Vysoké učení technické v Brně
Fakulta informačních technologií

Dokumentace k projektu z předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ17

Tým 080, varianta II

Členové týmu

Miroslav Válka	xvalka05	25 %	vedoucí
Jak Trněný	xtrnen03	25 %	
Lukáš Prokop	xproko37	25 %	
Pavel Bartoň	xbarto93	25 %	

1 Úvod

Dokumentace popisuje projekt do předmětů *Formální jazyky a překladače* a *Algoritmy*. Cílem tohoto projektu je vytvořit překladač programovacího jazyka `IFJ17`, který je podmnožinou programovacího jazyka `FreeBASIC`, do jazyka `IFJcode17`.

Projekt je složen z několika hlavních částí:

- lexikální analýza
- syntaktická analýza
- sémantická analýza
- generování kódu

Dle zadání varianty II je tabulka symbolů implementovaná pomocí tabulky s rozptýlenými položkami.

2 Tým

2.1 Rozdělení práce

Miroslav Válka – scanner, parser, ast, vypracování tabulek, dokumentace
Jan Trněný – symtable, generator, zkontrolování tabulek
Lukáš Prokop – token, conversion, zkontrolování tabulek, dokumentace
Pavel Bartoň – dstring, list, zkontrolování tabulek, testy

2.2 Průběh týmové práce

Již během prvního týdne zimního semestru jsme vytvořili tým o 4 členech. Hned druhý týden jsme uspořádali první schůzku, kde jsme si stanovili základní pravidla spolupráce jako například společná hlavička a patička u všech souborů, styl a velikost odsazování, styl psaní komentářů či názvů proměnných, funkcí a konstant. Bylo také důležité se dohodnout na pravidlech pro práci s repozitářem. Mezi ně patří například to, že nikdo nebude přímo commitovat do větve master, že každý řešený problém bude mít svoji větev a že pull request musí potvrdit jiný člověk než ten, který o něj požádal.

Pro společnou práci na projektu jsme využívali *GitHub*. Pro rozdělování úkolů nám posloužila služba *Trello*, která nabízí velmi přehlednou a efektivní organizaci úkolů. Kromě několika osobních setkání, na kterých jsme probírali a plánovali budoucí úkoly, jsme byli v kontaktu na skupinovém chatu na Facebooku.

3 Implementace

3.1 Lexikální analýza

O provedení lexikální analýzy se stará knihovna `scanner`, která je implementovaná v souborech `scanner.h` a `scanner.c`. Knihovna `scanner` dále využívá knihovny `token`, `error`, `conversion` a `dstring`, o kterých se podrobněji zmiňujeme v části *Pomocné knihovny*.

Scanner je využíván jako prostředník mezi vstupním souborem a dalšími částmi překladače. Konkrétně se scanner zabývá zpracováním vstupního kódu a následným převodem na tokeny. Během tohoto zpracování ignoruje scanner většinu bílých znaků (s výjimkou konce řádku) a komentářové sekvence jazyka IFJ17, přičemž postupně načítané lexémy převádí na tokeny. K určování typu tokenu je využíván konečný automat.

Scanner nezpracovává celý vstup v jeden okamžik, ale dochází k postupnému zpracování. V okamžiku, kdy je načten token, se jeho činnost pozastaví a je vyčkáváno na žádost o další token. Do struktury tokenu je ukládán textový řetězec reprezentující přečtenou část vstupu, číslo řádku, ze kterého bylo čteno, a typ tokenu. Pokud token představuje konstantu, je uložena její hodnota.

3.2 Syntaktická analýza

Syntaktickou analýzu zajišťuje knihovna `parser`, která je implementovaná v souborech `parser2.h` a `parser2.c`. Právě parser je hlavní částí překladače, protože obstarává celé řízení běhu programu. Ke své činnosti tedy využívá veškeré naše knihovny - `token`, `error`, `symtable`, `list`, `ast`, `scanner` a `generator`.

Jednou z úloh parseru je právě syntaktická kontrola, která probíhá v souladu s pravidly *LL-Gramatiky*, množinami *First*, *Empty* a *Predic*, a *LL-Tabulkou* pro rekurzivní sestup. Pro zpracování výrazů je použita *precedenční tabulka*.

Parser vždy požádá scanner o token, který je následně zpracován skrze rekurzivní volání funkcí, které reprezentují jednotlivé neterminály. Během zpracování tokenu se vytváří abstraktní syntaktický strom. Zpracovaný token je následně uložen na své místo v tomto abstraktním syntaktickém stromě. V případě výrazu je volána speciální funkce pro zpracování výrazu. Po zpracování výrazu je do uzlu abstraktního stromu uložen list tokenů v postfix podobě.

