VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

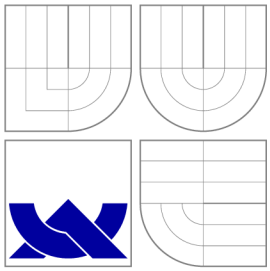
# JAZYK VYŠŠÍ ÚROVNĚ ABSTRAKCE PRO PROGRAMOVÁNÍ MOBILNÍCH INTELIGENTNÍCH AGENTŮ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

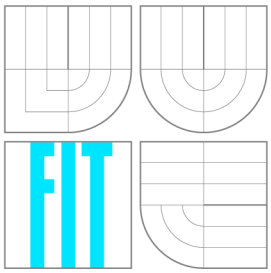
AUTOR PRÁCE
AUTHOR

RÓBERT KALMÁR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# JAZYK VYŠŠÍ ÚROVNĚ ABSTRAKCE PRO PROGRAMOVÁNÍ MOBILNÍCH INTELIGENTNÍCH AGENTŮ

LANGUAGE OF HIGHER LEVEL OF ABSTRACTION FOR PROGRAMMING MOBILE INTELLIGENT AGENTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — RÓBERT KALMÁR
AUTHOR

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

BRNO 2010

# Abstrakt

Cílem této práce je návrh jazyka vyšší urovně abstrakce pro programování mobilních inteligentních agentů a implementace překladače pro tento jazyk. Bude představen nástroj ANTLR pro generování syntaktických a lexikálnych analyzátorů. Čtěnář bude seznámen s teoretickou i praktickou stránkou implementace překladače tak jako s programovaním v tomto jazyce. V závěre práce bude představeno prostredí pro programování spomínaném jazyce a příklady agentních kódů.



# Abstract

The aim of this work is to design a language of higher level of abstraction for programming mobile intelligent agents and implement a compiler for this language. There will also be presented the ANTLR tool for generating syntax and lexical analyzers. The reader will become familiar with teorethical and practical aspects of implementation of the compiler and also with programming in this language. There will be shown the environment for programming in this language and some examles of agent codes at the end.



# Klíčová slova

Bezdrátové senzorové sítě, agent, ALLL, AHLL, překladač, ANTLR, programovací jazyk, programovacie prostredie

# Keywords

Wireless sensor networks, agent, ALLL, AHLL, compiler, ANTLR, programming language, programming environment

# Citace

Róbert Kalmár: Jazyk vyšší úrovně abstrakce pro programování mobilních inteligentních agentů, bakalářská práce, Brno, FIT VUT v Brně, 2010

# Jazyk vyšší úrovně abstrakce pro programování mobilních inteligentních agentů

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Františka Zbořila, PhD. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Róbert Kalmár
18. mája 2010

## Poděkování

V prvom rade by som sa chcel poďakovať pánovi Františkovi Zbořilovi za výborné vedenie pri tvorbe tejto bakalárskej práce, neúnavnú ochotu poradiť a pomôcť. Ďalej by som sa chcel poďakovať Pavlovi Spáčilovi za implementáciu aritmeticko-logických operácií, ako aj za pomoc pri tvorbe jazyka a jeho prekladu. V neposlednom rade ďakujem svojim rodičom a priateľom za ich neustálu podporu.

# Obsah

# Kapitola 1

# Úvod

V súčasnej dobe, vďaka rozvoju vedy a techniky, sme schopný konštruovať malé, ale relatívne výkonné mikrokontroléri umiestňované do bezdrôtových senzorových sietí. Začínajú byť dostatočne výkonné na to, aby miesto vopred naprogramovaných aplikácií na nich bežala len platforma a interpret agentného jazyka. Samotná logika aplikácie by sa jednotlivým uzlom v senzorovej sieti doručovala vo forme inteligentných agentov.

Táto práca nadväzuje na práce Ing. Jana Horáčka a Bc. Pavla Spáčila, ktorým sa podarilo implementovať funkčnú platformu a interpret agentného jazyka. Práca si dáva za cieľ pokračovať v tomto ich snažení. Budeme sa snažiť vyvinúť nový jazyk, ktorý bude mať vyššiu úroveň abstrakcie na rozdiel od spomínaného agentného jazyka. Tento nový jazyk by mal umožňovať pohodlne a plnohodnotne programovať mobilných inteligentných agentov do bezdrôtových senzorových sietí. Mal by kombinovať imperatívne a agentné princípy ale brať v úvahu aj možnosti daného agentného jazyka. K novému jazyku sa budeme snažiť implementovať prekladač tohto jazyka, kde agentný jazyk bude cieľovým jazykom tohto prekladača. Ďalej pre plnohodnotné využívanie tohto jazyka vytvoríme jednoduché prostredie so zvýraznením syntaxe apod.

V prvej kapitole zoznámime čitateľa s agentmi a bezdrôtovými senzorovými sieťami. Keďže cieľom práce je vytvorenie programovacieho jazyka a prekladača k tomuto jazyku ďalšia kapitola bude venovaná ľahkému úvodu do tvorby prekladačov. Nasledujúce kapitoly budú popisovať jednotlivé prostriedky použité v rámci tejto práce. Bude predstavený generátor lexikálnych a syntaktických analyzátorov – ANTLR a agentný jazyk ALLL. Ďalej predstavíme vyvinutý jazyk vyššej úrovne abstrakcie s podrobným popisom a vysvetlením jeho konštrukcií. Ukážeme si akým spôsobom je možné jednotlivé konštrukcie tohto jazyka zapísať v agentnom jazyku a zoznámme čitateľa so samotnou implementáciou prekladača. Záverečné časti tejto práce budú venované programovaniu v novom jazyku, hlavne z hľadiska vyvinutého prostredia a nakoniec budú nasledovať ukážkové kódy v tomto jazyku.

# Kapitola 2

# Agenti a WSN

V tejto kapitole predstavíme čitateľovi pojem agent a pojem bezdrôtových senzorových sietí – WSN. Ukážeme si niektoré príklady použitia agentov a takisto aj senzorových sietí. V poslednej časti tejto kapitoly bude uvedené možné spojenie agentov a WSN, výhody, ktoré táto integrácia môže poskytnúť, ako aj úskalia a problémy, ktoré treba mať na pamäti.

## 2.1 Agent

Pod pojmom agent si môžme predstaviť hocijakú entitu, ktorá vníma svoje prostredie, v ktorom sa nachádza, pomocou svojich senzorov a takisto mení toto prostredie pomocou svojich aktivátorov. Ako príklad môžme uviesť človeka ktorý ako senzory používa oči, uši apod. a ako aktivátory mu slúžia nohy, ruky a ostatné orgány. Príkladom z oblasti robotiky môže byť robot-agent, ktorý ako senzory používa kamery, laserové zameriavače. Aktivátormi sú rôzne ramená ovládané motormi apod.[9] V ďalšom texte sa pod pojmom agent bude myslieť už len umelý agent, ktorý bol vytvorený človekom.

Agent je riadený programom, ktorý beží na nejakej architektúre [9]. Takže agent je spojením architektúry a programu.

Agentov môžme rozdeliť do 3 základných kategórií. Reaktívni agenti len reagujú na podnety z okolia. Majú predom k dispozícii známu množinu akcií pre rôzne podnety. Deliberatívni agenti zvažujú svoje možnosti dosiahnutia cieľa a vytvárajú si k jeho dosiahnutiu plán. Poslednou skupinou sú sociálni agenti, ktorý už pracujú s explicitnými modelmi chovania ostatných agentov, tvoriacich spolu multiagentný systém. Dokážu si tieto modely upravovať a aktualizovať.[16]

## 2.2 WSN

WSN – bezdrôtové senzorové siete, sú zložené z uzlov. Tie sú vybavené mikrokontrolérmi so senzormi, modulom na bezdrôtovú komunikáciu a zdrojom energie. Tieto uzly sú riadené softvérom, snímajúcim a vyhodnocujúcim dáta namerané zo senzorov. Vďaka možnosti bezdrôtovej komunikácie, môžu jednotlivé uzly odosielať svoje výsledky ostatným uzlom alebo do centrálneho riadiaceho centra. Vzhľadom na to, že jednotlivé senzorové uzly nemajú v dosahu všetky ostatné uzly alebo riadiace stredisko ich komunikačný protokol je založený na tzv. multi-hop smerovacích algoritmoch. To znamená, že jednotlivé správy sa zo zdrojového uzlu do cieľového presúvajú cez uzly ktoré sú medzi nimi.[7]

Využitie takýchto senzorových sietí je napríklad na sledovanie šíriaceho sa požiaru v lesoch, monitorovanie rozsiahlych priestorov apod. Pri takomto využití sa od jednotlivých uzlov sa neočakáva veľký výpočtový výkon, ale na druhej strane je žiadané aby dokázali svoju činnosť vykonávať po relatívne dlhú dobu bez zásahu človeka. Sú napájané z batérií, ktoré majú obmedzenú kapacitu, alebo technológiami ako je získavanie elektrického napätia zo stromov [3], ktoré poskytujú obmedzený príkon. Z tohto dôvodu musia byť jednotlivé uzly energeticky nenáročné. Preto sú vybavené malými mikrokontrolérmi, s obmedzenou pamäťou a výpočtovým výkonom.

## 2.3 Agenti vo WSN

Spojením konceptu agentných systémov a bezdrôtových senzorových sietí získame možnosť mať staticky rozmiestené senzorové uzly, ale agenti, ktorí budú dáta zo senzorov vyhodnocovať sa môžu v tejto sieti pohybovať a využiť potenciál multiagentného prístupu k riešeniu problému. Nový agenti môžu môžu byť do WSN dynamicky vysielaný alebo naopak nepotrebný agenti zrušený.

Klasický agenti sa často programujú v jazykoch ako Java, alebo iných OO[1] jazykoch. Týchto agentov je obtiažne umiestniť do uzlov WSN, kde sú výrazne obmedzené dostupné zdroje oproti klasickým PC. Z tohto dôvodu bol na FIT vyvinutý nový jazyk – ALLL (viď kapitola 5) – na programovanie takýchto agentov[15]. ALLL má nízku úroveň abstrakcie, takže jeho analýza nepotrebuje veľké množstvo zdrojov, ale pritom poskytuje nástroje ako správu výnimiek, klonovanie agentov a metarozhodovanie.

