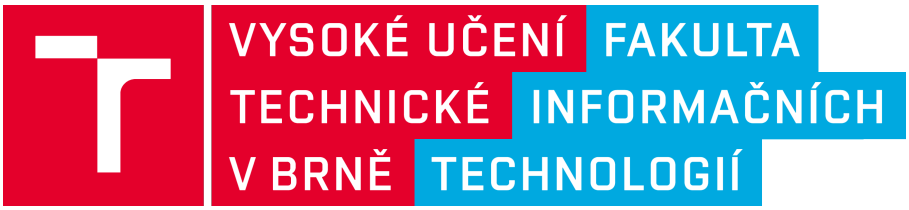


Implementace překladače imperativního jazyka IFJ23

Tým xjakub41, Varianta: TRP-izp

Milan Jakubec (xjakub41) 25%
Jiří Tesař (xtesar44) 25%
Norman Babiak (xbabia01) 25%
Jakub Ráček (xracek12) 25%

Brno, 6. prosince 2023

Obsah

1 Rozdělení práce

Milan Jakubec

- Návrh gramatiky
- Implementace rekurzivní syntaktické analýzy
- Implementace generátoru (obecně)
- Dokumentace

Jiří Tesař

- Lexikální analýza
- Dokumentace

Norman Babiak

- Implementace analýzy výrazů
- Implementace generátoru (obecně, výrazy)
- Implementace zásobníku
- Dokumentace

Jakub Ráček

- Implementace sémantické analýzy
- Implementace tabulky symbolů
- Implementace zásobníku
- Dokumentace

2 Implementace

Jednotlivé moduly překladače jsou rozděleny do samostatných souborů vyjímaje sémantické analýzy rekurzivního syntaktického parseru, která se nachází společně se samotným parserem v jednom souboru. Deklarace struktur a funkcí (výjimaje pomocných funkcí) se nachází ve stejnojmenných hlavičkových souborech, v nichž jsou uvedeny i Doxygen komentáře popisující jejich funkci.

2.1 Lexikální analýza

Lexikální analyzátor překladače je implementován za pomoci konečného automatu (FSM). Tento automat lze ve zdrojových souborech nalézt pod názvem **scanner.c**, jeho diagram se pak nachází dále v této dokumentaci. Vstup se tokenizuje do datové struktury *token_t*, tato struktura a její další pomocné struktury se nacházejí v souboru **token.c**. Každý token v sobě má svůj typ (typy tokenů jsou definovány pomocí enumu token_type), atributy a boolean eol, který pomáhá zjisit, zda se token nachází na novém řádku, či nikoliv.

Pro samotné načítání tokenů je klíčová funkce *get_next_token()*. Implementována je i možnost práce s dynamickými stringy, k nalezení v souboru **str.c**. Pokud jsou tedy součástí vstupního zdrojového kódu řetězcové hodnoty jako například řetězcový literál nebo identifikátor, jsou funkce ze str.c použity k akumulaci jednotlivých znaků do kompletního řetězce, který je pak přiřazen jako atribut příslušného tokenu. V případě výskytu chyby se vypíše chybová hláška a provádění se ukončí s odpovídající návratovou hodnotou. K tomu se využívá externí modul **error.c**

2.2 Syntaktická analýza

Syntaktická analýza parseru je implementována na základě doporučení dle rekurzivního sestupu za použití poměrně rozsáhlé LL gramatiky. Implementaci této gramatiky resp. jejích pravidel lze nalézt v souboru **parser.c**. Samotná LL gramatika včetně LL tabulky je k dispozici v pozdějších částech tohoto dokumentu. Jelikož je parser středobodem celého překladu, je to právě on, kdo využívá lexikální analyzátor a inicializuje v sobě veškeré potřebné struktury, včetně instance generátoru, který během překladu rovnou generuje výsledný mezikód. Jednotlivá pravidla parseru jsou implementována jako funkce, která má jako parametr pokaždé instanci parseru, a potom dle

potřeby různě pomocné struktury (pro parametry, argumenty funkce). Samotný parser potom pomocí přepínačů kontroluje svůj stav, jako například zda se nachází ve funkci, v cyklu, a podobně. Tokeny parser konzumuje pomocí vlastní funkce *load_token()*, která využívá rozhraní lexikálního analyzátoru. V každé funkci se také po konzumaci tokenu kontroluje, zda byl na vstupu přijat syntkaticky správný a očekávaný token, pakliže nebyl, vypíše se syntaktická chyba a program se okamžitě ukončí (viz. sekce "Lexikální analýza").

