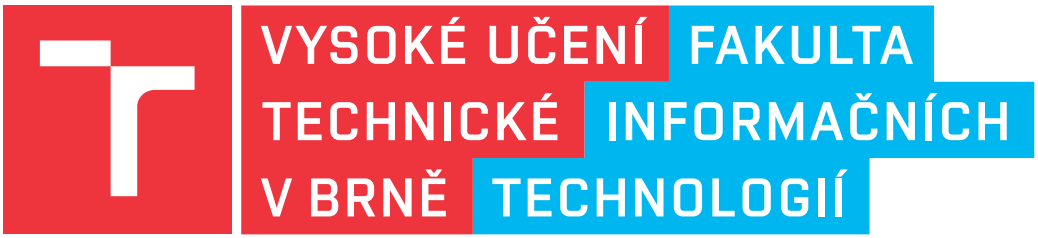


Implementace překladače imperativního jazyka IFJ21

Tým 005, varianta I.

Tadeáš Vintrlík	`xvintr04`	25%
Kryštof Albrecht	`xalbre05`	25%
Josef Škorpík	`xskorp07`	25%
Jakub Kozubek	`xkozub07`	25%

8. prosince 2021

Obsah

1 Spolupráce v týmu

Při práci na projektu jsme využívali metodu párového programování. Díky pravidelným setkáním, osobně i přes Discord, měl každý přehled o dění v projektu a nebylo nutné využívat nástroje, jako je Trello.

Pro verzování zdrojových kódu projektu jsme využívali GitHub.

1.1 Rozdělení práce

1.1.1 Tadeáš Vintrlík

- Tabulka symbolů
- Abstraktní datové typy - seznam, zásobník, AVL stromy
- Testy pro ADT a syntaktický analyzátor
- Generování kódu
- Sémantické kontroly
- Precedenční analyzátor

1.1.2 Kryštof Albrecht

- Syntaktický analyzátor shora-dolů
- Lexikální analyzátor
- LL-gramatika

1.1.3 Josef Škorpík

- Automat pro lexikální analýzu
- Lexikální analyzátor
- Syntaktický analyzátor shora-dolů

1.1.4 Jakub Kozubek

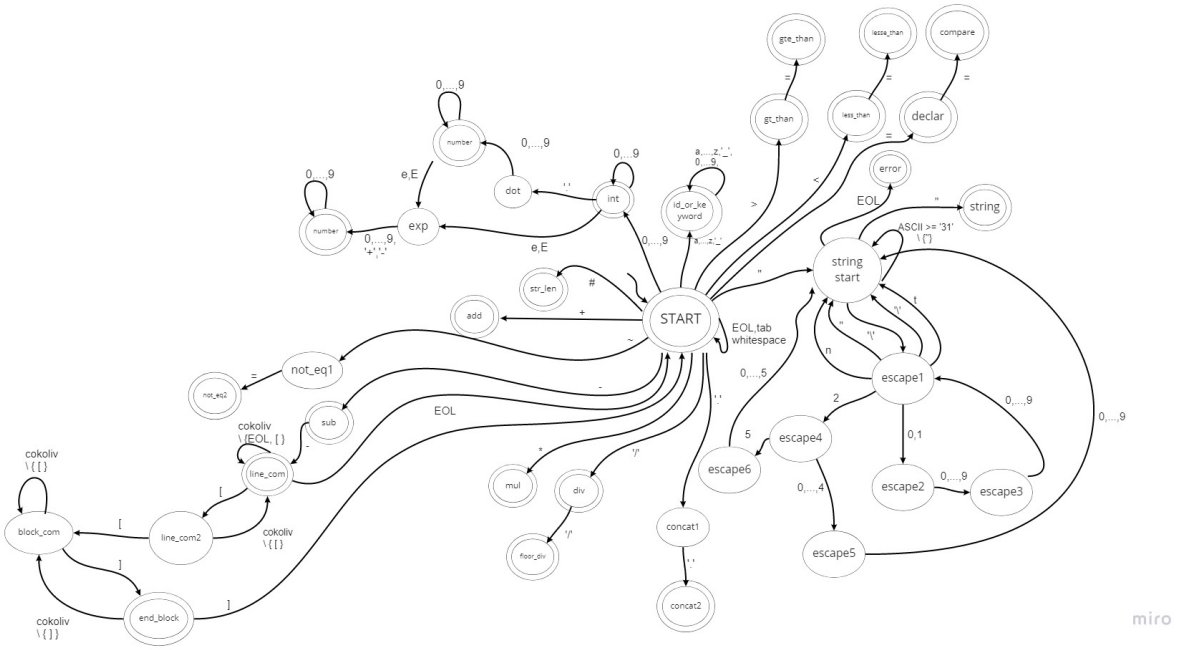
- Generování kódu
- ADT dynamický řetězec
- Testy pro lexikální analyzátor
- Precedenční analyzátor

2 Návrh překladače

2.1 Lexikální analyzátor

Lexikální analyzátor je implementován v souboru `scanner.c`.

