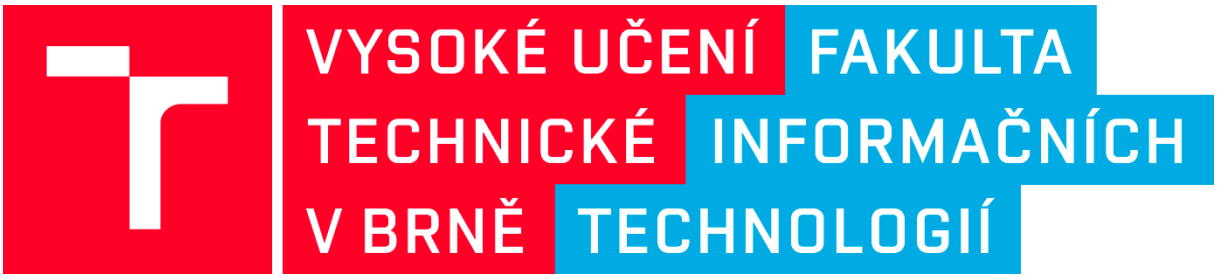


Dokumentácia k projektu z predmetu IFJ a IAL

Implementácia interpretu imperatívneho jazyka IFJ16

Tým 016, varianta b/4/II

5. decembra 2016

Riešitelia:

Sámel Šimon, xsamel02 , 0%
Patrik Sztefek, xsztef02 , 25%
Tomáš Szűcs, xszucs01 , 25%
Marek Šipoš, xsipos03 , 25%
Jakub Štol, xstolj00 , 25%

1. Úvod

V tejto dokumentácii je popísaný vývoj a implementácia interpreta imperatívneho jazyka IFJ16, ktorý je zjednodušenou podmnožinou jazyka Java SE 8. Celá dokumentácia je rozdelená do kapitol, v ktorých sú popísané jednotlivé časti interpreta. Od návrhu, implementácie až po samotnú prácu v tíme.

2. Návrh interpretu

Z dôvodu veľkosti a náročnosti projektu sme boli nútení rozdeliť na jednotlivé časti(spomenuté nižšie), aby sme si zachovali prehľadnosť . A hlavne zabráneniu zbytočnému chaosu. Preto sme projekt rozdelili do piatich hlavných častí:

- Lexikálna Analýza
- Syntaktická analýza
    a) prediktívna
    b) precedentná
- Sémantická analýza
- Generátor trojadresného kódu
- Interpret

3. Implementácia

V tejto kapitole sa popisuje implementácia jednotlivých častí interpretu.

3.1 Lexikálny Analyzátor

Lexikálny analyzátor je konečný automat s epsilon prechodmi(slúžia k čo najpresnejšiemu určeniu tokenu pomocou medzi stavov), ktorý načíta vstup zo zdrojového súboru. Následne rozlišuje od jednoduchých znakov až po kľúčové slová, ktoré prevádza na tokeny. Token je tvorený štruktúrou, v ktorej sa nachádza číslo riadku a typ.(obrázok konečného automatu viz. Príloha A)

3.2 Syntaktický analyzátor

Úlohou syntaktického analyzátoru je skontrolovať syntax a poskytnúť priebeh programom pre ostatné analýzy. Využíva sa kombinácia syntaktické analýzy zhora dole a zdola nahor. Pre výrazy sa využíva teda precedentná analýza (zdola nahor), pre zbytok analýza prediktívna(zhora dole).Využívame prediktívnu analýzu namiesto rekurzívnej, aj napriek tomu , že je doporučená rekurzívna. Je to z dôvodu, že umožňuje voľne meniť návrh gramatiky a LL tabuľky. Zároveň je rýchlejšou ale je o dosť obťažnejšou na implementáciu. Použitá gramatika je LL(1) gramatika. Pretože však jazyk IFJ16 nejde popísať čistou LL(1) gramatikou, bolo pre popísanie jazyka použito drobných heuristík, ktoré mierne “ohýbajú” princíp syntaktickej analýzy.(Príloha B)

