Vysoké učení technické v Brně
Fakulta Informačních technologií

IFJ projekt

Tým xnecas31

varianta TRP-izp

Rozšíření - FUNEXP

Jan Juračka	(xjurac07)	25%	
Jakub Janšta	(xjanst02)	25%	
Petr Nečas	(xnecas31)	25%	**vedoucí**
Jakub Slanina	(xslani12)	25%	

BRNO 2023

Návrh

Implementaci překladače jsme zahájili návrhem FSM, vypracováním lexikální analýzy a definováním možných tokenů které budou zpracovány v dalších krocích implementace.

Následovala příprava a zpracování syntaktické analýzy. Současně se realizovala první možná verze tabulky symbolů.

Ve finální části projektu se zpracovala sémantika, která následovala dokončením generace kódu.

Implementace

V kapitole Implementace se nachází jednotlivé podkapitoly jednotlivých částí implementace překladače. V této části dokumentace se také nachází informace o implementaci jednotlivých struktur použitých při implementaci projektu.

Tabulka Symbolů

Tabulka symbolů byla dle zadaní implementovaná pomocí tabulky s rozptýlenými položkami **s implicitním zřetězením položek (TRP-izp)**. Implementace se nachází v souboru **symtable.c** s hlavičkovým souborem **symtable.h**.

