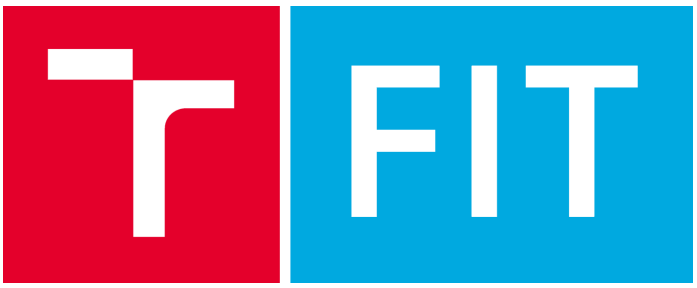


Dokumentace projektu z předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ21

Tým 070, varianta II

Vilém Gottwald (xgottw07) - 39 %
Pavel Marek (xmarek75) - 21 %
Daniel Ponížil (xponiz01) - 20 %
Štěpán Bílek (xbilek25) - 20 %

8. prosince 2021

1. Úvod

Cílem našeho týmového projektu byla implementace programu, který přeloží kód napsaný ve zdrojovém jazyce IFJ2021 do jazyka IFJcode21.
Program načítá zdrojový kód ze standardního vstupu a následně z něj vygeneruje mezikód na standardní výstup, v případě chyby vypíše chybové hlášení na standardní výstup.

2. Práce v týmu

Na projektu jsme začali pracovat v průběhu listopadu, neměli jsme konkrétní představu o jeho časové náročnosti, takže poslední dny před odevzdáním jsme museli výrazně zvýšit intenzitu práce.

Jako komunikační kanál jsme používali platformu discord, kde jsme měli vytvořený server právě pro tento projekt. Discord umožňuje použití jak chatu, tak i skupinových hovorů, které podporují hromadné sdílení obrazovek, to spolupráci výrazně usnadnilo.

Pro správu projektu jsme používali verzovací systém Git, jako vzdálený repozitář jsme používali GitHub.

3. Rozdělení práce

Vilém Gottwald (xgottw07) - Vedoucí týmu, lexikální analýza, tabulka symbolů, LL-gramatika, dynamický řetězec, syntaktická analýza, dohled nad ostatními částmi projektu, testování
Pavel Marek (xmarek75) - syntaktická a sémantická analýza, dokumentace, testování
Štěpán Bílek (xbilek25) - generování kódu, dokumentace, testování
Daniel Ponížil (xponiz01) - syntaktická a sémantické analýza výrazů, dokumentace, testování

4. Vývojový cyklus

Po vytvoření týmu jsme na první schůzce vytvořili pravidla pro LL-gramatiku, vytvořili LL-tabulku a rozdělili si práci. První byl vytvořen scanner, ten je zásadní pro fungování všech ostatních částí překladače. Následně začal vývoj syntaktického analyzátoru a části realizující precedenční syntaktickou analýzu. K syntaktickému analyzátoru jsme postupně přidali i sémantické kontroly. Na závěr byl vytvořen generátor kódu. Všechny části se průběžně testovaly. Nejprve probíhaly jednotkové testy, po dokončení programu jako celku jsme testovali systémovými testy.

