Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Vývojové prostředí pro kurz Paradigmata programování

2024

Jakub Mazel

Vedoucí práce:
doc. RNDr. Michal Krupka, Ph.D.

Studijní program: Informatika,
Specializace: Obecná informatika

Bibliografické údaje

Autor:	Jakub Mazel
Název práce:	Vývojové prostředí pro kurz Paradigmata programování
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2024
Studijní program:	Informatika, Specializace: Obecná informatika
Vedoucí práce:	doc. RNDr. Michal Krupka, Ph.D.
Počet stran:	39
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Jakub Mazel
Title:	Integrated development environment for the Programming Paradigms course
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2024
Study program:	Computer Science, Specialization: General Computer Science
Supervisor:	doc. RNDr. Michal Krupka, Ph.D.
Page count:	39
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Bakalářská práce se zabývá vytvořením vývojového prostředí s integrací výuky pro kurz Paradigmata programování. V první sekci obecně popisuji vývojové prostředí Polyglot a proč vznikla potřeba vytvořit alternativu k LispWorks Personal. V druhé sekci se zevrubně věnuji implementaci. V poslední části popisuji uživatelské rozhraní a používání programu.

Synopsis

The Bachelor thesis deals with creating an integrated development environment with the integration of tuition for the Programming paradigms course. In the first section, I generally describe IDE Polyglot and why the need to replace LispWorks Personal arose. In the second section, I dive deeply into implementation details. In the last part, I describe user interface and the use of the program.

Klíčová slova: vývojové prostředí; výuka; lisp

Keywords: integrated development environment; teaching; lisp

Děkuji doc. RNDr. Michalu Krupkovi, Ph.D. za jeho cenné rady, podnětné připomínky a pomoc při práci na programu.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

Seznam obrázků

Seznam zdrojových kódů

1 Úvod

Pro studenta, který se učí programovat, je důležité kvalitní vývojové prostředí. V této práci vytvořím vývojové prostředí, které budou používat studenti informatiky v kurzu Paradigmata programování, který je vyučován na Katedře informatiky Přírodovědecké fakulty Univerzity Palackého v Olomouci. Kurz je stěžejní součástí studia programování v prvních ročnících. Cílem bylo vytvořit sjednocující platformu pro studenty a vyučující v rámci kurzu. Základem je jednoduché uživatelské rozhraní, které je přizpůsobeno potřebám výuky, a integrace kurzu přímo do programu.

V současné době se ve výuce kurzu Paradigmata programování používá vývojové prostředí LispWorks Personal. LispWorks Personal má několik omezení, protože se jedná o bezplatnou verzi komerčního produktu. Mezi omezení LispWorks Personal patří časový limit pět hodin na používání aplikace [1]. Kromě omezení je LispWorks Personal také příliš komplexním nástrojem pro potřeby kurzu.

Vývojové prostředí Polyglot, které jsem v rámci práce naprogramoval, omezení LispWorks Personal nemá, a navíc lépe splňuje požadavky vyučujících i studentů pro výuku.

Vývojové prostředí má dva typy uživatelů, studenta a vyučujícího. Vyučující má přístup na server, na který může přidávat zdrojové soubory a zadávat úkoly z jednotlivých lekcí. Student má tyto zdrojové soubory a úkoly z lekcí v programu okamžitě dostupné.

Na začátku kurzu dostane student základní minimální verzi programovacího jazyka Lisp. V průběhu kurzu jej bude vyučující rozšiřovat skrze zdrojové soubory z jednotlivých lekcí. Student v rámci úkolů, ve kterých bude zejména programovat nové funkce, jazyk také rozšíří, což mu usnadní pochopení probíraného učiva. Tím se z vývojového prostředí stane platforma, která zahrne podstatnou část kurzu. Zajistí prostředí pro programování s integrací výuky skrz zdrojové soubory.

2 Vývojové prostředí

Vývojové prostředí[1] je program, který poskytuje nástroje pro vývoj softwaru. Minimální vývojové prostředí pro dynamický programovací jazyk by mělo obsahovat textový editor na úpravu zdrojových kódů a kompilátor (nebo interpret) na vyhodnocování výrazů.

Vývojové prostředí Polyglot vzniklo jako nástroj pro výuku v kurzu Paradigmata programování, ve kterém se studenti naučí různé styly programování (paradigmata). Doposud se jako primární vývojové prostředí používal LispWorks Personal, který má omezení. Mezi omezení LispWorks Personal patří pětihodinový časový limit a omezení velikosti zásobníku. LispWorks Personal také obsahuje nástroje, které student během kurzu Paradigmata programování nepoužije, což dělá uživatelské rozhraní složitější. Student může v částech kurzu, které nevyužívají nástrojů od LispWorks, používat i jiné distribuce Common Lispu a vývojová prostředí, nicméně většina studentů používá LispWorks Personal. Ve vývojovém prostředí Polyglot zmíněná omezení nejsou. Program je navržen tak, aby byl co nejjednodušší a zároveň poskytoval veškerou potřebnou funkcionalitu pro kurz Paradigmata programování. Polyglot navíc oproti LispWorks Personal i alternativám přináší nové možnosti práce v rámci kurzu jak pro vyučujícího, tak pro studenta.

Pokud student před začátkem kurzu již programoval, nejspíš se setkal s některým z nejpoužívanějších vývojových prostředí. Mezi ty podle počtu hledání na Googlu patří Visual Studio, Visual Studio Code, Eclipse a PyCharm [2]. U některých jsem se inspiroval při tvorbě uživatelského rozhraní. Za zmínku stojí číslování řádků, které v LispWorks Personal chybí, nebo přehledná indikace neuložených úprav v editoru. Cílem bylo, aby byl přechod na nové prostředí bezproblémový.

Uživatelské rozhraní vývojového prostředí Polyglot je složeno ze tří podoken, umístěných v rozhraní pod sebou. Nazývají se *editor*, *listener* a *helper*. V editoru uživatel píše zdrojový kód, například funkce nebo metody. Také v něm může plnit úkoly zadané od vyučujícího. Listener je interaktivní konzole, ve které lze vyhodnocovat výrazy a testovat funkce napsané v editoru. V helperu se zobrazují uživateli informace, které jsou nápomocné při psaní programu v editoru a listeneru.

Vývojové prostředí má dva typy uživatelů:

- Student – má přístup k souborům přednášek, ve kterých bude plnit úkoly,
- Vyučující – má přístup na server, přes který může sdílet soubory.

V textu budu oba typy uživatelů označovat souhrnně uživatel. Pokud bude potřeba, typ uživatele specifikuji.

[1] Anglicky Integrated development environment, neboli IDE.

2.1 Předmět Paradigmata programování

Předmět Paradigmata programování se vyučuje na Přírodovědecké fakultě Univerzity Palackého v Olomouci. Je vyučován v rámci programů Obecná informatika a Programování a vývoj software. Cílem tohoto předmětu je naučit studenty základy programování a seznámit je s různými přístupy řešení problémů. Předmět se dělí do čtyř částí, neboli kurzů. Každý kurz je jednosemestrální, všechny kurzy dohromady trvají čtyři semestry. Kurz Paradigmata programování 1 je zaměřen na funkcionální paradigma. První verze vývojového prostředí byla vyvinuta zejména pro první kurz. Kurz se dělí do dvanácti přednášek, které na sebe tématicky navazují.

Pro výuku kurzu se používá programovací jazyk, který vychází z Common Lispu. Výhodou jazyka vycházejícího z Common Lispu jsou nástroje umožňující jeho tvárnost. Common Lisp má například systém maker, díky kterému lze jednoduše rozšířit syntaxi jazyka. Je to také dynamický programovací jazyk, u kterého lze upravovat program za běhu. To podporuje inkrementální vývoj vhodný pro výuku. Díky těmto nástrojům ho lze jednoduše přizpůsobit paradigmatu, které se zrovna vyučuje.

2.2 Systém kurzů ve vývojovém prostředí

Hlavní myšlenka při tvoření systému kurzů a lekcí do vývojového prostředí byla taková, že student dostane během semestru ke každé přednášce (lekci) zdrojový kód s naprogramovanými funkcemi. Naprogramované funkce budou součástí přednášky. K funkcím z přednášky dostane student sadu úkolů. Náplní většiny úkolů bude napsat funkci, která doplní nebo použije funkce z přednášky. Na začátku kurzu student dostane k dispozici minimální programovací jazyk. Program se před tím, než vyučující přidá zdrojové soubory z přednášek, bude nacházet ve výchozím stavu. Výchozí stav obsahuje minimální jazyk, ve kterém je přístupná veškerá funkcionalita, kromě té, která je definována v souborech přednášky. Jazyk se v průběhu kurzu rozšíří o funkce vyučujícího i studenta.

Zdrojový kód k lekci vyučující přidá na server do složky s kurzem. Program si při spuštění tento soubor načte. Vývojové prostředí se bude při spuštění vždy nacházet ve výchozím stavu nebo stavu poslední přednášky. Je možné spustit vývojové prostředí bez přístupu k internetu. V tomto případě se nemůžou načíst žádné soubory ze serveru a načte se buď poslední uložená přednáška na zařízení nebo se vývojové prostředí spustí ve výchozím stavu. Výchozí stav obsahuje základní jazyk a umožňuje veškerou funkcionalitu nezávislou na připojení k internetu, jako například úpravu souborů.

Způsob průběhu představeného kurzu má své problémy. Může nastat situace, kdy student nebude stíhat naprogramovat funkce z úkolů do příští přednášky. Funkce budou zapotřebí v následujících přednáškách, jelikož nové funkce budou funkce z předchozích úkolů používat. Je možné, že během používání se tyto problémy neobjeví nebo vzniknou jiné.

