FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Projektová dokumentace do předmětů IFJ a IAL

Tým 031, varianta I

Seznam autorů:

Šulavík Jan	(xsulav01)	32% - vedoucí
Černík Tomáš	(xcerni14)	25%
Polonec Michal	(xpolon02)	25%
Zedník Matěj	(xzedni15)	18%

Rozšíření: FUNEXP

Obsah

Úvod

Naším cílem bylo implementovat v jazyce C překladač imperativního jazyka ifj20. Tento program má načíst zdrojový kód v jazyce ifj20, který je podmnožinou jazyka Go a přeložit jej do mezikódu, který je generován na standardní výstup. Případně program vrací příslušný chybový kód. Vybrali jsme si variantu I, ve které je tabulka symbolů tvořena binárním stromem.

Práce v týmu

Rozdělení práce mezi členy týmu

Projekt jsme si rozdělili na následující části: lexikální analýza, syntaktická analýza, sémantická analýza, precedenční syntaktická analýza, generace kódu a tabulku symbolů.

Jan Šulavík pracoval na lexikální analýze, sémantické analýze, testování a podílel se také na syntaktické analýze a tabulce symbolů.

Černík Tomáš pracoval na precedenční analýze výrazů a podílel se i na dokumentaci a testování.

Polonec Michal pracoval na tvorbe LL-gramatiky/tabuľky, syntaktickej analýze a podílel se i na testovani.

Zedník Matěj pracoval na generování kódu a podílel se na tabulce symbolů, kterou předělal do finální podoby Jan Šulavík. Proto je zde odchylka od rovnoměrného rozdělení bodů.

Komunikace

Většina komunikace probíhala přes program Discord většinou textově, ale konalo se i několik videokonferencí, kdy jsme se vždy po dokončení určité části práce poradili, jak budeme postupovat dále. Discord nám umožnil efektivní a rychlou komunikaci, a mohli jsme se bez problémů radit o projektu i když ne všichni byli zrovna za počítačem, což zcela jistě zabránilo mnoha nedorozuměním. Osobní setkání nebylo možné z důvodu epidemiologické situace.

Verzování

Pro spravování projektu a jeho verzí jsme používali Git a vzdálený repozitář GitHub, který nám umožňoval pracovat na více částech najednou a následně po kontrole tyto změny spojit v jeden celek.

Rozšíření

FUNEXP

Toto rozšíření je implementováno pouze částečně. Program podporuje výrazy jako parametry při volání funkce.

Lexikální analýza

Jádrem lexikální analýzy je funkce *get_token*, která načítá znaky ze standardního vstupu a převádí je na tokeny, které jsou reprezentovány strukturou *Token*. Struktura obsahuje dva prvky - atribut a typ. V atributu, který je typu MyString je uložena jeho hodnota a v proměnné type je uložen typ tokenu. Typy tokenů odpovídají jednotlivým termům, operátorům, znakům a klíčovým slovům, které jsou povolené v jazyce ifj20. Takto vytvořený token je dále zpracováván v parseru.

Lexikální analyzátor je implementován jako deterministický konečný automat, který je popsán diagramem. V hlavním cyklu switch, který přepíná jednotlivé stavy automatu se podle aktuálně načteného znaku rozhoduje, na jaký stav přejít a jaké akce je potřeba vykonat. Pokud je zrovna načtený znak povolený, je uložen do atributu tokenu. Toto představovalo jistou výzvu, protože v jazyce ifj20 není stanovena maximální délka termu. Bylo proto třeba implementovat dynamický řetězec, jehož implementace je popsána dále. Pokud není načtený znak platný, tak automat vrací chybový typ tokenu, který když parser přijme, tak přestane vykonávat další akce a program se přeruší s návratovou chybou 1. Při implementaci často používám funkci *ungetc*, aby se automat mohl podívat na další znak a podle něj rozhodnout, jak se zachovat, to ušetřilo tvorbu některých dalších stavů.