2.3 Sémantická analýza

Sémantická analýza je prováděna staticky, využívá přitom toho, že jazyk IFJ23 je silně staticky typovaný. Logika analýzy je tedy úzce spojena s analýzou syntaktickou a generátorem. Obzvláště kritický je zde zásobník, kterým jsme se rozhodli řešit rozsahy platnosti. Zásobník obsahuje tabulky symbolů s definicemi proměnných a funkcí s tím, že na vrcholu je vždy aktuální rozsah platnosti, a na dně je globální rozsah platnosti. V analýze se sleduje i současný stav parseru, reprezentovaný několika flagy. Jedná se převážně o typové a kontextové kontroly. Typové kontroly jsou přítomné převážně při přiřazení do proměnných, volání funkcí, a zpracování výrazů. Pro uchování a kontrolu parametrů funkce mimo jiné slouží struktura dynamického pole. Kontroly kontextové zamezují práci s nedefinovanými či neinicializovanými proměnnými a redefinicemi.

2.4 Analýza výrazů

Analýza výrazů probíhá v souboru **expression.c**, který využívá strukturu syntaktické analýzy a precedenční tabulku. Na začátku je inicializován zásobník a struktura **analysis_t** pro pozdější sémantickou analýzu výrazů. Na zásobník je uložen počáteční znak "$", poté je načten token z vrcholu zásobniku a následujíci token ze vstupu pomocí parserové funkce **load_token()**. Načtené tokeny potom vyhodnotí pomocná funkce **precedence()**. Funkce vrátí výsledek volající funkci, **prec_analysis()**, která se rozhodne, jestli načtený token ze vstupu uloží na vrchol zásobníku, nebo výraz v zásobníku zredukuje na jediný neterminál. Redukce jednotlivých terminálů se vyhodnotí po sémantické analýze podle gramatiky pro výrazy (viz. sekce 7.1). Funkce **reduce** rovnou generuje výsledný mezikód. Pro ukončení analýzy výrazů slouží funkce **handle_upcoming()**, která rozhodne, zda je další token součástí výrazu, nebo už má předat kontrolu zpátky řídící struktuře s příslušným datovým typem výsledku výrazu, s kterým pak řídící struktura dále pracuje.

2.5 Generování kódu

Generování kódu je řešeno v souborech **gen.c** a **buildin.c**, kde v prvním souboru je k nalezení základní struktura generování kódu, tzn. generování deklarací proměnných, generování smyček, podmíněných větví aj., a v druhém souboru se nachází definice vestavěných funkcí. Veškeré parametry funkcí a vlastně i vše ostatní se předává přes zásobník.

3 Datové struktury

3.1 Tabulka symbolů

Pro implementaci tabulky symbolů jsme zvolili variantu TRP-izp. Parser pracuje se zásobníkem tabulek symbolů, kde každá tabulka reprezentuje jiný kontext. Tabulka je postavena na znalostech z předmětu IJC a byla přizpůsobena variantě s otevřenou adresací (implicitního zřetězení položek) na základě prezentací předmětu IAL. V základu se jedná o dynamické kruhové pole, kde každá položka reprezentuje funkci anebo proměnnou existující v aktuálním kontextu. Pro hashovací funkci jsme zvolili řešení ze zadání letošního IJC projektu a dle publikovaného článku[1] jsme využili variantu hashovací funkce "sdbm".

3.2 Zásobník

Soubor **stack.c** obsahuje implementaci zásobníku, který zahrnuje základní funkce, jako jsou **stack_pop()**, **stack_init()**, **stack_push()**, a **stack_free()**. Dále obsahuje další pomocné funkce, které jsou využívány při analýze výrazů a tabulky symbolů.

3.2.1 Zásobník pro analýzu výrazů

Pro analýzu výrazů jsou implementovány funkce jako **stack_push_token()**, která uloží na vrchol zásobníku přicházející token v analýze výrazů. Kromě znaku se tam také uloží jeho datový typ pro sémantickou analýzu. Další pomocnou funkcí je **stack_push_after()**, která vloží token po vrchní terminál zásobníku. Tahle funkce slouží pro vložení tokenu určující konec pro redukci výrazů. Potom **stack_top_token()**, která vrací znak na vrcholu zásobníku, **stack_top_terminal()**, vracející terminál na vrcholu zásobníku. Pomocná funkce **stack_count_after()** slouží pro spočítání znaků nad znakem určující konec pro redukci, a pro vložení těchto znaků do struktury **analysis_t**, pro

sémantickou analýzu. Pro testovací účely je taktéž implementovaná funkce **print_stack**.

