**Formální jazyky a překladače**

2016/2017

**Dokumentace interpretu imperativního jazyka IFJ16**

**Tým 013, varianta b/3/I**

Rozšíření: BOOLOP

**Seznam autorů:**

Šuba Adam xsubaa00 25% vedoucí týmu  
Paliesek Jakub xpalie00 25%

Šuhaj Peter xsuhaj02 25%

Tóth Adrián xtotha01 25%

**Obsah**

[1. Úvod 2](#_Toc468468808)

[2. Návrh 3](#_Toc468468809)

[2.1. Vývojový cyklus 3](#_Toc468468810)

[3. Interpret 3](#_Toc468468811)

[3.1. Lexikálny analyzátor 3](#_Toc468468812)

[3.2. Syntaktický analyzator 3](#_Toc468468813)

[3.3. Sémantický analyzátor 4](#_Toc468468814)

[3.4. Interpret 4](#_Toc468468815)

[4. Algoritmy 5](#_Toc468468816)

[4.1. (b) vyhľadávanie s použitím Boyer-Moorovho algoritmu 5](#_Toc468468817)

[4.2. (3) radenie pomocou Shell sort algoritmu 5](#_Toc468468818)

[4.3. (I) implementácia tabuľky symbolov pomocou binárneho vyhľadávacieho stromu 5](#_Toc468468819)

[5. Práca v tíme 5](#_Toc468468820)

[6. Záver 5](#_Toc468468821)

[7. Literatúra 5](#_Toc468468822)

[8. Príloha 6](#_Toc468468823)

[8.1. Diagram konečného automatu lexikálnej analýzy 6](#_Toc468468824)

[8.2. LL-gramatika syntaktického analyzátora 7](#_Toc468468825)

[8.3. Precedenčná tabuľka syntaktického analyzátora 8](#_Toc468468826)

# Úvod

Dokumentácia popisuje vývoj interpretu imperatívneho jazyka IFJ16. Jazyk IFJ16 je podmnožina jazyka Java SE 8, čo je staticky typovaný objektovo orientovaný jazyk.

Úlohou nášho interpretu je kontrola zdrojového kódu napísaného v jazyku IFJ16 a následne jeho interpretácia.

Nami zvolený Tím 013 mal priradenú variantu b/3/I. Varianta obsahovala: (b) vyhľadávanie s použitím Boyer-Moorovho algoritmu, (3) radenie pomocou Shell sort algoritmu, (I) implementácia tabuľky symbolov pomocou binárneho vyhľadávacieho stromu.

Dokumentácia je rozdelená na kapitoly, ktoré bližšie popisujú jednotlivé časti interpretu a spôsob ich riešenia. Taktiež obsahuje aj popis algoritmov, popis rozdelenia úloh a spôsobu práce v tímu. V závere dokumentácie sa nachádza zhrnutie našej práce.

V prílohe sa nachádzajú: diagram konečného automatu lexikálnej analýzy, LL-gramatika a precedenčná tabuľka syntaktického analyzátora.

Dokumentace popisuje vývoj interpretu imperativního jazyka IFJ16. Jazyk IFJ16 je podmnožinou jazyka Java SE 8, což je staticky typovaný, objektově-orientovaný jazyk.

Úlohou našeho interpretu je kontrola zdrojového kódu napsaného v jazyku IFJ16 a jeho následná interpretace.

Zvolená varianta zadání Tým 013 má parametry b/3/I:

* (b) vyhledávání podřetězce s použitím Boyer-Mooreova algoritmu
* (3) řazení s použitím algoritmu Shell sort
* (I) implementace tabulky symbolů použitím binárního vyhledávacího stromu

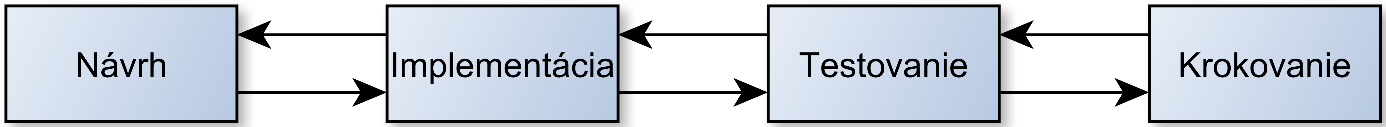
Dokumentace je rozdělená na kapitoly, které blíže popisují jednotlivé části intepretu a způsob jejich řešení. Taktéž obsahuje i popis použitých algoritmů, rozdělení úloh a způsob práce v týmu. V závěru dokumentace se nachází shrnutí naší práce.