Syntaktická analýza

Prvou, asi najdôležitejšou časťou bolo vytvoriť gramatiku jazyka. Pri jej tvorení som narazil na niekoľko problémov pri eps-pravidlách a taktiež pri počiatočnom neterminály POCIATOCNY_STAV pretože každý program *ifj20* musí obsahovať *package main* avšak iba raz a na začiatku programu. Je to zabezpečené prechodom do neterminálu ZACIATOCNY_STAV po precitani tokenov *package* a *main*. Eps-pravidlá mi tvorili neželané nedeterminizmy pri generovaní LL tabuľky. Postupne som ich odstraňoval, avšak nakoniec som jeden z nich zanechal kvôli lepšej znovupoužiteľnosti pri pravidle *PRAVA_STRANA_2* ktoré som potreboval ukončovať pri ich použití v cykle *for* a pri použití ako súčasti viacnásobného priradenia inými znakmi čo je zabezpečené definovaním týchto ukončovacích znakov v pravidlách ako terminálov *eol* a *{*. Po vygenerovaní LL- tabuľky mi vzniklo niekoľko nedeterminizmov, pri získaní tokenu *id* v pravidle *STATEMENT* a taktiež pri načítaní *tokenu (* v pravidle *ARGUMENTY_NAVRATOVE.* Tento problém bol prezentovaný spolu s ďalšími na demo cvičení z roku 2011 a riešil som ho uložením aktuálneho *tokenu* a načítaním daľšieho *tokenu,* aby som vedel rozhodnúť, ktorú verziu pravidla budeme uplatňovať. Následne som metódou rekurzívneho zostupu implementoval syntaktickú analýzu. Na často sa opakujúce žiadosti *scannera* o nový token a kontrolu typov som používal pomocné funkcie.

Sémantická analýza

Při sémantické analýze se využívá struktury typu *ParserData*, která obsahuje všechny potřebné informace o aktuálním stavu parseru. Je také využíváno globální proměnné symtable představující globální tabulku symbolů. Jelikož v jazyce ifj20 nejsou povoleny globální proměnné, jsou v ní uloženy definované funkce a speciální proměnná _. Při implementaci jsem musel řešit řadu problémů, které byly dány jazykem ifj20. Největší výzvou byla implementace kontroly platnosti proměnných v jednotlivých blocích. Nakonec jsem k řešení využil stack, který v sobě uchovává lokální tabulky symbolů. Na stack se s každým novým blokem vloží nová tabulka symbolů a do ní zkopíruje obsah tabulky z nadřazeného bloku. Když program z bloku vystoupí, je použita na stack funkce pop, která tabulku symbolů zahodí a na vrcholu je opět původní tabulka.

Dalším problémem byla implementace podpory volání funkce ještě před její definicí. To je nakonec řešeno tak, že každý záznam v tabulce symbolů má flag *declared*, který je pro proměnné vždy nastaven na jedna, a u funkcí se nastavuje podle toho, jak byly použity. Když se id funkce ukládá do tabulky symbolů z řádné definice, je flag nastaven na hodnotu 1 a funkce může být v programu použita. Pokud jde o volání funkce, která ještě není definovaná, tak se podle typu a počtu návratových hodnot a parametrů funkce před definuje, ale flag se nastaví na 0. Pokud je poté funkce později řádně definovaná, flag se nastaví na 1. Pokud parsování proběhne jinak v pořádku, tak po ukončení funkce parse je prohledána tabulka symbolů a zkoumá se nastavení flagu u každé položky v

tabulce. Pokud nějaká obsahuje flag na hodnotě 0, funkce byla použita ale nebyla definovaná a vrací se chyba.

Při sémantických kontrolách se využívá množství flagů, které pomáhají rozhodovat, jak se má parser zrovna zachovat.

