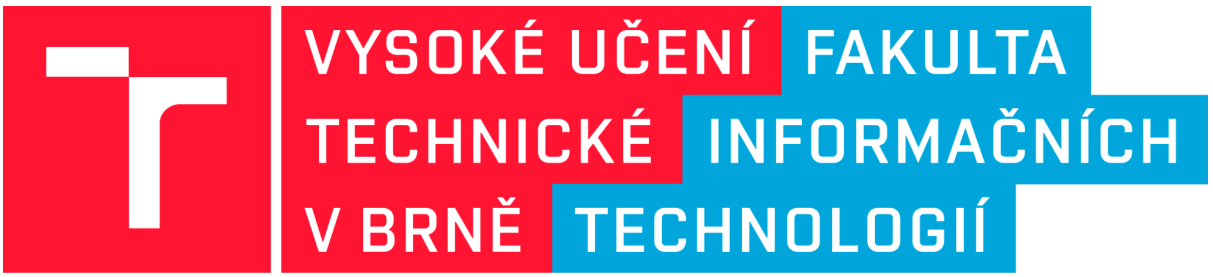


IFJ Projekt

Tím xvitkom00, varianta TRP-izp

Márk Vitkó	25%	xvitkom00 (vedúci tímu)
Daniela Jurišinová	25%	xjurisd00
Daniel Čerešňa	25%	xceresd00
Tadeáš Bujdosó	25%	xbujdot00

Rozdelenie práce medzi členmi tímu:

Návrh konečného automatu pre lexikálnu analýzu:
Márk Vitkó,
Daniela Jurišinová,
Daniel Čerešňa,
Tadeáš Bujdosó

Návrh gramatických pravidiel pre syntaktickú analýzu:
Márk Vitkó,
Daniela Jurišinová,
Daniel Čerešňa

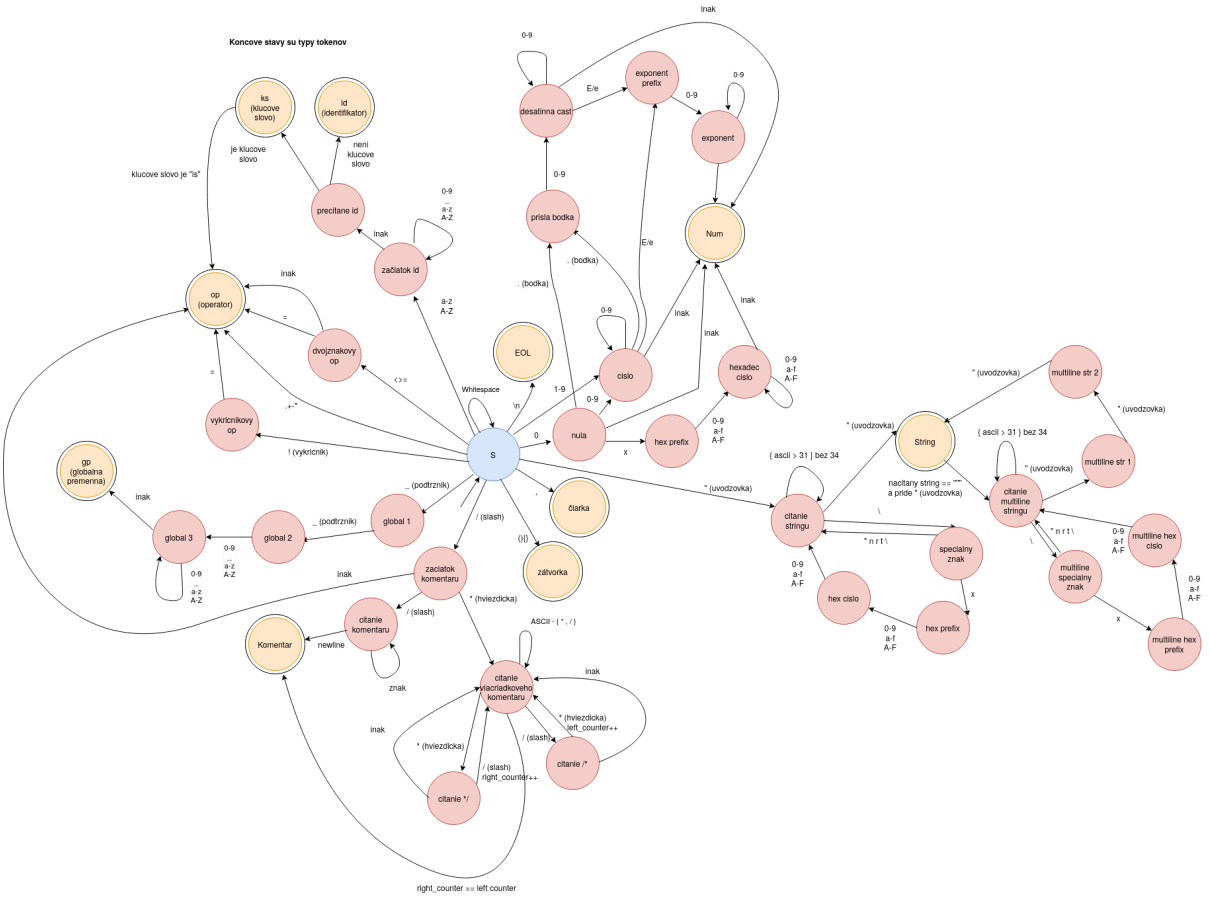
Implementácia dátových štruktúr:
Mark Vitkó,
Daniel Čerešňa