2.3 Editor

Editor je hlavní součástí vývojového prostředí, protože v něm uživatel pracuje většinu času. Pomocí editoru je možné upravovat a vytvářet nové soubory se zdrojovými kódy. Všechny úkoly a programy, student vypracovává v editoru. V programu Polyglot je podobný editoru z LispWorks Personal. Navíc přidává číslování řádků, čímž se vzhledově přibližuje moderním vývojovým prostředím.

Klávesové zkratky

Editor poskytovaný od LispWorks obsahuje příkazy, které lze namapovat na klávesové zkratky. Velkou část zkratek a příkazů jsem v mém vývojovém prostředí nepotřeboval. Zejména se jednalo o zkratky, které byly převzaté z Emacs[2] a zkratky na práci s editorem od LispWorks. Emacsové zkratky by byly dostačující pro potřeby kurzu, nicméně práce s Emacs se od moderních vývojových prostředí natolik liší, že by adaptace studentů na ně trvala dlouho. Zkratky pro práci s editorem LispWorks poskytují funkcionalitu, která je nad rámec kurzu a proto musely být odstraněny, aby výsledné vývojové prostředí bylo uživatelsky co nejjednodušší. Některé zkratky musely být doplněny, jelikož chyběly a zároveň jsou součástí moderních textových editorů. Tímto jsem chtěl používání přiblížit co nejblíže tomu, na co je student zvyklý u ostatních vývojových prostředí. Většina dostupných zkratek se vztahuje k navigaci v textu. Mezi základní zkratky, které jsou dostupné v první verzi vývojového prostředí patří například zkratky „Copy“, „Cut“, „Paste“ nebo pohyb kurzoru. Další skupina zkratek zahrnuje práci s upravovaným souborem. Zde patří ukládání, načítání a tvoření nového souboru. Následně zde jsou zkratky na vyhodnocování výrazů v editoru, zkratky vztahující se k listeneru a zkratky na přepínání zaměření mezi editorem a listenerem. Během výuky se může stát, že některé klávesové zkratky budou uživatelům chybět. Vyučující bude mít možnost, přidat soubor se zkratkami na server podobně jako soubory s lekcemi.

Číslování řádků

Moderní vývojová prostředí mají číslování řádků upravovaného souboru pro zlepšení přehlednosti kódu. Hlavní využití má při komunikaci, kde se programátoři (nebo program samotný) mohou odkázat na konkrétní řádek. Ve výuce programování je to důležitý prvek, protože diskuze nad zdrojovým kódem je důležitá součást výuky. V LispWorks Personal je v uživatelském rozhraní vidět pouze kolik řádků má otevřený zdrojový kód a v jakém rozsahu jsou v Editoru zobrazeny. Program Polyglot jsem rozšířil o podokno zobrazující čísla řádků u aktuálně upravovaného souboru. Výhodou je, že studentovi bude uživatelské rozhraní připomínat jiná běžně používaná vývojová prostředí, díky čemu si rychleji zvykne na nový program.

[2]Emacs je vývojové prostředí napsané v Lispu z roku 1985

Zobrazování lambda seznamů

Každá funkce má ve své definici určený seznam parametrů, se kterými je volána. Například funkce pro sčítání má libovolný počet parametrů. Z parametrů funkce lze vytvořit seznam, kterému se v Lispu říká lambda seznam. Lambda seznam funkce pro sčítání vypadá takto: `(&rest args)`, kde klíčové slovo `&rest` určuje libovolný počet parametrů. Pro funkci faktoriál by zase mohl vypadat následovně: `(n)`, protože faktoriál má na vstupu jedno číslo.

Zobrazení lambda seznamu během psaní programu je pro programátora, který je v začátcích nápomocné, protože nezná parametry všech funkcí, které používá. LispWorks Personal poskytuje způsob na zobrazení lambda seznamu funkce, ale studenti ji nepoužívají často. Může to být dáno tím, že si příslušnou zkratku nenajdou v dokumentaci. Ve vývojovém prostředí Polyglot se lambda seznam zobrazuje uživateli automaticky, když napíše název funkce.

Vyhodnocení definic

Vývojové prostředí pro dynamický programovací jazyk by mělo mít implementované nástroje, kterými lze definovat kompilované nebo interpretované funkce. V LispWorks Personal jsou dostupné oba způsoby vyhodnocení. Program Polyglot zatím poskytuje pouze možnost definovat funkce jako interpretované. Tento způsob byl zvolen z důvodu jednodušší implementace. Do budoucna by mohl být přidán způsob, kterým by bylo možné definice a výrazy kompilovat podobným způsobem jako v LispWorks Personal.

Uživatel může definice vyhodnotit dvěma způsoby, stejně jako v LispWorks Personal. Může vyhodnotit buď výraz nejblíž kurzoru nebo všechny výrazy v editoru. V LispWorks Personal je výsledek vyhodnocení zobrazen v podokně s názvem „Output“, na které lze překliknout z Editoru nebo Listeneru. Rozhodl jsem se podokno na zobrazování výsledků vyhodnocení neimplementovat a výsledky zobrazit uživateli do Listeneru. Tím se zjednoduší uživatelské rozhraní, protože je redukováno na menší počet podoken.

2.4 Listener

Listener je typickým nástrojem pro lispové jazyky. Jedná se o interaktivní konzoli, do které uživatel může zadávat příkazy. Listener příkaz přečte, vyhodnotí, vytiskne návratovou hodnotu a celý proces opakuje. Z tohoto důvodu se listeneru říkalo anglicky *read-eval-print loop* [3], což se později zkrátilo na označení REPL. První implementace listeneru vznikla roku 1964 [4]. Berkeley a Deutsch vytvořili listener pro počítač PDP-1.

Listener je součástí vývojového prostředí LispWorks. Během kurzu student často pracuje s Listenerem, jedná se o důležitou součást celého vývojového prostředí. Student může testovat napsaný kód okamžitě, bez potřeby kompilace a spouštění celého programu. Listener pro Polyglot jsem navrhl tak, aby lépe vyhovoval potřebám studenta kurzu.

Prvním krokem bylo zjednodušení uživatelského rozhraní. Common Lisp používá balíčky, což je obdoba jmenných prostorů. To v jakém se listener nachází balíčku LispWorks Personal ukazuje u promptu. Tato informace je důležitá pro programátory pracující na pokročilých projektech ve více balíčcích. Během kurzu Paradigmata programování se student s prací s balíčky nesetká. Zobrazení názvu aktuálního balíčku je proto pro studenta informace, kterou nepotřebuje vědět. Ve vývojovém prostředí se o aktuální balíček stará program interně a student není touto režií zatížen.

Listener v LispWorks Personal má nástroje na obsluhu chybového stavu. Pokud uživatel do listeneru zadá výraz na vyhodnocení, který vyvolá výjimku, přejde listener v LispWorks Personal do chybového stavu, který si žádá akci uživatele. Při každém takovém vstupu do chybového stavu listener vytiskne informaci o výjimce. Obvykle se zobrazí název výjimky a s jakými argumenty byla vyvolána. V této chvíli má podle typu výjimky uživatel na výběr z několika možností, jakými výjimku obslouží. Mezi tyto možnosti patří například návrat z chybového stavu bez obsloužení, vrácení jiné hodnoty z chybového výrazu, zavolání výrazu s jinými argumenty nebo vytisknutí zásobníku. Ve většině případů během kurzu student volí možnost vrácení se z chybového stavu bez obsloužení. Tato možnost je tedy jediná v listeneru mého vývojového prostředí. Z toho důvodu v něm neexistuje žádný chybový stav, který by byl třeba obsluhovat.

Listener v LispWorks Personal nebrání uživateli mazat již vyhodnocené výrazy. Pokud v LispWorks Personal vyhodnotíte výraz a listener vrátí hodnotu, lze zpětně kurzorem text upravit nebo označit a smazat. Možnost mazat již vyhodnocené výrazy nedává smysl, protože historie vyhodnocených výrazů by měla být neměnná. Mé vývojové prostředí řeší tento problém a mazat vyhodnocené výrazy nejde.

V LispWorks Personal chybí v listeneru jednoduchý způsob, jakým by se uživatel mohl dostat k předchozím vyhodnoceným výrazům. V LispWorks Personal je rychlý přístup možný, ale ve výchozím stavu nemá namapovanou klávesovou zkratku. Uživatel musí klávesovou zkratku sám v nastavení určit. Program Polyglot má tuto funkcionalitu implementovanou. Při implementaci jsem se inspiroval terminálem v MacOS, takže používání je velmi podobné.

2.5 Helper

Helper je nejmenší okno a jeho hlavní úkol je zobrazovat informace, například lambda seznamy funkcí. Uživatel do helperu nemusí nic zadávat, veškerá jeho zodpovědnost je zobrazování informací. Okno bude mít výchozí velikost jednoho řádku, protože většina zpráv zobrazovaných v helperu je krátkých. Pokud bude třeba, uživatel si okno bude moci zvětšit a přečíst si delší zprávu zobrazenou v helperu, popřípadě přečíst více zpráv.

Do helperu se v této verzi vypisuje lambda seznam funkcí a maker. Je možné, že se v budoucnu v helperu budou objevovat i další typy zpráv, například informace o přednáškách nebo špatně otevřených souborech.

2.6 Záplaty

Během semestru nastane situace, kdy některý z uživatelů objeví chybu v programu. Možným řešením je chybu v softwaru opravit a vydat jeho novou verzi, kterou si uživatelé stáhnou. Tento přístup má své potíže. Stahovat při každé opravě chyby novou verzi programu by bylo velmi nepraktické. Část uživatelů by si novou verzi nestáhla. K jiným by se nemusela dostat informace o vydání nové verze. Proto je třeba najít způsob, který by tento proces oprav chyb zautomatizoval a chyby v programu opravoval sám bez nutnosti stahování nové verze programu. Tento problém řeší tzv. záplata (anglicky patch).

