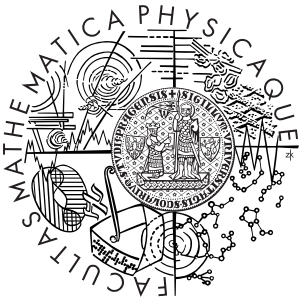
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

# BAKALÁŘSKÁ PRÁCE




Tomáš Křen

# Nástroj pro programování ve fyzikálním prostředí

**Katedra softwarového inženýrství**

Vedoucí bakalářské práce: RNDr. Petr Hnětynka Ph.D.

Studijní program: Informatika, Programování

Praha 2011

Děkuji vedoucímu práce RNDr. Petru Hnětynkovi Ph.D. za rady, které mi pomohly při této práci.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 15. května 2011 Tomáš Křen

**Název práce:** Nástroj pro programování ve fyzikálním prostředí
**Autor:** Tomáš Křen
**Katedra:** Katedra softwarového inženýrství
**Vedoucí bakalářské práce:** RNDr. Petr Hnětynka Ph.D., Katedra distribuovaných a spolehlivých systémů
**e-mail vedoucího:** hnetynka@d3s.mff.cuni.cz

**Abstrakt:** cílem práce je vytvořit program fungující jako nástroj pro programování v dvojrozměrném fyzikálním prostředí. z hlediska uživatelského rozhraní bude podobny hře the incredible machine. program bude umožňovat uživateli interaktivně vytvářet stroje sestavené z předmětů různých typů. interakci předmětů mezi sebou navzájem bude zajišťovat fyzikální simulátor. (...)

**Klíčová slova:** klíčová slova...

**Title:** Tool for programming in a physical environment
**Author:** Tomáš Křen
**Department:** Department of Software Engineering
**Supervisor:** RNDr. Petr Hnětynka Ph.D., Department of Distributed and Dependable Systems
**Supervisor's e-mail address:** hnetynka@d3s.mff.cuni.cz

**Abstract:** In the present work we study ... cílem práce je vytvořit program fungující jako nástroj pro programování v dvojrozměrném fyzikálním prostředí. z hlediska uživatelského rozhraní bude podobny hře the incredible machine. program bude umožňovat uživateli interaktivně vytvářet stroje sestavené z předmětů různých typů. interakci předmětů mezi (...)

**Keywords:** klíčová slova...

# Obsah

# 1 Úvod

## 1.1 Stručný popis programu a jeho cíle

Program *Kutil* je hra pojatá jako interaktivní fyzikální prostředí, ve kterém vkládáním, přesouváním a propojováním objektů pomocí myši v dvourozměrném prostoru hráč vytváří virtuální svět. Tento virtuální svět je hierarchický; tím myslíme to, že *uvnitř* každého objektu se nachází opět prostor, kam je možno vkládat další objekty. Hráč má možnost se touto hierarchií volně pohybovat.

Celý takto vytvořený svět, nebo případně jeho části, však zároveň reprezentují syntaxi programu. Toho je docíleno především tím, že ve hře jsou různé druhy objektů nazývané *funkce*, které zastávají stejnou roli, jako funkce v klasických programovacích jazycích. Jsou to navzájem propojitelné „krabičky“ s různým počtem vstupů (kterými objekty v roli argumentů padají dovnitř) a výstupů (ze kterých objekty v roli výsledků vypadávají ven).

Vedle těchto pasivních funkcí reagujících na vstup, jsou zde i aktivní objekty zastávající roli agentů ve virtuálním světě. Ti jsou řízeni buďto programem v podobě soustavy navzájem propojených funkcí ve vnitřku těchto agentů, nebo přímo hráčem z klávesnice.

Hráč má možnost kdykoliv spustit či zastavit běh simulace virtuálního světa, nebo se pohybovat zpět či vpřed v historii editačních změn podobně, jako je to běžné například v textových editorech.