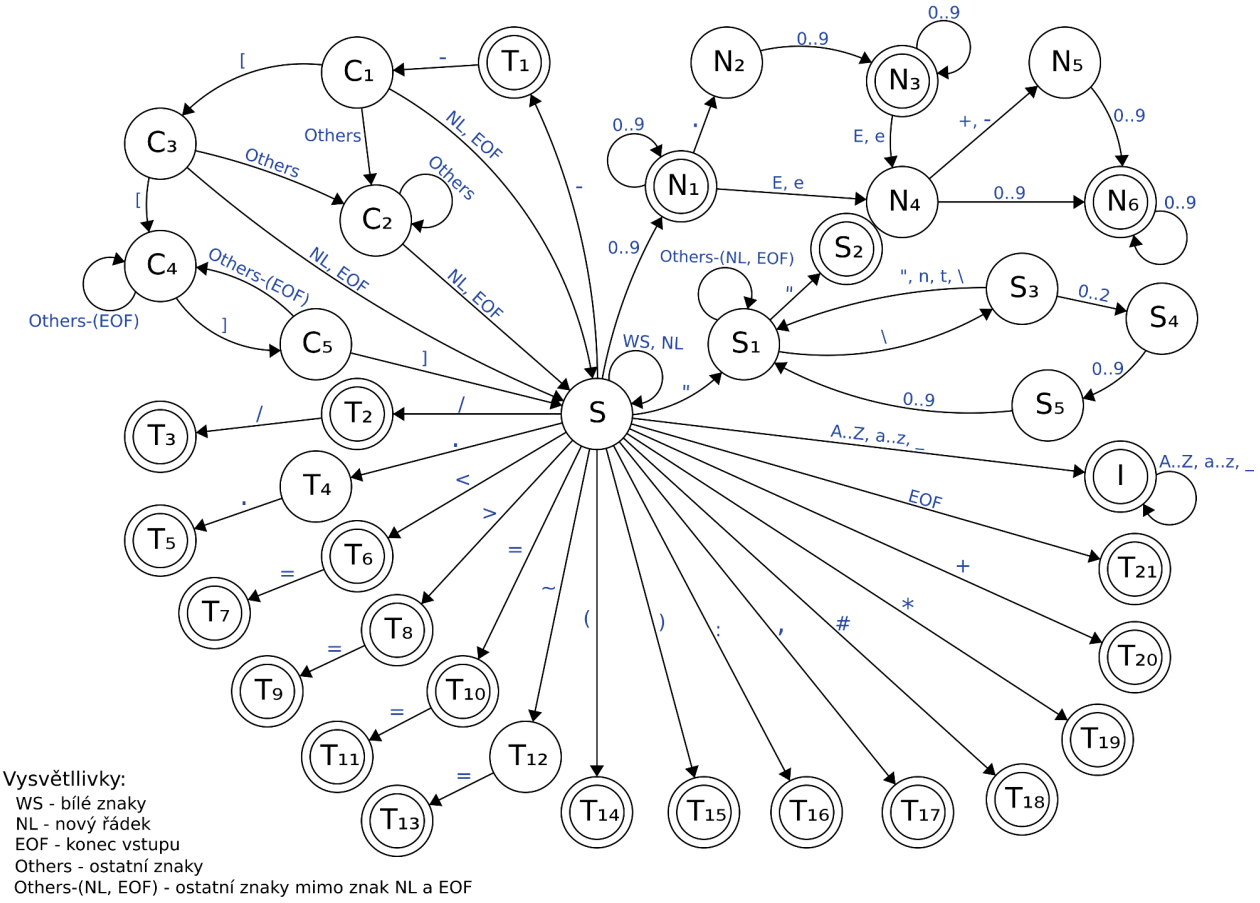
5. Lexikální analýza

Lexikální analýza je implementována na základě níže uvedeného konečného automatu, přičemž základní datovou strukturou syntaktické analýzy je token. Token je implementován jako struktura typu a atributu, kde atribut je *union*. K hodnotě atributu se přistupuje na základě vyhodnocení typu.

Činnost lexikálního analyzátoru je vyvolána zavoláním funkce *get_token*, které je jako parametr předán ukazatel na token, jehož hodnota má být aktualizována. Přechody, které nejsou v diagramu konečného automatu zobrazeny vedou na lexikální chybu s návratovou hodnotou 1. Neomezená délka řetězců a názvů identifikátorů je řešena využitím dynamicky alokovaného řetězce, do nějž jsou postupně vkládány znaky, přičemž při vyčerpání alokovaného prostoru dochází

k realokaci (implementace v *dynamic_string.c*). Zdrojové kódy lexikální analýzy se nachází v souborech *scanner.c* a *scanner.h*.

