VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

INTELIGENTNÍ EDITOR PRO JAZYK AHLL

INTELLIGENT EDITOR FOR THE AHLL LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — TOMÁŠ KUČERA
AUTHOR

VEDOUCÍ PRÁCE — Doc. Ing. FRANTIŠEK ZBOŘIL, Ph.D.
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/18322/2016/xkucer90

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Kučera Tomáš**
Obor: Informační technologie
Téma: **Inteligentní editor pro jazyk AHLL**
Intelligent Editor for the AHLL Language
Kategorie: Překladače

Pokyny:

1. Seznamte se s jazyky AHLL a ALLL a jejich implementacemi pro simulační nástroje a pro distribuované systémy. Dále se seznamtse se systémem WSAgeNt pro realizaci agentů v bezdrátových senzorových sítích.
2. Navrhněte prostředí, které umožní programování v jazyce AHLL a překlad z jazyka AHLL do ALLL s využitím existujícího překladače. Prostředí by mělo samostatně volit parametry, které optimalizují velikost výsledného kódu, nabízet programové konstrukce a dostupné služby podle zvolené agentní platformy.
3. Implementujte toto prostředí a integrujte jej do simulačního nástroje T-Mass.
4. Vytvořte poster formátu A2, který bude demonstrovat funkčnost navrženého řešení.

Literatura:

- Kalmár, R.: Optimalizace překladu agentních jazyků různé úrovně abstrakce, DP, 2011, Brno
- Kurti, S.: Grafické vývojové prostředí agentního jazyka ALLL, DP, 2011, Brno

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zbořil František, doc. Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Predmetom tejto bakalárskej práce je návrh a implementácia inteligentného editoru pre agentný jazyk AHLL. Čitateľ je najprv oboznámený s konceptom agentných systémov a platformou WSageNt. Nasleduje analýza a porovnanie existujúcich editorov a vývojových prostredí. Budú predstavené technológie ANTRL, RSyntaxTextArea a AutoComplete poskytujúce prostriedky pre efektívnu implementáciu niektorých kľúčových funkcií editora. Výsledný editor integruje existujúci AHLL prekladač do jazyka ALLL a simulátor distribuovaných systémov T-Mass. Na záver testujeme schopnosť editora vybrať najlepšiu kombináciu optimalizácií prekladača na základe dĺžky cieľového kódu.

Abstract

The subject of this bachelor's thesis is the design and implementation of an intelligent editor for the agent language AHLL. The reader is familiarized with the concept of agent systems and the WSageNt platform. Following is analysis and comparison of existing editors and integrated development environments. Presented will also be technologies ANTLR, RSyntaxTextArea and AutoComplete providing resources for effective implementation of some of the editor's key features. The resulting editor integrates existing AHLL to an ALLL compiler as well as the distributed system simulator T-Mass. Finally, we test the editor's ability to choose the best combination of compiler optimizations based on the target code's length.

Kľúčové slová

ALLL, AHLL, editor, ANTLR, zvýraznenie syntaxe, ponuka kľúčových slov, prekladač, optimalizácie, simulácia

Keywords

ALLL, AHLL, editor, ANTLR, syntax highlighting, code completion, compiler, optimization, simulation

Citácia

KUČERA, Tomáš. *Inteligentní editor pro jazyk AHLL.* Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. František Zbořil, Ph.D.

Inteligentní editor pro jazyk AHLL

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Doc. Ing. Františka Zbořila, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

. . . . . .
Tomáš Kučera
4. mája 2017

Poďakovanie

Týmto ďakujem vedúcemu práce Doc. Ing. Františkovi Zbořilovi, Ph.D. za odborné vedenie a pomoc pri tvorbe tejto bakalárskej práce.

Obsah

Kapitola 1

Úvod

Za desiatky rokov prešli vývojové prostredia mnohými evolučnými zmenami. Najskôr to bola schopnosť číslovania riadkov, či úpravy textu pomocou jednoduchých textových nástrojov. S komplexnosťou architektúr a programovacích jazykov však vzrástli aj požiadavky na efektivitu a pohodlie pri vývoji v daných prostrediach.

Editor popísaný v tejto práci má slúžiť na vývoj v jazyku AHLL vytvoreného Ing. Róbertom Kalmárom, určenom k programovaniu mobilných inteligentných agentov. V jeho prácach [6] a [7] bol tiež vytvorený prekladač tohto jazyka, ktorého úlohou je kompilácia do cieľového jazyka ALLL. Je taktiež schopný aplikovania rôznych optimalizácií, a tým zmenšiť veľkosť výsledného kódu.

Cieľom je teda vytvoriť program, ktorý bude prácu na programovaní agentov uľahčovať zvýrazňovaním kľúčových slov pre vyššiu prehľadnosť, ponukou dopĺňania ako kľúčových slov, tak aj celých programových konštrukcií a dynamicky pridaných identifikátorov. Taktiež má disponovať možnosťou prekladu zdrojového kódu pomocou vyššie uvedeného prekladača. Nakoľko tento kompilátor automaticky používa všetky implementované druhy optimalizácií, bude doň pridaná možnosť vybrať požadované optimalizačné techniky pomocou spúšťacích parametrov. To umožní editoru aplikovať rôzne kombinácie optimalizácií a vybrať tak najefektívnejší variant. Ďalšou hlavnou úlohou je integrovanie do simulačného nástroja T-Mass.

Prvá kapitola pojednáva o problematike agentov a s nimi súvisiacich multiagentných systémoch a platforme WSageNt. Nasledujúca kapitola popisuje spomenuté jazyky AHLL a ALLL, ich princípy a programové entity. Ďalšia časť zoznamuje čitateľa s už existujúcimi riešeniami editorov pre úpravu zdrojových kódov. Práca pokračuje návrhom, kde sú popísané použité technológie a spôsob, akým sa budú riešiť jednotlivé problémy. Posledná kapitola pojednáva o implementácii a testovaní jednotlivých funkcií editora.

Kapitola 2

Teoretická analýza

Táto kapitola pokladá teoretické základy k pojmu agent a multiagentné systémy (MAS). Teoretická analýza je základom pre pochopenie funkcionality daných princípov, a teda porozumenie, na čo sa pri tvorbe rozhrania pre programovanie v agentnom jazyku AHLL[1] zamerať.

2.1 Agent

Pojem agent predstavuje určitý autonómny prvok vnímajúci podnety a stavy vo svojom okolí. Činí tak senzormi, a takisto je schopný meniť stav daného prostredia tzv. aktivátormi. Ako príklad možno uviesť termostat, ktorý na základe teploty odmeranej na teplotných čidlách mení teplotu ohrievania miestnosti.

V oblasti informačných technológií je za agenta označovaný umelý agent - systém určený k plneniu nejakej úlohy. Schopnosť jeho samostatnosti ho odlišuje od ostatných pasívnych prvkov. Je to človekom vytvorené zariadenie založené na určitej architektúre a riadené programom z čoho vyplýva, že agent je akýmsi spojením programu a architektúry [12].

Za inteligentného agenta sa považuje agent vykazujúci inteligentné chovanie a majúci nasledujúce vlastnosti [14]:

- **Autonómnosť** - samostatné jednanie a jeho plná kontrola
- **Reaktivita** - schopnosť reakcie na zmeny
- **Proaktivita** - ovplyvňovanie svojho okolia vzhľadom na svoje ciele
- **Sociálnosť** - interakcia s ostatnými agentmi

Informácie zozbierané agentom sú spracúvané a môžu byť ďalej preposlané ostatným agentom na senzorovej sieti. Táto komunikácia môže prebiehať pomocou zvlášť vytvorenej káblovej architektúry alebo po čoraz viac sa rozširujúcej bezdrôtovej senzorovej sieti WSN[2].

2.2 Multiagentné systémy

MAS sú koncipované z množstva spolupracujúcich uzlov. Ako už bolo spomenuté vyššie, daní agenti v jednotlivých uzloch medzi sebou komunikujú a vytvárajú si obraz sledovanej

[1] Agent High Level Language
[2] Wireless sensor network - https://en.wikipedia.org/wiki/Wireless_sensor_network

oblasti, čím dochádza k učeniu agentov. Využívajú sa na riešenie problémov, ktoré sú ťažko riešiteľné, alebo dokonca neriešiteľné pre individuálneho agenta, resp. monolitický systém.

Z faktu, že sú zložené z viacerých komponentov vyplýva, že jednotlivé zložky môžu byť usporiadané v rôznych topológiách. Topológia, kde senzorové uzly priamo komunikujú s akýmsi spojujúcim bodom (základňová stanica), a taktiež ona komunikuje s uzlami naspäť sa nazýva hviezda.

Ďalším rozšíreným typom je usporiadanie do stromu, kde v menších oblastiach senzorových uzlov je vlastná základňová stanica komunikujúca s uzlami vo svojej oblasti. Táto stanica potom komunikuje s hlavnou základňovou stanicou pripojenou na centrálne stanovisko. Ďalšie topológie s podrobnejším vysvetlením a názornými ukážkami sú popísané v zdroji [9].

2.3 Platforma WSageNt

Platforma v MAS slúži na vytvorenie prostredia pre beh agentov. Agenta má na starosti riadiť a umožňovať mu vykonávanie určitých činností, ako napríklad pohyb alebo získavanie dát z jeho senzorov. Je tak akýmsi komunikačným kanálom medzi agentom a systémom [13].

Popis platformy WSageNt poskytnutý v tejto podkapitole je založený na informáciách uvedených na stránke projektu [5]. WSageNt je multiagentná platforma, ktorej úlohou je sledovanie, správa a práca s uzlami bezdrôtovej senzorovej siete. Návrh a vznik prebiehal na FIT VUT v Brne, kde je aj naďalej vyvíjaná. V nasledujúcich odsekoch bude platforma stručne popísaná.

Platforma poskytuje schopnosť behu agentov v bezdrôtových senzorových uzloch, konkrétne MICz a IRIS. Softvér je rozdelený do 2 hlavných častí – agentná platforma a interpret jazyka ALLL. Ten je interpretovaný na samotných senzorových uzloch a mení tak podstatu uzlu počas behu hostiteľskej aplikácie. Toto prispieva k možnosti behu agentov v senzorovej sieti, pozmenenia funkcionality uzlov podľa aktuálnych požiadaviek a prenosu agentov medzi jednotlivými uzlami.

Jadro platformy je naprogramované v TinyOS v2. Takáto platforma je schopná behu v simulačnom prostredí nástroja TOSSIM, čo umožňuje testovanie jednoduchých agentov pomocou simulačných skriptov. Výhoda tohto prístupu je reflektovaná v situácii, kedy nie sú uzly fyzicky k dispozícii. Kľúčovými vlastnosťami platformy sú:

- Agenti môžu byť klonovaní a pridaní do iného uzlu
- Komunikácia agentov je riešená pomocou zasielania a prijímania správ
- Kompatibilita vďaka TinyOS v2

Kapitola 3

Jazyky ALLL a AHLL

Kapitola poskytuje základné informácie o oboch jazykoch. Zameriava sa na ich charakteristiku po stránke teoretickej, ako aj praktickej vo forme znázornenia ich syntaktických konštrukcií a jednotlivých prvkov jazykov.

3.1 ALLL

Agentný jazyk ALLL je založený na procedurálnom prístupe a bol vyvinutý na FITe. Napriek tomu, že existovalo niekoľko agentných jazykov a kalkulov, bola potreba vytvoriť jazyk schopný obsluhy výnimiek, meta-rozhodovania a klonovania. Agent má vďaka tomu schopnosť vysporiadať sa s nečakanými chybami počas behu. Nová platforma mala taktiež umožňovať podporu riadenia, monitorovania a hlavne pohybu informácií a predovšetkým celých agentov medzi uzlami senzorovej siete. Informácie obsiahnuté v tejto sekcii sa opierajú o prácu Ing. Pavla Spáčila [13], a tiež [15] a [16], kde možno taktiež nájsť ukážky kódov a princíp použitia jazyka v praxi.

