VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VÝZKUMNÉ CENTRUM INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY
RESEARCH CENTRE OF INFORMATION TECHNOLOGY

EDITOR JAZYKA CODAL V PROSTŘEDÍ ECLIPSE

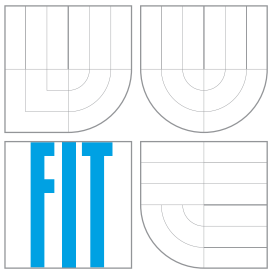
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ HYNEK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VÝZKUMNÉ CENTRUM
INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY
RESEARCH CENTRE OF INFORMATION TECHNOLOGY

EDITOR JAZYKA CODAL V PROSTŘEDÍ ECLIPSE

CODAL LANGUAGE EDITOR IN ECLIPSE FRAMEWORK

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ HYNEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZDENĚK PŘIKRYL, Ph.D.

BRNO 2013

Abstrakt

Tato diplomová práce se zabývá tvorbou editoru jazyka CodAL pro vývojové prostředí projektu Lissom, které je založené na prostředí Eclipse. Cílem této práce je analyzovat problém tvorby editorů a rozebrat doplňky existujících editorů, které zlepšují jejich uživatelskou přívětivost. V teoretické části diplomové práce je popsána tvorba parseru a následná analýza zdrojových kódů napsaných do editoru. Jsou vysvětleny syntaktické a sémantické aspekty jazyka CodAL. V praktické části je navržen nový editor jazyka CodAL a popsán postup k jeho vytvoření. Výsledkem práce je nový editor jazyka CodAL integrovaný ve vývojovém prostředí projektu Lissom.

Abstract

The Master thesis is focused on creation of an editor of CodAL language for the development toolkit of the project Lissom which is based on Eclipse framework. The goal of this thesis is to analyze the problem of editor creation and the features in existing editors which add some value to their usability. The outline of parser creation and subsequent code analysis of the source codes written into the editor is described in the theoretical part. It also explains the syntax and semantic aspects of the CodAL language. In the practical part the new CodAL language editor is designed and developed. The new CodAL language editor integrated into the development toolkit of the project Lissom is the final outcome of this thesis.

Klíčová slova

abstraktní syntaktický strom, CDT, CodAL, Codasip Studio, co-design, Eclipse, editor, gramatika, jazyk, LALR Parser Generator, lexer, Lissom, objektový model, parser, plug-in, refaktorizace, syntaxe, sémantika

Keywords

abstract syntax tree, CDT, CodAL, Codasip Studio, co-design, Eclipse, editor, grammar, language, lexer, LALR Parser Generator, Lissom, object model, parser, plug-in, refactoring, syntax, semantics

Citace

Jiří Hynek: Editor jazyka CodAL v prostředí Eclipse, diplomová práce, Brno, FIT VUT v Brně, 2013

Editor jazyka CodAL v prostředí Eclipse

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Zdeňka Přikryla Ph.D.

.....
Jiří Hynek
13. května 2013

Poděkování

Rád bych poděkoval rodině za podporu při tvorbě této práce. Rovněž děkuji vedoucímu diplomové práce – panu Ing. Zdeňku Přikrylovi Ph.D. za pomoc při analýze jazyka CodAL a tvorbě diplomové práce. Speciální poděkování patří rovněž panu Ing. Odřeji Ilčíkovi, s kterým jsem spolupracoval na vyvoji editoru jazyka CodAL.

Obsah

Kapitola 1

Úvod

Dnešní doba je charakteristická tím, že prakticky v každém zařízení můžeme najít čip, který se stará o řízení jednotky, v níž se nachází. Jedná se o běžná zařízení, jakými jsou domácí spotřebiče nebo elektronika. Například v autech jsou zastoupeny v několikanásobném počtu, přičemž každá jednotka se stará zpravidla o jednu konkrétní činnost. Systémy musí být spolehlivé a reagovat včas v reálném čase. Se vzrůstajícím počtem zařízení, která v sobě mají integrované vestavěné systémy, vzrůstají i požadavky na jejich návrh.

Důležitým faktorem je cena. Vestavěné zařízení musí být natolik dostupné, aby cena výsledného zařízení, v němž bude integrované, byla přijatelná pro dostatečný počet zákazníků, kteří vrátí investice vložené do vývoje vestavěného systému spolu se ziskem. Největší podíl investic zde hrají fixní náklady na návrh vestavěného zařízení – vymyšlení rozmístění logických členů na čipu. Marginální náklady vynakládané na fyzickou výrobu jsou oproti návrhu zanedbatelné. Odvíjejí se pouze od ceny samotného čipu a periferních zařízení, která využívá. Nahrání navrženého obvodu na FPGA čip zabere oproti jeho návrhu zlomek času. Výsledný produkt proto musí být vyroben v takovém množství, aby se náklady na návrh rozpočítaly do přijatelné ceny za jedno zařízení.

K tomu, aby toto bylo možné provést, musí být dostatečná poptávka. Roli v tomto případě hraje aktuální nabídka konkurence a čas, ve kterém firma dokáže přijít na trh s lepším zařízením, než její konkurence. Klíčová je rovněž volba fyzických komponent. Je nutné volit dlouhodobě perspektivní a dostupné architektury a komponenty, po kterých je poptávka. Často se volí méně výkonné hardwarové komponenty z důvodu toho, že k nim existuje lepší podpora a dokumentace, jsou levnější a nehrozí, že by nebyly dostupné. U mobilních systémů využívající vestavěná zařízení je rovněž důležitý příkon. Jedná se především o zařízení napájené bateriemi, u kterých je požadováno, aby vydržely v chodu co nejdelší dobu. Takové jednotky je proto důležité optimalizovat pouze na nejpodstatnější funkcionalitu.

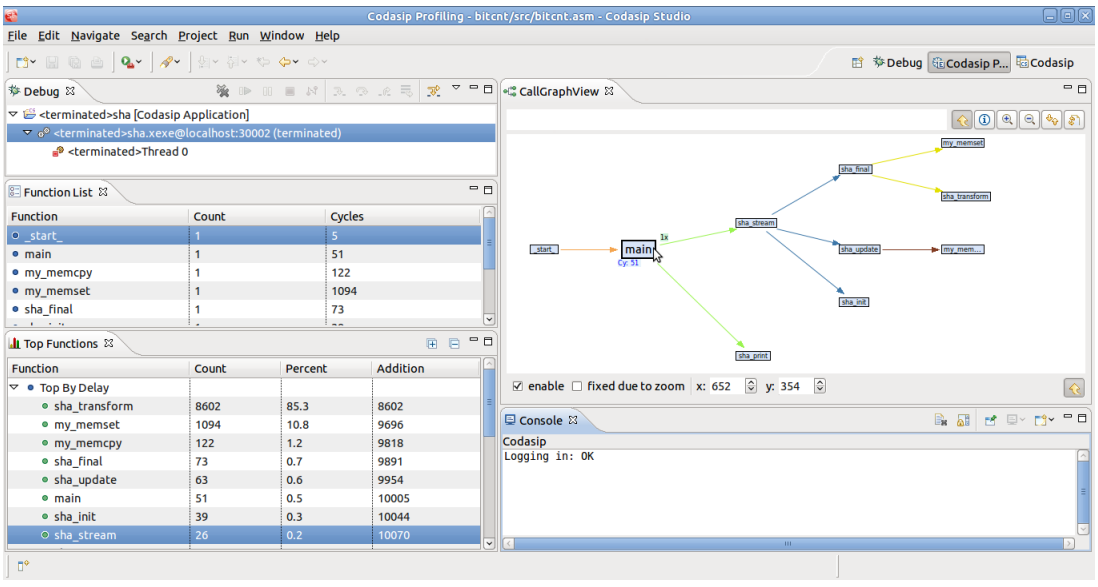
Tyto všechny faktory – cena, čas, kvalita (funkcionalita, výkon) a příkon se vzájemně ovlivňují a je důležité je zohlednit ve fázi návrhu. K tomu, aby bylo možné navrhovat vestavěná zařízení v požadovaném čase a požadované kvalitě, vzrůstají požadavky na návrhové techniky a nástroje na automatickou syntézu obvodů z abstraktních popisů. Návrhářů, kteří dobře rozumí hardwaru, není mnoho. Implementace v nízkoúrovňových jazycích (jazyk strojových instrukcí, VHDL, ...) je u složitějších systémů komplikovaná. Je proto nutné se snažit zvýšit úroveň abstrakce návrhu. V dnešní době již existují nástroje nabízející implementaci na úrovni jazyka C a následnou automatizovanou syntézu (např. nástroj Catapul C [3]). Důraz je rovněž kladen na souběžný návrh hardwaru a softwaru, díky kterému je možné optimálně rozdělit výpočetní sílu. Složité výpočty náročné na čas (například kódování videa v reálném čase) je možné zajistit speciálním obvodem nebo speciální instrukcí

