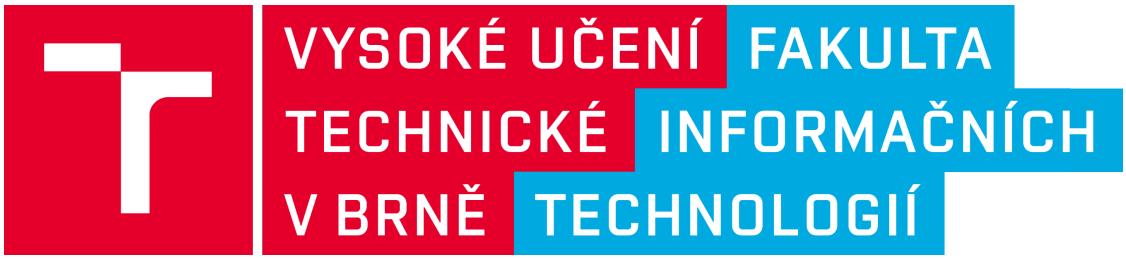


Dokumentace

Implementace překladače imperativního jazyka IFJ24

Tým xbohatd00, Varianta - TRP-izp

Daniel Bohata (xbohatd00) 25%
Tadeáš Horák (xhorakt00) 25%
Ivo Puchnar (xpuchn02) 25%
Adam Vožda (xvozdaa00) 25%

Rozšíření: FUNEXP

Brno, 4. prosince 2024

Obsah

1 Práce v týmu

1.1 Rozdělení práce mezi jednotlivé členy týmu

- Daniel Bohata
  - Vedení
  - Návrh ASS
  - Generace kódu
- Tadeáš Horák
  - Návrh ASS
  - Implementace ASS
  - Sémantická analýza
- Ivo Puchnar
  - Lexikální analýza
  - Dokumentace
- Adam Vožda
  - Implementace ASS
  - Syntaktická analýza

1.2 Průběh práce

Na projektu jsme začali pracovat hned. Po značném rozpracování tabulky symbolů, lexikální a syntaktické analýzy jsme ale práci pozastavili. K projektu jsme se naplno vrátili během 8. týdne po půlsemestrálních zkouškách. Následně se naše úsilí zvyšovalo s blížícím se termínem. Každý si vzal určitou část s tím, že jsme si navzájem pomáhali, když někdo potřeboval. Vedoucí nenechával nic náhodě, ale nevytvářel přílišný nátlak.

1.3 Sdílení kódu

Pro sdílení kódu jsme použili Git/GitHub. Systém za nás udržoval přehled o verzích a jejich kompatibilitě. Pro každou část jsme vytvořili zvláštní větev. Po dokončení části jsme větev spojili s větví *main*, kde jsme udržovali plně funkční program.

1.4 Komunikace

Seznámení týmu proběhlo osobně. Následná komunikace probíhala hlavně přes skupinové zprávy na Instagramu. Když bylo potřeba sdílet nějaký soubor nebo více obrázků, použili jsme skupinu na Discordu. Prezentaci na obhajobu jsme rovněž udělali na osobní schůzce.

2 Lexikální analýza

Lexikální analýza se provádí funkcí *get_ token()*, která je definována v souboru *Lexem_ an.c*. Její pomocné struktury a funkce jsou definovány v souborech *tokens.h, .c*.
Syntaktická analýza zavolá *get_ token()*, který vrací strukturu *Token*. Druh tokenu se určuje podle atributu *KeyWord kw*. V atributech *int i, double f, char *s* se předávají hodnoty tokenů pro identifikátory, čísla a stringy.
Lexikální analyzátor jsem vytvořil z lexikálních analyzátorů, které jsem udělal pro projekty posledních dvou let.

2.1 Inicializace

Na začátku se zavedou proměnné a alokuje paměť.
Proměnné *int n* a *double d* slouží k předání celých a desetinných čísel do tokenů. *Token new* drží strukturu *Token*, kterou funkce po úpravách lexikální analýzou vrací. *char letter* si pamatuje poslední znak ze vstupu. Pokud funkce končí a poslední znak se nevyužil, vrací ho zpět na vstup. *size_ t lex_ size* udržuje aktuální maximální alokovanou délku *char *lexem*. Při spuštění funkce začíná na 8. *char *lexem* je dynamicky alokovaný string pro uchovávání dlouhých lexémů. Před přidáním dalšího znaku funkce zkontroluje, zda má **lexem* dostatek místa. Pokud ne, realokuje jeho obsah s dvojnásobnou velikostí a tu zapíše do *lex_ size*. *char *p* je pomocný pointer. Při realokaci **lexem* na něm zjistíme, zda realokace proběhla úspěšně, a při předávání stringu do tokenu do něj alokujeme adekvátní velikost stringu. Alokovanou paměť následně předáme do atributu **s* tokenu.

