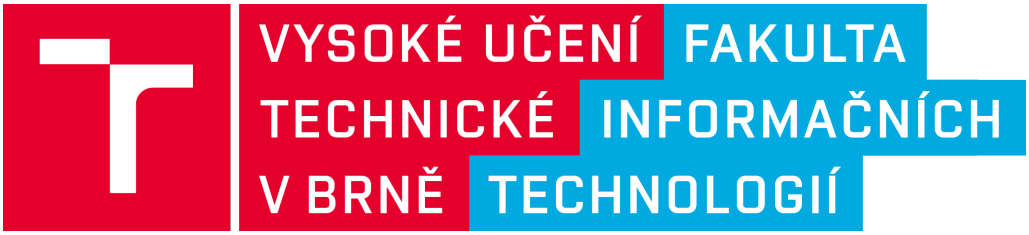


Projektová dokumentace implementace překladače

Tým xmasek19 varianta TRP

Členové týmu a rozdělení bodů

Vojtěch Kuchař	xkucha30	25
Jakub Mašek	xmasek19	25
Filip Polomski	xpolom00	25
Martin Zelenák	xzelen27	25

7. prosince 2022

Obsah

1 Struktura překladače

Náš projekt se skládá z několika menších celků. Lexikálního analyzátoru, syntaktického analyzátoru s rekurzivním sestupem pro kostru programu a precedenční analýzou pro výrazy, sémantického analyzátoru. Dále z generátoru kódu a pomocných funkcí, které využíváme pro usnadnění implementace.

1.1 Lexikální analyzátor

Lexikální analyzátor byl navržen a implementován v souboru *scanner.c* podle deterministického konečného automatu viz Obrázek 1. V našem řešení jsou jeho výstupem tokeny z funkce *getToken()* volané ze souboru *parser.c*. Lexikální analyzátor tvoří tokeny typů klíčová slova, EOF, identifikátory, void, aritmetické operátory, operátory přiřazení, začátek a konec programu, operátory porovnávání a speciální symboly (například závorky, středník, dolar).

1.2 Syntaktický a sémantický analyzátor

Syntaktický a statický sémantický analyzátor jsou sjednoceny a implementovány v souboru *parser.c*. Analýza kostry programu je implementována rekurzivním sestupem podle LL-gramatiky viz Sekce 3.2. Pro analýzu výrazů, včetně volání funkcí, je implementována precedenční analýza řídící se precedenční tabulkou viz Sekce 3.4. Implementace zahrnuje i zotavování z chyb v překládaném souboru.

1.3 Generátor kódu

Generátor kódu je převážně implementován v souboru *codeGenerator.c* a částečně v souboru *parser.c*. Kód se generuje včetně dynamických sémantických kontrol a případných chybových hlášek pro běhové chyby.

2 Implementace

V následující kapitole je popsána implementace překladače.

2.1 Členění implementačního řešení

2.1.1 *codeGenerator.c, codeGenerator.h*

Implementační řešení generování IFJcode22 kódu. V rámci našeho řešení byly navrženy pomocné funkce (například konverze typů), generované jako funkce jazyka IFJcode22 a volané pomocí instrukce CALL. Generovaný kód je postupně vkládán do dvou řetězců, ze kterých je po úspěšném dokončení všech kontrol tisknut výsledný kód na standartní výstup. Jako první je vytištěn řetězec s funkcemi a následně řetězec hlavního těla programu. Za zmínku stojí implementace univerzální pomocné funkce pro implicitní konverze typů, která je v kódu hojně využívána a nemusí se tak generovat kód pro každou konverzi zvlášť, například u aritmetických operacích.

2.1.2 *dynamicString.c, dynamicString.h*

Implementace dynamicky alokovaných řetězců a oparací nad nimi. Soubory *dynamicString.c* a *dynamicString.h* jsou využívané v *parser.c, scanner.c, codeGenerator.c*. Obsahuje funkce pro konkatenaci jednoho nebo více znaků za řetězec, pro vložení jednoho, či více znaků dovnitř řetězce a pro jejich mazání.

2.1.3 *error.h*

Soubor *error.h* byl implementován jako jeden enumerační list obsahující číselná označení chybových stavů.

2.1.4 *main.c*

V souboru *main.c* probíhá nastavení vstupního souboru pro *scanner.c* a zavolání funkce *parser()* ze souboru *parser.c*, která řídí celý následný překlad.

2.1.5 *Makefile*

Pro překlad používáme GNUMake, který jsme mimo jiné používali také pro testování jednotlivých částí projektu. Projekt je přeložen pomocí příkazu *make*. Příkaz *make clean* následně vymaže všechny binární soubory.

