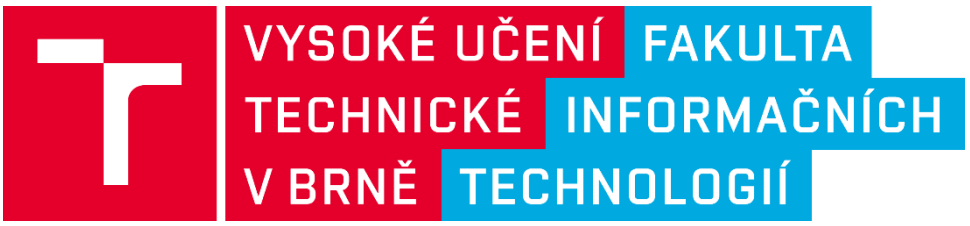


Dokumentace

Implementace překladače IFJ23

Tým xzaple40, varianta BVS

Václav Zapletal (xzaple40)	25%
Denys Dumych (xdumyc00)	25%
Artem Vereninov (xveren00)	25%
Oleg Andriichuk (xandri07)	25%

Úvod

Cílem tohoto projektu bylo vytvořit překladač pro podmnožinu programovacího jazyka Swift, nazvanou IFJ23. Vstupní program nám byl poskytnut prostřednictvím standardního vstupu v terminálu. Naším úkolem bylo generovat mezikód a v případě chyby souboru vypsat odpovídající chybový kód na standardní chybový výstup.

Lexikální analýza

Základem celého překladače je právě lexikální analýza, jeho tvorba byla založena na předem vytvořeném deterministickém konečném automatu.

Hlavní funkcí je **`getToken`**, která je volána ze syntaktického analyzátoru a zapisuje nový token, obsahující veškeré potřebné informace, jako je typ, jméno, atd., na adresu, kterou předáváme jako parametr. Lexikální analyzátor začíná načítat jednotlivé znaky ze vstupu, dokud nevytvoří lexém nebo znak odpovídající jazyku IFJ23. Lexikální analyzátor funguje pomocí `switch`, který zpracovává různé znaky a rozhoduje, zda je token již připraven nebo potřebuje další úpravy. **`getToken()`** využívá pomocné funkce, jako je **`checkKeyword`** pro zjištění lexému.

Tímto způsobem jsme vytvořili základní strukturu lexikálního analyzátoru pro jazyk IFJ23.

Syntaktická analýza

Napsáno pomocí rekurzivního sestupu s využitím LL gramatiky. Získáváme nový token pomocí funkce **`newToken()`** a využíváme pomocnou funkci **`type_check()`** pro jednodušší kontrolu syntaxe. Pro ukládání dat jsme využili následující globální proměnné:

- **`Stack main_stack`**: Definice zásobníku stromu pro definování rozsahu.
- **`Token token:`** Token, který upravujeme při použití funkce **`getToken`**.
- **`StringValue current_function:`** Název funkce, ve které se momentálně nacházíme.
- **`bool inside_ifwhile`**: Boolean hodnota pro zjištění, zda se nacházíme uvnitř cyklu.

- **`bool inside_function`**: Boolean hodnota pro zjištění, zda se nacházíme uvnitř funkce.

Hlavní funkcí je **`parse`**, která iniciovala syntaktickou analýzu. Zavola různé funkce pro počáteční inicializaci globálních proměnných, zaznamená v globálním rozsahu vnoření funkcí a deklarace funkcí. Zároveň zapsala všechny tokeny do pole, které bude využito v další analýze. Následně spusti analýzu voláním funkce **`program`**, která ověřuje, zda napsaný kód odpovídá LL gramatice.

Sémantická analýza

Pro provádění sémantické analýzy využíváme strukturu zásobníku, která obsahuje odkazy na binární vyhledávací stromy. V těchto stromech uchováváme uzly s informacemi relevantními pro sémantickou analýzu. Konkrétně jsou v uzlech uloženy odkazy na syny, klíč položky a odkaz na strukturu, která popisuje typ uzlu, jeho datový typ, množství parametrů (pro funkce) a odkaz na spojový seznam, který popisuje jméno parametru a jeho datový typ. Tato informace nám umožňuje provádět ověřování chyb typové kompatibility a deklarací funkcí nebo proměnných.

