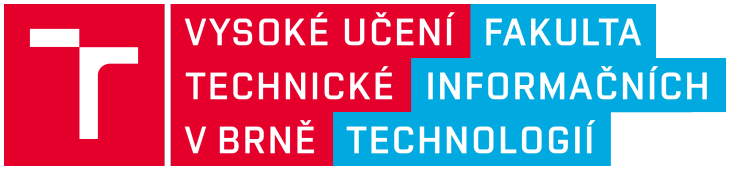


Dokumentace k projektu do předmětů IFJ a IAL
Implementace kompilátoru jazyka IFJ23 do IFJcode23
Tým xvetlu00, varianta - TRP-izp
Implementovaná rozšíření: FUNEXP

Vedoucí: Nikita Vetluzhskikh xvetlu00 25%
Neonila Mashlai xmashl00 25%
Murad Mikogaziev xmikog00 25%
Oleg Borshch xborsh00 25%

December 6, 2023

Obsah

Úvod

Cílem našeho projektu bylo implementovat překladač v programovacím jazyce C pro překlad jazyka IFJ23 do jazyka IFJcode23. Jazyk IFJ23 je podmnožinou programovacího jazyka Swift.

Lexikální analýza

Za prvé, abychom implementovali plnou funkční lexikální analýzu, museli jsme udělat FSM, které lze nalézt v dodatcích.

Hlavní funkcí souboru **scanner.c** (název našeho lexikálního analyzátoru) je funkce **next_token()** pro získání tokenů ze stdin po jednom. Vrací strukturu o každém tokenu, která obsahuje informace jako typ tokenu, číslo řádku, ve kterém se vyskytuje, a atribut tokenu (celé číslo, řetězec, desetinné číslo, klíčové slovo).

Lexikální analyzátor čte symboly jeden po druhém, dokud není přečten jiný symbol než bílý znak, poté začne analyzovat přečtený symbol. Hlavní kódová část je součástí přepínače s případy pro kontrolu přečteného znaku, pokud se jedná o posloupnosti znaků, které se v IFJ23 nemohou vyskytnout, vypíše chybové hlášení 1, jinak získá informaci o tokenu a zapíše jej do struktury. Případy jsou stejné co jsou zobrazené v našem FSM.

K dispozici jsou také čtyři pomocné funkce. Funkce **create_token**, která se používá ve funkci **next_token** k vytvoření struktury a přidělení místa pro každý načtený token, funkce **delete_token**, která se používá k uvolnění paměti, pokud načtený token vrátí chybové hlášení, funkce **single_token**, která se používá k zápisu informace o tokenu do struktury a funkce **scanning_finish_with_error**, která se používá pro chybové hlášení.

Syntaktická analýza

Syntaktická analýza má dvě části.

Prediktivní syntaktická analýza

První z nich je prediktivní syntaktická analýza, která je implementována v **parser.c** na základě LL-tabulky a LL-pravidel. LL-tabulka funguje pomocí zásobníku.
Hlavní funkcí je funkce **analyse()**, která zahajuje a řídí celý proces analýzy. Toto jsou makra používaná v kódu pro zjednodušení a zpřehlednění opakujících se úkonů:
GET_TOKEN(): Makro načte další token ze vstupního proudu. Pokud není token nalezen, funkce vrátí chybový kód.
CHECK_RULE(): Toto makro vyvolává dané pravidlo (funkce) pro analýzu a kontrolu syntaxe a sémantiky. Pokud pravidlo vrátí chybu, makro vrátí tento chybový kód.
VERIFY_TOKEN(): Makro nejprve načte token a poté ověří, zda je typu t_token. Pokud není, vrátí chybový kód pro syntaktickou chybu.
INSERT_SYM(): Makro vkládá symbol do tabulky symbolu. Pokud není dostupná nebo dojde k interní chybě, vrátí příslušný chybový kód. Pokud není symbol nalezen nebo dojde k jiné chybě, vrátí chybový kód sémantické chyby.