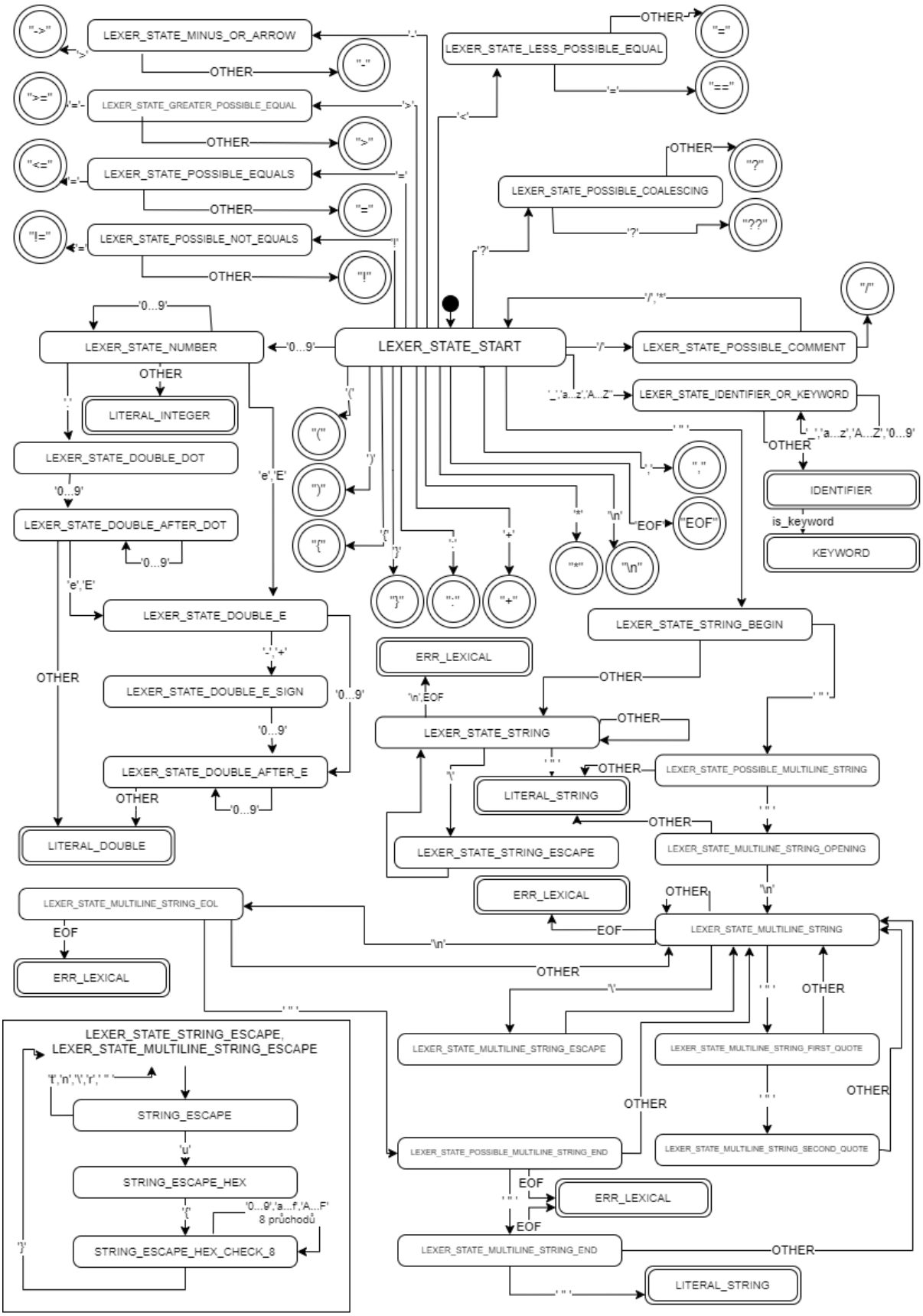
Lexikální analýza

Cílem lexikálního analyzátoru je zpracovat jednotlivé lexémy a vracet je jako zpracované tokeny. V případě, že token není jednoznačný (identifikátor, řetězec, desetinné a celé číslo), tak je jeho hodnota dostupná v pomocné struktuře `BufferString`, implementované v souborech **buffer_string.c** a **buffer_string.h** . Seznam všech možných tokenů lze nalézt v hlavičkovém souboru **lexer.h**.

Samotný lexer je vypracovaný podle předem navrhnutého, dle potřeby upraveného FSM návrhu. Seznam stavů implementovaného FSM je také definovaný v souboru **lexer.h**. Celková implementace analyzátoru se nachází v souboru **lexer.c**.

Pokud v návrhu není definován přechod pro nějaký vstup, tak implicitně spadá do chybového stavu a vrací token značící patřičnou chybu. Vyjímka je konečný stav identifier ve kterém pouze proběhne jednoduchá kontrola, jestli se jedná o klíčové slovo.

V návrhu se nachází další menší vnitřní automat, zobrazen v levém dolním rohu, který řeší escape sekvence řetězců.


LEXER_STATE_START
LEXER_STATE_MINUS_OR_ARROW
"->"
"-"
LEXER_STATE_GREATER_POSSIBLE_EQUAL
">="
">"
LEXER_STATE_POSSIBLE_EQUALS
"<="
"="
LEXER_STATE_POSSIBLE_NOT_EQUALS
"!="
"!"
LEXER_STATE_LESS_POSSIBLE_EQUAL
"=="
LEXER_STATE_POSSIBLE_COALESCING
"??"
"?"
LEXER_STATE_POSSIBLE_COMMENT
"/"
LEXER_STATE_NUMBER
LITERAL_INTEGER
LEXER_STATE_DOUBLE_DOT
LEXER_STATE_DOUBLE_AFTER_DOT
LEXER_STATE_DOUBLE_E
LEXER_STATE_DOUBLE_E_SIGN
LEXER_STATE_DOUBLE_AFTER_E
LITERAL_DOUBLE
LEXER_STATE_IDENTIFIER_OR_KEYWORD
IDENTIFIER
is_keyword
KEYWORD
"("
")"
"{"
"}"
"-"
"+"
"*"
"\n"
"EOF"
","
ERR_LEXICAL
LEXER_STATE_STRING_BEGIN
LEXER_STATE_STRING
LITERAL_STRING
LEXER_STATE_STRING_ESCAPE
LEXER_STATE_POSSIBLE_MULTILINE_STRING
LEXER_STATE_MULTILINE_STRING_OPENING
LEXER_STATE_MULTILINE_STRING
LEXER_STATE_MULTILINE_STRING_EOL
LEXER_STATE_MULTILINE_STRING_ESCAPE
LEXER_STATE_MULTILINE_STRING_FIRST_QUOTE
LEXER_STATE_MULTILINE_STRING_SECOND_QUOTE
LEXER_STATE_POSSIBLE_MULTILINE_STRING_END
LEXER_STATE_MULTILINE_STRING_END
LEXER_STATE_STRING_ESCAPE, LEXER_STATE_MULTILINE_STRING_ESCAPE
STRING_ESCAPE
STRING_ESCAPE_HEX
STRING_ESCAPE_HEX_CHECK_8
8 průchodů
OTHER
EOF

Syntaktická analýza

Implementovaná v **syntax_ll.c** a **syntax_precedent.c**. Dle zadání projektu jsou přiložena LL pravidla, podle kterých syntaktická analýza vyhodnocuje přicházející tokeny.