Diagram konečného automatu lexikálního analyzátoru



6. Syntaktická analýza

Při implementaci syntaktické analýzy jsme postupovali hlavně podle LL-gramatiky metodou rekurzivního sestupu podle pravidel v LL-tabulce.

Syntaktická analýza (kromě výrazů) je implementována v souboru *syntax_analysis.c* a její rozhraní pro implementaci v *syntax_analysis.h*.

Při běhu programu si program bere tokeny pomocí funkce get_token, která je implementována v souboru *scanner.c*, pomocí příkazu *get_token(data->token)*, kde *data* je struktura obsahující proměnné a ukazatele na další struktury potřebné k implementaci. Funkce následně nahradí aktuální token tím následujícím. Poté následuje podmínka, podle které se program rozhodne, kterou funkci spustí dál, případně vrátí chybové hlášení. Tento koloběh se neustále opakuje dokud dostává tokeny, nebo dokud nenajde chybu.

LL-gramatika

1. REQ ::= require str_val PROG
2. PROG ::= global id : function (PAR_TYPES) TYPES PROG
3. PROG ::= function id (PARAMS) TYPES BODY end PROG
4. PROG ::= id (ARGS) PROG
5. PROG ::= eof
6. TYPES ::= : TYPE NEXT_TYPE
7. TYPES ::= ε
8. TYPE ::= integer
9. TYPE ::= number
10. TYPE ::= string
11. TYPE ::= nil
12. NEXT_TYPE ::= , TYPE NEXT_TYPE
13. NEXT_TYPE ::= ε
14. PARAMS ::= id : TYPE NEXT_PARAM
15. PARAMS ::= ε
16. NEXT_PARAM ::= , id : TYPE NEXT_PARAM
17. NEXT_PARAM ::= ε
18. ARGS ::= TERM NEXT_TERM
19. ARGS ::= ε
20. NEXT_TERM ::= , TERM NEXT_TERM
21. NEXT_TERM ::= ε
22. TERM ::= integer_val
23. TERM ::= number_val
24. TERM ::= string_val
25. TERM ::= nil
26. TERM ::= id
27. BODY ::= local id : TYPE INIT BODY
28. BODY ::= if EXPRESSION then BODY else BODY end BODY
29. BODY ::= while EXPRESSION do BODY end BODY
30. BODY ::= return EXPRESSIONS
31. BODY ::= id ASSIGN BODY
32. BODY ::= ε
33. EXPR_OR_FNC ::= EXPRESSION
34. EXPR_OR_FNC ::= id (ARGS)
35. EXPRESSIONS ::= EXPRESSION NEXT_EXPRESSION
36. EXPRESSIONS ::= ε
37. NEXT_EXPRESSION ::= , EXPRESSION NEXT_EXPRESSION
38. NEXT_EXPRESSION ::= ε
39. ASSIGN ::= (ARGS)
40. ASSIGN ::= NEXT_ID = EXPRS_OR_FNC
41. NEXT_ID ::= , id NEXT_ID
42. NEXT_ID ::= ε
43. EXPRS_OR_FNC ::= EXPRESSION NEXT_EXPRESSION
44. EXPRS_OR_FNC ::= id (ARGS)
45. INIT ::= ε
46. INIT ::= = EXPR_OR_FNC
47. PAR_TYPES ::= TYPE NEXT_TYPE
48. PAR_TYPES ::= ε

* znak ε značí prázdný řetězec

7. Sémantická analýza

Sémantická analýza je prováděna typovými kontrolami ve vhodných částech derivačního stromu generovaného syntaktickou analýzou. Pomocné proměnné využívané sémantickou analýzou jsou definované v datové struktuře *data* ve zdrojovém souboru *expression.h.* V průběhu sémantické analýzy jsou do tabulek symbolů identifikárů a tabulky symbolů funkcí přidávány nově definované funkce a proměnné, včetně informací o jejich datových typech.

