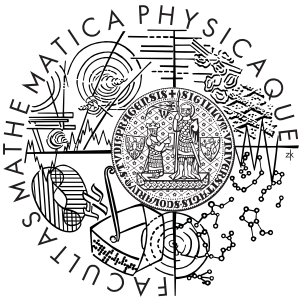
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

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 24. 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: Předmětem této práce je implementovat hru pojatou jako interaktivní fyzikální prostředí, ve kterém vkládáním, přesouváním a propojováním objektů v dvourozměrném hierarchickém prostoru uživatel vytváří virtuální svět. Tento 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. Dále program obsahuje aktivní agenty řízené vnitřním programem, který je poskládán z funkcí.

Klíčová slova: hra, programovací jazyk, virtuální svět

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: The subject of this work is to implement the game conceived as an interactive physical environment in which a user creates a virtual world in hierarchical two-dimensional space by inserting, moving and connecting objects. However, the world, or his parts, also represents the syntax of a program. This is achieved mainly because the game includes different kinds of objects called functions, which occupy the same role, as a function in classic programming languages. The program also includes active agents controlled by an internal program, which is made up of these functions.

Keywords: game, programming language, virtual world

Obsah

1 Úvod **1**

2 Analýza problému **4**

2.1 Výběr programovacího jazyka 4

2.2 Snaha o uchopení obecných principů na úkor popisu principů v reálných jazycích 4

2.3 Teoretické koncepty na úkor počtu předmětů 5

2.4 Poučení z minulé verze programu 5

3 Popis řešení **7**

3.1 Architektura programu 7

3.2 Stručný rozbor fází běhu programu 9

3.3 Provázanost XML reprezentace a objektů jazyka Java 10

3.3.1 Představa abstraktního popisu objektů virtuálního světa . 11

3.3.2 XML reprezentace jako zápis abstraktního popisu objektu virtuálního světa 11

3.3.3 XML reprezentace jako zápis objektu jazyka Java 14

3.4 Některé základní typy objektů 15

3.4.1 Basic 15

3.4.2 Time 16

3.4.3 Frame 17

3.4.4 Příklad minimalistického GUI 17

3.5 Objekty reprezentující data 20

3.5.1 Čísla 20

3.5.2 Symboly 20

3.5.3 Boolovské hodnoty 20

3.5.4 Směry 20

3.5.5 Seznamy 21

3.6 Funkce a Kisp 21

3.6.1 Kisp 23

3.6.2 Kisp a textová konzole . . . 24
3.6.3 Bílé funkce . . . 25
3.6.4 Černé funkce . . . 25
3.6.5 Rekurze . . . 26
3.6.6 Komunikace po internetu . . . 26
3.7 Agenti . . . 27
3.7.1 Vnitřní program . . . 27
3.7.2 Jablíčko, moucha a vosa . . . 28
3.7.3 Paměť mouchy . . . 29
3.7.4 Funkce *incubator* . . . 29
3.7.5 Funkce *read* a *write* . . . 31
3.8 Hráčem řízená postavička . . . 31
3.9 Uživatelská a programátorská dokumentace . . . 32

4 Diskuse řešení **33**
4.1 Existující programy s podobným zaměřením . . . 33
4.1.1 The Incredible Machine . . . 33
4.1.2 Karel . . . 34
4.1.3 AgentSheets . . . 35
4.2 Poznámky k XML reprezentaci objektů . . . 35
4.3 Dualita funkcí a agentů . . . 36
4.4 Souvislost s Genetickým programováním . . . 37

5 Závěr **39**

6 Přílohy **41**
6.1 Obsah CD . . . 41

1 Úvod

Schopnost programovat je považována za věc, která je netriviální a člověk se jí učí dlouho a pracně. Jaké jsou možnosti toho, jak naučit děti programovat?

Jedna z možností je učit je přímo reálně používané jazyky. Ty však trpí nedostatkem, že nejsou navrženy tak, aby byly jednoduše pochopitelné pro nováčky. Proto se nabízí možnost slevit z reálné používanosti a učit děti nějaký méně praktický, ale za to názornější jazyk.

Naprostá většina reálně používaných jazyků má textovou formu - program v nich má formu textu. Tato forma se zdá velice efektivní, otázkou však zůstává zda je to také forma názorná. Nebylo by možné navrhnout jazyk, který by měl místo textového základu nějaký názornější?

Pokus o navržení a implementaci takového jazyka je předmětem této práce. Jako alternativu k textovému popisu programu práce navrhuje popisovat program jakožto vzájemně propojené objekty virtuálního světa, podobající se spíše než slovům stavebnici. Snahou je udělat programovací jazyk, který se navenek tváří jako hra, tak aby si ten, kdo si v něm hraje, nemusel všimnou toho, že se učí programovat.

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 nebo sadou úkolů. Více než hotovou hrou s příběhem se program snaží být nástrojem na tvorbu takovýchto her.

2 Analýza problému

2.1 Výběr programovacího jazyka

Program byl naprogramován v jazyce *Java*. Jeho výhodou je multiplatformnost, rozsáhlost standardních knihoven a také relativně dobrá spolupráce s prohlížeči (to pro možnost udělat program i ve formě appletu). Další podstatnou výhodou je schopnost Javy dynamicky přidávat nahrané třídy za běhu programu a další vlastnosti podobné této, které zatím nebyly v programu použity, ale při výběru jazyka na to byl brán zřetel. Tato vlastnost by pak umožnila ještě těsnější provázání Kutila a Javy, což by bylo dobře využitelné při použití uživatelem vytvořených definic objektů virtuálního světa.

Proberme ještě krátce výhody jiných jazyků, které byly nakonec zavrženy. Zajímavou možností by bylo použít platformu *Adobe Flash*, pro její úzkou provázanost s grafickou stránkou a pro její velice dobrou spolupráci s prohlížeči, nevýhodou je však po mnoha stránkách řádově horší programovací jazyk *ActionScript*.

Pokud by prioritou byla rychlost a efektivita kódu, patrně by bylo vybráno *C++*, to však prioritou nebylo.

Pokud by nezáleželo na mé schopnosti programovat v daném jazyce, vybral bych nejspíš jazyk *Haskell*, pro jeho matematickou krásu a pro zajímavost výzvy, kterou přináši naprogramovat něco takového jako je Kutil v tak striktně funkcionálním jazyce.

2.2 Snaha o uchopení obecných principů na úkor popisu principů v reálných jazycích

Důležitým rozhodnutím při návrhu programu na výuku programování je rozhodnutí, zda dát přednost snaze o vysvětlení principů reálně používaných jazyků, nebo zda raději dát přednost snaze o vysvětlení obecných principů. V této otázce jsem se rozhodl pro plnou podporu snahy o uchopení obecných principů a to z toho důvodu, že považuji tuto schopnost za mnohem cennější, zvlášť v raném

věku.

To má za důsledek, že je hojně používána metoda „rozpouštění hranic“ mezi pojmy, které se často nacházejí na různých hladinách abstrakce. Jedná se o pojmy jako např. *data* versus *program*, *kód programu* versus *výstup programu* nebo *grafické uživatelské rozhraní* versus *to co je díky tomuto rozhraní editováno*. Snaha je, aby se hranice mezi těmito pojmy v programu moc neprojevovala. Aby byl program chápán spíše jako virtuální svět jehož zákonitosti odrážejí některé důležité obecné principy programování.