[1] Objektovo orientovaných

# Kapitola 3

# Jazyk a prekladač

Cieľom tejto práce je vytvorenie programovacieho jazyka pre mobilných inteligentných agentov a implementácia prekladača tohto jazyka. K tejto tematike neodmysliteľne patrí pojem jazyk. Podľa jednej z definícií [6]:

**Definícia 3.1.** Nech $\Sigma^*$ značí množinu všetkých podreťazcov nad abecedou $\Sigma$. Potom ľubovoľná podmnožina $L \subseteq \Sigma^*$ je jazyk. Ak podmnožina $L$ obsahuje konečný počet prvkov tak aj jazyk $L$ je konečný, inak je jazyk $L$ nekonečný.

Zvyšok tejto kapitoly bol prevzatý z [5] a [6].

Samotný preklad vstupného jazyka do výstupného robí tzv. prekladač. Prekladač (compiler) je program čítajúci vstupný jazyk (zdrojový kód), ktorý sa následne prekladá do cieľového jazyka[5].

Všeobecne proces prekladu zdrojového jazyka na cieľový prebieha cez niekoľko fáz. Týmito fázami sú lexikálna analýza, syntaktická analýza, sémantická analýza, generovanie vnútorného kódu, jeho optimalizácia a nakoniec generovanie cieľového kódu.

Lexikálnu analýzu vykonáva tzv. *lexer* prípadne *scanner*. Lexer číta vstupný text a rozdelí ho na jednotlivé lexémy (tokeny), pričom kontroluje ich správnosť. Lexémy reprezentuje v jednotnom tvare a prípadne preskakuje nepotrebné časti zdrojového textu (napr. komentáre, biele znaky apod.). Modelmi na popis lexikálnej analýzy sú konečné automaty a regulárne gramatiky.

Na lexikálnu analýzu nadväzuje syntaktická analýza. Syntaktickú analýzu vykonáva tzv. *parser*. Syntaktická analýza má za úlohu určiť syntaktickú štruktúru zdrojového programu, rozdeleného na jednotlivé tokeny lexerom. Syntaktická štruktúra programu je definovaná vo forme derivačného (syntaktického) stromu. Samotná syntax jazyka je definovaná gramatickými pravidlami. Z nich syntaktický analyzátor konštruuje spomínaný derivačný strom. V závislosti na tom ako prebieha stavba derivačného stromu rozoznávame 2 základné druhy parserov. Parser *zhora-dole* stavia derivačný strom vstupného jazyka od koreňového uzlu smerom k terminálnym symbolom, pričom parser *zdola-hore* to robí naopak, tj. od terminálnych symbolov ku koreňovému uzlu. Modelmi na popis a analýzu syntaxe sú bezkontextové gramatiky a zásobníkové konečné automaty.

Na generovanie syntaktických a lexikálnych analyzátorov existujú rôzne nástroje. Niektoré dokážu vygenerovať len lexikálny alebo syntaktický analyzátor. Existujú aj nástroje, ktoré generujú oba spomínané analyzátory. Takýmto nástrojom je aj nástroj ANTLR, ktorý bude podrobnejšie popísaný v kapitole 4.

Ďalšou fázou analýzy vstupného jazyka je sémantická analýza. Tá kontroluje správnosť sémantiky (významu) vstupného programu, tj. uskutočniteľnosť jednotlivých akcií. Jedná

sa napríklad o kontrolu cieľov skokov, typovú kontrolu apod.

Sémantický analyzátor využíva na ukladanie jednotlivých prvkov vstupného jazyka rôzne tabuľky. Jedná sa napríklad o tabuľku identifikátorov, tabuľku funkcií a procedúr alebo návestí.

Napríklad tabuľka symbolov je tabuľka implementovaná formou poľa, zreťazeného zoznamu, binárneho vyhľadávacieho stromu alebo hashovacej tabuľky. Obsahuje informácie o každom identifikátore vyskytujúcom sa v zdrojovom programe. Medzi informácie ukladané danom o identifikátore patrí napr. meno symbolu, jeho typ, riadok, kde bol definovaný, či je pomocným symbolom prekladača[1], či bol inicializovaný atď.

V prípade, že potrebujeme blokovo-štruktúrovanú tabuľku identifikátorov, tj. tabuľku ktorá rešpektuje blokovú štruktúru vstupného jazyka (prekrývanie symbolov apod.), je možné použiť jednu globálnu tabuľku symbolov alebo viacero tabuliek symbolov, kde jednotlivé tabuľky sú uložené v zásobníku. Pri hľadaní symbolu sa postupuje od vrcholu zásobníku k jeho koreňu.

Vnútorný kód je vnútornou reprezentáciou zdrojového kódu a je s ním funkcionálne ekvivalentný. Je nezávislý na cieľovom kóde. Najčastejšie sa jedná o tzv. abstraktný syntaktický strom (AST[2]) alebo 3-adresný kód. Typicky sa pri preklade buduje najprv AST, ktorého výhodou je jednoduchosť vybudovania. Tento sa následne prevedie do 3-adresného kódu. Ten reprezentuje príkazy zdrojového programu postupnosťou jednoduchých inštrukcií. 3-adresný kód je vhodnejší na následnú optimalizáciu.

Cieľom optimalizácie je zmenšenie a zrýchlenie vnútorného kódu. Môžme rozlíšiť 2 typy optimalizácií. Prvou je optimalizácia nezávislá na cieľovej architektúre (jazyku). Sem možno zaradiť presun invariantov z cyklov, eliminácia mŕtvych kódov a rôzne iné. Druhou optimalizáciou je optimalizácia, ktorá je závislá na cieľovej architektúre (jazyku). Sem sa zaraďuje prevažne prideľovanie dostupných registrov.

Pri generovaní cieľového kódu sa optimalizovaný vnútorný kód mapuje na cieľový kód alebo jazyk.

[1] slúžia ukladanie medzivýsledkov

[2] Abstract syntax tree

# Kapitola 4

# ANTLR

ANTLR (ANother Tool for Language Recongnition) je nástroj vytvárajúci framework pre tvorbu prekladačov a interpretov z gramatických pravidiel. Bol vyvinutý na University of San Francisco profesorom Terence Parr-om. ANTLR je napísaný v Jave, takže je bez problémov prenositeľný na rôzne platformy [12].

Z formálneho popisu gramatiky generuje zdrojový kód programu, prijímajúceho daný jazyk. Dokáže generovať program v rôznych jazykoch. Úplne podporované sú jazyky Java, C[1], C# a C#2, ActionScript a JavaScript . Dobrá podpora je aj pre jazyky C# 3 a Python, avšak niektoré vlastnosti chýbajú. Ďalšie jazyky ako napríklad Perl, Ruby, PHP a ďalšie sú zatiaľ vo vývoji alebo podporované len okrajovo[12].

Program vygenerovaný pomocou nástroja ANTLR využíva na analýzu jazyka LL(*) metódu. Je to metóda analýzy bezkontextových gramatík. Pracuje zhora-dole (od koreňa k terminálovým symbolom). Vstupné symboly analyzuje zľava doprava a konštruuje najľavejšiu deriváciu generovaného syntaktického stromu[11].

## 4.1 Syntax

Syntax jazyka ANTLR je vcelku jednoduchá a intuitívna. Na druhej strane možnosti nastavenia tohto nástroja a komplexný popis samotnej gramatiky presahuje rámec tejto práce. V nasledujúcich odstavcoch zhrnieme aspoň základnú syntax a použitie tohto jazyka. Pre komplexnejší popis by som čitateľa odkázal na [12], z ktorej táto podkapitola aj čerpá.

Syntax jazyka ANTLR sa skladá z globálnych nastavení, definícií typov a premenných a gramatických pravidiel. Gramatické pravidlá sú zapísané v syntaxi podobnej EBNF. Jednotlivé pravidlá sú oddelené bodkočiarkou.

Na začiatku popisu sa uvádza typ gramatiky (`lexer` alebo `parser`) kľúčové slovo `grammar` a jej meno. Meno gramatiky musí byť rovnaké ako meno súboru s koncovkou `.g`. V prípade že gramatika obsahuje lexikálne aj syntaktické pravidlá, tj. chceme spraviť lexer aj parser tak typ gramatiky vynecháme.

**Príklad 1.** *Pre gramatiku s názvom `AHLL`, bude meno súboru `AHLL.g`. Generujeme lexer aj parser.*

```
grammar AHLL;
```

[1] kompatibilné s C++

Po type generovaného analyzátora sa definujú nastavenia a to kľúčovým slovom `options`. V tejto časti sa definuje cieľový jazyk generovaného analyzátora. V prípade, že robíme prekladač môžme analyzátorom vygenerovať abstraktný syntaktický strom (AST).

V prípade, že cieľovým jazykom je jazyk C, je potrebné predefinovať typ AST stromu na `pANTLR3_BASE_TREE`.

**Príklad 2.** *Výstupným jazykom je jazyk C a generujeme abstraktný syntaktický strom.*

```
options {
language = C;
ASTLabelType=pANTLR3_BASE_TREE;
output=AST;                          //Abstract Syntax Tree
}
```

Virtuálne tokeny[2] sa deklarujú kľúčovým slovom `tokens`. Syntax je `tokens{t1,t2, ...}`. Tokeny, ktoré reprezentujú nejaký terminálny symbol vo vstupnom texte, napr. čísla, kľúčové slová, deklarovať nie je potrebné. Tieto sa deklarujú automaticky pri popise ich syntaxe.

Samotný popis gramatických pravidiel syntakticky zodpovedá EBNF. Na ľavej strane stojí nonterminálny symbol a na pravej strane je reťazec terminálnych a nonterminálnych symbolov. Užitočná je možnosť ovplyvniť stavbu AST tzv. prepisovacími pravidlami. Tieto pravidlá majú nasledovný tvar:

```
pravidlo -> ^(koreň potomok1 potomok2 ...) ;
```

kde `koreň` je terminálny symbol a `potomok` môže byť terminálny aj nonterminálny symbol. V prípade, že niektoré symboly nechceme zahrnúť do AST tak ich jednoducho v prepisovacom pravidle vynecháme ako ukazuje nasledujúci príklad.

**Príklad 3.** *Príklad ukazuje pravidlo* `main : MAIN ( ) blok` *, v ktorom sa pri stavbe AST vynechajú zátvorky.*

```
main     :     MAIN '(' ')' blok -> ^(MAIN blok)    ;
```

*Koreňom stromu v tomto prípade bude token* `MAIN` *a ako jeho potomok sa rozgeneruje pravidlo pre non-terminálny symbol* `blok`.

## 4.2 ANTLRWorks

ANTLRWorks je grafické vývojové prostredie pre ANTLR. Je takisto ako ANTLR napísané v Jave [12]. Výhodou tohto prostredia je možnosť jednoducho interpretovať a ladiť vyvíjanú gramatiku. Jednotlivé gramatické pravidlá je možné graficky zobraziť vo forme vývojového diagramu.

