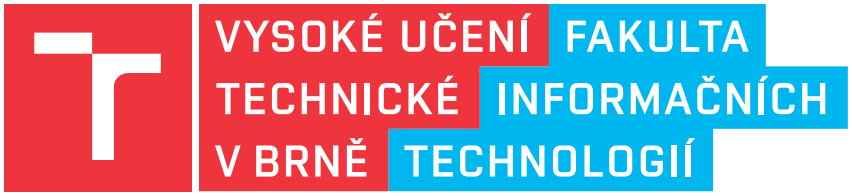


Překladač jazyka IFJ21 do IFJcode

IFJ - Formální jazyky a překladače

Tým 015, varianta I

Jakub Komárek **(xkomar33)** 30%
Křivánek Jakub (xkriva30) 24%
Matušík Adrián (xmatus35) 23%
Tverdokhlib Vladyslav V. (xtverd01) 23%

5. prosince 2021

Rozšíření: FUNEXP

Obsah

1 Úvod

Práce se zaobírá implementací překladače programovacího jazyka IFJ21(jazyk funkčně podobný jazyku Teal) na jazyk IFJ21code.

2 Struktura překladače

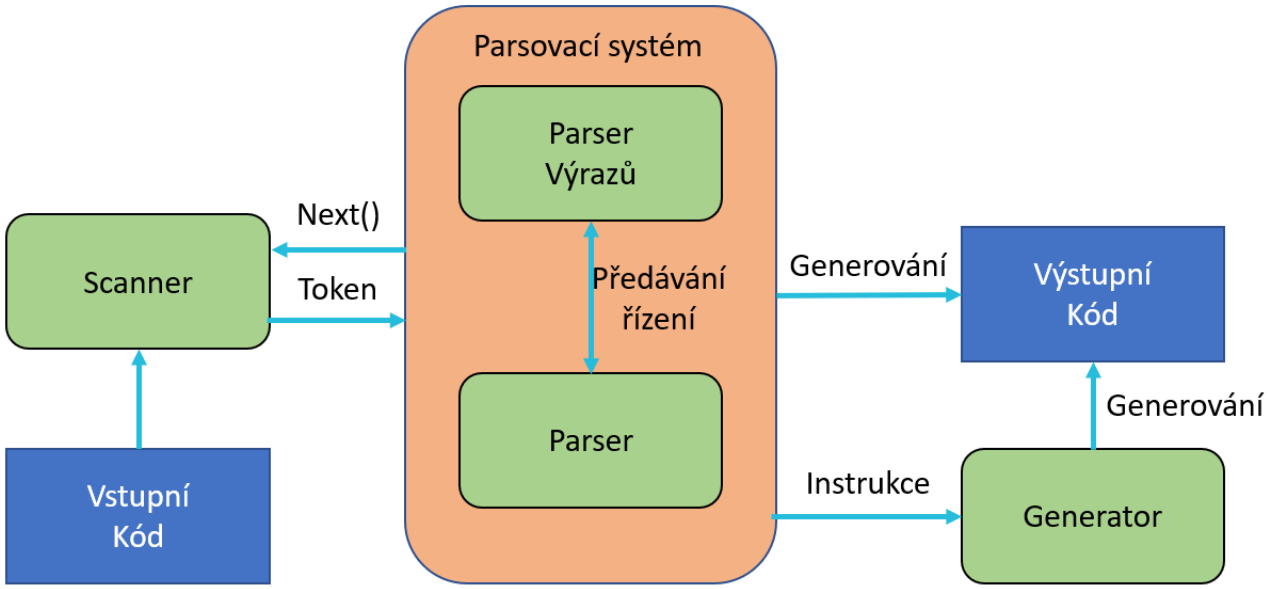
Tento překladač se skládá ze 3 hlavních částí - scanner, parser a generátor. Překlad řídí primárně parser, který se skládá ze dvou modulů - modul pro výrazy, který se stará o zpracování výrazů pomocí precedenční analýzy a modul pro parsování ostatních konstrukcí vstupního jazyka (pomocí LL(1) gramatiky4). Řídící logika parseru vhodně přepíná mezi těmito moduly.

Parser pracuje s tokeny, které mu na vyžádání poskytuje skener. Scanner čte vstupní kód po jednotlivých znakách a pomocí konečného automatu rozlišuje jednotlivé tokeny.

