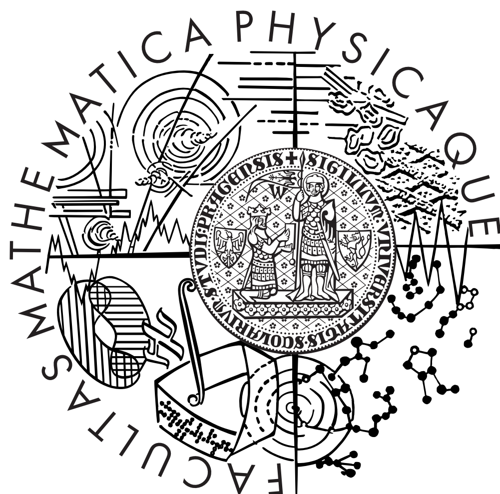
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Petr Štěpán

Prostředí pro animaci algoritmů — překladač a práce s daty

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Rudolf Kryl
Studijní program: Informatika, Programování

2006

Na tomto místě bych rád poděkoval vedoucímu bakalářské práce RNDr. Rudolfu Krylovi za podnětné připomínky a rady při vývoji AAnimu i při vypracování této bakalářské práce.

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

V Praze dne 29. května 2006 Petr Štěpán

Obsah

1 Představení programu AAnim **6**
- 1.1 Zkoumání již existujících algoritmů . . . 6
- 1.2 Psaní vlastních programů . . . 10
- 1.3 Rozšiřování AAnimu . . . 13

2 Projekt AAnim **15**
- 2.1 Základní popis . . . 15
- 2.2 Vznik projektu . . . 15
- 2.3 Slepé uličky ve vývoji AAnimu . . . 15
- 2.4 Hlavní komponenty AAnimu . . . 16
- 2.5 Specifika programu . . . 16
- 2.6 Použití Javy v AAnimu . . . 18

3 Překladač **19**
- 3.1 Jazyk AL . . . 19
- 3.2 Lexikální analýza . . . 22
- 3.3 Syntaktická analýza . . . 24
- 3.4 Rozhraní překladače . . . 25

4 Data algoritmu **26**
- 4.1 Přidružené datové soubory . . . 26
- 4.2 Správa dat v AAnimu . . . 27
- 4.3 Formát datových souborů . . . 28

5 Vizualizace **31**
- 5.1 Zobrazovače a editory typů . . . 31
- 5.2 Animátory . . . 33

6 Závěr **38**

A Zodpovědnost za jednotlivé části AAnimu **39**
- A.1 Základní komponenty AAnimu . . . 39
- A.2 Moduly . . . 40
- A.3 Animátory . . . 40

Název práce: Prostředí pro animaci algoritmů — překladač a práce s daty
Autor: Petr Štěpán
Katedra: Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. Rudolf Kryl
E-mail vedoucího: kryl@ksvi.mff.cuni.cz

Abstrakt: Tato bakalářská práce popisuje některé z hlavních komponent softwarového díla AAnim. Program AAnim poskytuje prostředí pro prohlížení a vývoj vlastních programů v jednoduchém, Pascalu podobném jazyku AL. Programy v AL mohou být krokovány, lze zobrazovat i měnit hodnoty jejich proměnných pomocí tabulky proměnných i pomocí pokročilejších zobrazovačů a editorů typů. Specializovanější animaci prováděného programu obstarávají vizualizační moduly, tzv. animátory. K algoritmu lze z prostředí AAnimu vytvořit a dále používat a spravovat pojmenovaná vstupní data (vstupní sady proměnných) a konfigurace animátorů. AAnim je rozšiřitelný o moduly, které přidávají do AL nové typy, funkce a procedury, a animátory. V této práci jsou popsány tyto hlavní součásti AAnimu: překladač, jednotná správa dat algoritmu, editory a zobrazovače typů a animátory.

Klíčová slova: AAnim, animace, algoritmy, překladač, animátor

Title: Environment for Animation of Algorithms — Translator and Data Manipulation
Author: Petr Štěpán
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Rudolf Kryl
Supervisor's e-mail address: kryl@ksvi.mff.cuni.cz

Abstract: This bachelor's thesis describes some of the main components of the AAnim software project. AAnim offers an environment for viewing and creating own programs in a simple Pascal-like language named AL. AL programs can be single-stepped, it is possible to display and change the values of their variables using the table of variables or using more advanced renderers and type editors. The more specialised animation of the running program is provided by visualisation modules (named animators). The titled input data of an algorithm (input variable sets) can be created, further used and managed by AAnim. AAnim is extensible by adding modules which deliver new types, functions and procedures to AL and animators. The following main components of the program are described in this work: translator, unified algorithm data management, type editors and renderers and animators.

Keywords: AAnim, animations, algorithms, translator, animator

Předmluva

Softwarové dílo AAnim bylo vytvářeno dvěma studenty — Pavlem Římským a Petrem Štěpánem. Vzhledem k provázanosti prací jsou první dvě kapitoly bakalářských prací společné. V dodatku A lze pak nalézt bližší informace o autorství jednotlivých částí AAnimu.

Abychom čtenáři umožnili získat alespoň základní představu o funkcích AAnimu, zařadili jsme první kapitolu, v níž se snažíme přiblížit základní dovednosti programu a používané pojmy. V druhé pak prezentujeme obecné informace o projektu (hlavní komponenty, popis vývoje atp.).

Pro úplné pochopení dalšího textu doporučujeme nejdříve přečíst uživatelskou dokumentaci dostupnou z prostředí programu AAnim (na CD přiloženém k této práci). Bylo by sice možné ji exportovat a po úpravách ji vytisknout, ale považovali jsme to za samoúčelné. Z nápovědy AAnimu je také dostupná generovaná programátorská dokumentace detailně popisující použité třídy a rozhraní.

Tato bakalářská práce pojednává o hlavních komponentách, jejichž autorem je Petr Štěpán.

V kapitolách 3 až 5 popisuji objektový návrh komponent, které jsem vytvořil. Kromě objektového návrhu se zabývám i jeho motivací a zvažovanými alternativami.

Kapitola 1

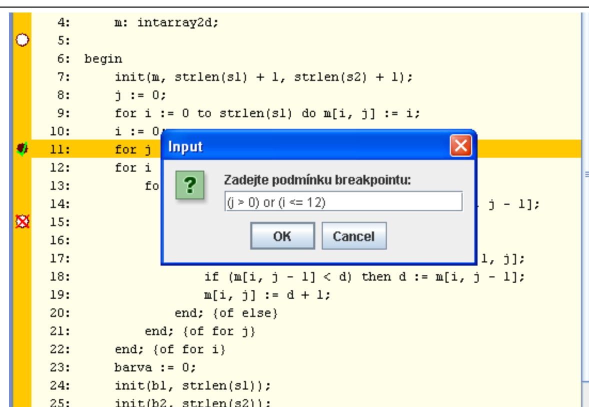
Představení programu AAnim

V této úvodní kapitole čtenáři pomocí velkého množství screenshotů představíme program AAnim z uživatelského hlediska. Každý obrázek má za cíl ukázat jednu konkrétní dovednost tohoto programu.

1.1 Zkoumání již existujících algoritmů

- sledování, která část zdrojového kódu algoritmu se zrovna provádí, nastavování breakpointů a podmíněných breakpointů

Obrázek 1.1 Nastavování breakpointů (levý pruh), podmínek breakpointů (dialog v popředí) a sledování, která část zdrojového kódu se provádí (oranžově zvýrazněný řádek)



- sledování a změna aktuální hodnoty proměnných

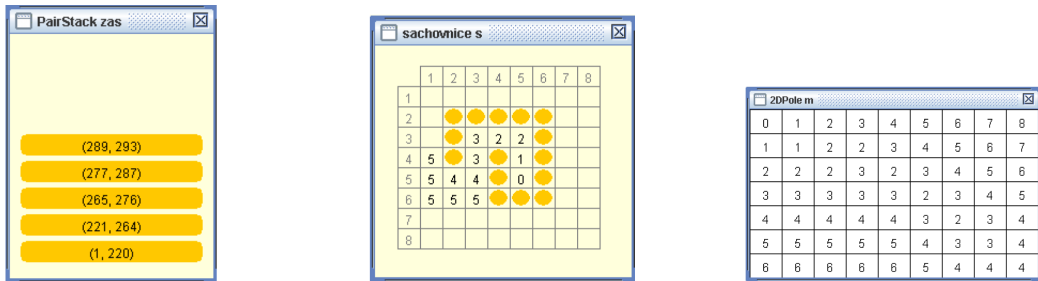
Obrázek 1.2 Sledování hodnot proměnných pomocí tabulky, hodnota proměnné j je právě měněna

Tabulka proměnných

cyklus	fáze slévání	4
fibo	pole Fib. čísel	[8, 12, 14, 15]
i	řídící proměnná ve for cyklech	4
j	řídící proměnná ve for cyklech	16
jeste	true, dokud není dotříděno	false
n	počet pásek	4
pomocne	pole pomocných pásek	[0, 0, 0, 1, 0]
sum	max. počet běhů při daném rozlož...	49
tmp	pomocná proměnná při generová...	8
vstupni	vstupní páska	0

- u některých datových typů sledování aktuální hodnoty proměnných pomocí speciálních zobrazovačů (tzv. *rendererů*)

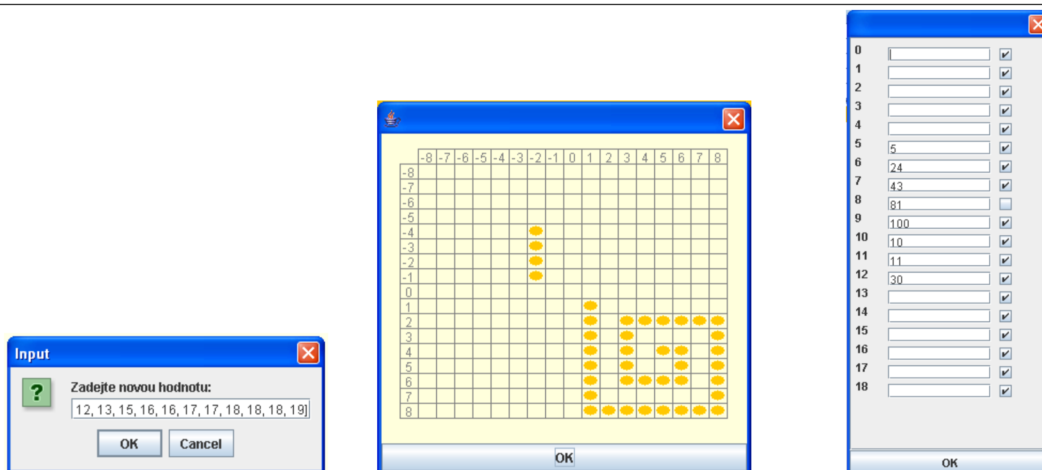
Obrázek 1.3 Zobrazovače typů



renderer typu `PairStack` (zásobník intervalů) | renderer typu `sachovnice` | renderer typu `2DPole`

- u některých datových typů změna aktuální hodnoty proměnných pomocí speciálních editorů

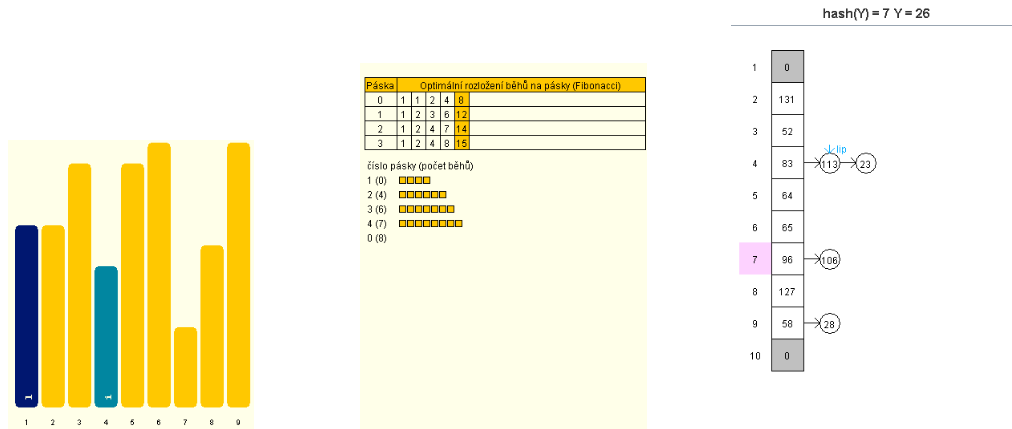
Obrázek 1.4 Editory typů



editor typu `intarray` | editor typu `polepozic` | editor typu `hashtable`

- animování velkého množství algoritmů v samostatném panelu (tvz. *animátoru*)