Současně s tvorbou abstraktního stromu je plněna také tabulka symbolů. Parser obsahuje globální tabulku symbolů, která obsahuje záznamy funkcí, přičemž každá funkce má svou vlastní tabulku symbolů, ve které jsou uchovávány záznamy o parametrech a proměnných dané funkce.

3.3 Sémantická analýza

Rovněž o sémantickou analýzu se stará knihovna `parser`, a to v průběhu vytváření abstraktního syntaktického stromu za pomoci globální tabulky symbolů a lokálních tabulek symbolů. Zvláštní pozornost je věnovaná kontrole datových typů u výrazů. Do listu v postfix podobě jsou na správná místa vloženy i tokeny, které představují implicitní konverze z celočíselné hodnoty na desetinnou.

3.4 Generování kódu

Poslední částí je generování kódu, které zajišťuje knihovna `generator`. Ta je implementovaná v souborech `generator.h` a `generator.c`. Úkolem generátoru je generovat kód v jazyce `IFJcode17`. Kód je generován v průběhu průchodu abstraktního syntaktického stromu, který nám dodává parser.

Generovaný kód využívá především zásobník a zásobníkové instrukce. Právě přes zásobník jsou předávány parametry funkcím. Zásobník je rovněž využíván k výpočtům výrazů. Instrukce, které nemají zásobníkovou variantu, jsou řešeny přes pomocné proměnné, přičemž výsledná hodnota je umístěna zpět na zásobník.

Generátor kódu zajišťuje také implicitní konverze mezi datovými typy integer a double, generuje také vestavěné funkce. Kódy vestavěných funkcí nejsou minimalistické a samozřejmě je lze do budoucna optimalizovat. Částečně byly vygenerovány samotným generátorem a následně byly manuálně upraveny.

3.5 Pomocné knihovny

3.5.1 Knihovna `error`

Knihovna `error` je implementována v souborech `error.h` a `error.c`. Tato knihovna obsahuje návratovou hodnotu programu a funkce pro nastavování různých typů chyb.

3.5.2 Knihovna `token`

Knihovna `token` je implementována v souborech `token.h` a `token.c`. Tato knihovna poskytuje funkce pro spravování tokenů, tedy funkce sloužící pro vytváření, uvolňování či kontrolní výtisk tokenů. Součástí této knihovny je také samotná struktura tokenu a typy tokenů.

3.5.3 Knihovna `dstring`

Knihovna `dstring` je implementována v souborech `dstring.h` a `dstring.c`. Tato knihovna poskytuje funkce pro práci s dynamickým textovým řetězcem.

3.5.4 Knihovna `conversion`

Knihovna `conversion` je implementována v souborech `conversion.h` a `conversion.c`. Tato knihovna slouží k upravování tokenu. Konkrétně je nad tokeny typu `ID` spuštěn filtr klíčových slov a případně je změněn typ tokenu na příslušný typ klíčového slova. Dále také u tokenů představujících čísla provede převod textu do číselné hodnoty typu `integer` nebo `double`, a tato převedená hodnota je uložena do tokenu.

3.5.5 Knihovna `list`

Knihovna `list` je implementována v souborech `list.h` a `list.c`. Tato knihovna obsahuje strukturu listu a funkce sloužící pro práci s ním. List lze využívat jako seznam, zásobník či frontu, vždy podle potřeby.

3.5.6 Knihovna `ast`

Knihovna `ast` je implementována v souborech `ast.h` a `ast.c`. Tato knihovna obsahuje strukturu abstraktního syntaktického stromu a funkce pro práci s ním.

3.5.7 Knihovna `symtable`

Knihovna `symtable` je implementována v souborech `symtable.h` a `symtable.c`. Tato knihovna obsahuje strutury a funkce pro práci s tabulkou symbolů. Tabulka symbolů je dle zadání implementována jako tabulka s rozptýlenými položkami.

4 Závěr

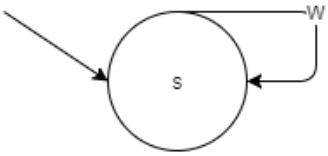
Práce na tomto projektu pro nás byla přínosem nejen z pohledu získání nových znalostí a schopností, ale také z hlediska získání nových zkušeností při týmovém vývoji, ať už se jednalo o vzájemnou komunikaci či spolupráci.

