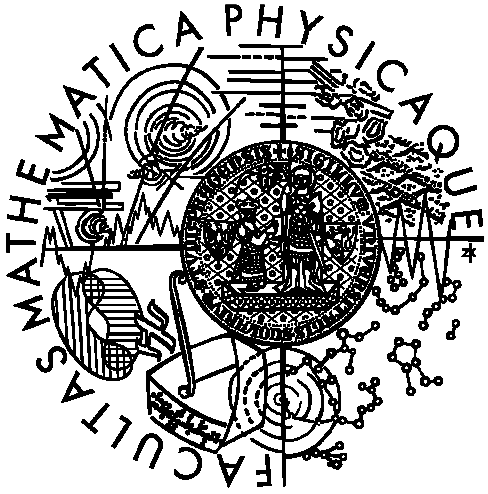
UNIVERZITA KARLOVA V PRAZE
MATEMATICKO-FYZIKÁLNÍ FAKULTA

DIPLOMOVÁ PRÁCE



JAN KRČEK

Jazyk pro řízení 2D her

Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. David Bednárek
Studijní program: Informatika, Softwarové systémy

Děkuji vedoucímu diplomové práce RNDr. Davidovi Bednárkovi za cenné rady a podněty, které přispěly k vytvoření této práce, a členům týmu Krkal za spolupráci na původní verzi. Také bych rád poděkoval rodině, přátelům a kolegům z MIS AG za podporu a trpělivost.

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

V Praze dne 10. srpna 2007 Jan Krček

Obsah

Seznam obrázků

Název práce: Jazyk pro řízení 2D her
Autor: Jan Krček
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. David Bednárek
e-mail vedoucího: David.Bednarek@mff.cuni.cz

Abstrakt: Práce se zabývá návrhem programovacího jazyka pro řízení tahových i real-time her s logickými či simulačními prvky, implementací překladače, komponenty běhové podpory a grafického uživatelského rozhraní, které slouží jak pro vývoj her, tak pro jejich běh a ladění.

Navrhovaný jazyk je určen pro popis systémů žijících a vzájemně se ovlivňujících objektů. Důraz je kladen na snadnou rozšiřitelnost, také na jednoduchost a bezpečnost. Jazyk obsahuje například vícenásobnou dědičnost, zprávy nebo speciální typ pro množiny. Primárním výstupem kompilace je zdrojový kód jazyka C++.

Klíčová slova: systém pro tvorbu her, kompilátor, vícenásobná dědičnost, simulace

Title: A controlling language for 2D games
Author: Jan Krček
Department: Department of Software Engineering
Supervisor: RNDr. David Bednárek
Supervisor’s e-mail address: David.Bednarek@mff.cuni.cz

Abstract: The thesis covers design of a programming language for controlling turn-based and real-time games with logical or simulative features; implementation of a compiler, a component for runtime support and a graphical user interface that serves both for game development and their running and debugging.

The designed language is intended for describing systems with living and communicating objects. Importance is given on easy extensibility, also on simplicity and safety. The language contains for instance multiple inheritance, messages and special type for sets. Primary output of the compiler is C++ source code.

Keywords: game engine, compiler, multiple inheritance, simulation

1 Úvod

Práce se zabývá návrhem programovacího jazyka pro řízení tahových i real-time her s logickými či simulačními prvky, implementací překladače, komponenty běhové podpory a grafického uživatelského rozhraní, které slouží jak pro vývoj her, tak pro jejich běh a ladění.

1.1 Motivace a cíl práce

Navrhovaný jazyk vychází z programovacího jazyka použitého v Systému Krkal[1] [2]. Systém Krkal je komplexní prostředí pro vývoj, editaci a hraní dvourozměrných her. Systém byl převážně navržen pro logické a simulační hry, které obsahují velké množství rozmanitých, vzájemně se ovlivňujících, herních prvků.

Protože takové hry se vyznačují vysokou provázaností, přišel Systém Krkal s řešením, které ji snižuje a usnadňuje rozšiřování her o nové prvky. Byl navržen specifický programovací jazyk, který herní prvky popisoval jako třídy. Komunikace prvků byla umožněna přes systém zasílání (opožděných) zpráv. Díky zprávám objekty začaly měnit svůj stav v průběhu času, je tedy možné mluvit o simulaci jejich života. Jazyk dále obsahoval celou řadu prvků usnadňujících právě rozšiřitelnost.

Systém Krkal je integrované prostředí, které obsahuje editor skriptů, kompilátor, editor levelů, nástroj pro import grafiky, grafický engine a prostředí běhové podpory. Snaží se pokrýt všechny fáze vývoje hry, od programování přes editaci levelů až k vlastnímu hraní. Zajímavý je například editor levelů, který dovoluje zasahovat do běžící hry: Přidávat či rušit objekty a měnit jejich vlastnosti.

V současné době existuje velké množství systémů pro tvorbu her, které se liší zaměřením na určitý typ her i implementací. Mezi nejznámější patří Game Maker, Torque Game Engine nebo WME. Další možností jsou systémy komerčních her, například Unreal Engine. Kromě integrovaných systémů existují i herní knihovny a pro skriptování se často používají jazyky jako Python nebo Lua.

Systém Krkal se liší od výše popsaných řešení právě zaměřením na programovací jazyk, na simulaci žijících objektů a na rozšiřitelnost. Nová verze se nesnaží konkurovat kvalitou grafického výstupu nebo množstvím funkcí, ale přichází s komponentovým modelem, který umožní integrovat části Systému Krkal do jiných aplikací.

Hlavním cílem práce je navrhnout jazyk, který by byl použitelný v nové verzi Systému Krkal a implementovat komponenty, které jsou s jazykem svázány: kompilátor, běhové prostředí a grafické uživatelské rozhraní, které umožní jak psaní skriptů, tak interaktivní sledování jejich běhu.

Práce se zaměřuje jak na vylepšení a dopracování klíčových vlastností původního jazyka, jako je vícenásobná dědičnost, práce s množinami, rozšiřitelnost a verzování, tak na odstranění nedostatků, na zjednodušení a celkové zpřehlednění syntaxe. Důležitou součástí návrhu je i důraz na prvky zaručující bezpečnost, nový jazyk neumožňuje pracovat s neinicializovanou pamětí, ruší pointerovou aritmetiku a nově využívá garbage collector.

[1] Jméno vzniklo v roce 1996 sloučením prvních písmen z příjmení **Kr**če**k** a **Al**tman.

Součástí jazyka je i vícenásobná dědičnost. Práce vysvětluje její přednosti, popisuje její implementaci a řeší problémy způsobené při identifikaci položek předků. Vícenásobná dědičnost úzce souvisí i s rozšiřitelností. Zajímavým prvkem je možnost přidávat už existujícím objektům nové předky.

Jazyk je navržen tak, aby bylo možné modifikovat i složité systémy a jednoduše je rozšiřovat o nové prvky či pravidla. A to tak, aby ve většině případů nebylo nutné zasahovat do už existujícího kódu. Práce popisuje scénáře rozšiřování, současné modifikace nezávislými uživateli a řeší i případy, kdy je potřeba originální kód nahradit novým.

Jako primární výstup kompilátoru byl zvolen jazyk C++. Generátor kódu tvoří jednak typovou informaci a dále zdrojové soubory jazyka C++, které jsou pak překládány běžným C++ kompilátorem do modulu dll. Tento přístup má řadu výhod, například absenci interpretace nebo možnost snadno kontrolovat a ladit výstup kompilace.

1.2 Členění práce

Následující kapitola představí Systém Krkal, popíše jeho architekturu a návrh komponentového modelu nové verze. Třetí kapitola představí jazyk Krkal C, následuje kapitola, která popisuje rozšiřitelnost a verzování. Pátá kapitola popisuje vývoj jazyka Krkal C a rozebírá jeho klíčové vlastnosti. Následují kapitoly o implementaci kompilátoru, runtimu a integrovaného vývojového prostředí. Závěrečná kapitola shrne výsledky této práce.

2 Systém Krkal

Tato kapitola přiblíží Systém Krkal [2], který byl vytvořen jako projekt na Matematicko-fyzikální fakultě Univerzity Karlovy. Kapitola popíše funkce systému, ukáže jak v něm tvořit hru a zhodnotí jeho vlastnosti, nedostatky a možnosti dalšího vývoje.

Autory projektu Krkal jsou Petr Altman, Jan Krček, Jiří Margaritov a Jan Poduška. Nechť Krkal 2.0 označuje verze systému od obhájení projektu po současnost. Krkal 3.0 je nově vyvíjená verze, jejíž součástí je i nový jazyk a kompilátor, kterým se zabývá tato práce.

2.1 Úvod

Systém Krkal je integrované prostředí, které umožňuje vytvářet, editovat a hrát 2D hry. Patří do rodiny tzv. *Game Enginů*, tedy nástrojů usnadňujících vývoj her.[2]

Hra v Krkalovi se odehrává na obdélníkovém plánu. Krkal se hodí pro logické a simulační hry, pro arkády a strategie. Vyniká ve hrách, které obsahují množství složitých herních objektů, vzájemně se ovlivňujících.

2.2 Tvorba hry v Systému Krkal

Programování

Hra v Krkalovi se programuje ve speciálním skriptovacím jazyku Krkal C. Právě tento jazyk a jeho vlastnosti dělají Krkala výjimečného mezi ostatními systémy podporující vývoj her.

Jazyk je objektově orientovaný, tedy se každý herní prvek (stěna, kámen, hlavní hrdina, dveře, mina, …) popíše jako třída. Stav herního prvku je uchován v členských proměnných, chování objektů definují metody. Díky tomu, že metody se dají volat opožděně (zasílání zpráv a reakce na události), je možné simulovat „život“ objektů. Velký důraz byl kladen i na rozšiřitelnost, možnost snadného přidávání objektů, či nových pravidel, do již existujících her. Třeba díky vícenásobné dědičnosti je možné, jen pouhou kombinací předků požadovaných vlastností, vytvořit nový herní prvek.

Propojení skriptů a Systému Krkal

Skripty samozřejmě musí komunikovat se zbytkem systému. To umožňují jednak volání ze skriptů do systému, což bývají různé knihovní funkce. Dále třída může mít definované metody „známých jmen“, které systém bude volat v případě, že dojde k určité události. Takovou metodou je v Krkalovi i konstruktor. Poslední možností je reflexe (v Krkalovi 2.0 fungovala jen částečně). Systém může procházet jednotlivé

[2] *Game Engine* je komponenta, která tvoří jádro hry nebo interaktivní aplikace, zpřístupňuje potřebné technologie, usnadňuje vývoj. Engine většinou obsahuje vrstvu pro vykreslovaní 2D či 3D grafiky, funkce pro vstup a výstup, řízení času, fyzikální engine, skriptování, detekci kolizí atd. Za součást enginu bývá považována i sada nástrojů pro přípravu herních dat, jako například editor levelů, prostředí pro psaní a ladění skriptů nebo nástroje pro import a úpravu grafiky. Důležitou vlastností je i abstrakce od hardwaru a znovu-použitelnost enginů. Příkladem enginů, kterými se Krkal inspiruje, jsou Game Maker a Torque Game Engine.

Obrázek 1: Editor levelů v Krkalovi

proměnné a číst je nebo do nich zapisovat. Navíc je možné třídy a jejich položky označit atributy (tagy) a upřesnit tak systému význam jednotlivých položek.

Propojení objektů a grafiky

V Krkalovi je grafika od objektů oddělena. To, která grafika se má v určité situaci použít pro určitý objekt, je popsáno mimo skripty takzvanými pravidly automatické grafiky. Automatická grafika může vybírat obrázek či animaci náhodně, v návaznosti na sousední obrázky, podle vzoru nebo podle hodnoty proměnné objektu (používá se reflexe). Pravidla se dají kombinovat.

Editor levelů

Editor umožňuje umísťovat objekty na plán levelu (do mapy), či vytvářet objekty mimo mapu. To, jaké objekty se dají umísťovat do mapy (a jakým způsobem), poznává editor podle atributů (ale šlo by to udělat i dědičností od třídy známého jména). U objektů se navíc dají nastavovat jejich členské proměnné. I zde je způsob editace popsán pomocí atributů.

Během editace skripty běží a editované objekty tedy mohou s editorem komunikovat, či se po vytvoření správně inicializovat.

2.3 Kompilace

Kompilátor v Krkalovi 2.0 převáděl vstupní kód do třech výstupů. Šlo o typovou informaci, tedy detailní popis tříd a jejich členů, dále kód metod přeložený do mezikódu určeného k interpretaci a nakonec kód metod přeložený do zdrojového kódu jazyka

C++, který pak byl připojen k Systému Krkal a spolu s ním zkompilován běžným C++ kompilátorem. Překlad do C++ má řadu výhod, například možnost ladění a odpadnutí nutnosti interpretace. V Krkalovi 2.0 to ale bylo přístupné jen autorům, nikoli uživatelům.

2.4 Běh hry

Hra může být spuštěna buď samostatně nebo jako součást editoru levelů. Při startu dojde k inicializaci *Runtimu* (komponenty, která řídí běh skriptů, v Krkalovi také často nazývané *Kernel*). Runtime nahraje skripty (typovou informaci, metody kompilované C++ kompilátorem i metody interpretované), dále může nahrát uložený level či rozehranou hru.

Během startu runtime tvoří objekty (ty mohou být součástí statických proměnných nebo součástí levelu) a volá jejich konstruktory. Tímto okamžikem objekty začínají žít, volat metody, zasílat zprávy, tvořit jiné objekty.

Běh skriptů je dělen na takty. Takt je diskrétní časový okamžik, ve kterém runtime předává řízení skriptům. Volání metod může být v Krkalovi opožděné, místo okamžitého zavolání metody dojde k poslání *zprávy*. U zpráv si volající určuje časový okamžik, kdy má být zpráva doručena, může to být třeba v tomto taktu, v příštím taktu nebo za 200 ms. Pro runtime tedy takt neznamená nic jiného než, že vyzvedává zprávy s aktuálním časem doručení a vyvolává metody, které je obsluhují.

Pro běh hry jsou kromě Runtimu potřeba ještě další komponenty, které zajišťují převážně vstupně výstupní operace. Například *Grafický Engine*, *File Systém* a *GUI* (grafické uživatelské rozhraní).

2.5 Architektura Krkala 2.0

Krkal 2.0 byl sice rozdělen na komponenty (viz Obrázek 2), ale ty byly propojeny v rámci jednoho modulu (Krkal.exe), což neumožňuje jejich výměnu, či rozšiřování. Například i kompilované skripty tvořily nedílnou součást souboru Krkal.exe, tedy běžný uživatel je nemohl měnit.

Vlastnosti

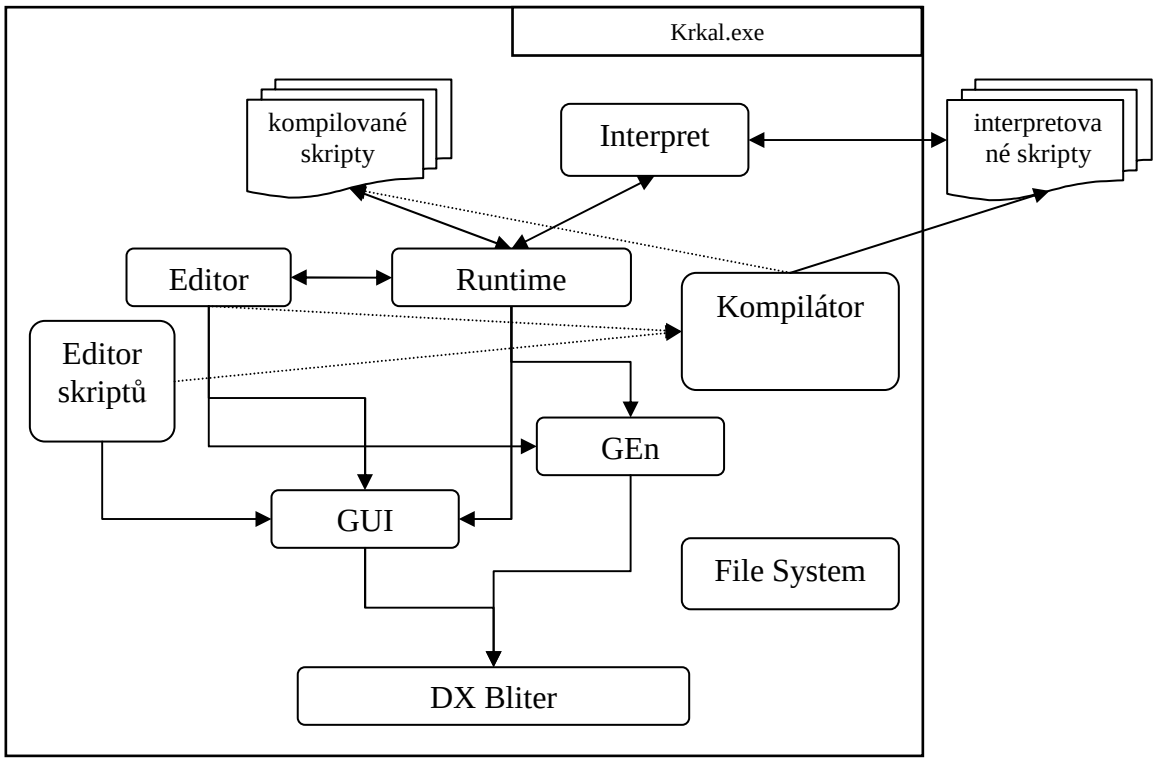
Projekt Krkal byl vyvíjen jako co nejobecnější syst0m pro tvorbu her, který by vyhovoval stejnojmenné hře Krkal.

Hra Krkal se vyznačuje velkým množstvím herních objektů se složitě definovanými interakcemi. Hra byla původně naprogramována v jazyce C a ukázalo se, že není vůbec jednoduché tyto interakce správně a bezchybně převést do zdrojového kódu. Ještě problematičtější bylo hru rozšiřovat o nové prvky. Každá změna si často vyžádala mnoho dalších změn na různých místech kódu.

Hra Krkal a zkušenosti s ní určily směr projektu. Bylo třeba navrhnout jazyk a runtime tak, aby umožňovaly jednoduše popsat prostředí žijících, vzájemně komunikujících objektů, a také, aby bylo možné tento systém jednoduše rozšiřovat.

Obecné vlastnosti Systému Krkal:

- **Krkal je objektově orientovaný.**
- **Objekty žijí a vzájemně se ovlivňují. Komunikaci a synchronizaci zajišťuje systém zasílání zpráv.**



Obrázek 2: Rozložení komponent v Krkalovi 2.0

- **Krkal je množinově orientovaný.** Jména entit (tříd, metod, grafiky, abstraktních pojmů,…) je možné seskupovat do množin a s množinami i jmény pak pracovat přímo v jazyce. Ve skriptech se mohou vyskytovat příkazy typu: Jestliže je objekt A z množiny *sebratelných objektů*, tak ho seber. Jazyk umožňuje kdykoli v budoucnu množiny rozšiřovat. Pokud tedy přibude nový *sebratelný objekt*, není nutné upravovat kód všude, kde se pracuje se *sebratelnými objekty*, stačí upravit množinu. Jména se dají využít i jako parametry některých jazykových konstrukcí, třeba k tvorbě objektu jména *n* nebo k volání metody jména *m*. A nakonec jména a množiny slouží jako základ vícenásobné dědičnosti.
- **Vícenásobná dědičnost.** Díky vícenásobné dědičnosti je možné poskládat novou třídu z předků, kteří představují požadované vlastnosti. Například nechť nová třída je: 1) pohyblivá 2) zničitelná 3) při zničení vybouchne 4) padá do děr 5) posouvají ji pásy 6) bojí se jí šneci...
- **Safe metody.** Safe metoda má řadu vlastností: Safe volání zajišťuje runtime. Volající si určí, jakou metodu zavolá a jaké ji předá parametry, není důležité vědět, jestli volaný objekt metodu obsahuje. Objekt může mít safe metod i více, v tom případě se zavolají všechny. Runtime se stará o převod argumentů, nezadané argumenty vyplní výchozími hodnotami, parametry navíc nevyužije. Safe metody se dají volat i jako zprávy.
- **Direct metody.** Tyto metody se volají běžným způsobem. Nepodporují rozšiřitelnost a posílání zpráv, ale jejich volání je rychlé, protože neobsahuje režii navíc.
- **Rozšiřitelnost.** Snadné rozšiřitelnosti napomáhají ostatní vlastnosti jazyka, hlavně množiny, vícenásobná dědičnost, safe metody a verzování.
- **Verzování.** Krkal je od základu navržen tak, aby umožňoval práci na hře více lidem najednou. Zabraňuje konfliktům jmen a umožňuje nezávislé modifikace

spojovat v jeden celek. Jménům jsou přiřazovány unikátní neměnné identifikátory. Tyto identifikátory jsou potřeba i v případech, kdy je na třídy navazována grafika, nebo když jsou objekty serializovány a jejich stav je ukládán do levelů. Jména pak spolehlivě identifikují objekty a jejich jednotlivé proměnné a umožňují level správně nahrát i v případě změny skriptů.

- **Jazyk není specializován na konkrétní systém.** Je jedno, zda bude použit pro 2D hru, 3D hru či aplikaci zabývající se diskrétní simulací. (Viz Propojení skriptů a Systému Krkal.)

Zhodnocení

V rámci projektu Krkal byl vytvořen nejen systém pro tvorbu her, ale byla implementována i hra Krkal, která poměrně důkladně prověřila funkčnost systému. Nyní je možné hodnotit dobré a špatné vlastnosti a zamyslet se nad dalším vývojem.

Hlavní vlastnosti systému (jména, vícenásobná dědičnost, safe metody, zprávy, rozšiřitelnost a verzování) se nejen osvědčily, ale ukázaly se jako nutné a klíčové prvky. Ale i zde je potřeba vylepšovat a dosáhnout větší obecnosti, přehlednosti, jednoduchosti, funkčnosti nebo korektnosti.

Jazyk Krkal C se vyznačoval poměrně nepřehlednou syntaxí. Podporoval složitý systém typů (základní typy, třídy, struktury, pointery, dva druhy polí a stringy). Nezaručoval bezpečnost a nechránil programátora například před nedovoleným přístupem do paměti. Pokud se k tomu přičtou ještě minimální možnosti ladění a značné množství chyb v samotném kompilátoru, je jasné, že je jazyk obtížně použitelný. Ukázalo se, že vhodnější, než pouze opravovat chyby, je celkově přepracovat návrh jazyka a implementovat nový kompilátor.

Dalším nedostatkem je již zmíněná uzavřenost systému v jenom `.exe` souboru.

2.6 Krkal 3.0

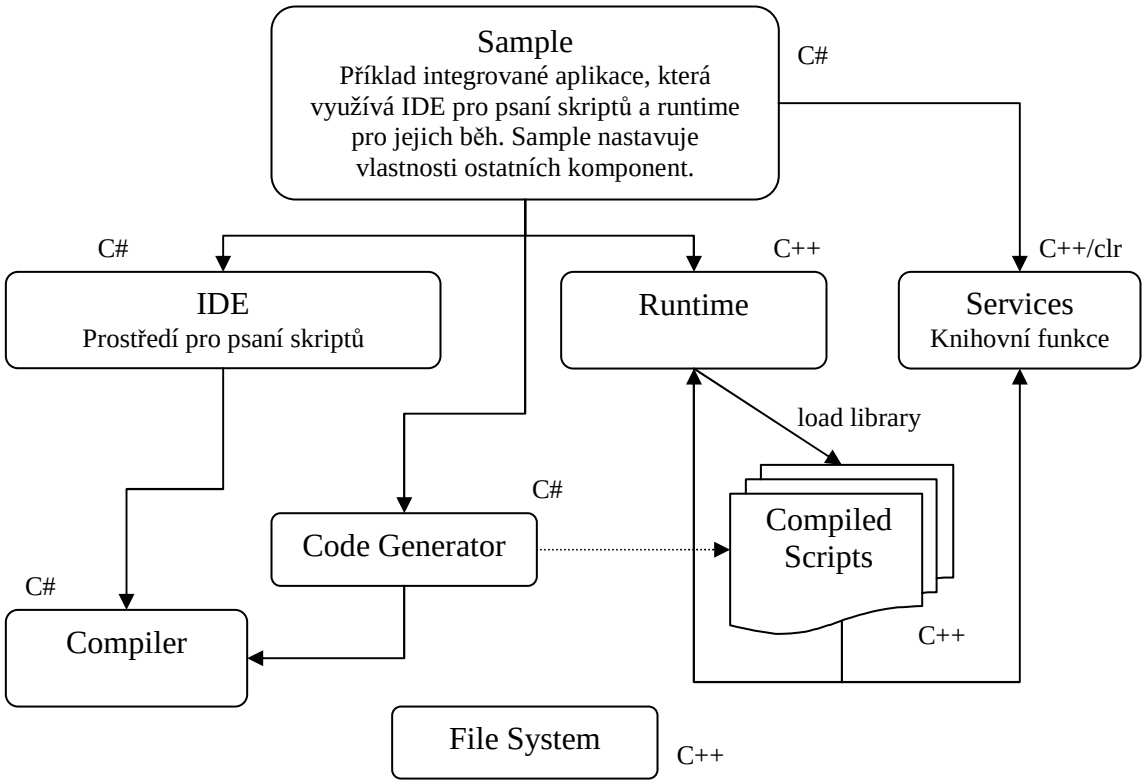
I Krkal 3.0 se soustředí na stejné klíčové vlastnosti jako předchozí verze a dále je rozšiřuje a vylepšuje.

Asi největší změna se týká samotného jazyka, jehož návrh byl od základu předělán. Jazyk byl zjednodušen a zpřehledněn. Důraz byl kladen i na bezpečnost programování a nízkou náchylnost k chybám. Jde asi o podobnou změnu, jakou je přechod z jazyka C++ do jazyka C#. Návrh nového jazyka bude podrobně popsán dále v této práci.

Radikální změna jazyka si vyžádá i rozsáhlé změny v celém systému. Kompletní přepsání kompilátoru, dále změny v runtimu, v editoru skriptů i editoru levelů a nakonec bude potřeba přepsat i hru Krkal. Všechny tyto změny není možné pokrýt jednou diplomovou prací, proto se práce soustředí hlavně na návrh jazyka a na implementaci nového kompilátoru.

Rozložení na komponenty

Krkal 3.0 už nebude homogenní celek, ale množina konfigurovatelných komponent. (Viz Obrázek 3.) Jiné aplikace budou moci tyto komponenty používat, můžou například převzít jazyk Krkal C spolu s kompilátorem a runtimem, ale dodat vlastní grafický engine, editor a ovládání.



Obrázek 3: Schéma komponent Krkala 3.0

Nová verze postupně vzniká z komponent, které jsou buď nově vytvořené a nebo převzaté z původního Systému Krkal a upravené. Tento postup je implementačně jednodušší, než pracovat od začátku s celým Systémem Krkal a každou změnu vždy promítat do všech jeho částí.

První krok představuje návrh nového jazyka a implementaci kompilátoru. Aby bylo možné skripty editovat, bylo implementováno IDE (integrované vývojové prostředí). Z původního Krkala byla převzata komponenta File System a komponenta běhové podpory (runtime), která umožňuje skripty spouštět a testovat. (Runtime byl zjednodušen a upraven pro nový jazyk.)

V současnosti jsou nové komponenty použity v aplikaci Sample, která slouží čistě k předvedení a testování nového jazyka. (Sample například neobsahuje grafický engine ani jiné prvky, které jsou potřeba ve hrách.)

Krkal C se stal obecně využitelným jazykem. Už není vázán na Systém Krkal ani na 2D hry, použit by mohl být například v aplikacích, které pracují se simulací.

Další vývoj Systému Krkal

Aby verze 3.0 byla kompletní, je třeba převzít a upravit i zbylé komponenty z původního Krkala, mimo jiné editor levelů a grafický engine.

Pro nového Krkala se plánuje i podpora jazykových lokalizací. Každé jméno bude mít dva textové atributy `UserName` a `Comment`, které budou překládány do lokálních jazyků. Navíc skripty v nové verzi pracují s unicodem.

Plánují se uživatelské profily a konfigurovatelný systém postupu po levelech a výběru levelů. Při dohrání levelu se podle určitých pravidel mohou odkrývat následující levely.

Další podstatnou věcí je hrou řízené GUI. Našlo by využití jak u klasických dialogů, tak u různých HUDů, či informačních a ovládacích panelů. Vzhled by měl být konfigurovatelný textově, například pomocí xml. Události by obsluhovaly skripty pomocí safe metod.

A nakonec třeba systém pro konfiguraci a vedení statistik. Kdo by nechtěl vědět, kolik času strávil v tom kterém levelu, kolikrát a jakým způsobem zemřel, nebo porovnávat skóre s ostatními hráči na internetu?

