Projektová dokumentace

Implementace překladače imperativního jazyka IFJ21

Tým 103, varianta I

Kuchta Samuel	**(xkucht11)**	**25% (vedoucí týmu)**
Dovbush Andrii	(xdovbu00)	25%
Dmitriievich Daria	(xdmitr00)	25%
Kramár Denis	(xkrama06)	25%

8. 12. 2021

Obsah

1. Úvod

Cílem projektu bylo vytvořit překladač imperativního jazyka IFJ21 v jazyce C. IFJ21 je zdrojový kód jenž je zjednodušenou podmnožinou jazyka TEAL. Program je konzolová aplikace, která načítá zdrojový kód ze standardního vstupu, zpracuje jej, a na standardní výstup zobrazí cílový mezikód IFJcode21. V případě chyby vrací patřičný chybový kód.

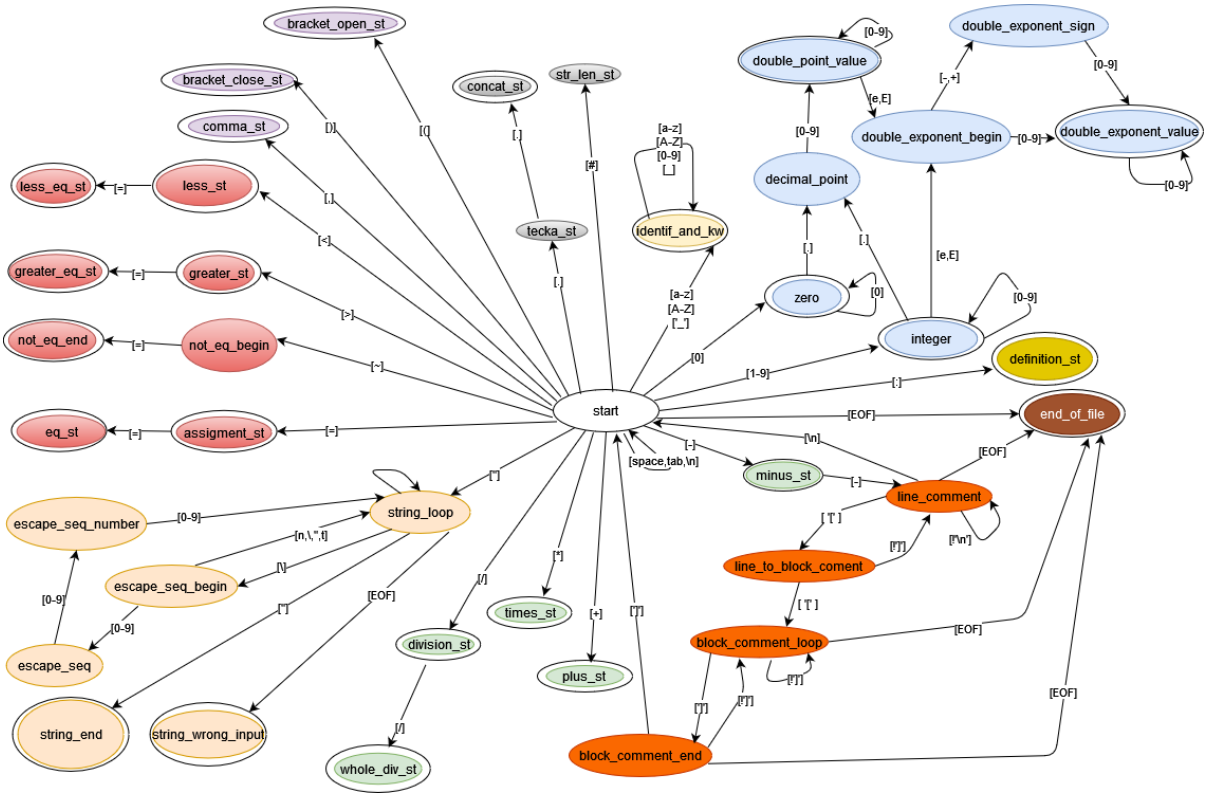
2. Práce v týmu

Projekt jsme začali vypracovávat v půlce října. Na začátku jsme si rozdělili práci, kde jednotlivci měli udělat své části, a nakonec se měl projekt dokončit jako celek. Jako verzovací systém se používal git a github jako vzdálený repozitář. Jako komunikační server jsme používali Discord.

Nejprve byla dokončena Lexikální analýza, poté tabulka symbolů, syntaktická analýza, generátor kódu, a nakonec sémantická analýza.

