Projektová dokumentace

Implementace překladače imperativního jazyka IFJ20

Tým 55, varianta I

Stepaniuk Roman Bc.	**(xstepa64)**	29%
Pastushenko Vladislav	(xpastu04)	23%
Bahdanovich Viktoryia	(xbahda01)	24%
Tomason Viktoryia	(xtomas34)	24%

Brno
09.12.2020

Obsah

1 Úvod

Výpracovaný projekt načítá zdrojový kód zapsaný ve zdrojovém jazyce IFJ20 ze standardního vstupu a generuje výsledný mezikód v jazyce IFJcode20 na standardní výstup nebo vrací odpovídající chybový kód v případě chyby.

Tato dokumentace popisuje návrh, implementaci, způsob práce v týmu.

2 Návrh a implementace

2.1 Lexikální analýza

Lexikální analýza je implementovaná ve zdrojovém souboru `scanner.c`. Ve `scanner.h` se nachází prototypy funkcí, a soubor `token.h` obsahuje enumerace typů tokenů. Analýza je implementována pomocí konečného automatu a definuje datovou strukturu `Token`, která je reprezentovaná jako jednosměrně vázaný lineární seznam se svými položkami: “type”, “data”, “size” a ukazatelem na další token. Během lexikální analýzy pomocí hlavní funkce `get_token` se ze vstupního souboru cyklické načítá symbol po symbolů. Na začátku přečteme první symbol odlišný od ”tabu”a ”mezery”. Podle prvního získaného symbolu se předem zvolíme typ tokenů (např. získáme-li ‘0-9‘ - nastavíme typ tokenu ‘TOKEN_TYPE_LITERAL_INT‘). Pak se cyklicky načítají další symboly, dokud nedosáhneme konce souboru, ověřuje se správnost vstupních symbolů a lexikálně se analyzuje získané tokeny. Pracujeme s tokeny pomocí konstrukce if-else a předem nastavených předpokládaných typu tokenů. A případně měníme typ tokenu. Počítáme znaky dopředu a v případě potřeby uložíme počítaný symbol do statické proměnné, která se zachová hodnotu mezi voláními funkcí. A příště budeme pracovat s tímto symbolem.

2.2 Syntaktická analýza

Nejdůležitější a hlavní část projektu je syntaktická analýza. Syntaktická analýza je realizovaná metodou rekurzivního sestupu. Program na začátku prací alokuje paměť pro strukturu `SymTable` i ostatní datové struktury a volá parser, který pomocí funkce `get_and_set_token` využivá scanner a s jeho pomocí čte soubor a generuje další token. Pak parser provádí syntaktickou analýzu podle pravidel, které jsou popsaný na poslední stránce . Pravidla mají svoje vlastni funkcí a čtou tokeny dokud nedosáhne konce souboru nebo pokud další token nebude patřit do pravidel. Když program začne načítat “expression”, udělá základní kontrolu syntaxe a předá “expression” precedenčnímu analyzátoru. Kvůli sémantické analýze program běží dvakrát a na prvním běhu syntaktický analyzátor prohází jen hlavičky funkci a jejích parametry. Při druhém běhu kontroluje celý program včetně těl funkci.

Funkce jsou implementovány v souboru `parser.c`

2.3 Zpracování výrazů

Když parser začíná zpracovávat výraz, on vytvoří zásobník pro tokeny, který bude ten výraz obsahovat. Kdy zpracování všech tokenů výrazu bude ukončeno, parser zavolá funkci `expression_processor`, která přijímá zásobník tokenů s výrazem a kontroluji pokud má chyby.

Taky, `expression_processor` zjisti typ výrazu a vrací číslo typu výrazu nebo -1 pokud došlo k chybě.

`Tabulka priority operátorů`

Priorita	Operátor
1	* /
2	+ -
3	< > =< >= == !=

Když výraz bude zkontrolován, uvolňuji paměť , a zavolá funkci `generateCode`, která přečte zásobník s výrazem a vygeneruje IFJcode20 pomocí precedenční syntaktické analýzy. Všechny operandy se zapisují do zásobníku IFJcode20, a operátory volájí odpovídající matematické instrukce, ve souladu s typem výrazu. Když `generateCode` ukončí spravovat výraz, uvolni paměť a ukončí práce.

