Projektová dokumentace

Implementace překladače imperativního jazyka IFJ24

Tým xpopov10, varianta vv-BVS

Popov Albert	**(xpopov10)**	25 %
Krasovskyi Oleh	(xkrasoo00)	25 %
Savin Ivan	(xsavini00)	25 %
Turar Nurdaulet	(xturarn00)	25 %

3. prosince 2024

1 Úvod

Cílem projektu bylo vytvořit program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ24, jenž je zjednodušenou podmnožinou imperativního programovacího jazyka Zig 0.13, a přeloží jej do cílového jazyka IFJcode24 (mezikód).

Program funguje jako konzolová aplikace, která načítá zdrojový program ze standardního vstupu a generuje výsledný mezikód na standardní výstup nebo v případě chyby vrací odpovídající chybový kód.

2 Práce v týmu

2.1 Způsob práce v týmu

Na začátku projektu jsme úkol rozdělili do 4 hlavních bloků a jednali podle vzájemné interakce kódů. V případě, že se něčí práce zdála méně důležitá, tato osoba pomáhala ostatním nebo psal testy.

2.1.1 Verzovací systém

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 v tzv. větvích. Většinu úkolů jsme nejdříve připravili do větve a až po otestování a schválení úprav ostatními členy týmu jsme tyto úpravy začlenili do hlavní vývojové větve.

2.1.2 Komunikace

Komunikace probíhala v obecné konverzaci nebo v soukromých zprávách.

Ke konci projektu jsme se setkali, abychom ukázali, jak naše programové bloky fungují, s cílem najít chyby a zvýšit porozumění projektu ostatním účastníkům.

2.2 Rozdělení práce mezi členy týmu

Práci na projektu jsme si rozdělili rovnoměrně s ohledem na její složitost a časovou náročnost. Každý tedy dostal procentuální hodnocení 25 %. Tabulka shrnuje rozdělení práce v týmu mezi jednotlivými členy.

Člen týmu	Přidělená práce
Popov Albert	vedení týmu, testování, syntaktická analýza
Vojtěch Hertl	sémantická analýza, testování
Savin Ivan	lexikální analýza , dokumentace, testování
Turar Nurdaulet	generování cílového kódu, organizace práce, testovaní, struktura projektu

Tabulka 1: Rozdělení práce v týmu mezi jednotlivými členy

3 Návrh a implementace

Projekt jsme sestavili z několika námi implementovaných dílčích částí, které jsou představeny v této kapitole. Je zde také uvedeno, jakým způsobem spolu jednotlivé dílčí části spolupracují.

3.1 Struktura překladače

Překladač se skládá z následujících modulů:

- **Lexikální analyzátor (lexer):** Převádí zdrojový kód na tokeny
- **Parser:** Analyzuje tokeny pomocí syntaktických a sémantických pravidel a poté vytvoří abstraktní syntaktický strom (AST).
- **Generátor kódu:** Generuje z AST cílový kód.

Diagram konečného automatu pro lexer a LL tabulka budou přiloženy v samostatných přílohách.

Folders structure

- `/docs`: Dokumentace projektu.
- `/include`: Hlavičkové soubory (`.h`) pro jednotlivé moduly.
- `/scripts`: Skripty pro spuštění a kontrolu projektu.
- `/src`: Implementace jednotlivých modulů překladače a hlavní soubor `main.c`.
- `/tests`: Testovací soubory a skripty na ověření funkčnosti.

3.2 Lexikální analýza

Jako první jsme začali s lexikální analýzou. Zpočátku jsme počítali s tím, že mezi všemi tokeny bude existovat nějaký oddělovací symbol, pomocí kterého lze zdrojový kód rozdělit na bloky. Každý blok by následně mohl být snadno přeložen na odpovídající tokeny. Později jsme však od této myšlenky museli upustit, protože v některých případech bylo nutné implementovat příliš mnoho výjimek, což ztěžovalo rozšíření lexeru a zhoršovalo čitelnost kódu.

Nová verze lexikálního analyzátoru používá deterministický konečný automat (FSM). Hlavní funkce `run_lexer()` načítá vstupní kód a spouští základní stav automatu. Nejprve se vyhledá první neprázdný znak a podle něj se přechází do příslušného stavu automatu. Po zpracování stavu se kód vrátí do funkce `run_lexer()`, která pokračuje ve čtení, dokud není dosažen konec souboru.

Pro zpracování identifikátorů se nejprve přečtou povolené znaky. Po jejich přečtení se zkontroluje, zda se jedná o klíčové slovo nebo identifikátor. To umožňuje oddělit klíčová slova a vytvořit odpovídající znaky.

Při zpracování čísel se čtou číslice, tečky a znak e/E. Pokud je posloupnost zadána správně, vytvoříme podle zadaných údajů token f64 nebo i32

Celý lexikální analyzátor je implementován jako deterministický konečný automat podle předem vytvořeného diagramu, který bude přiložen v příloze. Tento diagram specifikuje jednotlivé stavy a přechody mezi nimi, což usnadňuje údržbu a rozšíření lexeru v případě potřeby.

3.3 Syntaktická analýza

¡¡ PUT YOUR TEXT HERE ¿¿

3.3.1 Here is a subsection if needed

`OurCode.c`.

Some text

3.4 Sémantická analýza

Sémantická analýza

3.5 Generování cílového kódu

Generování cílového kódu

3.5.1 Possible subsection

Some data

4 Závěr

Projekt nás zprvu trochu zaskočil svým rozsahem a složitostí. Postupem času, až jsme získali dostatek znalostí o tvorbě překladačů na přednáškách IFJ, jsme projekt začali řešit.

