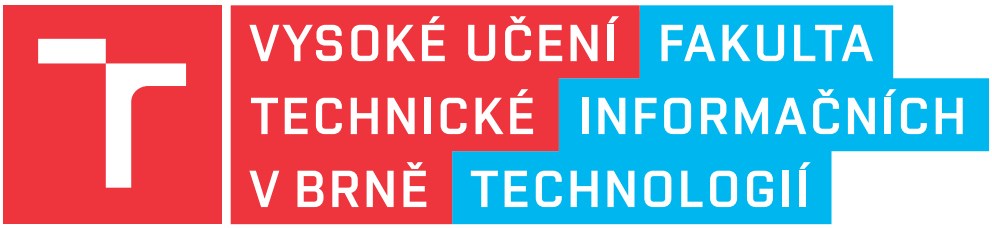


Projektová dokumentace

Implementace překladače imperativního jazyka IFJ23

Tým xdubro01

6. prosince 2023

Maksim Dubrovin	**(xdubro01)**	25 %
Anastasiia Samoilova	(xsamoi00)	25 %
Ilya Volkov	(xvolk02)	25 %
Anastasiia Mironova	(xmiron05)	25 %

Obsah

1 Úvod

Tento dokument byl vytvořen jako dokumentace pro projekt IFJ, popisující metody implementace všech komponent kompilátoru i problémy během uvedené implementace.

Cílem projektu je vytvořit program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ23 a přeloží jej do cílového jazyka IFJcode23

2 Implementace

Projekt jsme udělali z několika námi realizovaných částí, které jsou v této kapitole.

Kompilátor se skládá z následujících komponent:
1. Lexikální analyzátor
2. Syntaktický analyzátor
3. Sémantický analyzátor
4. Generátor kódu

Vstupní kód je zpracován každou komponentou ve stejném pořadí, v jakém je zapsán výše uvedený seznam.

2.1 Lexikální analýza

Lexikální analýza probíhá v `scanner.c`, tento soubor obsahuje implementaci skeneru. Skener čte vstupní zdrojový kód znak po znaku, přeskakuje prázdné místo a komentáře a identifikuje různé tokeny, jako jsou čísla, řetězce, identifikátory a symboly. V tomto souboru používáme pomocné funkce, které jsou implementovány v souborech `str.c` a `token.c`. Funkce **scanToken** v `scanner.c` je jádrem skeneru, který je zodpovědný za identifikaci a vytváření tokenů na základě vstupních znaků.

2.1.1 Podstata a souvislost procesu práce v souborech

V souboru `token.c` používáme funkce, které pomáhají při zpracování tokenů. Poskytované funkce umožňují vytváření, načítání a mazání tokenů a také dynamickou změnu velikosti tak, aby vyhovovala proměnné délce seznamu.

V souboru `str.c` funkce poskytují základní řetězcové operace potřebné při lexikální analýze a dalších částech kompilátoru. Zpracovávají výpočet délky řetězce, porovnání řetězců, kopírování řetězců a transformace, jako je přidání znaménka mínus do lexému tokenu.

Skener se inicializuje v `scanner.c` voláním **scanTokens**. Funkce **initScanner** nastavuje počáteční stav struktury skeneru.

Funkce **scanToken** je opakovaně volána pro tokenizaci vstupu. Tokeny jsou identifikovány na základě aktuálních znaků ve vstupním proudu. Identifikované tokeny jsou přidány do **TokenList** pomocí funkce **token_add**.

TokenList udržuje dynamické pole tokenů. Tokeny jsou přidány do seznamu, protože jsou identifikovány skenerem.

Konečným výsledkem je seznam tokenů představujících syntaktické prvky vstupního zdrojového kódu. Každý token má typ (např. celé číslo, řetězec, identifikátor) a lexém (skutečná textová reprezentace).

Celý lexikální analyzátor je implementován jako deterministický konečný automat podle vytvořeného diagramu. 1

2.2 Syntaktická analýza

Úkolem syntaktického analyzátoru je vyplnit tabulku symbolů (symtable). Syntaktická analýza, kterou lze nalézt v souborech `compiler.c` a `compiler.h`, vychází z LL – gramatiky 2 a LL tabulky. 3

Náš analyzátor syntaxe, je rekurzivní sestupný analyzátor, jako by při práci stavěl abstraktní syntaktický strom (abstract syntax tree), různé funkce zpracovávají různá gramatická pravidla na základě aktuálního tokenu a jeho priority. Sleduje také místní a globální proměnné pomocí tabulek znaků a seznamů místních proměnných.

