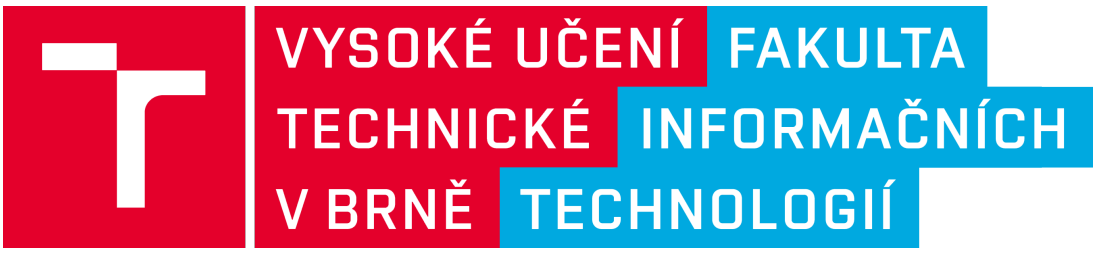


Dokumentace implementace překladače jazyka IFJ24

Tým xbircka00, varianta TRP-izp

Vedoucí: Andrea Birckmannová xbircka00 33%
Jan Štefan Hodák xhodakj00 34%
Tomáš Luštík xlusti03 33%
Barbora Šturcová xsturcb00 0%

4. prosince 2024

Obsah

1 Úvod

1.1 Popis projektu

Tento dokument popisuje implementaci týmového projektu do předmětů Formální jazyky a překladače a Algoritmy. Zadáním bylo vytvořit kompilátor, který přeloží kód v jazyce IFJ24, podmnožině jazyka Zig, do cílového jazyka IFJcode24

1.2 Struktura projektu

Projekt se skládá z několika různých částí a je rozložen do více skriptů.
abst.c/.h Soubory obsahující definici abstraktního sémantického stromu a operace nad ním.
codegen.c/.h Soubory pro generování výsledného kódu IFJcode24 z abstrakního syntaktického stromu
DLL_precedence.c/.h Subory obsahující Dvojitě provázaný seznam upravený pro využití precedenční analýzou
DLL_symtable.c/.h Subory obsahující Dvojitě provázaný seznam upravený pro využití tybulkou symbolů
lex_analyzer.c/.h Soubory obsahující implementaci lexikálního analyzátoru.
main.c Vstupní bod programu, spouští parser a následně generaci kódu
parser.c/.h Hlavní část programu, řídí celou analýzu a zároveň vytváří syntaktický strom.
precedence.c/.h Soubory obsahující precedenční syntaktickou analýzu pro výrazy
symtable.c/.h Soubory obsahující definice tabulky symbolů a operací nad ní.
utils.c/.h Soubory obsahující pomocné funkce

2 Implementace

2.1 Lexikální analýza

Lexikální analýza je naimplementována na základě konečného automatu, jehož návrh lze vidět v obrázku *A.1*. Pomocí konečného automatu je vizualizováno načítání jednotlivých symbolů ze vstupního zdrojového kódu a jejich následné spracování na jednotlivé tokeny. Názvy jednotlivých stavů jsou odvozeny od toho jaký typ tokenu načítají a jestli jsou ukončující. Tudíž stav MORE_F bude ukončující stav (označení F) a bude načítat symbol 'větší'.

Lexikální analýza je volána pokaždé, co je při syntaktické analýze potřeba zpracovávat další token. Jednotlivé symboly se vybírají ze zdrojového kódu pomocí funkce getc(soubor). Díky tomuto je taky možné podle potřeby symboly vracet funkcí ungetc(aktuální symbol, soubor).

Pro zpracování jednotlivého tokenu byla vytvořena struktura Token, která obsahuje typ tokenu, který je typu TokenType, což je enum obsahující všechny možné typy tokenů, a ukazatel na jeho hodnotu, která je v tomto případě reprezentována ukazatelem na char. Zatímco se načítají jednotlivé symboly jsou do pomocné proměnné str tyto symboly ukládány. Poté co se ukončí zpracování jednoho tokenu, alokuje se nová paměť pro další pomocnou proměnnou, která je potom předána do hodnoty tokenu.