V příloze se nachází:

* diagram konečného automatu lexikální analázy
* LL-gramatika pro syntaktickou analázu shora dolů
* precedenční tabulka pro syntatickou analýzu zdola nahoru

# Návrh

## Vývojový cyklus



DOPISAT

# Interpret

Implementaci interpretu lze rozdělit do 4 logických celků. Hlavní komponentou je syntaktický analyzátor, neboli “parser”, který řídí činnost lexikálního analyzátoru, “scanneru”, a také provádí sémantické akce sémantického analyzátoru. Poslední komponentou je samotný interpret, který provádí interpretaci instrukcí, vygenerovaných během kontroly vstupního programu.

Kontrola správnosti programu je prováděna ve dvou průchodech, v prvním se některé části zdrojového kódu úplně ignorují (zéjména výrazy, které jsou závislé na sémantických akcích, více dále). Účelem prvního průchodu je naplnění tabulky symbolů.

## Lexikálny analyzátor

Jediným úkolem lexikálního analyzátoru je na žádost syntaktického analyzátoru přečíst další symbol ze zdrojového souboru, klasifikovat jej a předat ve formě tokenu zpět syntaktickému analyzátoru.

Implementace je založena na konečném automatu. Bílé znaky a komentáře jsou ze zdrojového kódu odignorovány. Pokud řetězec dojde do koncového stavu reprezentující nějaký lexém a následující znak nepatří mezi znaky, které může daný lexém obsahovat a zárověň znak patří mezi znaky které mohou bezprostředně za daným lexémem následovat, je typ vraceného tokenu nastaven na typ odpovídající tomuto lexému. Některým tokenům je ještě přidělen atribut, obsahující užitečná data získaná z lexému. Pokud je v některém stavu přečten znak, který není v daném stavu podporován, popřípadě pokud je v koncovém stavu přečten znak, který nemůže následovat bezprostředně za daným lexémem, dojde k chybě. Poslední načtený znak, který rozhodl o ukončení konečného automatu, je vrácen zpět do vstupního souboru.

V případě lexikální chyby vrací interpet návratový kód 1.

Poznámka:

Při druhém průchodu syntaktické analýzy již není lexikální analyzátor činný. Při každém získání tokenu během prvního průchodu je daný token uložen do datového typu fronta a v druhém průchodu jsou tokeny brány z této fronty.

Lexikálny analyzátor, nazývaný aj ako „scanner“, je prvá časť interpretu. Lexikálny analyzátor je implementovaný je vo forme konečného automatu. Vstupom je textový súbor z ktorého sa čítajú jednotlivé znaky ktoré sú vyhodnocované.

Konečný automat prevádza znaky resp. postupnosť znakov t.j. lexémy do tokenov. Znaky typu medzera, komentáre sú ignorované. Lexikálny analyzátor má špecifikované určité znaky ktoré môže prijímať. Ak znak nepatrí medzi prijímané znaky, je vyhlásená lexikálna chyba. Pri chybnej štruktúre aktuálneho lexému je program ukončený s návratovou hodnotou 1.

Lexikálny analyzátor sa využíva keď syntaktická analýza žiada o token. Token je štruktúra ktorá obsahuje typ tokenu a typu zodpovedajúce dáta. Token sa z lexikálne analýzy posiela syntaktickej analýze práve vtedy keď konečný automat skončil v koncovom stave. Konečný automat sa spúšťa zakaždým keď je žiadaný token.