Obrázek 1.5 Ukázky animátorů



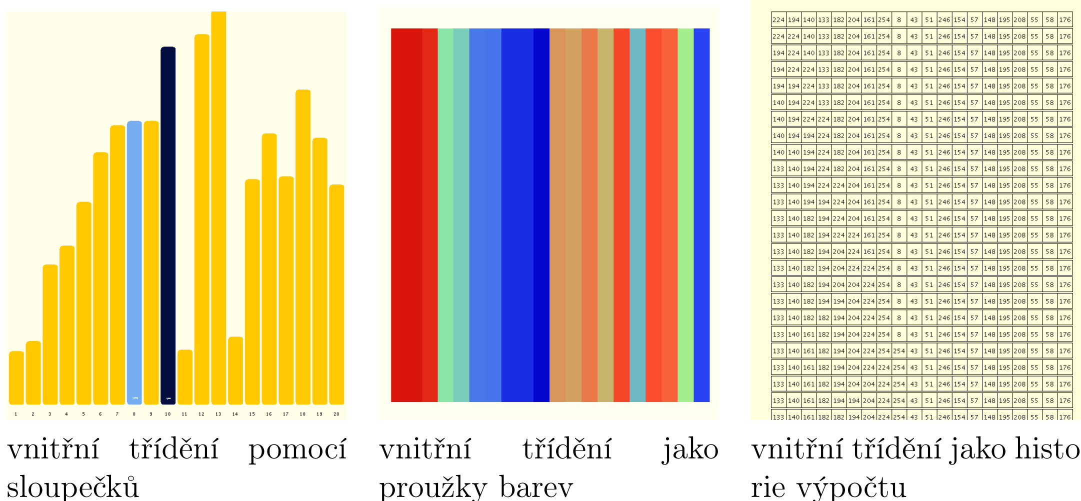
sloupečkový animátor algoritmů vnitřního třídění — polyfázové třídění — hashování s oblastí přetečení

- vybírání z více animátorů pro jeden algoritmus

Obrázek 1.6 Zdrojový kód algoritmu vnitřního třídění (konkrétně insertsort)

```
 1:  {TRIDICI ALGORITMUS INSERTSORT}
 2:
 3:  input a: intarray;
 4:
 5:  var i, j, tmp: int;
 6:      posunvpravo: bool;
 7:
 8:  begin
 9:      if sizeof(a) = 0 then a := randomSequence(20, 255);
10:      for i := 2 to sizeof(a) do begin
11:          tmp := a[i];
12:          j := i - 1;
13:          while (j > 0) and (a[j] > tmp) do begin
14:              a[j + 1] := a[j];
15:              j := j - 1;
16:          end;
17:          a[j + 1] := tmp;
18:      end;
19:  end;
```

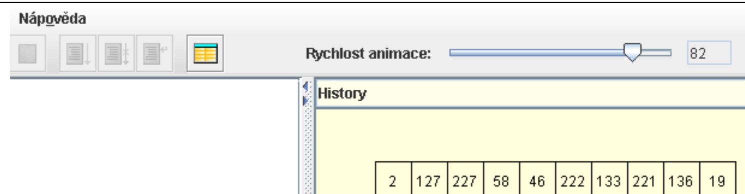
Obrázek 1.7 Tři animátory pro algoritmus insertsort



vnitřní třídění pomocí sloupečků | vnitřní třídění jako proužky barev | vnitřní třídění jako historie výpočtu

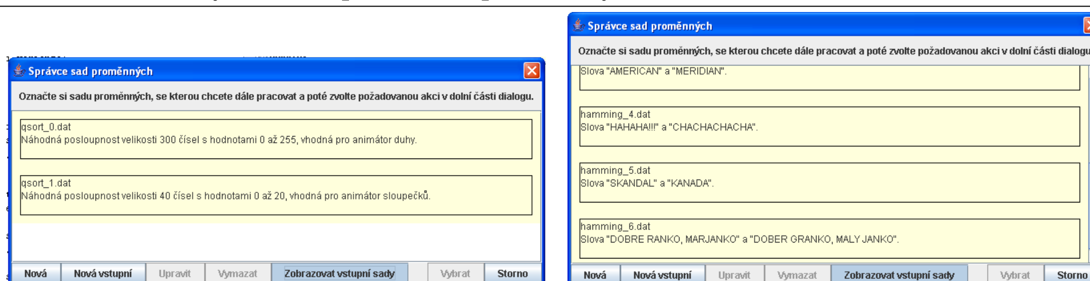
- nastavování rychlosti provádění algoritmu

Obrázek 1.8 Nastavování rychlosti provádění algoritmu



- výběr z nabídky ukázkových vstupních dat algoritmů

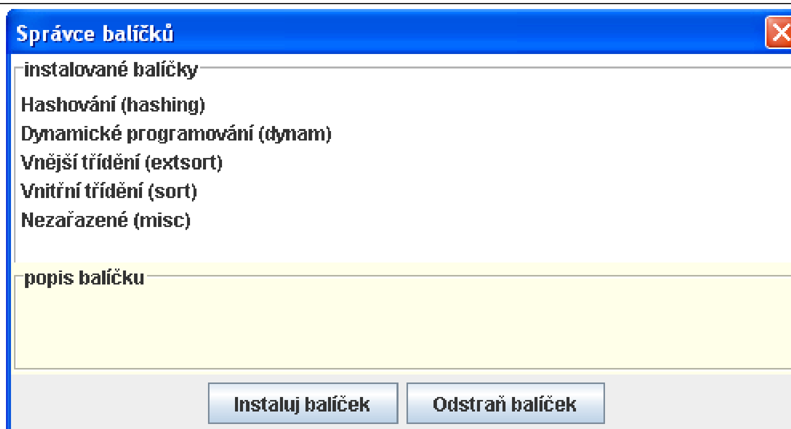
Obrázek 1.9 Výběr vstupních sad proměnných



vstupní data pro algoritmus quicksort | vstupní data pro algoritmus Hammingova vzdálenost

- rozšiřování AAnimu o nové algoritmy, animace či zobrazovače a editory hodnot proměnných instalací tzv. *balíčků*

Obrázek 1.10 Správce balíčků



1.2 Psaní vlastních programů

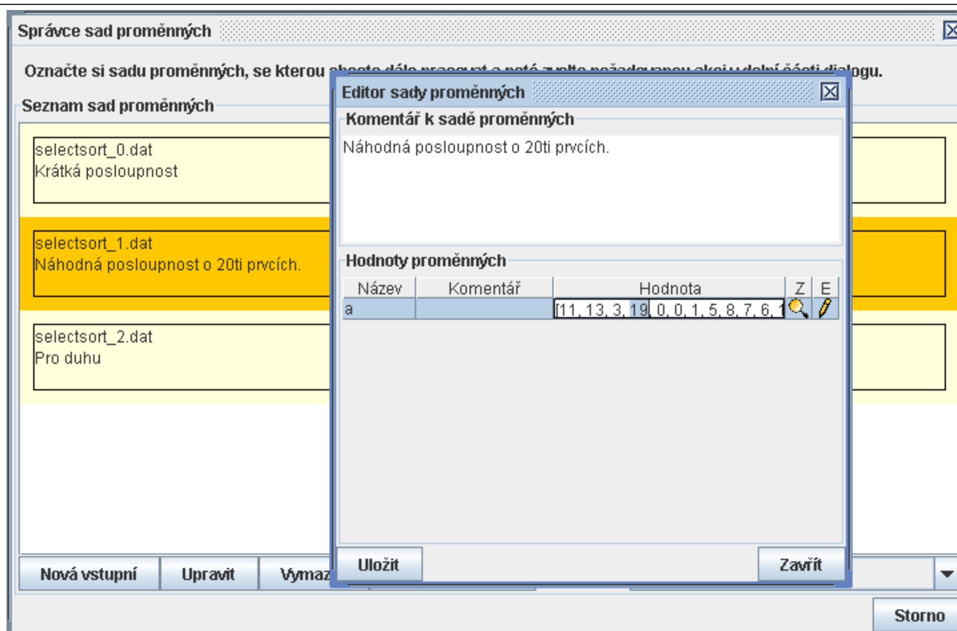
- vytváření vlastního zdrojového kódu v AL (AAnim Language), lehce zvládnutelném, Pascalu podobném jazyku

Obrázek 1.11 Algoritmus třídění výběrem v AL

```
 1:  input
 2:      a: intarray;
 3:  var
 4:      i, minindex, j, tmp: int;
 5:  begin
 6:      if sizeof(a) = 0 then a := randomSequence(100, 255);
 7:      for i := 1 to sizeof(a) do begin
 8:          minindex := i;
 9:          for j := i + 1 to sizeof(a) do begin
10:              if a[j] < a[minindex] then
11:                  minindex := j;
12:         end;
13:         tmp := a[i];
14:         a[i] := a[minindex];
15:         a[minindex] := tmp;
16:      end;
17:  end;
```

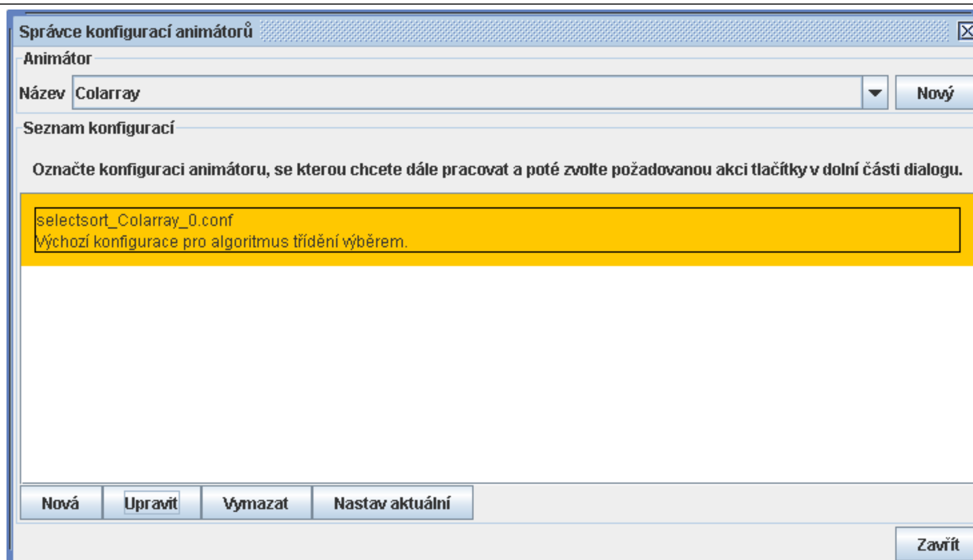
- vytváření, úprava a ukládání vstupních sad algoritmu v Editoru sad proměnných

Obrázek 1.12 Úprava vybrané vstupní sady pro algoritmus třídění výběrem. V Editoru sad proměnných lze pohodlně upravovat hodnoty proměnných vstupní sady.

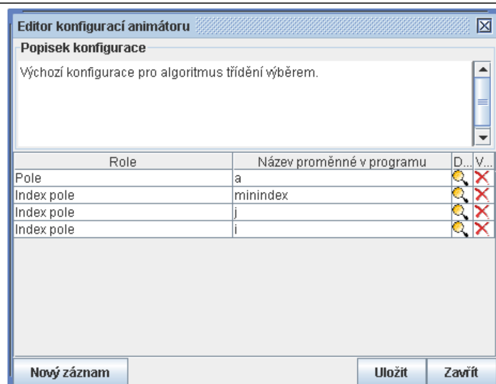


- možnost využít pro animace svého vlastního programu již existujících animátorů vytvořením konfigurace animátoru, souboru, v němž je vybraným proměnným z programu přiřazena jistá role v animaci

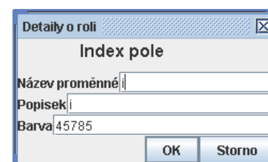
Obrázek 1.13 Správce konfigurací animátorů umožňuje vytvářet, mazat a upravovat již existující konfigurační soubory animátorů.



