Dokumentace projektu

Implementace překladače imperativního jazyka IFJ21

Tým 125, varianta I

Do Hung	**(xdohun00)**	25%
Kedra David	(xkedra00)	25%
Kvasnička Jaroslav	(xkvasn14)	25%
Kolařík Petr	(xkolar79)	25%

Implementovaná rozšíření jazyka IFJ21: BOOLTHEN

Obsah

1 Úvod

Naším úkolem bylo vytvořit program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ21, jenž je zjednodušenou podmnožinou jazyka Teal a přeloží jej do cílového jazyka IFJcode21 (mezikód).

Program má být konzolová aplikace bez grafického uživatelského rozhraní, která načítá zdrojový kód ze standartního vstupu a výsledný mezikód generuje na standartní výstup.

2 Návrh a implementace

Projekt jsme rozvrhli do několika dílčích částí, které jsme postupně implementovali.

2.1 Lexikální analýza

Hlavní funkcí této části je funkce *get_token*, která postupně načítá znaky ze zdrojového souboru, podle kterých se námi vytvořený konečný stavový automat přepíná mezi jednotlivými stavy. Znaky jsou ukládány do struktury *Istring*, která se dynamicky zvětšuje se zvětšujícím se počtem znaků. Jakmile dosáhne konečného stavu, volá funkci *token_create,* která token vytvoří, přiřadí mu typ, případně atribut a vrátí volajícímu. U načítání escape sekvencí a komentrářů. Oba tyto příklady jsou implementovány podle daných konečných automatů. Lexikální analýza se skládá ze souborů scanner.c, scanner.h, token.c, token.h.

2.2 Syntaktická analýza

Toto je nejdůležitější část celého programu, podle které se celý překlad řídí. Řeší, jestli tokeny, které dostane z lexikální analýzy odpovídají očekávaným tokenům podle LL – tabulky, kterou jsme vygenerovali z LL – gramatiky. Samotná gramatika se skládá z termů a netermů. Jednotlivé netermy jsou definované jako funkce. V každé funkci se nachází *switch*, se větví na jednotlivé typy tokenů (generováno podle LL – tabulky). Pokud je načten token (z lexikálního analyzátoru) ve switch neočekáván, jedná se o syntaktickou chybu a funkce hází chybu 2. Též je kontrolována existence prologu *require*. Načítání výrazů je pořešené pomocí precedenční analýzy (zdola nahoru). Hlavním souborem syntaktické analýzy je *parser.h*, *parser.c*.

2.3 Sémantická analýza

Sémantická analýza se provádí ihned po syntaktické kontrole. Makra *INIT_SCOPE* a *DESTROY_SCOPE* vytváří rozsah platnosti proměnných pro kontroly s tabulkou symbolů. Při

volání proměnných se sémantický analyzátor dívá do tabulky souborů a hledá, zda již již daná proměnná existuje či nikoliv, a to vrací uživateli do parametru *node_return* typu *node_ptr*. Funkce jsou ukládány v globální TS, proměnné jsou ukládány v lokální TS. Sémantický analyzátor dále provádí typovou kontrolu u přirazování výrazů či návrat z volání funkcí. Sémantická analýza je řízena syntaktickým analyzátorem – všechny sémantické akce jsou prováděny v *parser.c* a *expression.c.* Soubory využívané sémantickým analyzátorem jsou: symtable a stack.

2.4 Generování kódu

Pokud všechny předchozí kroky proběhly úspěšně, může se generovat kód. Všechen kód je generován (opět) v syntaktickém analyzátorovi. Hlavní funkcí generování kódu je *gen_code*, která bere pět parametrů: fronta příkazů, typ instrukce, cílová deestinace, první operand a druhý operand. Pokud není fronta příkazu při volání funkce definována (hodnota NULL), je příkaz okamžitě vypsán do výsledného souboru. Pro udržení jednotlivých indexů pro pojemnování (návěští a proměnných) je použita speciální struktura *gen_info,* která se na začátku kazdé funkce vynuluje. Jména promenných jsou generována v tomto formátu:

[FRAME]@[FUNC_NAME]_[if]_[elseif]_[else]_[while]_[VAR_NAME]

Řetězce „if“, „elseif“, „else“ a „while“ se připisují, pokud jejich indexy jsou větší než jedna, a to ve tvaru např. „_if1_elseif2“. Soubory použité pro generování kódu byly: generator, expression (generate_code_nterm).