Symboly jsou iterativně zpracovávány pomocí switch case a cyklu while a poté jsou pomocí různých podmínek ukládány do již zmíněné pomocné proměnné str, tak, aby ve výsledné hodnotě nebyly například symboly komentáře nebo symboly \\ v víceřádkovém řetězci.

2.2 Syntaktická analýza

Syntaktická analýza je implementována ve dvou částech, v parseru, kde se spracovává téměř vše pomocí rekurzivního sestupu (viz *A.2*) a v precedenční analýze, která spracovává výrazy. Parser kontroluje správnou syntaxi pomocí rekurzivního sestupu. První pravidlo, od kterého kontola začíná je `import`, které ověří správné importování `ifj24.zig` a následně pravidlo `commands`, ze kterého je možné následně projít celý vstupní kód. Pokaždé, kdy parser využije aktuální token zavolá funkci `getNextToken`, která vrátí následující Token, podle kterého se určí jaké pravidlo využít dále. Ve chvíli, kdy parser načte Token `EOF`, syntaktická kontrola končí. Pokud někde při průchodu kódu načtený token nebude odpovídat aktuálnímu pravidlu, kotrola končí s návratovým kódem 2 (Syntaktická chyba). Ve chvíli kdy parser narazí v pravidle na výraz, řízení přebírá precedenční analýza, která výraz zpracuje a následně vrací kontrolu parseru.

2.2.1 Precedenční syntaktická analýza

Precedenční analýza přebírá funkci parseru po dobu, kdy se nachází ve výrazu. Od parseru dostane informace o očekávaném typu, uzel, ke kterému se má připojit abstarktní syntaktický strom výrazu, a aktuálním rozsahu (scope) kvůli sémantické kontrole výrazu. Kontroluje jak syntaxi, tak sémantiku. Výraz je zpracováván na základě precedenční tabulky (viz *A.4*) pomocí obousměrně vázaného seznamu (Double Linked List) a následně redukován pomocí redukčních pravidel (viz *A.5*).

Precedenční analýza načítá tokeny pomocí lexikálního analyzátoru a průběžně je zpracovává. Načítání tokenů je přerušeno při prvním neočekávaném tokenu. Poté se dokončí zpracování výrazu a pokud se nevyskytla žádná chyba, činnost je předána zpět parseru.

Pro zpracování výrazů a jejich redukci je použit upravený obousměrně vázaný seznam, který nahrazuje funkci zásobníku popsaného na přednáškách. Vstupní výraz je na základě precedenční tabulky vložen do seznamu. Po ukončení zpracování výrazu by měl v seznamu zůstat pouze jeden neterminál E na vrcholu, který obsahuje odkaz na kořen stromu daného výrazu. Tento kořen stromu je připojen ke zbytku stromu ještě před opuštěním precedenční analýzy a předáním funkce parseru.

2.3 Sémantická analýza

Sémantická analýza je prováděna pomocí dvou průchodů kódem. Při prvním průchodu jsou pouze hledány definice funkcí, které jsou následně vloženy do globální tabulky funkcí `var_htable` společně s jejich návratovým typem a vyžadovynými parametry.

Při druhém průchodu kódem je sémantika kontrolována ve stejnou chvíli jako syntaxe. Sémantická kontrola při průběhu probíhá většinově komunikací s tabulkou symbolů (viz*3.1*). Při vytvoření nového rozsahu, se v aktuální funkci vytvoří nová tabulka `var_htable` do které se ukládájí nově vytvořené proměnné do chvíle, kdy daný rozsah končí a tabulka se maže. Před mazáním tabulky vždy proběhne kontrola, zda byly všechny proměnné využity, byla využita modifikovatelnost var.

