Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Softvérový nástroj pre potreby predmetu Sémantika programovacích jazykov

Diplomová práca

2018 **Iskender Yar-Muhamedov**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Softvérový nástroj pre potreby predmetu Sémantika programovacích jazykov

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	9.2.1. Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	Ing. William Steingartner, PhD.
Konzultant:	Ing. William Steingartner, PhD.

Košice 2018 **Iskender Yar-Muhamedov**

Názov práce:	Softvérový nástroj pre potreby predmetu Sémantika programovacích jazykov
Pracovisko:	Katedra počítačov a informatiky, Technická univerzita v Košiciach
Autor:	Iskender Yar-Muhamedov
Školiteľ:	Ing. William Steingartner, PhD.
Konzultant:	Ing. William Steingartner, PhD.
Dátum:	27. 4. 2018
Kľúčové slová:	Sémantika programovacích jazykov, programovací jazyk $\mathscr{J}ane$, strom abstraktnej syntaxe, postfixná notácia, vyhodnocovač výrazov, učebná pomôcka.

Abstrakt: Diplomová práca sa zaoberá navrhnutím a vytvorením aplikácie, ktorá dokáže spracovávať matematické výrazy, ich syntaktický rozbor, výstup vo forme postfixnej notácie a taktiež vo forme stromu abstraktnej syntaxe (AST), identifikáciu nesprávne zadaných reťazcov (implementáciou zotavenia) a interaktívny zásah používateľa pri vkladaní údajov s následným vyhodnotením výsledkov. Analytická časť práce pozostáva z formálnej definície jednoduchého programovacieho jazyka $\mathscr{J}ane$, lexikálnej analýzy, syntaktickej analýzy, gramatiky jazyka, postfixnej notácie, algoritmu shunting-yard a prístupov vyhodnotenia. Syntetická časť práce je zameraná na návrh a implementáciu aplikácie. Vďaka testovaniu koncových používateľov je potvrdená aj korektnosť implementácie. Aplikácia bude použitá ako učebná pomôcka pre študentov predmetu Sémantika programovacích jazykov.

Thesis title:	Software tool for course Semantics of programming languages
Department:	Department of Computers and Informatics, Technical University of Košice
Author:	Iskender Yar-Muhamedov
Supervisor:	Ing. William Steingartner, PhD.
Tutor:	Ing. William Steingartner, PhD.
Date:	27. 4. 2018
Keywords:	Semantics of programming languages, $\mathscr{J}ane$ programming language, abstract syntax tree, postfix notation, expression evaluator, learning tool.

Abstract: Thesis deals with proposing and creating application that can process mathematical expressions, their syntactic analysis, output in the form of a postfix notation and abstract syntax tree (AST), to identify incorrectly entered expressions (implementation of error recovery) and interactive user intervention during entering data with subsequent evaluation of the results. The analytical part of the thesis consists of a formal definition of the $\mathscr{J}ane$ simple programming language, lexical analysis, syntactic analysis, language grammar, postfix notation, shunting-yard algorithm and evaluation approaches. The synthetic part of the thesis is focused on the design and implementation of the application. Correctness of the implementation is even confirmed through the end-user testing. The application should be used as a learning tool for the student of the course Semantics of programming languages.

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra počítačov a informatiky

ZADANIE DIPLOMOVEJ PRÁCE

Študijný odbor: **Informatika**
Študijný program: **Informatika**

Názov práce:

Softvérový nástroj pre potreby predmetu Sémantika programovacích jazykov

Software tool for course Semantics of programming languages

Študent:	**Bc. Iskender Yar-Muhamedov**
Školiteľ:	**Ing. William Steingartner, PhD.**
Školiace pracovisko:	**Katedra počítačov a informatiky**
Konzultant práce:	**Ing. William Steingartner, PhD.**
Pracovisko konzultanta:	**Katedra počítačov a informatiky**

Pokyny na vypracovanie diplomovej práce:

1. Preštudovať teóriu sémantiky programovacích jazykov pre aritmetické a logické výrazy.
2. Analyzovať vlastnosti syntaktických objektov a ich sémantiky a identifikovať kritéria potrebné pre implementáciu analýzy a vyhodnocovania výrazov pre potreby predmetu Sémantika programovacích jazykov.
3. Formulovať kritéria pre úspešnú implementáciu zvolených algoritmov.
4. Implementovať analýzu, grafickú reprezentáciu a vyhodnocovanie výrazov v interaktívnom prostredí.
5. Vypracovať dokumentáciu podľa pokynov vedúceho práce.

Jazyk, v ktorom sa práca vypracuje: slovenský
Termín pre odovzdanie práce: 27.04.2018
Dátum zadania diplomovej práce: 31.10.2017

.....
doc. Ing. Jaroslav Porubän, PhD.
vedúci garantujúceho pracoviska

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY -12-

.....
prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som záverečnú prácu vypracoval(a) samostatne s použitím uvedenej odbornej literatúry.

Košice, 27.4.2018 .....

Vlastnoručný podpis

Poďakovanie

Chcel by som poďakovať vedúcemu a konzultantovi mojej diplomovej práce Ing. Williamovi Steingartnerovi, PhD. za jeho odborné rady a pripomienky k mojej práci, za jeho ústretovosť, ochotu pomôcť pri riešení problémov vzniknutých počas tvorby práce a profesionálny prístup.

Dalej by som sa chcel podakovat svojim blízkym a svojej rodine za ich podporu a pomoc.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Motivácia

Sémantika programovacích jazykov zohráva dôležitú úlohu pri hľadaní významu programov, ako aj pri potvrdení ich korektného priebehu. Práve tejto oblasti sa venuje táto diplomová práca. Konkrétne sa zaoberáme spracovaním vstupného aritmetického alebo logického výrazu, jeho vyhodnotením a zobrazením vo forme stromu abstraktnej syntaxe, a to nielen na teoretickej úrovni, ale aj praktickej, a teda vývoja aplikácie, ktorá dokáže tieto úlohy realizovať.

Pre navrhnutie a vytvorenie aplikácie je najprv nutné zozbierať bázu potrebných vedomostí. Preto sa hneď v prvej časti venujeme formálnej definícii jednoduchého programovacieho jazyka Jane, následne sme zadefinovali gramatiku jazyka. Ďalšia fáza bola lexikálna a syntaktická analýza. Ďalej sme sa venovali postfixnej notácii. Jednou z najpodstatnejších vedomostí pre nás bol algoritmus shunting-yard a prístupy vyhodnotenia vstupného výrazu. Akonáhle sme zozbierali všetky potrebné vedomosti, začali sme vývoj aplikácie, ktorá tvorí syntetickú časť tejto práce. Tu sme zužitkovali doterajšie vedomosti z oblasti sémantiky programovacích jazykov ako aj objektového programovania či samotného softvérového inžinierstva.

Výsledkom syntaktickej časti tejto diplomovej práce je softvérový nástroj pre potreby predmetu Sémantika programovacích jazykov, ktorý dokáže zanalyzovať vstupný aritmetický alebo logický výraz na syntaktickú správnosť, realizuje zotavenie pri výskyte chyby, poskytne jeho postfixnú formu zápisu a umožní interaktívny zásah používateľa pri vkladaní údajov s následným vyhodnotením výsledkov. Aplikácia poskytuje aj grafickú reprezentáciu vstupného výrazu vo forme stromu abstraktnej syntaxe, ktorú na požiadavku používateľa vie uložiť do PDF súboru na základe vektorovej grafiky. Softvérový nástroj má slúžiť ako učebná pomôcka pre študentov predmetu Sémantika programovacích jazykov, kde sa študenti učia rôzne sémantické princípy a ich aplikácie v praxi. Jednou z praktic-

kých znalostí tohto predmetu je vyhodnotenie aritmetických a logických výrazov a zobrazenie stromu abstraktnej syntaxe, a práve táto aplikácia študentom má pomôcť lepšie pochopiť samotný materiál štúdia a tiež má slúžiť ako kontrola písomných prác.

1 Formulácia úlohy

Diplomová práca je zameraná na vývoj grafickej, desktopovej aplikácie, ktorá spracováva matematické výrazy, ich syntaktický rozbor, výstup vo forme stromu abstraktnej syntaxe, identifikáciu nesprávne zadaných reťazcov (implementáciou zotavenia) a interaktívny zásah používateľa pri vkladaní údajov s následným vyhodnotením výsledkov. Výsledný softvér bude slúžiť ako učebná pomôcka pre predmet Sémantika programovacích jazykov.

Softvér bude mať nasledujúce funkcie:

- spracovanie aritmetických výrazov,
- spracovanie boolovských výrazov,
- vyhodnocovanie výrazov,
- syntaktická kontrola,
- zotavenie pri výskyte syntaktických chýb,
- interaktívne zmeny hodnôt premenných vo výrazoch,
- kreslenie stromu abstraktnej syntaxe.

Zameriavame sa na tvorbu nastroja, ktorý umožní uvedené činnosti realizovať. Pre tieto účely potrebujeme implementovať nasledujúce metódy:

- vstup výrazu,
- lexikálna analýza,
- syntaktická analýza,
- zmena hodnôt premenných,

- spracovanie výrazu,
- zotavenie pri výskyte chyby,
- vyhodnocovanie výrazu,
- kreslenie stromu abstraktnej syntaxe.

2 Použité typografické konvencie

Tu uvedieme typografické konvencie, ktoré sú platné pre túto diplomovú prácu.

Tenká kurzíva	používa sa na označenie aritmetického alebo boolovského výrazu a na zápis sémantických funkcií, ktoré zapisujeme písaným písmom (napr. $\mathscr{E}$, $\mathscr{B}$);
Kurzíva	používa sa na označenie názvu metódy;
`Neproporcionálne písmo`	používa sa na označenie názvu a skutočnej hodnoty premennej a na zápis syntaktických prvkov vyjadrujúcich hodnoty (napr. `true`, `false`);
Tučné bezpätkové písmo	používa sa na označenie prvkov grafického používateľského rozhrania;
Tučná antikva	používa sa na označenie údajového typu premennej alebo objektu, na zápis sémantických oblastí, ktoré zapisujeme s veľkým začiatočným písmenom, na označenie prvkov množiny terminálnych symbolov a na označenie celočiseľnych (napr. **0**, **10**, **-5**) a boolovských (napr. **tt**, **ff**) konštánt;
Tučná kurzíva	používa sa na označenie názvu triedy a balíka.

3 Analytická časť práce

Analytická časť práce sa zaoberá formálnou definíciou jednoduchého programovacieho jazyka $\mathscr{J}ane$, lexikálnej a syntaktickej analýzy vstupného výrazu. Taktiež v tejto časti diplomovej práce sa venujeme definícii gramatiky jazyka a postfixnej notácii, ktorá odstráni zátvorky vo výraze. Na implementáciu metódy prepisu infixného tvaru výrazu na postfixný použijeme algoritmus shunting-yard. Takisto v analytickej časti práce sa zaoberáme aj prístupmi vyhodnotenia vstupného výrazu.

3.1 Jednoduchý programovací jazyk $\mathscr{J}ane$

Jednoduchý programovací jazyk $\mathscr{J}ane$ je vzorovým programovacím jazykom, ktorý patrí do paradigmy imperatívneho (procedurálneho) programovania. Pre lepšiu názornosť pri formulovaní princípov sémantických metód predpokladáme, že pracujeme len s celočíselnými hodnotami, ktoré môžu nadobúdať premenné jazyka a s pravdivostnými hodnotami, ktoré sa používajú v boolovských výrazoch. Teda jazyk má preddefinované dva typy a typovanie v $\mathscr{J}ane$ je implicitné [1].

Vo vzorovom programovacom jazyku $\mathscr{J}ane$ máme nasledujúce produkčné pravidlá pre syntaktické oblasti aritmetických a boolovských výrazov [1]:

$$e ::= n \mid x \mid e + e \mid e - e \mid e * e \mid (e)\,, \tag{3.1}$$

$$b ::= \texttt{true} \mid \texttt{false} \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b \mid (b)\,, \tag{3.2}$$

kde

$n \in$ **Num** (reťazce číslic),
$x \in$ **Var** (premenné),
$e \in$ **Expr** (aritmetické výrazy),
$b \in$ **Bexp** (boolovské výrazy),

pričom

- n je atomická entita, pre abstraktnú syntax výrazov nie je dôležitý tvar čísel, preto **Num** nemá vnútornú štruktúru;
- x je atomická entita, pre abstraktnú syntax výrazov nie je dôležitý tvar identifikátorov, teda **Var** nemá vnútornú štruktúru;
- e je prvok syntaktickej oblasti **Expr**, ktorá má z hľadiska abstraktnej syntaxe vnútornú štruktúru;
- b je prvok syntaktickej oblasti **Bexp**, ktorá má taktiež z hľadiska abstraktnej syntaxe vnútornú štruktúru.

Zátvorky "(" a ")" sú pomocné symboly. Zátvorkované aritmetické a boolovské výrazy slúžia pre explicitný zápis priorít aritmetických, respektíve logických operácií [1].

3.2 Sémantika aritmetických výrazov

Aritmetický výraz v Jane slúži pre výpočet celočiselnej hodnoty. Syntakticky správny aritmetický výraz musí mať niektorý z tvarov produkčného pravidla [1]:

$$e ::= n \mid x \mid e + e \mid e - e \mid e * e \mid (e)\,, \tag{3.3}$$

pričom

- n je metapremená označujúca reťazec číslic, teda celočíselnú konštantu;
- x je premenná programu;
- $e \,\square\, e$ označuje niektorú operáciu na aritmetických výrazoch: súčet, rozdiel alebo súčin;
- (e) označuje zátvorkovaný aritmetický výraz.

Produkčné pravidlo pre sémantickú oblasť aritmetických výrazov môžeme rozšíriť nad rámec toho, ktoré sa využívajú v predmete Sémantika programovacích jazykov. Pridáme do produkčného pravidla binárnu aritmetickú operáciu celočíselného delenia, unárny operátor "$+$" a "$-$":

$$e ::= n \mid x \mid e + e \mid e - e \mid e * e \mid e/e \mid +e \mid -e \mid (e)\,. \tag{3.4}$$

Sémantické oblasti aritmetických výrazov jednoduchého programovacieho jazyka $\mathscr{J}ane$ sú [1]:

- **Z** je množina celých čísel, prvky ktorej sú hodnoty aritmetických výrazov;
- **State** je množina stavov, pričom stav s každej premennej aritmetického výrazu priradí jej aktuálnu hodnotu [1, 2, 3]:

$$\begin{aligned} s &\in \mathbf{State}, \\ \mathbf{State} &= \mathbf{Var} \to \mathbf{Z}. \end{aligned} \tag{3.5}$$

Aby sme definovali význam (hodnotu) aritmetických výrazov, musíme definovať sémantickú funkciu, ktorá každému aritmetickému výrazu $\mathscr{J}ane$ jednoznačne priradí jeho celočíselnú hodnotu. Aritmetické výrazy však obsahujú premenné, ktorých hodnota závisí od aktuálneho stavu. Preto sémantickú funkciu pre aritmetické výrazy špecifikujeme curryovským zápisom [1, 2, 3]:

$$\mathscr{E} : \mathbf{Expr} \to \mathbf{State} \to \mathbf{Z}.$$

$\mathscr{E}$ je funkcia dvoch argumentov. Po dosadení aritmetického výrazu za prvý argument dostaneme funkciu jedného argumentu [1, 2, 3]:

$$\mathscr{E}[\![x + y - 1]\!] : \mathbf{State} \to \mathbf{Z}.$$

Po aplikovaní tejto funkcie na druhý argument, na stav s, dostaneme hodnotu, význam aritmetického výrazu v stave s [1, 2, 3]:

$$\mathscr{E}[\![x + y - 1]\!] : s \in \mathbf{Z}.$$

Ďalej potrebujeme definovať sémantickú funkciu $\mathscr{E}$ pre jednotlivé tvary aritmetických výrazov pomocou sémantických rovností [1]:

$$\begin{aligned}
\mathscr{E}[\![n]\!]s &= \mathscr{N}[\![n]\!], \\
\mathscr{E}[\![x]\!]s &= s\ \ x, \\
\mathscr{E}[\![e_1 + e_2]\!]s &= \mathscr{E}[\![e_1]\!]s \oplus \mathscr{E}[\![e_2]\!]s, \\
\mathscr{E}[\![e_1 - e_2]\!]s &= \mathscr{E}[\![e_1]\!]s \ominus \mathscr{E}[\![e_2]\!]s, \\
\mathscr{E}[\![e_1 * e_2]\!]s &= \mathscr{E}[\![e_1]\!]s \otimes \mathscr{E}[\![e_2]\!]s, \\
\mathscr{E}[\![e_1/e_2]\!]s &= \mathscr{E}[\![e_1]\!]s \oslash \mathscr{E}[\![e_2]\!]s, \\
\mathscr{E}[\![+e]\!]s &= \mathscr{E}[\![e]\!]s, \\
\mathscr{E}[\![-e]\!]s &= \mathscr{N}[\![0]\!] \ominus\ \mathscr{E}[\![e]\!]s, \\
\mathscr{E}[\![(e)]\!]s &= (\mathscr{E}[\![e]\!]s).
\end{aligned} \tag{3.6}$$

3.3 Sémantika boolovských výrazov

Boolovský výraz v $\mathscr{J}ane$ slúži pre zistenie pravdivostnej hodnoty podmienky v podmieňovacom príkaze alebo v príkaze cyklu. Syntakticky správny boolovský výraz musí mať niektorý z tvarov produkčného pravidla [1]:

$$b ::= \texttt{true} \mid \texttt{false} \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b \mid (b)\,, \tag{3.7}$$

pričom

- `true`, `false` sú pravdivostné konštanty;
- $e = e$ je rovnosť aritmetických výrazov;
- $e \leq e$ je neostrá nerovnosť aritmetických výrazov;
- $\neg b$ je negácia boolovského výrazu;
- $b \wedge b$ je konjunkcia boolovských výrazov;
- (b) označuje zátvorkovaný boolovský výraz.

Produkčné pravidlo pre sémantickú oblasť boolovských výrazov môžeme rozšíriť nad rámec toho, ktoré sa využívajú v predmete Sémantika programovacích jazykov. Pridáme do produkčného pravidla boolovskú operáciu disjunkcie:

$$b ::= \texttt{true} \mid \texttt{false} \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b \mid b \vee b \mid (b)\,. \tag{3.8}$$

Význam boolovských výrazov je pravdivostná hodnota. Vytvoríme sémantickú oblasť [1]:

$$\mathbf{B} = \{\mathbf{tt}, \mathbf{ff}\},$$

pričom

- **tt** bude označovať pravdivosť;
- **ff** bude označovať nepravdivosť.

Definujeme sémantickú funkciu $\mathscr{B}$ pre boolovské výrazy curryovským zápisom [1, 2, 3]:

$$\mathscr{B} : \mathbf{Bexp} \to \mathbf{State} \to \mathbf{B}$$

Uvažujme o stavoch, lebo boolovské výrazy môžu obsahovať aritmetické výrazy, ktorých hodnota môže závisieť od aktuálnych hodnôt premenných. Sémantické rovnosti pre jednotlivé alternatívy produkčného pravidla definujú význam boolovského výrazu v stave s [1]:

$$\begin{aligned}
\mathscr{B}[\![\texttt{true}]\!]s &= \mathbf{tt},\\
\mathscr{B}[\![\texttt{false}]\!]s &= \mathbf{ff},\\
\mathscr{B}[\![e_1 = e_2]\!]s &= \begin{cases} \mathbf{tt}, \text{ ak } \mathscr{E}[\![e_1]\!]s = \mathscr{E}[\![e_2]\!]s, \\ \mathbf{ff}, \text{ ak } \mathscr{E}[\![e_1]\!]s \neq \mathscr{E}[\![e_2]\!]s, \end{cases}\\
\mathscr{B}[\![e_1 \leq e_2]\!]s &= \begin{cases} \mathbf{tt}, \text{ ak } \mathscr{E}[\![e_1]\!]s \leq \mathscr{E}[\![e_2]\!]s, \\ \mathbf{ff}, \text{ ak } \mathscr{E}[\![e_1]\!]s > \mathscr{E}[\![e_2]\!]s, \end{cases}\\
\mathscr{B}[\![\neg b]\!]s &= \begin{cases} \mathbf{tt}, \text{ ak } \mathscr{B}[\![b]\!]s = \mathbf{ff}, \\ \mathbf{ff}, \text{ ak } \mathscr{B}[\![b]\!]s = \mathbf{tt}, \end{cases}\\
\mathscr{B}[\![b_1 \wedge b_2]\!]s &= \begin{cases} \mathbf{tt}, \text{ ak } \mathscr{B}[\![b_1]\!]s = \mathbf{tt} \text{ a } \mathscr{B}[\![b_2]\!]s = \mathbf{tt}, \\ \mathbf{ff}, \text{ ak } \mathscr{B}[\![b_1]\!]s = \mathbf{ff} \text{ alebo } \mathscr{B}[\![b_2]\!]s = \mathbf{ff}, \end{cases}\\
\mathscr{B}[\![b_1 \vee b_2]\!]s &= \begin{cases} \mathbf{tt}, \text{ ak } \mathscr{B}[\![b_1]\!]s = \mathbf{tt} \text{ alebo } \mathscr{B}[\![b_2]\!]s = \mathbf{tt}, \\ \mathbf{ff}, \text{ ak } \mathscr{B}[\![b_1]\!]s = \mathbf{ff} \text{ a } \mathscr{B}[\![b_2]\!]s = \mathbf{ff}, \end{cases}\\
\mathscr{B}[\![(b)]\!]s &= \begin{cases} \mathbf{tt}, \text{ ak } \mathscr{B}[\![b]\!]s = \mathbf{tt}, \\ \mathbf{ff}, \text{ ak } \mathscr{B}[\![b]\!]s = \mathbf{ff}. \end{cases}
\end{aligned} \tag{3.9}$$

3.4 Prehľad existujúcich riešení

V dnešnej dobe je celosvetovým trendom modernizácia vzdelávania vo forme digitalizácie. Dôsledkom tohto trendu vzniká mnoho projektov, ktoré napomáhajú v modernizovaní vzdelávacieho procesu na všetkých úrovniach vzdelávania [4].

Podstata tejto diplomovej práce je v implementácii softvérového nástroja pre potreby predmetu Sémantika programovacích jazykov, ktorý dokáže analyzovať vstupný aritmeticky alebo boolovský výraz na syntaktickú správnosť, realizuje zotavenie pri výskyte chyby, poskytne jeho postfixnú formu zápisu a umožní interaktívny zásah používateľa pri vkladaní hodnôt premenných s následným vyhodnotením výsledkov. Taktiež aplikácia poskytuje grafickú reprezentáciu vstupného výrazu vo forme stromu abstraktnej syntaxe, ktorú na požiadavku používateľa vie uložiť do PDF súboru na základe vektorovej grafiky. Softvérový nástroj má slúžiť ako učebná pomôcka pre študentov predmetu Sémantika programovacích jazykov, kde sa študenti učia rôzne sémantické princípy a ich aplikácie v praxi. Táto aplikácia má študentom pomôcť lepšie pochopiť samotný materiál štúdia a tiež má slúžiť ako kontrola písomných prác.

V súčasnosti existuje niekoľko podobných riešení a pravé v tejto podkapitole zdôrazníme, v čom sa naše riešenie bude líšiť, aké sú jeho výhody a nevýhody.

Chris Johnson [5] v 2008 roku pod licenciou GNU zverejnil Java applet, ktorý demonštruje správanie shunting-yard algoritmu.

Výhody aplikácie:

- nie je závislý od konkrétnej platformy;
- názornosť správania shunting-yard algoritmu;
- názornosť vyhodnotenia vstupného výrazu.

Nevýhody aplikácie:

- absencia možnosti pracovania s boolovskými výrazmi (práca iba s aritmetickými výrazmi);
- absencia možnosti pracovania s premennými;
- absencia unárnych operátorov;

- absencia grafickej reprezentácie vstupného výrazu vo forme stromu abstraktnej syntaxe.

Ota Pavelek sa zaoberal v svojej bakalárskej práci implementáciou hlasom ovládanej kalkulačky [6].

Výhody aplikácie:

- možnosť prací pre ľudí s obmedzenými schopnosťami;
- možnosť práce s goniometrickými funkciami.

Nevýhody aplikácie:

- je závislá od konkrétnej platformy;
- absencia možnosti pracovania s boolovskými výrazmi (práca iba s aritmetickými výrazmi);
- absencia možnosti pracovania s premennými;
- názornosť vyhodnotenia vstupného výrazu;
- absencia grafickej reprezentácii vstupného výrazu vo forme stromu abstraktnej syntaxe.

Žiadna z uvedených aplikácií neumožňuje prácu s boolovskými výrazmi, premennými a neposkytuje reprezentáciu vstupného výrazu vo forme stromu abstraktnej syntaxe. Tieto body sú dôležité pre potreby predmetu Sémantika programovacích jazykov, ktorý zohráva dôležitú úlohu pri hľadaní významu programov, ako aj pri potvrdení ich korektného priebehu.

Existuje softvérový nástroj Wolfram Mathematica [7], ktorý poskytuje tieto funkcie, ale je spoplatnený.

Práve táto diplomová práca sa zaoberá riešením uvedenej problematiky a implementáciou softvérového nástroja pre potreby predmetu Sémantika programovacích jazykov.

3.5 Etapy analýzy

Pri práci akéhokoľvek kompilátora alebo interpretátora je prítomná fáza analýzy zdrojového programu zvyčajne reprezentovaného vo forme textu.

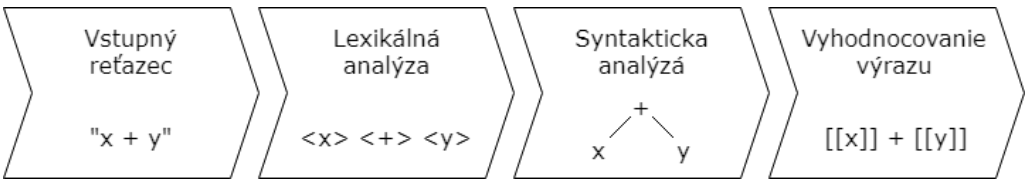
Úlohou prekladača je urobiť preklad jednej podoby reprezentácie programu do ekvivalentnej druhej formy reprezentácie. Kompilátor – prípad prekladača, ktorý robí preklad programu do cieľovej reprezentácie, ktorú môže použiť počítač – strojový kód alebo bajt-kód. Úlohou interpretátora je vykonať vstupný program bez konverzie do cieľovej reprezentácie [8].

Vstupný program uvedený v predchádzajúcom odseku by sa nemal chápať úzko len ako kód v programovacom jazyku, v konkrétnych prípadoch to môžu byť napríklad údaje uvedené v textovom formáte, XML, JSON atď. [8].

V našom prípade bude vstupná forma aritmetický alebo boolovský výraz, reprezentovaný na vstupe ako reťazec. Proces analýzy vstupného programu pozostáva z dvoch etáp: lexikálnej analýzy a syntaktickej analýzy. Lexikálna analýza je fáza prekladu, pri ktorej sú lexikálne jednotky atomizované a podávané na vstup syntaktického analyzátora vo forme symbolov. Lexikálny analyzátor je programový modul realizujúci lexikálnu analýzu, realizácia ktorého je reprezentovaná metódou rozpoznávajúcou lexikálne jednotky [9].

V etape lexikálnej analýzy sa vstupný program skladajúci sa zo sekvencie symbolov prekladá do sekvencie lexém (tokenov). Základnou úlohou syntaktickej analýzy je kontrola syntaxe (gramatickej správnosti vstupného kódu) – vstupného reťazca terminálnych symbolov. V tejto fáze sa sekvencia lexém porovnáva s určitým súborom formálnych pravidiel pre stavbu viet v prirodzenom či formálnom jazyku [8].

Vstupným reťazcom je v našom prípade aritmetický alebo boolovský výraz zapísaný vo vzorovom programovacom jazyku Jane, ktorého jednotky sú spracované do formy symbolov.