2.4.1 Generování funkce

Na začátku funkce ze zavolá *LABEL* a *PUSHFRAME* instrukce pro inicializaci rámce. Dále se nadefinují dočasné pomocné proměnné a parametry, ke kterým se přiřadí patřičné argumenty. Následuje definice návratových hodnot. Po vygenerování všech příkazů uvnitř fce se nakonec zavolá *POPFRAME* a *RETURN.*

2.4.2 Generování větvení a cyklení

Nejprve se vygenerují patřičný *LABEL*, poté se vygeneruje podmíněný výraz, podle kterého se (ne)provedou skoky. Výraz se poté vyhodnotí, zda je pravdivý nebo ne. Pokud je výraz nepravdivý, kráče program na návěští za koncem pravdivého kódu. Pokud je výraz pravdivý, vygenerují se příkazy uvnitř pravdivého bloku a na jeho konci se volá skok na konec větvení (pro if) nebo na začátek cyklu (pro while).

2.4.3 Generování výrazů

Výrazy se generují v souboru *expression.c.* Funkce *expression* v precedenčí analýze vytvoří abstraktní syntaktický strom pomocí struktury exp_nterm_t. Na ten se zavolá funkce

generate_code_nterm, která již kód vygeneruje. Výraz se generoval pomocí algoritmu PostOrder, kdy se nejprve vygeneroval kód pro levý operand, výsledek se dal do zásobníku, poté se vygeneroval pravý operand a výsledek se taky dal do zásobníku. Oba výsledky se poté vyjmuly a uložili se to dočasných proměnných. Provedla se typová kontrola a typová konverze, vložili se zpátky do zásobníku a provedla se na nich zásobníková instrukce dané operace. Výjimkou byly instrukce EQ a NEQ, kde se nevkládaly ihned do zásobníku, ale provedly se instrukce, výsledek se uložil do třetí dočasné proměnné a až ten se vložil na zásobník. Tyto operace se prováděli rekurzivně, tudíž nakonec byl na vrcholu zásobníku výsledek celého výrazu.

2.4.4 Generování mnohonásobného přiřazení

Nejprve se do *stack_var* uložily operandy (proměnné) na levé straně, poté se vygenerovaly výsledky výrazů na pravé straně a uchovávaly se v zásobníku. Nakonec se výrazy postupně odstraňovaly z vrcholu zásobníku a přiřazovaly se do proměnných uložených ve *stack_var*.

2.5 Precedenční analýza

Precedenční analýza využívá 2 soubory: exp_stack a expression. První soubor definuje všechny potřebné funkce a struktury potřebné pro rozjetí algoritmu analýza zdola nahoru. Ve druhém souboru je poté implementace tohoto algoritmu. V expression.c je vytvořena precedenční tabulka, která je uložena ve dvourozměrném poli. V ní jsou uložené 4 druhy akcí: R(educe), S(hift), E(qual) a U(ndefined). Jednotlivé akce jsou definováne v prezentaci o Syntaktické analýze zdola nahoru. Hlavním jádrem algoritmu zisk tokenu od lexikálního analyzátoru, porovnání s precedenční tabulkou a volání příslusné akce. Precedenční tabulka slouží pro syntaktickou analýzu a při redukci je jsou volány sémantické akce pro jejich kontrolu. Pokud na konci zbyde jediný *exp_nterm,* jde o úspěšnou kontrolu, v opačném případě je volána příslušná chybová hlášení. Speciální případem je, pokud poslední volaný token je identifikátor. V tomto případě se přepíná zpátky na analýzu shora dolů.

3 Použité techniky a algoritmy

3.1 Tabulka symbolů

Tabulka symbolů je implementována pomocí binárního vyhledávacího stromu. Jednotlivé proměnné/funkce se do ni vkládají pomocí příkazu *tree_insert*. Tabulka vyhledává podle knihovní funkce *strcmp*. Pokud je hodnota výrazu menší než 0, volá se vyhledávání na levý podstrom, pokud je hodnota větší než 0, hledá se v pravém podstromě. Pokud se uživatel snaží vložit proměnnou, která je již v tabulce, vrací ERR_SEM_DEF hodnotu (chybová hodnota 3).