2.2 Postupná úprava analyzátoru

První verze lexikálního analyzátoru fungovala tak, že se funkce zavolala zvlášť před ostatními analýzami a vytvořila jednostranně vázaný, který pak mohli procházet, ale pro splnění podmínky syntaxí řízené analýzy jsme ho změnili na vracení jednotlivých tokenů. V kódu můžete najít některé pozůstatky této verze, např. while cyklus pod inicializací. Pokud prvek neubíral funkčnosti, ponechali jsme ho.

2.3 Určování tokenu

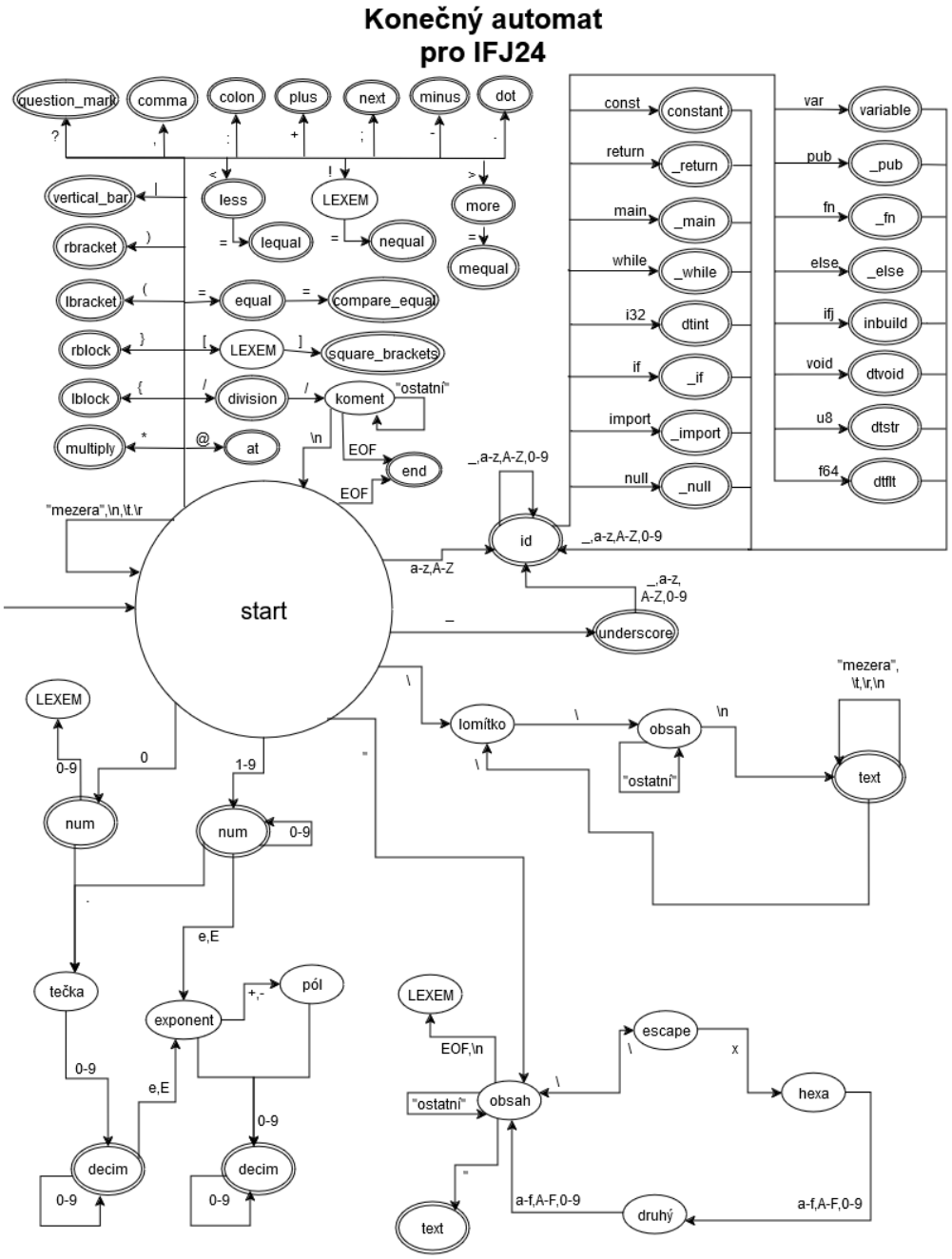
Funkce *get_ token()* rozhoduje o tokenu podle příchozího znaku. Tento znak vloží do velkého switche.
Na začátku switche najdeme bílé znaky, které analyzátor přeskočí. Komentáře se rovněž přeskakují, jen se nacházejí níže.
Následují jednoznačné tokeny. Jinak řečeno, tokeny vyžadující pouze 1-2 znaky.
Víceřádkové stringy se poznají podle dvou obrácených po sobě jdoucích lomítek "\\"a posílají se stejně jako řádkové, v atributu tokenu **s* s atributem *kw* nastaveným na *text*. Rozdíl je v jejich tvorbě. Do stringu se přidá vše, co se nachází za "\\", včetně konce řádku '\n'. Samotná lomítka se neposílají. Pokud se na dalším řádku, po bílých znacích, nachází další "\\", proces se opakuje, jinak se poslední '\n' smaže a token se pošle.
Jednořádkové stringy se poznají podle dvojitých uvozovek ". Pokud následují dvě "po sobě, jedná se o prázdný string a může se tedy ihned poslat token s *text*. V **s* se pošle pouze '\0'. Pokud následují znaky, přidají se do odevzdávaného stringu. Pokud cyklus narazí na obrácené