2.3 Teoretické koncepty na úkor počtu předmětů

Při práci na programu bylo nutno rozhodnout, zda se spíš snažit rozvíjet různé teoretické koncepty, které jdou v duchu toho, že se jedná o programovací jazyk (jako vestavěný jazyk Kisp pro skládání základních funkcí nebo agenta ovládaného vnitřní funkcí) nebo se spíš věnovat různým typům spíše mechanických objektů (jako například ozubená kola, pružiny, kladky, atd.) v duchu hry The Incredible Machines, kterou byla hra zásadně ovlivněna (podrobnější srovnání v části 4.1.1).

Toto rozhodnutí dopadlo tak, že se spíš rozvíjely teoretické koncepty, s tím, že mechanické předměty lze přidat později, zatímco teoretické koncepty by se přidávaly obtížněji, kdyby se s nimi nepočítalo od začátku.

Dalo by se říci, že byla snaha o vyrovnanost pohledů na Kutila jako na prohlížeč virtuálního fyzikálního světa a jako na interpret jazyka.

2.4 Poučení z minulé verze programu

Program Kutil má předchozí verzi, která je na podobném stupni vývoje, přičemž současná verze vznikla kompletním znovu napsáním a s původní verzí nemá prakticky žádný společný kód. Důvodem pro přepsání byla nespokojenost s návrhem architektury, kterou bylo obtížné rozšiřovat požadovaným způsobem.

Základní rozdíl mezi oběma verzemi je v tom, že původní verze byla centralizována kolem konceptu fyzikální simulace a další koncepty byly přidávány

do kontextu této fyzikální simulace. Zatímco současná verze je centralizována kolem hierarchické struktury objektů virtuálního světa a simulace je relativně hodně skrytá. Tato přílišná svázanost s uchopením celé věci v kontextu fyzikální simulace neumožňovala moc dobrou modularitu.

Další nevýhodou z pohledu metody „rozpouštění hranic“ bylo pevné oddělení ovládacích prvků a světa virtuální simulace a nepříliš kvalitní provázání XML reprezentace objektů virtuálního světa s objekty Javy.

Tím, že bylo už dopředu počítáno s některými problémy a tím, že návrh nové architektury byl kvalitnější, byl vývoj nové verze o poznání rychlejší a s lepšími výsledky. Proto považuji rozhodnutí kompletního přepsání programu jako sice komplikované, ale dobré rozhodnutí.

3 Popis řešení

3.1 Architektura programu

V této podkapitole se ve stručnosti podíváme na program z pohledu programátora. Zbytek textu pak bude spíše balancovat na pomezí programátorského a uživatelského pohledu na věc.

Třídy reprezentující objekty virtuálního světa implementují rozhraní `KObject`. Metody tohoto rozhraní pokrývají mnoho činností jako vykreslování, tvoření kopií objektu, převádění objektu do XML, atd. Ale patrně nejpodstatnější metodou tohoto rozhraní je metoda `step()`. Po zavolání této metody objekt udělá jeden krok své činnosti.

Základní typ objektu, který implementuje rozhraní `KObject` a od kterého jsou odvozeny prakticky všechny ostatní třídy implementující toto rozhraní, je třída `Basic`. Základní vlastností objektu třídy `Basic` je, že se skládá z vnitřních objektů. Při zavolání metody `step()` zavolá tuto metodu i u svých vnitřních objektů. Více informací o třídě `Basic` a o několika dalších důležitých třídách odvozených od třídy `Basic` je v části 3.4.

Virtuální svět je tedy tvořen hierarchií `KObjectů`, jejíž dynamiku zajišťuje metoda `step()`.

Jádro programu tvoří tři objekty; instance tříd `Scheduler`, `IdDB` a `Rucksack`.

`Scheduler` je jednoduchý plánovač, který můžeme chápat jako kořen celé hierarchie `KObjectů`, který volá u nejvyšších `KObjectů` z hierarchie metodu `step()`. Toto volání pak kaskádovitě projde celou hierarchií a má za důsledek všechny akce provedené v tomto kroku běhu virtuálního světa. Všechny akce pak jsou projevem jednotlivých `KObjectů`, které mají možnost manipulovat i se strukturou hierarchie. Moc manipulovat se strukturou hierarchie mají díky tomu, že mají jednak přímé reference na své vnitřní objekty i rodičovský objekt, dále pak mají nepřímý přistup k libovolnému jinému objektu skrze jeho *unikátní id*. Tento nepřímý přístup je zprostředkován díky výše jmenované instanci třídy `IdDB`. `IdDB` slouží jako aktuální databáze všech objektů, kteréžto jsou zde přístupné pod

svým unikátním id.

Instance třídy `Rucksack` má důležitou roli při interakci s uživatelem. Umožňuje manipulaci s objekty pomocí myši, implementuje schránku, zajišťuje ovládání běhu simulace (tzn. „play/pause“), zajišťuje pohyb zpět či vpřed v historii editačních změn, zajišťuje interakci s textovou konzolí programu, zajišťuje dialog pro ukládání a otevírání stavu, drží informaci o označených objektech, atd.

GUI programu je realizováno přes speciální třídu `Frame` implementující `KObject` (podrobněji viz 3.4.3). `Frame` je okno zobrazující vnitřek nějakého `KObjectu`. Pokud `Frame` splňuje podmínky popsané v 3.4.3, manifestuje se jako okno programu. Protože je `Frame KObject`, může být prvkem vnitřku právě zobrazovaného `KObjectu`; potom se zobrazí jako vnořené okno a je s ním možno manipulovat podobně, jako s jinými `KObjecty`. Třída `Frame` úzce spolupracuje s instancí třídy `Rucksack`, čímž společně zajišťují interakci s uživatelem.

Podívejme se ještě podrobněji na třídu `Function` implementující `KObject`: Její instance zastávají roli funkcí programovacího jazyka. Rozlišujeme dvě základní varianty funkcí: *bílou* a *černou* (viz 3.6). Černé funkce jsou definovány strukturou svých vnitřních objektů, zatímco bílé jsou definovány výrazem jazyka Kisp (viz 3.6.1). Podívejme se blíže na bílé funkce. Výraz jazyka Kisp je předpisem, podle kterého se vyrobí objekt implementující rozhraní `FunctionImplemetation`. V tomto rozhraní je hlavně důležitá metoda počítající pro pole vstupů pole výstupů dané funkce:

```
public KObject[] compute( KObject[] objects );
```

Výraz jazyka Kisp je složen ze speciálních znaků (tj. `\,'`), mezer, závorek a symbolů atomických funkcí (např. `+`, `head`, `pair`, `if`,...). Převod Kispových výrazů na objekty implementující rozhraní `FunctionImplementation` má na starosti třída `Kisp`. Uvnitř těla statické metody `newAtomImplementation(...)` je možno přiřadit nový symbol ke konkrétní `FunctionImplementation`.

Pro různé typy funkcí [1] existují různé abstraktní třídy implementující `FunctionImplementation` (např. `UnarImplementation`, `BinarImplementation`,

[1] Typem funkce zde chápeme počet vstupů a výstupů.

`UnarBinarImplementation`, atd.).