Funkce *tree_search* vyhledává stejným způsobem, jako *tree_insert*, akorát vrací nalezený uzel v případě úspěchu, a NULL v případě neúspěchu.

Funkce *tree_destroy* odstraňuje uzly pomocí algoritmu PostOrder.

Vše je implemetováno ve souboru *symtable*.

3.2 Zásobník tabulky symbolů

V zásobníku jsou ukládány lokální proměnné. Pokaždé, kdy se vytvoří nový rozsah platnosti (scope), je vytvořena nová tabulka symbolů a ta je vložena do zásobníku (INIT_SCOPE). V ní se vyhledává tak, že se vyhledává od vrcholu zásobníku po dno zásobníku (jediná výjimka nastává u deklarace proměnné, kde se prohledává jenom na v TS, která je na vrcholu zásobníku). Ukončení rozsahu platnosti se vyjme TS z vrcholu zásobníku (a její vyčištění).

Funkce *stack_reset_index* vrátí ukazatel zpátky na vrchol zásobníku, *stack_dec_index* naopak posouvá index hlouběji.

Vše je naimplementováno v souboru *stack*.

3.3 Zásobník mnohonásobného přiřazení

Do *stack_var* se ukládají jednotlivá jména proměnných, do kterých se ukládájí výsledky výrazů na pravé straně. Využívá k tomu klasické příkazy: *stack_var_pop, stack_var_push*. Zásobník se nachází v souboru *symtable*.

3.4 Fronta příkazů

Fronta příkazů slouží k v případě, kdy je potřeba si uložit kolekci příkazů předtím, než je vypsat na výstup. Hlavním příkazem fronty je *queue_flush*, která vyprázdní frontu a vypíše všechny příkazy ve frontě. Fronta příkazů je využívána především v cyklech, kde bylo potřeba všechny deklarace proměnných vypsat před samotným cyklem, a ukládání volání funkcí v „globálním

scopu“, kde mezi jednotlivými příkazy mohly stále být definice funkcí. Fronta je implementována v souboru *generator*.

4 Práce v týmu

Díky tomu, že členem našeho týmu je student 3. ročníku, který nás předem varoval, že na projekt je třeba spoustu času a úsilí, jsme na projektu začali pracovat už koncem září. Vedoucí nám hned od začátku rozdělil práci do skupin po dvou až třech lidech a určil předběžné deadliny jednotlivých částí.

4.1 Komunikace

Ke vzájemné komunikaci jsme si zvolili aplikaci Discord, přes kterou probíhala komunikace jak textová, tak občasné hovory. Občasná komunikace mezi jednotlivými členy probíhala také prostřednictvím aplikace Messenger. Nechybělo ani osobní setkávání v prostorách školy, ale to bylo kvůli pandemické situaci a nemocí členů značně postiženo. Pro přehledné zobrazení dat odevzdání jsme používali aplikaci ClickUp, která zároveň umožňuje sdílení dokumentu. Ten jsme používali především pro zaznamenání chyb, na které jsme během programování a testování přišli, abychom je měli všechny na jednom místě a mohli je vyřešit.

4.2 Verzovací systém

Pro umožnění pracovat na více souborech najednou a pro průběžné ukládání verzí souborů jsme využili verzovací systém Git. Jako vzdálený repozitář jsme použili GitHub.

4.3 Rozdělení práce mezi členy

Rozdělení práce bylo provedeno podle našeho vedoucího, který znal naše schopnosti a věděl, jak to bude nejrozumnější.

