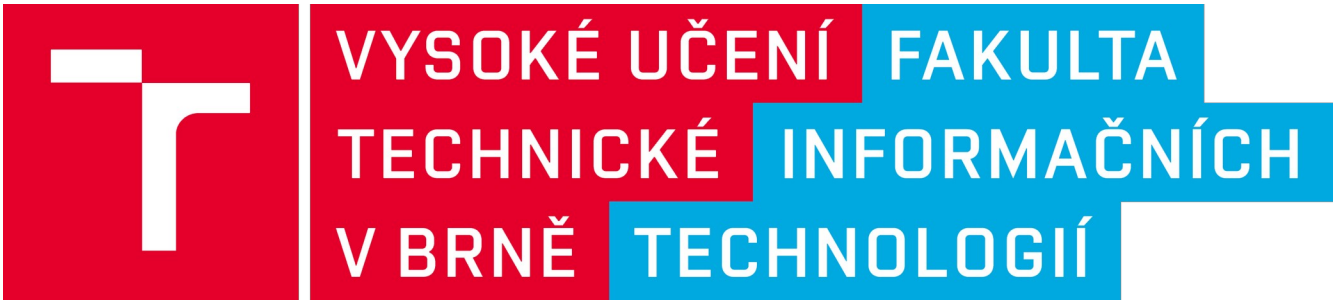


Překladač jazyka IFJ24

Dokumentace projektu

Tým xjenisf00, Varianta vv-BVS

Filip Jenis	xjenisf00	- vedoucí	25
Hugo Bohácsek	xbohach00		28
Josef Ambruz	xambruj00		19
Štefan Dubnička	xdubnis00		28

Podporovaná rozšíření:

FUNEXP

Práce v týmu:

Práci na projektu jsme si rozdělili na základě plánu, na kterém jsme se domluvili při návrhu překladače. Pravidelné schůze nám umožnily získat přehled o tom, jak se projekt vyvíjí a na základě toho přerozdělit práci či posunout naše interní termíny.

Práci na jednotlivých částech jsme si rozdělili již na začátku, postupem času bylo však čím dál víc potřeba na jednotlivých modulech spolupracovat a diskutovat o způsobech jejich vzájemného propojení.

Při vývoji jsme významně používali nástroj Git a platformu GitHub. Hlavní komunikační platformou byl Discord, kde probíhaly vyjímečně probíhaly schůzky, pokud se nám nepodařilo sejít se osobně.

Rozdělení práce:

- Filip Jenis (xjenisf00)
  - vedení týmu, organizace, generování cílového kódu, code review
- Hugo Bohácsek (xbohach00)
  - scanner, syntaktická analýza, sémantická analýza
- Josef Ambruz (xambruj00)
  - LL(1) gramatika a tabulka, konečný automat scanneru, testování, dokumentace
- Štefan Dubnička (xdubnis00)
  - syntaktická analýza výrazů, precedenční tabulka, sémantická analýza

Návrh překladače:

Překladač je založený na syntaxí řízeném překladu, provádí jednoprůchodovou analýzu kódu.

Lexikální analýza:

Lexikální analyzátor je implementován podle deterministického konečného automatu, jehož diagram je k dispozici v příloze.

Hlavní funkcí lexikálního analyzátoru je funkce *`get_next_token`*, která je volaná Syntaktickým analyzátorem. Její návratovou hodnotou je struktura `Token`. Ta obsahuje všechny informace potřebné pro vykonávání syntaktické analýzy: její typ, hodnota, přetypovaná hodnota (v případě float/ int). Zároveň uchovává informace, které potřebuje syntaktická analýza, a to konkrétně:

jestli je nullable, slice, její datový typ, a pro precedenční analýzu, jestli se jedná o speciální hodnotu epsilon (ε)

Analyzátor načítá bílé znaky, dokud nenarazí na lexém, který se snaží zpracovat, respektive ověřit, zda se může podle definice jazyka IFJ24 ve zdrojovém kódě vůbec vyskytovat. V případě, že narazí na sekvenci znaků, která není povolená, končí chybou 1 – lexikální chyba. Před tím se však postupně snaží zjistit, zda se nejedná o klíčové slovo, řetězec znaků, číslo, komentář, či identifikátor.