Tabulka symbolů funkcí je implementována jako tabulka s rozptýlenými položkami, nejdůležitějšími funkcemi pro práci s ní jsou *STF_add_symbol*, sloužící pro přidání symbolu do tabulky symbolů a *STF_search*, která slouží k vyhledání symbolu v tabulce.

Tabulka symbolů identifikátorů je implementována jako lineární seznam tabulek symbolů. K vytvoření nové hloubější dílčí tabulky symbolů slouží funkce *STV_L_deeper*, ta je volána při vstupu do nového bloku s vlastním rozsahem platnost. Naopak funkce *STV_L_uppper,* která ze seznamu vymaže nejhlubší tabulku symbolů, je volána ve chvíli, kdy z bloku naopak vystupujeme. U každé dílčí tabulky symbolů je zaznamenána úroveň její hloubky, tuto hodnotu lze pro momentálně nejhlubší tabulku zpřístupnit pomocí funkce *STV_L_get_current_level.* Pro vyhledání symbolu v seznamu tabulek slouží funkce *STV_L_search,* která prohledá všechny tabulky a vrátí nejhlubší výskyt tohoto symbolu. Funkce *STV_L_add* vloží vybraný symbol do nejhlubší tabulky symbolů proměnných. Implementaci tabulky symbolů lze nalézt v souborech *symtable.c* a *symtable.h.*

8. Výrazy

Zpracování výrazů probíhá pomocí precedenční syntaktické analýzy, která je implementovaná v souboru *expression.c* a hlavičkovém souboru *expression.h*. Výrazy zároveň pracují se zásobníkem, který je implementován v souboru *stack.c* a jeho hlavičkovém souboru *stack.h*. Při zpracování výrazů jsme vycházeli z precedenční tabulky viz. tabulka:

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

REL - relační operátory <, <=, >, >=, ==, ~=

… Ta nám určí prioritu jednotlivých operátorů. Některé operátory mají stejnou prioritu a asociativitu, tedy se vyskytují v jedné kolonce.

Precedenční analýzu spouštíme ze souboru *syntax_analysis.c*. Po spuštění výrazů si funkce pomocí aktuálního tokenu a terminálu na vrcholu zásobníku zjistí index z precedenční tabulky, a následně spustí operaci tomu určenou. Operace *O_LE* nám primárně vloží zarážku *STOP* na zásobník, dále vloží aktuální token a spustí funkci *get_token*, která získá nový token ze scanneru. *O_MO* naopak spustí funkci reduce, která pracuje s tokeny na zásobníku. Počet tokenů zjistí podle

zarážky *STOP*. Funkce *reduce* také zajišťuje spuštění funkce syntaktické precedenční analýzy, ze které získá pravidla pro sémantickou analýzu. Ta je volána téměř hned poté. Pokud projdou všechny pravidla, funkce vyčistí zásobník po zarážku *STOP* (vyčistí i zarážku). *O_EQ* operace se spustí v jediném případě, a to, když se v aktuálním tokenu a terminálu na vrcholu zásobníku objeví závorky. Vloží aktuální token na zásobník a zavolá *get_token*. Pokud nastane stav *O_ER*, znamená to, že by aktuální token a terminál měl být dolar. Pokud toto platí, analýza je úspěšná, v opačném případě končí neúspěchem. Poměrně složitější bylo implementovat práci programu s datovým typem *NIL*. Ten totiž neměl definovaný vlastní datový typ, ale univerzální datový typ *keyword*. Museli jsme s ním tedy pracovat jako s atributem datového typu *keyword*. Komplikaci nám také udělal možnost mít více výrazů na jednom řádku. Pokud nám přišel *identifikátor* bez jakékoliv operace (+,-,*...), vrátili jsme samotný *identifikátor* zpět do analýzy v *syntax_analysis.c*, která to následně dál zpracovala.

