Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Formální jazyky a překladače

Tvorba nástroje pro zobrazení a práci se zásobníkovým automatem

Plzeň, 2019 Michal Horký (A18N0087P)

Obsah

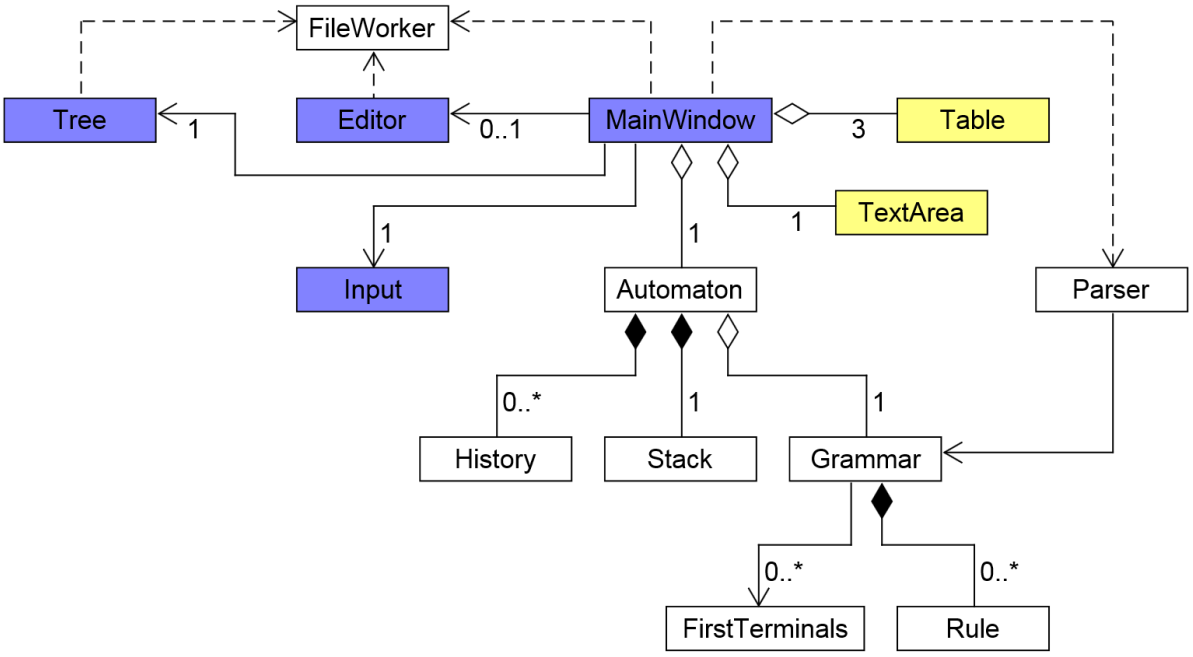
1 Zadání

Cílem práce je vytvořit nástroj, který bude schopen graficky zobrazit zpracování zásobníkového automatu. Automat bude mít na vstupu gramatiku a bude omezen na gramatiky typu `LL(1)`.

Gramatika bude zpracovaná aplikací bez použití speciálních nástrojů. Rozkladová tabulka bude sestavena programem.

Z důvodu jednoduššího vyvíjení a multiplatformního softwaru bude realizace v jazyce `C++` s využitím `Qt` frameworku.

2 Implementace



Obrázek 1 UML diagram tříd (modře okna, žlutě GUI komponenty, čárkované čáry volání statických metod)

2.1 Struktura aplikace, překlad

Hlavním pracovním adresářem projektu je `Visualization`. Důležitým souborem zde je projektový soubor `Visualization.pro`, ve kterém se nastavují využívané balíky, určuje se zde pracovní adresář, umístění hlavičkových a zdrojových souborů, dále se zde linkují knihovny, atd.

Projektový soubor je nepostradatelný pro překlad. Ten se provádí pomocí dvou příkazů v příkazové řádce.

1. `qmake` (vytvoření `makefilů`)
2. `mingw32-make` (`C` překladač pro `Windows`)

Samozřejmě lze projekt vyvíjet ve vývojovém prostředí `Qt Creator`, který se při spuštění automaticky postará o překlad. Nicméně nejde o moc propracované vývojové prostředí, takže osobně radši využívám textové editory, které mi zvýrazní syntaxi.

2.2 Adresářová struktura

- `css` – adresář s kaskádovými styly pro vzhled určitých komponent v aplikaci.
- `data` – ukázky gramatik, `tmp` složka pro vytvoření obrázku derivačního stromu.
- `img` – ikonky aplikace.

- `src` – zdrojové soubory.
- `debug` – složka, která vznikne po překladu a slouží pro debugování.
- `release` – druhá a poslední složka, která vznikne po překladu a obsahuje spouštěcí binární soubor.
- `Visualisation.pro`
- další konfigurační soubory

2.3 Zdrojové soubory

- `gui` – adresář s grafickými moduly, jednotlivé moduly představují buď okna, nebo grafické komponenty.
- `inner` – jedná se o adresář s moduly, které mají na starosti vnitřní práci s daty, ať už jde o držení dat v určitých strukturách nebo jejich úpravu.
- `io` – v tomto adresáři je jeden jediný modul `file_worker`, který obsahuje třídu `FileWorker` s pouze statickými funkcemi starajících se o čtení a zápisy do souborů.
- `visualization.cpp` – hlavní zdrojový soubor se spouštěcí funkcí `main()`.

