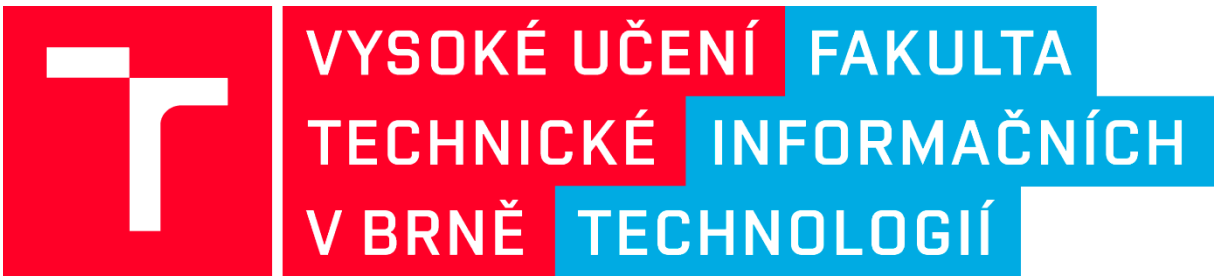
# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

# FAKULTA INFORMAČNÍCH TECHNOLÓGIÍ




Dokumentácia k projektu do predmetu IFJ a IAL

## Interpret jazyka IFJ16

Tím 066, varianta a/1/II

Dátum 11.12.2016


| | | |
|---|---|---|
| Vedúci tímu: | Jozef Urbanovský | (xurban66) – 20% |
| Členovia: | Adrián Tomašov | (xtomas32) – 20% |
| | Roman Dobiáš | (xdobia11) – 20% |
| | Adam Šulc | (xsulca00) – 20% |
| | Kristián Barna | (xbarna02) – 20% |


Rozšírenia: SIMPLE

# Obsah

# 1. Úvod

Táto dokumentácia popisuje návrh a implementáciu interpretu k jazyku IFJ16, ktorý je podmnožinou jazyka Java SE 8. V dokumentácii sa nachádza spôsob riešenia lexikálnej analýzy, syntaktickej analýzy, sémantickej analýzy a interpretu samotného. Zvolili sme si variantu a/1/II, ktorá zahŕňa implementáciu vstavanej funkcie find pomocou Knuth-Morris-Prattovho algoritmu, implementácia vstavanej funkcie sort pomocou Quick sortu a implementáciu tabuľky symbolov cez abstraktnú dátovú štruktúru tabuľky s rozptýlenými položkami. Súčasťou dokumentácie sú aj prílohy obsahujúce konečný automat lexikálneho analyzátoru, LL-gramatiku a precedenčnú tabuľku.

# 2. Cyklus vývoja projektu

## 2.1. Návrh a použitá metodika

Pri vývoji projektu sme si bližšie neurčili akým spôsobom budeme vyvíjať interpret IFJ16, ale dalo by sa to prirovnať k agilnej metodológii. Náš tím, zostavený už v na konci 2. semestra mal za cieľ nazbierať nutné informácie o tomto projekte už pred jeho zadaním. Po zadaní projektu sme si rozdelili prácu na jednotlivé celky, ktoré mali dopredu dohodnuté rozhranie a mohli byť tým pádom vyvíjané nezávisle na sebe. Po tom ako boli vyvinuté prototypy, resp. funkčné časti sme ich zintegrovali a poriadne otestovali. V návrhu sme vývoj projektu rozdelili na tieto časti, ktorých implementácia bude detailne popísaná v ďalších sekciách.

- Lexikálna analýza
- Syntaktická analýza
- Precedenčná analýza a sémantické kontroly
- Vstavané funkcie
- Tabuľka symbolov a dátové štruktúry
- Interpret

## 2.2. Spôsob práce v tíme

### 2.2.1. Zdieľanie a verziovanie

Nakoľko je na tomto projekte nutná spolupráca 4-5 študentov rozhodli sme sa pristúpiť k verziovaciemu systému *Git*, ktorý nám umožnil pracovať súčasne na projekte, vracať sa k predchádzajúcim verziám, vytvárať zálohy a zjednodušil následnej aj integráciu samotnú. Používali sme webovú službu *Github*, na ktorej ako študenti môžeme mať priváte repozitáre a patrí k najspoľahlivejším. V repozitári sme neuchovávali len zdrojové kódy, ale aj TODO, ktoré bolo upravované každý týždeň a prípadne na každej schôdzke. Následne sa v repozitári nachádza aj dokumentácia a všetky potrebné prílohy k projektu.