Jméno	**Práce**
Do Hung	vedoucí týmu, syntaktická analýza, sémantická analýza, testování, správa repozitáře
Kedra David	error handling, testování, precedenční analýza, příprava struktur
Kvasnička Jaroslav	lexikální analýza, generování kódu
Kolařík Petr	syntaktická analýza, sémantická analýza, testování, dokumentace

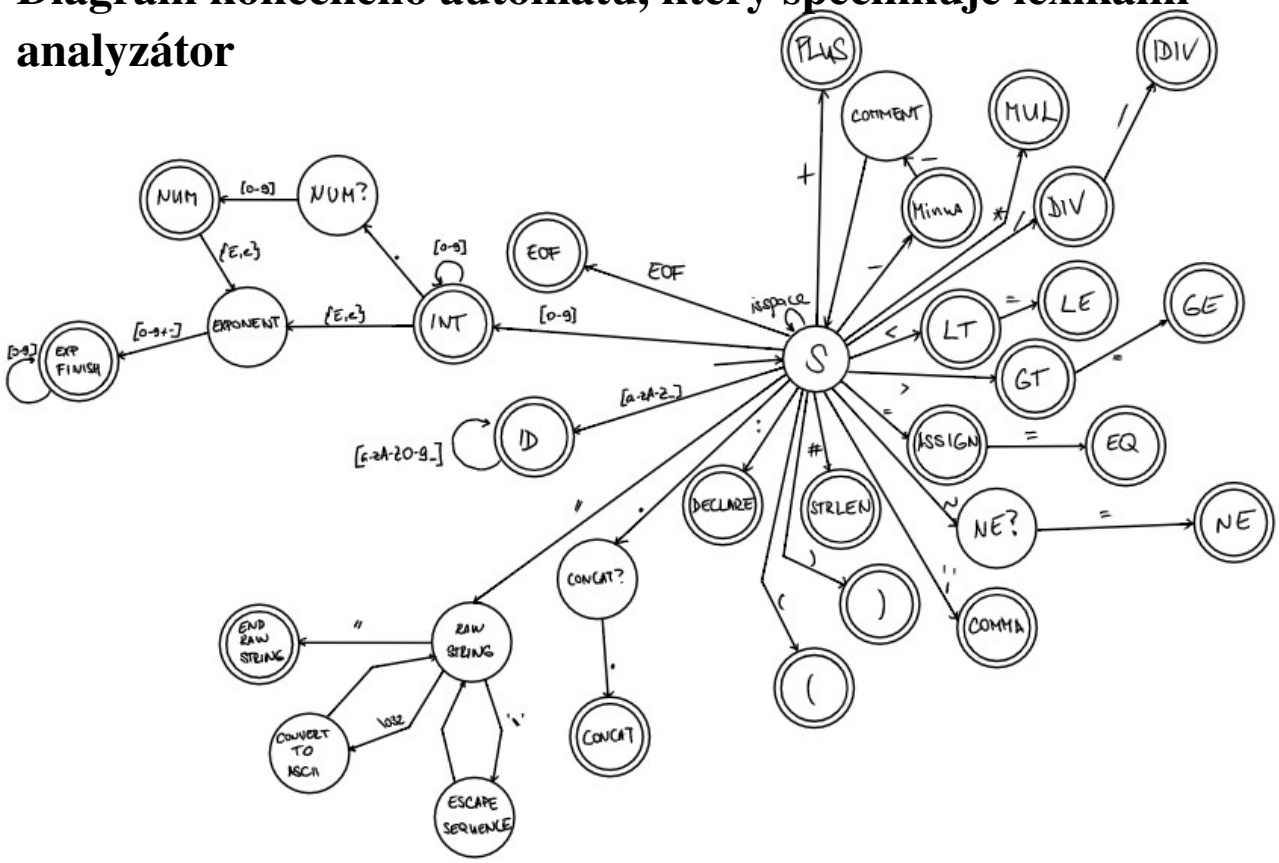
5 Závěr

I přesto, že jsme projekt začali řešit hned na začátku semestru, nás jeho rozsah zaskočil a dokončovali jsme ho na poslední chvíli. Velikou výhodou však bylo předčasné odevzdání, které nám ukázalo největší nedostatky projektu a ty jsme mohli do řádného data odevzdání v co největší míře odstranit.