aplikačně specifického systému (ASIP). Změny (například změnu codecu) lze promítnout v aplikačně specifickém systému v podobě nového firmwaru. Na rozdíl od toho je základní řízení systému vhodnější řešit pomocí softwaru, aby bylo možné v budoucnu kdykoliv aktualizovat toto řízení v závislosti na měnících se požadavcích.

1.1 Projekt Lissom

Na Fakultně informačních technologií Vysokého učení technického v Brně působí vědecká skupina Lissom [17], která se zabývá vývojem softwaru pro návrh procesorů s aplikačně specifickou instrukční sadou (ASIP) a systémů na čipu (SoC). Jednou z oblastí je vývoj specializovaného jazyka CodAL [5], s jehož pomocí je možné deklarativně popsat hardwarovou část (programový čítač, paměťové bloky, sběrnice, instrukční sadu, ...) a do stanovených bloků zapsat strukturální kód vycházející z jazyka C popisující chování definovaných instrukcí architektury. Vize projektu je nabídnout zákazníkovi komplexní nástroj umožňující navrhnout a implementovat ASIP nebo celý SoC v jazyce CodAL s následnou možností vygenerovat překladač jazyka C, assembler, linker, disassembler, debugger, simulátor a profiler. Tyto nástroje jsou určeny k implementaci a ladění softwaru pro danou platformu.

Dalším úkolem je poskytnutí integrovaného prostředí, pomocí kterého by bylo možné snadno ovládat tyto nástroje. Spolu s jazykem CodAL je proto vyvíjeno prostředí Codasip Studio [6]. Je založené na vývojovém nástroji Eclipse [7], jehož výhodou je, že se jedná o svobodný software. Je možné využívat všech existujících nástrojů prostředí Eclipse a libovolně je modifikovat pro komerční účely. Codasip Studio doplňuje Eclipse o sadu editorů a pohledů tvořící vývojové perspektivy pro souběžný návrh a vývoj hardwaru a softwaru. Pomocí tohoto nástroje je možné vytvářet, ladit a profilovat program pro ASIP.



Obrázek 1.1: Profilování pomocí nástoroje Codasip Studio

1.2 Obsah práce

Cílem diplomové práce bylo vhodně rozšířit stávající prostředí Codasip Studio. Konkrétně jsem se podílel na vývoji nového editoru jazyka CodAL.

Stávající editor je implementovaný v nástroji Xtext [23], což je generátor, který umí z definice gramatiky jazyka automatizovaně vytvořit parser spolu kompletním editorem. Výhodou je, že editor vygenerovaný pomocí nástroje Xtext obsahuje pomocné doplňky, jakými jsou zvýrazňování proměnných, nápověda, automatické doplňování zdrojového kódu apod. Nevýhodou je, že parser editoru nezohledňuje direktivy preprocesoru – konkrétně jejich odstranění překladačem jazyka CodAL. Tento nedostatek způsobuje například podvlnkování chybných konstrukcí na nesprávných řádcích. Rovněž byl problém se složitostí gramatiky jazyka CodAL. Výsledný parser byl velice komplikovaný a pomalý.

Členy vědeckého týmu Lissom bylo proto rozhodnuto, že editor bude nově založený na LALR Parser Generátoru (zkráceně LPG [16]). Vzhledem k tomu, že jazyk CodAL v sobě obsahuje bloky založené na jazyce ANSI C, bylo možné dále navazovat na projekt CDT (editory pro jazyk C a C++) [22]. Preprocesor je zde již vyřešen. Jazyk CodAL má navíc až na jednu výjimku stejnou syntaxi direktiv jako jazyk C. LPG dále pracuje s LR garamtikami narozdíl od nástroje Xtext, který pracoval s LL gramatikami (4.2.2). LR gramatiky popisují větší třídu jazyků a rovněž i gramatika překladače jazyka CodAL je v tomto formátu. Nemusí se proto provádět žádné převody mezi třídami gramatik.

Tento text jsem rozdělil na dvě části. V první (nutné pro splnění semestrální práce) rozebírám teoretické znalosti potřebné k implementaci, která je popsána v části druhé. Text je členěn do desíti kapitol. V následující kapitole je přiblížen jazyk CodAL. Je popsána jeho syntaxe a sémantika. V kapitole 3 jsou ukázány možnosti nástroje Eclipse. Jsou zde přiblíženy editory a projekt CDT. Kapitola 4 rozebírá obecný pohled na problematiku editorů, jakožto překladačů. Jsou zde popsány základní pojmy na úrovni teoretické informatiky a ukázány technologie, které umí zautomatizovat tvorbu překladačů (Xtext, LPG). V kapitole 5 je představen návrh nového editoru jazyka CodAL. Kapitoly 6, 7 a 8 se zabývají implementací výsledného editoru. Výstupem každé z těchto tří kapitol je funkční editor, kde každá následující kapitola přidává editoru novou funkcionalitu (od zvýrazňování syntaxe až po hromadné přejmenovávání proměnných). Dosažené výsledky jsou shrnuty v kapitole 9. Je zde vyhodnocena složitost zvoleného řešení. V závěru je shrnuta celá práce a jsou navrženy další rozšíření do budoucna.

Kapitola 2

Jazyk CodAL

Tato kapitola má za cíl představit základní koncept jazyka CodAL, jeho možnosti a základní popis syntaxe a sémantiky. Jednotlivé konstrukce jsou znázorněny krátkými příklady. Vycházel jsem přitom z oficiálního manuálu jazyka CodAL [1], ve kterém je možné podrobněji nastudovat danou problematiku.