Používáme zásobníkovou strukturu ke správě rozsahu proměnných. Při vstupu do funkce nebo cyklu přidáme strom na vrchol zásobníku a při opuštění ho smažeme pomoci funkce `Stack_Pop` a `Stack_Push`. Na začátku zásobníku máme strom pro globální proměnné, který přidáváme na začátku funkce parse().

Zpracování pro výrazy

Pro zpracování výrazů používáme symbol Stack_exp. Je implementován ve stejném souboru expression.c a expression.h. Ukazatele na další tokeny získáváme pomocí funkce exp_token(). Pomocí funkcí implementovaných v souborech expressions.c a expressions.h sbíráme přijaté tokeny do zásobníku a redukujeme výraz podle tabulky precedencí, která určuje, jaký operátor má vyšší prioritu. Při zpracování výrazu kontrolujeme typovou kompatibilitu operandů. Tento algoritmus také kontroluje, zda byla definována proměnná.

Generování kódu

Generování cilového kodu je implementováno v souboru codegen.c. Rozhraní se nachazí v souboru codegen.h. Implementováno je pouze generování deseti vnořených funkci. Funkce gen_ast zpracovává výrazy pomocí abstraktního syntaktického stromu a kontroluje typy operandu. Funkce create_string_4_gen zpracováva vstupní řetězec a formatuje ho do IFJcode23 string. Pro generování funkce používáme funkce generate_func_header, gen_func_return a gen_func_dec. Pro volání funkce využíváme gen_func_call.

Datové struktury

Dynamický řetězec

Navrhli jsme strukturu **StringValue** v souborech **str.c** a **str.h** pro manipulaci s řetězci, zejména v případech, kdy chceme ukládat řetězce s nedefinovanou délkou, například při ukládání tokenů. Tuto strukturu také používáme pro hledání různých identifikátorů v symbolické tabulce. Struktura uchovává odkaz na řetězec, jeho velikost a informaci o tom, kolik paměti bylo vyhrazeno. Implementovali jsme standardní operace pro přidávání na konec řetězce, kopírování řetězce, atd. Na začátku vyhradíme určitý objem paměti a při nedostatku paměti provedeme realokaci.

Binární vyhledávací strom

Navrhli jsme vyvážený binární vyhledávací strom, který slouží jako tabulka symbolů. V uzlech stromu jsou uloženy odkazy na potomky, klíč položky ve formě řetězce a odkaz na strukturu **node_data**. Tato struktura obsahuje informace o typu uzlu, jeho datovém typu, počtu parametrů (pro funkce), indikaci toho, zda může daný identifikátor být nebo vracet NULL, a odkaz na spojový seznam, který popisuje jméno parametru funkce a jeho datový typ. Implementovali jsme rovněž standardní funkce pro manipulaci s tabulkou a vyvážení stromu,

například **`bst_init`**, **`bst_insert`**, **`bst_delete`**, atd. Využíváme tuto strukturu k ukládání identifikátorů a pro rychlý přístup k jejich charakteristikám. Inspirací pro naši implementaci byla studijní opora předmětu IAL

Začátek práce

Projekt jsme začali společným pochopením zadání, následně jsme si rozdělili logické části překladače mezi sebou a studovali jejich funkčnost.

Způsob spolupráce

Pro správu verzí jsme zvolili GitHub kvůli jeho snadné použitelnosti a efektivnímu sdílení kódu. Společná práce probíhala pravidelně na schůzkách, které jsme si rezervovali minimálně jednou týdně. Během těchto setkání jsme prezentovali své poznatky z individuální práce. Pro sdílení materiálů a komunikaci mimo schůzky jsme využívali platformu Discord.