lomítko '\' vyzkouší, zda se jedná o escape sekvenci. Pokud ne, dojde k chybě 1. Pokud následuje 'x', zkontroluje se, zda následující dva znaky odpovídají zápisu hexadecimálního čísla. Pokud ano, hodnota se převede na znak, který se přidá do stringu, jinak nastává chyba 1. Druhé "ukončí nabírání znaků a pošle se token. String se posílá bez ohraničujících ".
Mezi celými a desetinnými čísly rozlišujeme podle přítomnosti tečky nebo exponentu. Čísla začínají jako celá po příchodu číslovky, tedy znak 0 až 9. Pokud jako první přijde 0 a následují další číslice, jedná se o chybu 1. Pokud po číslovkách následuje '.' nebo 'e' číslo je jednoznačně desetinné. Po '.' a alespoň jedné číslovce může stále přijít 'e'. Po 'e' musí také následovat alespoň jedna číslovka. Před ní se může objevit '+' nebo '-'. Pokud přijatý znak už není jeden z jmenovaných, string se převede na číslo, vloží se do atributu i pro celá a f pro desetinná čísla. Pro jistotu se číslo pošle i jako string v **s*.
Identifikátory a klíčová slova začínají buď písmeny nebo podtržítkem '_'. Pokud za '_' nenásledují písmena, čísla ani '_', jedná se o prázdnou proměnnou a odevzdá se její token. V opačném případě se do řetězce pomocí cyklu vezmou všechny zmíněné znaky. Aby se snížila režie, porovná se první znak řetězce s prvními znaky všech možných klíčových slov. Tam, kde se shodují, se řetězec porovná s konkrétním klíčovým slovem, díky čemuž se vynechají porovnávání s ostatními. Pokud se řetězec shoduje s celým slovem, vyšle se jeho token, jinak se vyšle token *id*, s řetězcem v **s*.
Pokud funkce najde EOF, pošle token *end* pro oznámení konce souboru.

2.4 Vysvětlení konečného automatu

- Ovály s jednoduchým okrajem jsou obyčejné stavy a ovály s dvojitým okrajem jsou stavy konečné.
- Pro přehlednost jsou některé přechody mezi stavy spojeny do jedné čáry. Pokud tomu tak je, symbol, který přechodu náleží, je vyobrazen tak, aby se nedaly zaměnit. Buď u jednotlivých přechodů, nebo u přechodů se stejným znakem směřujících do stejného stavu.
- Některé přechody mají šipku oběma směry. To znamená, že se při správném znaku lze vrátit do předchozího stavu.
- U hledání identifikátorů automat zůstává ve stavu *id*, dokud řetězec neodpovídá celému klíčovému slovu. Poté přejde do odpovídajícího stavu. Pokud ale řetězec pokračuje vrací se do *id*, kde už zůstává.
- Pokud automat dojde do konečného stavu a dostane znak neodpovídající žádnému z jeho přechodů, znak vrátí na vstup a odešle token odpovídající aktuálnímu konečnému stavu s případným obsahem (číslo/string).
- Pokud automat dojde do běžného stavu a dostane znak neodpovídající žádnému z jeho přechodů, dochází k chybě číslo 1, chyba v lexému, a program končí.
- Automat začíná velkým stavem *start* vlevo uprostřed. Vlevo nahoře se nachází oblast tokenů tvořených 1-2 znaky. Vpravo nahoře jsou klíčová slova a identifikároty proměnných a funkcí. Vpravo uprostřed jsou víceřádkové stringy. Vpravo dole jsou jednořádkové stringy

2.5 Diagram konečného automatu



Obrázek 1: Diagram konečného automatu

3 Syntaktická analýza

Syntaktická analýza je implementována v souboru `syntax_an.c` (s odpovídajícím hlavičkovým souborem `syntax_an.h`) a je realizována pomocí rekurzivního sestupu na základě LL gramatiky. Tento soubor obsahuje klíčovou funkci překladače, *syntax_ analyzer()*, která provádí veškeré potřebné operace před a po syntaktické analýze.

Proces začíná voláním funkce *headers()*, která načte počáteční pravidlo (import) a následně zpracovává hlavičky funkcí. Tyto hlavičky jsou uloženy do symbolické tabulky, zatímco těla funkcí jsou během prvního průchodu ignorována. Po načtení hlaviček se vstupní proud resetuje na začátek souboru pomocí funkce *rewind_ stdin()*.

