Dokumentace k projektu z předmětu IFJ a IAL

IFJ 2020 - Tým 005, varianta I

Petr Růžanský	xruzan00	25 %	Vedoucí
Radek Maňák	xmanak20	50 %	
Adrián Bobola	xbobol00	25 %	
Vráblik Matúš	xvrabl05	0 %	

9. prosince 2020

Obsah

1 Rozdělení práce **2**

2 Odchylky od rovnoměrného rozdělení bodů **2**
2.1 Práce v týmu 2
2.2 Komunikácia v týme 2

3 Jednotlivé části programu **3**
3.1 Lexikální analýza 3
3.2 Syntaktická analýza 4
3.3 Sémantická analýza 5
3.4 Generátor kódu 5

4 Tabulky symbolů **6**
4.1 Lokální tabulky symbolů 6
4.2 Globální tabulka symbolů 6

5 Závěr **6**
Príloha: LL gramatika 7

1 Rozdělení práce

Práci jsme si rozdělili na čtyři části podle hlavních části projektu. Lexikální, syntaktické, sémantické analýzy a generování kódu. Podrobnější rozdělení jsme nechtěli dělat, protože jsme na začátku nevěděli jak překladač funguje a jaká bude časová náročnost jednotlivých části.
Dohodli jsme se, že každý bude zodpovědný za svoji část a pokud s ní skončí dříve tak pomůže ostatním.

Rozdělení úkolů bylo tedy následovné:
Radek Maňák - Lexikální analyzátor
Petr Růžanský - Syntaktický analyzátor
Adrián Bobola - Sémantické kontroly
Matúš Vráblik - Generování kódu

2 Odchylky od rovnoměrného rozdělení bodů

Matúš Vráblík byl už od začátku špatně dosažitelný na komunikačních kanálech. Po pobízení aby něco začal dělat se pokusil udělat precedenční syntaktickou analýzu, ale dodal pouze částečné (asi 80 řádků kódu) řešení, které nefungovalo a bylo jednoduší ho zahodit a implementovat znovu. Poté s námi přestal komunikovat. Proto má v rozdělení 0 procent bodů.
Radek Maňák je hodnocen 50 procenty bodů, protože samostatně implementoval lexikální analýzu. Implementoval precedenční analýzu od její syntaktické části až po generování kódu. Implementoval volání funkcí, a značnou část generování kódu.

2.1 Práce v týmu

Pro jednodušší práci v týmu jsme použili verzovací systém git. Vzdálený repositář jsme hostovali na platformě Github. Ten nám umožnil práci více členů týmu na tom samém souboru současně.
Taktéž nám byl nápomocný pi návrzích vylepšení a vzájemné diskuzi jednotlivých častí programu mezi členy týmu.

2.2 Komunikácia v týme

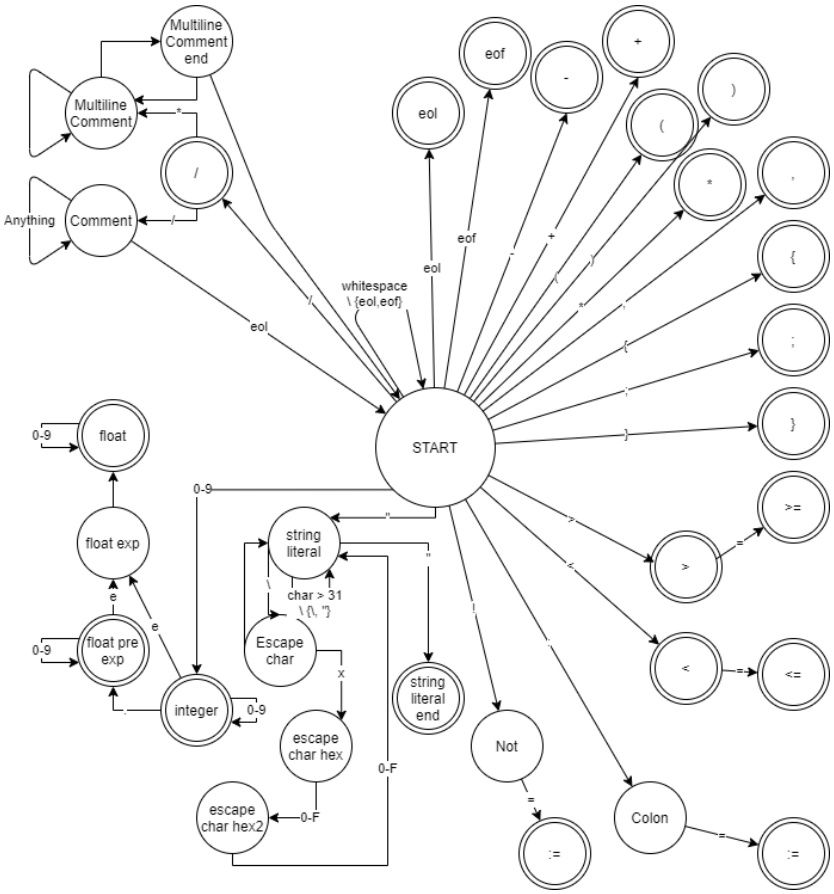
Na začátku semestru jsme se setkali na videohovoru pomocí platformy Discord. V průběhu práce nám více vyhovovala pouhá textová komunikace, protože ji nebylo nutné nijak organizovat a nevyžadovala účast všech členů v jednu dobu.

3 Jednotlivé části programu

3.1 Lexikální analýza

Lexikální analýzu jsme implementovali v souboru scanner.c pomocí stavového automatu. Tokeny vytvářené automatem se skládají z typu tokenu a atributu. Atribut je v závislosti na typu tokenu řetězec, celé číslo, číslo s pohyblivou řadovou čárkou nebo je token bez atributu.