2.4 Generování kódu

Generování kódu probíhá pomocí abstraktního binárního stromu. Strom je naplněný podle různých podmínek a poté předán do funkce generování kódu. Strom je tvořený uzly, které jsou složené z několika prvků: hodnoty uzlu (toto obsahuje buď názvy nebo číselné hodnoty), typ uzlu, pro který byl vytvořen seznam všech typů `NodeType`, a následně levý a pravý potomek tohoto uzlu. Strom se naplňuje pomocí funkcí `set_left_child` a `set_right_child`. Typy uzlů obsahuje například také typ uzlu ABST_NEWLINE, který označuje začátek nového řádku kódu původního zdrojového souboru.

Pomocí rekurzivní funkce traverze binárního stromu Preorder, se ukládají do jednoduché datové struktury `abst_items` pouze uzly typu NEWLINE a pomocí jednoduchého cyklu for se zpracují všechny tyto uzly postupně. Zpracování uzlů probíhá pomocí funkce switch case, která je v cyklu while a podle typu jednotlivých uzlů se zde generuje výsledný kód pomocí manipulace potomků uzlů.

Dále je pro účely generování kódu také vytvořená a používána datová struktura `LabelStack`, která zajišťuje korektní generování vnořených cyklů a případných podmíněných skoků. pomocí funkcí push, pop a peek. K tomuto se taktéž váže datová struktura `LabelInfo`, která obsahuje číslo návěští pro skoky a také o jaký typ návěští se jedná.

3 Datové struktury

3.1 Tabulka symbolů

Pro implementaci tabulky symbolů jsme si vybrali možnost tabulku s rozptýlenými položkami s implicitním zřetězením položek. Tabulku symbolů jsme se rozhodli implementovat pomocí dvou různých hashovacích tabulek, a to `func_htable` a `var_htable`. Jelikož v jazyce IFJ24 nejsou podporovány zanořené definice funkcí, a zároveň ani globální proměnné, tabulka `func_htable` slouží jako globální tabulka, do které jsou uloženy všechny funkce při prvním průchodu zdrojového souboru.

Funkce zapsané v tabulce jsou struktury, které mimo jiné obsahují dvojitě provázaný seznam, kde každý prvek seznamu ukazuje na tabulku `var_htable`. Tyto tabulky slouží pro uchovávání lokálních proměnných a zároveň také k určování jejich rozsahu platnosti v kódu. To je prováděno tak, že každá z tabulek obsahuje pouze proměnné, které jsou definovány v daném rozsahu. Při každém zanoření v kódu, například po použití if nebo while, se přidává nová tabulka na konec seznamu, která je platná pouze v tomto rozsahu a při vynoření se poslední tabulka uvolňuje.

3.2 DLList pro tabulku symbolů

Jak je zmíněno výše tento seznam je používaný pro uchovávání tabulek proměnných v každém rozsahu. Je implementovaný v souborech `dll_symtable.c` a `dll_symtable.h`. Jedná se o velmi zredukovanou verzi dvojitě provázaného seznamu upraveného a rozšířeného pro využití tabulkou symbolů.

3.3 DLList pro precedenční analýzu

Při precedenční analýze byl zásobník nahrazen upraveným dvojitě provázaným seznamem z důvodu jednodušší práce s ním. Je implementovaný v souborech `dll_precedence.c` a `dll_precedence.h`. Tato varianta má upravené funkce `DLL_DeleteLast` a `DLL_GetLast` tak aby vracely vymazanou respektive poslední položku z listu. Navíc je přidaná funkce `DLL_TopTerminal`, která vrátí ukazatel na terminál nacházející se nejblíže k vrcholu "zásobníku"(přeskakuje symboly, které nejsou terminály). Tyto funkce slouží k jednodušší implementaci precedenční analýzy.

4 Práce v týmu

Práce v týmu byla rozdělena na jednotlivé logické celky, n kterých se podíleli jednotliví členové týmu. Každý člen týmu konzultoval své řešení se zbytkem týmu. Komunikace probíhala osobně na pravidelných schůzích nebo přes aplikaci Discord. Pro správu verzí byl využit GitHub. Abychom

předešli desetinným číslům procent u rozdělení bodů, 1 % navíc jsme přenechali jednomu ze členů týmu.