Interpretovanie gramatík prebieha bez ich prekladu. Je to vhodné na rýchle prototypovanie gramatiky. Pri interpretovaní gramatiky sa žiaden výkonný kód umiestnený v gramatike nevykonáva, len sa simuluje stavanie derivačného stromu z daného vstupu. Problémom interpretácie je, že nefunguje v prípade, že je povolený backtracking v gramatike.

[2]Terminálne symboly, ktoré nezodpovedajú žiadnemu symbolu zo vstupného jazyka, ale chceme ich použiť ako uzol v AST

Ďalšou možnosťou kontroly správnosti gramatiky je jej samotné ladenie. Pri ladení sa gramatika aj s výkonným kódom preloží do Javy a spustí sa. Je možné sledovať stavbu derivačného a abstraktného syntaktického stromu, krokovať gramatiku a sledovať ktoré gramatické pravidlá sa použili. Takisto je k dispozícii aj zásobník volaní funkcií pri jednotlivých krokoch analýzy. Keďže ladenie prebieha preložením gramatiky a vygenerovaním analyzátora, bez problémov funguje aj backtracking.

Nevýhoda ladenia je, že dokáže ladiť len kód v Jave. Ak je generovaný analyzátor v inom programovacom jazyku ladenie nefunguje. V tomto prípade je výhodnejšie výkonný kód od gramatiky oddeliť. Analyzátor len vygeneruje a predá AST a ten sa ďalej analyzuje. Samotný výkonný kód a prekladová logika bude pracovať len s týmto stromom.

## 4.3 ANTLR3 C

V prípade, že analyzátor bude v inom jazyku ako Java, je potrebné mať k dispozíciu príslušnú runtime-ovú knižnicu. ANTLR3 C je runtime-ová knižnica pre jazyk C (alebo C++) [12].

ANTLR3 C je dostupné online na adrese http://www.antlr.org/download/C. Po rozbalení je potrebné knižnicu nainštalovať. Inštalácia pod operačným systémom Linux [1]:

```
tar xvzf antlrtgzname.tar.gz

./configure ; make
sudo make install
```

Inštalácia pod operačným systémom Windows je rovnaká, ale je potrebné mať nainštalovaný nástroj *Cygwin*. Podrobnejší popis inštalácie knižnice je uvedený v dokumentácii k tejto knižnici [1].

Jazyk C na rozdiel od Javy nie je objektový, ale samotná knižnica ja napísaná aby korešpondovala s OOP prístupom. V C-čku sa to dosiahlo použitím štruktúr a ukazateľov na funkcie (metódy). Pri práci s touto knižnicou treba mať na pamäti, že C nie je objektovo orientovaný jazyk. Tzv. nultý parameter[3] metódy sa na rozdiel napr. od C++ nedopĺňa automaticky. Je potrebné ho uviesť explicitne ako ukazuje nasledujúci príklad.

**Príklad 4.** *Vytvorenie parsera a spustenie analýzy vstupného kódu.*

```
psr = AHLLParserNew(tstream);   //Vytvorenie parsera
if(psr == NULL){                //kontrola či nedošla pamäť
        throw OutOfMemory();
}

langAST = psr->prog(psr);      //štart analýzy vstupu, psr == this
```

Kompletný popis vytvorenia analyzátoru pomocou tejto knižnice je uvedený v dokumentácii ku knižnici v časti [1, How to build Generated C Code].

[3] adresa objektu na ktorý je funkcia/metóda volaná, this

# Kapitola 5

# Agentný jazyk ALLL

Ako už bolo spomínané na FIT-e bol vyvinutý nový agentný jazyk – ALLL[1]. Jedná sa o agentný jazyk s nízkou úrovňou abstrakcie. Jazyk neposkytuje žiadnu formu premenných, len zoznamy podobné zoznamom v jazyku Prolog [14]. Jednotlivé prvky zoznamov môžu byť ďalšie zoznamy, takže povoľuje zanorovanie zoznamov.

Jazyk poskytuje niekoľko abstraktných štruktúr, každá obsahuje zoznamy s rôznym významom. Je tu tabuľka plánov (planBase), v ktorej jednotlivé zoznamy reprezentujú pomenované plány. Báza vstupov (inputBase), obsahuje zoznamy obdržané od platformy. Báza znalostí (BeliefBase), kam si agent ukladá informácie behom svojej činnosti. Na dočasné uloženie zoznamov a výsledkov niektorých služieb platformy sú k dispozícii 3 registre.

Poslednou štruktúrou je zásobník akcií. Zoznamy v tejto štruktúre reprezentujú akcie, ktoré sa majú previesť. Na rozdiel od predchádzajúcich štruktúr sú tieto zoznamy zoradené v poradí, v akom sa budú vykonávať [10].

## 5.1 Akcie poskytované interpretom ALLL

Interpret jazyka ALLL poskytuje základné jednoduché akcie. Jednotlivé akcie sú tabuľke 5.1.

Pre vyhľadávanie zoznamov v BeliefBase a InputBase sa využíva operácia unifikácie[10]. Operácia hľadá v tabuľkách rovnaký zoznam aký bol zadaný. V zozname je možné použiť tzv. anonymnú premennú, ktorá reprezentuje ľubovoľný prvok na mieste svojho výskytu. Takisto je možné pri unifikácii zoznamu zadať číslo registra v tvare `&<číslo>`. V tomto prípade sa za miesto registra dosadí jeho obsah. Podrobnejší popis jazyka ALLL možno nájsť v [15], [14] a hlavne v bakalárskej práci Bc. Pavla Spáčila [10].

## 5.2 Služby poskytované platformou

Platforma takisto poskytuje niektoré služby ako napríklad operácie car, cdr známe z jazyka Prolog, presun agenta na iný uzol v sieti, zastavenie interpretu apod. Služby platformy sa volajú z interpretu akciou, ktorá má tvar `$(<písmeno>,<parametre>)`, kde `<písmeno>` označuje typ služby, ktorá je volaná a nepovinná časť `<parametre>` označuje parametre danej služby. Podrobný popis služieb je uvedený v tabuľke 5.2.

[1] Agent Low Level Language

| Kód akcie | Parametre | Význam |
|---|---|---|
| `+` | n-tica\|register | Pridanie n-tice do BeliefBase |
| `-` | n-tica\|register | Odobranie n-tice z BeliefBase |
| `!` | číslo\|register n-tica\|register | Odoslanie správy zadanou n-ticou alebo registrom na mote so zadanou adresou |
| `?` | číslo\|register | Test inputBase na správu od mote/senzoru so zadanou adresou |
| `@` | zoznam akcií | Priame spustenie, akcie sa vložia na zásobník so zarážkou |
| `^` | meno\|register | Nepriame spustenie, hľadá sa plán v planBase s rovnakým menom |
| `&` | číslo | Zmena aktívneho registru |
| `*` | n-tica\|register | test BeliefBase na zadanú n-ticu alebo register. Výsledok sa uloží do aktívneho registru |
| `$` | písmeno {n-tica\|register} | Volanie služby platformy, prvý parameter je kód operácie, druhým sú parametre operácie |
| `#` | žiadne | zarážka za plánom, sémanticky táto akcia nemá žiaden význam |

Tabuľka 5.1: Prehľad akcií v jazyku ALLL [10]

| Kód | Parametre | Popis |
|---|---|---|
| a | žiadne | Aktivovanie sledovania prichádzajúcich správ pri bežiacom interprete |
| f | zoznam\|register | Služba vloží do aktívneho registru prvý prvok zoznamu ako jednoprvkovú n-ticu alebo prvý zo zoznamov ak je ich viac |
| k | žiadne | Zastavenie činnosti interpretu |
| l | zoznam\|register | Ovládanie LED-diód na Mote. Parameter obsahuje kód farby (r,g,b) a za ním sa môže nastaviť stav (0 – nesvieti, 1 – svieti). Ak stav nie je zadaný, dôjde k prepnutiu stavu |
| m | zoznam\|register | Parameter tejto služby obsahuje adresu platformy, kam sa má agent presunúť. Po skopírovaní celého kódu agenta na cieľovú platformu pokračuje vykonávanie agentného kódu na oboch platformách. Ak je za adresou parameter `s`, tak sa vykonávanie agentného kódu na zdrojovej platforme zastaví. |
| r | zoznam\|register | Služba vloží do aktívneho registru zvyšok zoznamu bez 1. prvku. Keď je zoznam jednoprvkový do aktívneho registru sa vloží prázdna n-tica. |
| s | žiadne | Zastavenie vykonávania kódu, kým nepríde správa z rádia. |
| w | zoznam\|register | Pozastavenie vykonávania kódu na čas zadaný v milisekundách |
| d | žiadne\|zoznam\| register | Bez parametrov dôjde k zmeraniu aktuálnej teploty. V opačnom prípade je potrebné zadať typ hodnoty (a – priemer, m – minimum, M – maximum) a počet hodnôt z ktorých sa hodnota vypočíta. |

Tabuľka 5.2: Služby platformy[10]

## 5.3 Nové služby platformy

V čase písania tejto bakalárskej práce boli služby platformy rozšírené o podporu matematických operácií Bc. Pavlom Spáčilom. Z tohto dôvodu bolo možné implementovať prekladač jazyka vyššej úrovne abstrakcie, viď. kapitola 6, s podporou matematických operácií, klasických podmienok a cyklov.

Matematické operácie boli pôvodne navrhnuté ako jednoduché operácie, podobné inštrukciám v jazyku symbolických inštrukcií. Jednalo sa o operácie typu

```
<typ><hodnota1><hodnota2>
```

Kde `<typ>` bol typ operácie, ktorá sa mala vykonať. `<hodnota1>` a `<hodnota2>` boli hodnoty, s ktorými sa operácia vykonávala. Tento prístup sa ukázal byť nevýhodný, keďže každá matematická operácia by vyžadovala nasledujúcu postupnosť krokov:

1. Načítanie operandov z báze znalostí do registrov pomocou testu BeliefBase na dané zoznamy.
2. Pri teste sa vkladajú nájdené zoznamy do ďalšieho zoznamu. Napríklad ak v BeliefBase sú n-tice `(a,2)(b,3)`, tak akcia `*(a,_)`, uloží do aktívneho registru `((a,2))`. Takže je potrebné získať prvý prvok zoznamu službou platformy, ktorá vráti 1. prvok zoznamu[2]. Takže v aktívnom registri sa nachádzala n-tica `(a,2)`.
3. Z každého operandu bolo potrebné získať 2. prvok zoznamu, keďže tam bola umiestnená skutočná hodnota. Prvý prvok sa využíval ako identifikátor. Jednalo sa o operáciu získanie zvyšku zoznamu[3]. Po tejto akcii sa v aktívnom registri objavila hodnota `(2)`.
4. Nasledoval samotný výpočet hodnoty.
5. Uloženie vypočítanej hodnoty operáciou pridania n-tice do báze znalostí.