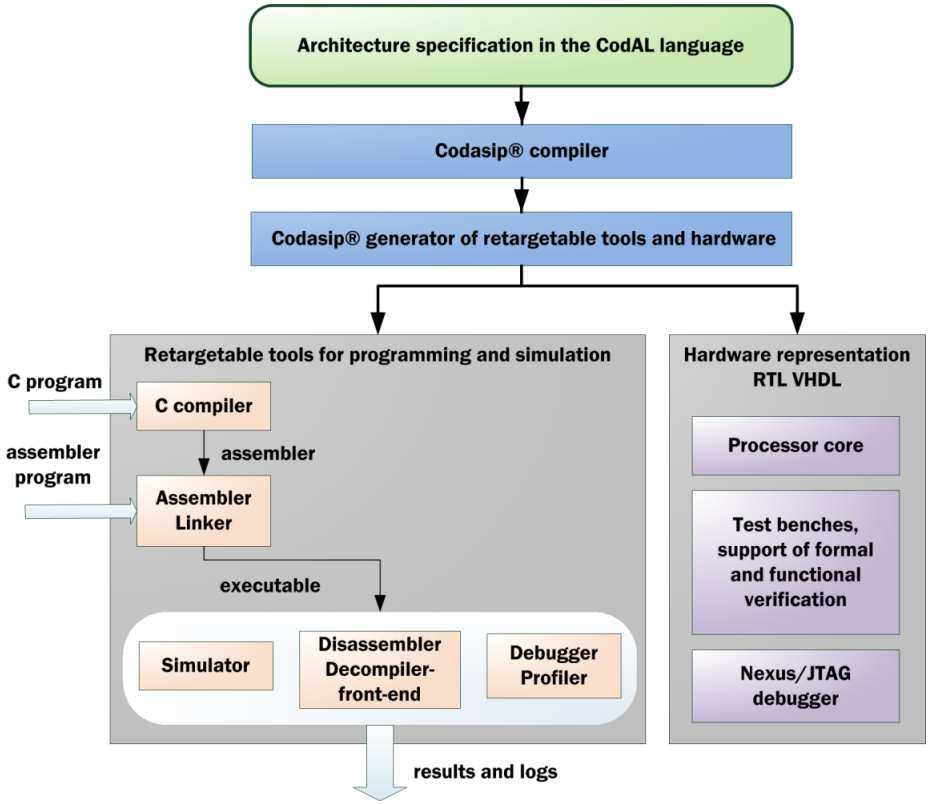
2.1 Obecný popis jazyka

Jazyk je určený k rychlému modelování procesorů s aplikačně specifickou instrukční sadou (ASIP). Instrukční sady takových procesorů obsahují konkrétní – specifické instrukce žádané daným druhem činnosti, který zastávají. Takový procesor je méně flexibilní, ovšem je optimalizovaný pro konkrétní činnost a poskytuje proto větší výkon. Je rovněž snadněji použitelný pro programátora, pokud je dotyčný dobře obeznámen s jeho možnostmi. Je jednodušší využít jednu specializovanou instrukci, jak tuto činnost modelovat pomocí posloupnosti několika elementárních instrukcí.

Jazyk byl navržen tak, aby podporoval souběžný návrh hardwaru a softwaru pro MPSoC. Uživatel nejprve popíše architekturu procesoru v jazyce CodAL. Tento zápis je následně přeložen do formátu XML. Získaný XML soubor je interní záležitost sloužící k následnému vygenerování nástrojů určených k implementaci softwaru pro danou architekturu. Jedná se o překladač jazyka C, assembler, linker, disassembler, debugger, simulátor a profiler. Nástroje jsou vygenerovány velice rychle a nabízí uživateli možnost implementovat software na úrovni jazyka C, simulovat přeložený program na dané architektuře a profilovat s cílem optimalizace.

Data získaná z kódu v jazyce CodAL nesou čtyři typy informací:

1. První kategorie popisuje model instrukční sady.
2. Druhá kategorie představuje model časování mikroarchitektury (*timing model*) určující aktivace jednotlivých stavebních bloků mikroprocesoru spolu s hierarchií dekodérů.
3. Třetí kategorie informací znázorňuje behaviorální model, který popisuje chování prvků mikroporcesoru.
4. Čtvrtá kategorie nese informaci o struktuře (*structural model*). Konkrétně se jedná o definici jednotlivých stavebních bloků mikroarchitektury a propojení mezi nimi. Elementy strukturálního modelu v sobě mohou zapouzdřovat části behaviorálního modelu.



Obrázek 2.1: Generování nástrojů pro implementaci, simulaci a ladění – zdroj [1]

2.2 Základní struktura CodAL souboru

V jazyce CodAL je možné modelovat odlišné architektury na různě složité úrovni popisu. Vždy je nutné dodržet základní strukturu souboru popsaného v jazyce CodAL nezbytnou pro překladač jazyka CodAL. Každý takový soubor musí být složen ze dvou částí – z popisu zdrojů a z popisu instrukční sady spolu s událostmi vyvolanými instrukcemi.

2.2.1 Popis zdrojů

Část popisující zdroje (*resources description*) obsahuje definici všech hardwarových komponent, z kterých se architektura skládá. Sekce může obsahovat specifikaci registrů, pamětí, sběrnic, mapování paměťového prostoru a ostatních prvků, jakými jsou například zřetězené linky (*pipeline*). Každá architektura musí obsahovat úseky kódu popisující tyto skutečnosti:

- právě jeden programový čítač (*program counter*);
- pamět pro uchování programu;
- mapování paměťového prostoru.

V následujícím příkladu je uvedena ukázka datové části základního souboru v jazyce CodAL, který jsem použil z dokumentace jazyka CodAL [1].

Příklad 2.1

```
program_counter bit[8] pc; // 8bitový programový čítač

ram bit[8] ram1 { // paměť
  .endianess = little,
  .lau = 8,
  .size = 256,
  .flags = {r,w,x}
};

bus bit[8] bus1 { // 8bitová sběrnice
  .lau = 8,
  .endianess = little
};

memorymapping defaultmap { // mapování paměťového prostoru
  bus bus1: 0..255 = ram1[7..0];
}
```

Tímto zápisem je stanoveno, že bude k dispozici 8bitový programový čítač `pc`, paměť `ram1` typu RAM obsahující 256 bloků o šířce osmi bitů a 8bitová sběrnice `bus1`. Dodatečné atributy dále stanovují, že endianita paměti `ram` je typu *little endian* a nad daty uloženými v paměti `ram1` jsou možné operace čtení, zápisu a vykonávání. Jako poslední věc je definováno mapování paměťového prostoru využívaného programovým čítačem prostřednictvím sběrnice `bus1`. 8bitová šířka představuje možnost přistoupit ke všem 256 blokům paměti `ram1`.

2.2.2 Popis instrukční sady a událostí

Druhá část obsahuje popis instrukční sady procesoru spolu s reakcemi na specifické události, jakými je například načítání instrukce. Každá architektura by měla obsahovat minimálně jednu instrukci. V následujícím výčtu jsou uvedeny základní události, jejichž reakce by měly být popsány v každém modelu procesoru.

- **Reset** reprezentuje popis reakce na událost *reset* procesoru. Může být prázdný, ale musí být obsažen.
- **Halt** popisuje reakci na událost, která je vyvolána simulátorem vždy na konci simulace. Obsahuje zpravidla ladící výpisy.
- **Main** obsahuje popis chování, které je vyvoláno s každou periodou hodin. Může zde být uveden například popis načítání instrukce, řízení přerušení apod.

Následující tři úseky kódu představují druhou část vzorového souboru jazyka CodAL.

Příklad 2.2

```
event halt { } // událost zastavení simulace

event reset { // událost reset
  semantics {
    pc = 0; // nastaví programový čítač na 0
} }
```

V prvním úseku zdrojového kódu jsou definovány reakce na dvě základní události: `halt` a `reset`. Událost `halt` má speciální semantiku, která zastaví procesor, proto není potřeba specifikovat žádnou posloupnost úkonů. Mohou zde být uvedeny například ladící výpisy. Událost `reset` v našem příkladě nastaví programový čítač na výchozí hodnotu 0. Toto chování je specifikováno v bloku `semantics`, který v sobě zaobaluje kód reprezentovaný modifikovanou verzí jazyka na ANSI C.

Příklad 2.3

```
element inop { // definice instrukce inop
  assembler { "nop" }; // textová reprezentace
  binary { 0b0000 }; // binární reprezentace
}

element ihalt { // instrukce pro zastavení simulace
  use halt; // využívá událost halt
  assembler { "halt" };
  binary { 0b1111 };
  timing { halt; }; // vyvolá tuto událost
}

set instructions = inop, ihalt; // definice instrukční sady
```

K tomu, aby bylo možné používat instrukce, je nutné provést jejich definici a provázat je s instrukční sadou. V příkladu 2.3 jsou definovány dvě instrukce – `ihalt` a `inop` a následně zaobaleny do instrukční sady `instructions`. Instrukce `inop` má definovaný jak textový (`nop`) tak binární zápis (`0b0000`). Jelikož se jedná o prázdnou instrukci, jejím voláním není vyvolána žádná událost. Na rozdíl od toho instrukce `ihalt` vyvolá událost `halt` prostřednictvím bloku `timing`. Proto je zde také uvedena konstrukce `use`, která říká, že bude použita událost `halt`.

Příklad 2.4

```
event main { // hlavní událost
  use instructions;
  start { { instructions; } };
  decoders (pc) { { instructions(bus[pc]); } };
}
```

Posledním vyžadovaným blokem je událost `main`. Účelem této události je dekódovat v každém hodinovém cyklu následující instrukci. To je popsané v blocích `start` a `decoders`. Instrukční sadu je opět nutné zahrnout pomocí konstrukce `use`.