Ve spouštěcí funkci `main()` se pouze nastaví okenní aplikace a vytvoří instance třídy `MainWindow` představující hlavní okno.

2.4 MainWindow

Zde se vytváří a rozmisťují veškeré grafické komponenty v hlavním okně. Kromě toho jsou zde funkce pro úpravu vzhledu a přístupnosti grafických komponent jakožto reakce na různé události, např.: kdy mají být přístupná jaká tlačítka, co se má stát po zpracování vstupu, při restartu běhu zpracování, apod. Dále se zde inicializují, případně zobrazují další okna, viz `Obrázek 1 UML diagram tříd`. Jinak veškeré signály potomků, které jsou zde odchyceny, jsou mapovány na sloty v této třídě (ani v jednom případě třída neslouží jako prostředník).

Jsou zde reference na všechny důležité prvky, které definují chod celé aplikace:

- Textová oblast pro průběžné výpisy stavů zásobníkového automatu.
- Tabulky pro gramatiku, zásobník a přechody mezi stavy.
- Okno pro nastavení vstupního textu.
- Okno pro zobrazení/uložení derivačního stromu.
- Objekt automatu.

Jen pro úplnost dodám informace k poslednímu oknu, které je možné vytvořit (vytváří se v `MainWindow::editGrammar()`). Jde o instanci třídy `Editor`, což je okno pro úpravu gramatiky. Reference na něj se nikde neuchovává, při vytvoření se nastaví ukončovací atribut, aby při zavření okna došlo k uvolnění paměti. Ve třídě `Editor` je jeden signál `changeGrammar(const QString &)`, který je namapován na

`MainWindow::setGrammar(const QString &)` – tahle funkce zastaví animaci, vymaže všechna data automatu až na zadaný vstupní text a načte nově zadanou gramatiku. Na závěr je uživatel informován o tom, jak se zpracování gramatiky povedlo.

2.4.1 Textová oblast v dolní části hlavního okna

Aneb instance třídy `TextArea`. Sem se vypisují informace o stavu automatu. Nejčastěji jde o výpis, co zbývá zpracovat na vstupu, aktuální stav zásobníku a id pravidel, které byly doposud využity. Ten je emitován v podobě signálu `message(bool, QString)` z `Automaton::printState()` a mapován na `MainWindow::message(bool, QString)`, kde se provede fyzické uložení textu do textové oblasti. Poté jsou zde výpisy o celkovém zpracování, tedy jestli byl řetězec akceptován nebo ne. Tento výpis je přidáván z funkce `MainWindow::processResult(bool)`, která je zavolána při ukončení zpracování vstupu.

2.4.2 Tabulky pro gramatiku, zásobník a přechody mezi stavy

Jde o instance třídy `Table`. Pro rozpoznání o jakou tabulku jde je zde vytvořen „výčtový typ“ v podobě tří proměnných `GRAMMAR_TABLE`, `PARSING_TABLE` a `STACK_TABLE`. V této třídě se pak definuje vzhled dané tabulky, dále tu jsou sloty pro reakce na vyvolané události, konkrétně: změna vybraného řádku (popř. buňky), při restartu zpracování odstranění výběru a slot, který znovu naplní danou tabulku. Dále tu jsou překryty `protected` funkce vyvolávané při jakékoli události s myší kvůli tomu, aby uživatel nemohl klikat do tabulek a kazit tím animaci. Pro přehlednost mají naplnění tabulek konkrétními daty na starosti odpovídající objekty, třeba pro tabulku s gramatikou jde o funkci `Grammar::setGrammarTable(const QTableWidget &)`.

2.4.3 Okno pro nastavení vstupního textu

Vytvoření tohoto okna má na starosti instance třídy `Input`. Je aktivní po celou dobu běhu aplikace. Při zavření se pouze skryje, fyzické zavření a uvolnění paměti nastane až při ukončování aplikace. Je to kvůli tomu, abychom nemuseli někde ukládat, co uživatel naposledy zadal. Může totiž zadat více řádků, ty se po uložení upraví na jednořádkový text, pro který je vizualizace optimalizována.

Ve třídě Input je jeden slot pro reakci na stisk tlačítka „`Uložit`“ a jeden signál `changeInput(const QString &)`, který je namapován na slot `MainWindow::setInput(const QString &)` – vypne a vrátí na začátek probíhající animaci, automatu nastaví vstupní text, který je při animaci vyhodnocován.

2.4.4 Okno pro zobrazení/uložení derivačního stromu

Tahle funkcionalita je ve třídě `Tree`. Má jednu veřejnou funkci `createNewTree(const QString &)`. Vstupním parametrem je stromová struktura v `DOT` jazyce. Funkce se třikrát pokusí o uložení stromové struktury do souboru tree.txt do složky `data/tmp_tree`.

V případě úspěšného uložení se spustí příkaz:
`dot -Tpng -O ../data/tmp_tree/tree.txt`, který vygeneruje obrázek s derivačním stromem ve formátu `png` (to ale pouze v případě, že má uživatel nainstalovanou službu `Graphviz` a správně nastavenou cestu).

