PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Demonstrační program pro roboty Khepera

2012 Jakub Bednarský

Anotace

Pro výuku základních konstrukcí v programování lze využít robotů Khepera. Cílem práce je vytvořit v programovacím jazyce Common Lisp jednoduchý program na ovládání a programování robotů Khepera. Program bude sloužit k demonstračním účelům ovládání robotů a jejich jednoduchému programování. Propojení hostitelského počítače a robota je vyřešeno pomocí virtualizace sériového rozhraní pro Bluetooth. Důležitou částí práce bylo vytvoření jazyka pro řízení chování robota. V příloze bakalářské práce je implementován program pro ovládání robota.

Děkuji vedoucímu práce Doc. RNDr. Michalu Krupkovi, Ph.D. za vedení této bakalářské práce a za přínosné konzultace.

Obsah

Seznam obrázků

1. Úvod

1.1. Motivace a cíle

Cílem práce je vytvořit v programovacím jazyce Common Lisp jednoduché rozhraní na ovládání robotů Khepera, kteří jsou k dispozici na katedře informatiky. Program by měl sloužit zejména demonstračním účelům při propagačních akcích pro studenty středních škol.
Hlavní požadavky na program jsou:

- vytvoření programové knihovny na ovládání robotů,
- čtení a nastavování hodnot parametrů robotů (čidla, motory),
- jednoduchý jazyk, ve kterém by studenti mohli řešit zadané úkoly.

Všechny moduly by měly být použitelné samostatně i pro jiné projekty. Moduly musí mít uživatelskou dokumentaci. Pro jejich implementaci je nutné použít programovací jazyk, který je volně dostupný a je používán na katedře informatiky.

1.2. Výběr programovacího jazyka

Lisp je druhý nejstarší dodnes používaný vyšší programovací jazyk (první je Fortran). Název pochází ze způsobu zpracovávání kódu jazyka – *Lis*t *P*rocessing. Byl původně specifikován roku 1958 a jeho autorem je John McCarthy [3]. Lisp je postaven na základech λ-kalkulu. Pro programování umělé inteligence byl dlouho výhradním programovacím jazykem a zároveň ovlivnil mnohé programovací jazyky.

Zvolil jsem programovací jazyk Lisp se záměrem jednoduchého a rychlého vývoje, možností představit jeho sílu a v neposlední řádě využít jeho základní vlastnosti – *program jsou data*. Zdrojový kód v Lispu je zapisován do seznamů. Díky tomu můžeme v jazyku, kterým budeme ovládat robota, využívat sémantiku jazyka Lisp a pozměnit si pro něj pouze syntax. Zároveň využijeme pro jednoduchost a transparentnost jazyka prefixový zápis.

Prefixový zápis je výhodný, protože definice jazyka nemusí řešit priority operací. Infixové operátory by nebylo možné jednoduchým principem předávat jako argumenty funkcím. Lisp obsahuje propracovaný objektově orientovaný systém, možnost programovat anonymní funkce pomocí *lambda výrazů* a možnost psát makra. Makra slouží k manipulaci se seznamy obsahující zdrojový kód. Zároveň mohou makra sloužit k implementaci líného vyhodnocování [4], které budeme využívat ve výrazech regulátoru[1].

[1] Regulátorem budeme rozumět soubor pravidel, které definují chování robota.

Všechny tyto vlastnosti Lispu jsou v programu ovládání robota využity a vedou ke zjednodušení implementace.

Jazyk Lisp má mnoha různých dialektů. Roku 1994 byl standardizován ANSI Common Lisp. Jedna z implementací ANSI Common Lispu je *LispWorks*, vývojové prostředí, které disponuje všemi potřebnými nástroji k vývoji – editorem kódu, `profiler`em, `debuger`em a mnoha dalšími. Tato implementace je volně ke stažení ve verzi *Personal edition* na adrese http://www.lispworks.com/downloads/index.html.

2. Uživatelský manuál

2.1. Popis robota Khepera

Robot Khepera III má dvě kolečka, jejiž otáčení ovládají 2 motory (každé kolečko má svůj motor) a všechny související vlastnosti s kolečky (rychlost motoru kolečka, ujetá dráha jednoho kolečka apod.) se označují anglickým prefixem `left` nebo `right` podle strany, na které se kolečko nachází. Dráha, kterou kolečko vykoná se počítá v pulsech a ve výchozím nastavení je jeden puls roven $0,047$mm. Během pohybu se zvyšuje hodnota ujeté dráhy a to i po přerušení pohybu a opětovném uvedení robota do pohybu. Pokud ujetá dráha dosáhne maximální hodnoty, je její čítač vynulován (samostatně pro každé kolečko). Hodnotu ujeté dráhy lze i nastavovat.

Robot má 2 programovatelné led diody - zelenou a červenou, které lze podle požadavků programově zapnout či vypnout. Robot je vybaven 11 infra-červenými senzory, které jsou pravidelně rozmístěny po celém obvodu těla robota, 2 senzory jsou umístěny zespodu robota na podvozku[2]. Hodnoty těchto senzorů se zvětšují s tím, jak se přibližuje robot k překážce. Hodnotu, při které robot stojí před překážkou, nelze přesně určit, protože intenzita navráceného světla závisí i na odrazové ploše a na materiálu plochy (ten může část vyzářeného světla pohltit[3]). Robot má i 5 ultrazvukových senzorů.

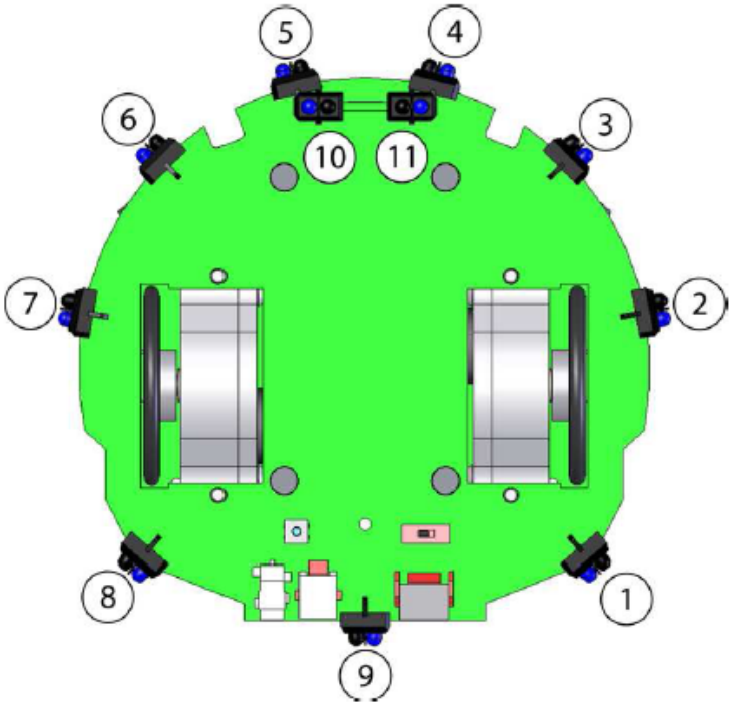
Robota lze ovládat pomocí zasílání příkazů nebo mu definovat chování pomocí jazyka, který je součástí práce.

