



Projektová dokumentace

Implementace překladače imperativního jazyka IFJ23

Tým xdubro01

23. listopadu 2023

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

Obsah

1	Úvod	1
2	Implementace	1
2.1	Lexikální analýza	1
2.1.1	Podstata a souvislost procesu práce v souborech	1
2.2	Syntaktická analýza	2
2.2.1	Tabulka symbolů	5
2.3	Sémantická analýza	5
2.4	Generátor kódu	6
3	Práce v týmu	6
3.0.1	Týmová komunikace a spolupráce	6
3.1	Rozdělení práce mezi členy týmu	6
4	Závěr	7
5	Literatura	7
6	Přílohy	8
.1	Diagram konečného automatu specifikující lexikální analyzátor	9
.2	LL – gramatika	9
.3	LL – tabulka	9
.4	Precedenční tabulka	9

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 zpracováván 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.

inc/FA_graph.pdf

2.2 Syntaktická analýza

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

inc/FA_graph.pdf

inc/FA_graph.pdf

Syntaktický analyzátor je inicializován pomocí seznamu vstupních tokenů a tabulek symbolů pro proměnné a funkce.

`parser.c` obsahuje implementaci analyzátoru. Definuje statickou proměnnou **currentToken** pro sledování aktuální pozice v seznamu tokenů a definuje statickou booleovskou proměnnou **isEOF** používanou k označení, zda bylo dosaženo konce souboru. On definuje ukazatele na tabulky symbolů (`var Table` a `funcTable`) pro proměnné a funkce, pak implementuje funkce pro analýzu deklarací proměnných, přiřazení, struktur řídicího toku a dalších programových konstrukcí. Funkce jsou navrženy tak, aby zpracovávaly různé části gramatiky, a analyzátor iteruje seznam tokenů při volání těchto funkcí.

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.

2.2.1 Tabulka symbolů

Při zpracovávání výrazů je použita precedenční tabulka 3.

inc/FA_graph.pdf

`symtable.c` a `symtable.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.

Analýzátor 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. Používáme strukturu `Parser`, ve které jsou potřebná data pro sémantické kontroly.

V souboru `parser.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. Tabulka symbolů se používá k ukládání 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,
Anastasiia Samoilova	kontrola, testování, struktura projektu, generování cílového kódu
Ilya Volkov	Lexikální analýza, testování, dokumentace
Anastasiia Mironova	Sntaktická analýza, sémantická analýza, testování
	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

