Dokumentace k projektu z předmětů IFJ a IAL Implementace překladače imperativního jazyka IFJ23

Tým xkukht01, varianta – vv-BVS

Kukhta Myron (xkukht01) 39%

Malashchuk Vladyslav (xmalas04) 33%

Burylov Volodymyr (xburyl00) 28%

Pikulin Artemii (xpikul03) 0%

6. prosince 2023

1 Úvod

Cílem projektu bylo vytvořit program na C, který načte zdrojový kód napsaný na IFJ23 a přeložil jste ho do jazzuka IFJ23code. Vrácenou hodnotou programu je chybový kód (vrátí 0, pokud překlad proběhl bez chyb). Jazyk IFJ23 je zjednodušenou podmnožinou jazyka Swift2 , což je moderní a stále populárnější jazyk používaný především při vývoji pro zařízení Apple.

2 Práce v týmu

2.1 Rozdělení práce

Rozdělili jsme práci mezi čtyři členy týmu. Každý dělal svou část, v případě potřeby mu pomáhali ostatní členové týmu. Celkové testování projektu jsme prováděli společně. Práce byla komplikována kvůli neplnění svých povinností jedním z členů týmu, který ji v posledních dnech opustil a nevykonal svou část práce(generátor kódu).

2.2 Způsob spolupráce

Při pracích na projektu jsme aktivně využívali systém pro správu verzí Git. Naše zdrojové kódy byly umístěny v repozitáři na online platformě GitHub, což zajistilo, že každý člen týmu měl snadný přístup k nejaktuálnější verzi projektu.

Pro specifické části projektu jsme vytvořili individuální větve, kde jsme prováděli testování a provedli nezbytné úpravy. Před začleněním do hlavní větve byl nezbytný tzv. "pull request", následovaný pečlivým posouzením kódu (code review) a schválením od jednoho či více členů týmu. Tímto postupem jsme zaručili kvalitu a integritu našeho projektu.

3 Návrh a implementace

3.1 Lexikální analýza

Lexikální analyzátor je implementován jako konečný automat, jak lze vidět ve zdrojovém souboru scanner.c a přidruženém hlavičkovém souboru scanner.h. Hlavní funkce, get_token, je zodpovědná za rozpoznávání tokenů a jejich zápis do struktury Token. Tato struktura obsahuje typ tokenu, atributy a informaci o tom, zda za ním následuje nový řádek. Atributy tokenu mohou být integer, floating, string(Dynamic_string_t), keyword(Klíčové slovo) v závislosti na tokenu.

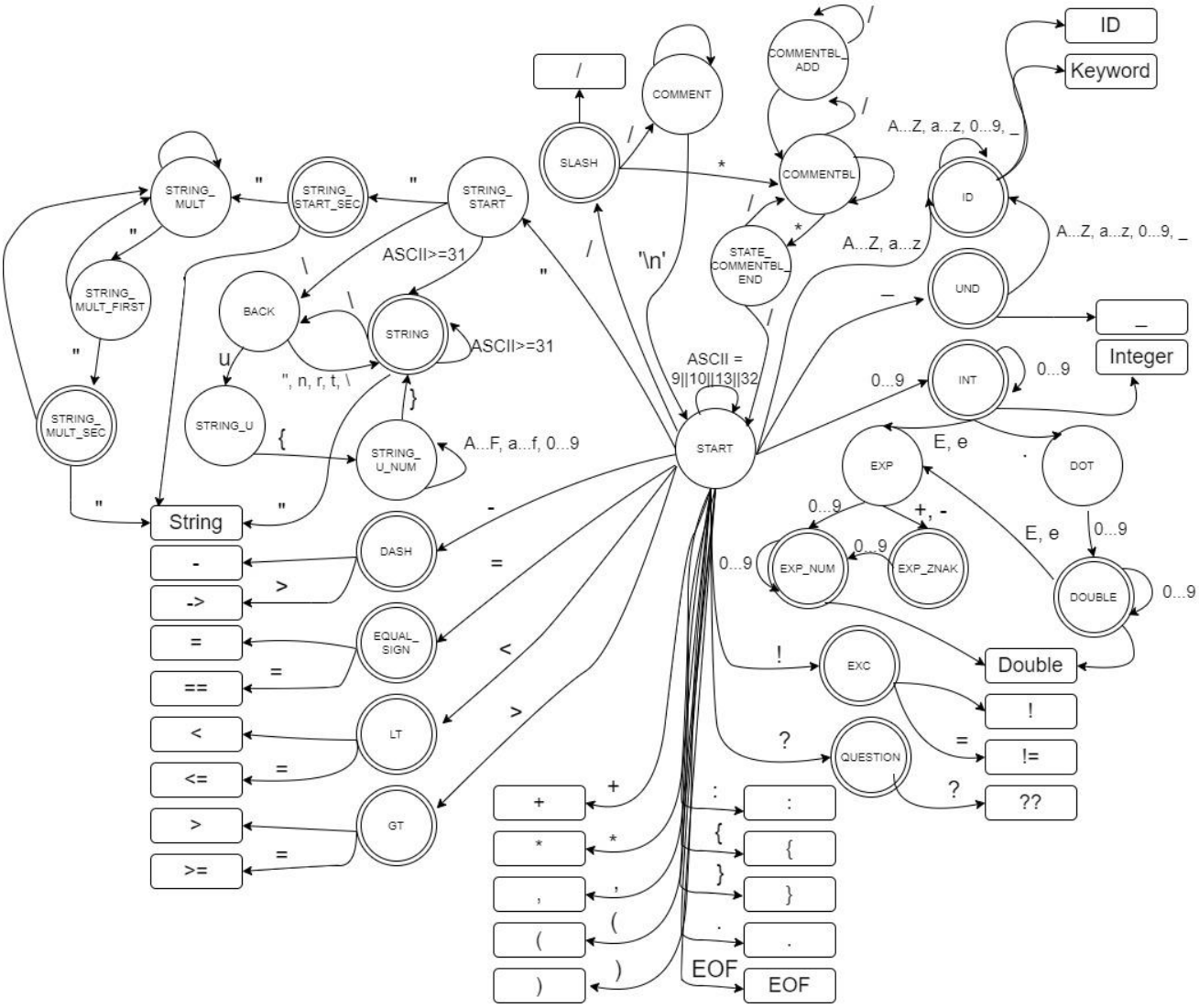
Konečný automat je implementován pomocí příkazu switch-case, kde každý stav odpovídá určitému znaku nebo skupině znaků. Funkce get_token čte znaky a přechází mezi stavy, dokud nedojde ke SCAN_END. V případě neshody s očekávanými znaky program signalizuje chybu (návratová hodnota 1).