2.2. Ovládání robota

2.2.1. Komunikace s robotem pomocí Bluetooth

[2]Lze je využít například pro zastavení robota po dosažení hrany stolu.

[3]Tento problém lze vyřešit počítáním rozdílu mezi aktuální (poslední) hodnotou senzoru a předposledně načtenou hodnotou senzoru. Pokud je tento rozdíl větší než 100 (hodnota 100 byla zjištěna empiricky), pak lze podle znaménka rozdílu určit, že se robot přiblížil k překážce a nebo oddálil od překážky. Pokud je rozdíl menší jak 100, tak se stav nemění.



Obrázek 1. Umístění infra-červených senzorů na těle robota Khepera III

Rozhraní Bluetooth se dnes nachází prakticky v každém počítači, práce s ním je jednoduchá. Samotná komunikace s robotem probíhá přes virtualizovanou sériovou linku a data se přenáší přes Bluetooth. Navázání spojení na straně počítače[4] se skládá z těchto kroků:

1. aktivování rozhraní Bluetooth,
2. kontrola, zdali je pro rozhraní Bluetooth vyhrazen některý COM port (například COM5),
3. nalezení robota pomocí funkce *Vyhledat rozhraní v dosahu* (je závislé na operačním systému hostitelského počítače),
4. vytvoření spojení s robotem a zadání bezpečnostního kódu 0000.

Poté již lze robota ovládat pomocí emulátoru terminálu či funkcí v libovolném programovacím jazyku pro komunikaci po sériovém rozhraní a pro práci s textovými řetězci. Jediným omezením je, že ***každý robot musí mít svůj*** vlastní ***vyhrazený sériový port na hostitelském počítači***[5].

2.2.2. Struktura příkazů a odpovědí pro komunikaci

Robot se ovládá pomocí příkazů. Příkazy robota jsou textové řetězce složené z ASCII znaků. Všechny příkazy mají stejnou syntaxi [5], začínají velkým písmenem označujícím význam příkazu a pokračují, pokud je to třeba, textovými či číselnými parametry oddělenými čárkou. Celý textový řetězec musí být ukončen znakem *line feed*. Pokud jsou číselné parametry větší než 8 bitů (větší než číslo 256), musí se přidávat prefix určující velikost parametru:

d pro 16 bitové hodnoty

l pro 32 bitové hodnoty

f pro hodnoty typu `float`

Pro nastavení rychlosti obou motorů robota na 20000 (32-bitová hodnota), se na sériový port pošle následující řetězec:

```
D,120000,120000
```

[4]Popsaný postup je použitelný pro počítač s operačním systém Windows 7. Postup spojení pro jiné operační systémy může být odlišný.

[5]Některé Bluetooth adaptéry využívají pro komunikaci dvou portů, pro každý směr jeden samostatný port.

Na každý úspěšně přijatý příkaz robot reaguje zasláním odpovědi, která má první písmeno stejné, jako bylo písmeno příkazu, ale v odpovědi je toto písmeno malé. Odpověď robota je textový řetězec. Pokud jsou v odpovědi předávány hodnoty, jsou oddělené čárkou, ale neobsahují prefixy podle velikosti čísla.

Následující příklad ukazuje formát odpovědi pro zjištění rychlosti:

```
;prikaz zjisteni rychlosti
E

;odpoved
e,20000,20000
```

2.2.3. Seznam příkazů

Robot Khepera III má komunikační protokol s dvaceti příkazy. V projektu se pracuje se základními příkazy uvedenými níže[6], příkazy pro konfiguraci firmware, načítání verze softwaru a další podobného charakteru nebyly potřeba.

A – příkaz spouští či ukončuje Braitenbergův mód.[7]
Syntax: `A,X`
X může nabývat hodnot 0 pro použití infra-červených senzorů, 1 pro ultrazvukové senzory a 2 pro zastavení Braitenbergova módu.

D – nastavení rychlosti dvou motorů.
Syntax: `D,speed_motor_left(32bits),speed_motor_right(32bits)`
Příklad: `D,l20000,l3234`

E – načtení rychlosti robota.
Syntax: `E`
Robot vrací odpověď ve tvaru `e,20000,3234`

F – nastavení pozice, kterou má robot dosáhnout s akcelerací a deakcelerací.
Syntax: `F,target_position_motor_left(32bits), target_position_motor_right(32bits)`
Jednotkou jsou pulsy, jeden puls je roven ve výchozím nastavení $0,047$mm.

I – nastavení aktuální pozice robota.
Syntax: `I,position_motor_left(32bits), position_motor_right(32bits)`
Jednotkou jsou pulsy.

[6] Příkazy jsou implementovány ve zdrojových kódech robota včetně reakcí na odpovědi.

[7] Pokud je spuštěn Braitenbergův mód, tak se robot samostatně pohybuje přímým směrem a když se přiblíží k překážce, tak se jí na základě hodnot senzorů vyhne a pokračuje dále.

K – zapne či vypne programovatelné led diody.
Syntax: `K,LED,X`
LED označuje, se kterou led diodou se bude pracovat, diody jsou zelená $LED = 0$ a červená $LED = 1$. X nabývá hodnot 0 pro vypnutí diody, 1 pro zapnutí diody a 2 pro přepnutí stavu.

M – inicializuje či restartuje motory. Používá se při chybě robota.
Syntax: `M`

N – načte hodnoty přiblížení senzorů
Syntax: `N`
Robot vrací odpověď ve tvaru `N,x1,...,x12` kde $x1$ až $x11$ jsou hodnoty senzorů podle obrázku 1. Hodnota $x12$ značí `timestamp` – relativní čas od posledního měření v ms.

0 – načte hodnoty infračervených senzorů
Syntax: `O`
Robot vrací odpověď ve tvaru `N,x1,...,x12` kde $x1$ až $x11$ jsou hodnoty senzorů podle obrázku 3.8 v uživatelském manuálu RobotKh3[5]. Hodnota $x12$ značí `timestamp` – relativní čas od posledního měření v ms.