Obr. 3.1: Etapy spracovania

Ďalej na implementáciu potrebujeme definovať gramatiku jazyka.

3.6 Gramatika jazyka

Preskúmame zátvorkový tvar zápisu aritmetických a boolovských výrazov, ktorých syntax je definovaná jeho gramatikou G, ktorá je vo všeobecnosti usporiadanou štvoricou [9, 10, 11, 12]:

$$G = (V_N, V_T, P, S),$$

kde

V_N je množina neterminálnych symbolov,
V_T je množina terminálnych symbolov,
P je množina pravidiel v tvare $\alpha \rightarrow \beta$,
S je začiatočný symbol $S \in V_N$,
α, β sú reťazce ľubovoľných symbolov.

Nech pravidlá gramatiky G sú nasledovné (prvky množiny terminálnych symbolov budú v úvodzovkách alebo apostrofoch).

Pre aritmetické výrazy:

$$
\begin{array}{ll}
S & \rightarrow Factorized\_lexical\_form \\
Factorized\_lexical\_form & \rightarrow NameLF \text{ “=” } Expr \\
NameLF & \rightarrow Identificator \\
 & \\
Expr & \rightarrow Term \ \{ \text{ “+” } \mid \text{ “−” } Term\} \\
Term & \rightarrow Factor \ \{ \text{ “∗” } Factor \ \} \\
Factor & \rightarrow Factor \mid Number \mid Var \mid \text{“(” } Expr \text{“)”} \\
 & \\
Identificator & \rightarrow Char \ \{ \ Char \mid Number \ \} \\
Char & \rightarrow \ \{ \text{ ‘\_’ } \mid \text{ ‘}\mathbf{A}\text{’ } \mid \text{ ‘}\mathbf{B}\text{’ } \mid ... \mid \text{ ‘}\mathbf{Z}\text{’ } \mid \text{ ‘}\mathbf{a}\text{’ } \mid \text{ ‘}\mathbf{b}\text{’ } \mid ... \mid \text{ ‘}\mathbf{z}\text{’ } \} \\
Number & \rightarrow Number \ \{ \text{ ‘}\mathbf{0}\text{’ } \mid \text{ ‘}\mathbf{1}\text{’ } \mid \text{ ‘}\mathbf{2}\text{’ } \mid ... \mid \text{ ‘}\mathbf{9}\text{’ } \} \\
Var & \rightarrow Char \ \{ \ Char \mid Number \ \}
\end{array}
$$

Pre boolovské výrazy:

$$
\begin{array}{ll}
S & \rightarrow Factorized\_lexical\_form \\
Factorized\_lexical\_form & \rightarrow NameLF \text{ “}=\text{” } Bexp \\
NameLF & \rightarrow Identificator \\
 & \\
Bexp & \rightarrow Term \,[\text{ “}=\text{” } \mid \text{ “}\leq\text{”} Term \,] \\
Term & \rightarrow Factor \,\{\text{ “}\wedge\text{”} Factor \,\} \\
Factor & \rightarrow \text{“}\neg\text{”} \mid Boolean\_constant \mid \text{“(”} Bexp\text{“)”} \\
 & \\
Identificator & \rightarrow Char \,\{ Char \mid Number \,\} \\
Char & \rightarrow \{\text{ ‘\_’} \mid \text{‘}\mathbf{A}\text{’} \mid \text{‘}\mathbf{B}\text{’} \mid ... \mid \text{‘}\mathbf{Z}\text{’} \mid \text{‘}\mathbf{a}\text{’} \mid \text{‘}\mathbf{b}\text{’} \mid ... \mid \text{‘}\mathbf{z}\text{’} \,\} \\
Number & \rightarrow Number \,\{\text{ ‘}\mathbf{0}\text{’} \mid \text{‘}\mathbf{1}\text{’} \mid \text{‘}\mathbf{2}\text{’} \mid ... \mid \text{‘}\mathbf{9}\text{’} \,\} \\
Boolean\_constant & \rightarrow \text{“}\texttt{true}\text{” } \mid \text{“}\texttt{false}\text{”}
\end{array}
$$

Rozšírené produkčné pravidlo vzorového programovacieho jazyka *Jane* pre syntaktickú oblasť aritmetických výrazov 3.4 zavedieme do gramatiky následne:

$$
\begin{array}{ll}
S & \rightarrow Factorized\_lexical\_form \\
Factorized\_lexical\_form & \rightarrow NameLF \text{ “}=\text{” } Expr \\
NameLF & \rightarrow Identificator \\
 & \\
Expr & \rightarrow Term \,\{\text{ “}+\text{” } \mid \text{ “}-\text{” } \,\} \\
Term & \rightarrow Factor \,\{\text{ “}*\text{” } \mid \text{“}/\text{”} Factor \,\} \\
Factor & \rightarrow Unary\_operator \mid Factor \mid Number \mid \\
 & \quad Var \mid \text{“(”} Expr\text{“)”} \\
 & \\
Identificator & \rightarrow Char \,\{ Char \mid Number \,\} \\
Char & \rightarrow \{\text{ ‘\_’} \mid \text{‘}\mathbf{A}\text{’} \mid \text{‘}\mathbf{B}\text{’} \mid ... \mid \text{‘}\mathbf{Z}\text{’} \mid \text{‘}\mathbf{a}\text{’} \mid \text{‘}\mathbf{b}\text{’} \mid ... \mid \text{‘}\mathbf{z}\text{’} \,\} \\
Number & \rightarrow Number \,\{\text{ ‘}\mathbf{0}\text{’} \mid \text{‘}\mathbf{1}\text{’} \mid \text{‘}\mathbf{2}\text{’} \mid ... \mid \text{‘}\mathbf{9}\text{’} \,\} \\
Var & \rightarrow Char \,\{ Char \mid Number \,\} \\
Unary\_operator & \rightarrow Unary\_operator \,\{\text{ “}+\text{” } \mid \text{ “}-\text{” } \,\}
\end{array}
$$

Rozšírené produkčné pravidlo vzorového programovacieho jazyka Jane pre syntaktickú oblasť boolovských výrazov 3.8 zavedieme do gramatiky následne:

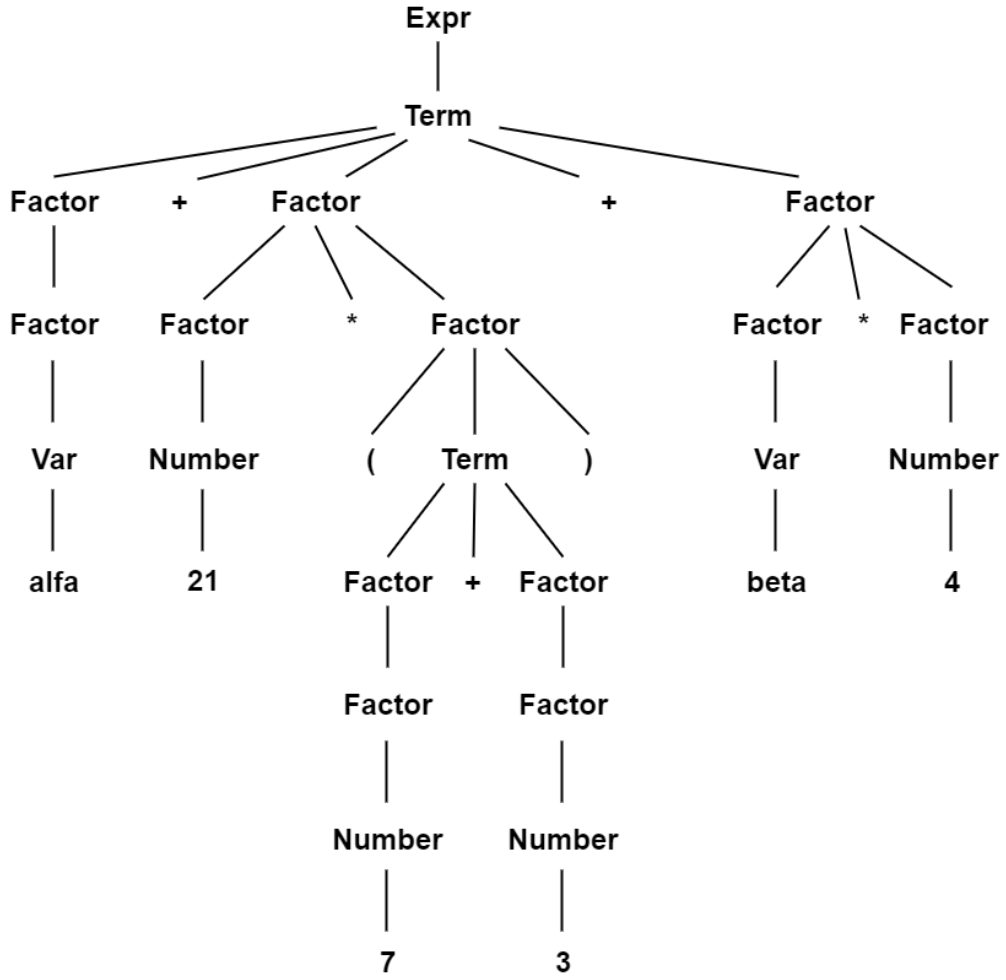
$$
\begin{aligned}
&S && \rightarrow Factorized\_lexical\_form \\
&Factorized\_lexical\_form && \rightarrow NameLF \text{ “=” } Bexp \\
&NameLF && \rightarrow Identificator \\
\\
&Bexp && \rightarrow Term \,[\text{ “=” } \mid \text{ “}\leq\text{”} Term\,] \\
&Term && \rightarrow Factor \,\{\text{ “}\wedge\text{” } \mid \text{ “}\vee\text{”} Factor\,\} \\
&Factor && \rightarrow \text{“}\neg\text{” } \mid Boolean\_constant \mid \text{“(”} Bexp\text{“)”} \\
\\
&Identificator && \rightarrow Char\,\{\,Char \mid Number\,\} \\
&Char && \rightarrow \{\text{ ‘\_’ } \mid \text{ ‘}\mathbf{A}\text{’ } \mid \text{ ‘}\mathbf{B}\text{’ } \mid ... \mid \text{ ‘}\mathbf{Z}\text{’ } \mid \text{ ‘}\mathbf{a}\text{’ } \mid \text{ ‘}\mathbf{b}\text{’ } \mid ... \mid \text{ ‘}\mathbf{z}\text{’ }\} \\
&Number && \rightarrow Number\,\{\text{ ‘}\mathbf{0}\text{’ } \mid \text{ ‘}\mathbf{1}\text{’ } \mid \text{ ‘}\mathbf{2}\text{’ } \mid ... \mid \text{ ‘}\mathbf{9}\text{’ }\} \\
&Boolean\_constant && \rightarrow \text{“}\texttt{true}\text{” } \mid \text{ “}\texttt{false}\text{”}
\end{aligned}
$$

3.7 Spracovanie výrazov na základe gramatiky

Analýza začína pôvodným pravidlom definovaným gramatikou jazyka, v našom prípade z pravidla “$Expr$”. Pre klesajúcu analýzu uplatňovania pravidiel môžu byť zastúpené v podobe volania funkcie, úlohou ktorej je analyzovať časť reťazca, ktorý zodpovedá vzoru zadanému na pravej strane pravidla. Každé pravidlo aplikujeme (vykonáva sa volanie funkcie) vo vstupnej časti reťazca ku časti reťazca, ktorá ešte nebola analyzovaná. V súlade s definovanou gramatikou pravidlá môžu byť aplikované rekurzívne. Na konci analýzy (návrat z funkcie, zodpovedajúci počiatočnému pravidlu) musí byť analyzovaný celý reťazec [8, 11, 12].

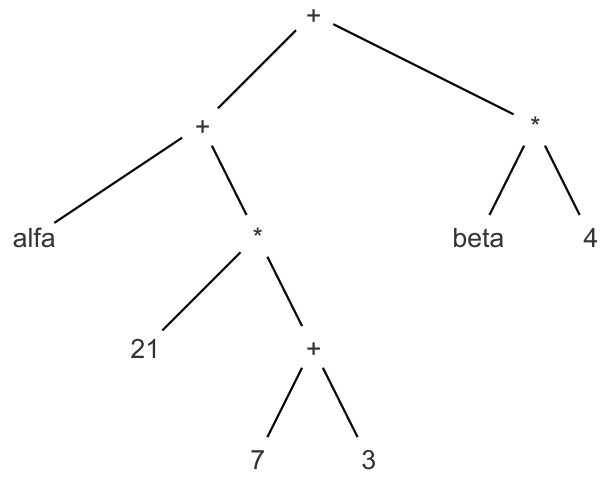
Ak časť reťazca nebude spracovaná, znamenalo by to, že vstupný reťazec nie je reťazec popísaný definovanou gramatikou, alebo má syntaktické chyby. Postup uplatňovania pravidiel gramatiky určených na analýzu vstupného reťazca bude reprezentovať derivačný strom [8, 10, 11, 12].

Na obrázku 3.2 je zobrazený derivačný strom, ktorý zodpovedá nasledujúcemu aritmetickému výrazu “$alfa + 21 * (7 + 3) + beta * 4$”.



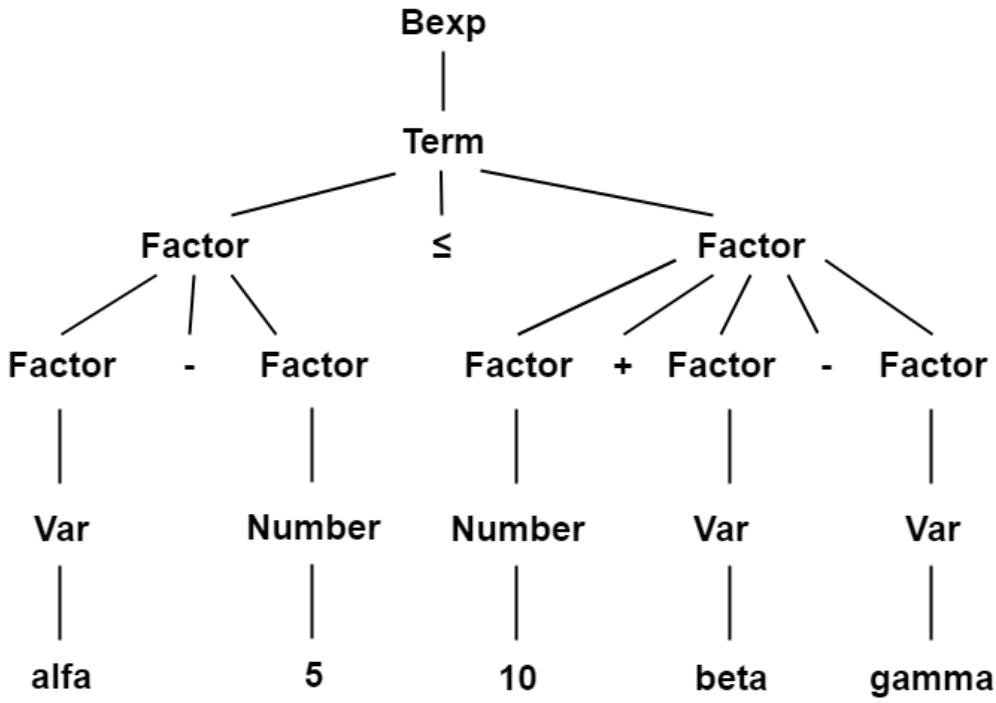
Obr. 3.2: Derivačný strom aritmetického výrazu

Na obrázku 3.3 je zobrazený strom abstraktnej syntaxe, ktorý zodpovedá vyššie uvedenému aritmetickému výrazu.



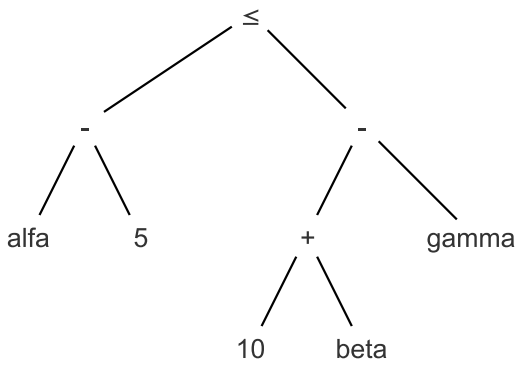
Obr. 3.3: Strom abstraktnej syntaxe aritmetického výrazu

Na obrázku 3.4 je zobrazený derivačný strom, ktorý zodpovedá nasledujúcemu boolovskému výrazu "$alfa - 5 \leq 10 + beta - gamma$".



Obr. 3.4: Derivačný strom boolovského výrazu

Na obrázku 3.5 je zobrazený strom abstraktnej syntaxe, ktorý zodpovedá vyššie uvedenému boolovskému výrazu.



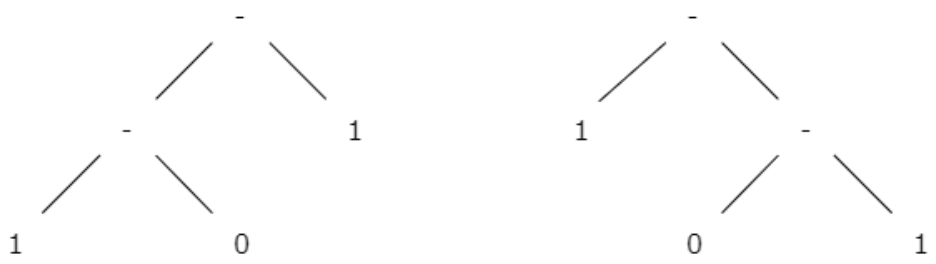
Obr. 3.5: Strom abstraktnej syntaxe boolovského výrazu

3.8 Syntaktická nejednoznačnosť

Gramatika jazyka je syntakticky nejednoznačnou, ak nejaký zo vstupných výrazov v jej jazyku má viac než jeden strom abstraktnej syntaxe. Programovacie jazyky môžu byť definované nejednoznačnými gramatikami. V prípade, ak existujú nejednoznačnosti, potrebujeme zavádzať konvencie, ktoré zaručujú odvodenie jedného stromu abstraktnej syntaxe pre každý vstupný výraz [13].
Napríklad, nasledujúca gramatika:

$$\begin{aligned} Expr &\rightarrow Term\ \{\ \text{“}-\text{”}\ Number\ \}\\ Number &\rightarrow Number\ \{\ \text{‘}\mathbf{0}\text{’}\ |\ \text{‘}\mathbf{1}\text{’}\ \} \end{aligned}$$

nie je jednoznačná, lebo vstupný výraz $\text{“}1-0-1\text{”}$ má dva stromy abstraktnej syntaxe, ktoré zodpovedajú nasledujúcim dvom variantom $\text{“}(1-0)-1\text{”}$ a $\text{“}1-(0-1)\text{”}$. Na obrázku 3.6 sú zobrazené možné verzie stromov abstraktnej syntaxe.



Obr. 3.6: Nejednoznačné syntaktické stromy aritmetického výrazu

3.9 Analýza aritmetických a logických výrazov

Vyvinutá aplikácia má analyzovať aritmetické a logické výrazy, ktoré zodpovedajú syntaxi podľa pravidiel 3.4 a 3.8, napríklad "$alfa + 21 * (7 + 3) + beta * 4$" alebo "$alfa - 5 \leq 10 + beta - gamma$". Pre uloženie komponentov výrazu a pre kontrolu parity zátvoriek možno použiť zásobník. Podobne možno zásobník použiť aj na analýzu samotných aritmetických alebo logických výrazov. Tieto metódy je možné implementovať rôznymi prístupmi. A napriek tomu je táto dôležitá praktická aplikácia zásobníku mimoriadne výhodná a mnohé jej aspekty sú zaujímavé samé osebe.

Ako sa ukázalo, je dosť ťažké vypočítať výsledok aritmetického alebo logického výrazu (v prípade pre počítačový algoritmus). Najjednoduchší spôsob spočíva v rozdelení tohto procesu na dva kroky [14]:

- transformovanie aritmetického alebo logického výrazu do postfixnej notácie, zvanej aj obrátená (reverzná) poľská notácia;
- vyhodnotenie výrazu v postfixnej notácii.

Krok 1 je pomerne komplikovaný, ale 2. krok je jednoduchý. V každom prípade dvojfázová schéma poskytuje jednoduchší algoritmus ako pokus priamo analyzovať aritmetické alebo boolovské výrazy. Oba kroky je možné implementovať rôznymi spôsobmi. Napríklad, rekurzívnymi algoritmami alebo pomocou zásobníka. V oboch prípadoch počas 2. kroku, postupného vyhodnotenia výrazu, bude zároveň prebiehať syntaktická kontrola správnosti vstupného výrazu.

V našej implementácii oba kroky, pre transformovanie aritmetického alebo boolovského výrazu v postfixnej notácii a pre jeho vyhodnotenie, budú implementované pomocou zásobníkov. Predtým, než začneme implementáciu, je potrebné sa zoznámiť s postfixnou notáciou.

3.10 Postfixná notácia

Pri zápise tradičných aritmetických alebo logických výrazov je operátor ($+$, $-$, $\leq$ atď.) umiestnený medzi dvoma operandmi. Tento zápis má názov infixná notácia. Píšeme

$$A + B$$

alebo

$$A * B + C.$$

Pôvodnú (nereverznú) poľskú notáciu alebo PN (angl. Polish Notation) navrhol poľský logik Jan Lukasiewicz už v roku 1924, pretože mu vyhovoval zápis, ktorý nepotreboval zátvorky a nemusel brať do úvahy priority operátorov. Notácia bola prefixná, teda operátor sa písal pred svojimi operandmi:

$$+ \; A \; B$$

alebo

$$+ \; C \; * \; A \; B.$$

Takýto výraz sa dá vyhodnotiť jedným prechodom zľava doprava.

Omnoho používanejšia a prehľadnejšia je obrátená (reverzná) poľská notácia, alebo RPN (angl. Reverse Polish Notation) zapisovaná v postfixnom tvare, kde je operátor umiestnený po dvoch operandoch. Týmto spôsobom, sa

$$A + B$$

z infixného zápisu transformuje na

$$A \; B \; +$$

v postfixnom zápise [13, 14, 15, 16, 17, 18].
Postfixná forma sa niekedy nazýva aj bezzátvorková forma, pretože zátvorky sú zbytočné [17, 18]. Tento variant notácie bol navrhnutý nezávisle od seba viacerými ľuďmi (prvý návrh bol v roku 1954), keď bolo potrebne implementovať vyhodnocovanie výrazov v počítačoch. Z hľadiska implementácie je veľmi výhodné minimum požadovanej pamäte: na medzivýsledky postačuje jeden zásobník.

Tabuľka 3.1: Vybrané infixné a postfixné výrazy

Infixné a postfixné výrazy	
Infixná notácia	Postfixná notácia
$A + B - C$	$AB + C-$
$A * B/C$	$AB * C/$
$A + B * C$	$ABC * +$
$A * B + C$	$AB * C+$
$A * (B + C)$	$ABC + *$
$A * B + C * D$	$AB * CD * +$
$(A + B) * (C - D)$	$AB + CD - *$
$((A + B) * C) - D$	$AB + C * D-$
$A + B * (C - D/(E + F))$	$ABCDEF + / - *+$

Hlavnou výhodou použitia obrátenej (reverznej) poľskej notácie pri implementácii našej aplikácie je jednoduchý spôsob vyhodnotenia akéhokoľvek zložitého vstupného výrazu.

Prefixná notácia je funkčne ekvivalentná postfixnej notácii, ale v praxi sa zriedka používa. Chceme upozorniť, že infixná notácia nemôže vzniknúť "otočením" zápisu prefixnej notácie: príčinou sú nekomutatívne operátory. Napríklad pre výraz "$+\ A\ B$" je reverzný zápis "$B\ A\ +$", ale pre "$/\ A\ B$" by sme po obrátení mali "$B\ A\ /$", čo dáva úplné iný výsledok.

3.11 Algoritmus shunting-yard

Shunting-yard algoritmus predstavil Edsger Wybe Dijkstra v 1960 roku pri implementácii jedného z prvých kompilátorov ALGOL 60. Taktiež v podstate rovnaký algoritmus je opísaný Friedrichom Bauerom a Klausom Samelsonom vo februári 1960 roku v Communications of the ACM, vol. 3, #2 "Sequential formula translation"[19, 6].

Hlavný princíp shunting-yard algoritmu je v tom, že transformuje aritmetický alebo boolovský výraz z infixnej formy do formy obrátenej poľskej notácie. Pre postfixnú notáciu je potrebné evidovať dva zásobníky. Prvý zásobník, ktorý bude nazvaný ako výstupný, bude uchovávať výstupnú množinu lexém a druhý zásob-

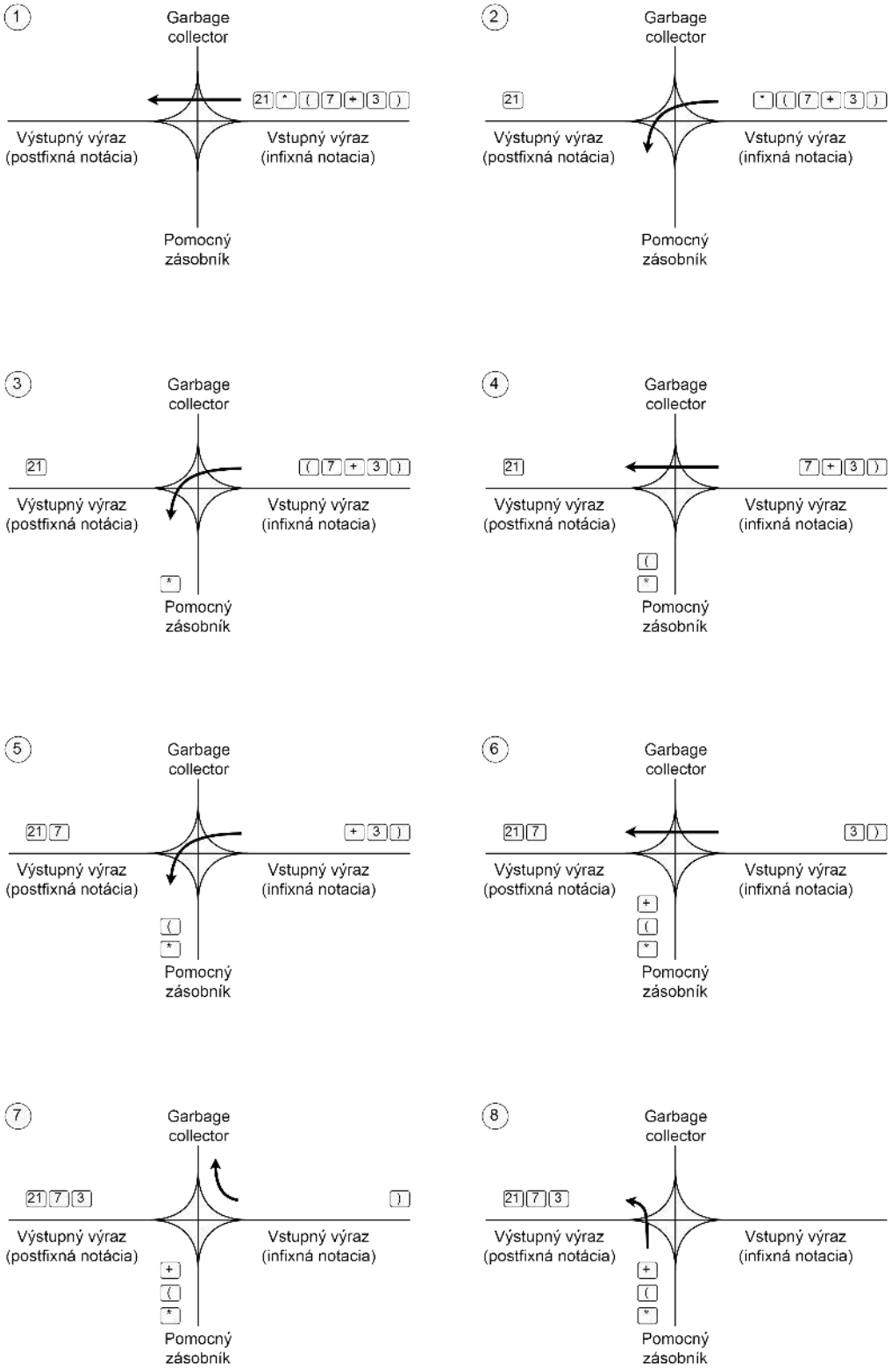
ník, nazvaný pomocný zásobník, bude uchovávať operátory [19, 20, 6].