### 2.2.2. Schôdzky tímu a komunikácia

Na začiatku semestru po zadaní projektu sme sa stretávali približne každé 2 týždne aby sme prediskutovali momentálnu situáciu a ako pokračovať ďalej. Na prvej schôdzke sme zhodnotili schopnosti nášho tímu a podľa toho rozdelili úlohy, kto bude pracovať na akej časti. Dohodli sme si isté konvencie ako budeme postupovať, aké rozhranie budeme používať a podobné. Schôdzky boli približne každé 2 týždne s tým, že sme neustále komunikovali v spoločnom chate na Facebooku a uskutočňovali videokonferencie, na ktorým sem sa radili ako postupovať ďalej, ak osobná schôdzka nebola možná. Členovia tímu, ktorí vyvíjali na sebe závislé časti sa stretávali a kontaktovali častejšie ako zbytok tímu.

### 2.2.3. Rozdelenie práce

**Jozef Urbanovský**: vedúci tímu, testovanie, dokumentácia, interpret, vstavané funkcie

**Roman Dobiáš**: lexikálny analyzátor, syntaktický analyzátor, sémantický analyzátor, testovanie

**Adrián Tomašov**: interpret, testovanie, vstavané funkcie, git, integrácia

**Adam Šulc**: syntaktický analyzátor, sémantický analyzátor, precedenčná tabuľka

**Kristián Barna**: tabuľka symbolov, radiaci algoritmus Quick-sort, Knuth-Morris-Pratt

# 3. Implementácia častí

## 3.1. Lexikálna analýza

Lexikálny analyzátor, skener, je prvá časť projektu, ktorá pracuje ako jediná so zdrojovým textom napísaným v jazyku *IFJ16*. Jeho úlohou je transformovať zdrojový text na tokeny. Celý skener je implementovaný ako končený automat (viď. príloha 9.1.), ktorý spracováva vstup znak po znaku a na základe stavu v ktorom skončil, určí typ tokenu, či zahlási chybu. Pokiaľ skener nájde lexikálnu chybu vo vstupnom súbore podľa návratovej hodnoty funkcie, ktorá spracováva lexém vieme zistiť o akú chybu sa jednalo. Náš skener implementuje funkciu *getToken(),* ktorá po volaní syntaktickým analyzátorom uloží práve jeden načítaný token do globálnej premennej *g_lastToken*, ktorá je typu *t_token*. Token obsahuje typ a hodnotu lexému, riadok a pozíciu na riadku, kde sa lexém nachádza. Tokeny sú interne uložené v dvojsmerne viazanom zozname, ktorý umožňuje znížiť réžiu na operácie, nakoľko dáta sú uchované a možno ich využiť aj pri druhom prechode parsera. Vďaka dvojsmerne viazanému zoznamu je možné implementovať funkciu *ungetToken(),* ktorá nám umožnuje riešiť kolíziu ak dva tokeny nezodpovedajú volaniu funkcie.

## 3.2. Syntaktická analýza

Syntaktický analyzátor je jadrom celého projektu a riadi všetky ostatné časti. Pri konštrukcii syntaktickej analýzy pre kontext jazyka založeného na LL gramatike (viď. príloha 9.2.), sme využili metódu rekurzívneho zostupu. Top-down parser simuluje vytvorenie najľavejšej derivácie abstraktného syntaktického stromu. Pokiaľ nie je možné odsimulovať konštrukciu derivačného stromu, dochádza k syntaktickej chybe. Vstupné tokeny od skenera sú spracovávané pomocou pravidiel LL gramatiky. Spracovanie výrazov syntaktickej analýzy je implementované precedenčným bottom-up parserom, ktorý je realizovaný zásobníkom a pre výber pravidla pre redukciu využíva precedenčnú tabuľku (viď. príloha 9.3.). Pri spracovávaní výrazov je využitý prevod do postfixovej notácie (poľskej notácie), kvôli následnému zjednodušeniu ich vyhodnocovania.