2.1.6 *parser.c, parser.h*

Obsahuje implementaci syntaktického a sémantického analyzátoru. Při implementaci LL-gramatiky bylo nutné použít pravidlo, nesplňující LL1 gramatiku pro příkaz přiřazení. Precedenční analýza je dále rozšířena o zpracování volání funkcí, které je tak vyřazeno z LL-gramatiky a to včetně jejich argumentů, tím je umožněno volání funkcí ve výrazech. Ve funkci pro pravidlo *¡stat-list¿* je implementován čítač otevřených složených závorek pro detekci ohraničených bloků kódu a ukončení funkce mimo tyto bloky. Při nedefinovaném volání funkce v jiné funkci je volání, včetně argumentů, uloženo na zásobník a do tabulky symbolů je uložen neúplný záznam. Před vytištěním kódu jsou volání ze zásobníku kontrolována pomocí tabulky symbolů, zda funkce byla dodatečně definována později.

2.1.7 *scanner.c, scanner.h*

Implementace lexikálního analyzátoru. V rámci našeho řešení částečně kontrolujeme správný zápis prologu a epilogu. Ve *scanner.c* zároveň probíhá překlad znaků v řetězcových literálech zapsaných v escape sekvenci ve formátu /xdd, kde dd je hexadecimální číslo. Obdobně zde pak probíhá překlad symbolů zapsaných jako oktalové číslo ve formátu /ddd.

2.1.8 *stack.c, stack.h*

Stack soubory obsahují implementaci zásobníku, schopného zpracovávat tokeny typu *Token*. Zásobník je v našem případě dynamicky alokovaný. Dále obsahuje implementaci řady funkcí pro operace se zásobníkem (například. push, pop). V našem řešení jsme naimplementovali i některé speciální funkce, například funkce *bottom*, která vrátí ukazatel prvek nacházející se na "dně"zásobníku. Nebo funkce *popBottom*, která "dno"zásobníku uvolní. Tyto funkce jsou dále využívány v souboru *parser.c*.

2.1.9 *symtable.c, symtable.h*

Implementace tabulky symbolů pomocí tabulky s rozptýleným přístupem (hash tabulky) s explicitním zřetězením položek. Tabulka je dynamicky alokovaná. Pro její rozšíření je implementována funkce *ST_expand()*, která krom rozšíření tabulky také znovu vypočítá klíče položek pro již vložené položky. Obdobně funguje funkce *ST_shrink()*. Tabulka ukládá ukazatele na struktury obsahující informace o funkcích a proměnných. O funkcích se ukládají následující informace: Zda-li je definovaná, její návratový typ a typ parametrů. U proměnných stačí jejich existence v tabulce (je definována).

2.2 Správa verzí a hosting

Pro správu verzí jsme používali distribuovaný systém správy verzí git a jako hosting GitHub. Zpočátku jsme implementovali v různých větvích, které jsme následně spojily, protože při dokončování projektu byl upravován převážně *codeGenerator.c*.

3 Závěr

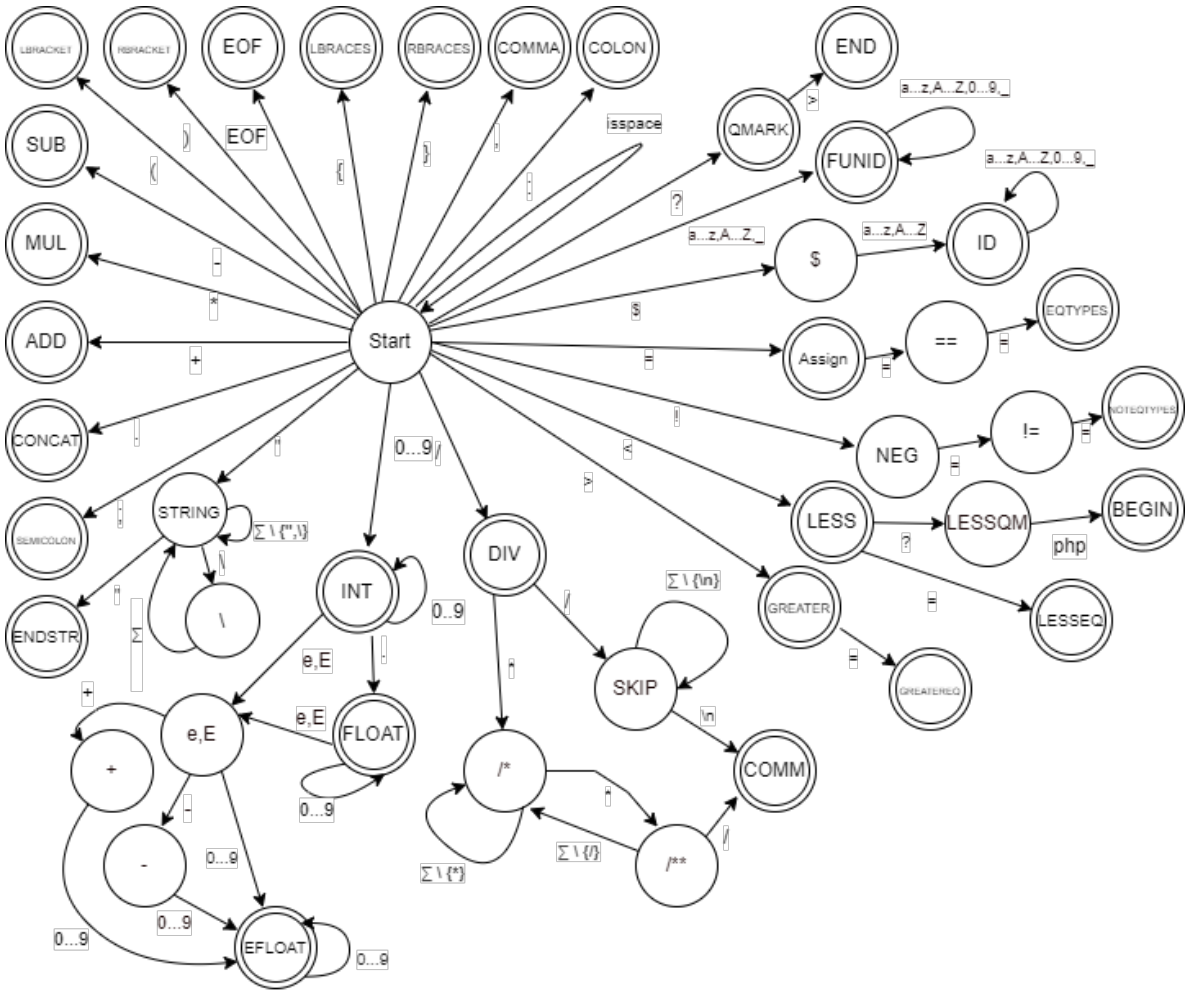
S celkovým vypracováním projektu jsme spokojení. S prací na projektu jsme začali brzy a díky tomu jsme měli dostatečnou časovou rezervu, ze které jsme později mohli čerpat. Práci jsme si na začátku rozdělili podle domluvy a tehdejších znalostí a schopností jednotlivých členů. Z prvu jsme si nebyli jistí,