Rozdělení práce

Václav Zapletal – Tvorba LL gramatiky, syntaktická analýza, testování a dokumentace

Denys Dumych – Scanner, Tvorba precedenční tabulky a Precedenční analýza

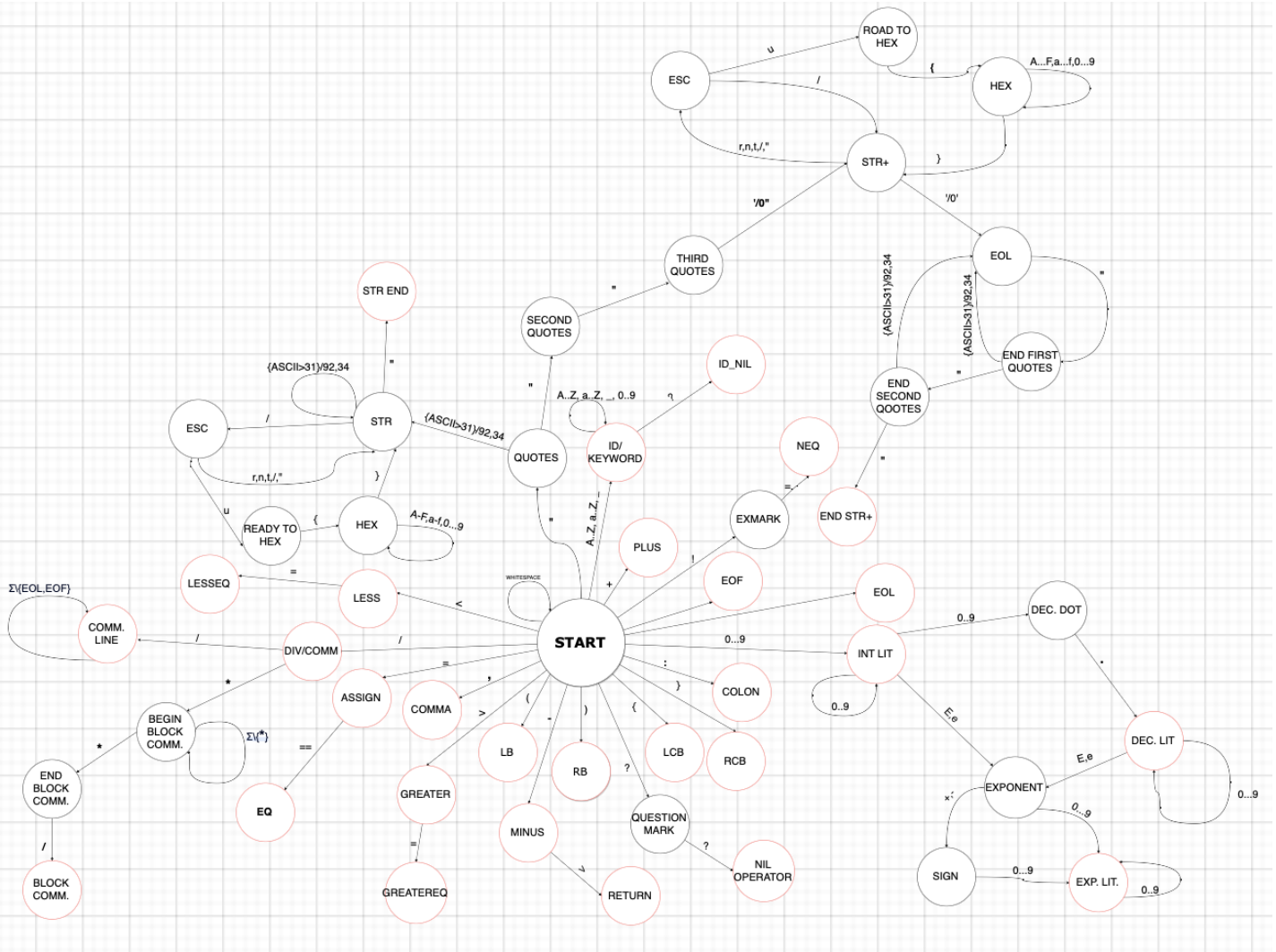
Artem Vereninov - Sémantická analýza, testování, dokumentace

Oleg Andriichuk - Tvorba konečného automatu, tabulka symbolů a generátor kódu

Závěr

Po prvním přečtení zadání jsme cítili určitou zmatenost, ale po několika přednáškách jsme začali stále lépe chápat. Rychle jsme rozdělili práci, a každý člen týmu obdržel téma, kterým se bude zabývat. Během semestru jsme měli mnoho schůzek, abychom mohli řešit všechny

problémy s kompatibilitou modulů, které jsme psali, a spolu porozumět principům fungování překladačů. Během práce jsme narazili na několik úskalí. Prvním z nich bylo zjištění, že budeme potřebovat dva průchody při sémantické analýze z důvodu volání funkce před její deklarací. Druhé bylo během generovani kodu.