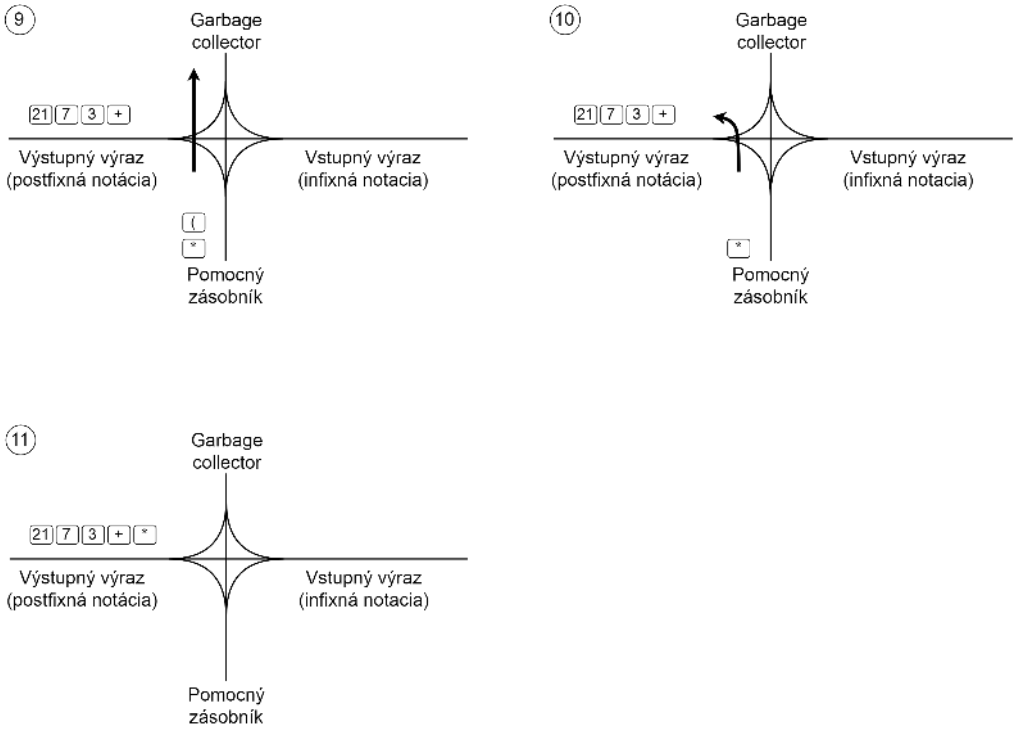
Algoritmus teda pracuje tak, že postupne číta lexémy zo vstupného poľa lexém a vloží prečítanú lexému do jedného z dvoch zásobníkov podľa nasledovných pravidiel [19, 20, 6]:

- Ak je lexéma konštanta alebo premenná, tak sa vloží tento token do výstupného zásobníka.
- Ak je lexéma operátor, tak ju vloží do pomocného zásobníka. Avšak predtým, než sa tak stane, skontroluje prioritu operátora na vrchu pomocného zásobníka. Ak sa tam nachádza operátor s vyššou alebo rovnakou prioritou a aktuálne spracovávaný operátor je asociatívny zľava, alebo aktuálne spracovávaný operátor je asociatívny sprava a na vrchu pomocného zásobníka je operátor s vyššou prioritou, potom sa vyberie operátor z vrchu pomocného zásobníka a vloží sa do výstupného zásobníka. Tento postup sa opakuje, pokým nie je pomocný zásobník prázdny, alebo podmienka pre odobratie operácie z vrchu pomocného zásobníka nie je splnená.
- Ak je lexéma ľavou zátvorkou, tak sa vloží do pomocného zásobníka. Avšak ak je lexéma pravou zátvorkou, potom sa vyberú lexémy z pomocného zásobníka a vkladajú sa do výstupného zásobníka, pokým nie je na vrchu pomocného zásobníka ľavá zátvorka. Potom sa ešte vyberie ľavá zátvorka z pomocného zásobníka. Pokiaľ sa nenarazí na ľavú zátvorku, potom zátvorky v textovom vstupe sú zadané nesprávne a textový vstup sa prehlási za neplatný.

Pokiaľ sa už spracujú všetky tokeny z textového vstupu, postupne sa presunú tokeny z pomocného zásobníka do výstupného zásobníka.

Na obrázku 3.7 je možné vidieť postupné spracovanie vstupného aritmetického výrazu “$21 * (7 + 3)$”.


1
Garbage collector
21 * (7 + 3)
Výstupný výraz (postfixná notácia)
Vstupný výraz (infixná notacia)
Pomocný zásobník
2
Garbage collector
21
* (7 + 3)
Výstupný výraz (postfixná notácia)
Vstupný výraz (infixná notacia)
Pomocný zásobník
3
Garbage collector
21
(7 + 3)
*
Výstupný výraz (postfixná notácia)
Vstupný výraz (infixná notacia)
Pomocný zásobník
4
Garbage collector
21
7 + 3)
(*
Výstupný výraz (postfixná notácia)
Vstupný výraz (infixná notacia)
Pomocný zásobník
5
Garbage collector
21 7
+ 3)
(*
Výstupný výraz (postfixná notácia)
Vstupný výraz (infixná notacia)
Pomocný zásobník
6
Garbage collector
21 7
3)
+ (*
Výstupný výraz (postfixná notácia)
Vstupný výraz (infixná notacia)
Pomocný zásobník
7
Garbage collector
21 7 3
)
+ (*
Výstupný výraz (postfixná notácia)
Vstupný výraz (infixná notacia)
Pomocný zásobník
8
Garbage collector
21 7 3
+ (*
Výstupný výraz (postfixná notácia)
Vstupný výraz (infixná notacia)
Pomocný zásobník



Obr. 3.7: Spracovanie aritmetického výrazu shunting-yard algoritmom

Po úspešnom spracovaní vstupného výrazu shunting-yard algoritmom na výstupnom zásobníku dostaneme obrátenú (reverznú) poľskú notáciu vstupného výrazu.

3.12 Stromové štruktúry

Pre implementáciu grafickej reprezentácie je potrebné sa zoznámiť so stromovými štruktúrami.

Štruktúra strom so základným typom **T** je buď prázdna štruktúra alebo vrchol typu **T** spolu s konečným počtom pripojených disjunktných stromových štruktúr so základným typom **T**, ktoré nazývame podstromy. Rekurzia je efektívny a elegantný prostriedok na definovanie stromovej štruktúry. Existuje viacero spôsobov reprezentácie stromovej štruktúry [17]:

- množín vnorených do seba;
- zátvoriek vnorených do seba;
- viacúrovňového zarovnávania;

- grafu.

Pre názornosť grafickej reprezentácie v našej práci použijeme grafovú štruktúru, lebo tá explicitne zobrazuje vzťahy vetvenia, ktoré viedli ku vzniku pojmu strom.

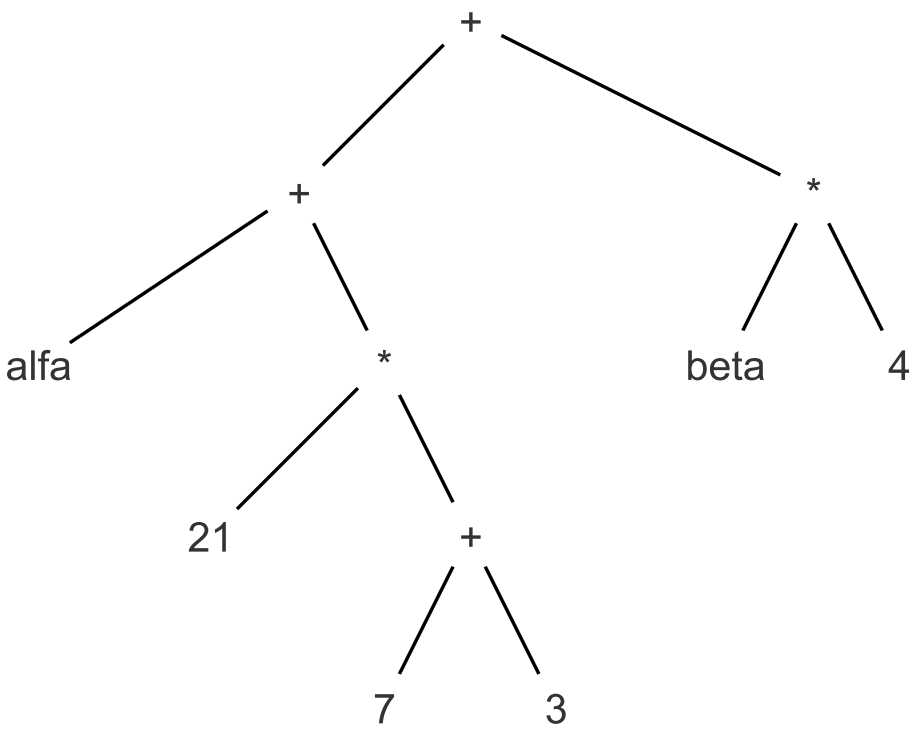
Najvyšší vrchol sa bežne nazýva koreň stromu. Koreň stromu je podľa definície na prvej úrovni. Maximálnu úroveň ľubovoľného prvku v strome nazývame jeho hĺbkou alebo výškou. Ak nejaký prvok nemá nasledovníkov, hovoríme, že je koncovým prvkom alebo listom stromu. Prvok, ktorý nie je listom, nazývame vnútorným vrcholom. Počet (priamych) nasledovníkov vnútorného vrcholu nazývame jeho stupňom. Maximálny stupeň spomedzi všetkých vrcholov určuje stupeň stromu [17, 18, 21, 22, 23].

Počet vetiev (hrán) alebo vrcholov, ktoré treba v strome prejsť, aby sme sa dostali od koreňa k vrcholu x, sa nazýva dĺžka cesty vrcholu x. Táto dĺžka sa pre koreň rovná jednotke, pre priameho potomka koreňa dvojke atď. Vo všeobecnosti platí, že dĺžka cesty vrcholu na i-tej úrovni sa rovná hodnote i. Dĺžka cesty celého stromu je definovaná ako súčet dĺžok ciest jednotlivých vrcholov stromu [17, 18, 21, 22, 23].

V našom prípade vnútornými vrcholmi sú aritmetické a boolovské operátory. Unárne operátory "$+$" a "$-$" pre aritmetické výrazy a "$\neg$" pre boolovské výrazy majú po jednom nasledovníkovi. Všetky binárne operátory majú dvoch nasledovníkov, a preto v našom prípade budeme mať strom druhého stupňa - binárny strom.

Definujeme binárny strom ako konečnú množinu prvkov (vrcholov), ktorá je buď prázdna, alebo sa skladá z koreňa (vrchola) a z dvoch disjunktných stromov nazývaných ľavý a pravý podstrom koreňa [17, 18, 21, 22, 23].

Operátory aritmetického alebo boolovského výrazu tvoria vrcholy vetvenia v strome a operandy ich podstromy, čo je uvedené na obrázku 3.8.



Obr. 3.8: Reprezentácia výrazu "$alfa + 21 * (7 + 3) + beta * 4$"

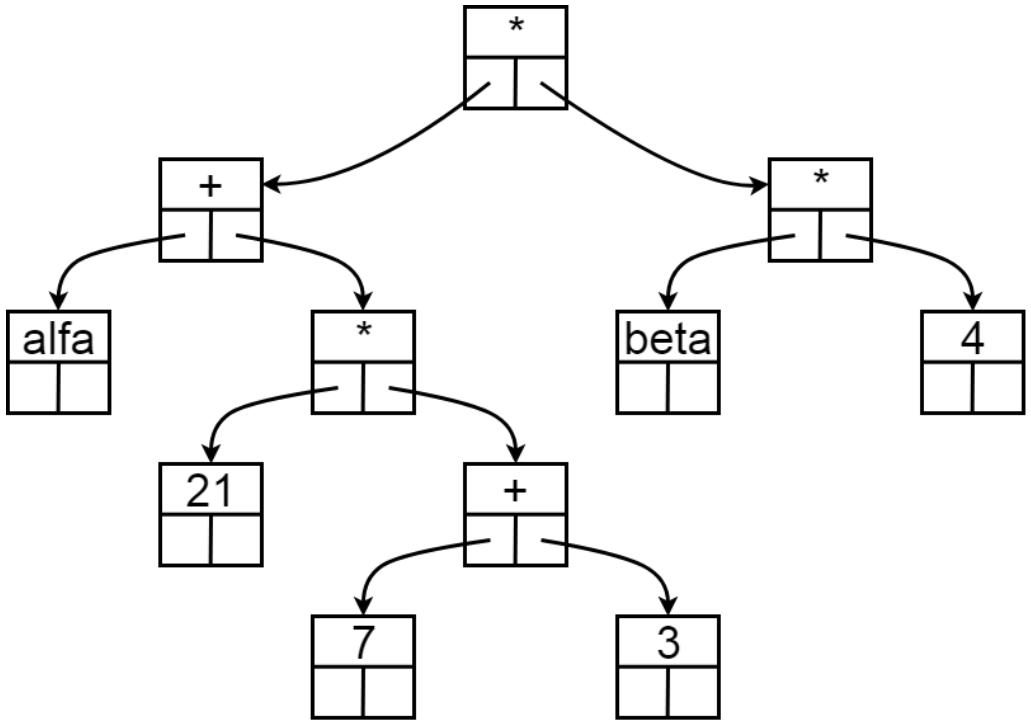
Vráťme sa teraz k problematike reprezentácie stromov. Vzhľadom na to, že ide o rozvetvené rekurzívne štruktúry, bude zaiste výhodné, ak využijeme dobre známe schopnosti smerníkov. Je pochopiteľné, že nemá zmysel definovať premenné s pevnou stromovou štruktúrou. Namiesto toho definujeme vrcholy stromu ako premenné s pevnou štruktúrou, t.j. pevne stanoveného typu, v rámci ktorého stupeň stromu určuje počet smerníkov na podstromy príslušného vrcholu. Referenciu na prázdny strom vyjadrujeme konštantou prázdneho smerníka, obvykle **nil**. Potom strom znázornený na obrázku 3.8 pozostáva z vrcholov, ktorých typ je definovaný takto:

```
public class Node
{
        NodeType type;
        String name;
        int value;
        boolean bValue;
```

```
        Node left, right;

        enum NodeType
        {
                UNARY_OPERATOR,
                BINARY_OPERATOR,
                VARIABLE,
                NUMBER,
                BOOLEAN_CONSTANT
        }
}
```

Reprezentácia takéhoto stromu je uvedená na obrázku 3.9.



Obr. 3.9: Strom zobrazený ako štruktúra údajov

Existujú aj iné spôsoby reprezentácie abstraktnej myšlienky stromovej štruktúry pomocou rôznych typov údajov [17, 18].

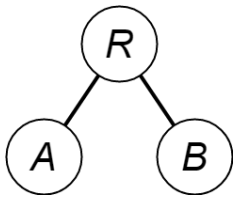
3.13 Základné operácie na binárnych stromoch

Existuje mnoho úloh, ktoré sa dajú riešiť na stromovej štruktúre. Najbežnejšou je uskutočnenie danej operácie P na každom prvku stromu. Operáciu P chápeme ako parameter všeobecnejšej úlohy navštívenia všetkých vrcholov stromu, ktoré sa zvyčajne nazýva prechod stromu [17].

Ak sa na túto úlohu pozeráme ako na jednoduchý sekvenčný proces, je jasné, že jednotlivé vrcholy v strome budú navštívené v určitom špecifickom poradí a možno sa na ne pozerať, ako keby boli lineárne usporiadané. V skutočnosti je opis mnohých algoritmov oveľa jednoduchší, ak vychádzame z toho, že spracovanie každého ďalšieho prvku v strome je možné na základe príslušného usporiadania [17].

Rozlišujeme tri základné usporiadania, ktoré sú prirodzeným dôsledkom stromovej štruktúry. Podobne ako samotná štruktúra, aj ony sa dajú vhodne rekurzívne vyjadriť. V súlade s binárnym stromom na obrázku 3.10, v ktorom symbolom R označujeme vrchol stromu, a symbolmi A, B ľavý a pravý podstrom, sú tieto tri usporiadania [17]:

- Priame (angl. preorder): R, A, B (najprv sa navštívil koreň a potom podstromy);
- Vnútorné (angl. inorder): A, R, B;
- Spätné (angl. postorder): A, B, R (koreň sa navštívi až po podstromoch).



Obr. 3.10: Binárny strom

Rozoznávame tri druhy výrazov: priamy prechod stromu, reprezentujúceho daný výraz, poskytuje prefixný zápis, spätný prechod postfixný a vnútorný prechod infixný zápis [17]. Všimnime si, že v prípadoch prefixného a postfixného zápisu nepotrebujeme zátvorky.

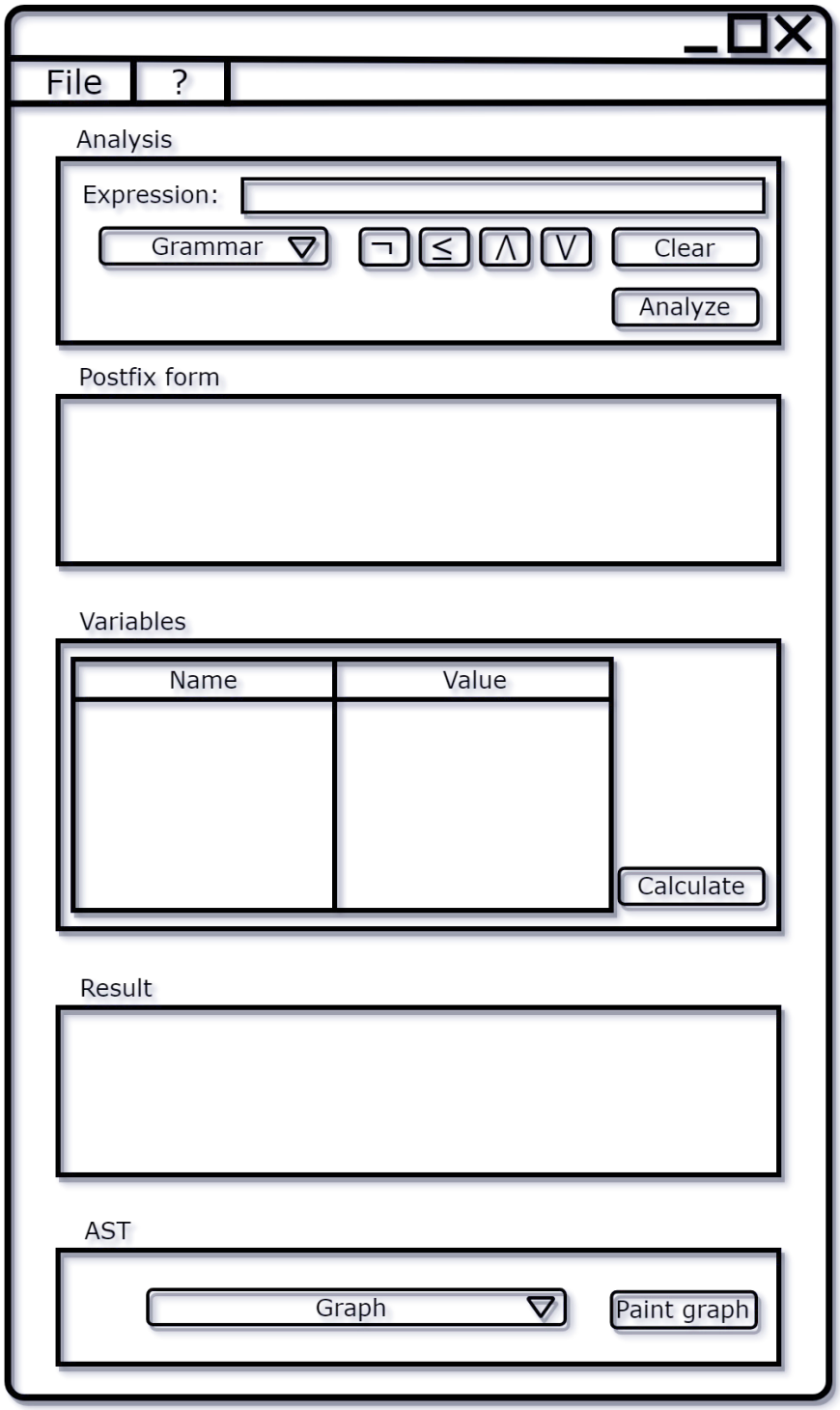
Z vyššie uvedeného možno povedať, že operácie na rekurzívne definovaných štruktúrach údajov sa najvhodnejšie definujú prostredníctvom rekurzívnych algoritmov.

4 Syntetická časť práce

Syntetická časť práce obsahuje popis samotného spôsobu implementácie programu. Pri implementácii je potrebné brať do úvahy to, že táto aplikácia má byť predovšetkým učebnou pomôckou pre študentov na zjednodušenie pochopenia teórie sémantiky programovacích jazykov pre aritmetické a boolovské výrazy. Aplikácia by preto nemala pôsobiť zložito, ale práve naopak, používateľské rozhranie by malo byť prehľadné a intuitívne jednoduché.

4.1 Návrh používateľského rozhrania

Na obrázku 4.1 je zobrazený návrh používateľského rozhrania.



Obr. 4.1: Návrh používateľského rozhrania

Používateľské rozhranie by malo obsahovať nasledujúce skupiny komponentov:

- **Hlavné menu**, ktoré bude obsahovať nasledujúce komponenty:
  - **File** bude obsahovať základnú operáciu **Exit**;
  - **?** bude obsahovať základnú informáciu o aplikácii **About**;
- **Analyze** pre zadanie vstupného výrazu, ktorá bude obsahovať nasledujúce komponenty:
  - **Expression** bude textové pole, do ktorého sa budú primárne zadávať aritmetické alebo boolovské výrazy pre spracovanie;
  - **Grammar** bude prepínač, ktorý umožňuje výber druhu gramatiky;
  - $\boxed{\neg}$, $\boxed{\leq}$, $\boxed{\wedge}$, $\boxed{\vee}$ budú tlačidlá na pridanie jednotlivých operátorov do vstupného výrazu;
  - **Clear** bude tlačidlo pre vymazanie vstupného poľa;
  - **Analyze** bude tlačidlo na spustenie lexikálnej a syntaktickej analýzy a prepis vstupného výrazu do postfixnej formy;
- **Postfix form** bude textové pole na výpis postfixnej formy vstupného výrazu;
- **Variables** pre zadanie hodnôt premenných, ktorá bude obsahovať nasledujúce komponenty:
  - **Table** bude tabuľka pre interaktívny vstup hodnôt premenných, ktorá sa vyskytujú vo vstupnom výraze;
  - **Calculate** bude tlačidlo na vyhodnotenie vstupného výrazu;
- **Result** bude textové pole na výpis výsledku vyhodnotenia vstupného výrazu;
- **AST** pre zadanie parametrov kreslenia stromu abstraktnej syntaxe, ktorá bude obsahovať nasledujúce komponenty:
  - **Graph** bude prepínač, ktorý umožňuje výber druhu vykreslenia stromu abstraktnej syntaxe;
  - **Paint graph** bude tlačidlo na vykreslenie stromu abstraktnej syntaxe;

4.2 Voľba programovacieho jazyka

Pre implementáciu aplikácie bude použitý programovací jazyk Java. Java je objektovo-orientovaný jazyk vyššej úrovne, umožňuje však aj klasické procedurálne programovanie. Je to platformovo nezávislý a najrozšírenejší programovací jazyk súčasnosti, ktorého zdrojové programy sa nekompilujú do strojového kódu ale do medzistupňa, tzv. „byte-code“, ktorý nie je závislý od konkrétnej platformy [21, 22, 23].

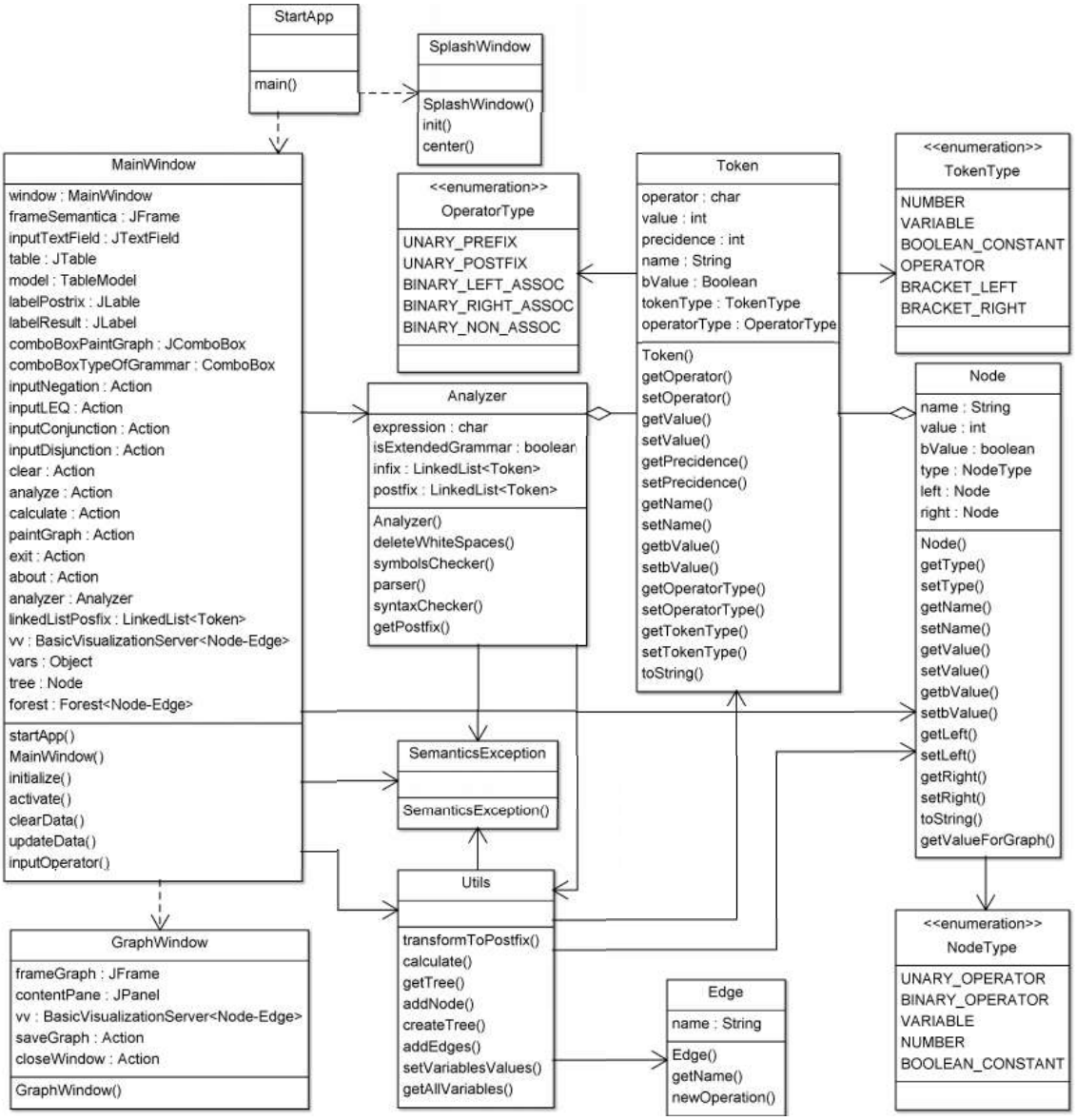
Na základe uvedeného bude navrhnutá aplikácia spustiteľná na rozličných operačných systémoch, ako napríklad Windows, Linux, macOS a Solaris. Táto vlastnosť je jedna z jeho najvýznamnejších, vďaka ktorej bude možné vyvíjanú aplikáciu spúšťať na ľubovoľnom operačnom systéme, a teda priaznivci akéhokoľvek prostredia nebudú ukrátení. K ďalším výhodám programovacieho jazyka Java je možné zaradiť zabezpečenosť, spoľahlivosť, podporu multivláknového vykonávania a taktiež paralelné spracovanie (angl. Message Passing Interface). Poskytuje tiež množstvo možností pre tvorbu používateľského rozhrania a používanie rôznych knižníc [21, 22, 23].