Během syntaktické analýzy se také generuje abstraktní syntaktický strom, který slouží jako vstup pro další části překladače.

Analýza kódu metodou rekurzivního sestupu

Celá syntaktická analýza, řízená LL gramatikou, je implementována pomocí rekurzivního sestupu. Výhodou této metody je její jednoduchost, nicméně může nastat přetečení zásobníku při její chybné implementaci, nebo dostatečně dlouhém vstupním programu, protože celý program je zpracován jako jeden celek. V LL gramatice jsou části, které jsou zpracovány precedenční analýzou značeny, jako **$EXPRESSION**, kdy po jejich vyhodnocení se přechází zpět na syntaktickou analýzu řízenou LL gramatikou.

Analýza výrazů precedenční metodou

Soubory **syntax_precedent.c** a **syntax_precedent.h**. Analýza pracuje se třemi zásobníky. `tokenStack` uchovává tokeny předávané lex. analyzátorem. Jeho obsah řídí operace algoritmu analýzy. `valueStack` uchovává hodnoty literálů a názvy proměnných. `nodeStack` obsahuje uzly výrazu. Precedenční tabulka je v kódu implementovaná funkcí `precedent_table`. Index na tabulce dává tokenům `token2index`. Hodnoty **4** a **5** na tabulce značí stavy, které řeší různé praktické problémy (volání funkcí, absence ε-pravidel). Při zpracování funkcí ve výrazu se na chvíli “vyskočí ven” z prec. analýzy, funkce se zpracuje ll-gramatikou (argumenty zpracovává prec. analýza) a pokud je úspěšná, vrátí se zpět do prec. analýzy.

Algoritmus se rozhoduje kde výraz končí na základě tokenů od lex. analyzátoru. Může jít o různé tokeny (např. “{”, “}”,...), včetně těch, kterými začínají nové příkazy (let, var, if,...). Pro tyto tokeny má prec. tabulka symbol **$**. Ale i ostatní tokeny v tabulce, jinak běžně se vyskytující ve výrazu, ho mohou někdy ukončovat (“)”, “!”, …). Tyto případy (oranžová políčka v tabulce) algoritmus řeší tak, že token nahradí ukončovacím tokenem `PRECEDENT_END` aby zbytek analýzy mohl proběhnout v pořádku. Například v takovémto kódu:

```
let x: Int = 1 + 2
x = x + 2
// prec. analýza ví, že výraz končí číslicí 2, protože po ní
následuje identifikátor “x”
```

LL-gramatika

```
<program> -> EOF
<program> -> EOL <program>
<program> -> <statements> <program>
<program> -> <func_definition> <program>

<type> -> Int
<type> -> Int?
<type> -> Double
<type> -> Double?
<type> -> String
<type> -> String?

<statements> -> <statement> <statements>
<statements> -> EOL <statements>
<statements> -> ε

<statement> -> <var_declaration>
<statement> -> <let_declaration>
<statement> -> <if>
<statement> -> <while>
<statement> -> <return>
<statement> -> <assign>
<statement> -> <func_call>

<let_var_declaration> -> $ID = $EXPRESSION
<let_var_declaration> -> $ID : <type> = $EXPRESSION
<let_var_declaration> -> $ID : <type>


<var_declaration> -> var <let_var_declaration>
<let_declaration> -> let <let_var_declaration>

<if> -> if $EXPRESSION { <statements> } <else>
<if> -> if let $ID { <statements> } <else>
<else> -> else { <statements> }

<while> -> while $EXPRESSION { <statements> }

<return> -> return
<return> -> return $EXPRESSION

<assign> -> $ID = $EXPRESSION

<func_call> -> $ID ( <func_call_args> )
<func_call_arg_without_name> -> $EXPRESSION
<func_call_arg_with_name> -> $ID : $EXPRESSION
<func_call_arg> -> <func_call_arg_without_name>
<func_call_arg> -> <func_call_arg_with_name>
<func_call_args> -> ε
<func_call_args> -> <func_call_arg>
<func_call_args> -> <func_call_arg> , <func_call_args>

<func_definition> -> <func_head> <func_body>

<func_head> -> func $ID ( <func_head_args> ) -> <type>
<func_head> -> func $ID ( <func_head_args> )

<func_head_arg> -> $ID $ID : <type>
<func_head_args> -> ε
<func_head_args> -> <func_head_arg>
<func_head_args> -> <func_head_arg> , <func_head_args>

<func_body> -> { <statements> }
```