`Tabulka precedenční syntaktické analýzy`

	+ -	* /	r	(	)	i	$
+ -	>	<	>	<	>	<	>
* /	>	>	>	<	>	<	>
r	>	<	>	<	>	<	>
(	<	<	<	<	=	<	err
)	>	>	>	err	>	err	>
i	>	>	>	err	>	err	>
$	<	<	<	<	err	<	err

2.4 Sémantická analýza

Kvůli sémantické analýze program dělá dva běhy a běží současně se syntaktickou analýzou. Při prvním běhu program kontroluje jména funkcí, jejích parametry a dodává informaci o nich do tabulky symbolů. Abychom mohli počítat s tím, že se tělo funkce může vyskytovat i po její volání, jsme museli udělat druhý běh, který už pracuje pouze s tělem funkci. Když syntaxe uvidí identifikátor, pro sémantickou analýzu bude znamenat že se jedna o proměnnou, a v záležitosti od místa v programu ta proměnná bude nebo přidaná do tabulky symbolů nebo bude zkontrolováno, jestli ta proměnná už existuje. Největší problém, se kterém jsme se setkali v sémantické analýze, bylo použití proměnných uvnitř

různých bloků buď na různých úrovních nebo na stejném úrovni. Řešením bylo využiti proměnnou pro hloubku 'deep' a seznam binárních stromu pro vyhledávaní proměnných. S její pomoci každá deklarace proměnné v novém bloku vytvářela nový binární strom na nové hloubce (jeden strom pro každou hloubku). A každý konec bloku znamenal, že proměnné na tomto úrovni už se nevyužijí a celý ten strom je odstraněn. Proměnné na 0 úrovni budou odstraněny když tělo funkce skončí. Vstupní parametry byly přeneseny do tabulky symbolů jako proměnné 0 úrovně. Funkci budou odstraněny z tabulky symbolů už před ukončením programu. Ještě jedna kontrola probíhá když musíme srovnávat jednu nebo několik proměnné s návratovými hodnotami funkce anebo z nějakým počtem výrazů. V těch případech využíváme dva zásobníky a tabulku symbolů pro shodu typů. Taky vždycky se kontrolují typy výstupních parametrů funkci a co ta funkce vrací.

Funkce jsou implementovány v souboru `parser.c`

3 Datové struktury a speciální algoritmy

Vybrali jsme variantu projektu s abstraktnou dátovou strukturou binární vyhledávací strom.Tak byly neimplementovány tabulky symbolů. Operace nad binárním stromem jsme prováděli rekurzivně.Ve struktuře `SymTab` máme 3 struktury pro různé druhy tabulek: tabulka symbolů pro funkce `function`, tabulka symbolů pro proměnné `variable`, a tabulka symbolů pro práci s generátorem kódu `genVariable`.

Při vytváření tabulky symbolů pro proměnné jsme se setkali s následujícím problémem. Proměnné mohou mít stejné názvy, ale se nacházet na různých úrovních. Abychom mohli to rozlišovat, vyvařili jsme nový binární strom na každé úrovni. Tím pádem jsme získali lineární seznam binárních stromů. Každý uzel stromu obsahuje identifikátor, ukazatele na jeho dva podstromy a data. V binárních stromech hledáme pomocí klíče, což je pro nás `token->data`. Implementovali jsme několik funkcí pro práci s tabulkou. To jsou následující funkce: inicializace, přidání nové položky, vložení typu(pro proměnné), přidání vstupních a výstupních argumentů (pro funkci), vyhledání položky, porovnání proměnných, odstranění položky, uvolnění tabulky z paměti.

Pro vestavěné funkce jsme udělali funkce `symTab_for_inbuilt_func` a ručně vytvořili tokeny tak, aby při volání tabulky byly tyto funkce vnořeny.