## Syntaktický analyzator

Mozkem celé kontroly správnosti zdrojového kódu je synktatický analyzátor. Samotnou syntaktickou analýzu můžeme rozdělit do dvou částí: kontrola rekurzivním sestupem a kontrola správnosti výrazů precedenční analýzou.

Rekurzivní sestup je metoda syntaktické analýzy shora dolů, která implementuje pravidla LL-gramatiky (viz příloha) na základě LL tabulky. Podle aktualniho stavu analyzátoru a vstupního tokenu je rozhodnuto, které pravidlo se zvolí. Metoda využívá rekurzivního volání funkce implementující dané pravidlo.

Pro kontrolu správnosti výrazů se využívá precedenční analýza, která je metodou syntaktické analázy zdola nahoru. Neboť bylo nutné nějakým způsobem napojit precedenční analýzu na rekurzivní sestup, přištoupili jsme k zavedení speciálního pseudoterminálu E do naší LL-gramatiky. V místě kontroly, kde by se měl nacházet tento terminál, se analyzátor přepne do režimu, kdy pouze žádá další a další tokeny a ukládá je do datového typu fronta. Tento režim je ukončen v moment, kdy narazí na token, který není ve výrazech podporován. Takto vytvořená fronta je poté předána precedenční analýze ke kontrole. Speciální případem, kdy je nutné lehce pozměnit chování tohoto procesu jsou výrazy podmínek ve větvení a v cyklu. Zde je nutné počítat levé a pravé závorky tak, aby byl analyzátor schopen poznat, kdy nastal konec podmínky a fronta je naplněná.

Neboť neimplementujeme rozšíření FUNEXP je pro nás funkce ifj16.print() speciální syntaktickou konstrukcí, neboť jí lze jako parametr dát jednoduchou konkatenaci, jejíchž výsledkem je vždy datový typ string. Z tohoto důvodu jsme se rozhodli nezahrnovat tohle chování do LL-gramatiky, ale vyřešit to jako speciální případ až při samotné implementaci analyzátoru. V druhém průchodu zdrojového souboru již díky sémantickým akcím analyzátor ví, že volaná funkce je ifj16.print() a na základě této informace se přepne do speciálního režimu, podobného přípavě na precedenční analýzu. Rozdíl je v tom, že povolené tokeny jsou pouze proměnné, literály a operátor konkatenace “+”. Při ukončení tohoto režimu je zkontrolováno, zda-li alespoň jeden token byl typu string. Poté je konkatenace vyhodnocena precedenční analýzou. V naší implementaci je tedy výsledek stejný jako při konkatenaci mimo volání této funkce.

V případě syntaktické chyby vrací interpret návratový kód 2.

Druhou časťou interpretu je syntaktická analýza „parser“. Úlohou syntaktického analyzátora je kontrola reťazcov tokenov či je syntakticky správne napísaný program. Ak k danému reťazci tokenov je úspešne vytvorený derivačný strom, program sa považuje za syntakticky správny. Niektoré časti syntaktickej kontroly sú spúšťané na základe sémantických akcii (DOPISAT).

Pri chybe je vyhlásená syntaktická chyba. Pri chybnej syntaxi programu je program ukončený s návratovou hodnotou 2.

## Sémantický analyzátor

Ďalšou časťou interpretu je sémantická analýza. Vstupom je derivačný strom od syntaktického analyzátora. Sémantický analyzátor skontroluje sémantické aspekty ktorými sú kontrola typov a kontrola deklarácii premenných stručne povedané správnosť operácii. Výstupom sémantického analyzátora je abstraktný syntaktický strom. Z abstraktného syntaktického stromu sú vytvorené 3-adresné inštrukcie ktoré sú potrebné pre samotný interpret.