Záplata je zdrojový kód, který opravuje chybu v programu. Pokud je chyba v nějaké funkci, záplata bude obsahovat novou definici této funkce s opravenou chybou. Při spuštění programu se nejprve načtou všechny stažené záplaty a předefinují funkce obsahující chyby. Tím se chyba opraví bez nutnosti stahovat celý program s opravenou chybou.

Výhodou tohoto systému je, že na chyby bude možné reagovat rychleji, než to je u LispWorks, které má velké množství uživatelů. LispWorks Personal nemá takto jednoduchý systém záplat, a proto je u něj doba čekání na opravu chyby delší. Zároveň je důležité zmínit, že LispWorks Personal nebude mít tolik chyb jako Polyglot, protože se jedná o komerční produkt, na kterém pracuje tým programátorů.

2.7 Delivery aplikace

Posledním krokem při vývoji programu je *delivery*[3]. V manuálu LispWorks [5] je delivery popsán jako proces, při kterém vznikne spustitelný program nebo dynamická knihovna. V mé práci jsem se věnoval tvorbě spustitelného programu. Při procesu delivery se z naprogramované aplikace vytvoří soubor spustitelný nezávisle na vývojovém prostředí LispWorks. Ve vzniklé aplikaci již není dostupné vývojové prostředí LispWorks a některé další funkce, aby velikost výsledného souboru byla co nejmenší. Tato aplikace je spustitelná pouze na platformě, na které byl delivery proveden. To znamená, že pro vytvoření aplikací pro všechny platformy je potřeba vlastnit licenci LispWorks Professional na každou platformu. Základem delivery je *delivery skript*, který je při delivery spuštěn. Skript načte celý program a spustí funkci `deliver`. Ve skriptu může uživatel určit název výsledné aplikace, kam se uloží a mnoho dalších parametrů.

Jakmile jsem z aplikace udělal spustitelný program, narazil jsem na problém v MacOS. Na všech zařízeních, kromě toho, na kterém byl delivery proveden při spuštění vyskočilo dialogové okno s upozorněním, že aplikace je poškozená a nejde spustit. Před vydáním bude potřeba program digitálně podepsat, aby operační systém mohl ověřit bezpečnost programu.

[3]Delivery je dostupný pouze ve verzi LispWorks Professional

3 Implementace

Následující kapitola práce je technická a zabývá se zejména implementací vývojového prostředí. Do ní patří vysvětlení hierarchií tříd, jakým způsobem spolu komunikují nebo jak jsou konkrétně řešeny problémy, na které jsem narazil.

Implementace celého programu probíhala zhruba od února 2023 do dubna 2024. Vývoj aplikace probíhal iterativně. Vedoucímu práce jsem prezentoval verze programu a další tvořil dle jeho připomínek. Všechny části práce kromě systému záplat, jsem naprogramoval sám s pomocí vedoucího. Vedoucí mi obvykle radil s problémy, na které jsem během vývoje narazil, a také mi pomohl pochopit Common Lisp více do hloubky. Během vývoje programu jsem se musel naučit pracovat s novými technologiemi, které celý vývoj usnadnily. Zejména se jedná o práci s knihovnou CAPI a Editorem od LispWorks. Pro udržování pořádku ve změnách zdrojových kódů a verzích jsem využíval Git a GitHub, skrze který jsem sdílel verze programu s vedoucím.

3.1 Použité technologie

Common Lisp

Vývojové prostředí Polyglot je napsané v jazyce Common Lisp. Common Lisp je dynamický, multiparadigmatický programovací jazyk. Umožňuje vyhodnocovat kód za běhu programu. Díky tomu se program pohodlně testoval a usnadnila se implementace některých jeho částí, například záplat.

První verze Lispu vznikla v letech 1958-1962 na MIT, kde ji poprvé implementoval John McCarthy se svým týmem [6]. Původně vznikl jako funkcionální jazyk [7], nicméně postupem času byly přidány nástroje pro psaní v jiných paradigmatech, například CLOS pro psaní objektově orientovaných aplikací. Existuje několik tzv. dialektů Lispu[4], mezi nejpoužívanější patří Scheme, Clojure a Common Lisp. Roku 1994 vznikla standardizovaná verze Lispu pod názvem ANSI Common Lisp [8]. Od té doby vzniklo mnoho distribucí Common Lispu. Mezi nejpoužívanější patří SBCL, CLISP nebo LispWorks.

Pro implementaci vývojového prostředí jsem se rozhodl použít distribuci LispWorks. Distribuci LispWorks jsem si vybral, protože jsem byl z kurzů Paradigmata programování zvyklý v prostředí pracovat a navíc poskytuje nástroje pro psaní okenních multiplatformních aplikací. Celá aplikace je naprogramována v prostředí LispWorks Professional. Vývojové prostředí je navrženo pro Lisp, tudíž veškeré zpracování zdrojových kódů je přímo v Common Lispu implementováno ve funkci `eval`. Volba jiného jazyka, například JavaScriptu, ve kterém je napsán Visual Studio Code, by byla možná, ale přinesla by problémy, které přímá implementace ve stejném jazyku řeší.

[4] Jedná se o různé programovací jazyky vycházející z původního Lispu.

CAPI a Editor

Použil jsem knihovnu Common Application Programmer's Interface (CAPI) poskytovanou od LispWorks. CAPI je objektová knihovna, která umožňuje programátorovi vytvářet okenní aplikace [9]. CAPI je multiplatformní knihovna, pomocí které lze vytvářet aplikace na všechny hlavní operační systémy (Windows, Linux a MacOS). Knihovna CAPI poskytuje třídy reprezentující okna, menu, rozhraní atd. Správným složením těchto tříd lze sestavit celé uživatelské rozhraní.

LispWorks má implementovaný editor, který poskytuje nástroje pro úpravu textových souborů [10]. Je plně programovatelný, což mi dalo možnost upravit editor pro potřeby kurzu Paradigmata programování. Editor LispWorks vychází ze staršího editoru Emacs, jehož prvky jsou v editoru LispWorks stále přítomny. CAPI poskytuje třídu `editor-pane`, která reprezentuje editor LispWorks. Tato třída je základem editoru ve vývojovém prostředí naprogramovaném v rámci této práce.

CLOS a OOP

Pro vývoj jsem zvolil zejména objektově orientované programování (OOP). Většina komunikace v aplikaci je řešena pomocí OOP, nicméně části jsou napsané v jiném paradigmatu, například funkcionálně. Knihovna CAPI, ve které je program implementován, je napsána v OOP. Využití OOP tak usnadňuje rozšíření tříd, které mi poskytuje CAPI.

Common Lisp pro objektové programování používá tzv. Common Lisp Object System (CLOS) [8]. Přístup k OOP je u CLOS jiný, než u jiných jazyků jako například C++ nebo C#. Jeden z rozdílů je vícenásobná dědičnost, kterou je možné využít i v C++, ale s rozdíly v implementaci.

Další rozdíl je možnost použít klíčová slova `:before` a `:after` v definicích metod [11]. Ty umožňují další kontrolu nad pořadím volání jednotlivých metod. Before metoda se volá před metodou stejného názvu. After metoda po metodě stejného názvu. Tímto systémem lze volání metody obalit a rozdělit do tří různých, což umožňuje jednoduše ovlivnit vytváření instancí tříd.

CLOS poskytuje i systém multimetod. Systém multimetod umožňuje rozhodnout za běhu aplikace, která z metod stejného názvu se zavolá na základě parametrů, se kterými je volána.

Delivery

Posledním krokem při vývoji aplikace je delivery. Delivery je proces, během kterého se ze zdrojových kódů vytvoří spustitelná aplikace na příslušnou platformu. Pro vytvoření spustitelné aplikace je zapotřebí vlastnit LispWorks Professional. LispWorks Professional poskytuje rozhraní pro delivery [5]. Je třeba vytvořit delivery skript, který po spuštění vytvoří spustitelnou aplikaci. V delivery skriptu se specifikuje název aplikace, kde bude výsledná aplikace uložena na disku, jaká funkce se zavolá při spuštění výsledné aplikace a další volitelné parametry.

Git

Během vývoje je důležité udržovat pořádek ve verzích aplikace. Proto jsem používal verzovací systém Git. Zdrojový kód je uložen v repozitáři, který v průběhu vývoje ukládá provedené změny. Tento nástroj umožňuje udržovat různé větve vývoje, komentovat a přidávat změny zdrojových kódů.

Celý Gitový repozitář jsem měl uložený na serveru pomocí služby GitHub. Jedná se o webovou aplikaci umožňující spravovat a sdílet repozitář online. Díky GitHubu se zjednodušila spolupráce s vedoucím práce, protože měl přístup k aktuálním zdrojovým kódům nebo vydaným verzím programu. V GitHubu je uložená dokumentace programu, která je napsána v markdown souboru.

Markdown je jednoduchý jazyk umožňující formátování textu. Lze v něm zapisovat seznamy, odkazovat a formátovat písmo. Pro svoji jednoduchost, dobrý vzhled a integraci v GitHubu jsem upřednostnil Markdown před alternativami.

3.2 Editor

Knihovna CAPI poskytuje třídu `editor-pane`, kterou jsem použil jako základ pro editor ve vývojovém prostředí. Původní třídu jsem rozšířil pro potřeby programu. Třída reprezentuje okno, ve kterém lze načítat a upravovat textové soubory.

Buffer

Pro práci s textem editor LispWorks využívá objekt buffer. Když uživatel otevře soubor, jeho obsah se načte do bufferu, jehož obsah je zobrazen v okně editoru. Buffer si zapamatuje, s jakým souborem je asociován (uloží si cestu k němu). Následně uživatel text upraví, přidá definice, opraví chyby atd. Provedené změny se stále neprovedly v souboru, jelikož veškeré úpravy proběhly v bufferu. Buffer ví, že jeho stav je jiný než ten při načtení a projeví se to ve stavu objektu. Jakmile se uživatel rozhodne soubor uložit, buffer svůj obsah vypíše do souboru. Tím jsou veškeré změny provedeny. Jak je psáno v dokumentaci [10], uživatel neupravuje soubor, nýbrž buffer. Buffer obsahuje data z pohledu editoru, soubor obsahuje data z pohledu operačního systému.