Tabulka klíčových slov je implementována jako jednorozměrné pole *`sc_keywords`* a je pro přehlednost odlišená od tabulky identifikátorů vestavěných funkcí, která je uložená v poli *`sc_builtin`*.

Pro přehlednost využívá funkce *`get_next_token`* další pomocné funkce, konkrétně:

- *`is_keyword`* - ověřuje, zda je lexém klíčové slovo
- *`is_builtin`* - ověřuje, zda je lexém vestavěnou funkcí jazyka IFJ24
- *`create_token`* - pomocná funkce pro vytváření struktury `Token`
- *`resize_buffer`* - mění velikost paměťového bloku, pokud je to potřeba (příliš dlouhý lexém)
- *`string_duplicate`* - imituje funkci *`strdup`*, která není součástí ISO C

Parser:

Parser je jádro celého překladače. Pro jeho implementaci jsme využili metody rekurzivního sestupu založeného na LL(1) gramatice. Výrazy jsou vyhodnocovány metodou precedenční syntaktické analýzy. Hlavní funkce parseru je *`parser_parse`*, která je efektivně vstupním bodem parseru.

Parser volá scanner, který předává tokeny, včetně jejich typu, na základě kterých se rozhoduje, kterým pravidlem se řídit. Pokud parser detekuje výraz, načte všechny tokeny výrazu na stack precedenční syntaktické analýzy a předá režii. V případě chyby vypíše vhodné chybové hlášení.

Precedenční syntaktická analýza:

Pro implementaci precedenční syntaktické analýzy využíváme vesměs standardních postupů, zmíněných na přednáškách. Precedenční syntaktická analýza je implementována pomocí precedenční tabulky a redukční gramatiky. Postupně načítá tokeny ze stacku popisujícího výraz a na základě tabulky rozhoduje o dalším kroku. Pokud dojde k úspěšné redukci výrazu, je režie předána zpátky syntaktické analýze shora dolů.

Sémantická analýza

Sémantická analýza probíhá zároveň se syntaktickou. Na začátku dojde k inicializaci a naplnění tabulky symbolů pro vestavěné a uživatelem definované funkce. Pro každou funkci se vytvoří seznam parametrů a jejich typů. Vrámci volání funkce probíhá kontrola počtu parametrů a jejich typová kompatibilita. Zároveň se také vytvářejí tabulky symbolů pro kontext každé funkce, které obsahují informace o identifikátorech a jejich datových typech. Při průchodu přes výrazy se kontroluje typová kompatibilita a výsledek se ukládá do tabulky symbolů.

Generátor cílového kódu:

Generátor cílového kódu generuje mezikód IFJcode24 na standardní výstup po dokončení veškerých analýz. Jednotlivé části kódu se generují už vrámci analýz do dočasné paměti překladače

Generování kódu je implementované v souboru *`generator.c`*, s rozhraním v *`generator.h`*. Obsahuje funkce generování jednotlivých částí programu, funkci pro inicializaci generování kódu, funkci pro uvolnění prostředků alokovaných pro generátor a funkci, která vygenerovaný kód přesune z dočasné paměti na standardní výstup. Funkce na generování jednotlivých částí programu jsou volané z příslušných pravidel syntaktické analýzy.

Na začátku generování se alokuje dočasná paměť v podobě dynamického řetězce (implementovaného v *`dynamic_string.c`* a *`dynamic_string.h`*). Do ní se následně přidává vygenerovaný mezikód. Na úvod se vygeneruje hlavička mezikódu, globální proměnné, skok do hlavní funkce těla programu a vygenerují se i vestavěné funkce napsané v IFJcode24.