9. Generování kódu

Generování do mezikódu probíhá přímo při syntaxí řízeném překladu, instrukce cílového jazyka IFJcode21 jsou generovány na standardní výstup. Na začátek každého programu se vygeneruje úvodní řádek `.IFJcode21.` Dále se generují instrukce, se kterými se generují i stručné komentáře, které dělají výsledný kód přehlednějším. Funkce pro generování instrukcí jsou implementovány v souboru *code_generator.c* jehož rozhraní se nachází v souboru *code_generator.h*. Instrukce vestavěných funkcí se pomocí maker generují až na konec programu. Každou funkci tvoří návěští s jejím názvem a vytvoří se lokální rámec, do kterého se nahraje její návratová hodnota. Před zavoláním funkce jsou parametry se svým indexem uloženy do dočasného rámce, před vstupem do funkce jsou přesunuty do lokálního rámce. Před návratem z funkce je její výsledek předán proměné pro návratovou hodnotu a je proveden skok na konec funkce. Generování výrazu probíhá přímo při jeho zpracování v souboru *expression.c*. Podle arity se rozhodne, zda se jedná o výraz obsahující pouze neterminál, nebo o výraz s binárním, či unárním operátorem. Pro generování proměnného počtu parametrů vestavěné funkce *write* bylo využito datového zásobníku, z něj jdou však vytahovat parametry jen v opačném pořadí, než jsou na vstupu, pro tento účel byl využit zásobník na tokeny implementovaný v souboru *param_stack.c* s rozhraním v *param_stack.h*. S využitím obou zásobníků následně dostaneme parametry v původním pořadí. Výrazy jsou generovány přímo s využitím tří adresného kódu. Pro odlišení jednotlivých vnitřních proměnných při generování výrazů je využit index výrazu.

10. Členění souborů

- *scanner.c*, *scanner.h* - lexikální analýza
- *syntax_analysis.c*, *syntax_analysis.h* - zpracování metodou rekurzivního sestupu
- *expression.c*, *expression.h* - zpracování výrazů metodou precedenční analýzy
- *code_generator.c*, *code_generator.h* - funkce pro generování kódu
- *symtable.c*, *symtable.h* - tabulky symbolů
- *error.h* - výčet návratových hodnot
- *stack.c*, *stack.h* - zásobník využívaný precedenční analýzou
- *param_stack.c*, *param_stack.h*, *dstr_queue.c*, *dstr_queue.h* - pomocné datové struktury pro generování kódu
- *dynamic_string.c*, *dynamic_string.h* - knihovna pro práci s dynamickými řetězci

Přílohy

	require	global	id	function	(	)	end	eof	integer	number	string	nil	,	integer_val	number_val	string_val	local	=	if	EXPRESSION	else	while	return
REQ	1.																						
PROG		2.	4.	3.				5.															
TYPES		7.	7.	7.		7.	7.	7.	6.	6.	6.	6.					7.		7.			7.	7.
TYPE									8.	9.	10.	11.											
NEXT_TYPE		13.	13.	13.		13.	13.	13.					12.				13.		13.			13.	13.
PARAMS			14.			15.																	
NEXT_PARAM						17.							16.										
ARGS			18.			19.						18.		18.	18.	18.							
NEXT_TERM						21.							20.										
TERM			26.									25.		22.	23.	24.							
BODY			31.				32.										27.		28.		32.	29.	30.
EXPR_OR_FNC			34.																	33.			
EXPRESSIONS							36.													35.	36.		
NEXT_EXPRESSION			38.				38.						37.				38.		38.		38.	38.	38.
ASSIGN					39.								40.					40.					
NEXT_ID													41.					42.					
EXPRS_OR_FNC			44.																	43.			
INIT			45.				45.										45.	46.	45.		45.	45.	45.
PAR_TYPES						48.			47.	47	47.	47.											

Příloha 1: LL - tabulka