2.3 Popis syntaxe a sémantiky

V předchozí sekci byla ukázána základní struktura souboru napsaného v jazyce CodAL. Jazyk CodAL poskytuje velké množství možností popisu hardwarových komponent a instrukčních sad MPSoC. V následující sekci bude předveden přehled syntaktických a sémantických možností jazyka CodAL, jejichž znalost je podstatnou podmínkou pro tvorbu editoru.

Pokud se podíváme na jazyk z pohledu jeho gramatiky, soubor napsaný v jazyce CodAL obsahuje několik sekcí:

- sekce pro importování jiných souborů (*include section*);
- definice programového čítače;
- definice základních zdrojů (registry, paměťové elementy, zřetězené linky, struktury a aliasy);
- mapování zdrojů do paměťového prostoru;
- deklarace entit a instance entit;
- pravidla architektury (události a elementy).

Lexikální a syntaktická stránka jazyka vychází ze stylu jazyka C. Komentáře i direktivy preprocesoru se píší stejně jako v jazyce C. Pro zanořování jsou používány složené závorky. Jazyk používá stejný typ operátorů pro porovnávání a přiřazování. Řetězce a čísla jsou zapisovány rovněž stejným způsobem jako v jazyce C. Lexikální a syntaktická podobnost obou jazyků je výhodná, jelikož jazyk C je rozšířený mezi velkým počtem programátorů. Učení jazyka CodAL je tak rychlejší.

Jazyk CodAL obsahuje velké množství bloků s odlišnou syntaxí, čemuž odpovídá i velké množství nových klíčových slov. Oproti jazyku C obsahuje jazyk CodAL 96 nových klíčových slov[1] a dvě množiny, které obsahují klíčová slova zřetězená s číslem určující bitovou šířku (např. `int32` a `vector16`). Jazyk CodAL oproti jazyku C dále obsahuje nový operátor – posloupnost dvou po sobě jdoucích teček (používané v rozsazích: `8..0`). Proměnné mohou v určitých blocích nabývat logických hodnot (`true`, `false`), hodnot v binární podobě a hodnot speciálních konstant používaných pro nastavení atributů.

Gramatika do sebe zaobaluje modifikaci gramatiky jazyka ANSI C, která pro představu zabírá šestinu kódu výsledné gramatiky jazyka CodAL. Jazyk má dostatečnou popisnou sílu, aby jím bylo možné modelovat komplexní mikroarchitektury včetně všech hardwarových bloků a instrukčních sad, které architektura obsahuje. Možnosti popisu jazyka CodAL jsou spolu s příklady ukázány v následujících odstavcích.

2.3.1 Sekce include

Jazyk CodAL umožňuje tak jako jazyk C modularizovat návrh. Je možné použít direktivu preprocesoru pro začlenění souvisejících souborů jazyka CodAL. Předtím, než je zavolán překladač jazyka CodAL, je zavolán preprocesor jazyka C. Díky tomu je možné začlenit i hlavičkové soubory jazyka C.

Jazyk nabízí dvě možnosti začlenění souborů. První možností je direktiva `#include` známá z jazyka C, která je vyhodnocena před kompilací překladačem jazyka CodAL. Druhou možností je klíčové slovo `INCLUDE`. Tento typ začlenění externího souboru je překladačem jazyka CodAL vynechán a je použit až generátory k tvorbě simulačních nástrojů.

[1] Dle gramatiky jazyka CodAL aktuální k prosinci 2012.

Konstrukce pomocí klíčového slova `INCLUDE` bude v budoucnu nahrazena speciální direktivou preprocesoru `#pragma include`, která obsahuje rovněž znak mříž (podrobněji v sekci 7.4.3). Je to z důvodu rozpoznávání těchto bloků preprocesorem nově vytvářeného editoru.

Příklad 2.5

```
#include "vexdefs.h" // pro překladač jazyka CodAL
INCLUDE "vexutils.h" // pro generátory simulátorů
```

2.3.2 Programový čítač

Programový čítač je základní registr pro uchovávání čísla právě vykonávané instrukce. V návrhu procesoru musí být právě jeden. Jak bylo ukázáno ve vzorovém souboru jazyka CodAL, je nutné definovat jeho bitovou šířku. To je provedeno klíčovým slovem `bit`.

Příklad 2.6

```
program_counter bit[8] pc;
```

2.3.3 Definice základních zdrojů

Tento blok má za úkol nést informaci o všech datových komponentách systému, jakými jsou registry, paměťové prvky, porty, signály, zřetězené linky, aliasy a jiné sdílené externí zdroje.

Registry

Registry jsou základními prvky každého procesoru. Hlavní výhodou je jejich rychlost. V těchto datových jednotkách jsou uložena všechna data, s kterými se momentálně pracuje. Je proto důležité zvolit optimální počet registrů s vhodnou bitovou šířkou.

Příklad 2.7

```
// 4 oddělené 16bitové registry
register bit[16] pc_slot_0, pc_slot_1, pc_slot_2, pc_slot_3;
register bit[32] regs[16]; // pole 16 registrů o šířce 32 bitů
```

Speciálním registrem je programový čítač, který má ovšem vlastní syntaxi rozebíranou v sekci 2.3.2.

Paměťové bloky

Jazyk CodAL nabízí možnost definovat několik druhů paměťových jednotek – `memory`, `ram`, `cache`, `bus`. Typ `memory` představuje ideální paměťový element, typ `ram` představuje paměť typu RAM[2], typ `cache` vyrovnávací paměť a typ `bus` sběrnici pro propojení paměťových prvků. Příkladem je propojení programové čítače s pamětí typu RAM, jak bylo ukázáno ve vzorovém příkladu sekce 2.2.1.

[2]Ve verzi jazyka CodAL aktuální v prosinci 2012 je konstrukce `ram` vedena pouze z důvodu zpětné kompatibility. Konstrukce `memory` byla rozšířena o syntaktické a sémantické vlastnosti konstrukce `ram`.

Paměťový prvek musí mít specifikovaný identifikátor, bitovou šířku a může mít uveden modifikátor `shared`, nebo `extern`[3]. Tyto modifikátory jsou podrobněji rozebírány v sekci 2.3.3.

Každý paměťový prvek obsahuje ve svém těle, které je obaleno složenými závorkami, atributy. Tyto atributy se vztahují ke konkrétnímu typu paměti (každý typ obsahuje vlastní sadu atributů). Především se jedná o dodatečné informace o paměťovém prvku, jakými jsou endianita, počet bloků, na které se dělí, nebo příznaky určující, jakým způsobem je možné manipulovat s daty v paměťovém bloku. Přesný výčet atributů spolu s jejich významem je možné nalézt v manuálu jazyka CodAL [1].

Příklad 2.8

```
cache bit[8] memory1_cache {
  .size = 8,
  .lau = 8,
  .flags = r, w,
  .connect = memory1,
  .endianess = little,
  .latency = 1, 1,
  .numways = 4,
  .rplpolicy = lru,
  .wapolicy = always,
  .wbpolicy = always
};
```

Výše je uveden příklad z dokumentace jazyka CodAL reprezentující osmibitovou paměť `register_cache` typu *cache*, která je napojena na paměť `memory1`. V těle jsou uvedeny jednotlivé atributy specifikující danou vyrovnávající paměť.

Porty

Porty slouží k propojení procesoru popsaného v jazyce CodAL s externími komponentami, jakým je například LCD displej. Aby bylo možné provádět simulaci těchto zařízení, je nutné zajistit jejich ovládání. To je možné provést zavedením nového komunikačního kanálu s externím zařízením – *portem*.

Příklad 2.9

```
// příklad 11bitového portu pro komunikaci s LCD displejem
port bit[11] lcd_ctrl;
```

Signály

Signály jsou speciální komponenty používané v synchronních modelech se zřetězenou linkou. Registry se v takových modelech chovají jako klopné obvody typu D. Hodnotu zapsanou do registru je možné číst vždy až s dalším taktem hodin. Aby bylo možné použít hodnotu ve stejném cyklu, kdy bylo do registru zapsáno, je nutné použít *signál*.