4.3 Implementácia

Aplikácia bude vyhodnocovať vstupný aritmetický alebo boolovský výraz zadaný ako reťazec a taktiež pre potreby výučby bude kresliť strom abstraktnej syntaxe. V súlade s analytickou časťou práce implementácia pozostáva z nasledujúcich bodov:

- lexikálna analýza vstupného výrazu;
- prepis vstupného výrazu do postfixnej notácie;
- syntaktická analýza vstupného výrazu;
- zadanie hodnôt premenných;
- vyhodnotenie používateľského vstupu;
- kreslenie stromu abstraktnej syntaxe.

Rozdelenie aplikácie do tried, ich usporiadanie a prepojenia su zobrazené pomocou diagramu tried 4.2.



Obr. 4.2: Diagram tried

4.3.1 Lexikálna analýza vstupného výrazu

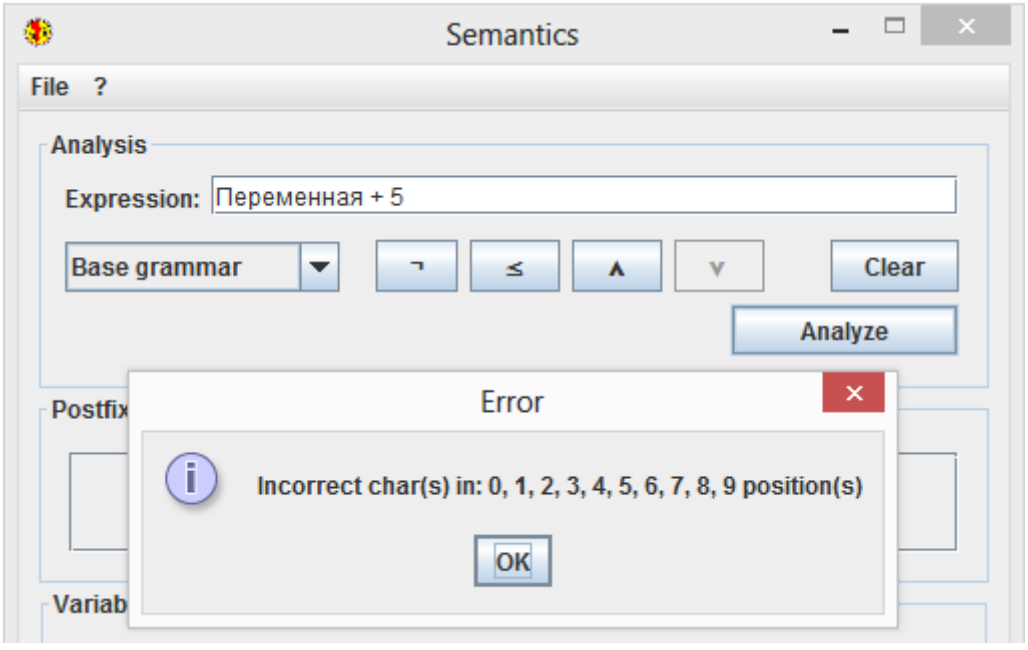
Lexéma je nedeliteľná časť výrazu, ktorá bežne pozostáva z niekoľkých symbolov. Rozpoznanie lexikálnych jednotiek je súčasťou lexikálnej analýzy. Počas lexikálnej analýzy je potrebné konkrétnu lexikálnu jednotku, napr. konkrétny identifikátor, rozpoznať a potom je potrebné ho preložiť. Preklad do formy vhodnej pre

ďalšie fázy prekladu programovacieho jazyka samozrejme nie je možný bez rozpoznania lexikálnej jednotky [9].

Pred začatím parsovania vstupného výrazu je potrebné odstrániť nevýznamné znaky (angl. whitespaces), také ako medzera, znak tabulácie, prechod na nasledujúci riadok, a podobne a sformovať celé lexémy. Tento postup nie je vyžadovaný ako súčasť algoritmu, môže však výrazne zjednodušiť pochopenie algoritmu a jeho implementáciu.

Pri implementácii lexikálneho analyzátora je potrebné oddeliť stavy lexikálneho analyzátora. Stav určuje aké symboly v daný moment môžu byť na vstupe lexikálneho analyzátora a aká bude reakcia na nich. Napríklad, ak na vstup lexikálneho analyzátora prišlo číslo, tak prejde do stavu "konštanta" a do skončenia konštanty (t.j. až do znaku operácie, zátvorky alebo na koniec riadku) na vstup lexikálneho analyzátora môžu prísť iba čísla.

V prípade, ak na vstup lexikálneho analyzátora príde symbol, ktorý nie je definovaný navrhnutou gramatikou, program sa zastaví, oznámi používateľovi správu o existujúcich chybách a poskytne používateľovi možnosť opraviť ich. Zotavenie aplikácie pri výskyte chyby počas spracovaní lexikálnej analýzy vstupného výrazu je uvedené na obrázku 4.3.



Obr. 4.3: Zotavenie aplikácie pri výskyte chyby počas spracovaní lexikálnej analýzy vstupného výrazu

Jeden stav je začiatočný – od neho začína práca lexikálneho analyzátora a jeden alebo niekoľko stavov musia byť koncovými.

Ďalej vytvorme tabuľku, ktorá definuje reakciu lexikálneho analyzátora pre vstupné symboly v závislosti na stave. Reakcia zvyčajne spočíva v zmene stavu lexikálneho analyzátora a iných operácií, napríklad, zápis aktuálneho symbolu do dočasnej premennej, zvýšenie alebo zníženie úrovne vnorenia s výskytom zátvoriek atď.

Zvyčajne je v programe lexikálny analyzátor implementovaný ako nekonečný cyklus s ukončením v prípade chyby alebo pri dosiahnutí konca vstupného reťazca. Takže chceme upozorniť, že dosiahnutie konca vstupného reťazca neznamená automaticky správnosť výrazu. Vnútri cyklu je podmieňovací blok, napríklad, po skupinách symbolov a v každom takom bloku je vlastný podmieňovací blok alebo prepínač po stavoch lexikálneho analyzátora.

Iný prístup, ktorý existuje pri vývoji analyzátora, je implementácia syntaktického analyzátora pomocou rekurzie [9].

Ak vstupný vyraz je správny, tak po spracovaní lexikálnym analyzátorom budeme mať zoznam lexém.

Časť lexém v aritmetických a logických výrazoch pozostáva z jedného symbolu, ale mená premenných a konštanty zvyčajne pozostávajú z niekoľkých symbolov. Preto je vstupné symboly potrebné ukladať do dočasnej premennej a keď sa táto lexéma skončí, skontrolovať, čo skutočne predstavuje.

Pre aritmetické a logické výrazy budeme mať nasledujúce lexémy:

- mená premenných;
- konštanty;
- operácie:
  - aritmetické (+, -, *, /);
  - logické ($\neg, \wedge, \vee$);
  - porovnania ($=, \leq$);
- pomocné symboly:
  - ľavá zátvorka;

– pravá zátvorka.

Rozoberme príklad, uvedený v predchádzajúcej kapitole (aritmetický výraz): "$alfa + 21 * (7 + 3) + beta * 4$". V procese lexikálnej analýzy bude vstupný reťazec konvertovaný do poľa lexém nasledujúcej štruktúry:

[0]	$\Rightarrow$	"$alfa$"	Premenná
[1]	$\Rightarrow$	"$+$"	Operátor - ľavo asociatívny
[2]	$\Rightarrow$	"21"	Konštanta
[3]	$\Rightarrow$	"$*$"	Operátor - ľavo asociatívny
[4]	$\Rightarrow$	"$($"	Ľavá zátvorka
[5]	$\Rightarrow$	"7"	Konštanta
[6]	$\Rightarrow$	"$+$"	Operátor - ľavo asociatívny
[7]	$\Rightarrow$	"3"	Konštanta
[8]	$\Rightarrow$	"$)$"	Pravá zátvorka
[9]	$\Rightarrow$	"$+$"	Operátor - ľavo asociatívny
[10]	$\Rightarrow$	"$beta$"	Premenná
[11]	$\Rightarrow$	"$*$"	Operátor - ľavo asociatívny
[12]	$\Rightarrow$	"4"	Konštanta

Takým spôsobom celá lexéma reprezentuje operátor (aritmetickú operáciu), alebo operand (číslo, skladajúce sa z jednej alebo viacerých číslic), alebo premennú (meno ktorej môže obsahovať číslice, ale musí sa začínať malým alebo veľkým písmenom) alebo zátvorku (ako element, ktorý mení prioritu vyhodnotenia aritmetických operácií).

Ďalej pre analýzu a spracovanie výrazu potrebujeme ho prepísať do postfixnej notácie.

4.3.2 Prepis vstupného výrazu do postfixnéj notácie

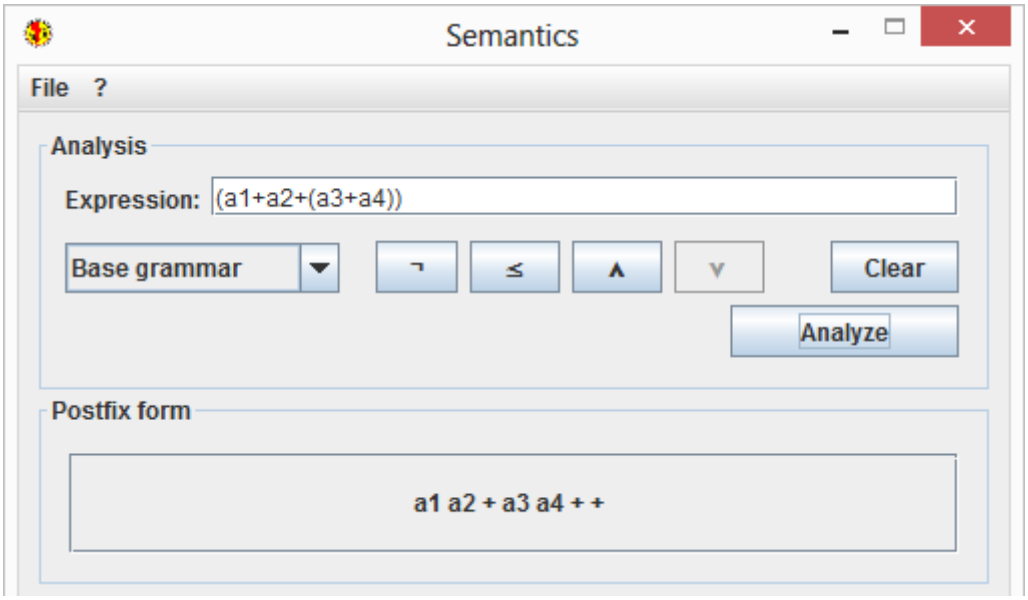
Ako sme už spomínali v analytickej časti práce, pre prepis vstupného výrazu do postfixnéj notácie, je vhodné použiť algoritmus shunting-yard, ktorý napríklad môže byť implementovaný pomocou dátovej štruktúry typu zásobník. Pri implementácii metódy, ktorá realizuje tento algoritmus použijeme dátovú štruktúru typu **LinkedList**, ktorá umožňuje pristupovať k zoznamu z oboch strán [21, 22].

Vytvoríme si metódu *transformToPostfix()*, parametrom ktorej je zoznam lexém v infixnom tvare a návratový parameter je zoznam lexém v postfixnom tvare.

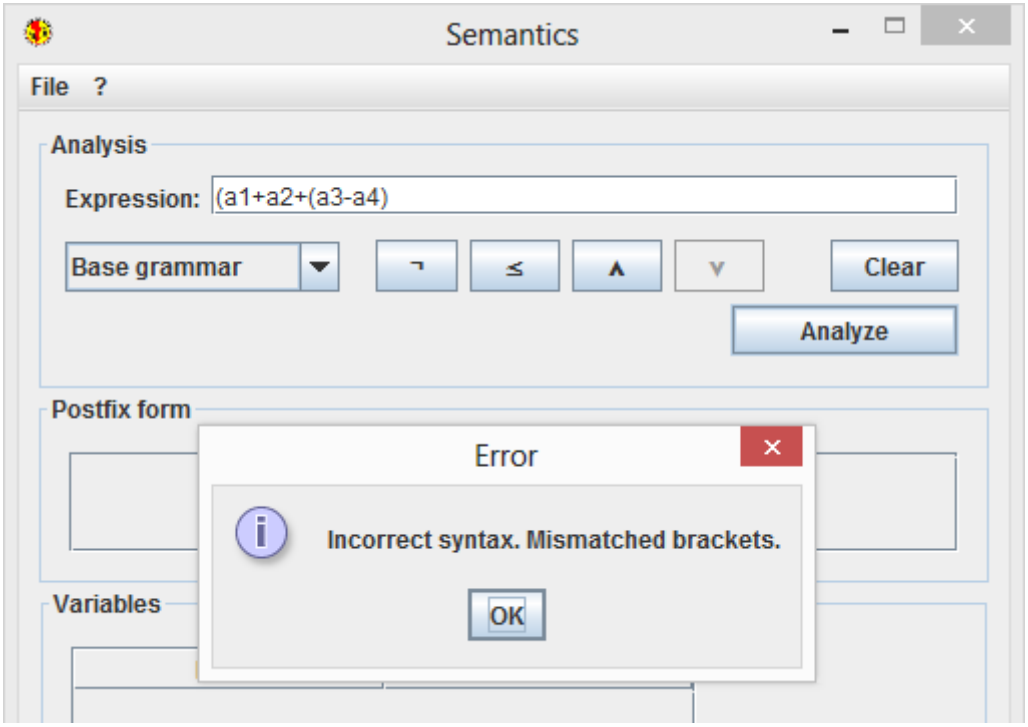
Algoritmus teda pracuje tak, že postupne vyberie prvú lexému zo vstupného zoznamu lexém pokiaľ vstupný zoznam nebude prázdnym a vkladá prečítanú lexému do jedného z dvoch zoznamov podľa nasledovných pravidiel:

- Ak je lexéma konštanta alebo premenná, tak vloží tento token na koniec výstupného zoznamu;
- Ak je lexéma unárny operátor, tak ju vloží na koniec pomocného zoznamu;
- Ak je lexéma binárny operátor, tak ju vloží na koniec pomocného zoznamu. Avšak predtým algoritmus skontroluje prioritu operátora na konci pomocného zoznamu:
  - Ak sa tam nachádza operátor s vyššou alebo rovnakou prioritou a aktuálne spracovávaný operátor je asociatívny zľava, alebo nie je asociatívny, vloží sa na koniec výstupného zoznamu;
  - Ak sa tam nachádza operátor s vyššou prioritou a aktuálne spracovávaný operátor je asociatívny sprava, potom sa vyberie operátor z konca pomocného zoznamu a vloží sa na koniec výstupného zoznamu;
  - Tento postup sa opakuje pokým nie je pomocný zásobník prázdny, alebo podmienka pre odobratie operácie z vrchu pomocného zásobníka nie je splnená;
- Ak je lexéma ľavou zátvorkou, tak ju vloží na koniec pomocného zoznamu. Avšak ak je lexéma pravou zátvorkou, potom ju odstráni a vyberá lexémy z konca pomocného zoznamu a vkladá ich na koniec výstupného zoznamu, pokým nie je na konci pomocného zoznamu ľavá zátvorka. Potom ešte vyberie a odstráni ľavú zátvorku z konca pomocného zoznamu. Pokiaľ nenarazí na ľavú zátvorku, potom zátvorky v textovom vstupe sú zadané nesprávne a prehlási vstupný zoznam lexém za neplatný, zastaví spracovanie výrazu a upozorní používateľa.

Vyššie uvedený algoritmus realizuje transformáciu infixnej notácie na postfixnú, ako je uvedené na obrázku 4.4 a taktiež kontrolu parity zátvoriek sprava. V prípade výskytu tejto syntaktickej chyby vo vstupnom výraze ju zachytí program, zastaví sa spracovanie výrazu, upozorní používateľa a dá mu možnosť ju opraviť, ako je uvedené na obrázku 4.5.



Obr. 4.4: Transformácia vstupného výrazu do postfixnej formy



Obr. 4.5: Zotavenie pri výskyte chyby počas kontroly parity zátvoriek

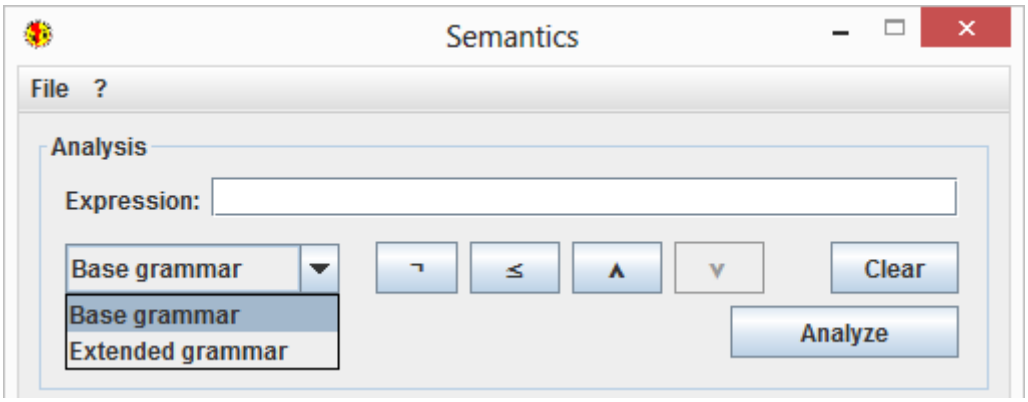
4.3.3 Syntaktická analýza vstupného výrazu

Základnou funkciou syntaktickej analýzy je kontrola syntaxe - gramatickej správnosti vstupného kódu - vstupného reťazca terminálnych symbolov [9]. Vstupným reťazcom v našom prípade je aritmetický alebo boolovský výraz zapísaný v jazyku $\mathscr{J}ane$, ktorého lexikálne jednotky sú spracované do formy symbolov. Vstupný reťazec je zapísaný v jazyku s definovanou gramatikou a syntaktická analýza môže byť založená na skúmaní, či existuje reťazec terminálnych symbolov produkovaný gramatikou, ktorý je rovnaký ako vstupný reťazec.

Jednou z dôležitých podúloh je kontrola mien premenných v zozname, lebo tá istá premenná môže byť prítomná niekoľkokrát v tom istom výraze, preto ak máme sformované meno premennej, potom musíme skontrolovať jej prítomnosť v zozname.

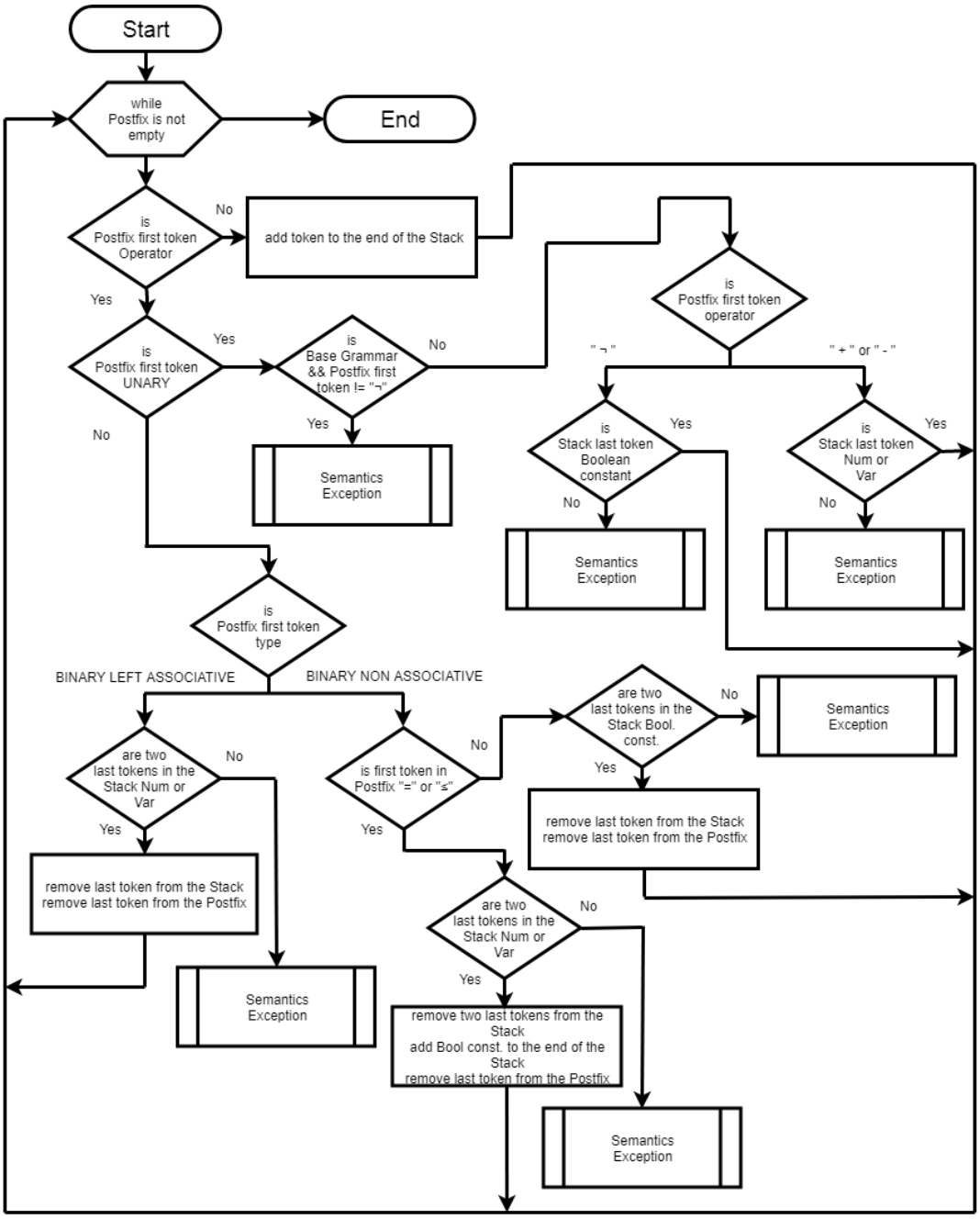
Okrem toho je dôležitou úlohou kontrola počtu ľavých a pravých zátvoriek, ktoré musia byť v rovnakom počte pri dosiahnutí konca spracovaného výrazu. Táto časť kontroly je už splnená počas transformácie vstupného výrazu do postfixnej notácie.

Vytvoríme metódu *syntaxChecker()*, parametrom ktorej je zoznam lexém v postfixnom tvare a boolovská premenná. Boolovská premenná určuje, či syntaktická kontrola zadaného výrazu bude prebiehať podľa základnej alebo podľa rozšírenej gramatiky. Hodnota boolovskej premennej `true` znamená vykonanie syntaktickej analýzy podľa rozšírenej gramatiky a `false` - vykonanie podľa základnej gramatiky. Hodnota boolovskej premennej sa určuje podľa polohy prepínača, ktorý umožňuje používateľovi zvoliť druh gramatiky, ako je ukázané na obrázku 4.6.



Obr. 4.6: Výber druhu gramatiky

Kontrola správnosti syntaxe bude implementovaná v súlade s definovanou gramatikou, použitím údajovej štruktúry zoznam a podľa vývojového diagramu ktorý je uvedený na obrázku 4.7.



Obr. 4.7: Vývojový diagram syntaktického analyzátora

Metóda *syntaxChecker()* sa začína stavom „Start“ a ukončí fungovanie jedným z možných stavov. Jeden z nich je *SemanticExeption*. Je to stav, kedy nastala vý-

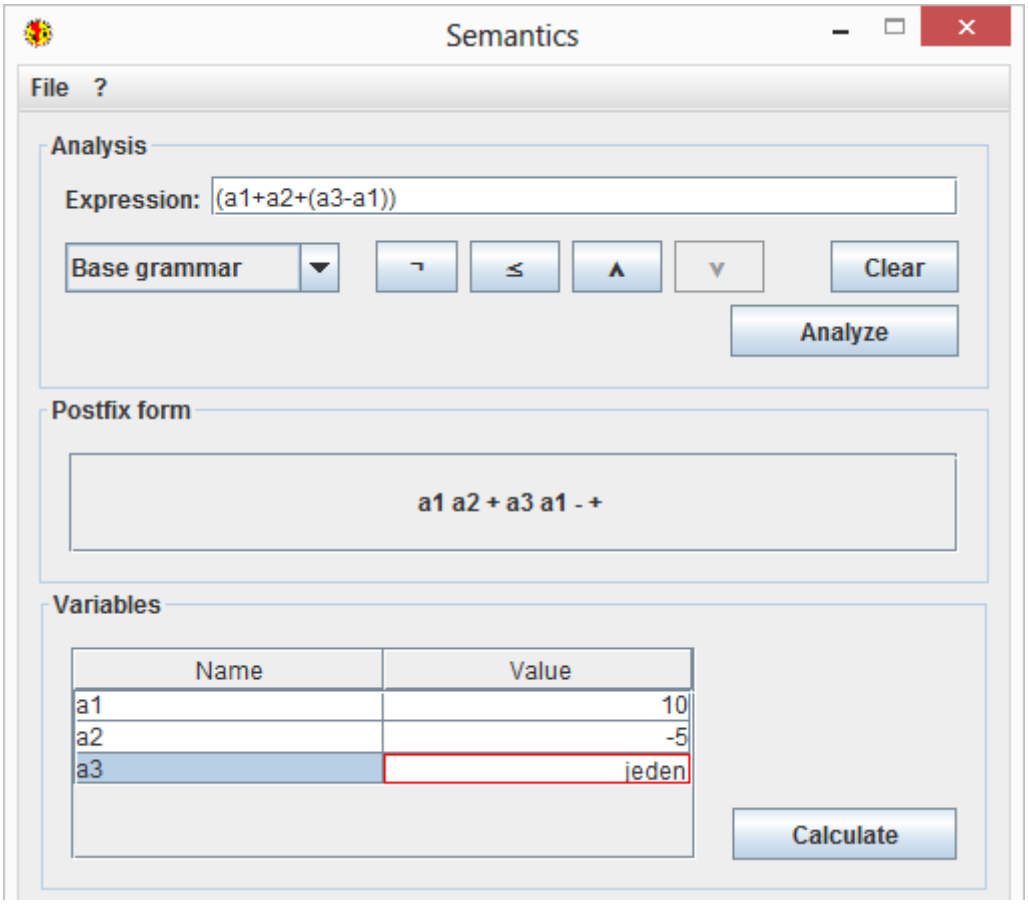
nimočná situácia kvôli nesprávnej syntaxi vstupného výrazu a metóda sa skončí. Zároveň používateľ bude oboznámený o existujúcej chybe. V prípade, že program dôjde do stavu „End“, znamená to, že vstupný výraz je syntakticky správny, metóda skončí a vráti riadenie.

4.3.4 Zadanie hodnôt premenných

Po stlačení tlačidla **Analyze** bude spustená lexikálna analýza, prepis vstupného výrazu do postfixnej notácie, syntaktická analýza a zároveň bude vytvorená štruktúra typu dvojrozmerné pole, ktoré bude obsahovať mená všetkých vyskytujúcich sa a neopakujúcich sa premenných. Na základe tohoto poľa následne vytvoríme model tabuľky potrebnej pre vytváranie komponentu **JTable**, ktorý sa zobrazí na používateľskom rozhraní a pomocou ktorého používateľ ma možnosť zadať hodnoty premenných.