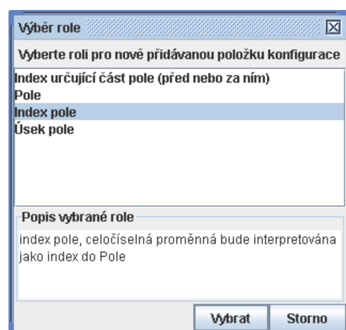
Obrázek 1.14 Konfigurace animátoru



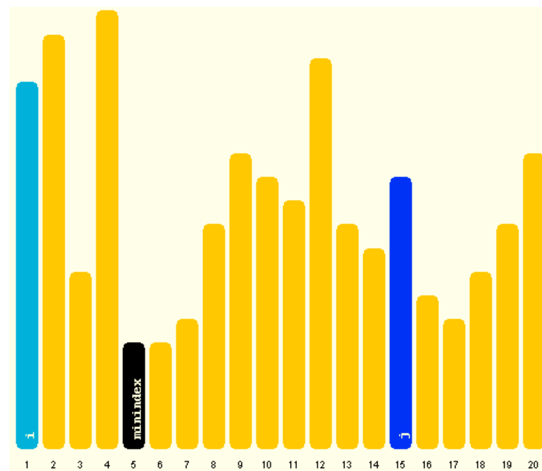
Každá proměnná, kterou animátor využívá má přiřazenu při animaci nějakou roli. V Editoru konfigurací animátoru lze s těmito přiřazeními (položkami konfiguračního souboru) pracovat.



Role však může mít další atributy, které ovlivňují její chování při animaci, např. vzhled. Tyto atributy lze pro každou roli editovat v dialogu Detaily o roli.



Při přidávání nových položek do konfigurace se nejprve uživateli nabídne seznam animátorem nabízených rolí včetně jejich krátkého popisu.



A konečně výsledek: animátor zobrazuje průběh algoritmu na základě informací z konfiguračního souboru.

1.3 Rozšiřování AAnimu

Každý, kdo ovládá Javu, může rozšířit jazyk AL o nové datové typy, operace na datových typech, zobrazovače typů, editory typů, funkce či procedury (balíku takovýchto rozšíření říkejme *modul*), a to naprosto nezávisle na zbytku zdrojového kódu AAnimu (autor vůbec nemusí mít k dispozici zdrojový kód AAnimu). Stejně snadno se dá AAnim rozšířit o nové animátory.

- vytváření nových datových typů

Obrázek 1.15 Výňatek ze zdrojového kódu Javy, kde se definuje nový datový typ. A použití takto definovaného typu v pseudokódu.

```
public class PCType_stack extends PCType_general{
    protected Stack s;
    public PCType_stack(){
            s = new Stack();
    }//PCType_stack
    public void push(int a){
            s.push(new Integer(a));
    }//push
    public int pop(){
            return ((Integer) s.pop()).intValue();
    }//pop
```

```
1:  {QUICKSORT - nerekurzivni varianta}
2:  module pairstack;
3:
4:  input pole: intarray;
5:
6:  var usek: intpair; {dvojice indexu urcujici aktualne trideny usek
7:      zas: pairstack; { zasobnik useku k prohledani}
8:      pivot, pivotindex, i, j, zac, kon: int;
9:
10: begin
11:     if sizeof(pole) = 0 then pole := randomSequence(40, 20);
12:     setpair(usek, 1, sizeof(pole));
13:     push(zas, usek);
14:     while not isEmpty(zas) do begin
15:         usek:= pop( zas);
```

- definování operací na těchto typech

Obrázek 1.16 Výňatek ze zdrojového kódu Javy, kde se definuje operace sčítání na řetězcích. A použití takto definované operace v pseudokódu.

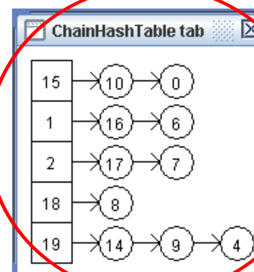
```
public static PCType_string aanim_add(PCType_string a, PCType_string b){
    return new PCType_string(a.stringValue() + b.stringValue());
}//aanim_add
```

```
1:  var s1, s2, s: string;
2:  begin
3:      s1 := "ahoj";
4:      s2 := " lidi";
5:      s := s1 + s2;
6:  end;
```

- definování vlastních zobrazovačů a editorů vlastních datových typů

Obrázek 1.17 Výňatek ze zdrojového kódu Javy, kde se definuje zobrazovač hashovací tabulky s oblastí přetečení. A jeho použití při animaci.

```
class ChainHashTableRenderer extends DefaultTypeRenderer{
    //hodnota, jež se vykresluje v zobrazovači
    private PCType_chainhashtable table;
    //konstanty určující vzhled
    private final int MARGIN= 10, ITEMSIZE= 30, CHAINITEMSIZE= 25 ;
    private final int ARROWLEN= 20, ARROWING= 9;
    private final Color bgColor= Color.white, emptyColor= Color.lightGray,
    fullColor= Color.white, fontColor= Color.black;
```



- definování vlastních funkcí a procedur — rozšiřování jazyka AL

Obrázek 1.18 Výňatek ze zdrojového kódu Javy, kde se definuje procedura `mergerunsto`. A použití takto definované procedury v pseudokódu.

```
public static void aanim_mergerunsto(PCType_polyphtapearray tapes, PCType_int destination){
    int x = destination.intValue();
    for (int i=0; i < tapes.numtapes(); i++){
        if (i == x)
            tapes.get(i).addRuns(1);
        else
            tapes.get(i).removeRun();
    }//for
}//aanim_mergerunsto
```

```
while sizeof(pomocne[cyklus mod (n+1)]) > 0 do
    mergeRunsTo(pomocne, (n+cyklus) mod (n+1));
```

- definování vlastních animátorů

Obrázek 1.19 Výňatek ze zdrojového kódu Javy, kde se definuje animátor Merge. A jeho použití při animaci.

```
/** Animátor nerekurzivní verze algoritmu vnitřního třídění slučováním */
public class Merge extends General {
    private MyPanel panel = new MyPanel();
    private RectArray first= new RectArray();
    private RectArray second= new RectArray();
```

l m

45	89	30	78	28	45	6	31	31	84	65	69	49	58	56	58

o

30	45	78	89	28	45	31	6	31	84	69	65	58	49	56	58

Kapitola 2

Projekt AAnim

2.1 Základní popis

Program AAnim (*Algorithm Animation*) slouží jako pomůcka studentům při studiu jednodušších algoritmů. Zahrnuje vývojové prostředí, které umožňuje editaci algoritmu v pseudokódu, jeho krokování a sledování hodnot proměnných. AAnim je možné rozšířit o moduly (napsané v jazyce Java) dodávající do pseudokódu další funkce a procedury. K algoritmu lze také připojit vizualizační moduly (nazývané *animátory*), které obstarávají animaci průběhu algoritmu.

2.2 Vznik projektu

Projekt AAnim začal vznikat na jaře roku 2005. RNDr. Rudolf Kryl potřeboval vytvořit několik animací algoritmů, které se přednášejí nebo by se mohly přednášet v kurzech programování na Matematicko-fyzikální fakultě UK v Praze. Po několika konzultacích se doktor Kryl rozhodl, že pro fakultu by byl vhodný program, který by dokázal nejenom zobrazovat průběh animace, ale šly by v něm nové animace i vytvářet. Algoritmy by se popisovaly jednoduchým jazykem, který by vycházel z Pascalu (dále v textu tento jazyk budeme nazývat *pseudokód* nebo jazyk AL). Pseudokód by měl být rozšiřitelný o nové funkce a procedury, přičemž ty by se nemusely psát přímo v pseudokódu, ale mělo by být možné o ně jazyk snadno obohatit.

Vzhledem k tomu, že se mělo jednat o poměrně náročný projekt, RNDr. Kryl usoudil, že by jej mohl svěřit hned dvěma studentům. Každý z nich by byl odpovědný za jednu část projektu. Po několika dalších konzultacích došlo k dohodě, že student Petr Štěpán vytvoří kompilátor pseudokódu (lexikální a syntaktickou analýzu) a student Pavel Římský vytvoří sémantickou analýzu a interpret.

Kompilátor a interpret však tvoří pouze jádro programu. Program se skládá z velkého množství komponent. Tyto komponenty jsou detailněji popsány v samostatné části každého z tvůrců programu.

2.3 Slepé uličky ve vývoji AAnimu

Původně jsme měli v úmyslu vytvořit několik na sobě nezávislých aplikací či apletů, každý z nich by zobrazoval jeden konkrétní algoritmus. Neuvažovali jsme však o tom,

že by se tyto algoritmy definovaly v pseudokódu – původní návrh počítal s tím, že logika animovaných algoritmů by byla obsažena přímo ve zdrojovém kódu v Javě. RNDr. Kryl však již za začátku projevil zájem o univerzálnější nástroj, který by byl snadno rozšiřitelný.

Konkrétnější podoba našeho projektu se začala rýsovat teprve po několika sezeních s RNDr. Krylem. Nejprve měl být jazyk pseudokódu velmi jednoduchý, obsahovat pouze datový typ pro celé číslo a logickou hodnotu. Je zřejmé, že pro předvádění pokročilejších algoritmů tyto dva typy nestačí. Co víc, pokud měl být náš projekt univerzálně použitelným nástrojem, bylo potřeba umožnit uživateli, aby si definoval vlastní datové typy. Nelze totiž dopředu předvídat, jaké různé algoritmy bude kdo chtít naším programem animovat a jaké datové typy budou tyto algoritmy používat (například zásobník je potřeba pro algoritmus quicksort, fronta pro některé grafové algoritmy).

Jedním z prvních (s odstupem času lze říct že kuriózních) návrhů, jak by mohl náš program vypadat, byl generátor zdrojového kódu pro jazyk Java. Princip by byl takový, že autor animace by napsal algoritmus v pseudokódu, své vlastní datové typy, procedury a animace by napsal v Javě a náš program by z pseudokódu vygeneroval Javovský zdrojový kód, který by připojil k Javovskému kódu napsanému autorem animace. Tím by vznikl Javovský aplet, který by animoval určitý algoritmus. Vzhled tohoto apletu by mohl autor ovlivnit vhodnou konfigurací. Jednalo by se de facto o „továrnu na animace“. Cílem by bylo uživateli ulehčit psaní apletu. Za normálních okolností by se totiž programátor musel starat o řízení rychlosti animace, krokování, zobrazování pseudokódu a zvýrazňování prováděných řádků a spoustu dalších technických záležitostí. S naším generátorem apletů by se programování výrazně ulehčilo – část kódu obstarávající tuto nutnou režii by byla automaticky generovaná.

V zásadě se nejednalo o špatný nápad. Samozřejmě by uživatel neměl možnost pseudokód editovat (leda že by si potom vygeneroval nový aplet). Paralelně s tímto návrhem jsme však již pracovali na alternativě, k níž jsme se nakonec přiklonili. Na smetišti dějin pak návrh generátoru skončil po otázce RNDr. Kryla „Jakou by to mělo výhodu oproti tomu druhému řešení?“ a naší odpovědi „žádnou“.

Během celého vývoje projektu jsme byli stavěni před další a další rozhodnutí, kudy se má projekt dále ubírat. I zde jsme narazili na několik slepých uliček. Ty jsou detailněji popsány v samostatných částech obou autorů.