3 Popis jazyka Krkal C ve verzi 3.0

Tato kapitola přiblíží nový jazyk, jeho syntaxi a ukáže principy programování v tomto jazyce. Vývojem jazyka a diskusí nad jeho klíčovými prvky se pak zabývá kapitola pátá.

3.1 Hello World

```
#head {
    version F5BF_21B4_F74B_E490;
    include "System_5615_A57B_A943_7EFF.kc" 1D5B_E586_5718_3379;
}
#attributes []
#names {
}



class name Program;

class Program {

    static void @Main() {
        @Error.PrintDebugMessage(text = "Hello" + " World!");
        WriteLine(text = "Scripts are running!") timed 3000;
    }

    static void WriteLine(string text) {
        @Error.PrintDebugMessage(text = text);
        WriteLine(text = text) timed 3000;
    }

}
```

Každý soubor s kódem začíná hlavičkou, ta se dělí na tři části (`#head`, `#attributes` a `#names`). V Krkalovi soubor s kódem představuje v určitém smyslu komponentu. Hlavička tuto komponentu identifikuje, popisuje, jak se uvnitř souboru budou tvořit jména, a říká, které další soubory se mají při kompilaci použít. Hlavička bude podrobně popsána v kapitole Rozšiřitelnost a verzování.

Následuje definice jména `Program`. V jazyce se používají strukturovaná jména, obohacená o šestnáctimístné hexadecimální číslo, které zabraňuje konfliktům jmen. Hexadecimálnímu číslu se říká *verze* a jménu se říká *KSID jméno*. (Jménu `Program` kompilátor přiřadí verzi automaticky.) KSID jména mohou být různých typů, v tomto případě jde o jméno třídy. Jazyk umožňuje mezi jmény definovat množinové vztahy.

Uvnitř třídy `Program` jsou definovány dvě statické safe metody:

Metoda `@Main` je pojmenována tzv. *Známým jménem*. (Zavináč uvozuje systémová jména.) Runtime volá metodu `@Main` při startu. Protože jde o safe metodu, může metod tohoto jména existovat více a runtime by zavolal všechny. Koncept metody `@Main` má tedy blíže ke statickým konstruktorům z jazyků .NET, než k metodě `main` z jazyka C.

Metoda pomocí systémového volání vypíše text `"Hello World!"` a pošle zprávu metodě `WriteLine` s parametrem `"Scripts are running!"`, která má

být doručena za 3000 milisekund. V obou případech jde o volání safe metod, při kterém si volající sám určuje, jaké parametry předá, proto v kulatých závorkách není jen hodnota parametru, ale i KSID jméno parametru.

Metoda `WriteLine` vypíše text[3] a pošle sama sobě zprávu za 3000 ms, se stejným parametrem.

3.2 Program

Program se skládá z jednoho či více souborů s kódem. Každý soubor začíná hlavičkou, která mimo jiné informuje kompilátor o tom, jaké další soubory má načíst.

Za hlavičkou mohou následovat:

- Deklarace KSID jmen
- Definice závislostí
- Definice tříd

V jazyce nezáleží na pořadí deklarací[4]. Dokonce jména mohou být deklarována v libovolném souboru, který je kompilován, a použita v libovolném jiném.

3.3 KSID jména

V Systému Krkal je možné deklarovat jména a mezi nimi vytvářet závislosti.

Závislosti

Jména a závislosti musí tvořit acyklický orientovaný graf. Závislosti se dají přidávat postupně na kterémkoli místě programu. Na závislost se dá dívat i jako na vztah *být prvkem množiny*. U tříd závislosti popisují objektovou dědičnost. Zajímavým důsledkem je, že díky možnosti přidávat nové závislosti dodatečně, je možné nejen odvodit potomka od předka, ale i už existující třídě přiřadit nového předka či předky.

Vztah závislosti je tranzitivní, tedy platí: `a < b && b < c → a < c`

Závislosti definují jen částečné uspořádání, tedy **ne**platí že: `!(a <= b) → a > b`

Závislost se definuje pomocí klíčového slova **depend** mimo objektové závorky.

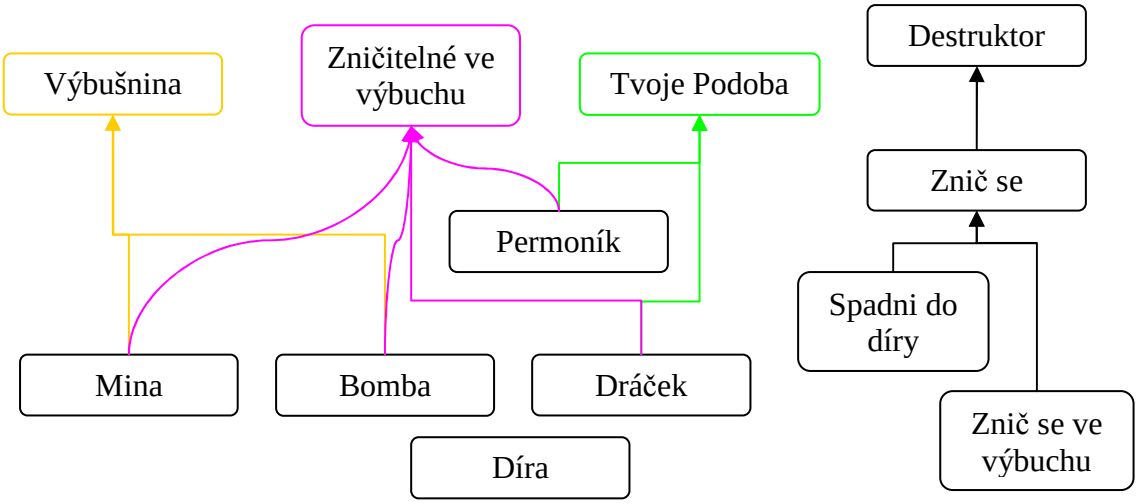
```
void name Sever, Jih, Zapad, Vychod;
void name Smery;

depend Smery << {Sever, Jih, Zapad, Vychod}; // Necht mnozina Smery obsahuje
                                             // Sever, Jih, Zapad, Vychod
class name Base, Left, Right, Derived;
depend Base << { Left, Right } << Derived;    // Diamant smrti
```

Operátory `<<` a `>>` určují směr závislosti a jeho význam vysvětlují následující příklady:

```
Predek << Potomek
Mnozina << Prvek
Vetsi << Mensi
```

[3] Při volání není povinnost parametr `text` zadat. V tom případě by měl hodnotu `null` a vytiskl by se prázdný řetězec. Metoda si může i zjistit, které její parametry byly zadány a které byly jen vyplněny výchozími hodnotami.

[4] Až na výjimky. Například pořadí deklarací safe metod stejného jména může ovlivnit pořadí jejich volání. Podle specifikace je pořadí volání safe metod nedefinované a programátor by neměl těchto mlhavých vlastností využívat.



Obrázek 4: Příklad KSID jmen a jejich uspořádání pomocí množinových vztahů

V kódu se pak dají závislosti testovat pomocí operátorů `<` `<=` `>` `>=` `==` `!=`.

```
if (obj <= $Moveable) ... // je objekt obj odvozen od tridy $Moveable?
```

Struktura KSID jmen

KSID jména jsou strukturovaná, tedy mohou se skládat z několika identifikátorů oddělených tečkou. Například `$jmeno1.jmeno2.jmeno3` nebo `$jmeno.jmeno` nebo `$MyClass.MyMethod.myParam`.

KSID jména mají globální platnost, tedy z jakéhokoli místa kódu jsou přístupná všechna jména.

Každý člen strukturovaného jména má přiřazenou verzi (16ti místné hexadecimální číslo), která zabraňuje konfliktům mezi jmény. Výjimku tvoří prvky systémových jmen, které verzi nemají. Více viz Rozšiřitelnost a verzování. Za normálních okolností je verze přiřazována automaticky, ale v případě nouze se jméno i s verzí zapisuje takto:

```
$SkakavaStena$F685_E5BE_1BD3_AA2F.ZavriSe$E3CB_FBA9_E57B_1ECD
```

Znak `$` uvozuje plně kvalifikovaná, uživatelem definovaná jména.

Znak `@` uvozuje plně kvalifikovaná systémová jména.

Není potřeba jméno plně kvalifikovat, v tom případě jeho prefix doplní kompilátor podle kontextu.

Příklad:

```
class Stena {              // $Stena
    int x;                 // $Stena.x
    void funkce1(int a) {}// $Stena.funkce1
                           // $Stena.funkce1.a
    void $funkce2() {}     // $funkce2
}

class Kamen {              // $Kamen
    int $Stena.x;          // $Stena.x – jmena maji globalni platnost a
                           // jdou pouzit i mimo svuj nativni kontext
                           // Stena i Kamen maji tedy promennou stejneho
                           // jmena
    double funkce1() {}    // $Kamen.funkce1
}
```

Globální kontext (prefix tvoří pouze `$` nebo `@`) platí mimo těla tříd a na místech, kde se očekává jméno typu, tedy u deklarací a za operátorem `new`.

Kontext třídy (prefix je tvořen jménem třídy) platí všude uvnitř definice třídy, pokud jiné pravidlo nestanoví jinak. Speciálně tento kontext platí i ve výrazech v závorce `()` za operátory `new`, `->` a ve výrazech v závorce, které mohou zastoupit jména parametrů safe metod.

Kontext jiného objektu platí bezprostředně za kódem `obj->`. Jako prefix se použije typ objektu `obj`.

Kontext safe metody je uplatněn v případě pojmenovávání parametrů dané metody. Jestliže je metoda volaná přes proměnnou typu `name`, není jméno metody v době kompilace známo a pro pojmenovávání parametrů se použije kontext třídy.

Příklad:

```
class A {
    static A A = new A(); // promenna $A.A je typu $A
}
```

Jména mohou být různých typů. Existují jména pro třídy, pro metody, pro proměnné i parametry, mohou existovat i jména pro grafiku, zvuky, ... Nové typy jmen lze do jazyka přidávat pomocí konfigurace. Jméno typu `void` nemá přiřazen význam a může být používáno třeba pro množiny abstraktních pojmů. Naopak specializovaná jména si s sebou nesou nějakou další informaci, třeba jméno třídy v sobě obsahuje definici té třídy. Jméno safe metody si pamatuje svůj návratový typ. I když daná safe metoda může mít mnoho implementací s naprosto rozdílnými argumenty, návratová hodnota musí zůstat všude stejná.

Jména se dají explicitně deklarovat mimo těla tříd:

```
class name MojeTrida;
void name Sever, Jih, Zapad, Vychod;
group name MyGroup;
```

Deklarace se skládá z klíčového dvojslova, kde první slovo představuje typ jména a druhé slovo je **`name`**. Výjimku tvoří deklarace jmen metod a proměnných, protože tam je součástí deklarace i typ proměnné či návratový typ metody. Deklarace pak má podobnou syntaxi jako definice proměnné či metody. U metod se neuvádí seznam argumentů.

```
int Promenna, Trida.Promenna;
static string Funkce(), JinaFunkce();
static double[] Pole;
```

Explicitně je nutné deklarovat jen jména tříd a všechna jména, mezi kterými jsou deklarovány závislosti příkazem `depend`.

3.4 Typy

Jazyk je staticky typován, tedy každá proměnná a každá část výrazu musí mít už v době kompilace jasně odvoditelný, konkrétní typ. V jazyce neexistuje typ `variant`, ani se nedají všechny typy převést na společný základ (na typ `object` jako je to v jazyce C#).

Jazyk obsahuje 5 základních typů:

- `int` – celočíselný znaménkový typ o velikosti 4 Byty
- `char` – bezznaménkový typ o velikosti 2 Byty, který může představovat číslo i unicodový znak
- `double` – typ s plovoucí desetinnou čárkou o velikosti 8 Bytů
- `name` – proměnná, do které se dají ukládat KSID jména
- `object` – společný předek všech tříd

2 specifické typy:

- `void` – prázdný typ
- `null` – má vždy pouze hodnotu `0` nebo `null`, kterou lze přiřadit do všech ostatních typů

Třídy.

Pole.

Typy lze rozdělit na *hodnotové* (`int`, `char`, `double`), u kterých proměnná obsahuje přímo hodnotu, operátor `=` ji kopíruje a porovnávací operátory pracují také přímo s hodnotou.

A na *referenční* (třída, pole), u kterých proměnná obsahuje referenci (pointer) na instanci, která se nachází někde na haldě. Třídy je třeba vytvořit operátorem `new`, pole se alokují sama automaticky. O uvolňování paměti se stará garbage collector, který automaticky uvolní referenční typ z paměti a to až tehdy, kdy na něj neexistuje žádný odkaz. Operátor `=` kopíruje vždy referenci, nikoli instanci. Analogicky operátory `==` a `!=` porovnávají pouze referenci. (Zbývající porovnávací operátory mají jiný význam.) V jazyce je zakázána jakákoli pointerová aritmetika vyjma prostého přiřazení a testu na shodu či nenulovost. Důsledkem je, že reference vždy odkazuje na platný objekt nebo je `null`.

Jazyk obsahuje `Destructor`, ten ale není volán automaticky, při uvolňování objektu z paměti (tehdy není volána žádná metoda), může být volán pouze explicitně uživatelem. Destruovaný objekt nepřestane existovat, ale přestane žít – runtime tomuto objektu přestane doručovat zprávy. Na destruktor mohou být vázány i další deinicializační kroky, například objekt přestane být vykreslován. Tyto kroky jsou konfigurovatelné.

Typ `name` stojí někde mezi hodnotovými a referenčními typy. Implementačně je to referenční typ. Může buď obsahovat pointer na KSID jméno a nebo `null`. Protože ale od každého KSID jména existuje jen jedna instance, která je platná po celý běh skriptů, chová se proměnná typu `name` jako hodnotová.

Typ `string` je jiné pojmenování typu `char[]` (pole charů).

Jazyk zná jenom implicitní (automatické) přetypování, které je definováno mezi typy `int` ↔ `char` ↔ `double` a mezi typy (objekt typu `A`) ↔ (objekt typu `B`). Navíc existuje implicitní přetypování z typu `null` na cokoli.

Existují dva druhy přetypování objektů. Přetypování z potomka na předka je rychlé a vždy uspěje, ale dá se použít jen tehdy, pokud je v době kompilace jasné, že jeden objekt je předek toho druhého. Ve všech ostatních případech se používá přetypování z libovolného objektu na libovolný jiný. Je pomalejší a může neuspět (pokud skutečný

objekt není cílový typ ani potomek cílového typu). Výsledkem neúspěšného přetypování je `null`.

```
class name Base, Derived1, Derived2;
depend Base << {Derived1, Derived2};

class Base {
    void @Main() {
        Derived1 d = new Derived1();
        Base b = d;  // ok, pretypovani na predka
        Derived1 d1 = b; // ok
        Derived2 d2 = b; // do d2 bude prirazeno null
    }
}
```

Jazyk nemá typ `bool`. Do logických výrazů mohou vstupovat všechny číselné typy, kde `0` znamená `false` a nenula znamená `true`. Navíc i všechny referenční typy, kde `null` znamená `false` a pointer na instanci znamená `true`. Výsledkem logických výrazů a porovnání je typ `int`.

3.5 Třídy

Třída je referenční typ, jehož data jsou alokována na haldě. Třída sdružuje členské proměnné, metody a metadata různého druhu. Třída může představovat jak žijící objekt (klíč, magnet, příšeru, ...), tak objekt zcela abstraktní, například prvek spojového seznamu.

Jazyk zatím nezná pojem privátních položek. Všechny položky třídy jsou přístupné v rámci celého kódu.[5] Programátor by i přesto měl dodržovat zásady objektově orientovaného programování a pracovat s daty pokud možno pouze uvnitř třídy a pro zpřístupnění dat mimo třídu použít safe metody.

Třída vzniká už deklarací svého KSID jména.

```
class name Trida;
```

Závorka třídy umožňuje definovat položky třídy.

```
class Trida {
    int a;
    int GetA() { return a; }
}
```

Protože jazyk Krkal C je založen na rozšiřitelnosti, umožňuje průběžné rozšiřování tříd o nové prvky. Závorka třídy se může v kódů vyskytovat vícekrát, v kterémkoli souboru. Závorka třídy tedy nepředstavuje definici třídy, jde spíše o druh namespacu.

```
class Trida {
    int b;
    int GetAPlusB() { return a+b; }
}
```

Proměnné

Uvnitř třídy je možné deklarovat *členské proměnné* i *statické proměnné*:

```
[static] <typ> <deklarace>[,…];
<deklarace> : <pojmenovani> [<atributy>] [= <inicializece>]
```

[5] Přístupová práva ve třídách by mohla ohrozit rozšiřitelnost. Pokud je něco deklarováno jako `private`, může v budoucnu nastat situace, že bude potřeba změnit typ na `public`, či naopak. Problém přístupových práv je otázka pro příští verze.

```
int a = 5, b;
double x = 1, y = 2;
static string text = “Ahoj!”;
int[][] pole;
```

Proměnné nikdy neobsahují nedefinovanou hodnotu. I v případě, že nejsou inicializovány explicitně, je runtime inicializuje na `0` nebo `null`.

Inicializace je výraz s následujícími omezeními: U inicializace členských proměnných není dovoleno přistupovat k ostatním členským proměnným či metodám a používat klíčové slovo `this`. Přístup ke statickým položkám a konstrukce objektů je povolen. `@Constructor` je volán až poté, co proběhly inicializace všech členských proměnných.

U inicializace statických proměnných je zakázán přístup k jiným statickým proměnným nebo metodám, ale konstrukce objektů je povolena. Varování: pokud by konstruktor přistupoval ke statickým proměnným, není zaručeno, zda už jejich inicializace proběhla či nikoli (neinicializované proměnné mají hodnotu `0` nebo `null`). Metoda `@Main` je volána až poté, co proběhly inicializace všech statických proměnných (u všech objektů).

Členská proměnná stejného jména může být definována vícekrát, pokud jde o definice u různých tříd. U všech výskytů proměnné se musí shodovat její typ.

Statická proměnná daného jména může být definována maximálně jednou.

Safe metody

Safe metody slouží ke komunikaci mezi objekty. Volání probíhá přes runtime a vazba mezi volajícím a volaným je volná. To umožňuje provádět nezávislé změny na jedné či druhé straně, tedy safe metody přispívají k rozšiřitelnosti. Na safe volání se lze dívat i jako na oznámení o události. Safe metoda pak onu událost může obsluhovat.

U jednoho objektu může existovat libovolný počet safe metod stejného jména. Metody musí být stejného typu, ale mohou přijímat libovolné argumenty. Pokud dojde k volání, runtime zavolá všechny metody v nedefinovaném pořadí. Příkladem nejhojněji vyskytované metody je `@Constructor`, není výjimkou třeba 12 konstruktorů u jednoho objektu. Každý z nich pak slouží k inicializaci své části, nezávisle na ostatních. V současné verzi je pořadí volání nedefinované, v budoucích verzích se plánuje, že bude zaručeno, že metody předků se zavolají vždy dříve než metody potomků.

Safe metody se deklarují podle následujícího vzoru:

```
[<modifikatory>] <typ> <pojmenovani> ([<paramentry>]) [<atributy>] <telo>
<parametry> : <parametr> [,…]
<parametr> : [<modifikator>] <typ> <pojmenovani> [<atributy>] [= <konstantni
vyraz>]
```

```
static name GetType(object o) {
    return o->Type();
}

int[] FillArray(int size, int value) {
    int[] arr;
    for (int f=0; f<size; f++) arr->AddLast(value);
    return arr;
}
```

Kde modifikátor u metody může být:

- `safe` – Nepovinné označení safe metody.
- `override` – Vynutí, že metoda bude u objektu jen jednou. `override` metoda u třídy nahradí jakékoli metody tohoto jména u předků. `override` metody se podobají virtuálním metodám z jiných jazyků, až na to, že v Krkalovi se z takové metody nedá zavolat implementace u předka. Tyto metody najdou své využití u polymorfních hierarchií, když je potřeba získat informaci specifickou pro každého potomka.
- `static` – Označuje statickou metodu.
- `retand`, `retor`, `retadd` – Existence více nezávislých těl stejného jména představuje problém pro návratovou hodnotu. Jak se zachovat, pokud několik metod najednou něco vrací? Jedna možnost je použít `override` a více metod zakázat, druhá možnost je specifikovat funkci, pomocí které bude runtime jednotlivé výsledky skládat. Možný je binární součet, binární součin a aritmetický součet.

U parametrů mohou být následující modifikátory:

- `ret` – Hodnota parametru se bude z funkce i vracet.
- `retand`, `retor`, `retadd` – Hodnota parametru se bude z funkce vracet a kombinovat jednou ze tří návratových funkcí.

Viz také kapitola Volání safe metod.

Direct metody

Direct metody jsou volány přímo, bez režie, která je nutná u safe metod. Může existovat maximálně jedna implementace direct metody daného jména. Nedají se použít jako zprávy, nefungují jako virtuální funkce, ani se nedají volat přes proměnno typu `name`.

```
direct [static] <typ> <pojmenovani> ([<parametry>]) [<atributy>] <telo>
<parametry> : <parametr> [,…]
<parametr> : [ret] <typ> <pojmenovani> [= <konstantni vyraz>]
```

```
direct int Sum(ret int result, int[] pole) {
    result = 0;
    if(!pole || !pole->GetCount())
        return 0;
    foreach (int f in pole) result += f;
    return 1;
}
```

I direct metody mohou být statické. Pořadí a typy argumentů jsou tentokrát pro volajícího závazné. Pokud je parametr označen klíčovým slovem `ret`, bude předán referencí, tedy direct metoda nebude pracovat s parametrem přímo, ale přes odkaz neboli pointer. Jména parametrů direct metod už nejsou KSID jména a proto nemohou být strukturovaná a nejsou globálně přístupná.

Direct volání je rychlejší, proto direct metody by měly být různé pomocné privátní rutiny s úzce specializovaným použitím, které se často volají během nějakého náročnějšího výpočtu. Vše ostatní by mělo být řešeno přes safe metody.

Skupiny a ovládací prvky

Jazyk dovoluje seskupovat položky třídy do skupin. Skupiny mohou být různých typů a jejich vlastnosti jsou konfigurovatelné. Skupina nemá žádný význam při vlastním psaní kódu, je používána editorem levelů ke správné interpretaci významu proměnných. Například, pokud byly dvě proměnné `x` a `y` uzavřeny ve skupině typu `Point2D`, editor věděl, že do těchto proměnných má ukládat souřadnice herního plánu a hodnoty proměnných se zadávaly jednoduše kliknutím na plán.

I ovládací prvky slouží k editaci. Může jít třeba o tlačítko nebo o separátor. Typ prvku je opět konfigurovatelný a případné parametry se nastavují přes atributy.

```
button control Tlacitko [OnAction = ObsluhaTlacitka, Parameter = 5];

cell3D group SouradniceObjektu [Editable] {
    int x, y, z; // souradnice
}

void ObsluhaTlacitka(int @ButtonParam) {
    ...
}
```

Dědičnost

Jazyk podporuje princip vícenásobné dědičnosti, který vychází z množinových vztahů. Množinové vztahy většinou věrně kopírují realitu. Třeba třída Bomba **je** Výbušnina, Sebratelná, Pohyblivá a Umístitelná, viz Obrázek 5.

Závislosti nemusejí vždy znamenat dědičnost, někdy může jít o prostý množinový vztah. Například `InventoryItem` může být jak třída, tak jméno typu `void`.

V Krkalovi je nejlepší se na dědění dívat jako na skládání (spojování) vlastností předků uvnitř potomka. Potomek přebírá od předků jejich proměnné a metody.

Nechť *položka* představuje konkrétní zápis nějaké proměnné, metody, či jiného prvku. Položky tříd jsou označeny KSID jménem, ale toto jméno většinou nestačí k jejich jednoznačné identifikaci, protože například safe metod může být u jedné třídy více stejného jména, proměnná stejného jména může být definovaná vícekrát u různých tříd. Položku a jméno položky je třeba rozlišovat.

Položky se dají rozdělit na *výlučné* a *nevýlučné*.

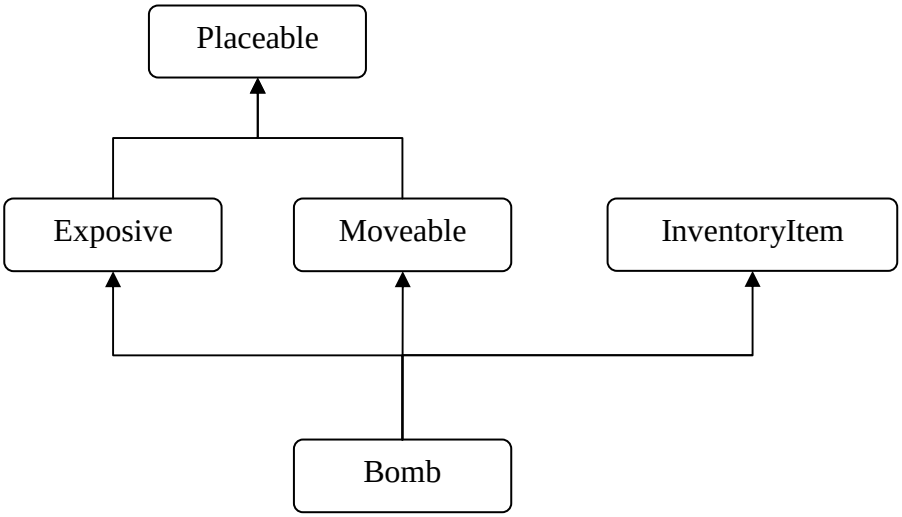
Výlučné položky vyžadují, aby třída obsahovala maximálně jednu položku stejného jména. Výlučné položky jsou proměnné, direct metody, ovládací prvky a safe metody označené jako `override`.

Nevýlučné položky nevyvolávají konflikty jmen. Třída může zdědit (nebo mít) libovolný počet položek jména X, pokud jsou všechny nevýlučné. Nevýlučné položky jsou safe metody (kromě `override`) a skupiny.

Jestliže třída P definuje výlučnou položku jména X, tak *zakryje* všechny položky jména X u svých předků. Tyto položky jsou zakryté (nebudou se dědit) jak pro třídu P, tak pro všechny její potomky.

Třída dědí od všech svých předků každou nezakrytou položku právě jednou.

Konflikt při dědění nastane, pokud by třída měla zdědit výlučnou položku jména X a alespoň jednu další položku jména X.



Obrázek 5: Schéma vícenásobné dědičnosti

Mějme třídy B(ase), L(eft), R(ight) a D(erived) s dědičností do kosočtverce (B << {L, R} << D). Konflikt vznikne ve třídě D, pokud L bude definovat výlučnou položku X_L jména X a R výlučnou položku X_R také jména X. Odstranit konflikt lze buď předefinováním položky v D (X_D) nebo odstraněním položek X_L a X_R a definováním položky X_B v B.

Příklad:

```
class name Placeable, Moveable, Explosive, Bomb;
void name InventoryItem;

depend Placeable << {Moveable, Explosive};
depend {Moveable, Explosive, InventoryItem} << Bomb;

class Placeable {
    int x,y; // coordinates
}

class Explosive {
    int power;
    double radius = 2.5;

    void @Constructor() {
        power = GetPower();
    }
    override int GetPower() {
        return 100;
    }

    void $Explode() {
        // ...
    }

    void $OnCrash() {
        $Explode() message;
    }
}

class Bomb {
    void @Constructor() { /* ... */ }

    void Activate() {
        $Explode() timed 300;
    }
```

```
    double $Explosive.radius = 3.5;

    override int $Explosive.GetPower() {
        return 50;
    }
}
```

Příklad znázorňuje také Obrázek 5. Třída `Placeable` definuje proměnné `$Placeable.x` a `$Placeable.y`, ty jsou děděny do tříd `Moveable` a `Explosive` a dále do třídy `Bomb`. Přestože `Bomb` je dědí ze dvou směrů, tak je bude obsahovat pouze jednou, protože jde o totožné položky. Ani kdyby `Placeable` měla safe metodu, tak by nedošlo k jejímu zdvojení ve třídě `Bomb`.

Třída `Explosive` definuje chování, které je společné všem výbušninám. Výbuch zajišťuje metoda `$Explode`. Dále třída reaguje na události, které by mohly způsobit výbuch (`$OnCrash`). Override metoda `GetPower` slouží k inicializaci proměnné `power`. Potomci mohou této metody využít k upřesnění síly výbuchu.[6] Alternativně lze použít i inicializaci přímo u členské proměnné (`radius`), pokud by ji podomek předefinoval, může určit jinou výchozí hodnotu.

Třída `Bomb` odvozuje své chování od všech předků. Definuje svůj konstruktor a minimálně jeden další dědí od třídy `Explosive`. Třída předefinovává metodu `$Explosive.GetPower` a proměnnou `radius`, protože jde o výlučné položky.