Předveďme příklad implementace jednoduché binární funkce; jedná se o implementaci sčítání:

```
class Plus extends BinarImplementation {
   public Plus(){ super("+"); }
   public KObject compute( KObject o1 , KObject o2 ) {
      if( o1 instanceof Num && o2 instanceof Num ){
         num1.set( ((Num) o1).get() + ((Num) o2).get()  );
      }
      return o1;
   }
}
```

Ještě se ve stručnosti podívejme na to, jak spolu jednotlivé funkce komunikují. Výstupy funkcí můžeme napojovat na vstupy jiných funkcí. Mechanizmus propojení funkcí je zajištěn tím, že funkce implementují rozhraní `Imputable`, jehož nejdůležitější metodou je metoda:

```
public void  handleInput( KObject input , int port );
```

Tato metoda slouží k dosazení vstupního `KObjectu` na konkrétní vstup funkce. Pokud má funkce všechny potřebné vstupy, vypočítá na základě své `FunctionImplementation` výstupy, které pak pošle dalším funkcím, se kterými je propojená, znovu voláním jejich metody `handleInput`. To má za důsledek kaskádu výpočtů končící typicky „vypadnutím“ výsledku.

3.2 Stručný rozbor fází běhu programu

Nyní 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.

3.3 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.

3.3.1 Představa abstraktního popisu objektů virtuálního světa

Nyní popíšeme abstrakci 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, objekt)*.

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. objekt). Takový objekt chápeme složený ze *součástek*, kde každá součástka je jeden objekt. A každá součástka má svůj klíč udávající, jakou roli v objektu má tato součástka. Více součástek může mít stejnou roli.

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

Jako příklad uveďme následující objekt:

```
[ ( key1, value1 ),
  ( key2, [ ( key3, []      ) ] ),
  ( key2, [ ( key4, value2  ) ] )
]
```

3.3.2 XML reprezentace jako zápis abstraktního popisu objektu virtuálního světa

Podívejme se na to, jak je XML reprezentace používaná v programu Kutil pro zápis 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. Pokud je více objektů pod stejným klíčem, zapíšeme je za sebou v rámci tohoto společného klíče:

4.
```
<key>
  <object>(...)</object>
  ...
  <object>(...)</object>
</key>
```

Pro příklad z 3.3.1 máme tedy následující zápis:

```
<object>
  <key1>value1</key1>
  <key2>
    <object><key3><object/></key3></object>
    <object><key4>value2</key4></object>
  </key2>
</object>
```

Využíváme dále toho, že v XML je pojem atribut. Využijeme toho pokud nějaký objekt (v roli hodnota ve dvojici klíč - hodnota) je textový řetězec. V našem příkladu to splňuje například 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“, o kterém budeme mluvit, je svázán s důležitým klíčem `inside`. Pokud je nějaký element `object` přímo v elementu `object`, znamená to, že je implicitně pod klíčem `inside`.

Uvažme následující dva zápisy, podle tohoto pravidla jsou ekvivalentní:

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

Druhý ekvivalentní zápis:

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

3.3.3 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 platný 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 (*seznam objektů* je seznam všech objektů, které jsou pod tímto klíčem v daném objektu). 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 správu 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 slouží 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 objektů v současném stavu programu využívá pouze klíče mající hodnotu textový řetězec, nebo dříve zmíněný důležitý klíč *inside*. 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* chápaný v kontextu GUI.

To, že většina objektů využívá pouze textové klíče nebo klíč *inside* neznamená to, že by koncept klíčů byl špatně navržený. Znamená to, že objekty implementované v současné fázi programu jsou dostatečně jednoduché a tak si s tím vystačí. Tuto typickou jednoduchou strukturu porušují například objekty v roli agentů, které využívají možnosti složitější struktury pro implementaci paměti oddělené od programu.

3.4 Některé základní typy objektů

Ve stručnosti se podíváme na některé typy objektů.

3.4.1 Basic

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

Na příkladě si ukážeme, jaké může mít vlastnosti:

```
<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 × 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 vnitřní fyzické objekty svět fyzikální simulace, 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.

3.4.2 Time

Výše 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 objektu opakovalo donekonečna) můžeme to specifikovat položkou `iterations`.

Ukážeme 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.

3.4.3 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.

3.4.4 Příklad minimalistického GUI

Na závěr této podkapitoly uvedeme příklad minimalistického GUI, které demonstruje, jak se používají výše zmíněné konstrukty. Následující XML kód je podoba celého souboru, který můžeme eventuálně 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`).

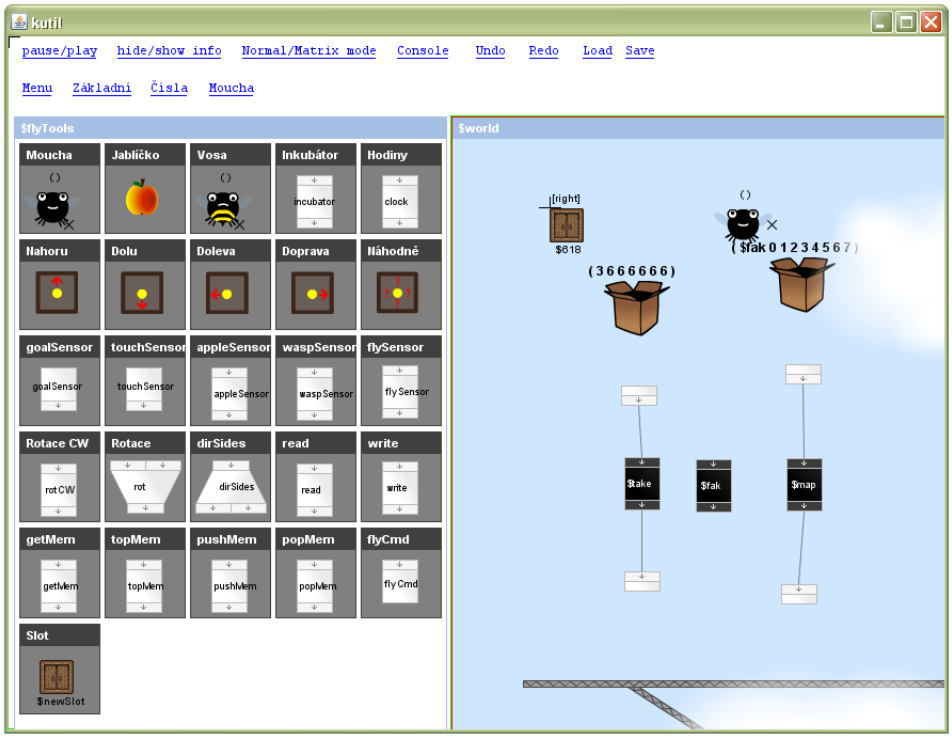
Další dva objekty jsou oba typu `frame`. První s id `$2` má za `target` implicitně sebe. Druhý má za `target` také objekt s id `$2`, navíc má položku `showXML` nastavenou na `true`, což má za důsledek, že ukazuje svůj `target` ve formě XML reprezentace.

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 vrátí do stavu před spuštěním simulace.

Pro srovnání ještě uvedeme obrázek toho, jak může vypadat složitější uživatelské rozhraní:



3.5 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.

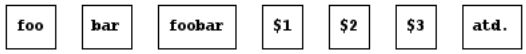
Nyní se na každý typ z výše zmíněných objektů velice stručně podíváme.

3.5.1 Čísla

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

3.5.2 Symboly

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



3.5.3 Boolovské hodnoty

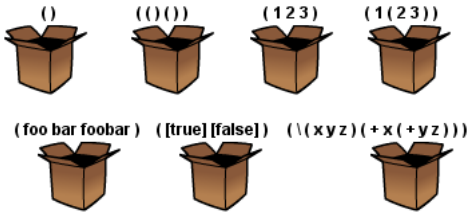
Boolovské hodnoty jsou reprezentovány bílým, respektive černým, míčkem:

3.5.4 Směry

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

3.5.5 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 3.6).

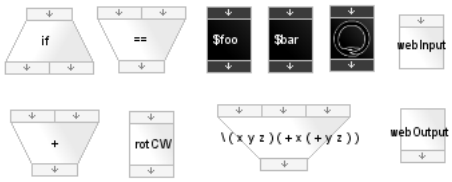


První seznam na obrázku je prázdný. Druhý obsahuje dva prázdné seznamy. Třetí obsahuje čísla 1, 2 a 3. Čtvrtý obsahuje číslo 1 a seznam obsahující čísla 2 a 3. Pátý obsahuje symboly *foo*, *bar* a *foobar*. Šestý obsahuje boolovské hodnoty *true* a *false*. A sedmý obsahuje symbol \, seznam obsahující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

3.6 Funkce a Kisp

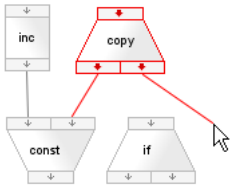
Funkce je objekt který má na svém vršku několik vstupů a na svém spodku několik výstupů.

Následující obrázek ukazuje několik příkladů různých funkcí:



Každý 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.

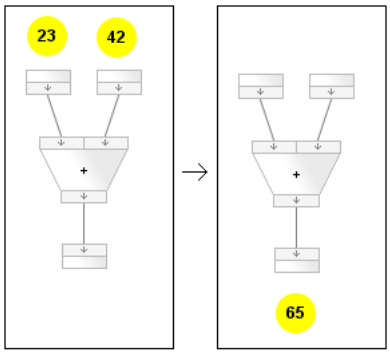
Na následujícím obrázku je tento princip naznačen:



Funkce jsou implicitně nefyzické objekty (nejsou součástí simulace, tzn. neinteragují 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 bílých funkcí je určeno

textově, je zapsáno pomocí jednoduchého jazyka Kisp. Naproti tomu chování černých funkcí je určeno strukturou jejich vnitřních objektů.

3.6.1 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.

Kisp umožňuje skládání základních funkcí (s omezením, že musí mít jeden výstup) a uživatelsky definovaných černých funkcí do složitějších funkcí.

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 proměnných 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 zapisuje 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 provázán 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, že 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. Do této funkce je dosazena hodnota `42`. Jako *argumenty-funkce* zde máme jednodušší možnost jediného symbolu. V tomto případě výpočet funkce probíhá 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 komplikovanější 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) )
```

