Formální jazyky a překladače

2020/2021

Implementace překladače imperativního jazyka IFJ20

Tým 018, varianta II

Vedoucí projektu: Šesták Pavel(xsesta07)

Spoluřešitelé: Kulíšek Vojtěch(xkulis03), Plevač Lukáš(xpleva07)

Brno, 9. prosince 2020

Obsah

1 Úvod

1.1 Komunikace mezi moduly

Komunikace mezi moduly probíhá převážně přes soubor globálních proměnných definovaných v rámci *global_variables.h*. Token reprezentuje aktuální zpracovávaný token od lexikální analýzy (Sekce 3 - Lexikální analyzátor). Proměnná *symtable* reprezentuje tabulku symbolů (Sekce 5.1 - Tabulka symbolů). Precedence_stack je globální zásobník pro precedenční analýzu zdola nahoru, použitou pro analýzu logických výrazů. Data_type_queue je fronta datových typů, která je použita u přiřazování výrazů do proměnných při kontrole datových typů. S frontou token_queue pracuje část syntaktické analýzy implementován rekurzivním sestupem a ukládá do ní tokeny do kterých se bude přiřazovat z důvodu kontroly datových typů.

1.2 Návratové hodnoty aplikace

Návratové hodnoty explicitně vyplývají ze zadání a jejich definice je součástí hlavičkového souboru *error_codes.h*.