2.1.1 Diagram konečného automatu



2.1.2 Reprezentace tokenů a symbolů

Datová struktura reprezentující tokeny je implementována v souboru `token_stack.h`.

Struktura je využívána nejen lexikálním analyzátorem, ale také tabulkou symbolů a během sémantických kontrol. Z tohoto důvodu obsahuje informace nejen o typu lexémy a atribut, ale také o datovém typu.

Při deklaraci proměnných a funkcí se jednoduše vezme token od scanneru, přiřadí se mu sémantické informace a je uložen do tabulky symbolů.

Umožňuje to také například přiřazovat datový typ literálům teprve vráceným lexikálním analyzátorem, což usnadňuje analýzu výrazů.

2.1.3 Funkce `unget_token(T_token*)`

Jelikož se syntaktický analyzátor nemůže u některých konstrukcí jazyka IFJ21 jednoznačně rozhodnout podle jednoho tokenu, implementuje scanner funkci pro vracení tokenů zpět.

Tato funkce natlačí token na zásobník tokenů [2.6.2], ze kterého potom bere funkce `get_next_token()` tokeny přednostně.

2.2 Syntaktický analyzátor

Syntaktický analyzátor je implementován v souboru `parser.c`.

Jako metodu pro analýzu shora-dolů jsme zvolili rekurzivní sestup. Pojmenování funkcí v kódu je vždy ve tvaru `rule_`NTER, kde NTER je jeden z neterminálů uvedených na levé straně gramatických pravidel [2.2.1].

2.2.1 LL-gramatika

1. PROG → **require** **lit_string** CODE
2. CODE → ϵ
3. CODE → TOP_ELEM CODE
4. TOP_ELEM → CALL
5. TOP_ELEM → DECL
6. TOP_ELEM → DEF
7. DECL → **global** **id** **:** **function** **(** TYPE_LIST **)** RET_LIST
8. DEF → **function** **id** **(** PARAM_LIST **)** RET_LIST BODY **end**
9. CALL → **id** **(** ARG_LIST **)**
10. RET_LIST → **:** TYPE NEXT_TYPE
11. RET_LIST → ϵ
12. TYPE_LIST → TYPE NEXT_TYPE
13. TYPE_LIST → ϵ
14. NEXT_TYPE → **,** TYPE NEXT_TYPE
15. NEXT_TYPE → ϵ
16. TYPE → **nil**
17. TYPE → **integer**
18. TYPE → **number**
19. TYPE → **string**
20. ARG_LIST → ARG NEXT_ARG
21. ARG_LIST → ϵ
22. NEXT_ARG → **,** ARG NEXT_ARG
23. NEXT_ARG → ϵ
24. ARG → **id**
25. ARG → **lit_number**
26. ARG → **lit_int**
27. ARG → **lit_string**
28. ARG → **nil**
29. PARAM_LIST → PARAM NEXT_PARAM
30. PARAM_LIST → ϵ
31. NEXT_PARAM → **,** PARAM NEXT_PARAM
32. NEXT_PARAM → ϵ
33. PARAM → **id** **:** TYPE
34. BODY → ϵ
35. BODY → STAT_LIST
36. STAT_LIST → VAR_DECL STAT_LIST
37. STAT_LIST → IF_ELSE STAT_LIST
38. STAT_LIST → WHILE STAT_LIST
39. STAT_LIST → **return** EXPR_LIST
40. VAR_DECL → **local** **id** **:** TYPE **=** EXPR
41. IF_ELSE → **if** EXPR **then** BODY **else** BODY **end**
42. WHILE → **while** EXPR **do** BODY **end**

2.2.2 LL-tabulka

	(	)	:	,	=	do	else	end	function	global	id	if	integer	lit_int	lit_number	lit_string	local	nil	number	require	return	string	then	while	$
PROG																				1					
CODE									3	3	3														2
TOP_ELEM									6	5	4														
DECL										7															
DEF									8																
CALL											9														
RET_LIST			10					11	11	11	11	11					11				11			11	11
TYPE_LIST		11											12					12	12			12			
NEXT_TYPE		15		14				15	15	15	15	15					15				15			15	15
TYPE													17					16	18			19			
ARG_LIST		21					21	21			20			20	20	20		20							
NEXT_ARG		23		22			23	23																	
ARG											24			26	25	27		28							
PARAM_LIST		30									29														
NEXT_PARAM		32		31																					
PARAM											33														
BODY							34	34				35					35				35			35	
STAT_LIST												37					36				39			38	
VAR_DECL																	40								
IF_ELSE												41													
WHILE																								42	

2.2.3 Konstrukce nepokryté LL-gramatikou

V jazyce IFJ21 jsou dvě konstrukce, které porušují LL(1) – jsou to příkazy *přiřazení* a *volání funkce bez přiřazení*:

1. `id ( ARG_LIST )`
2. `id , id , ... = EXPR , EXPR , ...`
3. `id , id , ... = id ( ARG_LIST )`

U pravidla `STAT_LIST` by tedy nastala kolize v množině *First*.