Na implementáciu modelu tabuľky vytvoríme triedu ***TableModel***, ktorá je následníkom triedy ***AbstractTableModel***. Štruktúra tabuľky pozostáva z dvoch kolóniek: **Name** (meno premennej) a **Value** (hodnota premennej). Počet riadkov tabuľky rovná sa celkovému počtu neopakujúcich sa premenných.

Okrem toho sme v modeli tabuľky prekryli metódu *getColumnClass()*, na základe ktorej sme definovali pre prvú kolónku (**Name**) hodnotu typu **String.class** a pre druhú kolónku (**Value**) - **Integer.class**. Prepis tejto metódy bude zároveň kontrolovať vloženie hodnôt premenných a nedovolí používateľovi vložiť akúkoľvek hodnotu premennej, dovolí vložiť iba hodnotu typu **Integer**. Princíp fungovania je uvedený na obrázku 4.8



Obr. 4.8: Zadanie hodnôt premenných

Po stlačení tlačidla **Calculate** proces priradí každej lexéme, ktorá je premenná v zozname lexém hodnotu, ktorú používateľ zadal v tabuľke a následné spustí proces vyhodnotenia.

4.3.5 Vyhodnotenie používateľského vstupu

Vyhodnotenie používateľského vstupu je možné implementovať viacerými spôsobmi.

Prvý z nich - vytvoriť rekurzívnu údajovú štruktúru strom, uzlami ktorého sú operátory a listami sú operandy. Potom je potrebné vytvoriť rekurzívnu metódu priameho (angl. preorder) prechodu stromu. Po aplikovaní vytvorenej metódy na koreň stromu dostaneme výsledok vyhodnotenia vstupného výrazu.

Druhý spôsob vyhodnotenia používateľského vstupu je pomocou zásobníka. Práve tento spôsob implementujeme v našej aplikácii.

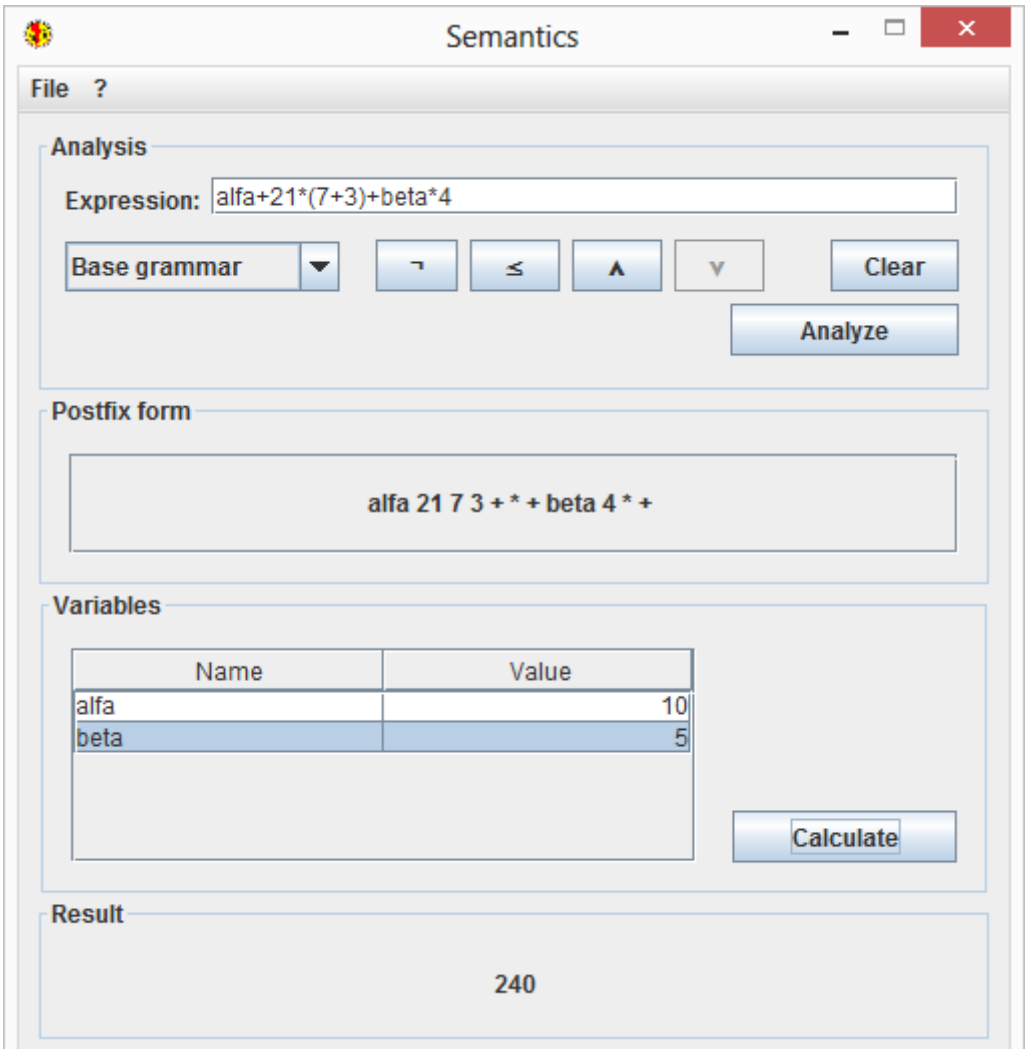
Vytvoríme statickú metódu *calculate()*. Vstupný parameter tejto metódy je údajová štruktúra typu **LinkedList** a výstupný je **Object**.

Algoritmus pre vyhodnotenie pracuje tak, že postupne vyberá lexémy zo vstupného zoznamu lexém a spracováva vybranú lexému podľa nasledovných pravidiel:

- Ak je lexéma konštanta alebo premenná, tak sa vloží tento token do výstupného zásobníka.
- Ak je lexéma unárny operátor, tak vyberie z vrcholu výstupného zásobníka token a aplikuje unárny operátor na vybraný token (operand). Výsledok vloží na vrchol výstupného zásobníka.
- Ak je lexéma binárny operátor, tak vyberie z vrcholu výstupného zásobníka dva tokeny a aplikuje na ne funkciu, ktorej argumentmi sú tieto dva tokeny. Výsledok vloží na vrchol výstupného zásobníka.

Po spracovaní algoritmu vstupný zoznam lexém bude prázdny a vo výstupnom zásobníku bude iba jeden token - to je výsledok vyhodnotenia vstupného výrazu. Po skončení metódy táto lexéma bude vrátená ako výstupný parameter.

Príklad vyhodnotenia vstupného výrazu "$alfa+21*(7+3)+beta*4$" je uvedený na obrázku 4.9



Obr. 4.9: Vyhodnotenie výrazu "$alfa + 21 * (7 + 3) + beta * 4$"

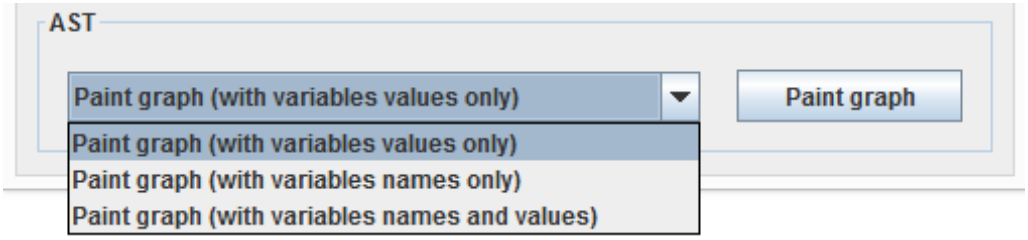
4.3.6 Kreslenie stromu abstraktnej syntaxe

Táto časť diplomovej práce je dôležitá, pretože poskytuje používateľovi grafickú reprezentáciu a prispieva k lepšiemu pochopeniu postupov aplikovania sémantických metód.

Vyvíjaná aplikácia poskytuje používateľovi možnosť výberu typu kreslenia listov stromu:

- iba hodnoty premenných,
- iba mená premenných,
- hodnoty aj mená premenných,

ako je to uvedené na obrázku 4.10



Obr. 4.10: Výber typu kreslenia listov stromu

Ďalej potrebujeme vytvoriť rekurzívnu údajovú štruktúru strom pre reprezentáciu vstupného výrazu. Na tieto účely vytvoríme triedu ***Node***, ktorá bude obsahovať nasledujúce členské premenné:

- `type` je typ uzla (enumeračný typ **NodeType**),
- `name` je meno operátora alebo premennej,
- `value` je hodnota konštanty alebo premennej,
- `bValue` je hodnota boolovskej konštanty,
- `left`, `right` sú potomkovia typu **Node**.

Enumeračný typ **NodeType** pozostáva z nasledujúcich položiek:

- `UNARY_OPERATOR` zahŕňa nasledujúce unárne operátory: “ $+$ ”, “ $-$ ”, “$\neg$”,
- `BINARY_OPERATOR` zahŕňa nasledujúce binárne operátory: “ $+$ ”, “ $-$ ”, “ $*$ ”, “$/$”, “ $=$ ”, “ $\leq$ ”, “ $\wedge$ ”, “ $\vee$ ”,
- `VARIABLE` je premenná,
- `NUMBER` je konštanta,
- `BOOLEAN_CONSTANT` je boolovská konštanta.

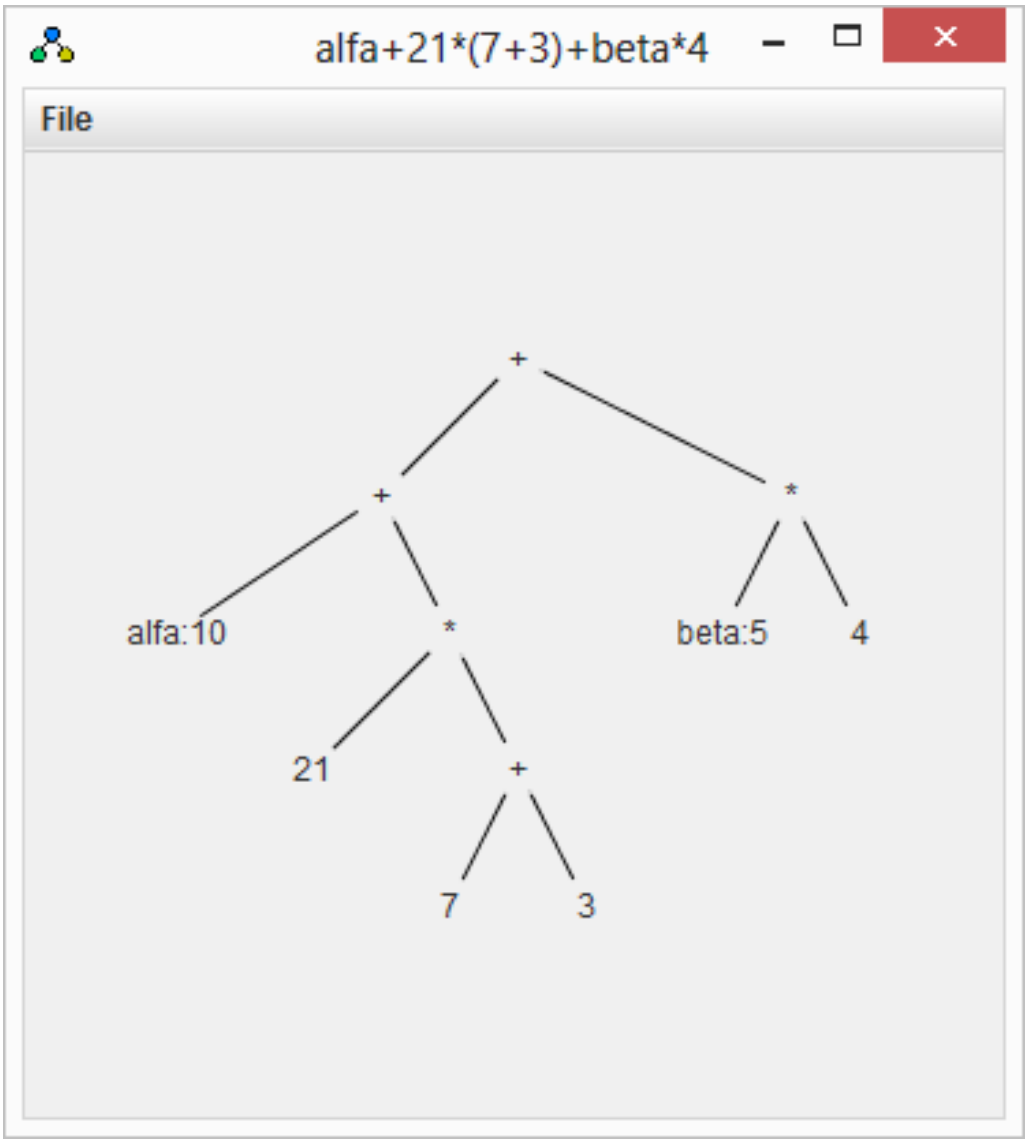
Ďalej vytvoríme rekurzívnu metódu priameho (angl. preorder) konštruovania stromu *getTree()*. Vstupný parameter funkcie je zoznam lexém a výstupný – rekurzívna údajová štruktúra typu **Node**.

Na kreslenie samotného grafu použijeme knižnicu "Jung" pre modelovanie a vizualizáciu grafov, s otvoreným zdrojovým kódom napísanú v jazyku Java [24].

Prvým krokom po stlačení tlačidla **Paint Graph** je potrebné skontrolovať polohu prepínača 4.10 a zistiť, v akom režime budeme kresliť listy stromu. V prípade, že poloha prepínača je na pozícii **Paint graph (with variables values only)** alebo **Paint graph (with variables names and values)**, je potrebne prečítať z objektu triedy ***TableModel*** všetky hodnoty zadané používateľom a priradiť ich premenným. Údajovú štruktúru typu **DelegateForest <Node, Edge>**, ktorá bude uchovávať uzly a hrany grafu, je potrebné vytvoriť v druhom kroku. Ďalej je potrebné vytvoriť rekurzívnu metódu *createTree()*, ktorá skonštruuje z údajovej štruktúry typu **Node** údajovú štruktúru typu **DelegateForest<Node, Edge>**.

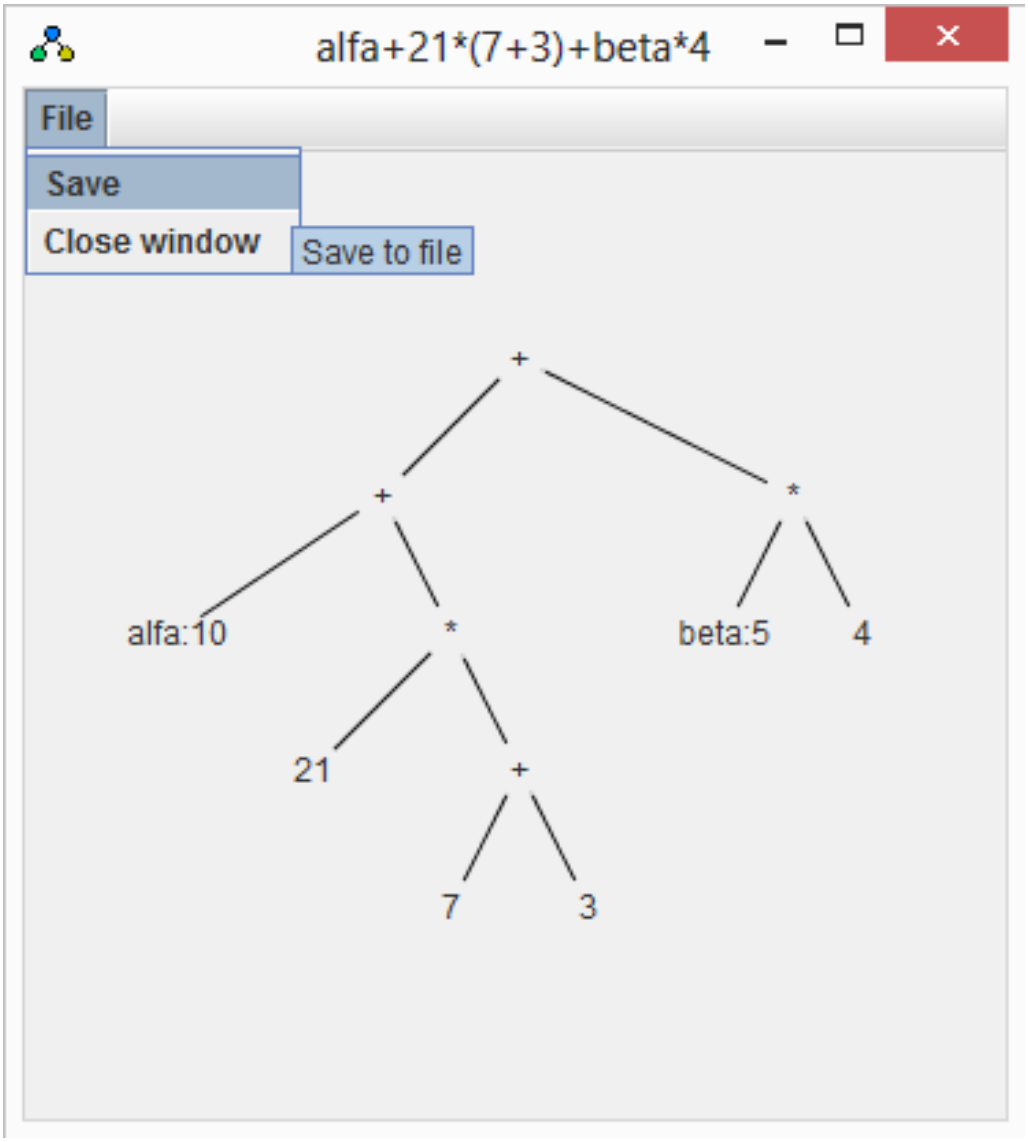
Na samotné vykresľovanie vytvoríme triedu ***PaintGraph***, ktorá dedí od triedy ***JFrame***. Konštruktor triedy ***PaintGraph*** bude obsahovať nasledujúce parametre: **String** je zadaný používateľský výraz– názov okna, **Forest<Node, Edge>** je údajová štruktúra a **int** je režim kreslenia listov stromu.

Po spracovaní konštruktora bude vytvorené okno, v ktorom bude vykreslený strom abstraktnej syntaxe, ktorý zodpovedá vstupnému výrazu, ako je to uvedené na obrázku 4.11.



Obr. 4.11: Kreslenie stromu abstraktnej syntaxe

Používateľ má možnosť uložiť skonštruovaný strom abstraktnej syntaxe voľbou položky **Save** v menu **File**, ako je to zobrazené na obrázku 4.12.



Obr. 4.12: Uloženie stromu abstraktnej syntaxe

Pre implementácii uloženia stromu abstraktnej syntaxe do PDF súboru sme použili knižnicu "freehep-graphicsio-pdf-2.1.1" [25] s otvoreným zdrojovým kódom napísanú v jazyku Java. Táto knižnica umožňuje uloženie obrázkov na základe vektorovej grafiky.

Implementácia samotného uloženia stromu abstraktnej syntaxe do PDF súboru poskytne veľkú výhodu používateľovi našej aplikácie, lebo takto má možnosť vygenerované obrázky použiť nielen pri študovaní predmetu Sémantika programovacích jazykov, ale aj pri výskumných prácach.

5 Vyhodnotenie

Neoddeliteľnou súčasťou v teoretickom a v praktickom vyučovacom procese sú moderné učebné pomôcky, ktoré obohacujú vyučovacie metódy a prispievajú k uľahčeniu didaktických zásad. Nemajú slúžiť iba k názornosti vyučovacieho procesu, ale umožňujú lepšie pochopenie učebného materiálu. Ich využitie je ďaleko širšie tým, že pomáhajú pochopiť aj abstraktné prvky učiva aj prispievajú k rozvoju myslenia. Učebné pomôcky majú veľmi dôležitú úlohu vo všetkých etapách výchovno-vzdelávacieho procesu a taktiež pri jeho uplatňovaní, prehlbovaní ako aj pri motivácii a spätnej väzbe.

Cieľom tejto diplomovej práce bolo vytvoriť softvérový nástroj, ktorý bude zavedený do praxe ako učebná pomôcka pre potreby predmetu Sémantika programovacích jazykov pre lepšie pochopenie teórie sémantiky programovacích jazykov pre aritmetické a logické výrazy.

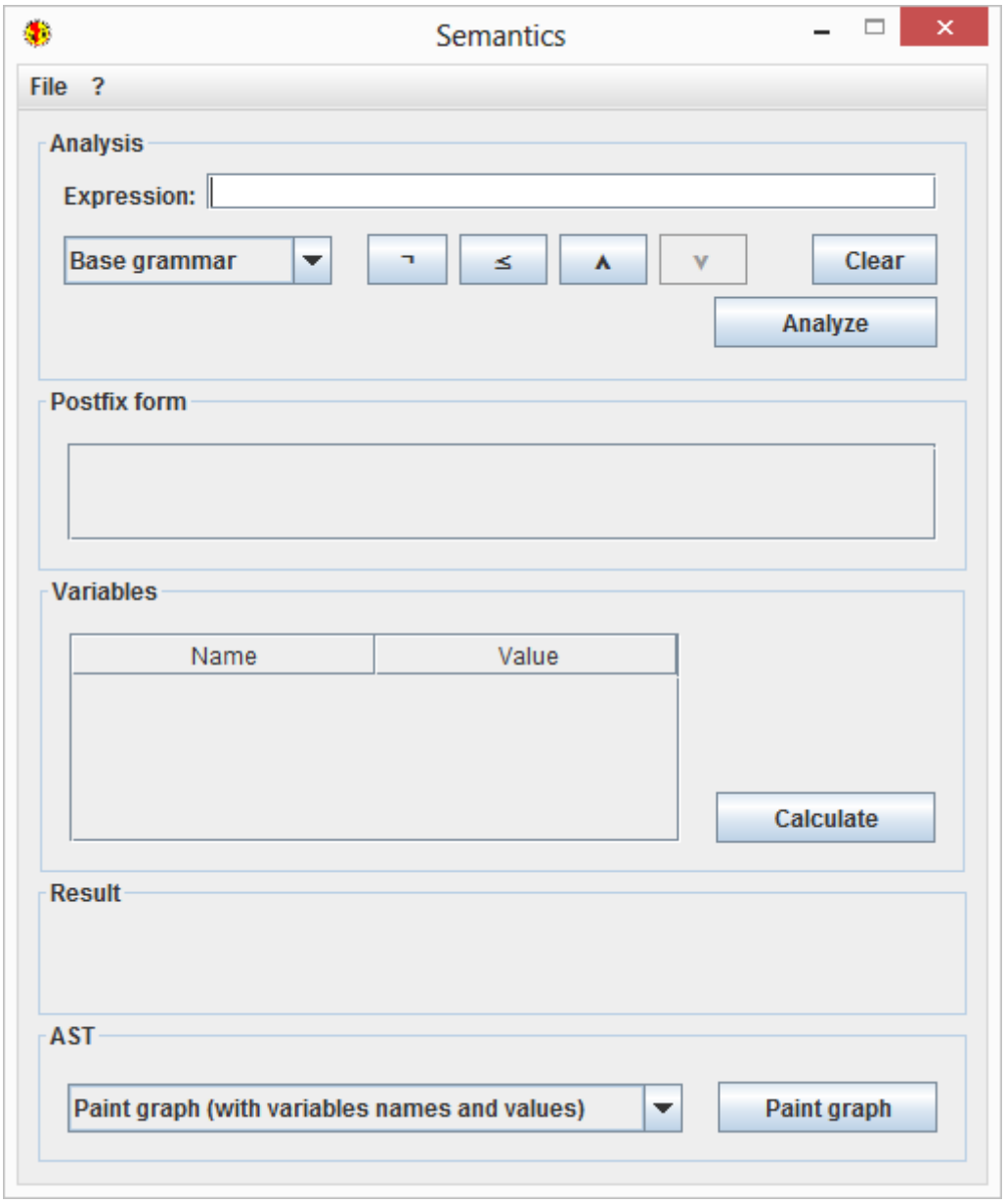
Cieľ sa nám úspešne podarilo naplniť a vytvorená aplikácia spĺňa požadované funkcionality, a tak bude už čoskoro, po dohode s vyučujúcim, zavedená priamo do vyučovacieho procesu.

Okrem bodov zadaných v zadávacom liste diplomovej prace boli v tejto práci vypracované aj úlohy nad rámec zadávacieho listu. V analytickej časti diplomovej prace sme rozšírili základné produkčné pravidlo jednoduchého programovacieho jazyka $\mathscr{J}ane$ pre sémantickú oblasť aritmetických výrazov o binárnu aritmetickú operáciu celočíselného delenia a unárny operátor "+" a "–" (3.4). Taktiež sme rozšírili základné produkčné pravidlo jednoduchého programovacieho jazyka $\mathscr{J}ane$ pre sémantickú oblasť boolovských výrazov o binárnu logickú operáciu disjunkcie " $\vee$ " (3.8). Všetky pridané operátory sme zaviedli aj do gramatiky jazyka (podľa kap. 3.6) a následne implementovali v našej aplikácii. Taktiež bola pridaná funkcionalita na uloženie stromu abstraktnej syntaxe do PDF súboru.

Vyvinutý softvérový nástroj umožňuje pracovať so základnou a rozšírenou gra-

matikou. Používateľ má možnosť zvoliť režim práce prepnutím príslušného prepínača, ako je to uvedené na obrázku 4.6.

Aplikáciu sme dali otestovať niekoľkým študentom a konzultantovi diplomovej práce. Výsledky boli veľmi pozitívne. Používateľské rozhranie aplikácie je veľmi jednoduché a intuitívne (podľa obrazku 5.1 a 4.11). Aj preto sa študenti ihneď v aplikácií zorientovali a pochopili jej funkcionality, ktoré si samozrejme aj úspešne vyskúšali.



Obr. 5.1: Používateľské rozhranie aplikácie

Popis jednotlivých komponentov používateľského rozhrania aplikácie je uvedený v kapitole 4.1.

Pre implementáciu aplikácie bol použitý platformovo nezávislý programovací jazyk Java, čo zabezpečuje funkčnosť našej aplikácie na každom z existujúcich operačných systémov. Vďaka tomu priaznivci akéhokoľvek prostredia nebudú ukrátení.

Pre implementáciu uloženia stromu abstraktnej syntaxe do PDF súboru sme použili vektorový systém, čo dáva používateľovi veľkú výhodu, pretože má možnosť vygenerované obrázky použiť nielen pri študovaní predmetu Sémantika programovacích jazykov, ale aj pri tlači plagátov a aj pri výskumných prácach.

Ako môžeme vidieť, softvérový nástroj pre potreby predmetu Sémantika programovacích jazykov je pre používateľa veľmi jednoduchý, užitočný a pritom obsahuje všetky potrebné funkcionality pre podporu vo výučbe. Veríme, že naša aplikácia sa stane medzi študentmi a učiteľmi obľúbenou a prispeje k zdokonaleniu vzdelávacieho procesu.

6 Záver

Táto diplomová práca sa zaoberá časťou z oblasti predmetu Sémantika programovacích jazykov na teoretickej aj praktickej úrovni.

Teoretická časť práce je popísaná v analytickej časti, ktorá sa zaoberá formálnou definíciou jednoduchého programovacieho jazyka Jane, sémantikou aritmetických a boolovských výrazov. Ďalšiu podkapitolu sme venovali prehľadu existujúcich riešení a ich porovnaniu. Taktiež sme sa v tejto časti diplomovej práce venovali lexikálnej a syntaktickej analýze vstupného výrazu, definícii gramatiky jazyka, postfixnej notácii a algoritmu shunting-yard pre prepis infixného tvaru výrazu na postfixný. Takisto v analytickej časti práce sa zaoberáme aj prístupmi vyhodnotenia vstupného výrazu.