KSID jména se přísně řídí pravidly kontextů (viz Struktura KSID jmen). Jestliže je třeba pojmenovávat položku některého předka, je nutné buď použít její úplné jméno nebo objekt nejprve přetypovat na předka a poté lze použít zkrácené KSID jméno.

```
class name A, B;
depend A << B;

class A {
    int a;
}

class B {
    void Metoda() {
        // a = 0; // chyba!
        $A.a = 1;
        // nebo
        A a = this;
        a->a = 1;
    }
}
```

Přívětivější přístup, kdy kompilátor proměnnou vyhledává u předků automaticky, v Krkalovi použít nelze. Jazyk dovoluje třídy dodatečně rozšiřovat a dokonce přidávat nové předky, což by mohlo měnit význam kódu v metodách. Například následující rozšíření třídy B, by změnilo chování metody `$B.Metoda`, aniž by kompilátor nahlásil chybu:

```
class B {
    int a;  // a puvodne v B nebyla.
}
```

[6] Inicializovat proměnnou `power` pomocí konstruktoru s parametrem by bylo problematické. V Krkalovi je běžné parametry konstruktoru nezadávat. Například editor při tvorbě bomby, by nepředal žádné parametry. Ani potomci nemají možnost ve svém konstruktoru předat parametr předkovi. Jde o safe volání, metody jsou volány v nedefinovaném pořadí a nemohou jedna druhé předávat argumenty. Potomci by se mohli pokusit inicializovat proměnnou přímo, ale tam by mohlo dojít ke konfliktu.

Proměnná stejného jména je u objektu fyzicky vždy jen jednou. Nezáleží na divokosti dědění ani na tom, zda je používaná z metod předků či potomků.

```
class name Base, Left, Right, Derived;
depend Base << {Left, Right} << Derived;

class Base {
    int $a;
}

class Left {
    void Write(int a) { $a = a;}
}

class Right {
    int Read() { return $a;}
}

class Derived {
    static void @Main() {
        Derived d = new Derived();
        d->$Left.Write(a = 5);
        int a = d->$Right.Read();

        @Error.PrintDebugMessage(
            text = (a == d->$a && a == 5)->ToString());
        // vypise 1
    }
}
```

Statické položky

Statické proměnné i metody netvoří součást objektů a jsou přístupné z kteréhokoli místa přes své KSID jméno.

Může existovat více statických safe metod stejného jména. Při volání by byly zavolány všechny. Ostatní statické položky mohou existovat maximálně v jednom exempláři.

```
class A {
    static void $Metoda() {};
}

class B {
    static void $Metoda() {
        $Metoda();   // volani obou dvou metod
    }
}
```

3.6 Pole

Pole patří mezi referenční typy. Runtime ho tvoří automaticky při jeho prvním použití. Pole uchovává n prvků stejného typu, ke kterým lze přistupovat přes index. První prvek má index 0, poslední n-1. Meze pole jsou kontrolovány a v případě neplatného indexu je vyvolána chyba. Jazyk nepodporuje vícerozměrná pole, ale je možné tvořit pole polí.

Deklaraci pole tvoří základní typ a hranaté závorky `[]`. U vnořených polí odpovídá počet `[]` počtu vnoření. Pole lze inicializovat výčtem prvků ve složených závorkách`{}`.

```
int[] PoleIntu = {5, 3, 1};
int[][] PolePoliIntu = { {1,1,1}, null, {8,16} };
A[] PoleTridA;
string PoleCharu = “text”;
char[] TakePoleCharu = {0, 1, 0};
```

Kromě přístupu k prvku, umí pole celou řadu užitečných operací. Operace s poli se můžou lišit podle použitého runtimu (jde o konfigurovatelnou vlastnost jazyka). Mezi základní operace patří:

- `int GetCount()` – Zjistí aktuální velikost pole.
- `void SetCount(int)` – Zmenší nebo zvětší velikost. Nově vytvořené prvky jsou inicializovány na `0` nebo `null`.
- `void AddLast(T)` – Přidá prvek na konec. Asymptotická složitost je O(1).
- `void AddFirst(T)` – Vloží prvek na začátek. Asymptotická složitost je O(1).
- `T RemoveLast()` – Přečte a odebere prvek z konce. O(1).
- `T RemoveFirst()` – Přečte a odebere první prvek. O(1).
- `void AddRangeLast(T[])` – Přikopíruje na konec pole. O(n).
- `void AddRangeFirst(T[])` – Přikopíruje na začátek pole. O(n).
- `void Insert(int,T)` – Vloží prvek na daný index. O(n).
- `void InsertRange(int,T[])` – Vloží pole. O(n).
- `int Compare(T[])` – Porovná s druhým polem podle hodnot. O(n).

Operátor `+` spojí dvě pole. Operátor pro výsledek tvoří nové pole.

Příklad:

```
int[] pole;

for (int f = 0; f < 5 f++) {
    pole->AddLast(f);    // postupne pridava do pole prvky
}

int suma = 0;
foreach(int a in pole) { // soucet vsech hodnot v poli
    suma += a;
}

int[] pole2;
pole2->SetCount(5);  // vytvori prazdne pole o peti prvcich
pole2[3] = suma;
```

3.7 Typ name

Proměnná typu `name` uchovává KSID jména. Operátory `< <= > >= == !=` lze použít k testování množinových vztahů. Operátory se dají použít nejen na jména samotná, ale i na výrazy typu `object` nebo na výrazy kombinující `object` a `name`, objekty jsou automaticky konvertovány na typ `name` funkcí `Type`, která vrací jejich typ.

Jazyk umožňuje přes proměnnou typu `name` volat safe metody, tvořit objekty a specifikovat jména safe parametrů. Výraz typu `name` je potřeba uzavřít do kulatých závorek.

```
name metoda = $Metoda;
name trida = this->Type();
name param = $Metoda.p;

object o = new (trida) ();
o -> (metoda) ( (param) = 1 );
```

3.8 Atributy

Atributy doplňují definici položky nebo KSID jména o metadata. Atributy nemají vliv na kompilaci, ale může se jimi řídit runtime, editor levelů nebo jiná část systému, která používá reflexi. Klíčovou roli hrají v editoru levelů, určují například, která položka bude editovatelná a jakým způsobem.

Syntaxe atributů je plně konfigurovatelná. Specifikuje se jméno atributu, oblast platnosti a typ. (Tag je atribut, který může být přítomen nebo nepřítomen. Atribut typu hodnota obsahuje konstantu určitého typu.)

Atributy se mohou zadávat v kterékoli deklaraci, bezprostředně za KSID jménem, uzavírají se do hranatých závorek a oddělují se čárkou. Některé typy se vážou na KSID jména, jiné na položky. Položkové atributy podléhají dědičnosti.

```
class Bomb [InMap, UserName = "Bomba"] {
    int IsActive [Editable, EditMode = $Bool];
}
```

3.9 Programování uvnitř metod a výrazů

Každá metoda může obsahovat *výrazy*, *příkazy*, *deklarace lokálních proměnných*, a *bloky*.

Výraz představuje základní větu programovacího jazyka. Skládá se z konstant, z přístupů k proměnným, volání funkcí a z operátorů. Výraz představuje výpočet hodnoty určitého typu, lze tedy říci, že výraz je určitého typu. Krkal C je staticky typovaný jazyk a proto typy všech výrazů a podvýrazů musí být určeny už během kompilace.

Příkazy ovlivňují běh programu, představují větvení, cykly a skoky.

Blok seskupuje výrazy, příkazy, deklarace a jiné bloky do jednoho celku.

Deklarace lokální proměnné definuje dočasnou proměnnou na zásobníku, která platí od místa deklarace do ukončení bloku. Identifikátor musí být jednoduchý, nejedná se o KSID jméno. Jméno proměnné nesmí být v kolizi s jinou lokální proměnnou ve vnořeném nebo nadřazeném bloku, ale může být v kolizi s KSID jmény, tedy například se členskou proměnnou.

Význam identifikátorů uvnitř výrazů

Výrazy pracují s proměnnými, proto je každý identifikátor přednostně chápán jako proměnná. Pokud lokální proměnná nebo parametr zastíní KSID jméno, dá se využít úplné kvalifikace k přístupu ke KSID jménu. Pokud se jedná o KSID jméno proměnné a je potřeba místo s proměnnou pracovat se jménem samotným, dá se využít operátor `&`, který vrací jméno identifikátoru jako typ `name`.

Za operátory `&`, `new`, `->`, u deklarací na místě, kde se očekává typ, a u safe volání na místě, kde se očekává KSID jméno parametrů, platí jiné kontexty a očekává se zde KSID jméno, nikoli proměnná. Výraz typu `name` lze použít v některých případech také, ale musí být uzavřen do kulatých závorek. Viz Typ `name`.

Parametry safe metod jsou pojmenovány KSID jmény, metoda k nim ale přistupuje přes lokální identifikátory, které vznikají z poslední části KSID jména každého parametru. U direct metod tento převod není nutný.

```
void @Constructor(int Trida.param, string $text) {
    int i = param + text->GetCount();
}
```

Proměnné a konstanty

Výraz může obsahovat konstanty (literály) číselných, znakových a textových typů. Způsob zadávání těchto konstant je shodný s jazykem C#. Krkal například dovoluje jak textové konstanty s escape sekvencemi, tak verbatim stringy.

```
string a = “slova\toddelena\ttabem”;
string b = @”C:\Program Files\Krkal\”;
```

Jazyk definuje klíčová slova, která představují důležité proměnné a konstanty: `true` (1), `false` (0), `null` (0), `this` (aktuální objekt, statické metody mají `this` rovný `null`) a `sender` (objekt, který zavolal metodu).

Výraz může pracovat s KSID jménem jako s typem `name`.

Výraz může přímo přistupovat ke čtyřem druhům proměnných: ke statické proměnné, ke členské proměnné vlastního objektu, k lokální proměnné a k parametru.

Výraz může přímo volat statické metody a členské metody, pokud je jméno metody zadáno KSID konstantou.

Operátory

Prefixový unární operátor získání KSID jména:

```
&
```

Postfixové unární operátory přístupu:

```
->
(), []
```

Postfixové unární aritmetické operátory:

```
++, --
```

Prefixové unární aritmetické operátory:

```
++, --, +, -, !, ~
```

Binární aritmetické operátory podle priority:

```
*, /, %
+, -
<<, >>
<, >, >=, <=
==, !=
&
|
&&
||
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=  (asociativita zprava doleva)
```

Nejvyšší prioritu má operátor `&`, následují unární postfixové operátory, pak unární prefixové a nakonec binární. Ve výrazech se dají používat kulaté závorky k určování priorit výpočtu.

Logické binární operátory `&&` a `||` nevyhodnocují svůj pravý operand pokud to není potřeba.

Přístup k prvkům

Operátor `[]` přistupuje k prvku pole. Nalevo od operátoru je výraz typu pole, uvnitř závorek výraz typu `int`, určující index.

Operátor `->` slouží k přístupu k položkám objektu. Výraz typu objekt je nalevo od operátoru, napravo se očekává KSID jméno členské proměnné nebo členské metody. Kompilátor kontroluje zda položka třídě patří.

Safe metody se dají volat i přes proměnnou typu `name`, tedy pomocí konstrukce:

```
obj->(n)()
```

Operátor `->` může sloužit i k zavolání takzvané systémové metody. Tyto metody jsou konfigurovatelné, mají jednoduchá jména a volají se stejně jako direct metody. Systémová metoda může být definována na libovolném typu (nebo skupině typů). Třeba typ pole má celou řadu systémových metod jako `GetCount()` nebo `AddLast(T)`. Číselné typy a typ `name` mají například metodu `ToString()`, objekty a pole mají metody `Compare()` a `Clone()` a tak dále.

Volání direct metod

Direct metoda je volána přímo. Parametry se zadávají do kulatých závorek, oddělují se čárkami. Typy parametrů musí být kompatibilní se signaturou volané metody. Parametry, které má metoda označeny jako `ret` se předávají referencí.

Volání safe metod

Způsob volání plně určuje volající. Určuje, u jakého objektu se metoda bude volat, jakého jména metoda bude, jaké budou parametry (jména, typy, počet i pořadí) a zda se metoda bude volat opožděně jako zpráva. Přitom vůbec nerozhoduje, zda volaný objekt metodu má (či kolik) a jaké ve skutečnosti přijímají parametry.

```
class name A, B;
param name x;

class B {
    int $Metoda(string text, int param) {}

    void @Constructor() {
        int i = 0;
        i += $Metoda(text = “safe”); // nebyl uveden objekt a protoze nejde
                                     // o statickou metodu, tak se pouzije this

        A a = new A();
        i += a->$Metoda(param = “safe”, text = 5, $x = null);
                        // Neni jasne zda objekt A ma metodu $Metoda.
                        // Volat neexistujici metody je povoleno.
                        // Volajici plne specifikuje jmena a typy
                        // parametru. Runtime se je pokusi
                        // zkonvertovat, v pripade neuspechu
                        // nahlasi chubu.
    }
}
```

S každým KSID jménem metody je svázán návratový typ (včetně informace, zda jde o statickou metodu) V předchozím případě tedy kompilátor mohl určit typ výrazů, kde byla volána metoda `$Metoda,` i zda šlo o statické volání či nikoli.

Jméno metody se ale dá uchovávat i v proměnné typu `name`, pak je třeba kompilátoru napovědět, zda jde o statické volání. Navíc za operátorem `->` musí být výraz typu `name` uzavřen v kulatých závorkách. Návratový typ se kompilátor pokusí odvodit podle kontextu.

```
class B {

    void @Constructor(name B.method, name B.param) {
        double d;
        d = this->(method)( $Metoda.text = “safe”); // volani u objektu
        d += static->(method) ((param) = “text”);   // staticke volani
                        // jak jmeno metody, tak jmeno parametru
                        // je ulozeno v promenne typu name
    }

}
```

V tomto příkladu si kompilátor odvodil návratový typ jako `double`. První volání je objektové a druhé statické (Metoda nemůže být zároveň statická a zároveň objektová a proto runtime minimálně v jednom případě nezavolá nic). U statického volání je i jméno parametru zadáno pomocí proměnné typu `name`. (Kulaté závorky jsou opět povinné.)

Volající může označit některé parametry klíčovým slovem `ret`. Pokud u volané metody existuje kompatibilní parametr a je také označen jako `ret`, runtime vrátí jeho hodnotu volajícímu. V opačném případě runtime vrátí `0` nebo `null`.

Volající může metodu zavolat jako zprávu. Zpráva narozdíl od přímého safe volání nemůže nic vracet. Při vyvolání zprávy se zpráva jen uloží do některé fronty zpráv a řízení se okamžitě vrátí do volající metody. Runtime později vyzvedává zprávy z front a volá příslušné metody. Více o zprávách viz Zprávy.

Runtime volá vždy všechny metody, které má skutečný typ objektu. Přetypování na předka nemá na safe volání vliv.

```
class name Trida, Base, Program;
depend Base << Trida;

class Trida {
    void $Metoda() {}
}

class Base {
    void $Metoda() {}
}


class Program {
    void @Main() {
        Trida trida = new Trida();
        trida->$Metoda();       // zavolaji se obe metody

        Base base = trida;
        base->$Metoda();        // zavolaji se obe metody

        object obj = base;
        obj->$Metoda();         // zavolaji se obe metody
    }
}
```

Jak runtime volá safe metodu?

Při volání safe metody runtime ověřuje, zde objekt není `null` a u zpráv navíc zda žije. Poté se pokusí u cílového objektu najít tělo nebo těla metod, jejichž KSID jména odpovídají volanému KSID jménu.

I safe metody se dají propojovat závislostmi. Když objekty implementují reakce na události, mohou implementovat buď specifickou metodu, která při volání dostane přednost, nebo obecnější metodu, která se zavolá, jen pokud u objektu specifická metoda neexistuje.

Provádí se zde takzvané *zobecňování*. Když runtime u objektu nenajde přímo volanou metodu, zkouší zavolat nějakou obecnější metodu, ale takovou, která má k volané metodě nejblíže.

Přesněji hledá metodu (metody) jména m, tak aby:

- m >= M
- objekt obsahoval alespoň jedno tělo jména m
- a neexistovalo žádné m', že m > m' >= M a objekt obsahoval alespoň jedno tělo jména m'.

Kde M je jméno volané metody.

Runtime implementace safe metody hledá pomocí jednoho dotazu do perfektní hashovací tabulky, která je připravena při startu. Vyhledávání nemá negativní vliv na rychlost.

Pokud objekt obsahuje nějaké metody, runtime je postupně prochází. Pro každou metodu:

- Runtime páruje parametry podle jejich KSID jmen. I zde se provádí zobecňování. Je možné předávat parametry v různém pořadí, některý parametr nezadat (tělo do nezadaných parametrů dostává výchozí hodnoty a může si zjistit, zda byl, či nebyl parametr zadán). Je možné předávat při volání i nějaké parametry navíc – tělo je prostě nedostane.
- Do cílových parametrů jsou překopírovány volané parametry. Typ je automaticky zkonvertován. (Runtime hlídá situace, kdy by do jednoho cílového parametru byly předávány dvě hodnoty a hlásí je jako chyby.)
- Zavolá se tělo metody
- Pokud jde o přímé volání a volající si některé parametry označil jako `ret` a i volaný má odpovídající parametr označen jako `ret`, dojde k takzvanému *vracení hodnotou*. Hodnota z volané funkce je kopírována zpátky volajícímu. (hlídají se tu případy, že se na jedno místo vrací vícekrát nebo že se naopak nevrací vůbec, a vše se hlásí jako chyby.)
- Stejně se postupuje i u případné návratové hodnoty funkce.

Tvorba objektů

Konstrukce nových objektů je velmi podobná safe volání. Jen místo dvojice objekt, metoda se zde zadává typ třídy. Následuje seznam parametrů, kde jména parametrů jsou lokalizována ke jménu `@Constructor`. Konstrukci dokonce lze odložit díky mechanismu zpráv.

```
name n = $Trida;

Trida trida = new Trida(Trida.param1 = 5);
object o = new (n) ();
new Trida() timed 500;
```

Pro pojmenování parametru konstruktoru bylo použito strukturované jméno `$Constructor.Trida.param1`. Tato konvence předchází konfliktům jmen, které by u konstruktorů mohly často vznikat. Na druhém řádku byla vytvořena třída jména `n`. Třetí řádek zajistí vytvoření objektu typu `Trida` až za 500 ms.

Runtime pro vytvářené objekty nejprve vyhradí paměť, poté zavolá inicializace jednotlivých členských proměnných a nakonec zavolá `@Constructor`, jako běžnou safe metodu.

Příklad jak v Krkalovi jednoduše implementovat class factory:

```
void @Constructor() {
    classTypes = {$Left, $Base, @Error, $Derived, $Right, $ClassGame};
    Run() timed 2222;
}

void Run() {
    if (classTypes->GetCount() > 0) {
        name c = classTypes->RemoveFirst();
        object o = new (c) (Left.Value = 5, Right.Value = 1,
            Base.Value = 100);

        string s = o->$WhoAmI();
        @Error.PrintDebugMessage(text = s);

        Run() timed 2222;
    }
}
```

LValue a reference

Do výrazu typu LValue se dá přiřadit hodnota. Operátor `=` vyžaduje LValue na své levé straně. Operátory `++` a `--` také potřebují LValue. LValue je jakákoli proměnná, parametr nebo prvek pole.

```
GetPole()[5] = 4;    // ok
GetPole()[5]+1 = 4;  // chyba
```

U direct volání metody, která má `ret` parametr, je potřeba vytvořit referenci. Aby to bylo možné, parametr jednak musí být LValue a jednak musí být naprosto stejného typu, jako je u volané metody.

U safe volání určuje typy volající. Volající může rozhodnout, že některý parametr bude `ret` a tedy se bude předávat referencí a musí být LValue. Shoda typů ale požadována není, runtime typy zkonvertuje pokud to bude možné.

Jestliže existuje reference na prvek pole, runtime pole uzamkne. Do uzamčeného pole nelze přidávat ani ubírat prvky. Toto opatření je nezbytné, aby reference nezačala odkazovat na neplatnou paměť. Při přidávání prvků totiž může dojít k realokaci paměti pole.

Naštěstí reference existuje jen po dobu volání metody. Pole je poté odemčeno. Zprávy nemohou mít `ret` parametry, takže není možné uzamknout pole na neomezenou dobu.

Dohledávání typu výrazů

Některé výrazy nemají specifikovaný typ a kompilátor pak typ musí dohledávat pomocí kontextu. Příkladem je safe metoda volaná přes proměnnou typu `name`.

```
name n = $Metoda;
this -> (n) ();                  // Kompilator urci, ze volani je typu void.
int i = this -> (n) ();          // Zde kompilator urci typ int.
int j = this -> (n) ()[4];       // A zde bude navratovy typ int[].
this -> (n) ($param = this->(n)())
                                 // Chyba, kompilator nedokaze odvodit typ
                                 // vnoreneho volani.
```

Podobné je to i s inicializacemi pole. Teprve podle kontextu se dá odvodit, zda jde o pole charů, intů nebo doublů.

```
double[] pole;
pole = {1,2,3};
```

Kompilátor dovoluje mít dočasně výraz nespecifikovaného typu. Poté pokračuje v analýze kódu a řeší první operátor, který s nespecifikovaným výrazem pracuje. Nastává jedna s následujících situací:

- Operátor vyžaduje specifický typ. Například operátor `->` na své levé straně vyžaduje typ objekt, uvnitř operátoru `[]`, je potřeba typ `int`. Pak kompilátor specifikuje neznámý typ jako typ požadovaný.
- Operátor očekává boolovskou hodnotu. Pak kompilátor specifikuje typ `int`.
- Jde o unární operátor, ze kterého se nedá nic odvodit. Pak kompilátor rozhodnutí odloží.
- Jde o binární operátor. Kompilátor zkusí odvodit typ z druhého operandu. Pokud to nejde, například proto, že oba operandy jsou nespecifikované, tak kompilátor hledání zastaví a nahlásí chybu.
- Nespecifikovaný výraz je použit jako parametr safe volání. Pak kompilátor nahlásí chybu, tato situace nelze rozhodnout z definice.
- LValue nesmí být nespecifikovaného typu.

I když kompilátor typ určí, musí ještě zpětně zkontrolovat nespecifikovaný výraz, zda typ všude sedí.

Příkazy

Jazyk obsahuje následující příkazy: `if` + `else`, `while`, `do`, `for`, `foreach`, `break`, `continue` a `return`. Jejich syntaxe i použití jsou stejné jako v jazyce C++. Příkaz `switch` je plánován do příští verze.

Příkaz `foreach` slouží k procházení přes všechny prvky pole. Prvky pole jsou do iterační proměnné kopírovány, takže změna iterační proměnné nezmění prvek pole. Je plánována varianta, kdy iterační proměnná bude moci být reference. Foreach dovoluje iterovat i přes více dimenzí.

```
int[] pole;
int[][] pole2D;

double suma = 0;
foreach (double i in pole) // konverze jsou mozne
{
    suma += i;
}
```

```
// foreach dovoluje specifikovat podminku a iterovat pres vice dimenzi
foreach (double i in pole2D; i < 5) {
    suma += i;
}
```

Ošetření chyb

Jazyk Krkal C zatím neobsahuje výjimky. Runtime se snaží spíže chyby pouze logovat a z chyb se zotavovat, než se při jakékoli chybě okamžitě ukončit. Simulovaný svět většinou dokáže běžet i tehdy, pokud v jednom z objektů dochází k chybám. Současný runtime dělí chyby do čtyř kategorií:

Chyba, která není považována za chybu: například neúspěšné přetypování objektu (výsledkem je `null`), volání neexistující safe metody, zaslání zprávy neexistujícímu nebo mrtvému objektu.

Chyba, ze které se může runtime okamžitě zotavit. To jsou většinou chyby při předávání parametrů safe metodám (parametr nelze zkonvertovat, volaná metoda má vracet, ale nevrací). V tomto případě runtime chybu zapíše do logu a pokračuje v činnosti. Nezadané parametry jsou nastaveny na 0 nebo `null`.

Chyba, ze které se nelze okamžitě zotavit. Jde o přístup mimo meze pole, přístup přes `null` pointer nebo dělení nulou. V takovém případě runtime chybu zapíše do logu a vyvolá výjimku. Výjimka je chycena runtimem na některém záchytném bodě. Záchytný bod obsahuje většina funkcí runtimu, kromě přímého volání safe metod.

Panická chyba. Při této chybě je třeba runtime okamžitě ukončit. Chybu může způsobit například vadný soubor, ze kterého jsou načítány skripty, nebo přetečení zásobníku, či nedostatek jiných klíčových zdrojů.

3.10 Zprávy

Přímé volání přeruší běh stávající metody, vykoná se kód metody volané a teprve potom se řízení vrátí zpět. Metody volané přímo mohou vracet hodnoty.

Volání metody jako *zpráva* probíhá jinak. Při zavolání se zpráva uloží do příslušné fronty, včetně všech předávaných parametrů. A volající funkce pokračuje v práci. Volaná metoda se provede až někdy později, až na ni přijde řada. Záleží na typu zprávy.

Jestliže zpráva není doručitelná, runtime nevyvolá chybu, zprávu pouze zahodí. Pro zprávy se používají fronty, tedy dříve volaná zpráva je i dříve doručena (pokud byla použita stejná fronta).

Takt

Každý takt má runtime k dispozici čtyři základní fronty zpráv: `message`, `end`, `nextturn` a `nextend`. Navíc frontu `timed`, kde jsou časované zprávy. Runtime nejprve vyzvedne zprávy z fronty `timed`, jejichž čas odpovídá aktuálnímu času, a přidá je na konec fronty `message`.

Pak vyzvedává postupně všechny zprávy z fronty `message` a volá příslušné safe metody. Když frontu `message` vyprázdní a ve frontě `end` něco je, prohlásí frontu `end` za frontu `message` a frontu `end` nastaví jako prázdnou. A vše se opakuje.

Vyprázdněním prvních dvou front skončí takt. Runtime převede zprávy z front `nextturn` a `nextend` do front `message` a `end` a tím je vše připraveno k vyvolání nového taktu.

Typy zpráv

- `message`
- `end`
- `nextturn`
- `nextend`
- `timed <čas>` – Časovaná zpráva. Zpráva se vyvolá až po uplynutí příslušného počtu milisekund. Pokud tato doba nastane někdy mezi takty, zpráva musí počkat až na další takt, vyvolá se tedy trošičku později.
- `callend <objptr>` – Tyto zprávy se vyvolávají po ukončení metody. Pokud se v rámci volání metody provádí více nezávislých těl, tak se `callend` zprávy vyvolávají až po ukončení činnosti všech těl. `callend` front je mnoho, s každým vnořeným voláním se vytvoří jedna, do které se pak ukládají zprávy, které čekají na ukončení právě tohoto volání. `Callend` zpráva se vyvolá co nejdříve, ale ne tehdy, pokud ještě běží nějaká metoda objektu `objptr`. `Callend` tedy čeká, až daný objekt ukončí veškerou svoji činnost.

3.11 Volání knihovních funkcí, `extern` metody

Jazyk zná dva způsoby, jak volat vnější knihovní funkce. První možností jsou systémové metody, které se vážou na určitý typ a volají se přes operátor `->`. Viz Přístup k prvkům a viz Pole. Systémové metody se volají podobně jako direct metody, jejich jména a signatury jsou konfigurovatelné a definují se mimo jazyk.

Druhou možností jsou externí metody. Externí metody se definují přímo v jazyce Krkal C, mohou být `safe`, `direct`, `static` i `override` a tvoří běžnou součást tříd. Metody jsou uvozeny klíčovým slovem `extern` a kompilátor nepřekládá jejich tělo. Text těla předává kompilátor generátoru kódu v nezměněné podobě.

Jestliže je výstupem kompilace C++ kód, můžou těla `extern` metod obsahovat kód v jazyce C++, který přímo komunikuje s vnějším světem. Kompilátor, ani generátor kódu tento text téměř nemusejí modifikovat. Uživatel má dokonce možnost si v jazyce C++ programovat svoje vlastní knihovní funkce a ty pak z `extern` metod volat.

Příklad:

```
class name Error, Math;

class Error {
    static extern void PrintDebugMessage(string text, int param) {
        KString wstr = ctx.prm<KString>(0);
        char *a = UnicodeToAnsi(wstr->c_str());
        KerMain->KerServices->LogDebugInfo(5,1,ctx.prm<int>(1),a);
        SAFE_DELETE_ARRAY(a);
    }
}
```

```
class Math {

    static extern direct int Random(int range) {
        return KerMain->KerServices->mtr->randInt(_KSL_range);
    }
}
```

Externí metody vypadají na první pohled složitě, protože vyžadují znalost runtimu a kompilovaných skriptů, ale ve skutečnosti představují velmi efektivní a přímočarý nástroj, který umožňuje přímo v jazyce Krkal C psát knihovní funkce.

