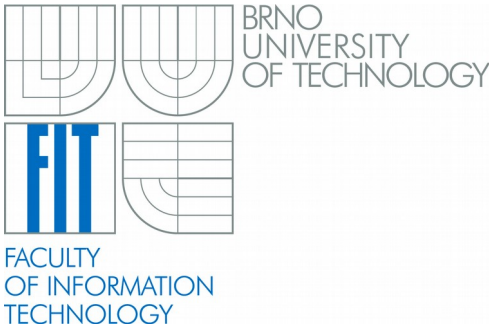


Documentace k projektu IFJ/IAL
Implementace interpretu imperativního jazyka IFJ16.

30. listopadu 2016

Tým číslo 056, varianta a/4/II

Kharytonov Danylo (xkhary00) – vedoucí – 16%
Kiselevich Roman (xkisel00) - 25%
Niahodkin Pavel (xniaho00) – 25%
Inhliziian Bohdan (xinhli00) - 17%
But Andrii (xbutan00) – 17%

Obsah

1. Úvod

Tato dokumentace popisuje implementaci překladače imperativního jazyka IFJ16, který je velmi zjednodušenou podmnožinou jazyka Java SE 8. Náš program načítá zdrojový soubor a hodnotí správnost kódu. Pokud kód je v pořádku, program interpretuje kód a vrátí 0. V případě, jestli v kódu najde chybu, vrací kód chyby.

2. Zadání

Program se skládá ze 3 hlavních částí- nejdůležitější části je **syntaktický analyzátor**. Načítá zdrojový soubor prostřednictvím **lexikálního analyzátoru** a překládá kód na posloupnost instrukcí, které následně předá **interpretu** k jejich vykonání.
Náš tým vybral variantu a/4/II a proto jsme museli použít **Knuth-Morris-Prattův algoritmus** pro vyhledávání podřetezce v řetězci, **algoritmus List-Merge sort** pro řešení a taky implementovat tabulku symbolů pomocí **tabulky s rozptýlenými položkami.**

3. Příprava

3.1 Verzování (VCS, Git)

Dobře organizovaná týmová práce představuje jeden z důležitých faktorů pro vytvoření projektu. Aby několik členů týmu mohli v klidu pracovat nad jednou úlohou a přitom neřešit zbytečné problémy jako přepsání nechtěných souborů, ztráta nebo dlouhé přenášení dát a podobně, vytvořili jsme repozitář ve verzovacím systému Git, kam jsme postupně ukládáli nové a nové soubory, ze kterých pak vznikl náš projekt.

3.2 Vytvoření stylu zápisu programu

První soubor, který se objevil v repozitáři, byl textový soubor, ve kterém náš vedoucí podrobně popsal pravidla, popisující styl napsání kódu našeho projektu. Tento soubor obsahoval pravidla, týkající se lokálních proměnných, funkci, konstant, struktur, vytváření modulů a dokonce úpravy příkazových závorek a cyklů. V budoucnu dodržování těchto pravidel velmi zpřehlednilo zdrojový kód.

3.3 Modul itrerror

Modul pro zpracování různých chyb, které mohou vyskytnout v době práci interpretu.
Obsahuje funkce *fatal_error()*, která ukončí program a vrací OS kód odpovídající chyby. Chyby jsou reprezentovany jako vyčtovy typ. Pokud dojde k chybě, funkce uvolní všechny zdroje, které OS alokovala programu a zavře zdrojový soubor.

3.4 Modul restab

Modul pro zjednodušení práci s dynamickou paměti.

Zdroje jsou uložené v Hašovací tabulce.

Pro alokace dynamické paměti modul používá wrapper funkci nad *malloc()* , *realloc() a free()* (*rtab_mallocr()*, *rtab_realloc() a rtab_free()* resp.)

Modul taky obsahuje funkce, která je schopna odstranit všechny dříve alokované dynamické zdroje.

3.5 Modul boofer

Modul byl vytvořen pro buferizaci toku symbolů.
Poskytuje 4 funkce pro práci se vstupním řetězcem:
buf_getch() a buf_ungetch() - pracuji analogicke standardním funkcím.
buf_setlex() - sděluje buferu o začátku lexemy.
buf_getlex() - vrací ukazatel na dynamické alokovanou lexemu.
Na konci bufer předává tok symbolů lexikálnímu analýzatoru.
Pro realizaci buferizaci použili jsme techniku, popsanou v knize "Compilers: Principles, Techniques, and Tools. Second Edition" v kapitole V kapitole "Input Buffering" na stránkách 115-117.