Náš tým jsme měli sestaven velmi brzy, byli jsme již předem domluveni na komunikačních kanálech, osobních schůzkách a na používání verzovacího systému, tudíž jsme s týmovou prací neměli žádný problém a pracovalo se nám společně velmi dobře.

Na projektu jsme začali pracovat trochu později, takže jsme neměli časovou rezervu, ale nakonec jsme všechno stihli. Jednotlivé části projektu jsme řešili většinou individuálně za použití znalostí z přednášek nebo materiálů do předmětů IFJ a IAL.

V průběhu vývoje jsme se potýkali s menšími problémy týkajícími se nejasností v zadání, ale tyto jsme vyřešili díky fóru k projektu. Správnost řešení jsme si ověřili automatickými testy a pokusným odevzdáním, díky čemuž jsme byli schopni projekt ještě více odladit.

Tento projekt nám celkově přinesl spoustu znalostí ohledně fungování překladačů, prakticky nám objasnil probíranou látku v předmětech IFJ a IAL a přinesl nám zkušennosti s projekty tohoto rozsahu.

A Grammar

```
[Program] ::= [Prolog] [FunctionDefList]

[Prolog] ::= 'const' 'ifj' '=' '@import' '(' '"ifj24.zig"' ')' ';'

[FunctionDefList] ::= [FunctionDef] [FunctionDefList]
[FunctionDefList] ::= ε

[FunctionDef] ::= 'pub' 'fn' [Identifier] '(' [ParamList] ')' [ReturnType] '{' [StatementList] '}'

[ParamList] ::= [Identifier] ':' [Type] [ParamListTail]
[ParamList] ::= ε

[ParamListTail] ::= ',' [Identifier] ':' [Type] [ParamListTail]
[ParamListTail] ::= ε

[ReturnType] ::= 'void'
[ReturnType] ::= [Type]

[StatementList] ::= [Statement] [StatementList]
[StatementList] ::= ε

[Statement] ::= 'const' [Identifier] ':' [Type] '=' [Expression] ';'
[Statement] ::= 'var' [Identifier] ':' [Type] '=' [Expression] ';'
[Statement] ::= [Identifier] '=' [Expression] ';'
[Statement] ::= 'if' '(' [Expression] ')' [NullableBinding] '{' [StatementList] '}' [OptionalElse]
[Statement] ::= 'while' '(' [Expression] ')' '{' [StatementList] '}'
[Statement] ::= 'return' [ReturnValue] ';'

[ReturnValue] ::= [Expression]
[ReturnValue] ::= ε

[NullableBinding] ::= '|' [Identifier]
[NullableBinding] ::= ε

[OptionalElse] ::= 'else' '{' [StatementList] '}'
[OptionalElse] ::= ε

[Expression] ::= [SimpleExpression] [RelationalTail]

[RelationalTail] ::= [RelationalOperator] [SimpleExpression]
[RelationalTail] ::= ε

[SimpleExpression] ::= [Term] [SimpleExpressionTail]

[SimpleExpressionTail] ::= '+' [Term] [SimpleExpressionTail]
[SimpleExpressionTail] ::= '-' [Term] [SimpleExpressionTail]
[SimpleExpressionTail] ::= ε
```

Tabulka 2: Grammar 1

B Grammar2

```
[Term] ::= [Factor] [TermTail]

[TermTail] ::= '*' [Factor] [TermTail]
[TermTail] ::= '/' [Factor] [TermTail]
[TermTail] ::= ε

[Factor] ::= '(' [Expression] ')'
[Factor] ::= [Identifier]
[Factor] ::= [Literal]

[Type] ::= '?' [BaseType]
[Type] ::= [BaseType]

[BaseType] ::= 'i32'
[BaseType] ::= 'f64'
[BaseType] ::= '[]u8'

[Identifier] ::= id [CallParam]
[Identifier] ::= '_'

[CallParam] ::= '(' [Expression] [CallParamTail] ')'
[CallParam] ::= ε

[CallParamTail] ::= ',' [Expression] [CallParamTail]
[CallParamTail] ::= ε


[Literal] ::= integer
[Literal] ::= floating-point
[Literal] ::= string literal
[Literal] ::= 'null'

[RelationalOperator] ::= '=='
[RelationalOperator] ::= '!='
[RelationalOperator] ::= '['
[RelationalOperator] ::= ']'
[RelationalOperator] ::= '[='
[RelationalOperator] ::= ']='
```

Tabulka 3: Grammar 2

C Precedenční tabulka

/x	*	/	+	-	“==”	“!=”	“<”	“>”	“<=”	“>=”	(	id	)	$
*	>	>	>	>	>	>	>	>	>	>	<	<	>	>
/	>	>	>	>	>	>	>	>	>	>	<	<	>	>
+	<	<	>	>	>	>	>	>	>	>	<	<	>	>
-	<	<	>	>	>	>	>	>	>	>	<	<	>	>
“==”	<	<	<	<	>	>	>	>	>	>	<	<	>	>
“!=”	<	<	<	<	>	>	>	>	>	>	<	<	>	>
“<”	<	<	<	<	>	>	>	>	>	>	<	<	>	>
“>”	<	<	<	<	>	>	>	>	>	>	<	<	>	>
“<=”	<	<	<	<	>	>	>	>	>	>	<	<	>	>
“>=”	<	<	<	<	>	>	>	>	>	>	<	<	>	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	
id	>	>	>	>	>	>	>	>	>	>			>	>
(	>	>	>	>	>	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<	<	<	<	<	<		

Tabulka 4: Precedenční tabulka použitá při precedenční syntaktické analýze výrazů