Výše jsme naznačili dva pohledy, jak se na hru Kutil můžeme dívat: Buďto jako na *interpret* programovacího jazyka, nebo jako na *prohlížeč* virtuálního světa. Jeho role prohlížeče, je v mnohém podobná webovému prohlížeči: V každém kroku simulace je k dispozici XML reprezentace aktuálního stavu virtuálního světa. Cílem této XML reprezentace je snaha o „uživatelskou přívětivost“ pro čtení a ruční editování člověkem. Dále pak je cílem snaha o dostatečnou modularitu této reprezentace, tak aby nebránila přidávání dalších nových typů objektů v budoucnu. Program také podporuje jednoduchou síťovou komunikaci skrze minimalistický webový server.

Samotné GUI programu je stejného charakteru jako ostatní prvky virtuálního světa, tedy není zde pevná hranice mezi GUI a virtuálním světem, který je skrze GUI editován - tyto dvě věci v sebe volně přecházejí.

Vedle základní manipulace s objekty pomocí myši program umožňuje pokročilejší variantu interakce; pomocí textové konzole. Interakce skrze konzoli umožňuje jednak zadávat programu nejrůznější příkazy, hlavně je však prostředkem pro vkládání složitějších objektů. Pro tento účel je v programu obsažen jednoduchý minimalistický programovací jazyk Kisp, inspirovaný programovacím jazykem Lisp. Ten na jedné straně umožňuje vkládat složitější objekty reprezentující hierarchické datové struktury, dále však také umožňuje vkládat složené funkce; tedy funkce složené z elementárních nebo uživatelsky definovaných funkcí. Toto skládání funkcí funguje na základě *částečné aplikace funkce* a definice funkcí pomocí *lambda výrazů*.

Cílem této práce je vytvoření programu přibližujícího dětem svět programování skrze intuitivnost fyzikálního světa. Pojem hra se zde chápe spíše jako „stavebnice“, než klasická počítačová hra s vyvíjejícím se příběhem. Více než hotovou hrou s příběhem se program snaží být nástrojem na tvorbu takovýchto her samotnými dětmi.

## 1.2 Struktura práce

# 2 Teoretické pozadí řešených problémů

## 2.1 Fyzikální simulace

## 2.2 Genetický programování

## 2.3 Kisp - včetně hrotby lambda kalkulu -> částečná aplykace funkce

## 2.4 XML

# 3 Analýza řešení

## 3.1 Stručný rozbor fází běhu programu

Nejprve se podíváme na velice stručný přehled jednotlivých fází běhu programu; od jeho spuštění po jeho ukončení. Podrobnější rozbor jednotlivostí pak následuje v dalších částech této kapitoly.

První akcí programu po jeho spuštění je nahrání XML reprezentace virtuálního světa ze souboru. Pokud je program zavolán s parametrem, interpretuje se první parametr jako jméno souboru, který bude nahrán. Pokud je program zavolán bez parametrů, načte se implicitní interní soubor.

Tento soubor popisuje strukturu objektů virtuálního světa, kterážto koresponduje se strukturou objektů v rámci jazyku Java, ve kterém je program napsán. Objekty tvoří hierarchii na jejímž vršku je jednoduchý plánovač, který vybízí jednotlivé objekty k akci; analogicky každý další objekt vybízí k akci objekty v hierarchii pod ním.

Součástí této hierarchie je i popis GUI včetně oken, ve kterých se vše zobrazuje. Tato okna mohou být navzájem vnořená.

Uživatel interaguje s programem pomocí myši a klávesnice, případně pomocí integrované textové konzole (ta není součástí hierarchie).

Pokud je zavřeno okno, které není vnořené (tzn. to které vystupuje jako okno programu), program je ukončen.

## 3.2 Provázanost XML reprezentace a objektů jazyka Java

Nyní popíšeme mechanizmus jakým jsou provázány objekty v Javě (reprezentující objekty prostředí) s jejich textovou XML reprezentací.

Je možno rozlišit tři různé věci:

- Objekt ve virtuálním světě hry,
- objekt jazyka Java, který je v pozadí tohoto objektu

- a XML reprezentaci objektu, kterou můžeme chápat jako předpis určující oba tyto objekty.

Program dělá to, že vezme XML reprezentaci a na jejím základě vytvoří nový objekt Javy. Během existence objektu Javy je pak kdykoliv k dispozici jeho aktuální XML reprezentace. Jinými slovy: XML reprezentaci můžeme přeložit na objekt Javy a obráceně objekt Javy můžeme přeložit na XML reprezentaci.