LL-tabulka

	EOL	EOF	#ide	nil	Int	Dou	Strin	var	let	if	else	whil	func	retu	{	}	(	)	<	>	<=	>=	==	!=	+	-	*	/	#int	#dou	#str	!	=	?	??	,	:	->
<prog>	x	x	x					x	x	x		x	x	x															x	x	x							
<type>					x	x	x																														
<statements>	x		x					x	x	x		x	ε	x		ε													x	x	x							
<statement>			x					x	x	x		x		x															x	x	x							
<let_var_declaration>			x																																		
<let_declaration>									x																													
<var_declaration>								x																														
<if>										x																												
<else>											x																											
<while>												x																										
<return>														x																								
<assign>			x																																		
<func_call>			x																																		
<func_call_arg_without_name>			x														x												x	x	x							
<func_call_arg_with_name>			x																																		
<func_call_arg>			x														x												x	x	x							
<func_call_args>			x														x	ε											x	x	x							
<func_definition>													x																									
<func_head>													x																									
<func_head_arg>			x																																		
<func_head_args>			x															ε																			
<func_body>															x																						

Precedenční tabulka

		0	1	2	3	4	5		6		7						8	9
		id	lit	(	)	!	*	/	+	-	==	!=	<	>	<=	>=	??	$
0	id			f	>	>	>	>	>	>	>	>	>	>	>	>	>	>
1	lit				>	>	>	>	>	>	>	>	>	>	>	>	>	>
2	(	<	<	<	=	<	<	<	<	<	<	<	<	<	<	<	<	
3	)				>	>	>	>	>	>	>	>	>	>	>	>	>	>
4	!				>	>	>	>	>	>	>	>	>	>	>	>	>	>
5	*	<	<	<	>	<	>	>	>	>	>	>	>	>	>	>	>	>
	/	<	<	<	>	<	>	>	>	>	>	>	>	>	>	>	>	>
6	+	<	<	<	>	<	<	<	>	>	>	>	>	>	>	>	>	>
	-	<	<	<	>	<	<	<	>	>	>	>	>	>	>	>	>	>
7	==	<	<	<	>	<	<	<	<	<	>	>	>	>	>	>	>	>
	!=	<	<	<	>	<	<	<	<	<	>	>	>	>	>	>	>	>
	<	<	<	<	>	<	<	<	<	<	>	>	>	>	>	>	>	>
	>	<	<	<	>	<	<	<	<	<	>	>	>	>	>	>	>	>
	<=	<	<	<	>	<	<	<	<	<	>	>	>	>	>	>	>	>
	>=	<	<	<	>	<	<	<	<	<	>	>	>	>	>	>	>	>
8	??	<	<	<	>	<	<	<	<	<	<	<	<	<	<	<	>	>
9	$	<	<	<		<	<	<	<	<	<	<	<	<	<	<	<	

Sémantická analýza

Implementovaná v souboru **semantic.c** a **semantic_expression.c**. Sémantická analýza zpracovává syntaxí generovaný abstraktní syntaktický strom tím způsobem, že tabulky symbolů vloží symbol s funkcí obsahující informace jako název, typ návratové hodnoty a vstupní argumenty funkce. Dále zpracovává tělo programu a pokud narazí na proměnnou, vloží ji do tabulky, která je později předána generátoru kódu. Po té se přejde na sémantickou kontrolu všech příkazů, výrazů a volání funkcí. Funkce jsou kontrolovány při každém jejich volání, aby se předešlo volání neinicializovaných globálních proměnných.

Proměnné jsou dále přejmenovány v abstraktním stromě tak, aby vyhovovaly generátoru kódu.

Generátor kódu

Implementován v souboru **generator.c**. Generace kódu pro interpret byla vytvořena jako poslední podle původního návrhu. Ke generaci se taky váží soubory **generator.h**, hlavičkový soubor, **generator_built_in.h**, ve kterém jsou definované vestavěné funkce. V souboru **generator_pre_defined.h** jsou definovány stavební bloky, jako třeba hlavička souboru, podmínky, cykly, nebo složitější operátory.

Generátor generuje cílový kód přímo z abstraktního syntaktického stromu, který je zároveň využit jako mezikód, který sémantická analýza upravuje. Původní plán zahrnoval i kódový optimalizátor, ale kvůli nedostatku časových zdrojů je kód generován přímo.