Po vykonání příkazu se spustí časovač, který každou vteřinu vyvolá slot (funkci) `Tree::controlPngExists()`. Ta zkontroluje, jestli už byl obrázek vygenerován, pokud ano, zavolá se funkce `Tree::showImage()`, kde se načte pixelová mapa obrázku a nastaví se do labelu `tree`, následně je zobrazeno celé okno. Pokud obrázek ještě vygenerován nebyl, čeká se dál, maximálně se však tento cyklus opakuje desetkrát. Výsledný obrázek je možnost uložit, k tomu je zde tlačítko `save` a slot `Tree::saveClicked()`.

2.4.5 Objekt automatu

Jde o instanci třídy `Automaton`, která má na starosti chod vizualizace. Jsou zde tedy funkce pro přehrávání [`play()`, `pause()`], krokování [`prevStep()`, `nextStep()`], ovládání rychlosti [`changeWaiting(double)`], apod. Dále jsou v tomto objektu struktury, bez kterých se automat neobejde, jde o gramatiku (třída `Grammar`) a o zásobník (třída `Stack`). Je zde i seznam změn v automatu, který se využívá při krokování zpět (`QList<History> changes`).

Pro přehrávání je vytvořen časovač, který po určité době zavolá soukromý slot `Tree::oneStep()` vykonávající jeden krok zpracování vstupu.

1. na počátku se na zásobník musí uložit počáteční symbol,
2. v dalších iteracích je vždy nejdříve odebrán vrchol zásobníku a:
    a. v případě prázdného symbolu se nic nevykoná (prázdný symbol hodnota nula),
    b. jinak se zkusí najít uživatelem zadané slovo v seznamu terminálů,
    c. pokud nebylo zadané slovo nalezeno, nebo se neshoduje s odebraným vrcholem zásobníku, nebo neexistuje přechod v přechodové tabulce – dojde k ukončení zpracování s výsledkem `NEAKCEPTOVÁNO`.
    d. jinak se provádí srovnání, případně expanze.
3. Jestliže je zásobník prázdný, jde o konec zpracování. Záleží, jestli je na vstupu ještě něco. Pokud ne, výsledek je `AKCEPTOVÁNO`.

Při každém kroku se uloží údaj o změně a vypíše se stav do textové oblasti v hlavním okně.

2.5 Grammar

Objekt této třídy představuje bezkontextovou gramatiku. Kromě datových struktur obsahuje třída veškeré funkce pro práci s gramatikou.

2.5.1 Použité datové struktury

Vytvoření vnitřní struktury gramatiky začíná načtením hrubého textu, ve kterém by měla být bezkontextová gramatika. To má na starosti funkce `QString FileWorker::loadFileContent(const QString &)`. Ta ovšem pouze načte text, neprovádí žádné kontroly, apod. Funkce `bool Parser::parseGrammar(Grammar &, const QString &)` nejdříve vymaže údaje ke staré gramatice, poté hrubý text rozdělí a pomocí funkce `Grammar::append(const QString &, const QList<QString> &)` provádí vytváření struktury gramatiky. V případě hrubého porušení formátu (např. chybí dvojtečka) ukončí vytváření gramatiky a provede kompletní smazání dosud zpracovaných dat.

Pro práci s gramatikou jsem se rozhodl pro použití čtyř datových struktur:

- seznam terminálů (`QList<QString> terminals`),
- seznam neterminálů (`QList<QString> nonterminals`),
- seznam přepisovacích pravidel (`QList<Rule> terminals`),
- mapa (`QMap<int, QList<int> > rulesIndices`) pro mapování indexu neterminálu na seznam odpovídajících indexů přepisovacích pravidel.

Uvedený seznam terminálů a neterminálů představuje jejich skutečné texty, většinou se ovšem pracuje pouze s jejich indexy. Kvůli odlišení indexů jsem zavedl pravidlo, že u terminálů je prováděn převod do záporných čísel. Navíc je v obou seznamech zablokovaný nultý index, na něm nic není. Ve většině případů nula (jakožto index terminálu či neterminálu) znamená prázdný symbol – pouze při práci s tabulkou přechodů odpovídá index prázdného symbolu velikosti seznamu terminálů. Je to z toho důvodu, že je prázdný symbol v posledním sloupečku tabulky.

Poslední datová struktura, mapa, je zde kvůli rychlejšímu přístupu k přepisovacím pravidlům. Jeden neterminál může mít více přepisovacích pravidel. Každé přepisovací pravidlo je schováno pod jednou instancí třídy `Rule`. Takže abychom nemuseli při zpracovávání vstupu procházet seznam přepisovacích pravidel a hledat, jaká pravidla patří danému neterminálu, na začátku se vytvoří záznam v mapě, kde jsou, pro konkrétní index neterminálu, uloženy všechny indexy odpovídajících přepisovacích pravidel.

Posledním instančním atributem je tabulka přechodů. Jedná se o dvourozměrné pole celých čísel. Hodnota `-1` je defaultní a znamená, že se z daného neterminálu nelze dostat na požadovaný terminál. Jinak tabulka obsahuje indexy přepisovacích pravidel, která se mají použít.