Ak nastala chyba, je vyhlásená sémantická chyba a program je ukončený s návratovou hodnotou 3,4,6 (hodnota závisí od typu chyby).,

4.2. Tabuľka symbolov

Tabuľka symbolov (ďalej len TS) je podľa zadania implementovaná ako binárny vyhľadávací strom. Vzhľadom na to, že v jazyku IFJ16 existujú rôzne rozsahy platnosti identifikátorov a triedy umožňujú vytvárať oddelené menné priestory, sme TS navrhli ako trojúrovňovú kaskádu vyhľadávacích štruktúr, kde na prvej úrovni je jediná tabuľka tried. Každá trieda má potom vlastnú tabuľku členov (spoločnú pre statické premenné a funkcie). Funkcie majú ďalej vlastné tabuľky lokálnych premenných.

Hodnoty statických premenných sú uložené priamo v odpovedajúcich položkách TS. Hodnoty lokálnych premenných kvôli možnému rekurzívnemu volaniu funkcií niesu uložené priamo v TS, ale ich hodnoty si ukladá interpret na svojom zásobníku počas vykonávania inštrukcií.

Počas syntakticko-sémantickej analýzy vzniknú ďalšie pomocné prvky, kde sú uložené literály a medzivýsledky z generovania aritmetických inštrukcií pre výrazy. Kedže nemajú identifikátory, niesu uložené vo vyhľadávacej štruktúre BVS, ale iba v lineárnych zoznamoch. V globálnom zozname pomocných elementov sú uložené všetky literály a pomocné premenné vzniknuté pri generovaní inicializačného 3AC pre statické premenné. V lokálnych zoznamoch sú uložené pomocné premenné z výrazov nachádzajúcich sa v telách daných funkcií.

## Interpret

Poslednou fázou je interpret. Interpret odsimuluje vstupnú inštrukčnú pásku ktorá sa skladá z trojadresných inštrukcii a na základe nich vykoná interpretáciu. Interpret má na starosti hlavne typovú konverziu dátových typov pri operáciách. Napríklad pri súčte int a double musí vykonať konverziu int na double.

# Algoritmy

Všetky nižšie popísané algoritmy sú implementované v súbore ial.c.

## (b) vyhledávání podřetězce použitím Boyer-Mooreova algoritmu

Algoritmus Boyera-Moora bol objavený Robertem S. Boyerem and J. Strotherem Moorem v roku 1977. Boyer-Moorov algoritmus je ten najefektívnejší algoritmus na vyhľadane v reťazci podreťazec. Funguje na základe dvoch heuristík a na princípe spracovania reťazca zprava doľava. Podreťazec vyhodnocuje od posledného znaku k prvému znaku. Heuristické funkcie spracujú podreťazec najskôr do poľa kam sa uložia hodnoty skokov.

Algoritmus byl objeven Robertem S. Boyerem a J Strotherem Moorem v roce 1977. Jedná se o nejefektivnější algoritmus na vyhledávání podřetězce v řetezci. Funguje na základě dvou heuristik a na principu zpracování řetězce zprava doleva. Podřetězec je vyhodnocován od posledního znaku k prvnímu. Heuristické funkce spracují podřetězec nejprve do pole, kam se uloží hodnoty skoků. ????

## (3) řazení použítím algoritmu Shell sort

Shell sort bol objavený Donaldom Shellom v roku 1959. Je to kvadratický triediaci algoritmus ktorého zložitosť je O(n^2) a slúži na usporiadanie postupnosti prvkov do poradia. Algoritmus pracuje „in situ“ a na základe vkladania.

Shell sort byl objevený Donaldem Shellem v roce 1959. Je to kvadratický radící algoritmus, jehož složitost je O(n2) ?????????? a slouží k uspořádání posloupnosti prvků do pořadí. Algoritmus pracuje „in situ“ a na základě vkládání.

## (I) implementace tabulky symbolů použitím binárního vyhledávacího stromu

Binárny vyhľadávací strom sa využíva pri tabuľke symbolov. Implementovaný bol vo forme rekurzie a s priechodom postorder. V strome sú uložené názvy funkcii a im zodpovedajúce dáta.

