm00)	25%

6. decembra 2021

Obsah

1 Úvod

Cieľom projektu bolo vytvoriť program v jazyku C, ktorý pracuje so zdrojovým kódom v jazyku IFJ21 (zjednodušená podmnožina jazyka Teal) načítaného zo štandardného vstupu a jeho následným preložením do cieľového jazyka IFJcode21. Program generuje výsledný medzikód na štandardný výstup.

Vybrali sme si druhú variantu zadania podľa, ktorej sme tabuľku symbolov implementovali pomocou tabuľky s rozptýlenými položkami.

2 Implementácia

Prekladač môžme rozdeliť na niekoľko základných modulov:

- **Scanner** – lexikálna analýza
- **Parser** – syntaktická a sémantická analýza
- **Generátor** – generátor kódu cieľového jazyka IFJcode21

K týmto základným modulom sme ešte vytvorili niekoľko menších modulov na zjednodušenie práce, hlavne jednoduchšie testovanie a s tým úzko spojené hľadanie bugov a ich oprava.

2.1 Scanner

Prvá časť, ktorú sme pri tvorbe prekladača implementovali bola lexikálna analýza. Hlavnou funkciou lexikálnej analýzy je spracovanie vstupného súboru. Scanner prechádza súbor znak po znaku a prevádza vstupné znaky na jednotlivé lexémy, ktoré v našom riešení reprezentuje štruktúra `token_struct`, ktorá sa skladá z typu a atribútov. Po načítaní lexému scanner určí jeho typ a atribúty, s ktorými následne pracuje parser. Lexikálny analyzátor je implementovaný ako deterministický konečný automat 1.

V našom riešení projektu sa o prácu scanneru stará primárne funkcia `get_token`, ktorá načíta token a uloží ho do štruktúry `token_struct`. Scanner vracia chybu 1 pokiaľ narazí na znak, ktorý nie je v jazyku IFJ21 definovaný a program je ukončený v opačnom prípade vracia ukazateľ na získaný token.

Pre jednoduchšie spracovanie reťazcov sme implementovali modul `str`, v ktorom sú implementované základné funckie pre prácu s reťazcami, čo nám značne uľahšilo prácu aj v iných častiach projektu.

2.2 Syntaktická analýza

Druhá časť, ktorú sme implementovali bol parser, ktorého súčasťou je aj syntaktická analýza. Pri implementácii sme použili metódu zhora-dole, konkrétne metódu rekurzívneho zostupu.

Syntaktická analýza sa riadi podľa pravidiel LL-gramatiky, kde väčšina pravidiel má svoju vlastnú funkciu. Syntaktická analýza žiada tokeny od lexikálneho analyzátoru (scanneru) pomocou makra `GET_TOKEN`.

Na členenie programu do jednotlivých rámcov (funkcie, cykly, if-else) využívame zásobník s funkciami, implementovaný v `data_stack`. Pri implementácii sme si brali inšpiráciu z projektu v predmete Algoritmy.
Keď narazíme na kľúčové slovo `end` tak odstránime prvok na vrchole zásobníka, čo je vždy najvnútornejší rámec. Lokálne premenné si následne ukladáme do tabuľky symbolov implementovanej v module `symtable`.

2.2.1 LL-Gramatika

1. **header** → require ”ifj21”**program**
2. **program** → global id : function (**params_g**) **rets** end **program**
3. **program** → function id (**params**) **rets body** end **program**
4. **program** → **call program**
5. **program** → ϵ
6. **params_g** → **data_type param_g**
7. **params_g** → ϵ
8. **param_g** → , **data_type param_g**
9. **param_g** → ϵ
10. **params** → id : **data_type param**
11. **params** → ϵ
12. **param** → , id : **data_type param**
13. **param** → ϵ
14. **rets** → : **data_type ret**
15. **rets** → ϵ
16. **ret** → , **data_type ret**
17. **ret** → ϵ
18. **body** → **command body**
19. **body** → ϵ
20. **command** → **def_var**
21. **command** → id **assign**
22. **command** → if **expression** then **body** else **body** end
23. **command** → while **expression** do **body** end
24. **command** → **call**
25. **command** → return **expression**
26. **command** → write (**expression**)
27. **def_var** → local id : **data_type assign**
28. **call** → id (**expressions**)
29. **data_type** → integer
30. **data_type** → number
31. **data_type** → string
32. **data_type** → ni
33. **assign** → **= assigns**
34. **assigns** → id (**expression**)
35. **assigns** → **expression**