Kód externí metody není vždy převeden do jazyka C++ beze změny. Jestliže se metoda odkazuje na jiné prvky z programu v Krkal C (třeba na členské proměnné), je třeba tento odkaz správně identifikovat a přeložit. Kompilátor odkazy vyhledává v těle externí metody a doplňuje k ním potřebné informace. Každý odkaz uvozuje text `_KS*_`, následují parametry odkazu v kulatých závorkách.

- `_KSID_(n)` – Odkaz na KSID jméno, `n` musí být KSID konstanta (pro její zadání platí kontext třídy).
- `_KSG_(sv)` – Odkaz na statickou proměnnou, `sv` je její KSID jméno (platí kontext třídy).
- `_KSV_(obj,v)` – Odkaz na členskou proměnnou, `obj` je typ objektu, ke kterému se přistupuje, `v` je KSID jméno proměnné. (Pro `obj` platí globální kontext, pro `v` platí kontext objektu `obj`.)
- `_KSC_(fromDerived,toBase)` – Přetypování z potomka na předka. (pro obě KSID jména platí globální kontext).
- `_KSDM_(m)` – Odkaz na direct metodu, `m` je její KSID jméno (platí kontext třídy).

4 Rozšiřitelnost a verzování

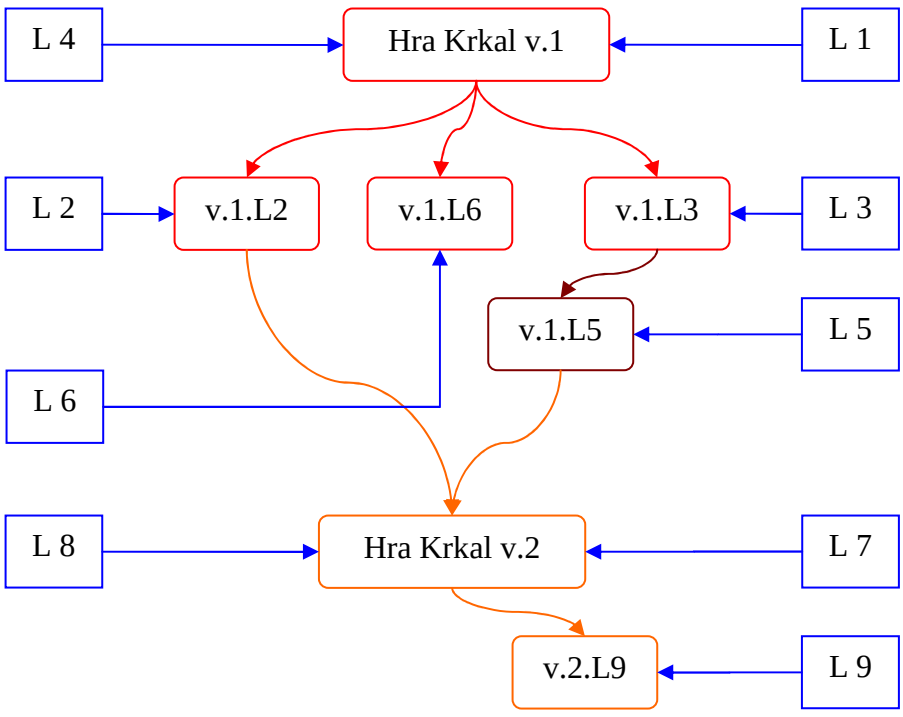
Tato kapitola popisuje propojení zdrojových souborů jazyka Krkal C, verzování souborů a verzování KSID jmen. Přiblíží možnosti a scénáře rozšiřování, nezávislý vývoj programu více uživateli a možnost nezávislé modifikace opět sloučit do jednoho celku.

4.1 Scénář rozšiřování

Jazyk Krkal C slouží k popisu hry nebo obecněji k popisu nějakého simulovaného světa. Svět, který jazyk popisuje, představuje jeden celek, který je vždy provázán mnoha složitými vztahy. Program v Krkalovi téměř nelze rozdělit na komponenty tak, jak to bývá zvykem v běžném programátorském světě.

Běžné programy se většinou skládají z komponent. Každou komponentu může vyvíjet jiná firma, některé komponenty poskytují služby, jiné komponenty tyto služby využívají. Komponenty jsou propojeny přes dobře definované interfacy a mezi komponentami neexistují cyklické vazby (až na výjimky callbacků). Při vývoji je snaha zachovat zpětnou kompatibilitu interfaců.

Program v Krkalovi se dá rozdělit maximálně na systémovou část, která zprostředkovává komunikaci s vnějším světem, a na vlastní hru nebo simulaci. Hra se dá dále rozdělit na třídy, či skupiny tříd, které k sobě logicky patří. Protože objekty spolu intenzivně komunikují a komunikace probíhá všemi směry, je téměř nemožné najít nějaké acyklické uspořádání, pomocí kterého by se dala hra rozdělit na skutečné komponenty. Krkal se o toto dělení ani nesnaží.



Obrázek 6: Vývoj hry Krkal. Téměř každý level využívá jinou verzi hry.

Krkal tedy musí jít vlastní cestou. Připouští složité vztahy mezi třídami a nedělitelnost programu na komponenty. Krkal umožňuje snadno modifikovat kterýkoli prvek a to tak, aby:

- Mohla nadále existovat stará verze programu a vše co ji používalo mohlo fungovat beze změny.
- Pokud dva lidé vytvoří nezávisle na sobě modifikaci stejného programu, neměly by mezi nimi vzniknout konflikty a obě modifikace by se měly dát jednoduše spojit do jednoho celku.

Příklad pro rozšiřování může poskytnout hra Krkal. Hra obsahuje přes 50 herních prvků propojených vztahy. Hra je rozdělena na levely a velmi si zakládá na originalitě, snaží se, aby každý level vypadal úplně jinak. Řekněme, že program hry Krkal je ve verzi 1 a první levely jsou postaveny na této verzi. Autoři dalších levelů už se s verzí 1 nespokojí a pro své nové levely si vždy vymyslí nějakou originální modifikaci či změnu pravidel – v každém levelu jinou. Pak nastane otázka, jestli by nebylo zajímavé tyto nové věci spojit do jednoho celku a zkusit jak fungují dohromady. Vznikne Krkal verze 2 a situace se může opakovat. Viz Obrázek 6.

4.2 Verzování zdrojových souborů

Soubor s kódem v Krkalovi nahrazuje komponentu. Ale nejde o komponentu ve smyslu klient server aplikací nebo ve smyslu Assemblies. Vlastnosti této „komponenty“ jsou jiné.

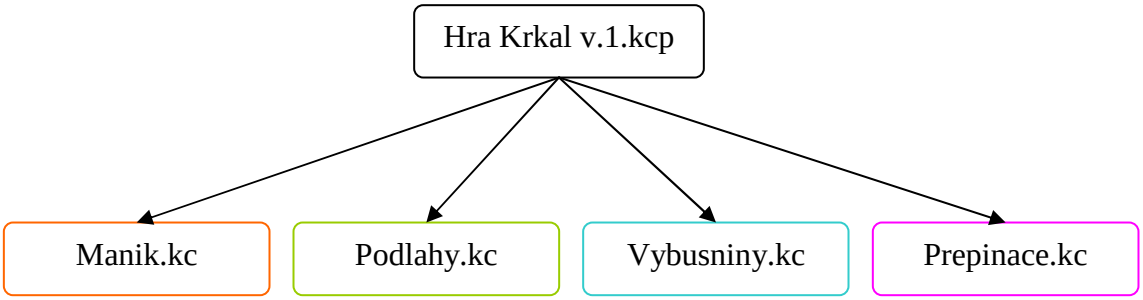
Každý soubor je identifikován svým souborovým jménem bez cesty. (Jazyk předpokládá, že buď soubory budou umístěny všechny v jednom adresáři a nebo je komponenta File System dokáže ve vnořených adresářích automaticky najít.) Součástí jména je i verze (náhodně vygenerované 16ti místné hexadecimální číslo, které zabraňuje konfliktům jmen). Například:

```
AntDeathMatch_3798_03CD_176A_36B2.kcp
```

V hlavičce souboru je uvedena verze komponenty (také náhodně vygenerované hexadecimální číslo) a seznam dalších souborů, které se mají použít při kompilaci. Odkazy na soubory nesmí tvořit cyklus. Význam odkazu není v tom, že daný soubor závisí na souboru, na který odkazuje. Kdyby to tak bylo, tak by nešlo zabránit cyklům, protože v Krkalovi potenciálně každý soubor závisí na každém. Kompilátor na začátku projde graf odkazů a vytvoří seznam použitých souborů. Jestliže je potřeba kompilovat soubor S, stačí, když na něj existuje jeden odkaz.

Soubory se rozdělují na projekty a komponenty. *Projekt* je takový soubor, který má význam samostatně kompilovat. Kompilací projektu vznikne funkční program. Soubory s projekty mají příponu `.kcp`. *Komponenta* tvoří součást projektů, k samostatné kompilaci určena není. Přípona komponenty je `.kc`. Rozdíl mezi projekty a komponentami je pouze formální, toto rozdělení napomáhá přehlednosti.

Pro zvýšení přehlednosti se doporučuje dávat odkazy pouze do projektů a naopak do komponent umísťovat vlastní program. Viz Obrázek 7.



Obrázek 7: Projekt se odkazuje na všechny své komponenty

Právo editovat soubor

Systém verzování v Krkalovi předpokládá, že každý soubor má svého autora a úpravy v daném souboru může dělat jen autor. (Autorem, může být i skupina lidí, pro verzování je důležité pouze to, aby se dokázali dohodnout na výsledné podobě změn.)

Autor vyvíjí svůj soubor po určitý čas, testuje jeho funkčnost a kompatibilitu a nakonec ho publikuje. Poté už by v souboru neměl dělat žádné změny, které můžou ohrozit kompatibilitu, protože jiní uživatelé už jeho soubor mohou používat ve svých projektech nebo levelech. Zejména by neměl měnit typy položek, jména položek a odebírat položky nebo závislosti. Přidávání nových položek je bezpečné, přidávání závislostí také, pokud se nestrefí nešťastně do cyklu. Změna kódu metod, by měla být prováděna opatrně.

Autor může svůj soubor prohlásit za uzavřený, tedy slíbí, že už v něm nebude dělat žádné změny.

Specifikace Systému Krkal 3.0 obsahuje i návrh, jak zamezit neautorům modifikovat soubory a jak zabránit modifikacím uzavřených souborů. Striktní hlídání práv na soubory, by ale pravděpodobně příliš omezovalo pohodlí uživatelů, politika založená na čestném slově může být pro Krkala vhodnější.

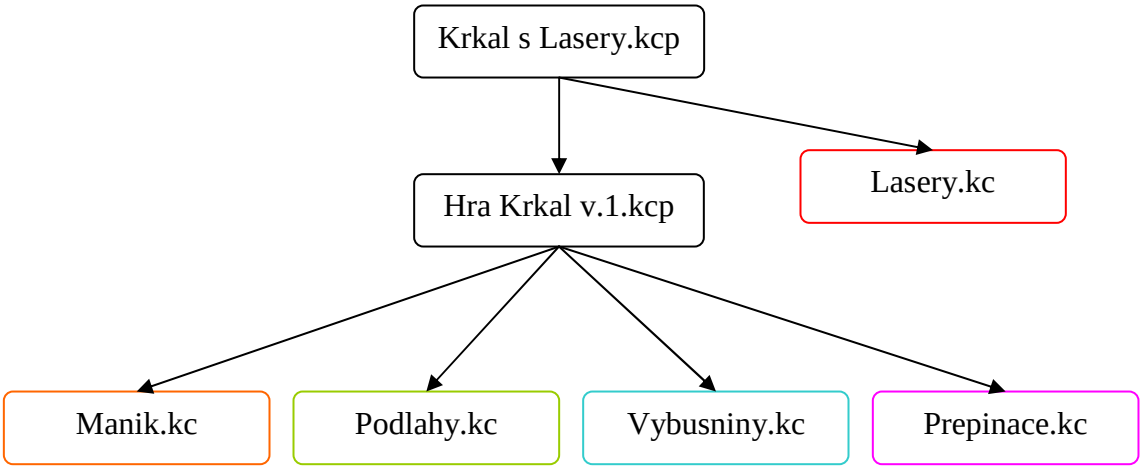
Modifikace přidáváním

Kterýkoli uživatel může rozšířit existující projekt o nové prvky tak, že vytvoří nový projekt, do kterého umístí odkaz na původní projekt a na svůj soubor s vylepšeními.

Jazyk je navržen tak, že většinu modifikací lze provádět v novém samostatném souboru, bez nutnosti měnit soubory existující. Už publikované soubory by neměly být měněny.

Přidání nové třídy A obvykle vyžaduje tyto kroky:

- Třída A je přidána do všech množin, kam patří. (Jsou jí tedy přiřazeni i předci, od kterých bude dědit.) (Jazyk dovoluje definovat nové množinové vztahy.)
- Jestliže A pracuje s množinou jiných tříd, je třeba tuto množinu definovat a třídy do ní přidat.
- Pokud jiné třídy mají na A nějak specificky reagovat, lze vytvořit buď novou metodu u jiné třídy nebo rovnou novou třídu B, která popisuje způsob komunikace s A, a všem třídám, u kterých je to potřeba, přiřadit B jako předka.



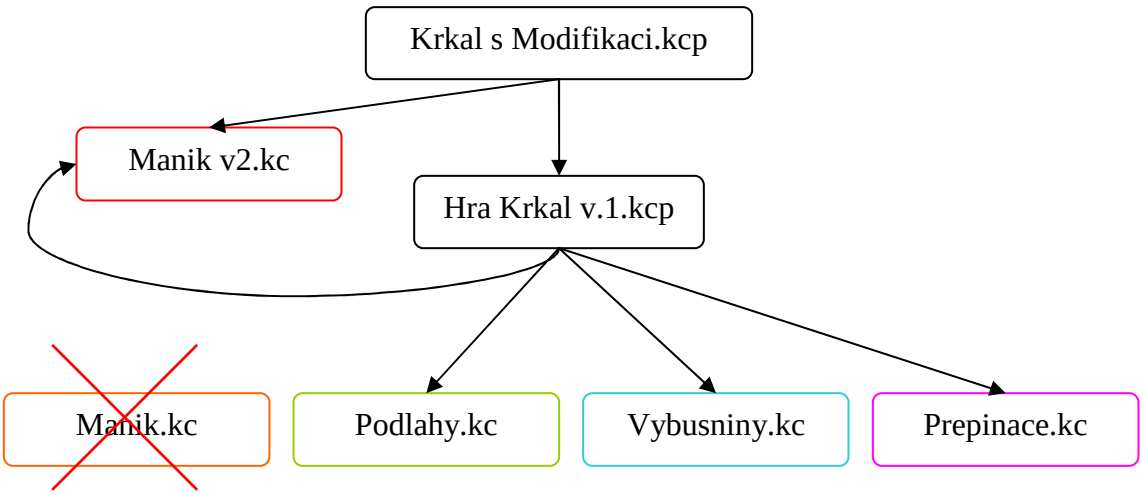
Obrázek 8: Rozšíření přidáním nové komponenty

(Jazyk dovoluje kdykoli otevřít objektovou závorku už existující třídy a rozšiřovat ji o nové položky. Jazyk dovoluje přidat existujícím třídám nového předka. Safe metody podporují více implementací, takže se dají rozšiřovat obsluhy událostí.)

Náhrada existujících souborů

Nastanou situace, kdy přidávání nestačí. V základní verzi může být chyba nebo základní verze není navržena dostatečně obecně, aby umožnila přidání nových věcí. Typickou chybou bývá implementace chování, které ovlivňuje všechny objekty, místo toho, aby se pracovalo s množinami. Pro každé pravidlo se najde výjimka.

Zasahovat přímo do souborů, které je potřeba změnit, je obvykle zakázáno. Proto jazyk dovoluje okopírovat chybný soubor pod jiné jméno, učinit v něm potřebné modifikace a nakonec připojit modifikovaný soubor k novému projektu tak, že nahradí soubor stávající. Viz Obrázek 9. Existující soubory včetně souborů s projekty (`Hra Krkal v.1.kcp`) modifikovány nebyly, proto hry založené na projektu `Krkal s Modifikaci.kcp` budou obsahovat změnu, ale hry založené na `Hra Krkal v.1.kcp` nikoli.



Obrázek 9: Náhrada existujících souborů

Kompilátor automatický přesměrovává odkazy v podřízených souborech, pokud některý nadřízený soubor obsahoval odkaz na jinou verzi komponenty.

Algoritmus výběru souborů

Soubor je identifikován unikátním jménem. Uvnitř souboru se nachází hlavička, kde se v části `#head` nachází verze komponenty (uvozená klíčivým slovem `version`) a seznam odkazů na jiné soubory. Odkaz je uvozen klíčovým sloven `include` a obsahuje jméno souboru a verzi komponenty toho souboru.

```
// soubor "DebugTest_C328_D5C4_4B70_15FB.kcp"
#head {
    version F5BF_21B4_F74B_E490;
    include "System_5615_A57B_A943_7EFF.kc" 1D5B_E586_5718_3379;
    include "Classes_BF89_9CCC_0137_BF96.kc" A3F1_32CF_6DCC_F0D7;
}
```

Jestliže vzniká nový soubor, IDE vygeneruje náhodné hexadecimální číslo a připojí ho ke jménu souboru. Dále IDE vytvoří hlavičku, kde vyplní verzi komponenty dalším náhodně vygenerovaným číslem.

Pokud vzniká modifikace existujícího souboru, je třeba ji nově pojmenovat. IDE vygeneruje nové číslo do jména souboru. Verze komponenty v hlavičce zůstává nezměněna.

Soubory ze skupiny S jsou vzájemné modifikace právě tehdy, jestliže všechny soubory ze skupiny S mají stejnou verzi komponenty. Z každé skupiny S, jejíž soubory jsou vzájemné modifikace, smí být při kompilaci použit maximálně jeden soubor.

Soubory propojené odkazy si lze představit také jako orientovaný graf. Kompilátor načte hlavičku kořenového souboru – to je soubor s projektem, který je právě kompilován – a začne prohledávat graf do hloubky.

Soubory, kde ještě kompilátor nebyl, jsou označeny jako bílé, soubory, které jsou právě zpracovávány, jsou šedé a už zpracované soubory jsou černé. Soubory navíc obsahují čítač, který říká, kolikrát byly viděny na cestě od kořene k právě zpracovávanému vrcholu. Zpracování souboru probíhá takto:

1. Kompilátor označí soubor jako šedý.
2. Kompilátor nahraje soubor (z disku, z cache nebo z editoru skriptů) a načte jeho hlavičku.
3. Kompilátor ověří, zda se shoduje verze komponenty v hlavičce s verzí komponenty, která byla uvedena v odkazu na tento soubor.
4. Pro každý odkaz na soubor A, komponenty A_K:
    a. Jestliže A_K se už někde vyskytla, nechť B je dříve nalezený soubor této komponenty. Pak buď A je B. Nebo B je jiný soubor, ale byl viděn na cestě od kořene k tomuto místu (jeho čítač je vyšší než 0), poté B má nahradit všechny soubory komponenty A_K, kompilátor tedy odkaz A, A_K nahradí odkazem B, A_K. Nebo B je jiný soubor, ale jeho čítač je roven 0 – v tom případě byly odkazy nalezeny v nezávislých větvích grafu a kompilátor musí nahlásit chybu.
    b. Kompilátor zvýší čítač viděných souborů u A o 1 a přidá komponentu A_K mezi už viděné komponenty.
    c. Jestliže soubor A je šedý, je v odkazech cyklus a kompilátor musí skončit s chybou.
5. Pro každý odkaz na soubor A, komponenty A_K:

    a. Jestliže je A bílý, tak kompilátor rekurzivně zpracuje A.
6. Pro každý odkaz na soubor A, komponenty A_K:
    a. Kompilátor sníží čítač u A.
7. Kompilátor zařadí zpracovávaný soubor mezi soubory určené ke kompilaci.
8. Kompilátor označí soubor jako černý.

Tento algoritmus zaručí, že od každé komponenty bude použit maximálně jeden soubor, detekuje cykly v odkazech, ověřuje zda se shoduje verze komponenty v odkazu a v samotném souboru. Kompilátor nahrazuje odkazy na staré verze komponent, pokud v nějakém nadřazenějším souboru byl použit jiný odkaz. Algoritmus neotvírá soubory s komponentami, které nebudou použity. Výstupem algoritmu je topologicky uspořádaný seznam souborů.

Nevýhodou je závislost na pořídí procházení. Existuje špatně navržené uspořádání odkazů, u kterého algoritmus nenahlásí chybu, ale vrátí správný výsledek.

4.3 Verzování KSID jmen

Pro rozšiřitelnost je také důležité zamezit konfliktům jmen, která vznikla nezávisle na sobě. Jazyk Krkal C nepoužívá namespacy jako jiné jazyky, ale KSID jména.

Kdyby v jazyce namespacy byly, jejich použití by mohlo vypadat například tak, že každý autor by měl svůj namespace a všechna jména, která by vytvořil, by se do tohoto namespacu přidávala. Kompilátor by do něj například automaticky vkládal i nové položky, definované autorem u cizích tříd.

Pracovat se jmény jiných autorů by šlo pomocí plně kvalifikovaných jmen. Otázkou zůstává, zda by šlo namespacy použít (zpřístupnit) i na začátku souboru. Co dělat, jestliže v namespacech vznikají nová jména a v souboru vznikají konflikty, které můžou změnit význam kódu?

Namespacy se pravděpodobně nehodí pro tak provázaný kód, jako bývá popisován jazykem Krkal C.

V Krkalovi existují strukturovaná KSID jména, k nimž kompilátor doplňuje verze.

Určení verzí KSID jmen

Kompilátor určuje verze postupně, pro KSID jméno `$a.b.c` kompilátor nejprve určí verzi jména `$a`, pak jména `$a$va.b` a nakonec verzi jména `$a$va.v$vb.c` a dostane výsledek `$a$va.v$vb.c$vc`. Tento postup je důležitý proto, aby se zachovaly společné prefixy. Jestliže by `$a` bylo již existující jméno třídy, které už je používáno a tudíž má fixovanou verzi `va1`, a `$a.b` by bylo jméno nové položky, pak kompilátor nejprve hledá verzi pro `$a`, najde `va1` a pokračuje s `$a$va1.b`, tady zjistí, že toto jméno ještě použito nebylo, a může sám přiřadit verzi.

Pro verze KSID jmen je důležitá jejich fixace. Nebylo by žádoucí, aby se KSID jména s každou kompilací měnila. Na jména by nešla navázat grafika, nešla by ukládat do levelů a podobně.

Starý Krkal fungoval tak, že přiřadil jménu verzi souboru, ve kterém bylo deklarováno. Tento postup zaručí, že nebudou vznikat konflikty jmen, pokud nezávislí autoři pracují každý s jiným souborem. Ale fixace jména je zaručena jen zdánlivě. Přidáváním nových souborů se může stát, že kompilátor začne kompilovat soubory v jiném pořadí a nejednou první použití jména bude v jiném souboru a jméno dostane

jinou verzi. Navíc zlý uživatel může deklaraci jména v jednom souboru zrušit a dát ji do jiného.

Nový Krkal fixuje jména pomocí tabulky jmen v hlavičce každého souboru. Při určování verze jména N v souboru S se kompilátor nejprve podívá do tabulky jmen v souboru S, když tam najde záznam, použije verzi z tabulky, jinak prohledá tabulky všech ostatních souborů a:

- najde jeden záznam, pak použije jeho verzi.
- najde více záznamů, pak existuje více jmen stejného jména a kompilátor neví, které má použít. Nahlásí chybu a uživatel buď musí zapsat správnou verzi přímo ke jménu N a nebo ručně upravit tabulku jmen souboru S (uživatel tak určí, jaká z variant jména bude platná v S).
- nenajde žádný záznam (jméno je nové), pak jménu přiřadí souborovou verzi souboru S.

Poté, co je verze určena, kompilátor aktualizuje tabulku souboru S.

Nakonec, jestliže kompilace proběhla bez chyb, by měl kompilátor aktualizované tabulky zapsat zpátky do souborů. (Tento krok bohužel v současnosti ještě není implementován.)

V tabulce mohou nejen jména přibývat, ale i ubývat, pokud uživatel identifikátor ze souboru odstraní. Kompilátor tato stará jména ale z tabulek neodstraňuje, jen je označí tagem `old`. Uživatel mohl jméno třeba jen dočasně zakomentovat, proto by bylo nepříjemné, kdyby byla zapomenuta původní verze jména. Když se jméno vrátí, kompilátor uživatele upozorní, aby překontroloval správnost verze.

Fixace jmen v jednotlivých souborech dovoluje kompilovat soubory v libovolném pořadí.

4.4 Sloučení nezávislých modifikací do jednoho celku

Sloučení může probíhat jednoduše nebo složitě. Záleží na kvalitě kódu a na štěstí.

Jednoduchý postup vypadá tak, že se vytvoří nový projekt, do něj se přidají odkazy na všechny slučované verze a ono to funguje. Pokud bylo rozšiřováno pouze přidáváním, půjdou pravděpodobně všechny modifikace opět přidat k sobě. Konflikty KSID jmen nevzniknou, díky jejich verzování a díky fixaci verzí v jednotlivých souborech.

Přidávání nových položek, jmen a závislostí nemá vliv na těla metod. Ta zůstanou funkční po jakékoli změně tohoto typu. Může ale vzniknout konflikt při dědění výlučných položek (třeba `override` metod) a může vzniknout cyklus KSID jmen. Oba případy pravděpodobně vznikají při špatném návrhu a půjde o řídký jev.

Komplikace nastanou, pokud jedna z verzí nahradila nějaký existující soubor. Tuto modifikaci je třeba zkontrolovat, případně opravit a přidat jí do top-level projektu, aby byla platná pro všechny větve.

Další problém může být v tom, že modifikace sice pracují správně se základní verzí, ale neumí komunikovat mezi sebou.

Výše popsané konflikty a chyby je nutné při slučování verzí opravit. (Samozřejmě bez zásahu do existujících souborů.)

5 Vývoj jazyka Krkal C

Tato kapitola popisuje rozdíly mezi jazykem Krkal C 2.0 a jazykem Krkal C 3.0. Vysvětluje principy, které formovaly podobu jazyka a zamýšlí se nad dalším vývojem i nad alternativními řešeními.

5.1 Vývoj původního jazyka

Vlastnosti původního jazyka určila již mnohokrát zmiňovaná hra Krkal. Bylo třeba navrhnout jazyk tak, aby umožnil jednoduše popsat prostředí žijících, vzájemně komunikujících objektů, a také, aby bylo možné tento systém jednoduše rozšiřovat.

Myšlenka jednoduchého čistě procedurálního jazyka byla brzy zamítnuta. Takový jazyk by nepřinesl nic nového a nijak by neusnadňoval vývoj simulačních her. Představit si herní prvky jako objekty je přirozené.

Důležitým požadavkem byla rozšiřitelnost. Přidávání nových prvků mělo být co nejjednodušší, bez nutnosti zasahovat do existujícího kódu, především bez nutnosti měnit kód uvnitř existujících metod. Tyto vlastnosti umožňují nejen nezávislý vývoj více uživateli, ale obecně výrazně zjednodušují přidávání nových prvků. Čím provázanější systém jazyk popisuje, tím užitečnější jsou vlastnosti usnadňující rozšiřitelnost.

Prvním prvkem podporujícím rozšiřitelnost byly množiny. Ve hře Krkal je typické, že pravidla platí vždy pro nějakou množinu objektů. Například magnet přitahuje objekty přitahované magnetem, do díry padají jen některé objekty, fotobuňka reaguje na určité objekty, atd. Díky množinám nemusí být v metodách pevně zadrátované podmínky, které by bylo třeba ručně upravovat při každé změně. Jazyk dovoluje metodám používat množinové testy a dovoluje průběžně rozšiřovat množiny o nové prvky.

Možnost rozšiřovat třídy v nových souborech o nové položky je důležitá pro nezávislý vývoj. Nezávislý uživatel tak může přidávat nové věci, aniž by musel modifikovat existující soubory.

Safe metody podporují rozšiřitelnost v mnoha ohledech. Volání je volné, argumenty specifikuje volající. Je možné volat i neexistující metodu. Runtime při vyhledávání metody používá princip zobecňování. Volaný objekt může mít více implementací dané safe metody. Volají se všechny metody (implementace), které má skutečný typ volaného objektu.

Volná vazba mezi volaným a volajícím minimalizuje nutnost dělat změny v kódu, když například přibude nový argument. Možnost mít více implementací minimalizuje nutnost měnit existující těla metod.

Například objekt hlavní hrdina může mít safe metodu `OnMoveEnded`, ve které si naplánuje pohyb na další políčko. V budoucnu se objeví sebratelné předměty a metoda `OnMoveEnded` může být rozšířena o sbírání předmětů. Aby nebylo nutné měnit stávající metodu, je možné naprogramovat metodu novou, stejného jména.

