Projektová dokumentace

Implementace překladače imperativního jazyka IFJ21

2021/22

Tým 131 - varianta I

Petr Hýbl (xhyblp01) 30%

Alexandr Čelakovský (xcelak00) 30%

Martin Zmitko (xzmitk01) 40%

Dominik Klon (xklond00) 0%

8.12.2021

Obsah:

1. Úvod

Cílem projektu bylo vytvořit program v jazyce C, který přeloží imperativní jazyk IFJ21. Jazyk IFJ21 je podmnožina jazyka Teal. Vybrali jsme si první variantu projektu – implementaci tabulky symbolů pomocí binárního vyhledávacího stromu.

2. Implementace

Náš zdrojový kód se skládá z několika na sobě závislých částí.

2.1 Lexikální analýza

Lexikální analýza je implementovaná samostatně v souboru *scanner.c.* Lexikální analýzu jsme začali implementovat jako úplně první. Hlavní funkce, která řídí celý lexikální analyzátor se nazývá *get_token.* Tato funkce čte jednotlivé znaky ze vstupu stdin a převádí je na strukturu token. Tato struktura si uchovává tyto informace:

- Typ tokenu jako jeden typ z výčtu TokenType
- Klíčové slovo, pokud je typ tokenu cokoliv jiného, než klíčové slovo hodnota není definována
- Obsah tokenu, pokud je typ tokenu string, identifikátor nebo klíčové slovo.
- Celé číslo tokenu, pokud je typ tokenu integer
- Reálné číslo tokenu, pokud je typ tokenu number
- Řádek, na kterém se token nachází

Lexikální analyzátor je vytvořený na základě našeho návrhu konečného deterministického automatu. Automat je implementován jako opakující se switch. Názvy stavů jsou uložené v proměnné *state.* Při čtení jednotlivých znaků automat prochází stavy. Při nalezení nepovolené sekvence znaku IFJ21 se automat ukončí chybou, jinak vrátí validně načtený token k dalšímu zpracování.

Pokud automat narazí na escape sekvenci ve stringu s ASCII hodnotou, vytvoří se pro ni samostatné pole znaků s délkou 3 do kterého se znaky načtou a pomocí přetypování se převedou na znak. Pokud je ASCII hodnota jiná než číslice 0-9 se automat ukončí chybou.

Pro implementaci bylo potřeba vytvořit další funkce, hlavně pro práci s obsahem tokenu. Funkce *token_data_init* inicializuje prázdné pole stringů a funkce *token_data_append* zajišťuje vkládání jednoho znaku na konec řetězce. Pomocí funkce *is_keyword* detekujeme, zda se jedná o klíčové slovo nebo nikoliv. Ta porovnává aktuální string s klíčovými hodnotami. Díky funkci *get_char_type* můžeme snadno rozhodnout do jaké skupiny znaků náš aktuální znak zařadit. Informuje nás, zda se jedná o číslo, písmeno abecedy nebo ostatní.

2.2 Syntaktická analýza

Syntaktická analýza je implementována metodou rekurzivního sestupu. Pro tuto metodu jsme vytvořili LL-gramatiku. Celá syntaktická analýza se nachází v souboru parser.c. Pro každý neterminál existuje funkce, která dané pravidlo výkonná. Při implementaci používáme několik maker, které nám usnadní práci, zejména pro kontrolu návratových kódů funkcí a případnou propagaci chyby. Na základě sekvence vstupních tokenů se parser rozhoduje o dalším postupu.

V případě pravidel, které můžou končit prázdným přechodem je použito makro *CALL_RULE_EMPTY*, které zajistí, že parser nepřejde při prázdném přechodu na další token, což by zavinilo přeskočení následujícího tokenu.

Přiřazení hodnot do více proměnných a argumenty funkce jsou řešeny pomocí jednosměrně vázaného seznamu. Při parsování jsou nejprve uloženy, a až na konci pravidla, buď po uložení všech v případě argumentů funkce, nebo po zpracování všech výrazů či zavolaní funkce v případě přiřazení se seznam prochází a generuje se odpovídající cílový kód.

LL-tabulka nebyla k implementaci potřeba.