6 Přílohy

.1 Diagram konečného automatu specifikující lexikální analyzátor

inc/FA_graph.pdf

Obrázek 1: Diagram konečného automatu specifikující lexikální analyzátor

.2 LL – gramatika

.3 LL – tabulka

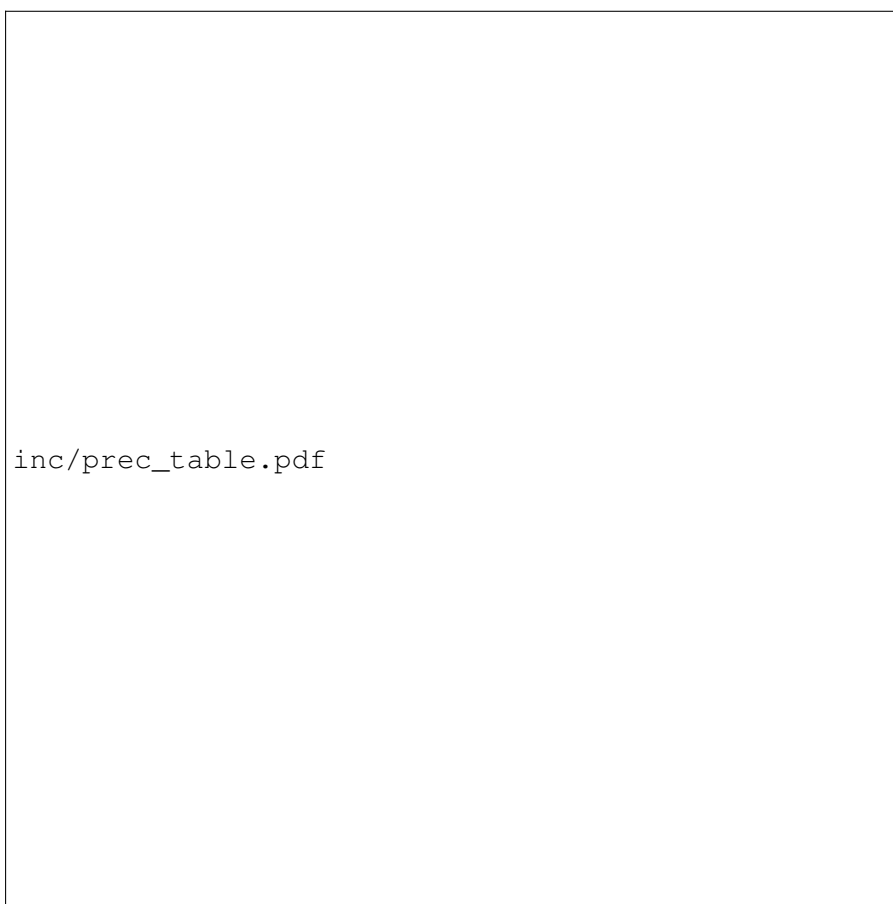
.4 Precedenční tabulka

1. `<prog>` -> DECLARE FUNCTION ID (`<params>`) AS `<type>` `<prog>`
2. `<prog>` -> FUNCTION ID (`<params>`) AS `<type>` EOL `<statement>` END FUNCTION `<prog>`
3. `<prog>` -> EOL `<prog>`
4. `<prog>` -> `<scope>`
5. `<scope>` -> SCOPE EOL `<statement>` END SCOPE `<end>`
6. `<end>` -> EOL `<end>`
7. `<end>` -> EOF
8. `<params>` -> ID AS `<type>` `<param_n>`
9. `<params>` -> ϵ
10. `<param_n>` -> , ID AS `<type>` `<param_n>`
11. `<param_n>` -> ϵ
12. `<statement>` -> DIM ID AS `<data_type>` `<def_var>` EOL `<statement>`
13. `<statement>` -> IF `<expression>` THEN EOL `<statement>` ELSE EOL `<statement>` END IF EOL `<statement>`
14. `<statement>` -> DO WHILE `<expression>` EOL `<statement>` LOOP EOL `<statement>`
15. `<statement>` -> ID = `<def_value>` EOL `<statement>`
16. `<statement>` -> INPUT ID EOL `<statement>`
17. `<statement>` -> PRINT `<expression>` ; `<print>` EOL `<statement>`
18. `<statement>` -> RETURN `<expression>` EOL `<statement>`
19. `<statement>` -> ϵ
20. `<def_var>` -> = `<expression>`
21. `<def_var>` -> ϵ
22. `<def_value>` -> ID (`<arg>`)
23. `<def_value>` -> ASC (`<arg>`)
24. `<def_value>` -> CHR (`<arg>`)
25. `<def_value>` -> LENGTH (`<arg>`)
26. `<def_value>` -> SUBSTR (`<arg>`)
27. `<def_value>` -> `<expression>`
28. `<arg>` -> `<value>` `<arg_n>`
29. `<arg>` -> ϵ
30. `<arg_n>` -> , `<value>` `<arg_n>`
31. `<arg_n>` -> ϵ
32. `<value>` -> INT_VALUE
33. `<value>` -> DOUBLE_VALUE
34. `<value>` -> STRING_VALUE
35. `<value>` -> ID
36. `<print>` -> `<expression>` ; `<print>`
37. `<print>` -> ϵ
38. `<type>` -> INTEGER
39. `<type>` -> DOUBLE
40. `<type>` -> STRING

Tabulka 1: LL – gramatika řídící syntaktickou analýzu

inc/LL_table.pdf

Tabulka 2: LL – tabulka použitá při syntaktické analýze



inc/prec_table.pdf

Tabulka 3: Precedenční tabulka použitá při precedenční syntaktické analýze výrazů