Po získaní bázy potrebných vedomostí sme sa venovali praktickej časti, ktorá je popísaná v syntetickej časti diplomovej práce.

Výsledkom syntetickej časti diplomovej práce je softvérový nástroj pre potreby predmetu Sémantika programovacích jazykov, ktorý dokáže skontrolovať syntaktickú správnosť akéhokoľvek zložitého vstupného aritmetického alebo boolovského výrazu, zobraziť používateľovi jeho postfixnú formu, výsledok vyhodnotenia a strom abstraktnej syntaxe. Okrem toho bola implementovaná doplnková funkcionalita aplikácie - možnosť uloženia stromu abstraktnej syntaxe do PDF súboru na základe vektorovej grafiky. Pre vývoj aplikácie bol použitý platformovo nezávislý programovací jazyk Java, čo zabezpečuje funkčnosť nášho softvérového nástroja na akomkoľvek z existujúcich operačných systémoch.

Vykonávanie pozostáva z dvoch základných funkcionalít. Primárna funkcionalita je vyhodnotenie zadaného vstupného aritmetického alebo boolovského výrazu a jeho zobrazenie vo forme stromu abstraktnej syntaxe. Sekundárna funkcionalita je zobrazenie vstupného výrazu v postfixnej forme a uloženie stromu abstraktnej syntaxe vstupného výrazu do PDF súboru, ktorý je vygenerovaný na

základe vektorovej grafiky, a preto je pripravený pre akékoľvek ďalšie použitie. Používateľské rozhranie aplikácie je pre používateľa intuitívne jednoduché a pochopiteľné.

K tejto aplikácii by bolo vhodné ako rozšírenie pridať online dodatok "Učiteľ-Študent", pomocou ktorého by učiteľ mohol zadať úlohu študentovi, napríklad boolovský výraz, a študent v grafickom prostredí by mal možnosť nakresliť zadanie a odovzdať ho.

Literatúra

[1] William Steingartner a Valerie Novitzká. *Sémantika programovacích jazykov*. 1. vyd. Technická univerzita v Košiciach, 2015. ISBN: 978-80-553-1951-3.

[2] Glynn Winskel. *The formal semantics of programming languages : an introduction*. Massachusetts Institute of Technology, 1993. ISBN: 0-262-23169-7.

[3] Tobias Nipkow a Gerwin Klein. *Concrete Semantics*. Okt. 2017. URL: `http://www.concrete-semantics.org/concrete-semantics.pdf`.

[4] Tomáš Baran. "Emulátor abstraktného stroja pre predmet Sémantika programovacích jazykov". Dipl. pr. Technická univerzita v Košiciach, 2014.

[5] Chris Johnson. *Shunting yard algorithm demo*. 2008. URL: `http://www.chris-j.co.uk/parsing.php`.

[6] Ota Pavelek. "Hlasem ovládaná kalkulačka". Bakalárska práca. Vysoké učení technicé v Brně, 2011.

[7] Wolfram. *Wolfram Mathematica*. 2017. URL: `https://www.wolfram.com/mathematica/`.

[8] Д.И. Соломатин, А.В. Копытин и А.И. Другалев. *Основы синтаксического разбора, построение синтаксических анализаторов*. Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования Воронежский Государственный Университет. Учебно-методическое пособие для вузов. Воронеж, Российская федерация, 2014.

[9] Ján Kollár. *Prekladače*. 1. vyd. Košice, Slovenská republika: Elfa s.r.o., 2009. ISBN: 978-80-8086-121-6.

[10] Ľudovít Molnár, Milan Češka a Bořivoj Melichar. *Gramatiky a jazyky*. 1. vyd. Bratislava: Alfa – Vydavateľstvo technickej a ekonomickej literatúry, 1987.

[11] Michal Chytil. *Automaty a gramatiky*. Praha: SNLT – Nakladatelství technické literatury, 1984.

[12] Peter Sestoft. "Grammars and parsing with Java". Department of Mathematics and Physics Royal Veterinary and Agricultural University, Denmark. Jan. 1999.

[13] Сергей Александрович Орлов. *Теория и практика языков программирования: Учебник для вузов. Стандарт 3-го поколения*. 1-е изд. Санкт-Петербург, Российская федерация: ООО Издательтсво "Питер", 2013. ISBN: 978-5-496-00032-1.

[14] Роберт Лафоре. *Структуры данных и алгоритмы в Java*. 2-е изд. Санкт-Петербург, Российская федерация: ООО Издательтсво "Питер", 2013. ISBN: 978-5-496-00740-5.

[15] Васильев А.Н. *Java. Объектно-ориентированное програмирование: Учебное пособие*. ООО Издательтсво "Питер", 2011. ISBN: 978-5-49807-948-6.

[16] Пышкин Евгений Валерьевич. *Структуры данных и алгоритмы: реализация на C/C++*. Санкт-Петербург, Российская федерация: Центр оперативной полиграции факультета технической кибернетики СПбГПУ, авг. 2009.

[17] Niklaus Wirth. *Algoritmy a štruktúry údajov*. 2. vyd. Bratislava: Alfa, sept. 1989. ISBN: 80-50-00153-3.

[18] Piotr Wróblewski. *Algoritmy*. 1. vyd. Brno: Computer Press, 2015. ISBN: 978-80-251-4126-7.

[19] Theodore Norvell. *Parsing Expressions by Recursive Descent*. 1999. URL: `http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm`.

[20] Patrik Rusnák. "Knižnica pre manipuláciu s logickými funkciami". Dipl. pr. Žilinská univerzita v Žiline, 2017.

[21] Герберт Шилдт. *Java. Полное руководство*. 8-е изд. Москва: Издательский дом Вильямс, 2012. ISBN: 978-5-8459-1759-1.

[22] Брюс Эккель. *Философия Java*. 4-е изд. ООО Издательтсво Питер, 2015. ISBN: 978-5-496-01127-3.

[23] Gilles Dowek. *Principles of Programming Languages*. Springer-Verlag London Limited, 2009. ISBN: 978-1-84882-031-9.

[24] *JUNG 2.0 Tutorial*. Apr. 2009.

[25] Mark Donszelmann, Tony Johnson, Cal Loomis et al. *FreeHEP PDF Driver*. Jún 2007. URL: `https://mvnrepository.com/artifact/org.freehep/freehep-graphicsio-pdf/2.1.1`.

Zoznam príloh

Príloha A CD médium – záverečná práca v elektronickej podobe

Príloha B Používateľská príručka

Príloha C Systémová príručka

A CD médium – záverečná práca v elektronickej podobe

B Používateľská príručka

B.1 Funkcia programu

Tento softvérový nástroj je vytvorený ako učebná pomôcka pre študentov predmetu Sémantika programovacích jazykov pre lepšie pochopenie teórie sémantiky programovacích jazykov pre aritmetické a logické výrazy. Aplikácia dokáže skontrolovať syntaktickú správnosť akéhokoľvek zložitého vstupného aritmetického alebo boolovského výrazu, zobraziť jeho postfixnú formu, výsledok vyhodnotenia a strom abstraktnej syntaxe. Okrem toho aplikácia obsahuje doplnkovú funkcionalitu - možnosť uloženia stromu abstraktnej syntaxe do PDF súboru vo forme vektorovej grafiky.

B.2 Súpis obsahu dodávky

Program je dodávaný na CD médiu, ktoré obsahuje nasledujúce adresáre:

- adresár /doc/
  - diplomová práca vo formáte PDF a TEX, súčasťou ktorej sú:
    - Používateľská príručka softvérového nástroja;
    - Systémová príručka softvérového nástroja;
- adresár /exe/
  - spustiteľný súbor softvérového nástroja vo formáte Executable JAR;
- adresár /src/
  - zdrojové kódy softvérového nástroja;

- adresár /lib/
  - knižnice pre správne kompilovanie zdrojových kódov softvérového nástroja.

B.3 Inštalácia a spustenie aplikácie

Aplikácia bola vytvorená pomocou objektovo-orientovaného, platformovo nezávislého programovacieho jazyka Java, a skompilovaná do jedného súboru typu Executable JAR. Takýto druh súborov je prenositeľný (angl. portable) a je možné ho priamo spúšťať bez akejkoľvek inštalácie na akomkoľvek zariadení, ktoré samozrejme obsahuje virtuálny stroj Java s minimálnou verziou 1.8.

B.3.1 Požiadavky na technické vybavenie

Požiadavky na technické vybavenie závisia iba od minimálnych systémových požiadaviek pre inštaláciu virtuálneho stroja Java verzii 1.8. Preto sem uvedieme iba minimálnu potrebnú konfiguráciu potrebnú pre spustenie aplikácie.

Minimálna konfigurácia

- procesor Intel Pentium II 266 MHz,
- 128 MB operačnej pamäte,
- 132 MB volného miesta na pevnom disku z ktorých:
  - 124 MB pre virtuálny stroj Java,
  - 2 MB pre obnovenie virtuálného stroja Java,
  - 6 MB pre softvérový nástroj pre potreby predmetu Sémantika programovacích jazykov.

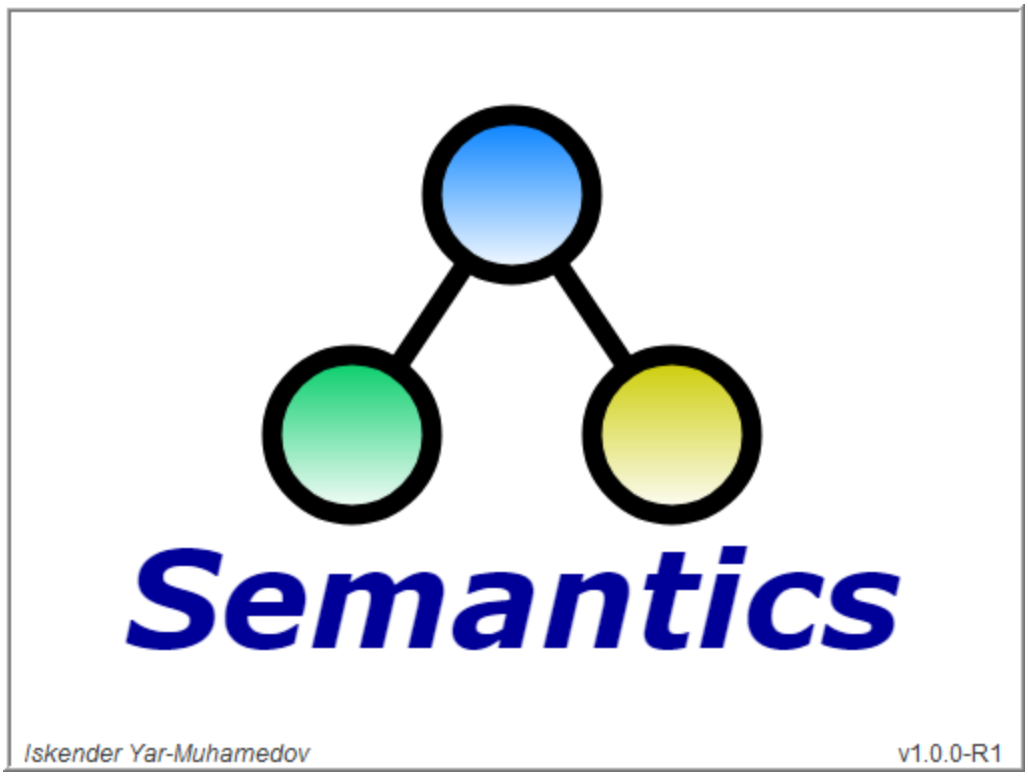
B.3.2 Požiadavky na programové vybavenie

Pre úspešné spustenie softvérového nástroja pre potreby predmetu Sémantika programovacích jazykov je potrebný nainštalovaný virtuálny stroj Java verzia 1.8 alebo vyššia na jednom z nasledujúcich operačných systémov:

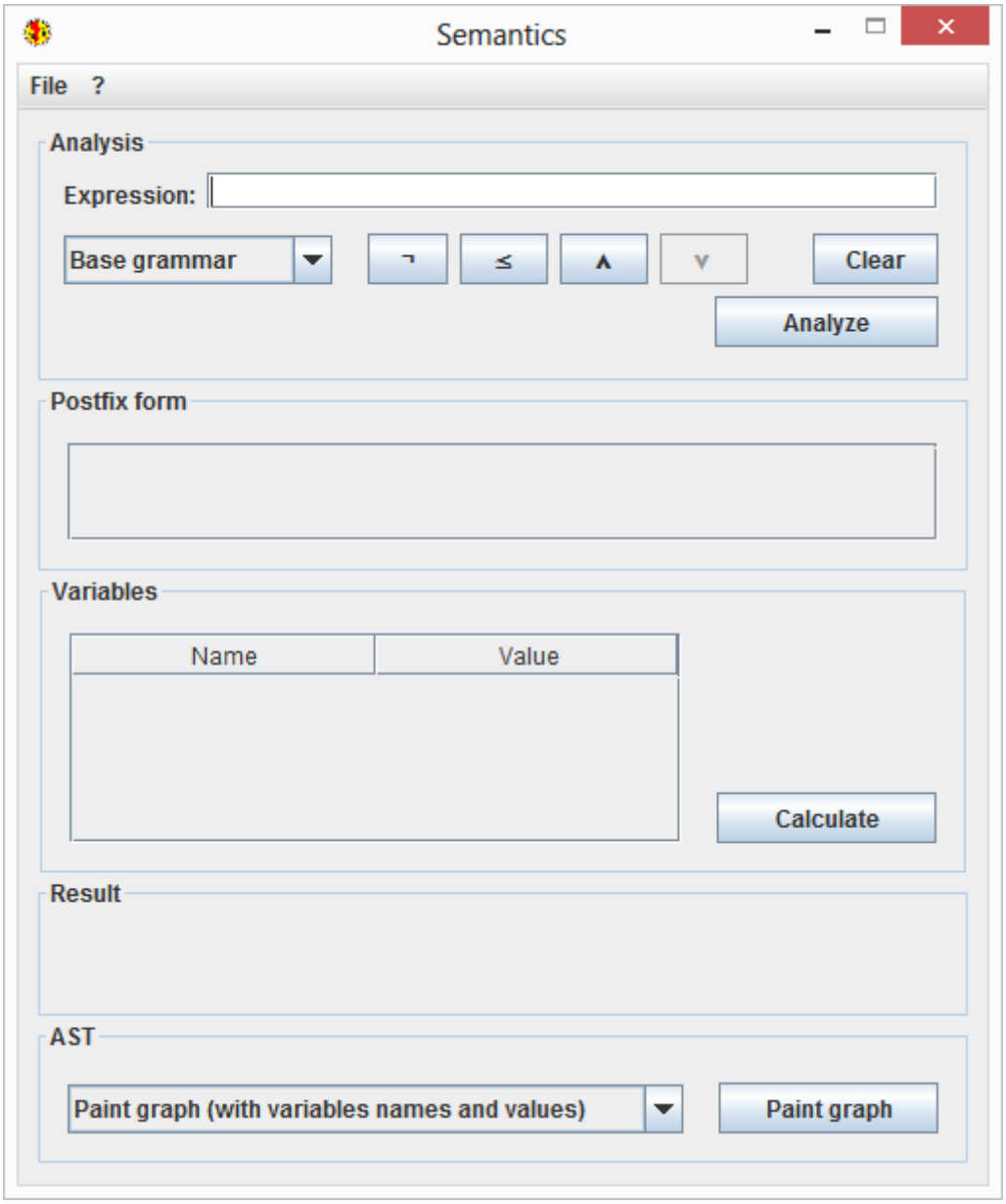
- Solaris, verzia 10 Update 9+ alebo vyššia,
- Windows verzia XP SP3 alebo vyššia (ale od 8. apríla 2014 už neexistuje oficiálna podpora pre Windows XP),
- Oracle Linux verzia 5.5+ alebo vyššia,
- Red Hat Enterprise Linux verzia 5.5+ alebo vyššia,
- Suse Linux Enterprise Server verzia 10 SP2+ alebo vyššia,
- Ubuntu Linux verzia 12.04 – LTS alebo vyššia,
- Ubuntu Linux (Hard-Float ABI) verzia 12.04 – LTS alebo vyššia,
- OS X verzia 10.8.3+ alebo vyššia.

B.4 Popis použitia aplikácie

Po spustení aplikácie sa na par sekúnd zobrazí splash-screen okno B.1 a následné hlavné okno programu B.2.



Obr. B.1: Screen-splash okno



Obr. B.2: Hlavné okno programu

Dizajn aplikácie a rozmiestnenie jednotlivých komponentov je veľmi jednoduché a intuitívne, aby používateľ ľahko pochopil funkcionalitu programu.

Aplikácia obsahuje tieto základné skupiny komponentov:

- **Hlavné menu**, ktoré obsahuje nasledujúce komponenty:
  - **File** obsahuje základnú operáciu **Exit**;

  - **?** obsahuje základnú informáciu o aplikácii **About**;

- **Analyze** pre zadanie vstupného výrazu, ktorá obsahuje nasledujúce komponenty:
  - **Expression** je textové pole, do ktorého sa primárne zadávajú aritmetické alebo boolovské výrazy pre spracovanie;
  - **Grammar** je prepínač, ktorý umožňuje výber druhu gramatiky;
  - $\boxed{\neg}$, $\boxed{\leq}$, $\boxed{\wedge}$, $\boxed{\vee}$ sú tlačidlá na pridanie jednotlivých operátorov vo výraze;
  - **Clear** je tlačidlo pre vymazanie vstupného poľa;
  - **Analyze** je tlačidlo na spustenie lexikálnej a syntaktickej analýzy a prepis vstupného výrazu do postfixnej formy;
- **Postfix form** je textové pole na výpis postfixnej formy vstupného výrazu;
- **Variables** pre zadanie hodnôt premenných, ktorá obsahuje nasledujúce komponenty:
  - **Table** je tabuľka pre interaktívny vstup hodnôt premenných, ktoré sa vyskytujú vo vstupnom výraze;
  - **Calculate** je tlačidlo na vyhodnotenie vstupného výrazu;
- **Result** je textové pole na výpis výsledku vyhodnotenia vstupného výrazu;
- **AST** pre zadanie parametrov kreslenia stromu abstraktnej syntaxe, ktorá obsahuje nasledujúce komponenty:
  - **Graph** je prepínač, ktorý umožňuje výber druhu vykreslenia stromu abstraktnej syntaxe;
  - **Paint graph** je tlačidlo na vykreslenie stromu abstraktnej syntaxe;

B.4.1 Vyhodnotenie vstupného výrazu

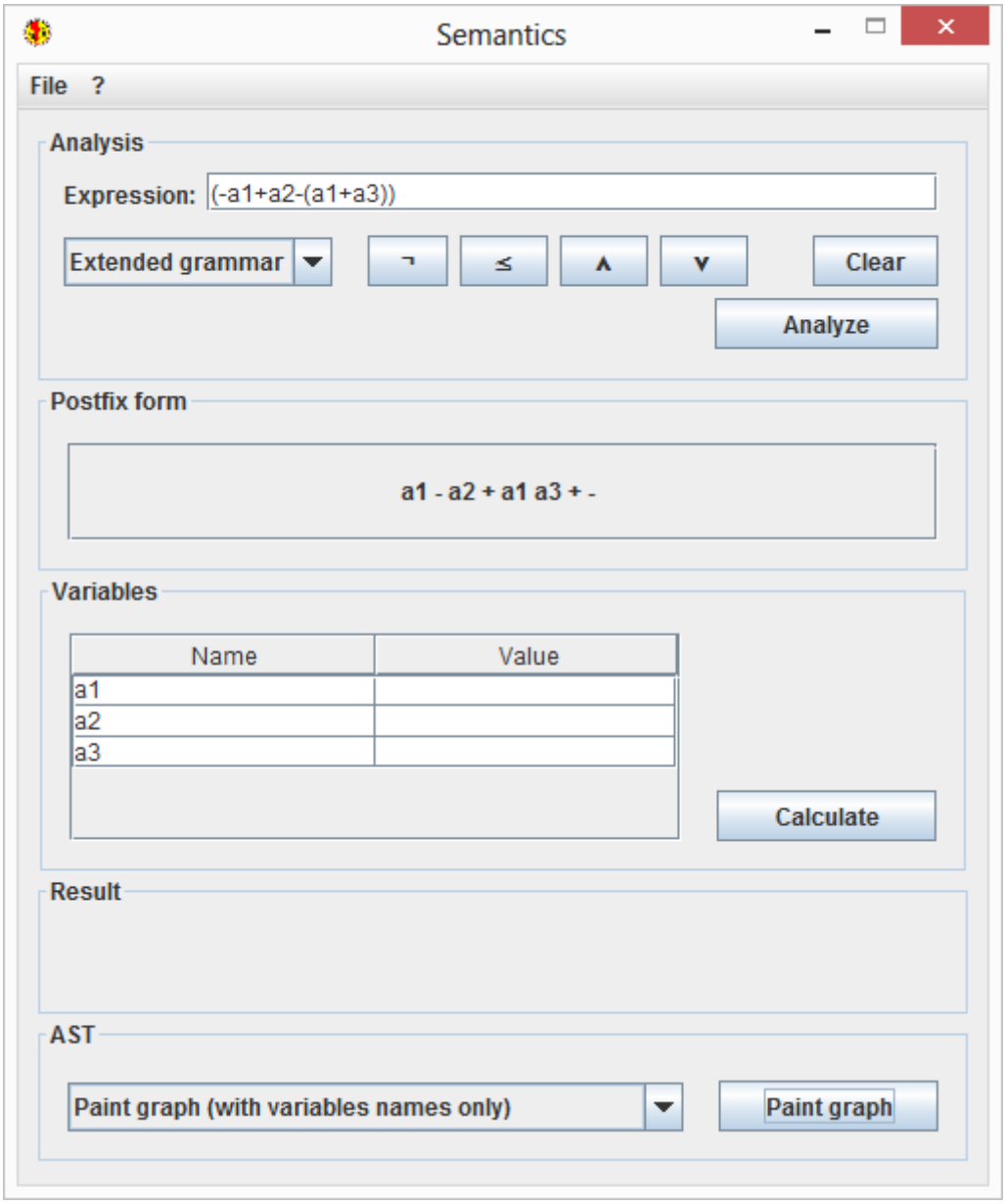
Po spustení aplikácie je možné do textového poľa **Expression** zapísať vstupný aritmetický alebo boolovský výraz v infixnom tvare. Používateľ ma možnosť zvoliť typ gramatiky s ktorej chce pracovať, pomocou príslušného prepínača. Ak sa používateľ rozhodne pridať do vstupného výrazu operátory, tlačidla ktorých neexistujú na počítačových klávesniciach, na to je možné použiť príslušné tlačidlá aplikácie:

- $\neg$ - pridá na príslušnú pozíciu kurzora unárny operátor negácie;
- $\leq$ - pridá na príslušnú pozíciu kurzora binárny operátor neostrej nerovnosti aritmetických výrazov;
- $\wedge$ - pridá na príslušnú pozíciu kurzora binárny operátor konjunkcie;
- $\vee$ - pridá na príslušnú pozíciu kurzora binárny operátor disjunkcie.

Okrem toho používateľ má možnosť vložiť celý výraz alebo jeho časť do textového pola **Expression** zo schránky použitím klávesovej skratky "Crtl+V".

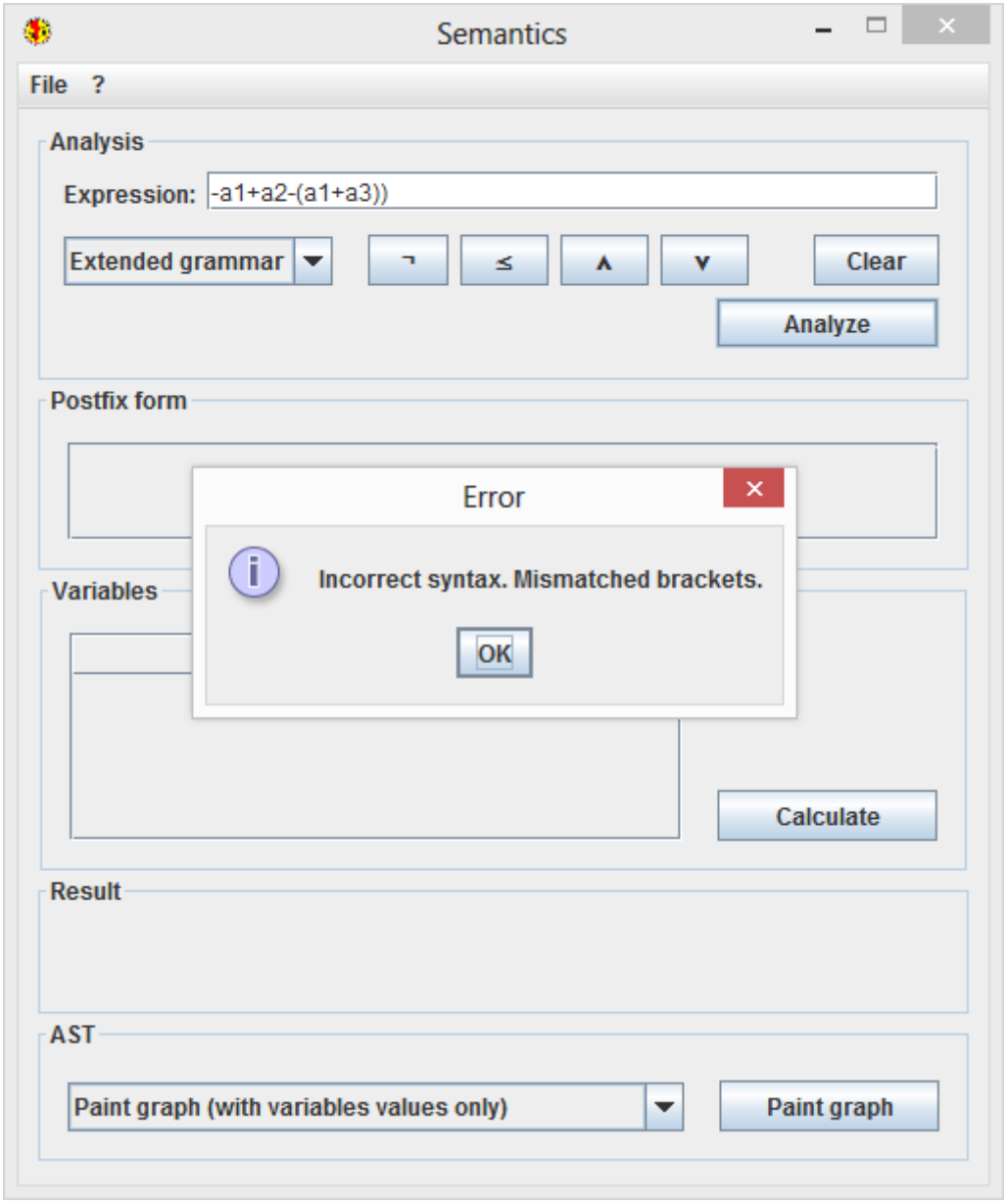
V prípade, že sa používateľ rozhodne zmazať zapísaný vstupný výraz, môže na to použiť tlačidlo **Clear**, ktoré vymaže vstup.

Po stlačení tlačidla **Analyze** dostane používateľ výstup vo forme postfixnej notácie a zobrazí sa mu zoznam všetkých vyskytujúcich sa premenných, ako je to uvedené na obrázku B.3.



