VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace projektu do předmětů IFJ a IAL

Název projektu:

Implementace překladače imperativního jazyka IFJ18.

Tým:

Číslo 033, varianta I

Členové týmu:

Sedláček Adam (vedoucí)	xsedla1e (25%)
Kurka David	xkurka04 (25%)
Havlíček Lukáš	xhavli46 (25%)
Katrušák Jaroslav	xkatru00 (25%)

Rozšíření:

-

Obsah:

1) Práce v týmu:

1.1) Spolupráce týmu

První schůzka celého týmu proběhla v prvním týdnu po jeho registraci. V návaznosti na tuto schůzku byla dohodnuta jednotná komunikační platforma a také byl vybrán vhodný verzovací nástroj. Společně s tímto bylo dohodnuto rozdělení projektu na 4 hlavní části, které byly rozděleny mezi členy týmu.

Komunikace týmu probíhala nejčastěji elektronickou formou na soukromém Discord serveru, kde probíhaly velice intenzivní diskuze, přičemž výsledkem je, v době psaní dokumentace, přes 13 000 zpráv. Přes jasně dané prvotní rozdělení práce se k diskuzi o konkrétním problému připojili vždy všichni členové týmu, kteří tak mnohdy pomáhali řešit komplikace nesouvisející s jejich částí.

1.2) Správa zdrojového kódu

Jelikož je tento projekt týmový, bylo potřeba zajistit, aby měli všichni členové týmu přístup k aktuálním zdrojovým souborům a rovněž aby bylo možné sledovat aktuální stav prací. Z tohoto důvodu byla hned na začátku vybrána verzovací služba – GitHub, která nám pomohla odhalit mnoho chyb, které vznikaly při přidávání funkcionality.

1.3) Vývoj překladače

Zpočátku vývoj probíhal velice pomalu, jelikož bylo potřeba si vytvořit představu, co vlastně celý projekt obnáší. První vyvíjenou částí překladače byl scanner, který se po prvotních návrzích automatu rýsoval velmi pomalu. Po prvotním tápání a následném zorientování se v problematice, se vývoj začal posouvat rychle kupředu, avšak vzhledem k dalším školním povinnostem se ne vždy dařilo dodržovat stanovené termíny. Každá další část pak byla vyvíjena vždy po dokončení a otestování předchozí části, což značně zjednodušovalo hledání chyb.

1.4) Dělení práce

Rozdělení stanovené na začátku vývoje se nakonec ve své původní podobě příliš neuplatnilo a bylo postupně změněno do následující podoby:

Sedláček Adam:	Generování kódu, pomocné soubory pro generování, testování
Kurka David:	Generování kódu, pomocné soubory pro syntaktickou a sémantickou analýzu, garbage collector
Havlíček Lukáš:	Syntaktická a sémantická analýza a pomocné soubory
Katrušák Jaroslav:	Lexikální analyzátor a jeho pomocné soubory, dokumentace

2) Implementace součástí

2.1) Struktura

Překladač se skládá ze čtyř hlavních částí – lexikálního analyzátoru (scanner.c), syntaktického analyzátoru (syntax.c) a sémantického analyzátoru (semantic.c), jenž spolu tvoří parser, a nakonec generátoru kódu (generate.c). Rozhodnutí rozdělit parser do dvou souborů, bylo učiněno z důvodu snadnějšího hledání chyb a zvýšení čitelnosti jednotlivých částí.

2.2) Lexikální analýza

Je implementována v souboru scanner.c, na základě zavolání ze syntaktického analyzátoru posílá tokeny. Tokeny, které jsou získávány ze vstupního souboru, jsou vytvářeny na základě lexikálních pravidel, která jsou znázorněna ve schématu konečného automatu.

Každý token je struktura, která obsahuje typ tokenu a jeho hodnotu. Struktura tokenu je implementována v souboru token.h.