Z tohto dôvodu boli matematické operácie navrhnuté sofistikovanejšie. Matematické operácie sú rozdelené na binárne a unárne. Binárne operácie majú tvar

```
$(o,<typ>,<operand1>,<operand2>,<meno1>,<meno2>,<výsledok>)
```

Kde:

- `o` označuje službu platformy pre matematické operácie (ALU).
- `<typ>` označuje typ operácie (súčet, rozdiel apod.). Jednotlivé typy operácií a ich kódy sú uvedené v tabuľke 5.3.
- `<operand>` je zoznam alebo register kde sa nachádzajú operandy danej matematickej operácie.
- `<meno>` je n-tica, ktorá reprezentuje prvý prvok operandu, ktorý slúži ako identifikátor. Tento prvok sa skontroluje s prvým prvkom operandu a pri výpočte sa neberie do úvahy. V prípade, že operand bol priamy, tj. priamo zadaný číslom bez identifikátora, zodpovedajúce meno je prázdna n-tica[4].

[2] označená písmenom f
[3] označená písmenom r
[4] ()

- `<výsledok>` je meno identifikátora pod akým sa jednotlivé výsledky uložia.

Unárne operácie majú podobný tvar, ale z pochopiteľných dôvodov obsahujú len jeden operand a jedno meno. Tvar unárnych operácií je

`$(o,<typ>,<operand>,<meno>,<výsledok>)`

Význam jednotlivých položiek je rovnaký ako v predchádzajúcom prípade.

Každá operácie predpokladá, že na vstupe dostane zoznam n-tíc v tvare

`(<meno>,<hodnota>)`

Týchto n-tíc môže byť vo všeobecnosti viac. Ako výsledok sa v prípade binárnych operácií dáva kartézsky súčin jednotlivých operandov medzi sebou. V prípade unárnej operácie sa na každú n-ticu aplikuje táto operácia. Výsledok je uložený v aktívnom registri opäť ako zoznam n-tíc, ktoré majú tvar `(<výsledok>,<hodnota>)`. Toto demonštrujú príklady 5 a 6.

**Príklad 5.** *Binárna operácia súčtu. V jednotlivých registroch sú nasledujúce n-tice:*

- *1. register* – `(((a),1)((a),2))`
- *2. register* – `(((b),3))`
- *3. register – nastavený ako aktívny register*

*Potom operácia* `$(o,add,&1,&2,(a),(b),(c))` *uloží do aktívneho registru n-ticu* `(((c),4)((c),5))`.

**Príklad 6.** *Unárne mínus. V jednotlivých registroch sú nasledujúce n-tice:*

- *1. register* – `(((a),1)((a),2))`
- *2. register – nastavený ako aktívny register*

*Potom operácia* `$(o,min,&1,(a),(c))` *uloží do aktívneho registru n-ticu* `(((c),-1)((c),-2))`.

| **Kód operácie** | **Význam** | Poznámky |
|---|---|---|
| mul | násobenie 2 čísel | |
| div | celočíselné delenie | $operand\_1 \div operand\_2$ |
| mod | zvyšok po celočíselnom delení | $operand\_1 \mod operand\_2$ |
| add | súčet | |
| sub | rozdiel | $operand\_1 - operand\_2$ |
| les | menšie než | $operand\_1 < operand\_2$ |
| leq | menšie, rovno | $operand\_1 <= operand\_2$ |
| mor | väčšie než | $operand\_1 > operand2\_$ |
| meq | väčšie, rovno | $operand\_1 >= operand\_2$ |
| equ | rovno | |
| neq | nerovno | |
| and | logický súčin | |
| orr | logický súčet | |
| min | unárne mínus | |
| not | negácia | |
| cpy | kopírovanie obsahu premenných | $operand\_1 = operand\_2$ |

Tabuľka 5.3: Kódy operácií pre službu aritmeticko-logických operácií

# Kapitola 6

# Jazyk AHLL

AHLL (Agent High Level Language) je jazyk vyššej úrovne abstrakcie pre programovanie mobilných inteligentných agentov. Bol vytvorený v rámci tejto bakalárskej práce. Jedná sa o imperatívny štruktúrovaný jazyk. Syntax jazyka je podobná jazyku C [8], avšak je prispôsobená programovaniu inteligentných agentov a možnostiam agentného jazyka ALLL.

Jazyk dovoľuje prácu s premennými (globálnymi aj lokálnymi), základné konštrukcie z imperatívnych jazykov (cyklus, podmienka), ako aj štruktúrovanie kódu do tzv. plánov[1].

## 6.1 Základná štruktúra jazyka

Program v jazyku AHLL sa dá rozdeliť do 3 základných častí. Prvou je deklarácia globálnych premenných programu. Nasleduje definícia plánov. Poslednou časťou je špeciálny plán nazvaný `main`, ktorý slúži ako vstupný bod programu.

Deklarácia premenných je uvedená kľúčovým slovom `var`. Premenné je možné inicializovať priradením konštanty do nej. Premenné sa reálne vytvoria až po ich inicializovaní, prípadne po priradení hodnoty do nich v rámci programu. Použitie neinicializovanej premennej vyvolá chybu. Deklarovanie a inicializovanie globálnych premenných je voliteľné.

Po deklarácii a prípadnej inicializácii globálnych premenných nasleduje definícia plánov. Táto časť je takisto voliteľná. Definované plány sú uložené v bázy plánov[2]. Plány je možné definovať aj s parametrami. Definícia plánu má nasledujúcu syntax:

```
plan <meno_plánu>(<zoznam_parametrov>) {
  <zoznam_príkazov>
}
```

Poslednou časťou kódu je vstupný bod programu, ktorý je povinnou súčasťou. Uvádza sa kľúčovým slovom `main`. Je to špeciálny plán, ktorým sa začína beh programu. Nemôže mať parametre. Kód tohto plánu je umiestnený na zásobník interpretu. Syntax je nasledujúca:

```
main(){
  <zoznam_príkazov>
}
```

Neodmysliteľnou súčasťou programovacieho jazyka sú komentáre, ktoré sprehľadňujú kód. Komentáre v AHLL sú syntakticky rovnaké ako riadkové komentáre v jazyku C, uvádzajú sa dvoma lomítkami (`//`)[8].

[1] Plán je obdobou procedúr a funkcií známych z iných imperatívnych jazykov

[2] štruktúra interpretu, PLAN BASE

**Príklad 7.** *Základná štruktúra jazyka AHLL*

```
// komentár

var a;  //deklarácia premennej a
a = 8;  //inicializácia premennej

plan p1(b, c)   //definícia plánu p1 s~2 parametrami b, c
{
  //kód plánu
}

main()        //povinná časť, vstupný bod programu
{
  //kód umiestnený na zásobník
}
```

## 6.2 Premenné

Premenné môžu byť globálne, ktoré majú platnosť v celom programe a lokálne, ktoré majú platnosť len v bloku kde boli deklarované. Lokálne premenné sa po skončení bloku vymažú z pamäte a nie sú ďalej prístupné.

Všetky premenné je potrebné deklarovať a pri použití inom ako priradenie hodnoty do nej, musia byť inicializované. Použitie neinicializovanej premennej vyvolá chybu pri preklade. Inicializázia premennej prebieha operáciou priradenia hodnoty (operátor `=`).

Globálne aj lokálne premenné je možné inicializovať konštantou, alebo výrazom, v ktorom sú všetky použité premenné inicializované (tj. výraz je možné vyhodnotiť). Výrazom je možné inicializovať premenné len v plánoch alebo v pláne `main`. Globálne premenné je možné inicializovať aj mimo plánov v tzv. globálnom kontexte, ale len konštantou.

Syntax deklarácie premennej je nasledujúca:

```
var <zoznam_premenných> ;
```

Jednotlivé premenné v zozname premenných sú oddelené čiarkou.

## 6.3 Plány

Plány sú obdobou funkcií v jazyku C. Plán je možné definovať aj s parametrami. Parametre sú predávané hodnotou. Definícia plánu má nasledujúcu syntax:

```
plan <meno_plánu>(<zoznam_parametrov>) {
  <zoznam_príkazov>
}
```

kde `<meno_plánu>` je jednoznačné meno plánu. V prípade, že sa definuje plán s menom, pod ktorým bol už definovaný iný plán, vyvolá to chybu pri preklade. Takisto `<zoznam_parametrov>` je postupnosť identifikátorov oddelených čiarkami, kde každý musí byť jedinečný pre daný plán. Zoznam príkazov plánu je uzavretý v bloku. Lokálne parametre plánu sa po jeho skončení z pamäte odstránia. Predávanie výsledkov z plánu je možné pomocou globálnych premenných.

| Názov príkazu | Syntax |
|---|---|
| blok | `{ <príkazy> }` |
| podmienený príkaz | `if( <podmienka> ) <blok1> else <blok2>` |
| cyklus | `while ( <podmienka> ) <blok>` |
| volanie plánu | `<plan> ( <parametre> ) ;` |
| deklarácia premennej | `var <zoznam_premenných> ;` |
| volanie služby platformy | `platform ( <zoznam parametrov> ) => <identifikátor> ;` |
| odoslanie správy | `send (<adresa_mote>, <sprava>) ;` |
| prijatie správy | `receive ( <adresa_mote> ) => <identifikátor> ;` |
| výraz | viď podkapitola 6.4.9 |
| prázdny príkaz | `;` |

Tabuľka 6.1: Zoznam príkazov jazyka AHLL

## 6.4 Príkazy

Príkazy sú základným výkonným prvkom jazyka. Pomocou nich sa definuje samotné správanie agenta. Príkazy je možné združovať do blokov. Zoznam príkazov je uvedený v tabuľke 6.1.

### 6.4.1 Blok

Blok kódu sám o sebe nemá žiadnu výkonnú funkciu. Jeho funkciou je združovanie príkazov. Blok príkazov je možné použiť, všade tam, kde by sa očakával príkaz. Niektoré konštrukcie jazyka AHLL (podmienený príkaz, cyklus) priamo vyžadujú použitie bloku.