Následuje druhý průchod, zahájený voláním funkce *program()*, který provádí podrobnou analýzu celého programu. V tomto kroku se také provádí sémantická kontrola a vytváří se abstraktní syntaktický strom (ASS). ASS je generován pomocí funkcí, které vytvářejí jednotlivé uzly a propojují je do hierarchické struktury. Po dokončení syntaktické analýzy je ASS předán do fáze generace kódu.

Tokeny jsou načítány lexikálním analyzátorem prostřednictvím funkce *read_ token()*, která volá funkci *get_ token()*. Tato funkce slouží jako rozhraní mezi lexikální a syntaktickou analýzou. Parser zpracovává všechny části LL gramatiky s výjimkou výrazů. Výrazy jsou předávány metodě precedenční syntaktické analýzy, která je samostatně vyhodnocuje.

3.1 Přechod mezi rekurzivním sestupem a precedenční analýzou

Přechod mezi rekurzivním sestupem a precedenční analýzou je realizován voláním funkce *expressionParser(bool tokenRead, ast_ node_ exp_ t **resultPointer)* ve souboru `expression_parser.c`. Tato funkce je volána na místech, kde se očekává výraz, například za symbolem `=`. Parametr `tokenRead` indikuje, zda byl první token již přečten, a parametr `resultPointer` slouží jako ukazatel, kam je uložen výsledný strom výrazu.

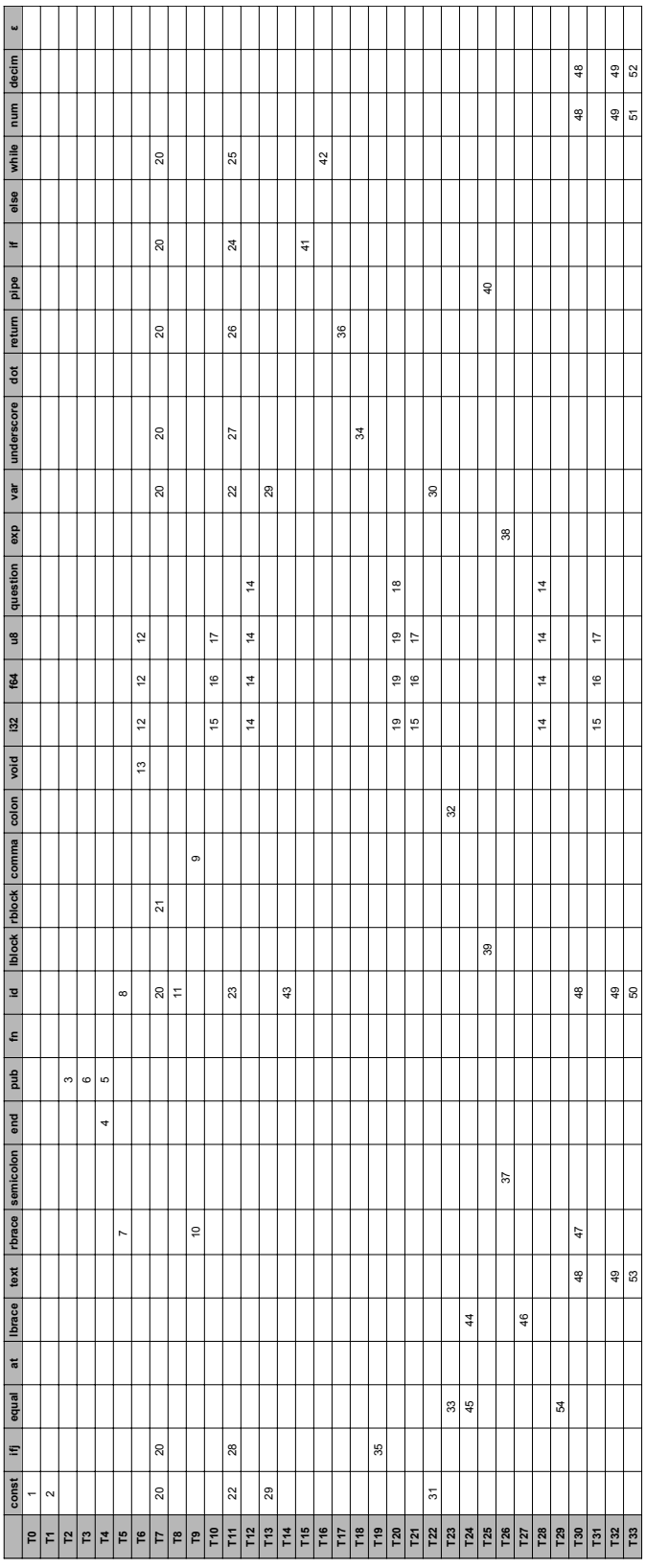
3.2 LL gramatika

1. Program → Import Function_list
2. Import → `const ifj equal at lbrace text rbrace semicolon`
3. Function_list → Function_analysis Function_next
4. Function_next → `end`
5. Function_next → Function_list
6. Function_analysis → `pub fn id lbrace` Param_def `rbrace` Return_type `lblock` Code_list `rblock`
7. Param_def → eps
8. Param_def → Param Param_list
9. Param_list → `comma` Param_def
10. Param_list → eps
11. Param → `id colon` Var_type
12. Return_type → Data_type
13. Return_type → `void`
14. Var_type → Data_type_optional Data_type
15. Data_type → `i32`
16. Data_type → `f64`
17. Data_type → `u8`
18. Data_type_optional → `question`
19. Data_type_optional → eps
20. Code_list → Code Code_list
21. Code_list → eps
22. Code → Variable_definition
23. Code → Call_or_assignment
24. Code → If_else
25. Code → While_syntax
26. Code → Return_syntax
27. Code → Empty_variable

