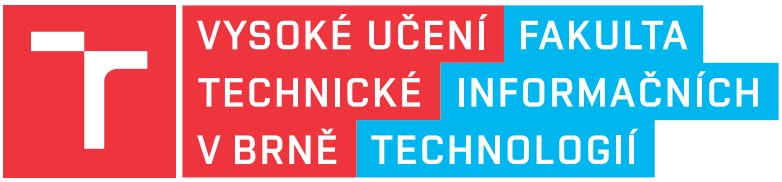


Projektová dokumentace
Překladač jazyka IFJ22

Tým xstrel03
Varianta TRP

8. prosince 2022

Matyáš Strelec	`xstrel03`	25%
Ondřej Seidl	`xseidl06`	25%
Maxmilián Nový	`xnovym00`	25%
Dominik Klon	`xklond00`	25%

Obsah

1 Práce v týmu

Společně jsme vytvořili návrh modulů a rozhraní pro komunikaci a předávání dat mezi nimi. Každý pracoval na své části, vzájemně jsme komunikovali postup ostatním, pokud nastaly chyby, byly zaznamenány a později vyřešeny. Pro sdílení kódu a zaznamenávání chyb byl využit `GitHub`, pro moduly byly vytvořeny samostatné větve, které se slučovaly do hlavní větve, pokud byla funkcionalita vylepšena. Pravidelně jsme se scházeli prezenčně nebo on-line, abychom ostatní informovali o stavu kódu a domluvili se na dalším postupu řešení. V poslední fázi jsme náš program testovali vlastními programy v `IFJ22` a pokud jsme narazili na chybu, společně jsme ji opravili.

Jednotlivé části projektu byly rozděleny mezi členy týmu následovně.

1.1 Rozdělení práce

Matyáš Strelec

- Lexikální analýza
- Syntaktická analýza

Ondřej Seidl

- Implementace tabulky symbolů
- Zpracování výrazů
- Precedenční analýza

Maxmilián Nový

- Návrh LL-gramatiky
- Vestavěné funkce
- Sémantická analýza

Dominik Klon

- Generování kódu

Společná práce

- Testování
- Dokumentace
- Drobné opravy

1.2 Odchylky od rozvnoměrného rozdělení

Body jsou rozděleny rovnoměrně mezi všechny členy týmu.

2 Lexikální analýza

Implementace lexikální analýzy je obsažena v souborech `lexer.c` a `lexer.h`.

2.1 Datové struktury

Pro potřeby lexikálního analyzátoru byly vytvořeny datové struktury které pomáhají při práci s tokeny a konečným automatem. Výčtový typ `fsm_state_t` obsahuje všechny možné stavy konečného automatu dle návrhu, výčtový typ `token_type_t` definuje typy tokenů.

Struktura `token_t` obsahuje informace o tokenu, jeho typ, pozici v souboru, délku, a jeho předchůdce a následníka ve spojovém seznamu. Struktura `token_list_t` obsahuje ukazatele na první, poslední, a aktuální token.

2.2 Funkce

Všechny funkce jsou ve zdrojových souborech popsané v komentářích, včetně jejich funkcionality, parametrů a návratových hodnot.