Práce lexikálního analyzátoru spočívá v zapamatování si aktuálního stavu automatu, na základě čehož je posléze rozhodnuto, co se má udělat s dalším načteným znakem.

Načítání zdrojového souboru probíhá vždy po řádcích, kdy je poté postupně z každého načteného řádku analyzován každý znak, přičemž je uchovávána informace o aktuální pozici načteného znaku. Této informace je například využito při rozhodování, zda znak „=“ uvozuje značku blokového komentáře, či nikoliv.

2.3) Syntaktická analýza

Syntaktická analýza (syntax.c) je spouštěna jako první část překladače a vyvolává činnost scanneru. Pracuje na základě rekurzivního sestupu pomocí LL pravidel.

Při syntaktické analýze je každý získaný token nejdříve uložen do obousměrně vázaného seznamu, následně je posouzen podle syntaktických pravidel. Pokud vyhovuje pravidlům, tak je načten další token, pokud nevyhovuje je zavolána funkce syntax_error s odpovídající chybovou hláškou. Rovněž jsou při tomto průchodu ukládány názvy funkcí do tabulky symbolů a je provedena část sémantické kontroly redefinice funkcí.

Pokud je syntaktická analýza ukončena úspěchem, je zavolána sémantická analýza, které je předán obousměrně vázaný seznam tokenů, společně s tabulkou symbolů.

2.4) Sémantická analýza

Sémantická analýza (semantic.c), která tvoří druhý průchod, využívá tabulku symbolů vytvořenou při syntaktické analýze. V tabulce se mimo jména funkce nachází i informace o počtu parametrů, se kterými byla funkce definována, jako klíč je v tabulce využíván název funkce.

Do této tabulky je nahlíženo v případě, kdy se sémantická analýza nachází uvnitř funkce. Při průchodu funkcí je vytvářena další tabulka symbolů, tentokrát obsahující proměnné.

V případě, že se sémantická analýza nachází v hlavním těle programu, tak nahlíží do druhé sémantické tabulky, kterou si sama vytváří, při průchodu kódem.

Pokud je sémantická analýza ukončena úspěchem, je zavolána funkce ast_build, které je předán obousměrně vázaný seznam tokenů společně s druhou tabulkou symbolů, která byla vytvořena sémantickou analýzou. Funkce ast_build zajistí vytvoření abstraktního syntaktického stromu, na základě kterého bude generován kód v generátoru kódu (generate.c).

2.5) Generování kódu

Generátor kódu (generate.c) generuje kód na základě abstraktního syntaktického stromu, v jehož levé části se nacházejí definice funkcí, jež se při generování vkládají před „main“ a v pravé části se nachází volání funkcí a samotný „main“.

Samotné generování funguje na základě rekurzivního volání, kdy je postupováno shora dolů, v průběhu čehož jsou volány jednotlivé funkce zajišťující generování příslušného kódu.

3) Závěr

Projekt byl zvláště svým rozsahem mimořádně časově náročný a jeho vyhotovení zabralo každému členovi týmu mnoho desítek hodin. Přestože přípravné práce na projektu začali velice brzy, tak samotná implementace započala až za několik týdnů. Byla provázena mnoha zpožděními oproti původnímu plánu, zejména z důvodu mnoha dalších školních povinností. Na pomalý počáteční postup však měla vliv i problematika, kterou se tento projekt zabýval, jelikož byla pro všechny členy týmu do této doby naprosto neznámá a bylo potřeba se s ní podrobně seznámit.

V průběhu vývoje jsme naráželi na mnohá úskalí, která bylo potřeba vyřešit, aby mohl vývoj postoupit dále. Prvním řešeným problémem bylo testování, které jsme se, po prvotním testování scanneru, rozhodli pro rychlejší a efektivnější práci zautomatizovat, jelikož opakované manuální testování a procházení všech výsledků bylo značně časově náročné.