Hlavní hrdina by si mohl ukládat všechny sebratelné předměty do jednoho kontejneru. Nebo by mohl mít pro každý typ předmětu novou metodu pro sbírání (`OnMoveEnded`) a pro každý předmět by měl speciální proměnnou, do které by si ukládal jejich počet. Celkem pěknou vlastností budoucích verzí jazyka by mohlo být

pole indexované množinovými jmény. Pak by bylo možné mít jednu metodu pro sbírání a počty sebraných předmětů uchovávat v tomto poli.[7]

Systém zasílání zpráv se vyvinul spolu se safe metodami. Jestliže runtime převádí parametry mezi volajícím a volaným, může je jednoduše i uložit do fronty a voláni pozdržet. Zprávy v Krkalovi představují velmi silný a obecný prostředek. Zprávy se volají synchronně, všechny metody jsou vykonávány jedním vláknem. Runtime nemusí řešit problémy se synchronizací a program v Krkalovi se chová deterministicky v tom smyslu, že je vždy jasné, kdy se která zpráva vyzvedne. Při návrhu jazyka na multithreading mezi objekty myšleno nebylo.

Zprávy představují jiný přístup než například latentní (dlouho běžící) funkce používané v Unreal Scriptu [6] nebo v jazyce Simula. Tyto jazyky umožňují pracovat se synchronizačními primitivy, blokováním a příkazy typu `Sleep()`.

Vícenásobná dědičnost byla do jazyka přidána až později, když se ukázalo, že rozdělení na objekty, které nepodporují dědičnost, by vedlo ke kódu psaného stylem copy+paste. Dědičnost byla navržena tak, aby splynula s množinovými vztahy, bylo tedy přirozené, že bude vícenásobná. Vícenásobná dědičnost je komplikovanější na implementaci, ale reálněji vystihuje skutečný svět. Jednoduchá dědičnost by hlavně nezapadala do filosofie rozšiřování.

Základní principy jazyka Krkal C a Systému Krkal jsou asi nejpodobnější systému Game Maker [5]. Jde hlavně o čistě objektově orientovaný přístup, komunikaci pomocí zpráv či událostí a o samostatný popis tříd, grafiky a herního plánu (levelu).

5.2 Krkal C verze 2.0

Původní jazyk byl objektově orientovaný. Záleželo na pořadí deklarací a pořadí kompilace jednotlivých souborů. (Verzování souborů bylo tehdy řešeno jinak.) Jazyk obsahoval i základní direktivy preprocesoru, podobně jako jazyk C. (Direktivy se neukázaly jako užitečné a tak v nové verzi už nejsou.)

Jazyk umožňoval deklarovat jména, závislosti, globální proměnné a třídy. Třída mohla obsahovat členské proměnné, safe a direct metody, skupiny a ovládací prvky. Chyběly statické metody, override metody, inicializace proměnných přímo v deklaraci a syntaxe byla trochu jiná. Ke třídám, jménům i položkám se daly přiřazovat atributy.

Jazyk už tehdy obsahoval KSID jména, jenom nebyla používána tak důsledně jako nyní. Například členské proměnné a direct metody nebyly identifikovány KSID jménem. Způsob lokalizace ke kontextu se řešil jinak (ne tak důkladně, elegantně a složitě jako nyní) a obsahoval chyby, které mohly vést ke změně významu kódu metod při rozšiřování.

Příklad původního jazyka:

```
////////////////////////////////////////////////////////////////////
// Vlastní elektrický a životu nebezpečný výboj

object oProud {
    objptr Otec1, Otec2;
    char @CollisionCfg;
    constructor() {
        Otec1 = onull; Otec2 = onull;
```

[7] Uživatel by si už nyní takový kontejner mohl sám naprogramovat jako hashovací tabulku. Stačila by k tomu systémová metoda `GetHashCode` nad typem `name`.

```
            @CollisionCfg = @eKCConeCell|@eKCCnothing;
        }

        // ....

        void ::DeaktivujIDruhyho() {
            if (Otec1) Otec1->oElektroda::Deactivate() message;
            if (Otec2) Otec2->oElektroda::Deactivate() message;
        }

        objptr triger; // triger pro zabíjení objektů

        void @MapPlaced() {
            triger = onull;
            if (@IsEditor()) return;
            triger = new oAreaTrigger;
            if (smer == Sever)
                triger->SetPosSz(::X1:@ObjPosX-7,::X2:@ObjPosX+7,
                ::Y1:@ObjPosY-20,::Y2:@ObjPosY+20);
            else
                triger->SetPosSz(::X1:@ObjPosX-20,::X2:@ObjPosX+20,
                ::Y1:@ObjPosY-7,::Y2:@ObjPosY+7);
            triger->SetClzGr(::AddGr:BlokujeProud, ::Redirect:this);
            @PlaceObjToMap(triger);
            @MvConnectObjs(triger,this);
        }

        void @TriggerOn(objptr @Object) {
            if (typeof(@Object) <= ZnicitelneProudem)
                @Object->ZnicSeProudem() timed 165;
            if (Otec1) Otec1->oElektroda::Prerus(pObject:this);
            if (Otec2) Otec2->oElektroda::Prerus(pObject:this);
        }

        void @MapRemoved() {
            delete triger;
            triger = onull;
        }

    }
```

Implementace vícenásobné dědičnosti, která byla tehdy použita, umožňovala přístup ke členským proměnným pouze uvnitř metod třídy.

Typový systém vycházel z jazyka C. Základními typy byly `int`, `char`, `double`, `name` a `objptr`. (Ve starém jazyce byl jen jeden typ pro pointery na objekty a nebylo nutné přetypovávat objekty z jednoho typu na jiný.)

Jazyk dále obsahoval pointerovou aritmetiku, stejnou jako v C. Protože třídy neumožňovaly přímý přístup ke členským proměnným, existovaly navíc struktury. Bylo například možné přes dvojitý pointer procházet spojový seznam, pracovat s poli nebo alokovat a uvolňovat paměť.

Nechyběl typ `string`, který byl tak nešikovně navržen, že nakonec nebyl nikdy plně implementován.

Už ve starém jazyce se objevila pole proměnné velikosti.

5.3 Krkal C verze 3.0

Možná hlavní vylepšení jazyka je jeho zjednodušení. Ale také zaměření na bezpečnost a dopracování věcí okolo rozšiřitelnosti, verzování a KSID jmen.

Typový systém

Typový systém byl radikálně změněn a zjednodušen. Typy `int`, `double` a `name` zůstaly, `char` byl zvětšen na 2 Byty. Pointery, pointerová aritmetika, struktury a pole z jazyka C byly odstraněny. Pole proměnné velikosti bylo vylepšeno a použito jako základní typ. Typ `string` byl zrušen a nahradilo ho pole proměnné velikosti – `char[]`.

Třída nyní dovoluje přístup ke svým členským proměnným a proto mohla plně nahradit strukturu. Aby to bylo možné, musela být jinak řešena vícenásobná dědičnost (viz Vícenásobná dědičnost) a typ `objptr` musel být nahrazen hierarchií tříd.

Výhoda hierarchie typů pro reference na třídy je i v tom, že kompilátor může už během kompilace ověřit zda daná položka je objektem vlastněna či nikoli.

O paměť se nyní stará garbage collector, takže není nutné (ani možné) uvolňovat objekty a pole.

Byl zaveden typ `null`, který obsahuje nulu a je kompatibilní se všemi typy. U volání safe metody, kde typ parametru určuje volající, bylo totiž potřeba vědět zda 0 je typu `int` nebo `double[]` a proto existovalo 9 speciálních konstant pro nulu. Zavedení typu `null` umožnilo jejich zrušení.

Krkal nyní obsahuje velmi jednoduchý systém typů, téměř minimální. Jednoduchost typů je pro skriptovací jazyk vhodná a usnadňuje implementaci kompilátoru, runtimu a celého systému.

Navíc tento jednoduchý systém umožňuje úplnou reflexi. Runtime nebo editor levelů může procházet data jakéhokoli typu a rozumí jim. To u starého jazyka, který obsahoval pointery, nešlo. Jedna z důležitých funkcí runtimu je schopnost serializovat veškeré žijící objekty a uložit jejich data například na disk. Nový runtime to může dělat naprosto automaticky. Starý runtime potřeboval podporu skriptů, třídy, které používaly pointery, musely mít naprogramovánu vlastní serializaci.

Jazyk je staticky typován, tedy kompilátor musí umět určit typ každého výrazu nebo podvýrazu. Skriptovací jazyky často bývají typovány dynamicky, to znamená, že typ proměnné není znám v době kompilace, ale až za běhu. Například výraz

```
a = b + c;
```

může jednou znamenat sčítání intů, podruhé konkatenaci stringů. Záleží na tom, jaká přijdou data. Krkal touto cestou nejde. Staticky typovaný jazyk je méně náchylný na chyby. To, že proměnná je určitého typu, zaručuje, že i její hodnota bude toho typu. Staticky typovaný jazyk je rychlejší, runtime nemusí za běhu měnit podle typů své chování. A je kompatibilní s C++, což je klíčové pro kompilaci skriptů do jazyka C++.

Přesto i Krkal by potřeboval nějaký typ `object` nebo `variant`, do kterého by šly uložit všechny ostatní typy. Takový typ v jazyce zatím není. Společný předek všech typů je potřeba pro tvorbu kolekcí.

Jazyk obsahuje všechny prostředky nutné pro tvorbu kolekcí (spojových seznamů, stromů, hashovacích tabulek, ...), ale absence univerzálního typu a absence šablon, znesnadňují tvorbu knihovny kolekcí. Pro každý typ by bylo třeba kolekci napsat znovu.

Generické programování by určitě bylo nejelegantnější, ale jeho implementace by byla pravděpodobně velmi složitá.

Naštěstí pole v Krkalovi samo o sobě dokáže zastoupit několik nejzákladnějších kolekcí: rozšiřitelné pole, frontu a zásobník.

Změny v syntaxi

Zjednodušena a přepracována byla i syntaxe. Například používání výrazu typu `name` při volání metod nebo při konstrukci objektů. Ubylo čtyřteček `::`, přibyl znak `$` uvozující plně kvalifikovaná KSID jména nebo příkaz `foreach` ...

Jazyk nově vyhodnocuje identifikátory různým způsobem podle kontextu, což zabraňuje konfliktům jmen proměnných a jmen typů a usnadňuje to psaní kódu. Například v těle metody lze napsat tyto deklarace:

```
a a = new a();
a b = new a();
a c = a;
```

Konstrukce objektů je nyní sjednocena se safe voláním. To znamená, že konstruktoru lze předávat parametry jako jakékoli jiné safe metodě, dokonce je možné konstrukci objektu zpozdit pomocí zasílání zpráv.

Část syntaxe jazyka je nyní konfigurovatelná (typy jmen, skupin a ovládacích prvků, definice atributů a systémových metod).

Bylo nově navrženo volání knihovních funkcí. Vznikly systémové metody a externí metody.

KSID jména

KSID jména jsou specifický prvek jazyka Krkal C. Jazyk potřebuje prostředek, který umožní globálně identifikovat veškeré entity. KSID jména řeší problémy při rozšiřování (stabilně přidělované verze pomáhají řešit konflikty jmen).

Jména jsou strukturovaná. Aby uživatel nemusel psát jména celá, kompilátor mu napomáhá tím, že jméno (podle kontextu) lokalizuje k nějakému prefixu. Uvnitř třídy je většinou platný prefix třídy a proto se položky dané třídy dají identifikovat krátkými jmény. Bohužel pro položky zděděné od předků už to neplatí. V jazyce je nutné, aby identifikace položek byla jednoznačná a neměnná během rozšiřování (a to i v případě vícenásobné dědičnosti).

Uživatel je nucen identifikovat položky předků dlouhými, plně kvalifikovanými jmény. Protože pracovat s dlouhými jmény není příjemné, brzy začne uživatel využívat KSID jména, která jsou tvořená jen jedním identifikátorem. To může, tato jména jsou zamýšlena třeba pro pojmenovávání frekventovaných safe metod, které jsou často volány z různých tříd. Ale pokud bude uživatel tato globální jména používat příliš často, způsobí konflikt jmen.

```
class name Base, Derived;
depend Base << Derived;

class Base {
    void $GlobalName() {}        // $GlobalName
    void PrivateName() {}        // $Base.PrivateName
}

class Derived {
    void Method() {              // $Derived.Method
        $GloalName();            // pristup k polozkam predka
        $Base.PrivateName();
        Method();                // pristup k me vlastni polozce
```

```
        }
    }
```

5.4 Plány do budoucna

Jazyku chybí univerzální typ a díky tomu je obtížné napsat knihovnu kolekcí. Viz také Typový systém.

Původní jazyk měl všechny členské proměnné přístupné jen z vnitřku objektu. V současné verzi jsou naopak všechny veřejné. Jazyku chybí zapouzdření tříd, nějaký systém práv. Systém práv výrazně napomáhá orientaci mezi položkami třídy. V současné verzi práva úmyslně chybí. Jazyk je příliš nový, je třeba ho nejprve důkladněji vyzkoušet a pak navrhnout systém práv tak, aby neohrozil rozšiřitelnost. Bohužel rozšiřitelnost může být ohrožena právě tím, že absence práv umožní špatný styl psaní kódu.

Zajímavou otázkou jsou property. Property je termín pro položku třídy, která se navenek tváří jako proměnná, ale uvnitř je implementována jako dvojice funkcí, které umožňují číst a zapisovat hodnotu. Mají být property řešeny pomocí direct metod nebo safe metod? Safe metoda má na property příliš velkou režii, ale také spoustu užitečných vlastností. Možná v Krkalovi bude vypadat property zcela netradičně: Může být implementována pomocí dvojice direct metod a volitelné safe metody, která bude obsluhovat událost změnění property.

Užitečné by také bylo zavedení abstraktních tříd. Při intenzivním používání vícenásobné dědičnosti může situace vypadat tak, že funkcionalita je pouze v abstraktních třídách. Třídy, které mají popisovat žijící objekty, jsou pak sestaveny jen kombinací předem připravených abstraktních tříd. Tato vlastnost by zvýšila přehlednost a také efektivitu runtimu.

Velké množství vylepšení si pořád vyžadují pole. To, že jde o referenční typ, který se sám alokuje, může uživatele zmást. Dále si uživatelé musejí dávat velký pozor na stringy. Pole je vždy předáváno referencí a každý, komu je předáno, může změnit jeho obsah, změna se pak projeví všude. Řešením by mohla být možnost pole po vytvoření zamknout a vytvořit tak nezměnitelný objekt. Další možností je vytvořit nový typ – konstantní pole.

Jazyk neobsahuje výjimky. Otázkou je, jak moc jsou pro tento jazyk užitečné. Ve hře Krkal zatím většinou potřeba nebyly.

Dalším vylepšením může být typ delegát. Jazyk už nyní obsahuje obrovskou svobodu, jak volat safe metody. Pro volání je třeba zadat řadu parametrů: objekt, jméno metody a pro každý parametr jeho jméno, typ a hodnotu. Delegát by mohl tyto data sloučit do jednoho celku a umožnit snadné volání takto zadaných metod.

Celou řadu vylepšení lze navrhnout i pro safe metody a zprávy. Užitečná je například možnost zprávy přesměrovat na jiný objekt(y) bez nutnosti zprávu obsloužit a vyvolat jinou. Zajímavé je i programování řízené stavy objektů, jak popisuje Sweeney [6] ve specifikaci Unreal Scriptu. Každý objekt se v daném okamžiku nachází v určitém stavu. Při volání metody se zavolá jen ta implementace, která aktuálnímu stavu odpovídá. Tato technika eliminuje příkazy `switch` a přispívá rozšiřitelnosti.

V jazyce není přetěžování operátorů ani metod. Jeho absence zjednodušuje jazyk a implementaci kompilátoru. Přetěžování metod je nahrazeno safe metodami. U přetížených metod kompilátor vybírá jednu metodu, ze skupiny metod stejného jména,

podle seznamu parametrů použitého při volání. Naproti tomu safe metody jsou zavolány všechny a každá z metod si vybírá ze seznamu parametrů ty, které potřebuje. Tělo safe metody může ověřit, které argumenty byly skutečně zadány, a proto jedna safe metoda může plně nahradit skupinu přetížených metod.

6 Implementace kompilátoru

Následující kapitola přiblíží implementaci kompilátoru. Nejprve vysvětlí, v jakém prostředí je kompilátor používán a jaká rozhodnutí ovlivnila jeho implementaci. Následně popíše obecné principy kompilace, cacheování a rozdělení kompilace na jednotlivé fáze a kroky. Netriviálnější části a algoritmy jsou vysvětleny do větších detailů.

6.1 Použití kompilátoru

Úkolem kompilátoru je přeložit zdrojový kód jazyka Krkal C do výstupního kódu nebo do výstupních dat. Výstup kompilátoru se může značně lišit.

Kompilátor je od začátku navrhován tak, aby uměl spolupracovat s IDE, kompilátor musí umět rychle překládat zdrojový kód a poskytovat IDE informace, které potřebuje – to je zejména typová informace (popis tříd), která najde uplatnění v „class view“, a informace potřebná pro kontextovou nápovědu během psaní kódu (automatické doplňování kódu a nápověda parametrů).

Výstupem kompilátoru jsou i chybová hlášení, jsou opět navržena tak, aby IDE mohlo už během psaní automaticky podtrhávat chyby.

Kompilátor negeneruje přímo spustitelný kód. Filosofie Krkala 3.0 je postavená na vyměnitelných komponentách, kompilátor má být použitelný v různých prostředích a podle použití se může lišit i výstup. Kompilátor generuje pouze obecná, snadno použitelná data a nechává další krok na jiné komponentě – na generátoru kódu.

Ve starém Krkalovi byl výstupem (kromě typové informace) jednak kód v jazyce C++, který byl dále překládán C++ kompilátorem, a dále mezikód podobný assembleru, který byl interpretován. Metody mohly být přeloženy oběma způsoby a runtime umožňoval běh obou druhů kódu a uměl transparentně volat metodu jednoho světa z metody druhého světa.

Nyní kompilátor generuje data, která obecně popisují kód metod, a umožňuje připojení libovolného generátoru kódu, který pak může generovat C++, mezikód, přímo strojový kód nebo kód nějakého jiného jazyka, který je dostatečně silný, aby kompilované skripty zvládl (C#, Java?). Dokonce je možné generátor kódu nepřipojovat vůbec, ale připojit přímo interpret, který by si vyžádal překlad metody právě tehdy, kdy by byla zavolána, a interpretoval by její příkazy přímo podle dat poskytnutých kompilátorem. Současná verze obsahuje generátor kódu do jazyka C++.

Zadání diplomové práce předpokládalo, že i nový kompilátor bude podobně jako ten původní generovat jak C++ kód, tak interpretovatelný mezikód. Ale došlo zde k posunu. Nový návrh je jednak obecnější (lze připojit libovolný generátor kódu a generovat tak libovolný výstup), jednak jednodušší. Protože proces kompilace do C++ byl zpřístupněn všem uživatelům a byly odstraněny všechny jeho nevýhody, paralelní existence interpretovatelného mezikódu přestala být potřebná.

Kompilátor negeneruje výstup do žádného externího souboru. Výsledky kompilace jsou pouze v paměti a ostatní komponenty k nim mají přístup přes veřejný interface kompilátoru, který je založen na třídách jazyků .NET.

6.2 Obecný návrh kompilátoru

Než započaly práce na kompilátoru, bylo třeba učinit několik důležitých rozhodnutí.

Bylo rozhodnuto, že ze starého kompilátoru nebude použito nic a nový kompilátor bude navržen a implementován znovu. Důvody jsou následující: Nový a starý jazyk se značně liší. Architektura původního kompilátoru nebyla optimální, nový kompilátor kompiluje jinak, v jiném pořadí, využívá cacheování a podobně. Starý kompilátor obsahoval chyby.

Jako programovací jazyk byl vybrán jazyk C#. Zkušenosti s implementací původního kompilátoru ukázaly, že největší obtíže způsobují chyby v programu. Protože kompilátor je komplexní komponenta s vysokou náchylností k chybám, byl zvolen C# jako jazyk, který pomáhá minimalizovat množství programátorských chyb. Další výhodou je zázemí .NET Frameworku, především generické kolekce, na kterých je kompilátor postaven. Nevýhodou je, že pro C# existuje daleko méně nástrojů, usnadňujících lexikální nebo syntaktickou analýzu, než pro C++. Další nevýhodou je nutnost propojovat staré komponenty z původního Krkala, které jsou psány v C++ s komponentami psanými v C#. Výhody jazyka C# ale přesto značně převažují nad nevýhodami.

Bylo rozhodnuto, že kompilátor bude obecně použitelná komponenta s konfigurovatelnými prvky. Kompilátor bude představovat pouze frontend, backend, tedy generátor kódu, bude řešen samostatnou komponentou.

Návrh kompilátoru vychází z klasického schématu, popsaného v [1]. Postup kompilace je navržen tak, že kompilátor kompiluje jen to, co je třeba. Například pokud IDE potřebuje typovou informaci, kompilátor kompiluje jen deklarace položek tříd, nikoli těla metod nebo inicializace proměnných. Kompilace je víceprůchodová s náhodným přístupem. Například kompilátor přeloží deklarace položek, přitom si zapamatuje, kde jsou těla metod, a pokud později bude potřeba kompilovat metodu, kompilátor začne od místa, které si zapamatoval. Kompilace jednotlivých souborů je cacheovaná.

Další otázkou bylo použití nástrojů usnadňující lexikální a syntaktickou analýzu. Obzvláště pro lexikální analýzu by bylo vhodné použít existující řešení, protože syntaxe lexikálních tokenů je v Krkalovi téměř stejná jako v jazycích C# nebo C. Bohužel se ukázalo, že čas potřebný na nalezení a integraci takového řešení by byl delší než implementace vlastní lexikální analýzy. (Nástroje jsou většinou psány v C/C++ a jejich integrace do .NET prostředí si vyžaduje úsilí navíc.) Výhoda vlastního řešení je v plné kontrole a v možnosti maximálního přizpůsobení.

U syntaktické analýzy byl preferován jednoduchý, přirozený přístup metodou shora dolů, převážně z následujících důvodů: Jazyk Krkal C má jednoduchou syntaxi, ale složitou sémantiku. Při analýze je důležité mít informaci o kontextu. Kompilátor navíc může snadno kompilovat jenom části kódu, například kompilovat typovou informaci, ale vynechávat těla tříd, ty pak kompilovat později. Obavy, že by tento přístup byl neefektivní nebo vedl ke špatně udržovatelnému kódu se naštěstí nenaplnily.

6.3 Kompilátor, první přiblížení

Kompilátor obsahuje tři hlavní třídy:

- Singleton `KrkalCompiler` – ten obsahuje pouze cache a odkaz na objekt typu `CustomSyntax`, který slouží ke konfiguraci veškeré volitelné syntaxe a dalších parametrů, mimo jiné je zde třeba zadat delegáta, který tvoří instanci generátoru kódu.
- Statickou třídu `CompilerConstants`, která obsahuje například metody pro zjištění zda text je klíčové slovo, zda text je operátor a podobně.
- Třídu `Compilation` – Tu je třeba vytvořit, jako parametr konstruktoru zadat jméno souboru s projektem a zavolat metodu `Compile()`. Tímto se provede kompilace, součástí které může být i generování kódu. Výstup, tedy typová informace a seznam chyb, je přístupný také z tohoto objektu.

Kompilace probíhá v několika krocích. V některých krocích kompilátor analyzuje zdrojové soubory. Kompilátor má přístup k libovolné části libovolného souboru a může začít analýzu z kteréhokoli místa. Lexikální analýza, syntaktická analýza i část sémantické analýzy jsou propojeny v jeden procesní řetěz. V jiných krocích je prováděna pouze sémantická analýza.

Syntaktická analýza

Analýza je řízena shora. Kompilátor není založen na automatu, který by byl vygenerován podle pravidel gramatiky, ale na ručně psaných funkcích, které vyhodnocují gramatická pravidla, ověřují jejich správnost, rozhodují o dalším postupu a generují výstupní data. Kompilátor nepracuje s formálně zadanou gramatikou[8]. Pro přiblížení jeho práce je možné použít gramatiku, kde přepisovací pravidla na levé straně obsahují jeden neterminál a na pravé straně regulární výraz složený z neterminálů i terminálů. Každý neterminál je na levé straně právě jednoho pravidla.

Příklad:

Příklad obsahuje ilustraci několika přepisovacích pravidel. Neterminály jsou zapsány jako slova začínající velkým písmenem. Kulatá závorka slouží k označení skupiny, hvězdička označuje, že předchozí část se opakuje n krát, kde $n \geq 0$. Alternativy jsou naznačeny svislítkem | a otazník představuje nepovinný výskyt. Ostatní znaky jsou terminály.

```
File -> Header Content
Header -> (Head|Headattr|Nametable)*
Head -> #head {(Headstatement)*}
Headattr -> #attributes Attributes
Attributes -> [(Attribute(,Attribute)*)?]
Content -> (Namedeclaration|Dependency|Class)*
Class -> class Ksid (Attributes)? {(Field)*}
Field -> (Modifier)* Type Ksid (Argumentlist)? (Attributes)?
(Variable|Method|Group|Control)
```

Gramatika může být definována různými způsoby (třeba bez použití regulárních pravých částí) a stále generovat tentýž jazyk. Takto zapsaná gramatika nejvěrněji vystihuje současnou implementaci kompilátoru. Kdyby kompilátor používal LR analýzu (analýzu zdola nahoru), bylo by třeba zapsat gramatiku v jiné formě.

[8] Gramatika je definována množinou neterminálů V_N, množinou terminálů V_T, počátečním stavem $S \in V_N$, a množinou přepisovacích pravidel. Slovo $\alpha \in V_T^*$ je obsaženo v jazyce, generovaného gramatikou, pokud v gramatice existují přepisovací pravidla, která odvodí slovo α z počátečního stavu S. Analogicky text t je programem jazyka Krkal C, pokud v gramatice jazyka Krkal C existují přepisovací pravidla, pomocí kterých lze text t odvodit z počátečního stavu S. V opačném případě text t obsahuje syntaktické chyby.

Na příkladu jsou vidět i místa, kde dochází k větvení, jde o výčty alternativ a o opakující se sekvence, kde je potřeba určit počet opakování. V této gramatice neexistují alternativy mezi pravidly, protože každý neterminál je na levé straně právě jednoho pravidla. Kompilátor prochází strom shora dolů a zleva doprava jedním průchodem. Aby toto bylo možné musí se kompilátor umět v každém okamžiku rozhodnout mezi všemi alternativami, které gramatika nabízí.

Kompilátor průběžně kontroluje správnost odvození v textu. Nechť se v textu nachází na pozici p. Pak nalevo od p je už zpracovaný text $t_1 \in V_T^*$ a kompilátor rozepsal počáteční neterminál S pomocí přepisovacích pravidel na text $t_1 n$, kde n jsou rozpracované pravé strany pravidel, tedy regulární výraz, který obsahuje terminály i neterminály. Kompilátor postupně zpracovává pravou stranu aktuálního pravidla.

- Jestliže narazí na terminál x, zkontroluje, zda i zdrojový text obsahuje x a posune pozici p o jedna doprava.
- Jestliže narazí na neterminál N, přeruší práci na aktuálním neterminálu a rekurzivně začne zpracovávat N.
- V případě, že regulární výraz nabízí alternativy, kompilátor si musí jednu vybrat.
    - Kompilátor může buď použít výhled. Pro každou alternativu A má kompilátor k dispozici množinu FIRST(A), která obsahuje všechny možné začátky alternativy A (začátky jsou sekvence terminálů, kterými může alternativa po odvození začínat). Kompilátor se podívá na dosud nezpracovaný terminál t_p na pozici p (většinou se používá výhled délky 1) a zvolí takovou alternativu A, kde $t_p \in$ FIRST(A). Množiny FIRST musí být disjunktní, aby nedocházelo ke konfliktům[9]. Příkladem může být pravidlo `Header -> (Head | Headattr | Nametable)*`, kde se kompilátor musí umět rozhodnout mezi třemi variantami a ukončením cyklu. Každá z variant může začínat pouze určitým klíčovým slovem (`#head`, `#attributes` nebo `#names`), proto, pokud výhled bude obsahovat jedno z těchto klíčových slov, zvolí kompilátor odpovídající variantu, jinak vyskočí z cyklu.
    - Rozhodování mohou ovlivnit i už zpracované terminály. Například u zpracovávání položky se může na místě typu objevit klíčové slovo `group`, pak je jasné že tato položka se bude zpracovávat jako skupina a není potřeba se rozhodovat podle výhledu.

Protože kompilátor nepracuje s exaktně definovanou gramatikou, ani výše uvedený příklad není přesný. Například pravidlo

```
Header -> (Head|Headattr|Nametable)*
```

by bylo třeba rozšířit o tyto dodatečné podmínky: Část `#head` je povinná. Každá z částí se může vyskytovat maximálně jednou.