Rozdelení práce:
Kuchta Samuel: vedoucí týmu, Lexikální analyzátor, Generátor kódu
Dovbush Andrii: Syntaktická analýza, Sémantická analýza
Dmitriievich Daria: Tabulka symbolů, Generátor kódu
Kramár Denis: Syntaktická analýza, Sémantická analýza

3. Diagram LA



Nákres byl vytvořen pomocí webové aplikace draw.io, v repozitáři je také soubor ve kterém je nákres, pokud by tohle nebylo čitelné.

4. LL-gramatika, LL-tabulka a precedenční tabulka

4.1. LL-gramatika

Terminály:
integer, double , string, identifier, left_bracket, right_bracket, comma, division, whole_division, times, plus, minus, EoF, definition, less, less_equals, greater, greater_equals, not_equals, assignment, equals, concat, str_len, key_if, key_else, key_then, key_end, key_do, key_while, key_function, key_return, key_local, key_global, key_nil, key_number, key_integer, key_string, key_require, key_main

Neterminály:
S, Arg_def, Arg_type, Code, Def_arg_list, Def_arg_list_inner, Expression_list, Expression_list _inner, Else_block, Code, Function_call_args, Function_def, Global_def, Global_type, If_block, Identifier_list, Identifier_list_inner, Line_of_code, Local_def, Program, Return_list, Return_statement, Type_def, Types_list, Types_list_inner, While_block, main_func, main_start

Neterminály které vedou k zmeně typu analýzy na precedenční (výrazy):
Expression, condition

1. S -> key_require + "ifj21" + program + main_start + EoF
2. program -> global_def + program
3. program -> function_def + program
4. program -> ε
5. global_def -> key_global + identifier + definition + global_type
6. global_type -> arg_type + type_def
7. global_type -> key_function + left_bracket + def_arg_list + right_bracket + return_list
8. type_def -> assignment + expression
9. type_def -> ε
10. def_arg_list -> arg_def + def_arg_list_inner
11. def_arg_list_inner -> comma + arg_def + def_arg_list_inner
12. def_arg_list_inner -> ε
13. arg_def -> identifier + definition + arg_type
14. arg_def -> ε
15. arg_type -> key_string
16. arg_type -> key_integer
17. arg_type -> key_number
18. function_def -> key_function + identifier + left_bracket + def_arg_list + right_bracket + return_list + code + key_end
19. return_list -> definition + types_list
20. return_list -> ε
21. types_list -> arg_type + types_list_inner
22. types_list_inner -> comma + arg_type + types_list_inner
23. types_list_inner ->
24. code -> line_of_code + code
25. code -> if_block + code

26. code -> while_block + code
27. code -> local_def + code
28. code -> return_statement + code
29. code -> ε
30. line_of_code -> identifier + identifier_contin
31. if_block -> key_if + condition + key_then + code + else_block + key_end
32. else_block -> key_else + code
33. else_block -> ε
34. while_block -> key_while + condition + key_do + code + key_end
35. local_def -> key_local + identifier + definition + arg_type + type_def
36. return_statement -> key_return + expression_list
37. identifier_contin -> left_bracket + expression_list + right_bracket
38. identifier_contin -> identifier_list_inner + assignment + expression_list
39. identifier_list_inner -> comma + identifier + identifier_list_inner
40. identifier_list_inner -> ε
41. expression_list -> expression + expression_list_inner
42. expression_list -> ε
43. expression_list _inner -> comma + expression + expression_list _inner
44. expression_list _inner -> ε
45. main_start -> key_main + left_bracket + right_bracket

4.2. precedenční tabulka

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

Terminaly:
Concat, str_len, plus, minus, division, whole_division, times, l_bracket, r_bracket, identifier, string, integer, double, key_nill, less, less_equals, greater, greater_equals, not_equals, equals, comma

Neterminály:
Condition, expression, str_len_arg

1. Str_len_arg -> string
2. Str_len_arg -> identifier
3. Str_len_arg -> l_bracket + expression + r_bracket
4. Condition -> expression + less + expression
5. Condition -> expression + less_equals + expression
6. Condition -> expression + greater + expression
7. Condition -> expression + greater_equals + expression
8. Condition -> expression + not_equals + expression
9. Condition -> expression + equals + expression
10. Expression -> identifier + l_bracket + expression_list + r_bracket
11. Expression -> identifier
12. Expression -> string
13. Expression -> integer
14. Expression -> double
15. Expression -> key_nil
16. Expression -> str_len + str_len_arg
17. Expression -> l_bracket + expression + r_bracket
18. Expression -> expression + concat + expression
19. Expression -> expression + plus + expression
20. Expression -> expression + minus + expression
21. Expression -> expression + division + expression
22. Expression -> expression + whole_division + expression
23. Expression -> expression + times + expression