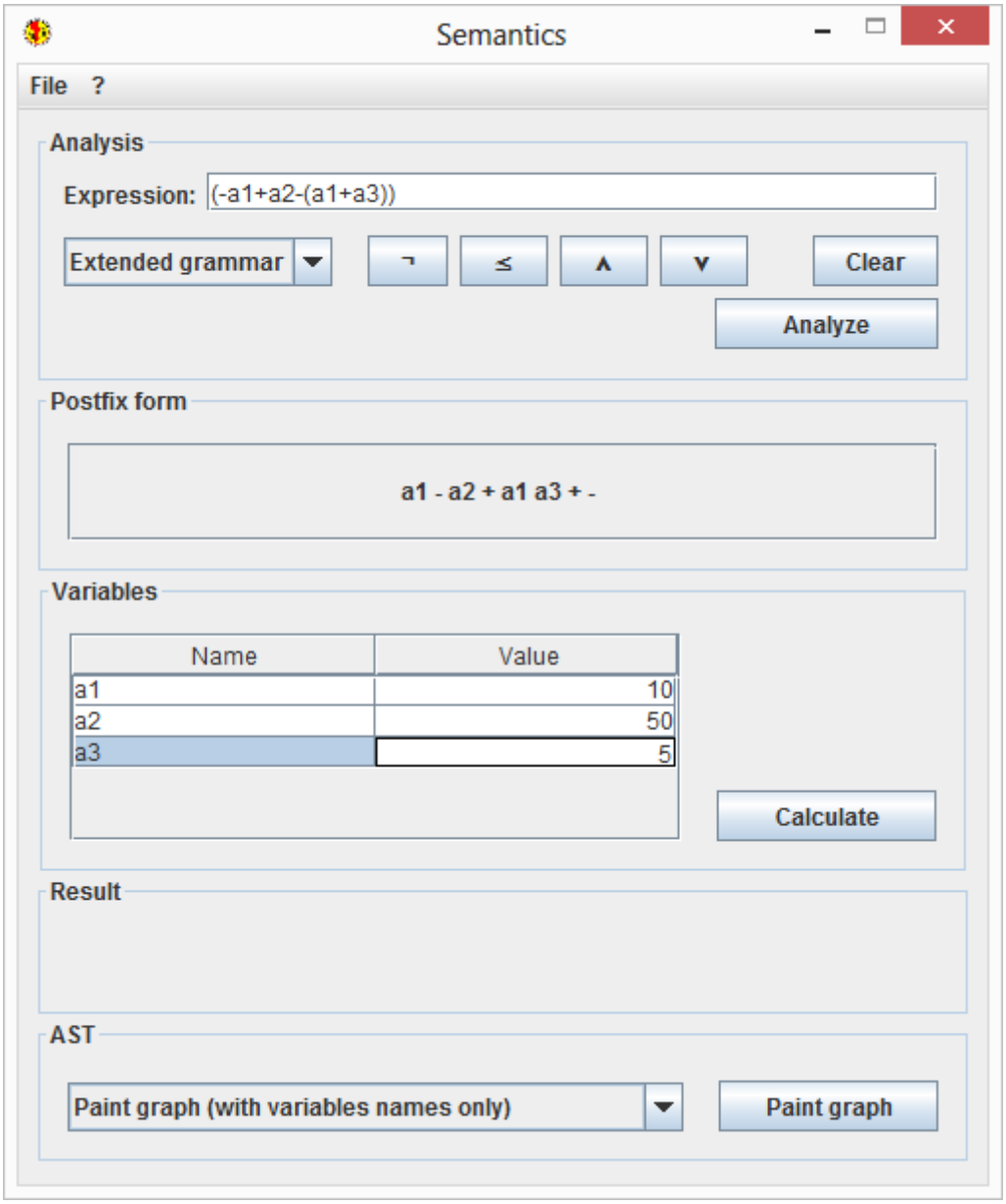
Obr. B.3: Príklad spracovania analyzátora

V prípade zadaného chybného alebo syntakticky nesprávneho výrazu, program sa zastaví a oznámi používateľovi správu o chybe, ktorú používateľ má možnosť opraviť. Príklad zotavenia programu počas výskytu chyby je uvedený na obrázku B.4.



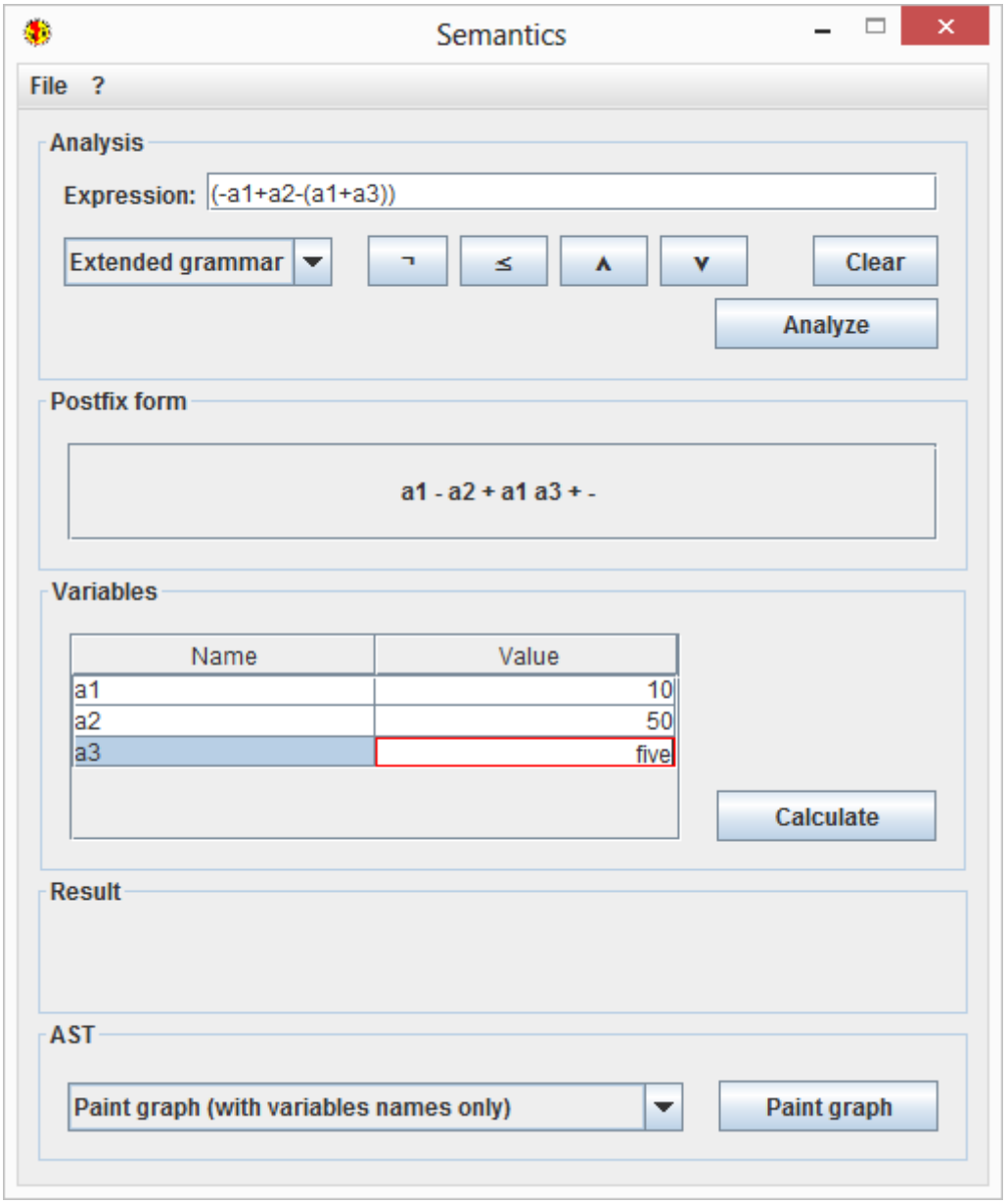
Obr. B.4: Zotavenie programu počas výskytu chyby

Nasledujúcim krokom je vloženie hodnôt premenných. Pre vloženie hodnoty premennej je potrebné dvakrát myšou kliknúť na bunku v stĺpci **Value**, nachádzajúcu sa pri danej premennej. Po dvojkliku bude bunka aktívna a umožní zadanie číselnej hodnoty pomocou klávesnice, ako je uvedené na obrázku B.5.



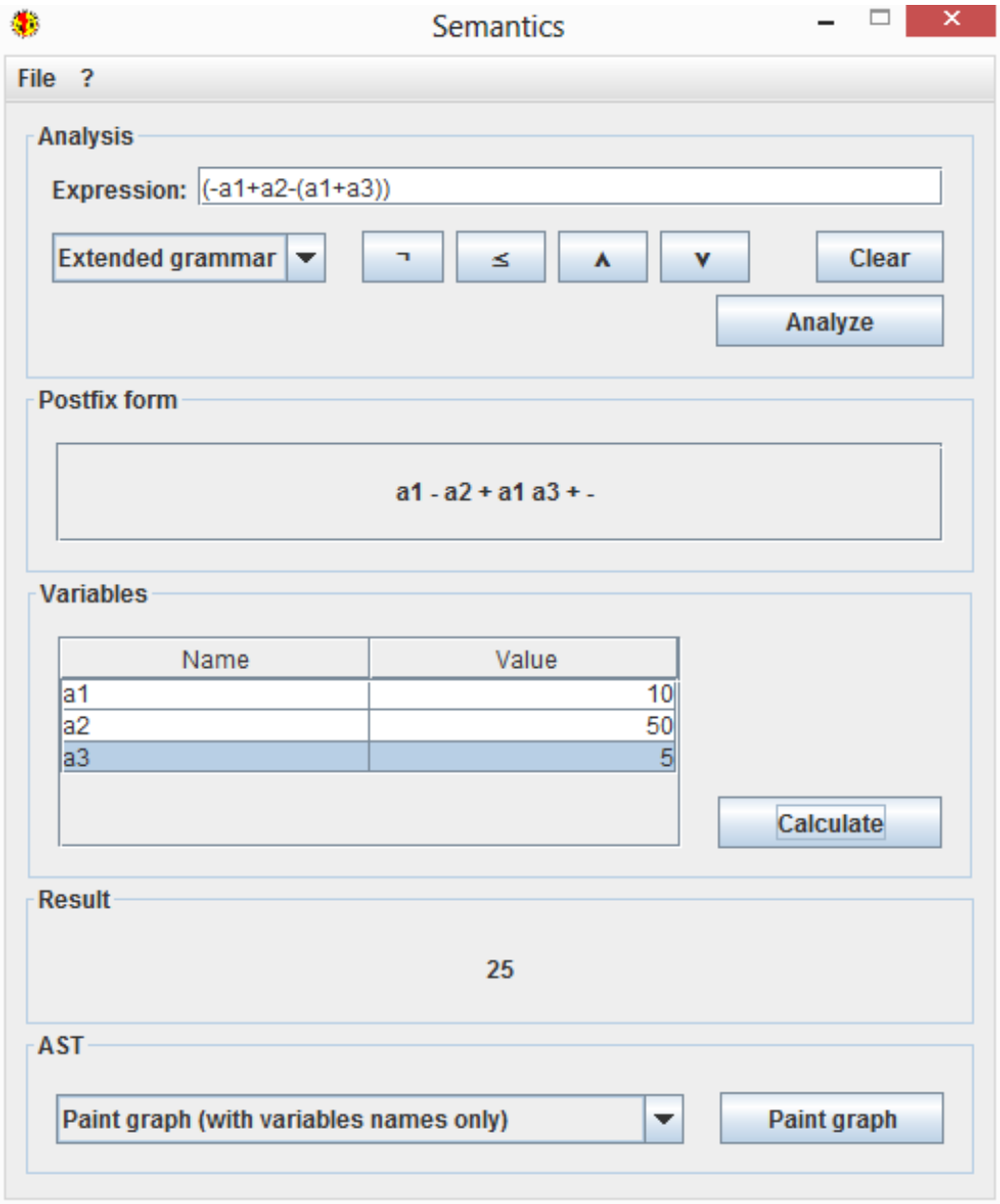
Obr. B.5: Vkladanie hodnôt premenných

Pri pokuse vložiť nejakú inú hodnotu okrem celočíselného typu program označí príslušnú bunku červeným okrajom a poskytne používateľovi možnosť ju opraviť, ako je uvedené na obrázku B.6.



Obr. B.6: Chyba počas vkladania hodnôt premenných

Tlačidlo **Calculate** obsahuje jednu z primárnych funkcionalít aplikácie. Po jeho stlačení aplikácia vyhodnotí vstupný výraz a vypíše výsledok v textovom pole **Result**, ako je uvedené na obrázku B.7.



Obr. B.7: Vyhodnotenie vstupného výrazu

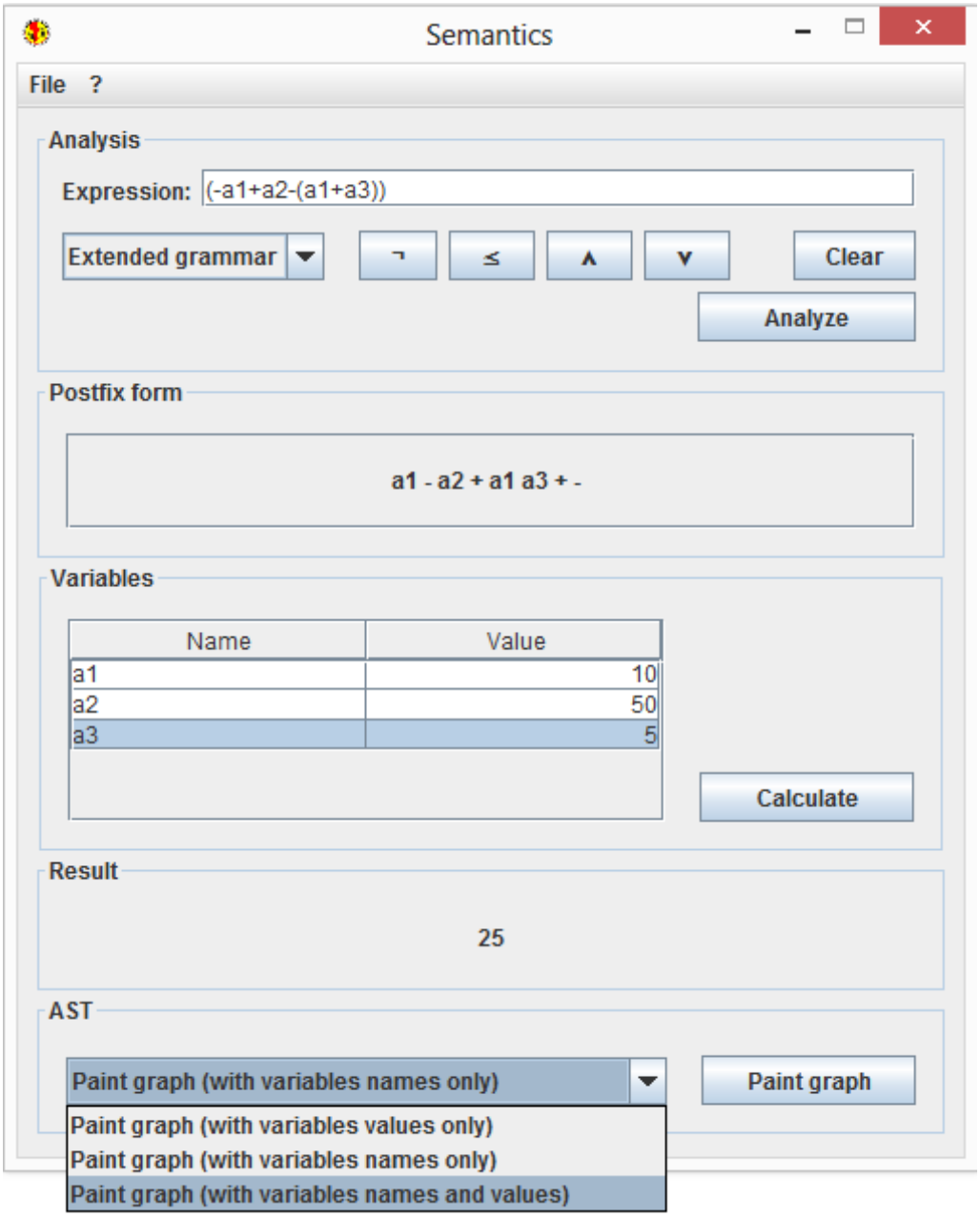
B.4.2 Kreslenie stromu abstraktnej syntaxe

Druhou z primárnych funkcionalít aplikácie je kreslenie stromu abstraktnej syntaxe. Tá poskytuje používateľovi grafickú reprezentáciu vstupného výrazu vo forme stromu abstraktnej syntaxe. Tato grafická reprezentácia poskytuje používateľovi lepšie a hlbšie pochopenie samotnej teórie sémantiky programovacích jazykov pre aritmetické a logické výrazy.

Vyvinutá aplikácia poskytuje používateľovi možnosť výberu typu kreslenia listov stromu:

- iba hodnoty premenných,
- iba mena premenných,
- hodnoty aj mená premenných,

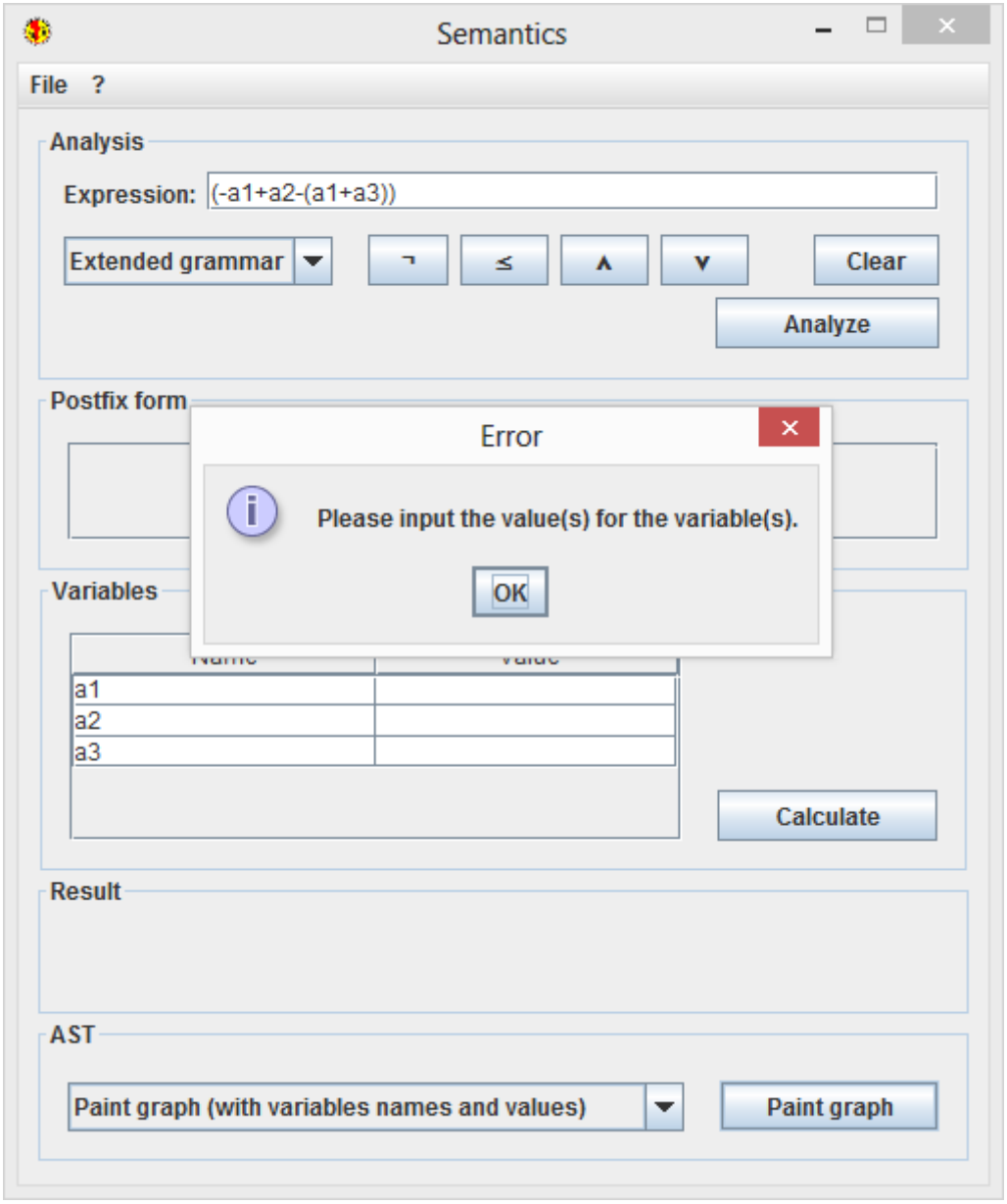
ako je to uvedené na obrázku B.8



Obr. B.8: Výber typu kreslenia listov stromu

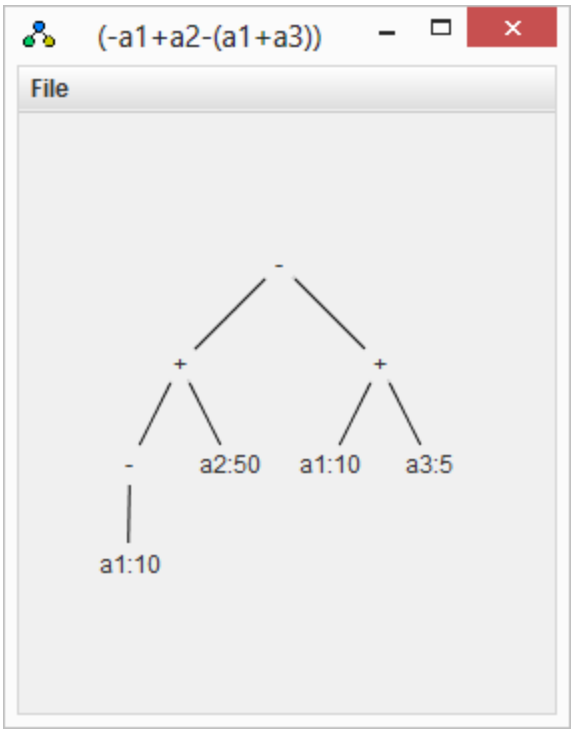
V prípade, že si používateľ zvolil položku “iba hodnoty premenných” alebo “hodnoty a mená premenných”, tak pred samotným vykreslením je potrebné vložiť hodnoty premenných, čo nie je povinné pri kreslení stromu abstraktnej syntaxe, v ktorom listy stromu budú obsahovať iba mená premenných. Ak premenným nebudú priradené hodnoty, program sa zastaví a upozorní používateľa o nutnosti vloženia hodnôt premenných pred kreslením stromu abstraktnej syntaxe,

ako je uvedené na obrázku B.9.



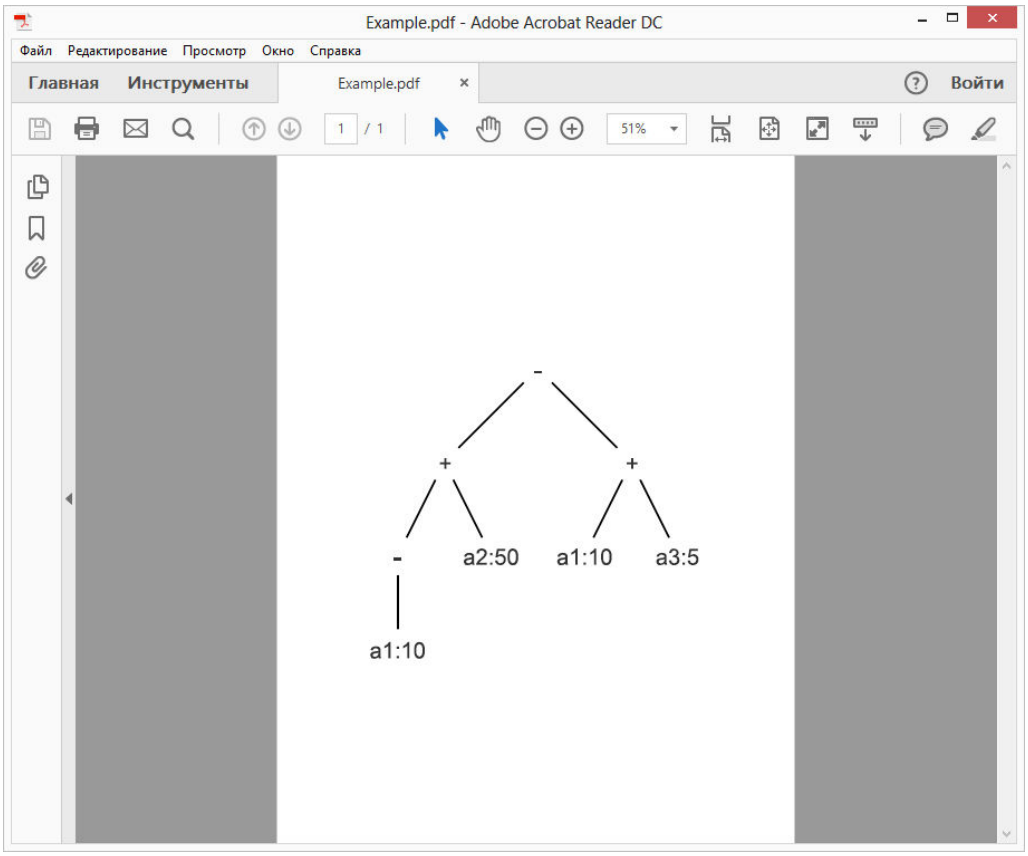
Obr. B.9: Zotavenie aplikácie pri pokuse kreslenia stromu abstraktnej syntaxe bez priradenia hodnôt premenných

Samotná grafická reprezentácia vstupného výrazu vo forme stromu abstraktnej syntaxe je uvedená na obrázku B.10. Používateľ aplikácie má samozrejme možnosť uložiť do súboru vygenerovaný obrázok. Pre uloženie je potrebné vybrať položku **Save** v menu **File** a zadať názov nového súboru.



Obr. B.10: Strom abstraktnej syntaxe zodpovedajúci výrazu $(-a1 + a2 - (a1 + a3))$

Pri uložení obrázka do PDF súboru sa používa vektorový systém, čo dáva používateľovi možnosť použiť tento obrázok aj pre tlač veľkých plagátov, aj pri budúcich výskumných prácach. Vygenerovaný súbor je možné otvoriť v akejkoľvek aplikácii určenej na čítanie a editáciu PDF súborov, napríklad Adobe Acrobar Reader B.11.



Obr. B.11: Otvorenie PDF súboru v aplikácie Adobe Acrobar Reader

Používateľ má možnosť hocikedy ukončiť prácu aplikácie stlačením kombinácie kláves “Alt + F4” (platforma Windows) alebo voľbou položky **Exit** v menu **File**, alebo kliknutím na krížik v pravom hornom rohu okna aplikácie.

C Systémová príručka

C.1 Systémové požiadavky a spustenie projektu

Aplikácia je implementovaná v programovacom jazyku Java. Java je objektovo orientovaný jazyk vyššej úrovne, platformovo nezávislý, ktorého zdrojové programy sa nekompilujú do strojového kódu ale do medzistupňa, tzv. „byte-code“, ktorý nie je závislý od konkrétnej platformy. Systémovými požiadavkami sa v tomto prípade myslí to, že v počítači je nutné mať nainštalovanú správnu verziu JDK 1.8 (angl. Java Development Kit) alebo vyššiu a akéhokoľvek vývojového prostredia (napr. Eclipse od spoločnosti Eclipse Foundation, Inc. alebo IntelliJ IDEA od spoločnosti JetBrains). Samotná inštalácia prebieha tak, že stačí skopírovať samotný projekt z CD média a importovať knižnice z priečinka ***lib*** CD média.

C.2 Dokumentácia Java

Systémová dokumentácia je súčasťou diplomovej práce a zároveň prílohy A, teda CD média. Táto dokumentácia je generovaná vývojovom prostredí Eclipse od spoločnosti Eclipse Foundation, Inc. Dokumentácia obsahuje opis všetkých tried projektu s popisom verejných metód a atribútov.