	+	*	(	)	i	$	-	/	<	>	<=	>=	==	!=
+	R	S	S	R	S	R	R	S	R	R	R	R	R	R
*	R	R	S	R	S	R	R	R	R	R	R	R	R	R
(	S	S	S	W	S	N	S	S	S	S	S	S	S	S
)	R	R	N	R	N	R	R	R	R	R	R	R	R	R
i	R	R	N	R	N	R	R	R	R	R	R	R	R	R
$	S	S	S	N	S	N	S	S	S	S	S	S	S	S
-	R	S	S	R	S	R	R	S	R	R	R	R	R	R
/	R	R	S	R	S	R	R	R	R	R	R	R	R	R
<	S	S	S	R	S	R	S	S	N	N	N	N	N	N
>	S	S	S	R	S	R	S	S	N	N	N	N	N	N
<=	S	S	S	R	S	R	S	S	N	N	N	N	N	N
>=	S	S	S	R	S	R	S	S	N	N	N	N	N	N
==	S	S	S	R	S	R	S	S	N	N	N	N	N	N
!=	S	S	S	R	S	R	S	S	N	N	N	N	N	N

1. <program> -> <statement> <program>
2. <program> -> EOF
3. <statement> FUNC FUNC_ID (<params>) <is_return> { <statement>} <statement>
4. <statement> -> VAR ID <type_assign> <value_assign> <statement>
5. <statement> -> LET ID <type_assign> <value_assign> <statement>
6. <statement> -> ID = <value_assign_list> <statement>
7. <statement> -> FUNC_ID(<args>) <statement>
8. <statement> -> WHILE <expression> { <statement> } <statement>
9. <statement> -> IF <islet> <expression> { <statement> } else { <statement> } <statement>
10. <statement> -> RETURN <return_types> <statement>
11. <statement> -> eps
12. <params> -> PARAM_NAME ID : <type> <params_n>
13. <params> -> eps
14. <params_n> -> , PARAM_NAME ID : <type> <params_n>
15. <params_n> -> eps
16. <is_return> -> -> <type>
17. <is_return> -> eps
18. <return types> -> ID
19. <return types> -> <expression>
20. <return types> -> FUNC_ID()
21. <return types> -> eps
22. <type> -> INT
23. <type> -> DOUBLE
24. <type> -> STRING
25. <islet> -> IF_LET
26. <islet> -> eps
27. <type_assign> -> : <type>
28. <type_assign> -> eps
29. <value_assign> -> = <value_assign_list>
30. <value_assign> -> eps
31. <assign_value_list> -> ID
32. <assign_value_list> -> <expresion>
33. <assign_value_list> -> FUNC_ID(args)
34. <assign_value_list> -> <value>
35. <args> -> <args_list> <args_n>
36. <args_list> -> <args_name>
37. <args_list> -> <id_or_value>
38. <args_n> -> eps
39. <args_name> -> A_NAME: <id_or_value>
40. <args_name> -> eps
41. <id_or_value> -> ID
42. <id_or_value> -> <value>
43. <args> -> eps
44. <args_n> -> , <args_list> <args_n>
45. <value> -> INT_VALUE
46. <value> -> DOUBLE_VALUE
47. <value> -> STRING_VALUE

	<statement>	EOF	FUNC	VAR	LET	ID	FUNC_ID	WHILE	IF	RETURN	PARAM_NAME	,	- >	<expression>	INT	DOUBLE	STRING	IF_LET	:	=	<value>	<args_list>	<args_name>	<id_or_value>	A_NAME	INT_VALUE	DOUBLE_VALUE	STRING_VALUE	eps
<program>	1	2																											
<statement>			3	4	5	6	7	8	9	10																			11
<params>											12																		13
<params_n>												14																	14
<is_return>						18							16																17
<return_types>							20							19															21
<type>															22	23	24												
<is_let>																		25											26
<type_assign>																			27										28
<value_assign>																				29									30
<assign_value_list>						31	33							32							34								
<args>																						35							43
<args_list>																							36	37					
<args_name>																									39				40
<id_or_value>						41															42								
<args_n>												44																	38
<value>															45	46	47												