2.2.2 Precedenčná syntaktická analýza

Precedenčná syntaktická analýza je použitá pre spracovanie výrazov a je založená na precedenčnej tabuľke. Implementovali sme ju v module `exp_handle`. V našej implementácii používame zásobník z modulu `exp_stack`, na ktorý si zapisujeme jednotlivé tokeny vstupného výrazu. Hlavným cieľom je skrátiť tokeny na zásobníku pomocou pravidiel z precedenčnej tabuľky, ktorú sme zjednodušili keďže $+$ a $-$ alebo $*$, $/$ a $//$ majú rovnakú asociativitu a prioritu. Id v našej tabuľke reprezentuje názov premennej, string, double alebo integer hodnotu.

Hlavnú úlohu v precedenčnej analýze tvorí funkcia `exp_processing`, ktorá má za úlohu zjednodušiť výraz zo zásobníku (modul `exp_stack`). Funckia načíta tokeny až kým nedojde na koniec výrazu alebo nenarazí na nejakú sémantickú chybu kedy program skončí s chybovým hlásením na výstupe.

	+ -	/ * //	(	)	id	relations	#	..	$
+ -	>	<	<	>	<	>	<	>	>
/ * //	>	>	<	>	<	>	<	>	>
(	<	<	<	=	<	<	<	<	ERR
)	>	>	ERR	>	ERR	>	ERR	>	>
id	>	>	ERR	>	ERR	>	ERR	>	>
relations	<	<	<	>	<	ERR	<	<	>
#	>	>	<	>	<	>	>	>	>
..	<	<	<	>	<	>	<	>	>
$	<	<	<	ERR	<	<	<	<	END

Tabulka 1: Precedenčná tabuľka

2.3 Sémantická analýza

Sémantickú analýzu sme implementovali spoločne so syntaktickou a to hlavne v moduloch `expr_handle` a `parser`. Väčšina sémantických kontról spočíva v kontrole počtu parametrov funkcie prípadne počtu návratových typov či typovej kompatibility dátových typov. V prípade narazenia na nejakú zo spomínaných chýb sa program ukončí a vracia konkrétne číslo chyby.

2.4 Tabuľka symbolov

Tabuľku symbolov sme implementovali ako tabuľku s rozptýlenými hodnotami. V našej implementácii sme si brali veľkú inšpiráciu z druhého projektu v predmete Algoritmy kde sme implementovali tabuľku s rozptýlenými hodnotami. Na ukladanie globálnych premenných (názvy funkcií) používame zásobník, kde vrcholom je vždy najvnútornejší (aktuálne spracovávaný rámec). Tabuľku symbolov využívame na ukladanie lokálnych premenných (premenné z aktuálne spracovaváneho rámca) a následne k prístupu k nim z modulov `parser` a `exp_handle`. Kľúč v tabuľke zastupuje názov premennej či funkcie podľa, ktorých následne v tabuľke vyhľadávame pre detekciu sémantických chýb.

2.5 Generátor cieľového kódu

Generátor cieľového kódu sa v našom riešení skladá z funkcií, ktoré volá `parser` a `expr_handle`. Úlohou generátoru je vytvárať cieľový kód **IFJcode21**. V priebehu syntaktických kontról, výsledný kód postupne zapisujeme do dynamického reťazca (štruktúra `string_struct`), ktorý na začiatku inicializujeme. V prípade narazenia na nejakú chybu uvoľnujeme alokovanú pamäť.