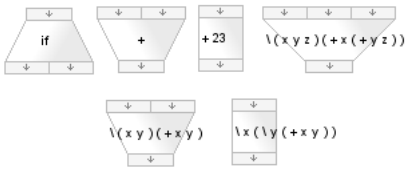
Neboli odstraní se první prvek ze seznamu *argumenty-funkce* a dále se pokračuje analogicky jako v předchozím příkladě.

3.6.2 Kisp a textová konzole

Do textové konzole programu je možno napsat výraz Kispu. Výsledek vzniklý vyhodnocením výrazu se projeví tak, že se objeví jako vkládaný objekt na kurzoru myši. Pokud se výraz vyhodnotí jako zápis funkce, je vkládaným objektem příslušný objekt funkce. Jinak je výsledným objektem nějaký objekt reprezentující data (viz 3.5).

3.6.3 Bílé funkce

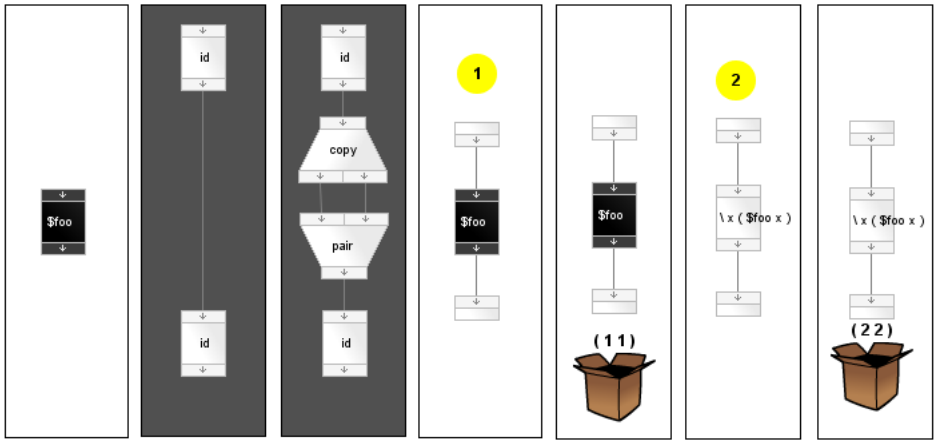
Bílá funkce je definována zápisem funkce v Kispu. Následující obrázek ukazuje několik příkladů:



Zajímavý je rozdíl mezi předposlední a poslední funkcí. Zatímco předposlední má dva vstupy, poslední má jenom jeden. Předposlední své vstupy sečte. Naproti tomu poslední vrátí pro vstup N jako výstup funkci sčítající svůj vstup a N.

3.6.4 Černé funkce

Chování černé funkce je dáno její vnitřní strukturou, konkrétně zapojením funkcí nalézajících se v jejím vnitřku. Následující obrázek použijeme pro objasnění fungování černých funkcí:



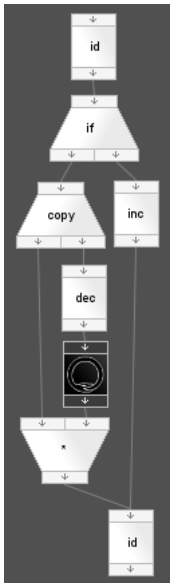
První okénko ukazuje vzhled černé funkce. U černé funkce je zvlášť důležité její id, které vystupuje jako jméno dané funkce, kterým se můžeme na tuto funkci odkazovat jinými funkcemi. Druhé okénko ukazuje, jak vypadá vnitřek

nové černé funkce. Dvě bílé funkce[2] zde reprezentují vstup a výstup, mezi nimi se bude nacházet program černé funkce. Třetí okénko ukazuje příklad triviálního programu, který udělá kopii vstupního objektu a následně vrátí dvouprvkový seznam s prvky původní objekt a jeho kopie. Čtvrté a páté okénko demonstruje přímé zavolání funkce. Šesté a sedmé okénko ukazuje příklad nepřímého zavolání funkce pomocí bílé funkce, jejíž definující Kispový výraz obsahuje id černé funkce.

3.6.5 Rekurze

Speciálním typem objektu je rekurze. Ta umožňuje odkazovat se na černou funkci uvnitř jí samotné.