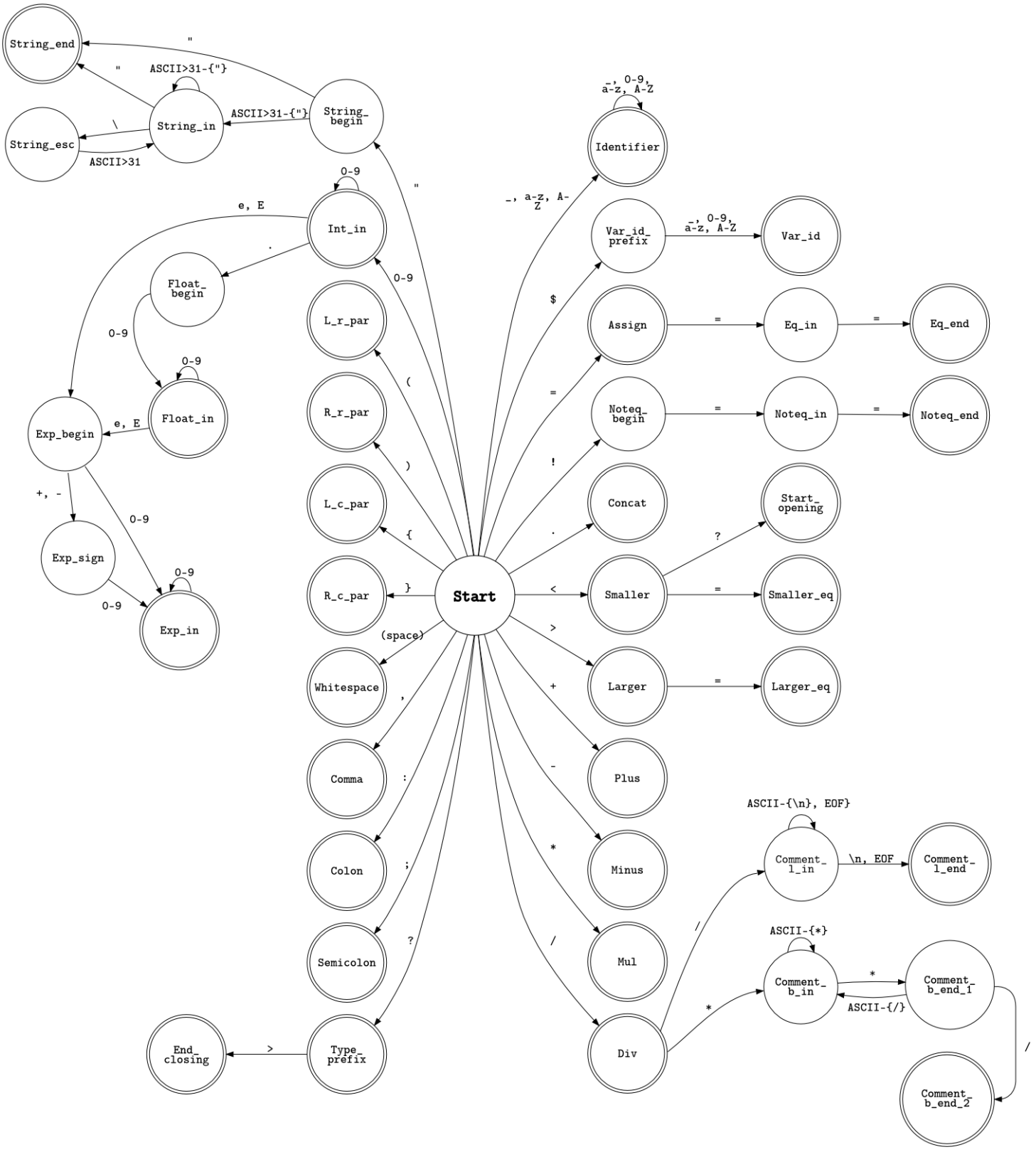
Funkce lexeru volaná z hlavního programu je funkce `fillTokenList()`, která jako parametr dostává ukazatel na strukturu `token_list_t`, kterou naplní seznamem tokenů pomocí volání funkce `getNextToken()`. Funkce `getNextToken()` je volána v cyklu, dokud není dosažen token typu konec souboru.

Funkce `getNextToken()` je implementována pomocí konečného automatu. Dle posloupnosti znaků na vstupu určuje typ a vyplňuje data tokenu. V případě, že je na vstupu znak, který nelze podle automatu dále číst, je kontrolováno, jestli momentální stav automatu je koncový, pokud ano, token je validní. Dále jsou rozpoznána klíčová slova a odstraněny úvozovky z řetězců. Pokud automat není v koncovém stavu, ale na vstup přijde znak, který automat nemůže přečíst, funkce vrací chybu 1. Funkce také řeší práci s escape sekvencemi v řetězcích.