C.2.1 Trieda SemanticsException

Trieda ***SemanticsException*** sa používa pre vytvorenie okna z popisom chyby. Táto trieda dedí od triedy ***Exception***, ktorá je súčasťou balíka ***java.lang***.

exceptions

Class SemanticsException

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            exceptions.SemanticsException

All Implemented Interfaces:

`java.io.Serializable`

```
public class SemanticsException
extends java.lang.Exception
```

Uses to create a window with an error description.

Version:

`1.0.0-R1`

Author:

`Iskender Yar-Muhamedov iskender.eu@gmail.com`

See Also:

`Serialized Form`

Constructor Summary

Constructors

Constructor and Description
`SemanticsException(java.lang.String message)`

Method Summary

Methods inherited from class java.lang.Throwable

`addSuppressed, fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, getSuppressed, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString`

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

SemanticsException

```
public SemanticsException(java.lang.String message)
```

Parameters:

`message - error description.`

C.2.2 Trieda Analyzer

Trieda ***Analyzer*** sa používa pre lexikálnu a syntaktickú analýzu vstupného výrazu. Táto trieda dedí od triedy ***Object***, ktorá je súčasťou balíka ***java.lang***.

main

Class Analyzer

java.lang.Object
    main.Analyzer

```
public class Analyzer
extends java.lang.Object
```

Uses for the lexical and syntax analyze.

Version:

`1.0.0-R1`

Author:

`Iskender Yar-Muhamedov iskender.eu@gmail.com`

Constructor Summary

Constructors

Constructor and Description
`Analyzer(int isExtendedGrammar, java.lang.String expression)`

Method Summary

All Methods	Instance Methods	Concrete Methods

Modifier and Type	Method and Description
`java.util.LinkedList<main.Token>`	`getPostfix()` The getter for the postfix field.

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Analyzer

```
public Analyzer(int isExtendedGrammar,
                java.lang.String expression)
         throws exceptions.SemanticsException
```

Parameters:

`isExtendedGrammar - a flag that indicates the type of grammar used.`

`expression - input expression.`

Throws:

`exceptions.SemanticsException`

Method Detail

getPostfix

```
public java.util.LinkedList<main.Token> getPostfix()
```

The getter for the postfix field.

Returns:

`Returns the LinkedList of Token type in postfix form.`

C.2.3 Trieda Edge

Trieda ***Edge*** sa používa pre grafickú vizualizáciu grafov. Táto trieda dedí od triedy ***Object***, ktorá je súčasťou balíka ***java.lang***.

main

Class Edge

java.lang.Object
    main.Edge

```
public class Edge
extends java.lang.Object
```

Uses for the visualization.

Version:

1.0.0-R1

Author:

Iskender Yar-Muhamedov iskender.eu@gmail.com

Constructor Summary

Constructors

Constructor and Description
`Edge(java.lang.String name)`

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`java.lang.String`	`getName()` The getter for the name field.

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Edge

```
public Edge(java.lang.String name)
```

Parameters:

```
name - name of the edge.
```

Method Detail

getName

```
public java.lang.String getName()
```

The getter for the name field.

Returns:

```
Returns the name of name field.
```

C.2.4 Trieda Node

Trieda ***Node*** je rekurzívna údajová štruktúra, ktorá sa používa pre reprezentáciu stromu. Táto trieda dedí od triedy ***Object***, ktorá je súčasťou balíka ***java.lang***.

main

Class Node

java.lang.Object
    main.Node

```
public class Node
extends java.lang.Object
```

Recursive data structure of tree.

Version:

```
1.0.0-R1
```

Author:

```
Iskender Yar-Muhamedov iskender.eu@gmail.com
```

Constructor Summary

Constructors

Constructor and Description
`Node(main.Token token)`

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`boolean`	`getbValue()` The getter of bValue field.
`Node`	`getLeft()` The getter of left field (left node).
`java.lang.String`	`getName()` The getter of name field.
`Node`	`getRight()` The getter of right field (right node).
`main.Node.NodeType`	`getType()` The getter of type field.
`int`	`getValue()` The getter of value field.
`java.lang.String`	`getValueForGraph(int mode)` The method which used for node visualization.
`void`	`setbValue(boolean bValue)` The setter of bValue field.
`void`	`setLeft(Node left)` The setter of left field (left node).
`void`	`setName(java.lang.String name)` The setter of name field.
`void`	`setRight(Node right)` The setter of right field (right node).
`void`	`setType(main.Node.NodeType type)` The setter of type field.
`void`	`setValue(int value)` The setter of value field.
`java.lang.String`	`toString()`

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

Node

```
public Node(main.Token token)
     throws exceptions.SemanticsException
```

Parameters:

`token - type of token`

Throws:

`exceptions.SemanticsException`

Method Detail

getType

```
public main.Node.NodeType getType()
```

The getter of type field.

Returns:

`Returns type field.`

setType

```
public void setType(main.Node.NodeType type)
```

The setter of type field.

Parameters:

`type - type of Node.`

getName

```
public java.lang.String getName()
```

The getter of name field.

Returns:

`Returns name field.`

setName

```
public void setName(java.lang.String name)
```

The setter of name field.

Parameters:

```
name - name of Node.
```

getValue

```
public int getValue()
```

The getter of value field.

Returns:

```
Returns value field.
```

setValue

```
public void setValue(int value)
```

The setter of value field.

Parameters:

```
value - value of Node.
```

getbValue

```
public boolean getbValue()
```

The getter of bValue field.

Returns:

```
Returns bValue field.
```

setbValue

```
public void setbValue(boolean bValue)
```

The setter of bValue field.

Parameters:

```
bValue - value of Node.
```

getLeft

```
public Node getLeft()
```

The getter of left field (left node).

Returns:

```
Returns left field (left node).
```

setLeft

```
public void setLeft(Node left)
```

The setter of left field (left node).

Parameters:

```
left - left Node.
```

getRight

```
public Node getRight()
```

The getter of right field (right node).

Returns:

```
Returns right field (right node).
```

setRight

```
public void setRight(Node right)
```

The setter of right field (right node).

Parameters:

```
left - right Node.
```

toString

```
public java.lang.String toString()
```

Overrides:

```
toString in class java.lang.Object
```

getValueForGraph

```
public java.lang.String getValueForGraph(int mode)
                                  throws exceptions.SemanticsException
```

The method which used for node visualization.

Parameters:

`mode - mode of visualization. 0 - Node contain variables values only, 1 - Node contain variables names only, 2 - Node contain variables names and values.`

Returns:

`Returns string for node visualization.`

Throws:

`exceptions.SemanticsException`

C.2.5 Trieda StartApp

StartApp je trieda, ktorá spúšťa aplikáciu. Táto trieda dedí od triedy ***Object***, ktorá je súčasťou balíka ***java.lang***.

main

Class StartApp

java.lang.Object
    main.StartApp

```
public class StartApp
extends java.lang.Object
```

Class with launch configuration.

Version:

`1.0.0-R1`

Author:

`Iskender Yar-Muhamedov iskender.eu@gmail.com`

Constructor Summary

Constructors

Constructor and Description
`StartApp()`

Method Summary

All Methods	Static Methods	Concrete Methods

Modifier and Type	Method and Description
`static void`	`main(java.lang.String[] args)` Launch the application.

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

StartApp

```
public StartApp()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Launch the application.

C.2.6 Trieda TableModel

Trieda ***TableModel*** je používaná komponentom JTable, ktorý reprezentuje mená a hodnoty premenných. Táto trieda dedí od triedy ***AbstractTableModel***, ktorá je súčasťou balíka ***javax.swing.table***.

main

Class TableModel

java.lang.Object
  javax.swing.table.AbstractTableModel
    main.TableModel

All Implemented Interfaces:

`java.io.Serializable, javax.swing.table.TableModel`

```
public class TableModel
extends javax.swing.table.AbstractTableModel
```

A JTable component uses this table model for represent names and values of variables.

Version:

1.0.0-R1

Author:

Iskender Yar-Muhamedov iskender.eu@gmail.com

See Also:

Serialized Form

Constructor Summary

Constructors

Constructor and Description
TableModel(java.lang.Object[][] data)

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
java.lang.Class	getColumnClass(int c)
int	getColumnCount()
java.lang.String	getColumnName(int col)
int	getRowCount()
java.lang.Object	getValueAt(int row, int col)
boolean	isCellEditable(int row, int column)
void	setValueAt(java.lang.Object value, int row, int col)

Methods inherited from class javax.swing.table.AbstractTableModel

addTableModelListener, findColumn, fireTableCellUpdated, fireTableChanged, fireTableDataChanged, fireTableRowsDeleted, fireTableRowsInserted, fireTableRowsUpdated, fireTableStructureChanged, getListeners, getTableModelListeners, removeTableModelListener

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

TableModel

```
public TableModel(java.lang.Object[][] data)
```

Parameters:

data - an array of variables.

Method Detail

getColumnCount

```
public int getColumnCount()
```

getRowCount

```
public int getRowCount()
```

getColumnName

```
public java.lang.String getColumnName(int col)
```

Specified by:

getColumnName in interface javax.swing.table.TableModel

Overrides:

getColumnName in class javax.swing.table.AbstractTableModel

getValueAt

```
public java.lang.Object getValueAt(int row,
                                   int col)
```

getColumnClass

```
public java.lang.Class getColumnClass(int c)
```

Specified by:

getColumnClass in interface javax.swing.table.TableModel

Overrides:

getColumnClass in class javax.swing.table.AbstractTableModel

isCellEditable

```
public boolean isCellEditable(int row,
                              int column)
```

Specified by:

`isCellEditable in interface javax.swing.table.TableModel`

Overrides:

`isCellEditable in class javax.swing.table.AbstractTableModel`

setValueAt

```
public void setValueAt(java.lang.Object value,
                       int row,
                       int col)
```

Specified by:

`setValueAt in interface javax.swing.table.TableModel`

Overrides:

`setValueAt in class javax.swing.table.AbstractTableModel`

C.2.7 Trieda Token

Trieda ***Token*** reprezentuje lexému. Táto trieda dedí od triedy ***Object***, ktorá je súčasťou balíka ***java.lang***.

main

Class Token

java.lang.Object
    main.Token

```
public class Token
extends java.lang.Object
```

Data structure of Token.

Version:

`1.0.0-R1`

Author:

`Iskender Yar-Muhamedov iskender.eu@gmail.com`

Constructor Summary

Constructors

Constructor and Description
`Token(boolean bool)`
`Token(char operator, main.Token.OperatorType operatorType, int precedence)`
`Token(int value)`
`Token(java.lang.String str)`
`Token(java.lang.String name, int value)`
`Token(main.Token.TokenType tokenType)`
`Token(Token token)`

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`java.lang.Boolean`	`getbValue()` The getter of bValue field.
`java.lang.String`	`getName()` The getter of name field.
`char`	`getOperator()` The getter of operator field.
`main.Token.OperatorType`	`getOperatorType()` The getter of operatorType field.
`int`	`getPrecidence()` The getter of precedence field.
`main.Token.TokenType`	`getTokenType()` The getter of tokenType field.
`int`	`getValue()` The getter of value field.
`void`	`setbValue(java.lang.Boolean bValue)` The setter of bValue field.
`void`	`setName(java.lang.String name)` The setter of name field.
`void`	`setOperator(char operator)` The setter of operator field.

void	`setOperatorType(main.Token.OperatorType operatorType)` The setter of operatorType field.
void	`setPrecidence(int precedence)` The setter of precedence field.
void	`setTokenType(main.Token.TokenType tokenType)` The setter of tokenType field.
void	`setValue(int value)` The setter of value field.
java.lang.String	`toString()`

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

Token

```
public Token(Token token)
```

Parameters:

`token - token`

Token

```
public Token(char operator,
             main.Token.OperatorType operatorType,
             int precedence)
```

Parameters:

`operator - operator of token.`

`operatorType - operator type of token.`

`precedence - precedence of token.`

Token

```
public Token(int value)
```

Parameters:

`value - of token.`

Token

```
public Token(java.lang.String name,
             int value)
```

Parameters:

`name - name of Token.`

`value - value of Token.`

Token

```
public Token(java.lang.String str)
```

Parameters:

`str - value of token.`

Token

```
public Token(boolean bool)
```

Parameters:

`bool - value of token.`

Token

```
public Token(main.Token.TokenType tokenType)
```

Parameters:

`tokenType - type of token.`

Method Detail

getOperator

```
public char getOperator()
```

The getter of operator field.

Returns:

`Returns operator field.`

setOperator

```
public void setOperator(char operator)
```

The setter of operator field.

Parameters:

```
operator - operator of token.
```

getValue

```
public int getValue()
```

The getter of value field.

Returns:

```
Returns value field.
```

setValue

```
public void setValue(int value)
```

The setter of value field.

Parameters:

```
value - value of token.
```

getPrecidence

```
public int getPrecidence()
```

The getter of precedence field.

Returns:

```
Returns precedence field.
```

setPrecidence

```
public void setPrecidence(int precedence)
```

The setter of precedence field.

Parameters:

```
precedence - precedence of token.
```

getName

```
public java.lang.String getName()
```

The getter of name field.

Returns:

```
Returns name field.
```

setName

```
public void setName(java.lang.String name)
```

The setter of name field.

Parameters:

```
name - name of token.
```

getbValue

```
public java.lang.Boolean getbValue()
```

The getter of bValue field.

Returns:

```
Returns bValue field.
```

setbValue

```
public void setbValue(java.lang.Boolean bValue)
```

The setter of bValue field.

Parameters:

```
bValue - value of token.
```

getOperatorType

```
public main.Token.OperatorType getOperatorType()
```

The getter of operatorType field.

Returns:

```
Returns operatorType field.
```

setOperatorType

```
public void setOperatorType(main.Token.OperatorType operatorType)
```

The setter of operatorType field.

Parameters:

```
operatorType - operator type of token.
```

getTokenType

```
public main.Token.TokenType getTokenType()
```

The getter of tokenType field.

Returns:

```
Returns tokenType field.
```

setTokenType

```
public void setTokenType(main.Token.TokenType tokenType)
```

The setter of tokenType field.

Parameters:

```
tokenType - type of token.
```

toString

```
public java.lang.String toString()
```

Overrides:

```
toString in class java.lang.Object
```

C.2.8 Trieda Utils

Trieda ***Utils*** obsahuje statické metódy pre transformáciu výrazu z infixnej formy na postfixnú formu, vyhodnotenie výrazu, vytvorenie stromovej štruktúry, zadanie hodnôt premenných a získanie zoznamy premenných. Táto trieda dedí od triedy ***Object***, ktorá je súčasťou balíka ***java.lang***.

main

Class Utils

java.lang.Object
    main.Utils

```
public class Utils
extends java.lang.Object
```

Contains the static methods to transform an expression in infix form to postfix form, to calculate an expression, to build tree structure, to set variables values and to get list of variables.

Version:

`1.0.0-R1`

Author:

`Iskender Yar-Muhamedov iskender.eu@gmail.com`

Constructor Summary

Constructors

Constructor and Description
`Utils()`

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`static java.lang.Object`	`calculate(java.util.LinkedList<main.Token> postfix)` The method for evaluating an expression, which was transformed to postfix form.
`static edu.uci.ics.jung.graph.Forest<main.Node,main.Edge>`	`createTree(main.Node node)` The method creates recursive data structure of Forest type.
`static java.lang.Object[][]`	`getAllVariables(java.util.LinkedList<main.Token> linkedList)` The method for finding a names of all existing variables.
`static main.Node`	`getTree(java.util.LinkedList<main.Token> postfix)` The method creates recursive data structure of Node type.
`static java.util.LinkedList<main.Token>`	`setVariablesValues(java.util.LinkedList<main.Token> linkedList, main.TableModel model)` The method assigns values to variables.
`static java.util.LinkedList<main.Token>`	`transformToPostfix(java.util.LinkedList<main.Token> infix)` The method to transform a LinkedList of Token type in infix form to the LinkedList of postfix form.

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Utils

```
public Utils()
```

Method Detail

transformToPostfix

```
public static java.util.LinkedList<main.Token> transformToPostfix(java.util.LinkedList<main.Token> infix)
                                                            throws exceptions.SemanticsException
```

The method to transform a LinkedList of Token type in infix form to the LinkedList of postfix form.

Parameters:

`infix - LinkedList of Token type in infix form.`

Returns:

`Returns the LinkedList of Token type in postfix form.`

Throws:

`exceptions.SemanticsException`

calculate

```
public static java.lang.Object calculate(java.util.LinkedList<main.Token> postfix)
                                  throws exceptions.SemanticsException
```

The method for evaluating an expression, which was transformed to postfix form.

Parameters:

`postfix - LinkedList of Token type in postfix form.`

Returns:

`Returns the result of the calculations in Object type.`

Throws:

`exceptions.SemanticsException`

getTree

```
public static main.Node getTree(java.util.LinkedList<main.Token> postfix)
                         throws exceptions.SemanticsException
```

The method creates recursive data structure of Node type.

Parameters:

`postfix - LinkedList of Token type in postfix form.`

Returns:

`Returns the recursive data structure of Node type.`

Throws:

`exceptions.SemanticsException`

createTree

```
public static edu.uci.ics.jung.graph.Forest<main.Node,main.Edge> createTree(main.Node node)
```

The method creates recursive data structure of Forest type.

Parameters:

`node - recursive data structure of Node type.`

Returns:

`Returns the recursive data structure of Forest type.`

Throws:

`exceptions.SemanticsException`

setVariablesValues

```
public static java.util.LinkedList<main.Token> setVariablesValues(java.util.LinkedList<main.Token> linkedList,
                                                                  main.TableModel model)
                                                           throws exceptions.SemanticsException
```

The method assigns values to variables.

Parameters:

`linkedList - LinkedList of Token type.`

`model - table model of TableModel type.`

Returns:

`Returns the LinkedList of Token type in postfix form. Values were assigned to variables.`

Throws:

`exceptions.SemanticsException`

getAllVariables

```
public static java.lang.Object[][] getAllVariables(java.util.LinkedList<main.Token> linkedList)
```

The method for finding a names of all existing variables.

Parameters:

`linkedList - LinkedList of Token type.`

Returns:

`Returns the two-dimensional array. In the first contains names of variables.`

C.2.9 Trieda GraphWindow

Trieda ***GraphWindow*** reprezentuje okno pre grafickú vizualizáciu grafu. Táto trieda dedí od triedy ***JFrame***, ktorá je súčasťou balíka ***javax.swing***.

windows

Class GraphWindow

java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
          javax.swing.JFrame
            windows.GraphWindow

All Implemented Interfaces:

`java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable,`
`javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants`

```
public class GraphWindow
extends javax.swing.JFrame
```

The window for graph visualization.

Version:

`1.0.0-R1`

Author:

`Iskender Yar-Muhamedov iskender.eu@gmail.com`

See Also:

`Serialized Form`

Nested Class Summary

Nested classes/interfaces inherited from class java.awt.Window

`java.awt.Window.Type`

Nested classes/interfaces inherited from class java.awt.Component

`java.awt.Component.BaselineResizeBehavior`

Field Summary

Fields inherited from class javax.swing.JFrame

`EXIT_ON_CLOSE`

Fields inherited from class java.awt.Frame

`CROSSHAIR_CURSOR, DEFAULT_CURSOR, E_RESIZE_CURSOR, HAND_CURSOR, ICONIFIED, MAXIMIZED_BOTH, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MOVE_CURSOR, N_RESIZE_CURSOR, NE_RESIZE_CURSOR, NORMAL, NW_RESIZE_CURSOR, S_RESIZE_CURSOR, SE_RESIZE_CURSOR, SW_RESIZE_CURSOR, TEXT_CURSOR, W_RESIZE_CURSOR, WAIT_CURSOR`

Fields inherited from class java.awt.Component

`BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT`

Fields inherited from interface javax.swing.WindowConstants

`DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE, HIDE_ON_CLOSE`

Fields inherited from interface java.awt.image.ImageObserver

`ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH`

Constructor Summary

Constructors

Constructor and Description
`GraphWindow(java.lang.String title, edu.uci.ics.jung.graph.Forest<main.Node,main.Edge> forest, int mode)`

Method Summary

Methods inherited from class javax.swing.JFrame

`getAccessibleContext, getContentPane, getDefaultCloseOperation, getGlassPane, getGraphics, getJMenuBar, getLayeredPane, getRootPane, getTransferHandler, isDefaultLookAndFeelDecorated, remove, repaint, setContentPane, setDefaultCloseOperation, setDefaultLookAndFeelDecorated, setGlassPane, setIconImage, setJMenuBar, setLayeredPane, setLayout, setTransferHandler, update`

Methods inherited from class java.awt.Frame

`addNotify, getCursorType, getExtendedState, getFrames, getIconImage, getMaximizedBounds, getMenuBar, getState, getTitle, isResizable, isUndecorated, remove, removeNotify, setBackground, setCursor, setExtendedState, setMaximizedBounds, setMenuBar, setOpacity, setResizable, setShape, setState, setTitle, setUndecorated`

Methods inherited from class java.awt.Window

`addPropertyChangeListener, addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener, applyResourceBundle, applyResourceBundle, createBufferStrategy, createBufferStrategy, dispose, getBackground, getBufferStrategy, getFocusableWindowState, getFocusCycleRootAncestor, getFocusOwner, getFocusTraversalKeys, getIconImages, getInputContext, getListeners, getLocale, getModalExclusionType, getMostRecentFocusOwner, getOpacity, getOwnedWindows, getOwner, getOwnerlessWindows, getShape, getToolkit, getType, getWarningString, getWindowFocusListeners, getWindowListeners, getWindows, getWindowStateListeners, hide, isActive, isAlwaysOnTop, isAlwaysOnTopSupported, isAutoRequestFocus, isFocusableWindow, isFocusCycleRoot, isFocused, isLocationByPlatform, isOpaque, isShowing, isValidateRoot, pack, paint, postEvent, removeWindowFocusListener, removeWindowListener, removeWindowStateListener, reshape, setAlwaysOnTop, setAutoRequestFocus, setBounds, setBounds, setCursor, setFocusableWindowState, setFocusCycleRoot, setIconImages, setLocation, setLocation, setLocationByPlatform, setLocationRelativeTo, setMinimumSize, setModalExclusionType, setSize, setSize, setType, setVisible, show, toBack, toFront`

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalPolicy, getInsets, getLayout, getMaximumSize, getMinimumSize, getMousePosition, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, preferredSize, print, printComponents, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusTraversalKeys, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setFont, transferFocusDownCycle, validate

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, contains, contains, createImage, createImage, createVolatileImage, createVolatileImage, disable, dispatchEvent, enable, enable, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBaseline, getBaselineResizeBehavior, getBounds, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getGraphicsConfiguration, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocation, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isPreferredSizeSet, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, requestFocus, requestFocusInWindow, resize, resize, revalidate, setComponentOrientation, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale, setMaximumSize, setName, setPreferredSize, show, size, toString, transferFocus, transferFocusBackward, transferFocusUpCycle

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.awt.MenuContainer

getFont, postEvent

Constructor Detail

GraphWindow

```
public GraphWindow(java.lang.String title,
                   edu.uci.ics.jung.graph.Forest<main.Node,main.Edge> forest,
                   int mode)
```

Parameters:

title - window title with inserted expression.

forest - data structure of Forest type.

mode - mode of visualization. 0 - Node contain variables values only, 1 - Node contain variables names only, 2 - Node contain variables names and values.

C.2.10 Trieda MainWindow

Trieda ***MainWindow*** reprezentuje hlavne okno aplikácie. Táto trieda dedí od triedy ***Object***, ktorá je súčasťou balíka ***java.lang***.

windows

Class MainWindow

java.lang.Object
    windows.MainWindow

```
public class MainWindow
extends java.lang.Object
```

The main window of Semantics application. Uses for input expression, calculate expression and to display infix notation of inserted expression.

Version:

1.0.0-R1

Author:

Iskender Yar-Muhamedov iskender.eu@gmail.com

Constructor Summary

Constructors

Constructor and Description
`MainWindow()` Create the application.

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`static void`	`activate()` The static method for setting visibility of the JFrame component.
`static void`	`startApp()` The static method for creation and initialization the MainWindow component.

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

MainWindow

```
public MainWindow()
```

Create the application.

Method Detail

activate

```
public static void activate()
```

The static method for setting visibility of the JFrame component.

startApp

```
public static void startApp()
```

The static method for creation and initialization the MainWindow component.

C.2.11 Trieda SplashWindow

Trieda ***SplashWindow*** reprezentuje splash okno aplikácie. Táto trieda dedí od triedy ***JWindow***, ktorá je súčasťou balíka ***javax.swing***.

windows

Class SplashWindow

java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        javax.swing.JWindow
          windows.SplashWindow

All Implemented Interfaces:

`java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, javax.accessibility.Accessible, javax.swing.RootPaneContainer`

```
public class SplashWindow
extends javax.swing.JWindow
```

The window for Splash screen.

Version:

`1.0.0-R1`

Author:

`Iskender Yar-Muhamedov iskender.eu@gmail.com`

See Also:

`Serialized Form`

Nested Class Summary

Nested classes/interfaces inherited from class java.awt.Window

`java.awt.Window.Type`

Nested classes/interfaces inherited from class java.awt.Component

`java.awt.Component.BaselineResizeBehavior`

Field Summary

Fields inherited from class java.awt.Component

`BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT`

Fields inherited from interface java.awt.image.ImageObserver

`ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH`

Constructor Summary

Constructors

Constructor and Description
`SplashWindow()` Constructor.

Method Summary

Methods inherited from class javax.swing.JWindow

`getAccessibleContext, getContentPane, getGlassPane, getGraphics, getLayeredPane, getRootPane, getTransferHandler, remove, repaint, setContentPane, setGlassPane, setLayeredPane, setLayout, setTransferHandler, update`

Methods inherited from class java.awt.Window

`addNotify, addPropertyChangeListener, addPropertyChangeListener, addWindowFocusListener, addWindowListener, addWindowStateListener, applyResourceBundle, applyResourceBundle, createBufferStrategy, createBufferStrategy, dispose, getBackground, getBufferStrategy, getFocusableWindowState, getFocusCycleRootAncestor, getFocusOwner, getFocusTraversalKeys, getIconImages, getInputContext, getListeners, getLocale, getModalExclusionType, getMostRecentFocusOwner, getOpacity, getOwnedWindows, getOwner, getOwnerlessWindows, getShape, getToolkit, getType, getWarningString, getWindowFocusListeners, getWindowListeners, getWindows, getWindowStateListeners, hide, isActive, isAlwaysOnTop, isAlwaysOnTopSupported, isAutoRequestFocus, isFocusableWindow, isFocusCycleRoot, isFocused, isLocationByPlatform, isOpaque, isShowing, isValidateRoot, pack, paint, postEvent, removeNotify, removeWindowFocusListener, removeWindowListener, removeWindowStateListener, reshape, setAlwaysOnTop, setAutoRequestFocus, setBackground, setBounds, setBounds, setCursor, setFocusableWindowState, setFocusCycleRoot, setIconImage, setIconImages, setLocation, setLocation, setLocationByPlatform, setLocationRelativeTo, setMinimumSize, setModalExclusionType, setOpacity, setShape, setSize, setSize, setType, setVisible, show, toBack, toFront`

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalPolicy, getInsets, getLayout, getMaximumSize, getMinimumSize, getMousePosition, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, preferredSize, print, printComponents, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusTraversalKeys, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setFont, transferFocusDownCycle, validate

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, contains, contains, createImage, createImage, createVolatileImage, createVolatileImage, disable, dispatchEvent, enable, enable, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBaseline, getBaselineResizeBehavior, getBounds, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getGraphicsConfiguration, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocation, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isPreferredSizeSet, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, prepareImage, prepareImage, printAll, remove, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, requestFocus, requestFocusInWindow, resize, resize, revalidate, setComponentOrientation, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale, setMaximumSize, setName, setPreferredSize, show, size, toString, transferFocus, transferFocusBackward, transferFocusUpCycle

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

SplashWindow

```
public SplashWindow()
```

Constructor. Uses to create and initialize Splash window.