3.3 Semanticky analyzátor

Preberá symboly zo syntaktickej analýzy, u ktorých kontroluje ich význam. Zaznamenané symboly ukladá do tabuľky symbolov . Tabuľka pre triedy ukladá všetky statické premenné pre danú triedu a ich funkcie v triede. U premenných sa ukladá identifikátor(v tabuľke symbolov skrátený), dátový typ a poradie v deklarácii triedy. U funkcii ukladá identifikátor(v tabuľke symbolov úplný), návratový dátový typ, typ parametra a ich presné poradie

(kontrola pri volaní funkcie), pozíciu inštrukcie začiatku funkcie na páske. Tabuľka pre funkcie zaznamenáva všetky premenné definované vo vnútri funkcie a ich parametre. U premenných sa ukladá identifikátor, dátový typ, poradie deklarácie v danej funkcii, kde parametre sú prvé. Do tejto tabuľky sa uložia všetky parametre funkcie. Pri kontrole významu symbolov prebieha ku kontrole dátových typov a kontrole deklarácií/redeklarácií. Kontrola dátových typov prebieha pokiaľ zaznamená výraz, vyhodnotí výsledný dátový typ ten sa potom používa ďalej - napr. v if, while, alebo aj ako priradení. Pokiaľ ide o priradenie musíme vždy zistiť, či pravý výraz odpovedá ľavému, prípadne, či sa dá pravý na ľavý previesť. Kontrola dátových typov sa uplatňuje aj pri typov parametrov. Overí sa, či všetky parametre volané funkciou sú správne – vrátane ich poradia a počtu. Dochádza k spolupráci s tabuľkami symbolov tried. Kontrola očakávanej právd. hodnoty u rôznych jazykových konštrukcií. V ktorej overujeme, že tam, kde to je potrebné, sa naozaj vyskytuje výraz obsahujúci právd. hodnotu. Týka sa to týchto prípadov:

- if (PRAVD. HODNOTA) { … } else { … }
- while (PRAVD. HODNOTA) { … }

Pri kontrole deklarácií/redeklarácií sa kontroluje či funkcia/premenná existuje a všetko potrebné je vyriešené.

3.4 Interpret a Generátor trojadresného kódu

Rozhodli sme sa použiť generátor trojadresného kódu, používajúci naše vlastné inštrukčné sady. Generátor rozlišuje či sa jedná o vytvorenie plnohodnotnú inštrukciu a inštrukciu skoku(návestie). Toto rozlišovanie sa deje kvôli prehľadnosti kódu. Aby mohol interpreter využiť tieto inštrukcie, sú zaznamenané v tabuľke inštrukcií. Hneď vedľa nej je tabuľka obsahujúca návestie pre skoky, ako doplnok k tabuľke inštrukcií. Okrem inštrukcii skoku a návrat z funkcie, interpreter spracúva inštrukcie lineárne. U skokovej inštrukcie a návratu z funkcie závisí od parametrov ako sa zmení číslo práva v danej inštrukcii.

3.5 Booyer-Moore algoritmus

Tento algoritmus sme použili pri vstavanej funkcii **find**, ktorá má za úlohu nájsť pod reťazec v reťazci. Booyer-Mooruv algoritmus používa viacero variant heuristík. Slúžia na nájdenie zhodných znakov v reťazci a pod reťazci. Vďaka tomuto vyhľadávaniu zhodných, respektíve nezhodných znakov môže algoritmus preskočiť určité znaky, ku ktorým sa už nevracia. Vďaka tejto schopnosti môže algoritmus rýchlo prejsť daný reťazec. Počas implementácie algoritmu Booyer-Moore Prvá heuristika sme sa inšpirovali z opory IAL-2016-verze-16.C.