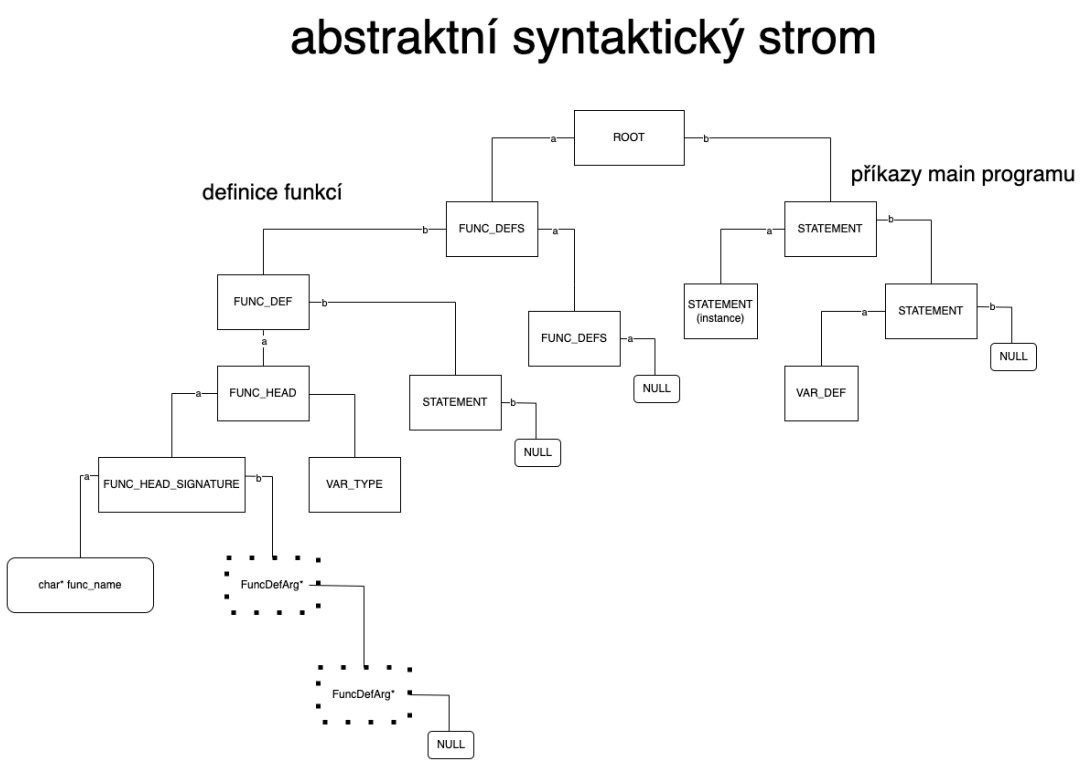
Datové struktury

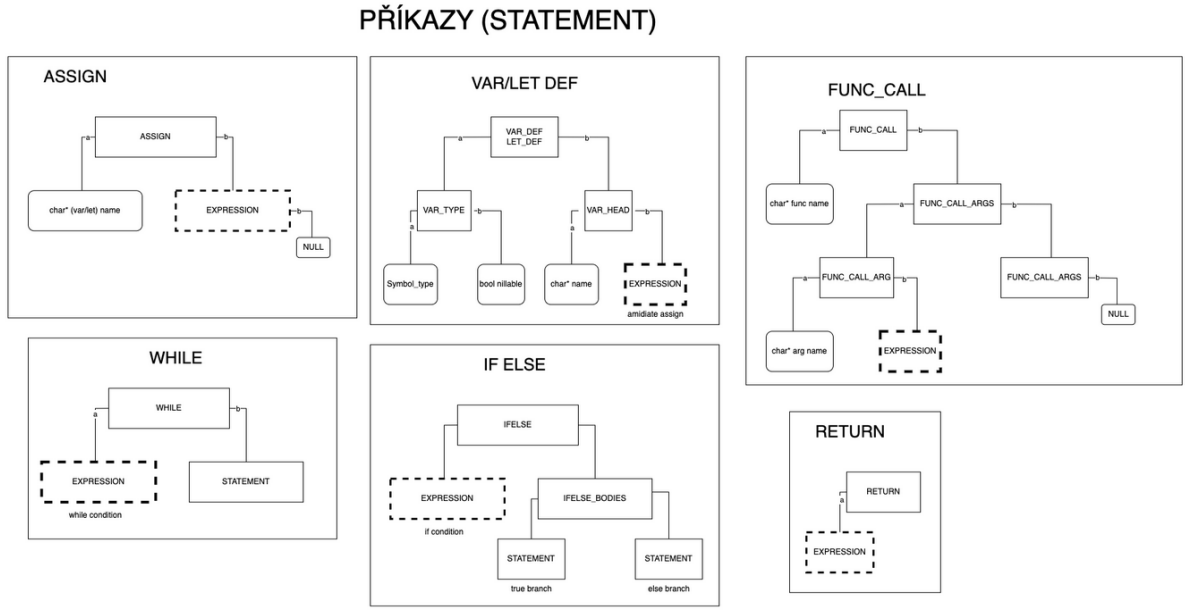
Struktura `buffer_string` funguje jako pomocná struktura pro práci se stringy, a zpracování tokenů (předávání). Definice se nachází v souboru **buffer_string.h** a je implementován v souboru **buffer_string.c**.