Jazyk je využívaný na programovanie MAS (2.2), ktoré sa vyznačujú zníženou energetickou náročnosťou hardwaru, ako aj jeho malou výkonnosťou, čo je spôsobené nízkou úrovňou abstrakcie, ktorú tento jazyk ponúka. Z tohto faktu vyplýva, že programátorovi nie je poskytnutá žiadna forma premenných, avšak pracuje priamo s registrami, do ktorých je možné ukladať medzivýsledky operácií. Taktiež sú k dispozícii zoznamy, podobné zoznamom v jazyku Prolog, a to v rôznych úrovniach zanorenia.

V súčasnosti existujú 2 verzie jazyka, jedna je určená priamo pre bezdrôtové senzorové uzly, tá druhá pre simuláciu v programe T-Mass, ktorý je určený na tvorbu a simulovanie distribuovaných systémov [17]. Použitý prekladač vuyžíva druhú verziu, preto sa zvyšok práce zaoberá práve týmto variantom jazyka.

3.1.1 Prvky jazyka

Základom jazyka sú atómy a zoznamy. Atóm definujeme ako neprázdnu postupnosť malých, velkých písmen, číslic, znakov podtržník, plus, mínus alebo bodka:

```
atom : ( [a-zA-Z0-9] | '_' | '+' | '-' | '.' )+
```

Medzi druhy zoznamov patria dotazovacie a bežné zoznamy. Dotazovacím zoznamom rozumieme postupnosť elementov uzavretých v hranatých zátvorkách '[' a ']'. Elementom

sa v tomto prípade myslí atóm, register, číslo, ďalší dotazovací zoznam alebo anonymná premenná reprezentovaná znakom podržníka '_'.

Druhým typom zoznamu je tzv. bežný zoznam. Ten predstavuje postupnosť atómov, registrov, čísel alebo iných bežných zoznamov uzavretých okrúhlymi zátvorkami '(' a ')'.

Plány

Podstatnými prvkami ALLL sú vety, ktoré sú reprezentované plánmi `plan`. Tie sú zas len vety zložené z jednotlivých akcií `action`. Jazyk poskytuje hierarchické zanorovanie plánov, čo pri chybe vykonávania nejakej akcie v pláne na nižšej úrovni umožňuje zmazať daný podplán a pokračovať vo vykonávaní plánu na vyššej hierarchickej úrovni.

```
plan : '<' action ( ',' action )* '>'
```

Abstraktné štruktúry a registre

Agent popísaný týmto jazykom disponuje 7 časťami (4 zoznamy, 3 registre) [13]:

- PlanBase (báza plánov) – neusporiadaný zoznam plánov
- BeliefBase (báza znalostí) – obsahuje obdržané informácie o prostredí agenta
- InputBase (báza vstupov) – informácie získané od platform
- Plan (plán) – zámer agenta
- Register 1 – univerzálny register
- Register 2 – univerzálny register
- Register 3 – univerzálny register

Prvou z častí, a zároveň prvým popísaným zoznamom bude PlanBase. Táto časť je akýmsi zoznamom akcií, ktoré sa počas behu agenta môžu vykonať. Každý zoznam musí byť pomenovaný kvôli tomu, aby sa daný plán mohol opakovane volať v iných častiach programu.

Báza znalostí, pomenovaná taktiež beliefBase, je úložným priestorom dát, ktoré sú potrebné pre výpočty. Tieto dáta sú vytvárané buď za behu agenta – zberom informácií počas interakcie s okolým, alebo pred prvým spustením činnosti agenta. Štruktúra definície báze znalostí v kóde jazyka ALLL je ďalej popísaná v podsekcii 3.1.2.

InputBase je zoznam, do ktorého sú informácie o nameraných hodnotách a správach ukladané samovoľne. Záznamy sú zložené z n-tíc s údajmi o type hodnoty (prijatá správa, získaná hodnota meraním. . .) a samotnej hodnoty. Rozdielom medzi bázou znalostí a bázou vstupov je v tom, že do báze znalostí si agent ukladá údaje sám, a sám tiež môže rozhodovať o tom, ktoré údaje do nej zaradí, čo prispieva k väčšej autonómnosti agenta.

Ďalším zoznamom je zámer agenta, označovaný aj Plan. Je vyjadrením akcie, ktorú bude agent po svojom spustení vykonávať. Okrem uvedenej postupnosti akcií môže zámer obsahovať názvy zoznamov akcií z báze plánov.

Ako už bolo spomenuté, registre slúžia programátorovi na ukladanie dočasných hodnôt. Využívajú sa napríklad v akciách namiesto fyzických hodnôt, kde sú reprezentované symbolom. Ten je za reálnu hodnotu nahradený pred započatím vykonávania akcie. Register môže

byť teda použitý ako vstup akcie alebo ako časť n-tice. V určitom okamihu je nastavený iba jeden z registrov ako aktívny.

Akcie

Hlavným atomickým prvkom jazyka je inštrukcia, alebo taktiež akcia. Sekvencia akcií tvorí plán, v ktorom sú dané akcie vykonávané postupne. Nakoľko boli služby platformy v práci [13] rozšírené o podporu matematických operácií, riadiace konštrukcie iterácia a selekcia sú v jazyku priamo podporované a nie je teda potreba docieliť danej funkcionality pomocou báze znalostí, ako to bolo v predchádzajúcej verzii jazyka.

Podrobný popis jednotlivých akcií:

- Priame spustenie plánu `'@' plan`

  Toto spustenie sa využíva pri plánoch, u ktorých je vyššia pravdepodobnosť zlyhania niektorej z akcií a je nutné sa vyhnúť taktiež zlyhaniu nadradeného plánu. Akcia má len jediný parameter, ktorý predstavuje zoznam akcií určených k vykonaniu. Na zásobník je umiestnená zarážka, ak ju už neobsahuje, a pred ňu sa ukladajú akcie. Ak dôjde k pretečeniu, akcia zlyháva.

- Nepriame spustenie plánu `'^(' atom ')'`

  Parametrom danej inštrukcie je názov plánu hľadaného v báze plánov. Pokiaľ bol plán nájdený, vkladá sa akcia na zásobník (prípadne so zarážkou, ako je popísané pri priamom spustení), inak akcia zlyháva.

- Odoslanie správy `'!(' item ',' generalList ')'`

  Služba odoslania správy cez rádio je poskytovaná platformou. Obsahuje 2 parametre – `item` predstavuje adresu cieľovej platformy a `generalList` bežný zoznam definovaný v podsekcii 3.1.1 určený k odoslaniu. Parameter `item` je atóm, no je ho možné zadať taktiež registrom. Cieľová adresa musí obsahovať iba 1 element, inak akcia zlyháva. Správa k odoslaniu je najprv skopírovaná, následne sú v nej nahradené označenia registrov za ich hodnoty a potom je správa pripravená na odoslanie.

- Prijatie správy `'?(' item2 ')'`

  Správy v podobe n-tíc sú pridané do báze znalostí implicitne. Jediným parametrom inštrukcie je adresa stanice, z ktorej má byť daná správa prijatá. Položka `item2` môže byť definovaná podobne ako položka `item` spomenutá vyššie - atómom, registrom, a navyše symbolom unifikácie '_'. Pri nájdení elementu v tvare „adresa, n-tica“ vo vstupnej báze je správa uložená do aktívneho registra.

- Pridanie zoznamu do báze znalostí `'+' generalList`

  Inštrukcia pridania n-tice do báze znalostí vykoná vyhľadanie danej n-tice v tabuľke, a pokiaľ táto operácia neprebehla úspešne, čiže sa v tabuľke ešte nenachádza, uloží sa na prvú pozíciu. Pokiaľ nastane situácia, že tabuľka je preplnená a došlo by k pretečeniu, akcia končí chybou.

- Pridanie plánu do báze znalostí `'+{' item ',' plan '}'`

  Podobné ako pridanie zoznamu, plán je pomenovaný hodnotou `item`. Kľúčové slovo `plan` je definované v podsekcii 3.1.1.

- Vymazanie zoznamu z báze znalostí `'-' (generalList | queryList)`

  Inštrukcia vymazania n-tice z báze znalostí vykoná vyhľadanie danej n-tice v tabuľke, a pokiaľ táto operácia prebehla úspešne, čiže sa v tabuľke nachádza, je táto n-tica z tabuľky vymazaná. V prípade, že by sa n-tica v tabuľke nenachádzala, akcia aj napriek tomu končí úspechom. Pokiaľ sa jedná o dotazovací zoznam `queryList`, zmazané sú všetky odpovedajúce n-tice.

- Vymazanie plánu z báze znalostí `'-(' item2 ')'`

  Podobné ako zmazanie zoznamu, akcia vymaže plán odpovedajúci výrazu `item2`.

- Dotaz do báze znalostí `'*' queryList`

  Akcia vyhľadáva dotazovacie zoznamy v báze znalostí. Ak sa zoznam v báze nenachádza, akcia končí neúspechom, inak sa nájdené n-tice následne uložia do aktívneho registru

- Volanie služby platformy `'$' generalList`

  Parameter `generalList` popisuje službu, ktorá sa má vykonať. Prvý prvkom je názov služby, zvyšné popisujú parametre danej služby. Podrobný popis jednotlivých služieb a ich parametrov sa nachádza v tabuľke 3.1.

- Zmena aktívneho registra `'&' value`

  Spôsobí nastavenie registra daného parametrom value na aktívny. Tým pádom je úložným priestorom pre akcie s výstupom.

3.1.2 Štruktúra jazyka

Program skonštruovaný v ALLL po stránke štruktúry zahŕňa časť so zámerom agenta a sekciu s bázou znalostí, zapísané ako:

```
program : intention (beliefBase)?
intention : '>intention' ':' plan ';'
beliefBase : '>database' ':' (generalList | planRecord)* ';'
```

Z formálnej definície vyplýva, že uvedenie báze znalostí je nepovinné. Kľúčové slovo `generalList` označuje bežný zoznam a `planRecord` reprezentuje plán označený identifikátorom a uložený v báze znalostí:

```
generalList: '(' atom ( ',' atom)* ')' ';'
planRecord : '{' atom ';' plan '}' ';'
```

Kód	Parametre	Popis
a	žiadne	Aktivuje sa sledovanie prichádzajúcich správ pri bežiacom interpretri.
f	zoznam| register	Vloží do aktívneho registra prvý prvok zoznamu ako jednoprvkovú n-ticu alebo prvý zo zoznamov, ak ich je viac.
k	žiadne	Zastaví činnosť interpretru
l	zoznam| register	Ovládanie LED na uzle, parameter musí obsahovať kód farby LED (r, g, y) a za ním môže byť stav LED (0 - vypnuté, 1 - zapnuté), pokiaľ nie je zadaný, dojde k prepnutiu aktuálneho stavu.
m	zoznam| register	Parameter služby obsahuje adresu platformy, kam sa má kód agenta presunúť, celý kód sa skopíruje do cieľovej platformy a vykonávanie pokračuje na oboch platformách ďalej. Ak je cieľová adresa rovnaká ako tá, na ktoré práve agent je, vykonaním tejto služby sa nič nestane. Za adresou môže byť parameter `s`, ktorý značí príkaz zastavenia vykonávania kódu na zdrojovej platforme.
r	zoznam| register	Vloží do aktívneho registru zbytok zoznamu bez prvého prvku alebo všetky zoznamy bez prvého z nich. Ak je zoznam iba jeden a obsahuje len jeden prvok, do aktívneho registru sa vloží prázdna n-tica.
s	žiadne	Zastaví vykonávanie kódu, pokým nepríde správa z rádia. Ak je aktivované sledovanie a správa uz je prijatá, vykonávanie pokračuje ďalej.
w	zoznam| register	Pozastavenie vykonávania kódu na zadaný čas v milisekundách.
d	žiadne| zoznam| register	Volanie bez parametru zmeria aktuálnu teplotu, v opačnom prípade je potreba zadať typ hodnoty (a - priemer, m - minimum- M - maximum) a počet hodnôt, z koľkých sa má hodnota vypočítať. Ak nie je nameraný dostatočný počet hodnôt pre výpočet, dochádza k zlyhaniu služby.