Sémantický význam bloku je, že vytvára vlastný priestor pre lokálne premenné. To znamená, že premenné deklarované v bloku majú platnosť len v danom alebo v zanorených blokoch. Po ukončení bloku sú lokálne premenné vymazané z pamäte.

**Príklad 8.** *Ukážka bloku kódu Príklad ukazuje deklaráciu 2 premenných* `a,b`*. Premenná* `b` *je deklarovaná v bloku. Po ukončení bloku sa vymaže z pamäte.*

```
var a;      // deklarácia premennej a
{           // začiatok bloku kódu
  var b;    // deklarácia premennej b, premenná b je použiteľná len v tomto
            // bloku
}           // koniec bloku, premenná b je vymazaná z pamäte
```

### 6.4.2 Podmienený príkaz

Podmienený príkaz umožňuje vetvenie programu. Podmienkou je ľubovoľný výraz, viď. podkapitola 6.4.9. Výraz je pravdivý v prípade, že číselná hodnota výrazu je rôzna od nuly. Príkazy v `<blok1>` sa vykonajú v prípade, že podmienka bola pravdivá. Úplný podmienený príkaz obsahuje aj vetvu `else`, ktorej kód (`<blok2>`) sa vykoná v prípade nepravdivosti podmienky. Táto vetva nie je povinná.

**Príklad 9.** *Podmienený príkaz.*

```
var a,b;
a = 2;
if (a < 3){
  b = 1;
}
else{
  b = 0;
}
```

*Premenná* `b` *má hodnotu 1 alebo 0, v závislosti na tom či je hodnota premennej* `a` *menšia ako 3.*

### 6.4.3 Cyklus

Cyklus umožňuje opakované vykonanie príkazov v bloku `<blok>`, pokiaľ je podmienka pravdivá. Jedná sa o cyklus s podmienkou na začiatku. V prípade, že podmienka neplatí príkazy v bloku sa nevykonajú ani raz. Podmienkou je ľubovoľný výraz, viď. podkapitola 6.4.9.

**Príklad 10.** *Cyklus s podmienkou na začiatku*

```
var i;
i = 1;
while(i <= 50){
  i = i + 1;
}
```

*Cyklus sa zopakuje 50 krát. Po jeho ukončení ma premenná* `i` *hodnotu 51.*

### 6.4.4 Volanie plánu

Príkaz volania plánu umožňuje plán definovaný v programe spustiť. V prípade, že plán má svoje parametre, tie sú predávané hodnotou. Parametrom plánu môže byť ľubovoľný výraz. Jednotlivé parametre sú oddelené čiarkou.

Plán musí byť pred zavolaním definovaný v sekcii pre definovanie plánov. V prípade, že plán nie je definovaný, preklad končí chybou. Takisto sa kontroluje správny počet parametrov. Napríklad, ak je plán definovaný s 2 parametrami, je potrebné oba parametre zadať, inak preklad aj v tomto prípade končí chybou.

**Príklad 11.** *Volanie plánu V sekcii pre definovanie plánov bol definovaný plán* `sucet(a,b)`*, ktorý má 2 parametre. Bola aj deklarovaná globálna premenná* `c`*, slúžiaca na návrat hodnoty z plánu.*

```
var c;                  // globálna premenná c

plan sucet(a, b) {      // plan sucet, ktorý do premennej c,
                        // uloží súčet svojich parametrov.
  c = a + b;
}
```

*Správanie nasledujúcich volaní:*

| | |
|---|---|
| `nasob(a,b);` | *Volanie plánu* `nasob` *vyvolá chybu pri preklade, lebo plán nebol definovaný.* |
| `sucet(2);` | *Príkaz vyvolá chybu pri preklade, lebo plán* `sucet` *má 2 parametre. Zadaný bol len jeden.* |
| `sucet(2,3);` | *Vyhodnotia sa parametre plánu, zavolá sa plán* `sucet` *s týmito parametrami. Parametre sú predané hodnotou. Po vykonaní plánu bude v premennej* `c` *hodnota 5.* |

### 6.4.5 Deklarácia premennej

Používané premenné treba deklarovať. Po deklarácii sú premenné sprístupnené na zápis hodnoty. Po oboch spomínaných úkonoch (deklarácia a inicializácia) sa premenná označí za inicializovanú a je ju možné použiť aj ako tzv. r-value[3]. Použitie neinicializovanej premennej vyvolá chybu pri preklade.

Sémantický význam deklarácie je, že sa vytvorí záznam o premennej v príslušných tabuľkách prekladača jazyka AHLL, ale premenná samotná sa nevytvorí. Premenná sa vytvorí až po jej inicializácii.

V prípade deklarácii viacerých premenných, sú jednotlivé premenné oddelené čiarkou.

**Príklad 12.** *Deklarácia premennej.*

```
var a,b;
```

*Deklarácia 2 premenných.*

### 6.4.6 Volanie služby platformy

Platforma poskytuje rôzne rozširujúce služby ako napríklad pozastavenie alebo uspanie interpretu, blikanie LED diódami, meranie dát zo senzorov apod. Volanie služby platformy je uvedené kľúčovým slovom `platform`. Syntax je nasledujúca:

```
platform ( <služba> , <zoznam_parametrov> ) => <identifikátor> ;
```

kde `<služba>` je konštanta typu reťazec, identifikujúca danú službu. Nepovinná položka `<zoznam_parametrov>` je zoznam parametrov danej služby. Prvkami zoznamu môžu byť konštanty alebo identifikátory. Z dôvodu, že interpret obsahuje len 3 registre je možné v použiť maximálne dva identifikátory. Výsledok služby platformy sa ukladá do premennej `<identifikátor>`. Táto položka je nepovinná. Pri službách, ktoré neposkytujú žiaden výsledok, napr. zastavenie činnosti interpretu, použitie konštrukcie `=> <identifikátor>` môže spôsobiť chybu pri vykonávaní agentného kódu.

Ďalej treba pamätať na to, že doterajšia implementácia prekladača nekontroluje existenciu služby platformy, jej parametre ani to či poskytuje nejaký výsledok alebo nie. Je na programátorovi, aby na tieto veci dal pozor.

**Príklad 13.** *Pozastavenie činnosti ipterpretu na daný čas. Služba je identifikovaná reťazcom* `w`*, interpret sa zastaví na 300ms.*

```
platform("w",300);
```

*V prípade, že čas uspania je uložený v premennej*

[3] Jedná sa o hodnotu, ktorá sa môže vyskytovať na pravej strane výrazu, volaniach plánu atď.

```
var a;
a = 300;
platform("w",a);
```

**Príklad 14.** *Tento príklad ukazuje službu platformy, ktorá vracia nejaký výsledok. Jedná sa o službu získanie prvého prvku zoznamu. Služba je identifikovaná písmenom* `f`*. Výsledok operácie sa uloží do premennej* `a`*.*

```
var a;
platform("f",[1,2,3]) => a;
```

### 6.4.7 Odoslanie správy

Odosielanie správy na iný senzorový uzol sa deje pomocou príkazu `send`. Tento príkaz má 2 parametre, oba sú povinné. Prvým parametrom je adresa uzlu, kam sa má správa poslať, druhým je samotná správa. Oba parametre môžu byť zadané vo forme konštanty alebo premennej. Syntax príkazu je nasledujúca:

```
send(<adresa_uzlu>, <správa>);
```

**Príklad 15.** *Odoslanie správy na uzol s adresou* `2`*.*

```
send(2,"done");
```

*Oba parametre môžu byť zadané vo forme premenných. V tomto prípade by to vypadalo nasledovne:*

```
var adresa, sprava;
adresa = 2;
sprava = "done";
send(adresa,sprava);
```

### 6.4.8 Prijatie správy

Prijatie správy z iného senzorového uzlu sa deje pomocou príkazu `receive`. Príkaz má nasledujúcu syntax:

```
receive(<adresa_uzlu>) => <identifikátor> ;
```

`<adresa_uzlu>` je označenie uzlu, z ktorého chceme správu prijať. Parameter je nepovinný a v prípade jeho absencie sa výbere prvá prijatá správa. Parameter môže byť zadaný ako konštanta alebo premenná.

`<identifikátor>` je meno premennej, kam chceme prijatú správu uložiť, je nepovinný a v prípade absencie sa prijatá správa nikam neukladá a je vymazaná.

**Príklad 16.** *Prijatie správy z uzlu s adresou 2. Obsah správy sa uloží do premennej* `a`*.*

```
var a;
receive(2) => a;
```

| |
|---|
| *Unárne operátory:*<br>+ − !<br>Unárne operátory majú asociativitu zľava-doprava.<br>*Binárne operátory:*<br>* / %<br>+ −<br>< > <= >=<br>== != <br>&&<br>\|\|<br>Uvedené binárne operátory majú asociativitu zľava-doprava.<br>*Operátor priradenia:*<br>=<br>Operátor priradenia má najnižšiu prioritu, asociativita tohto operátora je sprava-doľava. |

Tabuľka 6.2: Priorita operátorov jazyka AHLL

### 6.4.9 Výrazy

Výrazy sa používajú pri vyhodnocovaní podmienok, ako aj pri inicializácii hodnôt. Jazyk AHLL podporuje základné aritmetické a logické operácie. Syntax a správanie operácií boli prevzaté z jazyka C [8].

Výraz sa skladá z jednotlivých operácií, ktoré sa dajú rozdeliť na binárne a unárne. Unárne operátory majú vyššiu prioritu ako operátory binárne. Priorita operátorov je uvedená v tabuľke 6.2.

Výraz sa stáva príkazom ak je ukončený bodkočiarkou. Výraz má nasledujúcu syntax:

```
<vyraz> ;
```

Výrazom je postupnosť konštánt alebo identifikátorov spojených pomocou binárnych a unárnych operátorov.