Vytvořené tokeny jsou ukládány do pomocného bufferu aby mohli být znovu načteny při druhém průchodu syntaktické analýzy.



Obr.1: Graf stavového automatu lexikální analýzy

3.2 Syntaktická analýza

Až na výrazy se celá syntaktická analýza řídí LL tabulkou a pro implementaci byla použita metoda rekurzivního sestupu. Každý neterminál tvoří funkci. Ty se pak rekurzivně volají a každá funkce načítá a kontroluje tokeny.

Pro zpřehlednění volání neterminálů a kontrolu terminálů, jsou implementovány dvě pomocná makra:
První `CHECK_AND_LOAD_TOKEN([očekávaný_typ_tokenu])`, které zkontroluje, jestli aktuální token, nacházející se v globální proměnné `token`, odpovídá typu, zadaného v parametru makra `[očekávaný_typ_tokenu]`. Pokud ano, načte další token, jinak vrátí syntaktickou chybu (2).
Druhé pomocné makro je `CHECK_AND_CALL_FUNCTION([FUNKCE()])`, které pouze rekurzivně zavolá daný neterminál (funkci), zkontroluje její návratový typ a v případě, že rekurzivně volaný neterminál vrátí chybu, pošle ji dále.

Pro výrazy byla použita metoda zdola-nahoru a řízena byla tabulkou výrazů. Implementace syntaktické analýzy se nachází v souboru parser.c.

E	E + E
E	E - E
E	E * E
E	E / E
E	E r E
E	(E)
E	id
E	id()
E	id(E)
E	id(E, E)
E	id(E, E, E, ... Ei)

Obr.2: Tabulka gramatických pravidel precedenční analýzy

3.3 Sémantická analýza

Sémantická analýza pracuje spolu se syntaktickou analýzou ve dvou průchodech. První průchod je zjednodušený a pouze načte jména funkcí jejich parametry a návratové hodnoty. Informace o funkcích jsou poté uloženy do globální tabulky symbolů aby byly k dispozici při druhém hlavním průchodu.

V druhém průchodu se sémantická analýza stará o sběr, ukládání a následné načtení dat o funkcích a proměnných. Kontroluje zda jsou funkce volány se správnými parametry, že datové typy identifikátorů na levé straně přiřazení jsou stejné jako identifikátory na straně pravé.

Po zpracování výrazu je do globálního zásobníku pro výsledky výrazů vložen symbol s informacemi o datovém typu a název lokální proměnné na které je výsledek uložen.

3.4 Generátor kódu

Generování výsledného kódu jsme implementovali v souboru codegen.c, kde jsme vytvořili pomocné funkce na vygenerování určitého typu výsledného kódu.
Aby jsme zajistili unikátní názvy pro každé volání funkce (názvy proměnných, navésti a pod.), použili jsme globální proměnné na jejich číslování. Tím jsme zamezili chybám při spouštění výsledného interpretu.

Jednotlivé generovací funkce jsou volané průběžně při procházení programu sémantickou analýzou (implementace v souboru parser.c).

4 Tabulky symbolů

4.1 Lokální tabulky symbolů

Tabulku symbolů jsme implementovali jako binární vyhledávací strom, podle zvolené varianty zadání (soubor symtable.c).
Na rozlišení rozsahu platnosti jednotlivých proměnných jsme se v naší implementaci řídili pravidlem "nový rozsah platností = nová lokální tabulka symbolů". Tabulky symbolů jsme ukládali do globálního zásobníku tabulek. (implementace v souboru symstack.c).

Tím jsme zaručili, že aktuálně zpracovaný kód bude mít k dispozici právě tabulku platnou pro svůj rozsah platnosti, který získá pomocí operace "Push"ze zásobníku lokálních tabulek.

4.2 Globální tabulka symbolů

Kromě lokálních tabulek jsme implementovali i globální tabulku symbolů.
Ta nám slouží na ukládání základních informací o funkcích, jejich návratových typech a pod.

5 Závěr

Projekt jsme se snažili dělat průběžně během celého semestru podle stanoveného "time managementu". Řešení projektu bylo celkem rozsáhlé, ale díky přednáškám a demonstračním cvičením jsme to úspěšně zvládli.

Tvorba projektu nás obohatila o znalosti z oboru fungování překladačů a umožnila nám vyzkoušet vytvořit si svůj vlastní překladač. Tímto nám pomohla k lepšímu pochopení probrané problematiky i na praktické úrovni.

<start>	<preamble><body>
<preamble>	package id
<body>	<func><body>
<body>	EOF
<func>	func id (<params>) <ret_types>{ <state_list>}
<params>	eps
<params>	<param><params_n>
<params_n>	eps
<params_n>	, <param><params_n>
<param>	id <data_type>
<ret_types>	eps
<ret_types>	<data_type>
<ret_types>	(<types>)
<types>	eps
<types>	<data_type><types_n>
<types_n>	eps
<types_n>	, <data_type><types_n>
<data_type>	int
<data_type>	string
<data_type>	float64
<state_list>	eps
<state_list>	<state><state_list>
<state>	<declr>
<state>	if <expression>{ <state_list>} <else>
<state>	for <for_declr><expression>; <expression>{ <state_list>}
<state>	return <expression>
<state>	id (<func_param>)
<state>	id <id_n>= <expr>
<id_n>	eps
<id_n>	, id <id_n>
<func_param>	eps
<func_param>	<expr><func_param_n>
<func_param_n>	eps
<func_param_n>	, <expr><func_param_n>
<declr>	id := <expression>
<for_declr>	id := <expression>
<for_declr>	;
<else>	else { <state_list>}

Obr.3: LL gramatika