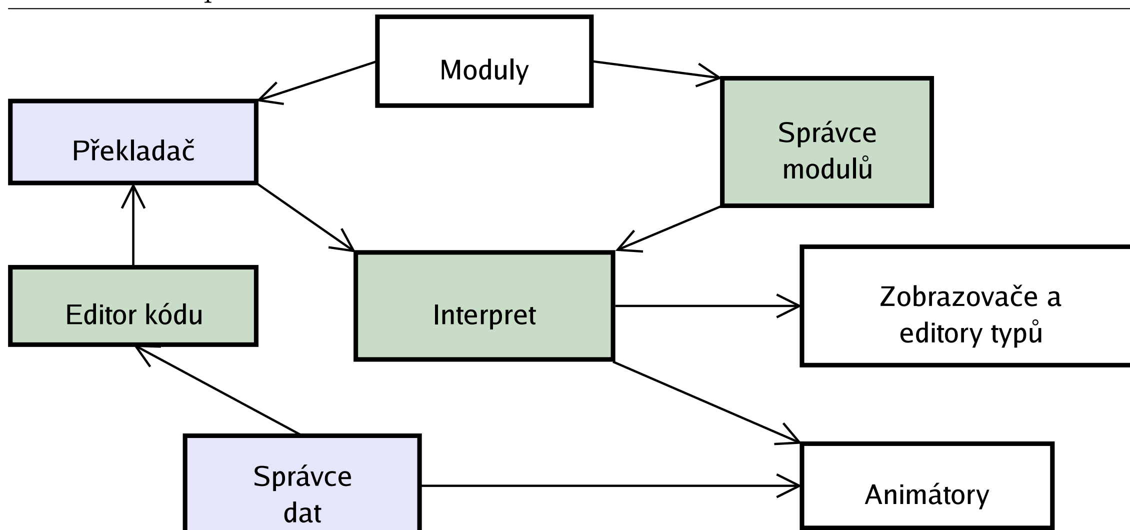
2.4 Hlavní komponenty AAnimu

AAnim sestává z mnoha komponent. Přikládáme velmi zjednodušené schéma (obrázek 2.1) a krátké popisy nejdůležitějších z nich, abychom zjednodušili čtenáři orientaci v dalším textu. Podrobnější informace lze nalézt v dalších kapitolách prací obou autorů (viz přílohu Zodpovědnost za jednotlivé části AAnimu).

2.5 Specifika programu

Většina ostatních programů, které mají za cíl přiblížit studentovi princip fungování algoritmu pomocí animace jeho průběhu, se charakterem svého návrhu podobá filmu a studenta staví spíše do role pasivního pozorovatele s malými možnostmi ovlivnit

Obrázek 2.1 Hlavní komponenty AAnimu — modře zvýrazněné komponenty vytvořil Petr Štěpán



průběh animace. Ty se ve většině případů omezují na změnu rychlosti probíhající animace či výběr z několika autorem předem připravených kolekcí vstupních dat. Zdrojový kód, pokud je vůbec možné ho zobrazit, potom slouží spíše k informaci uživateli, v jaké části algoritmu se právě vizualizace nachází.

My jsme se snažili (po různých peripetiích popsaných v předchozí části) navrhnout AAnim tak, aby poskytl studentovi prostředky k hlubšímu a aktivnímu zkoumání principu algoritmu. Hlavní vlastnosti a nástroje, které mu to umožní, shrnuje následující seznam.

- interpretovaný pseudokód

  Student může modifikovat již existující algoritmy a vidět, jak provedené změny ovlivní chování algoritmu, nebo vytvářet své vlastní programy.

- zobrazení a změna hodnot proměnných v průběhu algoritmu

  Pomocí tabulky proměnných lze zobrazovat i upravovat hodnoty jednodušších typů přímo, pro složitější typy existuje mechanismus zobrazovačů a editorů, které může uživatel využít i při absenci předpřipravené animace, tedy například pro své vlastní algoritmy.

- vstupní sady

  Načítání iniciálních hodnot vstupních proměnných využijí zejména autoři algoritmů, kteří mohou připravit dostatečně „vysvětlující“ data.

- konfigurovatelné animační moduly (*animátory*)

  Hlavní prostředek vizualizace algoritmů, na základě hodnot proměnných, jež mu jsou poskytovány prostředím, vytváří animaci algoritmu. Možnost sdílet jeden animátor mezi více algoritmy a pomocí konfiguračních souborů přizpůsobit např. různému pojmenování proměnných stejného významu, zjednodušuje práci autorům animátorů, neboť je činí univerzálnějšími. Naproti tomu jeden

algoritmus může být vizualizován více animátory, což může zlepšit studentovo porozumění danému programu.

Podrobnější výčet funkcí a dalších možností, jak rozšířit AAnim pomocí Javy, nalezne čtenář v dokumentaci (přístupné z nápovědy AAnimu).

2.6 Použití Javy v AAnimu

Ačkoliv jazyk Java nepatří k nejuniverzálnějším, pro tento typ projektu nabízí řadu výhod. Nejprve si shrňme obecné klady Javy. Java eliminuje chyby programátora, je jednoduchá. To umožňuje, aby se programátoři soustředili převážně na užitečnou práci a neztráceli tolik času hledáním chyb. Další výhodou Javy je bohatá standardní knihovna. Není třeba složitě instalovat přídavné moduly, takřka vše, co programátor bude při své práci potřebovat, je součástí standardní knihovny Javy. Tato knihovna je navíc podrobně a přehledně zdokumentovaná. Za vývojové nástroje a vývojové prostředí se nemusí nic platit a programy napsané v Javě běží pod velkým množstvím běžně používaných operačních systémů.

Nyní je třeba si zodpovědět otázku, zda se nevýhody Javy nějak zásadně projeví na našem projektu. Javu řadíme mezi interpretované jazyky. Psát interpret v interpretovaném jazyce se může zdát nerozumné. Proto jsme se museli zamyslet nad tím, jestli rychlost Javy nebude představovat problém. Jelikož náš projekt má sloužit k vizualizaci algoritmů a typickým uživatelem bude student, který se snaží pochopit princip tohoto algoritmu, rychlost interpretace není podstatná. Algoritmy budou typicky krokovány, aby byl uživatel schopný zamyslet se nad jejich principem, často si budou uživatelé nastavovat breakpointy. Jelikož je projekt již téměř hotový, lze zpětně konstatovat, že jsme se v předpokladu nemýlili. Rychlost Javy opravdu není omezujícím faktorem.

Malá univerzalita Javy u tohoto typu projektu nevadí. Žádné speciální operace se nepoužívají. Naopak se ukázaly užitečné některé speciální vlastnosti Javy. Jde především o takzvané *Reflection API*, tedy možnost uložit zkompilovanou třídu do souboru a pomocí standardního rozhraní zjišťovat, jaké obsahuje metody a jakého jsou typu, a navíc vytvářet instance těchto metod. Reflection API je využito pro rozšiřování pseudokódu o nové funkce, procedury, datové typy a animace. Nutno ovšem podotknout, že ne vždy používání knihovny Javy urychluje práci. Existují i výjimky. O tom blíže pojednává část o editoru kódu v práci Pavla Římského.

Kapitola 3

Překladač

První fází ve zpracování zdrojového souboru algoritmu je překlad kódu do objektových struktur, se kterými dále pracuje interpret. Tento úkol mají na starosti třídy z balíčku `aanim.translator`.

Tato kapitola popisuje proces překladu počínaje lexikální analýzou (a souvisejícími třídami `Token` a `Tokenizer`) až po syntaktickou analýzu, jejíž úkoly plní potomci třídy `Parser`. Kromě popisu objektového návrhu tříd (jehož technické detaily se čtenář nejlépe dozví z programátorské a generované dokumentace k jednotlivým třídám) se budeme snažit rozebírat také důvody, které nás vedly k jejich konkrétní implementaci, jakož i výhody a nevýhody alternativních postupů.

3.1 Jazyk AL

Ještě dříve než přistoupíme k popisu samotného překladače, musíme něco říci o překládaném jazyku. Jazyk vytvořený pro účely zápisu algoritmů v AAnimu jsme nazvali AL (*AAnim Language*). AL byl navržen tak, aby byl

- dostatečně mocný

  Byl sice navrhován v době, kdy už jsme byli rozhodnuti, že nebude obsahovat definice funkcí a procedur, které budou dodávány externími moduly, a že v něm budou psány programy podobné např. hlavnímu bloku pascalského programu. Ale uživatele jsme nechtěli příliš omezovat, a proto jsme zařadili cykly, podmínky a široký repertoár operátorů.

- intuitivní a lehce zvládnutelný

  Vymýšlet samoúčelně nějaké exotické syntaktické konstrukty jen proto, abychom AL odlišily od stávajících jazyků by bylo kontraproduktivní, neboť naším cílem bylo co nejméně omezovat uživatele a naopak jej podporovat v práci s AL. Rychlé a bezproblémové zvládnutí jazyku AL je jistě nutná podmínka k tomu, aby uživatelé celé prostředí AAnimu používali. Vzhledem k cílové skupině – studentům učícím se nové algoritmy, jsme zvolili jako základ jazyk Pascal.

- snadno přeložitelný

  Jednoduchá konstrukce překladače nebyla samozřejmě hlavním vývojovým kritériem, ale jistě jsme k ní v nejednoznačných případech přihlíželi. V tomto

ohledu také pomohl fakt, že jsme si za základ zvolili jazyk Pascal, který sám je známý jako nepříliš na překlad náročný.

Díky specifickému použití jsme mohli do AL zařadit prvky, které se v běžných, univerzálních jazycích nevyskytují. Jedná se hlavně o

- *vstupní proměnné*, speciální druh proměnných, které se v algoritmu užívají jako parametr v těle procedury, jako vstupní hodnota, s níž program dále v průběhu manipuluje. Byla pro ně zavedena zvláštní deklarace uvozená klíčovým slovem **input**. U programu obsahujícího vstupní proměnné je při jeho spuštění uživatel dotázán, zda chce načíst vstupní data (iniciální hodnoty vstupních proměnných) ze souboru, z tzv. *vstupní sady proměnných* (více o sadách proměnných v kapitole Data algoritmu).
- popisné komentáře. Jedná se opět o rozšíření deklarace proměnných. Tentokrát se její součástí může stát řetězec, jenž je AAnimem interpretován jako popis úlohy dané proměnné v algoritmu, a je za běhu programu zobrazen v tabulce proměnných. Tato funkce umožní uživateli snadnější orientaci v tabulce při sledování hodnot proměnných. AL obsahuje i klasické komentáře (pascalské `{}`), ale ty si zachovávají svou nulovou sémantickou hodnotu a jsou odstraňovány z dalšího zpracování již ve fázi lexikální analýzy.