### 3.3. Sémantická analýza

Sémantická analýza je implementovaná ako priama súčasť syntaktického parsera. Popri generovaní kódu a kontrolovaní syntaktickej korektnosti, parser spolupracuje s tabuľkou symbolov a overuje, či jednotlivé konštrukcie sú aj typovo kompatibilné, prípadne konvertovateľné. Samotné implicitné konverzie hodnôt premenných sú realizované priamo v operáciach interpretu. Sémantická analýza rozširuje výrazový parser o redukciu dátových typov, podľa povolených konverzií. Výstupom tejto časti je vygenerovaný kód pre interpret. Tabuľka symbolov je pred začiatkom sémantickej analýzy predvyplnená signatúrami vstavaných funkcií a triedy IFJ16. Po skončení kompilácie je v nej každému statickému symbolu, ktorý nebol inicializovaný, prenastavený typ na neinicializovaný. Táto operácia má za úlohu pomôcť detekovať chyby pri práci s neinicializovanou premennou.

### 3.4. Interpret

Interpret pracuje s inštrukčnou páskou, ktorá uchováva trojadresné inštrukcie, uložené v lineárne spájanom zozname. V hlavnom cykle interpretu sa sekvenčne prechádza inštrukčná páska a každá inštrukcia má zodpovedajúcu funkciu. Jednotlivé funkcie pracujú s tabuľkou symbolov, prípadne so zásobníkom, ktorého položky sú relatívne indexované, vzhľadom na vzdialenosť od začiatku rámca pre danú funkciu – base pointer. V inštrukciách pre vyhodnocovanie aritmeticko-logických operácií sa nachádza nadstavba na implicitnú konverziu dátových typov (int → double). Interpretácia končí v momente, keď dosiahne inštrukcie *halt(),* prípadne konca inštrukčnej pásky. Interpret je pripravený na možné optimalizácie.

## 4. Implementácia algoritmov

### 4.1. Knuth-Morris-Pratt algoritmus

Algoritmus zrýchľuje vyhľadávanie podreťazca v reťazci, za využitia informácie o výskyte podreťazcov v hľadanom reťazci, ktoré sa zhodujú s prefixom hľadaného reťazca. Túto informáciu uchováva pomocná tabuľka, ktorá sa zostaví pred hľadaním, postupným porovnávaním podreťazcov hľadaného reťazca s jeho prefixom. V prípade že neexistuje podreťazec o veľkosti väčšej než 1 zhodujúci sa s prefixom, bude tabuľka plná núl. Pri vyhľadávaní sa v prípade nezhody podľa tabuľky nastaví ktorým písmenom z podreťazca sa má pokračovať.

Základný vyhľadávací algoritmus porovnáva podreťazec s reťazcom postupne znak po znaku. Algoritmus sa vyhýba porovnávaniu toho istého znaku v prípade, že už bol porovnaný a nie je možné aby bol súčasťou podreťazca ktorý je prefixom hľadaného reťazca. Samotné vyhľadávanie má zložitosť $O(n)$. Pomocná tabuľka indexov uchováva pozície, od ktorých sa musí začať opäť porovnávať.

### 4.2. Quick-sort algoritmus

Quick-sort je veľmi rýchly a rekurzívne jednoducho implementovateľný radiaci algoritmus založený na princípe „divide and conquer“ s najhoršou možnou časovou zložitosťou $O(n^2)$ a s očakávanou zložitosťou $O(n * \log n)$. Prvým krokom pri implementácii tohto algoritmu je zvolenie si ľubovoľného prvku v poli, pivotu. V našom prípade sa ako pivot vyberie pseudomedián, tj.

$$pivot = \frac{(left\ boundary + right\ boundary)}{2}$$

Algoritmus Quick-sort následne prechádza a preusporiadáva pole tak, aby na jednej strane boli prvky väčšie než pivot, na druhej strane prvky menšie než pivot a pivot samotný bol umiestnený približne v strede medzi týmito časťami.