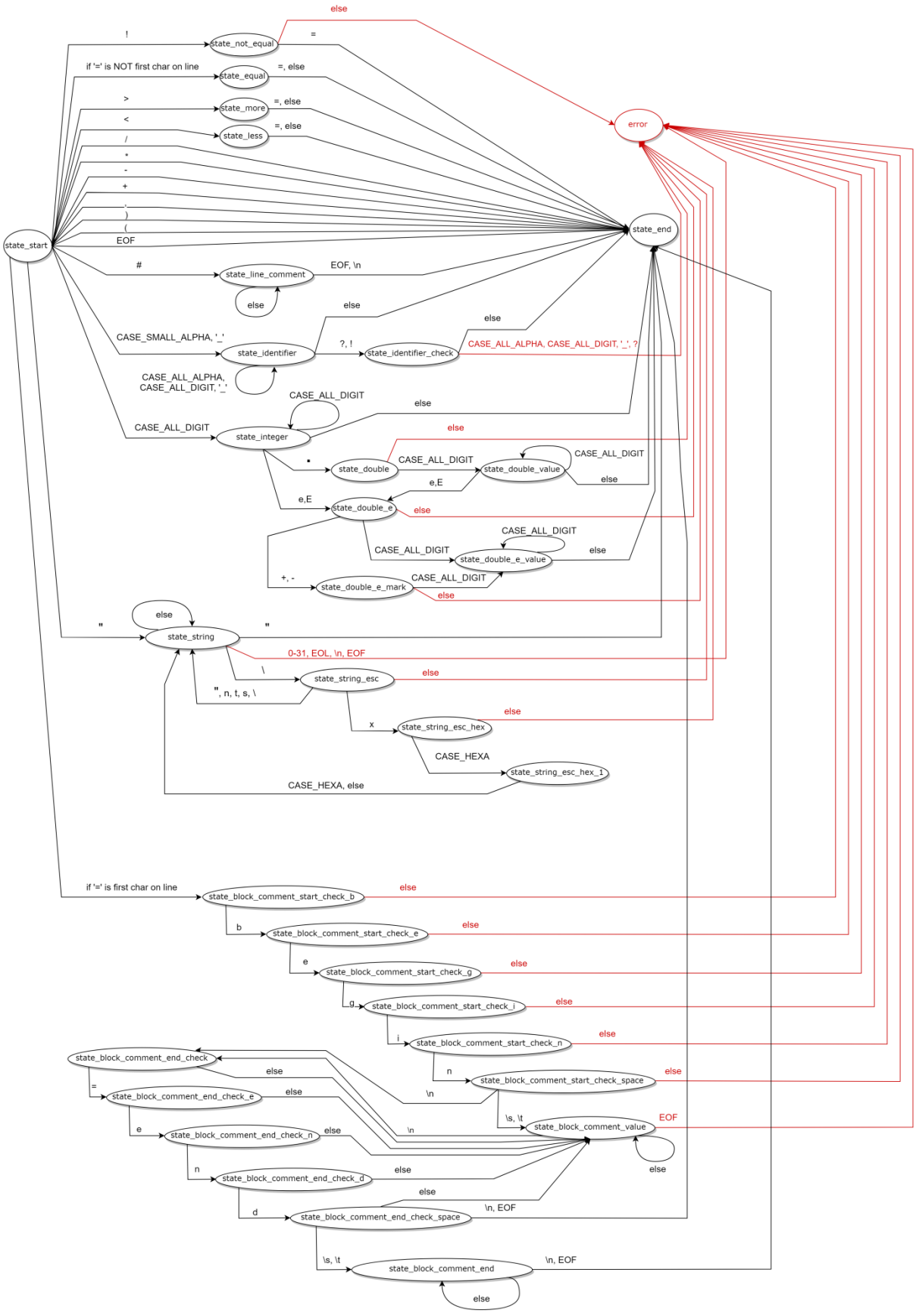
Jedním z dalších problémů, se kterým jsme přišli do styku, byla práce s pamětí, kdy po začátku prací na syntaktické a sémantické analýze nám začínalo být jasné, že bude potřeba se o dealokaci paměti začít starat centralizovaně pro celý program najednou. Toto jsme vyřešili napsáním tzv. garbage collectoru (my_malloc.c), který nám práci s pamětí značně zjednodušil napříč celým programem.