Dále soubor obsahuje funkce na práci se seznamem tokenů jako vázaným seznamem a funkce pro ladění.

2.3 Diagram konečného automatu

Vizte obrázek 1.



Obrázek 1: Diagram konečného automatu vytvořený nástrojem `Graphviz`

3 Syntaktická analýza

3.1 Implementace

Implementace syntaktické analýzy je obsažena v souborech `parser.c` a `parser.h`.

Hlavní funkce syntaktického analyzátoru je `parser()`. Parametrem dostává ukazatel na seznam tokenů, se kterými pracuje, a pro účely sémantické analýzy ukazatel na strukturu tabulek symbolů a číslo průchodu. Funkce `parser()` nejprve kontroluje správnost prologu. Ten kontroluje funkce `checkProlog`, která prochází tokeny prologu a v případě chyby vrací chybový kód 1.

Pro začátek syntaktické analýzy je volána funkce `rule_Prog`, jedna z funkcí reprezentující neterminály gramatiky. Pro každý neterminál existuje funkce, která na základětypu následujícího tokenu pravidlo, které se pro zpracování použije.

Na základě pravidel LL-gramatiky a LL-tabulky funkce `parser()` vybírá podle typu následujícího tokenu pravidlo, které se pro zpracování použije.

Postupně se zpracovávají terminály, které zpracovává funkce `parseTerminal()`, jejíž parametr typu tokenu určuje typ tokenu, který by měl být další. Pokud není očekávaný typ, vrací chybu. Pokud se narazí na neterminál, je volaná funkce pro tento neterminál. Zpracovávání probíhá dokud není načten token typu `EOF`, tedy konec souboru.

3.2 Precedenční analýza

Precedenční analýza je řešena v `exp_parser.c` a má rozhraní v `exp_parser.h`. Obsahuje 2 funkce.
Funkce `parse_expression_with_tree()` slouží k syntaktické kontrole výrazů a sestavení parsovacího stromu, který se sestaven na základě precedence operátorů se kterým dále pracuje generátor kódu. Funkce `exp_parser()` slouží čistě ke zpřehlednění kódu. Implementace je dále doplněna o funkci `precedence()` která vrací prioritu operátoru.

3.3 Syntaktický strom

Syntaktický strom je implementován v `parse_tree.c` se stejnojmeným rozhraním. Nejdůležitější je funkce `makeOpNode()`, která vytvoří strom s operátorem v kořenovém uzlu a 2 potomky jako operandy. `printPtree()` je pak dále využívána generátorem kódu viz sekce 6. Zbytek funkcí slouží pro vkládání nových uzlů a pro alokování a uvolňování zdrojů.

3.4 LL-gramatika

Pro jazyk IFJ22 byla navržena následující gramatika.

```
1:  <prog> -> <stat> <prog>
2:  <prog> -> function func-id ( <params> ) : type { <st-list> } <prog>
3:  <prog> -> <eof>

4:  <eof> -> ?> EOF
5:  <eof> -> EOF

6: <params-cont> -> , type $id <params-cont>
7: <params-cont> -> eps

8: <params> -> type $id <params-cont>
9: <params> -> eps

10: <args-cont> -> , <term> <args-cont>
11: <args-cont> ->

12: <args> -> <term> <args-cont>
13: <args> -> eps

14: <stat> -> $id = <assign> ;
15: <stat> -> while ( <expr> ) { <st-list> }
16: <stat> -> if ( <expr> ) { <st-list> } else { <st-list> }
17: <stat> -> return <expr> ;
18: <stat> -> <expr> ;
19: <stat> -> func-id ( <args> ) ;

20: <st-list> -> <stat> <st-list>
21: <st-list> -> eps

22: <assign> -> <expr>
23: <assign> -> func-id ( <args> )

24: <term> -> $id
25: <term> -> val
```