2.2.1 Zpracování výrazů

Pro zpracování výrazů se nepoužívá metoda svrchu dolu ale zdola nahoru. Pro tento typ analýzy je potřeba vytvořit precedenční tabulku. Naše tabulka má rozměry 9x9 a operátory se stejnou prioritou jsme sloučili do jedné buňky pro přehlednost. Celé zpracování výrazů je implementováno v souboru expression.c a funkce, která výraz zpracuje se nazývá solvedExpression. Tuto funkci volá parser, když narazí na výraz, který je třeba zpracovat. Použili jsme algoritmus uváděný na přednáškách.

Pro převod tokenu do indexu tabulky jsme připravili 3 pomocné funkce *TokentoIden*, která nám změní TokenType na IdentType, protože potřebujeme více variant, než je v TokenType. Funkce *get_index_to_table a stack_to_table* vrací index tokenu nebo vrcholu zásobníku do naší precedenční tabulky. Díky těmto hodnotám se můžeme orientovat v tabulce. Pro výsledek v tabulce < pushneme na znak pomocný znak I_HALT a pak znak, který čteme. Pro > hledáme na zásobníku znak I_HALT, pokud najdeme a posloupnost IdenType na zásobníku sedí na jedno z pravidel, odstraníme posloupnost ze zásobníku, nahradíme I_NON_TERM a načteme další token. Pokud žádné pravidlo nesedí ukončujeme analýzu chybovou hláškou. Pro = vložíme na zásobník I_PAR_R a načteme další token.

Tento algoritmus se opakuje, dokud můžeme číst další výrazově validní tokeny a následně provádíme pravidlo > do té doby, než nenarazíme na začátek zásobníku (I_DOLAR). Řešíme zde i typovou kompatibilitu výrazu a sémantiku výrazu např. 13 .. “string”.

2.2.2 Zásobník pro zpracování výrazů

Pro ulehčení práce při zpracování výrazů jsme také samostatatně do souboru *stack.c* implementovali zásobník jako lineárně vázaný seznam. Funkce *Stack_Init* nám provede inicializaci zásobníku. *Stack_Push* nám vkládá do zásobníku druhý argument funkce typ tokenu a také třetí argument, který ukládá obsah. *Stack_Top_Ptr*, *Stack_Top_Type* a *Stack_Top_Data* vrací ukazatel zásobníku nebo pouze typ tokenu či obsah. *Stack_InsertBeforeNonTerm* vloží pomocný znak I_HALT před symbol funkce a záměrně vynechává I_NON_TERM. *Stack_Pop* odstraní poslední prvek zásobníku. Funkce Stack_Destroy plně odstraní zásobník.

2.3 Sémantická analýza

Při syntaktické analýze probíhá souběžně i analýza sémantická.

Sémantická kontrola probíhá pomocí implementaci tabulky symbolů v souboru *symtable.c.* Díky výběru zadání jsme implementovali pomocí vyhledávacího binárního stromu. Zvolili jsme rekurzivní verzi binárního stromu díky své jednodušší konstrukci. Každý uzel stromu obsahuje informace:

- Ukazatel na levý a pravý podstrom
- Booleovskou proměnou, zda je funkce definovaná
- Parametry funkce
- Typ – u proměnné její typ, u funkce návratová hodnota
- Klíč – název proměnné nebo funkce
- Id – číslo zanoření

V jednotlivých binárních stromech vyhledáváme pomocí klíče, kterým je pro nás identifikátor. Do tabulky symbolů se vždy uložila proměnná nebo funkce a pomocí funkcí *table_search, table_search_first a table_search_all* jsme prohledávali, zda tato proměnná již byla definována v určitém kontextu. V globálních tabulkách ukládáme funkce a v lokálních tabulkách ukládáme proměnné.

2.4 Generování kódu

Pro generování kódu jsme zvolili řešení, kdy generování tříadresového kódu přímo v *parser.c.* Během parsování se v předem zvolených místech kódu generuje vždy určitá část tříadresového kódu dle celkového kontextu. Návěští funkcí se generují podle jména funkce v kódu a unikátnost názvů proměnných je zajištěna přidáním čísla s hloubkou zanoření do názvu. Návěští pro skoky u podmínek a cyklů se tvoří přidáním hloubky zanoření a inkrementovanou hodnotou pořadí dané konstrukce v kódu.