### 3.2.1 Představa abstraktního popisu objektů virtuálního světa

Nyní popíšeme abstakci objektů virtuálního světa na základě níž je pak definována XML reprezentace těchto objektů.

V rámci této abstrakce objektem chápeme:

- *Textový řetězec*, nebo
- seznam dvojic *( textový řetězec, seznam objektů )*.

První možnost je *elementární objekt*, sestávající pouze z textového řetězce. Druhá možnost je *složený objekt*, přičemž ony dvojice chápeme ve smyslu dvojice *klíč* (tj. textový řetězec) a *hodnota* (tj. seznam objektů). Takový objekt chápeme složený ze *součástek*, kde každá součástka je jeden seznam objektů. A každá součástka má svůj klíč udavající, jakou roli v objektu má tato součástka.

Takto reprezentovaný objekt můžeme jednoduše zapsat. (Seznam zapišeme jako řadu hodnot v hranatých závorkách.)

Jako příklad uveďme následující objekt:

```
[ ( key1, [value1]        ),
  ( key2, [ [( key3, []    )],
            [( key4, [[]]  )] ] )
]
```

Je vidět že tento způsob zápisu je velice nepřehledný (a proto ho nebudeme vůbec používat).

### 3.2.2 XML reprezentace jako zápis abstraktního popisu objektu virtuálního světa

Podívejme se na to, jak je XML reprezentace používaná v programu Kutil zápisem výše popsané abstrakce objektu virtuálního světa.

1. Textový řetězec zapíšeme jako textový řetězec, ve kterém nahradíme speciální znaky XML odpovídajícím kódem.

2. Složený objekt
   `[ (key1 , (...) ) , (key2 , (...) ) , ... , (keyN , (...) ) ]`
   zapíšeme jako:

```
<object>

    <key1> (...) </key1>
    <key2> (...) </key2>
    ...
    <keyN> (...) </keyN>

</object>
```

3. Seznam objektů zapíšeme tak, že jednotlivé objekty zapíšeme za sebou jak následují.

Pro příklad z 3.2.2 máme tedy následující zápis:

```
<object>

    <key1>value1</key1>
    <key2>
        <object><key3></key3></object>
        <object><key4><object/></key4></object>
    </key2>

</object>
```

Využíváme dále toho, že v XML je pojem atribut. Využijeme toho pokud nějaký seznam objektů (v roli hodnota ve dvojici klíč - hodnota) je jednoprvkový

a jeho prvek je textový řetězec. V našem příkladu to splňuje dvojice `(key1, [value1] )`. Potom můžeme psát:

```
<object key1="value1">
    <key2>(...)</key2>
</object>
```

Celá tato konstrukce má následující motivaci: V XML není nijak pevně stanoveno, kdy pro nějaký konstrukt použít atribut a kdy vnořený element. Je zde však jedno podstatné omezení: atributy nemohou obsahovat strukturovaná XML data, pouze textový řetězec. Chápáním atributu jako „syntaktického cukru“ nám umožňuje nesvazovat jednotlivé klíče s konkrétním typem objektu.

Poslední „syntaktický cukr“ je svázán s důležitým klíčem `inside`. Uvažme tyto dva zápisy podle tohoto pravidla jsou ekvivalentní, první:

```
<object>
    <inside>
        <object>(...)</object>
        <object>(...)</object>
    </inside>
    <key1>
        <object>(...)</object>
        <object>(...)</object>
    </key1>
</object>
```

Druhý:

```
<object>
    <object>(...)</object>
    <object>(...)</object>
    <key1>
        <object>(...)</object>
        <object>(...)</object>
    </key1>
</object>
```

Čili pokud je nějaký element `object` přímo v elementu `object`, znamená to, že je implicitně pod klíčem `inside`.

### 3.2.3 XML reprezentace jako zápis objektu jazyka Java

Nyní se podíváme jak jsou na základě XML reprezentace konstruovány objekty Javy.