#### Konštanty a identifikátory

Konštanta môže byť číselná, reťazcová alebo typu zoznam. Identifikátor je postupnosť písmen a číslic, začínajúca písmenom. Ich syntax je nasledujúca:

```
<identifikátor>  ::= (('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9') *)
<číslo>          ::= '0'..'9'+
<reťazec>        ::= '"'('a'..'z'|'A'..'Z'|'0'..'9')* '"'
<zoznam>         ::=    '[' ']'
                 |  '[' (konstant)  (',' (konstant) )*  ']'
<konstanta>      ::= <číslo> | <reťazec> | <zoznam>
```

#### Unárne operátory

Unárne operátory sa zapisujú pred výraz. Ich syntax je nasledujúca:

```
<unárny_operátor>  ::= '+' | '-' | '!'
<výraz>            ::=  <unárny_operátor> <výraz>
```

Význam jednotlivých operátorov je v tabuľke 6.4

| Operátor | Sémantika |
|---|---|
| $*$ | násobenie |
| $/$ | delenie |
| % | modulo |
| $+$ | súčet |
| $-$ | rozdiel |
| $>$ | väčší než |
| $>=$ | väčšie rovno |
| $<$ | menší než |
| $<=$ | menšie rovno |
| $==$ | rovno |
| $!=$ | nerovno |
| && | logický súčin |
| $\|\|$ | logický súčet |
| $=$ | operátor priradenia |

Tabuľka 6.3: Binárne operátory

| Operátor | Sémantika |
|---|---|
| $+$ | unárne plus |
| $-$ | unárne mínus |
| ! | negácia |

Tabuľka 6.4: Unárne operátory

### Binárne operátory

Binárne operátory sa zapisujú medzi 2 výrazy. Ich Syntax je nasledujúca:

```
<výraz> ::= <výraz> <binárny_operátor> <výraz>
```

Význam jednotlivých operátorov je v tabuľke 6.3

# Kapitola 7

# Preklad vybraných konštrukcií z AHLL do ALLL

Jazyk AHLL poskytuje konštrukcie známe z iných imperatívnych jazykov. Medzi tieto konštrukcie patria premenné, cykly, podmienky, definície plánov atď. Niektoré konštrukcie boli uvedené v [13, Príloha C]. Možnú implementáciu týchto konštrukcií v agentnom jazyku ALLL popisuje táto kapitola.

## 7.1 Premenné

Premenné sa v jazyku ALLL ukladajú do štruktúry BeliefBase (BB). Ukladajú sa ako n-tice v tvare `(<meno>, <hodnota>)` Ich načítavanie do jednotlivých registrov je riešené operáciou unifikácie a získania prvého prvku zoznamu[10].

Globálne premenné sa ukladajú pod menom akým boli deklarované. Napríklad globálna premenná s deklaráciou

```
var a;
a = 5;
```

je uložená v BB v tvare `((a),5)`.

Pri lokálnych premenných, ktoré môžu byť deklarované kdekoľvek v programe je situácia odlišná. Názov premennej sa skladá z mena plánu, čísla bloku kde bola deklarovaná a samotného názvu premennej. Napríklad lokálna premenná deklarovaná v pláne `plan1` a prvom bloku tohto plánu (prvý blok je zároveň hlavným blokom plánu, nultý blok sa využíva na ukladanie parametrov plánu viď ďalej) s deklaráciou `var a;` je uložená v BB v tvare `((plan1,1,a),5)`.

Medzi premenné ukladané do BB patria aj parametre plánov. Ako už bolo spomenuté, predávajú sa hodnotou. Po vyhodnotení parametrov sa uložia do BB pod názvom tvoreným menom plánu, ku ktorému parametre patria, ako nultý blok plánu a názvu parametra. Napríklad majme definíciu plánu s parametrom `plan plan1(a)`. Tento parameter bude uložený v BB v tvare `((plan1,0,a),<hodnota>)`.

Prepis hodnoty premenných je riešený takým spôsobom, že n-tica reprezentujúca danú premennú sa najprv z BB vymaže a následne sa do BB uloží nová n-tica s aktuálnou hodnotou.

Pri vymazávaní lokálnych premenných blokov a plánov sa využíva operácia unifikácie n-tice s možnosťou anonymných premenných. Napríklad na vymazanie lokálnych premenných

bloku sa vygeneruje operácia `-((<meno_plánu>,<číslo_bloku>,_),_)`. `<meno_plánu>` je plán v ktorom sa daný blok nachádza, `<číslo_bloku>` je blok, z ktorého mažeme premenné. Prvá anonymná premenná reprezentuje všetky premenné z daného bloku a plánu. Druhá anonymná premenná reprezentuje, že hodnoty týchto premenných sú ľubovoľné.

Vymazávanie premenných z celého plánu (vrátane parametrov) je riešené obdobným spôsobom, len miesto čísla bloku sa vloží anonymná premenná.

## 7.2 Podmienený príkaz

Pri implementovaní podmieneného príkazu sa využíva vlastnosť unifikácie, ktorou je, že ak sa unifikovaná n-tica nenachádza v BB akcia skončí chybou a zo zásobníka akcií je vymazaný celý plán až po zarážku [10].

Symbolický zápis podmienky vo forme makra:

```
macro if $podmienka $then $else
        @($podmienka_true, $then)
        @($podmienka_false, $else)
        -($podmienka)
macro-end
```

Po vyhodnotení podmienky sa do BB uloží n-tica v tvare `(<názov_podmienky>,<1/0>)`. Názov podmienky je pomocná premenná, ktorá identifikuje danú podmienku. Hodnotou je číslo 1 alebo 0 v závislosti na tom či bola podmienka splnená alebo nie. Následne sa využije tzv. priame spustenie plánu [10, strana 16]. Ako prvou operáciou tohto plánu je unifikácia `*(<názov_podmienky>,1)` pre vetvu „then“, kde je podmienka splnená a `*(<názov_podmienky>,0)` pre vetvu „else“.

V prípade, že bola podmienka splnená, v BB sa nachádza n-tica `(<názov_podmienky>,1)`. Unifikácia vo vetve then skončí úspechom a pokračuje sa vo vykonávaní akcií tejto podmienky. Po skončení vetvy then, sa spustí vetva else. Tá sa ako prvé pokúsi unifikovať n-ticu `(<názov_podmienky>,0)`, ale táto sa v BB nenachádza, operácia skončí neúspechom a zmaže sa celý plán, ktorý obsahoval akcie pre vetvu else. V prípade nesplnenia podmienky je situácia opačná.

## 7.3 Iterácia

Pri implementácii cyklu `while` sa takisto využíva vlastnosť unifikácie popísaná vyššie. V tomto prípade je potrebné cyklus spúšťať, kým podmienka platí. Preto sa cyklus uloží ako plán do Plan Base (PB). Po ukončení akcií tohto plánu sa vyvolá jeho opätovné spustenie. V prípade, že podmienka cyklu ešte stále platí, vykoná sa ďalšia iterácia[13, príloha C].

Symbolický zápis cyklu vo forme makra:

```
macro while $meno_iterácie $podmienka $akcia:
            +π ($meno_iterácie,@($podmienka_true, $akcia, @π($meno_iterácie)))
            @π(meno_iterácie)
macro end  [13]
```

V prípade cyklu sa podmienka vyhodnocuje v pláne reprezentujúcom cyklus. Podmienka sa po vyhodnotení uloží do BB opäť v tvare `(<názov_podmienky>,1/0)`. Po vyhodnotení sa unifikuje n-tica `(<názov_podmienky>,1)`. V prípade, že sa unifikácia podarí,

podmienka sa vymaže, vykonajú sa akcie v tele cyklu a následne sa opäť spustí plán tohto cyklu. Cyklus končí, keď podmienka nebude splnená. V tom prípade sa unifikácia n-tice `(<názov_podmienky>,1)` nepodarí a plán sa vymaže. Po ukončení cyklu sa podmienka vymaže z BB.

# Kapitola 8

# Implementácia prekladača

Implementácia prekladača je rozdelená na niekoľko častí. Prvou časťou je implementácia gramatiky jazyka AHLL. Tá sa nachádza v súbore `AHLL.g` a je písaná v ANTLR. Prekladom gramatiky sú vygenerované moduly `AHLLLexer` a `AHLLParser`.

Druhou časťou je implementácia samotného prekladača. Prekladač je písaný v jazyku C++ s využitím knižníc ANTLR3 C a STL[1].

Implementácia prekladača prebiehala spočiatku hlavne ako návrh gramatiky a možnosti jej prekladu do cieľového jazyka. O všetko sa starala jedna trieda. Rozširovanie gramatiky a funkcionality prekladača robilo značné problémy. Preto bol návrh prerobený a rozdelený do viacerých tried. Popis jednotlivých tried a ich hlavnej funkcionality je obsahom tejto kapitoly.

Paralelne s návrhom jazyka a implementáciou jeho prekladača prebiehala aj implementácia nových služieb platformy potrebných v mote-och. Toto spôsobilo, že zo začiatku sa overenie funkčnosti konfrontovalo s dohodnutou špecifikáciou. Akonáhle boli potrebné služby implementované bol aj prekladač v štádiu, keď ho bolo možné otestovať na reálnych príkladoch.

V tomto štádiu sa objavili nové problémy, na ktoré sa neprišlo v čase návrhu. Avšak vďaka dobrému návrhu tried prekladača, takisto aj komponentov platformy boli tieto problémy odstránené v relatívne krátkom čase.

Ďalšia práca na prekladači bola spojená hlavne s rozšírením príkazov. Pribudla možnosť volania služieb platformy, práca s bezdrôtovou komunikáciou apod.

V nasledujúcich podkapitolách je popis jednotlivých tried prekladača a funkcie poskytované týmito triedami.

## 8.1 AHLLLexer a AHLLParser

Tieto 2 komponenty implementujú lexikálny a syntaktický analyzátor. Sú napísané v jazyku C a boli generované prekladom gramatiky.

## 8.2 ASTGenerator

Táto trieda je zapuzdrením lexikálneho a syntaktického analyzátora, do triedy. Okrem konštruktoru, ktorý má povinný parameter meno súboru, v ktorom sa nachádza zdrojový kód v jazyku AHLL obsahuje ešte 2 metódy.

[1] Standard Template Library

Metóda `runGrammarTest()` sa stará o vytvorenie lexikálneho a syntaktického analyzátoru, tzv. streamu tokenov, inicializáciu a vytvorenie AST stromu a spustenie gramatickej analýzy.

Druhá metóda s názvom `pANTLR_BASE_TREE getTree()` vráti ukazateľ na AST. AST je typu pANTLR_BASE_TREE, čo je štruktúra definovaná v hlavičkovom súbore knižnice ANTLR3 C.

## 8.3 ByteCode

Táto trieda je generátorom jednotlivých inštrukcií v cieľovom jazyku (ALLL). Návratovou hodnotou každej metódy je reťazec typu `std::string`. Poskytuje základné inštrukcie, ktoré sú priamo odvodené z jazyka ALLL. Medzi tieto metódy patrí napríklad pridanie/odobranie n-tice/registra z BB, nastavenie registra na aktívny, volanie plánu, priame spustenie, unifikácia a operácie car a cdr známe z jazyka LISP.

Okrem týchto inštrukcií poskytuje aj sofistikovanejšie inštrukcie odvodené z potrieb prekladača. Medzi tieto inštrukcie patrí testovanie BB na prítomnosť n-tice, načítanie premennej z BB do registra, uloženie premennej, generovanie unárnej/binárnej matematickej operácie a kopírovanie hodnoty premennej do inej premennej.