P – nastaví pozici, kterou má robot dosáhnout konstantní rychlostí.
Syntax: `P,target_position_motor_left(32bits), target_position_motor_right(32bits)`
Jednotkou jsou pulsy. Dosažení znamená, že kolečko vykoná dráhu rovnu délce rozdílu zadaných pulsů a aktuálního počtu pulsů. Příklad: `P,l1000,l3000`

R – načte pozici robota.
Syntax: `R`
Robot vrací odpověď ve tvaru `r,position_motor_left,position_motor_right`.

2.3. Jazyk pro definici chování robota

Pro ovládání robota byl navržen jazyk, který je podobný regulátorovým kontrolním systémům a je použit ve vývojovém prostředí na přiloženém CD.

Regulátorové kontrolní systémy se používají v průmyslu [10], k řízení strojů a v dalších odvětvích lidské činnosti [9]. Jedná se o zařízení složené ze vstupních senzorů, výpočetní jednotky a výstupu. Regulátorem označujeme předpis, jak se má zachovat výpočetní jednotka v závislosti na vstupních hodnotách. Po rozhodnutí regulátoru se předají výsledné hodnoty na výstup. Příkladem je regulace teploty při ohřevu vody. Pokud je na čidle měřícím teplotu větší teplota, než nastavená na regulátoru, začne regulátor teplotu snižovat tím, že sníží výkon ohřívacího tělesa.

Každé zapsané pravidlo se skládá z přiřazení a případně konečně mnoha predikátů.

$$assignment\ p_0\ \ldots\ p_n \qquad n \in N_0,$$

kde *assignment* je příkaz, který se má provést, jedná se o přiřazení hodnoty či hodnoty proměnné do proměnné. Pravidlo může obsahovat predikáty p_0 až p_n – podmínky, které musí být splněny pro provedení *assignment*. Pokud neobsahuje žádný predikát ($n = 0$), je *assignment* vždy proveden a jedná se o fakt.
Přepsaný příklad do jazyka je:

$$\underbrace{sniz\ teplotu}_{assignment} \quad \underbrace{pokud\ je\ teplota\ vetsi\ nez\ 30°C}_{p_0}$$

Hodnoty používané v regulátoru se ukládají v prostředí jako proměnné. Prostředí je seznam názvů proměnných a jejich hodnot. Ve většině řešení úkolů v definovaném jazyku je použito více pravidel.

Definujeme jeden výpočetní krok jazyka, který se skládá z částí:

1. Načtení stavu všech čidel a uložení jejich hodnot do prostředí $\mathcal{P}$.
2. Vyhodnocení všech pravidel a faktů – interpret má přístup k hodnotám z prostředí $\mathcal{P}$ a v něm je i mění. Pokud není hodnota nalezena v aktuálním prostředí $\mathcal{P}$, hledá se v prostředí, které bylo na konci minulého kroku $\mathcal{P}_{last}$. Pokud ani v tomto prostředí hodnota neexistuje, je vytvořena v aktuálním prostředí $\mathcal{P}$.
3. Přenesení hodnot z aktuálního prostředí $\mathcal{P}$ do prostředí pro minulý krok $\mathcal{P}_{last}$. Přenáší se pouze hodnoty, které nejsou stavem čidel. Díky tomu lze ukládat uživatelsky definované proměnné pro příští krok.
4. Nastavení regulovaných hodnot z aktuálního prostředí $\mathcal{P}$ na výstupní zařízení.

Tento výpočetní krok stále opakujeme v intervalu určeném dle potřeby a tím jsme vytvořili jednoduchý regulátorový kontrolní systém.
Příklad použití robota a regulátorů:

```
- zastavení robota při dosažení konce desky, po které
   se pohybuje (aby z ní nespadl):
pokud je hodnota senzoru na spodní straně robota rovna 0,
   pak nastav rychlost motorů obou koleček na 0

- změna směru zatáčení robota
pokud je rychlost motoru levého kolečka větší, než rychlost
```

```
motoru pravého kolečka nebo rychlost motoru levého kolečka
je menší, než rychlost motoru pravého kolečka,
   pak nastav hodnotu rychlosti levého kolečka do
   proměnné pomocná,
   nastav hodnotu rychlosti pravého kolečka do proměnné
   rychlost levého kolečka,
   nastav hodnotu proměnné pomocná do proměnné rychlost
   pravého kolečka
```

2.3.1. Definice syntaxe jazyka

Jazyk je navržen tak, aby měl stejnou syntax zápisu jako LISP.

Přiřazení (`assignment`)
: syntax: `(variable value)`
`variable` označuje název proměnné, do které má být uložena hodnota `value`. Místo hodnoty `value` může být použit název definované proměnné. V tomto případě se do `variable` uloží hodnota uložená v proměnné s názvem `value`. Pouze v přiřazení dochází k vedlejšímu efektu.

Predikáty
: syntax: `(p symbol-1 symbol-2)`
Symbol `p` představuje binární relační symbol ($>, <, =, <=, \ldots$), který se aplikuje na `symbol-1` a `symbol-2`. Oba symboly `symbol-1` a `symbol-2` mohou být názvem proměnné či hodnotou. Pokud je symbol názvem proměnné, je použita pro vyhodnocení hodnota proměnné s tímto názvem. Vyhodnocení predikátu vrací booleovskou hodnotu a nemá žádný vedlejší efekt. Pokud je predikátů více, musí být všechny vyhodnoceny jako pravdivé, aby se provedlo přiřazení (`assignment`).

Pravidlo
: syntax: `(<- assignment p0 ... pn)`
Pokud se v pravidlu vyskytuje pouze `assignment` (přiřazení), je pravidlo faktem a provede se vždy.

Příklady pravidel s popisem:

```
;nastavi speed na 400, pokud je speed mensi nez 400
(<- (speed 400) (< speed 400))

;nastavi speed na 400, pokud speed neni 400
(<- (speed 400) (< speed 400) (> speed 400))

;vzdy nastavi speed-user na 1000
```

```
(<- (speed-user 1000))

;nastavi speed na hodnotu v speed-user
(<- (speed speed-user))

;nastavi speed na speed
(<- (speed speed))
```

3. Implementace

3.1. Vrstvy programu

Pro větší přehlednost bylo třeba rozdělit zdrojový kód do menších celků a vytvořit tak logické části programu. V této sekci jsou popsána rozhraní jednotlivých částí, nastíněn způsob jejich fungování a práce s nimi[8].