5 Implementace

5.1. Speciální algoritmy a datové struktury

„knihovny.h a knihovny.c“ :
Pro lexikální analyzátor, a generátor kódu bylo vytvořeno dynamické pole znaků *inflatable_text* pro ukládání řetězce. z důvodu neznalosti délky identifikátorů, řetězců a výsledné délky vygenerovaného kódu při kompilaci programu.
Toto pole obsluhují obslužné funkce. Funkce *inflatable_text_init* allokuje paměť o velikosti definované pomocí *INC_INFLATABLE_TEXT_LENGTH* na 16 znaku a nastaví proměnnou *alloc_size*.
Funkce *inflatable_text_add_char* ukládá znak do pole a pokud je pole plné, pokusí se jej zvětšit a uložit do něj znak. Pokud bylo dokončeno načítání identifikátoru nebo stringu, obsah bufferu je zkopírován do patřičného tokenu pomocí funkce *store_identif_or_string_to_token,* funkce také alokuje paměť pro uložení řetězce do tokenu.
Pro uvolnění paměti *inflatable_textu* slouží funkce *inflatable_text_free.*

„lexikalni_analyza.h“:
Dále byla vytvořena struktura LA_token která je výstupem lexikální analýzy.
Struktura obsahuje proměnnou *token_type* která určuje typ tokenu, což je informace o tom, jakou informaci token ponese (např operator, klíčové slovo, celočíselná hodnota nebo řetězec znaků) a dále atribut union *token_data,* ve kterém je uložená informace co přesně bylo načteno.
Union podle typu tokenu může být ukazatel na řetězec (pokud byl token identifikátor nebo řetězec), hodnota typu double, pokud vstupem bylo desetinné číslo. Dále celočíselná hodnota pokud byla na vstupu. Pokud typ tokenu bylo klíčové slovo, typ *key_word_value* nám určí, jaké bylo klíčové slovo.
Token se vytváří pomocí funkce *create_token*, která vrací ukazatel na LA_token
Funkce *delete_token* uvolní paměť, která byla potřeba k uložení tokenu a pokud typ tokenu byl identifikátor nebo string, uvolní také paměť, která byla potřebná pro jeho uložení.

„SA_tree.h“ a „SA_tree.c“:
Pro syntaktickou analýzu, byla vytvořena struktura syntaktického stromu *SATree*, díky které syntaktická může tvořit jednotlivé terminály/neterminály, a připojovat je jako uzly *TreeNode* do syntaktického stromu, kde neterminální uzel může mít víc potomků, adresovatelných pomocí struktury *Node_Vector (první, poslední), a pak Node_Vector_Node* (další, předešlí)

5.2. Lexikální analýza

Lexikální analýza se z velké části odehrává ve funkci *get_token.* Tato funkce je naimplementována jako konečný deterministický automat v souboru *„lexikalni_analyza.c“*, podle předem vytvořeného diagramu.
Automat je naimplementován jako nekonečný cyklus, ve kterém je switch, a podle stavu, ve kterém se nachází a znaku, který byl přečtený ze vstupu, se určuje přechod mezi stavy.
Pokud přečteme znak, který neodpovídá žádnému pravidlu pro přechod, vyhodnotíme, zda jsme v konečném stavu, a podle toho ukončíme práci (uložíme identifikátor, číselnou hodnotu apod. do tokenu nebo nastavíme chybu do proměnné *exit_code* a vrátíme false).

5.3. Tabulka Symbolů

Tabulky symbolů jsou implementovány jako binární vyhledávací stromy v souborech *symtable.c* a *symtable.h.*
Každý uzel stromu obsahuje kromě identifikátoru a ukazatelů na jeho dva podstromy také data. V nich je uložen typ identifikátoru, informace, jestli byl identifikátor již definován, a v případě identifikátoru funkce taky ukazatel na lokální tabulku symbolů, počet parametrů, pole typů argumentů a pole typů returnů.
Vyhledávání v binárních stromech je implementováno pomocí klíče, kterým je identifikátor.

5.4. Syntaktická analýza

Nejdůležitější částí celého programu je syntaktická analýza, implementovaná v souborech *„syntakticka_analyza.c“* a *„syntakticka_analyza.h“*. Až na výrazy se syntaktická analýza řídí LL – gramatikou a metodou rekurzivního sestupu podle pravidel v LL – tabulce. Každé pravidlo má svou vlastní funkci, která dostává přes parametr ukazatel na otcovský uzel *SA_tree_node*.