## 8.4 Symbol

Táto štruktúra nesie informácie o jednotlivých symboloch[2] definovaných v zdrojovom kóde. Patrí k sémantickej analýze. O každom deklarovanom symbole sa zaznamenáva meno, pod akým bol symbol deklarovaný, riadok deklarácie, adresa pod ktorou symbol vystupuje v ALLL, či je symbol lokálnym symbolom[3] a či bol symbol inicializovaný. Riadok deklarácie sa využíva pri výpise sémantických chýb počas prekladu.

## 8.5 SymbolTable

Táto trieda implementuje časť sémantickej analýzy, ktorá sa stará o deklarácie premenných. Jedná sa o tzv. tabuľku symbolov. Tabuľka symbolov uchováva záznamy o jednotlivých symboloch (štruktúry `Symbol`).

Keďže jazyk AHLL je blokovo orientovaný, táto skutočnosť sa odzrkadľuje aj na implementácii tabuľky symbolov. Tabuľka symbolov je implementovaná formou zásobníku. Každý záznam v zásobníku reprezentuje premenné deklarované v rovnakom bloku. Na vrchole zásobníka sa nachádza aktuálny blok, hlbšie sú nadradené bloky. Premenné v bloku sú vyhľadávané pomocou mapovacej funkcie. Jedná sa o štruktúru `std::map`.

Pri vyhľadávaní konkrétneho symbolu sa postupuje od vrcholu zásobníka. Prehľadávanie končí pri prvom výskyte symbolu alebo prehľadaním všetkých úrovní zásobníka. V tom prípade sa symbol vyhlási za nedefinovaný.

Tabuľka symbolov poskytuje metódy na pridanie nového symbolu do tabuľky a vyhľadanie symbolu podľa mena. Ďalej poskytuje metódy na vstup/výstup do nového plánu/bloku. Jedná sa o metódy `enterPlan()`, `leavePlan()`, `enterBlok()` a `leaveBlok()`. Tieto metódy zvyšujú/znižujú vrchol zásobníku.

[2] premenných

[3] symbol, ktorý je pomocným symbolom prekladača, slúži ako dočasné úložisko dát

Pre ladenie je k dispozícii metóda `print()`, ktorá vypíše aktuálny obsah tabuľky symbolov.

## 8.6 PlanTable

Táto trieda implementuje druhú časť sémantického analyzátora. Jedná sa o tzv. tabuľku plánov. Trieda ukladá informácie o jednotlivých plánoch definovaných v zdrojovom kóde.

Tabuľka plánov pre každý definovaný plán ukladá jeho meno, adresy jeho parametrov a riadok, na ktorom bol plán definovaný.

Tabuľka plánov poskytuje metódy pre pridanie nového plánu do tabuľky. Jedná sa o metódu `enterPlan(sdt::string name, int declLine)`. Táto metóda spôsobí pridanie nového plánu do tabuľky plánov. Na pridávanie parametrov k aktuálnemu plánu je k dispozícii metóda `addParamAddress(std::string addr)`.

Medzi ďalšie služby, ktoré poskytuje trieda patria vyhľadanie plánu a zistenie či tabuľka obsahuje zadaný plán.

Na ladenie sa používa metóda `print()`, ktorá vypíše aktuálny obsah tabuľky plánov.

## 8.7 CodeGenerator

Táto trieda implementuje priechod abstraktným syntaktickým stromom a riadi generovanie cieľového kódu. Trieda obsahuje inštancie tried `ByteCode`, `PlanTable` a `SymbolTable`. Má jedinú verejnú metódu `generateCode()`. Pomocou tejto metódy sa spustí generovanie cieľového kódu.

Stromom prechádza rekurzívnym zostupom. Každý uzol stromu má v tejto triede svoju metódu, ktorá sa stará o jeho vyhodnotenie. Jedná sa napr. o metódy `evalCOND` (vyhodnotenie podmienky v cykloch a podmienenom príkaze), `evalEXPR` (vyhodnotenie výrazov) atď.

Výsledný kód je uložený v dvojici reťazcov `outCode` a `planBase`. V reťazci `outCode` sa nachádzajú akcie umiestnené na zásobník agenta. Tieto akcie sú vykonávané ako prvé po spustení interpretu. V reťazci `planBase` sú umiestené jednotlivé plány agenta.

Väčšina metód v triede `CodeGenerator`, ktoré prechádzajú stromom a generujú cieľový kód, má nasledovné parametre:

- `TREE t` – koreň podstromu, ktorý daná metóda vyhodnocuje
- `std::string & code` – reťazec, kam má metóda generovať kód. Parameter ovplyvňuje miesto, kam sa generuje cieľový kód (zásobník, báza plánov atď.).

## 8.8 Printer

Táto trieda slúži prevažne na ladiace účely a kontrolu. Jej úlohou je výpis vygenerovaného agentného kódu v štruktúrovanej podobe, tzn. každá akcia je na samostatnom riadku. Toto sa deje pomocou jej metódy `print(std::string code)`.

# Kapitola 9

# Prostredie pre programovanie v AHLL

Mnoho textových editorov umožňuje zvýraznenie syntaxe v programovacích jazykoch. Jedným z takýchto editorov je aj textový editor *Kate*, dostupný pod grafickým prostredím KDE v OS Linux.

Definícia zvýraznenia syntaxe pre tento textový editor uložená vo formáte XML[2]. Toto XML obsahuje zoznam kľúčových slov jazyka, definíciu štýlov a pravidiel. Tieto XML súbory sú uložené väčšinou v adresári `/usr/share/apps/katepart/syntax/`. V tomto prípade sú definície syntaxe dostupné pre všetkých užívateľov systému.

Súbor s definíciou syntaxe je možné uložiť aj do adresára
`~/.kde/share/apps/katepart/syntax/`
alebo
`~/.kde4/share/apps/katepart/syntax/`

## 9.1 Štruktúra XML súboru

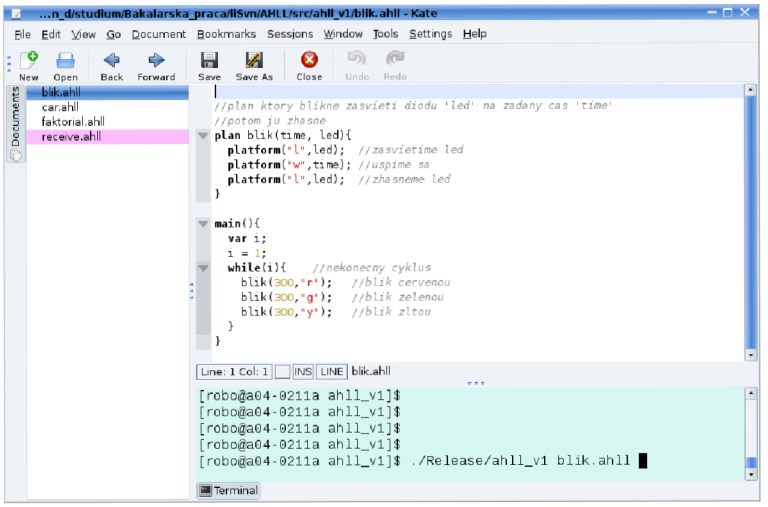
XML súbor popisujúci syntax jazyka začína hlavičkou popisujúcou verziu XML dokumentu a typ dokumentu. V našom prípade hlavička vypadá nasledovne:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE language SYSTEM "language.dtd">
```

Ďalej nasleduje samotný popis jazyka. Jedná sa o element `<language>`. Tento uzol má 2 podstromy. `<highlighting>` definuje štýl zvýraznenia syntaxe jazyka a nepovinný element `<general>`, ktorý môže obsahovať informácie o kľúčových slovách, komentároch a odsadzovaní a nadväznosti kódu.

Element `<highlihting>` obsahuje voliteľný element `<list>` obsahujúci zoznam kľúčových slov. Ďalej obsahuje povinné elementy `<contexts>` a `<itemDatas>`. Element `<contexts>` obsahuje všetky kontexty pre zvýrazňovanie syntaxe. Pod pojmom kontext sa myslí napríklad to či daný znak je uprostred reťazca, v komentári, čísle apod. Element `<itemDatas>` obsahuje popis farieb a typov písma pre jednotlivé kontexty.

Obdobným spôsobom bol vytvorený aj popis syntaxe pre programovací jazyk AHLL. Popis je umiestnený v súbore `ahll.xml`. Tento súbor je potrebné dať do jednej z vyššie spomínaných zložiek. Odteraz bude textový editor *Kate* rozpoznávať súbory `*.ahll` ako programy v programovacom jazyku AHLL a tomu prispôsobí zvýraznenie syntaxe.




Obrázok 9.1: Textový editor Kate so zvýraznením syntaxe pre AHLL

Ukážka textového editoru *Kate* so zvýraznením syntaxe je na obrázku 9.1.

# Kapitola 10

# Príklady kódov

Na testovanie prekladača boli vytvorené jednoduché programy, ktoré demonštrujú jeho a správne generovanie kódu v jazyku ALLL. Prvým programom je reimplementácia programu na blikanie LED diód vo novovytvorenom jazyku AHLL. Pôvodná verzia tohto programu bola vytvorená v rámci diplomovej práce Ing. Jana Horáčka[4] a bakalárskej práce Bc. Pavla Spáčila[10].

Keďže služby platformy boli rozšírené o podporu aritmeticko-logických operácií druhý ukážkový program je výpočet faktoriálu na Mote.

Oba programy boli odskúšané v dostupnom simulačnom nástroji TOSSIM.

## 10.1 Blikanie LED na Mote

Počas implementácie jazyka ALLL a platformy bol vytvorený program, blikajúci LED diódami [10]. Tento program bol prepísaný do jazyka AHLL. Program v jazyku AHLL vypadá nasledovne:

```
//plan ktory blikne zasvieti diodu 'led' na zadany cas 'time'
//potom ju zhasne
plan blik(time, led){
  platform("l",led);  //zasvietime led
  platform("w",time); //uspime sa
  platform("l",led);  //zhasneme led
}

main(){
  var i;
  i = 1;
  while(i){    //nekonecny cyklus
    blik(300,"r");   //blik cervenou
    blik(300,"g");   //blik zelenou
    blik(300,"y");   //blik zltou
  }
}
```

Pomocný plán `blik` má 2 parametre. Parameter `time` je čas, na ktorý sa má dióda zasvietiť. `led` určuje, ktorou diódou sa má bliknúť. Plán pomocou príkazu volania služby

platformy najprv zasvieti zadanú LED. Následne pozastaví interpret na daný čas. Po uplynutí tohto času diódu zhasne a plán sa ukončí.

Kód na zásobníku spustí v nekonečnom cykle blikanie jednotlivých diód pomocou spomínaného plánu `blik`.

## 10.2 Faktoriál

Nasledujúci kód ukazuje výpočet faktoriálu na Mote.