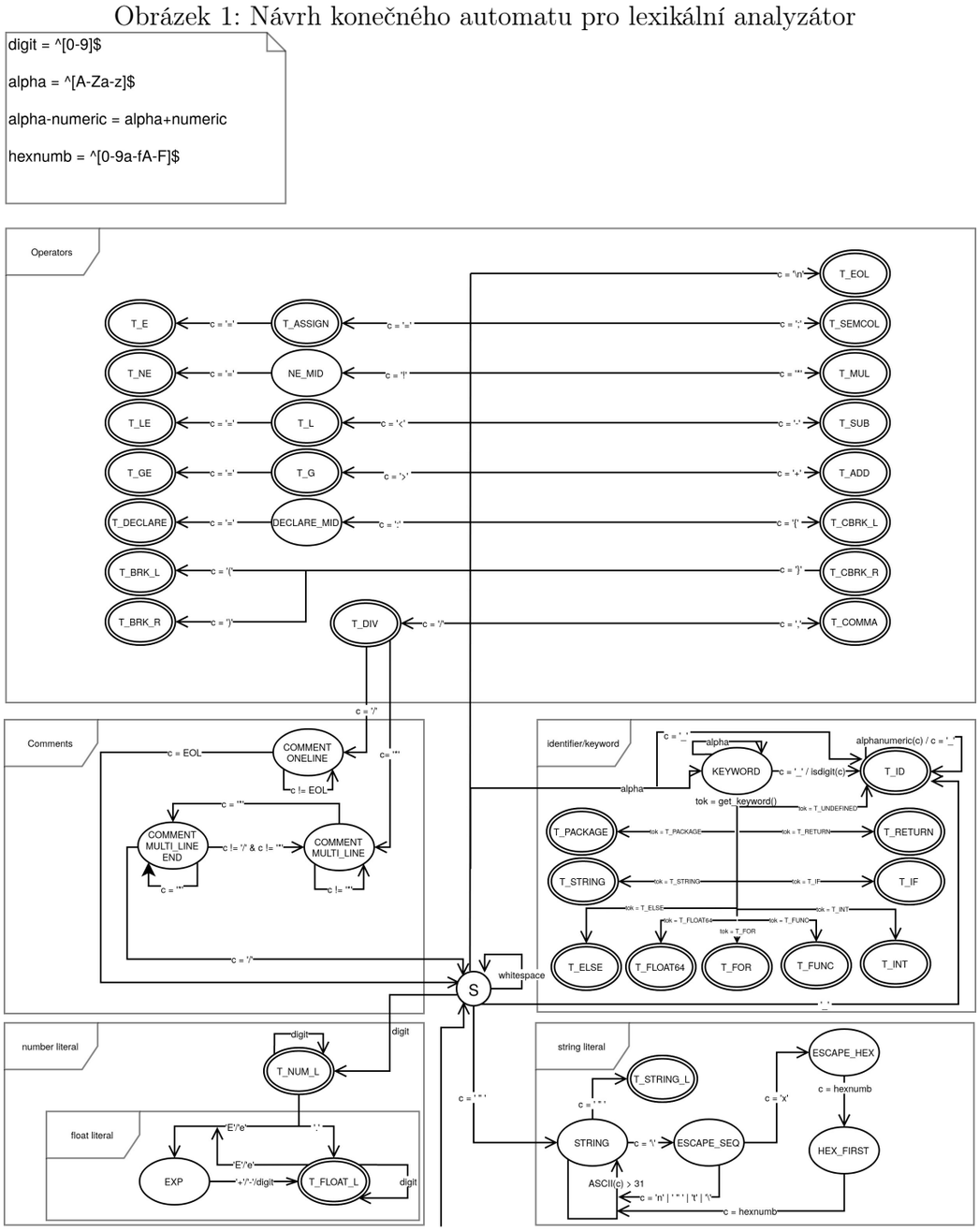
2 Práce v týmu

Práce na projektu probíhala v týmu, jehož složení je konstantní od začátku studia. Pro verzování projektu jsme zvolili git hostovaný na portálu GitLab. Kořenový adresář projektu je rozdělen do čtyř složek, pro oddělení jednotlivých částí projektu. Složka */doc* obsahuje popis těchto složek, dokumentaci v TeXu a popis jak je formátován zdrojový kód v rámci projektu. Druhá sekce */include* obsahuje hlavičkové soubory s definicemi a externími funkcemi s komentářem, které slouží pro komunikaci mezi jednotlivými moduly. Sekce /src obsahuje zdrojové soubory překladače. Poslední sekcí je */tests*, v rámci které jsou implementovány automatické testy na jednotlivé moduly a datové struktury. Při commitu byl využit pipelining pro build a následné provedení automatických testů. Komunikace ohledně projektu probíhala skrz Facebookovou skupinu a schůzky pořádané na platformě Discord. Práce byla rozdělena do separátních bloků, které byly rozděleny průběžně mezi členy týmu. Schůzky probíhali převážně z důvodu upřesnění funkcionality daných modulů a integrace jednotlivých modulů do funkčního celku.

3 Lexikální analyzátor

První blok překladače je lexikální analyzátor známý také jako skener. Jeho úkolem je rozložit zdrojový kód ze standardního vstupu na sekvenci tokenů. Lexikální analyzátor byl implementován pomocí konečného automatu(Obrázek 1). Konečné stavy automatu reprezentují jednotlivé tokeny. V případě, že automat se nachází v koncovém stavu a znak na vstupu není schopný zpracovat, vrací token do syntaktického analyzátoru. V opačném případě kdy automat není schopen znak přijmout a automat se nenachází v koncovém stavu končí překlad na lexikální chybu. Každý stav je reprezentován funkcí a aktuální stav je uložen pomocí ukazatele na funkci.
Stavy konečného automatu byly reprezentovány funkcemi, kdy další stav byl uložen v ukazateli na funkci, který byl volán v hlavní smyčce. Konečný automat je doplněn mnoha boolovskými proměnnými, které mu pomáhají se rozhodovat, zda-li číst další znak ze vstupu nebo zda byl načten již kompletní token.
Token je reprezentován strukturou obsahují informace o typu tokenu, jeho atributu a datovém typu. Jako atribut se rozumí hodnota literálu či název identifikátoru. Typ tokenu a datový typ je realizován pomocí výčtového typu. Atribut je realizován pomocí datového typu union.
Komunikační rozhraní s lexikálním analyzátorem je pomocí funkce *get_ token()*, která aktualizuje hodnotu globální proměnné token.
Lexikální analyzátor má počítadlo aktuálního řádku *lex_ line_ counter*, které je přístupné syntaktické i sémantické analýze při chybovém výstupu pro informování uživatele, na kterém řádku byl překlad ukončen.

Obrázek 1: Návrh konečného automatu pro lexikální analyzátor



4 Syntaktický analyzátor

Syntaktická analýza řídí překlad a je rozdělena do dvou celků - analýza pomocí rekurzivního sestupu a precedenční analýzy. Hlavní část syntaktické analýzy je implementována rekurzivním sestupem, kde každý neterminál reprezentuje samostatnou funkci. Syntaktická analýza volá lexikální analyzátor, který jí vrací další token ze vstupu. V rámci rekurzivního sestupu se snaží pokračovat na základě tokenu podle jednoho z pravidel definovaných v rámci LL gramatiky. V případě, že narazí na výraz tak syntaktický analyzátor volá precedenční analýzu, která výraz vyhodnotí a uloží datový typ na zásobník datových typů. Syntaktická analýza provede kontrolu datových typů. V průběhu simulace derivačního stromu se volají sémantické kontroly a provádí se generování cílového kódu.