První co je potřeba určit je, konstruktor jaké třídy má být pro daný objekt zavolán. Tato informace se nachází v klíči *type*. Pokud tento klíč objekt nemá, nebo je-li obsahem tohoto klíče něco jiného než plátný kód třídy, je použit konstruktor třídy *Basic*, což je základní typ objektu od kterého dědí všechny ostatní složitější objekty.

Tomuto konstruktoru se předají všechny dvojice *klíč - seznam objektů*, které tento objekt obsahuje. Tyto objekty už dostává ve formě objektů Javy (tedy vnitřní objekty se inicializují dříve, než samotný objekt). Každý konstruktor tedy dostane balíček (implementovaný třídou *KAtts*) dvojic *klíč - seznam objektů*. Není pevně určeno jak by měl konstruktor na tento balíček reagovat, v samotném kódu programu se však všude dodržují jednoduché konvence pro zprávu těchto záležitostí, tak aby každý objekt mohl jednoduše na požádání vrátit svou aktuální XML reprezentaci.

Dalším důležitým klíčem je klíč *id*, předpokládá se, že je jeho hodnota textový řetězec. Ten složí jako unikátní identifikační symbol onoho objektu. Díky

*id* spolu můžou interagovat objekty nezávisle na své pozici v hierarchii. Pokud objekt nemá uvedeno explicitní *id*, dostane přiděleno nějaké unikátní. Konvence je taková, že píšeme id začínající symbolem *$* a dále obsahující jen alfanumerické znaky a podtržítko. Program obsahuje globální databázi všech objektů přístupnou všem objektům, v níž je přístup k objektům zajištěn na základě znalosti jejich *id*. *Id* však sebou přináší problém, díky tomu, že v době konstrukce objektů ještě není známo *id* nadřazených objektů v hierarchii. To je v programu řešeno tak, že vytvoření objektu probíhá ve dvou fázích. První fáze je zavolání konstruktoru a druhá fáze je zavolání funkce *init()*. Ve chvíli zavolání této funkce už jsou všechna *id* známá a tak může objekt dokončit své vytvoření. Objekty nevyužívající přímé reference na jiné objekty tuto funkci mohou ignorovat.

Většina objekty v současném stavu programu využívá pouze klíče mající hodnotu textový řetězec, nebo dříve zmíněný důležitý klíč *inside*.[1] Proto se na klíč *inside* podívejme podrobněji. Každý objekt mající svůj odraz v GUI programu je umístěn ve *vnitřku* nějakého objektu, a naopak má svůj *vnitřek,* v němž mohou být umístěny další objekty. A součástí GUI je možnost volně se pohybovat hierarchií vzniklou tímto vztahem *vnitřku* a *vnějšku.* Klíč *inside* je právě tento *vnitřek.*

## 3.3 Některé základní typům oběktů

Ve stručnosti se podíváme na některé typy objektů.

### 3.3.1 Basic

Třída *Basic* představuje základní typ objektu virtuálního světa, od něhož všechny ostatní typy předmětů dědí.

Na příkladě si ukážeme, jaké může mít vlastnosti:

[1] Tuto typickou jednoduchou struturu porušují například objekty v roli agentů, kteří využívají možnosti složitější struktury pro implementaci paměti oddělené od programu.

```
<object type="basic" id="$1" pos="100 150"
        shape="rectangle 200 100"
        physical="true" attached="false">

    <object id="$2" pos="0 0"/>
    <object id="$3" pos="50 50"/>

</object>
```

Jedná se o základní objekt, který má ve svém vnitřku dva další základní objekty (ty sice nemají uvedený typ, typ *basic* je však implicitní). Unikátní id tohoto objektu je `$1`. Pozice objektu v rámci svého *rodiče* (tzn. objektu jehož vnitřku je součástí) je `[100, 150]`. Má tvar obdélníku o stranách $200 \times 100$. Položka `physical="true"` určuje, že objekt je fyzický. Tím se myslí, to že se účastní fyzikální simulace uvnitř svého rodiče, a tedy interaguje s ostatními fyzickými předměty. Položka `attached="false"` určuje, že se může volně pohybovat (nemá pevně fixovanou pozici).

Každý základní objekt má pro své vnitří fyzické objekty svět fyzikální smulace, ve kterém má každý vnitřní objekt odpovídající těleso, podle kterého si aktualizuje svou pozici.