Zpracování výrazů

Výrazy jsou zpracovávány precedenční syntaktickou analýzou, která je zpracována v souboru *expression.c*, ke které patří knihovna *expression.h*. Hlavní funkce se nazývá expression, ze které se následně volají její obslužné funkce. Tato hlavní funkce je volána právě tehdy, když se má zpracovávat výraz. Výrazy jsou zpracovávány token po tokenu, které se postupně ukládají v podobě symbolů do struktury *Stack* a dále jsou zpracovávány pomocí precedenční tabulky, která je uvedena ve vlastní kapitole. Pomocí této tabulky je zjištěno jaká činnost bude v souvislosti s nově obdrženým tokenem vykonána. Tato funkce dále obsahuje část pro kontrolu počtu závorek a to i v případě že se jedná o výraz v podmínce či for cyklu. V odpovídajících situacích jsou volány funkce pro generaci kódu. Funkce *expression* vrací v případě úspěchu hodnotu nula a v případě chyby její konkrétní kód.

Generování kódu

Kód se generuje pomocí souboru code_gen.c, ke kterému patří hlavičkový soubor code_gen.h. Generování je rozděleno do několika podsekcí. Mezi ně patří například podsekce pro funkce, podmínky, cykly a další. Nejdříve bylo zamýšleno udělat zapisování kódu do dočasného souboru tmpfile. To ale není v souladu se zadáním a je použitý dynamický string a pro pár funkcí i pole charů. Bohužel se generování výrazu časově protáhlo a to zapříčinilo problémy s jeho integrací.