### 4.3. Tabuľka s rozptýlenými položkami

Na tabuľku pre uchovávanie symbolov sme podľa zadania využili tabuľku s rozptýlenými položkami (TRP). Výhodou TRP je rýchlosť hľadania symbolov v nej uložených. Jej priemerná doba vyhľadávania je $O(1)$ a v najhoršom prípade až $O(n)$. Tabuľka pozostáva z poľa ukazovateľov na jednosmerne viazaný zoznam synoným, kde každá položka zoznamu obsahuje symbol a ukazovateľ na nasledujúcu položku zoznamu. V TRP hľadáme symbol pomocou takzvanej hashovacej funkcie, ktorá na základe hľadaného reťazca vypočíta index do poľa ukazovateľov na zoznam, v ktorom by sa hľadaný symbol mal nachádzať. Následne sa prechádza zoznamom, pokiaľ nie je prvok nájdený, prípadne kým sa nedostaneme na koniec zoznamu.

Naše riešenie zahŕňa využitie jedinej tabuľky symbolov, ktorá uchováva lokálne a statické symboly a zároveň aj konštanty potrebné pre interpret.

## 5. Implementácia rozšírení

### 5.1. SIMPLE

Rozšírenie SIMPLE so sebou prináša úpravu LL gramatiky, pri ktorej dochádza ku kolízii v LL(1) gramatike kvôli nejednoznačnému príkazu ELSE. Táto situácia bola vyriešená manuálnou úpravou LL tabuľky podľa pokynov zadávateľov a v prípade výskytu sa ELSE vzťahuje vždy ku najbližšiemu IF.

# 6. Testovanie

Nakoľko sme si projekt rozdelili na niekoľko nezávislých častí bolo nutné otestovať ich funkčnosť jednotkovými testami. Samotné jednotkové testy boli najväčšou výzvou, nakoľko sme museli simulovať vstupné, prípadne výstupné dáta na overenie správnosti danej časti. Testovanie nášho interpretu po integrácií prebiehalo na základe automatizovaného bashového skriptu, ktorý mal za úlohu overiť správnosť projektu.

Testy boli rozdelené do 3 kategórií, správne, nesprávne a runtime testy. Správne testy obsahovali zdrojový kód, vstup a výstup, ktorý bol následne skontrolovaný, či sa správne interpretoval, vyhodnotil a vrátil správny návratový kód. Nesprávne a runtime testy obsahovali v hlavičke zdrojového kódu typ chyby a jej návratový kód, ktorý očakávame. Následne skript vyextrahoval dáta z hlavičky a po vykonaní daného testu skontroloval jeho výstup a či vrátil očakávaný návratový kód. V prípade chybne vyhodnoteného testu bol oboznámený člen, ktorý mal daný celok na starosť, aby danú chybu opravil. Taktiež sme využili šancu pokusných odovzdaní, ktoré nám pomohli verifikovať náš interpret. Na pokusných odovzdaniach sme obstáli s 93% a 97% respektívne.

- Počet správnych testov: 49
- Počet nesprávnych testov: 85
- Počet runtime testov: 26

# 7. Záver

Návrh a implementácia vlastného interpretu jazyka IFJ16 bol pre nás všetkých výzvou, nakoľko s takým rozsahom školského projektu sme sa doteraz ešte nestretli. Vďaka tomuto projektu sme sa naučili ako používať verziovací systém *Git*, zlepšili si svoje programátorské schopnosti v jazyku C a osvojili si prácu v tíme. Komunikácia v rámci päťčlenného tímu bola občas náročná, ale naučila nás ako to chodí v praxi a čo všetko je nutné robiť v tímovom projekte.

