Projektová dokumentácia

k Implementace překladače imperativního jazyka IFJ22

Tým xpekny00, varianta TRP

Adam Pekný (vedúci tímu), xpekny00 - 25
Samuel Slávik, xslavi37 - 25
Jakub Kontrík, xkontr02 - 25
Michal Ľaš, xlasmi00 - 25

Práca v tíme

Rozdelenie práce

Adam Pekný (xpekny00):
Spracovanie výrazov, generovanie kódu, rekurzívna syntaktická analýza, sémantická analýza, LL gramatika, precedenčná tabuľka

Samuel Slávik (xslavi37):
Generovanie kódu, vstavané funkcie, zásobník, dynamické pole znakov, dokumentácia, LL gramatika

Jakub Kontrík (xkontr02):
Rekurzívna syntaktická analýza, sémantická analýza, generovanie kódu, obojstranne viazaný zoznam, LL gramatika

Michal Ľaš (xlasmi00):
Lexikálna analýza, konečný automat, dynamické pole znakov, tabuľka symbolov, LL gramatika

Verzovanie kódu a komunikačné kanály:

Pre verzovanie kódu sme používali už tradične nástroj git s privátnym repozitárom na Githube. Pri práci sme však často využívali aj rozšírenie VSCode o plugin LiveShare, ktorý umožňuje prácu viacerých developerov naraz na jednom projekte.
Na komunikáciu bola najviac používaná aplikácia Discord, avšak do karát nám hral aj fakt že, sa spolu všetci dobre poznáme a často sa stretávame aj fyzicky

Implementácia

Lexikálna analýza

Náš lexikálny analyzátor sme začali tvoriť podľa navrhnutého konečného automatu. Hlavnú časť našej lexikálnej analýzy tvorí funkcia *get_token()*, ktorá berie ako argument číslo, ktoré určuje stav automatu. Funkcia načíta znaky zo štandardného vstupu a vracia postupnosť tokenov. V prípade načítania nevalidnej postupnosti znakov funkcia vracia typ tokenu T_ERROR. Automat má 3 stavy. V stave 0 funkcia *get_token()* načítava prológ jazyka IFJ22, v stave 2 načítava vstup za epilógom a v

stave 1 funkcia načítava všetky ostatné tokeny. Načítanie prológu a epilógu je implementované v dvoch samostatných pomocných funkciách *automat_state_prolog()* a *automat_state_epilog()*.
Token reprezentujeme ako štruktúru, ktorá obsahuje typ tokenu, riadok na ktorom sa token nachádza a dáta, reprezentované ako typ union, ktoré obsahujú jeden z typov long, double alebo string. String je uložený ako typ dynamický buffer.
Hlavná časť funkcie *get_token()* pozostáva z implementácie switch prepínača, ktorý bol vytvorený z nášho konečného automatu pre lexikálnu analýzu, na začiatku tejto funkcie sa volá pomocná funkcia change_state(), ktorá určí prvý stav automatu podľa prvého načítaného znaku.
V našej implementácii sme vytvorili pomocné funkcie, ktoré nám pomáhajú spracovávať vstup v jednotlivých stavoch:

- Funkcia *load_string()* - slúži na načítavanie reťazcov
- Funkcia *load_var_id()* - slúži na načítanie názvov premenných
- Funkcia *load_type_id()* - slúži na načítanie typov, ?int, ?float, ?string
- Funkcia *load_num()* - slúži na načítanie čísel všetkých typov
- Funkcia load_letter() - slúži na načítanie názvov funkcii alebo kľúčových slov ako while, if, else, function, return,...

Ďalšie pomocné funkcie sú:

- Funkcia *string_to_num()* - slúži na prevedenie reťazca na číslo
- Funkcia *is_reserved_id()* - slúži na overenie či načítaný reťazec znakov je kľúčové slovo

Náš lexikálny analyzátor vie presne určiť typ tokenu vrátane odlíšenia názvov premenných, názvov funkcií alebo rozpoznania kľúčových slov.