Funkce pro prací s tabulkou symbolů jsou implementovany ve souboru `symtable.c`. Hlavičkový soubor je `symtable.h`

4 Generování cílového kódu

Funkce pro generování kódu se nachází v souboru `code_generator.c`, a je volána v hlavním běhu `parseru` během syntaktické a sémantické analýzy. Její hlavička se nachází v souboru `code_generator.h`. Spuštění těchto funkcí se nastává při druhém průchodu kódu. Pro práci s generátorem kódu používáme tabulku symbolů `genVariable`. Máme dva průchody těla funkcí. Při prvním průchodu všechny proměnné zapisujeme do tabulky symbolů. Pro vyhýbání opakované definici proměnné, vytýkáme každou novou definici přes DEFVAR před tělem funkcí. Toto pravidlo se používáme pro cyklus “for” a v případě

deklarace několika proměnných se stejnými názvy na stejné úrovni, ale v různých blocích. Při druhém průchodu dochází ke zpracovávání funkcí s již deklarovanými proměnnými.

Nepoužíváme globální rámec. Globálně vyhlašujeme jen podtržítko.

`Code_generator.c` nabízí různé funkcí pro generování samostatných částí programu, to jsou generování funkce "main", generování začátku funkce, generování konce funkce, vytvoření dočasného rámce, volání funkce, deklarace proměnné, funkce pro skoky cyklů "for", "if" atd. Každá funkce je volána ve správném pořadí a se správnými parametry. Při generování funkcí každá funkce má svůj lokální rámec a je tvořena návěštím podle názvu funkce. Před voláním funkce definujeme hodnoty parametrů v dočasném rámci, a po vstupu do funkce přesouváme jeho na zásobník rámců a se stává aktuálním lokálním rámcem. Po provedení funkce, výsledek je uložen do proměnné s návratovou hodnotou funkce. Vestavěné funkce jsou předepsány přímo v jazyce IFJcode20. Jsou vyvolány na začátku programu.

Pro podmíněné skoky generujeme unikátní návěští, jsou generovány ve tvaru `$token->data$for/if$count`, kde `token->data` zajistí rozlišování mezi jednotlivými funkcemi, for/if ukazuje typ cyklu, a `count` je hodnota, která se inkrementuje s každým dalším návěštím ve stejné funkce.

Ve většině případů pracujeme se zásobníkem, ukládáme tam všechny výrazy a parametry. Operace s nimi se provádí na vrcholu zásobníku.

5 Práce v týmu

5.1 Způsob práce v týmu a komunikace

Nejprve jsme vytvořili plán a nějakou strukturu projektu. Každý pátek jsme měli schůzku, kde jsme analyzovali, co bylo provedeno, co je třeba změnit a rozhodnout nad čím pracovat dál. Na jednotlivých úlohách jsme pracovali jednotlivě nebo dvojice členů týmu. A pořad jsme konzultovali mezi sebou.

Komunikace probíhala na začátku osobně, pak prostřednictvím aplikace Discord, kde jsme měli skupinové konverzace nebo psali přímo mezi sebou. Tam jsme probírali a řešili problémy týkající se různých částí projektu.

5.2 Verzovací systém

Pro správu souborů projektu jako verzovací systém jsme zvolili Git. Jako vzdálený repositář jsme používali GitHub.

Každý z nás měl svou vlastní větev. Díky GitHubu jsme mohli pracovat na více úkolech současně. Při změnách jsme si nechali komentáře, co bylo změněno. Po týmovém schválení a otestování jsme tyto úpravy spojovali do hlavní větve.

5.3 Rozdělení práce mezi členy týmu

Práci jsme rozdělili následovně:
Stepaniuk Roman Bc.(xstepa64) - syntaktická analýza, sémantická analýza, vedení tymu, dokumentace,generace kódu
Pastushenko Vladislav (xpastu04) - syntaktická analýza, sémantická analýza, dokumentace
Bahdanovich Viktoryia (xbahda01) - dokumentace, lexikální analýza, generace kódu, oprava chyb
Tomason Viktoryia (xtomas34) - tabulka symbolů, generace kódu, dokumentace, oprava chyb