Podobné je to i s pravidlem popisujícím položku. Pravidlo začíná částí, která dovoluje číst modifikátory, kompilátor v tento okamžik ještě neví, zda půjde o proměnnou, metodu či ovládací prvek, tak modifikátory zatím čte všechny. Teprve později zkontroluje, zda pro daný typ položky je načtená kombinace modifikátorů platná.

[9] Gramatika je navržena tak, aby tato podmínka platila.

Implementace syntaktické analýzy

Analýza je řešena soustavou ručně programovaných funkcí, které představují jednotlivá gramatická pravidla. Funkce jsou volány rekurzivně, volání kopíruje analýzu shora dolů.

Aby nevznikl nepřehledný a špatně udržovatelný kód, existuje třída `SyntaxTemplates`, která obsahuje podpůrné funkce pro syntaktickou analýzu. Je zde poměrně komplexní funkce, která analyzuje cykly v gramatických pravidlech (Detekuje začátek a konec cyklu, hlídá oddělovače, prázdné příkazy, ale i vyhodnocení chyb a zotavení se z chyb.) Dále například funkce, která načítá KSID jméno, nebo funkce, která čte středník a když ho nenajde, tak nahlásí odpovídající chybu.

Lexikální analýza

Lexikální analýza zpracovává zdrojový text do tokenů, které jsou dále využívány syntaktickou analýzou. Analýza lexikálních tokenů je inspirována gramatikou popsanou ve specifikaci jazyka C# [4].

Každý token (třída `LexicalToken`) obsahuje typ, data a svou pozici v textu (včetně velikosti tokenu). Funkce lexikální analýzy vždy vracejí token, ani v případě chybových stavů (třeba při čtení mimo soubor) nevyvolávají výjimky, některé chyby jsou ovšem logovány.

Kompilátor pracuje s následujícími typy tokenů: `Keyword`, `HeaderKeyword`, `Operator`, `Identifier`, `Char`, `Int`, `Double`, `String`, `VersionString`, `Error` a `Eof`. Tokeny tedy představují klíčová slova, operátory, literály (konstanty), identifikátory – lexikální analýza načítá celý strukturovaný identifikátor, včetně teček a uvozujících znaků `@` a `$` – a chybové stavy.

Lexikální analýza (třída `Lexical`) pracuje nad souborem s textem. Většina funkcí se vztahuje k aktuální pozici v textu. Tuto pozici je možné měnit a to jak číselným zadáním, tak zadáním pozice od určitého tokenu.

Nejdůležitějšími funkcemi jsou funkce `Read`, která přečte token a posune pozici na další, a funkce `Peek`, která pouze přečte token, ale pozici neposouvá. Lexikální analýza obsahuje i další funkcionalitu, například `Peek` n-tého tokenu nebo přeskakování částí kódu.

Analýza automaticky přeskakuje bílé znaky a komentáře. Tokeny jsou cacheovány pomocí spojového seznamu. Token je tvořen ze zdrojového textu jen jednou, při prvním dotazu, všechny následující dotazy vracejí už vytvořený token.

Hlášení chyb

Kompilátor obsahuje i třídu `ErrorLog`, která složí k logování chyb. Záznam o chybě může obsahovat, kromě běžného popisu chyby a čísla chyby i pozici ve zdrojovém souboru. Tato pozice obvykle bývá zadána tokenem, na který je chyba vázána.

Chyby je možné buď pouze logovat, nebo logovat a vyvolat výjimku. Výjimka je vyvolávána tehdy, jestliže chyba znemožňuje na daném místě pokračovat. Kompilátor se snaží výjimky chytat a z chyb se zotavovat. Neskončí tedy s první chybou, ale raději se snaží v překladu pokračovat a poskytnout uživateli kompletní seznam chyb. Kvalita tohoto seznamu samozřejmě závisí na kvalitě zotavování. Může se stát, že kompilátor toho vynechá příliš, nebo díky zotavení na špatném místě, vzniknou falešné chyby.

Navíc vznikají zavlečené chyby čistě díky tomu, že kompilátor vyžaduje něco, co díky dřívější chybě neexistuje. Kompilátor řeší zotavování se z chyb téměř nejjednodušším možným způsobem, proto kvalita zotavování pravděpodobně nebude vysoká.

U syntaktické analýzy řeší zotavování třída `SyntaxTemplates`. Výjimka může být chycena funkcí, která analyzuje cykly. Ta pak přeskočí část kódu – hledá buď oddělovač (středník) nebo koncovou závorku `}` – a poté pokračuje v analýze od tohoto místa.

Sémantická analýza

Sémantická analýza přiřazuje význam jednotlivým syntaktickým prvkům. Část sémantické analýzy je prováděna společně s analýzou syntaktickou. Část je prováděna v samostatných krocích.

Sémantická analýza například vyhodnocuje identifikátory (význam identifikátoru se často mění podle kontextu) nebo kontroluje a vyhodnocuje typy.

Samostatným krokem je zařazování položek do tříd a dědění.

6.4 Fáze kompilace

Kompilátor pracuje v několika fázích či krocích. Kroky nemusí být vždy prováděny všechny, záleží na účelu kompilace. Kompilátor využívá dva druhy cache: cache jednotlivých souborů a cache celé kompilace.

Cacheování

Kompilátor není určen jen pro prosté přeložení zdrojového kódu do určitého výstupu, ale je používán i integrovaným vývojovým prostředím. IDE využívá kompilátor k získávání aktuální typové informace a tu pak prezentuje uživateli. Například uživatel může připsat do třídy novou proměnnou a udělat v zápisu syntaktickou chybu. IDE chybu podtrhne, uživatel ji následně opraví a nová proměnná se objeví v class view mezi ostatními členy třídy.

Aby toto bylo možné, musí IDE opakovaně spouštět kompilátor a číst z něj potřebné informace. Kompilace musí být dostatečně rychlá, aby nebrzdila práci uživatele, v ideálním případě by si uživatel neměl ničeho všimnout.

Při implementaci kompilátoru měly většinou přednost vlastnosti jako jednoduchost, přehlednost a funkčnost. Požadavek na rychlost byl většinou až na druhém místě. Například vyšší rychlosti by bylo dosaženo volbou C++, přesto je kompilátor psán v C#. Podobně by vyšší rychlost mohla přinést implementace lexikální a syntaktické analýzy pomocí generovaných automatů.

Rychlost kompilátoru je založena na myšlence: „Nekompiluj nic, co není potřeba.“ Například, jestliže IDE požaduje typovou informaci, kompilátor kompiluje jen položky tříd, není třeba překládat těla metod, inicializace proměnných ani atributy. Navíc uživatel od poslední kompilace provedl jen několik málo změn, proč tedy nevyužít informace z předešlé kompilace?

Nejrychlejší strategií by bylo vzít výsledky předchozí kompilace a promítnout do nich uživatelem provedené změny. Bohužel tento postup je implementačně náročný. I malá změna může kaskádovitě ovlivnit mnoho věcí. Může dojít ke změně typu položky nebo typu KSID jména (což ovlivní sémantický význam na mnoha místech), nová

složená závorka může znamenat úplně jiné uzávorkování bloků, stávající chyby mohou být opraveny, ale nové chyby mohou vzniknout. Vznik nebo zrušení závislosti může dramaticky ovlivnit dědičnost. Tato strategie byla zamítnuta pro přílišnou komplikovanost a náchylnost k chybám. Je těžké přesně definovat dopad jednotlivých změn.

Kompilátor využívá členění programu na soubory a do cache ukládá výsledky kompilace jednotlivých souborů. Soubory ovšem nefungují samostatně, jazyk Krkal C se vyznačuje vysokou provázaností. Do souborové cache se ukládají jen takové výsledky kompilace, které nejsou závislé na ostatních souborech ani na projektu, který je právě kompilován. Kompilace je rozdělena na několik kroků, v první fázi jsou samostatně překládány jednotlivé soubory. Informace z jiných souborů není využívána, proto výsledky této fáze zůstávají platné, i když dojde ke změně v jiných souborech. Ve druhé fázi jsou informace z jednotlivých souborů sloučeny do jednoho celku.

Jestliže uživatel změní soubor, je tento soubor znovu přeložen, u ostatních souborů je využita informace z cache. Změněný soubor je kvůli jednoduchosti překládán celý. I fáze slučování je kvůli jednoduchosti provedena celá znovu.

Kompilace je prováděna v krocích, kompilátor dělá jen ty kroky, které jsou potřeba, a naopak nedělá ty kroky, které už byly učiněny v minulosti. I výsledky celkové kompilace jsou cacheovány.

Například pokud si IDE vyžádá typovou informaci, nejsou už kompilována těla metod. Jestliže následně uživatel vyvolá úplnou kompilaci, kompilátor zjistí, že nebyly změněny žádné soubory. Může tedy vynechat všechny kroky, které byly provedeny předtím, a spustí pouze překlad metod a generování výstupu.

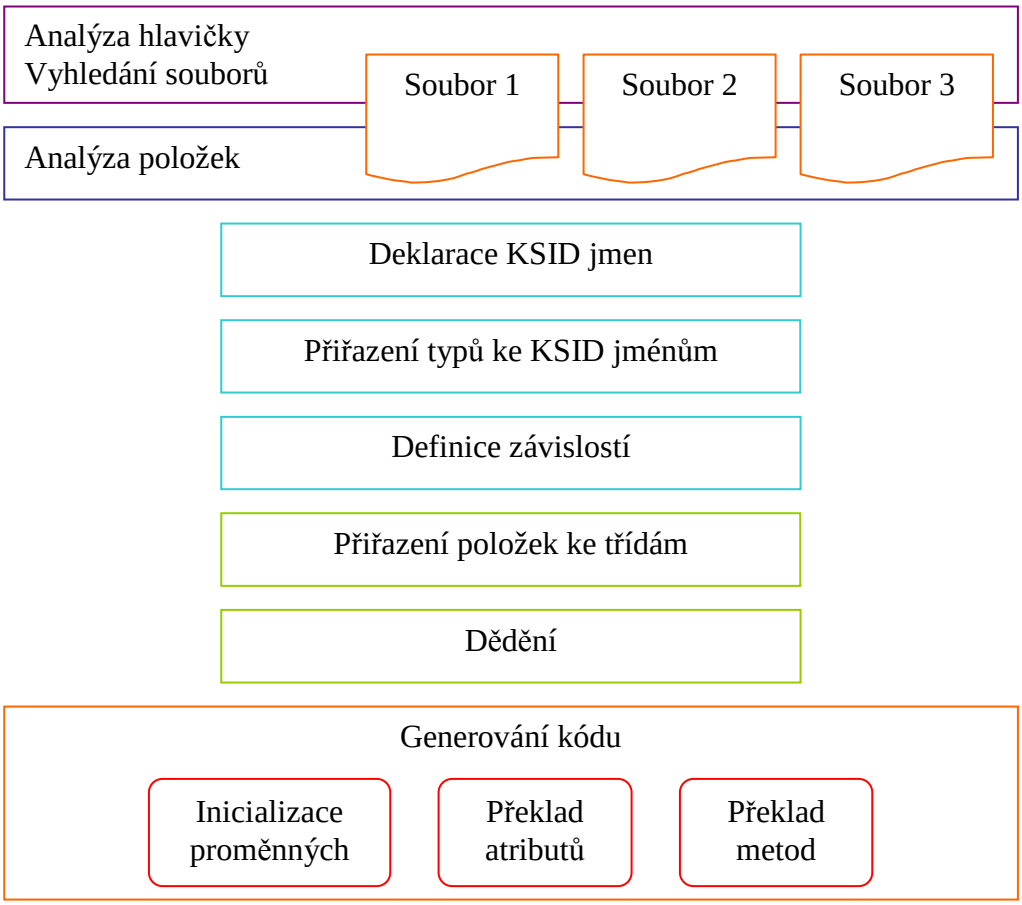
Kompilace souborů

V prvním kroku metody `Compile` třídy `Compilation` je třeba nalézt všechny soubory, které se kompilace účastní. Používá se algoritmus popsaný v kapitole Algoritmus výběru souborů.

Soubor je reprezentován třídou `Lexical`. Kompilátor nahrává soubory pomocí samostatné komponenty File System. Komponenta File System umí vyhledat soubor. Kompilátor neví, kde se přesně soubor nachází, protože pracuje jenom se jménem souboru, nikoli s cestou. Navíc soubor může být otevřen v editoru, pak komponenta File System poskytuje kompilátoru aktuální, právě editovanou, verzi. Dále komponenta informuje kompilátor, zde se soubor změnil od posledního načtení. Pokud ne, kompilátor nebude tvořit nový objekt `Lexical`, ale nahraje ho cache.

Třída `Lexikal` neobsahuje jen funkcionalitu pro lexikální analýzu, ale i odkazy na veškeré výsledky souborové kompilace. Kompilátor váže k souboru co nejvíce informací, jejichž význam nemohou ovlivnit změny v jiných souborech.

Jedná se o syntaktickou analýzu hlavičky a všech položek, tedy deklarací jmen, závislostí, tříd, proměnných, metod, skupin, ovládacích prvků a argumentů metod. V této fázi není možné provádět kompletní sémantickou analýzu, protože ta vyžaduje informaci ze všech souborů. Není například možné porozumět ani KSID jménům, nemusí být známy verze KSID jména ani jeho typ, proto si kompilátor zatím nemůže dávat KSID jména do souvislostí a musí sémantickou analýzu odložit na později. Důsledkem je i to, že kompilátor zatím neví, které položky patří do kterých tříd.



Obrázek 10: Fáze kompilace

Kompilace těl metod, inicializací a atributů je odložena.

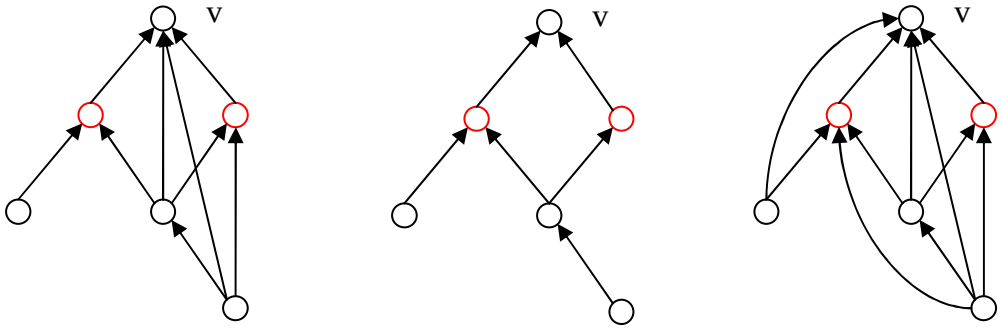
Například během výběru souborů, které se budou kompilovat, kompilátor potřebuje data z hlavičky souboru. Tato data jsou získávána syntaktickou analýzou a protože jsou závislá pouze na vlastním souboru, můžou být u souboru uložena. Proto, dokud se soubor nemění, kompilátor nemusí analyzovat hlavičku znovu.

V prvním kroku kompilátor nalezne soubory potřebné ke kompilaci a překládá hlavičku (pokud soubor nebyl cacheován). Kompilace může být nyní ukončena, jestliže cílem byl jen seznam souborů. Ve druhém kroku jsou překládány položky, ale je prováděna jen syntaktická analýza (opět jen pokud nebyla využita cache).

Graf KSID jmen

Základem typové informace je acyklický orientovaný graf KSID jmen. Třída, která graf reprezentuje, podporuje následující funkcionalitu:

- Přidávání a ubírání vrcholů.
- Přidávání a odstraňování hran (závislostí).
- Tranzitivní test závislosti. (Je předkem? Je potomkem?) Tento test je implementován pomocí hashovací tabulky, která je vytvořena u jednoho z vrcholů při prvním dotazu. Tabulka je naplněna všemi předky (či potomky) daného vrcholu, používá se prohledávání do hloubky. Při změně závislostí v grafu je třeba všechny tyto tabulky odstranit jako neplatné.
- Enumerace všech vrcholů.



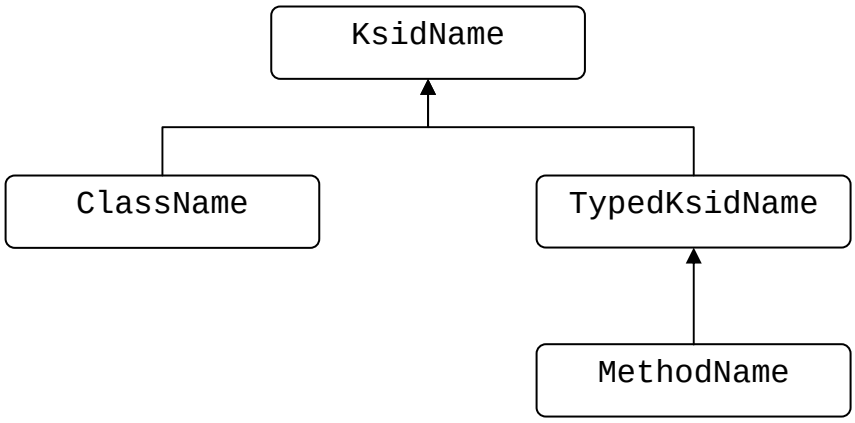
Obrázek 11: Graf a jeho tranzitivní redukce a tranzitivní uzávěr. Červeně jsou označeni nejbližší synové vrcholu v.

- Topologické setřídění grafu. Vrcholy jsou seřazeny topologicky, pokud pro každé dva vrcholy v_1 a v_2, v_1 je před v_2, platí, že z v_1 nevede hrana do v_2. Pokud jsou vrcholy topologicky uspořádány, tak hrany vedou jen jedním směrem. Topologické třídění je implementováno algoritmem, který je založen na prohledání do hloubky.
- Enumerace nejbližších synů (otců). Syn s_1 vrcholu v ($s_1 < v$) je nejbližší syn, pokud neexistuje žádný jiný syn vrcholu v, který by byl předkem s_1. Nejbližší syn vrcholu v, zůstane synem vrcholu v, i když by byla na grafu provedena tranzitivní redukce[10], viz také Obrázek 11. Použitý algoritmus testuje závislosti mezi každými dvěma syny, pracuje v čase $O(n^2)$, kde n je počet synů. Algoritmus testuje tranzitivní závislost pomocí předpočítané hashovací tabulky.
- Enumerace nejbližších potomků (předků), pro které platí predikát P. Jde o obecnější verzi předchozího bodu. Pokud pro vrchol neplatí predikát a přesto by splňoval podmínku na nejbližší vrchol, algoritmus se pokouší místo vrcholu přidávat jeho syny.

Tyto grafové algoritmy jsou dále využívány například pro výpočet dědičnosti, testování množinových vztahů nebo v IDE pro prezentaci stromových struktur. Je zde potřeba řešit problém tranzitivního uzávěru a tranzitivní redukce, což je výpočetně náročné. Například Gries a kol. navrhují pro tranzitivní redukci algoritmus pracující s maticí sousednosti v čase $O(n^3)$ [3]. Nevýhoda tohoto postupu je v paměťové náročnosti (matice o velikosti n^2) a v tom, že algoritmus nevyužívá možného nízkého počtu hran.

V Krkalovi implementované algoritmy se jednak snaží vyhnout paměťově náročným maticím, dále se snaží problém řešit líně – jen u těch vrcholů, kde je to aktuálně požadováno – a využívají DFS (prohledávání do hloubky). DFS má složitost O(n + m), kde n je počet vrcholů a m počet hran. Pro acyklický orientovaný graf je počet hran $O(n^2)$, ale v případech, které skutečně nastávají, půjde spíše o O(n). Proto generování tranzitivního uzávěru založeného na DFS má sice složitost $O(n^3)$ (pro každý vrchol se provádí DFS), ale v případě, že počet hran je pouze O(n), bude jeho složitost $O(n^2)$.

[10] Tranzitivní redukce může být definována jako minimální podgraf grafu G, jehož tranzitivní uzávěr je stejný jako tranzitivní uzávěr grafu G. U orientovaného acyklického grafu je tranzitivní redukce určena jednoznačně.



Obrázek 12: Hierarchie KSID jmen

Kompilace typové informace

V této fázi kompilátor slučuje informace z jednotlivých souborů a buduje typovou informaci. Typová informace představuje KSID jména a jejich závislosti, popis tříd a jejich položek. Jestliže informace u souborů byla generována převážně syntaktickou analýzou, pak generování typové informace je čistě sémantická analýza.

Pokud nebyl změněn ani jeden z použitých souborů, kompilátor využije výstup předchozí kompilace. Kroky, které byly dříve úspěšně dokončeny mohou být nyní vynechány.

Kompilátor nejprve tvoří explicitně deklarovaná KSID jména. KSID jména mohou být různých typů, typ KSID jména ovlivňuje to, jakou další informaci si s sebou KSID jméno nese. Polymorfní hierarchii KSID jmen znázorňuje Obrázek 12. `KsidName` nenese žádnou specifickou informaci. `ClassName` je jméno třídy a obsahuje popis té třídy. `TypedName` je jméno proměnných a metod, obsahuje typ proměnné nebo návratový typ metody. `MethodName` dědí od `TypedName` a obsahuje navíc seznamy argumentů.

Ve druhém kroku kompilátor přiřazuje typy k typovaným, explicitně deklarovaným, KSID jménům. U tříd je součástí typu opět KSID jméno, proto je tento krok odložen na dobu, kdy už musí být jména všech tříd deklarována.

Ve třetím kroku kompilátor tvoří závislosti mezi jmény. Je povinnost explicitně deklarovat jména tříd a všechna jména, mezi kterými je deklarována závislost. Deklarace závislosti tedy pracuje se jmény, která jsou vytvořena už v prvním kroku. Kompilátor následně jména topologicky setřídí a otestuje, zda mezi nimi neexistuje cyklus.

I v dalších krocích mohou vznikat KSID jména (jména jsou deklarována při jejich prvním použití), ale už mezi nimi nejsou závislosti. Tato nová jména už nemohou ovlivnit vztahy mezi jmény ani topologické uspořádání jmen.

Ve čtvrtém kroku kompilátor zařazuje položky ke třídám. Kontroluje duplicity výlučných položek.

V pátém kroku kompilátor dědí položky z předků na potomky. Kompilátor využije topologicky setříděných jmen a zpracovává třídy v pořadí od předků k potomkům. U

každé třídy dědí položky jen od nejbližších předků (u nich už byla dědičnost spočítána, takže nejbližší předkové už obsahují vše, co zdědili).

Položky jsou děděny pouze, pokud třída neobsahuje výlučnou položku stejného jména. Jestliže je děděno více různých položek stejného jména a alespoň jedna z nich je výlučná, kompilátor nahlásí chybu.

Na tomto místě kompilace skončí, pokud byla požadována pouze typová informace.

Generování kódu

Kompilátor neprovádí tuto fázi, pokud ve fázi generování typové informace došlo k chybě nebo pokud fáze generování kódu úspěšně proběhla v minulosti a od té doby nedošlo k žádné změně.

Kompilátor nejprve překládá atributy a inicializace položek. U položek si pamatuje místo ve zdrojovém souboru, kde začíná hranatá závorka s atributy nebo kde je operátor přiřazení. Kompilátor analyzuje zdrojový kód od tohoto místa a výsledek přidá k typové informaci.

Poté kompilátor vytvoří instanci generátoru kódu (přes callback delegáta). Generátor kódu musí implementovat interface `ICodeGenerator`.

Následně jsou kompilovány jednotlivé metody. Výsledkem je stromová struktura, která je reprezentuje. Jestliže zatím nedošlo k žádné chybě a existuje generátor kódu, je mu struktura předána k následnému zpracování. Tato struktura má mít pouze dočasnou životnost, ani kompilátor, ani generátor kódu by si na ni neměly držet odkaz, aby ji garbage collector mohl uvolnit. Kompilace metod není cacheována.

Nakonec kompilátor oznámí generátoru kódu (pokud existuje a nedošlo k žádné chybě), že má vytvořit výstup. Generátor by měl do výstupu zařadit jak veškerou typovou informaci, tak těla metod a inicializace, která průběžně zpracovával v předchozím kroku.

6.5 Kompilace metod

Výsledkem kompilace metod je stromová struktura, jejíž uzly odpovídají syntaktickým prvkům jazyka. První úroveň je tvořena uzly odvozenými od třídy `Statement`, to může být `CodeBlock`, `Expression`, `LocalDeclaration` nebo potomek třídy `Command`. Třída `Expression` reprezentuje výraz. Ten je opět uložen ve stromové struktuře, uzly tentokrát mohou být tvořeny třídami jako `ExprUnary`, `ExprBinary`, `ExprSafeCall` atd.

Výstup ve formě stromu byl zvolen i přesto, že jeho generování je pomalejší a paměťově náročné. S reprezentací ve formě stromu se totiž nejsnáze pracuje. Kdyby měl být výstupem lineární proud instrukcí, výhoda by byla v jeho kompaktnosti a v tom, že generátor kódu by tento proud mohl okamžitě zpracovávat. Nevýhoda je v tom, že kompilátor by musel instrukce v proudu nějakým způsobem řadit. Post-order by vyhovoval generátoru mezikódu, in-order by vyhovoval generátoru jazyka C++. Je velmi pravděpodobné, že pořadí instrukcí generovaných kompilátorem by generátoru kódu nevyhovovalo a musel by si tyto instrukce přeuspořádat. Stromovou strukturu může každý generátor kódu procházet tak, jak potřebuje. Navíc se stromovou

reprezentací se pracuje snáze i kompilátoru, například v případech, kdy je třeba se k nějakému uzlu dodatečně vracet.

Metody jsou kompilovány kompletně v jednom kroku od lexikální analýzy až po analýzu sémantickou. Nepoužívá se cacheování. V budoucnu by kompilátor cacheování požívat mohl. Jazyk definuje množinu podmínek, které, pokud jsou splněny, zaručují, že metoda nemusí být překompilována. Podmínky vypadají takto:

- Nesmí být změněn vlastní kód metody ani její signatura.
- Všechna KSID jména použita uvnitř metody musí existovat a zůstat stejného typu.
- Signatury použitých direct volání se nesmí změnit.
- Návratový typ jména safe metody, použitého při volání, se nesmí změnit.
- Nesmí se změnit typy statických a členských proměnných, ke kterým metoda přistupuje.
- Členské direct metody a proměnné musí stále existovat u tříd, přes které k nim metoda přistupuje.
- Jestliže metoda používá přetypováni z potomka na předka, musí být předek stále předkem potomka.
- Volitelně: Jestliže metoda přetypovává objekt A na B a B nebyl předek A, B stále nesmí být předek A.

Cacheování by měl provádět generátor kódu, aby byla cacheována už výsledná podoba metody, nikoli její stromová reprezentace.

Analýza metod je implementována rekurzivně volanými funkcemi, které analyzují zdrojový kód podle pravidel gramatiky. Kompilátor musí kontrolovat a určovat typy výrazů i podvýrazů. Vlastnosti unárních a binárních operátorů (jako požadované a výsledné typy, druh operace, priorita, ...) nejsou napevno zabudovány do kódu kompilátoru, ale jsou zadány jako data.

Zajímavostí jsou výrazy, u kterých typ není jasně dán, ale musí být později odvozen kompilátorem. Více viz Dohledávání typu výrazů.

Na místech, kde gramatika připouští více alternativ se musí kompilátor umět rozhodnout, která z alternativ je platná. Kompilátor se rozhoduje podle výhledu a množin FIRST.

Zajímavostí je konflikt mezi lokální deklarací a výrazem. Kompilátor musí umět rozlišit tyto entity ještě dříve, než je začne analyzovat. Pokud by entita začínala identifikátorem, kompilátor ho nemůže analyzovat, dokud neví, v jakém se nachází kontextu. V případě lokální deklarace by šlo o jméno třídy, očekávalo by se KSID jméno v globálním kontextu. V případě výrazu, by identifikátor mohl být buď lokální proměnná nebo KSID jméno v kontextu třídy.

Aby kompilátor poznal lokální deklaraci musí použít výhled větší než 1. Lokální deklarace začíná právě jedním z následujících způsobů:

- Klíčovým slovem, které označuje typ (`int`, `object`, `string`, ...)
- Dvěma identifikátory za sebou. (Jde o jméno třídy a jméno deklarované proměnné.)
- Identifikátorem a hranatými závorkami `[]`. (Jde o deklaraci pole tříd.)