`Nejdůležitější struktury pro parser:`

- Syntaktický analyzátor se pohybuje v seznamu tokenů pomocí struktury **Parser**.
- Funkce **expression** analyzuje výrazy na základě priority.
- Pravidla syntaktické analýzy pro různé typy tokenů jsou definována ve struktuře **ParseRule**.
- Informace o místních proměnných jsou spravovány pomocí struktur **Local** a **Local List**.

2.2.1 Použité techniky

Při práci na syntaktické analýze jsme pro parsing použili následující techniku **Recursive Descent Parsing**.

Recursive Descent Parsing je založená na principu rekurzivních postupů. **Recursive Descent Parsing** se používá k analýze gramatiky jazyka. Každé pravidlo gramatiky je reprezentováno funkcí, která rekurzivně volá sebe a další rozebíratelné funkce pro kontrolu podstruktur.

Princip práce:

- Každý neterminál v gramatice je reprezentován funkcí.
- Pro každý neterminál je vytvořena funkce rozboru.
- Funkce rozboru způsobuje sebe nebo jiné funkce podle pravidel gramatiky.

Při práci na parsingu výrazů jsme použili **Pratt's Parsing**. Zaměřuje se na zpracování operátorů a explicitně definuje prioritu a asociativnost operátorů.

Princip práce:

- Každý operátor je reprezentován funkcí rozboru.
- Operátoři jsou seskupeni podle priority.
- S tím se počítá i asociativnost operatorů.
- Rekurzivní sestup se používá k analýze výrazů.

Jako výsledek parser kontroluje strukturu zdrojového kódu podle definovaných gramatických pravidel. Analyzované informace lze použít pro další fáze kompilátoru, jako je sémantická analýza a generování kódu.

2.2.2 Tabulka symbolů

Při zpracovávání výrazů je použita precedenční tabulka 4.

`symbtable.c` a `symbtable.h` tyto soubory implementuje tabulku symbolů, což je datová struktura pro ukládání párů klíč-hodnota. Používá se ke správě proměnných a funkcí během kompilace. Tabulka symbolů používá otevřené adresování a lineární sondování pro řešení kolizí.

Analyzátor interaguje se dvěma tabulkami symbolů: **varTable** pro proměnné a **funcTable** pro funkce. Funkce jako **symtable_insert_variable** a **symtable_insert_function** se používají k vložení informací o proměnných a funkcích do příslušných tabulek symbolů.

2.3 Sémantická analýza

Sémantická analýza je prováděna podle sémantických pravidel jazyka IFJ23. Sémantická a syntaktická analýza u nás probíhá v jednom souboru `compile.c`

V souboru `compile.c` používáme tabulky symbolů ke správě informací o proměnných a funkcích. Tabulky symbolů jsou vyplněny a dotazovány během analýzy, což zajišťuje správné a efektivní ukládání a vyhledávání informací o proměnných a funkcích během analýzy. Sémantická analýza zahrnuje kontrolu duplicitních deklarací a zajištění správného použití.

2.4 Generátor kódu

Generátor kódu je implementován v `compiler.c`. Jeho úkolem je vytvořit konečný kód. Kód začíná definováním struktur a výčtů, jako je **Precedence** a **ValueType**, které se používají k reprezentaci priority operací a typů hodnot.

Jsou oznámeny globální proměnné a struktury, jako je Parser, Local, a další, které budou použity v procesu kompilace. Existuje pole **rules**, které představuje pravidla parsingu pro různé tokeny. Tato pravidla se používají při analýze výrazů a příkazů.

Generátor našeho kódu obsahuje: funkce, které zpracovávají vestavěné funkce generováním odpovídajícího kódu pro virtuální stroj; funkce jako **advance, consume, match, expression** a další, které zpracovávají tokeny a výrazy v kódu; funkce pro zpracování deklarací proměnných; funkcí, podmíněných příkazů a cyklů (declaration, funcDeclaration, ifStatement, whileStatement, atd.).

Kompilace začíná vyvoláním funkce **initCompiler**, která inicializuje kompilátor. Pak nastává kompilační cyklus, ve kterém se zpracovávají funkce a deklarace. Kompilace je ukončena vyvoláním funkce **endCompilation**. Proces kompilace generuje kód pro virtuální stroj, který předpokládá provedení programu napsaného v daném programovacím jazyce.

Celkový tok práce spočívá ve zpracování tokenů, generování příslušného kódu pro virtuální stroj a vytváření datových struktur pro správu místních proměnných.

3 Práce v týmu

Sešli jsme se o týden později, jak byl úkol zveřejněn. Vedoucí týmu - Maxim Dubrovin-rozdělil práci na základě našich dovedností, složitosti a času pro jednoho nebo druhého člena týmu. Tímto způsobem jsme měli plán oddělení naší práce a přibližné termíny, kdy musí být jedna nebo druhá část dokončena.