Pokud funkce get_token identifikuje token typu TOKEN_ID, používá dvě další funkce: keyword_or_id a keyword_or_not. První kontroluje, zda je identifikátor klíčovým slovem, a případně přiřazuje odpovídající hodnotu atributu. Druhá se stará o případ, kdy na konci identifikátoru následuje otazník (?). Pokud je to klíčové slovo, přiřazuje hodnotu atributu keyword KW_INT_NIL, KW_DOUBLE_NIL nebo KW_STRING_NIL.

Kromě toho jsou v kódu vytvořeny pomocné proměnné, jako je State_string_help_str pro zpracování escape sekvencí (\u{dd}), helpcomblock pro počítání otevřených komentářů v bloku a Dynamic_string_t pro dočasné uložení dat tokenu pro další zpracování.

Během vývoje jsem se setkal s problémy při zápisu do znakového řetězce pomocí escape sekvence '\u{dd}', kde je vyžadována práce s hexadecimálními číslicemi. Tento problém jsem vyřešil pomocí pomocných znaků a funkce strtol, která umožňuje převést číselný řetězec čísla zapsaného v hexadecimálním systému do desítkové soustavy.

3.1.1 Diagram konečného automatu



3.2 Syntaktická analýza

V projektu kompilátoru pro Swift hraje syntaktická analýza klíčovou roli při přeměně vstupního zdrojového kódu do strukturovaného formátu v souladu se syntaxí jazyka. Tento proces zahrnuje interakci mezi parserem a skenerem, kde parser žádá o tokeny od skeneru pomocí funkce get_token.

Lexikální analýza prováděná skenerem převádí symboly zdrojového kódu na tokeny. Při úspěšné lexikální analýze skener vrátí token(lexém ze vstupního kódu). Parser na zakladí LL gramatiku, začíná rekurzivně zpracovávat pravidla odpovídající aktuálnímu stavu. Pokud aktuální stav parseru nesouhlasí s očekávaným tokenem, vratíme syntaktická chyba. Tento proces se opakuje až do úplného rozboru vstupního kódu.

3.2.1 LL-gramatika(tabulka)

	let	var	func	ID	LITERAL	if	else	while	{	}	(	)	:	,	Int	Int?	Double	Double?	String	String?	nil	return	=	EOF	e	expression	_	->	$
<SWIFT_PROG>	3	3	2	3		3		3																1					
<STATEMENT_LIST>	6	7		8		4		5																	9				
<IF_EXPRESSION>	11																									10			
<VAR_DEFINITION>				12							11												10						
<ID_TYPE_DEFITION>													13												14				
<ID_ACTION>											16												15						
<ASSIGMENT>				18																						17			
<ARG_LIST>				19																					20				
<ARG>				21																					22				
<ARG_NAME>				23																					23				
<NEXT_ARG>														24											25				
<FUNCTION_DEFINITION>			25																										
<PARAM_LIST>				26																					27		26		
<PARAM>				28																							28		
<FUNCTION_STATEMENT_LIST>	29	29		29		29		29														30							
<RETURN_EXPRESSION>																									32	31			
<NAME_PARAM>				33																							34		
<ID_PARAM>				35																							36		
<NEXT_PARAM>														37											38				
<FUNC_TYPE_DEFINITION>																									40			39	
<TERM>				41	42																42								
TYPE															43	44	45	46	47	48									