Ako výsledok našej skupinovej práce predkladáme funkčný a nami otestovaný interpret jazyku IFJ16 obsahujúci rozšírenie SIMPLE, zahŕňajúci zjednodušený syntaktický zápis u podmienených výrazov.

# 8. Použitá literatúra

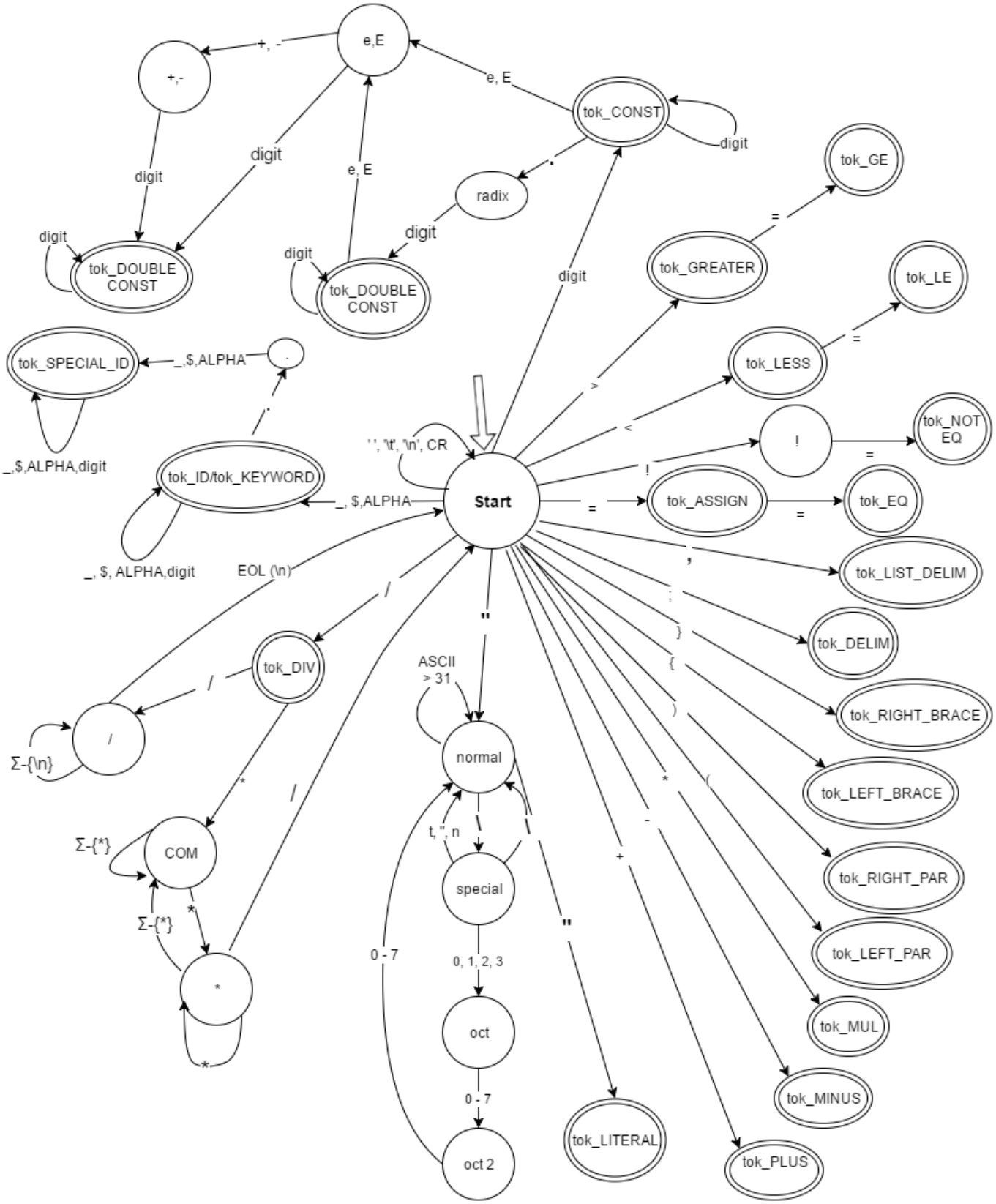
HONZÍK, Jan M., FIT VUT V BRNĚ, *Algoritmy: Studijní opora*, Verzia: 16-D, Brno, 2016.