4.1 Rekurzivní sestup

Rekurzivní sestup syntaktické analýzy je implementován v rámci souboru *syntax_ analyzer.c.*

4.1.1 LL gramatika

G_{LL} = (N, Σ, P, E)

N = { <statement>, <can_EOL>, <expression>, <multy_assign>, <expression_list>, <param_list>, <param>, <comparison>, <expression>, <rel_operator>, <for_declaration>, <for_comparison>, <for_assignment>, <func>, <param_define>, <return_list>, <return_data_type>, <data_type>, <mu_param_define>, <prolog> }

Σ= { T_EOF, T_ELSE, T_FLOAT64, T_FOR, T_FUNC, T_IF, T_INT, T_PACKAGE, T_RETURN, T_STRING, T_ID, T_NUM_L, T_FLOAT_L, T_STRING_L, T_COMMA, T_CBRK_R, T_CBRK_L, T_SEMCOL, T_SUB, T_ADD, T_NE, T_MUL, T_E, T_GE, T_G, T_EOL, T_LE, T_L, T_BRK_R, T_ASSIGN, T_DECLARE, T_BRK_L, T_DIV }

P = {
1: <can_EOL> → ε
2: <can_EOL> → T_EOL <can_EOL>
3: <prolog> → <can_EOL> T_PACKAGE main T_EOL <func> //define package
4: <func> → <can_EOL> T_FUNC T_ID T_BRK_L <param_define> T_BRK_R <return_list> T_CBRK_L <statement> T_CBRK_R <can_EOL> <func>
5: <func> → T_EOF
6: <param_define> → ε
7: <param_define> → T_ID <data_type>
8: <param_define> → T_ID <data_type> T_COMMA
<mu_param_define> 9: <mu_param_define> → T_ID <data_type>
10: <mu_param_define> → T_ID <data_type> T_COMMA <mu_param_define>
11: <return_list> → T_BRK_L <return_data_type> T_BRK_R
12: <return_list> → ε
13: <data_type> → T_INT
14: <data_type> → T_FLOAT64
15: <data_type> → T_STRING
16: <return_data_type> → <data_type>
17: <return_data_type> → <data_type> T_COMMA <return_data_type>
18: <statement> → <can_EOL>
19: <statement> → ε
20: <statement> → <can_EOL> T_ID T_DECLARE <can_EOL> <expression> T_EOL <statement>
21: <statement> →<can_EOL> T_ID <multy_assign> T_ASSIGN <can_EOL> <expression_list> T_EOL <statement>
22: <multy_assign> → ε
23: <multy_assign> → T_COMMA <can_EOL> T_ID <multy_assign>
24: <statement> → <can_EOL> T_ID T_BRK_L <can_EOL> <param_list> T_BRK_R T_EOL <statement>
25: <statement> → <can_EOL> T_ID T_BRK_L T_BRK_R T_EOL <statement>
28: <param_list> → <param>
29: <param_list> → <param> T_COMMA <can_EOL> <param_list>

30: <param> → T_STRING_L
31: <param> → T_ID
32: <param> → T_NUM_L
33: <param> → T_FLOAT_L
34: <statement> → <can_EOL> T_IF <comparison> T_CBRK_L T_EOL <statement> T_CBRK_R T_ELSE T_CBRK_L T_EOL <statement> T_CBRK_R T_EOL <statement>
35: <comparison> → <expression> <rel_operator> <can_EOL> <expression>
36: <rel_operator> → T_L
37: <rel_operator> → T_LE
38: <rel_operator> → T_G
39: <rel_operator> → T_GE
40: <rel_operator> → T_E
41: <rel_operator> → T_NE
42: <statement> → <can_EOL> T_FOR <for_declaration> T_SEMCOL <for_comparison> T_SEMCOL <for_assignment> T_CBRK_L T_EOL <statement> T_CBRK_R T_EOL <statement>
43: <for_declaration> → T_ID T_DECLARE <can_EOL> <expression>
44: <for_declaration> → ε
45: <for_comparison> → <comparison>
46: <for_comparison> → ε
47: <for_assignment> → T_ID <multy_assign> T_ASSIGN <can_EOL> <expression_list>
48: <for_assignment> → ε
49: <statement> → <can_EOL> T_RETURN <expression_list> T_EOL <statement>
50: <statement> → <can_EOL> T_RETURN T_EOL <statement>
51: <expression_list> → <expression>
52: <expression_list> → <expression> T_COMMA <expression_list>

}

E = { <prolog> }

4.2 Precedenční analýza

Precedenční analýza zdola nahoru je v rámci překladače využita k analýze výrazů. Analýza výrazu je implementována v rámci souboru *expression_ analyser.c* a přístupná přes volání *expression_ analyser()*. Uchovává indexy do precedenční tabulky, a implementuje práci se zásobníkem pomocí pravidel definovaných v tabulce. Výsledný datový typ výrazu je uložen do fronty datových typů.