Syntaktická analyza si tvoří Syntaktický strom tak, že volá Lexikální analýzu pomocí *get_token(),* a na základě tokenu, aplikuje pravidlo podle deterministické gramatiky. Pravidla se mohou skládat z libovolného počtu neterminálů a terminálů. Když funkce aplikující pravidla narazí na terminál, tak pro něj vytvoří uzel pomocí makra MANAGE_TERM, které do něj vloží informace tokenu. Když narazí na neterminál, tak také vytvoří uzel pomocí makra MANAGE_NON_TERM, které zavolá obslužnou podfunkci, která se o uzel rekurzivně postará.

Když Syntakticka analyza proběhne bezchybně, tak se nad vytvořeným stromem zavolá sémantická analýza, a nakonec se pomocí průchodu výsledným stromem budou pro jednotlivé uzly volat ekvivalentní funkce generátoru kódu, které na stdout vytisknou výsledný program.

Precedenční analýza výrazů je v syntaktické analýze definována a volána stejně jako ostatní pravidla v LL – gramatice. při zpracovávání výrazů je použita precedenční tabulka, a vlastní gramatika.

5.5. Sémantická analýza

Ve struktuře *psData* jsou uloženy data pro konkrétní uzel v stromu. (podrobněji popsáno v 5.3. Tabulka Symbolů). Implementovaná sémantická analýza v souborech “*semanticka_analyza.c*” a “*semanticka_analyza.h*”. Jsou kontrolovány následující vlastnosti kódu: definice proměnných, počet a typ argumentů pro function call, počet a typ výrazů pro return statement.

Když je zavolán sémantický analyzátor, vestavěné funkce jsou přidány do tabulky symbolů, aby byly deklarovány předem. Ve výrazech je zkontrolována definice proměnných a funkcí vyhledaváním v lokální tabulce symbolů. V připadě, že identifikátor nebyl nalezen v lokální tabulce symbolů, bude vyhledáván v globalní tabulce symbolů. Během kontroly výrazů dochází k porovnání typů a případně jejich převodu na požadovaný typ (*integer->number, number->integer*) .

Existuje i speciální typ *nul_val*. Pokud se celý výraz skládá z jednoho tokenu *nul_val*, výraz vrátí *nul_val*. Jestli je výraz sestaven z 2 a více tokenů, tak když jeden z nich je *nul_val*, tak bude vrácen *exit_code* = SEM_NIL_ERR (error č. 8). Kontrola *Condition* bloků umožní, že nějaka čast Condition bude *nil_val*, ale v tomto připadě umožní jenom poůžití == nebo ~= operatorů.

5.6. Generování kódu

Generuje mezikód do vnitřní paměti programu pomocí struktury *inflatable_text*, a po skončení generování je vypsán na stdout pomocí funkce *code_gen_flush*. Skládá se ze souboru hlavičkového „*generator_mezikodu.h*“ a implementačního „*generator_mezikodu.c*“.
Jednotlivé části generování:

Začátek: Vygenerovaný začátek mezikódu, definice globálních proměnných na GF, vytvoření vestavěných funkcí a skok do funkce „*main*“.

Výrazy: Ukládané na datový zásobník, na jehož vrcholu se dělají určité aritmeticko logické operace. Výsledky operací se ukládají do proměnných na globální rámec.

Návěstí: Ve tvaru „*function_id% label_depth% label_index“,* kde *function_id* je identifikátor pro rozlišení návěští jednotlivých (unikátních) funkcí. *label_depth* je hloubka zanoření jednotlivých návěští pro oddělení podmínek a cyklů. *label_index* je index návěstí v dané funkci pro oddělení více návěští v jednom bloku.

Funkce: Funkce mají nadefinován jejich lokální rámec, své počáteční návěstí, výchozí návratovou hodnotu, na základě datového typu uloženou na lokální rámec LF, ze kterého se po odchodu z funkce stane dočasný rámec TF. Po zavolání funkce se hodnoty parametrů uloží do lokálního rámce. Při návratu z funkce se hodnota uloží do návratové proměnné a skočí se za volání funkce.

5.7. Main

Zavolá vygenerování hlavičky mezikódu, a zavolá syntaktickou analýzu, vytiskne zbytek mezikódu.

6. Závěr

projekt jsme začali řešit, až jsme získali dostatek znalostí o tvorbě jednotlivých komponent na přednáškách IFJ.

Náš tým jsme měli sestaven velmi brzy, byli jsme již předem domluveni na komunikačních kanálech, osobních schůzkách a na používání verzovacího systému, tudíž jsme s týmovou prací neměli žádný problém, a pracovalo se nám společně velmi dobře.

Bohužel jsme nestihli dořešit volání generátoru v sémantické analýze pro jednotlivé části programu.

Tento projekt nám celkově přinesl spoustu znalostí ohledně fungování překladačů, prakticky nám objasnil probíranou látku v předmětech IFJ a IAL a přinesl nám zkušennosti s projekty tohoto rozsahu.