Tabuľka 3.1: Prehľad služieb platformy. Prevzaté z [13].

3.2 AHLL

AHLL je imperatívny štruktúrovaný jazyk pre programovanie mobilných inteligentných agentov. Vytvorený bol v bakalárskej práci [6] Ing. Róbertom Kalmárom a ďalej upravený v jeho diplomovej práci [7]. V daných dielach bol taktiež navrhnutý a implementovaný prekladač tohto jazyka do jazyka ALLL.

Syntaktické konštrukcie a prvky jazyka sú podobné jazyku C[1]. AHLL poskytuje ako lokálne, tak aj globálne premenné, namiesto funkcií sa definujú tzv. plány, vstupným bodom programu je špeciálny plán nazvaný `main`, umožňuje deklarovať novú službu platformy atď.

3.2.1 Prvky jazyka

Premenné

Lokálne premenné majú platnosť len v bloku, v ktorom boli deklarované a od miesta deklarácie. Po skončení bloku premenná prestáva byť prístupnou a je z pamäte vymazaná. Naopak

[1] Jazyk C - https://en.wikipedia.org/wiki/C_(programming_language)

globálne premenné je možné použiť v ktorejkoľvek časti programu. Premenné sa deklarujú pomocou kľúčového slova `var`. Je zároveň možná deklarácia viacerých premenných naraz pri oddelení jednotlivých identifikátorov znakom čiarky. Použitie neinicializovanej premennej vedie k chybe. Inicializácia premennej je následne vykonaná pomocou znaku '='. Do jazyka boli taktiež pridané konštanty, definované pomocou kľúčového slova `const`, a od miesta ich inicializácie nie je možné meniť ich hodnotu:

```
'var' identifier ';'
identifier '=' value ';'
'const' identifier '=' value ';'
```

Plány

Hlavným prvkom plánu je blok. Je to zoznam príkazov `commands` ukončených znakom bodkočiarky ';' a uzavretý v zložených zátvorkách '{' a '}' :

```
block : '{' commands '}'
```

Plán môžeme prirovnať k funkciám v jazyku C. Je možné ich definovať pomocou kľúčového slova `plan`, priradiť im názvy `identifier` a parametre `parameters`, ktoré sú predávané hodnotou. Plány nemajú návratovú hodnotu, túto funkcionalitu je však možné docieliť pomocou globálnych premenných. Po definícii je možné plán zavolať:

```
'plan' identifier '(' parameters ')' block
identifier '(' parameters ')' ';'
```

Selekcia a iterácia

Podmienený príkaz je realizovaný pomocou známeho if-else. Syntax je nasledovná:

```
'if' '(' condition ')' block_1 'else' block_2
```

`condition` je podmienka, ktorá sa vyhodnocuje, a v prípade jej platnosti je vykonávaným `block_1`, inak sa skáče na `block_2`.

Jazyk AHLL zatiaľ ponúka 2 konštrukcie pre vykonávanie cyklov. Prvým z nich je cyklus while:

```
'while' '(' condition ')' block
```

Pokiaľ platí podmienka `condition`, sú vykonávané príkazy uvedené v `block`. Po čase bol do jazyka pridaný cyklus na podporu priechodu zoznamom, ktorý má nasledovnú syntax:

```
'foreach' tmp 'in' (identifier | list) 'do' block
```

`tmp` je premenná určená na uloženie aktuálneho prvku v prechádzanom zozname. Táto premenná nemusí byť predom deklarovaná. `identifier` (`list`) je premenná (zoznam), cez ktorú sa iteruje.

Služby platformy

Ako už bolo spomenuté, jazyk podporuje definíciu nových služieb platformy. Syntax definície služby:

```
'service' identifier '(' parameters ')' '{'
     'call' list ';'
     'return' ('null')? ';'
'}'
```

`identifier` označuje meno služby, `parameters` parametre a `list` n-ticu, ktorá sa má zavolať pri spustení služby. Služby môžu mať návratovú hodnotu, keď sa však za kľúčovým slovom `return` uvedie `null`, služba končí bez nej.

Správy

Syntax jazyka obsahuje funkcie pre odosielanie a prijímanie správ z jedného senzorového uzlu na druhý. Príkaz na odoslanie správy obsahuje 2 parametre – adresa uzlu `mote` a správa určená k odoslaniu `message`.

Prijatie správy disponuje naopak len jedným parametrom `mote` reprezentujúcim uzol, z ktorého má byť správa prijatá. Parameter je volitelný a pri jeho absencii je zachytená prvá prijatá správa bez ohľadu na odosielateľa.Tento príkaz môže byť takisto použitý vo výrazoch, o ktorých pojednáva nasledujúca kapitola.

```
'send' '(' mote ',' message ')' ';'
'receive' '(' mote ')' ';'
```

Výrazy a identifikátory

Výraz môže obsahovať identifikátory, číselné konštanty, služby platformy s návratovými hodnotami, príkaz `receive` a operátory. AHLL implementuje 2 druhy operátorov - binárne operátory sú popísané v tabuľke 3.2 a unárne v tabuľke 3.3. Výrazy sa používajú pri inicializácii premenných alebo vyhodnocovaní podmienok. Identifikátor je definovaný ako postupnosť malých a veľkých písmen, číslic, pričom názov musí začínať písmenom. Formálne zapísané:

```
identifier : (('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9') *)
```

Príklad 3.2.1. *Ukážka štruktúry programu na príklade kódu AHLL agenta.*

```
var g;   // deklaracia globalnej premennej
g = 3;   // inicializacia

service hlavicka(lst) { // sluzba vrati 1. prvok zoznamu lst
    call ["lst", ["h", lst]];
    return;
}

plan plan_1(a, b) {   // definicia planu plan_1
    var c;   // deklaracia lokalnej premennej
```

```
    c = hlavicka(a);   // volanie sluzby

    // zvysok planu
}

main() {   // vstupny bod programu
    var d, e;
    d = [4, 5, 6];
    e = 3;

    plan_1(d, e);   // volanie planu

    // zvysok main-u
}
```

Operátor	**V**ýznam
*	násobenie
/	delenie
%	modulo
+	súčet
-	mínus
>	väčšie než
>=	väčšie rovno
<	menšie než
<=	menšie rovno
==	rovnosť
!=	nerovnosť
&&	logický súčin
||	logický súčet

Tabuľka 3.2: Binárne operátory. Prevzaté z [7].

Operátor	**V**ýznam
+	plus
-	mínus
!	negácia
++	inkrementácia
–	dekrementácia

Tabuľka 3.3: Unárne operátory. Prevzaté z [7] a upravené.

Kapitola 4

Existujúce riešenia

Štúdium existujúcich riešení dopomáha k porozumeniu problémov, na ktoré je potreba pri vývoji aplikácie dbať. Analýza dôležitých prvkov je stavebným kameňom pre vytvorenie programu splňujúceho požiadavky cieľového užívateľa.

Nakoľko žiaden inteligentný editor pre kód jazyka AHLL neexistuje, je potreba študovať iné editory. Jedným z variantov je grafické vývojové prostredie jazyka ALLL od Ing. Szabolca Kürtiho [8]. Keďže tento jazyk svojou programovou štruktúrou nie je veľmi podobný jazyku AHLL, je potreba taktiež čerpať poznatky o ďalších editoroch zameraných na jazyky s vyššou úrovňou abstrakcie. Prostredia tohto typu majú totižto vzhľadom na nároky jazyka iné vlastnosti a funkcie, podobné produktu vyvíjanému v tejto práci.

4.1 Grafické vývojové prostredie ALLL

Integrované vývojové prostredie (IDE[1]) je akousi nadstavbou editoru, ktorý je úlohou tejto práce, avšak je veľmi vhodný na inšpiráciu pri tvorení editoru s podporou pre určitý jazyk. IDE vo všeobecnosti pozostáva z nasledujúcich častí:

- Editor zdrojového kódu
- Kompilátor/interpret zdrojového jazyka
- Debugger

Toto prostredie ponúka možnosť vytvorenia, ukladania a upravovania projektu, ktorého súčasťou sú kódy agentov a topológia siete vo formáte XML[2]. Upravovanie agentného kódu je možné po označení jedného z agentov v danej topológii a následnom výbere akcií, resp. služieb platformy, ktoré požaduje užívateľ pridať do výsledného kódu agenta.

Editor taktiež ponúka debugger simulujúci činnosť jedného agenta s možnosťou nastavenia rôznych parametrov, ako napríklad stav registrov, hodnota senzoru atď.

4.2 Sublime Text

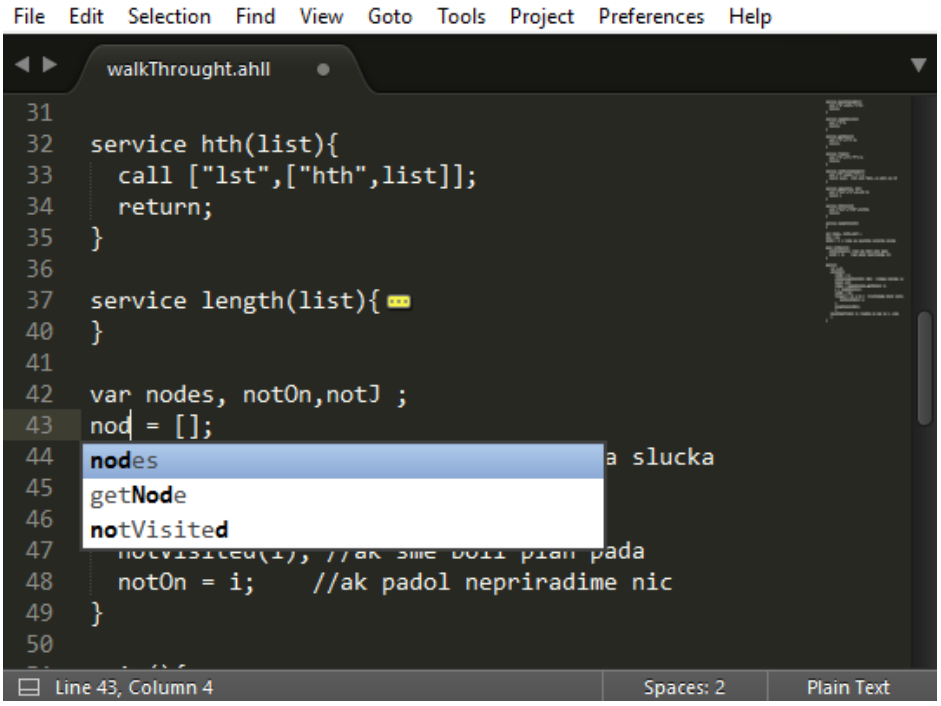
Sublime Text[3] je príkladom editoru pre upravovanie zdrojového textu. Ponúka štýly pre zvýraznenie syntaxe mnohých jazykov, ktoré sú volené samostatne na základe prípony otvore-

[1] Integrated Development Environment - https://en.wikipedia.org/wiki/XML
[2] eXtensible Markup Language
[3] Sublime Text oficiálna stránka - https://www.sublimetext.com/

ného súboru, alebo ručne v spodnom panely. Editor taktiež disponuje funkciou zvýraznenia, skrytia a znovuzobrazenia blokov. Táto funkcia sa využíva pre dosiahnutie väčšieho počtu užitočných riadkov na obrazovke pre minimalizáciu nutnosti skrolovania v dokumente.

K dispozícii sú tiež klávesové skratky pre zrýchlenie práce s najpoužívanejšími funkciami, zostavovací systém pre preklad a mnoho ďalšieho.