Příklad 2.10

```
signal bit[4] id_rA; // příklad 4bitového signálu
```

[3] Ve verzi gramatiky aktuální k prosinci 2012 je k dispozici rovněž modifikátor `msi`.

Zřetězené linky

Jazyk CodAL nabízí možnost členit výpočet procesoru do fází. Přidáním registrů na vhodná místa může být výpočet rozčleněn na několik částí a výsledky mohou být získávány s větší frekvencí. V jazyce CodAL je toho možné docílit konstrukcí `pipeline`.

Příklad 2.11

```
pipeline pipe2 { // linka se čtyřmi fázemi
  FE : l1,l2 : pr1, pr2;
  DE : l3,l4;
  EX;
  WB : : pr3,pr4;
};
```

V příkladu 2.11 použitém z dokumentace jazyka CodAL je znázorněna zřetězená linka `pipe2` se čtyřmi fázemi `FE`, `DE`, `EX`, `WB`. U jednotlivých fází je znázorněna možnost přidání registrů. Registry `l1 - l4` jsou typu *latch* a jsou používány pro uchování dat mezi fázemi. Registry `pr1 - pr4` jsou interní registry zřetězené linky.

Aliasy

Aliasy jsou konstrukce, které je možné použít v případě, že je nutné se odkazovat na konkrétní část paměti nebo registru jiným jménem. Je to vhodné zejména, když je konkrétní byte používán jako pole příznaků. Těmto konstrukcím nenáleží žádné klíčové slovo, ale jsou realizovány pomocí přiřazení.

Příklad 2.12

```
register bit[8] psw;
bit[1] zero = psw bit[3]; // reference na 4. bit registru psw.
bit[1] ov = psw bit[5]; // reference na 6. bit registru psw
```

Sdílené a externí zdroje

V případech, kdy je na čipu více procesorů, může být žádané zajistit komunikaci mezi těmito procesory. Toho lze docílit například prostřednictvím přerušení nebo sdílených paměťových zdrojů. Jazyk CodAL nabízí možnost definice takových zdrojů. Před definicí paměťového prvku je možné uvést modifikátory `shared` nebo `extern`. Modifikátor `shared` indikuje, že se jedná o deklaraci zdroje, který je využíván jinými procesory. Modifikátor `extern` komplementárně k tomu slouží k označení paměťových zdrojů, které nejsou vlastněny procesorem, který je chce používat.

Příklad 2.13

```
shared cache l2 {...}; // L2 je sdíleno s druhým jádrem
cache l1 { .connect = l2, ... }; // lokální L1 využívá L2

// externí sběrnice používaná pro komunikaci s jinými jádry v MPSoC
extern bus arm5.sysbus  ... ;
```

V příkladu 7.5 je ukázán popis jádra A, které sdílí vyrovnávací paměť `l2` s jádrem B a obsahuje lokální vyrovnávací paměť `l1`, která je spojená s lokální `l2`. Dále jádro A obsahuje sběrnici, která je používána pro komunikaci s ostatními prvky v MPSoC.

2.3.4 Mapování paměťového prostoru

Podstatnou činností při modelování architektury je mapování zdrojů do paměťového prostoru tak, aby k nim bylo možné snadno přistupovat.

Příklad 2.14

```
memorymapping {
  bus sysbus: 0..1023 = mem[12..0];
  bus sysbus: 1025..5117 = bmem[12..10] [9..0];
};
```

Na příkladu 2.14 jsou znázorněny 2 druhy mapování zdrojů. První možností je mapovat celou paměť (pole paměťových buněk). Druhou, složitější možností je mapovat paměť rozdělenou do bloků (*banků*). K takové paměti se pak přistupuje přes adresu, která je poté rozdělena na dvě (adresu banku a adresu prvku).

2.3.5 Entity

Jazyk CodAL umožňuje implementaci VLIW (*very long instruction word*) mikroarchitektur. Jedná se o architekturu, jejíž instrukce jsou složeny z několika dílčích *subinstrukcí*. Každá z těchto subinstrukcí je vykonávána zřetězenou linkou (*slotem*) využívající sadu zdrojů (*cluster*). Tyto zdroje (*clustery*) jsou v jazyce CodAL implementovány zapouzdřeně ve vlastních entitách. Při simulaci je pak primárně vyhledáván zdroj v přiřazeném clusteru daného slotu a až v případě, že není pod daným jménem nalezen, použije se zdroj ze zdrojů globálních.

Příklad 2.15

```
entity t_regs_a (out_reg) {
  register bit[16] results[8];
};
```

V příkladu 2.15 je znázorněna entita `t_regs_a`, který obsahuje sadu osmi registrů o bitové šířce 16. Obsahuje jeden parametr `out_reg`, který je použit při tvorbě instance dané entity znázorněné v dalším příkladě.

Příklad 2.16

```
register bit[32] reg1;
t_regs_a regs_a (reg1);
```

Jako parametr je uveden 32bitový registr `reg1`. Instance `regs_a` pak bude obsahovat osm 16bitových registrů a referenci na 8bitový registr `reg1`. Tímto způsobem je možné vytvářet několik těchto instancí. K prvkům instance je možné přistupovat přes tečkovou notaci: `regs_a.results[1]`.

2.3.6 Pravidla architektury

Pravidla slouží pro definici chování architektury. K tomu jsou v jazyce CodAL vymezeny bloky označené klíčovými slovy `element` pro instrukce a `event` pro události. Skupiny instrukcí jsou pak mapovány do instrukční sady označované klíčovým slovem `set`. Příklad použití těchto bloků byl ilustrován v sekci 2.2.2.

Základní formát pravidel `event` a `element` je ve zjednodušené formě následující:

Příklad 2.17

```
<RULE> <ID> {
    use ... ;
    assembler { ... };
    binary { ... };
    semantics { ... };
    return { ... };
    timing { ... };
    start { ... };
    decoders { ... };
    bundle { ... };
    debundle { ... };
}
```

V příkladě 2.17 je znázorněné pravidlo `<RULE>`, které může být typu `event` nebo `element`. Je pojmenované identifikátorem `<ID>`. Pravidlo obsahuje tělo, v němž jsou bloky, které specifikují chování pravidel. Tyto bloky nejsou povinné. Jejich význam je přiblížen v následujících odstavcích.

Sekce use

V případě, že chceme lokálně použít událost, instrukci nebo instrukční sadu v jiném bloku `event` nebo `element`, je nutné použít konstrukci `use`. Můžeme přitom vytvářet několik instancí pomocí aliasů.

Příklad 2.18

```
element reg { ... }

element add {
  use reg as reg_src, reg_dst;
  ...
}
```

V příkladu 2.18 je ilustrována tvorba dvou instancí instrukce `reg`.

Sekce assembler a binary

Tyto konstrukce slouží pro označení instrukce (nebo její části) její textovou a binární podobou. Informace jsou využity profilerem a při generování assembleru a disassembleru. Nejjednodušší varianta zápisu je k vidění na příkladu 2.19.

Příklad 2.19

```
element ihalt {
  assembler { "halt" };
  binary { 0b1111 };
}
```

Jazyk CodAL nabízí velké možnosti v popisu reprezentace instrukce. Více informací je k dispozici v dokumentaci tohoto jazyka [1].

Sekce semantics

Do těla `semantics` je implementováno chování události nebo instrukce. Hlavním účelem této sekce je implementovat přesun dat mezi zdroji, případně jejich modifikaci.

Pro popis je použita modifikovaná verze jazyka ANSI C, která obsahuje pouze některé konstrukce. Nepovoluje například ukazatele, struktury, příkaz `goto`, deklaraci a inicializaci proměnné v jednom příkazu. Funkce je možné používat stejně jako v ANSI C s tím rozdílem, že parametry je možné předávat pouze hodnotou. Funkce vrací pouze standardní datové typy.