Následující obrázek demonstruje použití rekurze uvnitř funkce počítající faktoriál:



3.6.6 Komunikace po internetu

Pro komunikaci po internetu existují dvě speciální funkce. Funkce *webInput* má jeden vstup a žádný výstup, slouží k odeslání objektu na server (jedná se o mini-

[2]U těchto dvou bílých funkcí nápis *id* značí, že jde o identitu.

malistický server napsaný v jazyku PHP, v současné chvíli se nachází na doméně `kutil.php5.cz`), kde je do databáze uložena jeho XML reprezentace. Společně s XML reprezentací se na server odesílá i identifikační port, což je textový řetězec fungující jako klíč, pod kterým je daný objekt uložen.

Funkce *webOutput*, která má jeden výstup a žádný vstup, pokud je použita ve virtuálním světe pravidelně kontroluje, zda se na severu pod sledovaným identifikačním portem neobjevil nějaký objekt. Pokud ano, ze serveru je odstraněn a funkce ho vrátí jako svůj výstup.

Tato funkcionalita je zatím spíše v experimentální fázi, takže komunikace například probíhá stále na stejném identifikačním portu.

3.7 Agenti

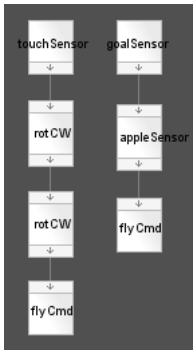
Dalším důležitým prvkem virtuálního světa jsou *agenti*, samostatně aktivní objekty řízené svým vnitřním programem na základě vstupů přicházejících ze speciálních funkcí nazývaných *senzory*. Prostřednictvím vedlejších efektů funkcí nazývaných *efektory* agent reaguje na vstupy ze senzorů různými akcemi.

Program nyní obsahuje dva typy agentů: *mouchu* a *vosu*. Mouchou se zde budeme dále zabývat, vosa je pak speciálním případem mouchy.

Moucha je volný fyzický objekt, který není ovlivněn gravitací. Každá moucha má svůj cíl, což je pozice ke které tato moucha přímou cestou letí. Souřadnice cíle nejsou ve stejných jednotkách jako všeobecně používané souřadnice objektů, místo toho se uvažuje čtvercová mřížka, s přibližně stejnou velikostí políčka, jako je velikost mouchy. Tedy moucha je součástí fyzikální simulace, ale její pohyb je řízen změnou pozice cíle uvažovaného v rámci této čtvercové mřížky.

3.7.1 Vnitřní program

Podobně jako černá funkce má moucha ve svém vnitřku program sestávající z navzájem propojených funkcí. Senzory jsou funkce sloužící pro mouchu jako zdroj informací o vnějším světě. Podívejme se na příklad jednoduchého vnitřního programu mouchy. Jako příklad nám pomůže následující obrázek:



Na obrázku můžeme vidět dvě posloupnosti funkcí. Každá z nich začíná senzorem. U mouchy se předpokládá, že data s kterými vnitřně pracují funkce, které tvoří její program, jsou *směry* (viz 3.5.4). Nejprve se podívejme na senzor *touchSensor*. Ve chvíli, kdy se moucha dotkne jiného fyzického předmětu, funkce *touchSensor* vrátí jako výstup směr, ve kterém se dotkla daného předmětu. Funkce *rotCW* rotuje směr ve směru hodinových ručiček. A funkce *flyCmd* posune cíl o jedno políčko ve směru, který jí byl předán jako vstup. To má za následek, že první posloupnost reaguje na kolizi s objektem posunem cíle o jedno políčko ve směru pryč od kolize. Druhá posloupnost začíná funkcí *goalSensor*. Ta se chová tak, že při příchodu mouchy do cíle vrátí jako svůj výstup směr, ve kterém bylo naposledy posunuta pozice cíle. Následuje funkce *appleSensor*, která pro libovolný vstup vrací směr, ve kterém se vzhledem k mouše nachází nejbližší objekt *jablíčko*[3]. Čili tato posloupnost má za důsledek, že moucha při příchodu do cíle změní pozici cíle na pozici, která je blíž nejbližšímu jablíčku.

3.7.2 Jablíčko, moucha a vosa

Zmiňovaný objekt jablíčko je fyzický objekt s fixní pozicí, sloužící k „rozmnožování“ much. Ve chvíli, kdy se dotkne moucha jablíčka, jablíčko zmizí a místo něj se objeví kopie této mouchy. Další typ agenta *vosa*, je v principu svého fungování totožná s mouchou (jen je o něco pomalejší než moucha). Vosa slouží jako protiváha jablíčkům; ve chvíli, kdy dojde ke kolizi mouchy a vosy, moucha je

[3]Pokud takové jablíčko neexistuje, vrátí *appleSensor* pátý (náhodný) směr.

smazána. Dále pak na rozdíl od mouchy vosa neumí „sníst“ jablíčko.

Pro to, aby mohla moucha reagovat na přítomnost vos a aby mohla vosa reagovat na přítomnost much, existují dva senzory *flySensor* a *waspSensor*. Ty fungují analogickým způsobem jako *appleSensor*.

3.7.3 Paměť mouchy

Moucha má dále paměť v podobě seznamu, se kterou může operovat. K manipulaci s touto pamětí slouží například funkce *pushMem*, *popMem* a *topMem*, které k tomuto seznamu přistupují, jako by se jednalo o zásobník. Obsah této paměti není ve vnitřku mouchy (ve vnitřku je pouze program mouchy); jak jsme ukázali výše (viz 3.3.3) vnitřek mouchy se skrývá pod klíčem *inside*, obsah paměti je pod klíčem *mem*. Kispová podoba obsahu paměti je zobrazena nad mouchou (podobně jako ji zobrazuje seznam).

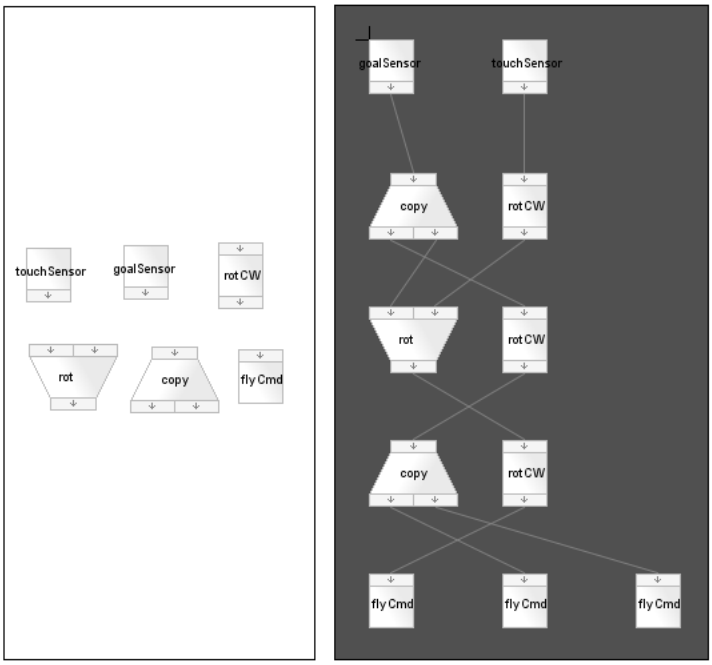
Podoba jablíčka, mouchy a vosy je na následujícím obrázku:

3.7.4 Funkce *incubator*

Mouchy mohou vznikat dvojím způsobem: buďto je vytvoří uživatel ručně, nebo můžeme použít speciální funkce *incubator*, která slouží k vytváření much s náhodným vnitřním programem. Ve stručnosti se podíváme na to, jak funkce *incubator* takový program vytváří.