Za zmienku určite stojí ponuka kľúčových slov na základe výberu požadovaného editovacieho štýlu, a hlavne dynamické pridávanie identifikátorov do ponuky slov. To znamená, že akonáhle je v jazyku deklarovaný nový identifikátor, je pri prichádzajúcom požiadavku na ponuku slov zobrazený v zozname, pokiaľ vyhovuje už napísanému textu pred kurzorom v textovej časti. Popísané prvky a funkcionalitu prezentuje obrázok 4.1



Obr. 4.1: Ukážka editoru Sublime Text.

4.3 Vývojové prostredie Eclipse

Eclipse[4] primárne slúži ako vývojové prostredie pre jazyk Java, avšak podporované sú taktiež C/C++ alebo PHP. Jednoducho kombinovateľné jazykové podpory a iné služby ponúkané v predvolených balíčkoch robia prostredie silným nástrojom pre vývoj. K dispozícii je aj Eclipse Marketplace, z ktorého možno získať rôzne rozšírenia pre upravenie tohto IDE.

Za spomenutie stoja optické rozdiely oproti grafickému prostrediu a editoru spomenutým vyššie. Prvým podstatným prvkom je element GUI reprezentujúci hierarchiu súborového systému otvoreného projektu. Takisto je pri rozkliknutí triedy možné vidieť zoznam implementovaných metód, ktorý slúži ako referencia na ich implementáciu.

Ďalším prvkom uľahčujúcim prácu je panel nástrojov umiestnený pod menu aplikácie. Ten obsahuje najčastejšie používané akcie pre rýchlejší prístup. Je tak akýmsi kompromisom medzi používaním klávesových skratiek a hľadaním požadovaných akcií v menu.

[4]Eclipse oficiálna stránka - https://eclipse.org/

Kapitola 5

Návrh aplikácie

Návrh aplikácie začínal výberom programovacieho jazyka a vývojového prostredia vhodného na riešenie danej úlohy. Nasledovalo rozdelenie problému na podproblémy. Ich postupné riešenie zahrňovalo okrem iného aj výber technológií schopných uľahčiť samotný vývoj editoru. Technológie museli byť volené na základe vybraného jazyka a prostredia. Konceptuálny návrh na obrázku 5.2 zahrňuje len tie najpodstatnejšie balíčky, triedy, metódy a asociácie. Množstvo elementov bolo z obrázka vynechaných kvôli zachovaniu prehľadnosti. Obsiahnuté triedy sú ďalej popísané v kapitole o implementácii (6).

Pre lepší výsledný efekt boli následne navrhnuté vylepšenia, ktoré by výsledný produkt priblížili k skúmaným existujúcim editorom. Tieto vylepšenia majú programátorovi (užívateľovi aplikácie) poskytnúť nástroje pre zrýchlenie tvorby kódu, a tým zlepšenia celkového pocitu z produktu.

5.1 Programovací jazyk a prostredie

Zložitosť projektu vyžaduje návrh tried, ktorý sa dá následne aplikovať do objektovo orientovaného programovacieho jazyka. Nevyhnutnou požiadavkou je vlastnosť grafickej knižnice, pomocou ktorej bude možné vytvoriť grafické užívateľské rozhranie. Výhodou je, ak vybraný jazyk prenositeľný medzi rôznymi architektúrami a operačnými systémami.

Zvoleným kritériám vyhovuje jazyk Java. Jej vývoj začal v roku 1991 firmou Oak (neskôr Sun Microsystems, teraz Oracle). Stavaná bola na základoch jazyka C a C++. Krátku dobu po predstavení v roku 1995 bol o Javu veľký záujem, nakoľko predstavovala pre vtedy začínajúci Internet a WWW veľký pokrok [4].

Je to multiplatformný[1] jazyk, o čo sa stará preklad do bajtkódu namiesto klasického strojového kódu. Bajtkód je nezávislý na cieľovej platforme, nakoľko je interpretovaný pomocou Java platformy. Tá je zložená z virtuálneho stroja JVM (Java Virtual Machine), na ktorom beží výsledná aplikácia a Java Core API poskytujúce rozhranie, ktoré obsahuje knižnice pre jednoduchší a rýchlejší vývoj v tomto jazyku [4].

Úlohu grafickej knižnice bude plniť Swing na základe vybranej technológie RSyntaxTextArea (kapitola 5.2.2) rozširujúcej triedu grafického komponentu tejto knižnice.

Java disponuje prvkami pre zvýšenie programovacieho komfortu, ktorými sú napríklad pokročilé vývojové prostredia podporujúce rôzne aspekty jazyka od dopĺňania programových syntaktických konštrukcií, po ladenie programu. Jednou z výhod je aj garbage collector

[1] Cross-platform - https://en.wikipedia.org/wiki/Cross-platform

starajúci sa o automatické uvoľňovanie alokovanej pamäte. Ten prechádza zabratú pamäť a uvoľňuje položky, na ktoré už neexistuje v programe referencia.

Čo sa týka integrovaného vývojové prostredia, Eclipse poskytuje všetko potrebné pre splnenie požadovanej úlohy. Taktiež obsahuje vývojársky nástroj ANTLR 4 (kapitola 5.2.1), ktorý pomáha pri tvorbe gramatík zvýraznením syntaxe gramatických konštrukcií, a takisto generovaním syntaktických a lexikálnych analyzátorov z týchto gramatík v cieľovom jazyku (v tomto prípade Java).

5.2 Použité technológie

Pre dosiahnutie požadovanej funkcionality bolo možné použiť externé open source technológie. Ich všeobecný popis a predstavenie ich použitia pri tvorbe editora sú obsiahnuté v tejto kapitole. Najprv bolo nutné naštudovať si, ktoré možné riešenia sú schopné poskytnúť požadované správanie. Následne prebehlo porovnanie ich výhod a nevýhod, bol zvolený najvyhovujúcejší spôsob a jeho podrobné naštudovanie pre schopnosť jeho aplikovania.

5.2.1 ANother Tool for Language Recognition v4

ANother Tool for Language Recognition (ANTLR) je nástroj na generovanie lexikálnych či syntaktických parserov slúžiacich pri vývoji prekladačov pomocou gramatických pravidiel. Je schopný čítania, spracovania alebo prekladania štruktúrovaného textu alebo binárnych súborov. Šírený je pod licenciou BSD. Vytvorený bol profesorom Terence Parr, ktorý sa vývojom jazykových nástrojov zaoberá ako roku 1989 [11].

Ako prvé je nutné naštudovať a vytvoriť gramatiku jazyka, z ktoréj ANTLR produkuje spomenuté parsery v zvolenom programovacom jazyku. Plne podporovanými jazykmi sú Java, C#, Python 2 a 3, JavaScript, Go, C++ a Swift [11].

Oproti minulým verziám ANTLR bolo vykonaných veľa zmien. Namiesto predtým generovaného abstraktného syntaktického stromu generuje derivačný strom. Známym operátorom priradenia '=' je možné definovať názvy a význam jednotlivých pravidiel, a zároveň ich združovať operáciou '+=' [10]. Tokeny nepotrebné pre syntaktickú analýzu je možné preskočiť pri parsovaní konštrukciou `->skip`, avšak ak tieto tokeny majú byť využité v iných častiach programu (priradenie štýlu, počítanie medzier u bielych znakov), tak sa posielajú do tzv. skrytého kanálu príkazom `->channel(číslo_kanálu)`.

Príklad 5.2.1. *Príklad definovania lexikálneho pravidla pre biele znaky v ANTLR v4.*

```
WHITESPACE : [\t\f ]+ -> channel(1) ;
```

Gramatické a lexikálne pravidlá je možné rozdeliť do 2 súborov, pričom sa môžu skladať pomocou kľúčového slova `import`. Kombinácia do jedného súboru je taktiež možná, avšak menej prehľadná. Súbor na začiatku obsahuje definíciu toho, či sa jedná o lexikálne alebo syntaktické pravidlá, nasleduje kľúčové slovo `grammar` a názov gramatiky. Všetky súbory s gramatikami majú príponu .g4 [10].

Príklad 5.2.2. *Príklad definovania súboru **AHLLLexer.g4** s lexikálnymi pravidlami.*

```
lexer grammar AHLLLexer;
```

5.2.2 RSTA, AutoComplete

Na základe zvoleného implementačného jazyka Java a knižnice Swing bolo možné nájsť open source komponenty uľahčujúce editovanie a štylizovanie textu. Pri analýze možných riešení bolo nájdených mnoho frameworkov, do užšieho výberu sa však dostali len dva – Xtext a RSTA (RSyntaxTextArea).

Xtext[2] je framework, ktorý pre svoju činnosť potrebuje gramatiku a ako výsledok ponúkne celú infraštruktúru editoru zahrňujúcu parser, linker, kontrolu typov, prekladač, zvýraznenie syntaxe, ponuku kľúčových slov a mnoho ďalšieho.

Druhý variant (RSTA) ponúka textový Swing komponent rozširujúci triedu `JTextArea`. Implicitne podporuje zvýraznenie syntaxe pre viac ako 40 jazykov a ponuku kľúčových slov pre Java, JavaScript, PHP atď. Povoľuje taktiež definíciu vlastného spôsobu spracovania textu v textovej časti, čiže napríklad použiť vygenerovaný lexikálny analyzátor a jednotlivým tokenom priradiť rôzne štýly fontu [2]. Jej sesterský projekt AutoComplete disponuje rozhraním pre ponuku kľúčových slov [1], a tým pádom je možné poskladať editor podľa seba, a takmer od základov. Kvôli tomu bol za výsledný produkt použitý v tejto práci zvolený RSTA spolu s AutoComplete, oba distribuované pod upravenou BSD licenciou.

5.2.3 JTattoo

JTattoo pozostáva z niekoľkých rôznych Look and feel[3] pre Swing aplikácie. Programátorovi je poskytnutý vzhľad jednotlivých prvkov rozdielny od toho klasického pribaleného v štandardnom JDK. Je distribuovaný pod licenciou GNU General Public License v3. Zaujímavými vlastnosťami frameworku sú [3]:

- Interaktívne správanie prvkov a rollover efekty
- Vysoký výkon kvôli optimalizovaným vykreslovacím algoritmom
- Nízka veľkosť frameworku obsahujúceho všetky poskytnuté vzhľady

5.3 Úlohy editoru a návrh GUI

Pri návrhu jednotlivých funkcií a grafických komponentov hrala podstatnú rolu analýza existujúcich editorov a vývojových prostredí (kapitola 4). Každý z nich ponúka užívateľovi nástroje špecifické pre to, k čomu je daná aplikácia určená. Nasledujúce kapitoly pojednávajú o funkciách, ktorými má výsledný editor disponovať, a o návrhu grafického rozhrania slúžiacom na komunikáciu užívateľa s procesmi na pozadí.

5.3.1 Editovanie textu

Písanie a upravovanie zdrojového textu bude podporovať nasledujúce funkcie:

Práca so súbormi

Pri vytvorení nového súboru bude inicializovaný komponent typu `RSyntaxTextArea`. Potom môže užívateľ využívať jeho textovú časť na písanie zdrojového kódu. Pri otvorení už existujúceho súboru je takisto vytvorený tento komponent, avšak je mu priradený odkaz

[2] Xtext - https://eclipse.org/Xtext/
[3] LaF - https://en.wikipedia.org/wiki/Look_and_feel

na daný súbor a po vytvorení textového komponentu je následne skopírovaný obsah súboru do textovej časti metódou `setText()`.

Vďaka uloženiu odkazu na súbor je možné progres v editácii pravidelne ukladať. Pri pokuse o uloženie obsahu RSTA bez priradeného súboru je možné vytvoriť súbor s požadovaným menom a obsah textovej časti je doň skopírovaný. Pri snahe o kompiláciu obsahu takéhoto komponentu sa daná akcia nevykoná, nakoľko prekladaču nie je možné na vstup poslať neexistujúci súbor.