Upravená varianta jazyka C nabízí možnost deklarovat celočíselné proměnné na přesný počet bitů. Umožňuje to pomocí klíčových slov `int`N, kde N je celé číslo. Dodatečná sémantická kontrola pak povoluje pouze některá celá čísla. Pro výpis je možné používat metodu `printf` jako v jazyce ANSI C.

Sekce return

V tomto bloku je uveden jeden výraz, který reprezentuje návratovou hodnotu. Konstrukce je používaná v případě vytváření instancí bloků `element` a přístupu k jejich hodnotám.

Příklad 2.20

```
element inst_add { return { INST_ADD; }; }
element inst_sub { return { INST_SUB; }; }
set inst_arithm = inst_add, inst_sub;

element exec_arithm {
  use inst_arithm as operation;
  semantics {
    switch (operation) {
      case INST_ADD: ... // udělej operaci sčítání
      case INST_SUB: ... // udělej operaci odčítání
} } }
```

V příkladu 2.20 je znázorněna instrukční sada `inst_arithm` s instrukcemi `inst_add` a `inst_sub`, jejichž návratové hodnoty jsou konstanty `INST_ADD`, respektive `INST_SUB`. V popisu chování instrukce `exec_artihm` pak můžeme testováním návratové hodnoty zjišťovat, která instrukce je aktivní a provádět v závislosti na tom patřičné operace.

Sekce timing

V sekci `timing` jsou uvedeny všechny události, které mají být uskutečněny v případě vyvolání dané instrukce nebo události. Tato konstrukce je důležitá v případě, kdy požadujeme synchronní model procesoru. V sekci jsou povoleny příkazy `delay`, které zajišťují zpoždění za účelem synchronizace.

Příklad 2.21

```
switch (reg_arch_size) {
  case 16:
    delay 1; writeback16; break;
  case 32:
    delay 1; writeback32; break;
}
```

V příkladu 2.21 je ukázka těla bloku `timing`. V závislosti na velikosti registru, do kterého je plánované zapisovat, je s prodlením jednoho hodinového cyklu vyvolána příslušná událost.

Sekce start

Tato sekce slouží především pro generátor assembleru a disassembleru. V této sekci je uvedena konstrukce, která reprezentuje startovní neterminál pro generování gramatiky jazyka strojových instrukcí. Může zde být uvedený například kořenový prvek nesoucí instrukční sadu (deklarovaný pomocí `set`).

Sekce decoders

Konstrukce `decoders` musí být součástí stejného elementu jako sekce `start` a tento element musí být unikátní. Je zde popsáno, kde jsou instrukce uloženy, jejich adresy, jakým způsobem adresovat zdroje a které dekodéry mají být aktivovány.

Sekce bundle a debundle

V sekcích `bundle` a `debundle` je uveden kód pro kódování a dekódování VLIW instrukcí. Používá se při popisu na úrovni architektur a tyto informace jsou pak používány generátory assembleru a disassembleru.

Sekce přiřazení

Při definici události je možné do deklarační části volitelně uvést sekci přiřazení (*assignment section*). Používá se v případě, že danou událost chceme asociovat s konkrétní fází zřetězené linky.

Příklad 2.22

```
pipeline pipe { FE: ; DC: ; EX: ; WB: ; };
event address1 : pipe.FE { ... }
event read1 : pipe.FE { ... }
event decode1 : pipe.DC { ... }
event add1 : pipe.EX { ... }
```

2.4 Integrace s vývojovým prostředím

Existují dvě možnosti, jak z popisu architektury v jazyce CodAL vygenerovat nástroje pro implementaci softwaru pro danou architekturu. První možností je použít nástroj na úrovni příkazové řádky (*CLI - Command Line Interface*). Druhou možností je použít grafické vývojové prostředí (*Codasip Studio*). Cílem této práce je vytvořit editor pro grafické vývojové prostředí, proto budu v této kapitole rozebírat pouze druhou možnost.

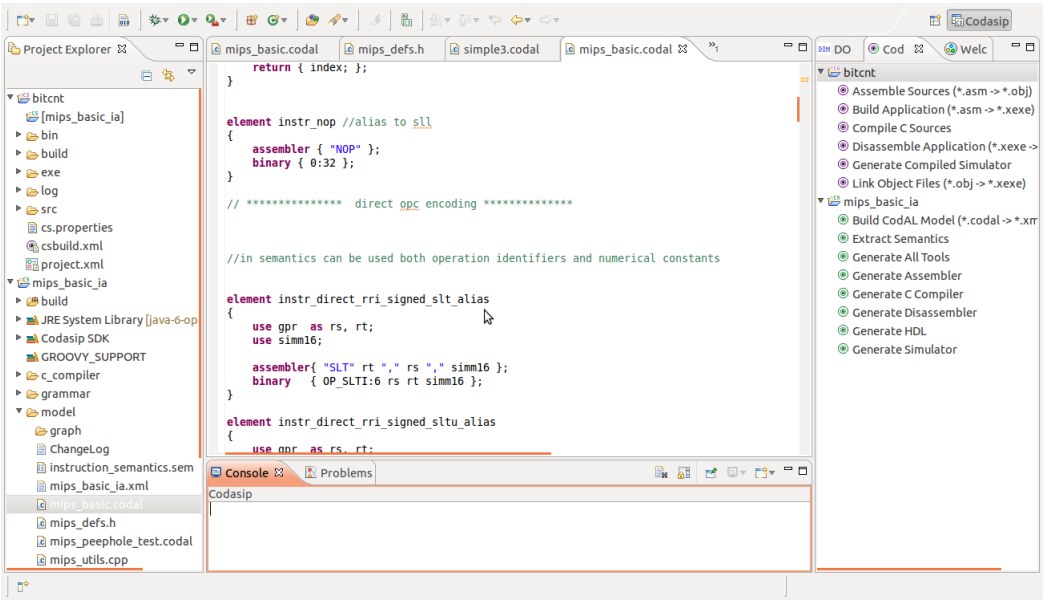
Vývojové prostředí Codasip Studio představuje v rámci celého nástroje prezenční vrstvu. Pomocí této vrstvy uživatelé komunikují se střední vrstvou (*middleware*). Jedná se například o příkazy generování simulačních nástrojů, spouštění a ukončování simulace. Střední vrstva provádí tyto příkazy a oznamuje jejich výsledky zpět vývojovému prostředí (prezenční vrstvě). Děje se tak prostřednictvím TCP/IP protokolu.

Střední vrstva může využívat jednoho nebo více vzdálených počítačů, na kterých jsou nainstalovány simulátory (*cloud simulation*). Tímto způsobem je možné provádět rychlé simulace víceprocesorových architektur.

2.4.1 Uživatelské rozhraní Codasip Studia

Vývojové prostředí nabízí tři sady nástrojů – perspektivu pro vývoj, pro debuggování a profilování.

- **Základní perspektiva pro vývoj** je složena z editorů pro modelování hardwaru nebo softwaru. Perspektiva obsahuje mimo jiné editor jazyka CodAL. K dispozici jsou dále pohledy pro orientaci ve zdrojových kódech, generování simulačních nástrojů a konzole.



Obrázek 2.2: Ukázka vývojové perspektivy

- **Debuggovací perspektiva** umožňuje uživateli zastavovat simulace na označených místech a sledovat hodnoty jednotlivých registrů či hodnot v paměti. Je možné krokování kódu jako v klasických debuggerech.

- **Profilovací perspektiva** je složena ze skupiny pohledů, které zobrazují informace získané ze simulace softwaru na zvolené architektuře. Je zde možné podrobně sledovat například využití vyrovnávacích pamětí, počty cyklů. Tuto perspektivu jsem rozšiřoval v rámci bakalářské práce o pohled demonstrující tok programu (*call graph*).

Výsledkem této práce by měl být nový editor jazyka CodAL, který bude součástí základní perspektivy pro souběžný vývoj hardwaru a softwaru.

2.4.2 Tvorba projektů

Pro tvorbu projektů jsou k dispozici průvodci, ve kterých uživatel specifikuje všechny nutné parametry vyžadované pro tvorbu projektu.