Na začiatku spracovania vstupného súboru voláme generátor, ktorý vygeneruje vstavané funkcie a hlavičku (label `main`). Postupne prechádzame celý vstupný súbor pomocou parseru, v ktorom voláme rôzne funkcie generátoru kódu a tým pripisujeme kód do (výstupného) reťazca. V prípade, že sme nenarazili na žiadnu chybu generátor vypíše na štandardný výstup obsah reťazca uloženého v štruktúre `string_struct` kde máme zapísaný cieľový kód.

3 Práca v tíme

Na začiatku riešenia projektu sme si stanovili deň v týždni, kedy sme sa schádzali a diskutovali sme pokrok jednotlivých častí projektu a následne sme si rozdeľovali ďalšiu prácu. V pokročilej fáze sme niektoré zo zložitejších častí projektu implementovali vo dvojiciach, kde sme spoločne diskutovali všetky možné stavy a tým odhalili čo najväčší počet syntaktických a sémantických chýb, ktoré by mohli nastať.

Ako hlavnú komunikačnú službu sme používali **Discord**, kde sme si vytvorili vlastný server. Na discorde prebiehala všetka naša komunikácia a rovnako sa tam odohrávali aj všetky naše schôdzky, kde sme využívali aj možnosť zdieľania obrazovky k lepšiemu vysvetleniu problematiky. Ako verzovací systém sme použili **GIT** hosťovaný na stránke **GitHub**.

3.1 Rozdelenie práce

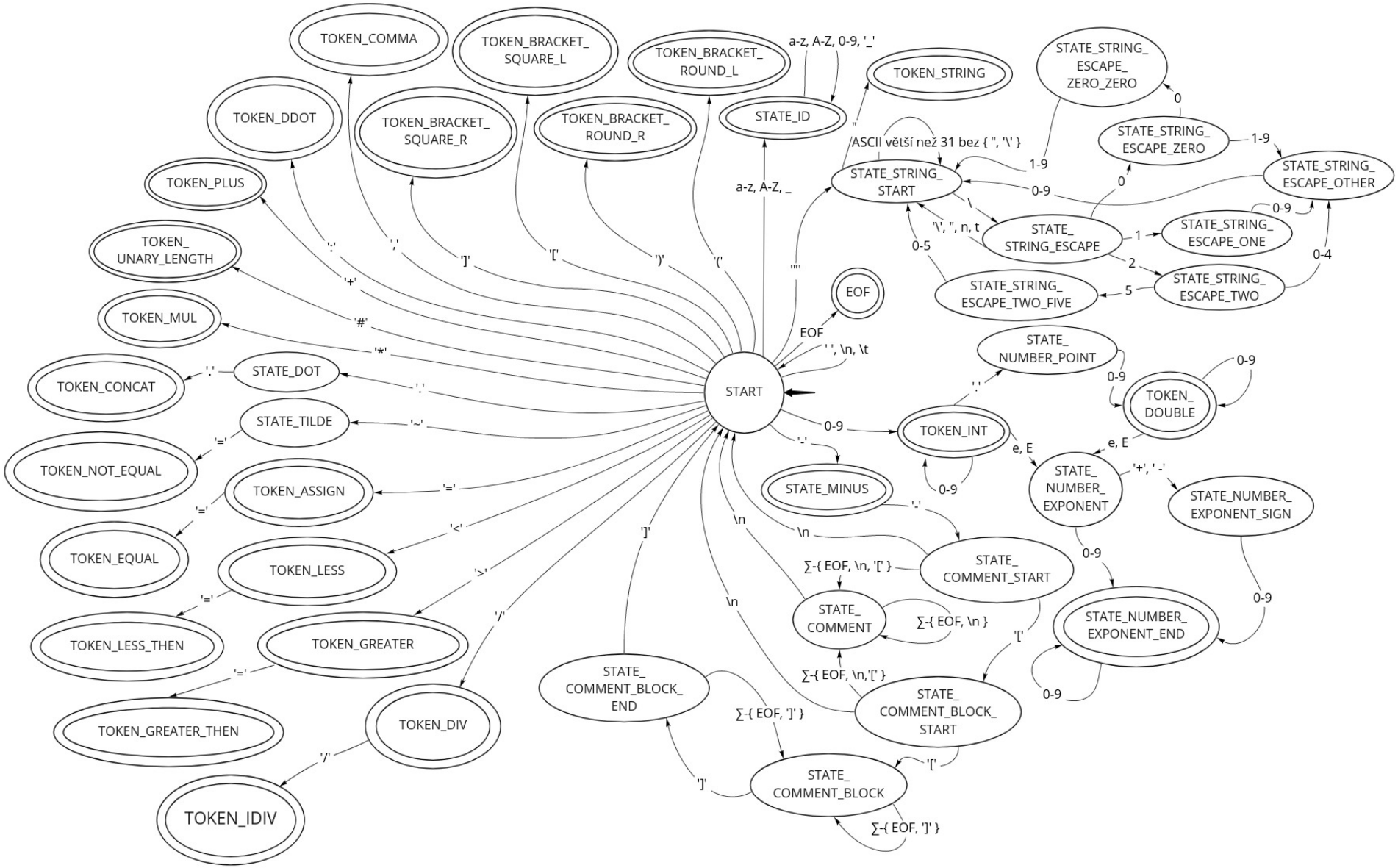
Medvedev Anton	– implementácia syntaktický a sémantický analýzátor, precedenčný analyzátor, testovanie
Verevkin Aleksandr	– implementácia generátoru výsledného kódu a tabuľky symbolov, návrh precedenčnej analýzy
Tsiareshkin Ivan	– návrh lexikálneho analyzátoru a jeho implementácia, implementácia precedenčnej analýzy
Helc Matej Alexej	– implementácia syntaktický a sémantický analýzátor, návrh LL-gramatiky, dokumentácia