Pri použití vhodného grafického komponentu je možné mať otvorených viac súborov naraz, ukladať a zatvárať ich súčasne. Pred zavretím RSTA alebo ukončením aplikácie je nutné skontrolovať, či sa v nejakom z textových polí nenachádzajú neuložené zmeny (text v komponente bez súboru, obsah súboru sa nezhoduje s obsahom v textovej časti). V tom prípade je nutné zobraziť dialógové okno s možnosťou uložiť tieto zmeny.

Zvýraznenie syntaxe

Zásadným prvkom pri písaní zdrojového kódu je zvýraznenie syntaxe. Hlavnú rolu hrá trieda implementujúca rozhranie `TokenMaker` (súčasť RSTA) s metódou `getTokenList()`. Tá ma za úlohu rozdeliť aktuálne editovaný riadok na tokeny [2], pričom bude možné využiť vygenerovaný lexer.

Zvýraznené budú kľúčové slová jazyka, ako aj identifikátory premenných a funkcií. Slová nevyhovujúce žiadnym lexikálnym pravidlám budú ostro zvýraznené a podčiarknuté pre možnosť rýchlej úpravy programátorom.

Výpis definovaných plánov a služieb

Parsovanie textu na tokeny je možné využiť nie len na zvýrazňovanie textu, ale taktiež aj na výpis deklarovaných plánov a služieb platformy. Je nutné vytvoriť `DocumentListener`, ktorý pri každej editácii prechádza obsah dokumentu kvôli novému obsahu tokenov. Pokiaľ poradie tokenov odpovedá konštrukcii definície plánu alebo služby, je automaticky pridaný bez duplicít do zoznamu. Každý plán a služba vyskytujúca sa v tomto zozname bude slúžiť ako odkaz do textovej časti na miesto deklarácie, čiže pri ukladaní tokenov do zoznamu je nutné zaznamenať aj jeho polohu v textovej časti.

Ponuka kľúčových slov a konštrukcií

Pomocou projektu AutoComplete je možné zostaviť mechanizmus na inteligentnú ponuku kľúčových slov. Nakoľko nemá k dispozícii strom generovaný syntaktickým analyzátorom, vynahrádza tento nedostatok možnosťou ponuky slov na základe toho, či sa doplnenie vyžaduje v lexikálnom pravidle komentára alebo na doplnenie kľúčového slova či celej konštrukcie. Pri kľúčových slovách sa jedná o jednoslovné výrazy, zatiaľ čo pri definovaní main-u, plánu či služby platformy je doplnený celý blok. Pri takomto doplnení je vhodné, aby sa kurzor vrátil na miesto pre doplnenie mena plánu. Túto funkciu AutoComplete nezvláda, preto bude nutné implementovať danú funkciu od základov ručne.

Editor by mal taktiež podporovať ponuku užívateľom definovaných identifikátorov premenných, plánov a služieb platformy. Táto vlastnosť tiež nie je implicitne podporovaná. Novovzniknuté identifikátory budú následne pridávané do zoznamu ponuky kľúčových slov zobrazenej kombináciou kláves CTRL+SPACE.

Zo zadania plynie, že služby platformy majú byť k dispozícii na základe toho, ktorá platforma je zvolená, avšak po novom sa definujú priamo v AHLL kóde agenta, tým pádom sú dynamicky pridávané rovnakým spôsobom ako plány.

Hľadanie a nahradenie

Neodmysliteľnou súčasťou editora je funkcia hľadania a nahradenia slov, alebo ich častí. RSTA ponúka engine pre hľadanie vpred, vzad, nahradenie jedného alebo všetkých výskytov výrazu. Hľadaný vzor je možné definovať priamo, alebo pomocou regulárneho výrazu. Taktiež je možné nastaviť hľadanie na rozlišovanie malých a veľkých písmen (case sensitive) a porovnávanie na zhodu celých slov.

Ostatné funkcie

RSTA ponúka rôzne metódy implementujúce často používané akcie pre editovanie textu, ako napríklad operácie *undo, redo, copy, paste, cut* a podobne. Ich invokáciu je možné zaistiť známymi klávesovými skratkami.

Po naštudovaní zdrojových kódov RSTA bolo zistené, že je možné taktiež implementovať tzv. code folding[4], zvýrazňovanie párových hranatých, okrúhlych či zložených zátvoriek alebo taktiež odsadzovanie textu pri prechodoch medzi úrovňami zanorených blokov [2]. Keďže tieto funkcie závisia na type jednotlivých tokenov (napríklad code folding súvisí so zloženými zátvorkami), je nutné rozšíriť triedy a preťažiť metódy implementujúce spomenuté chovanie tak, aby typy tokenov používaných frameworkom RSTA boli adekvátne k typu tokenov generovaných z nami použitého lexikálneho analyzátoru. Druhým spôsobom je definovanie lexikálnych pravidiel rovnakým spôsobom, akým sú definované pre implicitne podporované jazyky.

Príklad 5.3.1. *Príklad pridania podpory pre parsovanie textu zadaného do dokumentu RSTA.*

```
AbstractTokenMakerFactory atmf =
    (AbstractTokenMakerFactory)TokenMakerFactory.getDefaultInstance();
atmf.putMapping("text/ahll", "ahll.editor.AHLLTokenMaker");
```

`ahll.editor.AHLLTokenMaker` *je cesta k triede implementujúcej rozhranie* `TokenMaker` *a* `text/ahll` *je reťacový popis jazyka.*

5.3.2 Preklad

V úvode bolo spomenuté, že AHLL editor musí podporovať preklad zdrojových textov do cieľového jazyka ALLL. Pre splnenie tejto úlohy je plánované použiť prekladač vyvinutý, a ďalej rozšírený Ing. Róbertom Kalmárom v jeho diplomovej práci [7], z ktorej vychádza táto podsekcia. Jeho forma bude ďalej popísaná a naznačená na obrázku 5.1.

Kompilátor na vstupe očakáva zdrojový text jazyka AHLL. Vstup je najprv spracovaný preprocesorom povoľujúcim vkladanie a rozvinutie hlavičkových súborov s validným AHLL kódom. Tieto hlavičky sú predovšetkým určené k abilite definovania plánov alebo služieb platformy.

V ďalšej fáze je výstup preprocesoru odoslaný na spracovanie lexikálnym a syntaktickým analyzátorom. Prvý menovaný spracuje zdrojový kód na postupnosť tokenov a prepošle ich

[4] Code folding - https://en.wikipedia.org/wiki/Code_folding

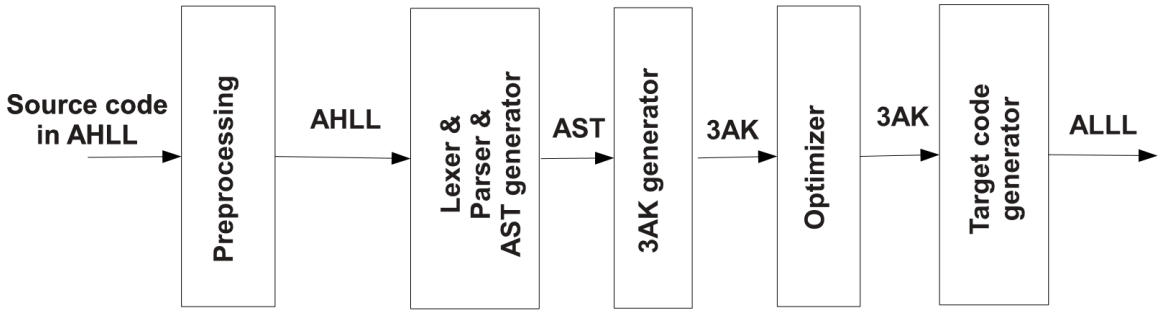
na vstup parseru. Jeho úlohou je vytvorenie abstraktného syntaktického stromu (AST[5]). Obe časti boli generované nástrojom ANTLR. Vytvorený AST je v generátore 3-adresného kódu (3AK) transformovaný na 3AK.

Očakáva sa, že editor bude sám voliť druhy optimalizácií tak, aby výsledný kód mal čo najmenšiu veľkosť. Existujúci kompilátor poskytuje 4 druhy optimalizácií: vyhodnotenie konštantných výrazov, propagácia konštánt, propagácia kópií a eliminácia mŕtveho kódu. Nakoľko u terajšieho prekladača sú implicitne zvolené všetky optimalizácie a nie je možné ich voliť spúšťacími parametrami, je nutné ich to prekladača pridať. Pre korektné fungovanie takéhoto postupu je potrebné, aby všetky optimalizačné moduly boli schopné produkovať správnu formu 3AK. Ten je prezentovaný ako určitá forma algoritmov určených k prekladu. Vychádzajú zo syntaktických konštrukcií AHLL, avšak autor prekladača mal snahu priblížiť sa taktiež forme cieľovému jazyku prekladu. 3AK je reprezentovaný inštrukciami, ktoré sú rozdelené na 4 druhy:

- matematické, logické a relačné inštrukcie
- riadiace inštrukcie
- skokové inštrukcie
- ostatné inštrukcie

Vstup a výstup optimalizátoru je 3AK, takže optimalizácie je taktiež možné z prekladu úplne vynechať. Tým pádom je možné spustiť samotný preklad bez aplikovania všetkých kombinácií optimalizácií a porovnávania jednotlivých výsledkov kvôli výberu toho najefektívnejšieho, čo samozrejme urýchli samotný preklad.

Pri syntaktickej chybe podáva kompilátor konkrétne informácie o chybe spolu s číslom riadku a stĺpca, na ktorom sa nachádza. V editore je tým pádom vhodné užívateľovi túto informáciu podať určitou formou výpisu do stavového riadku.



Obr. 5.1: Štruktúra prekladača. Prevzaté z [7].

5.3.3 Integrovanie do T-Mass

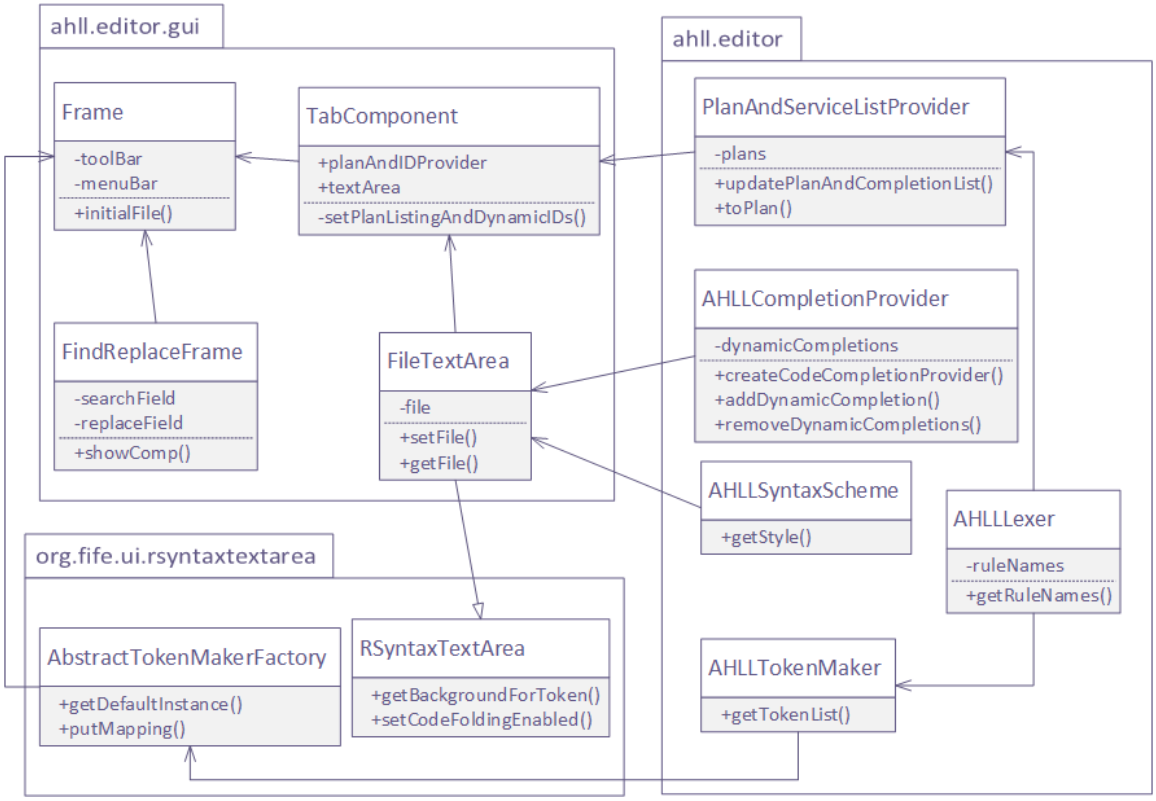
Integrácia do simulačného nástroja T-Mass je navrhnutá tak, že program editoru bude možné spustiť priamo zo simulátoru. Preto bude nutné pridať editoru funkciu spracovania parametrov. Parametrami budú cesty k súborom, a nakoľko je editor navrhnutý tak, že je možné editovať viac súborov súčasne, je možné zadať taktiež viac parametrov.