Nejlepší představu o jazyku AL si čtenář může udělat sám z gramatiky (v Backus-Naurově formě), kterou přikládáme. Konkrétnější návod (včetně struktury programu v AL) lze nalézt v Uživatelské dokumentaci.

```
<PROGRAM>                ::= <DIR_MODUL><DEKLARACE_PROMENNYCH><TELO>

<DIR_MODUL>              ::= λ | module <ALPH_ALPHNUM>;

<DEKLARACE_PROMENNYCH>   ::= <DEKL_VSTUP_PROM><DEKL_PROM>
<DEKL_VSTUP_PROM>        ::= λ | input <DEFINICE_PROMENNE><DEFINICE_PROMENNYCH>
<DEKL_PROM>              ::= λ | var <DEFINICE_PROMENNE><DEFINICE_PROMENNYCH>
<DEFINICE_PROMENNE>      ::= <NAZEV_PROMENNE><POPIS_PROMENNE><DALSI_PROMENNE>:
                             <TYP_PROMENNE>;
<POPIS_PROMENNE>         ::= λ | <RETEZEC>
<DALSI_PROMENNE>         ::= λ | ,<NAZEV_PROMENNE><POPIS_PROMENNE><DALSI_PROMENNE>
<NAZEV_PROMENNE>         ::= <ALPH_ALPHNUM>
<TYP_PROMENNE>           ::= <ALPH_ALPHNUM>
<DEFINICE_PROMENNYCH>    ::= λ | <DEFINICE_PROMENNE><DEFINICE_PROMENNYCH>

<TELO>                   ::= <BLOK>
<PRIKAZ>                 ::= <IF_PRIKAZ> | <FOR_PRIKAZ> | <PRIRAZENI> |
                             <ZAPIS_DO_POLE> | <WHILE_PRIKAZ> |
                             <VOLANI_PROCEDURY> | <BLOK>

<IF_PRIKAZ>              ::= if <VYRAZ> then <PRIKAZ> |
                             if <VYRAZ> then <PRIKAZ> else <PRIKAZ>

<FOR_PRIKAZ>             ::= for <NAZEV_PROMENNE> := <VYRAZ> to <VYRAZ> do
```

```
                           <PRIKAZ> |
                           for <NAZEV_PROMENNA> := <VYRAZ> downto <VYRAZ> do
                           <PRIKAZ>

<PRIRAZENI>             ::= <NAZEV_PROMENNE> := <VYRAZ>;
<ZAPIS_DO_POLE>         ::= <NAZEV_PROMENNE>[<VYRAZ><INDEXY>] := <VYRAZ>;
<INDEXY>                ::= λ | <INDEXY>,<VYRAZ>

<WHILE_PRIKAZ>          ::= while <VYRAZ> do <PRIKAZ>

<VOLANI_PROCEDURY>      ::= <NAZEV_PROCEDURY>(<PARAMETRY>);
<NAZEV_PROCEDURY>       ::= <ALPH_ALPHNUM>
<PARAMETRY>             ::= λ | <NEPRAZDNE_PARAMETRY>
<NEPRAZDNE_PARAMETRY>   ::= <VYRAZ> | <NEPRAZDNE_PARAMETRY>,<VYRAZ>

<BLOK>                  ::= begin <POSL_PRIKAZU> end;
<POSL_PRIKAZU>          ::= λ | <POSL_PRIKAZU><PRIKAZ>

<VYRAZ>                 ::= <LOGICKY_TERM> | <LOGICKY_TERM> or <VYRAZ>
<LOGICKY_TERM>          ::= <LOGICKY_LITERAL> |
                            <LOGICKY_LITERAL> and <LOGICKY_TERM>
<LOGICKY_LITERAL>       ::= <PREDIKAT> | not <PREDIKAT>
<PREDIKAT>              ::= <ARITMETICKY_VYRAZ> |
                            <ARITMETICKY_VYRAZ><REL_OP><ARITMETICKY_VYRAZ>
<REL_OP>                ::= = | < | > | <= | >= | <>
<ARITMETICKY_VYRAZ>     ::= <TERM> | <ARITMETICKY_VYRAZ><ADD_OP><TERM>
<ADD_OP>                ::= + | -
<TERM>                  ::= <FAKTOR> | <TERM><MUL_OP><FAKTOR>
<MUL_OP>                ::= * | / | div | mod
<FAKTOR>                ::= <UNARNI_OP><FAKTOR> | <PODFAKTOR>
<UNARNI_OP>             ::= + | -
<PODFAKTOR>             ::= <CISLO> | <RETEZEC> | <LOG_KONSTANTA> | (<VYRAZ>) |
                            <VOLANI_FUNKCE> | <NAZEV_PROMENNE> | <PRVEK_POLE>

<CISLO>                 ::= <NUM><NUMS> | <NUM><NUMS>.<NUMS>
<RETEZEC>               ::= "<ALPH_ALPHNUM>"
<LOG_KONSTANTA>         ::= true | false

<VOLANI_FUNKCE>         ::= <NAZEV_FUNKCE>(<PARAMETRY>)
<NAZEV_FUNKCE>          ::= <ALPH_ALPHNUM>

<PRVEK_POLE>            ::= <NAZEV_PROMENNE>[<VYRAZ><INDEXY>]

<ALPHA>                 ::= a | ... | z | A | ... | Z | _
<NUM>                   ::= 0 | ... | 9
<NUMS>                  ::= λ | <NUMS><NUM>
<ALPHNUM>               ::= <NUM> | <ALPHA>
<ALPHNUMS>              ::= λ | <ALPHNUMS><ALPHNUM>
<ALPH_ALPHNUM>          ::= <ALPHA><ALPHNUMS>
```

Na ukázku si ještě můžete prohlédnout následující příklad velmi jednoduchého programu v AL. Všimněte si zejména deklarační sekce.

```
{výpočet průměru prvků pole a}
module general;
input a "vstupní pole čísel": IntArray;
var i: Int;
    prumer "počítaný průměr": Real;
begin
    prumer:= 0;
    for i:= 1 to sizeof(a) do
        prumer:= prumer + a[i];
    prumer:= prumer / sizeof(a);
end;
```

3.2 Lexikální analýza

První fáze překladu převádí vstupní řetězec – kód algoritmu v pseudokódu – na posloupnost lexikálních elementů, logicky souvisejících částí vstupního řetězce (klíčová slova jazyka, řetězce v uvozovkách, čísla, jednoznakové symboly), nazývaných též *tokeny*. Současně se ještě filtrují v kódu obsažené sémanticky nevýznamné informace jako jsou komentáře a bílé znaky.

V mnoha otázkách týkajících se lexikální analýzy a její implementace v Javě jsme inspiraci k řešení nalezli v knize Stevena Johna Metskera *Building Parsers With Java*.

Tokenizer

Hlavní třídou této části překladače je `Tokenizer`, která převádí vstupní řetězec znaků (vstupující jako parametr konstruktoru) na posloupnost tokenů a tak realizuje úlohu lexikální analýzy.

Při vyžádání tokenu ze vstupního proudu se nejprve klasifikuje první znak (číslice, symbol, desetinná čárka, bílý znak, uvozovka, začátek komentáře, atp.) a na základě tohoto určení se provede rutina zpracovávající daný lexikální element. Pro určení znaků používáme pole, jež přiřazuje každému znaku (z US ASCII, neboť všechny ostatní v AL nemají speciální význam) jeho třídu. Tato implementace má řadu výhod

- klasifikace znaku je velmi rychlá (jeden přístup do pole)
- lze velmi flexibilně měnit význam speciálních symbolů (změna uvozovek pro řetězce, symbolů uvozujících komentáře ap.).

Ovšem není to univerzální řešení, problém představují víceznakové symboly (např. `<=` či `:=`). Obecnějším řešením by mohla být stromová struktura použitá pro uložení příznaků vstupních znaků. Její vrcholy by představovaly znaky (spolu s příznakem jejich významu) a hrana z uzlu A do B by existovala, právě když by znak v A předcházel znaku B v nějakém vícepísmenném symbolu. Při načítání znaků ze vstupu by se sestupovalo ve stromě tak dlouho, dokud by to bylo možné (z aktuálního uzlu již

nevede hrana do uzlu reprezentujícího další znak na vstupu) a poté se vrátil příznak spojený s tímto uzlem.

Jelikož však AL obsahuje víceznakových symbolů velmi málo (přesněji čtyři, a sice `:=`, `<=`, `>=`, `<>`) a všechny pouze o délce dva, používáme jiný přístup využívající možnost nahlédnout do vstupního proudu znaků na jeden znak dopředu v metodě zpracovávající jednoznakové symboly. V případě, že dostane ke zpracování první znak nějakého dvouznakového symbolu, ověří, zda další znak v proudu odpovídá druhému znaku symbolu a je-li tomu tak, přečte i tento druhý znak a vrátí příslušný token.

Další funkčnost, kterou `Tokenizer` vyšším vrstvám překladače nabízí, tkví v potřebě náhledu více tokenů dopředu (*look-ahead*) při zpracování zdrojového textu v pseudokódu. Pokud například syntaktický analyzátor získá informaci o tom, že první token příkazu je identifikátor, nemůže vědět, jestli překládaný příkaz bude přiřazení do proměnné, přiřazení do pole nebo volání procedury. Bylo by sice možné tuto nejednoznačnost odstranit formální úpravou gramatiky a místo pro člověka čitelnějšího

```
příkaz    =    přiřazení_do_proměnné |
               přiřazení_do_pole     |
               volání_procedury
```

přepsat a otrocky zjednoznačnit, vlastně „vytknout" společnou část a vytvořit pro ni nový element gramatiky. Místo tohoto přístupu jsme zvolili pro vyšší části překladače pohodlnější alternativu, kdy mohou nahlédnout na příští token pomocí metody `peek()`. Pokud potřebují získat informaci o větším počtu lexikálních elementů, mohou je po vyžádání od `nextToken()`vrátit zpět pomocí `pushBack(Token)`. `Tokenizer` udržuje zásobník vrácených tokenů a umožňuje tak provádět syntaktickou analýzu s výhledem libovolné délky.

Hlavním významem třídy `Tokenizer` je, že zjednodušuje činnost dalším (vyšším) částem překladače tím, že významně zmenší počet objektů, které musí tyto části rozpoznávat, neboť namísto stovek znaků zpracovávají o dva řády menší počet druhů tokenů.

Tokeny

Tokeny vracené metodou `Tokenizer.nextToken()` jsou instance třídy `Token` mající údaje o pozici ve zdrojovém textu a samozřejmě o typu lexikálního elementu, jenž zapouzdřují. V závislosti na typu tokenu lze zjistit jeho hodnotu voláním některé z metod `sval()`, `dval()`, `intval()` lišících se návratovým typem (`String`, `double`, `int`).

Zajímavou alternativou implementace tokenů se jeví možnost reprezentovat tokeny různých typů jinými třídami, potomky nějakého abstraktního tokenu. Tyto třídy by potom nemusely obsahovat všechny výše zmíněné metody pro zjištění hodnoty tokenu, ale vždy jen tu, která je pro daný typ přirozená. Rozpoznání druhu tokenu by pak mohlo být realizováno buď pomocí RTTI (běhová identifikace typů) a operátoru **instanceof** nebo stejně jako dosud pomocí metody `getType()`. Nikoliv však pomocí na první pohled lákavější varianty *statické* metody, jakkoliv to vypadá přirozeně, neboť přece všechny instance třídy příslušející danému typu tokenu musí

vracet stejnou hodnotu. Toto řešení znemožňuje realizovat vlastnost Javy, jež statickým metodám, na rozdíl od běžných metod, nedovoluje býti virtuální (ukazuje-li reference typu předka na potomka a zavolá se skrze tento odkaz metoda, je použita její verze z předka). A tuto vlastnost nezbytně potřebujeme, neboť metoda `Tokenizer.nextToken()` musí vracet tokeny jako reference na předka. Ať už bychom zjistili typ tokenu jakýmkoliv způsobem, museli bychom jej pro zjištění jeho lexikální hodnoty přetypovat na typ příslušného potomka a poté zavolat metodu, pomocí níž daný potomek vrací obsah tokenu.

Celkově je tedy použitý přístup daleko úspornější z hlediska délky zdrojového kódu (stačí jen jediná třída) i z pohledu jednoduchosti používání ve vyšších částech překladače, což je podstatnější argument.

3.3 Syntaktická analýza

V situaci, kdy je překládaný program v AL převeden na posloupnost tokenů, překladač zjišťuje, zda vstupní slovo patří do jazyka generovaného gramatikou, a v případě, že tomu tak je, sestrojí objektovou reprezentaci programu (derivační strom). V opačném případě zahlásí program chybu s označením příslušné pozice v pseudokódu, kde k ní došlo. V AAnimu probíhá tato fáze paralelně s lexikální a sémantickou analýzou, jedná se o *syntaxí řízený překlad*. O sémantické analýze, která zahrnuje např. typovou kontrolu, lze více informací získat v práci kolegy Římského, v kapitole o interpretu.

Zvolené řešení používá metodu rekurzivního sestupu, kdy si proud tokenů předává soustava vzájemně volajících se procedur, v níž každá zpracovává jeden neterminál gramatiky. Toto řešení bylo zvoleno na základě kombinace několika příčin. Jednak je tato metoda poměrně dobře implementovatelná. Dalším argumentem pro byla snaha překladač vytvořit samostatně a nepoužívat tak dostupné nástroje pro generování překladačů z poskytnuté gramatiky (javacc, ANTLR, ...). Důvodem, proč jsme se rozhodli nekonstruovat zásobníkový automat, zase byla znalost použití pseudokódu: zapisují se v něm jednoduché, krátké algoritmy, a proto vytváření nějaké složitější formy překladu by bylo (pro AAnim) zbytečné. Tento názor se zpětně potvrdil, neboť překlad zdrojového textu trvá zlomky vteřiny a při spouštění algoritmu je to operace nepostřehnutelná.

Parser a jeho potomci

Základ všech syntaktických analyzátorů (*parserů*) pseudokódu tvoří abstraktní třída `Parser`. Všechny parsery přetěžují jednu její důležitou metodu, `getResult()`, jež iniciuje analýzu syntaktického elementu a vrací objektovou reprezentaci neterminálu, který daný parser zpracovává. V případě syntaktické resp. sémantické chyby tato metoda vyhodí výjimku `ParsingException` resp. `ETypeCheck`.

Typický parser je tedy objekt, potomek třídy `Parser`. V konstruktoru dostává vstupní proud tokenů (reprezentovaný třídou `Tokenizer`), a dále modul a tabulku typů (ty jsou využívány při sémantické analýze). Návratový typ se liší podle zpracovávaného elementu gramatiky. Pokud se analyzuje výraz, vrací se instance třídy `Expression`, pokud příkaz, je to `Command`, a jestliže se jedná o deklaraci, má vrácený objekt typ `VariableDeclaration`.