Akce prováděné objektem v jednom kroku simulace se provedou zavoláním metody `step()`. To má mimo jiné za následek krok vnitřního světa fyzikální simulace a zavolání metody `step()` u svých vnitřních objektů.

### 3.3.2 Time

Víše jsme uvedli, že aby objekt provedl krok simulace, musí být zavolána jeho metoda `step()`. Tu typicky volá rodič objektu. Problém nastává, když objekt nemá rodiče, tedy když je kořenem hierarchie. K tomuto účelu slouží instance třídy `Time`.

Při startu programu jsou všechny kořenové objekty tohoto typu předány jednoduchému plánovači. Ten potom v pravidelných intervalech volá metodu step těchto objektů. To jak často je ten který objekt zavolán je dáno jeho položkou `ups` (updates per second). Pokud chceme jen určitý počet iterací (tzn. nechceme,

aby se volání tohoto objekt opakovalo donekonečna) můžeme to specifikovat položkou `iterations`.

Ukažeme to na příkladu:

```
<object type="time" ups="80">

    <object/>

</object>
<object type="time" ups="35" iterations="100">

    <object/>
    <object/>

</object>
```

První `time` bude volán 80 krát za sekundu stále dokola, zatímco druhý `time` bude volán 35 krát za sekundu, ale jen celkově stokrát.

### 3.3.3 Frame

Další důležitou třídou je `Frame`. Ta zajišťuje základní prvky uživatelského rozhraní, její objekty se totiž manifestují jako okna.

Pokud je rodičem `framu` přímo `time`, pak se `frame` manifestuje jako okno programu. Pokud ne, je takzvaným vnořeným oknem a zobrazí se až uvnitř jiného okna, podobně jako ostatní objekty.

`Frame` má položku `target`, což je id objektu, jehož vnitřek toto okno zobrazuje. Pokud není uveden, je automaticky `targetem` tento `frame` samotný. Dále má položku `cam`, která určuje pozici, kterou ve vnitřku `target` objektu zobrazuje.

### 3.3.4 Příklad minimalistického GUI

Na závěr této podkapitoli uvedeme příklad minimilastického GUI, které demonstruje jak se používají výše zmíněné konstrukty. Následující XML kód je podoba celého souboru, který můžeme eventuelně spustit programem:

```
<?xml version="1.0" encoding="UTF-8"?>
<kutil>
  <object type="time" ups="80">
    <object type="frame" main="true" id="$1" size="610 260"
            pos="200 200">
      <object type="button" title="pause/play" cmd="play" pos="30 30" />
      <object type="button" title="Undo" cmd="undo" pos="30 60"/>
      <object type="frame" id="$2" size="130 130" pos="10 100">
        <object pos="50 10" physical="true" attached="false"/>
        <object pos="14 103" shape="rectangle 100 10" physical="true"
                attached="true"/>
      </object>
      <object type="frame" target="$2" showXML="true" size="450 130"
              pos="150 100" />
    </object>
  </object>
</kutil>
```

Kdybychom spustili tento soubor programem (například pomocí `java -jar kutil.jar` *`filename.xml`*), dostaneme následující GUI:


Rozeberme tento příklad podrobněji. Objekt s id `$1` typu `frame` zobrazuje svůj vlastní vnitřek. Protože se nachází přímo v `time`, manifestuje se jako okno programu. Jeho první dva vnitřní objekty jsou typu `button`. `Button` vypadá jako

odkaz, po kliknutí na něj se provede akce s kódem v položce `cmd`. První tlačítko přepíná stav simulace mezi stavy „play“ a „pause“. Druhé tlačítko vrátí stav o jednu editační změnu zpět tomu objektu, který má položku `main` nastavenu na `true` (takový by měl být v celém GUI právě jeden a zde je to objekt `$1`).

Další dva objekty jsou oba typu `frame`. První s id `$2` má za `target` implicitně sebe. Druhý má za `target` také objekt s id `$2`, navíc má položku `showXML` nastavenou na `true`, což má za důsledek, že ukazuje svůj `target` ve formě XML reprezentace.