[5] Abstract syntax tree

Korektný beh simulácie vyžaduje 3 druhy súborov:

- ALLL súbor s kódom agenta
- TML (T-Mass Modelling Language) definujúci elementy simulácie - agentov, platfory, prostredia a ich vzťahy
- PD definujúce platformu a jej služby

Súbor ALLL je vytvorený editorom hned po preklade zdrojového AHLL. Súbory TML a PD je nutné vytvoriť ručne, pričom je potreba kontrolovať, či výsledný ALLL kód nevolá služby, ktoré T-Mass nepodporuje. Príkladom takého kódu je napríklad testovací súbor walkThrough.ahll prevzatý z práce [7] a spomenutý v kapitole testovania 6.7. Nakoľko prekladač generuje rovnakú verziu ALLL jazyka, aká je implementovaná v simulátore [7], je integrácia plne validná s ohľadom na vyššie spomenutú absenciu niektorých služieb v simulátore.



Obr. 5.2: Minimalistická verzia konceptuálneho návrhu tried.

5.3.4 Grafický návrh

Návrh grafického užívateľského rozhrania bol zo značnej miery inšpirovaný existujúcimi editormi. Rokmi osvedčené rozloženie prvkov má výhodu v tom, že si programátor nemusí zvykať na nové prvky a praktiky, pracuje s niečím, čo pozná a rýchlo sa v tom orientuje.

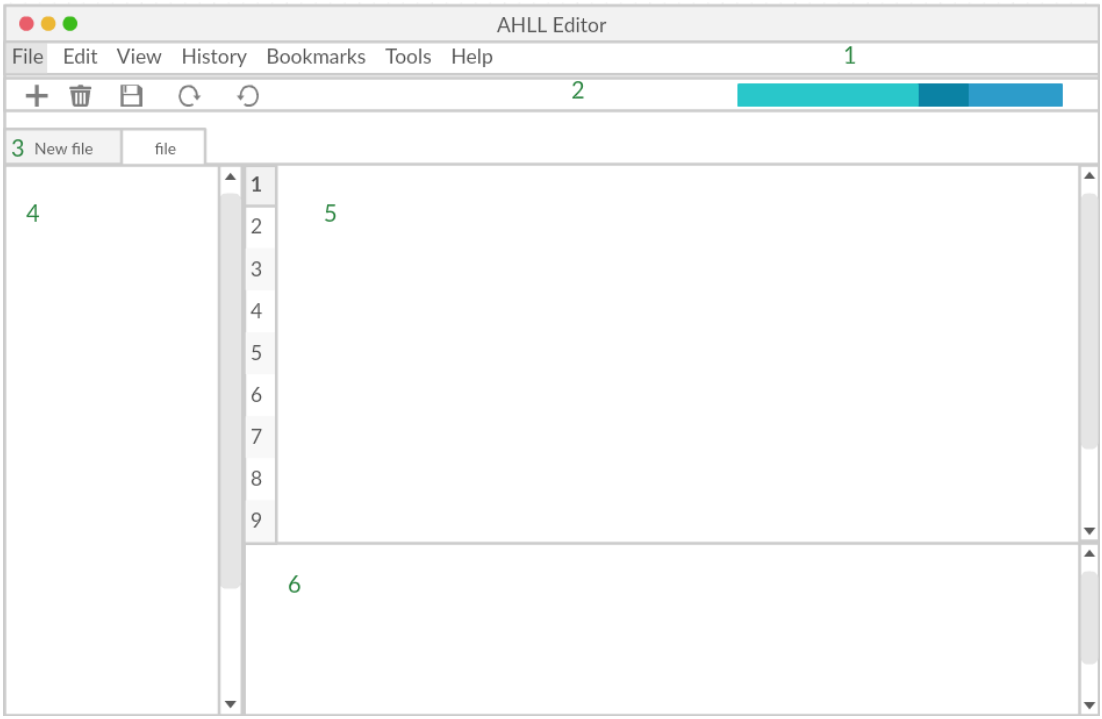
Nasledujúci popis vychádza z obrázku 5.3. Statická časť rozhrania zahŕňa klasické menu(1) s pokročilými možnosťami, z ktorých sú niektoré vo forme tlačidiel na panely

nástrojov(2) umiestneného pod menu pre rýchlejší prístup. Akcie sú na tlačidlách reprezentované obrázkami a ponúkajú taktiež textovú nápovedu, resp. tooltip. Tento panel obsahuje okrem iného aj komponent pre zobrazenie priebehu prekladu s optimalizáciami, aby bola užívateľovi poskytnutá spätná väzba o procesoch na pozadí, a tým zvýšená viditeľnosť stavu systému. Často používané akcie ponúknuté v menu je taktiež možné vykonať pomocou klávesových skratiek, pre príklad otvorenie nového súboru, uloženie atď. Klávesové kombinácie sú zobrazené po rozkliknutí prvkov menu.

Dynamická časť disponuje tabuľkovým panelom(3), ktorý obsahuje karty s jednotlivými otvorenými súbormi. Karta pozostáva z názvu určeného podľa súboru, ľavého panelu(4) na zobrazenie definovaných plánov a služieb, hlavnej textovej časti(5) a spodného panelu(6) pre vypísanie informácií o preklade.

Pri akcii *otvoriť súbor* alebo *uložiť ako* je zobrazené nemodálne okno s obsahom štruktúry súborového systému. Pri požiadavke na hľadanie slov je zobrazené naopak modálne okno nakoľko chceme, aby užívateľ mohol interagovať s oknom aplikácie. Pri stratení zamerania nesmie byť okno skryté, tento jav nastane až po zavretí okna. Tiež je potreba zaistiť, aby v jednom okamihu mohol byť otvorený len jeden takýto komponent.

Výhodou je, ak prostredie užívateľovi umožňuje nastaviť veľkosť jednotlivých komponentov. Napríklad v prípade maximalizácie textovej plochy pre možnosť zobrazenia čo najväčšieho počtu riadkov je možné odsunúť panel nástrojov a zmenšiť veľkosť stavového riadku.



Obr. 5.3: Mockup grafického rozhrania editoru.

Kapitola 6

Implementácia a testovanie

Implementačná časť prebiehala v niekoľkých fázach. Najprv sa jednalo o implementáciu lexikálných pravidiel a prepojenie vygenerovaného lexera s dokumentom z RSTA. Toto vyžadovalo kompletné naštudovanie dokumentácií oboch frameworkov. Následne bola snaha o implementáciu syntaktických pravidiel a teda možnosť syntaktickej kontroly za behu. Túto časť sa však nepodarilo dokončiť kvôli nedostatku zdrojov v dokumentácii RSTA, tým pádom prebieha kontrola len na lexikálnej úrovni.

Neskôr boli pridávané ďalšie funkcie plynúce zo zadania, ako ponuka kľúčových slov a konštrukcií, ďalej výpis plánov a služieb. Nasledovala integrácia prekladača, čo zahrňovalo pridanie spúšťacích parametrov do jeho zdrojových kódov a na základe nich výber požadovaných optimalizácií. Ďalej boli implementované zvyšné komponenty grafického rozhrania so službou vyhľadania a nahradenia, a ako posledná integrácia do simulátoru T-Mass.

6.1 Parsovanie tokenov

`AHLLLexer` je trieda vygenerovaná frameworkom ANTLR reprezentujúca lexer parsujúci vstupný text na tokeny. Generovaná je z lexikálnych pravidiel definovaných v súbore AHLLLexer.g4 a uvedených v prílohe B. Získanie tokenov zo zdrojového kódu je využité na 2 hlavné činnosti editoru.

Zvýraznenie syntaxe a odsadenie blokov

Výstup lexikálnej analýzy je spracovaný v `AHLLTokenMaker`, kde sú jednotlivé tokeny uložené do zreťazeného zoznamu pre možnosť ďalšieho spracovania. To zaisťuje preťažená metóda `getTokenList()`. Jej výstup je upravený konverziou tokenov na objekty korešpondujúce tokenom, s ktorými pracuje RSTA. Typy jednotlivých tokenov sú kontrolované v `AHLLSyntaxScheme`, kde sa následne aplikuje štýl pre zobrazenie v textovej časti (farba fontu, podčiarknutie atď), pričom sú zmeny pri zadávaní textu okamžite vykreslené. Neznámy typ tokenu je zvýraznený červenou farbou a podčiarknutý.

Úlohou spomenutej triedy na spracovanie tokenov je tiež rozpoznanie tokenu začiatočnej zloženej alebo okrúhlej zátvorky, čo slúži napríklad na odsadenie textu tabulátormi pri jednotlivých prechodoch do nižších úrovní zanorených blokov.

Pri definovaní lexikálnych pravidiel boli jednotlivé tokeny písané v rovnakom poradí, ako sú definované pre jazyky implicitne podporované RSTA. Týmto bolo dosiahnuté rozpoznanie tokenov zložených zátvoriek pre funkčný code folding, a tiež ostatných zátvoriek

pre zvýraznenie ich párov. Túto vlastnosť interne zaisťuje `CurlyFoldParser` (z RSTA), ktorý zaisťuje rozdielnu podporu od napríklad značkovacích jazykov.

Výpis plánov a služieb

Úlohou triedy `PlanAndServiceListProvider` je aktualizovať zoznam definovaných plánov a služieb. V konštruktore je inicializovaná premenná reprezentujúca zoznam plánov typu `HashSet`. Nakoľko do nej nie je možné pridávať duplicity, pri pokuse o pridanie už existujúceho identifikátoru končí operácia neúspechom (viacnásobný výskyt definície rovnakého plánu, resp. služby je nežiadúci). Metóda `updatePlanAndCompletionList()` zaisťuje parsovanie dokumentu lexerom, vymazanie prvkov zo zoznamu (v každom cykle sú vymazané a nanovo pridané kvôli získaniu najaktuálnejšej verzie - inak je veľká šanca výskytu neaktuálnych hodnôt) a následnom prechode zoznamu tokenov. Pri nájdení tokenu `plan` (`service`) nasledovaného neobmedzeným počtom bielych znakov, a následne tokenom identifikátoru je tento identifikátor pridaný do zoznamu. Do zoznamu je tiež pridávaný aj hlavný plán `main`. Spolu s názvom identifikátorov je uložená aj pozícia definície, nakoľko každá položka zoznamu slúži ako odkaz do textovej časti (kliknutím na položku zoznamu je kurzor presunutý do dokumentu na miesto definície).

V cykle realizujúcom tento prechod je tiež kvôli zvyšeniu výkonu pridaná aktualizácia CompletionProvider-u spomenutá v nasledujúcej podsekcii (nie je nutné znova iterovať cez všetky tokeny).

6.2 Ponuka jazykových konštrukcií

Ponuku slov riadi trieda `AHLLCompletionProvider` (rozširujúca bázovu triedu z AutoComplete), ktorá rozpoznáva sekciu kódu, v ktorej sa nachádza (komentár, reťazec...) a na základe toho vyberá slová do provider-u. Ten je zobrazený klávesovou skratkou CTRL+SPACE.