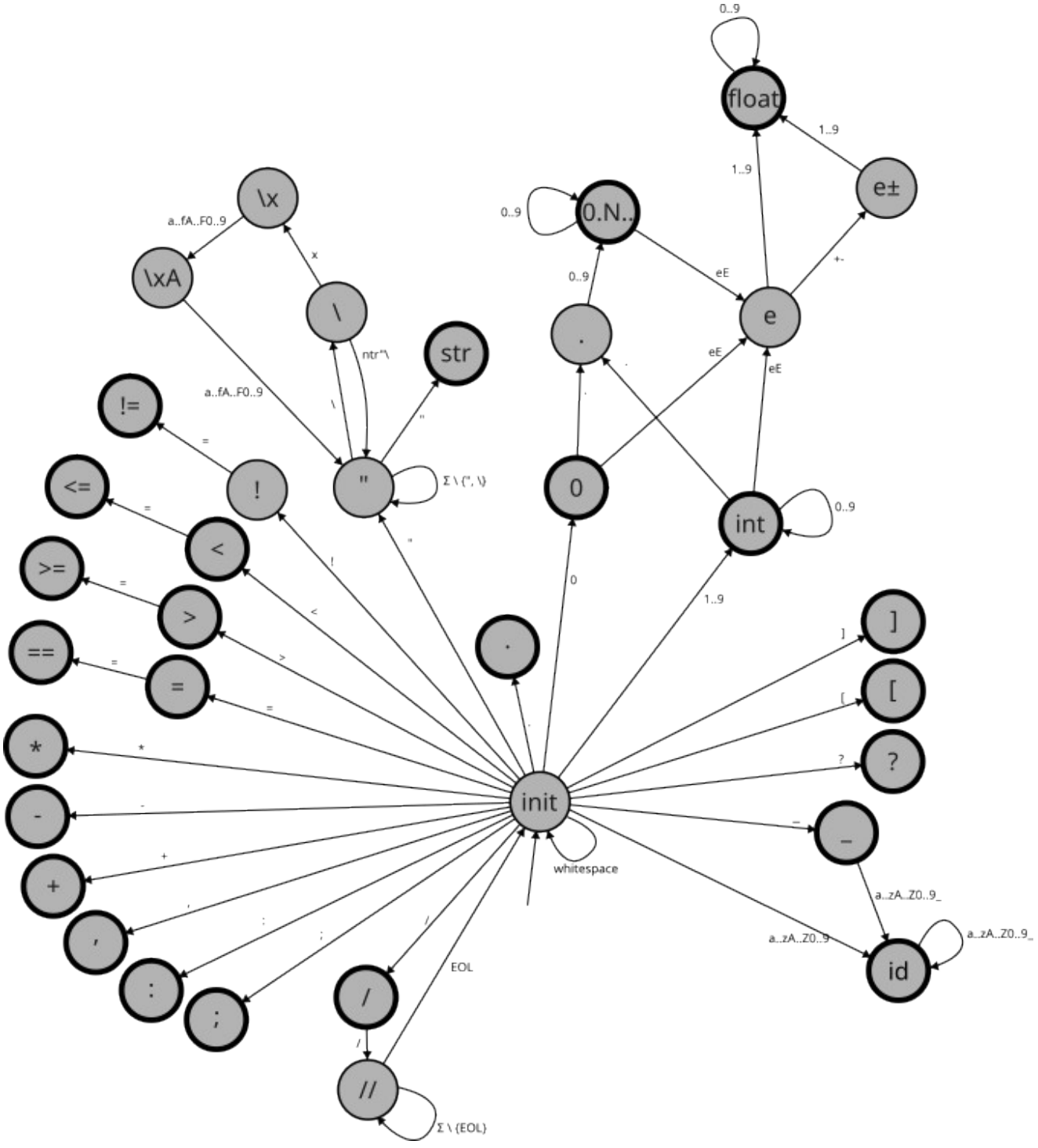
Každá funkce začíná návěštím *`$identifikátor`* a má vlastní lokální rámec. Parametry funkce se předávají pomocí dočasného rámce s pořadovým číslem daného argumentu. Při návratu z funkce se výsledek zpracovaného výrazu ukládá do speciální proměnné a vykoná se skok na konec funkce.

Všechny výrazy jsou ukládané a vykonávané na datovém zásobníku v pořadí, v jakém je zpracovává precedenční analýza výrazů. Výsledek výrazů se ukládá do globální proměnné.

Pro příkazy typu *`if`* a *`while`* se generují speciální návěští tvaru *`$identifikátor_funkce$if/while$pořadí_ve_funkci`* (následně s koncovkami *`$end`*, *`$else`*, …). Tyto návěští se dočasně ukládají do seznamu (implementovanému v *`label_list.c`* a *`label_list.h`*), kde se vždy kontroluje vrchol, aby bylo dodržené správné pořadí generovaných návěští. Po vygenerování konce příkazu typu *`if/while`* se jeho návěští odstraní ze seznamu.