4. Implementace

4.1 Lexikální analýza

Lexikální analýzator používá bufer a poskytované jím funkce pro práci se vstupním řetězcem pro transformaci toku symbolů v potok tokenů. Komentáře a bílé znaky se zpracujií také, ale scanner je nikam neuloží a pokračuje ve zpracování dalšího tokenu. Zpracované lexémy (= tokeny) jsou předávány syntaktickému analyzátoru.
Na rozpoznání klíčových slov používá modul *kwrecognizer*. Práce našeho lexikálního analýzatoru je realizovaná pomocí následujících funkcí:
lex_token_get_next() - provádí potřebnou konverzaci potoku symbolů, ukládá příští token a vrátí ukazatel na něho.
lex_token_unget() - vrátí token do potoku tokenů.
lex_token_get_current() - vrátí ukazatel na aktuální uložený token.
Na závěr, lexikální analýzator vrátí potok tokenů a předává ho syntaktickému analýzatoru. Analýza a konverzaci potoku symbolů se provádí pomocí konečného automatu, zobrazeného na straně 9.

4.2 Tabulka symbolů

Ve vybraném námi variantu zadání, datová struktura musí být reprezentovaná pomocí hašovací tabulky.
Data, která jsou uložená v tabulce, se dělí na dva typů: informace o proměnné a informace o funkcí.
Vkládání dát do tabulky se provádí pomocí funkce *symtab_insert_item()*
Pro výběr dát z tabulky používáme funkce *symtab_get_item_info()*
Každý item je reprezentovan ve formě vyčtového typu *type_item*.
Pro hašovaní klíčů používáme algoritmus **SDBM**, protože může dobře kódovát bity, což způsobuje lepší distribuce klíčů a menší rozdělení.

4.3 Syntaktický analyzátor

Na vstupu syntaktický analyzátor očekává porok tokenu, který mu předává lexikální analyzátor pomocí funkce l*ex_token_get_next()*. Pro ověření syntaxe načtených tokenů je implementován rekurzivní sestup podle pravidel precedenční a prediktivní LL gramatiky. Výrazy se zpracovávají pomocí precedenční gramatiky metodou zdola nahoru, vše ostatní prediktivní gramatikou metodou shora dolů. Instrukční list se generuje pomocí funkce *init_list_instructions()*, která se nachází v modulu *ilist*.

4.4 Interpret

Interpret má na starosti vykonávání programu, zapsaného ve zdrojovém souboru. Provádí typovou kontrolu, tzn. zda je urcitá operace nad danými datovými typy definovaná. Interpret postupně prochází instrukční pásku a na základě aktuální instrukce provádí odpovídající akce.

Na začátku interpretace instrukce vyvolání funkce vytváří simulace zásobníkového rámce, ve kterém se nachází hodnoty všech lokálních proměnných, vztahující se k této funkci.

4.5 Knuth-Morris-Prattův vyhledávací algoritmus

Tento algoritmus slouží k vyhledávání podřetezce v jiném řetezci. Má lineární časovou složitost $O(m + n)$ a je význacný tím, že zachovává minimální možný počet celkových porovnání, tzn. neporovnává žádnou odpovídající dvojici (vzoru a řetezce) dvakrát. Algoritmus ke své práci využívá konečný automat, ve kterém z každého uzlu vychází tolik hran, kolik je znaků abecedy. Automat načítá znaky, a pokud dojde ke shodě, posune se na další stav, v opačném případě se vrátí na stav předchozí.

4.6 List-Merge sort

List Merge Sort pracuje na principu setřiďování posloupností, přičemž využívá řazení bez přesunu položek. V prvním kroku se zřetězí všechny neklesající posloupnosti, jejich začátky se přitom uloží do pomocného seznamu. Potom se v každém kroku setřídí dvě seřazené posloupnosti do jedné. Algoritmus končí, když je v seznamu jen jedna seřazená posloupnost.