Zvolený přístup k objektovému návrhu parserů preferuje jejich jednoduchý a kompaktní tvar. Další pojetí, jež jsme zvažovali, se blížilo k tradičnímu „procedurálnímu“, totiž mít jednu velkou třídu, která by vlastnila proud tokenů a namísto objektů (parserů) by používala metody. V tomto případě by odpadla nutnost vytvářet při překladu nové objekty a předávat parametry jejich konstruktorům, ale výsledná třída by byla obrovská a těžko spravovatelná. Mnohdy si také parser definuje vlastní pomocné metody, které by byly sdíleny celou třídou, což by dále přispívalo k nepřehlednosti.

Naopak volba mechanismu hlášení chyb při překladu velmi přirozeně vyplynula z prostředků nabízených jazykem Java. Požadavek přerušení normálního toku programu a návrat k nějakému centrálnímu místu ošetření splňují výjimky a princip jejich šíření. Toto plně dostačuje pro hlášení chyb tak, jak je prováděno v současné verzi, kdy se po první objevené chybě překlad přeruší a vypíše se příslušná informace. Jedno uvažované rozšíření překladače počítá právě s výpisem všech kompilačních chyb. To by však vyžadovalo opuštění mechanismu výjimek a jeho nahrazení nějakým vlastním řešením, které by nepřerušilo překlad a místo toho by se snažilo z chybového stavu zotavit.

3.4 Rozhraní překladače

Pro interpret, který výstup překladače využívá, není podstatná zvolená metoda lexikální či syntaktické analýzy, a proto může být od nich zcela odstíněn. Komunikační rozhraní pro části programu, které chtějí využít služeb překladače, reprezentuje třída `Translator`.

Hned v konstruktoru se jí předá vstupní znakový proud určený k překladu a tamtéž se také okamžitě spustí překlad: načtení deklaračních sekcí a kompilace hlavního bloku programu. Vyskytne-li se při překladu chyba, konstruktor vyhodí patřičnou výjimku. Je tedy povinností volajícího ošetřit formu chybového výstupu. Pokud překlad proběhne v pořádku, dojde k naplnění vnitřních struktur třídy `Translator` výsledky kompilace (tabulka deklaračních informací, objekt `Block` představující hlavní blok programu), které lze získat voláním některé z příslušných metod `getTypes()`, `getVariableDeclarations()`či `getBody()`.

Kapitola 4

Data algoritmu

V této části popíšeme, jaká data doprovázejí samotný kód algoritmu a jakými prostředky jsou v AAnimu spravována.

4.1 Přidružené datové soubory

Kromě souboru obsahujícího zdrojový kód programu v pseudokódu mohou být součástí onoho celku, který se uživateli jeví po otevření AAnimem jako algoritmus, další, podpůrné, datové soubory. Může se jednat o

- soubory obsahující vstupní sady proměnných
- konfigurační soubory animátorů.

Sady proměnných

V prostředí AAnimu by měl student poznávat a zkoušet si sám upravovat nové algoritmy. Abychom mu poskytli co nejvíce nástrojů, které mohou pomoci v jeho snažení, nabízíme mu možnost exportovat (a importovat) hodnoty proměnných v libovolném okamžiku průběhu algoritmu do souboru (a ze souboru). Vzniká tak potřeba pracovat s pojmenovaným souborem hodnot proměnných, *sadou proměnných*. Tato funkce s sebou přinesla požadavek na typy použité v pseudokódu, potomky třídy `PCType_general`, uložit hodnotu daného typu do řetězce a naopak rekonstruovat hodnotu z řetězce (při načítání proměnné). Uložení hodnot v textové podobě jsme zvolili kvůli lepší čitelnosti souboru se sadou proměnných i mimo prostředí AAnimu (viz jednoduchou vstupní sadu v následujícím příkladu).

Příklad 4.1.1 Soubor vstupní sady pro algoritmus vnitřního třídění. Obsahuje iniciální hodnotu pro vstupní proměnnou `a` typu pole celých čísel.

```
#Krátká posloupnost
input
a [5, 8, 5, 3, 8, 9, 0, 4, 9]
```

Sada proměnných je reprezentována třídou `VarSet`, se kterou pracuje správce sad proměnných. Kromě tabulky přiřazující názvu proměnné její textovou reprezentaci

je součástí třídy i komentář usnadňující uživateli orientaci mezi více sadami téhož algoritmu. O správě sad proměnných jakož i dalších datových souborů příslušných k jednomu algoritmu viz odstavec o správě datových souborů algoritmu.

Speciálním případem sady proměnných je množina hodnot vstupních proměnných nazývaná *vstupní sada*. Vstupní proměnná je proměnná v pseudokódu, která je v algoritmu využívána podobně jako parametr v těle procedury, tedy jako vstupní hodnota, jež je v průběhu algoritmu dále využívána. V kódu se deklaruje promocí direktivy **input**. Po spuštění algoritmu, v němž jsou deklarovány vstupní proměnné, AAnim zobrazí dialog s možností výběru uložených vstupních sad.

Konfigurační soubory animátorů

Dalším podpůrným datovým souborem, který musí existovat, má-li být spuštěný algoritmus vizualizován animátorem, je konfigurační soubor daného animátoru. Na tomto místě jen poznamenám, že se jedná o posloupnost záznamů přiřazujících proměnným z pseudokódu jistou úlohu při animaci algoritmu, tzv. *roli*, a že je reprezentována třídou `AnimatorConfig`. Více informací o významu v konfiguračním souboru obsažených položek lze nalézt v kapitole Vizualizace.

4.2 Správa dat v AAnimu

V okamžiku, kdy jsme vytvořili koncept sad proměnných, museli jsme řešit otázku, jak se postavit k jejich správě. Uživatel si mohl (potenciálně) vybírat z mnoha sad, proto jsme mu chtěli poskytnout nějaký nástroj, který by mu to usnadnil. Vyvinul se z něj Správce sad proměnných, dialog zobrazující všechny dostupné sady umožňující dále s nimi pracovat (mazat, přidávat, upravovat existující, nastavovat výchozí vstupní sadu). Zatímco podoba správce a vyžadovaná funkcionalita byla téměř okamžitě jasná, otázka začlenění správy sad do objektového návrhu tak přímočaré řešení neměla a zvažovali jsme jich několik.

Jedna idea řešení umístila těžiště práce se sadami do jejich správce. Nechť se stará o jejich načítání a ukládání při editaci, což má jistě své opodstatnění, neboť jedině on takto k sadám přistupuje. Ovšem hodnoty ze vstupních sad je potřeba též načítat při inicializaci algoritmu a tedy přidejme správci i tyto schopnosti z důvodů úspory kódu (již umí sady načítat ze souboru, bude také umět předávat hodnoty proměnných prostředí).

V podobném záměru jsme spatřovali řadu nedostatků. Jednak v případě, že by správce sad v sobě spojoval funkci grafického editoru a zároveň objektu zpřístupňujícího hodnoty proměnných ve vstupních sadách dalším komponentám programu, jednalo by se o porušení zásad objektového programování vytvářet třídy s jasně a jednoduše definovaným úkolem se všemi svými negativními důsledky například v podobě komplikovaného, špatně udržovatelného kódu. Ale i kdyby došlo k oddělení těchto dvou rolí, byli jsme si vědomi faktu, že sady proměnných jsou jen další datovou entitou potřebnou pro fungování algoritmu (kromě zdrojového kódu). A pokud by se v budoucnosti přidávaly další podobné „doprovodné“ datové soubory a každý by měl nějakého svého samostatného správce, začala by být situace nepřehledná. Rozhodli jsme se proto přístup ke správě dat algoritmu centralizovat v podobě třídy `AlgorithmKit`.

Tento obecnější přístup se osvědčil, když jsme o několik měsíců později rozšiřovali možnosti animátorů o mechanismus rolí, což si vyžádalo existenci konfiguračních souborů. Díky tomuto jednotnému přístupu stačilo však už jen systematicky rozšířit `AlgorithmKit` přidáním několika metod pro práci s konfiguracemi.

AlgorithmKit — správce dat algoritmu

Tato třída představuje rozhraní, skrze nějž mohou ostatní programové komponenty AAnimu přistupovat k algoritmu přidruženým datům. Tento návrh přináší několik výhod.

- Centralizace správy dat soustředí veškerou práci s externími datovými soubory do jednoho místa v programu. Kód se stává lépe spravovatelným a snadněji se tedy provádějí změny.
- Umožňuje odstínit všechny další součásti programu od implementačních detailů uložení datových entit, mezi které patří například konvence pojmenování souborů či jejich formát.

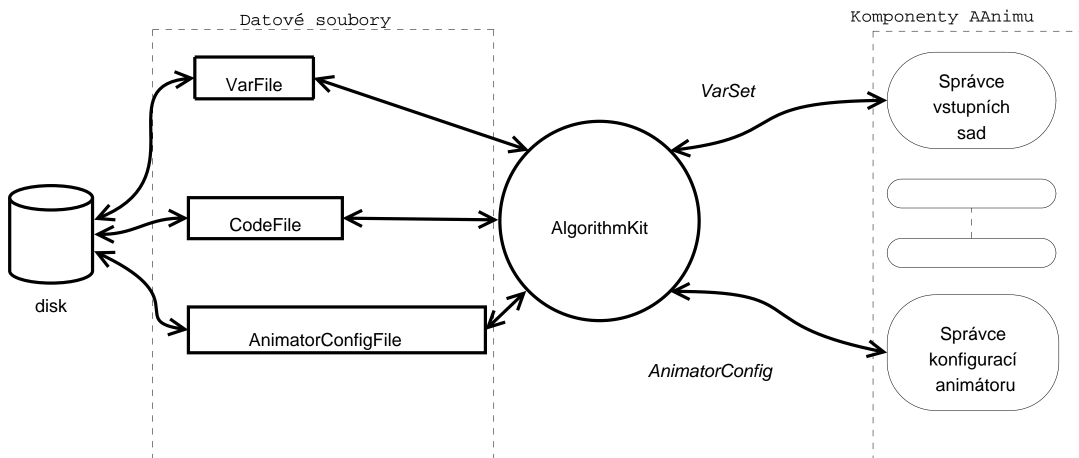
Odstranění závislostí dalších částí aplikace na formátu uložení je realizováno následovně. Každá komponenta AAnimu pracující s nějakou datovou entitou (správce sad proměnných, správce konfigurací, GUI aplikace při ukládání či načítání zdrojového kódu) operuje nad datovým objektem, který zcela abstrahuje od detailů svého uložení a pouze poskytuje logický obsah. Takto například sada proměnných (třída `VarSet`) poskytuje metody, pomocí nichž lze přečíst nebo zapsat název sady, komentář k ní, příznak jedná-li se o sadu vstupní a tabulku přiřazující názvu proměnné její hodnotu. Ale to, že název sady odpovídá názvu souboru, v němž je uložena a že tento má jméno odvozené od názvu souboru se zdrojovým kódem, žádná komponenta programu mimo správce dat netuší.

`AlgorithmKit` tedy navenek pracuje s abstraktními datovými objekty. Další vrstvou pro práci s nimi, mimo `AlgorithmKit` již neviditelnou, jsou datové soubory. Jedná se o třídy, potomky třídy `SmartFile` (představuje soubor a stará se o logiku operací s ním jako např. otevření či uložení, více o ní generovaná dokumentace), které realizují ukládání a načítání ze souborů obsahujících ony od detailů uložení abstrahované datové entity a které tyto objekty poskytují správci dat. Jasnější představu o tom, jak tento řetězec tříd funguje, snad dá schéma na obrázku 4.1.

4.3 Formát datových souborů

Všechny datové objekty připojené k algoritmu jsou ukládány v textovém formátu do samostatných souborů do stejného adresáře, v němž se nachází soubor obsahující pseudokód. Textovou podobu uložení jsme zvolili, abychom umožnili uživatelům prohlížet si obsah těchto souborů i mimo AAnim a v nutném případě je tam i upravovat. Tyto připojené soubory musí dodržovat jmennou konvenci, v níž je jejich jméno odvozeno od názvu souboru s pseudokódem. Správce dat pak vždy, když je dotázán na nějaké datové objekty, prohledá adresář a na základě názvů v něm obsažených souborů tyto objekty vyhledá.

Obrázek 4.1 Schéma přístupu k datovým entitám



Konkrétní konvence zde nebudeme zmiňovat (více viz dokumentace). V této části chceme spíše demonstrovat univerzálnost použití správce dat na příkladu poměrně podstatné změny formátu a způsobu uložení datových souborů, která však neovlivní ostatní komponenty programu.