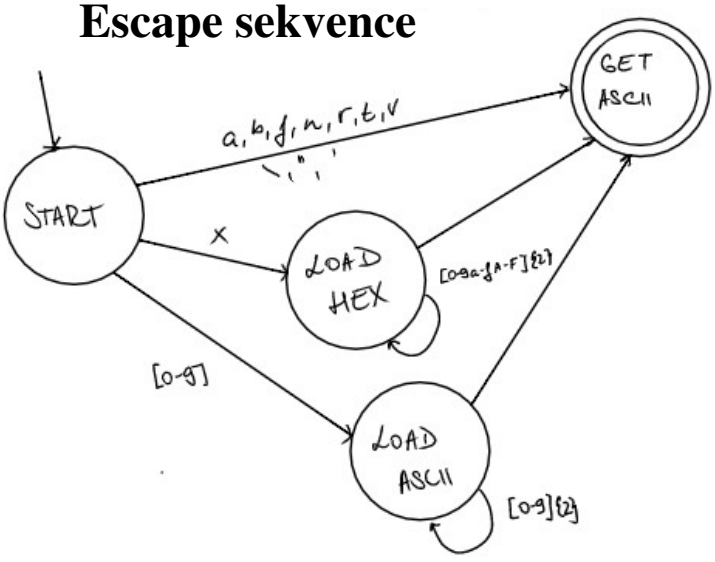
Práce v týmu byla téměř bezchybná. Po menších skupinkách jsme na jednotlivých částech projektu pracovali efektivně a pokud jsme se u něčeho zasekli, nebo si s něčím nevěděli rady, ostatní členové týmu byli vždy k dispozici.

Z projektu jsme si odnesli hodně zkušeností především co se týče práce v týmu. Zároveň jsme pořádně pochopili funkcionalitu překladače a zároveň zlepšili své programátorské dovednosti.

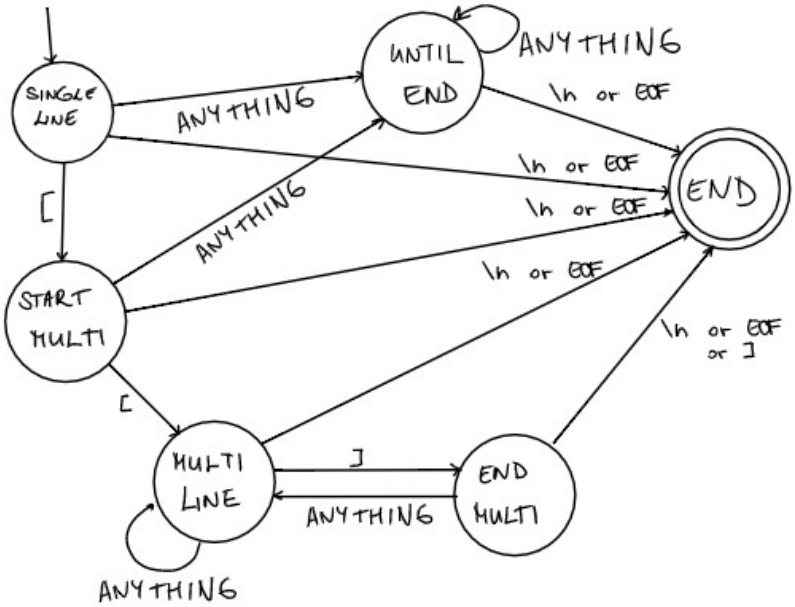
6 Diagram konečného automatu, který specifikuje lexikální analyzátor