Vrstva komunikace robota – slouží pro posílání a příjem textových řetězců mezi hostitelským počítačem a robotem. Použita byla komunikace pomocí Bluetooth, který vytváří virtuální sériový port. V této vrstvě dochází k navázání spojení s robotem a uživatel nastavuje název sériového portu.

Každý robot může používat jiný druh komunikace, může být virtuálním robotem a proto bylo třeba vytvořit abstraktní třídu robota a použít dědičnost a polymorfizmus pro odvození různých typů robotů.

Třída khepera – abstraktní třída. Definuje společné metody všech odvozených tříd třídy `khepera`. Hlavní metodou je metoda `do-command`, která je použita pro zaslání příkazu robotovi a zpracování odpovědi na daný příkaz. Třída `khepera` obsahuje i implementaci metod pro práci se stavy objektu robota.

Třída robot-khepera je třídou, jejíž instance komunikuje již s fyzickým robotem Khepera III. Obsahuje implementaci zasílání příkazů přes komunikační vrstvu a implementaci metod pro úpravu stavu po příjmu odpovědi robota.

Další částí projektu je jazyk a jeho interpret. Jazyk slouží pro popis chování robota v situacích, ve kterých se může nacházet a interpret se stará o interpretaci tohoto jazyka. Jazyk je zapisován jako prostý text v textovém souboru nebo může být zadáván z vývojového prostředí LispWorks.

Třída regulator-control - instance této třídy se stará o provádění naprogramovaného kódu a úpravu hodnot v prostředí, ve kterém se s robotem pracuje. Obsahuje makro pro definování příkazů a další metody interpretu.

[8] V textu předpokládáme základní znalost programovacího jazyka LISP – zápis syntaxe, volání funkcí a metod, kompilace zdrojového kódu ...

Object-data je třída, jejíž instance slouží ke komunikaci mezi objektem robota (či jiných zařízení) s instací třídy `regulator-control`. Odstiňuje vnitřní implementace mezi různými objekty a interpretem jazyka.

3.2. Třída khepera

Každý objekt představující robota má svůj fyzický stav, kterým je rychlost, ujetá dráha jednotlivých koleček, hodnoty senzorů a hodnoty rozsvícených led diod. Všechny tyto vlastnosti se uchovávají v atributu `stats` třídy `khepera`, který je pojmenovaným seznamem neboli `p-listem`.

Dále má objekt atributy `wheel-pitch` obsahující rozteč koleček robota. Atribut `pulse-length` má hodnotu délky jednoho pulsu. Všechny délky robot počítá v pulsech. Délky se používají pro nastavení umístění, kam má robot dojet a zastavit či k načítání ujeté dráhy. Pro převod na normální délku je třeba pulsy násobit velikostí jednoho pulsu, který je ve výchozím nastavení roven $0,047$mm.

3.2.1. Programové ovládání robota

Příkaz, který má být vykonán, se pošle robotovi a očekává se odpověď od robota. Po příjmu odpovědi se kontroluje, zdali to je odpověď na předešlý příkaz (potvrzení zpracování příkazu robotem). Pokud odpoveď potvrzuje předešlý příkaz, je odpověď podle potřeby dále zpracována, v opačném případě je odpověď zahozena a signalizována chyba.

Po potvrzení příkazu je možné na ni automaticky reagovat a ošetřit vzniklý stav. Každý objekt musí dle typu příkazu aktualizovat svůj stav a k tomu využít zpracovanou odpověď.
Příklad průběhu zaslání příkazu `D,l10000,l20000` a jeho zpracování:

1. zaslání příkazu pro nastavení rychlosti robota,
2. navrácení odpovědi robotem potvrzující, že byla rychlost nastavena[9],
3. instance robota generuje příkaz pro dotaz na aktuální rychlost `E`,
4. příjem potvrzující odpovědi robota,
5. aktualizace stavu instance robota podle odpovědí na předešlý příkaz `E`.

Možné řešení automatické aktualizace je pro každý příkaz definovat funkci nebo metodu, ve které se volá metoda `do-command` a poté další příkaz pro načtení změněného stavu a zpracování odpovědi. Další řešení automatické aktualizace je pomocí kombinace metod a možnosti použití multimetod, které umožňuje programovací jazyk Lisp. Při použití tohoto řešení odpadá nutnost definovat nově pojmenované funkce nebo metody pro provedení příkazu a v nich práci s odpovědí robota.

[9]Robot nezaručuje přesné nastavení rychlosti, ta se může lišit od nastavované hodnoty.

3.2.2. Metody pro ovládání robota

Pro ovládání robota přes Bluetooth je třeba vytvořit instanci třídy `robot-khepera`. Instance obsahuje komunikační rozhraní, proměnné pro hodnoty vlastností robota potřebné pro zadávání příkazů a příjem odpovědí na ně. Všechny možné příkazy, které lze použít, jsou k nalezení v seznamu `*commands*` a popsány v oddílu 2.2.2. na straně 10.

Instance třídy `robot-khepera` se vytváří pomocí generické funkce `make-instance`. Ta vytvořený objekt vrací, je třeba jej uložit do proměnné, se kterou se bude dále pracovat. Při vytvoření instance není automaticky navázáno Bluetooth spojení. Spojení s robotem je vytvořeno až při volání metody `start`, které se mohou předat argumenty označující názevy portů pro příjem a zasílání příkazů. Během práce s robotem lze testovat, zdali objekt představující robota má spojení s robotem a při ukončení práce se musí zabrané porty uvolnit. Poté je možné využít objekt robota pro komunikaci s jiným robotem, popřípadě přes jiný sériový port či pro opětovnou komunikaci.

Metoda make-instance

syntax: **make-instance** *robot-khepera => object*
Metodou make-instance se vytvoří instance třídy `robot-khepera`, se kterou se dále pracuje.

Funkce commandp

syntax: **commandp** *symbol => boolean*
Funkcí lze zjistit, jestli je předaný symbol příkazu v třídě implementován.

Metoda start

syntax: **start** *robot-khepera* `&optional` *(port-name-read "COM5") (port-name-write "COM5") (time-out 0.0) => robot-khepera*
Pro zahájení práce s robotem se musí zavolat metoda `start` instance třídy `robot-khepera`, která naváže komunikaci přes Bluetooth. Lze určit název portu použitého pro čtení odpovědí robota, název portu použitého pro zápis příkazů a čas vypršení při čtení znaků[10].

Metoda end

syntax: **end** *robot-khepera => robot-khepera*
Metoda ukončí práci s robotem, zruší všechny zabrané zdroje a uvolní komunikační port.