4.1 Rozdělení práce

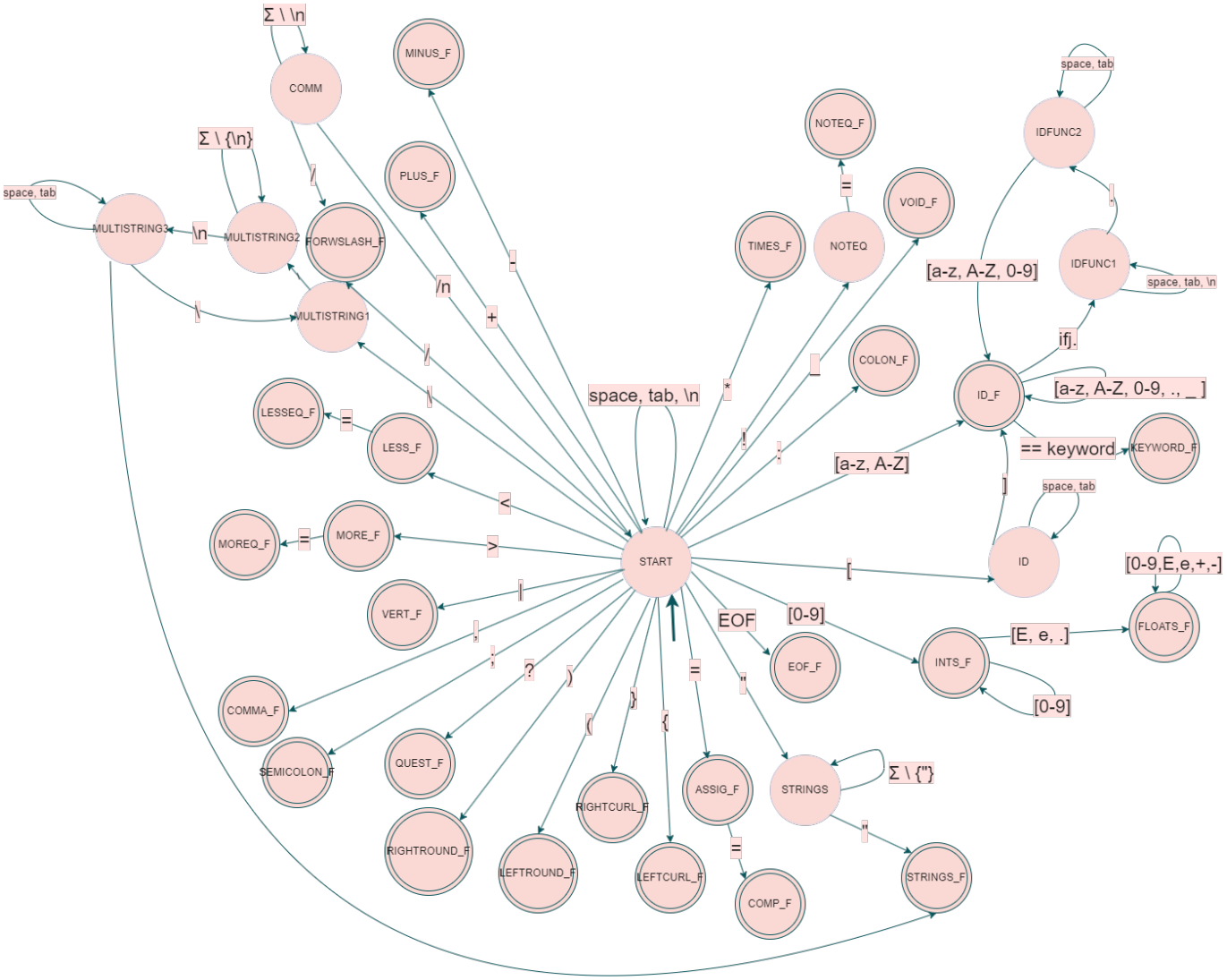
- Andrea Birckmannová
  - Lexikální analýza
  - Generování kódu
- Jan Štefan Hodák
  - Syntaktická analýza
  - Sémantická analýza
  - Testování
  - Dokumentace
- Tomáš Lusštík
  - Precedenční syntaktická analýza
  - sémantická analýza
  - Správa repozitáře
- Barbora Šturcová
  -

5 Závěr

Tento projekt nám pomohl hlouběji porozumnět struktuře a návrhu kompilátorů, a také datových struktur. Během implementace jsme se naučili základní strukturu jazyka Zig, na kterém byl postaven jazyk IFJ24, ale také rozšířili naše znalosti v jazyce C.