3.2.2 Zásobník pro tabulku symbolů

Pro sémantickou analýzu je implementováno několik pomocných funkcí pro práci se zásobníkem tabulek symbolů. Kromě funkcí jako je *stack_push_table()* nebo *stack_pop_table()* existují také funkce pro orientaci v kontextech, *stack_top_table()* reprezentující aktuální kontext a *stack_bottom_table()* reprezentující globální kontext. Pro prohledání stacku a nalezení proměnných a funkcí slouží *stack_lookup_var()* a *stack_lookup_func()*. Mimo jiné existuje i funkce *stack_lookup_var_in_global()*, která pomáhá generátoru rozhodnout jaký rámec má použít.

3.3 Dynamické pole

V naší implementaci máme vlastní strukturu jménem *vector_t* pro práci s dynamickým polem. Vector je datová struktura dynamického pole, která v sobě uchovává parametry funkce. Každý zápis funkce v tabulce symbolů má vlastní vector, který je mu přidělen při prvním zavolání nebo definici funkce a slouží pro sémantickou kontrolu parametrů v následujících voláních nebo definici.

4 Závěr

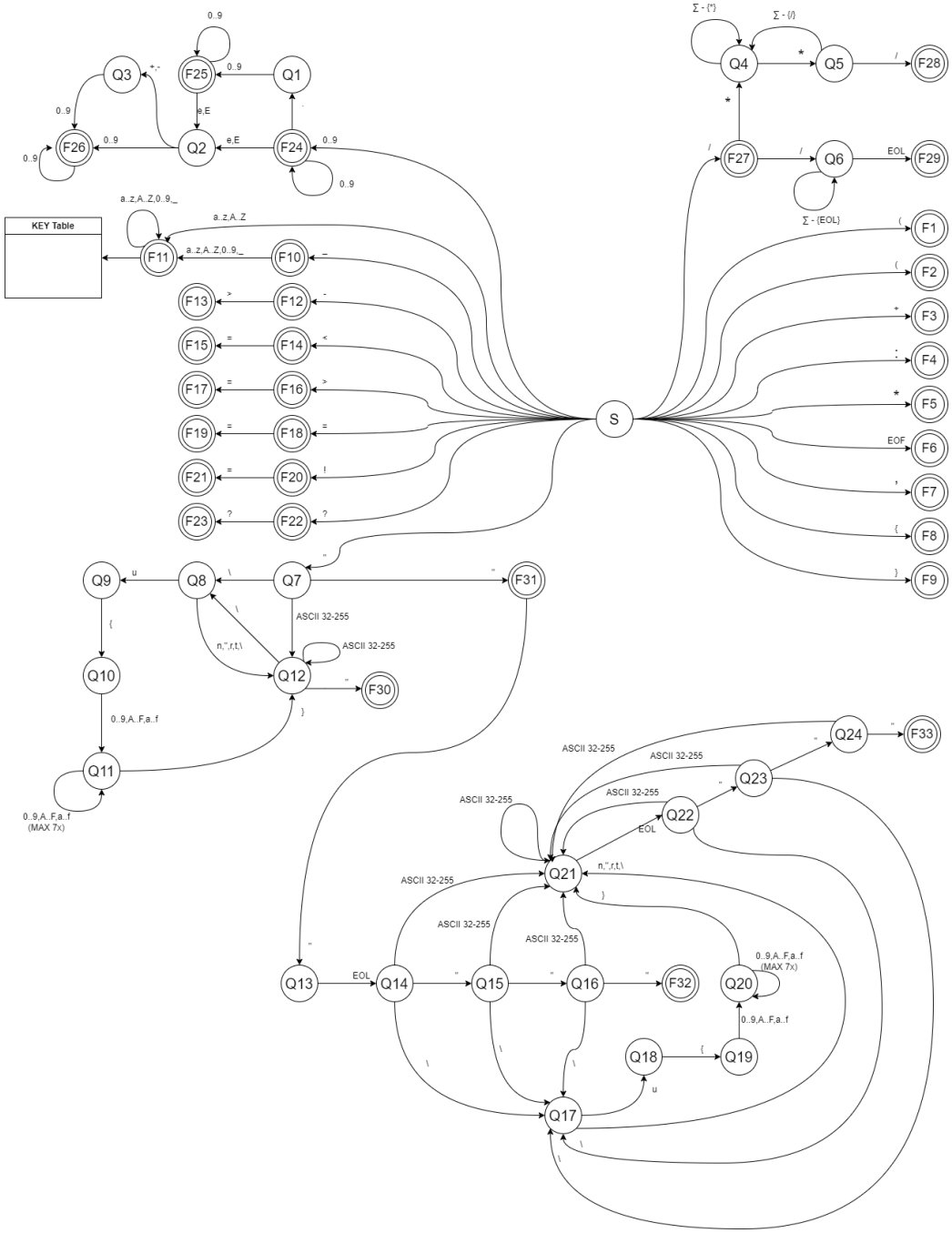
Práci na projektu jsme si navzdory jeho složitosti užili. Myslíme si, že každý z nás se naučil spoustu nových a zajímavých věcí nehledě na zvýšení schopnosti programovat a řešit problémy, přičemž pro každého člena týmu to samozřejmě platí převážně pro modul, který daný člen implementoval. Ale i tyto moduly spolu musely nějak fungovat, pracovat a komunikovat, a domníváme se, že to se nám povedlo zrealizovat dobře. Při řešení projektu u nás vyvstávaly během vývoje různé problémy, které jsme společnými silami byli schopni vyřešit. Velkou zásluhu na tom mají automatizované testy poskytnuté našimi skvělými kolegy z ročníku a samotný pan doktor Křivka, který při tvorbě překladače byl všem velmi nápomocný. Ironicky jsme projekt do stavu, jaký jsme si představovali, dostali až den po druhém testovacím odevzdání, kdy se nám naráz povedlo opravit sémantickou analýzu, naplno zprovoznit základní generování a vestavěné funkce. Závěrem bychom dodali, že velkou výzvou pro náš tým byl souboj s časem, pro každého z nás z ruzných důvodů, ale nakonec jsme se s tím, dle našeho názoru, takříjakíc poprali statečně.