Syntaktická analýza

Syntaktická analýza je implementovaná metodou rekurzívného zostupu zhora nadol. Funkcie sú implementované pomocou LL gramatiky. Pre každý neterminál existuje funkcia, v ktorej sú implementované dané pravidla. Využívaná je funkcia *get_token()*, ktorá pošle ďalší token, keď je potrebné. Ak nastane, že nejaké gramatické pravidlo nie je splnené, využívame makro ASSERT_ERROR, ktoré skontroluje podmienku a prípadne zavolá funkciu *error_exit()*, ktorá ukončí program s navrátovym typom chyby a vypíše chybu na chybový výstup spolu s riadkom kde sa daná syntaktická chyba nachádza. Na spracovanie výrazov voláme funkciu *parse_expression()*, ktorej je daný očakávaný token, ktorý ukončuje výraz a poprípade token, ktorý je načítaný navyše pri priraďovaní.

Statická sémantická analýza

Keďže ifj22 je dynamicky typovateľný jazyk, staticky kontrolujeme základné veci. A to pomocou globálnej tabuľky symbolov. Tabuľka symbolov je implementovaná ako

tabuľka s rozptýlenými položkami. Do ktorej vkladáme definované funkcie a každá funkcia má vlastnú tabuľku symbolov pre lokálne premenné. Kontrola prebieha počas rekurzivneho zostupu. Pri volaní sa kontroluje či sa daná funkcia alebo premenná nachádza v tabuľke. Staticky kontrolujeme aj počet argumentov funckie.

Generovanie kódu

Pre generovani kódu využívame obojstranne viazaný zoznam, ktorý uchováva všetky inštrukcie, ktoré sa pripisujú počas rekurzivneho zostupu. Používame zásobníkový kód. Pre správne generovanie kódu máme v zozname pridané ukazovateľe ako aktívny prvok, main prvok a while alebo if inštrukcia, pred ktorú sa vkladajú inštrukcie na definovanie premenných aby nevznikla chyba. Unikátnosť náveští zabezpečujeme pomocou funkcie *label_name_gen()*, ktorá priradzuje k zadanému menu náveštia unikátne číslo. Pri generovaní kódu sa používa makro DETECT_MAIN, s pomocou ktorého sa vyhodnotí či inštrukcia patrí do hlavného tela programu alebo do funkcie. Po celej analýze sa prejde zoznam a každý prvok vypíše na štandardný výstup.

Vstavané funkcie

Vstavané funkcie a ich náveštia generujeme pri spustení programu, kedy ich aj manuálne pridávame do tabuľky symbolov a určujeme typ argumentov a ich počet. Typy netreba kontrolovať pri funkciách, kde je zadaný argument term.

Dynamická sémantická analýza

Počas generovania kódu, využívame stack a inštrukcie ifjcode22 s ktorými kontrolujeme typy premenných aby sme zaručili korektné volanie funkcií a návratové typy funkcií. Argumenty funkcií sú načítané v tabuľke symbolov a pri volaní funkcie sa kontroluje, či typ zodpovedá definovánému argumentu.

Spracovanie výrazov

Spracovanie výrazov sme implementovali pomocou precedenčnej tabuľky a zásobníkového automatu. Hlavnou funkciou nášho spracovania výrazov je funkcia *parse_expression()*, ktorá dostáva ako parametre začiatočný token výrazu, token s typom ktorý ukončuje výraz, obojsmerne viazaný zoznam pre generované inštrukcie IFJcode22 a prípadne token, ktorý bolo potrebné načítať navyše pre rozhodnutie či sa jedná o priradenie do premennej alebo len výraz, pri nevyužití tohto parametru v ňom očakáva hodnotu NULL. Táto funkcia využíva funkciu z lexikálnej analýzy *get_token()*, pomocou ktorej získava postupnosť tokenov výrazu. Na začiatku si funkcia inicializuje zásobník pre spracovanie výrazu a vloží naň terminál s tokenom typu ukončujúceho tokenu. Následne začne spracovávať postupne tokeny až kým nie je zásobník v koncovom stave a na vstupe je správny ukončujúci token. Ukončujúcim stavom rozumieme stav kedy sa na zásobníku nachádzajú len dva

prvky, na vrchole výrazový prvok a pod ním terminál s tokenom typu ukončujúceho tokenu.
Spracovanie jednotlivých tokenov prebieha nasledovne:

1. Overí sa valídnosť vstupného tokenu, v prípade premennej jej definícia
2. Prekonvertuje sa typ tokenu na typ prvku výrazu a následne sa pomocou typu najvrchnejšieho terminálu na zásobníku a prekonvertovaného typu tokenu vyhodnotí index v precedenčnej tabuľke
3. Z precedenčnej tabuľky sa získa informácia o tom aká operácia sa má vykonať:
    a. Vložiť nový prvok vytvorený na základe vstupného tokenu na zásobník a načítať nový token
    b. Označiť že za najvrchnejším terminálom na zásobníku je potrebná redukcia, vložiť nový prvok vytvorený na základe vstupného tokenu na zásobník a načítať nový token
    c. Vygenerovať kód pre časť výrazu na zásobníku, ktorý je pripravený na redukciu pomocou funkcií z modulu generátor, a následne túto časť na zásobníku zredukovať pomocou syntaktických pravidiel pre výrazy

V prípade že sa nepodarí zredukovať výraz, typ vstupného tokenu sa vyhodnotí ako neznámy z pohľadu výrazu, alebo token neprejde jednou zo sémantických kontrol, je z funkcie vrátená hodnota false alebo priamo sa program ukončuje s chybovou návratovou hodnotou a chybovou hláškou, ktorá obsahuje popis chyby a číslo riadku na ktorom bola detekovaná. Špeciálne pre token typu pravej oblej zátvorky sa v prípade keď je ukončujúcim tokenom overuje pomocou úrovne zanorenia v dvojiciach zátvoriek a pomocnej funkcie *is_expr_end_token()* či sa skutočne jedná o ukončujúci token alebo obyčajnú zátvorku vo výraze.

Dátové štruktúry

Obojstranne viazaný zoznam

Štruktúra, do ktorej vkladáme inštrukcie alebo skupiny inštrukcií na generovanie kódu. Obsahuje ukazovatele na prvý prvok, posledný prvok, aktívny prvok, prvok hlavného tela programu, prvok na if alebo while. Aj pomocné ukazovatele, ako ukazovateľ na terajšiu funkciu, funkciu v ktorej je inštrukcia spracovaná, terajšiu premennú, ukazatel na meno vonkajšieho náveštia if alebo while, stack s volanými argumentmi funkcie a počet volaných argumentov.

Dynamické pole znakov

Pole, ktorá sa zväčšuje ak sa ma presiahnuť alokovaná veľkosť tejto štruktúry. Veľkosť sa zväčšuje vždy na dvojnásobok aktuálne alokovaného priestoru. Túto štruktúru používame v lexikálnej analýze na postupné načítanie znakov zo vstupu a na ukladanie generovaného kódu. Implementácia obsahuje pomocné funkcie na

inicializáciu, pridanie jedného znaku do poľa, pridanie reťazca do poľa a uvoľnenie alokovaných zdrojov.