Funkce *incubator* je bílá funkce s parametry, to znamená, že krom vstupní hodnoty její chování ovlivňuje obsah jejího vnitřku. Jejími parametry jsou funkce, které chceme použít jako stavební díly vnitřního programu nové mouchy. Tyto funkce jednoduše vložíme do vnitřku funkce *incubator*. Na libovolný vstup pak *incubator* reaguje tak, že z těchto funkcí sestaví vnitřní program mouchy.

Tento program má vrstevnatou strukturu, podobně jako neuronové sítě. První vrstva sestává pouze z funkcí s nula vstupy, poslední vrstva sestává z funkcí s nula výstupy a ostatní vrstvy sestávají z funkcí s alespoň jedním vstupem a jedním výstupem. Algoritmus pro stavbu tohoto programu nejprve náhodně určí počet vrstev (např. z rozmezí 3 až 6) a pak pro první vrstvu náhodně určí počet funkcí (např. z rozmezí 1 až 6), které zde budou. Pak náhodně vybírá z dostupných funkcí ty, které na dané místo vloží. Pro každou další vrstvu platí, že přidává funkce do vrstvy, dokud celkový počet vstupů v této nové vrstvě nepřekračuje počet výstupů v předchozí vrstvě. Vstupy a výstupy sousedních vrstev jsou pak náhodně propojeny. Následující obrázek ukazuje názorný příklad:



První okénko ukazuje příklad vnitřku funkce *incubator*, druhé okénko ukazuje příklad vzniklého programu uvnitř mouchy.

3.7.5 Funkce *read* a *write*

Krom vlastního pohybu může moucha ovlivňovat své prostředí ještě jedním způsobem: Může na políčko, na kterém se právě nachází, zapsat nebo načíst libovolná data, pomocí funkcí *read* a *write*. Tato data jsou uložena v objektu *slot*, který se nachází na místě daného políčka (jedná se o nefyzický předmět, podobně jako funkce). Pokud moucha zapisuje na místo, kde není *slot*, automaticky se vytvoří nový.

To nám dává nový pohled na mouchu jakožto na jakési „rozšíření“ základního pojetí Turingova stroje. Moucha zde představuje jakousi hlavu, která se pohybuje místo na jednorozměrné pásce na pásce dvojrozměrné. Vnitřní program mouchy pak lze chápat jako analogii přechodové funkce, neboť podobně jako přechodová funkce tento program ovlivňuje jak pohyb, tak zápis.

3.8 Hráčem řízená postavička

Typickým prvkem mnoha her je postavička ovládaná hráčem přímo z klávesnice. Tento prvek nechybí ani ve hře Kutil, zde je reprezentován objektem *budha*.

Budha je fyzický volný objekt. Pomocí klávesnice je možno ovládat jeho pohyby. Navíc je možno s budhou provádět nejrůznější manipulace s objekty. Je schopný do svého vnitřku „nasát“ objekt, kterého se právě dotýká. Budhu tedy můžeme chápat jako klávesnicí ovládaný seznam.

Přesněji má budha následující schopnosti:

- Umí přesunou objekt, kterého se právě dotýká na první pozici ve svém vnitřku.
- Umí přesunout objekt, kterého se právě dotýká na první pozici ve vnitřku objektu, který má ve svém vnitřku na první pozici. Čili jakási hlubší varianta první schopnosti, která umožňuje měnit strukturu jiných objektů, nejen svoji vnitřní.
- Umí inverzní operaci k prvním dvěma zmíněným operacím, čili objekt na

první pozici svého vnitřku (respektive na první pozici ve vnitřku prvního prvku svého vnitřku) umí dostat ven, na pozici těsně pod sebou.

- Umí rotovat obsah svého vnitřku doleva a doprava (rotací máme na mysli rotaci seznamu: první prvek se stane druhý, druhý třetím,..., poslední prvním).
- Umí naráz vrátit všechny vnitřní objekty na místa, kde je sebral.

Pokud je vložen do virtuálního světa víc než jeden budha, je možno přepínat právě aktivního budhu, nebo případně ovládat všechny budhy naráz.

Jak ukazuje následující obrázek, obsah vnitřku budhy je zobrazován podobně jako u seznamu.

3.9 Uživatelská a programátorská dokumentace

Uživatelská dokumentace je součástí programu, je ve formě interaktivního návodu, který provází uživatele jednotlivými vlastnostmi programu. Návod na spuštění programu je v dodatku zabývajícím se obsahem přiloženého CD.

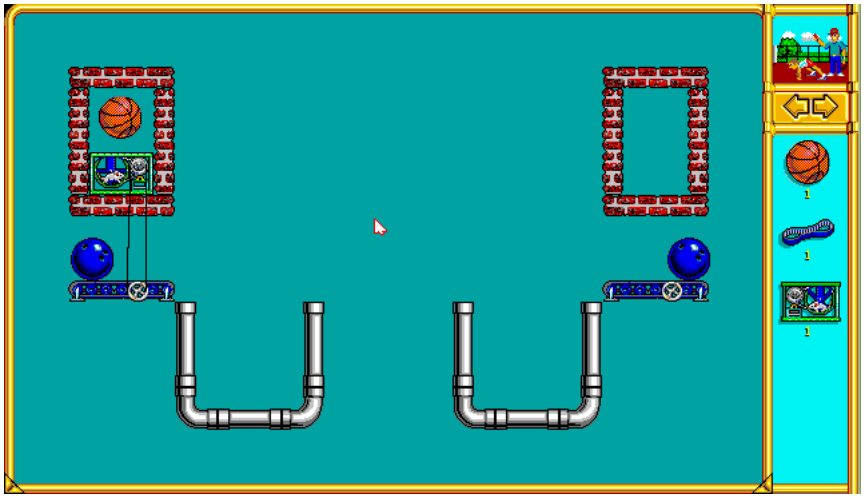
Podrobnější programátorská dokumentace je také součástí CD.

4 Diskuse řešení

4.1 Existující programy s podobným zaměřením

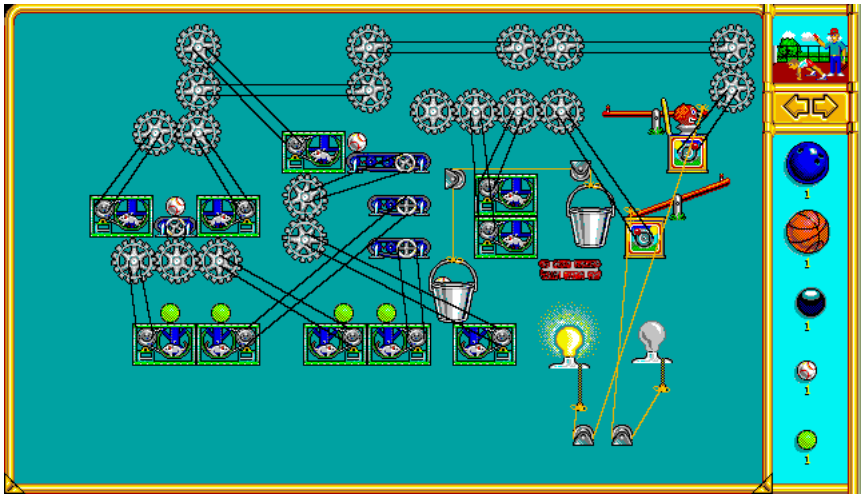
4.1.1 The Incredible Machine

Počáteční inspirací programu Kutil byla hra *The Incredible Machine* (zkráceně TIM) z roku 1993. V této hře hráč řeší sérii problémů, kde každý problém sestává z nekompletního stroje a několika součástek určených ke vložení do tohoto stroje tak, aby stroj vykonal daný požadavek, typicky nějakou jednoduchou mechanickou operaci. Následující obrázek ukazuje, jak vypadá příklad jednoduchého problému:



Vedle výše zmíněného módu hry, kde hráč postupně řeší sérii problémů má hra ještě mód, ve kterém si hráč může stavět stroje neomezeně dle svého uvážení. Následující obrázek ukazuje příklad takového stroje, který jsem vytvořil. Implementuje binární dvoubitovou sčítačku[4] :

[4]Přítomnost zelených míčků na myší kleci představuje jedničku, výstup je v podobě rozsvícené nebo zhasnuté lampičky. Zde jsou přítomny všechny čtyři míčky, tedy sčítačka počítá 11 + 11 = (1)10. Zobrazují se však pouze první dvě místa, proto ukazuje 10.



Je vidět, že na TIM se můžeme dívat jako na zvláštní programovací jazyk, kde stroje představují programy. Zajímavou součástkou, kterou však hra TIM neobsahuje je krabička, do které by bylo možné zavřít již funkční stroj. Stroj takto zavřený v krabičce by bylo možno následně využít v jiném stroji jako součástku.

Původní myšlenka pro hru Kutil byla právě vytvořit napodobeninu hry TIM, s touto součástkou. Postupem času, jak práce na programu Kutil pokračovala a byly objevovány další možnosti, jak se k programu stavět, se program vzdaloval od této původní představy. Program Kutil tak jak vypadá v současné chvíli můžeme v tomto kontextu považovat za přechod mezi hrou TIM a klasickým programovacím jazykem. Kutil je navržen tak, aby byl rozšiřitelný a tak možnost obsahovat objekty podobné součástkám z hry TIM není znemožněna, spíš se v současné chvíli zaměřuje na obecnější konstrukty.

4.1.2 Karel

Karel je výukový programovací jazyk mimo jiné používaný na Stanfordově univerzitě. Program napsaný v jazyce Karel slouží k ovládání chování robota pohybujícího se po čtvercové mřížce. K tomu jsou používány příkazy *turnleft* (Karel se otočí o 90 ° doprava), *putbeeper* (Karel položí bzučák na aktuální políčko),

pickbeeper (Karel zvedne bzučák) a *turnoff* (Karel se vypne, čímž končí běh programu).

Základní rozdíl mezi Karlem a Kutilem je, že v Kutilovi je program tvořen objekty na stejné rovině abstrakce jako je agent sám (jak agent, tak funkce tvořící program jsou předměty, které vkládáme do virtuálního světa). Zatímco program v Karlovi má jinou formu, než robot a jeho prostředí: Program je text.

4.1.3 AgentSheets

Příkladem komerčně úspěšného programu pro výuku programování je program AgentSheets. Je designován k tomu, aby učil děti vytvářet počítačové hry. Začíná jednoduchými hrami s cílem „dostat žabáka přes cestu“ a komplikovanost těchto her se postupně zvyšuje až dosahuje k hrám typu *The Sims* s využitím prvků umělé inteligence.

Základní rozdíl mezi AgentSheets a Kutilem je v tom, že AgentSheets je mnohonásobně rozsáhlejší a profesionálně propracovanější projekt s dlouhou tradicí. Další rozdíl je podobný jako byl u programovacího jazyku Karel, totiž v tom, že odděluje program od výsledného produktu. Interakce probíhá skrze různá okna, kde se nastavují nejrůznější vlastnosti chování agentů a kde se na principu „drag-and-drop“ skládá program dohromady. Kutil se naproti tomu snaží být systém více podobný virtuálnímu samočinnému světu, kde nechceme pevně rozdělovat systém na program a výstup.

4.2 Poznámky k XML reprezentaci objektů

XML reprezentaci objektů, tak jak byla popsána v části 3.3, bychom mohli vytknout na první pohled redundantní používání slova *object* jakožto jména elementu. Mohlo by se zdát, že přirozenější reprezentací by bylo použít místo slova *object* typ daného objektu a zajistit, aby jméno objektu nemohlo být stejné jako jméno klíče (aby bylo jasně poznat co je klíč a co je vnitřní objekt). To proč se o tomto částečně ne tolik podstatném problému zmiňujeme takto výslovně je to, že důležitý důvod pro použití slova *object* jakožto jména elementu se skrývá v za-

tím neimplementované součásti programu, která je však do budoucna plánována jako rozšíření.

Součástí XML reprezentace by v budoucnu měl být konstrukt nazývaný *macro*. Což by byl jakýsi preprocesor umožňující v rámci XML reprezentace vytvářet objekty i jinou cestou, než jejich explicitním popsáním. Uvedu hypotetický příklad makra:

```
<macro type="kisp">
   ( \ ( x y ) (+ x y ) )
</macro>
```

Na základě tohoto makra by pak vznikl objekt reprezentující funkci popsanou v Kispu.

Pokud bychom taková makra používali často, pro lidskou čitelnost je myslím dobré velice jasně odlišovat objekty a makra. A jejich typ chápat jako až druhořadou vlastnost.

Takováto makra by pak měla různá další zajímavá použití. Každý typ makra je de facto překladačem z nějakého jazyka do objektové reprezentace v rámci Kutila. Představme si například, že máme nějaký přirozený zápis Turingova stroje. Potom by mohlo existovat makro, které převádí tento přirozený zápis do reprezentace v Kutilovi, konkrétně by se k tomu hodila moucha (tzn. vnitřní program mouchy by odpovídal její přechodové funkci a počáteční hodnota pásky by se manifestovala řadou *slotů* s patřičným obsahem).

4.3 Dualita funkcí a agentů

Dva základní prvky virtuálního světa jsou funkce a agenti. Funkce můžeme chápat jako pasivní; čekající na vstup na základě kterého něco provedou. Agenti jsou naproti tomu aktivní; na akci nečekají tolik zvenku, jde spíše zevnitř a ovlivňují tak své okolí.

Tuto dualitu můžeme v informatice vidět na mnoha místech. Příkladem jsou například dvě vzájemně ekvivalentní uchopení programu: lambda kalkul a Turingův stroj. Toto uchopení vzniklo takřka na začátku informatiky a dualita v

něm obsažená přetrvává dále, dnes dobře patrná při rozlišování mezi deklarativními a imperativními jazyky. Deklarativní jazyky v tomto ohledu chápeme jako následníky tradice lambda kalkulu, ve kterých se i samotná konstrukce lambda výrazu častokrát vrací do syntaxe těchto jazyků. Naproti tomu imperativní jazyky uchopující program jako sadu příkazů stroji je možno chápat jako následníka myšlenky Turingova stroje.

V duchu takto chápané tradice pak funkce v Kutilovy chápeme jako deklarativní a agenty jako imperativní konstrukty.