Přestože jsme v průběhu vývoje museli mnoho prvotních rozhodnutí přehodnotit, tak ne všechna tato rozhodnutí se ukázala být špatná. Zejména se nám vyplatilo rozdělení programu na více částí, např. rozdělení syntaktické a sémantické analýzy do dvou souborů, což zlepšilo čitelnost a usnadnilo testování a izolaci chyb.

Mimo to, že jsme díky tomuto projektu mohli nahlédnout do, pro nás, dosud neprobádané problematiky, nám celý tento projekt také názorně předvedl problematiku týmového vývoje se všemi jeho pozitivy a negativy společně s potřebou dynamicky reagovat na nově zjištěné poznatky.

3.1) Použitá literatura

Námi použitými zdroji při práci na projektu byly přednášky z předmětů IFJ a IAL.



Příloha 1: Deterministický konečný automat

S	→	<inside_command> <S>
S	→	def <def_func> <S>
S		EOF
inside_command	→	EOL
inside_command	→	id <id_came>
inside_command	→	<if>
inside_command	→	<while>
def_func	→	id (<param>) EOL <inside_command> end EOL
param	→	epsilon
param	→	id <params>
param	→	<item> <params>
params	→	epsilon
params	→	id <params>
params	→	<item> <params
id_came	→	EOL //expression or call_func without params
id_came	→	<call_func>
id_came	→	operator <expr> EOL
id_came	→	<assign>
call_func	→	(<param>) EOL
call_func	→	id, <params> EOL
if	→	if <expr_bool> then EOL <command> else EOL <command> end EOL
while	→	while <expr_bool> do EOL <command> end EOL
assign	→	id <call_func>
assign	→	id operator <expr> EOL
assign	→	id EOL
assign	→	(<expr>)
expr	→	<expr> + <expr>
expr	→	<expr> - <expr>
expr	→	<expr> * <expr>
expr	→	<expr> / <expr>
expr	→	(<expr>)
expr	→	id
expr	→	<item>

Příloha č.2: LL-gramatika

expr_bool	→	<expr_bool> + <expr_bool>
expr_bool	→	<expr_bool> - <expr_bool
expr_bool	→	<expr_bool> * <expr_bool>
expr_bool	→	<expr_bool> / <expr_bool>
expr_bool	→	(<expr_bool>)
expr_bool	→	<expr_bool> == <expr_bool>
expr_bool	→	<expr_bool> != <expr_bool>
expr_bool	→	<expr_bool> > <expr_bool>
expr_bool	→	<expr_bool> < <expr_bool>
expr_bool	→	<expr_bool> => <expr_bool>
expr_bool	→	<expr_bool> =< <expr_bool>
expr_bool	→	id
expr_bool	→	<item>
item	→	nil
item	→	str
item	→	float
item	→	int

Příloha č.2: LL-gramatika

	+	-	*	/	(	)	<	>	<=	>=	=	=	!=	id	$
+	>	>	<	<	<	>	>	>	>	>	>	>	>	<	>
-	>	>	<	<	<	>	>	>	>	>	>	>	>	<	>
*	>	>	>	>	<	>	>	>	>	>	>	>	>	<	>
/	>	>	>	>	<	>	>	>	>	>	>	>	>	<	>
(	<	<	<	<	<	=	<	<	<	<	<	<	<	<	!
)	>	>	>	>	!	>	>	>	>	>	>	>	>	!	>
<	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
>	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
<=	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
>=	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
=	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
=	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
!=	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
id	>	>	>	>	!	<	>	>	>	>	>	>	>	!	>
$	<	<	<	<	<	!	<	<	<	<	<	<	<	<	!

Příloha č.3: Precedenční tabulka