Obsah objektu `$2` jsou dva fyzické předměty, první volný, druhý vázaný ke své pozici.

Když spustíme simulaci stisknutím tlačítka *„play/pause“*, vrchní fyzický předmět začne padat až se zastaví pádem na druhý fyzický předmět.

Stisknutím tlačítka *„Undo“* se stav vráti do stavu před spuštěním simulace.

## 3.4 Objekty reprezentující data

V programu jsou různé objekty reprezentující data. Zde se zmíníme o následujících: čísla, symboly, seznamy, boolovské hodnoty a směry.

Všechny tyto objekty mají společné, že to jsou fyzické volné objekty. Slouží pak jako vstup a výstup funkcí.

K jejich rychlému vytvoření můžeme použít textovou konzoly programu.

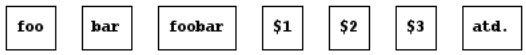
Nyní se na každý typ z výše zmíněných objektů velice ve stručnosti podíváme.

### 3.4.1 Čísla

Čísla, přesněji celá čísla, jsou reprezentována takovýmito žlutými míčky:

### 3.4.2 Symboly

Symboly (textové řetězce bez mezer) jsou reprezentovány takovýmito bílými obdélníky:




### 3.4.3 Boolovské hodnoty

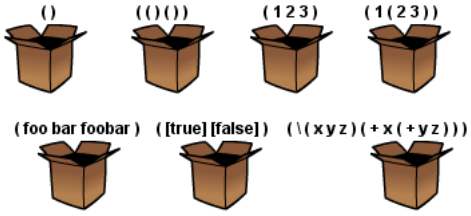
Boolovské hodnoty jsou reprezentvány bílým, respektive černým, míčkem:


### 3.4.4 Směry

Je pět druhů směru (nahoru, dolu, doleva, doprava, náhodně). Směry jsou reprezentovány takovýmito čtverci:

### 3.4.5 Seznamy

Seznam slouží jako uspořádaná posloupnost objektů. Oproti chování základního objektu se jeho vnitřním objektům nevolá metoda `step()`, tzn. po dobu co je objekt uvnitř seznamu je „zamrzlý“. Seznam má tvar krabice, nad níž je napsána jeho reprezentace v Kispu (podrobně o Kispu v části 3.5).

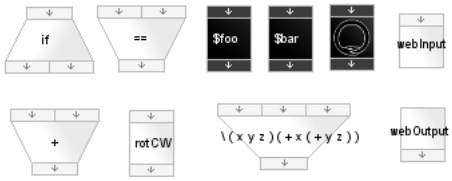


První seznam na obrázku je prázdný. Druhý obsahuje dva prázdné seznamy. Třetí obsahuje čísla 1, 2 a 3. Čtvrtý obsahuje číslo 1 a seznam obsahující čísla 2 a 3. Pátý obsahuje symboly *foo*, *bar* a *foobar*. Šestý obsahuje boolovské hodnoty *true* a *false*. A sedmý obsahuje symbol \, seznam obsahující symboyl $x$, $y$ a $z$ a seznam obsahující: symboly $+$ a $x$ a seznam obsahující symboly $+$, $y$ a $z$.

## 3.5 Funkce a Kisp

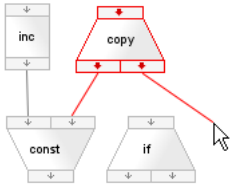
Funkce je objekt který má na svém vršku několik vstupů a na svém spodku několik výstupů.

Následující obrázek ukazuje několik příkladů různých funkcí:




Každy vstup a výstup je označen šipkou. Funkce můžeme navzájem napojovat tak, že klikneme na výstup nějaké funkce, tím se nám u kurzoru objeví čára, která symbolizuje propojení, když nyní klikneme na nějaký vstup, tak tím propojíme vstup a výstup.