V nynějším stavu programu je vzájemný vztah funkcí a agentů převážně charakterizován tím, že vnitřní program agentů je tvořen funkcemi, neboli že funkce definují agenty. Dále se podíváme na opačný vztah, který však ještě v současné verzi programu není implementován. Opačným vhodným vztahem se mi zdá schopnost agentů měnit strukturu zapojení funkcí. V současné fázi agenti mají prostředky pro vkládání dat na jednotlivá políčka mřížky po které se pohybují. Nápad který se zde naznačuje, je schopnost agentů vkládat do svého okolí funkce a měnit jejich vzájemné propojení.

Tím bychom dosáhli nové možnosti hierarchického tvoření agentů: Podobně jako se funkce mohou skládat ze svých vnitřních funkcí, tak by ve vnitřku agenta mohly být vnitřní agenti modifikující vnitřní program definující chování tohoto agenta.

4.4 Souvislost s Genetickým programováním

Genetické programování (zkráceně GP) [2] je paradigma v oblasti evolučních algoritmů, které pracuje s populací v níž jsou jedinci programy (typicky stromové struktury). Stručně řečeno funguje následujícím způsobem: Programy v populaci řeší nějaký konkrétní problém a my jsme schopni ohodnotit toto řešení pomocí *fitness funkce*. Na základě tohoto hodnocení pak probíhá výběr jedinců pro další generaci, tito jedinci jsou buď přímo vloženi do další generace, případně zkříženi s jiným jedincem nebo zmutováni. Tak dostáváme novou populaci a takto postupujeme stále dokola, dokud nedostaneme program řešící náš problém dostatečně

dobře. Nejdůležitějšími parametry při aplikaci GP jsou volba fitness funkce a funkcí, ze kterých se následně programy sestavují.

Program Kutil se snaží nabídnout základní konstrukty pro tvorbu systémů inspirovaných nebo přímo implementujících principy GP. Sada těchto konstruktů je zatím v počáteční fázi vývoje. Zkusme nyní nahlédnout motivaci ekologie mouchy, jablka a vosy. Moucha představuje jedince populace, zatímco vosa a jablko společně s prostředím ve kterém se mouchy nalézají tvoří implicitní fitness funkci (ve smyslu že je implikována prostředím). Moucha se musí snažit vyhnout o něco pomalejší vose a navíc čím víc jablíček sní, tím víckrát se rozmnoží a její kód má větší šanci na přežití. Například si můžeme představit bludiště, ve kterém by mouchy hledaly jablíčka a snažily se vyhnout střetům s vosami. Vhodným využitím funkce *incubator* (viz 3.7) a nějakých dalších zatím neimplementovaných funkcí pro křížení a mutaci much by bylo možné udělat toto „bludiště“ tak, aby fungovalo podle výše popsaných principů GP a přitom nepoužívalo žádnou explicitní fitness funkci.

Myslím že studium implicitních fitness funkcí v tomto kontextu je důležité, protože umožňuje pochopit obecnější principy spojené s fenoménem vytváření nových řešení a samoorganizací, tak jak k tomu dochází v přírodě, kde je fitness funkce implicitní.

5 Závěr

Cílem této práce bylo vytvoření programu přibližujícího dětem svět programování skrze intuitivnost fyzikálního světa. Jak práce na programu pokračovala, čím dál víc se zdálo, že takovýto systém může být nápomocný nejen dětem, ale i mě samotnému. Lidská přirozenost vychází z přirozenosti opice, která si potřebuje věci pořádně „osahat“, aby pochopila hlouběji principy, kterými funguje. Prvním prostředkem pro usnadnění osahatelnosti je to, že kontext spojující celý program do jednotného celku je fyzikální simulace, což je pro lidi intuitivní rozhraní, díky tomu, že žijeme ve fyzikální realitě. Jako další prostředek usnadnění této osahatelnosti byla zvolena metoda „rozpouštění hranic“ mezi pojmy, které se často nacházejí na různých hladinách abstrakce.

Těmito pojmy jsou pojmy jako *data* versus *program*, *kód programu* versus *výstup programu* nebo *grafické uživatelské rozhraní* versus *to co je díky tomuto rozhraní editováno*. Minimalistický jazyk Kisp byl inspirován Lispem právě proto, že v něm je mezi daty a programem tak malý rozdíl: Zápis programu a zápis dat, s kterými program zachází v sebe volně přecházejí. Vždyť i samotný vynález počítač čerpá svou moc z toho, že má program, který určuje jeho chování uložený jako data, se kterými manipuluje.

Program je psán tak, aby jednotlivé hladiny abstrakce na sebe volně navazovaly. Aby uživatel, pokud má zájem, mohl začínaje v grafickém rozhraní pokračovat dále do XML reprezentace, případně do Kispové reprezentace, nebo dokonce do kódu napsaného v Javě.

V matematice si často vypomáháme geometrickými znázorněními abstraktních pojmů. Mnohdy tyto pojmy vycházejí přímo z inspirace nějakým geometrickým principem. A díky tomu tento princip pak dále přenášíme do abstraktnějších rovin. Jeden z pohledů na program Kutil je právě jako na takovouto pomůcku pro tvorbu intuicí, která by pomohla v tom, že uvidíme nějakou souvislost mezi dvěma problémy tím, že jejich více či méně metaforické uchopení bude souviset v Kutilovi, díky tomu že se snaží věci implementováno „fyzicky“ a „osahatelně“.

Kdyby to takhle opravdu fungovalo, tak by nám to dávalo spojitý přechod

mezi doménou učících se dětí a bádajících dospělých.

Jednoduchost s jakou jde docílit Turingovské kompletnosti jazyka nám dává zajímavou možnost snažit se vytvořit intuitivní nástroj, který by však neztrácel na své teoretické síle.

Program Kutil je hnán snahou o kompromis mezi konceptuálním náčrtkem sjednocení mnoha různých nápadů do jednoho a funkční hrou. Díky tomu má i řadu neduhů. Některé koncepty jsou zatím jen naznačeny a nejsou dodělány do plně uspokojivé kvality. Nebyl kladen maximální důraz na efektivitu a na stabilitu, proto se při složitějších situacích může zdát pomalý a někdy je nestabilní. Nebylo cílem udělat kompletně hotovou věc, ale spíš uplést konzistentní košík nápadů, ve kterém je vidět další potenciál.

Do budoucna vidím tři hlavní priority dalšího vývoje:

- Práce na efektivitě, stabilitě a eleganci kódu.
- Vytvoření rozhraní umožňujícího dynamicky propojovat program s uživatelem napsaným kódem v Javě tak, aby mohly vznikat uživatelsky definované funkce, nové typy objektů a makra.
- Obohatit program o nástroje umožňující experimenty v doméně Genetického programování.

6 Přílohy

6.1 Obsah CD

Na přiloženém CD se nachází následující soubory a adresáře:

`kutil.jar` Spustitelný soubor programu. V systému Windows je možno ho spustit přímo, na Unixovém systému ho můžeme spustit příkazem z konzole:

```
java -jar kutil.jar
```

`bakalarska-prace.pdf` Elektronická verze této práce ve formátu PDF.

`readme.txt` Podrobnější informace o obsahu CD, případně aktuálnější než ty uvedené v tomto seznamu.

`src` Adresář obsahující zdrojové kódy.

`doc` Adresář obsahující programátorskou dokumentaci.

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/*