Pro generování výrazů je využíván zásobníkový kód, pro předávání parametrů a návratových hodnot funkcí je taktéž využíván zásobník.

2.4.1 Generování vestavěných funkcí

Generování vestavěných funkcí jsme vyřešili pomocí vygenerování vestavěných funkcí na začátku programu a pokud tuto funkci nalezneme v kódu programu tak jenom pomocí vygenerovaní kódu CALL skočíme na začátek programu, kde tato funkce čeká na zavolání.

3. Spolupráce v týmu

3.1 Rozdělení práce

Původně jsme se snažili rozdělit práci rovnoměrně a spravedlivě dle náročnosti. Bohužel jeden člen týmu nás opustil, a tak vzniklo nerovnoměrné rozdělení.

Petr Hýbl (xhyblp01)	zpracování výrazu, zásobník, dokumentace
Alexandr Čelakovský (xcelak00)	lexikální analýza, vestavěné funkce, FSM
Martin Zmitko (xzmitk01)	sémantická, syntaktická analýza, generátor kódu, tabulka symbolů
Dominik Klon (xklond00)	

3.2 Verzovací systém

Pro programovou část byl zvolen verzovací systém Git, přičemž pro uložení našeho repozitáře a vzdálenou správu jsme zvolili platformu GitHub.

3.3 Komunikace

V týmu byly nastaveny dva komunikační kanály - Discord a Messenger. Discord sloužil pro obecné dotazy ohledně projektu, sdílených souborů a následně hlasovou komunikaci. Messenger měl sloužit

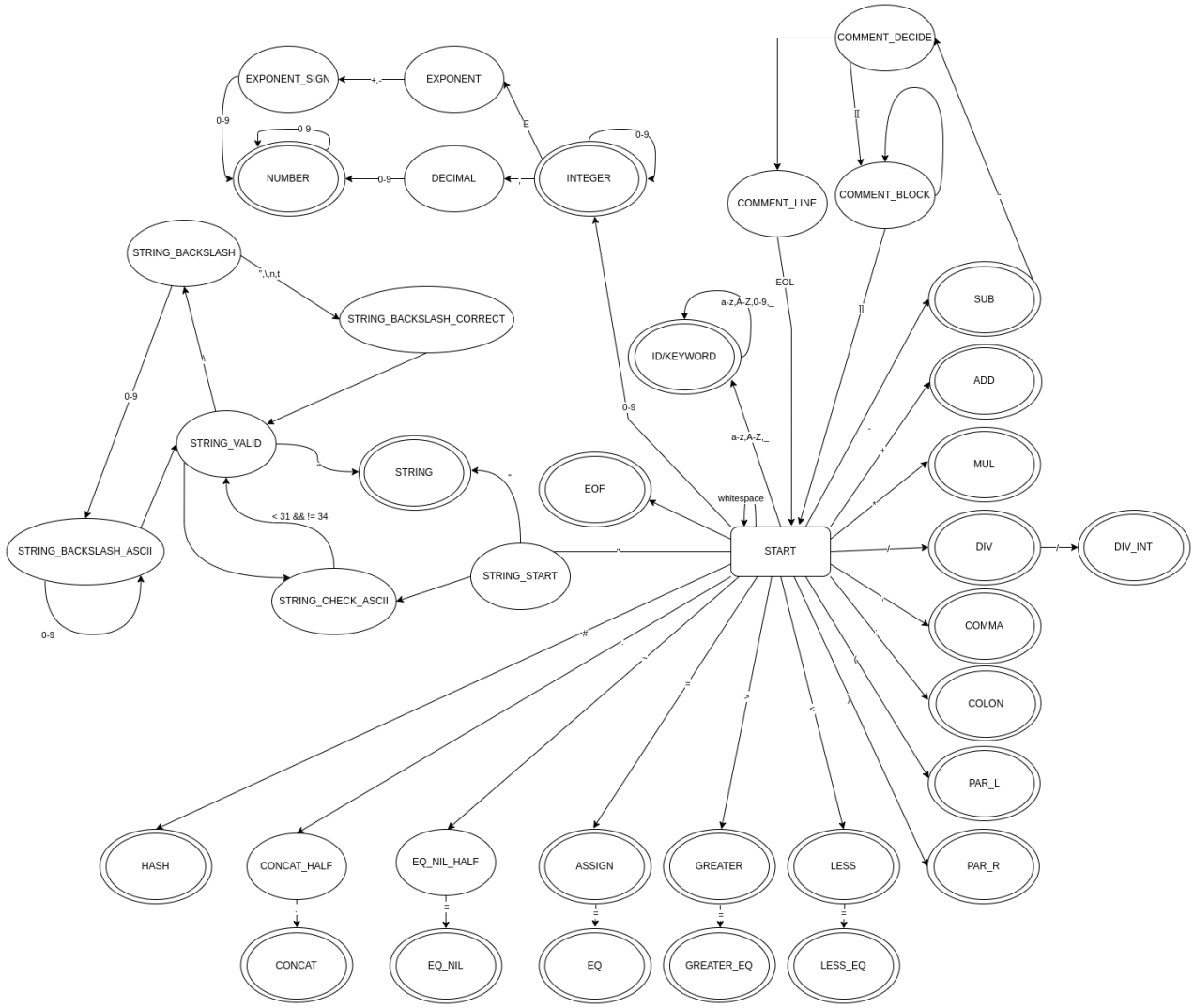
jako bod pro naléhavou komunikaci, protože ho má každý na svém mobilu, kdyby bylo potřeba něco naléhavě zodpovědět.