Metoda startedp

syntax: **startedp** *robot-khepera => boolean*
Metoda se používá ke zjištění, zdali je robot v pohotovostním stavu – robot

[10]Pokud je překročen čas čtení jednoho znaku, tak při čekání na odpověď robota je případná načtená část odpovědi zahozena a čtení zahájeno od začátku.

může přijímat příkazy a je navázáno spojení mezi robotem a hostitelským počítačem.

Příklad vytvoření instance robota, jeho spuštění a ukončení práce:

```
;vytvoreni instance robota
CL-USER 1 > (setf *k* (make-instance 'robot-khepera))
#<ROBOT-KHEPERA 21C7010B>

;jeho spusteni, komunikuje pres port COM1
CL-USER 2 > (start *k* "COM1")
#<ROBOT-KHEPERA 21C7010B>

;pracujeme s robotem

;uvolneni zabraneho portu
CL-USER 3 > (end *k*)
#<ROBOT-KHEPERA 21890C3F>

;test, jestli je mozne komunikovat pres seriovy port
CL-USER 4 > (startedp *k*)
NIL
```

3.2.3. Generická funkce do-command

syntax: **do-command** *khepera command-symbol* `&optional` *arguments* => *robot-khepera*

Generická funkce `do-command` slouží pro zadávání příkazů. Vyžaduje symbol příkazu s prefixem : (dvojtečka) a případný seznam argumentů příkazu. Pokud se příkaz provede, vrací funkce objekt třídy robota `khepera` (či objekt potomka třídy `khepera`), v opačném případě končí vyjímkou. Pokud je použit příkaz vyžadující parametr, u kterého se musí velikost určit prefixem (oddíl 2.2.2., strana 9), je možné použít funkci `make-arguments-with-prefix`.

syntax: **make-arguments-with-prefix** *prefix* `&rest` *args* => *list*
funkce vytvoří seznam argumentů a každému přidá zadaný prefix.

Příklad různých typů příkazů a jejich zadání:

```
;vraceni informace o rychlosti
CL-USER 5 > (do-command *k* :e)
#<ROBOT-KHEPERA 21890C3F>
;odpoved robota lze nalezt ve vlastnosti stats instance
```

```
;robot-khepera (viz. dale)

;zastaveni Braitenbergova modu
CL-USER 6 > (do-command *k* :a '(2))
#<ROBOT-KHEPERA 21890C3F>

;nastaveni rychlosti
CL-USER 7 > (do-command *k* :d (make-arguments-with-prefix "l" 2000
2000))
#<ROBOT-KHEPERA 21890C3F>
```

3.2.4. Ukládání a načítání hodnot vlastností robota

Objekt robota má po svém předkovi `khepera` atribut `stats`, ve kterém se uchovávají vlastnosti objektu (rychlost levého a pravého motoru, zapnutý Braitenbergův mód ...). Pokud se načítá hodnota vlastnosti, která ještě není v objektu uložena, je navrácen `nil`.

Metoda get-stat

syntax: **get-stat** *khepera stat-key => stat-value*
Metoda se využívá pro zjištění hodnoty vlastnosti `stat-key` robota. `stat-key` představuje název vlastnosti, pro kterou hodnotu zjišťujeme.

Metoda set-stat

syntax: **set-stat** *khepera stat-key stat-value => stat-value*
Změní hodnotu vlastnosti s názvem `stat-key` na hodnotu `stat-value`.

Příklad práce s vlastnostmi objektu robota:

```
;vraci t v pripade zapnuteho Braitenbergova modu, jinak nil
CL-USER 8 > (get-stat *k* :braitenberg)
NIL

;nastavi hodnoty vlastnosti speed na predany seznam
CL-USER 9 > (set-stat *k* :speed (list :left 1000 :right 2000))
(:LEFT 1000 :RIGHT 2000)
```

3.2.5. Kombinace metod a multimetody

Jazyk Lisp dovoluje definovat nejen standardní metody, ale i metody se specifikátory `:before`, `:after` a `:around`. Klíčová slova `:before`, `:after` a `:around` určují pořadí, kdy bude metoda se specifikátorem volána. Specifikátory se používají u stejně pojmenovaných metod a umožňují před voláním standardní metody volat `:before` metodu, poté standardní (primární) metodu, a nakonec

`:after` metodu. Pokud je definována alespoň jedna `:around` metoda, tak je volána jako první i přes to, že je definována i metoda se specifikátorem `:befor`. Pokud `:arround` metoda nevolá příkaz `(call-next-method)`, tak již nedojde k volání ostatních metod se specifikátory. Všem metodám se specifikátory, které se volají, jsou předávány stejné argumenty jako standardní metodě.

V projektu bylo třeba zajistit, aby při volání `:after` metody `do-command` byl v parametru metody uložen výsledek práce předchozí standardní metody. První možnost je ukládat výsledek do atributu třídy a nebo druhá možnost je vložit výsledek do prázdného argumentu ve volání metody. Byla zvolena druhá varianta, která požaduje, aby argument zůstal identický v průběhu volání jednotlivých metod se specifikátory (jinak by se jednalo o chybu).

Identitu lze zachovat pomocí předávání argumentů ve tvaru `(cons :args '(1 2))`. Měnit se může pouze druhý objekt v `cons` páru, který musí být vždy seznamem.

Výpis funkce pro změnu `cdr` hodnoty předávaného argumentu tečkového páru:

```
(defun set-answer (answer-cons &optional answer-list)
  (setf (cdr answer-cons) answer-list))
```

Nyní lze využívat vedlejšího efektu při předávání argumentu mezi voláním jednotlivých `:before`, `:after`, `:around` metod a standardní metody[11].

Multimetody umožňují každou metodu specializovat na přesně určený objekt, který se vyskytuje v argumentu volání metody. V projektu je tato vlastnost použita při určování, která metoda má být vybraná pro odpověď robota v závislosti na zaslaném příkazu.

3.2.6. Metody do-command :after

syntax: **do-command** *khepera command-symbol* **`&optional`** *arguments answer => result*

After metoda se používá při zaslání příkazu, stará se o obsluhu odpovědi. V projektu je definováno 9 after metod pro všechny příkazy, které zjišťují stav robota, jedná se o příkazy `:a`, `:d`, `:e`, `:f`, `:i`, `:n`, `:o`, `:p` a `:r`, které byly popsány v oddílu 2.2.2. na straně 10.

V těle metody je přístupný seznam hodnot vrácené odpovědi robotem. Pro převedení textové odpovědi na seznam hodnot odpovědi od robota je třeba použít funkci `get-answer`.

syntax: **get-answer** *answer-cons => list*

[11] Jedná se o jednu generickou funkci a o její metody se specifikátory pro kombinaci metod. Tato vlastnost byla použita u metody do-command.