2.5.2 Důležité funkce pro zpracování

Ve třídě `Grammar` je celkem hodně funkcí. Jsou zde funkce pro převod indexu na pravidlo (a naopak), pro zjištění velikosti seznamů terminálů a neterminálů, pro naplnění tabulek představujících gramatiku a přechodovou tabulku v hlavním okně, pro převod indexů v instanci `Rule` do řetězce kvůli výpisu, pro vytvoření textu v `DOT` jazyce kvůli vykreslení

derivačního stromu, apod. Většinou se ale jedná o pouhé gettry a jinak nejde o nic složitého. Například (pro jednoduchost zde nebudu uvádět vstupní parametry):

- U vytváření řetězce v `DOT` jazyce – jde o dvě funkce `toDotLanguage()` a `toDotLanguageRec()`, kde ta druhá je rekurzivní. První udělá hlavičku souboru a nadefinuje počáteční symbol, druhá pak zpracovává výsledná pravidla, která byla použita při zpracování vstupního řetězce, a přepisuje je do jednotlivých vrcholů výsledného stromu.
- Naplnění tabulky gramatiky a přechodové tabulky je realizováno v `setGrammarTable()` a v `setParsingTable()`. Jsou zavolány hned po vytvoření struktury gramatiky. Vždy se nastaví počet řádků a určí se, co se má uživateli zobrazit a jak – uložené indexy terminálů a neterminálů se musí převést zpět na řetězec, `-1` u přechodové tabulky nesmí být vidět, apod.
- Myslím si, že funkce `toString()` se ani nepoužívá, byla využívána při debugování na počátku vývoje.
- atd.

Mezi nejdůležitější funkce patří vytvoření přechodové tabulky, tedy `first` a `follow` funkce:

- `bool createTable()`
- `QList<FirstTerminals> first(int)`
- `FirstTerminals applyFirstRules(int, int)`
- `QList<int> follow(int, QList<int> &)`

2.5.3 bool Grammar::createTable()

Po načtení hrubého textu, jeho rozparsování a uložení do datových struktur gramatiky, je zavolána funkce `bool Grammar::isCorrect()`, která pouze vrací výsledek funkce `bool createTable()` – tedy jestli je gramatika `LL(1)` či nikoli. Ta vytváří tabulku přechodů. U každého neterminálu je zavolána funkce `QList<FirstTerminals> first(int)`, kde vstupním parametrem je index daného neterminálu. Funkce vrací nalezené první terminály. Jestliže je v seznamu i prázdný symbol, provede se navíc výpočet funkce `QList<int> follow(int nonterminalIndex, QList<int> &)`. Rovnou vrací konkrétní indexy terminálů, které se ovšem mohou opakovat (u funkce `first()` se vrací instance `FirstTerminals`, protože tam potřebujeme znát indexy pravidel, kdežto zde je už máme zjištěné). Jestliže má dojít při vykonávání (po funkci `first` či `follow`) k přepsání jakékoli buňky jiným indexem pravidla, dojde k předčasnému ukončení a výpisu chyby, že nastala určitá kolize a gramatika tedy není `LL(1)`.

2.5.4 QList<FirstTerminals> Grammar::first(int)

Tahle funkce prochází všechna přepisovací pravidla patřící neterminálu na daném indexu a ukládá výsledky funkce `FirstTerminals applyFirstRules(int, int)`.

2.5.5 FirstTerminals Grammar::applyFirstRules(int, int)

Jedná se o rekurzivní funkci, ve které je ovšem jednoduché vyhodnocování. Jestliže je na pravé straně pouze prázdný symbol, uloží se nula a dojde k ukončení. Pokud je jako první terminál postup je obdobný. Jestliže je prvním symbolem neterminál, zavolá se funkce `QList<FirstTerminals> first(int)`, jejíž výsledek se uloží do výsledného seznamu. V posledním případě může akorát nastat výjimka a to ta, když se ve vráceném seznamu najde prázdný symbol a za neterminálem je něco dalšího. To se pak řeší odstraněním prázdného symbolu a rekurzivním zavoláním sebe sama.

2.5.6 QList<int> follow(int, QList<int> &)

Jde o funkci, která hledá následující první terminály. Druhým vstupním parametrem je seznam zakázaných neterminálů, aby nedošlo k zacyklení. Funkce vždy projde všechna přepisovací pravidla, a pokud v pravidle najde daný neterminál, hledá pomocí funkce `FirstTerminals applyFirstRules(int, int)` následující terminály.

1. Jestliže vrácený seznam obsahuje prázdný symbol, vymaže se, zbytek se uloží do výsledných terminálů a provede se zavolání sebe sama akorát s jiným prvním parametrem (konkrétně indexem neterminálu, který patří k danému přepisovacímu pravidlu).
2. Pokud je terminál na konci přepisovacího pravidla, vyplní se údaj v posledním sloupečku tabulky (pro prázdný symbol, konec textu) a provede se to samé jako v předchozím případě, zavolání sebe sama se stejnými parametry jako v předchozím případě.
3. Jinak se pouze provede uložení nalezených terminálů do výsledného seznamu, a buď se pokračuje další iterací cyklu (dalším přepisovacím pravidlem), nebo dojde k ukončení funkce.