Číslování řádků

Pro modernější vzhled jsem přidal číslování řádků, které zlepší přehlednost a poskytne uživateli informaci o tom, kde ve zdrojovém kódu se nachází. Tato funkcionalita je napsána prakticky od základu. Protože podokno vykreslující řádky je jiná třída než samotný editor, musel jsem mezi nimi zajistit komunikaci. Pro tento účel jsem využil komunikaci přes interface, který obsahuje jak instanci editoru, tak číslovače řádků. Editor posílá zprávu o změně číslovači přes tento interface. Používá pro to `display-callback`, který se volá při každé změně zobrazení v okně editoru (tj. změna v textu, scrollování).

Počet řádků a číslo prvního viditelného řádku jsem získal pomocí systémových funkcí z bufferu. Aktuální výška okna se předává ve zpětném volání pro překreslení okna. Tyto informace stačí pro vykreslení viditelných řádků. Jednoduše jsem vykreslil číslo prvního viditelného řádku (pokud uživatel zascrolluje dolů, může to být například řádek 30) a zbytek jsem dokreslil v cyklu, protože jsem věděl, kolik dalších řádků je možné v okně zobrazit.

Pro zvýraznění aktuálního řádku není dostupná žádná systémová funkce. Proto jsem napsal algoritmus 1 na výpočet aktuálního řádku z pozice kurzoru. Na začátku si algoritmus zjistí offset[5] kurzoru, text, který právě uživatel edituje a položí aktuální řádek roven jedné. Následně v cyklu spočítá veškeré výskyty zalomení řádku (v Lispu je to znak `#\Newline`). Při každém výskytu znaku zalomení se proměnná držící nový řádek inkrementuje o jedna. Tento algoritmus má časovou složitost $O(n)$, protože v jednom cyklu iteruje přes prvky textového řetězce.

Hierarchie tříd

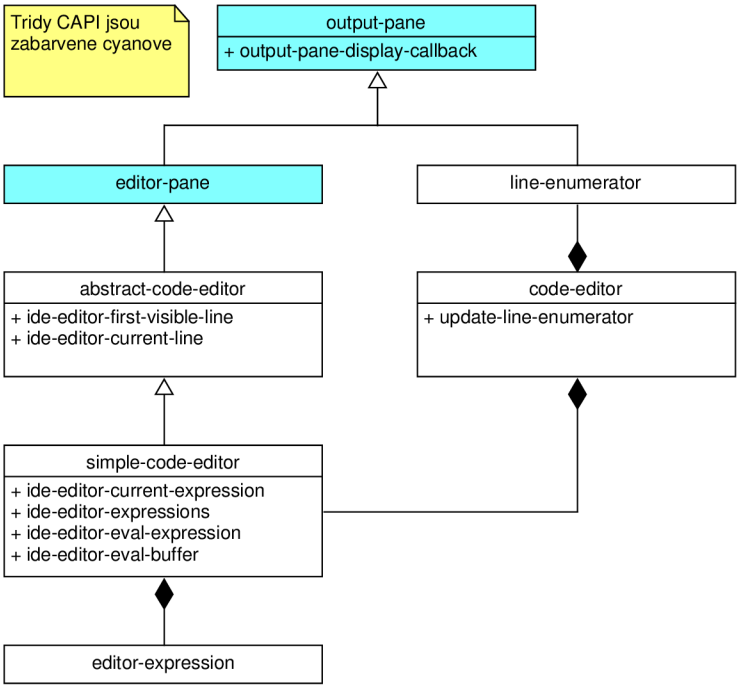
Správnou funkčnost editoru zajišťuje systém tříd. Třídy spolu navzájem komunikují. Základem je abstraktní třída `abstract-code-editor`, která dědí od třídy `editor-pane`, kterou poskytuje CAPI. Její funkcionalitu rozšiřuje o metody, které buď sjednocují pojmenování metod (funguje tedy jako adaptér) nebo rozšiřují základní funkčnost. Veškeré metody spojené s tímto obecným editorem jsem pojmenoval s předponou `ide-editor-`. Na této třídě jsou definovány metody vracející informace o řádcích, jako například aktuální řádek nebo počet řádků. Třída `abstract-code-editor` má definované metody pro vkládání textu do editoru. Ve zpětném volání této třídy je definována metoda na vypisování lambda seznamu. Od třídy `abstract-code-editor` dědí třída `simple-code-editor`, která funkčnost rozšiřuje o vyhodnocování výrazů. Vyhodnocování výrazů využívá třídu `editor-expression`. Instance třídy reprezentuje výraz zapsaný v bufferu. Další důležitá třída je `line-enumerator`, dědící od třídy CAPI `output-pane`. Tato třída reprezentuje okno vykreslující řádky. Je koncipována jako doplněk k `simple-code-editor`. Pro správné fungování jí stačí znát číslo prvního řádku, který má vykreslit, počet řádků na vykreslení a aktuální řádek, který zvýrazní. Třídy `simple-code-editor` a `line-enumerator` je třeba sjednotit, aby spolu mohly komunikovat. To zajišťuje třída `code-editor`, která dědí od třídy CAPI `interface`. Její úkol je správně zobrazit číslovač řádků a editor. To znamená, že veškerá komunikace ohledně změny obsahu v editoru se díky třídě `code-editor` přepíšou do číslovače řádků. Celá hierarchie je znázorněna na obrázku 1.

[5] Vzdálenost v počtu znaků od začátku souboru.

Vyhodnocení

Vyhodnocování výrazů je v kompetenci třídy `simple-code-editor`. Základem je třída `editor-expression`, jejíž instance má uložený výraz a jeho pozici v bufferu. Při stisknutí tlačítka na vyhodnocení aktuálního výrazu editor vytvoří seznam všech výrazů uložených jako instance třídy `editor-expression`. V něm podle pozice kurzoru najde nejbližší výraz ke kurzoru. Následně vybraný výraz vyhodnotí a výsledek vyhodnocení pošle listeneru pro zobrazení. Při vyhodnocení všech výrazů je rozdíl v tom, že se přes všechny prvky seznamu s výrazy iteruje a vyhodnotí se všechny nehledě na aktuální pozici kurzoru. Vyhodnocení probíhá zavoláním funkce `eval` v aktuálním balíčku kurzu. Další možností je funkci zkompilovat. Definice by se musela obalit do lambda výrazu a aplikovat funkce `compile`. Poté bychom měli funkci zkompilovanou namísto interpretované. Hlavní výhodou kompilovaných funkcí oproti interpretovaným by byla jejich rychlost, nicméně během kurzu není rychlost výpočtů (a jejich optimalizace) důležitá.

Při vyhodnocení v editoru musíme zobrazit informaci o výsledku. Může nastat několik situací. Pokud výraz není definice, vyhodnocení proběhne stejným způsobem, jako kdyby byl zadán do listeneru. To znamená, že je na něj zavolána funkce `eval` a výsledek vytisknut v listeneru. Je-li výraz definicí, může vyhodnocení proběhnout s nebo bez chyby. Pokud se vyskytne chyba, je vytištěna do listeneru. Při bezchybné definici (syntakticky) se v listeneru objeví název nově definované funkce. V LispWorks je informace o vyhodnocení z editoru zobrazena v samostatném okně nazvaném „Output“. Rozhodnutí neimplementovat okno „Output“ a použít pro zobrazení informace o vyhodnoceném výrazu listener zjednodušuje uživatelské rozhraní.



Obrázek 1: Diagram tříd editoru

```
1  (defun ide-editor-current-line (ide-pane)
2    (let ((current-line 1)
3          (point-offset (ide-editor-point-offset ide-pane))
4          (text (ide-editor-text ide-pane)))
5      (dotimes (i point-offset)
6        (when (char= (aref text i) #\Newline)
7          (incf current-line 1)))
8      current-line))
```

Zdrojový kód 1: Algoritmus pro zjištění aktuálního řádku

3.3 Listener

Listener jsem implementoval jako rozšíření třídy `abstract-code-editor`. Druhou možností bylo použít třídu `listener-pane` poskytovanou od CAPI. Nakonec jsem se rozhodl implementovat listener od základu, abych měl větší kontrolu nad jednotlivými částmi listeneru. Při implementaci stačilo napsat systém, který bude kontrolovat vstup od uživatele a bude umět vyhodnotit výraz, který uživatel zadá do promptu.

Kontrola vstupu

LispWorks Personal nebrání uživateli mazat text z promptu. Ve vývojovém prostředí jsem tuto možnost odstranil. Při stisku klávesy „backspace“ se kontroluje, zda uživatel může smazat daný znak. Pokud se nachází za bodem pro zápis je vše v pořádku a změna se provede, v opačném případě není uživateli dovoleno text smazat. Uživatel může mít kurzor za bodem pro zápis a mazat text před ním, pokud ho má označený. Je proto nutné kontrolovat, není-li výběr mimo oblast, do které se může zapisovat.

Metoda `before-input-callback` obsluhuje vstup pouze pokud uživatel zadá „enter“ pro vyhodnocení výrazu nebo „backspace“ pro smazání znaku. Při stisknutí klávesy „enter“ se kurzor musí před vyhodnocením a vytisknutím výsledku posunout na konec textu. Tímto je zamezeno situaci, kdy uživatel vyhodnotí výraz s kurzorem uprostřed slova, které by se bez ošetření zalomilo.