Jiné části implementace

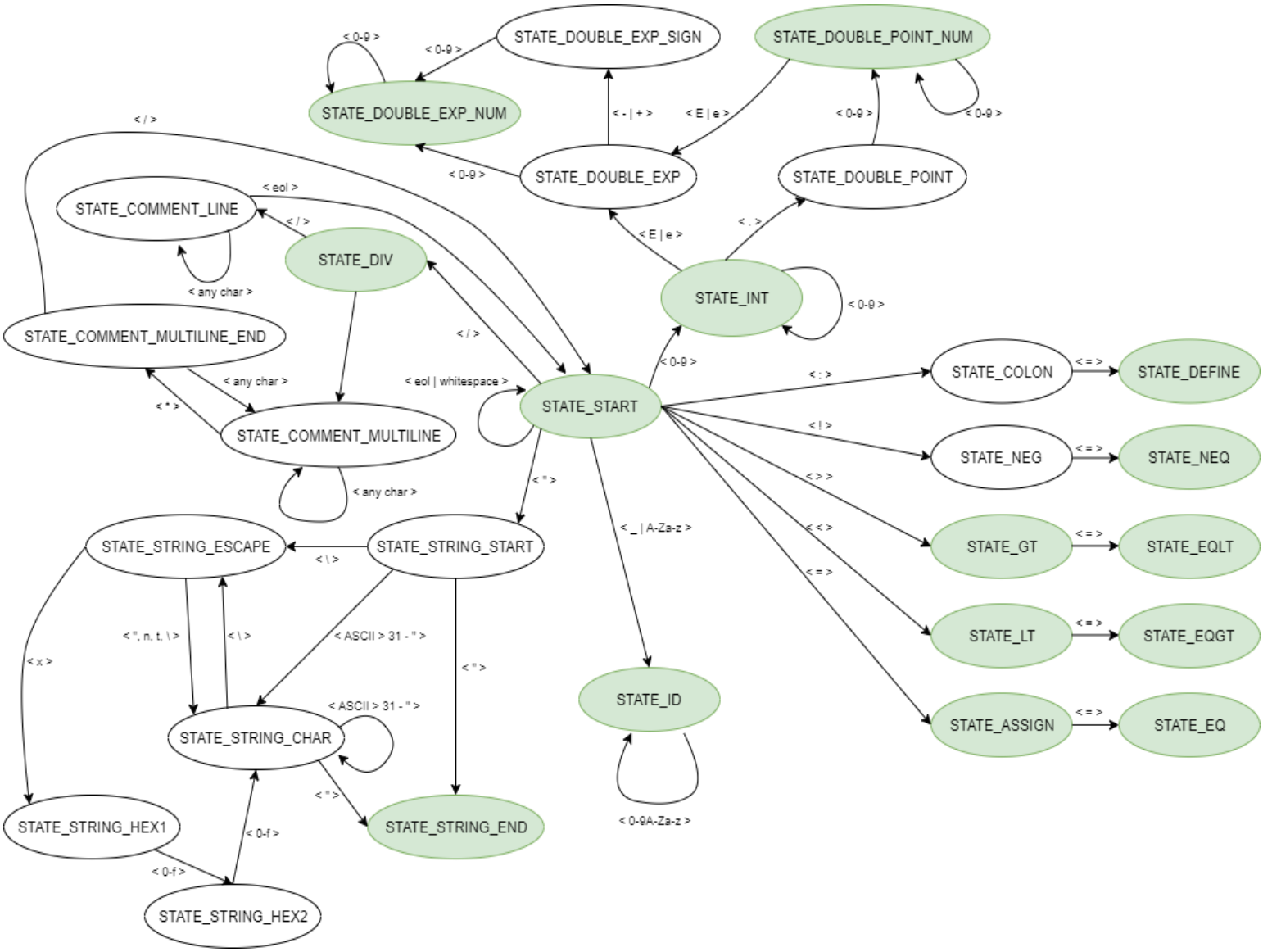
Dynamický řetězec

Implementace dynamického řetězce byla nutná kvůli specifikaci v zadání, která neomezovala maximální délku termu. Pro jeho implementaci je využita struktura typu *MyString*, která obsahuje proměnnou *lenght*, která uchovává aktuální délku řetězce, proměnnou *allocated_lenght*, která obsahuje délku řetězce, pro který je zrovna alokovaná paměť a proměnnou *value*, která drží samotný obsah řetězce. Při vytvoření řetězce se nejprve alokuje začáteční délka 50 znaků. Dále jsou implementovány základní operace pro práci s řetězcem, jako je například konkatenace se znakem či jiným řetězcem. Dynamičnost je implementována tak, že při konkatenaci se kontroluje délka řetězce s maximální alokovanou délkou a pokud je alokovaná délka menší, než aktuální, tak se nealokuje více místa.

Tabulka symbolů

Jelikož máme variantu zadání I, tak jsme tabulku symbolů implementovali jako binární vyhledávací strom. Při implementaci tabulky jsme se potýkali s určitými problémy. Jelikož jsme ji implementovali relativně brzy, neměli jsme přesnou představu, co všechno bude muset tabulka uchovávat a proto bylo třeba ji několikrát předělávat a upravit tak, ať se dá využít k řešení sémantických kontrol a pro generování kódu. Jsou implementovány základní funkce jako *insert* a *search*, které jsou hojně využívány pro implementaci jiných obslužných funkcí pro parser nebo v něm samotném. Za zmínku stojí netypická funkce *table_insert_function_args*, která do záznamu v tabulce, kde se již nachází definovaná funkce k ní uloží její návratové typy a typy parametrů, aby se k nim mohlo přistupovat a pracovat s nimi při sémantických kontrolách.

Diagram konečného automatu specifikující lexikální analyzátor



Zelené stavy jsou stavy konečné.