3.0.1 Týmová komunikace a spolupráce

Dali jsme přednost osobním setkáním před komunikací, scházeli jsme se jednou týdně nebo jednou za čtrnáct dní, probírali jsme naše další kroky a říkali, kdo, co a jak ve svém úkolu realizoval. Ale také jsme byli v kontaktu prostřednictvím messengeru ”Telegram”.

Pro správu souborů projektu jsme používali verzovací systém Git. Jako vzdálený repositář jsme používali GitHub.

Git nám umožnil pracovat na více úkolech na projektu současně.

3.1 Rozdělení práce mezi členy týmu

Práce byla rozdělena rovnoměrně na tolik, kolik je možné, s ohledem na její složitost a časovou náročnost, aby každý mohl zvládnout svůj úkol. Každý dostal procentuální hodnocení 25 %. Tabulka ukáže práci každého člena týmu:

Člen týmu	Přidělená práce
Maksim Dubrovin	Vedení týmu, organizace práce, kontrola, testování, struktura projektu, generování cílového kódu
Anastasiia Samoilova	Lexikální analýza, testování, dokumentace, prezentace
Ilya Volkov	Syntaktická analýza, sémantická analýza, testování, tabulka symbolů
Anastasiia Mironova	Lexikální analýza, testování, dokumentace

4 Závěr

Tento projekt je jedním z největších projektů na naší fakultě, tým jsme sestavili okamžitě, protože ještě před úkolem jsme se dohodli, že budeme spolupracovat. Tento projekt se ukázal být užitečný tím, že nám dal zkušenosti v týmu, naučil nás správný přístup k práci a rozdělování úkolů. Projekt vyžaduje spoustu času na realizaci, takže k němu potřebujete zodpovědný přístup a schopnost soustředit se na ponoření do plnění úkolu. Celkově nám tento projekt přinesl mnoho znalostí o práci překladačů, prakticky pro nás vysvětlil probíranou látku v předmětech IFJ a IAL.

5 Literatura

- Alexandr Meduna, Roman Lukáš - Formal Languages And Compilers, Lecture slides
- Jan M. Honzík, Ivana Burgetová, Bohuslav Křena, Algorithms, Lecture slides
- Robert Nystrom ”Crafting Interpreters”.