Historie vyhodnocení

Vývojové prostředí umožňuje rychlý přístup k vyhodnoceným výrazům. Pro implementaci bylo potřeba vytvořit systém, který uchovává historii všech vyhodnocených výrazů. Když uživatel zažádá o výraz z historie, systém mu ho vrátí.

Pro tento účel jsem definoval třídu `listener-history` uchovávající historii zadaných výrazů. Ve slotu `expressions` má v seznamu uložené výrazy jako řetězce. Pomocí metod `older-expression` a `newer-expression` může uživatel listovat ve výrazech a vybrat, který chce použít. Když listener vyhodnotí výraz a potřebuje ho uložit do historie, zavolá metodu `add-expression`, které předá výraz v řetězci.

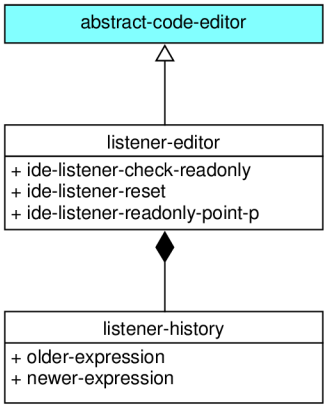
Hierarchie tříd

Na obrázku 2 je znázorněný diagram tříd listeneru. Třída `listener-editor` dědí od `abstract-code-editor` a je rozšířena o kontrolu vstupu a jeho vyhodnocení. Instance třídy `listener-editor` má ve slotu uloženou instanci `listener-history`, která zajišťuje rychlý přístup k vyhodnoceným výrazům.

Vyhodnocení výrazu

Funkce `after-input-callback` je zpětné volání, které se volá po vložení znaku do editoru. Ve vývojovém prostředí reaguje pouze na stisknutí klávesy „enter“, při kterém spustí funkci `handle-enter`. Funkce `handle-enter` 2 zajistí, že se buď vyhodnotí výraz v promptu nebo zalomí řádek.

Začíná tím, že si získá vyhodnocovaný výraz jako řetězec (podřetězec od začátku promptu do konce textu). Získaný řetězec následně převede do lispového výrazu funkcí `ide-editor-read` (výraz se přečte v aktuálním balíčku). Během čtení funkcí `ide-editor-read` může nastat `end of file` chyba. To znamená, že vyhodnocovaný výraz není správně uzávorkován a provede se pouze výchozí akce klávesy, což je zalomení řádku. Díky tomu můžeme v listeneru psát víceřádkové výrazy a klávesa `enter` vyhodnotí výraz pouze v případě správného uzávorkování. Pokud je výraz správně uzávorkovaný, uloží se do historie a vyhodnotí pomocí `ide-editor-eval`. Výsledek vyhodnocení se uloží do proměnné `result`. Výraz uložený v `result` se převede na řetězec a vytiskne se do listeneru.



Obrázek 2: Diagram tříd listeneru

```
(defun handle-enter (pane)
  (let ((input-string (subseq (editor-pane-text pane)
                              (read-only-point pane))))
    (unless (or (ide-listener-new-line-p pane)
                (empty-string-p input-string))
      (ide-editor-new-line pane))
    (handler-case
       (let* ((input (prog1
                       (ide-read-from-string pane input-string)
                     (add-expression (history pane) input-string)))
              (result (ide-editor-eval input)))
         (set-env-variables result)
         (ide-listener-build-message pane result))
      (end-of-file))))
```

Zdrojový kód 2: Algoritmus pro vyhodnocení výrazu v listeneru

3.4 Helper

Implementace třídy `helper-editor` je jednoduchá. Využívá vícenásobné dědičnosti. Dědí od třídy `abstract-code-editor` a `collector-pane`, kterou poskytuje CAPI. Od třídy `abstract-code-editor` dědí, protože využívám metody definované v této třídě. Třída `collector-pane` umožňuje pomocí proudů přijímat text, který je mu poslán. Jediná metoda třídy `helper-editor` je `ide-helper-editor-message`. Přijímá jeden argument, a to řetězec, který následně zobrazí.

3.5 Systém kurzů

Základem systému kurzů ve vývojovém prostředí je třída `course`, která reprezentuje aktuální kurz. Ve slotu `course-path-system`, je instance třídy `file-system` zajišťující práci s adresáři. Ve slotu `lectures` je uložen seznam dostupných lekcí a v `current-lecture` je aktuální lekce.

Při inicializaci instance třídy je potřeba poskytnout `course-path-system`, aby bylo možné načítat soubory s lekcemi. Během inicializace se načte název kurzu a zjistí se dostupné soubory jednotlivých lekcí. Vytvoří se základní lispový balíček, do kterého se nastaví aktuální přednáška. V tomto balíčku pak vývojové prostředí vyhodnocuje všechny výrazy.

Když uživatel změní lekci pomocí zpětného volání v menu, zavolá se metoda `activate-lecture`. Přednáška k aktivaci se předá v rámci zpětného volání. Metoda `activate-lecture` nastaví balíček s aktuální přednášku do výchozího stavu a načte do něj všechny zdrojové kódy z předchozích přednášek.

3.6 Webové rozhraní

Nejdůležitější součástí výukového rozhraní vývojového prostředí je komunikace mezi vyučujícím a studenty. Webové rozhraní ve vývojovém prostředí implementuje způsob, kterým může vyučující sdílet soubory s uživateli programu.

Webové rozhraní obstarává získávání dat z webové stránky. Umí získat obsah stránek ve formátu HTML a také jako lispový zdrojový soubor. Součástí je možnost navigace v rozhraní Apache serveru, který se používá v první verzi vývojového prostředí pro sdílení souborů. Pokud by do budoucna server změnil formu, stačilo by přepsat nebo přidat metody pro práci s novým rozhraním.

Připojení je reprezentováno třídou `connection`, která má slot `url` a dva sloty bez writeru `crlf` a `return-string`. Sloty `crlf` a `return-string` jsou používány při navazování spojení pomocí balíčku `COMM` od LispWorks. Balíček `COMM` zprostředkovává rozhraní pro TCP/IP komunikaci [12]. Metoda `ping` zjišťuje stav připojení k síti. Funguje tak, že se pokusí získat obsah výchozí stránky (slot `url`) a vrátí pravdivostní hodnotu dle výsledku dotazu. Metody `url-file-content` a `url-site-content` získávají obsah stránky v závislosti na jeho typu. Stránka může být buď ve formátu HTML nebo přímo obsah souboru. Pomocí metody `url-download-file` je možné stáhnout soubor do adresáře, který se předá jako argument.

3.7 Souborový systém

Při práci s textovým editorem je potřeba, aby měl implementovaný systém, který si bude udržovat pořádek v přiděleném adresáři, do kterého patří například log, ve kterém budou zapsány informace o chybách v aplikaci. Ve vývojovém prostředí je třeba do adresáře aplikace uložit také soubory záplat, zdrojových kódů z přednášek nebo klávesových zkratek. Minimální požadovaná funkcionalita zahrnuje vytváření, ukládání a načítání souborů. Pokud upravujeme nový soubor, který zatím není nikde uložen, musíme se při uložení dotázat uživatele, kam chce soubor uložit. Klávesovými zkratkami se tedy spouští zpětná volání na ukládání, vytvoření nového souboru a zobrazení dialogu pro výběr souboru k načtení. Oproti LispWorks je tu navíc ještě jedna možnost a to otevřít soubor kurzu. Tento soubor je uživateli schovaný a otevírá se podle toho, jaký kurz je aktuálně zvolený v editoru. Soubor kurzu má v sobě uložené definice jak z přednášek, tak ze cvičení, které si sám naprogramuje.

Při implementaci rozhraní jsem se inspiroval terminálem z UNIXových operačních systémů. Jednotlivé funkce jsou naprogramovány tak, aby používání sytému odpovídalo používání terminálu. Implementoval jsem základní příkazy z terminálu:

- `ls` – vypíše obsah adresáře,
- `cdir`[6] – změní aktuální adresář,

[6]Symbol `cd`, který používá terminál je zabraný, proto jsem zvolil `cdir`.

- `touch` – vytvoří soubor,
- `cat` – zobrazí obsah souboru,
- `mkdir` – vytvoří adresář,
- `rm` – smaže soubor,
- `rmdir` – smaže adresář,
- `echo` – zapíše řetězec do souboru.

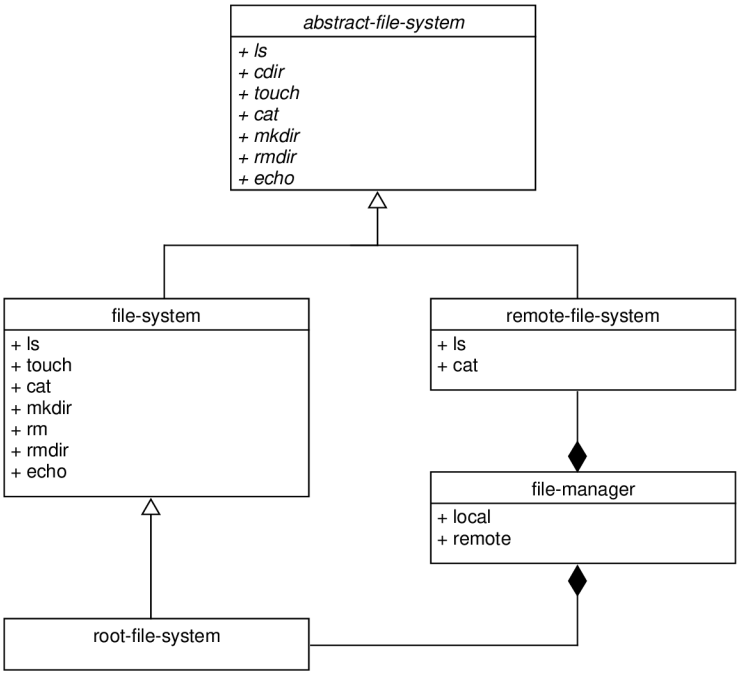
Základem je abstraktní třída `abstract-file-system`, která si ukládá atribut `working-dir` reprezentující aktuální adresář. Všechny zmíněné metody kromě `cdir` jsou abstraktní.