3.6 List-Merge Sort

Pre ďalšiu našu vstavanú funkciu – sort bol použitý List-Merge Sort. Na jeho implementovanie sme si vytvorili pomocne pole indexov. Do nami vytvoreného poľa sme vložili už zlúčené položky, ktoré neboli zoradené. Ukazatele, ktoré ukazujú na prvú položku sme uložili do listu. Následne sa vyberú prvé dve nezoradené položky. Zoradia a zlúčia sa do jednej , ktorá je následne presunutá na koniec zoznamu. Tieto úkony sa vykonávajú až kým nie sú zoradené. . Počas implementácie algoritmu List-Merge Sort sme sa inšpirovali z opory IAL-2016-verze-16.C.

3.7 Tabuľka symbolov

Tabuľku symbolov sme implementovali, ako tabuľku s rozptýlenými položkami, alebo prezývaná aj ako **hashtable.** Má spoločnú vlastnosť s binárnym stromom, že sú prispôsobené na rýchle vyhľadávanie. Je unikátna svojou funkciou tzv. hashovaciou funkciou. Hashovacia funkcia ma tzv. vyhľadávací kľuč, vďaka ktorému vieme kde máme našu položku hľadať alebo umiestňovať.

4.Vývoj interpretu

V tejto kapitole je popísane rozdelenie práce na interpretu, až po použité nástroje pri vývoji.

4.1 Rozdelenie práce

Prácu pred začatím vývoja interpretu rozdeľoval náš vedúci Patrik Sztefek. Rozdeľoval ju spôsobom, kto bol ako zručný v programovaní. Ale zároveň nám dal nám možnosť si vybrať, ktorú časť chceme riešiť. Aj keď sa môže zdať, že rozdelenie bodov neprináleží k odvedenej práce jednotlivých členov, každý sa snažil ako len mohol. Nižšie uvedené popisy slúžia pre predstavu, kto akú časť problematiky riešil.

Patrik Sztefek - Generátor, algoritmy, interpret, testovanie
Tomáš Szűcs – Lexikálny analyzátor, výpomoc, dokumentácia,
Marek Šipoš – Syntaktický analyzátor, Sémantický analyzátor, testovanie, Generátor
Jakub Štol - Lexikálny analyzátor, vstavané funkcie, výpomoc

4.2 Komunikácia a schôdze

Už pred začatím vývoja sme vedeli, že nebude čas na pravidelne schôdze z dôvodu odlišnosti rozvrhu každého člena, tak sme sa stretávali len vo vážných prípadoch. Namiesto schôdze sme zvolili variantu Facebook-ovej skupiny a Facebook instant-messaging. Výhoda spočíva v tom, že vieme komunikovať z ktoréhokoľvek miesta. Zároveň podávať správu o pokroku na svojej práci.

4.3 Použité nástroje

Na vývoj interpretu sa použil verzovaci systém **Git**. Ako úložný priestor pre náš projekt poslúžil **GitHub**. Z dôvodu dostupnosti na internete a zároveň má zabudované funkcie. Ktoré sprehľadňujú zmeny, a to v konkrétnom súbore na konkrétnom riadku. Aj z dôvodu poskytnutia voľnej licencie pre študentov, aj kvôli chuti vyskúšať moderné trendy. Ako ďalší nastroj sme použili **Google docs**. Najviac užitočný bol vo fázach vývoja, keď sme mali obrovské množstvo hlavičkových a zdrojových súborov. Zároveň slúžil ako poznámkový blok, či už pre brainstorming alebo popis jednotlivej časti interpretu.