28. Code → Library_function
29. Variable_definition → Rewrite_type `id` Variable_definition_type `equal exp`
30. Rewrite_type → `var`
31. Rewrite_type → `const`
32. Variable_definition_type → `colon` Var_type
33. Variable_definition_type → eps
34. Empty_variable → `underscore equal exp`
35. Library_function → `ifj dot` Function_call
36. Return_syntax → `return` Return_syntax_next
37. Return_syntax_next → `semicolon`
38. Return_syntax_next → `exp semicolon`
39. Unwrapped → eps
40. Unwrapped → `pipe id pipe`
41. If_else → `if lbrace exp rbrace` Unwrapped `lblock` Code_list `rblock else lblock` Code_list `rblock`
42. While_syntax → `while lbrace exp rbrace` Unwrapped `lblock` Code_list `rblock`
43. Call_or_assignment → `id` Call_or_assignment_next
44. Call_or_assignment_next → Function_call
45. Call_or_assignment_next → Assignment
46. Function_call → `lbrace` Parse_Params `rbrace semicolon`
47. Parse_Params → eps
48. Parse_Params → Term_list Parse_Params
49. Term_list → Term `comma`
50. Term → `id`
51. Term → `num`
52. Term → `decim`
53. Term → `text`
54. Assignment → `equal exp`

Poznámka: "exp"- označení pro výraz

3.3 LL tabulka



	const	ifj	equal	at	lbrace	text	rbrace	semicolon	end	pub	fn	id	lblock	rblock	comma	colon	void	i32	f64	u8	question	exp	var	underscore	dot	return	pipe	if	else	while	num	decim	ε
T0	1																																
T1	2									3																							
T2										6																							
T3									4	5																							
T4							7					8																					
T5																	13	12	12	12													
T6		20										20		21									20	20		20		20		20			
T7	20											11			9																		
T8							10											15	16	17			22	27		26		24		25			
T9																		14	14	14	14		29										
T10	22	28										23																41		42			
T11																32							30										
T12	29																	19	19	19	18			34		36							
T13												43						19	19	19		38					40						
T14		35											39					15	16	17													
T15								37																									
T16					46													14	14	14	14												
T17			54			48	47					48																			48	48	
T18						49						49						15	16	17											49	49	
T19						53						50																			51	52	
T20																																	
T21	31																																
T22			33																														
T23			45		44																												
T24																																	
T25																																	
T26																																	
T27																																	
T28																																	
T29			54																														
T30						48	47					48																			48	48	
T31																																	
T32						49						49						15	16	17											49	49	
T33						53						50																			51	52	

Obrázek 2: LL tabulka, část 1

3.4 Zpracování výrazů

Precedenční analýza začíná inicializací zásobníku, do kterého je nejprve vložen ukončující terminál. Následně jsou postupně čteny tokeny, přičemž funkce *getOperation()* vyhodnocuje operaci na základě vrcholu zásobníku a aktuálně čteného tokenu. V případě operace `<` je nad nejvrchnější terminál umístěna zarážka `<` a čtený token je vložen na vrchol zásobníku. Při operaci `=` je čtený terminál jednoduše vložen na vrchol zásobníku. Pokud nastane operace `>`, dochází k pokusu o redukci voláním funkce *tryToMatchRule(t_Stack *stack)*. Pokud se najde shoda s některým z pravidel gramatiky, probíhá redukce, která zároveň generuje abstraktní syntaktický strom.

Redukce je realizována pomocí funkcí:

- *createUnaryExp(t_Stack *stack)* pro vytvoření unárního výrazu,
- *createBinaryExp(t_Stack *stack)* pro vytvoření binárního výrazu,
- *createValueExp(t_Stack *stack)* pro vytvoření výrazu s hodnotou, nebo id,
- *reduceBracketNonTerminal(t_Stack *stack)* pro redukci v případě (E).

Pokud funkce `getOperation()` vrátí hodnotu `0`, znamená to chybu, a funkce `expressionParser(bool tokenRead, ast_node_exp_t **resultPointer)` skončí syntaktickou chybou. V případě, že výsledkem operace je `1`, provede se kontrola, zda zásobník odpovídá očekávanému stavu `$E$`. Pokud ano, je abstraktní syntaktický strom vložen do ukazatele `**resultPointer`.