Třída `file-system` reprezentuje souborový systém na disku. Dědí od třídy `abstract-file-system`. Implementuje zbylé zmíněné metody.

Rozšiřující třída `root-file-system` dědí od `file-system` a přidává pouze jinou inicializaci instance. Při inicializaci vytvoří adresáře pro záplaty, logování chyb, kurzy a klávesové zkratky.

Třída `remote-file-system` reprezentuje souborový systém na serveru. Implementace počítá pouze se čtením vzdáleného adresáře. Třída proto implementuje pouze metody `ls` a `cat`. Jaku atribut má uloženou instanci třídy `connection`, která zprostředkovává síťovou komunikaci.

Vývojové prostředí potřebuje synchronizovat zdrojové kódy ze serveru s domovským adresářem. O to se stará třída `file-manager`. Třída `file-manager` má dva atributy, `local` a `remote`. V atributu `local` je uložena instance třídy `root-file-system` a v atributu `remote` je uložena instance třídy `remote-file-system`. Při spuštění aplikace `file-manager` zkontroluje, jsou-li adresáře na serveru a disku stejné. Pokud nejsou, soubory ze serveru chybějící na disku stáhne do příslušného adresáře.



Obrázek 3: Diagram tříd souborového systému

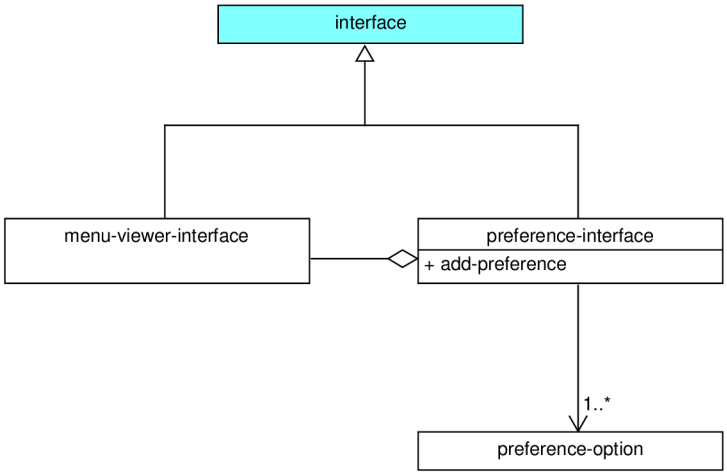
3.8 Předvolby

Předvolby jsou pro uživatele důležitou součástí celého dojmu z programu. V programu Polyglot je zatím okno s předvolbami implementováno velmi jednoduše. Možnosti výběru jsou omezeny na zvolení aktivní lekce. Do budoucna bude přidáno více možností výběru.

Implementace je složena ze tří tříd:

- `menu-viewer-interface` – CAPI interface reprezentující část okna zobrazující nastavení v rámci kategorie,
- `preference-interface` – CAPI interface reprezentující celé okno „Preferences“,
- `preference-option` – třída reprezentující kategorii nastavení.

Při spuštění programu se vytvoří instance objektu `preference-interface`. Nová kategorie nastavení lze instanci přidat metodou `add-preference-option`. Této metodě se předá název kategorie, možnosti a zpětné volání, které se zavolá při změně nastavení.



Obrázek 4: Diagram tříd předvoleb

3.9 Systém záplat

Systém záplat, který program Polyglot používá, naprogramoval vedoucí práce. Jedná se o systém využívající funkce `load`. Při psaní záplaty je potřeba dodržet předem definovanou strukturu, aby záplaty správně fungovaly. Jedinou podmínkou této předepsané struktury je použití makra `patch` na začátku zdrojového kódu záplaty. Systém záplat si ukládá interní verzi programu a podle ní načítá ostatní záplaty. Nové záplaty může vyučující uložit na server. Uživatelům budou automaticky staženy a při spuštění programu načteny.

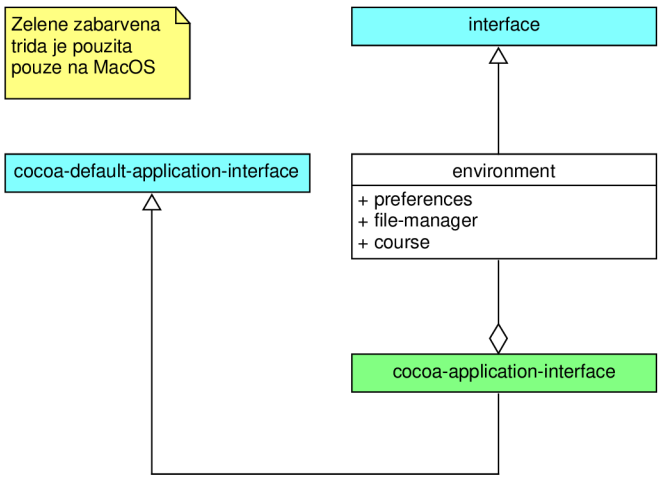
3.10 Okno aplikace

Hlavní okno aplikace je prvek programu, který spojuje všechny jednotlivé části programu, o kterých jsem psal výše. Zajišťuje jejich správný chod a bezproblémovou komunikaci. Definice pro aplikace na Windows a MacOS se liší v detailech, které zejména zahrnují rozdílné přístupy k uživatelskému rozhraní těchto operačních systémů.

Hlavní okno aplikace je reprezentováno třídou `environment`, která dědí od třídy `interface` poskytovanou CAPI. Třída má sloty `preference` s instancí okna předvoleb, `file-manager` starající se o místní i vzdálené adresáře a `course` reprezentující aktuální kurz. V definici třídy `environment` specifikuji jednotlivá menu, sekce a položky, kterým určuji zpětná volání. Ve slotu `panes` určuji jednotlivá podokna, která jsou v tomto případě instance tříd `code-editor`, `listener-editor`, `helper-editor`. Pro správné rozmístění podoken je třeba specifikovat `layout`, kde určuji poměr mezi jednotlivými podokny a jak jsou v okně umístěny. Dalšími argumenty při definici upravuji velikost okna při spuštění a zpětné volání, které se volá při pokusu zavřít aplikaci.

Třídě `environment` upravuji inicializaci instance, přidáním `:before` a `:after` metod metodě `initialize-instance`. V `:before` metodě vytvářím instanci třídy `file-manger`. Následně, pokud má uživatel připojení k internetu, program stáhne nové soubory ze serveru. V opačném případě dostane uživatel upozornění, že není připojen k internetu. V dalším kroku je třeba načíst záplaty. Ty je třeba načíst co nejdříve, aby byla záplata schopná ovlivnit co největší část programu. Nakonec se načtou klávesové zkratky. V `:after` metodě se vytvoří instance reprezentující okno předvoleb. V dalším kroku se inicializují jednotlivé předvolby. Posledním krokem inicializace okna je nastavení názvu okna.

V aplikaci pro MacOS je potřeba definovat třídu, která dědí od třídy poskytovanou CAPI `cocoa-default-application-interface`, aby bylo možné správně využít uživatelské rozhraní MacOS. Jedná se zejména o aplikační menu a další prvky uživatelské zkušenosti specifické pro MacOS. V této třídě se definují menu s aplikačním menu specifické pro MacOS. Lze zde dodat zpětná volání reagující na různé akce, například spuštění aplikace přetáhnutím souboru na ikonu programu. Třídu jsem pojmenoval `cocoa-application-interface` a o její nasazení se stará hlavní funkce programu `main`.



Obrázek 5: Diagram tříd hlavního okna

3.11 Hlavní funkce

Když se snažíme vytvořit spustitelnou aplikaci, je třeba určit, která funkce se zavolá při kliknutí na ikonu. Taková funkce nemá žádné parametry[7]. Funkce, která je volána při spuštění vývojového prostředí, se jmenuje `main`. Zdrojový kód funkce `main` 3 je zjednodušen, má vynechané implementační detaily pro zlepšení přehlednosti. Prvním krokem je vytvořit instanci třídy `environment`. Během inicializace se provedou všechny akce specifikovány v sekci 3.10. Do globální proměnné `*debugger-hook*` se nastaví funkce, která se volá při vyvolání výjimky v aplikaci. Funkce, kterou vrací `make-debugger-hook`, zapíše informaci o chybě do logu a zobrazí dialogové okno, ve kterém informuje o typu chyby. Následující tři výrazy v 3 jsou označeny symbolem `#+MacOSX`, který zajišťuje, že následující výraz se vyhodnotí pouze na MacOS. Jedná se o konfiguraci, která zajistí správné rozmístění položek v menu na MacOS. Vytvořím instanci rozhraní na cocoa, které nastavím jako rozhraní aplikace a následně instanci okna předám jako hlavní rozhraní aplikace. Poslední krok funkce `main` je zobrazení okna pomocí funkce CAPI `display`.

```
1  (defun main ()
2   (let* ((environment (make-instance 'environment))
3          (*debugger-hook* (make-debugger-hook (logs environment)))
4          #+MacOSX
5          (application (make-instance 'cocoa-application-interface)))
6     #+MacOSX
7     (set-application-interface application)
8     #+MacOSX
9     (setf (cocoa-application-ide-interface application)
10          environment)
11    (display environment)))
```

Zdrojový kód 3: Zjednodušená funkce `main`

[7] Případně volitelné pro testování.

4 Uživatelská dokumentace

V následující kapitole budu psát dokumentaci pro uživatele. Na začátku představím uživatelské rozhraní a v další sekci představím základní možnosti práce s vývojovým prostředím. Dokumentace je v této práci pro první verzi vývojového prostředí. Je pravděpodobné, že se do budoucna bude rozšiřovat o nové funkce, které budou v dalších verzích programu přidány.