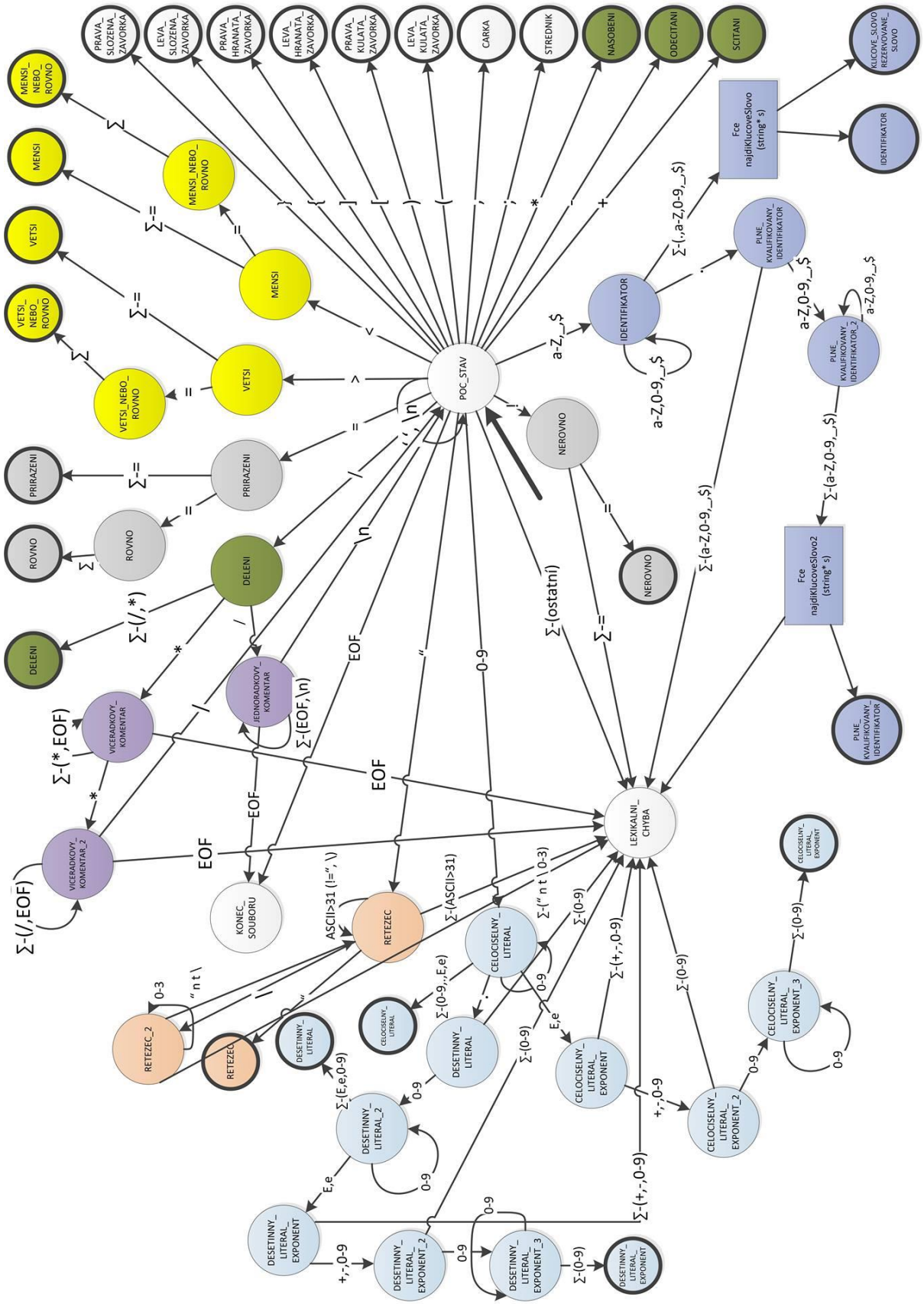
4.4Vývojový cyklus

Vývoj sa od samého mal charakteristiky k V-modelu, či už z pohľadu testovania alebo samotnej implementácie. Z dôvodu, že každá časť interpretu sa navrhla, implementovala následne testovala.