Během práce na projektu se samozřejmě vyskytla celá řada problémů - počínaje malými chybami jako například chybné commity na GitHubu a konče chybami většími jako například zcela nefunkční syntaktická analýza. Nicméně i to je součástí každého projektu a je třeba se umět s problémy vypořádat.

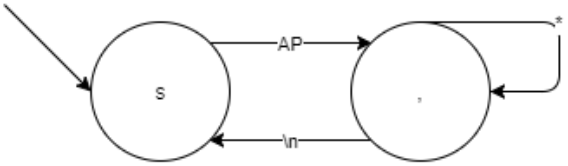
5 Přílohy

5.1 Konečný automat - lexikální analýza

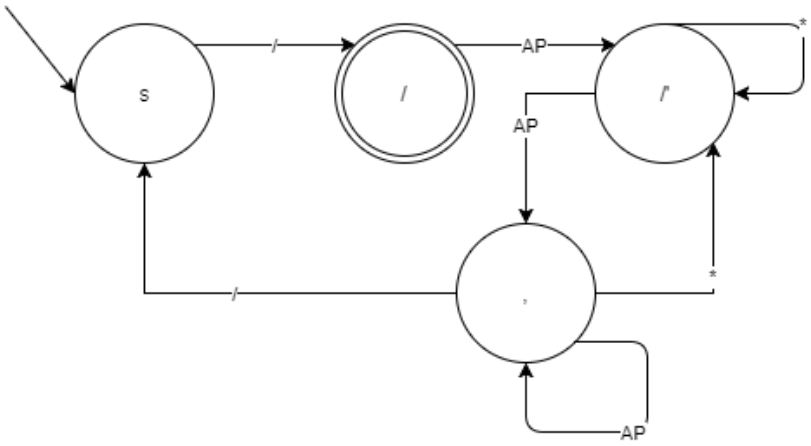
AZ = A...Z - Velká písmena
az = a...z - Malá písmena
N = 0...9 - číslice
W = { ,\n,\t,\r} - Bílé znaky
AP = { ' } - Apostrof
* = Všechny jiné znaky, které stav nebere
+/- = Pamatovat si znak v stavo pro další zpracování
R = { ,\n,\t,\r,\,+,-,*,/, , , <,>, = ,(,),[,],{,}} - znaky, které oddělují tokeny
d - možná rozšíření