Zvolený způsob uložení totiž není ideální. Při větším počtu připojených souborů může, pokud je navíc v adresáři umístěno více algoritmů, být obsah dané složky značně nepřehledný. I řádkově orientovaný formát datových souborů má své nevýhody: záleží například na pořadí řádků, a tak může při neopatrné editaci externím editorem snadno dojít k poškození obsahu souboru. Proto mezi jedno z uvažovaných rozšíření AAnimu patří uložení všech s algoritmem spojených datových entit do jednoho strukturovaného souboru. Velmi dobře by tomuto účelu vyhovoval populární jazyk pro popis dat XML (*Extensible Markup Language*). Pokusíme se nyní ukázat, jak relativně snadno lze tuto změnu uskutečnit.

Namísto nějaké striktní definice přípustných elementů a atributů, stačí nám pro představu uvést příklad, jak by mohl takový XML dokument nesoucí datové objekty algoritmu vypadat.

```
<?xml version="1.0"?>
<algorithm name="Název algoritmu">
  <program>
    <!-- část obsahující zdrojový kód -->
  </program>

  <varsets>
    <varset name="Vstupní sada 1" input="true">
      <variable>
        <name>nazev promenne</name>
        <value>10</value>
      </variable>
    </varset>
  </varsets>

  <animconfigs>
```

```
    <animator name="anim1">
      <animconfig name="Konfigurace 1 pro anim1">
        <!-- obsah konfigurace -->
      </animconfig>
    </animator>
  </animconfigs>
</algorithm>
```

Knihovny jazyka Java poskytují nástroje pro práci (čtení i úpravy) s XML soubory. Jeden z přístupů k manipulaci dat uložených v XML se nazývá DOM (*Document Object Model*), kdy (zhruba řečeno) se XML dokument jeví jako strom s uzly odpovídajícím elementům. Tohoto mechanismu by využívala třída, která by datový soubor obalovala. Zastupovala by potomky třídy `SmartFile` (viz obrázek 4.1) a disponovala metodami, které by už pracovaly s na uložení nezávislými datovými objekty (`VarSet`, `AnimatorConfig`). Ve správci dat by se potom pouze změnila, a nutno říci značně zjednodušila, těla většiny metod, neboť by více méně volala metody poskytované oním objektem obalujícím XML dokument.

Už z tohoto nepříliš podrobného nástinu je jasné, že ostatní komponenty programu by se vůbec měnit nemusely. Drtivou většinu programátorské práce by zkonzumovala konstrukce třídy obalující XML dokument, což je pochopitelné, avšak nutné, chceme-li takovou funkčnost.

Kapitola 5

Vizualizace

V této kapitole se budeme zabývat nástroji pro snazší pochopení nějakého algoritmu, které AAnim uživateli nabízí. Průběh algoritmu lze charakterizovat posloupností hodnot všech v něm použitých proměnných v čase (po krocích programu). Obecně tedy úkol vizualizace průběhu algoritmu spočívá ve výběru „důležitých" proměnných a v jejich vhodném vykreslení. AAnim používá v zásadě dva druhy zobrazovacích technik, jež se liší tím, jakou část z předchozího tvrzení více naplňují. Jedná se o

- tabulku proměnných a zobrazovače typů

  Tyto zobrazují hodnoty *všech* v programu definovaných proměnných (v případě zobrazovačů těch, o něž si uživatel požádá) bez nějaké „znalosti" o významu a starají se o *vhodné* vykreslení jejich obsahu. Což u tabulky proměnných znamená prostě zobrazení textové reprezentace hodnoty a v případě zobrazovačů vykreslení téhož v nějaké člověku snáze pochopitelné formě.

- animátory

  Tento nástroj se naopak snaží využít „znalosti" vizualizovaného algoritmu (nebo alespoň třídy podobných) k tomu, aby tu samou informaci (aktuální hodnoty proměnných) zobrazil nějakým pro uživatele pochopitelnějším způsobem. Při tomto se může selektivně zaměřit na nějakou konkrétní stránku algoritmu (úhel pohledu), nebo zvolit podobu zobrazovaných dat velmi speciální, kterou by si obecné zobrazovače typů vůbec nemohly dovolit.

5.1 Zobrazovače a editory typů

Zobrazovače typů

Primární entitou, která uživateli poskytuje aktuální hodnoty všech proměnných deklarovaných v algoritmu, je tabulka proměnných (viz uživatelskou dokumentaci). Zobrazuje textovou reprezentaci hodnot proměnných (to je mimochodem důvod, proč musí typy přetěžovat metodu `toString()`). Představuje obdobu ovládacího prvku známého z různých grafických debuggerů. Některé typy jsou ovšem dosti složité nebo jejich instance obsahují příliš mnoho dat, a tak se nemůžou vměstnat do prostoru jediné textové buňky, jež tabulka proměnných nabízí. Pro tyto případy je

AAnim vybaven mechanismem, který rozšiřuje zobrazovací schopnosti tabulky proměnných tím, že zobrazuje okno vykreslující hodnotu vybrané proměnné v nějaké (obvykle již netextové) člověku srozumitelné reprezentaci. Jedná se o zobrazovače (*renderery*) typů.

Tyto třídy implementují jednoduché, obecné rozhraní `TypeRenderer`.

```
public interface TypeRenderer extends VarsChangeListener{
    void render(VarTableModel vtm, String var);
    boolean isShown();
    public void discard();
}
```

Obsahuje hlavně dvě důležité metody:

- `render()`, jež je volána tabulkou proměnných, chce-li uživatel zobrazit renderer, a jež mu sdělí název vizualizované proměnné a v objektu typu `VarTableModel` mimo jiného i prostředí, od něhož potom bude dostávat hodnoty proměnné.

- `varsChanged()`, která je zděděná od rozhraní `VarsChangeListener` a v které je renderer prostředím informován, že hodnota jím sledované proměnné se změnila.

Zbylé metody `isShown()` a `discard()` slouží potom spíše pro potřebu správce zobrazovačů (více viz práce kolegy Římského), první informuje, zda je okno rendereru zobrazené, druhou se pak zobrazovači sděluje, aby zmizel (skryl své okno).

Jak patrno již z názvu, jsou zobrazovače typů svázány s typem, který zobrazují. Typ si s sebou nese svůj zobrazovač (vrací jej metodou `getRenderer()`).

Editory typů

Ačkoliv se nejedná o nástroj pro vizualizaci, zařadíme do této kapitoly i krátký paragraf o editorech typů, protože se v mnohém podobají zobrazovačům typů.

Jsou totiž zcela analogickým řešením potřeby tabulky proměnných měnit hodnoty proměnných jinak než textově u složitějších typů. Po iniciaci editace se zobrazí okno sofistikovanějšího editoru typu, třídy, jež se zavazuje implementovat prosté rozhraní `TypeEditor`.

```
public interface TypeEditor {
    public void edit(VarTableModel vtm, String var);
}
```

Alternativní návrh

Závěrem pasáže o zobrazovačích a editorech si dovolíme opět jeden malý exkurz do historie vývoje AAnimu. Jeden z původních návrhů, jak řešit otázku rozšíření zobrazovacích a editačních schopností tabulky proměnných, se podobal vylepšení textové buňky v tabulce. Předpokládal totiž, že se po iniciaci z tabulky objeví okno, které uživateli zprostředkuje zobrazení i editaci hodnoty vybrané proměnné. Převedeno

do řeči objektového programování, místo objektu implementujícího nějaká rozhraní, jednalo by se o potomka třídy `java.awt.Window`, který by byl schopen hodnotu proměnné zobrazit i změnit. Od tohoto návrhu se tedy udály dvě hlavní změny, které mechanismus zjednodušily a zároveň zobecnily.

1. Došlo k oddělení funkcí zobrazení a úpravy proměnné. Což stále umožňuje mít jeden objekt provádějící obě tyto činnosti, stačí mu jen implementovat obě rozhraní. Na druhou stranu, tím že se toto sloučení nepožaduje, může dojít k výraznému zjednodušení návrhu obou částí.

2. Namísto požadavku, *čím* má renderer či editor být (potomek třídy `Window`), chce se nyní po nich jen, *co* mají dělat. Tedy při zachování rozhraní `TypeRenderer` by se mohly zobrazovat (po dotvoření patřičných mechanismů) například jako záložky v nějakém centrálním správci zobrazovaných proměnných, nikoli pouze jako jednoduchá okna.

5.2 Animátory

Co jsou animátory

Z úvodu této kapitoly již víme, že animátor je jedním z vizualizačních nástrojů AAnimu a že na rozdíl od zobrazovačů disponuje určitými znalostmi o povaze algoritmu, který animuje, a může proto užité proměnné zobrazovat v dosti speciálních podobách (např. pole celých čísel jako haldu, stromovitou strukturu). Animátory jsou tedy vizualizačními moduly, o něž může být AAnim rozšiřován podobně jako o moduly obsahující funkce, procedury a typy, které se mohou použít v pseudokódu (viz práci kolegy Římského).

Techničtěji řečeno, animátory jsou objekty, potomci třídy `General`, které jsou programem načteny po spuštění algoritmu a v jeho průběhu dostávají od prostředí informace o změnách hodnot proměnných (implementují stejně jako zobrazovače nám již známé rozhraní `VarsChangeListener`). V reakci na tyto zprávy si zjistí hodnoty pro animaci důležitých proměnných a následně překreslí svůj panel v pravé části hlavního okna aplikace (více viz uživatelskou nápovědu, sekci o GUI).

Panel animátoru (potomek třídy `JPanel`) si AAnim od animátoru vyžádá voláním metody `getPanel()`. Je tedy vytvářen samotným animátorem, nikoli aplikací, což dovoluje, aby události iniciované uživatelem nad oblastí panelu byly zpracovávány metodami animátoru, a obohacuje tak animátor o možnost interakce.

Mechanismus rolí

Již několikrát jsme použili poměrně vágního tvrzení, že animátory mají nějaké „znalosti“ o vizualizovaném algoritmu. To v podstatě znamená, že jejich autor počítal při návrhu s tím, že animátor bude vizualizovat nějaký určitý algoritmus nebo třídu podobných. A ony významově neznámé proměnné, na jejichž změny jej bude prostředí upozorňovat, získají najednou pro animátor nějaký smysl, začnou vystupovat v roli objektů, s nimiž algoritmus na konceptuální úrovni pracuje. Jak však svázat informaci o tom, která proměnná má v algoritmu jakou úlohu?

Velmi jednoduchým řešením, které se dokonce v raných verzích AAnimu, kdy se vývoj soustředil na jiné komponenty, používalo, je přímo do kódu animátoru explicitně zadat, na hodnoty jakých proměnných se má dotazovat. To je pro aplikaci, jež se snaží co nejvíce podnítit studenty k experimentování s algoritmy, takřka nepoužitelné. Stačí totiž pouhá změna názvu proměnné v pseudokódu a animátor se bude v lepším případě dotazovat po neexistující proměnné, což vyvolá výjimku a zastaví provádění algoritmu, v horším případě, kdy by uživatel používal proměnnou téhož názvu ale v jiném významu, by animace nedávala pozorovateli smysl a její pedagogický přínos by byl spíše záporný. Dalším negativním důsledkem byla nemožnost vytvořit obecnější animátor použitelný pro více podobných algoritmů, neboť díky pevnému svázání se jmény proměnných byl každý animátor závislý dokonce na jedné konkrétní implementaci.

Řešení otázky, jak se zbavit závislosti na pojmenování proměnných a jak svázat informaci o tom, která proměnná má v algoritmu jakou úlohu, je velmi přirozeným zobecněním oné první, naivní myšlenky. Animátor již nebude pracovat se jmény proměnných, ale s konceptuálními *rolemi*, které tyto proměnné v algoritmu hrají. Oddělením přiřazení rolí proměnným od implementace animátoru vznikne konfigurační soubor unikátní pro každou dvojici algoritmus – animátor, na který je vlastně přenesena ta úplná závislost animátoru na jedné implementaci algoritmu kritizovaná v předchozím odstavci.

Tato separace má ještě jeden pozitivní důsledek, zvyšuje konfigurovatelnost animátoru. Využili jsme záznamů konfiguračního souboru, abychom rozšířili samotné přiřazení názvu proměnné roli o možnost navázat na sebe další atributy (např. barva, popisek), které ovlivní vzhled či chování animace.