6 Závěr

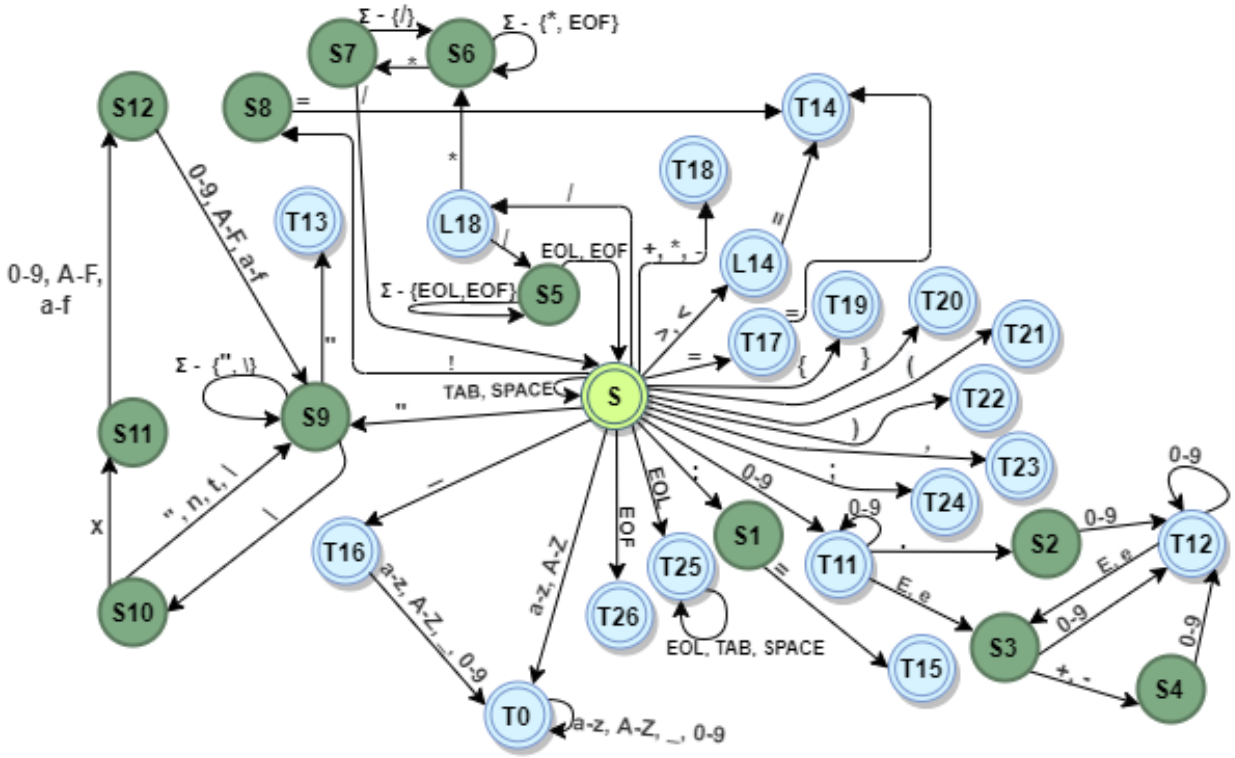
Reference

[1] Slajdy z přednášek předmětu Formální jazyky a překladače

[2] Slajdy z přednášek předmětu Algoritmy

[3] Wiki stránka o binárním stromu
https://cs.wikipedia.org/wiki/Binární_vyhledávací_strom

7 Diagram konečného automatu



LEGEND: S - START STATE
T0 - TOKEN_TYPE_IDENTIFIER/TOKEN_TYPE_INT/TOKEN_TYPE_FLOAT/
TOKEN_TYPE_STRING/TOKEN_TYPE_FOR/TOKEN_TYPE_IF/
TOKEN_TYPE_ELSE/TOKEN_TYPE_FUNC/TOKEN_TYPE_PACKAGE/
TOKEN_TYPE_RETURN/TOKEN_TYPE_COMMAND_FUNCTION
T11 - TOKEN_TYPE_LITERAL_INT
T12 - TOKEN_TYPE_LITERAL_FLOAT
T13 - TOKEN_TYPE_LITERAL_STRING
T14 - TOKEN_TYPE_LOGICAL_OPERATOR
L14 - MORE THAN/LESS THAN (TOKEN_TYPE_LOGICAL_OPERATOR)
T15 - TOKEN_TYPE_DECLARE
T16 - TOKEN_TYPE_UNDERSCORE
T17 - TOKEN_TYPE_EQUATING
T18 - TOKEN_TYPE_MATH_OPERATOR
L18 - SLASH (TOKEN_TYPE_MATH_OPERATOR)
T19 - TOKEN_TYPE_START_BLOCK
T20 - TOKEN_TYPE_END_BLOCK
T21 - TOKEN_TYPE_LEFT_BRACKET