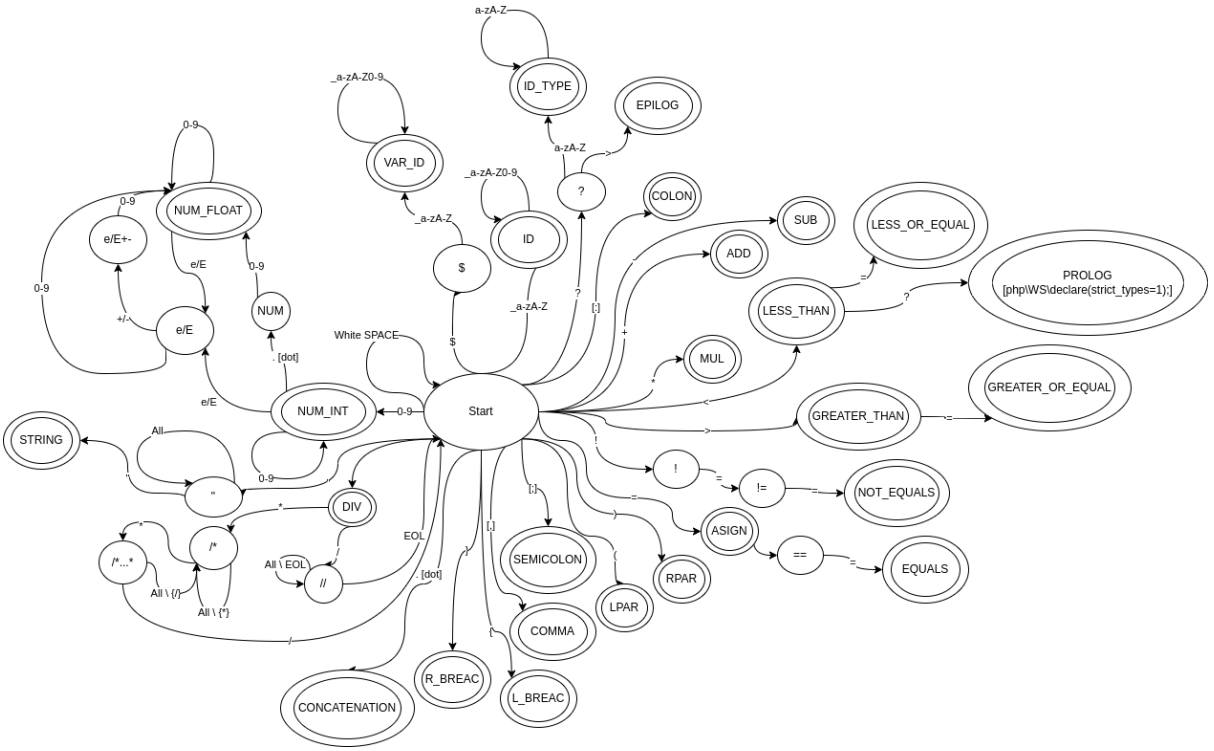
Zásobník

Ako ďalšiu pomocnú dátovú štruktúru sme implementovali zásobník. V zásobníku uchovávame typ ukazovatele na void. Naša implementácia obsahuje štandardné operácie so zásobníkom ako Init, Push, Pop, Top a Empty.

Tabuľka s rozptýlenými položkami

Pre implementáciu našej tabuľky symbolov sme si vybrali variantu tabuľky s rozptýlenými položkami. V tabuľke uchovávame informácie o premenných a funkciách. Pri premenných zaznamenávame názov premennej a či bola premenná inicializovaná. Pri funkciách ukladáme názov funkcie, počet parametrov, parametre a ich typy, či má funkcia návratovú hodnotu, typ návratovej hodnoty, ukazovatele na lokálnu tabuľku symbolov danej funkcie a názov náveštia funkcie. Implementácia našej tabuľky symbolov obsahuje aj funkciu na zmenu veľkosti tabuľky, ktorá umožňuje efektívnejší prístup k prvkom tabuľky pri veľkom množstve uchovávaných dát. Rozptyľovaciu funkcie sme prevzali zo stránky predmetu IJC (Programovanie v jazyku C) [http://www.fit.vutbr.cz/study/courses/IJC/public/DU2.html].
Nad našou tabuľkou s rozptýlenými položkami sme implementovali aj funkcie, ktoré nám umožňujú efektívnu prácu s touto tabuľkou. Napríklad funkciu *st_fun_create()*, ktorá vytvorí a inicializuje dáta o funkcií, *st_vat_create(),* ktorá vytvorí a inicializuje dáta o premennej, *st_fun_table_create()*, ktorá vytvorí tabuľku symbolov pre konkrétnu funkciu a ďalšie pomocné funkcie.