Binární operátory

U binárních operátorů bylo použito řešení které minimalizuje hloubku rekurze. V jazyce existuje 11 prioritních hladin a pokud by kompilátor volal funkci pro každou prioritní hladinu, bylo by to neefektivní. Kompilátor provádí vnořené volání jen tehdy, kdy je to nutné. Kompilátor uzávorkovává výrazy zleva doprava.

```
a + b + c;   // je uzávorkováno jako ((a + b) + c)
```

Proto rekurzivní volání je potřeba provést jen tehdy, kdy je nutné výraz uzávorkovat opačně. Například:

```
a + b * c;   // je nutné uzávorkovat jako (a + (b * c))
```

Následující kód vyhodnocuje binární výrazy:

```
internal ExprNode DoBinaryOperator() {
    ExprNode node1 = DoUnaryOperator();
    return DoBinaryOperator(1, node1);
}


private ExprNode DoBinaryOperator(int lowestPriority, ExprNode node1) {
    while (true) {

        LexicalToken token = Syntax.Lexical.Peek();
        if (token.Type != LexicalTokenType.Operator ||
        token.Operator.Priority < lowestPriority) {
            // I cannot handle this
            return node1;
        }


        Syntax.Lexical.Read();
        ExprBinary node = new ExprBinary(this, token);
        node.Left = node1;
        int currentPriority = token.Operator.Priority;

        ExprNode node2 = DoUnaryOperator();

        token = Syntax.Lexical.Peek();
        if (token.Type == LexicalTokenType.Operator &&
        (token.Operator.Priority > currentPriority ||
        (token.Operator.Priority == currentPriority &&
        currentPriority == RightToLeftAssociativity))) {
            // I need to do higher priority operator first
            node2 = DoBinaryOperator(currentPriority + 1, node2);
        }

        node.Right = node2;
        node.CheckBinaryNode();

        node1 = node;
    }
}
```

Funkce `DoBinaryOperator` dostává dva parametry: `lowestPririty` je nejnižší možná priorita, kterou může funkce zpracovat, `node1` je levá strana operátoru.

Funkce se nejprve podívá jestli následující token je binární operátor s prioritou alespoň `lovestPriority`. Jestliže ne, funkce vrátí `node1`.

Následuje přečtení operátoru a vyhodnocení unárního výrazu na jeho pravé straně.

Nyní musí funkce ověřit, zda může společně uzávorkovat levou a pravou stranu operátoru. Načte další token a podívá se, jestli to je binární operátor s vyšší prioritou

než právě zpracovávaný operátor. Pokud ano musí ho řešit přednostně, zavolá tedy rekurzivně funkci `DoBinaryOperátor` s `lowestPriority` nastavenou na prioritu o jedna vyšší, než je priorita aktuálního operátoru.

Funkce spojí levou a pravou stranu operátoru v jeden binární uzel. Výsledný výraz uloží do proměnné `node1` a zopakuje všechny předchozí kroky.

6.6 Analýza rychlosti

Je těžké kvantitativně vyjádřit rychlost kompilátoru. Měření rychlosti ovlivňuje jak rychlost hardwaru, tak samotný způsob testování. Subjektivní odpověď na otázku, zda je kompilátor dostatečně rychlý, zní ano. Přesná měření rychlosti prováděna nebyla. Výsledky následujícího měření jsou pouze orientační.

Test byl prováděn na počítači s procesorem AMD Athlon 64, 3200+ s 1 GB operační paměti. Byl prováděn úplný překlad jednoho 160kB souboru (12 000 řádků kódu), byly potlačeny cache. Pro srovnání celá hra Krkal, implementována v Systému Krkal 2.0, má cca 170kB. Předpokládá se, že běžně bude program rozdělen na několik menších souborů a změny budou prováděny jen v jednom z nich. Proto v běžné situaci bude kompilace podstatně rychlejší než v tomto testu.

Celá kompilace, včetně generování výstupu do jazyka C++, trvala přibližně 1 sekundu. Poté byl spuštěn C++ kompilátor, aby zkompiloval C++ soubory do knihovny dll. C++ kompilace trvala přibližně 5 sekund.

Nasazení kompilátoru v IDE se ukázalo také jako velice rychlé. Kompilace je spouštěna na pozadí v samostatném vlákně, práci uživatele by neměla nijak významně ovlivnit. Jako pomalý se ukázal použitý editor textu (Compona Syntax Box), který pravděpodobně není stavěný na tak velké soubory. Pomalý může být i update vizuálních komponent, které reprezentují typovou informaci. Kompilace je sice poměrně rychlá a je prováděna na pozadí, update vizuálních komponent je ale prováděn opět v hlavním vlákně a pokud je třeba v class view zobrazeno příliš mnoho řádků, začne být jejich aktualizace pomalá a uživatel to nepříjemně pocítí.

7 Implementace runtimu

Jazyk Krkal C pro svůj běh vyžaduje speciální runtime. Změny v jazyce si vyžádaly i změny v runtimu a implementaci několika nových prvků. Tato kapitola popisuje tyto změny a nové prvky. Jde například o přepracování vícenásobné dědičnosti a o garbage collecting. Celkově změněny byly i kompilované skripty.

7.1 Změny v runtimu

Runtime byl v první řadě zjednodušen. Některé časti, jako nahrávání levelů, serializace a deserializace objektů nebo podpora pro zobrazování objektů na herním plánu, byly z runtimu (zatím) úplně odstraněny.

Runtime byl připraven na paralelní běh ve více vláknech. Byly odstraněny všechny statické proměnné. Celý běh runtimu je nyní uzavřen v jednom objektu. Nyní je možné spustit více runtimů najednou, nezávisle na sobě. Mohou být spuštěny jak v jednom vlákně, tak paralelně v samostatných vláknech. Poznámka: V rámci jednoho runtimu objekty paralelně ve více vláknech neběží.

Změna typového systému si vyžádala i značné změny v runtimu. Díky tomu, že typový systém je nyní jednodušší, byl i runtime zjednodušen.

Dále byla přepracována pole. Pole jsou nyní implementována pomocí C++ šablon a jejich funkcionalita byla značně rozšířena. Jako typ string je nyní chápáno pole charů.

Typy jazyka jako `name`, `object` a pole jsou implementovány třídami, které definují celou řadu operátorů tak, jak je jazyk Krkal C využívá, například porovnání KSID jmen nebo porovnání a přetypování pointerů na objekty.

I runtime používá orientovaný acyklický graf KSID jmen. Jeho implementace měněna nebyla. Runtime, narozdíl od kompilátoru, používá k tranzitivnímu testu, zda jméno je potomkem nebo předkem, matice o velikosti souvislé komponenty v grafu. (Kompilátor místo paměťově náročných matic využívá líně tvořené hashovací tabulky.)

Implementace safe volání a zasílání zpráv také zůstala téměř beze změny.

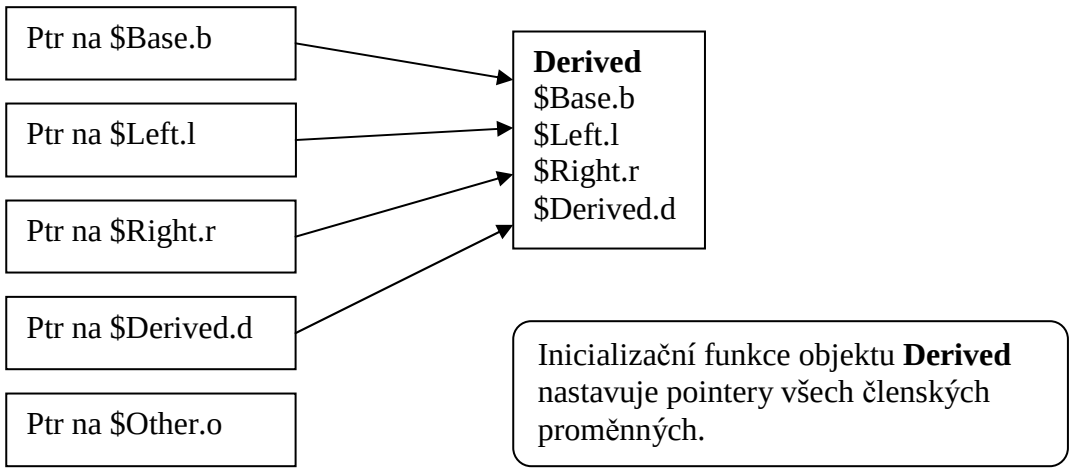
Změněn musel být start runtimu a načítání skriptů, protože skripty jsou nyní ukládány v jiném formátu.

Vícenásobná dědičnost

Implementace vícenásobné dědičnosti ovlivňuje přístup ke členským proměnným a způsob jejich uložení. Třídy dědí proměnné od svých předků. V jazyce Krkal C je každá proměnné obsažena v objektu maximálně jednou, nezávisle na způsobu dědění.

Různé strategie implementace vícenásobné dědičnosti budou ukázány na příkladu. Přistupovat se bude k proměnným třídy `Derived`, která je definována takto:

```
class name Base, Left, Right, Derived;
depend Base << {Left, Right} << Derived;

class Base {
    int b;
}

class Left {
    int l;
```



Obrázek 13: Implementace dědičnosti v Krkalovi 2.0

```
}

class Right {
    int r;
}

class Derived {
    int d;
}
```

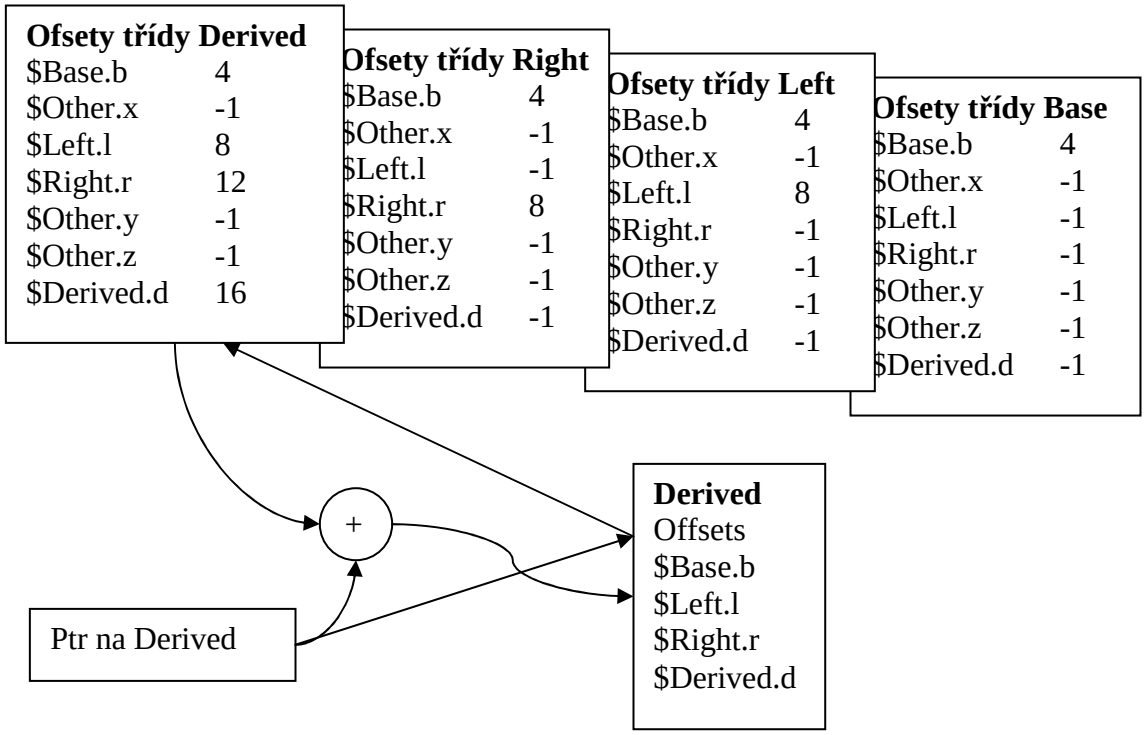
Třída `Derived` dědí od svých předků 3 proměnné: `$Base.b`, `$Left.l` a `$Right.r` a přidává jednu svou: `$Derived.d`.

Krkal 2.0 neumožňoval přístup k proměnným z vnějšku. Dalo by se říci, že všechny proměnné byly `protected`. Data objektu byla tvořena jeho členskými proměnnými, na jejich pořadí nezáleželo, navíc objekt obsahoval určitá systémová data, která ale nejsou pro tento příklad zajímavá. Tedy datová oblast objektu třídy `Derived` byla tvořena pouze čtyřmi inty. Kompilované skripty obsahovaly, pro každé jméno členské proměnné, proměnnou, do které se ukládal pointer na členskou proměnnou toho jména. Přes tyto pointery se přistupovalo ke členským proměnným.

Před voláním libovolné metody byla zavolána inicializační funkce objektu, jehož metoda byla volána (šlo o skutečný typ objektu). Tato metoda inicializovala pointery daného objektu tak, že ukazovaly na členské proměnné toho objektu. Pointery na proměnné mimo aktuální objekt zůstaly neinicializované a nebyly používány.

Druhá varianta vícenásobné dědičnosti už umožňuje přístup z vnějšku. Místo s pointery pracuje s ofsety. Tyto ofsety jsou uspořádány do tabulky o velikosti „počet všech jmen proměnných“ x „počet všech tříd“. Ofsety inicializuje runtime při startu. Objekt obsahuje kromě proměnných i pointer na ofsety na své proměnné.

Při přístupu ke členské proměnné je nejprve z dat objektu přečten pointer na tabulku ofsetů. Z ní je přečten ofset příslušné proměnné a ten je přičten k pointeru na objekt.



Obrázek 14: Druhá implementace dědičnosti

Krkal 3.0 nepoužívá tabulky ofsetů, ale objekty s více vstupními body. Pointery na proměnné jsou obsaženy přímo v každém objektu. Pro přístup ke členské proměnné p je potřeba pointer typu A na objekt a $ofset_{pA}$, který je závislý jak na členské proměnné p, tak na typu A. Pointer na proměnnou p se nachází na adrese (ptr typu A na objekt + $ofset_{pA}$).

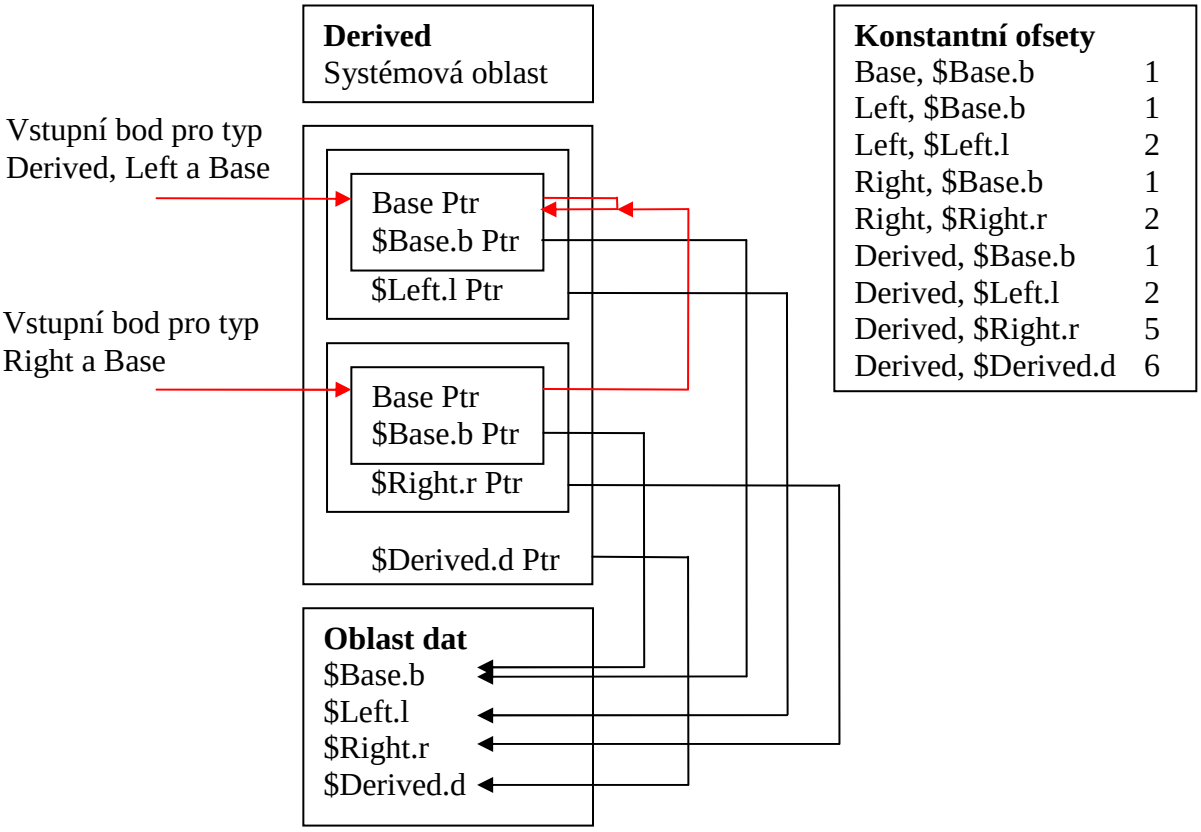
Objekt se dělí na systémovou část, část pointerů a datovou část, viz Obrázek 15. Při tvorbě objektu musí runtime, kromě alokace paměti, správně inicializovat systémovou a pointerovou část, to dělá podle šablony, kterou si připraví při startu pro každý typ. Datovou část runtime inicializuje na nuly. Teprve poté je volána funkce, která inicializuje členské proměnné a konstruktor.

Aby mohly existovat konstantní $ofsety_{pA}$ platné nejen pro objekt A, ale i pro všechny jeho potomky, mohou mít objekty více vstupních bodů a pointery na proměnné mohou být duplicitní. Pointer na objekt nemíří na začátek dat objektu, ale na některý vstupní bod. Při přetypování je třeba pointer posunout.

U přetypování z potomka na předka ví kompilátor, že přetypování existuje a o jaký ofset bude pointer posunut.

U obecného přetypování, se používá hashovací tabulka, ta je uložena u skutečného typu objektu. Runtime zkusí vyhledat v hashovací tabulce skutečného typu typ, na který má být pointer přetypován. Jestliže takový typ najde, získá ofset, který přičte k prvnímu vstupnímu bodu. (Na tento bod míří Base Ptr.) V opačném případě je výsledkem přetypování `null`.

Base Ptr se používá pro přetypování, pro porovnání pointerů na shodu a pro přístup do systémové oblasti.



Obrázek 15: Implementace dědičnosti v Krkalovi 3.0

Výhody a nevýhody jednotlivých přístupů shrnuje následující tabulka:

	Krkal 2.0	**2.varianta**	**Krkal 3.0**
Umožňuje přístup z vnějšku:	Ne	Ano	Ano
Přístup přes:	Pointer na proměnnou	Ptr na objekt, $ofset_p$	Ptr na objekt, $ofset_{pA}$
Počet operací pro přístup:	1	6	4
Příprava za běhu:	Inicializační fce objektu	Žádná	Posun vstupních bodů
Statická data:	Pointery	Tabulka ofsetů	Hashovací tabulky a šablony.
Velikost statických dat:	proměnné	proměnné * objekty	objekty * x
Data u objektů navíc:	0	1	pointerová část

Bylo těžké se rozhodnout pro jednu z variant, protože každá má své výhody a své nevýhody. Použitá varianta nakonec zvítězila pro svůj relativně rychlý přístup k proměnným a menší paměťovou náročnost v případě, že žijících objektů není příliš. Má ovšem i své nevýhody, například komplikovanější implementaci a vysokou paměťovou náročnost v případě, že počet žijících objektů značně přesáhne počet tříd.

Pro srovnání paměťové náročnosti je možné použít původní hru Krkal. Hra obsahovala 115 tříd a 258 jmen členských proměnných. Paměťová náročnost druhé varianty by tedy byla 115 * 258 * 4 (velikost ofsetu) = 118680. Předpokládejme že typický level má velikost 32 * 22 políček a na každém políčku jsou dva objekty. Jestliže by průměrný počet záznamů v pointerové části byl osm, tak je paměťová náročnost 32 * 22 * 2 * 8 * 4 = 45056 (statická data započítána nebyla).

I pro druhou variantu je třeba správně přetypovávat pointery na objekty, ač posouvat vstupní body není potřeba. Pokud jde o přetypování z libovolného objektu na libovolný jiný, je třeba ověřit platnost přetypování a v případě neúspěchu musí být

výsledkem přetypování `null`. U přetypování z potomka na předka není potřeba provádět žádnou akci.

Garbage collecting

Runtime obsahuje velmi jednoduchý garbage collector. Garbage collector automaticky uvolňuje objekty, na které už neexistuje žádný odkaz.

Všechny objekty, které jsou spravovány garbage collectorem mají společného předka – `CManagedObject`. Spravovaný objekt se při konstrukci přihlásí do garbage collectoru, obsahuje proměnnou `mark` a musí implementovat virtuální metodu `MarkLinks`.

```
class KRKALRUNTIME_API CManagedObject {
friend CGarbageCollector;
public:
    CManagedObject(CGarbageCollector & collector) : _mark(0)
    {
        collector.Add(this);
    }
    void MarkMe(int mark, CGarbageCollector & collector) {
        if (_mark != mark) {
            _mark = mark;
            collector.AddToQueue(this);
        }
    }
    // This method has to call MarkMe on all nested objects
    virtual void MarkLinks(int mark, CGarbageCollector & collector) =0;
    int GetMark() { return _mark;}
private:
    int _mark;
protected:
    virtual ~CManagedObject() {};
};
```

Garbage collector uvolňuje paměť jednou za čas nebo pokud bylo vytvořeno mnoho nových objektů. Přesné podmínky spuštění se dají nastavit pomocí konstant. Garbage Collector pracuje pouze na konci taktu, tedy v situaci, kdy neprobíhá žádné volání a neexistují žádné lokální proměnné. Garbage collector prohledává objekty a sleduje jejich odkazy na jiné objekty. Objekty, které takto navštíví si označí. Uvolní ty objekty, které označeny nebyly.

Prohledávání musí mít začátek. Garbage collector proto implementuje funkce `Hold` a `Release`. Držený objekt je zařazen do speciálního seznamu, ze kterého Garbage collector začíná prohledávat. Držený objekt samozřejmě nemůže být uvolněn. Funkci `Hold` je možné zavolat vícekrát, garbage collector si počet volání `Hold` počítá. Ke každému `Hold` je třeba v páru zavolat `Release`.

Runtime volá `Hold` na objekt `Static` (tato systémová třída slouží jako kontejner pro všechny statické proměnné) a na všechny spravované typy, které se ocitnou mezi parametry zprávy. Systém Krkal bude pravděpodobně volat `Hold` i v mnoha dalších případech.

Při prohledávání je nejprve třeba označit objekty, které nebudou uvolněny. Garbage collector je značí číslem, které se při každém prohledávání o jedna zvýší. Protože je značka pokaždé nová, není třeba ji mazat.

Garbage collector nejprve zavolá metodu `MarkMe` na všechny držené objekty. Tato metoda, pokud objekt ještě nebyl označen, označí objekt a zařadí ho do fronty.

Dokud fronta není prázdná, garbage collector vyzvedává objekty z fronty a volá na nich virtuální metodu `MarkLinks`, z té objekt zavolá metodu `MarkMe` u všech objektů, na které si drží referenci. Pro implementaci této metody využívá runtime znalost typové informace.

Výsledkem tohoto procesu je, že všechny přístupné objekty mají značku. Ty bez značky garbage collector uvolní z paměti. (V jazyce Krkal C není destruktor, takže objekty nemohou vstávat z mrtvých.)

Garbage collector mohl být implementován i jinak, třeba počítáním referencí. Toto řešení má výhodu v tom, že neobsahuje fázi značení, takže se systém chová plynuleji. Nevýhoda je v samotném počítání referencí, které je pomalejší, a v paměti mohou zůstávat objekty, které se odkazují samy na sebe.

Protože runtime disponuje vlastní správou paměti, mohlo být s objekty pracováno i inverzně. Místo alokace v běžné paměti by byly objekty alokovány v paměti runtimu. Před prohledáváním by byla celá paměť označena za volnou a při značení objektu, by si objekt svou paměť opětovně zarezervoval.

Řešení, které bylo implementováno, bylo zvoleno pro svou jednoduchost a pro mírnou preferenci prohledávání odkazů nad referencemi.

7.2 Kompilované skripty

Kompilované skripty představují kód jazyka Krkal C převedený do C++. Kompilované skripty pro svůj běh využívají funkcí runtimu. Tato kapitola popisuje, jak kompilované skripty fungují.

V současnosti nejsou skripty rozděleny na moduly. Ke každému projektu generuje generátor kódu jeden .code soubor s typovou informací a C++ zdrojové soubory, ze kterých bude vytvořen jeden .dll soubor. Existence jediného modulu usnadňuje identifikaci.

Identifikace entit v kompilovaných skriptech

Ideální způsob identifikace jsou KSID jména, protože jsou stabilní a jednoznačná. Jejich nevýhoda je délka, zvláště pokud se pracuje s entitami jako ofset členské proměnné, který je identifikován dvěma KSID jmény. Takový identifikátor může vypadat takto:

```
_KSV_Map_3798_03CD_176A_36B2__M_1Map_3798_03CD_176A_36B2__M_Size_3798_03CD_176A_36B2
```

Kompilované skripty vnitřně používají kratší identifikátory. Kratší identifikátor obsahuje číslo které ho činí jednoznačným. Krátký identifikátor vypadá takto:

```
_KSV_21_Map_Size
```

Protože pořadové číslo nemůže být stabilní, je krátký identifikátor používán pouze uvnitř modulu. Pro vnější komunikaci se používají KSID jména.

Bohužel je třeba pracovat i s entitami, které nemají KSID jméno vůbec. To jsou například konkrétní implementace metod.

Kompilátor přiřazuje všem položkám pořadové číslo, to je možné využít pro tvorbu identifikátoru. Takový identifikátor je jednoznačný, ale není stabilní (změna ve zdrojovém kódu může vyvolat přečíslování). Kompilovaným skriptům tato nestabilita nevadí, protože výstup je při každé změně generován celý znovu.

Pokud by byly kompilované skripty složeny z více modulů, které nejsou vždy kompilovány všechny, runtime už nemůže použít číslované identifikátory, ale mohl by vyhledávat (identifikovat) metodu podle invariantních vlastností popsaných v kapitole Kompilace metod.

V kompilovaných skriptech je třeba identifikovat následující entity:

- Implementace metody – součástí identifikátoru je pořadové číslo.
- KSID jméno – identifikátor tvoří KSID jméno.
- Pointer na statickou proměnnou – identifikátor tvoří KSID jméno.
- Ofset členské proměnné – identifikátor tvoří 2 KSID jména.
- Ofset přetypování na předka – identifikátor tvoří 2 KSID jména.
- Pointer na direct metodu – identifikátor tvoří KSID jméno.
- Inicializace objektu – identifikátor tvoří KSID jméno.

Inicializace kompilovaných skriptů

Runtime nejprve nahraje do paměti modul dll s kompilovanými skripty. Pak zavolá jedinou exportovanou metodu, která vytvoří objekt třídy `CScript` a vrátí na něj pointer.

Objekt `CScript` během své konstrukce vytvoří hashovací tabulku pointerů na implementace všech kompilovaných metod. V této tabulce se dá vyhledávat podle jména.

Následně runtime nahrává typovou informaci (také připravuje šablony pro pointerové části objektů a perfektní hashovací tabulky pro safe volání). Pokud narazí na metodu, zeptá se kompilovaných skriptů na její pointer. (Kompilované skripty ho vyhledají v připravené hashovací tabulce.)

Jakmile je typová informace kompletní, může dojít k inicializaci kompilovaných skriptů. V této části se naopak kompilované skripty ptají runtimu a zjišťují pointery na KSID jména, na statické proměnné a na direct metody. Dále ofsety členských proměnných a ofsety nutné pro přetypování. Kompilované skripty se ptají jen na ty údaje, které jsou skutečně využívány v některé metodě.

Tímto je inicializace hotova a runtime může začít volat metody.

Metody

Kompilované skripty kromě objektu `CScript`, který drží všechny potřebné parametry, obsahují implementace safe a direct metod. Jsou to globální funkce, protože runtime si na ně potřebuje držet pointery.

Přestože metody jsou statické, tak nepracují s žádnými statickými daty, aby byl možný paralelní běh několika runtimů. Metody potřebují pro svůj běh přístup k následujícím klíčovým objektům: k hlavnímu objektu runtimu, k objektu `CScript` a ke kontextu. Kontext je vytvářen s každým voláním metody a obsahuje specifická data týkající se aktuálního volání, například parametry safe metody, `this` a `sender`.