2.5.7 Parser

Obsahuje pouze jednu statickou funkci `parseGrammar(Grammar &, const QString &)`, která zajistí, že informace o staré gramatice budou smazány [zavoláním `Grammar::clear()`] a bude vytvořen nový seznam terminálů, neterminálů a přepisovacích pravidel.

2.5.8 Rule

Instance této třídy představují přepisovací pravidla. Obsahují index neterminálu a seznam indexů (terminálů, neterminálů), na které se má daný neterminál přepsat. Obsahuje důležitou funkci pro `first` a `follow` funkce, `QList<int> findNonterminal(int)`. Tahle funkce hledá určitý index neterminálů v přepisovacím pravidle. Vrací, na jakých indexech se hledaný neterminál nachází.

2.5.9 FirstTerminals

Instance jsou využívány při vyhledávání prvních terminálů. Slouží k uchování informací o nalezených prvních terminálech pro konkrétní přepisovací pravidlo. Třída obsahuje funkce pro nalezení a odstranění prázdného symbolu, což je potřebné u `first` a `follow` funkcí, `bool containsEmptySymbol` a `removeEmptySymbol()`.

2.6 Stack

Instance třídy `Stack` představuje zásobník celých čísel (viz význam hodnot), který je využíván při zpracování zásobníkového automatu. Obsahuje klasické funkce, např.: `push(int)`, `pop()`, apod. Funkce `append(Rule)` zkopíruje postupně všechny indexy z daného pravidla do zásobníku.

Důležitou funkcí je `setStackTable(QTableWidget &, Grammar &)`, která naplní tabulku představující zásobník. Funkce `toString(Grammar &)` je pak využívána při vypisování aktuálního stavu automatu.

Jakmile je obsah zásobníku změněn, je vyvolán signál `changed()`, který způsobí obnovu vzhledu tabulky.

2.6.1 Význam hodnot

- rovno nule => prázdný symbol,
- vetší než nula => neterminál,
- menší než nula => terminál.

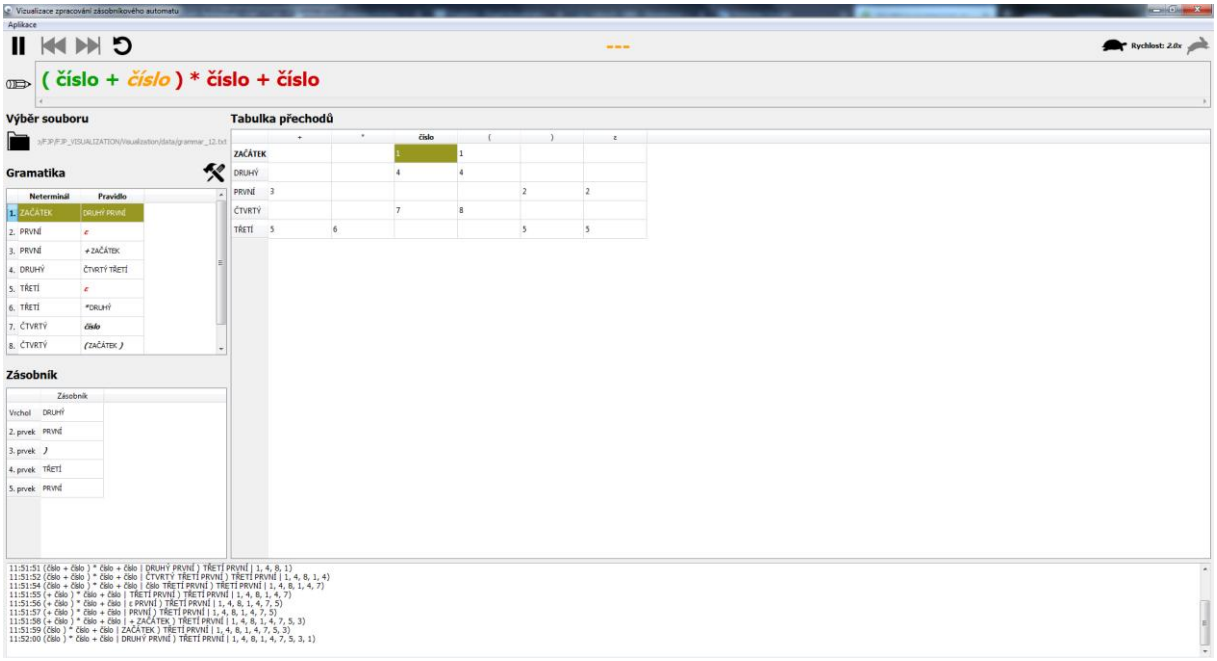
2.7 History

Instance třídy `History` jsou využívány pro zaznamenávání změn v automatu tak, aby bylo možné posunout se v historii zpět. Jedna instance je rovna jednomu kroku.

- Když se má během jednoho kroku změnit označená buňka v přechodové tabulce, nastaví se aktuální souřadnice do atributu `QPoint selectedCell`. Při návratu v historii jsou pak potřeba předchozí instance `History`. Při návratu označení v přechodové tabulce se musí změnit i vyznačený řádek v tabulce gramatiky, ten je shodný s řádkem přechodové tabulky, tedy souřadnicí `x`.
- Jestliže proběhla expanze, zaznamenat index vloženého pravidla na zásobník do atributu `int stackAdded`. Při návratu je ze zásobníku odebrán odpovídající počet záznamů. Navíc je odebrané i poslední pravidlo ze seznamu výsledných pravidel – kromě úplně prvního kroku vizualizace = přidání počátečního symbolu, pro tohle odlišení je zde atribut `bool start`. Ten je nastaven na `true` pouze při prvním kroku.