Přílohy:

Konečný automat pro Scanner:



LL-gramatika:

1. ⟨program⟩ → const id = @ import ("ifj24.zig") ; ⟨function_def⟩
2. ⟨function_def⟩ → ε
3. ⟨function_def⟩ → pub fn id (⟨params⟩) ⟨ret_type⟩ { ⟨statement⟩ } ⟨function_def⟩
4. ⟨params⟩ → ε
5. ⟨params⟩ → id : ⟨type⟩ ⟨params_n⟩
6. ⟨params_n⟩ → ε
7. ⟨params_n⟩ → , id : ⟨type⟩ ⟨params_n⟩
8. ⟨ret_type⟩ → ε
9. ⟨ret_type⟩ → void
10. ⟨ret_type⟩ → ⟨type⟩
11. ⟨type⟩ → ⟨nullable⟩ ⟨slice⟩ ⟨type_val⟩
12. ⟨type_val⟩ → i32
13. ⟨type_val⟩ → f64
14. ⟨type_val⟩ → u8
15. ⟨nullable⟩ → ε
16. ⟨nullable⟩ → ?
17. ⟨slice⟩ → ε
18. ⟨slice⟩ → [ ]
19. ⟨statement⟩ → ε
20. ⟨statement⟩ → ⟨var_def_kw⟩ id ⟨colon⟩ = expression ; ⟨statement⟩
21. ⟨statement⟩ → expression ; ⟨statement⟩
22. ⟨statement⟩ → return ⟨return_val⟩ ; ⟨statement⟩
23. ⟨statement⟩ → if (expression) ⟨id_no_null⟩ { ⟨statement⟩ } ⟨else⟩ ⟨statement⟩
24. ⟨statement⟩ → while (expression) ⟨id_no_null⟩ { ⟨statement⟩ } ⟨statement⟩
25. ⟨colon⟩ → ε
26. ⟨colon⟩ → : ⟨type⟩
27. ⟨id_no_null⟩ → ε
28. ⟨id_no_null⟩ → | id |
29. ⟨var_def_kw⟩ → const
30. ⟨var_def_kw⟩ → var
31. ⟨return_val⟩ → ε
32. ⟨return_val⟩ → expression
33. ⟨else⟩ → ε
34. ⟨else⟩ → else { ⟨statement⟩ }

LL-tabulka:

	id	=	;	pub	fn	(	)	{	}	:	,	void	i32	f64	u8	?	[	]	|	return	if	while	const	var	expression	else	$
⟨program⟩																							1				
⟨function_def⟩				3																							2
⟨params⟩	5						4																				
⟨params_n⟩							6				7																
⟨ret_type⟩								8				9	10	10	10	10	10										
⟨type⟩													11	11	11	11	11										
⟨type_val⟩													12	13	14												
⟨nullable⟩													15	15	15	16	15										
⟨slice⟩													17	17	17		18										
⟨statement⟩									19											22	23	24	20	20	21		
⟨colon⟩		25								26																	
⟨id_no_null⟩								27											28								
⟨var_def_kw⟩																							29	30			
⟨return_val⟩			31																						32		
⟨else⟩																				33	33	33	33	33	33	34	

Precedenční gramatika:

1. E → id
2. E → E + E
3. E → 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)

Precedenční tabulka:

	+	-	*	/	(	)	id	$	ε	==	!=	<	>	<=
+	>	>	<	<	<	>	<	>	<	>	>	>	>	>
-	>	>	<	<	<	>	<	>	<	>	>	>	>	>
*****	>	>	>	>	<	>	<	>	<	>	>	>	>	>
/	>	>	>	>	<	>	<	>	<	>	>	>	>	>
(	<	<	<	<	<	>	<	=	<	<	<	<	<	<
)	>	>	>	>	=	>	=	>	>	<	<	<	<	<
id	>	>	>	>	=	>	=	>	=	>	>	>	>	>
$	<	<	<	<	<	=	<	=	=	>	>	>	>	>
ε	<	<	<	<	=	>	=	>	=	<	<	<	<	<
==	<	<	<	<	<	>	<	>	>	=	=	=	=	=
!=	<	<	<	<	<	>	<	>	>	=	=	=	=	=
<	<	<	<	<	<	>	<	>	>	=	=	=	=	=
>	<	<	<	<	<	>	<	>	>	=	=	=	=	=
<=	<	<	<	<	<	>	<	>	>	=	=	=	=	=
>=	<	<	<	<	<	>	<	>	>	=	=	=	=	=