Příklad definice after metody pro zjištění aktuální rychlosti a uložení do stavových vlastností objektu robot-khepera:

```
;definice after metody nacitani rychlosti
(defmethod do-command :after ((robot-khepera robot-khepera)
                              (command-symbol (eql :e))
                              &optional arguments answer)
  (let ((answer (get-answer answer)))
    (set-stat robot-khepera :speed (list :left (first answer)
                                         :right (second answer)))))
```

3.2.7. Provedení příkazu - shrnutí

Pro naprogramování metody `do-commands`, která se stará o samotné ovládání robota (Provedení příkazu 3.2.1. na straně 15), byla využíta kombinace metod a multimetody.

Kombinace metod a princip multimetod je využit pro aktualizaci změněných vlastností robota, jejíž hodnoty jsou zasílány robotem. Samostatná kombinace metod je zároveň využita pro testování, zdali se příkaz provedl a byla přijata odpověď.

Metody jsou volány v tomto pořadí:

```
(do-command khepera command-symbol arguments answer)
(do-command :around khepera command-symbol arguments answer)
(do-command :after khepera command-symbol arguments answer)
```

`:After` metoda má ovšem pozměněný argument `answer` ve kterém je odpověď robota. Zároveň všechny `:after` metody jsou specializovány podle argumentu `command-symbol`. V třídě `khepera` je definována pouze jedna standardní metoda `do-command`. Ta obsahuje kód pro zaslání textového řetězce příkazu robotovi, příjem odpovědi robota a nastavení odpovědi do svého argumentu `answer`.

Implementována je opět pouze jedna metoda `do-command` se specifikátorem `:around`, která vypisuje případné chyby a konečný počet metod `commad-symbol` se specifikátorem `:after`, které se starají o aktualizaci stavu a další chování závislé na zaslaném příkazu.

Metodu se specifikátorem `:after` může být definována ke každému symbolu zastupujícímu příkaz robota. Pokud je definována, tak se provede a v jejím těle je přístupný argument `answer`, který obsahuje odpověď robota na příkaz v podobě tečkového páru, kde jednotlivé hodnoty odpovědi jsou uloženy až v `cdr` části jako seznam hodnot.

3.3. Třída robota Khepera III

Ke komunikaci robota Khepera s počítačem můžeme použít dvě metody. První možností je propojit robota pomocí propojovacího kabelu s počítačem přes COM či USB port. Tato možnost vyžaduje, aby byl v robotovi zapojen KoreBotLE modul. Druhou alternativou je bezdrátový přenos signálu přes rozhraní Bluetooth, při kterém není potřeba, aby byl robot vybaven KoreBotLE modulem. Proto byla tato možnost zvolena.

3.3.1. Komunikace Bluetooth v Lispu

Pro komunikace Bluetooth je použita virtualizace sériového rozhraní přes Bluetooth rozhraní. Po imlementační stránce je třeba se věnovat pouze programování komunikace přes sériové rozhraní.

LispWorks pro platformu Windows obsahuje již v Personal Edition knihovnu pro práci se sériovým rozhraním. Tato knihovna má název `serial-port` a pro její použití v programu je třeba tento modul načíst pomocí příkazu `(require "serial-port")`. Tento příkaz je již ve zdrojovém kódu `serial-port.lisp` a proto se jeho načtení může vynechat.

Otevření sériového rozhraní pro komunikaci

syntax: **open-port** `&optional` *(port-name "COM5") (time-out :none)*

Při otevření komunikačního portu se automaticky použijí stejné inicializační hodnoty `baud-rate, data-bits, stop-bits, parity` jako jsou v manuálu k robotu [5]. Je třeba určit název portu, přes který se bude komunikovat. Funkce vrací objekt `serial-port`, který slouží jako handle pro další funkce pracující s portem.

Zaslání textového řetězce

syntax: **write-string-port** *port string*

Pro zaslání textového řetězce robotovi se používá funkce `serial-port:write-serial-port-string`. Robot přijímá řetězec po jednotlivých znacích a k ukončení příkazu je třeba poslat jako poslední znak `LineFeed`. Funkce přebírá textový řetězec, přidá k jeho konci ukončovací znak a zašle jej robotovi.

Příjem textového řetězce

syntax: **read-string-port** *port*

Robot jako svou odpověď zasílá jednotlivé znaky a ty je třeba spojit dohromady, aby vytvořily textový řetězec s odpovědí. Pro příjem jednoho znaku je využita funkce `serial-port:read-serial-port-char`, které předáváme ještě jeden speciální znak jako příznak, že vypršel čas pro příjem znaku. Tento znak jsem zvolil `#\$`, jelikož se v odpovědích, které se mohou od robota vrátit, nevyskytuje.

Samotný příjem znaků probíhá ve smyčce, která postupně přidává přijaté znaky na konec seznamu znaků vytvářené odpovědi a končí při přijetí ukončovacího znaku `#\Return` nebo `#\$`. Pokud byl přijat ukončující znak, tak funkce vrací řetězec vytvořený ze seznamu přijatých znaků, v opačném případě vrací `nil` značící neúspěch čtení.

Ukončení práce se sériovým portem

syntax: **close-port** *port*

Pokud se sériový port nebude dále využívat, je třeba jej uzavřít, aby byl použitelný při příštím navázování komunikace.

3.4. Jazyk pro definici chování robota

Jazyk, který se používá pro programování chování robota, má lispovou syntax. Pro definici pravidla je definováno makro `<-`. Podle definice pravidla 2.3. na straně 11 je pravidlo rozděleno na dvě částí. Na přiřazení – `asignment`, které se provede vždy[12] nebo v případě splněných podmínek a na případné podmínky – `predikáty`. Pravidlo se skládá z příkazu `when`. Makro přepisuje predikáty do podmínky `when`, které jsou zřetězeny pomocí makra `and`. Pokud je makro `and` vyhodnoceno bez podmínek, vrací `t`. Je tedy vždy splněna podmínka `when`, pokud nejsou definovány žádné `predikáty`.

```
CL-USER 1 > (when (and) 1)
1

CL-USER 2 > (when (and (= 1 1) (= 2 3)) 1)
NIL
```

Dále se musí přepsat názvy proměnných na metodu, která k nim přistupuje – vrací a nastavuje jejich hodnoty. Jedná se o metody `get-variable` a `set-variable`.

Celý zdrojový kód pravidla je uvnitř anonymní funkce, které se předává prostředí, ve kterém má být vyhodnoceno. Součástí reprezentace pravidla je i textový řetězec zdrojového kódu, kterým bylo makro vytvořeno, aby bylo pravidlo čitelné i pro člověka.