Reference

[1] YIGIT, Ozan - Hash Functions: http://www.cse.yorku.ca/~oz/hash.html.

5 FSM pro lexikální analýzu

5.1 Diagram



6 LL

6.1 LL gramatika

1. <program> -> <statement> <program>
2. <program> -> <function_definition> <program>
3. <program> -> ϵ

4. <statement_list> -> <statement> <statement_list>
5. <statement_list> -> ϵ
6. <statement> -> <variable_definition_let>
7. <statement> -> <variable_definition_var>
8. <statement> -> <assignment>
9. <statement> -> <conditional_statement>
10. <statement> -> <loop>
11. <statement> -> <function_call>
12. <statement> -><return_statement>

13. <function_definition> -> func <identifier> (<parameter_list>) <function_return_type_and_body>
14. <function_return_type_and_body> -> { <void_function_body> }
15. <function_return_type_and_body> -> "->"<type> { <nonvoid_function_body> }
16. <nonvoid_function_body> -> <statement_list>
17. <void_function_body> -> <statement_list>
18. <void_function_body> -> ϵ
19. <parameter_list> -> <parameter> <more_parameters>
20. <parameter_list> -> ϵ

21. <more_parameters> -> , <parameter> <more_parameters>
22. <more_parameters> -> ϵ
23. <parameter> -> <no_name_parameter>
24. <parameter> -> <identifier_parameter>
25. <no_name_parameter> -> _ <identifier> : <type>
26. <identifier_parameter> -> <identifier> <rest_of_identifier_parameter>
27. <rest_of_identifier_parameter> -> _ : <type>
28. <rest_of_identifier_parameter> -> <identifier> : <type>

29. <type> -> Double | Int | String | Double`?` | Int`?` | String`?`
30. <variable_definition_let> -> `let` <identifier> <definition_types>
31. <variable_definition_var> -> `var` <identifier> <definition_types>
32. <definition_types> -> <type_def>
33. <definition_types> -> <initialization>
34. <type_def> -> : <type> <type_def_follow>
35. <type_def_follow> -> <initialization>
36. <type_def_follow> -> ϵ
37. <initialization> -> = <init_type>
38. <init_type> -> <function_call>
39. <init_type> -> <expression>

40. <assignment> -> <identifier> = <assignment_type>
41. <assignment_type> -> <function_call>
42. <assignment_type> -> <expression>