3.2.2 LL-gramatika(pravidla)

1. <SWIFT_PROG> -> EOF
2. <SWIFT_PROG> -> <FUNCTION_DEFINITION> <SWIFT_PROG>
3. <SWIFT_PROG> -> <STATEMENT_LIST> <SWIFT_PROG>
4. <STATEMENT_LIST> -> if <IF_EXPRESSION> { <STATEMENT_LIST> } else { <STATEMENT_LIST> } <STATEMENT_LIST>
5. <STATEMENT_LIST> -> while EXPRESSION { <STATEMENT_LIST> } <STATEMENT_LIST>
6. <STATEMENT_LIST> -> let <VAR_DEFINITION>
7. <STATEMENT_LIST> -> var <VAR_DEFINITION>
8. <STATEMENT_LIST> -> ID <ID_ACTION>
9. <STATEMENT_LIST> -> e
10. <IF_EXPRESSION> -> EXPRESSION
11. <IF_EXPRESSION> -> LET ID
12. <VAR_DEFINITION> -> ID <ID_TYPE_DEFITION> <ASSIGMENT><local_scope>
13. <ID_TYPE_DEFITION> -> : <TYPE>

14. <ID_TYPE_DEFITION> -> e
15. <ID_ACTION> -> = <ASSIGMENT>
16. <ID_ACTION> -> (<ARG_LIST>)
17. <ASSIGMENT> -> EXPRESSION
18. <ASSIGMENT> -> ID(<ARG_LIST>)
19. <ARG_LIST> -> <ARG><NEXT_ARG>
20. <ARG_LIST> -> e
21. <ARG> -> <ARG_NAME> <TERM>
22. <ARG_NAME> -> ID :
23. <ARG_NAME> -> e
24. <NEXT_ARG> -> ,<ARG><NEXT_ARG>
24. <NEXT_ARG> -> e
25. <FUNCTION_DEFINITION> -> FUNC ID (<PARAM_LIST>) <FUNC_TYPE_DEFINTION> { <FUNCTION_STATEMENT_LIST> }
26. <PARAM_LIST> -> <PARAM> <NEXT_PARAM>
27. <PARAM_LIST> -> e
28. <PARAM> -> <NAME_PARAM> <ID_PARAM> : <TYPE>
29. <FUNCTION_STATEMENT_LIST> -> <STATEMENT_LIST><FUNCTION_STATEMENT_LIST>
30. <FUNCTION_STATEMENT_LIST> -> return <RETURN_EXPRESSION> <FUNCTION_STATEMENT_LIST>
31. <RETURN_EXPRESSION> -> EXPRESSION
32. <RETURN_EXPRESSION> -> e
33. <NAME_PARAM> > ID
34. <NAME_PARAM> -> _
35. <ID_PARAM> -> ID
36. <ID_PARAM> -> _
37. <NEXT_PARAM> -> ,<PARAM><NEXT_PARAM>
38. <NEXT_PARAM> -> e
39. <FUNC_TYPE_DEFINTION> -> -> <TYPE>
40. <FUNC_TYPE_DEFINTION> -> e
41. <TERM> -> ID
42. <TERM> -> LITERAL
43. <TYPE> -> INT
44. <TYPE> -> INT?
45. <TYPE> -> DOUBLE
46. <TYPE> -> DOUBLE?
47. <TYPE> -> STRING
48. <TYPE> -> STRING?

3.3 Sémantická analýza

Během procesu sémantické analýzy prouřívame tabulku symbolů, implementovanou ve formě vyváženého binárního stromu podle výšky. Tabulku kterou uchováváme na zásobníku. První tabulka symbolů v dolní části zásobníku obsahuje informace o globálních proměnných a funkcích. Nad ní jsou umístěny další tabulky, zodpovědné za lokální oblasti viditelnosti, kde jsou uchovávány lokální proměnné, které mohou překrývat globální proměnné. Každá proměnná a funkce jsou uloženy v uzle stromu a obsahují důležité informace, jako je název, datový typ, definice, inicializace atd. Tento přístup nám umožňuje efektivně sledovat, zda všechny proměnné a funkce zapojené v kódu byly správně inicializovány a odpovídají očekávanému typu.