Model reprezentace pravidla:

```
(cons (lambda (p) (when (and predicates)
                        (assignment)))
      (source))
```

Příklad definice pravidla:

[12] Poté se jedná o fakt.

```
CL-USER 3 > (<- (rychlost 2))
(#<anonymous interpreted function 20099DF2> . "(<- (rychlost 2))")
```

Samostatné pravidlo je anonymní funkce – bez volání této funkce se nevykonává žádný kód, nemá vedlejší efekt. Pro vyhodnocení pravidla se používá funkce `force`, která má dva parametry, kterými jsou pravidlo a prostředí, ve kterém se má pravidlo vyhodnotit.

3.4.1. Používání jazyka

Jazyk je implementován ve třídě `regulator-control`. Pro jeho použití je třeba vytvořit instanci této třídy představující interpret jazyka. Interpretace se spouští voláním metody `start-regulator`, které se předává čas při jehož dovršení se má opakovat výpočetní krok regulátorového jazyka (oddíl 2.3., strana 12). Samostatný výpočetní krok lze provést voláním metody `regulator-stage`.

Pravidla se mohou přidávat pomocí makra `add-rule` a odstraňovat metodou `remove-rule`, pro odstranění všech pravidel je definována metoda `remove-all-rules`. Uložení pravidel do souboru je implementováno metodou `save-rules-to-file` a jejich načtení ze souboru metodou `load-rules-from-file`.

Generická funkce make-instance
syntax: **make-instance** *regulator-control*
Generická funkce vytvoří instanci interpretu jazyka.

Metoda start-regulator
syntax: **start-regulator** *regulator-control (timespan* 1*)*
Voláním metody se spouští interpretace regulátoru (pokud byla předtím interpretace zastavena, tak ji opět spouští). `timespan` určuje časový úsek, po kterém se má opět provádět jeden výpočetní krok interpretace regulátoru. Čas se nastavuje v sekundách, defaultní hodnota je 1 sekunda. Delší časy jsou vhodné pro testování, kratší čas se nedoporučuje, protože může docházet k chybám způsobených delší odezvou robota na zasílané příkazy.

Metoda stop-regulator
syntax: **stop-regulator** *regulator-control*
Metoda zastavuje interpretaci regulátoru.

Metoda regulator-stage
syntax: **regulator-stage** *regulator-control*
Metoda provede jeden výpočetní krok regulátoru.

Makro add-rule
syntax: **add-rule** *regulator-control rule => list of rules*
Makro přidá pravidlo do seznamu pravidel regulátoru.

Metoda source

syntax: **source** *regulator-control (string-source nil)*
Metoda vypíše seznam vložených pravidel[13] s číslem označujícím index pravidla. Pokud není nepovinný parametr `string-source` roven `nil`, pak vrací metoda zdrojový kód pravidel včetně komentářů.

Metoda remove-rule

syntax: **remove-rule** *regulator-control index => list of rules*
Metoda odstraní pravidlo ze seznamu pravidel na pozici `index`, který lze zjistit pomocí metody `source`.

Metoda remove-all-rules

syntax: **remove-all-rules** *regulator-control => regulator-control*
Metoda odstraní všechna pravidla, která jsou v seznamu pravidel.

Metoda save-rules-to-file

syntax: **save-rules-to-file** *file-name* `&optional` *(overwrite t) => file-name*
Metoda uloží všechna pravidla, která byla přidána do seznamu pravidel a nebo byla načtena ze souboru, do souboru `file-name`. Argument `overwrite` určuje, jestli se má soubor přepsat, pokud již existuje.

Metoda load-rules-from-file

syntax: **load-rules-from-file** *file-name => list of rules*
Metoda načte zdrojový kód regulátoru ze souboru `file-name`.

Metoda set-source-code

syntax: **set-source-code** *regulator-control source-string => list of rules*
Metoda načte zdrojový kód regulátoru z řetězce `source-string`.

Příklad použití metod interpretu:

```
;vytvoreni instance interpretu
CL-USER 1 > (setf *r* (make-instance 'regulator-control))
#<REGULATOR-CONTROL 20094E83>

;pridani pravidla
CL-USER 2 > (add-rule *r* (<- (rychlost 100)))
((#<anonymous interpreted function 200E26E2> . "(<- (RYCHLOST 100) )"))

;pridani pravidla
CL-USER 3 > (add-rule *r* (<- (rychlost 10) (< rychlost 200)))
((#<anonymous interpreted function 200E26E2> . "(<- (RYCHLOST 100) )")
```

[13] Pouze pravidla, ignoruje komentáře.

```
(#<anonymous interpreted function 200B628A> .
"(<- (RYCHLOST 10) (< RYCHLOST 200))"))

;odebrani pravidla na indexu 1
CL-USER 4 > (remove-rule *r* 1)
((#<anonymous interpreted function 200E26E2> . "(<- (RYCHLOST 100) )"))

;ulozeni do souboru
CL-USER 5 > (save-rules-to-file *r* "D:/regulator.re")
"D:/regulator.re"

;smazani vsech pravidel
CL-USER 6 > (remove-all-rules *r*)
#<REGULATOR-CONTROL 21987327>

;nacteni pravidel ze souboru
CL-USER 7 > (load-rules-from-file *r* "D:/regulator.re")
((#<anonymous interpreted function 21C337B2> . "(<- (RYCHLOST 100) )"))
```

3.4.2. Komunikace mezi robotem a interpretem jazyka

Aby byl interpret použitelný pro různé druhy robotů, je třeba definovat rozhraní předávání dat mezi robotem a interpretem – třídu, jejíž instance bude všechny objekty představující roboty v interpretu zastupovat.

Třída zastřešující objekty řízené regulátorem je `regulator-objects`, instance `regulator-objects` je automaticky vytvořena při inicializaci instance třídy `regulator-control`. Umožňuje přidávat instance robotů a ukládat hodnoty proměnných, které mají roboti. Přístupná je pomocí metody `regulator-objects` instance třídy `regulator-control`.

Metoda add-object

syntax: **add-object** *regulator-objects o*
Metoda přidá objekt `o` (například instanci třídy `robot-khepera`) do instance `regulator-objects`. Tím lze v interpretu používat stavové hodnoty objektu `o` a lze s ním pracovat.

Metoda add-data

syntax: **add-data** *regulator-objects keyword value*
Metoda uloží hodnotu `value` do proměnné `keyword`, která bude použitelná při interpretaci regulátoru.