5. Gramatika

program -> class_list

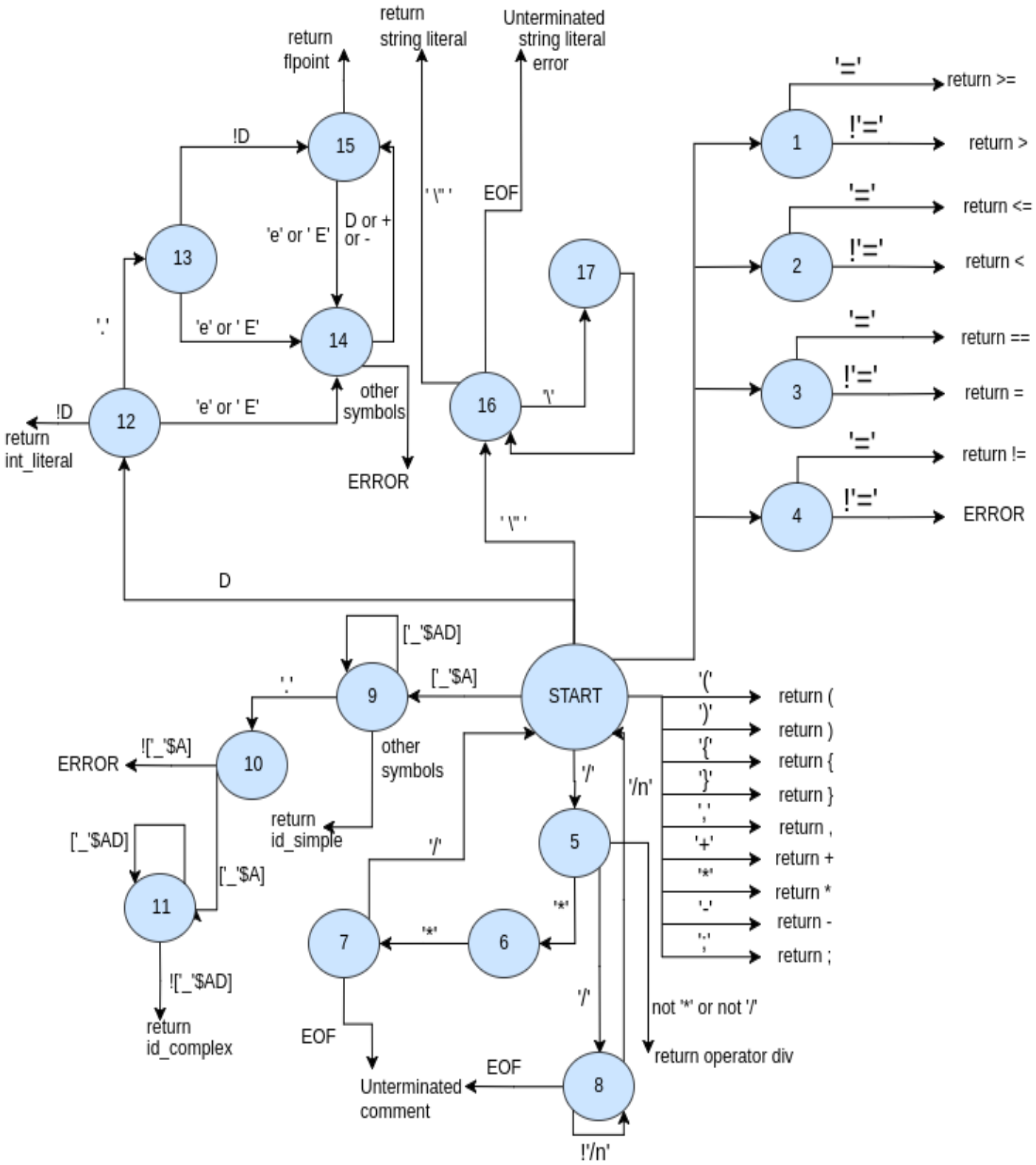
class_list -> class_def class_list | EPSILON
class_def -> 'class' ID { class_instance_list }
class_instance_list -> func_def class_instance_list | EPSILON
class_instance_list -> var_global_dec class_instance_list
var_global_dec -> 'static' type ID dec_init
dec_init -> ';' | '=' expr ';'

func_def -> 'static' ret_type ID (arg_list) { func_body }
arg_list -> arg arg_next | EPSILON
arg_next -> ',' arg arg_next | EPSILON
ret_type -> type | 'void'
arg -> type ID
type -> 'int' | 'double' | 'String'
func_call -> 'ID' assignment_void_call
assignment_void_call -> '=' ID '(' param_list ')' ';' | (' param_list ')' ';'
param_list -> param param_next | EPSILON
param_next -> ',' param param_next | EPSILON
param -> INT_LIT | DOUBLE_LIT | STRING_LIT | ID
func_body -> statement func_body | EPSILON
func_body -> var_local_dec var_def_init func_body
var_def_init -> ';' | '=' expr ';'
var_local_dec -> type ID

statement_seq -> { statement_list }
statement_list -> statement statement_list | EPSILON
statement -> ID '=' expr ';'
statement -> condition | loop
statement -> func_call
statement -> 'return' return_expr
return_expr -> ';' | expr ';'

condition -> 'if' '(' expr ')' statement_seq 'else' statement_seq
loop -> 'while' '(' expr ')' statement_seq

6. Diagram přechodů konečného automatu



7. Rozdělení práce podle jednotlivých modulů

Kiselevich Roman (xkisel00) se jako první hridně vydal po velmi těžké cestě samotné implementace projektu. Je zodpovědný za napsání většiny modulů:

boolean.h*, *ifj16.c*, *buffer.c*, *buffer.h*, *qoui.c*, *qoui.h*, *restab.h*, *restab.h*, *soui.c*, *soui.h*, *parser.c*, *parser.h*, *soexpitems.c a ***soexpitems.h*.**

But Andrii (xbutan00) a Kharytonov Danylo (xkhary00) většinu času pracovali spolu, a výsledkem jejich práce byly následující moduly:

***lexer.c*, *lexer.h*, *prectab.c*, *prectab.h*, *token.h*, *ilist.c*, *ilist.h*.**

Inhliziian Bohdan (xinhli00) měl na starosti moduly ***fcallstack.c*, *fcallstack.h*, *ifj16funcs.c*, *ifj16funcs.h*, *symtab.c a symtab.h*.**

Pavel Niahodkin (xniaho00) vypracoval moduly ***cmptcheck.c*, *cmptcheck.h*, *exprtree.c*, *exprtree.h*, *inter.c*, *inter.h*, *itrerror.c*, *itrerror.h*, *kwrecognizer.c*, *kwrecognizer.h*** a udělal tuto dokumentci :-)