jakým způsobem projekt pojmout a implementovat. Nejdříve jsme vypracovali návrh konečného automatu popisujícího činnost lexikálního analyzátoru a následovali tvorbou syntaktického analyzátoru. Tehdy jsme doplnili rozdělení práce o nově nabyté úkoly a doladili nedostatky s původním rozdělením. Při zkušebním odevzdání nám byly odhaleny některé nedostatky, na jejichž opravy jsme měli dostatek času.

3.1 Práce v týmu a rozdělení práce

Práce v týmu probíhala plynule. Společně jsme v týmu komunikovali. Komunikace probíhala převážně osobně, případně přes komunikační kanál discord.

Vojtěch Kuchař	dokumentace, generování kódu, implementace zásobníku a pomocných funkcí, testování
Jakub Mašek	syntaktický a sémantický analyzátor, zpracování výrazů, generování kódu, testování
Filip Polomski	lexikální analyzátor, dokumentace, testování
Martin Zelenák	syntaktický a sémantický analyzátor, tabulka symbolů, generování kódu, testování



Obrázek 1: Deterministický konečný automat lexikálního analyzátoru

3.2 *LL - gramatika*

1. <prog> => BEGIN DECLARE_ST <stat_list>
2. <stat_list> => EPILOG EOF
3. <stat_list> => EOF
4. <stat_list> => <stat> <stat_list> <return>
5. <stat> => IF (<expr>) <stat_list> ELSE <stat_list>
6. <stat> => WHILE (<expr>) <stat_list>
7. <stat> => FUNCTION FUNID (<params>) <funcdef>
8. <stat> => <assign>
9. <funcdef> => <stat_list>
10. <funcdef> => : TYPE <stat_list>
11. <assign> => ID = <expr> ;
12. <assign> => ID <expr> ;
13. <assign> => <expr> ;

14. <param> => TYPE ID
15. <param> => ? TYPE ID
16. <params> => <param> <params_2>
17. <params> => epsilon
18. <params_2> => , <param> <params_2>
19. <params_2> => epsilon
20. <return> => RETURN ;
21. <return> => RETURN <expr> ;
22. <return> = > epsilon

3.3 *LL - tabulka*

	BEGIN	EPILOG	$	IF	WHILE	FUNCTION	:	{	ID	TYPE	?	,	RETURN
<prog>	1												
<stat_list>		2	3	4	4	4			4				
<stat>				5	6	7			8				
<funcdef>							9	10					
<assign>									11, 12				
<param>										14	15		
<params>										16	16		
<params_2>												18	
<return>													20, 21

3.4 *Precedenční tabulka*

	+-	*/	(	)	id	c1	c2	$
+-	>	<	<	>	<	>	>	>
*/	>	>	<	>	<	>	>	>
(	<	<	<	=	<	<	<	
)	>	>		>		>	>	>
id	>	>		>		>	>	>
c1	<	<	<	>	<		>	>
c2	<	<	<	>	<	<		>
$	<	<	<		<	<	<	

*c1: <, >, <=, >=
*c2: ===, ! ==