Implementácia lexikálnej analýzy:
Mark Vitkó,
Daniela Jurišinová

Implementácia syntaktickej a sémantickej analýzy:
Mark Vitkó,
Daniel Čerešňa

Návrh a implementácia generátoru kódu:
Tadeáš Bujdosó

Diagram KA pre lexikálnu analýzu



modrý stav	**počiatočný** stav
červený stav	klasický **“medzistav”**
oranžový stav	**koncový** stav

LL1 gramatika a tabuľka pre syntaktickú analýzu

1 CODE_BLOCK := '{' EOL SEQUENCE '}'

2.1 SEQUENCE := EOL SEQUENCE_PRIME
2.2 SEQUENCE := INSTRUCTION SEQUENCE_PRIME

3.1 SEQUENCE_PRIME := EOL SEQUENCE_PRIME
3.2 SEQUENCE_PRIME := INSTRUCTION SEQUENCE_PRIME

4.1 INSTRUCTION := "return"
4.2 INSTRUCTION := "if"
4.3 INSTRUCTION := "while"
4.4 INSTRUCTION := "var"
4.5 INSTRUCTION := "static"
4.6 INSTRUCTION := '{'
4.7 INSTRUCTION := identif
4.8 INSTRUCTION := global_identif

5.1 RETURN := "return" EOL
5.2 RETURN := "return" EXPRESSION

6 IF := "if" '(' EXPRESSION ')' CODE_BLOCK "else" CODE_BLOCK

7 WHILE := "while" '(' EXPRESSION ')' CODE_BLOCK

8 DECLARATION := "var" identif

9.1 ALLOWED_EOL := ',' ALLOWED_EOL_PRIME
9.2 ALLOWED_EOL := operator ALLOWED_EOL_PRIME

10 ALLOWED_EOL_PRIME := EOL ALLOWED_EOL_PRIME

11.1 ASSIGNMENT := identif '=' EXPRESSION_OR_FUNCTION
11.2 ASSIGNMENT := global_identif '=' EXPRESSION_OR_FUNCTION

12.1 EXPRESSION_OR_FUNCTION := string EXPRESSION
12.2 EXPRESSION_OR_FUNCTION := num EXPRESSION
12.3 EXPRESSION_OR_FUNCTION := '(' FUNCTION_CALL
12.4 EXPRESSION_OR_FUNCTION := '.' FUNCTION_CALL

13.1 FUNCTION_DECLARATION_BEGIN := "static" identif '(' FUNCTION DECLARATION
13.2 FUNCTION_DECLARATION_BEGIN := "static" identif '=' SETTER_DECLARATION
13.3 FUNCTION_DECLARATION_BEGIN := "static" identif GETTER_DECLARATION

14 FUNCTION_DECLARATION := "(" FUNCTION_PARAMETERS ")" CODE_BLOCK

15.1 FUNCTION_CALL := identif "(" FUNCTION_PARAMETERS ")"
15.2 FUNCTION_CALL := "Ifj" ALLOWED_EOL identif "(" FUNCTION_PARAMETERS ")"

16.1 FUNCTION_PARAMETERS := identif FUNCTION_PARAMETERS_PRIME
16.2 FUNCTION_PARAMETERS := string FUNCTION_PARAMETERS_PRIME
16.3 FUNCTION_PARAMETERS := EXPRESSION FUNCTION_PARAMETERS_PRIME