# 9. Prílohy

## 9.1. Konečný automat lexikálnej analýzy

ALPHA = a-zA-Z

DIGIT = 0-9



## 9.2. LL gramatika

<source-program> → <class-definition-list>
<class-definition-list> → <class-definition> <class-definition-list>
<class-definition-list> → ε
<class-definition> → class simple-identifier { <class-body> }
<class-body> → ε
<class-body> → static <definition> <class-body>
<definition> → <type-specifier> simple-identifier <more-definition>
<definition> → void simple-identifier <function-definition>
<more-definition> → <function-definition>
<more-definition> → ;
<more-definition> → = <expr> ;
<function-definition> → ( <function-parameters-list> ) { <function-body> }
<function-body> → <statement> <function-body>
<function-body> → <local-definition> <function-body>
<function-body> → ε
<local-definition> → <type-specifier> simple-identifier <variable-initialization> ;
<variable-initialization> → = <more-next>
<variable-initialization> → ε
<parameter-definition> → <type-specifier> simple-identifier
<function-parameters-list> → <parameter-definition> <more-function-parameters>
<function-parameters-list> → ε
<more-function-parameters> → , <parameter-definition> <more-function-parameters>
<more-function-parameters> → ε
<argument-definition> → <term>
<function-arguments-list> → <argument-definition> <more-function-arguments>
<function-arguments-list> → ε
<more-function-arguments> → , <argument-definition> <more-function-arguments>
<more-function-arguments> → ε
<statement> → <compound-statement>
<statement> → <assign-statement>
<statement> → <selection-statement>
<statement> → <iteration-statement>
<statement> → <jump-statement>
<block-items-list> → <statement> <block-items-list>
<block-items-list> → ε
<compound-statement> → { <block-items-list> }
<assign-statement> → <identifier> <next> ;
<selection-statement> → if ( <expr> ) <statement> <selection-else>
<selection-else> → else <statement>
<selection-else> → ε
<iteration-statement> → while ( <expr> ) <statement>
<jump-statement> → return <expr> ;
<next> → ( <function-arguments-list> )
<next> → = <more-next>
<more-next> → <identifier> ( <function-arguments-list> )
<more-next> → <expr>
<identifier> → simple-identifier
<identifier> → fully-qualified-identifier
<term> → simple-identifier
<term> → fully-qualified-identifier
<term> → decimal-constant
<term> → floating-point-constant
<type-specifier> → double
<type-specifier> → int
<type-specifier> → String
<expr> → expression

### 9.3. Precedenčná tabuľka

| | == | != | < | > | <= | >= | + | - | * | / | ID | ( | ) | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **==** | > | > | < | < | < | < | < | < | < | < | < | < | > | > |
| **!=** | > | > | < | < | < | < | < | < | < | < | < | < | > | > |
| **<** | > | > | > | > | > | > | < | < | < | < | < | < | > | > |
| **>** | > | > | > | > | > | > | < | < | < | < | < | < | > | > |
| **<=** | > | > | > | > | > | > | < | < | < | < | < | < | > | > |
| **>=** | > | > | > | > | > | > | < | < | < | < | < | < | > | > |
| **+** | > | > | > | > | > | > | < | < | < | < | < | < | > | > |
| **-** | > | > | > | > | > | > | < | < | < | < | < | < | > | > |
| ***** | > | > | > | > | > | > | > | > | > | > | < | < | > | > |
| **/** | > | > | > | > | > | > | > | > | > | > | < | < | > | > |
| **ID** | > | > | > | > | > | > | > | > | > | > | - | - | > | > |
| **(** | < | < | < | < | < | < | < | < | < | < | < | < | = | - |
| **)** | > | > | > | > | > | > | > | > | > | > | - | - | > | > |
| **$** | < | < | < | < | < | < | < | < | < | < | < | < | - | = |