A Přílohy

A.1 Graf KA lexikálního analyzátoru



A.2 LL-pravidla

1. exp -> <exp_symbol> <exp_remain>
2. exp_symbol -> (<exp>)
3. exp_symbol -> i32
4. exp_symbol -> f64
5. exp_symbol -> id
6. exp_symbol -> []u8
7. exp_symbol -> <call_function>
8. exp_remain -> operator <exp>
9. exp_remain -> ϵ
10. define_const -> const id <: type> = <exp> ;
11. define_var -> var id <: type> = <exp> ;
12. assign_variable -> id = <exp> ;
13. void_result -> _ = <exp> ;
14. : type -> ϵ
15. : type -> : <type>
16. type -> ? <type>
17. type -> []u8
18. type -> i32
19. type -> f64
20. define_function -> pub fn id (<params>) <return_type> < commands >
21. return_type -> void
22. return_type -> <type>
23. params -> ϵ
24. params -> id : <type> <params_remain>
25. params_remain -> ϵ
26. params_remain -> , <params>
27. _return -> return <return_value> ;
28. return_value-> <exp>
29. return_value -> ϵ
30. if_statement -> if (<exp>) <if_remain>
31. if_remain -> < commands > else < commands >

32. if_remain -> | id | < commands > else < commands >
33. while_statement -> while (<exp>) <while_remain>
34. while_remain -> < commands >
35. while_remain -> | id | < commands >
36. call_function -> id (<exp_list>)
37. exp_list -> ϵ
38. exp_list -> <call_function_exp> <exp_list_remain>
39. call_function_exp -> <exp>
40. call_function_exp-> ϵ
41. exp_list_remain -> ϵ
42. exp_list_remain -> , < call_function_exp > <exp_list_remain>
43. commands -> <command> <commands>
44. commands -> ϵ
45. command -> <define_const>
46. command -> <define_var>
47. command -> <assign_variable>
48. command -> <void_result>
49. command -> <if_statement>
50. command -> <while_statement>
51. command -> <_return>
52. command -> <define_function>
53. command -> <call_function>
54. import -> const id = @import ([]u8) ;

A.3 LL(1) tabulka

Nonterminal	**(**	**i32**	**f64**	**u8**	**id**	**operator**	**const**	**var**	**_**	**:**	**?**	**func_id**	**pub**	**void**	**return**	**if**	**while**	**,**	**{**	**|**
exp	1	1	1	1	1							1								
exp_symbol	2	3	4	6	5							7								
exp_remain						8														
define_const							10													
define_var								11												
assign_variable					12															
void_result									13											
:type										15										
type		18	19	17							16									
define-function													20							
return_type		22	22	22							22			21						
params					24															
params_remain																		26		
_return															27					
return_value	28	28	28	28	28							28								
if_statement																30				
if_remain																			31	32
while_statement																	33			
while_remain																			34	35
call_function												36								
exp_list	38	38	38	38	38							38								
call_function_exp	39	39	39	39	39							39								
exp_list_remain																		42		
commands					43		43	43	43			43	43		43	43	43			
command					47		45	46	48			53	52		51	49	50			
import							54													

Tabulka 1: LL(1) tabulka

A.4 Precedenční tabulka

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

Tabulka 2: Precedenční tabulka

A.5 Redukční pravidla

1. E → E + E
2. E → E − E
3. E → E ∗ E
4. E → E / E
5. E → (E)
6. E → i
7. E → E == E
8. E → E ! = E
9. E → E < E
10. E → E > E
11. E → E <= E
12. E → E >= E