Diagram konečného automatu



LL gramatika

<start> => [T_PROLOG] <prog>
<prog> => <body> <prog>
<prog> => <fn> <prog>
<prog> => [T_EOF]
<prog> => [T_EPILOG] [T_EOF]
<fn> => [T_FUNCTION] [FUN_ID] [T_L_PAR] <fn_param> [T_COLON] <fn_type> <fn_dedf>
<fn_param> => [T_R_PAR]
<fn_param> => [T_STRING_TYPE, T_FLOAT_TYPE, T_INT_TYPE] <fn_dedf_param_type>
<fn_dedf_param_type> => [T_VAR_ID] <fn_dedf_param_var>
<fn_dedf_param_var> => [T_COMMA] <fn_dedf_param>
<fn_dedf_param_var> => [T_R_PAR]
<fn_dedf_param> => [T_STRING_TYPE, T_FLOAT_TYPE, T_INT_TYPE] <fn_dedf_param_type>
<fn_dedf> => [T_L_BRAC] <in_body>
<in_body> => <body> <in_body>
<in_body> => [T_R_BRAC]
<fn_type> => [T_VOID, T_STRING_TYPE, T_FLOAT_TYPE, T_INT_TYPE]
<fn_call_l> => [T_STRING, T_NUM_INT, T_NUM_FLOAT, T_VAR_ID] <fn_call_lc>
<fn_call_l> => [T_R_PAR]
<fn_call_lc> => [T_COMMA] <fn_call_lparam>
<fn_call_lc> => [T_R_PAR]
<fn_call_lparam> => [T_STRING, T_NUM_INT, T_NUM_FLOAT, T_VAR_ID] <fn_call_lc>
<body> => [T_VAR_ID] [T_ASSIGN] <body_var>
<body_var> => <expr> [T_SEMICOLON]
<body_var> => [T_FUN_ID] [T_L_PAR] <fn_call_l> [T_SEMICOLON]
<body> => <expr> [T_SEMICOLON]
<body> => [T_FUN_ID] [T_L_PAR] <fn_call_l> [T_SEMICOLON]
<body> => [T_RETURN] <body_ret>
<body_ret> => [<expr>] [T_SEMICOLON]
<body_ret> => [T_SEMICOLON]
<body> => [T_IF] [T_L_PAR] <expr> [T_R_PAR] [T_L_BRAC] <in_body> [T_ELSE] [T_L_BRAC] <in_body>
<body> => [T_WHILE] [T_L_PAR] <expr> [T_R_PAR] [T_L_BRAC] <in_body>

LL tabuľka

	a	b	c	d	e	f	g	h	i	j	k	l	m
A	A → a B												
B		B → b	B → c b	B → C B								B → M B	
C				C → d e f D g I G									
D								D → h	D → i E	D → j E	D → k E		
E												E → l F	
F								F → h					F → m P
P									P → i E	P → j E	P → k E		
G													
H												H → M H	
I									I → i	I → j	I → k		
J								J → h				J → l K	
K								K → h					K → m L
L												L → l K	
M												M → l t N	
N													
O													

	o	p	q	r	s	t	v	u	w	x	y	z	ž
A													
B							B → M B		B → M B	B → M B	B → M B		B → M B
C													
D													
E													
F													
P													
G													
H	H → o						H → M H		H → M H	H → M H	H → M H		H → M H
I		I → p											
J			J → q K	J → r K	J → s K								
K													
L			L → q K	L → r K	L → s K								
M							M → v u		M → w ť J u	M → x O	M → y ť v h n H z n H		M → ž ť v h n H
N							N → v u		N → w ť J u				
O							O → v u	O → u	O → w ť J u				

Legenda k LL tabuľke:

A	<start>
B	<prog>
C	<fn>
D	<fn_param>
E	<fn_dedf_param_type>
F	<fn_dedf_param_var>
G	<fn_dedf>
H	<in_body>
I	<fn_type>
J	<fn_call_l>

K	<fn_call_lc>
L	<fn_call_lparam>
M	<body>
N	<body_var>
O	<body_ret>
P	<fn_dedf_param>

a	[T_PROLOG]
b	[T_EOF]
c	[T_EPILOG]
d	[T_FUNCTION]
e	[FUN_ID]
f	[T_L_PAR]
g	[T_COLON]
h	[T_R_PAR]
i	[T_STRING_TYPE]
j	[T_FLOAT_TYPE]
k	[T_INT_TYPE]
l	[T_VAR_ID]
m	[T_COMMA]
n	[T_L_BRAC]
o	[T_R_BRAC]
p	[T_VOID]
q	[T_STRING]
r	[T_NUM_INT]
s	[T_NUM_FLOAT]
t	[T_ASSIGN]
u	[T_SEMICOLON]
v	<expr>
w	[T_FUN_ID]
x	[T_RETURN]
y	[T_IF]
z	[T_ELSE]
ž	[T_WHILE]

Precedenčná tabuľka

stack\expr	+	-	*	/	.	<	>	<=	>=	===	!==	(	)	i	$
+	>	>	<	<	>	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
.	>	>	<	<	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
===	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>
!==	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>	>		>		>
i	>	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<	<		<	