Rozhodnutí, který případ ve zpracovaném zdrojovém kódu nastal, probíhá získáním tokenu za identifikátorem.

Jestliže tento token je `(`, oba tokeny jsou vráceny scanneru [2.1.3] a pokračuje se pravidlem `rule_CALL`. V opačném případě se předpokládá přiřazení – sesbírají se identifikátory na levé straně (`left_side_function`) a předají se funkci zpracovávající přiřazení (`right_side_function`).

Zda na pravé straně je volání či seznam výrazů se určuje téměř identickým způsobem.

2.3 Precedenční analýza

Analyzátor výrazů je implementován v souboru `exp_parser.c`.

Jelikož v jazyce IFJ21 nejsou ukončovače výrazů, precedenční analyzátor rozlišuje následující případy:

1. Doposud přečtený výraz je platný a byl přečten nečekaný token – výraz je označen za platný, nečekaný token vrácen a řízení předáno top-down parseru
2. Přečtený výraz je zatím prázdný a hned první token je nečekaný – výraz je označen za prázdný, nečekaný token vrácen a řízení předáno top-down parseru (ten nepřítomnost výrazu může buď schválit, či zamítnout)
3. Přijatý výraz není platný a byl přečten nečekaný token – syntaktická chyba

2.3.1 Precedenční tabulka

	#	*, /, //	+, -	..	<, =, ...	(	)	id	$
#	>	>	>	>	>	<	>	<	>
*, /, //	<	>	>	>	>	<	>	<	>
+, -	<	<	>	>	>	<	>	<	>
..	<	<	<	<	>	<	>	<	>
<, =, ...	<	<	<	<	X	<	>	<	>
(	<	<	<	<	<	<	=	<	X
)	X	>	>	>	>	X	>	X	>
id	X	>	>	>	>	<	>	X	>
$	<	<	<	<	<	<	X	<	X

2.4 Sémantické kontroly

Sémantické kontroly jsou implementovány v souboru `semantics.c`.

2.5 Generování kódu

Vypisování cílového kódu na standardní výstup probíhá průběžně voláním funkcí implementovaných v souboru `code_gen.c`.

2.5.1 Řešení kolizí v návěštích u podmíněných příkazů a funkcí

Všechna návěští začínají pomlčkou, kterou nelze použít v IFJ21 v identifikátorech.

Návěští podmíněných příkazů jsou číslována podle čítače, který při každém použití příkazu `if` či `while` inkrementuje. Jsou navíc zakončeny pomlčkou, což znemožňuje kolize s funkcemi, které pomlčkou nekončí.

2.5.2 Parametry a návratové hodnoty funkcí

Před zavoláním funkce je vytvořen Temporary Frame, do kterého jsou přiřazeny potřebné parametry (`TF@%pN`), včetně případného přetypování.

Funkce po zavolání provede `PUSHFRAME` a přesune získané parametry do patřičných lokálních proměnných.

Návratové hodnoty se ukládají do `LF@%retvalN`. Po dokončení funkce se provede `POPFRAME` a volající z Temporary Frame přesune návratové hodnoty do lokálních proměnných (u volání s přiřazením).

2.5.3 Vyhodnocování výrazů

Kód pro vyhodnocování výrazů je generován přímo při precedenční analýze při redukci. K vyhodnocování je v IFJcode21 využíván datový zásobník.

Při přiřazování jsou nejdříve všechny výrazy vyhodnoceny (tím pádem zůstanou na datovém zásobníku) a poté pomocí `POPS LF@...` přiřazeny.

2.6 Tabulka symbolů

Podle zvolené varianty zadání jsme měli implementovat Tabulku symbolů pomocí binárních vyhledávacích stromů. Pro implementaci bylo použito hned několik abstraktních datových struktur.

2.6.1 AVL Strom

Jako základní datová struktura tabulky byl použit binární vyhledávací strom, konkrétně jejich samovyvažující varianta dvou Ruských matematiků Adeľson-Velského a Landise. Základ zdrojového kódu byl přebrán z druhého projektu z předmětu Algoritmy a doplněn, aby odpovídal specifikaci AVL stromů. Implementace se nachází v souboru `avl.c`.

2.6.2 Jednosměrně svázaný seznam

Jednosměrně svázaný seznam byla další datová struktura použitá nejen v tabulce symbolů, ale v rámci celého projektu a sloužila jako základ pro abstraktní datový typ zásobník. Implementace se nachází v souborech `sll.c` a `token_stack.c`.

V tabulce symbolů jsou tyto dvě struktury použity dohromady, kdy celá tabulka symbolů má následující rozložení:

1. Globální AVL strom použitý pro funkce.
2. Zásobník AVL stromů pro lokální proměnné.

Zásobník zde byl zvolen pro jeho vhodnost při simulování rámců platnosti.