3.5 Precedenční tabulka

	+	-	*	/	(	)	i	<	>	==	<=	>=	!=	;	x
+	>	>	<	<	<	>	<	>	>	>	>	>	>	>	<
-	<	>	<	<	<	>	<	>	>	>	>	>	>	>	<
*	>	>	>	>	<	>	<	>	>	>	>	>	>	>	<
/	>	>	<	>	<	>	<	>	>	>	>	>	>	>	<
(	<	<	<	<	<	=	<	<	<	<	<	<	<	0	<
)	>	>	>	>	0	>	0	>	>	>	>	>	>	>	0
i	>	>	>	>	0	>	0	>	>	>	>	>	>	>	0
<	<	<	<	<	<	>	<	0	0	0	0	0	0	>	<
>	<	<	<	<	<	>	<	0	0	0	0	0	0	>	<
==	<	<	<	<	<	>	<	0	0	0	0	0	0	>	<
<=	<	<	<	<	<	>	<	0	0	0	0	0	0	>	<
>=	<	<	<	<	<	>	<	0	0	0	0	0	0	>	<
!=	<	<	<	<	<	>	<	0	0	0	0	0	0	>	<
;	<	<	<	<	<	0	<	<	<	<	<	<	<	1	<
x	>	>	>	>	0	>	0	>	>	>	>	>	>	>	0

Obrázek 3: Precedenční tabulka

3.6 Gramatika pro výrazy

1. $E \rightarrow E + E$
2. $E \rightarrow E * E$
3. $E \rightarrow (E)$
4. $E \rightarrow id$
5. $E \rightarrow num$
6. $E \rightarrow float$
7. $E \rightarrow string$
8. $E \rightarrow E - E$
9. $E \rightarrow E/E$
10. $E \rightarrow E > E$
11. $E \rightarrow E < E$
12. $E \rightarrow E \neq E$
13. $E \rightarrow E \geq E$
14. $E \rightarrow E \leq E$
15. $E \rightarrow E == E$
16. $E \rightarrow -E$

4 Tabulka symbolů

Dle výběru varianty zadání TRP-izp je tabulka symbolů implementována tabulkou s rozptýlenými položkami s implicitním zřetězením položek. Ta je implementována v souboru *symtable.c (.h)*.
Na tabulce symbolů jsme začali pracovat dříve než byl zadán druhý projekt z předmětu IAL, a kvůli tomu jsme nemohli využít vzor z něj. Proto jsme rozhodli místo implementace z projektu IAL upravit a použít implementaci tabulky s rozptýlenými položkami z druhého projektu z předmětu IJC.

4.1 Návrh tabulky symbolů

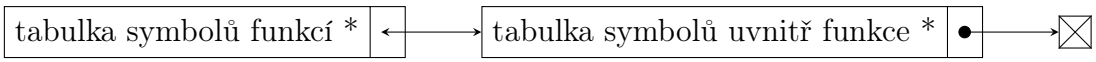
Prvek v tabulce symbolů obsahuje následující elementy:

- key - název symbolu
- depth - hloubka zanoření symbolu
- type - datový typ proměnné
- returnType - datový typ, který funkce vrací
- params - ukazatel na pole parametrů funkce
- paramCound - počet parametrů
- hasReturn - funkce má v těle return
- isConst - Symbol je typu const
- isNullable - Symbol může vracet NULL
- isDefined - Symbol byl definován
- isUsed - Symbol byl použit
- isMutable - Symbol může být měněn

4.2 Uložení tabulek symbolů do DLL seznamu

V *symtable.h* je kromě tabulky symbolů definován i seznam *htabs_l* , který obsahuje odkaz na první a poslední tabulku symbolů. Dvousměrně vázaný lineární seznam je pro odevzdávaný projekt redundantní, ale na začátku projektu jsme počítali s implementací generátoru kódu bez vytváření abstraktního syntaktického stromu, kde byl dvousměrně vázaný seznam vhodnější.

V odevzdávaném projektu existují zároveň nanejvýš dvě tabulky symbolů. První se vytváří na začátku syntaktické analýzy při analýze hlaviček funkcí a ukládají se do ní informace o funkcích.
Druhá tabulka se vytváří při vstupu do analýzy funkce a zaniká při ukončení této analýzy.



Obrázek 4: Diagram dvousměrně vázaného seznamu tabulek symbolů

5 Sémantická analýza

Sémantická analýza je prováděna zároveň se syntaktickou analýzou v *syntax_an.c*. Pomocné funkce pro kontrolu sémantiky jsou implementovány v *semantic_an.c* a definovány v *semantic_an.h*. Sémantická analýza vrací chybové kódy podle zadání.

6 Generování kódu

Generování kódu funguje na základě průchodu abstraktního syntaktického stromu. Při průchodu se pomocí funkce *printf()* vytváří spustitelný mezikód v instrukční sadě IFJcode24 na standardní výstup. Zdrojový kód pro generování kódu se nachází v souborech *codegen.c* a *codegen.h*. Generátor je tvořen ze dvou částí - tisknutí hlavičky a tisknutí mezikódu na základě vstupu.