1. Prvním úkolem je vytvoření **hardwarového projektu**. V něm uživatel zadá jméno souboru typu CodAL, ve kterém bude popsaný model. V dodatečném formuláři je pak možné zadat parametry pro simulaci a profilování.

2. Jakmile je vytvořený hardwarový projekt, je možné vytvořit **softwarový projekt**. Opět se to provede pomocí průvodce, v kterém je nutné asociovat softwarový projekt s hardwarovým modelem.

Po vytvoření projektů je možné souběžně navrhovat hardwarový model a program k němu asociovaný. Projekty je možné rovněž importovat.

Kapitola 3

Vývojové prostředí Eclipse

V minulé kapitole byl představen jazyk CodAL a jeho integrace ve vývojovém prostředí Codasip Studio. Tento nástroj je založen na vývojovém prostředí Eclipse. Ve své podstatě se jedná o sadu modulů, které rozšiřují základní verzi prostředí Eclipse. Za dobu existence Eclipse bylo implementováno mnoho takových rozšíření. Stejně tak jako Codasip Studio mají tyto rozšíření za cíl vytvořit nástroj pro konkrétní skupinu vývojářů. Prostředí Eclipse bylo pro tento účel navrženo a v sadě svých modulů obsahuje nástroj pro tvorbu nových rozšíření.

V této kapitole proto popíši základní logiku modularizace vývojového prostředí Eclipse. Zaměřím se přitom na editory, které jsou podstatou této diplomové práce. Zdrojem mi byla kniha Eclipse, Building Commercial-Quality Plugins [4], oficiální dokumentace nástroje Eclipse [20] a znalosti získané ze své předchozí práce [12].

3.1 Historie a přítomnost

Eclipse je momentálně značně rozšířené open source vývojové prostředí. Bylo vyvinuté firmou IBM jako náhrada za vývojové prostředí VisualAge. Jednalo se o kvalitní vývojové prostředí, které bylo implementované v jazyce SmallTalk. Se vzrůstající popularitou jazyka Java se firma IBM rozhodla vyvinout prostředí založené na tomto jazyce. Vývojáři mohli využít osvědčených návrhů použitých v prostředí VisualAge vzhledem k tomu, že jazyk SmallTalk je rovněž objektově orientovaný jazyk. Na přelomu tohoto tisíciletí proto vzniklo vývojové prostředí Eclipse implementované v jazyce Java.

Vývojové prostředí Eclipse je značně modularizované. Jeho hlavní výhodou je snadná rozšiřitelnost v podobě *plug-inů*. Nabízí podporu pro jejich vývoj v podobě PDE (*Plug-in Development Enviroment*). PDE je volitelnou součástí prostředí Eclipse. Poskytuje nástroje pro tvorbu, testování a debugování plug-inů. Díky plug-inům se z Eclipse stává nástroj, který se dá dobře přizpůsobit požadavkům vývojáře či vývojového týmu. Nabízí velký výběr volitelných nastavení, které je možné migrovat z jednoho prostředí na druhé. Je multiplatformní, není tedy problém ho provozovat na celé řadě operačních systémů. Zdrojové kódy vývojového prostředí jsou veřejné. Nástroj je aktuálně šířen pod licencí EPL (*Eclipse Public Licence*) a projekt spravován nevýdělečnou organizací Eclipse Foundation. Projekt je podporován velkými firmami, jakými jsou například IBM, Oracle nebo Google. Vývojové prostředí Eclipse je využíváno značným počtem nástrojů pro vývoj specializovaného softwaru (například Android SDK pro vývoj aplikací pro mobilní platformu Android). Díky

rozšířenosti mezi uživateli a podpoře ze strany mnoha firem se jedná o výhledově perspektivní nástroj[1].

3.2 Architektura Eclipse

Eclipse není monolitický program. Je tvořen malým jádrem, které slouží jako zavaděč pluginů. Konkrétně se jedná o projekt Equinox, který vznikl ve verzi 3 vývojového prostředí Eclipse. Projekt v sobě nese implementaci standardu OSGi verze R4, která je hojně využívaná například aplikačními servery jakými jsou GlashFish nebo JBoss.

OSGi je standardizovaný framework, který specifikuje systém modularizace v programovacím jazyce Java. V takovém systému je možné za jeho běhu instalovat, aktualizovat, spouštět, vypínat a odinstalovávat přídavné moduly systému. Přitom není nutné systém restartovat. Každý modul (*bundle*) je složen z archívů typu `jar`, které obsahují třídy poskytující funkcionalitu modulu. Mimo jiné musí každý modul obsahovat dodatečné konfigurační soubory, které tuto funkcionalitu deklarují. Jednotlivé moduly mohou záviset na jiných. Vzniká tím tedy graf závislostí, která je sestavena pomocí informací uvedených v konfiguračních souborech. Díky tomu je možné řídit životní cyklus jednotlivých modulů a zajišťovat jejich správnou inicializaci.

Ve vývojovém prostředí Eclipse jsou moduly reprezentovány prostřednictvím *plug-inů*. Každý plug-in musí mít specifickou strukturu, aby byl vývojovým prostředím Eclipse rozpoznán jako nový modul. Musí proto obsahovat následující konfigurační soubory:

- **META-INF/MANIFEST.MF:** Tento soubor předepisuje základní atributy modulu, které jsou nutné pro běh plug-inu. Jedná se o identifikátor, verzi a jméno. V souboru je dále uvedena cesta k hlavní třídě plug-inu nazývaná `Activator`. Důležité jsou rovněž informace o závislostech na jiných modulech. Specifikovat je možné konkrétní verze závislých modulů, které spouštěný modul vyžaduje. Uvést je možné i specifické verze, které jsou kompatibilní se spouštěným modulem.

- **plugin.xml:** Tento soubor se použije při zjišťování závislostí. V souboru `MANIFEST.MF` se nejprve zjistí, jaké moduly jsou vyžadovány. Soubor `plugin.xml` dále zjistí, jakou funkcionalitu (rozšíření – *extension*) konkrétního externího modulu daný modul využívá. Komplementárně k tomu jsou v tomto souboru specifikovány rozšíření modulu, které modul poskytuje pro své okolí (*extension points*). Týká se to například specifikace pohledů, editorů, tlačítek doplňujících menu apod.

- **build.properties:** V tomto souboru jsou uvedeny informace potřebné k překladu aplikace. Jsou zde specifikovány například informace o tom, které soubory budou zahrnuty ve výsledném archivu typu `jar` daného modulu.

O tvorbu konfiguračních souborů se stará automaticky PDE vývojového prostředí Eclipse. K dispozici jsou editory formulářového typu pro snadnou editaci atributů. Soubor minimální množiny modulů (*plug-inů*) nutných k sestavení samotné klientské aplikace je v Eclipse nazýván *Eclipse Rich Client Platform* (RCP).

[1] Pravidelně každý rok v červenci vychází nová verze. Momentálně je aktuální verze 4.2 Juno vydaná 27. 7. 2012.

3.3 Uživatelské rozhraní

Vzhled vývojového prostředí Eclipse je založen na grafické knihovně SWT (*Standard Widget Toolkit*). Tato knihovna byla navržena firmou IBM přímo pro vývojové prostředí Eclipse. Vývojáři knihovny se zaměřili především na nevýhody, které obsahují knihovny AWT (*Abstract Windowing Toolkit*) a JFC (*Java Foundation Classes*), více známá jako Swing.

Knihovna AWT byla vyvinuta jako první. Obsahuje pouze základní grafické komponenty (tlačítka, jednoduchá textová pole, atd.), což je pro návrh složitějších programů nedostačující. SWT proti tomu obsahuje širokou paletu grafických prvků. Pro implementaci složitějších komponent, jakými jsou například dialogy, průvodci, tabulky a seznamy, byla navržena knihovna JFace. Tento grafický toolkit má za cíl usnadnit tvorbu často implementovaných komponent, jejichž programování je zdlouhavé. Knihovna JFace vychází z knihovny SWT a snaží se ji doplnit.