DOPICI UZ. Vyhledávací binární strom je binární strom, platí tedy, že je složen z kořene a levého a pravého podstromu. Oba podstromy jsou také vyhledávacími binárními stromy. Aby byl birnární strom vyhledávací, musí platit, že všechny uzly nalevo od kořene jsou menší než kořen a všechny uzly napravo jsou větší než kořen. Implementace tabulky symbolů byla popsána detailně v kapitole

# Práce v týmu

## Rozdělení práce

* Adam Šuba – lexikální analyzátor, syntaktická analyza shora dolů a zdola nahoru, dokumentace, vedoucí
* Jakub Paliesek – sémantický analyzátor, tabulka symbolů, algoritmy, generátor 3-adresného kódu
* Peter Šuhaj – interpret, algoritmy
* Adrián Tóth – testy, algoritmy, dokumentace

## Způsob práce

Pro práci byl zvolen systém správy verzí Git, hostovaný na serveru GitHub. Kromě osobní komunikace byla hlavním komunukačním kanálem skupinová konverzace na sociální síti Facebooku.

Práca prebiehala rýchlo a efektívne a tak sme mali plne funkčný interpret s menšími implementačnými chybami pripravený na prvé pokusné odovzdanie.

# Záver

Zmyslom tohto projektu bolo porozumenie práci interpretu a na získanie skúseností v tímovej práce.

Výsledkom našej tímovej práce je interpret imperatívneho jazyka IFJ16 doplnený o boolop rozšírenie.

# Literatúra

1. HONZÍK, Jan M., Algoritmy: Studijní opora. Verzia: 16-B, Brno: Vysoké učení Technické, 2014
2. https://sk.wikipedia.org/wiki/Shell\_sort

# Príloha

## Diagram konečného automatu lexikálnej analýzy

Konecny automat VLOZIT

## LL-gramatika syntaktického analyzátora

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Rule** | **Empty** | **First** | **Follow** | **Predict** |
| C-LIST → **class id {** MEMB-LIST **}** C-LIST | 0 | class, $ | $ | class |
| C-LIST → **$** | $ |
| MEMB-LIST → C-MEMB MEMB-LIST | 1 | static | } | static |
| MEMB-LIST → ε | } |
| C-MEMB → **static** C-MEMB1 | 0 | static | static, } | static |
| C-MEMB1 → TYPE **id** C-MEMB2 | 0 | void, int, double, String, boolean | static, } | int, double, String, boolean |
| C-MEMB1 → **void id** C-MEMBFUNC | void |
| TYPE → **int** | 0 | int, double, String, boolean | id | int |
| TYPE → **double** | double |
| TYPE → **String** | String |
| TYPE **→ boolean** | boolean |
| C-MEMBFUNC → **(** FN-DEF-PLIST **) {** FN-BODY **}** | 0 | ( | static, } | ( |
| C-MEMB2 → **=** E**;** | 0 | ;, =, ( | static, } | = |
| C-MEMB2 → **;** | ; |
| C-MEMB2 **→** C-MEMBFUNC | ( |
| FN-DEF-PLIST → ε | 1 | int, double, String, boolean | ) | ) |
| FN-DEF-PLIST → PAR-DEF FN-DEF-PLIST1 | int, double, String, boolean |
| FN-DEF-PLIST1 → **,** PAR-DEF FN-DEF-PLIST1 | 1 | "," | ) | , |
| FN-DEF-PLIST1 → ε | ) |
| PAR-DEF → TYPE **id** | 0 | int, double, String, boolean | ",", ) | int, double, String, boolean |
| FN-BODY → STAT FN-BODY | 1 | id, fqid, {, if, while, return, int, double, String, boolean | } | id, fqid, {, if, while, return |
| FN-BODY → TYPE **id** OPT-ASSIGN **;** FN-BODY | int, double, String, boolean |
| FN-BODY → ε | } |
| STAT-COM → **{** STAT-LIST **}** | 0 | { | else, id, fqid, {, if, while, return,  int, double, String, boolean, } | { |
| STAT-LIST → STAT STAT-LIST | 1 | id, fqid, {, if, while, return | } | id, fqid, {, if, while, return |
| STAT-LIST → ε | } |
| ID → **id** | 0 | id, fqid | (, =, ), “,” | id |
| ID → **fqid** | fqid |
| STAT → IDAS-CA **;** | 0 | id, fqid, {, if, while, return | fqid, id, {, if, while, return, int,  double, String, boolean, } | id, fqid |
| STAT **→** STAT-COM | { |
| STAT → **if** **(** E **)** STAT-COM **else** STAT-COM | if |
| STAT → **while** **(** E **)** STAT-COM | while |
| STAT **→ return** RET-VAL **;** | return |
| AS-CA → **(** FN-PLIST **)** | 0 | (, = | ; | ( |
| AS-CA → **=** ASSIGN | = |
| ASSIGN → E | 0 | E, id, fqid | ; | E |
| ASSIGN → ID **(** FN-PLIST **)** | id, fqid |
| OPT-ASSIGN → ε | 1 | = | ; | ; |
| OPT-ASSIGN → **=** ASSIGN | = |
| FN-PLIST → ε | 1 | id, fqid, string-literal, int-literal,  double-literal,  boolean-literal | ) | ) |
| FN-PLIST → VAL-ID FN-PLIST1 | id, fqid, string-literal,  int-literal, double-literal, boolean-literal |
| FN-PLIST1 → ε | 1 | "," | ) | ) |
| FN-PLIST1 → **,** VAL-ID FN-PLIST1 | , |
| VAL-ID → ID | 0 | id, fqid, string-literal, int-literal,  double-literal,  boolean-literal | ",", ) | id, fqid |
| VAL-ID → **string-literal** | string-literal |
| VAL-ID **→ int-literal** | int-literal |
| VAL-ID **→ double-literal** | double-litera |
| VAL-ID **→ boolean-literal** | boolean-literal |
| RET-VAL → E | 1 | E | ; | E |
| RET-VAL → ε | ; |

