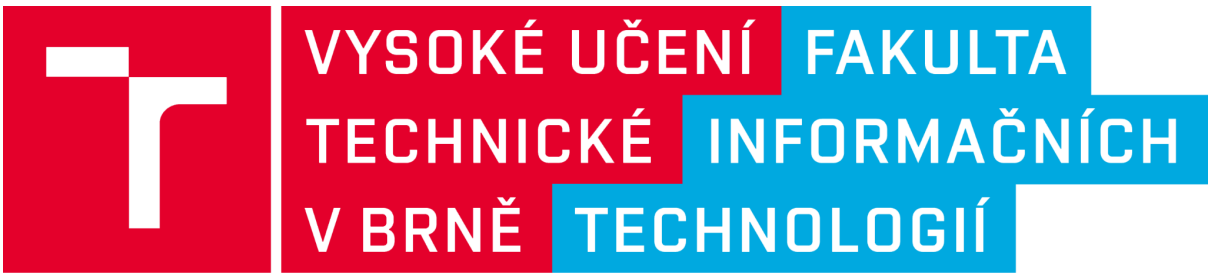


Projektová dokumentace

Implementace překladače imperativního jazyka IFJ22

Tým xkralr06, varianta TRP

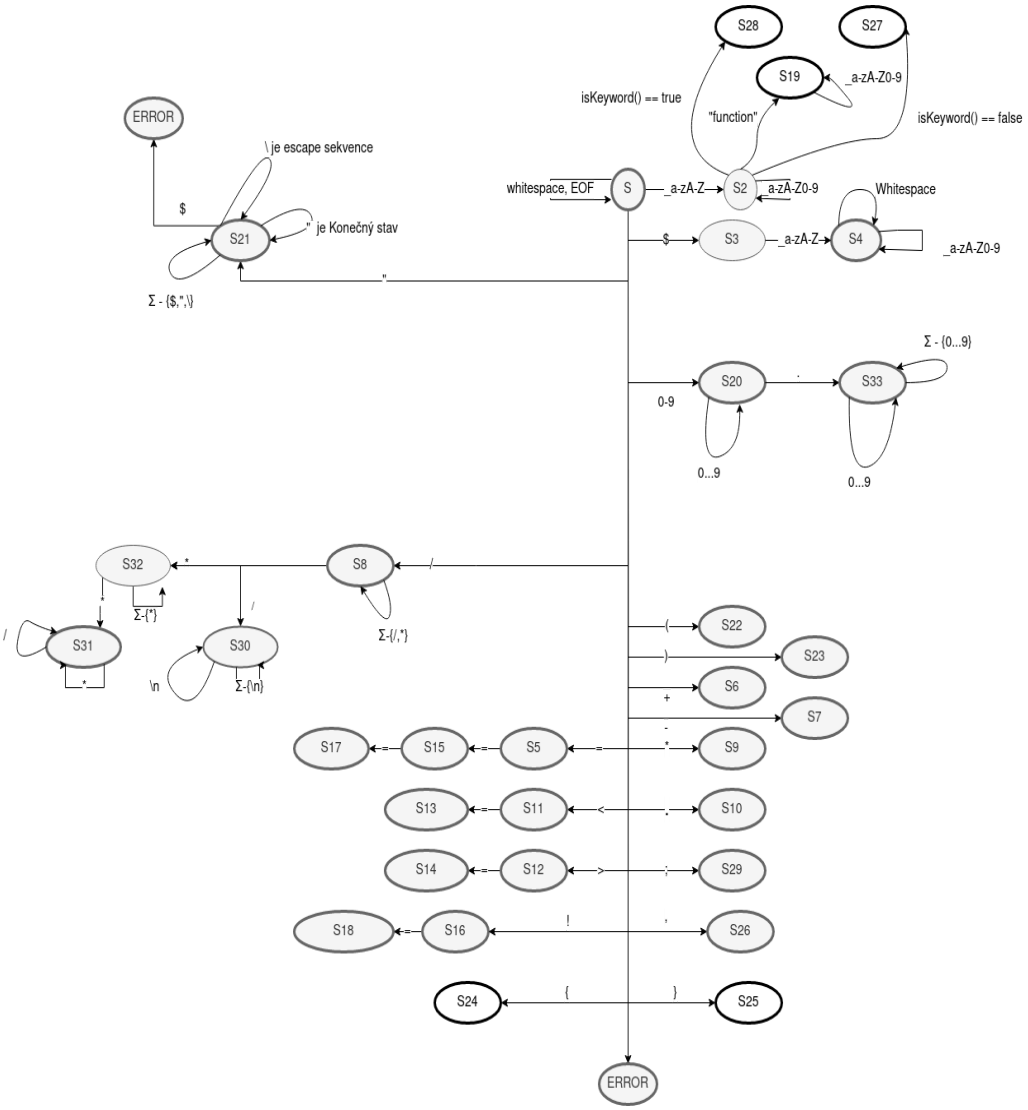
Řešitelé:

07.12.2022
xkralr06 - Rostislav Král (Vedoucí) - 20%
xnguye22 - Hoang Nam Nguyen - 25%
xsnieh00 - Nikita Sniehovskyi - 20%
xjezek19 - Lukáš Ježek - 35%

1 Lexikální analýza

1.1 Diagram

V diagramu jsou konečné přechody vyjádřeny křivkami(nerovné šipky) a konečné stavy dvojitými okraji stavu.



ST_START => S
ST_READ => S2
ST_VAR_PREFIX => S3
ST_VAR => S4
ST_OP_ASSIGN => S5
ST_OP_PLUS => S6
ST_OP_MINUS = S7
ST_OP_DIVIDE => S8
ST_OP_MULTIPLY => S9
ST_OP_CONCAT => S10
ST_OP_LESSER_THAN => S11
ST_OP_GREATER_THAN => S12
ST_OP_LESS_EQUAL => S13
ST_OP_GREATER_EQUAL => S14
ST_OP_EQUALS => S15
ST_OP_NOT_EQUAL => S16
ST_OP_TYPE_EQUALS => S17
ST_OP_TYPE_NOT_EQUALS => S18
ST_FUNC => S19
ST_INT_LITERAL => S20
ST_STRING_LITERAL => S21
ST_LEFT_PARENTHESES => S22
ST_RIGHT_PARENTHESES => S23
ST_LEFT_CURLYBRACKET => S24
ST_RIGHT_CURLYBRACKET => S25
ST_COMMA => S26
ST_KEYWORDS => S27
ST_FUNC_CALL => S28
ST_SEMICOLON => S29
ST_SLINE_COMMENT => S30
ST_MLINE_COMMENT _END => S31
ST_MLINE_COMMENT => S32
ST_DOUBLE_LITERAL => S33

1.2 Popis lexikálního analyzátoru

Lexikální analýza je část, která pracuje se vstupním souborem (načítaný ze standardního vstupu) a jeho hlavní funkcionalita se nachází v funkci `TOKEN_T* get_next_token()` a je realizovaná přes konstrukci jazyka C - `switch`, kde jednotlivé case statementy znamenají nějaký stav .Vytvořili jsme funkci `char* lexer_fget()`, která čte ze standardního vstupu a počítá na jakém řádku a znaku se nachází daný lexém/token, který je následně přijat konečným automatem, přičemž může dojít k přechodu viz. Diagram výše. Již zmíněná funkce

vrací tokeny podle přijatých znaků do Syntaktického analyzátoru, kde je token dál vyhodnocován.

2. Syntaktická a sémantická analýza

Jak syntaktickou, tak sémantickou analýzu jsme implementovali v souboru `syntax_and_semantic_analyzer.c`. Pro syntaktickou analýzu jsme si zvolili prediktivní postup. Tento způsob nám umožňoval lepší reakce na změny gramatiky a větší možnosti modifikace v průběhu vývoje. Syntaktická analýza se řídí LL pravidly popsanou níže.

Syntaktická analýza si volá postupně tokeny pomocí funkce `TOKEN_T* get_next_token()`, která je implementovaná v lexer.c, tato funkce načítá tokeny ze zdrojového souboru a zároveň provádí lexikální analýzu tokenu. Jedinou výjimkou je v tomto ohledu zpracování výrazů, kde syntaktickou a sémantickou kontrolu provádíme ve funkci `BSTnode* analyze_precedence(DLList* list)`. Do ní vkládáme seznam reprezentující výraz, každá položka seznamu odkazuje na `TOKEN_T`. Syntaktická a sémantická analýza pracuje se strukturou `TOKEN_T*`, která je určena pro práci s tokeny. Dále obsahuje pomocnou strukturu `parseFunctionHelper functionHelper`, ta se používá při deklaraci funkcí pro kontrolu názvu, parametrů a návratového datového typu. Další pomocnou strukturou je scope, ta se používá pro kontrolu otevřených závorek, a zároveň je v proměnné `scope.num` uložena informace o aktuálním scopu pro proměnné a funkce.