4.1 Uživatelské rozhraní

Uživatelské rozhraní z obrázku 6 je navrženo tak, aby v jednom okně uživatel viděl veškeré potřebné informace k práci na úkolech zadaných v kurzu. Na obrázcích 6 a 7 můžete vidět rozdíly ve verzích pro MacOS a Windows. Největší rozdílem je umístění menu, které je v rozhraní pro Windows umístěno přímo pod názvem okna aplikace, zatímco v MacOS je na to vyhrazena horní lišta na obrazovce. V následující části budu popisovat uživatelské rozhraní na MacOS z obrázku 6. Uživatelská rozhraní jsou si podobná, a proto je popis platný pro obě platformy.

Hlavní okno se skládá z několika částí:

- Název okna – zde vidíme v jakém kurzu a jaké přednášce se aktuálně nacházíme,
- Číslovač řádků – zobrazuje kolik řádků má soubor a který je aktuální,
- Editor – zde upravujeme zdrojový kód,
- Listener – vyhodnocuje výrazy zadané do promptu,
- Helper – zobrazuje informaci o lambda seznamu.

Název okna nám dává čtyři důležité informace. V hranatých závorkách se zobrazuje aktuální kurz a lekce. Na obrázku 6 je aktuální kurz „PP1“, tedy kurz Paradigmata programování 1 a „Lecture 02“ nám říká, že se nacházíme v druhé lekci. Text „Untitled Document“ nám říká, že text v editoru není asociován s žádným souborem. Symbol hvězdičky na Windows nebo tečky u tlačítka pro zavření okna na MacOS nám indikuje, že v editoru byly provedeny neuložené změny.

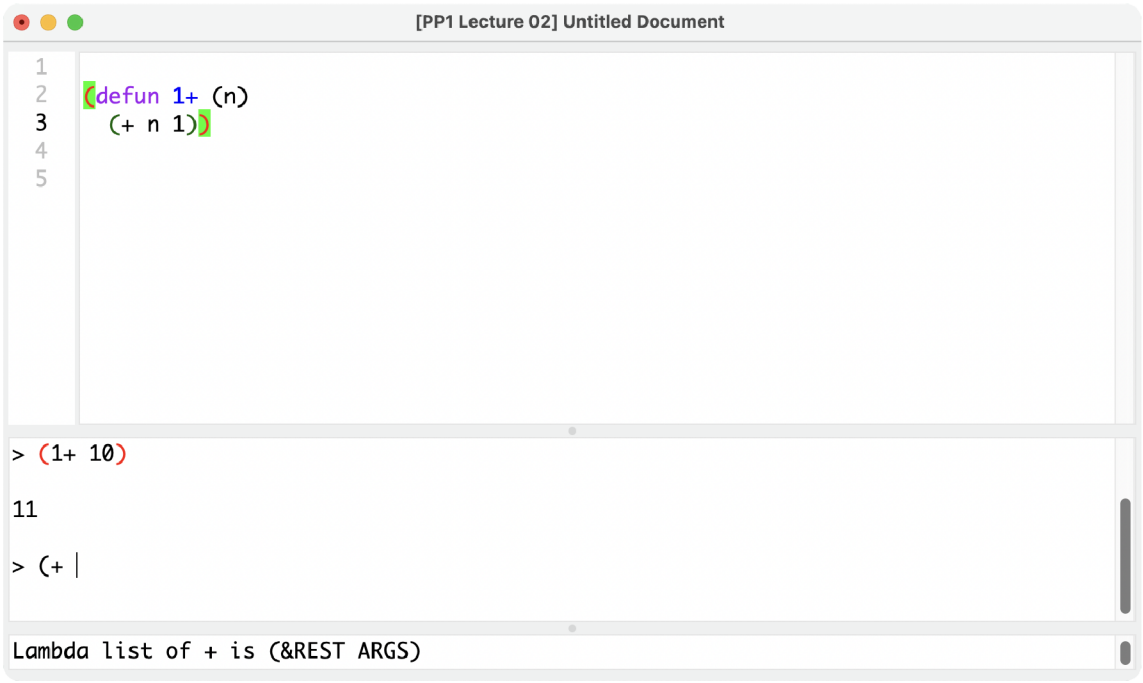
Z číslovače řádků můžeme vyčíst, že upravovaný soubor má pět řádků a kurzor se aktuálně nachází na třetím řádku. Číslovač se automaticky aktualizuje podle akcí, které uživatel provede v editoru.

V editoru můžeme upravovat zdrojový kód. Poskytuje zabarvení syntaxe, díky které lze snadno rozlišit účel symbolu. Na obrázku 6 také můžeme vidět zabarvení závorek, které k sobě patří.

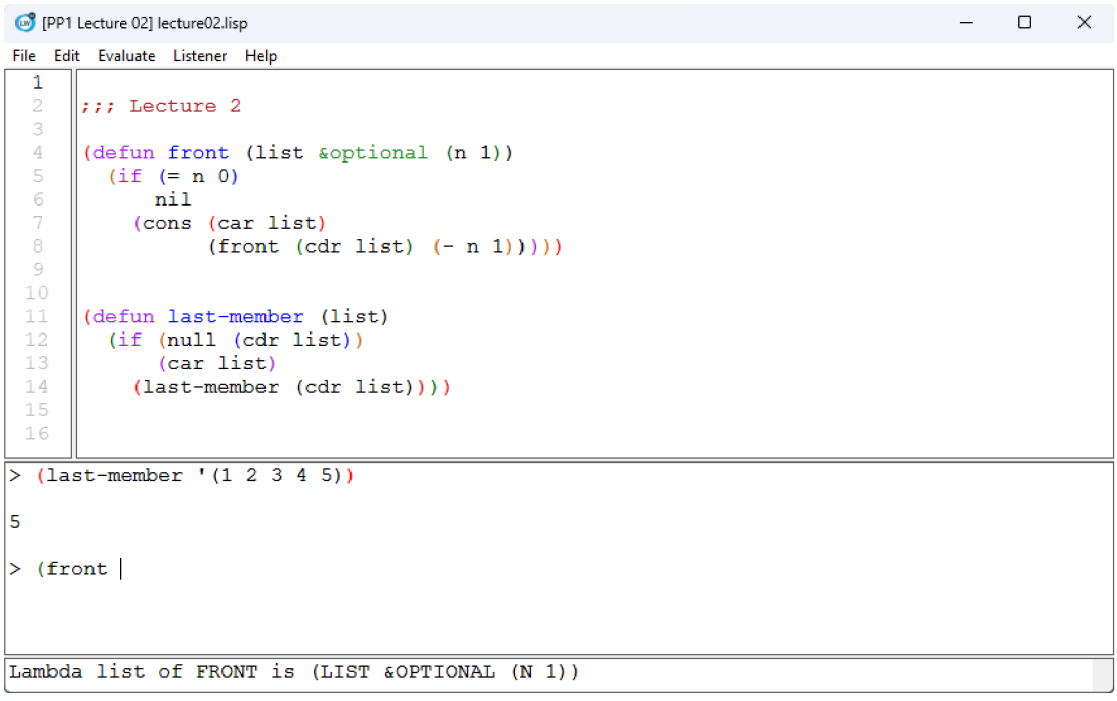
V listeneru lze vyhodnocovat výrazy jazyka a výrazy definované v editoru. Na obrázku 6 můžeme vidět vyhodnocený výraz `1+`. Listener je také zaměřené

podokno, protože se v něm aktuálně nachází kurzor. Zaměření lze přepínat mezi editorem, listenerem a helperem klávesovými zkratkami nebo kliknutím myší.

Helper aktuálně zobrazuje lambda seznam funkce +, který se zobrazil automaticky po zadání symbolu do listeneru a stisknutí klávesy „mezerník“.



Obrázek 6: Uživatelské rozhraní na MacOS



Obrázek 7: Uživatelské rozhraní na Windows

Menu

Menu se na Windows a MacOS liší, jedná se o platformě specifické detaily. Menu budu popisovat obecně a není-li uvedeno jinak, informace jsou platné pro Windows i MacOS.

Menu se skládá se z šesti sekcí:

- Polyglot (MacOS) – zde je možnost otevřít předvolby a akce specifické operačnímu systému MacOS,
- File – zde se nacházejí akce se soubory,
- Edit – zde jsou akce s editorem,
- Evaluate – zde jsou akce vyhodnocení,
- Listener – zde jsou akce listeneru,
- Help – zde může uživatel nahlásit chybu.

Polyglot je sekce menu, která je dostupná pouze na MacOS. Jsou v ní následující možnosti:

- „Settings...“ – otevře předvolby,
- „Quit Polyglot“ – ukončí aplikaci.

V sekci File najdeme následující možnosti práce se souborem:

- „New“ – otevře nový soubor ,
- „Open...“ – otevře nový soubor,
- „Open Lecture File“ – otevře soubor z aktuální lekce,
- „Save“ – uloží soubor,
- „Save as...“ – uloží soubor jako.

Následující možnosti jsou v sekci File pouze na Windows:

- „Settings...“ – otevře předvolby,
- „Exit“ – ukončí aplikaci.

V sekci Edit najdeme možnosti na základní práci s textovým souborem:

- „Cut“ – vyjmout,
- „Copy“ – kopírovat,
- „Paste“ – vložit,

- „Select All“ – vybrat vše,
- „Undo“ – zpět,
- „Set Focus to Editor“ – nastavit zaměření na editor,
- „Set Focus to Listener“ – nastavit zaměření na listener.

Sekce Evaluate obsahuje možnosti:

- „Evaluate Expression“ – vyhodnotí výraz nejblíž kurzoru,
- „Evaluate All Expressions“ – vyhodnotí všechny výrazy v editoru.