Metoda data

syntax: **data** *regulator-objects*

Návratovou hodnotou jsou všechny hodnoty proměnných, které obsahuje interpret jazyka. Proměnné jsou uloženy v `plistu`.

Příklad přidání instance robota do interpretu:

```
;vytvorime interpret
CL-USER 1 > (setf *r* (make-instance 'regulator-control))
#<REGULATOR-CONTROL 21A2552F>

;vytvorime instanci robota
CL-USER 2 > (setf *k* (make-instance 'robot-khepera))
#<KHEPERA 2009CA7F>

;pridame instanci robota
CL-USER 3 > (add-object (regulator-objects *r*) *k*)
(#<KHEPERA 2009CA7F>)
```

Každý objekt, který představuje robota, musí implementovat metody `get-data-from-object` a `set-data-from-language`.

Metoda get-data-from-object

definice: **get-data-from-object** *o regulator-objects => plist*
Metoda slouží pro předávání stavu objektu (hodnot senzorů, rychlosti motorů ...) pro který je definována, instanci `regulator-objects`. V metodě je implementováno předávání proměnných pomocí metody `add-data`. Proměnné nemohou být předávány do interpretu jazyka jiným způsobem.

Metoda add-data

definice: **add-data** *regulator-objects keyword value*
Metoda nastaví hodnotu proměnné `keyword` na `value` v aktuálním prostředí, které používá interpret. Pokud proměnná neexistuje, je prostředí vytvořena.

Metoda set-data-from-language

definice: **set-data-from-language** *o regulator-objects*
Metoda nastavuje hodnoty proměnných objektu (zapnutí Braintenbergova módu, rychlost motorů ...) hodnotami, které jsou uloženy v předáváné instanci `regulator-objects`. K datům se přistupuje pomocí metody `data`.

Příklad implementace metod robota:

```
(defmethod get-data-from-object ((khepera khepera) regulator-objects)
  (add-data regulator-objects 'rychlost-leva 1000)
  (add-data regulator-objects 'rychlost-prava 1000))
```

```
;function-set-stat-to-robot - metoda pristupujici
;k promennym v robotovi
(defmethod set-data-from-language ((khepera khepera) regulator-objects)
  (let ((data (data regulator-objects)))
    (function-set-stat-to-robot (getf data :rychlost-leva))
    (function-set-stat-to-robot (getf data :rychlost-prava))))
```

4. Rychlý návod k použití

Následuje jednoduchý návod, který popisuje postup, jak spustit robota a vývojové prostředí. Předpokladem je operační systém Windows, spuštění LispWorks a otevírání zdrojových souborů a jejich kompilace[14]. Zároveň je třeba mít aktivováno sériové rozhraní přes Bluetooth (viz. 2.2.1. na straně 7)[15].

1. Otevřte soubor *run.lisp* a ten zkompilujte,
2. napište název sériového portu, který se má použít ke komunikaci[16] a stiskněte *Use serial-port*,
3. načtěte uložený zdrojový kód regulátoru pomocí volby *Load source code* nebo jej vytvořte v editoru,
4. spusťte interpretaci pomocí tlačítka *Start interpreting* (pokud probíhá interpretace, nelze měnit zdrojový kód a ani načíst jiný zdrojový kód ze souboru),
5. interpretaci můžete zastavit pomocí tlačítka *Stop interpreting*.

Výpis proměnných, které lze používat ve vývojovém prostředí:

1. `:left-speed`
2. `:right-speed`
3. `:left-position`
4. `:right-position`
5. `:proximity-X`, kde X je číslo od 1 do 11

[14]Kompilovat lze soubor pomocí volby Wokrs/Buffer/Compile nebo kliknutím na ikonu "listu papíru se závorkou a bleskem" v editoru zdrojového kódu.

[15]Tento postup je pro pracovní stanice s operačním systémem Windows, pro ostatní systémy může být odlišný.

[16]Je možné zvolit rozdílné porty pro směr komunikace do robota (zasílání) a od robota (příjem).

6. `:ambient-X`, kde X je číslo od 1 do 5
7. `:braintenberg`

Závěr

Cílem této práce bylo implementovat programové ovládání robotů Khepera III včetně ukládání jejich stavů a vytvořit jednoduchý jazyk, ve kterém lze definovat chování robotů.

Roboty lze ovládat z příkazové řádky LispWorks pomocí jedné funkce a zároveň ovlivňovat jejich chování pomocí ovládacího jazyka. Implementace umožňuje definovat automatické chování na zadaný příkaz. Jazyk má intuitivní syntaxi – vyžaduje znát názvy vlastností robota.

Předmětem dalšího rozšíření by mohla být implementace virtuálního robota vhodného pro testování, virtuální desky, na které by se virtuální robot pohyboval mezi překážkami a grafické zobrazení virtuální desky.

Conclusions

The aim of this work was to implement robots KheperaIII software control including the imposition of conditions and create a simple language in which you can specify the behavior of robots.

Robots can be controlled from the LispWorks command line using one functions and influence their behavior using controll language. Implementation allows definition automatic behavior on the specified command. Language syntax is intuitive - it requires to know the robot properties names.

Subject to further extension could be the implementation of a virtual robot suitable for testing, virtual board, which would the virtual robot moving among obstacles and graphical display of virtual board.

Reference

[1] GRAHAM, P. *ANSI Common Lisp.* Prentice Hall 1995.

[2] GRAHAM, P. *On Lisp.* Prentice Hall 1993.

[3] McCarthy, John. *Recursive Functions of Symbolic Expressions and Their Computation by Machine.* Massachusetts Institute of Technology, Cambridge, Mass 1960.

[4] BARSKI, C. *Land of Lisp.* San Francisco 2011

[5] *Uživatelský manuál k robotum Khepera 3 (s.a.)*

[6] *LispWorks User Guide and Reference Manual (s.a.)*

[7] *Common Lisp HyperSpec (s.a.)*

[8] *The Common Lisp Cookbook Project (s.a.)*

[9] *Regulator (automatic control) (s.a.)*

[10] *Regulators / Control Systems (s.a.)*

A. Obsah přiloženého CD

Přiložené CD obsahuje zdrojové texty programu a diplomové práce.

`doc/`
: Dokumentace práce ve formátu PDF, vytvořená dle závazného stylu KI PřF pro diplomové práce, včetně všech příloh, a všechny soubory nutné pro bezproblémové vygenerování PDF souboru dokumentace (v ZIP archivu), tj. zdrojový text dokumentace, vložené obrázky, apd.

`src/`
: Kompletní zdrojové texty programu se všemi potřebnými (převzatými) zdrojovými texty, knihovnami a dalšími soubory (v ZIP archivu).