Precedenční syntaktická analýza

Druhá je precedenční syntaktická analýza, v našem kódu je implementována jako součást souboru **semantics.c**.
Precedenční syntaktický analyzátor, stejně jako prediktivní verze, využívá zásobník a speciální tabulku – v tomto případě precedenční tabulku. Tato tabulka obsahuje symboly pro určení vztahu mezi

operandy, jako jsou "<", ">", "=", a prázdné sloty pro neplatné kombinace tokenů. Analyzátor se opírá o funkci **expression()**, která zajišťuje posunutí tokenů (shift) a zápis do hashovací tabulky, a funkci **reduce()**, jež aplikuje pravidla pro redukci tokenů na zásobníku. Funkce **reduce()** navíc generuje instrukce pro operátory a díky rozšíření FUNEXP také zpracovává volání funkcí, což je důvod, proč tyto elementy nejsou zahrnuty přímo v LL gramatice a tabulce. Další důležitou funkcí je **get_index()**, která určuje odpovídající řádek a sloupec v precedenční tabulce pro dané tokeny. Samotný výběr pravidla pro redukci je řízen funkcí **reduce()** a závisí na specifickém součtu hodnot symbolů na zásobníku. Pokud součet neodpovídá žádnému pravidlu a nejde o funkci, dojde k syntaktické chybě typu 2.

Sémantická analýza

Soubor **semantics.c** je implementace sémantické analýzy.
Hlavní funkce a jejich funkčnost jsou:
expression: Hlavní funkce pro analýzu výrazů. Řídí celý proces analýzy výrazů pomocí precedenční tabulky a zásobníku. Tato funkce zajišťuje, že syntakticky správné výrazy jsou také sémanticky správné.
reduce: Funkce pro redukci výrazů na základě pravidel definovaných v precedenční tabulce. Tato funkce odstraňuje symboly ze zásobníku a nahrazuje je novými, vyhodnocenými symboly.
check_semantics: Funkce, která kontroluje sémantickou správnost výrazů podle definovaných pravidel. Tato funkce se zabývá typy operandů a určuje, zda jsou kombinace typů v operacích platné.
check_rule: Funkce pro určení, jaké pravidlo se má použít v kontextu redukce. Tato funkce rozhoduje, jak se mají symboly na zásobníku zpracovávat.
check_param a **check_func_call**: Funkce pro kontrolu parametrů a volání funkcí. Tyto funkce zajišťují, že parametry předávané do funkcí odpovídají očekávaným typům a počtům.
Kód také obsahuje definice pomocných datových struktur a typů, jako jsou enumerace pro indexy precedenční tabulky a pravidla pro redukci, a makra pro časté operace, například pro získání tokenu nebo ověření tokenu.

Řetězec s dynamickou velikostí

Struktura **string_ptr** zahrnuje obsah řetězce, pozici posledního indexu a počet bytů který má alokovaný. V případě naplnění řetězce se alokuje dvakrát více místa v paměti. Tento řetězec používáme pro uložení hodnoty lexému, nebo názvu proměnné v scanner.c. Taky použili jsme funkci **string_concat** v generátoru, pro uložení celého kódu v globální proměnné pro tisk na konci generace.

Tabulka symbolů

Soubor **hash.c** implementuje tabulku symbolů pro kompilátor pomocí hashtable s implicitním řetězením pro řešení kolizí.
Datové struktury: **item_data** (data spojená s každým symbolem v tabulce symbolů), **symbol** (název, item_data a ukazatele na další symbol v případě kolize), **hash_table** (tabulka symbolů se zadanou velikostí a polem ukazatelů symbolů)
Zde jsou hlavní funkce pro vytvoření (**create_hash_table**) a zničení (**destroy_hash_table**) hashtable, vytvoření výchozí struktury (**create_default_item**) a funkce pro vložení (**insert_symbol**), nalezení (**find_symbol**) a odstranění (**remove_symbol**) symbolů a také funkce pro výpočet indexového řetězce v hashtable pomocí jednoduchého algoritmu (**hash**).
Celkově tato implementace poskytuje základní operace pro správu tabulky symbolů, takže je vhodná

pro použití v syntaxi a sémantické analýze kompilátoru. Hashtable využívá implicitní řetězení k efektivnímu řešení kolizí.

Zásobník symbolů

Soubor **stack.c** poskytuje implementaci zásobníku používaného pro syntaxi a sémantickou analýzu v kompilátoru. Zásobník definovaný strukturou **t_stack** ukládá prvky typu **t_stack_elem**, z nichž každý obsahuje **item_data** a **Precedence_table_symbol**. Zásobník podporuje standardní operace, jako je inicializace, push, pop a uvolnění paměti. Zahrnuje další funkce, jako je nalezení horního terminálu, zatlačení položky za horní terminál, získání řetězcové reprezentace symbolu tabulky priorit, počítání prvků, přístup k hornímu prvku a tisk všech symbolů v zásobníku. Tento zásobník je navržen tak, aby usnadnil analýzu výrazů a symbolů během procesu kompilace.