5. Použitá literatúra

Ifj2016.pdf
Opora-IAl-2016-verze-C.pdf
Jak-na-projekt.pdf

Príloha A: Konečný automat Lexikálneho analyzátoru



Priloha B: LL gramatika a precedencna tabulka

01 PROGRAM → TRIDA PROGRAM
02 PROGRAM → eof
03 TRIDA → class identifikator { SEZNAM-DEFINIC-STATIC }

04 SEZNAM-DEFINIC-STATIC → static DATOVY-TYP DEFINICE-STATIC SEZNAM-DEFINIC-STATIC
05 SEZNAM-DEFINIC-STATIC → ε
06 DEFINICE-STATIC → DEFINICE-FUNKCE
07 DEFINICE-STATIC → DEFINICE-PROMENNA ;
08 DEFINICE-PROMENNA → identifikator DEF-PROM-KONEC
09 DEF-PROM-KONEC → PRIRAZENI
10 DEF-PROM-KONEC → ε

11 DEFINICE-FUNKCE → f_identifikator (SEZNAM-PARAMETRU) SLOZENY-PRIKAZ
12 SEZNAM-PARAMETRU → PARAMETR-PRVNI PARAMETR-DALSI
13 SEZNAM-PARAMETRU → ε
14 PARAMETR-PRVNI → primitivni_typ identifikator
15 PARAMETR-DALSI → , primitivni_typ identifikator PARAMETR-DALSI
16 PARAMETR-DALSI → ε
17 SEZNAM-VSTUPU → vyraz VSTUP-DALSI
18 SEZNAM-VSTUPU → identifikator VSTUP-DALSI
19 SEZNAM-VSTUPU → ε
20 VSTUP-DALSI → , VSTUP-KONEC
21 VSTUP-DALSI → ε
22 VSTUP-KONEC → vyraz VSTUP-DALSI
23 VSTUP-KONEC → identifikator VSTUP-DALSI

24 SLOZENY-PRIKAZ → { BLOK-PRIKAZU }
25 BLOK-PRIKAZU → PRIKAZ BLOK-PRIKAZU
26 BLOK-PRIKAZU → ε
27 PRIKAZ → primitivni_typ DEFINICE-PROMENNA ;
28 PRIKAZ → vyraz ;
29 PRIKAZ → identifikator POUZITI ;
30 PRIKAZ → f_identifikator VOLANI-FUNKCE ;
31 PRIKAZ → return NAVRAT-KONEC ;
32 PRIKAZ → if (vyraz) SLOZENY-PRIKAZ else SLOZENY-PRIKAZ
33 PRIKAZ → while (vyraz) SLOZENY-PRIKAZ
34 POUZITI → PRIRAZENI
35 POUZITI → ε
36 VOLANI-FUNKCE → (SEZNAM-VSTUPU)
37 NAVRAT-KONEC → vyraz
38 NAVRAT-KONEC → identifikator
39 NAVRAT-KONEC → ε
40 PRIRAZENI → = PRAVA-STRANA
41 PRAVA-STRANA → vyraz
42 PRAVA-STRANA → identifikator
43 PRAVA-STRANA → f_identifikator VOLANI-FUNKCE

44 DATOVY-TYP → void
45 DATOVY-TYP → primitivni_typ

vyraz → Předá se precedenční synt. analýze - symboly: identifikator(proměnné) číslo řetězec ( ) aritmetické + relační operátory

Precedentná tabuľka

	n	(	)	+	-	*	/	<	>	<=	>=	==	!=	$
n			>	>	>	>	>	>	>	>	>	>	>	>
(	<	<	=	<	<	<	<	<	<	<	<	<	<	
)			>	>	>	>	>	>	>	>	>	>	>	>
+	<	<	>	>	>	<	<	>	>	>	>	>	>	>
-	<	<	>	>	>	<	<	>	>	>	>	>	>	>
*	<	<	>	>	>	>	>	>	>	>	>	>	>	>
/	<	<	>	>	>	>	>	>	>	>	>	>	>	>
<	<	<	>	<	<	<	<					>	>	>
>	<	<	>	<	<	<	<					>	>	>
<=	<	<	>	<	<	<	<					>	>	>
>=	<	<	>	<	<	<	<					>	>	>
==	<	<	>	<	<	<	<	<	<	<	<			>
!=	<	<	>	<	<	<	<	<	<	<	<			>
$	<	<		<	<	<	<	<	<	<	<	<	<	E