6.1 Tisknutí hlavičky

Vytisknutí správné hlavičky zajišťuje funkce *printHeaders()*. Tato funkce nejdříve vytiskne povinné záhlaví pro mezikód - *.IFJcode24*. Poté vytiskne skok do funkce main, kterou musí každý vstupní program obsahovat. Následně vytiskne definici jedinné globální proměnné *GF@_%* a definice vestavěných funkcí jazyka IFJ24. Definice vestavěných funkcí byly nejdříve napsány a otestovány v pomocném souboru který nebyl odevzdán. Pro názvy proměnných ve vestavěných funkcích je využit znak '%' na konci, aby nemohlo dojít ke kolizi s proměnnými vytvářenými ve vstupním programu, jelikož znak '%' není v názvu proměnných povolen.

6.2 Generování na základě vstupu

Tato část generování závisí na průchodu abstraktním syntaktickým stromem vytvořeným v předchozích částech. První volanou funkcí je funkce *codebody()*, která prochází tělo(: funkce, while cyklu, if-else) a volá příslušné podpůrné funkce. Jelikož je vstupní program tvořen z funkcí (i main je funkce), začíná se generováním definic funkcí.

6.2.1 Začátek generování funkcí

Nejdříve se vytiskne nezměněné jméno funkce pomocí instrukce *LABEL*. Jako první se v každé funkci vytvoří rámec pomocí *CREATEFRAME*, posune se na zásobník pomocí *PUSHFRAME* a následně se nadefinují parametry a návratová hodnota. Poté se zavolá funkce *defAllVars()*, která projde tělo funkce do hloubky(= i do těl cyklů a if-else) a nadefinuje všechny použité proměnné, aby nemohlo dojít ke kolizi kvůli redefinicím. Následně se začne vyhodnocovat samotné tělo uložené v ASS.

6.2.2 Generování těla funkce

Tělo funkce prochází funkce *codebody()*. Pokud narazí na uzel přiřazení, např. *x = 2 + 3*, nejdříve se pravá strana převede do polské notace, tzn. *23+*, a za využití zásobníku je vyhodnocena. Výsledek je následně pomocí instrukce *POPS* uložen do proměnné x. Pokud *codebody()* narazí na uzel volání funkce, vloží se na zásobník argumenty funkce a poté se funkce zavolá pomocí instrukce *CALL*. Pokud *codebody()* narazí na while cyklus nebo if-else vytiskne se *LABEL WHILE%d*, resp. *LABEL IF%d*, kde %d značí číslo určené pomocí globální proměnné *labelCounter*, a to aby bylo možné odlišit vnořené cykly a if-else klauzule. Následně se vytiskne kontrola podmínky a podmíněný skok na *LABEL WHILEEND%d*, resp. *LABEL ELSE%d*, pokud nebude podmínka splněna. Následně se zavolá funkce *codebody()* nad tělem while cyklu, resp. if-u. Pokud se jednalo o while cyklus, vytiskne se po tělu cyklu skok na začátek cyklu a *LABEL WHILEEND%d*. Tímto končí while cyklus. Pokud se jednalo o if, vytiskne se skok za else na *LABEL IFEND%d* a začátek else *LABEL ELSE%d*. Pokud se jednalo o if klauzuli i s

else částí, zavolá se *codebody()* nad tělem else. Nakonec se vytiskne *LABEL IFEND%d*. Tímto končí if-else klauzule.

6.2.3 Konec funkce

Pokud *codebody()* narazí na return, uloží jeho výsledek do *LF@retval%*. Na konci funkce se vytiskne instrukce *POPFRAME* a *RETURN*, resp. *EXIT 0* pokud se jednalo o funkci main. Návratová hodnota se následně dá najít jako *TF@retval%*. Tento proces se provede pro každou funkci ve vstupním zdrojovém kódu.

7 GNU Make

Další požadavek projektu bylo vytvoření souboru *Makefile* pro přeložení projektu pomocí příkazu `make`. Pro kompilaci používáme kompilátor *gcc* s *gnu99* standardem jazyka C. Kompilátor je taky nastavený, aby přerušil překlad při jakékoliv chybě a optimalizace jsou vypnuté.
Nástroj GNU Make lze také využít k vytvoření zabaleného archivu pro odevzdání pomocí příkazu `make zip`.

8 Datové struktury

8.1 Abstraktní syntaktický strom (ASS)

Abstraktní syntaktický strom, zkráceně ASS, je struktura, kterou vytváříme během syntaktické analýzy a využíváme pro generaci kódu. Jedná se o systém navzájem propojených uzlů. Kořenový uzel je typu *ast_default_node* a obsahuje uzly definic funkcí *ast_node_fn_def* a jejich počet. Ty obsahují informaci o funkci s tělo definující funkci. Tělo funkce je pole *ast_default_node* ukazujících na uzly níže definované.

8.1.1 ast_node_fn_call

Uzel *ast_node_fn_call* znamená volání funkce. Uchovává informaci o názvu funkce a argumentech s kterými byla volána.