4 Záver

Projekt aj napriek jeho náročnosti bol veľmi zaujímavý a prínosný. Začiatok projektu bol trochu náročnejší a vďaka rozsiahlemu zadaniu vyzeral naozaj veľmi náročne ale s pribúdajúcim časom a vedomosťami z prednášok sa projekt pre nás stával stále viac zaujímavejším. Bolo veľmi ťažké začať, pretože sme nemali takmer žiadne vedomosti z oblasti problematiky projektu a preto sme mali na začiatku veľmi voľné tempo, ktoré sme museli v neskoršej fázi projektu dobiehať.

Projekt nám priniesol veľa a to nie len z oblasti vyučovaných predmetov ale hlavne skúsenosti s prácou v tíme a s prácou s verzovacím nástrojom.

Práca v tíme aj napriek počiatočným obavám a menšej jazyokvej bariére vyzerala ukážkovo, komunikovali sme takmer na dennej báze a spoločne sme pristupovali ku všetkým vzniknutým problémom.



Obrázek 1: Deterministický konečný automat

	require	global	:	function	id	(	)	end	if	then	else	while	do	return	write	local	integer	number	string	nil	=	,	$
header	1																						
program			2	3	4	5																	40
params_g							7										6	6	6	6			
param_g							9															8	
params					10		11																
param																						12	
rets			13														14	14	14	14			
ret																						15	
body																							
call					33	34	35																
data_type																							
command					25	38	39	29,32	26	27	28	30	31	36	37								
def_var			18		17											16	19	19	19	19			
assign																					20		
assigns					21	22	23																

Tabulka 2: LL-tabuľka

5 Použité zdroje

1. Prednášky z predmetu Formálne jazyky a prekladače
2. Prednášky z predmetu Algoritmy
3. Aplikáciu miro.com na tvorenie konečného automatu