4.2.1 Gramatika pro precedenční analýzu

G_{exp} = (N, Σ, P, E)

N = { <E> }

Σ= { T_NUM_L, T_FLOAT_L, T_STRING_L, T_ID, T_ADD, T_SUB, T_MUL, T_DIV, T_BRK_L, T_BRK_R, T_EOF }

P = {
1: <E> → <E> T_ADD <E>
2: <E> → <E> T_SUB <E>
3: <E> → <E> T_MUL <E>
4: <E> → <E> T_DIV <E>
5: <E> → T_BRK_L <E> T_BRK_R
6: <E> → T_NUM_L
7: <E> → T_FLOAT_L
8: <E> → T_STRING_L
9: <E> → T_ID
}

E = { <E> }

4.2.2 Precedenční tabulka

Precedenční tabulka je implementována jako dvou rozměrné pole znaků

	T_NUM_L	T_FLOAT_L	T_STRING_L	T_ID	T_ADD	T_SUB	T_MUL	T_DIV	T_BRK_L	R_BRK_R	T_EOF
T_NUM_L	_	_	_	_	⌋	⌋	⌋	⌋	_	⌋	⌋
T_FLOAT_L	_	_	_	_	⌋	⌋	⌋	⌋	_	⌋	⌋
T_STRING_L	_	_	_	_	⌋	⌋	⌋	⌋	_	⌋	⌋
T_ID	_	_	_	_	⌋	⌋	⌋	⌋	_	⌋	⌋
T_ADD	⌊	⌊	⌊	⌊	⌋	⌋	⌊	⌊	⌊	⌋	⌋
T_SUB	⌊	⌊	⌊	⌊	⌋	⌋	⌊	⌊	⌊	⌋	⌋
T_MUL	⌊	⌊	⌊	⌊	⌋	⌋	⌋	⌋	⌊	⌋	⌋
T_DIV	⌊	⌊	⌊	⌊	⌋	⌋	⌋	⌋	⌊	⌋	⌋
T_BRK_L	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	=	_
T_BRK_R	_	_	_	_	⌋	⌋	⌋	⌋	_	⌋	⌋
T_EOF	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	_	_

5 Sémantický analyzátor

Sémantický analyzátor je volán syntaktickým analyzátorem. Kontroluje korektní volání funkcí a jejich deklarace.

5.1 Tabulka symbolů

Tabulka symbolů podle naší verze zadání měla být implementována jako hashovací tabulka. Položka hashovací tabulky je definována jako datový typ *symtable_item_T*, který v sobě uschovává informace o datovém typu, názvu a zda byla již deklarovaná proměnná. V případě funkcí ještě nese informaci o datových typech parametrů a návratových hodnot. V rámci projektu je použito pole hashovacích tabulek, které je celé implementováno jako dvou rozměrné pole. Rozsah pole je definován pomocí konstant SYM_MAX_SCOPE a SYM_TOP_ITEMS. Tabulka symbolů dále ošetřuje predikci funkcí, které nebyli doposud deklarovány. Za tímto účelem byly přidány do tabulky symbolů příznaky pro zjištění, zdali byla funkce deklarována nebo volána.

5.2 Kontrola datových typů

Kontrola datových typů je založena na porovnávání datových typů ve dvou frontách. V případě že se jedná o přiřazení tak Syntaktická analýza jakmile narazí na přiřazení, tak proměnné do kterých se bude přiřazovat, uloží do globální fronty *token_queue*. Precedenční analýza výrazů ukládá výsledný datový typ do globální fronty *data_type_queue*. Jakmile precedenční analýza zpracuje výrazy, syntaktická analýza provede porovnání těchto zásobníků, jakmile nesedí počet prvků nebo se datové typy neshodují, tak je překlad ukončen sémantickou chybou.
V případě že se kontroluje výraz za klíčovým slovem *return* tak syntaktický analyzátor nevkládá nic do *token_queue*, ale volá tabulku symbolů, která vrací návratový typ dané funkce definovaný v prototypu funkce.

6 Generování kódu

Funkce pro generování kódu jsou implementovány v rámci souboru *code_ generator.c*. Podpůrné funkce jsou volány ze syntaktické analýzy a precedenční analýzy výrazů. V projektu není implementován optimalizátor. Generovaný kód pracuje se zásobníkem a díky tomu nevzniká velké množství kompilátorových proměných. Většinu konstrukcí jsme schopni generovat přímo, vyjímkou je deklarace uvnitř cyklu. Deklarace proměných uvnitř cyklů je přepsána na přiřazení. Všechny proměnné, které mají být v bloku cyklů deklaravaná se ukládájí do struktury nafukovacího pole a po ukončení bloku for cyklů je vygenerováno návěští, na které se skáče před vstupem do prvního cyklu v rámci bloku cyklů. Za toto návěští jsou generovány deklarace a následně se skáče zpět na začátek cyklu.