17.1 FUNCTION_PARAMETERS_PRIME := ',' ALLOWED_EOL EXPRESSION FUNCTION_PARAMETERS_PRIME
17.2 FUNCTION_PARAMETERS_PRIME := ',' ALLOWED_EOL string FUNCTION_PARAMETERS_PRIME

18 GETTER_DECLARATION := CODE_BLOCK EOL

19 SETTER_DECLARATION := '=' '(' identif ')' CODE_BLOCK EOL

	{	EOL	return	if	while	var	static	identif	global_identif	,	operator	string	num	(	.	lfj	=	Null (kw)	-
CODE_BLOCK	1																		
SEQUENCE	2.2	2.1	2.2	2.2	2.2	2.2	2.2	2.2	2.2										
SEQUENCE_PRIME	3.2	3.1	3.2	3.2	3.2	3.2	3.2	3.2	3.2										
INSTRUCTION	4.6		4.1	4.2	4.3	4.4	4.5	4.7	4.8										
RETURN			5.1, 5.2 *																
IF				6															
WHILE					7														
DECLARATION						8													
ALLOWED_EOL										9.1	9.2								
ALLOWED_EOL_PRIME		10																	
ASSIGNMENT								11.1	11.2										
EXPRESSION_OR_FUNCTION												12.1	12.2	12.3	12.4				
FUNCTION_DECLARATION_BEGIN							13.1, 13.2, 13.3 **												
FUNCTION_DECLARATION														14					
FUNCTION_CALL								15.1								15.2			
FUNCTION_PARAMETERS								16.1, 16.3	16.3			16.2, 16.3	16.3	16.3				16.3	16.3
FUNCTION_PARAMETERS_PRIME										17.1, 17.2 ****									
GETTER_DECLARATION	18																		
SETTER_DECLARATION																	19		

* poz.: v implementácii sa pozrieme o token ďalej a rozhodneme, ktoré pravidlo použijeme
** poz.: v implementácii sa pozrieme o 2 tokeny ďalej a rozhodneme, ktoré pravidlo použijeme
*** poz.: pri pravidle 16.3 sa orientujeme podľa neterminálu EXPRESSION, na ktoré bola použitá precedenčná analýza (v tab. zobrazené v stĺpcoch, ktorými sa EXPRESSION môže začínať)
**** poz.: v implementácii sa pozrieme o token ďalej a rozhodneme, ktoré pravidlo použijeme

20 EXPRESSION := precedenčná tabuľka

	*	/	+	-	<	>	<=	>=	is	"=="	!=
*	=	=	*	*	*	*	*	*	*	*	*
/	=	=	/	/	/	/	/	/	/	/	/
+	*	/	=	=	+	+	+	+	+	+	+
-	*	/	=	=	-	-	-	-	-	-	-
<	*	/	+	-	=	=	=	=	<	<	<
>	*	/	+	-	=	=	=	=	>	>	>
<=	*	/	+	-	=	=	=	=	<=	<=	<=
>=	*	/	+	-	=	=	=	=	>=	>=	>=
is	*	/	+	-	<	>	<=	>=	=	is	is
"=="	*	/	+	-	<	>	<=	>=	is	=	=
!=	*	/	+	-	<	>	<=	>=	is	=	=

Popis členenia implementačného riešenia
Implementácia kódu prekladača je rozdelená nasledovne:

lexer.h - dátová štruktúra a hlavičky funkcií pre lexer
lexer.c - implementácia funkcií pre lexer

syntactic.h - dátová štruktúra a hlavičky funkcií pre syntaktický analyzátor
syntactic.c - implementácia funkcií pre syntaktický analyzátor a precedenčnej syntaktickej analýze

semantic.h - dátová štruktúra a hlavičky funkcií pre semantický analyzátor
semantic.c - implementácia funkcií pre semantický analyzátor

generator.h - dátová štruktúra a hlavičky funkcií pre generátor kódu
generator.c - implementácia funkcií pre generátor kódu

symbol.h - definícia typov symbolov, definícia dátovej štruktúry pre symboly, hlavičky funkcií relevantné symbolom
symbol.c - implementácia funkcií relevantné pre symboly

symtable.h - dátová štruktúra a hlavičky funkcií pre tabuľku symbolov
symtable.c - implementácia funkcií pre tabuľku symbolov