- Jestliže byl zpracován další kus vstupního řetězce, nastaví se na `true` atribut `bool inputIndexChanged`, který nám při návratu říká, že musíme posunout zpět index ukazující na právě zpracovávané slovo.
- Pak je ještě potřeba zaznamenávat, když se ze zásobníku něco odebere. To, co se odebírá, se ukládá do atributu `int stackRemoved`. Zde je problém, že se může odebrat jakékoli celé číslo (viz `2.6.1 Význam hodnot`), právě proto je zde ještě poslední atribut `bool removeHappened`, který je nastaven na `true` v případě, že se ze zásobníku opravdu něco odebere. Při návratu se nejdříve kontroluje právě tento atribut, poté až se řeší, co vlastně bylo odebráno.

3 Uživatelská příručka



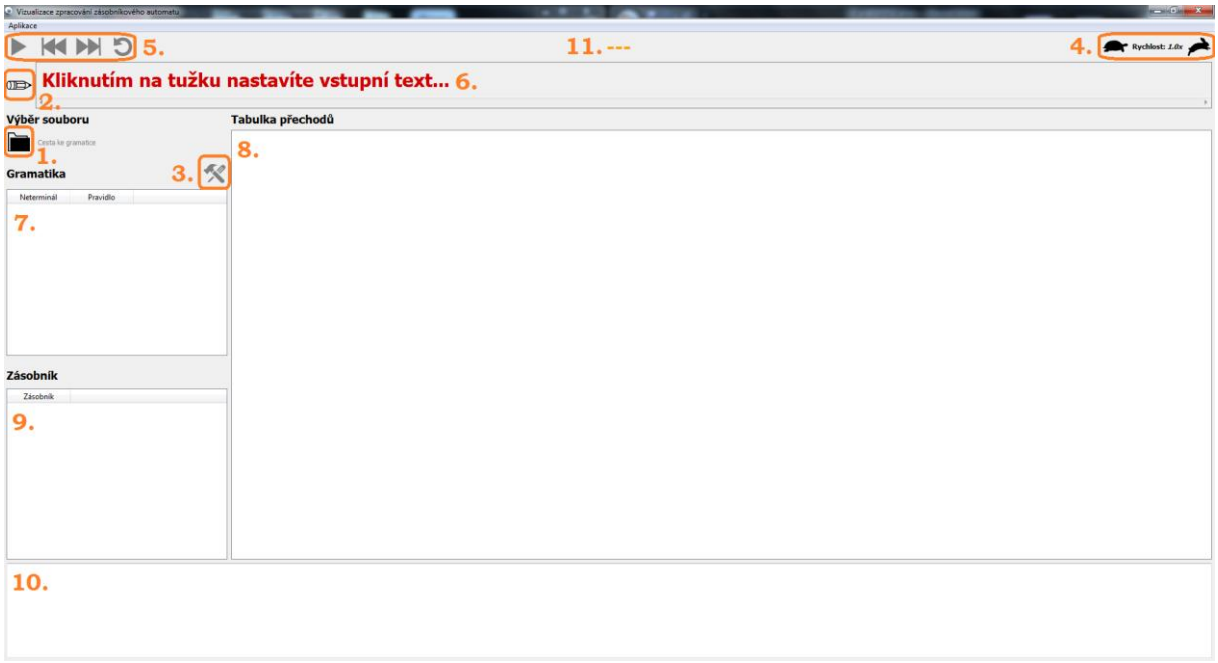
Obrázek 2 Vzhled aplikace

Ze všeho nejdříve doporučuji pro plnou funkcionalitu aplikace stáhnout a nainstalovat aplikaci `Graphviz` pro vykreslování grafů. Je důležitá pro vykreslení derivačního stromu. Jestliže aplikace nebude nainstalována, derivační strom nebude vykreslen. Balík pro `Windows` lze stáhnout z:
https://graphviz.gitlab.io/_pages/Download/Download_windows.html

Po nainstalování je ještě zapotřebí nastavit systémovou proměnnou `Path`, aby bylo možné spustit příkaz `dot`.