T22 - TOKEN_TYPE_RIGHT_BRACKET
T23 - TOKEN_TYPE_COMMA
T24 - TOKEN_TYPE_SEMICOLON
T25 - TOKEN_TYPE_EOL
T26 - TOKEN_TYPE_EOFILE
S1 - START STATE FOR TOKEN_TYPE_DECLARE
S2 - START STATE FOR TOKEN_TYPE_LITERAL_FLOAT
S3 - START STATE FOR TOKEN_TYPE_LITERAL_FLOAT WITH EXPONENT
S4 - ADDITIONAL STATE FOR TOKEN_TYPE_LITERAL_FLOAT WITH EXPONENT
S5 - START STATE FOR COMMENT
S6 - STATE FOR START AND IGNORING OF BLOCK COMMENT
S7 - STATE FOR END OF BLOCK COMMENT
S8 - START STATE FOR 'NOT EQUAL'(TOKEN_TYPE_LOGICAL_OPERATOR)
S9 - START STATE FOR TOKEN_TYPE_LITERAL_STRING
S10 - STATE FOR ESCAPE SEQUENCE IN TOKEN_TYPE_LITERAL_STRING
S11 - START STATE FOR HEXADECIMAL ESCAPE SEQUENCE IN TOKEN_TYPE_LITERAL_STRING
S12 - SECOND STATE FOR HEXADECIMAL ESCAPE SEQUENCE IN TOKEN_TYPE_LITERAL_STRING
SPACE - Space
TAB - Horizontal Tab
COMMA - Comma
EOL - End Of Line(Line feed)
EOF - End Of File
Σ - the alphabet we are working on(not only allowed printable characters)
Σ - {..} - the alphabet we are working on without one or more characters in brackets
A-F, a-z - a letter from the lists
+, * - a character from the list
0-9 - a digit from the list

8 LL-tabulka

Nonterminal	package	ID	EOL	EOF	func	(	)	{	}	ε	,	if	else	for	return	;	EXPR	:=	eq_symb	LITERAL	int	float64	string
PROGRAM_START	1																						
FUNCTION_CHECK					2																		
INPUT_PARAMETERS		4								3													
INPUT_SINGLE_PARAMETERS		5																					
INPUT_SINGLE_NEXT										6	7												
OUTPUT_PARAMETERS						9				8													
OUTPUT_SINGLE_PARAMETERS																					10	10	10
OUTPUT_SINGLE_NEXT										11	12												
FUNCTION_BODY		19	15					14	13	19		16	17	18	20								
FOR_CONSTRUCTION		21								21													
IF_CONSTRUCTION																	22						
DEFINE_FUNC		24								23													
DEFINE_FUNC_NEXT						27												25	26				
DEFINE_OPERANDS		28																					
DEFINE_OPERANDS_NEXT										29	30												
COUNT_OPERANDS																	31						
COUNT_OPERANDS_NEXT										33	32												
ALLOWED_EOL			34							35													
EXPRESSION_FUNC_ARGUMENTS		36																		36			
EXPRESSION_FUNC_SINGLE_ARGUMENT		37																		38			
EXPRESSION_FUNC_SINGLE_NEXT										40	39												
RETURN_CONSTRUCTION			41														42						
RETURN_CONSTRUCTION_NEXT			44								43												
START_BLOCK_NEW_LINE								45															
TYPE																					46	48	49