Hlavní složitost naší analýzy spočívá v použití jednoprůchodového přístupu kódu, což přináší určité obtíže při práci funkcemí, které používame do jejích definice. Jsem musíl vytvořít spravní algoritm, abych předpokladat spravní typ funkcí a jejich parametry.

3.4 Zpracování výrazu

Zpracování výrazu je implementováno v souboru expression.c a provádí se funkcí rule_expression, kterou volá syntaktický analyzátor. Výrazy se zpracovávají pomocí precedenční tabulky a zásobníku symbolů. Zpracování jednotlivých výrazů začíná inicializací zásobníku symbolů a uložením symbolu dolaru na jeho vrchol. Pak token po tokenu začíná funkce analyzovat vstupní výraz. Z každého tokenu funkce bere potřebné informace (symbol, datový typ - pokud je typ tokenu TOKEN_IDE (identifikátor), pak ho najde v tabulkách symbolů a předá typ jeho hodnoty). Pak na základě nejbližšího k vrcholu zásobníku terminálu a vstupního symbolu, který je získán z následujícího tokenu, funkce provádí jednu z 4 operací - shift, reduce, equal, error (ne znamená 100% chybu, ale provede několik kontrol), všechny možné kombinace vstupních symbolů a prvních terminálů na vrcholu jsou popsány precedenční tabulkou.

Pokud znak z tabulky je <(S), vloží menšítko před prvním terminálem, vloží načtený symbol na vrchol zásobníku (funkce shift) a zavolá funkci get_token.

Pokud znak z tabulky je >(R), ověří se, že mezi < a > je validní výraz (funkce test_rule pro nalezení vhodného pravidla a funkce test_semantics pro ověření typové kompatibility, vhodnosti typů operandů pro konkrétní operace a možná i, pokud je to nutné, přetypování některých operandů) a zredukuje získaný výraz do neterminálu E s výsledným datovým typem (funkce reduce).

Pokud znak z tabulky je =, provede push a zavolá funkci get_token.

Pokud znak z tabulky je "N", začne redukovat všechno, co se zjistalo v zásobníku až do okamžiku, kdy prvním terminálem v zásobníku bude dolar. Pak zkontroluje symbol následujícího tokenu a v případě korektnosti výrazu nastaví logickou proměnnou success na true, což způsobí východ z loopu.

Na konci zkontroluje typ výrazu a na základě toho provede přetypování.

3.4.1 Precedenční tabulka

	+-	*/	!	??	r	(	)	i	$
+-	>	<	<	>	>	<	>	<	>
*/	>	>	<	>	>	<	>	<	>
!	>	>	#	>	>	<	>	#	>
??	<	<	<	<	<	<	>	<	>
r	<	<	<	>	#	<	>	<	>
(	<	<	<	<	<	<	=	<	#
)	>	>	>	>	>	#	>	#	>
i	>	>	>	>	>	#	>	#	>
$	<	<	<	<	<	<	#	<	#

3.5 Generování cílového kódu

Chybí kvůli nedodržení práce jednoho z členů týmu.

4. Algoritmy

4.1 Zasobník

V rámci našeho kompilátoru pro Swift jsme implementovali efektivní systém pro správu tabulek symbolů pomocí struktury zásobníku. Tato organizace umožňuje pružně a hierarchicky uchovávat informace o symbolech během celého procesu sémantické analýzy. V úvodu inicializujeme dno zásobníku. Dno zásobníku slouží jako úložiště pro tabulku symbolů globální oblasti viditelnosti, obsahující informace o globálních proměnných a funkcích. Během průchodu kódem postupně přidáváme nové tabulky symbolů na vrchol zásobníku. Každá nová tabulka symbolů reprezentuje lokální oblast viditelnosti. Tabulky symbolů se na vrcholu zásobníku ukládají a odstraňují dle aktuálního kontextu analýzy.

4.2 Binární Výškově Vyvážený Strom

Tabulka symbolů je implementována jako uzly v binárním vyváženém stromu. Uzly obsahují informace o symbolech, jako je název, datový typ, definice, inicializace. Při přidání nového symbolu provádíme vyvažování stromu pomocí speciálně připravené funkce. Tato operace nám umožňuje optimalizovat časovou složitost při hledání symbolu v tabulce.

4.3 Dynamický řetězec

Dynamický řetězec je pomocná struktura, která se používá k zápisu znaků do řádku, když předem neznáme počet těchto znaků. Dynamický řetězec se skládá ze tří prvků: samotného řádku, délky řádku a velikosti vyhrazené pro řetězec paměti. Má funkce ds_init pro inicializaci řádku, ds_add_char pro přidání jednoho znaménka do řádku, ds_add_chars pro přidání více znaků do řádku, ds_copy pro kopírování jednoho dynamického řádku do druhého, ds_to_string pro kopírování dynamického řádku do char a ds_free pro uvolnění paměti zvýrazněné pod dynamickým řádkem.