3.1 Popis ovládání

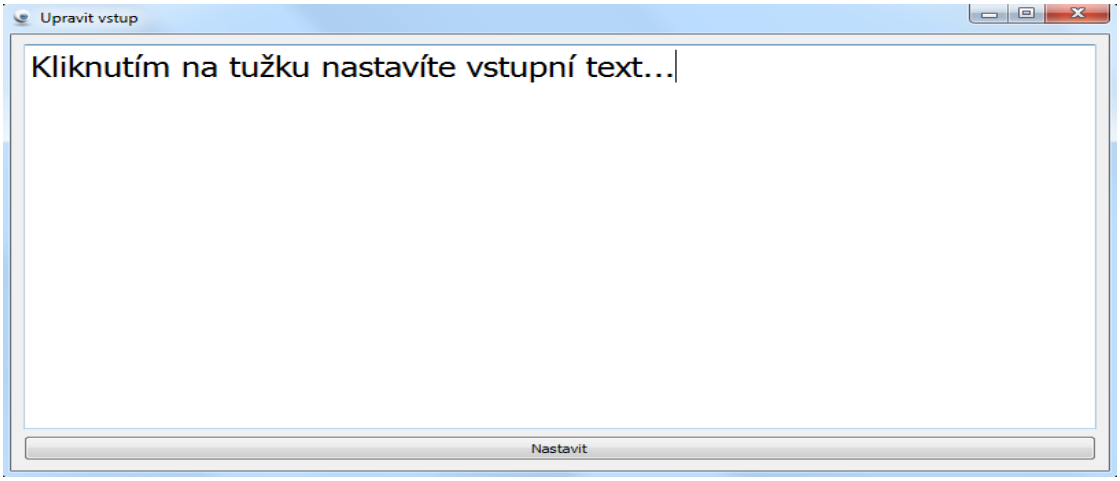
Aplikaci lze spustit přes spouštěcí binárku ve složce `bin`. Po spuštění se objeví následující okno (první spuštění může trvat trochu déle z důvodu zavádění dynamických knihoven):



Obrázek 3 Vzhled po spuštění

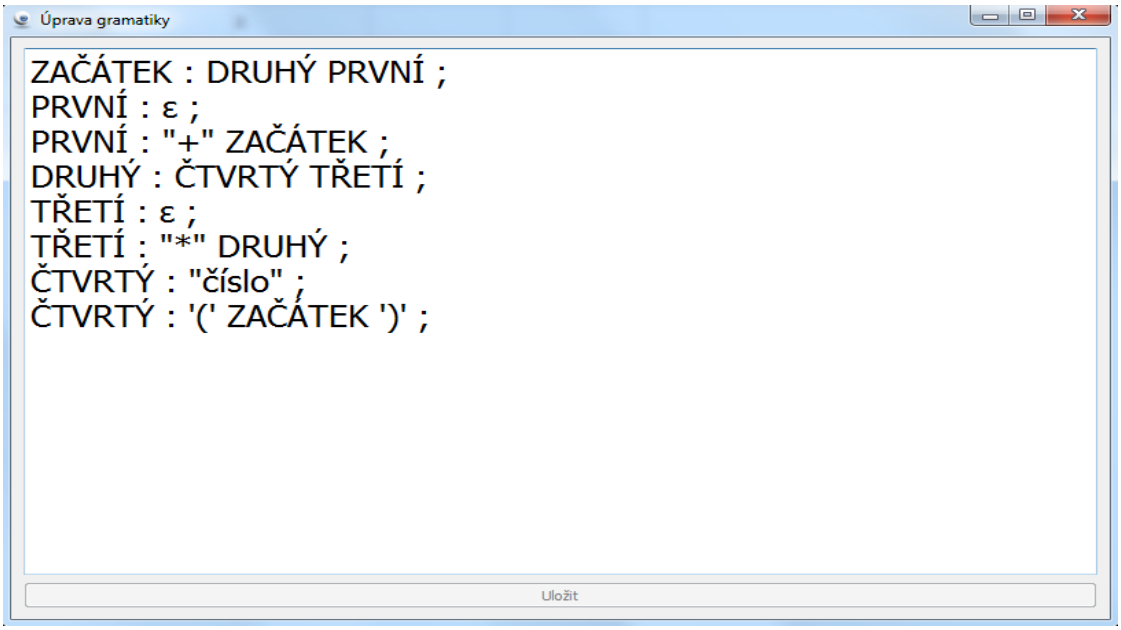
1. **Výběr gramatiky** – gramatika musí být uložena v textovém souboru, její formát musí odpovídat `Backus-Naurově` formě, konkrétně syntaxi `ANTLR` (až na regulární výrazy). Ve složce data jsou nějaké ukázky gramatik, které v aplikaci fungovaly tak, jak mají.

2. **Zadání vstupního textu** – po kliknutí na tužku se zobrazí okno, kam lze zadat i více řádkový text, ten je po uložení převeden na jednořádkový pro větší přehlednost vizualizace (je tomu samozřejmě přizpůsoben label, ve kterém je text zobrazován).



Obrázek 4 Okno pro nastavení vstupního řetězce

3. **Úprava gramatiky** – po kliknutí na tlačítko s kladivem a šroubovákem vyběhne velmi jednoduchý editor gramatiky (obsah v okně je vždy načten ze souboru). Vámi upravený výsledek můžete uložit do původního souboru nebo ho uložit pouze pro aplikaci (POZOR při opětovném spuštění okna se načte opět text ze souboru).



Obrázek 5 Okno pro úpravu gramatiky

4. **Nastavení rychlosti animace** – tlačítko se želvou slouží ke zpomalování animace, se zajícem naopak ke zrychlení. Minimální rychlost je poloviční, maximální dvojnásobná.

5. **Ovládání animace** – pro ovládání jsou zde čtyři tlačítka, konkrétně spuštění/zastavení, krokování zpět, dopředu a restart zpracování.

6. **Zpracovávaný vstupní text** – v tomto labelu je zobrazen uživatelem zadaný vstupní text. Barva zpracovaných slov je zelená, právě zpracovávaného slova je oranžová, a slova, která čekají na zpracování, jsou červená. Celý text je uživateli vždy zobrazen jako jednořádkový a label, ve kterém text je, má funkci automatického posouvání, což se hodí u delších textů.