Výstupní kód se generuje přímo v modulech parseru, složitější konstrukce se generují přes modul generátor.

Jednotlivé moduly mezi sebou komunikují pomocí sdílené struktury nebo její části, kterou si předávají pomocí ukazatele. Tato struktura je členěna na části podle účelu. Tento přístup byl zvolen kvůli potřebě přistupovat ke stejným zdrojům z různých modulů.

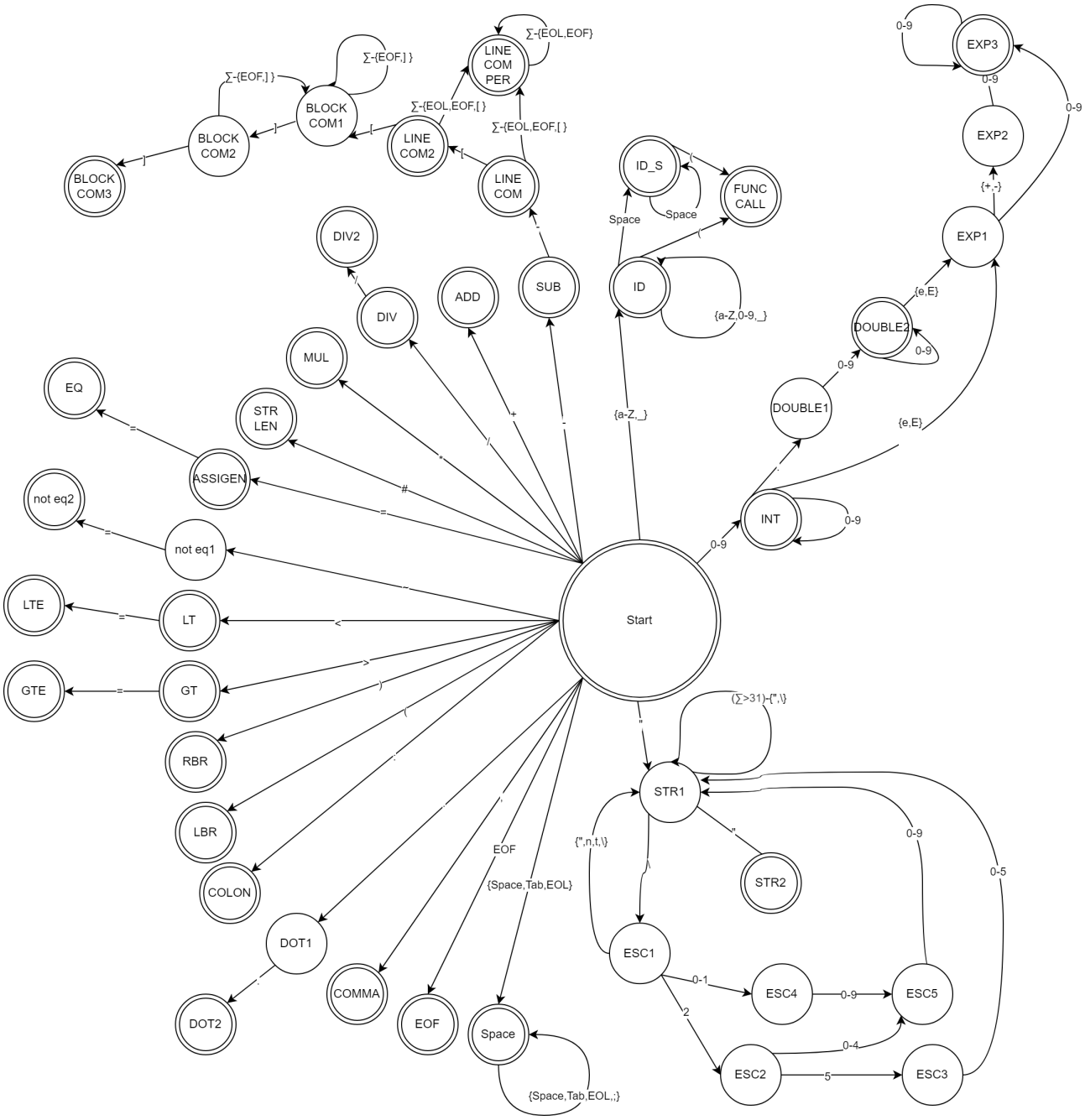
Pokud nastane v některé části překladu chyba, je zavolána obslužná funkce, která vygeneruje krátký report o chybě a následně skočí do funkce main na návěstí, kde v případě, že chyba nebyla vyvolána chybnou alokací paměti, dealokují zdroje aplikace a program vrátí příslušnou návratovou hodnotu.