2.1 Zpracování výrazů pomocí precedenční analýzy

Zpracování výrazu se provádí v souboru expressions.c. K tomu jsou vytvořeny pomocné struktury v souboru `data_structures.c`, jako `BSTnode` (binární vyhledávací strom), `Stack` (zásobník) a `DLList` (dvousměrně vázaný lineární seznam). Hlavní funkcí je `BSTnode* analyze_precedence(DLList* list)`. Bere si jako parametr seznam tokenů, který reprezentuje nezpracovaný výraz. V této funkci se pomocí predecenční tabulky určuje, jak se má výraz dále zpracovávat. Výraz se podle LL pravidel upravuje, pravidla lze najít níže (Seznam LL pravidel). Pokud je výraz syntakticky nebo sématicky špatně, vrátí funkce patřičný `exit code`. Pokud je správně, vrátí funkce `BSTnode*` strom reprezentující výraz a způsob, jak se má zpracovat.

2.2 Seznam LL pravidel pro redukci výrazů

```
E -> E  *  E        E -> E  >  E
E -> E  /  E        E -> E  <= E
E -> E  +  E        E -> E  -  E
E -> E  .  E        E -> E  <  E
E -> E  >= E        E -> E === E
E -> E !== E        E -> ID
```

```
E -> (E)
```

2.3 Precedenční tabulka - redukční pravidla

X	null	*	/	+	-	.	<	>	<=	>=	=	!=	===	!==	(	)	ID	$
NULL																		
*		>	>	>	>	>	>	>	>	>			>	>	<	>	<	>
/		<	<	>	>	>	>	>	>	>			>	>	<	>	<	>
+		<	<	>	>	>	>	>	>	>			>	>	<	>	<	>
-		>	>	>	>	>	>	>	>	>			>	>	<	>	<	>
.		>	>	>	>	>	>	>	>	>			>	>	<	>	<	>
<		<	<	<	<	<									<	>	<	>
>		<	<	<	<	<									<	>	<	>
<=		<	<	<	<	<									<	>	<	>
>=		<	<	<	<	<									<	>	<	>
=																		
!=																		
===		<	<	<	<	<									<		<	>
!==		<	<	<	<	<									<		<	>
(		<	<	<	<	<	<	<	<	<			<	<	<	=	<	
)		>	>	>	>	>	>	>	>	>			>	>		>		>
ID		>	>	>	>	>	>	>	>	>			>	>		>		>
$		<	<	<	<	<	<	<	<	<			<	<	<		<	

LEGENDA: Prázdná buňka = chyba

2.4 LL gramatika

1. <start> -> PROG_START -> <statement> -> DECLARE_STRICT_TYPES
2. <statement> ->
3. <func_definition> -> ID_NAME -> <function_param>
4. <function_params> -> ***e***

5. <function_param> -> ID -> <function_param>
6. <function_param> -> COMMA -> <function_param>
7. <function_param> -> RPAR -> <func_return_type>
8. <func_return_type> -> ID

3. Algoritmy a datové struktury

3.1 Tabulka symbolů - Tabulka s rozptýlenými položkami

Soubory `symtable.c` a `symtable.h`

Tabulka symbolů je implementována pomocí hashovací tabulky. Každá položka tabulky je jednostranně vázaný lineární seznam, který obsahuje všechny položky se stejným indexem po hashování klíče.

Jako unikátní klíč pro proměnné a funkce jsme používali jejich jména. Navíc u proměnných je záznam o jejich oblasti viditelnosti (scope), pomocí které se dá vytvářet "rámec", který umožňuje překrytí proměnných se stejným jménem.

Každá položka s proměnnou obsahuje její jméno, typ, obor viditelnosti.
Každá položka s funkcí obsahuje její jméno, návratovou hodnotu a seznam typů parametrů.

Aby hashovací tabulka zůstala hashovací tabulkou i když bude mít hodně záznamů, implementovali jsme automatickou funkci resize, která zvětší tabulku při přetížení.