Rozšírením triedy `ShorthandCompletion` a `AutoCompletion` (obe z AutoComplete) a preťažením konštruktorov s pridaním privátneho slotu pre uloženie pozície kurzoru bolo dosiahnuté to, že pri požiadavku na doplnenie plánu alebo služby je doplnená celá jazyková konštrukcia a kurzor je po doplnení vrátený na pozíciu, kde je požadovaný identifikátor plánu, resp. služby.

Dynamické pridávanie deklarovaných identifikátorov plánov, služieb a premenných nie je implicitne podporované, preto bolo nutné zaistiť túto funkciu pridaním metódy s parametrom identifikátora, ktorý je následne pridaný do abstraktnej dátovej štruktúry `HashSet` (z rovnakého dôvodu ako pri výpise plánov). Návratová hodnota akcie je použitá ako podmienka pre vytvorenie a pridanie doplnenia do provider-u.

Pri upravovaní dokumentu je nutné dynamicky pridané identifikátory vymazávať a znova pridávať (tak, ako pri výpise plánov) kvôli nežiadúcim efektom, ktoré môžu v opačnom prípade nastať. Tento spôsob je neefektívny, avšak rozdiel nie je pocitovať ani pri väčších dokumentoch.

Funkčnú ukážku ponuky slov možno vidieť na obrázku 6.3.

6.3 Preklad a optimalizácie

Preklad realizuje trieda `CompileAction`, ktorej hlavná metóda spúšťa .jar súbor s prekladačom jazyka AHLL. O spracovanie výstupu kompilátoru sa stará trieda `CompilationLog`, ktorá nielen ukladá výstup v podobe ALLL kódu do novo vytvoreného súboru v aktuálnom

adresári, ale taktiež prijíma chybový výstup a poskytuje ho užívateľovi ako spätnú väzbu do stavového riadku (chýbajúci prekladač, syntaktická chyba atp.).

Pri spustení prekladu s možnosťou nájdenia najlepšej kombinácie optimalizácií je invokovaná hlavná metóda triedy `OptimizeCompileAction`. Tá funguje taktiež ako observer[1] pre zobrazenie, aktualizáciu a skrytie panelu zobrazujúceho progres porovnania výstupov prekladača (užívateľ vie, že systém nezamrzol, ale pracuje na pozadí).

Pri práci na spúšťacích parametroch prekladača bolo zistené, že pri zadaní ktoréhokoľvek variantu optimálizácii, v ktorom sa nevyskytovala možnosť propagácie konštánt nebol prekladač schopný vygenerovať validný kód, prípade celý program spadol. Kvôli tomu je pri každom spustení natvrdo použitá propagácia konštánt (s výnimkou spustenia bez optimalizácií). Celkovo teda existuje 9 kombinácií zložených z nasledujúcich možností: vyhodnotenie konštantných výrazov, propagácia konštánt, propagácia kópií, eliminácia mŕtvych inštrukcií, žiadne optimalizácie.

`OptimizeCompileAction` v každom cykle porovná dĺžku kódu práve spusteného prekladu (hodnotu poskytuje trieda `CompilationLog`) s doterajšou najmenšou hodnotou. Ak je aktuálna hodnota menšia, je nastavená ako najmenšia a index kombinácie optimalizácií je uložený pre finálne spustenie prekladu s týmito optimalizáciami a následné uloženie do súboru. Pokiaľ vznikne rovnako dlhý kód vo viacerých iteráciách, za výslednú kombináciu je zvolená tá s najmenším počtom aplikovaných optimalizačných techník. V prípade chyby prekladu je cyklus okamžite ukončený a popis chyby je užívateľovi poskytnutý rovnako ako pri klasickom preklade.

6.4 Funkcia hľadania a nahradenia

Ako už bolo spomenuté, vyhľadávací engine je v RSTA už podporovaný, bolo však nutné ošetriť niektoré chyby a vytvoriť rozhranie pre komunikáciu s užívateľom. Pre zobrazenie okna vyhľadávania (obrázok 6.2) bola zvolená klávesová skratka CTRL+F (možné spustiť aj z menu alebo panela nástrojov). Pri návrhu okna bol zvolený návrhový vzor Singleton[2], nakoľko je nutné predísť vzniku viacerých vyhľadávacích okien. Okno rozširuje knižničnú Swing triedu `JInternalFrame` a obsahuje polia na zadanie hľadaného vzoru, reťazca nahradenia, možnosť zvolenia aplikovania možností: regulárny výraz, rozlišovanie malých a veľkým písmen, celé slová.

Tieto možnosti sú aplikované do vyhľadávacieho kontextu (z RSTA) a po zvolení operácie (*ďalšie, predchádzajúce, nahradiť, nahradiť všetky*) je spustený engine. V ňom boli upravené niektoré vlastnosti:

- Po neúspechu hľadania vpred (vzad) je vyhľadávanie presunuté na začiatok (koniec) dokumentu a hľadanie pokračuje.

- Pri nahradení je implicitne nastavené hľadanie vpred s následným nahradením. Pokiaľ nebolo nájdené, resp. nahradené žiadne slovo, nasleduje hľadanie vzad a následný pokus o nahradenie.

[1] Observer - https://en.wikipedia.org/wiki/Observer_pattern
[2] Singleton pattern - https://en.wikipedia.org/wiki/Singleton_pattern

6.5 Integrovanie T-Mass

V návrhu bolo spomenuté, že integrácia simulátoru bude spočívať hlavne v možnosti spustenia procesu editoru zo simulačného nástroja. Do editora bolo teda pridané spracovanie spúšťacích argumentov. Tie sú uložené v poli a môže ich byť zadané neobmedzené množstvo. Parametre reprezentujú absolútne, resp. relatívne cesty k súborom, ktoré sú určené k instantnému otvoreniu v editore. Nie je teda potreba po spustení editoru otvárať požadované súbory, tie sú otvorené automaticky pri spustení. Editor je exportovaný vo forme spustiteľného JAR súboru, ktorý je spustený po zadaní požadovaných súborov priamo zo simulátoru.

Do budúcna sa plánuje iný spôsob prepojenia týchto dvoch nástrojov s pridaním možnosti poskytnutia zoznamu implementovaných služieb v AHLL kóde, a následnej kontrole prítomnosti týchto služieb v nástroji T-Mass. Alternatívou je tiež zohľadnenie služieb implementovaných v T-Mass na strane editora, za ktorým by nasledovalo upozornenie užívateľa v prípade použitia nedostupnej služby.

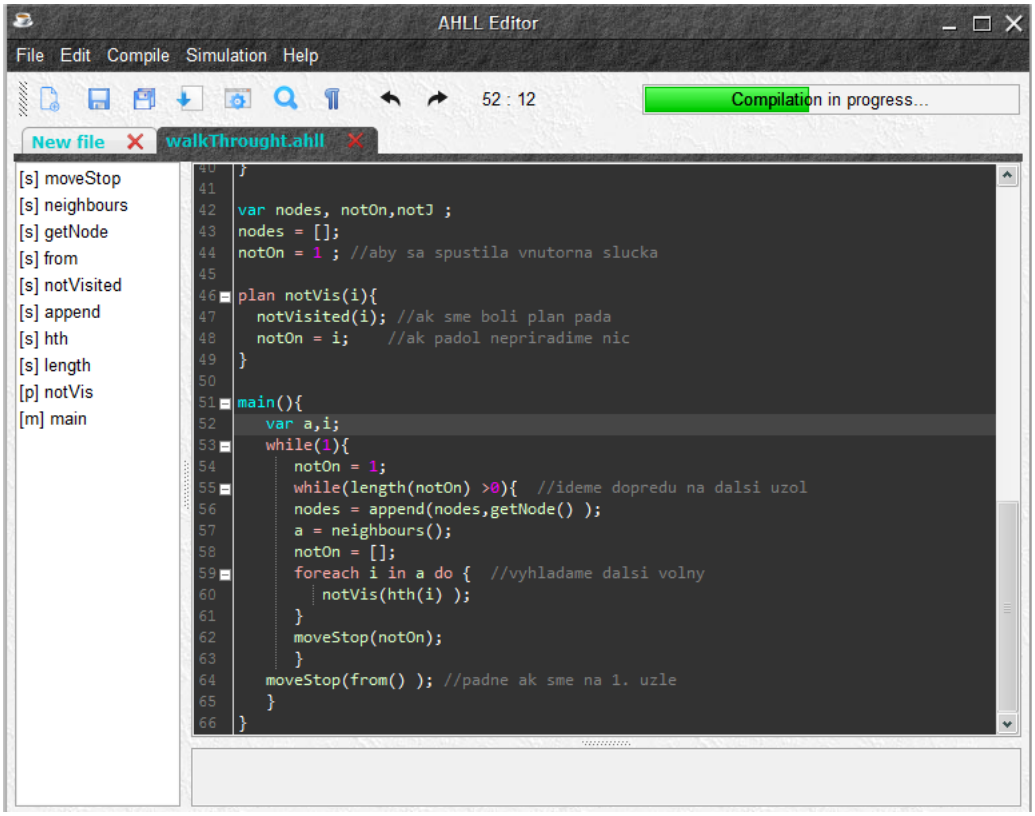
6.6 GUI

Trieda `Frame` reprezentuje hlavné okno aplikácie. Obsahuje hlavnú metódu `main`, v ktorej je explicitne nastavený Look and Feel programu aplikovaním témy z JTattoo (podsekcia 5.2.3). Taktiež obsahuje globálne nastavenia spracovaného jazyka pre RSTA, ako napríklad definovanie nami implementovaného `AHLLTokenMaker` a použitie `CurlyFoldParser`.

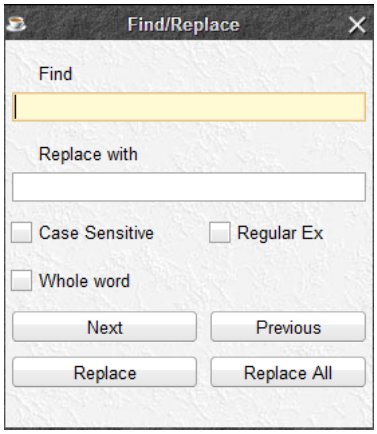
Nasledujúci popis vychádza z obrázku 6.1. Okno obsahuje panel menu (`MenuBar`), ktorý zobrazuje vačšinu nástrojov podporovaných editorom. Jediné vynechané sú akcie spojené s textovou časťou - tie je možné zobraziť pravým klikom v dokumente. Panel nástrojov pod menu (`ToolBar`) zobrazuje niektoré z najpoužívanejších akcií vo forme tlačidiel - vytvorenie nového súboru, preklad atď. Na ich aktiváciu reagujú triedy v balíčku `ahll.editor.actions`. Toolbar rovnako obsahuje časť pre výpis riadku a stĺpca, v ktorom sa nachádza kurzor (zobrazí sa až po otvorení prvej karty), a už spomínaný panel s progresom aplikovania optimalizácií (zobrazený iba pri zvolení akcie optimalizovania).

Hlavnú časť editora tvorí `JTabbedPane` z knižnice Swing. Doň sa pri vytvorení dokumentu pridávajú karty s panelmi pre výpis definovaných plánov a služieb (vľavo), stavový riadok pre výpis informácií o preklade(naspodu) a hlavnú textovú časť reprezentovanú triedou `FileTextArea`. Tá rozširuje bázovú triedu z RSTA kvôli možnosti priradenia súborov jednotlivým kartám. Odkazy na súbory priradené kartám sú využívané na priebežné ukladanie textu do súborov alebo kontrolu aktuálnosti zadaného textu v textovej časti s obsahom súboru. Pokiaľ sa obsahy nezhodujú, je pri zatvorení karty, alebo celej aplikácie užívateľovi poskytnutá možnosť zmeny uložiť alebo zahodiť.