43. <conditional_statement> -> if <if_statement>

44. <if_statement> -> <classical_statement>
45. <if_statement> -> <variable_statement>
46. <classical_statement> -> <expression> { <statement_list> } else { <statement_list> }
47. <variable_statement> -> let <identifier> { <statement_list> } else { <statement_list> }
48. <loop> -> while <expression> { <statement_list> }
49. <function_call> -> <identifier> (<arguments>)
50. <arguments> -> <argument> <more_arguments>
51. <arguments> -> ϵ
52. <argument> -> <arg_name> <arg_value>
53. <arg_name> -> <identifier> :
54. <arg_name> -> ϵ
55. <arg_value> -> <identifier> | <int> | <double> | <string> | <nil>
56. <more_arguments> -> , <argument> <more_arguments>
57. <more_arguments> -> ϵ
58. <return_statement> -> return <returned_expression>
59. <returned_expression> -> <expression >
60. <returned_expression> -> ϵ

6.2 LL tabulka

	func	if	else	var	let	return	Int	Double	String	(	)	{	}	:	while	->	<VarID>	<FunID>	=	<Undec>	<Expr>	,	<Literal>
<program>	2	1		1	1										1		1	1					
<statement_list>		4		4	4								5		4		4	4					
<statement>		9		7	6	12									10		8	11					
<function_definition>	13																						
<function_return_type_and_body>												14				15							
<nonvoid function_body>		16		16	16	16									16		16	16					
<void function_body>		17		17	17	17							18		17		17	17					
<parameter_list>											20						19			19			
<more_parameters>											22											21	
<parameter>																	24			23			
<no_name_parameter>																				25			
<identifier_parameter>																	26						
<rest_of_identifier_parameter>																	28			27			
<type>							29	29	29														
<variable_definition_let>					30																		
<variable_definition_var>				31																			
<definition_types>														32					33				
<type_def>														34									
<type_def follow>																			35				
<initialization>																			37				
<init_type>																		38			39		
<assignment>																	40						
<assignment_type>																		41			42		
<conditional_statement>		43																					
<if_statement>					45																44		
<classical_statement>																					46		
<variable_statement>					47																		
<loop>															48								
<function_call>																		49					
<arguments>											51						50						50
<argument>																	52						
<arg_name>																	53						54
<arg_value>																							55
<more_arguments>											57											56	
<return_statement>					58																		
<returned_expression>																					59		

7 Precedenční analýza

7.1 Gramatika pro výrazy

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

7.2 Precedenční tabulka

	+	-	*	/	==	!=	>=	<=	<	>	??	!	(	)	ID	INT	FLO	STR	NIL	$
+	>	>	<	<	>	>	>	>	>	>	>	<	>	>	<	<	<	<	>	>
-	>	>	<	<	>	>	>	>	>	>	>	<	>	>	<	<	<	<	>	>
*	>	>	>	>	>	>	>	>	>	>	>	<	>	>	<	<	<	<	>	>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	>	<	<	<	<	>	>
==	>	>	<	<	X	X	X	X	X	X	<	<	>	>	<	<	<	<	>	>
!=	>	>	<	<	X	X	X	X	X	X	<	<	>	>	<	<	<	<	>	>
>=	>	>	<	<	X	X	X	X	X	X	<	<	>	>	<	<	<	<	>	>
<=	>	>	<	<	X	X	X	X	X	X	<	<	>	>	<	<	<	<	>	>
<	>	>	<	<	X	X	X	X	X	X	<	<	>	>	<	<	<	<	>	>
>	>	>	<	<	X	X	X	X	X	X	<	<	>	>	<	<	<	<	>	>
??	>	>	<	<	>	>	>	>	>	>	<	<	>	>	<	<	<	<	>	>
!	<	<	<	<	X	>	<	<	<	<	<	<	>	=	<	<	<	<	X	X
(	<	<	<	<	<	<	<	<	<	<	<	<	<	X	<	<	<	<	X	X
)	>	>	>	>	>	>	>	>	>	>	>	X	X	>	X	X	X	X	X	>
ID	>	>	>	>	>	>	>	>	>	>	>	X	<	>	X	X	X	X	X	>
INT	>	>	>	>	>	>	>	>	>	>	>	X	<	>	X	X	X	X	X	>
FLO	>	>	>	>	>	>	>	>	>	>	>	X	<	>	X	X	X	X	X	>
STR	>	>	>	>	>	>	>	>	>	>	>	X	<	>	X	X	X	X	X	>
NIL	>	>	>	>	>	>	>	>	>	>	>	X	<	>	X	X	X	X	X	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	X	<	<	<	<	X	X

> Redukce, < Uložení na zásobník