```
var c;   //globalna premenna kde bude vysledok

main(){
  c = 1;
  var a;
  a = 1;  //pocitadlo
  while(a <= 5){
   c= c *a;
   a = a+1;
  }
  send(2,c);  //odosleme vysledok
}
```

Program vypočíta do globálnej premennej `c` faktoriál čísla 5. Po výpočte faktoriálu sa výsledok odošle na uzol s adresou `2`.

# Kapitola 11

# Záver

Cieľom tejto práce bolo navrhnúť jazyk vyššej úrovne abstrakcie, pre programovanie mobilných inteligentných agentov, prekladač tohto jazyka a jednoduché vývojové prostredie, umožňujúce pohodlne programovať v tomto jazyku. Ako už bolo spomenuté práca nadviazala na práce Ing. Jana Horáčka a Bc. Pavla Spáčila, ktorým sa podarilo implementovať spomínanú agentnú platformu a interpret agentného jazyka ALLL.

Nami navrhnutý jazyk kombinuje imperatívne a agentné princípy, ktoré sú vhodné na programovanie mobilných inteligentných agentov. Umožňuje prácu s aritmeticko-logickými operáciami, ako aj využívanie služieb agentnej platformy. Takisto sa podarilo úspešne naprogramovať prvú funkčnú verziu prekladača tohto jazyka a implementovať zopár ukážkových programov. Pre pohodlné programovanie v tomto jazyku bol vytvorený popis syntaxe jazyka pre textový editor *Kate*.

Ďalšia práca na jazyku a prekladači by mohla byť zameraná na zahrnutie optimalizačných algoritmov do prekladovej logiky. Takisto by bolo možné mierne upraviť gramatiku jazyka a viac ju priblížiť agentným princípom a práce so zoznamami, ktoré sú silnou stránkou jazyka ALLL. Čo sa týka vývojového prostredia, bolo by ho možné spojiť s už existujúcim simulátorom pre PC.

# Literatúra


[1] ANTLR3 C Runtime API and Usage Guide. [online], mar 2009 [cit 1.5.2010].
URL <http://www.antlr.org/api/C/index.html>

[2] The Kate Syntax Highlight System. [online], [cit. 10.5.2010].
URL <http:
//docs.kde.org/development/en/kdesdk/kate/katehighlight-system.html>

[3] Himes, C.; Carlson, E.; Ricchiuti, R.; aj.: Ultralow Voltage Nanoelectronics Powered Directly, and Solely, From a Tree. *Nanotechnology, IEEE Transactions on*, ročník 9, č. 1, jan. 2010: s. 2–5, ISSN 1536-125X, doi:10.1109/TNANO.2009.2032293.

[4] Horáček, J.: *Platforma pro mobilní agenty v bezdrátových senzorových sítích.* diplomová práca, Brno, FIT VUT v Brně, 2009.

[5] Meduna, A.: *Elements of Compiler Design*. Taylor & Francis Informa plc, 2008, ISBN 978-1-4200-6323-3, 304 s.

[6] Meduna, A.; Lukáš, R.: Formálne jazyky a prekladače. 2008, podklady k prednáškam z kurzu IFJ.

[7] Raghavendra, C. S.; Sivalingam, K. M.; Zhati, T.: *Wireless Sensor Networks*. New York: Springer, první vydání, 2004, ISBN 1-4020-7883-8, 3–20 s.

[8] Ritchie, D. M.: *C Reference Manual*. Bell Telephone Laboratories, May 1975 [cit. 24.4.2010].
URL <http://cm.bell-labs.com/cm/cs/who/dmr/cman.pdf>

[9] Russell, S.; Norvig, P.: *Artificial Intelligence: A modern Approach*. New Jersey: Prentice Hall, druhé vydání, 2003, ISBN 0-13-790395-2, 31–58 s.

[10] Spáčil, P.: *Mobilní agenti v bezdrátových senzorových sítích.* bakalárska práca, Brno, FIT VUT v Brně, 2009.

[11] Volkmann, R. M.: ANTLR 3 [online]. *Java News Brief*, Jún 2008 [cit 24.4.2010].
URL <http://jnb.ociweb.com/jnb/jnbJun2008.html>

[12] WWW stránky: ANTLRv3. [online], [cit. 24.4.2010 ].
URL <http://www.antlr.org>

[13] Zbořil, F.: *Plánování a komunikace v multiagentních systémech.* dizertačná práca, Brno, FIT VUT v Brně, 2004.

[14] Zbořil, F., Jr.; Spáčil, P.: Automata for Agent Low Level Language Interpretation. *Computer Modeling and Simulation, International Conference on*, ročník 0, 2009: s. 455–460, doi:http://doi.ieeecomputersociety.org/10.1109/UKSIM.2009.82.

[15] 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:http://doi.ieeecomputersociety.org/10.1109/UKSIM.2008.56.

[16] Štepánková, O.; Mařík, V.; Lhotská, L.: *Umělá inteligence*, kapitola 4. Distribuovaná umělá inteligence. Praha: Academia, první vydání, 1997, ISBN 80-200-0504-8, s. 144–147.

# Dodatok A

# Obsah CD

| | |
|---|---|
| `REDME` | návod na preloženie a spustenie prekladača |
| `doc/projekt.pdf` | dokument bakalárskej práce |
| `doc/tex` | zdrojová podoba dokumentu bakalárskej práce |
| `src/ahll_v1` | zdrojový text prekladača |
| `src/antlr` | gramatika prekladača |
| `src/kate` | prostredie pre programovanie v AHLL |

# Dodatok B

# EBNF jazyka AHLL

```
//vstupny bod analyzy
<prog>           ::= ( <var_definition>|<var_init> )* <plan>* <main> ;

//definicia planov
<plan>           ::= <PLAN> <ID> '(' <plan_params>?  ')' <blok> ;
//parametre planov
<plan_params>      ::=   <ID> ( ',' ID )* ;
//pociatocny plan na zasobniku
<main>           ::= <MAIN> '(' ')' <blok> ;

//prikaz
<statement>        ::=     <blok> | <if_stat> | <while_stat> | <plan_call>
                 |  <var_definition>  | <var_init> | (<expression> <SC>)
                 | <plat_call> | <send_call> | <receive_call> | <SC>  ;

//blok prikazov
<blok>             ::= '{' <statement>* '}' ;
//podmienka
<if_stat>          ::= <IF> '(' <expression> ')'  <blok> (<ELSE> <blok>)?  ;
//cyklus while
<while_stat>       ::= <WHILE> '(' <expression>  ')' <blok>  ;

<var_definition>   ::= <VAR> <ID> ( ',' <ID> )*  <SC>  ;
<var_init>         ::= <ID> <IS> <konstant> <SC>   ;
<plat_call>        ::= <PLATFORM> '(' <STR>  ( ',' <expression> )*  ')'
                       ( '=>' <ID>  )?  SC ;
<send_call>        ::= <SEND> '(' <expression> ',' <expression> ')' <SC> ;
<receive_call>     ::= <RECEIVE> '(' <expression>?  ')' ('=>' <ID> )? <SC>     ;
<plan_call>        ::= <ID> '('  <plan_call_params>  ')' <SC> ;

<plan_call_params> ::= (<expression>  ( ',' <expression> )*)?  ;
//=============================================================
//syntakticke pravidla pre vyrazy
<expression>       ::= <ID> <IS> <expression>
                 |   <or_op> ( <OR>  <or_op>  )*
```

```
                ;

<or_op>             ::= <and_op> ( <AND> <and_op> )*  ;
<and_op>            ::= <equal_op> ( (<EQUAL> | <NOT_EQUAL>)  <equal_op> )*  ;
<equal_op>          ::= <more_less_op> ( (<LESS> | <MORE> | <LESS_EQ> | <MORE_EQ> )
                      <more_less_op> )*  ;
<more_less_op>      ::= <aditive_op> ( ( <PLUS> | <MINUS> ) <aditive_op> )*  ;
<aditive_op>        ::= <multiplicative_op> ( ( <MULT> | <DIV> | <MODULO> )
                      <multiplicative_op> )*  ;
<multiplicative_op>::=  ( (<MINUS>|<PLUS>|<NEG>) )?   ( <atom>|<bracket>)  ;

<bracket>           ::= '(' <expression> ')' ;
<atom>              ::= <ID> | <konstant>    ;
<konstant>          ::= <INT> | <STR> | <zoznam>  ;

<zoznam>            ::=  '[' ']'
                    |   '[' ( <konstant> )  (',' (<konstant>) )*  ']' ;
//=========================================================================
//*************************************************************************
//                                LEXIKALNE PRAVIDLA
//*************************************************************************
//operatory:
<IS>                ::= '='  ;
<OR>                ::= '||' ;
<AND>               ::= '&&' ;
<EQUAL>             ::= '==' ;
<NOT_EQUAL>         ::= '!=' ;
<LESS>              ::= '<'  ;
<LESS_EQ>           ::= '<=' ;
<MORE>              ::= '>'  ;
<MORE_EQ>           ::= '>=' ;
<PLUS>              ::= '+'  ;
<MINUS>             ::= '-'  ;
<MULT>              ::= '*'  ;
<DIV>               ::= '/'  ;
<MODULO>            ::= '%'  ;
<NEG>               ::= '!'  ;

//KLUCOVE SLOVA
<IF>                ::= 'if' ;
<ELSE>              ::= 'else' ;
<WHILE>             ::= 'while'  ;
<VAR>               ::= 'var' ;
<PLAN>              ::= 'plan' ;
<MAIN>              ::= 'main' ;
<PLATFORM>          ::= 'platform' ;
<SEND>              ::= 'send' ;
<RECEIVE>           ::= 'receive' ;
```

```
// ostatne
<ID>              ::=  ( ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9')* ) ;
<INT>             ::=  '0'..'9'+ ;
<STR>             ::=  '"' STR_frag '"'   ;
<STR_frag>        ::=  ('a'..'z'|'A'..'Z'|'0'..'9')*  ;
<SC>              ::=   ';' ;
<COMMENT>         ::= '//'( ~('\r'|'\n') ) * '\r'? '\n' ;
```