Obrázek 1: Struktura

2.1 Scanner

Scanner je implementován podle konečného automatu 2 hladovou metodou - čte se vstup dokud automat nezhavaruje, pokud byl poslední stav konečný, vrací se jemu odpovídající token, jinak je nahlášena chyba na úrovni lexikální analýzy. Scanner zároveň rozpoznává klíčová slova a volání funkce.



Obrázek 2: Automat scanneru

2.2 Parser

Parser je implementován pomocí metody rekurzivního sestupu. Implementace je stavěna LL(1) tabulce4, kdy neterminál je vyobrazen v programu jako samostatná funkce. Vzhledem k podobnosti některých neterminálů, jsme některé neterminály vhodně spojili. Na principu kontroly tento zásah však nemá vliv.

Kontroly deklarací, datových typů, seznamů parametrů, atd. se provádí přímo při průchodu gramatickými pravidly. Stejně tak se provádí generování výsledného kódu. Je tedy zvolen jednoprůchodový způsob parsování vstupního kódu. Situace, kdy nelze generovat kód přímo (např.: dekalarace proměnných ve WHILE cyklu) jsme řešili pomocí pomocného zásobníku, který se ve vhodnou chvíli (typicky při konci WHILE cyklu) vyprázdnil.

2.3 Parser pro výrazy

Využívá metody precedenční analýzy. Generování kódu výrazů probíhá během průchodu algoritmem a to s využitím zásobníku(v cílovém kódu). Samotný výpočet se poté kvůli přetypování operátorů a kontrol na výskyt hodnoty nil provádí za pomocí před připravených registů, které jsou par využity u tříadresních instrukcí.

Algoritmus operuje s precedenční tabulkou 3, která je v kódu implementovaná jako konstantní dvojrozměrné pole.

2.3.1 Precedenční tabulka

Při implementaci se zjistilo, že některé operátory mají stejnou precedenci a tak byly některé operátory v tabulce sjednoceny.

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

Obrázek 3: Precedenční tabulka

2.4 Generování

Generování výstupního kódu probíhá při průchodu gramatickými pravidly. Po přečtení hlavičky vstupního kódu se vygeneruje hlavička výsledného kódu s vestavěnými funkcemi. Názvy proměnných se v jednotlivých rozsazích platnosti se odlišují pomocí dekorátoru, který zajišťuje, že daná proměnná nebude redefinována. Stejným způsobem se odlišují jednotlivé vygenerované struktury typu IF/WHILE.

Předávání argumentů mezi jednotlivými vygenerovanými funkcemi probíhá za pomocí zásobníku.

2.5 Datový model/tabulky symbolů

Tabulka symbolů je implementována pomocí binárního stromu. Jednotlivé rozsahy platnosti jsou rozděleny pomocí zásobníku na rámce - každá položka představuje jeden binární strom. Při vyhledání v datovém modelu se jde od hora zásobníku směrem ke spodu. Pokud se ve struktuře vyhledává lokální proměnná, vyhledává se v zásobníku, dokud se nenarazí na zarážku - představuje konec rozsahu platnosti funkce.

2.6 Rozšíření

Ačkoliv následující funkce nebyly v zadání, implementovali jsme do projektu podporu globálních proměnných a podporu lokálních funkcí. Zároveň překladač nehlásí chybu, pokud programátor omylem definoval funkci před deklarací.

Z rozšíření FUNEXP jsme implementovali výrazy v parametrech volání funkce (funkce ve výrazech není implementovaná).

3 Rozdělení práce

Důvod odchylky od rovnoměrného rozdělení bodů je z důvodu nerovnoměrného rozsahu odvedené práce.

Jakub Komárek - implementace parseru, scaneru, části generátoru a pomocných knihoven
Křivánek Jakub - generování výsledného kódu a vestavěných funkcí, tvorba testovací sady
Matušík Adrián - testování, návrh gramatiky a konečného automatu
Tverdokhlib Vladyslav V. - Implementace tabulky symbolů a datových struktur pro ukládání vlastností symbolů

4 Závěr

Překladač umí všechny požadované funkce, pomocí testovací sady jsme otestovali základní funkčnost.