LL GRAMATIKA

1.POCIATOCNY_STAV-> eof
2.POCIATOCNY_STAV-> package main ZACIATOCNY_STAV
3.ZACIATOCNY_STAV-> DEFINICIA_FUNKCIE ZACIATOCNY_STAV
4.ZACIATOCNY_STAV-> eof

5.DEFINICIA_FUNKCIE-> func id (ARGUMENTY_VSTUPNE ARGUMENTY_NAVRATOVE eol STATEMENT_LIST
6.ARGUMENTY_VSTUPNE->)
7.ARGUMENTY_VSTUPNE-> id TYPE ARGS

8.ARGS-> , id TYPE ARGS
9.ARGS->)
10.ARGUMENTY_NAVRATOVE-> (TYPE NAVRATOVE {
11.ARGUMENTY_NAVRATOVE-> (NAVRATOVE_2 {
12.ARGUMENTY_NAVRATOVE-> {
13.NAVRATOVE_2->)
14.NAVRATOVE->)
15.NAVRATOVE-> , TYPE NAVRATOVE

16.STATEMENT_LIST-> }
17.STATEMENT_LIST-> STATEMENT STATEMENT_LIST

18.STATEMENT-> if EXPR { eol STATEMENT_LIST else { eol STATEMENT_LIST eol
19.STATEMENT-> eol
20.STATEMENT-> id := EXPR eol
21.STATEMENT-> id LAVA_STRANA PRAVA_STRANA eol
22.STATEMENT-> id (VYRAZY eol
23.STATEMENT-> for FOR_DEFINITION FOR_ARGS FOR_PRIRAD eol STATEMENT STATEMENT_LIST
24.STATEMENT-> return RETURN_LIST

25.LAVA_STRANA-> , id LAVA_STRANA
26.LAVA_STRANA-> =
27.PRAVA_STRANA-> EXPR PRAVA_STRANA2
28.PRAVA_STRANA-> id (VYRAZY
29.PRAVA_STRANA2-> , EXPR PRAVA_STRANA2
30.PRAVA_STRANA2-> epsilon

31.VYRAZY-> EXPR VYRAZY_NEXT
32.VYRAZY->)
33.VYRAZY_NEXT-> , EXPR VYRAZY_NEXT
34.VYRAZY_NEXT->)

35.FOR_DEFINITION-> id := EXPR ;
36.FOR_DEFINITION-> ;
37.FOR_ARGS-> EXPR ;
38.FOR_PRIRAD-> id LAVA_STRANA PRAVA_STRANA {
39.FOR_PRIRAD-> {

40.RETURN_LIST-> EXPR RETURN_STATEMENT
41.RETURN_LIST-> eol
42.RETURN_STATEMENT-> , EXPR RETURN_STATEMENT
43.RETURN_STATEMENT-> eol

44.TYPE-> int

45.TYPE-> float64

46.TYPE-> string

LL TABULKA

	eof	package	main	func	id	(	eol	)	,	{	}	if	else	:=	for	return	=	;	int	float64	string
POCIATOCNY_STAV	1	2																			
ZACIATOCNY_STAV	4			3																	
DEFINICIA_FUNKCIE				5																	
ARGUMENTY_VSTUPNE					7			6													
ARGUMENTY_NAVRATOVE						10,11				12											
STATEMENT_LIST					17		17				16	17			17	17					
TYPE																			44	45	46
ARGS								9	8												
NAVRATOVE								14	15												
NAVRATOVE_2								13													
STATEMENT					20,21,22		19					18			23	24					
EXPR																					
LAVA_STRANA									25								26				
PRAVA_STRANA					28																
VYRAZY								32													
FOR_DEFINITION					35													36			
FOR_ARGS																					
FOR_PRIRAD					38					39											
RETURN_LIST							41														
PRAVA_STRANA2							30		29	30											
VYRAZY_NEXT								34	33												
RETURN_STATEMENT							43		42												

Precedenční tabulka

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

r - relační operátory

Závěr

Prací na tomto projektu jsme získali velké množství zkušeností, ať už z tématu překladačů nebo práce v týmu. Tento projekt byl velmi časově náročný, což jsme zpočátku podcenili a poté se v závěru potýkali s časovou tísní při implementaci generace kódu. Jelikož práce na generaci kódu začala pozdě, nestihli jsme ji už úplně dodělat.