Na následujícím obrázku je tento princip naznačen:

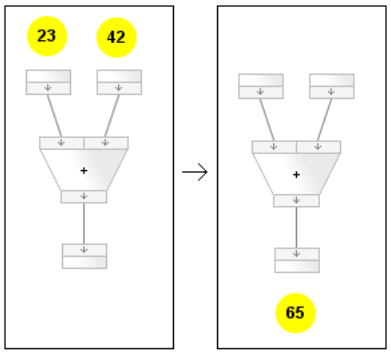



Funkce jsou implicitně nefyzické objekty (nejsou součástí symulace, tzn neiteragují s ostatními fyzickými předměty). Pro to abychom mohli do funkce dosadit nějaký objekt se používá objekt *in* (což je de facto speciální funkce s nula vstupy

a jedním výstupem). *In* je fyzický předmět s pevnou pozicí, kterého když se nějaký objekt dotkne, tak ten dotyčný objekt zmizí a stává se „daty“ dosazenými do funkce. Jeho protějšek *out* naopak svůj vstup zhmotní.

Čili objekty „padají“ do objektu *in*, následně s nimi funkce provedou nějaké operace, načež výstupní objekty „vypadnou“ z objektu *out*.

Následující obrázek ukazuje příklad průběhu výpočtu funkce počítající součet dvou čísel.




Rozlišujeme dva základní typy funkcí: *bíle* funkce a *černé* funkce. To co je odlišuje, je způsob jakým je určeno jejich chování. Chování bílý funkcí je určeno textově, je zapsáno pomocí jednoduchého jazyka Kisp. Naproti tomu chování černých funkcí je určeno strukturou jejich vnních objektů.

### 3.5.1 Kisp

Kisp je jednoduchý minimalistický programovací jazyk určený pro definování bílých funkcí a pro rychlé vytváření složitějších objektů pomocí textové konzole programu. Syntaxí se víceméně jedná o zjednodušený Lisp.

Z jazyku Haskell si Kisp vypůjčuje koncept částečné aplikace funkce: Ke každé funkci se chová jako by to byla funkce jedné proměnné, pokud se jedná o funkci n proměnných, vrací tato funkce n proměných jako výsledek funkci n-1 proměnných.

Vezměme si jako příklad funkci +, chápanou jako funkci dvou proměnných. Všechny funkce v Kispu jsou brány jako prefixové. Dosazení do funkce se zpisuje

jako jméno funkce následované dosazovaným výrazem, odděleno mezerou. Mezeru můžeme chápat jako infixový operátor aplikace funkce. Představme si, že chceme sečíst čísla 23 a 42. Zápis této operace v Kispu je následující:

( + 23 ) 42

Protože aplikace funkce se chápe v Kispu jako asociativní zleva, je tento výraz ekvivalentní výrazu:

+ 23 42

V našem příkladě po dosazení čísla 23 do funkce + vzniká nová funkce „+ 23“, do níž když dosadíme 42 , tak sečte 23+42 a vrátí 65.

Podobně jako v Lispu je v Kispu úzce provázan výraz v závorkách a seznam. K odlišení seznamu od výrazu při vyhodnocování slouží symbol '. Pro následující příklad využijeme funkci head, která vrací první prvek seznamu.

head '( + 2 3 )

Tento výraz se vyhodnotí na hodnotu symbol +, to díky tomu, před výrazem ( + 2 3 ) je symbol ', který zajistí že se výraz nevyhodnotí a místo toho se bude interpretovat jako seznam, jehož první prvek je symbol +.

Další konstrukcí Kispu je takzvaný lambda výraz. Ten umožňuje definovat nové funkce. Lambda výraz odpovídá následujícímu schematu:

( \ *argumenty-funkce tělo-funkce* )

Kde argumenty-funkce může být buď jeden symbol nebo seznam symbolů a kde tělo-funkce je nějaký výraz Kispu. Předveďme si to na názorném příkladu:

( \ x ( + x x ) ) 42

Máme zde funkci definovanou lambda výrazem a do níž je dosazena hodnota 42. Jako argumenty-funkce zde máme jednodušší možnost jediného symbolu. V tomto případě výpočet funkce probýhá tak, že se všechny výskyty tohoto symbolu v těle-funkce nahradí dosazenou hodnotou a následně se tento vzniklý výraz vyhodnotí, jeho hodnota je návratovou hodnotou funkce. Díky tomu je výsledná hodnota našeho příkladu číslo 84.