## Precedenčná tabuľka syntaktického analyzátora

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **+** | **-** | **\*** | **/** | **<** | **>** | **<=** | **>=** | **==** | **!=** | **(** | **)** | **&&** | **||** | **!** | **id** | **$** |
| **+** | **>** | **>** | **<** | **<** | **>** | **>** | **>** | **>** | **>** | **>** | **<** | **>** | **>** | **>** | **<** | **<** | **>** |
| **-** | **>** | **>** | **<** | **<** | **>** | **>** | **>** | **>** | **>** | **>** | **<** | **>** | **>** | **>** | **<** | **<** | **>** |
| **\*** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **<** | **>** | **>** | **>** | **<** | **<** | **>** |
| **/** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **<** | **>** | **>** | **>** | **<** | **<** | **>** |
| **<** | **<** | **<** | **<** | **<** |  |  |  |  | **>** | **>** | **<** | **>** | **>** | **>** | **<** | **<** | **>** |
| **>** | **<** | **<** | **<** | **<** |  |  |  |  | **>** | **>** | **<** | **>** | **>** | **>** | **<** | **<** | **>** |
| **<=** | **<** | **<** | **<** | **<** |  |  |  |  | **>** | **>** | **<** | **>** | **>** | **>** | **<** | **<** | **>** |
| **>=** | **<** | **<** | **<** | **<** |  |  |  |  | **>** | **>** | **<** | **>** | **>** | **>** | **<** | **<** | **>** |
| **==** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **>** | **>** | **<** | **>** | **>** | **>** | **<** | **<** | **>** |
| **!=** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **>** | **>** | **<** | **>** | **>** | **>** | **<** | **<** | **>** |
| **(** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **=** | **<** | **<** | **<** | **<** |  |
| **)** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** |  | **>** | **>** | **>** | **>** |  | **>** |
| **&&** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **>** | **>** | **>** | **<** | **<** | **>** |
| **||** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **>** | **<** | **>** | **<** | **<** | **>** |
| **!** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **<** | **>** | **>** | **>** | **<** | **<** | **>** |
| **id** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** | **>** |  | **>** | **>** | **>** | **>** |  | **>** |
| **$** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** | **<** |  | **<** | **<** | **<** | **<** |  |