Poznámky

`$id` - identifikátor proměnné
`func-id` - identifikátor funkce
`val` - číselný nebo řetězcový literál
`type` - datový typ (`int`, `double`, `string`, `void`
`func-id` - identifikátor funkce
`eps` - ε

3.5 LL-tabulka

	function	func-id	(	)	:	type	{	}	?>	EOF	,	$id	=	;	while	<expr>	if	else	return	val
<prog>	2	1							3	3		1			1	1	1		1	
<eof>									4	5										
<params-cont>				7							6									
<params>				9		8														
<args-cont>				11							10									
<args>				13								12								12
<stat>		19										14			15	18	16		17	
<st-list>		20						21				20			20	20	20		20	
<assign>		23														22				
<term>												24								25

Obrázek 2: LL-tabulka s pravidly přechodů

3.6 Precedenční tabulka

	*	/	+	-	.	<	>	<=	>=	===	!==	(	)	var	$
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
+	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
-	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
.	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
===	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>
!==	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>	>		>		>
var	>	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<	<		<	

Obrázek 3: Precedenční tabulka pro práci s výrazy

4 Sémantická analýza

Sémantická analýza probíhá v souborech `parser.c` a `parser_tree.c` ve formě různě rozmístněných kontrol.

4.1 Nedefinování/redefinováni funkce

Název funkce, kterou kontrolujeme, si ukládáme v globální proměnné `functionName`. Ten se vyhledává v tabulce symbolů, nebo se porovnává s názvy vestavěných funkcí. Pokud jde o neexistující funkci, nebo o redefinici už existující funkce, ukončíme program s návratovou hodnotou 3.

4.2 Argumenty a návratová hodnota funkce

Počet parametrů funkce a typ její návratové hodnoty si při definici funkce ukládáme do tabulky symbolů. Při použití zadefinované funkce později v programu počítáme počet argumentů, s kterými jsme funkci zavolali. Vyjímkou je však vestavěná funkce `write`, která má libovolný počet parametrů. Kontrola typu návratové hodnoty probíhá vygenerováním příkazů v cílovém jazyku, který porovnává typ hodnoty, kterou má funkce vracet, a typ hodnoty, kterou skutečně vrací. Tato kontrola proběhne až při spuštění kódu vygenerovaného překladačem. Kontrolujeme i zda ve funkcích nechybí příkaz `return` za pomoci globální proměnné `hasReturn`, pokud v sobě tato proměnná má uloženou hodnotu false a zároveň funkce, kterou definujeme, nemá návratový typ `void`, potom, stejně jako s ostatními kontrolami, ukončíme program s návratovou hodnotou 4.

4.3 Použití nedefinované proměnné

Kontrola použití nedefinovanné proměnné probíhá v souboru `parser_tree.c`. Když ve výrazu používáme proměnnou, vyhledáme jí v tabulce symbolů. Pokud jí nenajdeme, program se ukončí s návratovou hodnotou 5.

4.4 Chybějící/přebývající výraz v návratu z funkce

Při použití příkazu `parser_tree.c`, skontrolujeme, jaký návratový typ má funkce, v které příkaz používáme, pokud má návratový typ `void` a při příkazu `return` se nachází výraz, ukončíme program s chybou 6. Stejně ukončíme program, pokud funkce nemá navratový typ `void` a při příkazu `return` výraz chybí.

5 Tabulka symbolů

Tabulka symbolů je implementována v souborech `symtable.c` a `symtable.h` jako tabulka symbolů s rozptýlenými položkami.