token.h - definícia typov tokenov, definícia dátovej štruktúry pre tokeny, hlavičky funkcií relevantné tokenom
token.c - implementácia funkcií relevantné pre tokeny

tree.h - definícia typov uzlov, definícia dátovej štruktúry pre strom, hlavičky funkcií relevantné stromu
tree.c - implementácia funkcií relevantné pre strom

utils.h - definícia číselných chybových kódov

Lexikálna analýza

Základný princíp fungovania: Lexikálna analýza bola implementovaná ako konečný automat pomocou rekurzívneho volania funkcií, pričom každá volaná funkcia je ekvivalentná nejakému stavu v diagrame konečného automatu. Jednotlivé prechody sú realizované pomocou čítania nasledujúceho znaku zo štandardného vstupu (príp. porovnania so zoznamom kľúčových slov) a následným zavolaním funkcie stavu, do ktorého daný prechod vedie. V prípade, že automat príde do koncového stavu sa vytvorí token pomocou funkcií určených na vytvorenie tokenu, jednotlivé koncové stavy majú im dedikované konkrétne funkcie. Novo vytvorený token sa následne pridá do interného poľa tokenov a automat prejde do počiatočného stavu.

V prípade, že sa token vytvorí vo finálnom stave identifikátora, tak sa pridá aj do tabuľky symbolov spolu s dostupnou informáciou o čisle riadku, stĺpca a identifikačným číslom rozsahu platnosti.

Chybové stavy: Pri prečítaní znaku, ktorý nepatrí do abecedy konečného automatu alebo pri prečítaní znaku, ktorého prechod z daného stavu nie je definovaný sa nastaví príznak lexikálnej chyby a program sa ukončí s chybovým návratovým kódom.

Prechod k syntaktickej analýze: V prípade korektného spracovania dát zo štandardného vstupu prekladač prejde do fázy syntaktickej analýzy.

Syntaktická analýza

Základný princíp fungovania: Syntaktická analýza sa spúšťa po úspešnom dokončení lexikálnej analýzy a ako prvé dve kontrolované gramatické pravidlá sú pre základnú kostru programu. Po úspešnej kontrole sa prechádza do fázy, kde syntaktický analyzátor žiada tokeny od lexikálneho analyzátora a podľa nich sa rozhoduje o nasledujúcom volanom pravidle**.** Pri volaní pravidla sa v určitých prípadoch priamo vytvárajú až uzly syntaktického stromu a na základe uplatňovaného pravidla sa vytvorí buď terminálny alebo neterminálny uzol.

Pri implementovaní vytvárania stromu AST (resp. Parse tree) sme sa inšpirovali menšími prekladačmi, ktoré preskakujú vytváranie uzlov, ktoré nie sú “dôležité” (neskôr by boli tak či tak odstránené/nahradené) resp. sme takto odľahčili sémantickú analýzu od ich eliminácie. Napríklad pri pravidle expression sa do stromu pripája rovno najvyšší binárny operátor alebo pri pravidle instruction sa neterminálny uzol nevytvorí pre neterminál instruction ale priamo pre nasledujúce pravidlo konkrétnej inštrukcie.

Precedenčná syntaktická analýza bola implementovaná rekurzívnym spracovaním výrazu pomocou troch rekurzívne volaných pravidiel, ktoré sa volajú na základe priority operátorov.

Chybové stavy: Pri vykonávaní syntaktickej analýzy sa v prípade chybového stavu program skončí s dedikovaným návratovým kódom. V prípade, že žiadna chyba nenastane sa posiela AST do sémantickej analýzy.

Sémantická analýza

Základný princíp fungovania: Po úspešnom vykonaní syntaktickej analýzy sa začne prechádzanie stromu pre-order algoritmom. V každom uzle stromu sa najprv skontroluje, čo daný uzol predstavuje.

Uzol v strome môže predstavovať napríklad deklaráciu/redeklaráciu premennej, volanie funkcie, počet argumentov funkcie alebo nejaký výraz. Vo väčšine prípadov na samotné pravidlo môžu nadväzovať ďalšie pravidlá ako napríklad pri volaní vstavanej funkcie môžeme nadviazať pravidlom pre kontrolu počtu a typu argumentov.