7. **Načtená gramatika** – tabulka, kde na každém řádku je jedno přepisovací pravidlo. Je to kvůli lepší přehlednosti při odkazování z tabulky přechodů.

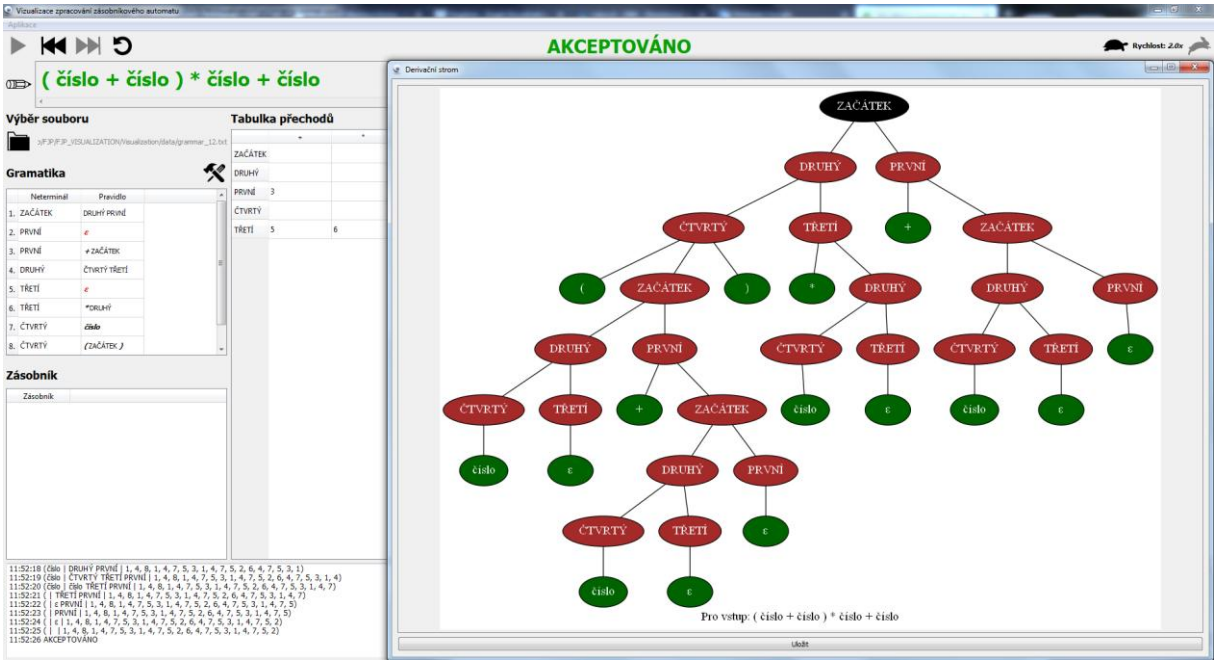
8. **Tabulka přechodů** – přechody, které neexistují, nemají v buňce nic. Jinak jde o id přepisovacího pravidla.

9. **Zásobník** – zde se zobrazuje, co všechno je na zásobníku, na jednom řádku je vždy buď jeden terminál, nebo neterminál.

10. **Textová oblast pro průběžné výpisy stavu automatu** – zde jsem se snažil dodržet formát výpisu ze cvičení.

11. **Výsledek zpracování** – zde mohou být buď oranžové tři pomlčky (ještě nedošlo k vyhodnocení), červený text `NEAKCEPTOVÁNO` (řetězec neodpovídá gramatice), nebo zelený text `AKCEPTOVÁNO` (řetězec byl v pořádku zpracován).

Výsledek doběhnutí aplikace v případě akceptovaného řetězce může vypadat třeba následovně:



Obrázek 6 Ukázka úspěšného zpracování

Zobrazený derivační strom si uživatel může uložit tam, kam si vybere. Jinak v menu je ukázka akceptovatelných řetězců pro konkrétní čísla gramatik (`Aplikace/Scénáře`). Na závěr bych chtěl dodat, že do tabulek je zakázáno klikat, je to z toho důvodu, aby uživatel nemohl zasahovat do vizualizace.

4 Závěr

Pro bezproblémové spuštění na počítači, kde není nainstalovaný `Qt framework`, jsou ke spustitelné binárce přidány `dll` knihovny. Licenční práva by neměla být porušena, jelikož jde o projekt, který není komerčně používaný a je veřejně dostupný.

Myslím si, že zadání bylo splněno, nicméně nedostatky tam jsou. Aplikaci by šlo například rozšířit o zpracování `LL(2)` gramatik. Také by šel vylepšit návrh datových struktur pro uložení gramatiky (třeba spojit seznam terminálů a neterminálů, dělící hranici uložit do nějaké celočíselné proměnné). Rozhodně by šlo zapracovat na editoru gramatiky, atd.

Použité zdroje

Studijní materiály z předmětu `KIV/FJP`, Prof. Ing. Karel Ježek CSc. a Ing. Richard Lipka Ph.D.
Webové stránky www.algoritmy.net, www.graphviz.org, atd.