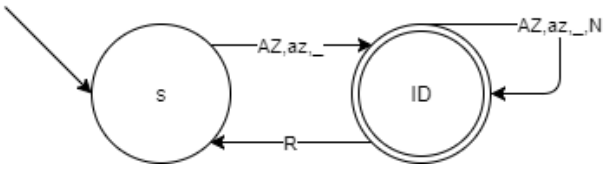
Ignorování bílých znaků



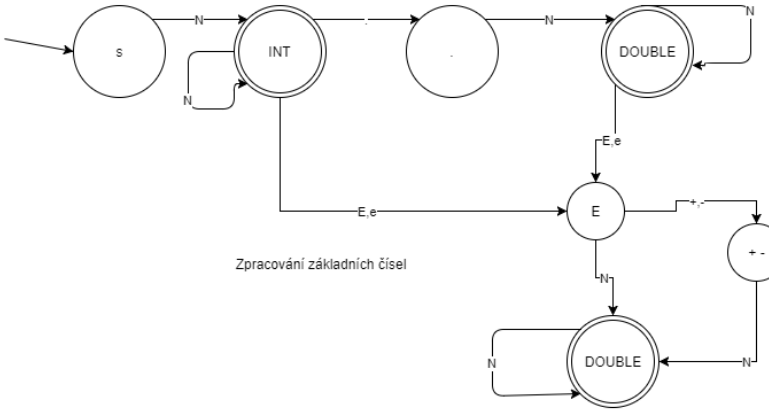
Ignorování jednořádkových komentářů



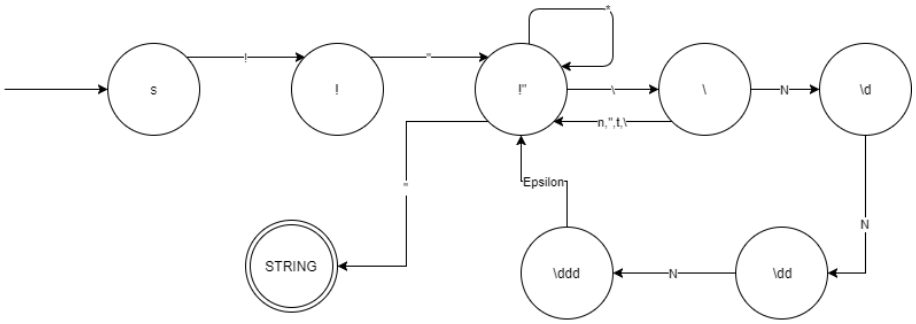
Ignorování víceřádkových komentářů



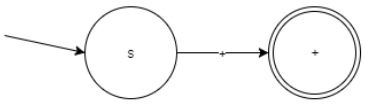
Identifikátory



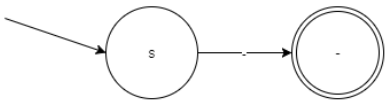
Zpracování základních čísel



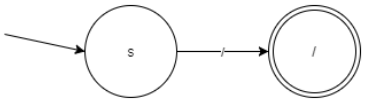
Textový řetězec



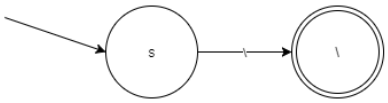
Plus



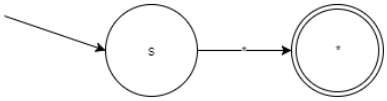
Mínus



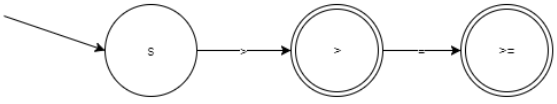
Klasické dělení



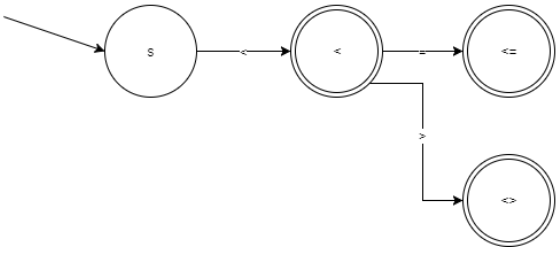
Celočíselné dělení



Násobení/ukazatel



Porovnání



5.2 LL-Gramatika

č.	PRAVIDLA
1	Program → ListDecDef ScopeDef
2	ListDecDef → declare FunctionHead ListDecDef
3	ListDecDef → FunctionHead FunctionBody FunctionEnd ListDecDef
4	ListDecDef → ”
5	FunctionHead → function id (ListParam) as DataType eol
6	ListParam → Param NextParam
7	ListParam → ”
8	Param → id as DataType
9	NextParam → , Param NextParam
10	NextParam → ”
11	DataType → integer
12	DataType → double
13	DataType → string
14	FunctionEnd → end function eol
15	ScopeDef → ScopeHead FunctionBody ScopeEnd
16	ScopeHead → scope eol
17	ScopeEnd → end scope ScopeAfter
18	ScopeAfter → eol ScopeAfter
19	ScopeAfter → ”
20	FunctionBody → ListVarDef ListCommand
21	ListVarDef → dim id as DataType VarDefAssigment eol ListVarDef
22	ListVarDef → ”
23	VarDefAssigment → = Expression
24	VarDefAssigment → ”
25	ListCommand → Command eol ListCommand
26	ListCommand → ”
27	Command → do while Condition eol ListCommand loop
28	Command → if Condition then eol ListCommand else eol ListCommand end if
29	Command → input id
30	Command → print ListExpression
31	ListExpression → Expression ;ListExpression
32	ListExpression → ”
33	Condition → Expression
34	Command → id = Assignment
35	Assignment → id (ListInParam)
36	Assignment → Expression
37	ListInParam → InParam NextInParam
38	ListInParam → ”
39	InParam → Term
40	NextInParam → , InParam NextInParam
41	NextInParam → ”
42	Term → id
43	Term → int
44	Term → float
45	Term → str
46	Command → return Expression
47	Assignment → length (ListInParam)
48	Assignment → substr (ListInParam)
49	Assignment → asc (ListInParam)
50	Assignment → chr (ListInParam)