8.1.2 ast_node_fn_return

Uzel *ast_node_fn_return* znamená návrat z funkce. Uchovává informace o návratovém typu a návratové hodnotě uložené v *ast_node_exp*.

8.1.3 ast_node_var_assign

Uzel *ast_node_var_assign* udává přiřazení do proměnné. Uchovává název proměnné a hodnotu v *ast_node_exp*, která se do ní má vložit.

8.1.4 ast_node_var_def

Uzel *ast_node_var_def* udává deklaraci proměnné a uchovává název proměnné. Dále pok1ud došlo i k přiřazení do proměnné, uloží jej v *ast_node_var_assign*.

8.1.5 ast_node_if_else

Uzel *ast_node_if_else* udává if-else klauzuli. Uchovává informaci jestli se jedná o if akceptující hodnotu null a tělo a délku těla if i else částí.

8.1.6 ast_node_while

Uzel *ast_node_while* znamená while cyklus. Uchovává informaci jestli se jedná o if akceptující hodnotu null a tělo a délku těla while cyklu.

8.1.7 ast_node_exp

Uzel *ast_node_exp* ukládá binární strom vytvořený v expression parseru. Navíc má dva speciální ukazatele - v případě volání funkce a v případě znaménka mínus, např. -2.

8.2 Dynamické pole

Dynamické pole je použito při generaci kódu pro uchování již definovaných proměnných *array_vars_t* a pro řazení exp uzlů v post orderu *array_items_t*. Obsahuje pole ukazatelů na ukládané hodnoty, maximální kapacitu a počet momentálně uložených položek.

8.3 Zásobník

Zásobník je implementován jako propojený seznam. Obsahuje následující struktury:

- `t_Stack`: Struktura reprezentující zásobník.
  - `struct StackItem *top` - ukazatel na vrchol zásobníku.
  - `int size` - aktuální počet položek v zásobníku.
- `t_StackItem`: Struktura reprezentující jednu položku v zásobníku.
  - `stackElementType type` - typ položky (`TERMINAL`, `NON_TERMINAL`, `PRECEDENT_LESS`).
  - `Token *token` - ukazatel na token, který položka obsahuje.
  - `struct StackItem *next` - ukazatel na následující položku v zásobníku.
  - `struct StackItem *prev` - ukazatel na předchozí položku v zásobníku.
  - `ast_node_exp_t *node` - ukazatel na ASS uzel asociovaný s touto položkou.

Funkce pro práci se zásobníkem:

- *void stackInit(t_Stack *stack)*
  - Inicializuje prázdný zásobník.
- *int stackPush(t_Stack *stack, stackElementType type, Token *tkn, ast_node_exp_t *node)*
  - Přidá novou položku na vrchol zásobníku.
  - Vrací `0` při úspěchu nebo `INTERNAL_COMPILER_ERROR` při chybě alokace paměti.
- *void stackPop(t_Stack *stack)*
  - Odstraní položku z vrcholu zásobníku.
- *t_StackItem *stackTop(t_Stack *stack)*
  - Vrací ukazatel na položku na vrcholu zásobníku bez jejího odstranění.
- *void stackClear(t_Stack *stack)*
  - Odstraní všechny položky ze zásobníku.
- *int stackPushPrecedentLess(t_Stack *stack)*
  - Vloží položku typu `PRECEDENT_LESS` za nejbližší `TERMINAL` položku.
  - Vrací `0` při úspěchu, `INTERNAL_COMPILER_ERROR` při chybě alokace paměti nebo `SYNTACTIC_ANALYSIS_ERROR`, pokud `TERMINAL` neexistuje.
- `t_StackItem *topTerminal(t_Stack *stack)`
  - Vrací ukazatel na nejbližší položku typu `TERMINAL` v zásobníku, počínaje od vrcholu.

Tento zásobník je klíčovou součástí při implementaci syntaktické precedenční analýzy, kde slouží k ukládání a manipulaci s elementy jako jsou terminály, neterminály a speciální položky pro precedenční analýzu.

9 Rozšíření

Naše řešení částečně implementuje rozšíření FUNEXP.
Funkce mohou být volány ve výrazech. Nepodařilo se nám však zavést výrazy jako parametry funkcí, jelikož to syntaktická analýza nepovoluje.

10 Závěr

Projekt ze začátku působil velice rozsáhle, ale látku jsme vstřebávali pozvolna a postupně jsme stavěli naše řešení. Znalosti jsme čerpali především z přednášek IFJ a IAL. Nejasnosti ze zadání jsme doplňovali z diskuzního fóra a otázkami na ostatní studenty na Discordu. Jelikož se většina týmu již znala, neměli jsme problém spolupracovat. Správnost řešení jsme ověřovali pokusným odevzdáním a automatickými testy. Projekt jsme stihli vypracovat s jistým předstihem, což nám umožnilo vypracovat tuto dokumentaci. Z projektu jsme si odnesli zkušenosti nejen z programování, ale také z týmové práce a hospodaření s časem.