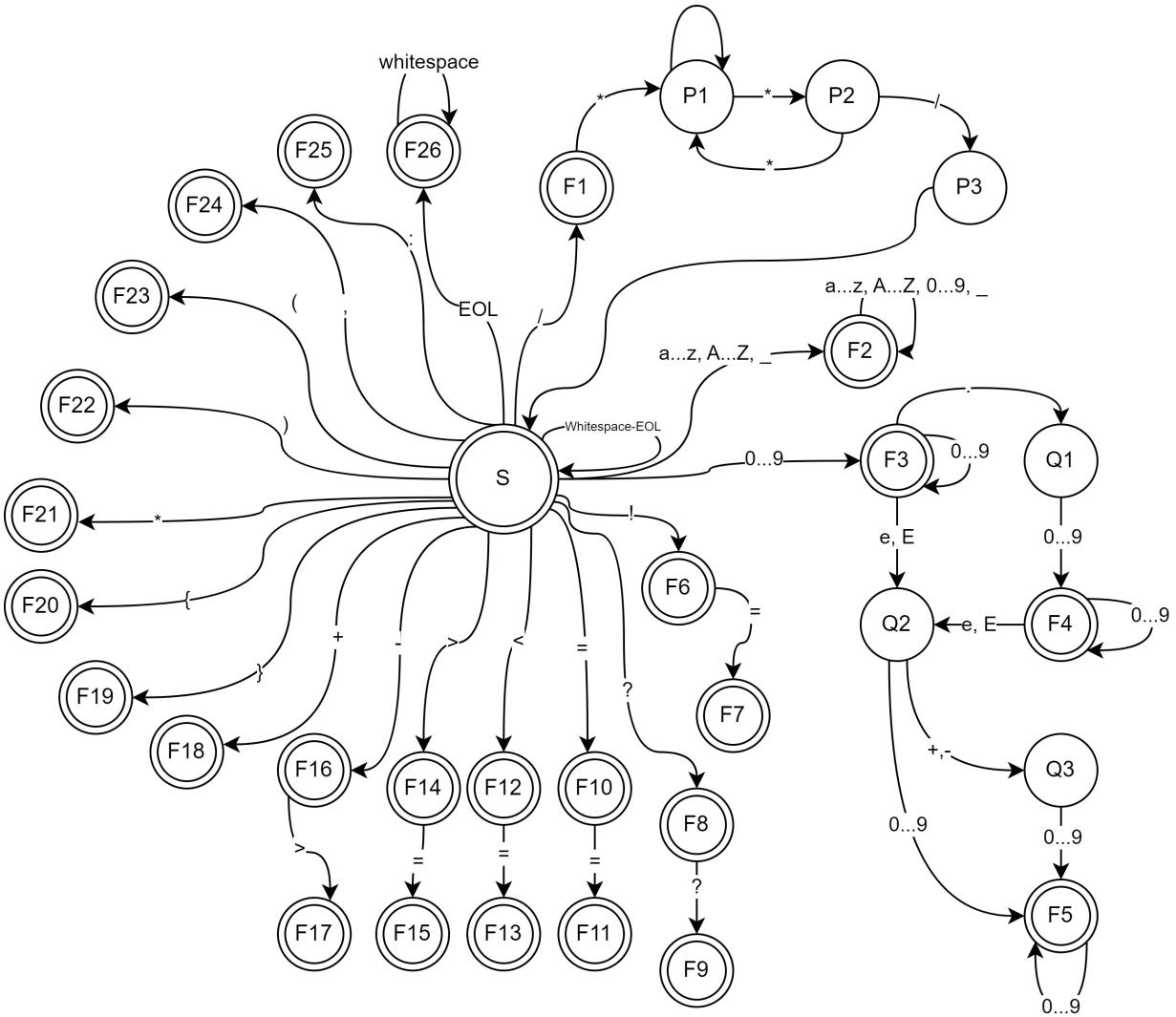
6 Přílohy

.1 Diagram konečného automatu specifikující lexikální analyzátor

.2 LL - gramatika

.3 LL – tabulka

.4 Precedenční tabulka



Legenda:

S	START
F1	SLASH
P1	BLOCK_COMMENTARY
P3	BLOCK_COMMENTARY_END
F2	IDENTIFIER
F3	FLOAT
Q1	NUMBER_POINT
F4	FLOAT
Q2	NUMBER_EXPONENT
Q3	NUMBER_EXPONENT_SIGN
F5	FLOAT
F6	BANG
F7	BANG_EQUAL
F8	QUESTION
F9	DOUBLE_QUESTIONS
F10	EQUAL
F11	EQUAL_EQUAL
F12	LESS
F13	LESS_EQUAL
F14	GREATER
F15	GREATER_EQUAL
F16	MINUS
F17	ARROW
F18	PLUS
F19	RIGHT_BRACE
F20	LEFT_BRACE
F21	STAR
F22	RIGHT_PAREN
F23	LEFT_PAREN
F24	COMMA
F25	COLON
F26	EOL

Obrázek 1: Diagram konečného automatu specifikující lexikální analyzátor

1. <program> → <statement>* EOF
2. <statement> → EOL
3. <statement> → var | let ID (: <type>)? = <expression>
4. <statement> → ID = <expression>
5. <statement> → if <logic_exp> <frame> (else <frame>)?
6. <statement> → while <logic_exp> <frame>
7. <statement> → func ID ((_ | ID ID : <type>)? (, _ | ID ID : <type>)*) (-> <type>)? EOL? <frame>
8. <statement> → <func_call>
9. <statement> → return <expression>?
10. <type> → Double (?)?
11. <type> → String (?)?
12. <type> → Int (?)?
13. <expression> → value
14. <expression> → ID
15. <expression> → <func_call>
16. <expression> → <term> | <expression> <add_op> <term>
17. <term> → <factor> | <term> <mult_op> <factor>
18. <factor> → ID | value | <factor> | (<expression>)
19. <add_op> → + | -
20. <mult_op> → * | /
21. <logic_exp> → <expression> logic_sign <expression>
22. <logic_exp> → (<logic_exp>)
23. <frame> → { <statement_list> }
24. <func_call> → ID ((_ | ID <expression>)? (, _ | ID <expression>))

Obrázek 2: LL – gramatika řídící syntaktickou analýzu

	EOF	EOL	var	let	ID	if	while	func	Double	String	Int	value	+	-	*	/	(	{
<program>	1	2	3	3	4, 24	5	6	7										
<statement>		2	3	3	4,24	5	6	7										
<type>									10	11	12							
<expression>					14, 24,18							13,18					18	
<term>					18							18					18	
<factor>					18							18					18	
<add_op>													19	19				
<mult_op>															20	20		
<logic_exp>					14,24,18							13,18					18,22	
<frame>																		23
<func_call>					24													

Obrázek 3: LL – tabulka použitá při syntaktické analýze

	!	+-	*/	(	)	logic	??
!	=	>	>	>	>	>	>
+-	<	=	<	<	>	>	>
*/	<	>	=	<	>	>	>
(	<	>	>	=	>	>	>
)	<	<	<	<	=	>	<
logic	<	<	<	<	<	=	>
??	<	<	<	<	>	<	=

Obrázek 4: Precedenční tabulka použitá při precedenční syntaktické analýze výrazů