Zásobník tabulek

Vytvořili jsme soubor **table_stack.c** který implementuje zásobník místních tabulek, abychom mohli pracovat s proměnnými v různých úrovních v cyklu. Nachází se v souboru **table_stack.c**, hlavní funkce byly použity k inicializaci (**table_stack_init**) a jsou založeny na funkcích zásobníku, jako je pop, push, top, free.

Generator

Generování kódu je implementováno v soubor generator.c. Kód je generován pomocí instrukcí generovaných v parser.c.
Hlavní funkce generátoru jsou:
bool generator_start(), která spouští proces generování kódu.
bool generator_end(), která dokončí proces generování kódu.
void generator_builtin(), která generuje kód pro vestavěné funkce nebo operace.
void codegen_flush(), která vygenerovaný kód odešle na výstup.
Existuje mnoho dalších funkcí téměř pro každý návod, ale tyto jsou hlavní.

Práce v týmu

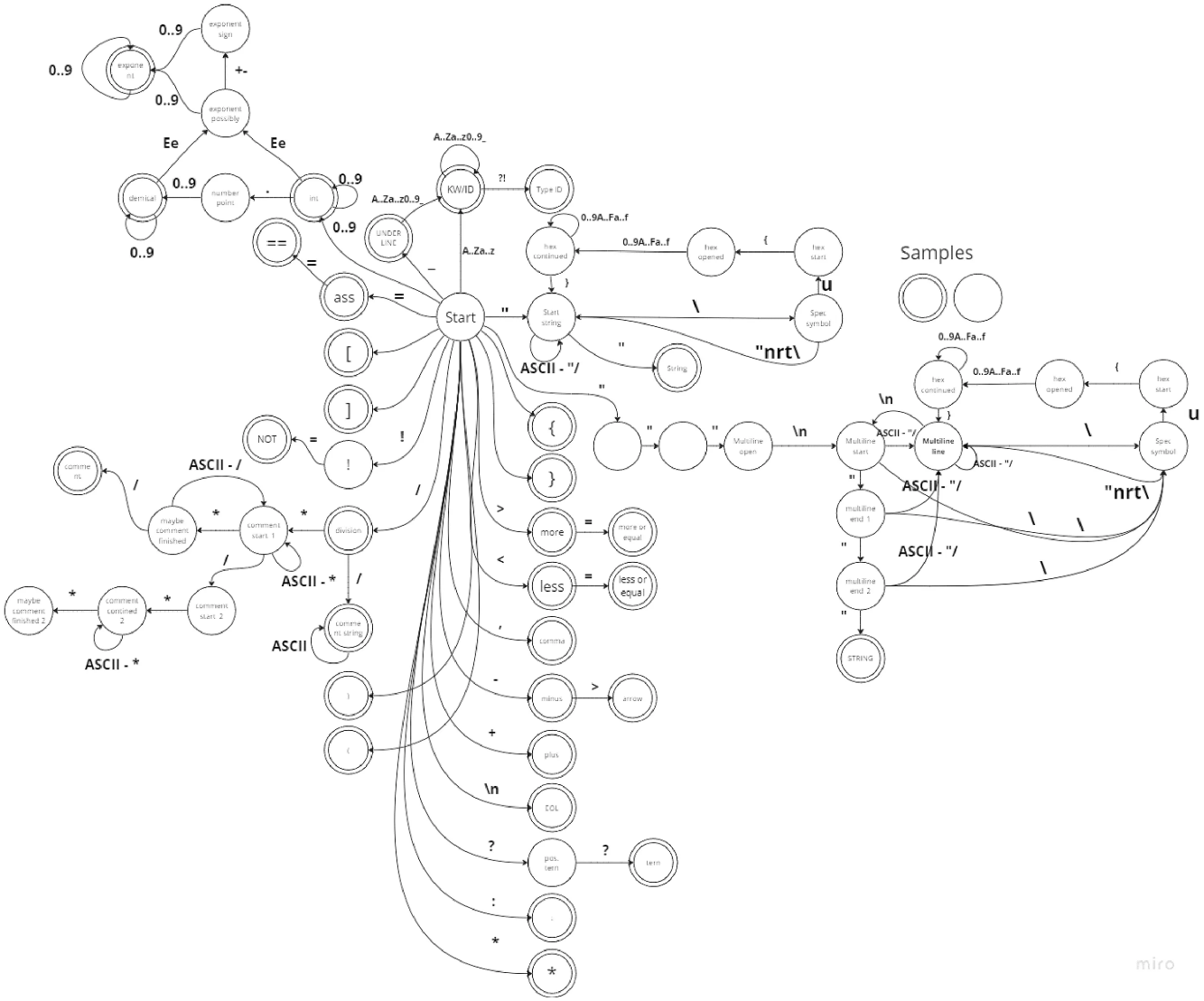
Měli jsme každý týden schůzky offline a někdy jsme se potkali na Discordu. Jako komunikační aplikace jsme také používali Discord a Telegram. Pro sdílení kódu jsme použili Github.