Ještě nám zbývá koplikovanější možnost, kdy argumenty-funkce je seznam symbolů. Tato možnost reprezentuje funkci více argumentů.

( \ ( x y z ) (+ x (+ y z) ) ) 23

Tato složitější varianta funguje tak, že po dosazení dostáváme:
( \ ( y z ) (+ 23 (+ y z) )

Neboli odstraní se první prvek ze seznamu argumenty-funkce a dále se pokračuje analogicky jako v předchozím příkladě.

### 3.5.2 Bílé funkce

Bílá funkce je definována

### 3.5.3 Černé funkce

### 3.5.4 Rekurze

### 3.5.5 promluvit tu o web input output

miniaturní webovej server na kutil.php5.cz, spíš experimentální fičura

## 3.6 Agenti

budha a moucha

budha umí manipulovat věcma krs klavesy

mužeme hejbat jednim nebo všema, de přepínat

moucha ma svuj vnitřní program

na to má sensory

## 3.7 Ostatní

tlačítko, tool

### 3.7.1 Příkazy - ty na tlačítkách a do konzole

### 3.7.2 uživatelský save a load - to patří do toho podtim

### 3.7.3 rucksack respektive všecko v třídě Global

jen v rychlosti vyjmenovat funkce co to umí

# 4 Diskuse řešení

## 4.1 Existující programy s podobným zaměřením

skopčit ze starýho

## 4.2 Stručná historie projektu

Že to mělo předchozí verzi, která byla kompletně předělána, některý koncepty byli zjednodušený/předělaný (funkce) a některý byly zahozený (přímý hratky s lambda výrazama), zvuky, houpačka, domina, rotace.

## 4.3 snaha spíš než o to udělat finalní hotový projekekt je udělat náčrtek něčeho většího lepšího, takže to neni tak slazený do konzistentě fungujícího stroje s příběhem, ale zas je tam víc konceptů naznačenejch - na programu chci dál pracovat a tak je ted ho chapu jako v počáteční fázi

chápu ho spíš jako konceptuální demo, než jako hotový produkt : chci aby se ho chytla komunita která by ho dále vyvíjel, pro potencialní komunitu myslim neni tak důležité vidět hotovou věc ale spíš vidět nápad, nemusí byt dokonalý a efektivní ale musí v něm být vidět další potenciál. (tohle asi dát do závěru spíš)

## 4.4 Dualita funkce a moucha = dualita lambda kalkul a turingův stroj

## 4.5 proč lisp - protože je to odá na volný přecházení z jednoho do druhýho a protože je to megasimplistický

## 4.6 proč je chytrý že je v XML ekvivalentní vnořený element a atribut: pač když to budou stavět třeba agenti tak to pro ně bude jednodušší když to bude mít takovejdle jednotnej charakter

## 4.7 metoda reinvent wheel je dobrá v tom, že dítě jak se tuní se může čim dál víc hrabat kódem až přejde v ten javovskej kerej zase vysvětluje to celý

operačák -> okýnkovej systém -> atd.

klikačka->kisp,xml->java

## 4.8 metody zjednodušení

### 4.8.1 co se dá ohmatat je lehký

### 4.8.2 dítě se v tom může hrabat a není odrazeno barierami typu tady začína gui a to už je kód jinýho charakteru, snaha o volný přecházení jednoho v druhý

### 4.8.3 gui volně přechází v syntaxi programu, takže si dítě nemusí budovat speciální koncept pro program a přesto může programovat

# 5 Uživatelská a programátorská dokumentace

programátorská je javadoc

uživatelská je vestavěná v podobě průvodce

# 6 Filosofická odbočka k abstraktnímu pojmu rozhraní

Viz novej Havel.

# 7 Závěr

## 7.1 Jak se to povedlo podle planu

-co se povedlo

-co se nepovedlo

## 7.2 Nápady a plány do budoucna

- možnost linkovat přímo .java soubory z xml dokumentů

-důraz na GP.

# Reference

[1] Davison, A.: *Killer Game Programming in Java*, O'Reilly, 2005

[2] Koza, J. R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992

[3] Phys2D - a 2D physics engine based on the work of Erin Catto. *http://www.cokeandcode.com/phys2d/*