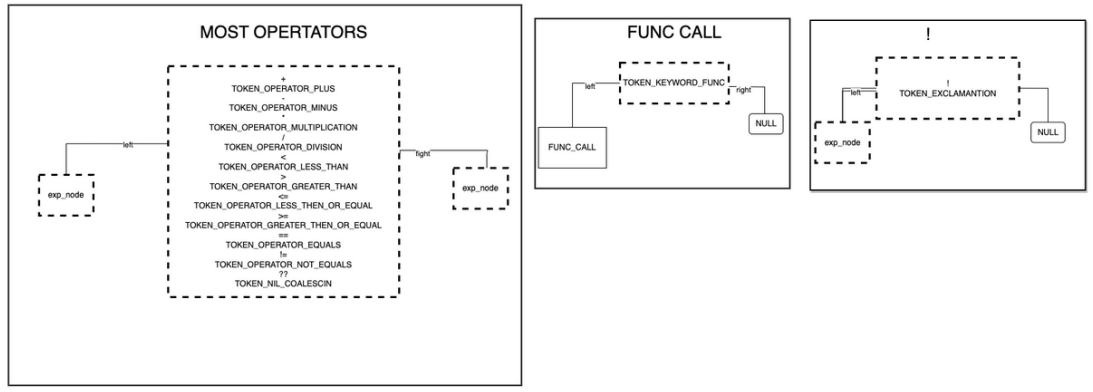
Struktury pro práci s tabulkou symbolů jsou definovány v **symtable.h**. Struktura `Symbol` slouží pro definování unikátních symbolů v kódu, které budou ukládány do specifických tabulek. Struktura `SymTable` slouží pro evidenci symbolů, pro dané rozsahy funkcí.

Struktury v souboru **ast.h**. V souboru je definovaná struktura syntaktického stromu ast, `ast_node`, do kterého je generován syntaxní strom který je dále zpracován v sémantickém analyzátoru a využít pro generování kódu. Strom výrazů, `exp_node`, je součástí struktury ast a ukládá specifické výrazy příkazů. Jednoduchý seznam `FuncDefArg`, ten slouží pro ukládání argumentů funkcí. Pro lepší orientaci je přiložena ilustrace struktur.

Ilustrace struktur







Práce v týmu

Rozdělení práce

Tabulka symbolů	- Petr Nečas
Lexikální analýza	- Jakub Slanina, Jakub Janšta
Syntaktická analýza	- Jakub Janšta, Jan Juračka
Sémantická analýza	- Petr Nečas, Jakub Janšta
Generátor kódu	- Jakub Janšta, Jakub Slanina

Použité nástroje

git,
Notion,
Office 360,
Google Docs

Závěr

Projekt nás zaskočil svou složitostí, kterou jsme poznávali více a více s blížícím se datem odevzdání. Zároveň jsme implementovali velké množství naivních řešení, takže se některé části celé přepisovali (např.: syntaktická analýza byla přepsána 3x) a nebo jsme měli na první pohled skvělé nápady, které se projevily jako zcela nepraktické (např.: vyhledávání v tabulce symbolů podle identifikátoru a bloku platnosti).

I přes všechny náročností rozšíření FUNEXP vyšlo jako vcelku spolehlivé, i když jako krajní případ nedovoluje sémantický správné volání funkce jako argument funkce s lokálním argumentem volající funkce. Je to výsledkem limitace paměťového modelu rámců, kterou jsme nedokázali efektivně obejít a skončí chybou interpretu.