5.3 LL-Gramatika – množiny

FIRST	FOLLOW	Nonterminal
{declare,",function,scope}	{$}	Program
{declare,",function}	{scope}	ListDecDef
{function}	{declare,scope,function,dim,do,if,input,print,id,return,end}	FunctionHead
{",id}	{)}	ListParam
{id}	{„)}	Param
{„"}	{)}	NextParam
{integer,double,string}	{eol„,),=,end,do,if,input,print,id,return}	DataType
{end}	{declare,scope,function}	FunctionEnd
{scope}	{$}	ScopeDef
{scope}	{dim,$,do,if,input,print,id,return,end}	ScopeHead
{end}	{$}	ScopeEnd
{eol,"}	{$}	ScopeAfter
{dim,",do,if,input,print,id,return}	{end}	FunctionBody
{dim,"}	{end,do,if,input,print,id,return}	ListVarDef
{=,"}	{eol}	VarDefAssigment
{",do,if,input,print,id,return}	{end,loop,else}	ListCommand
{do,if,input,print,id,return}	{eol}	Command
{Expression,"}	{eol}	ListExpression
{Expression}	{eol,then}	Condition
{id,Expression,length,substr,asc,chr}	{eol}	Assignment
{",id,int,float,str}	{)}	ListInParam
{id,int,float,str}	{„)}	InParam
{„"}	{)}	NextInParam
{id,int,float,str}	{„)}	Term

5.4 LL-Tabulka

viz str. 9 a 10

5.5 Precedenční tabulka

viz str. 11

Nonterminal	declare	function	id	(	)	eol	as	,	integer	double	string	end	scope	dim	while	loop
Program	1	1											1			
ListDecDef	2	3											4			
FunctionHead		5														
ListParam			6		7											
Param			8													
NextParam					10			9								
DataType									11	12	13					
FunctionEnd												14				
ScopeDef													15			
ScopeHead													16			
ScopeEnd												17				
ScopeAfter						18										
FunctionBody			20									20		20	20	
ListVarDef			22									22		21	22	
VarDefAssignment						24										
ListCommand			25									26			25	26
Command			34												27	
ListExpression						32										
Condition																
Assignment			35													
ListInParam			37		38											
InParam			39													
NextInParam					41			40								
Term			42													

Nonterminal	if	else	input	print	Expression	;	=	int	float	str	return	length	substr	asc	chr	$
Program																1
ListDecDef																
FunctionHead																
ListParam																
Param																
NextParam																
DataType																
FunctionEnd																
ScopeDef																
ScopeHead																
ScopeEnd																
ScopeAfter																19
FunctionBody	20		20	20							20					
ListVarDef	22		22	22							22					
VarDefAssignment							23									
ListCommand	25	26	25	25							25					
Command	28		29	30							46					
ListExpression					31											
Condition					33											
Assignment					36							47	48	49	50	
ListInParam								37	37	37						
InParam								39	39	39						
NextInParam																
Term								43	44	45						

	TK_PLUS	TK_MINUS	TK_MUL	TK_DIV	TK_DIV_INT	TK_MOD	TK_EQUAL	TK_NOT_EQUAL	TK_LESS	TK_GREATER	TK_LESS_EQUAL	TK_GREATER_EQUAL	TK_ID	TK_NUM_INTEGER	TK_NUM_DOUBLE	TK_NUM_STRING	TK_BRACKET_ROUND_LEFT	TK_BRACKET_ROUND_RIGHT	$
TK_PLUS	>	>	<	<	<	<	>	>	>	>	>	>	<	<	<	<	<	>	
TK_MINUS	>	>	<	<	<	<	>	>	>	>	>	>	<	<	<	<	<	>	
TK_MUL	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	>	
TK_DIV	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	>	
TK_DIV_INT	>	>	<	<	>	>	>	>	>	>	>	>	<	<	<	<	<	>	
TK_MOD	>	>	<	<	>	>	>	>	>	>	>	>	<	<	<	<	<	>	
TK_EQUAL	<	<	<	<	<	<							<	<	<	<	<	>	
TK_NOT_EQUAL	<	<	<	<	<	<							<	<	<	<	<	>	
TK_LESS	<	<	<	<	<	<							<	<	<	<	<	>	
TK_GREATER	<	<	<	<	<	<							<	<	<	<	<	>	
TK_LESS_EQUAL	<	<	<	<	<	<							<	<	<	<	<	>	
TK_GREATER_EQUAL	<	<	<	<	<	<							<	<	<	<	<	>	
TK_ID	>	>	>	>	>	>	>	>	>	>	>	>						>	>
TK_NUM_INTEGER	>	>	>	>	>	>	>	>	>	>	>	>						>	>
TK_NUM_DOUBLE	>	>	>	>	>	>	>	>	>	>	>	>						>	>
TK_NUM_STRING	>	>	>	>	>	>	>	>	>	>	>	>						>	>
TK_BRACKET_ROUND_LEFT	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	
TK_BRACKET_ROUND_RIGHT	>	>	>	>	>	>	>	>	>	>	>	>						>	>
$													<	<	<	<	<		