Role je tedy logická úloha, kterou proměnná v algoritmu může vykonávat, doplněná o další atributy. Animátor pak namísto s proměnnými pracuje s instancemi rolí.

Implementace mechanismu rolí

Role jsou přirozeně implementovány jako třídy, potomci abstraktní třídy `Role`, jejíž (pro účely této práce upravený – odstraněny těla metod) zdrojový kód vypadá takto:

```
abstract public class Role {
    /** jméno proměnné, jejíž úlohu instance této třídy určuje */
    protected String varName;

    public Role(){}
    public String getVarName(){}
    public void setVarName(String varName){}
    public Role(aanim.modules.PCType_string varName){}

    /** Metody využité editorem konfigurací */
    public String getName(){}
    public String getDescription(){}
    public String[] getOwnAttributeNames(){}
}
```

Příklad 5.2.1 Příklad konfiguračního souboru, který přiřazuje role využívané animátorem zobrazujícím pole jako sloupečky (`colarray`) proměnným z algoritmu vnitřního třídění výběrem. Záznamy jsou odděleny řádkem s třemi pomlčkami. Na prvním řádku záznamu je název role, na druhém název proměnné a na dalších ostatní atributy dané role.

```
colarray.Array
a
---
colarray.ArrayIndex
minindex
Index minimálního prvku pole
0
---
colarray.ArrayIndex
j
j
13045
---
colarray.ArrayIndex
i
i
45785
---
```

Vidíme, že jediným atributem je jméno proměnné, které je přijímáno konstruktorem a manipulují s ním metody `getVarName()` a `setVarName()`. Další metody jsou využívány editorem konfigurací, o němž se zmíníme v další části, a pro animátor nenesou žádnou podstatnou informaci. Každý animátor musí vytvořit potomky třídy `Role` nebo využít role vytvořené jinými animátory, které chce používat. V těchto potom dohodnutým způsobem může dodefinovat jejich další atributy. S vytvářením rolí a animátorů je spojeno mnoho konvencí, které jsou blíže popsány v Návodu pro autory animátorů (v nápovědě AAnimu).

Nyní si raději přiblížíme, jakým způsobem se informace obsažené v konfiguračním souboru k animátoru dostanou. Děje se tak při spuštění algoritmu. Grafické prostředí si nejprve od správce dat algoritmu (`AlgorithmKit`) vyžádá výchozí konfiguraci pro uživatelem vybraný animátor. Ten jí vrátí již na zvoleném formátu uložení konfiguračního souboru nezávislý objekt typu `AnimatorConfig`, který obsahuje posloupnost záznamů, v nichž se dané roli přiřazuje název proměnné, jež bude v této roli v algoritmu vystupovat, a event. další atributy s rolí spojené. Pro lepší představu uveďme příklad: proměnné `i` se přiřazuje role *Ukazatel do pole* a atribut *Popisek* s hodnotou "Nejmenší prvek".

V tomto okamžiku zavolá grafické prostředí metodu předka všech animátorů, třídy `General`, `readConfig()`, v níž mu předá získanou konfiguraci. V metodě se z textových záznamů konfiguračního souboru stávají instance jednotlivých rolí. K tomu opět jako v mnoha jiných komponentách AAnimu používáme *Java Reflection API*, název role v konfiguračním souboru totiž odpovídá názvu třídy a atributy role jsou použity jako parametry konstruktoru. Poté jsou role (jako objekty) předány

metodě `addRole()`, již každý animátor musí přetížit a postarat se o zpracování předaných rolí. Obvykle si role uloží do svých datových struktur a v průběhu algoritmu pak k získání hodnoty proměnné, kterou tato role reprezentuje, užívají metodu `getVarVal()`.

Editor konfigurací animátoru

Zatím jsme úplně pominuli otázku, jak vytvářet soubory konfigurací. Původně jsme je vytvářeli ručně a jejich existence byla uživatelům AAnimu skrytá. Posléze jsme uživatelům chtěli alespoň umožnit prohlížet si, jaké mají proměnné z algoritmu přiřazené role. Zároveň jsme si byli vědomi skutečnosti, že ruční vytváření souborů konfigurací je značně nepohodlné, a proto jsme se rozhodli vytvořit komfortnější nástroj pro práci s konfiguracemi — editor konfigurací animátoru.

Stručně nyní pro představu popíšeme, jak editor konfigurací funguje. To ale není vůbec cílem, spíše chceme ukázat, jak editor ovlivnil objektový návrh rolí a animátorů. Editor je přístupný ze Správce konfigurací animátoru. Zobrazí se po té, co uživatel začne upravovat nějakou existující konfiguraci nebo vytvářet novou. Nejdůležitější jeho částí je tabulka, jejíž řádky odpovídají záznamům v souboru konfigurací (zobrazují název role a jméno proměnné v algoritmu, které je role přiřazena). Po kliknutí na ikonu lupy se zobrazí dialog, kde může uživatel nastavit jednotlivé atributy role. Uživatel může přidat nový záznam (zobrazí se mu nejdříve seznam animátorem využívaných rolí a po vybrání jedné z nich i její editační dialog) i smazat již existující.

Co činí editor konfigurací speciálním například vůči editoru sad proměnných je fakt, že objekty, které upravuje nebyly v čase překladu známy. Díky modularitě animátorů (a tím pádem i rolí) musí si informace o rolích získávat až za běhu a dialogy, jež je upravují, také konstruovat dynamicky. To si vyžádalo úpravu objektového návrhu rolí i animátorů. Rolím oproti původní podobě přibyly metody `getName()`, `getDescription()`a `getOwnAttributeNames()`, které vrací název, popis a názvy atributů role. Animátory musely být rozšířeny o metodu `getExportedRoles()` vracející jimi užívané role.

Na příkladu posledně jmenované metody lze také ilustrovat, jak nesnadným úkolem je zavádění i zdánlivě banálních konvencí. Původně jsme totiž měli představu, že každý animátor přetíží tuto metodu a bude v ní vracet seznam užívaných rolí. To ovšem autory animátorů zatěžovalo vyplňováním těla této metody. Společně s faktem, že většinou animátor definuje své vlastní role, které také používá pouze on sám, nás to ve snaze ulehčit autorům animátorů přimělo zavést novou konvenci: animátor si bude role definovat ve svém (podle konvence pojmenovaném) balíčku. Poté jsme metodu `getExportedRoles()` s využitím vlastnosti jazyka Java, že balíček existuje na disku jako nějaký adresář a třídy v něm jako soubory, a opět *Reflection API*, implementovali ve společném předku animátorů tak, že vracela role umístěné v balíčku animátoru. Chybu, které jsme se přitom dopustili, jsme odhalili, až když jsme chtěli přidat animátor využívající roli, kterou sám nevytvořil, a přetížili jsme metodu `getExportedRoles()`. S údivem jsme potom hleděli na hlášku překladače, v níž nám sděloval, že tuto metodu přetížit nelze. Metodu jsme totiž deklarovali jako `final` ve snaze uchránit autory animátorů před přepisováním jejího těla, ale byli jsme až příliš restriktivní.

Vylepšení

Největším kandidátem na další zlepšování je editor konfigurací. V současné verzi se ještě nevyužívá jeho plného potenciálu, neboť se při úpravách konfigurací neprovádí téměř žádné kontroly vstupů. Zejména by bylo možné a vhodné testovat (v některých případech by to vyžadovalo rozšíření třídy `Role` o nové metody) existenci a typovou správnost přiřazení role nějaké proměnné. Další užitečnou pomůckou by mohla být kontrola správného počtů instancí rolí v konfiguračním souboru. Například animátor B-stromu potřebuje mít v konfiguraci právě jednu roli B-strom, což v současné verzi není při úpravách jeho konfigurace vynucováno.

Kapitola 6

Závěr

Vývoj programu AAnim byl pro mě cennou zkušeností. Především proto, že jsem poprvé vytvářel softwarové dílo takového rozsahu a po tak dlouhou dobu (více než jeden rok).

Dalším novým prvkem oproti mým dosavadním projektům byl fakt, že jsme AAnim vytvářeli v týmu (s Pavlem Římským), což s sebou přineslo mnohé pro návrh aplikace pozitivní důsledky: důkladnější promýšlení návrhů vynucené již samotnou potřebou snadno a srozumitelně je vysvětlit v diskuzi, tvorba co nejjednodušších rozhraní mezi komponentami vytvářenými různými autory či nutnost kvalitnější dokumentace „styčných" částí programu. Mnohdy také vzešlo z vzájemné diskuse řešení lepší než obě konfrontované varianty.

Společný vývoj vyvolal také potřebu synchronizovat vytvářený kód, spojovat po úpravách verze obou autorů. Zvolili jsme technicky jednoduchý avšak ne příliš elegantní a pro větší počet programátorů již stěží fungující způsob „skokového slévání", kdy po větší změně poslal její autor druhému novou verzi programu a podnikl-li mezitím úpravy i druhý, slil (většinou jsme používali program WinMerge) je do nové verze, která se stala základem pro další „vývojové kolečko". Dostalo se nám tak poučení, že je vhodné, i za cenu počáteční vyšší časové investice, zřídit nějaké centrální úložiště se sofistikovanější správou verzí (např. CVS či SVN).

V neposlední řadě byla pro mě nová a obohacující skutečnost, že jsme se v raných fázích vývoje podíleli v interakci s RNDr. Krylem na vytváření specifikace funkcí a podoby AAnimu. Nebyli jsme postaveni před nějakou pevnou a neměnnou specifikaci, ale měli jsme spíše určen základní směr: vytvořit pomůcku pro studenty podporující formou vizualizací jejich aktivní pochopení vybraných algoritmů.

Dodatek A

Zodpovědnost za jednotlivé části AAnimu

V této příloze je shrnuto, kdo z autorů je zodpovědný za které části projektu AAnim. Najdete tu informace o autorství jednotlivých komponent AAnimu, modulů a animátorů. Na některých komponentách pracovali oba studenti společně, v takovém případě bude tato informace podrobněji rozvedena.

A.1 Základní komponenty AAnimu

Komponenty vytvořené Pavlem Římským

- editor kódu (celý balíček `aanim.codeedit`)
- třídy reprezentující prostředí, v němž algoritmus běží (celý balíček `aanim.environment`)
- podstatná část grafického uživatelského rozhraní
  - téměř celý balíček `aanim.gui` (Petr Štěpán v něm prováděl změny pouze tehdy, pokud to bylo nutné, aby se projevily některé úpravy, které prováděl ve svých komponentách)
  - tabulka proměnných (balíček `aanim.vartable`), drobné úpravy v rodičovských třídách editorů a rendererů
  - balíček `aanim.utils`, až na dvě metody ve třídě `CGU`
- správa modulů (celý balíček `aanim.moduleman`)
- správa balíčků AAnim (celý balíček `aanim.packageman`)
- třída `aanim.smartfile.SmartFile`
- sémantická analýza a interpretace derivačního stromu (třídy `Command` a `Expression`, Petr Štěpán v nich podnikl drobné úpravy)

Komponenty vytvořené Petrem Štěpánem

- překladač – lexikální a syntaktická analýza (většina balíčku `aanim.translator`)
- jednotná správa dat algoritmu (balíček `aanim.algorithmkit`)

- mechanismus rolí v animátorech (`aanim.animators.Role`, související části v `aanim.animators.general.General`)
- správce konfigurací animátoru (balíček `aanim.animators.configedit`)
- správce sad proměnných (balíček `aanim.setman`)
- mechanismus zobrazovačů a editorů typů (balíček `aanim.datatypes`)

A.2 Moduly

Moduly vytvořené Pavlem Římským

- extsort
- general (spolu s Petrem Štěpánem)
- hashtable
- polyphase
- radixsort
- sachovnice

Moduly vytvořené Petrem Štěpánem

- btree
- general (spolu s Pavlem Římským)
- hashing
- heap
- pairstack

A.3 Animátory

Animátory vytvořené Pavlem Římským

- colarray (spolu s Petrem Štěpánem)
- hamming
- hashtable
- history
- polyphase
- radixsort
- rainbow
- tapes

Animátory vytvořené Petr Štěpánem

- btree
- chainhash
- colarray (spolu s Pavlem Římským)
- heap

- heaparray
- matrixmultiply
- merge