V každom uzle stromu sa v určitom poradí skontrolujú sémantické pravidlá a v prípade zhody podstromu s nejakým pravidlom sa presunieme na nasledujúci uzol. Po úspešnej semantickej analýze sa do generátoru kódu posunie ten istý strom, ktorý sémantický analyzátor dostal na kontrolu.

Chybové stavy: Pri nezhode s aspoň jedným pravidlom alebo pri nezhode s očakávaným typom argumentu vstavanej funkcie semantický analyzátor vráti chybu a zastaví sa chod programu.

Poznámka k AST: Keďže sme strom pre generátor kódu spracovávali už v syntaktickom analyzátori, tak ho semantická analýza nijak nemodifikuje, len dbá na jeho semanticky správnu konštrukciu.

Generátor kódu

Základný princíp fungovania: V záverečnej fáze prekladača sa vytvorí pole, ktoré obsahuje všetky globálne premenné a následne sa spustí priamo generátor cieľového kódu bez predošlej generácie vnútornej reprezentácie a optimalizácie.

Generátor kódu pracuje predovšetkým so stromom, kde prechádza neterminálnymi uzlami a podľa typu daného neterminálu sa rozhoduje o následných volaniach funkcii. Po zavolaní funkcie môže daná funkcia napríklad volať pomocné funkcie (konverzia z desatinného čísla na celé číslo) alebo sa pozrieť do tabuľky symbolov (napríklad v prípade rozlišovania getteru a setteru) ale predovšetkým funkcie generujú na štandardný výstup jednotlivé inštrukcie kódu IFJcode25. V prípade, že nasledujúci uzol je terminál sa zavolá funkcia určená na jeho vygenerovanie.

Poznámka k problémom pri implementácii generátora: Jeden z hlavných problémov bola implementácia deklarácie parametrov pre vnorené bloky kódu v tele funkcie. Problém bol vyriešený pomocou pomocnej funkcie, ktorá rozdeľuje lokákne premenné od parametrov volanej funkcie a zabezpečuje ich dostupnosť vrámci celého telo danej funkcie.

Tabuľka symbolov

Tabuľka symbolov z hľadiska lexeru: S tabuľkou symbolov začíname pracovať už v lexikálnej analýze, kde z tokenov typu identifikátor a globálna premenná vytvárame symboly pomocou rozhrania pre lexikálny analyzátor spolu s dostupnými informáciami napríklad o riadku tokenu, stĺpci a rozsahu platnosti. Novo vytvorené symboly vložíme do tabuľky symbolov pomocou tomu vytvorenej funkcie, ktorá zahŕňa vytváranie hashu a indexu pre daný symbol.

Tabuľka symbolov z hľadiska parsera: Neskôr v syntaktickej analýze sa tabuľka symbolov dopĺňa informáciami ako je napríklad konkrétny typ identifikátora (variable, function, getter …), riadok deklarácie a prípadné parametre funkcie.

Vyhľadávanie v tabuľke symbolov: Tabuľka symbolov vyhľadáva identifikátory na základe lexémy symbolu a na zákalde určeného rozsahu platnosti. V prípade pridávania symbolu na index, na ktorom sa už nejaký iný symbol nachádza sa začne lineárne vyhľadávanie najbližšieho voľného indexu v tabuľke.

Poznámka: Tabuľka symbolov je implementovaná ako nafukovacie pole.

Strom

Strom z hľadiska syntaktickej analýzy: Pri inicializácií stromu vzniká koreňový uzol a postupom syntaktickej analýzy sa doň pridávajú uzly, pričom každý uzol predstavuje buď použité gramatické pravidlo relevantné abstraktnému syntaktickému stromu alebo teminál.

Strom z hľadiska semantickej analýzy: Semantický analyzátor prechádza strom preorder algoritmom a na základe typu uzlu sa začne kontrola semantických pravidiel pre daný typ uzlu.

Strom z hľadiska generátoru kódu: Generátor kódu prechádza strom preorder algoritmom a na základe typu uzlu sa začne generácia kódu na základe typu uzlu.

Vývojový cyklus a spôsob práce v tíme

Prekladač bol vyvíjaný vodopádovým modelom s občasnými prezenčnými alebo online stretnutiami (Discord). Počas vývoja prekladača bola často využívaná metóda párového programovania a debugovania. Pri vývoji bol použitý verzovací systém Github a boli určené formálne normy pre písanie kódu ako napríklad typ pomenovávania funkcií a premenných (snake case) a spôsob odriadkovania.