Escape sekvence



Komentáře



7 LL – gramatika

```
1   PRG -> ε
2   PRG -> require _string_ PRG
3   PRG -> global id : function ( PARM ) RET PRG
4   PRG -> function id ( DEF_PARM ) RET CODE end PRG
5   PRG -> id ( LOF_E ) PRG
6   LOF_E -> ε
7   LOF_E -> E LOF_E_N
8   LOF_E_N -> ε
9   LOF_E_N -> , E LOF_E_N
10  PARM -> ε
11  PARM -> D PARM_N
12  PARM_N -> ε
13  PARM_N -> , D PARM_N
14  RET -> ε
15  RET -> : D RET_N
16  DEF_PARM -> ε
17  DEF_PARM -> id : D DEF_PARM_N
18  DEF_PARM_N -> ε
19  DEF_PARM_N -> , id : D DEF_PARM_N
20  CODE -> ε
21  CODE -> local id : D VAR_INIT_ASSIGN CODE
22  CODE -> if E then CODE ELSEIF_BLOCK ELSE_BLOCK end CODE
23  CODE -> while E then CODE end CODE
24  CODE -> return LOF_E
25  CODE -> id FUNC_OR_ASSIGN CODE
26  VAR_INIT_ASSIGN -> ε
27  VAR_INIT_ASSIGN -> = FUN_OR_EXP
28  FUN_OR_EXP -> id ( LOF_E )
29  FUN_OR_EXP -> E
30  ELSEIF_BLOCK -> ε
31  ELSEIF_BLOCK -> elseif E then CODE ELSEIF_BLOCK
32  ELSE_BLOCK -> ε
33  ELSE_BLOCK -> else CODE
34  FUNC_OR_ASSIGN -> ( LOF_E )
35  FUNC_OR_ASSIGN -> MUL_VAR_N = FUN_OR_MULTI_E
36  MUL_VAR_N -> ε
37  MUL_VAR_N -> , id MUL_VAR_N
38  FUN_OR_MULTI_E -> id ( LOF_E )
39  FUN_OR_MULTI_E -> E MUL_E_N
40  MUL_E_N -> ε
41  MUL_E_N -> , E MUL_E_N
42  E -> id
43  E -> (
44  E -> _string_
45  E -> _boolean_
46  E -> _integer_
47  E -> nil
48  E -> _number_
49  D -> string
50  D -> boolean
51  D -> integer
52  D -> number
53  E -> #
```

```
54  E -> not
55  RET_N -> ε
56  RET_N -> , D RET_N
57  D -> nil
```

	$	require	_string_	global	id	:	function	(	)	end	,	local	if	then	while	return	=	elseif	else	_boolean_	_integer_	nil	_number_	#	not	string	integer	boolean	number
PRG	1	2		3	5		4																						
PARM									10													11				11	11	11	11
RET	14	14		14	14	15	14			14		14	14		14	14													
DEF_PARM					17				16																				
CODE					25					20		21	22		23	24		20	20										
LOF_E			7		7			7	6	6								6	6	7	7	7	7	7	7				
E			44		42			43												45	46	47	48	49	50				
LOF_E_N									8	8	9							8	8										
D																						57				51	52	53	54
PARM_N									12		13																		
RET_N	55	55		55	55		55			55	56	55	55		55	55													
DEF_PARM_N									18		19																		
VAR_INIT_ASSIGN					26					26		26	26		26	26	27	26	26										
ELSEIF_BLOCK										30								31	30										
ELSE_BLOCK										32									33										
FUNC_OR_ASSIGN								34			35						35												
FUN_OR_EXP			29		29*			29												29	29	29	29	29	29				
MUL_VAR_N											37						36												
FUN_OR_MULTI_E			39		39*			39												39	39	39	39	39	39				
MUL_E_N					40					40	41	40	40		40	40		40	40										

8 LL – tabulka

`FUN_OR_EXP x id` – Může být použito i pravidlo 28. Záleží, zda je identifikátor funkce či proměnná, což se zjistí z tabulky symbolů.

`FUN_OR_MULTI_E x id` – Může být použito i pravidlo 38. Záleží, zda je identifikátor funkce či proměnná, což se zjistí z tabulky symbolů.

9 Precedenční tabulka

	(	)	^	#	not	* /	+ -	..	< > =	and	or	,	const	$
(	<	=	<	<	<	<	<	<	<	<	<	=	<	
)		>	>			>	>	>	>	>	>	>		>
^	<	>	<	<	<	>	>	>	>	>	>	>	<	>
#	<	>	<		<	>	>	>	>	>	>	>	<	>
not	<	>	<	<	<	>	>	>	>	>	>	>	<	>
* /	<	>	<	<	<	>	>	>	>	>	>	>	<	>
+ -	<	>	<	<	<	<	>	>	>	>	>	>	<	>
..	<	>	<	<	<	<	<	<	>	>	>	>	<	>
< > =	<	>	<	<	<	<	<	<	>	>	>	>	<	>
and	<	>	<	<	<	<	<	<	<	<	>	>	<	>
or	<	>	<	<	<	<	<	<	<	<	>	>	<	>
,	<	=	<	<	<	<	<	<	<	<	<	=	<	
const		>	>			>	>	>	>	>	>	>		>
$	<		<	<	<	<	<	<	<	<	<		<	