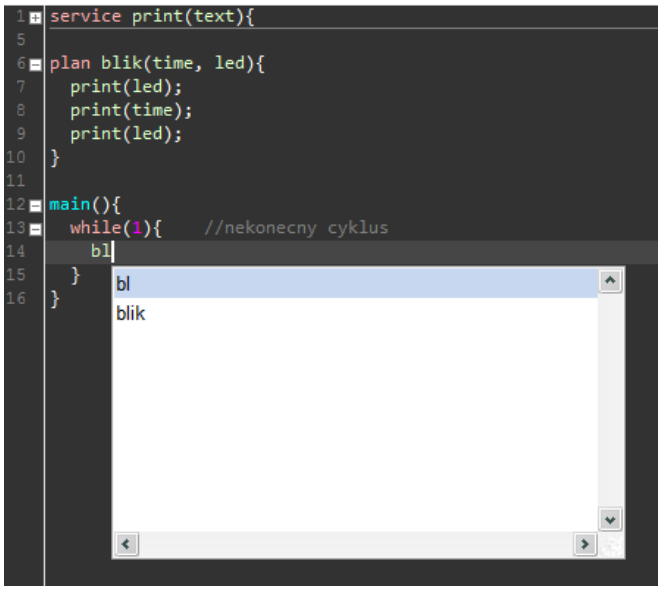
Na záhlavie karty bola potreba pridať vlastný panel s názvom súboru a tlačidlom zatvorenia, nakoľko Swing toto tlačidlo implicitne neposkytuje. Všetky obrázky použité pri vytvorení grafického rozhrania (prevažne tlačidiel) spadajú pod licenciu Free for commercial use. Licencie k open source projektom použitým v práci sú priložené v súbore LICENSE.txt.



Obr. 6.1: Screenshot výslednej aplikácie



Obr. 6.2: Okno vyhľadávania.



Obr. 6.3: Ukážka code folding-u a ponuky slov.

6.7 Testovanie

Testovanie aplikácie prebiehalo na systéme Windows 10 s verziou JRE 1.8, a to vo viacerých smeroch - testy výberu vhodnej kombinácie optimalizácií na zdrojový kód a užívateľského rozhrania - správne fungovanie jednotlivých modulov programu. Testy rozhrania prebiehali vykonávaním klasických operácií (uloženie súboru, preklad...) a prípadné chyby boli ihneď odstránené. Testovanie použitia optimalizácií sa opiera o fakt, že výstupom kompilátora je validný ALLL kód.

Preklad s optimalizáciami

Pri akcii prekladu so zameraním na výber najvhodnejšej kombinácie optimalizácií boli využité ukážkové súbory na výpočet faktoriálu (faktorial.ahll), blikanie LED diódiami (blikNew.ahll) a priechod agenta sieťou (walkThrough.ahll), ktoré boli prevzaté z práce [7] a sú priložené k odovzdávanému archívu.

Z výsledkov v tabuľke 6.1 je možné vidieť, že pri rôznych kódoch bola vybraná iná kombinácia optimalizácií. Ako už bolo spomenuté, pri zhodnej veľkosti výsledných kódov vyberie editor tú s najmenším počtom optimalizácií.

Význam skratiek použitých v zhrnutí výsledkov:

NONE - bez optimalizácií

CF (Constant Folding) - vyhodnotenie konštantných výrazov

CCP (Conditional Constant Propagation) - propagácia konštánt mŕtvych inštrukcií

CP (Copy Propagation) - propagácia kópií

DCE (Dead Code Elimination) - eliminácia mŕtveho kódu

	faktorial		**blikNew**		**walkThrough**	
Optimalizácie	dĺžka	výber	dĺžka	výber	dĺžka	výber
NONE	465	*	294		1244	
CCP	465		258	*	1208	
CCP, CF	465		258		1208	
CCP, CP	465		258		1209	
CCP, DCE	465		258		1146	
CCP, CF, CP	465		258		1209	
CCP, CF, DCE	465		258		1146	
CCP, CP, DCE	465		258		1089	*
CCP, CF, CP, DCE	465		258		1089	

Tabuľka 6.1: Výsledky aplikovania optimalizácií.

Kapitola 7

Záver

Cieľom tejto práce bolo vytvoriť inteligentný editor uľahčujúci programovanie v agentnom jazyku AHLL. Pre porozumenie konceptu agentného programovania bolo nutné naštudovať si teoretické princípy spojené s multiagentnými systémami. Ďalej boli podrobne naštudované materiály popisujúce syntax jazyka AHLL, ako aj jeho cieľového jazyka prekladu ALLL.

Návrh editora vychádza z analýzy existujúcich editorov a vývojových prostredí. Hlavnými funkcionálnymi požiadavkami editora boli jednoduchá práca so súbormi, zvýraznenie syntaxe, textové operácie (undo, redo...), ponuka kľúčových slov a konštrukcií, možnosť skrytia a zvýraznenia blokov kódu atď. Podrobnejší popis všetkých implementovaných funkcií sa nachádza v kapitole 6.

Výsledná implementácia tiež integruje existujúci prekladač vytvorený Ing. Róbertom Kalmárom. Ten realizuje preklad z jazyka vyššej úrovne abstrakcie do cieľového jazyka ALLL. Poskytuje pár optimalizačných techník, ktoré je možné kombinovať medzi sebou. Editor tiež zvláda funkciu výberu najvhodnejšej kombinácie optimalizácií na základe dĺžky výsledného kódu. Hlavným krokom práce bola tiež integrácia do nástroja T-Mass, určeného na tvorbu a simuláciu distribuovaných systémov.

Predmetom budúcej práce na editore je jednoznačne lepšia interakcia simulátoru T-Mass so samotným editorom. To predstavuje napríklad poskytnutie informácií o službách implementovaných simulátorom a možnosť ich využitia pri programovaní v editore. Takisto by bolo vhodné implementovať podporu zvýraznenia syntaktických chýb, nakoľko je táto funkcia implementovaná len na lexikálnej úrovni. Na strane prekladača je v pláne jeho autora implementovať väčšie množstvo optimalizačných algoritmov, čím by sa tiež zlepšil výsledok práce editora pri ich kombinovaní a aplikovaní počas kompilácie.

Literatúra

[1] Futrell, R.: AutoComplete. [Online; navštíveno 20.3.2017].
URL https://github.com/bobbylight/AutoComplete

[2] Futrell, R.: RSyntaxTextArea. [Online; navštíveno 20.3.2017].
URL https://github.com/bobbylight/RSyntaxTextArea

[3] Hagen, M.: JTattoo. [Online; navštíveno 20.3.2017].
URL http://www.jtattoo.net/index.html

[4] Herout, P.: *Učebnice jazyka Java*. České Budějovice: Kopp, třetí vydání, 2010, ISBN 978-80-7232-398-2, 144-147 s.

[5] Horáček, J.; Zbořil, F.; Gábor, M.: *WSageNt: Multiagent Platform for Wireless Sensor Networks*. FIT VUT v Brně, 2010, [Online; navštíveno 24.3.2017].
URL http://www.fit.vutbr.cz/~ihoracek/WSageNt/

[6] Kalmár, R.: *Jazyk vyšší úrovně abstrakce pro programování mobilních inteligentních agentů*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2010.
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=9625

[7] Kalmár, R.: *Optimalizace překladu agentních jazyků různé úrovně abstrakce*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2012.
URL http://www.fit.vutbr.cz/study/DP/DP.php?id=12492

[8] Kürti, S.: *Grafické vývojové prostředí agentního jazyka ALLL*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2012.
URL http://www.fit.vutbr.cz/study/DP/DP.php?id=13888

[9] Molák, J.: *Základnová stanice pro agentní platformu WSageNt s využitím GSM modulu*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2012.
URL http://www.fit.vutbr.cz/study/DP/DP.php?id=13974

[10] Parr, T.: ANTLR v4 Documentation. [Online; navštíveno 20.3.2017].
URL https://github.com/antlr/antlr4/blob/master/doc/index.md

[11] Parr, T.: ANTLR v4. 2014, [Online; navštíveno 20.3.2017].
URL http://www.antlr.org/

[12] Russel, S. J.; Norvig, P.: *Artificial Intelligence: a Modern Approach*. Upper Saddle River : Prentice Hall, třetí vydání, 2010, ISBN 978-0-13-207148-2, 34-59 s.

[13] Spáčil, P.: *Mobilní agenti v bezdrátových senzorových sítích*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2009.
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=7959

[14] Wooldridge, M.; Jennings, N. R.: Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, ročník 10, 1995: s. 115–152.

[15] Zbořil, F., Jr.; Spáčil, P.: Automata for Agent Low Level Language Interpretation. *Computer Modeling and Simulation, International Conference on*, ročník 00, 2009: s. 455–460, doi:doi.ieeecomputersociety.org/10.1109/UKSIM.2009.82.

[16] Zbořil, F., Jr.; Zbořil, F.: Simulation for Wireless Sensor Networks with Intelligent Nodes. *Computer Modeling and Simulation, International Conference on*, ročník 0, 2008: s. 746–751, doi:doi.ieeecomputersociety.org/10.1109/UKSIM.2008.56.

[17] Zbořil, F.; Kočí, R.; Zbořil, V. F.; aj.: T-Mass v.2, State of the art. In *Second UKSIM European Symposium on Computer Modeling and Simulation*, IEEE Computer Society, 2008, ISBN 978-0-7695-3325-4, str. 6.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=8720

Prílohy

Príloha A

Obsah priloženého pamäťového média

src/	zdrojové texty a ANTLR lexikálne pravidlá
lib/	použité frameworky
Resources/images	obrázky použité v GUI
Resources/lib/Compiler	AHLL prekladač
Resources/lib/T-Mass	T-Mass simulátor
text/	PDF s plagátom a bakalárskou prácou
text/src	zdrojové texty bakalárskej práce
ahlleditor.jar	spustiteľný JAR súbor
examples/	ukážkové AHLL kódy (prevzaté z práce Ing. Kalmára)
build.xml	skript Apache Ant pre preklad
LICENSE.txt	licencie použitých technológií
README.txt	súbor s informáciami o použití

Príloha B

AHLL Lexikálne pravidlá v ANTLRv4

```
COMMENT_EOL                 : '//' ~[\r\n]* channel(1) ;

SC_COMMA_DOT                : ',' |
                              ';' |
                              '.';

STR                         : '"' STR_frag '"' ;
fragment STR_frag           : ('a'..'z'|'A'..'Z'|'0'..'9')* ;

NEWLINE                     : '\r'? '\n' ;

OPERATOR                    : '||' |
                              '&&' |
                              '!=' |
                              '<'  |
                              '<=' |
                              '>'  |
                              '>=' |
                              '+'  |
                              '-'  |
                              '*'  |
                              '/'  |
                              '%'  |
                              '!'  |
                              '++' |
                              '--' ;

RESERVED_WORD               : 'if' |
                              'else' |
                              'while' |
                              'for' |
                              'foreach' |
```

```
                                'const';

RESERVED_WORD_2               : 'return';

PLAN                          : 'plan';

SERVICE                       : 'service' ;

LITERAL_NUMBER_DECIMAL_INT    : '0'..'9'+ ;

MESSAGE_ENABLE                : 'FI' |
                                'OI';

IN_DO                         : 'in' |
                                'do' ;

FUNCTION                      : 'main' |
                                'car' |
                                'cdr' |
                                'call';


NULL                          : 'null' | 'NULL' ;

ASSIGN                        : '=' ;

EQUALS                        : '==' ;

VARIABLE                      : 'var' ;

SEND                          : 'send' ;

RECEIVE                       : 'receive' ;

IDENTIFIER                    : (('a'..'z'|'A'..'Z')
                                 ('a'..'z'|'A'..'Z'|'0'..'9')*) ;

WHITESPACE                    : [\t\f ]+ -> channel(1) ;

SEPARATOR                     : '{' |
                                '}' |
                                '[' |
                                ']' |
                                ')' |
                                '(' ;

UNMATCHED                     : . -> channel(1);
```