Další pomocné funkce hashovací tabulky:

- `htab_find_var` Hledání proměnné
- `htab_insert_var` Vložení proměnné
- `htab_find_function` Hledání funkce
- `htab_insert_function` Vložení funkce
- `htab_remove_scope` Odstranění rámce

3.2 Dynamický řetězec

Soubory `dyn_string.c` a `dyn_string.h`

Abychom mohli analyzovat tokeny neznámé délky, vytvořili jsme strukturu dynamického řetězce, která obsahuje délku a ukazatel na řetězec.

Funkce `str_conc` připojuje nový řetězec na konec daného a alokuje větší paměť, pokud je potřeba.

3.3 Dvojitě vázaný lineární seznam

Jako pomocnou strukturu pro precedenční analýzu, syntaktickou a sémantickou kontrolu parametrů funkce, a kontrolu otevřených závorek jsme zvolili dvojitě vázaný lineární seznam (Dále jen DLL) deklarovaný v souboru `data_structures.c`. DLL je popsaný strukturou `DLList`, ta obsahuje odkaz na první a poslední prvek seznamu a také počet prvků uložených v seznamu, jeho položky jsou popsány strukturou DLLItem, každá položka obsahuje `TOKEN_T *token`, odkaz na další a předchozí prvek seznamu, typ uloženého tokenu `enum T_TOKEN_T`YPE tokenType a pokud je token operátor, pak i `enum T_OPERATOR tokenOperator`. Pro práci se seznamem používáme funkce ze souboru `data_structures.c`. Inicializaci provádíme funkcí `DLL_init(DLList* list)`. Pro vložení prvků se používá převážně `DLLItem* DLL_insert_first/last/after(DLList* list, TOKEN_T* token)`, pro vybrání prvků se používají funkce `DLLItem* DLL_pop_first/last(DLList *list)`. Pomocná funkce `bool DLL_find_token(DLList* list, TOKEN_T *token)` vrací hodnotu `true`, pokud najde zaslaný token v daném seznamu, jinak vrací `false`.

3.4 Zásobník

Jako další pomocnou strukturu jsme použili zásobník. Zásobník sloužil pro ukládání zarážek analýzu výrazů, proto každá položka zásobníku ukazuje na položku `DLLItem` . Jinak struktura funguje jako klasický zásobník.

3.5 Binární vyhledávací strom

Binární vyhledávací strom je rekurzivní struktura reprezentující výraz a určující pořadí, ve kterém se mají operace v něm obsažené vykonávat. Každý uzel stromu odkazuje na `TOKEN_T`, který může být typu `OPERATOR`, `LITERAL` nebo `ID` (idetifier). Obsahuje taky `type`, který určuje návratový typ výrazu či podvýrazu. Závorky se ve stromu již nevyskytují. Pokud je v `root` uzlu `TOKEN_T` typu `OPERATOR`, jeho 2 operandy jsou jeho levý a pravý podstrom (resp. jeho `TOKEN_T`). Operace ve stromu se provádí od listového uzlu. Výsledný datový typ operace je uložen v `BSTnode->type`.

4. Práce v týmu

4.1 Způsob práce

Projekt jsme začali řešit na konci října.

4.2 Komunikace

Hlavní komunikační nástroj jsme používali aplikaci Discord, Facebook Messenger nebo klasické telefonní číslo. Discord nám umožnil vytvářet vlákna pro jednotlivé části projektu a tím si udržet celkový přehled o projektu. Ostatní platformy jsme používali primárně pro domlouvání skupinových setkání na platformě Discord.

4.3 Verzovací systém

Pro verzování a sdílení kódu jsme použili verzovací systém Git, jako vzdálený repozitář jsme použili službu GitHub. Pro nové funkcionality jsme používali větve (branche), které jsme poté spojovali příkazem merge do hlavní větve (master).

4.4 Rozdělení práce

- Lukáš Ježek - Syntaktická a sémantická analýza
- Nikita Sniehovskyi - Tabulka symbolů, dynamické řetězce, generování kódu
- Hoang Nam Nguyen - precedenční analýza
- Rostislav Král - Lexikální analýza

5. Závěr

Je to bomba

Použitá literatura

Studijní materiály předmětu Jazyky a formální překladače (záznamy z přednášek a prezentace)
Studijní materiály předmětu Algoritmy