9 LL - gramatika

1. <PROGRAM_START> -> package ID EOL <FUNCTION_CHECK> EOF
2. <FUNCTION_CHECK> -> func ID (<ALLOWED_EOL> <INPUT_PARAMETERS>) <OUTPUT_PARAMETERS> { EOL <FUNCTION_BODY> } EOL
3. <INPUT_PARAMETERS> -> ε
4. <INPUT_PARAMETERS> -> <INPUT_SINGLE_PARAMETERS>
5. <INPUT_SINGLE_PARAMETERS> -> ID <TYPE> <INPUT_SINGLE_NEXT>
6. <INPUT_SINGLE_NEXT> -> ε
7. <INPUT_SINGLE_NEXT> -> , ID <TYPE> <INPUT_SINGLE_NEXT>
8. <OUTPUT_PARAMETERS> -> ε
9. <OUTPUT_PARAMETERS> -> (<OUTPUT_SINGLE_PARAMETERS>)
10. <OUTPUT_SINGLE_PARAMETERS> -> <TYPE> <OUTPUT_SINGLE_NEXT>
11. <OUTPUT_SINGLE_NEXT> ->ε
12. <OUTPUT_SINGLE_NEXT> -> , <TYPE> <OUTPUT_SINGLE_NEXT>
13. <FUNCTION_BODY> -> } EOL <FUNCTION_BODY>
14. <FUNCTION_BODY> -> <START_BLOCK_NEW_LINE>
15. <FUNCTION_BODY> -> EOL <FUNCTION_BODY>
16. <FUNCTION_BODY> -> if <IF_CONSTRUCTION>
17. <FUNCTION_BODY> -> else <START_BLOCK_NEW_LINE>
18. <FUNCTION_BODY> -> for <FOR_CONSTRUCTION>
19. <FUNCTION_BODY> -> <DEFINE_FUNC> <FUNCTION_BODY>
20. <FUNCTION_BODY> -> return <RETURN_CONSTRUCTION>
21. <FOR_CONSTRUCTION> -> <DEFINE_FUNC> ; <EXPRESSION> ; <DEFINE_FUNC> { EOL <FUNCTION_BODY>
22. <IF_CONSTRUCTION> -> <EXPRESSION> { EOL <FUNCTION_BODY>
23. <DEFINE_FUNC> -> ε
24. <DEFINE_FUNC> -> <DEFINE_OPERANDS> <DEFINE_FUNC_NEXT>
25. <DEFINE_FUNC_NEXT> -> := <COUNT_OPERANDS>
26. <DEFINE_FUNC_NEXT> -> = <COUNT_OPERANDS>
27. <DEFINE_FUNC_NEXT> -> (<ALLOWED_EOL> <EXPRESSION_FUNC_ARGUMENTS>
28. <DEFINE_OPERANDS> -> ID <DEFINE_OPERANDS_NEXT>
29. <DEFINE_OPERANDS_NEXT> -> ε
30. <DEFINE_OPERANDS_NEXT> -> , ID <DEFINE_OPERANDS_NEXT>
31. <COUNT_OPERANDS> -> <EXPRESSION> <COUNT_OPERANDS_NEXT>
32. <COUNT_OPERANDS_NEXT> -> , <EXPRESSION> <COUNT_OPERANDS_NEXT>
33. <COUNT_OPERANDS_NEXT> -> ε
34. <ALLOWED_EOL> -> EOL <ALLOWED_EOL>
35. <ALLOWED_EOL> -> ε
36. <EXPRESSION_FUNC_ARGUMENTS> -> <EXPRESSION_FUNC_SINGLE_ARGUMENT>)
37. <EXPRESSION_FUNC_SINGLE_ARGUMENT> -> ID <EXPRESSION_FUNC_SINGLE_NEXT>
38. <EXPRESSION_FUNC_SINGLE_ARGUMENT> -> LITERAL <EXPRESSION_FUNC_SINGLE_NEXT>
39. <EXPRESSION_FUNC_SINGLE_NEXT> -> , <EXPRESSION_FUNC_SINGLE_ARGUMENT>
40. <EXPRESSION_FUNC_SINGLE_NEXT> -> ε
41. <RETURN_CONSTRUCTION> -> EOL <FUNCTION_BODY>
42. <RETURN_CONSTRUCTION> -> <EXPRESSION> <RETURN_CONSTRUCTION_NEXT>
43. <RETURN_CONSTRUCTION_NEXT> -> , <EXPRESSION> <RETURN_CONSTRUCTION_NEXT>
44. <RETURN_CONSTRUCTION_NEXT> -> EOL <FUNCTION_BODY>
45. <START_BLOCK_NEW_LINE> -> { EOL <FUNCTION_BODY>
46. <TYPE> -> int
47. <TYPE> -> float64
48. <TYPE> -> string