Safe metoda je volána z runtimu, pouze s jediným parametrem a to pointerem na runtime. Metoda si z runtimu vezme pointer na `CScript`, připravený kontext a `this`. Případné parametry a návratovou hodnotu spravuje runtime na svém vlastním zásobníku. Safe metoda k nim přistupuje přes kontext.

```
// safe metoda v jazyce Krkal C:
void @Constructor(string Ant._name) {
    Map = sender;
    Name = _name;
    Move() timed @Math.Random(600);
}

// a její kompilovaná verze:
// class: $Ant$3798_03CD_176A_36B2
// method: @Constructor
void _KSF_22_Constructor(CKerMain *KerMain) {
    CKerContext &ctx(*KerMain->KerContext);
    OPointer thisO = ctx.KCthis;
    CScript *Script = (CScript*)KerMain->KS;

    thisO.get<OPointer>(Script->_KSV_10_Ant_Map)
        = ctx.Sender.Cast(Script->_KSID_4_Map);
    thisO.get<ArrPtr<wchar_t>>(Script->_KSV_12_Ant_Name)
        = ctx.prm<ArrPtr<wchar_t>>(0);
    KerMain->message(101, thisO, Script->_KSID_13_Move, 5,
        Script->_KSDM_15_Random(KerMain, 0, 600), 0, 0);
}
```

Direct metoda je volána přímo z jiné kompilované metody. Parametry jsou předávány běžným způsobem a metoda může vracet hodnotu. Direct metody dostávají navíc pointer na runtime a `this`. `CScript` si metoda vezme z runtimu a kontext si tvoří sama, jde o objekt na zásobníku, takže se na něj automaticky zavolá destruktor, když direct metoda končí.

```
// direct metoda v jazyce Krkal C:
direct void Place(Placeable obj, int x, int y) {
    obj->x = x;
    obj->y = y;
    Map[x][y] = obj;
}

// a její kompilovaná verze:
// class: $Map$3798_03CD_176A_36B2
// method: $Map$3798_03CD_176A_36B2.Place$3798_03CD_176A_36B2
void _KSF_12_Place(CKerMain *KerMain, OPointer thisO, OPointer _KSL_obj,
int _KSL_x, int _KSL_y) {
    CScript *Script = (CScript*)KerMain->KS;
    CKerContext ctx(KerMain, Script->_KSID_32_Place, thisO, "_KSF_12_Place");

    _KSL_obj.get<int>(Script->_KSV_41_Placeable_x) = _KSL_x;
    _KSL_obj.get<int>(Script->_KSV_43_Placeable_y) = _KSL_y;
    thisO.get<ArrPtr<OPointer,2>>(Script->_KSV_35_Map_Map)[_KSL_x][_KSL_y]
        = _KSL_obj;
}
```

Velká část kódu jazyka Krkal C lze převádět do C++ beze změny. Runtime obsahuje speciální typy pro pointer na objekt, KSID jméno a pole. Tyto typy definují operátory a metody tak, aby je kompilované skripty mohly snadno používat.

Přístup ke členským proměnným je umožněn metodou `get` u pointeru na objekt. Při volání je třeba specifikovat typ členské proměnné a její $ofset_{pA}$. $Ofset_{pA}$ je uložen v proměnné objektu `CScript` (Při startu ho runtime spočítá z typové informace.)

```
_KSL_obj.get<int>(Script->_KSV_41_Placeable_x) = _KSL_x;
```

Metoda `get` přistupuje k objektu podle pravidel popsaných v kapitole Vícenásobná dědičnost. K pointeru, který míří na přístupový bod objektu, přičte $ofset_{pA}$, výsledek dvakrát dereferencuje a přetypuje na referenci na požadovaný typ. Metoda `get` zapsaná v C++ vypadá takto:

```
template<typename T>
T & get (int offset) const {
    if (ptr == 0)
        throw CKernelError(eKRTEaccessingNullObject);
    return *((T**)(ptr))[offset];
}
```

Safe metodám runtime nepředává parametry přímo, ale přes kontext. Metoda s nimi pracuje podobně jako s členskými proměnnými, jen místo ofsetu se používá pořadové číslo parametru.

```
ctx.prm<ArrPtr<wchar_t>>(0)
```

Jestliže metoda potřebuje KSID jméno, pointer na statickou proměnnou, pointer na direct metodu nebo ofset pro přetypování, přečte tyto údaje z objektu `CScript`, kde jsou uloženy jako členské proměnné. K jejich identifikaci se využívají zkrácené identifikátory s pořadovým číslem. Například pro obecné přetypování je potřeba KSID jméno cílového typu.

```
ctx.Sender.Cast(Script->_KSID_4_Map);
```

Kompilované skripty v jiných jazycích

Výhoda kompilovaných skriptů spočívá ve snadném generování, absenci interpretu, možnosti snadno kontrolovat a ladit výsledek. Další výhoda je možnost snadno propojit skripty a běžný kód v cílovém jazyce.

Ideálním jazykem pro kompilované skripty je C++. C++ umožňuje programování jak na nízké úrovni – přímá práce s pamětí, pointery, přetypování – tak na vyšší úrovni – C++ šablony, třídy s přetíženými operátory a podobně. To vše umožňuje vytvořit vrstvu efektivně podporující kompilované skripty.

Otázkou zůstává, jak by vypadaly kompilované skripty v jiných jazycích, které nedisponují takovou silou, jako C++. Třeba v C# nebo Javě. Hlavními problémy by bylo předávání parametrů safe metod a simulace Krkal C tříd, tedy přístup ke členským proměnným. Není pevně dáno, jak má v Krkalovi vypadat pole nebo jak konkrétně má být implementována vícenásobná dědičnost. I když v C++ byl zvolen určitý přístup, v jiných jazycích může být vhodný přístup jiný.

8 Integrované vývojové prostředí

Integrované vývojové prostředí obsahuje nástroje pro editaci zdrojového kódu i grafické rozhraní, které umožňuje sledovat a ladit běh vytvářených programů. Prostředí se skládá z celé řady komponent (včetně výše popsaného kompilátoru a runtimu). Tato kapitola se věnuje zbývajícím komponentám a popisuje jejich integraci do jednoho celku.

8.1 Komponenta File System

Tato komponenta byla převzata z původního Systému Krkal a mírně rozšířena. File System představuje abstrakci nad souborovým systémem, dovoluje místo skutečných cest k souborům pracovat se zástupnými identifikátory, podporuje archivy i binární soubory se stromovou strukturou – tzv. registry. Registr umožňuje přistupovat k položkám různých typů podle textového klíče. Nyní je používán pro ukládání typové informace, původní verze do těchto souborů ukládala i levely a různá další strukturovaná data.

File System umožňuje také sdílet data mezi editorem skriptů a kompilátorem. Kompilátor je informován, zda byl soubor změněn. Pokud je soubor otevřen v editoru, File System tam přesměrovává požadavky na čtení.

Specifikace Krkala 3.0 předpokládá, že datové soubory jako skripty, grafika a levely budou identifikovány jménem (a verzí) bez cesty. Přesto bude možné použít adresářovou strukturu. Bude třeba definovat pravidla, jak bude File System soubory vyhledávat a co dělat v případě existence dvou souborů stejného jména. Může být například vyžadována totožnost těchto souborů a File System bude číst z některého z nich, ale zapisovat do všech. Toto jsou plány do budoucna, současný File System vyhledávat v adresářích ještě neumí, proto je potřeba ukládat soubory do jediného adresáře.

Také bude potřeba přesněji definovat a stabilizovat interface komponenty File System tak, aby mohla být nahrazena jiným souborovým systémem, který například místo souborů pracuje s databázemi nebo místo registrů používá formát xml.

Jedním z požadavků na File System je i možnost použití ve více vláknovém prostředí. File System musí implementovat synchronizační mechanismy a být připraven na zpracování paralelních požadavků z různých vláken. Synchronizace je v současnosti implementována částečně, je to celkem obtížný úkol, protože původně se s tím vůbec nepočítalo.

8.2 IDE – prostředí pro psaní skriptů

IDE je obecná komponenta, která obsahuje editor skriptů a grafické rozhraní sloužící k prezentaci typové informace a chybového výstupu.

IDE je implementováno pomocí knihovny `System.Windows.Forms` z .NET Frameworku. Jsou použity i komponenty třetích stran: Pro správu oken byl použit DockPanel, Weifen Luo, http://sourceforge.net/projects/dockpanelsuite. Pro editaci zdrojového textu byla vybrána komponenta Compona SyntaxBox, http://www.puzzleframework.com.

IDE má být komponenta, která umožní v uživatelsky příjemném prostředí psát skripty jazyka Krkal C. Už nyní IDE využívá výsledků kompilátoru, který je opakovaně

spouštěn na pozadí, k aktualizaci typové informace i chybového výstupu, chyby jsou například průběžně podtrhávány červenou vlnovkou zároveň s tím, jak uživatel píše kód.

Do budoucích verzí jsou plánována i další vylepšení. Od komfortního vyhledávání v textu po kontextové napovídání a doplňování kódu. Tento požadavek si vyžádá úzké propojení editační komponenty a kompilátoru. Bohužel SyntaxBox se neukázal jako plně použitelný, bude potřeba ho nahradit nějakou jinou, pravděpodobně placenou, komponentou s lepší dokumentací, podporou a s méně chybami.

8.3 Generátor kódu

Implementovaný generátor kódu převádí výstup kompilátoru do jazyka C++. Během své inicializace připraví pracovní prostředí v adresáři `Compiler\CppSource`. Pokud tam ještě neexistuje adresář se jménem kompilovaného projektu, tak ho vytvoří a překopíruje do něj výchozí soubory, které se nacházejí v adresáři `Compiler\CppSource\Krkal.KS`. Mezi těmito soubory je projektový soubor, soubory `Scipt.*` a soubory s příkazy `#include`. Adresáře `Compiler\CppSource\Shared` a `Compiler\CppSource\Include` obsahují zdrojové soubory, které jsou sdílené mezi všemi projekty. Uživatel může projekt a zdrojové soubory modifikovat, kromě souborů `Scipt.*`, které jsou při každé kompilaci přepsány.

Následně generátor kódu zpracovává těla metod tak, jak mu je kompilátor poskytuje, výsledky zatím ponechává pouze v paměti.

Během závěrečného kroku generátor tvoří výstup. Typovou informaci uloží do souboru s příponou `.code`. Těla metod, inicializace a další pomocný kód ukládá do souborů `Script.*`. Poté použije pro překlad C++ projektu Microsoft ® Visual C++ Project Builder, který je přístupný z .NET Frameworku. Jestliže kompilace skončí úspěšně, generátor kódu překopíruje výsledný modul dll do adresáře `Compiler\Bin` a typovou informaci do adresáře `Compiler\Code`, tam je očekává runtime.

8.4 Sample.Services

Tato komponenta má obsahovat veškeré knihovní funkce používané skripty. Instance třídy, ze které jsou služby přístupné, je tvořena paralelně s runtimem. Pointer na tuto instanci je při inicializaci předán jak runtimu, tak dále hlavnímu objektu kompilovaných skriptů.

V současnosti Services obsahují pouze funkce, které pracují s dvourozměrným herním plánem. Objekty mohou být umísťovány na pole plánu, nebo naopak odebírány čí přesouvány. Na jednom poli může být více objektů.

Komponenta Sevices může rozšířit i objekty jazyka Krkal o systémová data či funkcionalitu – toto rozšíření je řešeno pomocí dědění od třídy, která definuje objekt v komponentě Runtime. Dále Services mohou reagovat na událost zabití objektu, při které mohou například odstranit objekt z herního plánu.

Smyslem tohoto přístupu je zaručit nezávislost skriptů na konkrétním systému, kde jsou používány. Například pro účely této práce je použita komponenta Services, která obsahuje pouze velmi jednoduchý dvourozměrný herní plán. Herní plán v původním Krkalovi podporoval více pater a umísťování objektů ve třech souřadnicích, s přesností

na pixel. Nic ale nebrání řídit jazykem Krkal C objekty nějaké 3D hry, tam pravděpodobně bude komponenta Services implementována velmi odlišně.

8.5 Integrace v aplikaci Sample

Všechny konfigurovatelné komponenty jsou propojeny do jedné aplikace nazvané Sample. Viz také Obrázek 3 z druhé kapitoly.

V aplikaci Sample je implementován formulář, který dědí od hlavního formuláře z komponenty IDE a přidává několik nových položek do menu a nástrojových lišt, například tlačítko pro zobrazení okna s výstupem kompilace nebo tlačítko pro spuštění skriptů. Tyto věci součástí IDE nebyly, protože IDE není vázáno ani na generátor kódu ani na runtime.

Sample také inicializuje kompilátor a propojuje ho s generátorem kódu. Součástí je i definování konfigurovatelné syntaxe včetně popisu systémových metod.

Po aktivování tlačítka Run, je vytvořen runtime a komponenta Services, runtime je nastartován. Je otevřeno nové okno s nástroji, které zobrazují běh skriptů. Běhové prostředí se podobá editoru, který byl v původním Krkalovi, jen není zdaleka tak propracované. Prostředí obsahuje konzoli s výstupem chybových, informačních a uživatelských hlášení. Je zde grafická reprezentace herního plánu, kam je možné umísťovat nové objekty, dále prohlížeč členských proměnných objektů. Editor v původním Krkalovi umožňoval proměnné nejen číst, ale i měnit, editace byla řízena atributy.

Aplikace Sample má ukázat možnosti celého systému, předvést jak psaní skriptů a kompilaci, tak jejich běh. V současnosti napsané komponenty je potřeba dále vylepšovat a pak je použít buď v Systému Krkal 3.0 nebo nějaké nezávislé aplikaci.

9 Závěr

Hlavním cílem této práce bylo navrhnout jazyk, který bude použitelný v nové verzi Systému Krkal a implementovat komponenty, které jsou s jazykem svázány: kompilátor, běhové prostředí a grafické uživatelské rozhraní, které umožní jak psaní skriptů, tak interaktivní sledování jejich běhu.

Práce představila návrh nové verze Systému Krkal, který předpokládá, že systém bude sestaven z konfigurovatelných a vyměnitelných komponent. Některé z těchto komponent byly nově implementovány: kompilátor, generátor kódu a prostředí pro psaní skriptů, jiné byly převzaty z původního Krkala a upraveny: runtime a komponenta File System. Zbývají ještě komponenty jako grafický engine nebo editor levelů, jejichž implementace je nad rámec této práce. Díky komponentám lze rozdělit vývoj Krkala do menších kroků, navíc už nyní implementované komponenty mohou najít využití v nezávislých aplikacích. Příkladem je i aplikace Sample, která současné komponenty sdružuje do jednoho celku a slouží tak k demonstraci výsledků této práce.

Jazyk Krkal C byl od základů přepracován. Byla zjednodušena jeho syntaxe včetně systému typů, důraz byl kladen na bezpečnost, na zamezení práce s neinicializovanou pamětí, o dynamicky alokované objekty se nově stará garbage collector. Klíčové prvky z původní verze, jako KSID jména a množiny, safe metody, zprávy a vícenásobná dědičnost, byly hlouběji propracovány a vylepšeny. Jazyk například obsahuje nový mechanismus propojování a verzování souborů. I kompilátor je konfigurovatelná komponenta, proto jazyk obsahuje celou řadu konfigurovatelných prvků, mimo jiné i nově navržené systémové a externí metody, které slouží k volání vnějších funkcí. Krkal C díky tomu není vázán na konkrétní systém či platformu a stává se obecně využitelným jazykem.

Kompilátor byl implementován tak, aby umožnil nejen kompilaci, ale i spolupráci s vývojovým prostředím, které potřebuje v reálném čase získávat aktuální typovou informaci. Pro zvýšení rychlosti se kompilátor snaží kompilovat jen to, co je třeba a využívá cacheování. Fáze kompilace jsou navíc navrženy tak, aby nezáleželo na pořadí deklarací a užití jednotlivých prvků jazyka. Samotný kompilátor negeneruje výstup, výsledky ponechává přístupné uvnitř objektu `Compilation`, ze kterého je může přebírat jak IDE, tak libovolný generátor kódu.

Implementovaný generátor kódu převádí výstup kompilátoru do zdrojových souborů jazyka C++, které jsou dál kompilovány C++ kompilátorem do modulu dll. Návrh původně počítal s paralelním generováním mezikódu určeného k interpretaci. Protože ale byly odstraněny všechny nevýhody kompilovaných skriptů, mohlo být od generování mezikódu upuštěno. Výhody kompilovaných skriptů jsou v rychlosti a jednoduchosti implementace, není potřeba interpretovat žádný mezikód, výstup generátoru kódu ve formě jazyka C++ je snadno čitelný a je možné ho ladit. Hlavní nevýhodou byla uživatelská nepřívětivost, ale nyní lze celý proces kompilace a generování výsledného dll spustit jedním kliknutím.

Runtime byl upraven tak, aby zvládal nový jazyk a novou verzi kompilovaných skriptů. Nově byla navržena vícenásobná dědičnost a byl implementován jednoduchý garbage collector.

Budoucí vývoj se může ubírat jak k dokončení nové verze Systému Krkal, tak k použití existujících komponent v nezávislých aplikacích. Jazyk Krkal C může být

například nasazen v masivně multiplayerových online hrách, které budou simulovat velké množství rozmanitých objektů. Díky vlastnostem podporujících rozšiřitelnost a reflexi, bude možné do běžícího systému zasahovat, měnit pravidla nebo přidávat nové prvky.

Implementace prostředí podporujícího vývoj her, je velmi náročný a komplexní úkol. Existující systémy se většinou soustředí na určitou oblast, kde vynikají nad ostatními. Systém Krkal se zaměřil na programovací jazyk, na simulaci žijících objektů a na rozšiřitelnost.

Literatura

[1] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D. (1986): Compilers Principles, Techniques, and tools. Addison-Wesley Publishing Company.

[2] Altman, P., Krček, J., Margarirov, J., Poduška, J. (2004): Krkal, Softwarový projekt, MFF UK Praha. http://www.krkal.org.

[3] Gries, D., Martin, A. J., van de Snepscheut, J. L., and Udding, J. T. (1989): An algorithm for transitive reduction of an acyclic graph. *Sci. Comput. Program.* **12,2** (Jul. 1989), 151-155.

[4] C# Language Specification 1.2. Microsoft Corporation. http://msdn2.microsoft.com/en-us/vcsharp/Aa336809.aspx.

[5] Overmars, M.: Game Maker. http://www.yoyogames.com/gamemaker.

[6] Sweeney, T.: Unreal Script. Epic Games. http://wiki.beyondunreal.com/wiki/UnrealScript_Language_Reference.

Příloha A – Ovládání vývojového prostředí

Spuštění programu a systémové požadavky

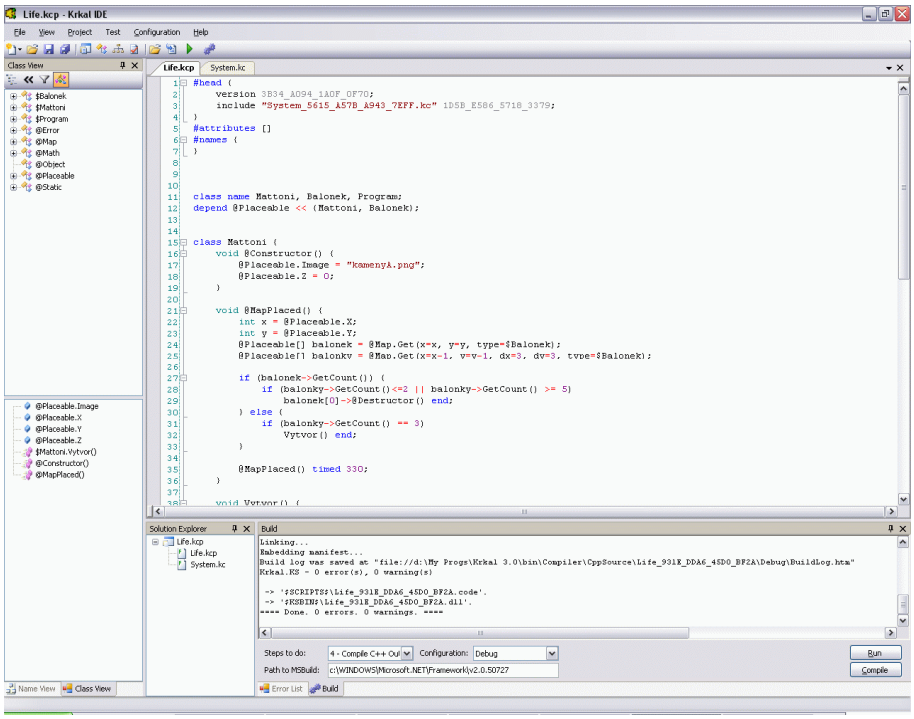
- Je třeba okopírovat adresář `Krkal 3.0` na pevný disk.
- Je požadován .NET Framework 2.0.
- Libovolná verze Microsoft Visual Studia 2005 se service packem 1, součástí instalace musí být C++ kompilátor. Je možné použít i Express edici, v tom případě je potřeba stáhnout i Platform SDK a nastavit cesty k jeho hlavičkovým souborům a `.lib` souborům u všech projektů v adresáři `Krkal 3.0\bin\Compiler\CppSource`.
- C Runtime, který byl použit při kompilaci Krkala 3.0 (je součástí VS 2005 SP1 a je přiložen na CD).
- Vývojové prostředí se spouští přes aplikaci Sample (nikoli IDE), která se nachází zde: `Krkal 3.0\bin\Bin\Krkal.Sample.exe`.

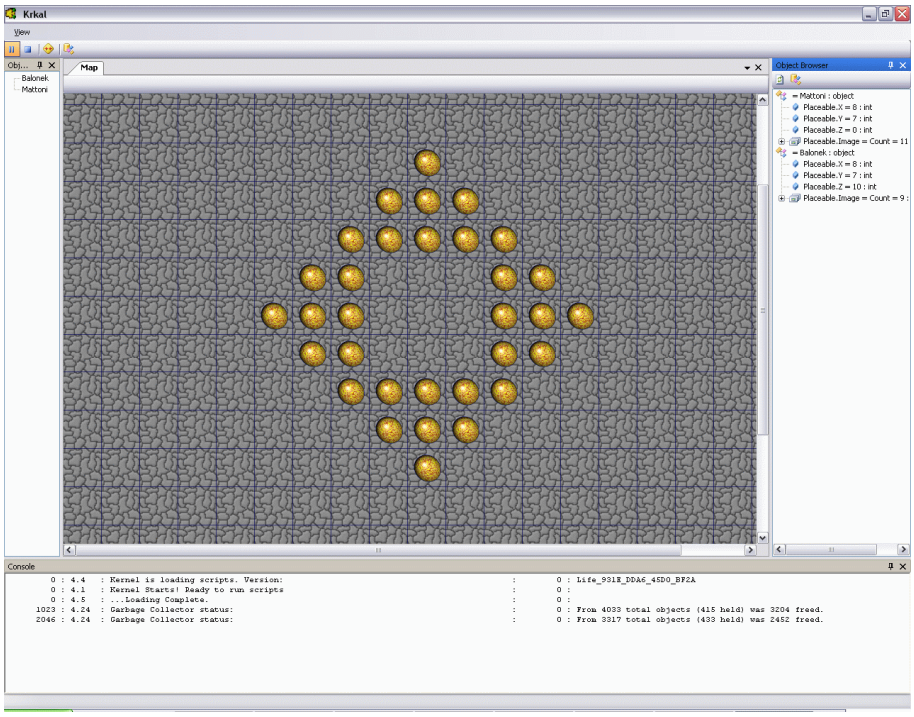
Editace skriptů

Po startu vývojového prostředí je nejprve nutné zvolit projekt. Volba projektu určuje, co se bude kompilovat a spouštět. V daném okamžiku může být zvolen maximálně jeden projekt. Editované soubory nemusejí s projektem souviset.

Projekt se volí například v menu *Project/Choose Project*. Otevře se dialog s nabídkou souborů, projekty mají příponu `.kcp`. Je možné zvolit jeden z existujících nebo vytvořit a zvolit nový. Důležité je nepoužívat funkce dialogu, které mění aktuální adresář, aplikace Sample na to není připravena.

Po zvolení projektu se soubory v něm obsažené objeví v okně *Solution Explorer* (pokud je zapnuta kompilace na pozadí). Dvojitým kliknutím na některý z vypsaných souborů se soubor otevře v editačním okně.





Nyní je možné psát kód a sledovat aktuální výsledky kompilace pomocí oken *Error List*, *Class View* a *Name View*. Dvojité kliknutí na některou položku ji zvýrazní ve zdrojovém textu. Prohlížeče jmen a tříd jsou poměrně široce konfigurovatelné pomocí tlačítek na jejich nástrojové liště.

Editor v současné době nedisponuje kontextovou nápovědou, ale přesto v něm fungují některé základní funkce jako Undo, kopírovíní, vyhledávání nebo odsazování textu. **Přes kontextové menu lze jednoduchým způsobem vkládat příkazy `include` do hlavičky.**

Kompilace

Současná implementace požaduje, aby kompilované skripty byly kompilovány pod stejným kompilátorem, jako byl kompilován zbytek systému. Musí se shodovat konfigurace (Debug / Release) i verze C Runtimu. Tato striktní závislost je považována za chybu a v nejbližší době bude odstraněna.

Je třeba otevřít okno *Build Settings and Output* a zkontrolovat nastavení. Verze dodaná na CD byla zkompilovaná pod VS 2005 SP 1 jako Release, proto i na počítači, kde je aplikace Sample testována, musí být VS 2005 SP 1 a konfiguraci kompilace je třeba nastavit na Release (je to výchozí hodnota). Dále je třeba zkontrolovat, že cesta k souboru MSBuild, který je součástí .NET Frameworku, je správná.

Pokud ano může být spuštěna kompilace, včetně generování výstupů a kompilace C++ souborů. Kompilaci je možné odstartovat klávesou F7, úplné překompilování všeho (*Rebuild All*) lze spustit z menu nebo pomocí zkratky Ctrl + Alt + F7.

Běh

V případě, že kompilace proběhla úspěšně, je možné skripty spustit (F5). Otevře se nové okno s nástroji *Map*, *Object List*, *Object Browser* a *Console*. Na hlavní nástrojové liště jsou tlačítka, která dovolují běh dočasně pozastavit nebo ukončit, a je zde i tlačítko, které dovoluje explicitně spustit garabage collector.

Jestliže skript pracuje s mapu, je možné sledovat život objektů na herním plánu. Levé tlačítko myši vybere všechny objekty na příslušné buňce a vypíše jejich členské proměnné v okně *Object Browser*. V okně *Object List* se dají volit umístitelné objekty a ty pak pravým tlačítkem umísťovat do mapy.

Object Browser slouží k prohlížení stavu objektů. Hodnoty proměnných se načítají v okamžiku rozbalení uzlu, poté se nemění do té doby, než je uzel zabalen a znovu rozbalen. *Object Browser* obsahuje i tlačítko *Refresh* a tlačítko *Static Variables*, které zobrazí všechny statické proměnné. Pokud je v prohlížeči vybrán uzel s objektem, je možné tento objekt zničit klávesou Delete. Zničený objekt bude odstraněn z mapy.

Posledním oknem je *Console*, kde se vypisují chybové, informační a uživatelské hlášky.

Runtime je navržen tak, aby bylo možné spustit více běhů najednou, to skutečně funguje, stačí se vrátit do editačního prostření a odstartovat tentýž nebo jiný projekt znovu.

Příloha B – Obsah CD

- C Runtime — Instalační soubor C Runtimu
- Diplomova prace — Diplomová práce ve formátu pdf
- Krkal 2.4 — Instalační soubor původního Systému Krkal
- Krkal 3.0 — Binární a zdrojové soubory komponent, které popisuje tato práce
  - bin — Binární a datové soubory; výstup kompilace adresáře src
    - Bin — Moduly .dll a .exe
    - Compiler — Výstup kompilátoru a generátoru kódu
      - Bin — Kompilované skripty přeložené do modulů dll
      - Code — Typová informace
      - CppSource — Zdrojové soubory kompilovaných skriptů v jazyce C++
    - Data — Data
    - Games — Zdrojové soubory jazyka Krkal C
    - Help — Popis příkladů implementovaných v jazyce Krkal C
  - src — Zdrojové soubory komponent systému, solution.
    - Krkal.CodeGenerator — Generátor kódu. Vytvořen v rámci této práce.
    - Krkal.Compiler — Kompilátor. Vytvořen v rámci této práce.
    - Krkal.Compiler.Launcher — Testovací utilita. Vytvořena v rámci této práce.
    - Krkal.FileSystem — File System. Převzat z původního Krkala a mírně upraven.
    - Krkal.FileSystem.Net — .Net wrapper. Vytvořen v rámci této práce.
    - Krkal.Ide — Prostředí pro editaci skriptů. Vytvořeno v rámci této práce.
    - Krkal.Runtime — Runtime. Převzat z původního Krkala a značně upraven.
    - Krkal.Runtime.Net — .Net wrapper. Vytvořen v rámci této práce.
    - Krkal.Sample — Hlavní aplikace. Vytvořena v rámci této práce.
    - Krkal.Sample.Services — Služby. Vytvořeny v rámci této práce.
    - Puzzle.CoreLib — SyntaxBox. Převzatá komponenta.
    - Puzzle.SyntaxBox — SyntaxBox. Převzatá komponenta.
    - Puzzle.SyntaxDocument — SyntaxBox. Převzatá komponenta.
    - Shared — Sdílené soubory.
    - Utils.ErrorGenerator — Utilita pro generování dat. Převzato z původního Krkala.
    - WinFormsUI — DockPanel Suite. Převzatá komponenta.
    - zlib — zlib komprese. Převzatá komponenta.