Sekce Listener obsahuje možnosti:

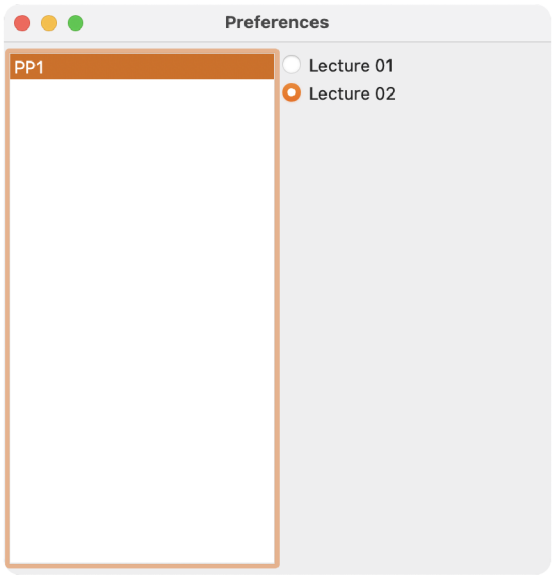
- „Restart Listener“ – restartuje listener,
- „Previous Expression“ – zobrazí v listeneru starší vyhodnocený výraz,
- „Next Expression“ – zobrazí v listeneru novější vyhodnocený výraz.

Sekce Help obsahuje možnost:

- „Report Bug“ – umožňuje podat zprávu o chybě v programu prostřednictvím emailu.

Předvolby

Předvolby se v rámci uživatelského rozhraní nacházejí v menu aplikace v sekci Polyglot (nebo File na Windows). Po kliknutí na tlačítko „Settings...“ se zobrazí okno s názvem „Preferences“, které je zobrazeno na obrázku 8. Okno „Preferences“ se dělí vertikálně na dvě části. V levé části se nachází volba kategorie nastavení. Kategorie můžou být například nastavení kurzu, nastavení vzhledu a další. V pravé části se nachází nastavení v rámci konkrétní kategorie. Na obrázku 8 vidíme možnost volby kurzu a výběr konkrétní lekce.



Obrázek 8: Předvolby na MacOS

4.2 Klávesové zkratky

V následující sekci bude výčet důležitých klávesových zkratek. Nové klávesové zkratky bude možné přidávat. Zde uvádím pouze ty nejdůležitější, které se nebudou zásadně měnit. Ve verzi na Windows jsou zkratky stejné, ale místo „Cmd“ používají „Ctrl“.

- Změna zaměření na editor – `Cmd + 1`,
- Změna zaměření na listener – `Cmd + 2`,
- Předchozí výraz v listeneru – `Cmd + šipka nahoru`,
- Další výraz v listeneru – `Cmd + šipka dolů`,
- Vyhodnocení výrazu v editoru – `Cmd + F7`,
- Vyhodnocení všech výrazů v editoru – `Cmd + F8`.

Vynechávám běžně používané zkratky, které používají i ostatní programy (například zkratku na kopírování textu).

4.3 Vyučující

Vyučující má přístup na server, na který může přidávat soubory. Server má danou strukturu souborů, podle které vývojové prostředí stahuje a načítá soubory. Pro první verzi má server následující strukturu adresáře:

- `courses/` – složka s kurzy,
- `patches/` – složka se záplatami,
- `shortcuts/` – složka s klávesovými zkratkami.

Složka `courses` obsahuje další složky:

- `global/` – složka s knihovnami,
- složky s kurzy – složky pro každý kurz.

Do těchto adresářů může vyučující přidávat nové soubory. Novou lekci přidá do složky příslušného kurzu.

4.4 Student

Při spuštění se vývojové prostředí nachází ve výchozím stavu nebo ve stavu poslední přednášky. Přednášky lze přepínat v předvolbách. Soubor přednášky, která je právě aktivní, se otevře kliknutím na položku „Open Lecture File“ v menu v sekci „File“. Po otevření souboru z přednášky můžete upravovat jeho obsah a plnit úkoly.

Pokud se v programu vyskytne chyba, vývojové prostředí zobrazí informaci o chybě s hláškou, ve které jsou uvedeny bližší informace o chybě. V sekci menu „Help“ máte možnost nahlásit chybu v aplikaci a zaslat ji vývojářům kliknutím na položku „Report Bug“.

Závěr

Cílem práce bylo vytvořit minimální vývojové prostředí vhodné pro výuku v kurzu Paradigmata programování na Katedře informatiky Přírodovědecké fakulty Univerzity Palackého v Olomouci.

V první sekci „Vývojové prostředí“ jsem představil kurz Paradigmata programování, obecně popsal jednotlivé části programu a porovnal je s vývojovým prostředím LispWorks Personal.

V sekci „Implementace“ jsem pronikl hlouběji do programu. Představil jsem použité technologie a uvedl implementační detaily jednotlivých částí programu.

V poslední sekci „Uživatelská dokumentace“ jsem představil uživatelské rozhraní a popsal používání aplikace jak z pohledu studenta, tak vyučujícího.

Cíle práce se částečně povedlo splnit. Podařilo se mi vytvořit vývojové prostředí obsahující funkční editor a listener. Vývojové prostředí také obsahuje integraci výuky v podobě přizpůsobení prostředí aktuální přednášce. V rámci práce byl naprogramován základ, na kterém bude možné do budoucna postavit kvalitní vývojové prostředí určené pro výuku v kurzu.

Části vývojového prostředí se nepodařilo z časových důvodů implementovat. Jedná se o funkcionalitu, kterou bude třeba doplnit, než Polyglot nahradí LispWorks Personal ve výuce. Mezi zásadnější nedostatky řadím chybějící implementaci knihoven, nástrojů na ladění kódu a integraci učebních textů z přednášek. Dále chybí možnost upravit vzhled aplikace (například font a barvy syntaxe) nebo verze pro Linux.

Conclusions

The aim of the thesis was to create minimalistic integrated development environment (IDE) for Programming Paradigms course at the Department of Computer Science in the Faculty of Science at Palacky University in Olomouc.

In the first section, "Development Environment", I introduced Programming Paradigms course, described parts of the program, and compared them with LispWorks Personal.

In the section "Implementation", I dived deeper into the program. I introduced the used technologies and showed implementation details of certain parts of program.

In the last section, "User Documentation", I introduced the user interface and described the use of the application from students and teachers point of view.

The aims of the thesis were partly accomplished. I managed to create integrated development environment with a working Editor and Listener. The IDE also includes an integrated tuition system which adapts the IDE to current lecture. In this work I programmed foundation for quality IDE developed for use in Programming Paradigms course.

Parts of the IDE weren't implemented due to time constraints. The missing functionality will need to be implemented in the future in order to substitute LispWorks Personal in the Programming paradigms course. Among the important missing features I'll mention the absence of implemented libraries, debugging tools, or no integration of course texts. Other missing features include the ability to edit the appearance of the application and Linux version of the program.

A Obsah elektronických dat

`text/`
: Adresář s textem práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech (textových) příloh, a všechny soubory potřebné pro bezproblémové vytvoření PDF dokumentu textu, tj. zdrojový text textu a příloh, vložené obrázky, apod.

`src/`
: Zdrojové kódy vývojového prostředí.

`delivery/`
: Spustitelné aplikace pro operační systémy Windows a MacOS.

`README.txt`
: Návod ke spuštění programu na operačních systémech Windows a MacOS.

Literatura

[1] LispWorks. *LispWorks Personal Edition*. 2021. Dostupný z: ⟨https://www.lispworks.com/downloads/index.html⟩.

[2] Carbonnelle, Pierre. *Top IDE index*. 2015. Dostupný z: ⟨https://pypl.github.io/IDE.html⟩.

[3] Seibel, Peter. *Practical Common Lisp*. Berkeley, CA: Apress, 2011. 499 s. Dostupný také z: ⟨https://gigamonkeys.com/book/⟩. ISBN 1430242906.

[4] Edmund Berkeley, L. Peter Deutsch. *The LISP Implementation for the PDP-1 Computer*. Maynard, MA: Digital Equipment Computer Users Society (DECUS), 1964. Dostupný také z: ⟨http://archive.computerhistory.org/resources/text/DEC/pdp-1/DEC.pdp_1.1964.102650371.pdf⟩.

[5] LispWorks. *Delivery User Guide*. 2021. Dostupný také z: ⟨https://www.lispworks.com/documentation/pdf/lw80/deliv-8-0.pdf⟩.

[6] McCarthy, John. History of Lisp. In Wexelblat, Richard L. (ed.). *History of Programming Languages*. New York, NY: Association for Computing Machinery, 1981, s. 784. Dostupný také z: ⟨http://jmc.stanford.edu/articles/lisp/lisp.pdf⟩. ISBN 0127450408.

[7] McCarthy, John. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*. 1960, roč. 3, č. 4, s. 184–195. Dostupný také z: ⟨http://dx.doi.org/10.1145/367177.367199⟩. ISSN 1557-7317.

[8] Graham, Paul. *ANSI Common Lisp*. Upper Saddle River, NJ: Prentice Hall, 1996. 432 s. ISBN 0133708756.

[9] LispWorks. *CAPI User Guide and Reference Manual*. 2021. Dostupný také z: ⟨https://www.lispworks.com/documentation/pdf/lw80/capi-m-8-0.pdf⟩.

[10] LispWorks. *Editor User Guide*. 2021. Dostupný také z: ⟨https://www.lispworks.com/documentation/pdf/lw80/editor-m-8-0.pdf⟩.

[11] Keene, Sonya E. *Object-oriented programming in COMMON LISP: A programmer's guide to CLOS*. Boston, MA: Addison-Wesley, 2001. 266 s. ISBN 0201175894.

[12] LispWorks. *LispWorks® User Guide and Reference Manual*. 2021. Dostupný z: ⟨https://www.lispworks.com/documentation/lw80/lw/lw.htm⟩.