5.1 Rozptylovací funkce

Je použita varianta pro sdbm s magickou konstantou 65 599. Jak funkce tak i konstanta jsou převzaty z předmětu IJC vyučovaného v letním semestru 2021/22. [1] [2]

5.2 Typy a struktury

Tabulka symbolů využívá typu `symtable_t` což je struktura obsahující struktury symbolů typu `symbol_t`. Symboly pak obsahují potřebné informace pro definice, generov vyhodnocování výrazů a sémantickou analýzu. Další typ `symtables_t` pak slouží pro uchování tabulek symbolů pro funkce, proměnné a pro pohodlnou práci s rámci.

5.3 Funkce

Tabulka musí z tokenů používané v syntaktické analýze udělat symboly. Na to slouží funkce `token_to_symbol()`. Tabulka symbolů je dynamicky alokována a její implementace obsahuje funkce na kontrolu velikosti, její následné zvětšení a další pomocné funkce pro prácí s tabulkou(insert, print,...).

[1] Stránka předmětu IJC: `http://www.fit.vutbr.cz/study/courses/IJC/public/DU2.html.cs`
[2] Popis sdbm: `http://www.cse.yorku.ca/~oz/hash.html`

6 Generování kódu

Generování kódu probíhalo v modulech syntaktického parseru a parseru pro výrazy. Kód se nachází v jednotlivých funkcích a generuje se postupně s jejích voláním, tj. postupně s průchodem syntaktického parseru.

6.1 Práce s tabulkou symbolů

Pro správné generování bylo potřeba využít tabulky symbolů, do které se při prvním průchodu uloží proměnné a na začátku hlavního těla a na začátku těla každé funkce se vygenerují definice proměnných naležících pro danné tělo. Nekompletní kód vygenerovaný prvním průchodem se přeskočí pomocí instrukce `JUMP` a za správný se považuje až kód z druhého průchodu.

6.2 Generování kódu pro výrazy

Generování kódu probíhá při Inorder průchodu binárním stromem vygenerovaných precedenční analýzou. Zároveň se tiskne kód, který sám o sobě umí zkontrolovat, jestli operace, které provádíme jsou validní a mají správné operandy.

6.3 Konverze typů pro výrazy v podmínce a cyklu

Cykly a podmínky jsou řešené pomocí porovnávání s hodnotou `bool@false`, proto bylo potřeba nejprve převést výsledek z generování výrazu na tento typ. Převádění hodnoty se provádí až v IFJcode22 kódu, který se generuje, když parser narazí na podmínku nebo cyklus.

6.4 Vestavěné funkce

Vestavěné funkce jazyka `IFJ22` byly přímo napsány v cílovém jazyku `IFJcode22` a tisknou se na konci kódu vygenerovaného překladačem.

7 Hlavní program

Hlavní program obsahuje soubor `main.c`. Hlavní program nejprve plní strukturu tokenů voláním funkce lexeru `FillTokenList()`. Následovně se vytvoří veškeré potřebné tabulky symbolů, které se používají v syntaktické a sémantické analýze. Spustí se první průchod funkce `parser()`, který zajistí naplnění tabulky symbolů indetifikátory funkcí a proměnných, poté se spouští druhý průchod, který už kontroluje správnou sémantiku. Na konec je tištěn kód pro vestavěné funkce. Po vykonání překladu je uvolněna alokovaná pamět pro tabulky symbolů a vázaný seznam tokenů. Pokud je hlavnímu programu v nějaké fázi vrácen chybový kód, je program ukončen s náležitým chybovým kódem a hláškou typu chyby.

8 Členění implementačního řešení

Jednotlivé části programu jsou rozděleny do zdrojových souborů následovně.

- `error.c/h` - Chybové výpisy a kódy
- `exp_parser.c/h` - Zpracování výrazů
- `lexer.c/h` - Lexikální analyzátor
- `main.c` - Hlavní program
- `parser.c/h` - Syntaktický a sémantický analyzátor, generování kódu
- `parse_tree.c/h` - Abstraktní syntaktický strom pro práci s výrazy
- `stack.c/h` - Zásobník
- `symtable.c/h` - Tabulka symbolů implementovaná jako tabulka s rozptýlenými prvky