Zjistili jsme nedostatek v podobě deklarace proměnné v cyklu WHILE a její inicializaci pomocí proměnné stejného názvu z předchozího rozsahu platnosti. Tato chyba nelze v našem překladači triviálně vyřešit, podkopává principy na kterých je překladač postavený.

5 Gramatika

1. START → PROLOG PROG $
2. PROLOG → require string
3. PROG → id ID_NEXT assigen EXP_OR_FUNC PROG
4. PROG → FUNCTION PROG
5. PROG → DECLARATION PROG
6. PROG → WHILE PROG
7. PROG → IF PROG
8. PROG → RETURN
9. PROG → FUNC_CALL PROG
10. PROG → ϵ
11. EXP_OR_FUNC → expresion N_EXPRESIONS
12. EXP_OR_FUNC → FUNC_CALL
13. FUNC_CALL → func_call F_ARG rbr
14. F_ARG → ϵ
15. F_ARG → expresion F_ARG_N
16. F_ARG_N → ϵ
17. F_ARG_N → comma expresion F_ARG_N
18. ID_NEXT → comma id ID_NEXT
19. ID_NEXT → ϵ
20. N_EXPRESIONS → comma expresion N_EXPRESIONS
21. N_EXPRESIONS → ϵ
22. WHILE → while expresion do PROG end
23. IF → if expresion then PROG ELSE_M end
24. ELSE_M → else PROG
25. ELSE_M → ϵ
26. FUNCTION → function func_call ARG rbr RETURN_D PROG end
27. ARG → ϵ
28. ARG → id colon TYPE ARGNEXT
29. ARGNEXT → comma id colon TYPE ARGNEXT
30. ARGNEXT → ϵ
31. RETURN_D → colon TYPE RETURN_DN
32. RETURN_D → ϵ
33. RETURN_DN → comma TYPE RETURN_DN
34. RETURN_DN → ϵ
35. RETURN → return RETURN_ARG
36. RETURN_ARG → expresion RETURN_ARG_N
37. RETURN_ARG → ϵ
38. RETURN_ARG_N → comma expresion RETURN_ARG_N
39. RETURN_ARG_N → ϵ
40. DECLARATION → RANGE id colon DECLARATION_T
41. DECLARATION_T → TYPE ASSIGEN_MAY
42. DECLARATION_T → function lbr ARG_D rbr colon RET_D
43. ARG_D → ϵ
44. ARG_D → TYPE ARG_DN
45. ARG_DN → comma TYPE ARG_DN
46. ARG_DN → ϵ
47. RET_D → ϵ

48. RET_D → TYPE RET_DN
49. RET_DN → comma TYPE RET_DN
50. RET_DN → ϵ
51. ASSIGEN_MAY → ϵ
52. ASSIGEN_MAY → assigen EXP_OR_FUNC
53. TYPE → integer
54. TYPE → nil
55. TYPE → number
56. TYPE → string
57. RANGE → local
58. RANGE → global

6 LLtabulka

	$	require	string	id	assigen	expresion	func_call	rbr	comma	while	do	end	if	then	else	function	colon	return	lbr	integer	nil	number	local	global
START		1																						
PROLOG		2																						
PROG				3			9			6			7			4		8					5	5
EXP_OR_FUNC						11	12																	
FUNC_CALL							13																	
F_ARG						15																		
F_ARG_N									17															
ID_NEXT									18															
N_EXPRESIONS									20															
WHILE										22														
IF													23											
ELSE_M															24									
FUNCTION																26								
ARG				28																				
ARGNEXT									29															
RETURN_D																	31							
RETURN_DN									33															
RETURN																		35						
RETURN_ARG						36																		
RETURN_ARG_N									38															
DECLARATION																							40	40
DECLARATION_T			41													42				41	41	41		
ARG_D			44																	44	44	44		
ARG_DN									45															
RET_D			48																	48	48	48		
RET_DN									49															
ASSIGEN_MAY					52																			
TYPE			56																	53	54	55		
RANGE																							57	58

Obrázek 4: LL(1) tabulka