Rozdělení práce

FSM a LL-tabulku vyrobili všichni členové týmu.
Nikita Vetluzhskikh :
lexikální analýza, precedenční syntaktická analýza, prediktivní syntaktická analýza, sémantická analýza, Makefile, zásobník symbolů.
Neonila Mashlai :
lexikální analýza, generátor, dokumentace, zásobník tabulek.
Murad Mikogaziev :
Precedenční syntaktická analýza, prediktivní syntaktická analýza, sémantická analýza, řetězec s dynamickou velikostí, zásobník symbolů.
Oleg Borshch :

sémantická analýza, tabulka symbolů, generátor, řetězec s dynamickou velikostí, tabulka symbolů.

FSM


exponent sign
exponent
exponent possibly
0..9
+-
Ee
decimal
number point
int
A..Za..z0..9_
KW/ID
Type ID
?!
UNDER LINE
A..Za..z
_
0..9A..Fa..f
hex continued
hex opened
hex start
{
u
Start
Start string
Spec symbol
"nrt\
String
ASCII - "/
ass
=
==
[
]
NOT
!
{
}
more
more or equal
less
less or equal
comma
minus
arrow
plus
EOL
pos. tern
tern
:
*
division
/
comme nt
maybe comment finished
comment start 1
ASCII - /
ASCII - *
comment start 2
comment continued 2
maybe comment finished 2
comme nt string
ASCII
Samples
Multiline open
Multiline start
Multiline line
multiline end 1
multiline end 2
\n
STRING
miro

LL pravidla

1 <program> -> <stm> EOF

2 <stm> -> var + let id : <var_type> = <expression> \n <stm>

3 <stm> -> var + let id = <expression> \n <stm>

4 <stm> -> var + let id : <var_type> \n <stm>

5 <stm> -> func_id(<call_params>) \n <stm>

6 <stm> -> id = <expression> \n <stm>

7 <stm> -> func func_id (<func_params>) -> <var_type> { <stm> <return> } \n <stm>

8 <stm> -> func func_id (<func_params>) { <stm> <return_void> } \n <stm>

9 <stm> -> if (<expression>) { <stm> } \n <stm>

10 <stm> -> if (<expression>) { <stm> } else { <stm> } \n <stm>

11 <stm> -> while (<expression>) { <stm> } \n <stm>

12 <stm> -> <return> + <return_void>

13 <stm> -> ε

14 <func_params> -> ε

15 <func_params> -> var_name var_id: <var_type> <func_params_not_null>

16 <func_params> -> _ var_id: <var_type> <func_params_not_null>

17 <func_params_not_null> -> , <func_params>

18 <func_params_not_null> -> ε

19 <call_params> -> var_name : var_id <call_params_n>

20 <call_params> -> var_id <call_params_n>

21 <call_params_n> -> , <call_params>

22 <call_params_n> -> ε

23 <var_type> -> String

24 <var_type> -> Int

25 <var_type> -> Double

26 <var_type> -> String?

27 <var_type> -> Int?

28 <var_type> -> Double?

29 <nil_flag> -> ! + ε

30 <return> -> return <expression> <nil_flag>

31 <return_void> -> return

32 <return_void> -> ε

LL-tabulka

	var	let	id	func	if	while	return	_	,	String	Int	Double	String?	Int?	Double?	!
program	2,3,4	2,3,4	5,6	7,8	9,10	11	12									
stm	2,3,4	2,3,4	5,6	7,8	9,10	11	12									
func_params			15					16								
func_params_not_null									17							
call_params			19,20													
call_params_n									21							
var_type										23	24	25	26	27	28	
nil_flag																29
return							30									
return_void							31									

Precedenční tabulka

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