LL- gramatika

```
<prog> -> global ID : function ( <fdec_args> <f_types> <prog>
<prog> -> function ID ( <fdef_args> <f_types> <stat> <prog>
<prog> -> ID ( <args> <prog>
<prog> -> require STRING <prog>
<prog> -> EOF
<fdec_args> -> <type> <fdec_args_n>
<fdec_args> -> )
<fdec_args_n> -> , <type> <fdec_args_n>
<fdec_args_n> -> )
<fdef_args> -> ID : <type> <fdef_args_n>
<fdef_args> -> )
<fdef_args_n> -> , ID : <type> <fdef_args_n>
<fdef_args_n> -> )
<f_types> -> : <types>
<f_types> -> e
<types> -> <type> <types_n>
<types_n> -> , <type> <types_n>
<types_n> -> e
<args> -> <term> <args_n>
<args> -> )
<args_n> -> , <term> <args_n>
<args_n> -> )
<stat> -> ID ( <args> <stat>
<stat> -> local ID : <type> <stat>
<stat> -> local ID : <type> = EXPR <stat>
<stat> -> local ID : <type> = ID ( <args> <stat>
<stat> -> <IDs> <EXPRs> <stat>
<stat> -> if EXPR then <stat> else <stat> <stat>
<stat> -> while EXPR do <stat> <stat>
<stat> -> return <EXPRs> <stat>
<stat> -> end
<stat> -> else
<IDs> -> ID <IDs_n>
<IDs_n> -> , ID <IDs_n>
<IDs_n> -> =
<EXPRs> -> ID ( <args>
<EXPRs> -> EXPR <EXPRs_n>
<EXPRs_n> -> , EXPR <EXPRs_n>
<EXPRs_n> -> e
<type> -> number
<type> -> integer
<type> -> string
```

```
<type> -> nil
<term> -> NUMBER
<term> -> INTEGER
<term> -> STRING
<term> -> nil
```

Diagram konečného automatu



Precedenční tabulka

	+ -	* / //	(	)	i	$	<>	..	#
+ -	>	<	<	>	<	>	>	>	<
* / //	>	>	<	>	<	>	>	>	<
(	<	<	<	=	<		<	<	<
)	>	>		>		>	>	<	
i	>	>		>		>	>	>	
$	<	<	<		<	D	<	<	<
<>	<	<	<	>	<	>	>	<	<
..	<	<	<	>	<	>	>	<	<
#	>	>	<	>	<	>	>	>	<