Druhou knihovnou je Swing. Byla vyvinuta firmou Sun jako reakce na malé možnosti knihovny AWT. Tento toolkit již obsahuje velké množství nástrojů určených k vytváření kvalitních programů v jazyce Java. Jeho nevýhodou je jeho malá rychlost způsobená vlastní implementací vzhledu (označovaného pojmem *look and feel*). Program implementovaný pomocí knihovny Swing má na všech platformách stejný vzhled. Vývojáři Eclipse se snažili o to, aby program Eclipse vzhledově zapadl mezi ostatní programy instalované na daném operačním systému. Díky tomu, že využívá systémové knihovny operačního systému, je zajištěna i jeho větší rychlost.

3.3.1 Pracovní prostředí

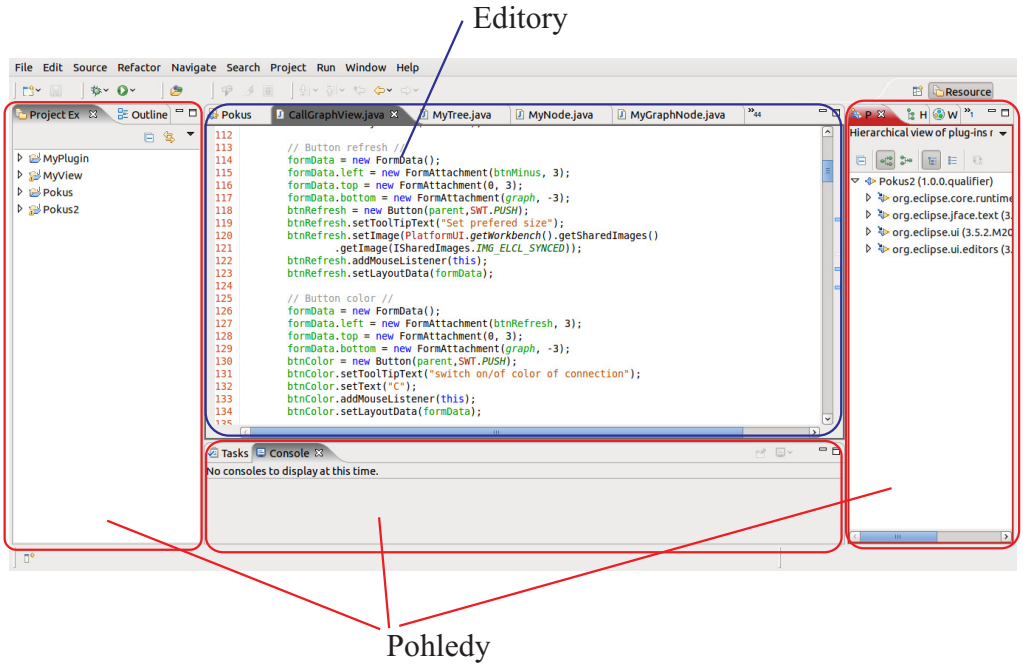
Pracovní prostředí nástroje Eclipse se nazývá *Workbench*. Jeho základem jsou perspektivy doplněné o menu tlačítka a dialogy. Každá perspektiva je určena pro konkrétní druh činnosti. Například vývojové prostředí Codasip Studio nabízí perspektivy pro vývoj, debuggování a profilování, jak bylo popsáno v sekci 2.4.1. Základní verze vývojového prostředí Eclipse obsahuje například perspektivu pro vývoj plug-inů (*Plug-in Development Perspective*), pro vývoj Java aplikací (*Java Perspective*) nebo pro verzování (*CVS Repository Exploring Perspective*). Každá perspektiva je složena z pohledů a editorů, které zastávají konkrétní činnost perspektivy. Je možné je používat i samostatně nebo zobrazit spolu s jinou perspektivou.

3.3.2 Editory

Editor je okno určené k editaci zdrojových kódů. Může mít několik podob. Základní variantou je klasický textový editor, do kterého se přímo píše zdrojový kód. Rozšířenou variantou je editor formulářového typu. Ten druh editoru slouží k editaci konkrétních částí kódu (například atributů konfiguračních souborů vytvářených plug-inů)[2]. Editory jsou zpravidla umístěny ve středu pracovního prostředí. Je možné mít současně otevřeno více editorů a přepínat mezi nimi.

Editory bývají rozšiřovány o dodatečnou funkcionalitu, která průběžně provádí analýzu zdrojového kódu s cílem usnadnit programátorovi práci. Týká se to například zvýrazňování syntaxe, nápovědy slov, automatických oprav nebo hromadného přejmenování proměnných.

[2] V dnešní době jsou vyvíjeny také grafické editory, pomocí kterých je možné například z modelů tříd vygenerovat zdrojový kód. V Eclipse se touto oblastí zabývá projekt GMF – *Graphical Modeling Framework*.



Obrázek 3.1: Pracovní prostředí nástroje Eclipse

Vize editoru jazyka CodAL je taková, že by měl v budoucnu obsahovat nástroje pro analýzu kódu, které jsou běžné v kvalitních editorech. V následujících odstavcích je uveden přehled takových nástrojů. Vycházel jsem z funkcionality známých editorů jakým je například editor jazyka Java prostředí Eclipse nebo projekt CDT určený pro jazyk C a C++.

Zvýrazňování zdrojového kódu

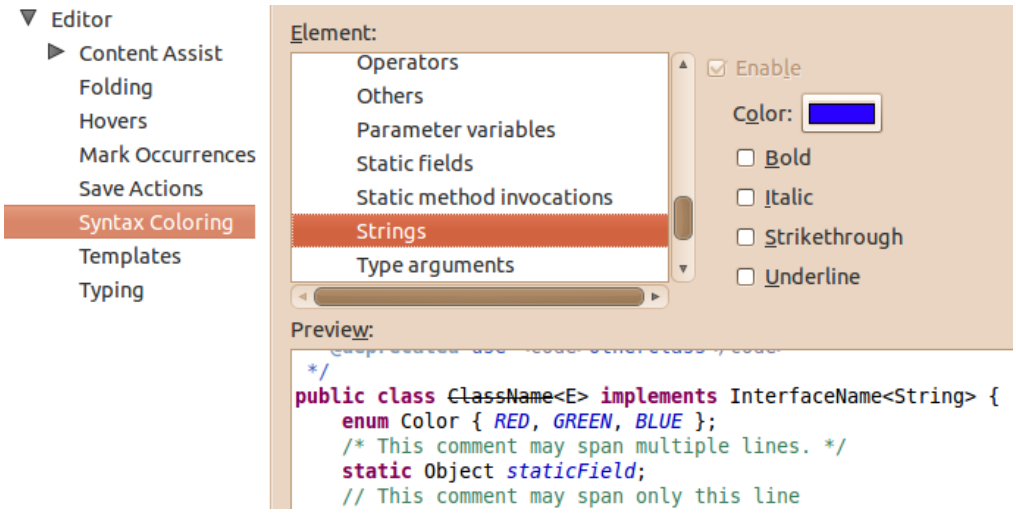
Smyslem tohoto nástroje je rozlišovat (barvami nebo stylem písma) jednotlivé druhy slov – *lexémy* (například klíčová slova, čísla, řetězce, proměnné). Rovněž by měly být odlišeny bloky, které jsou ve zdrojovém kódu dodatečné a nepoužijí se pro překlad (například komentáře). Výhodou je, pokud si uživatel může nadefinovat vlastní profil, v kterém určí styly pro jednotlivé typy lexémů. V takovém zdrojovém kódu se programátor může rychleji orientovat a snadněji objeví například překlepy klíčových slov.

Jedná se o základní nástroj každého editoru. Primárně je kód zpracováván na úrovni lexikální analýzy. Dodatečně mohou být zohledněny direktivy preprocesoru jako je tomu například u jazyka C. V takové situaci je nutné analyzovat například direktivy `#ifdef`, v kterých se testuje definice konstant. Jedná se pak o dodatečnou sémantickou kontrolu.

Příklad 3.1

```
#ifdef PRINT_TRACE // není definováno
  fprintf(stderr, "%6d: pc: %d, ins 0x%8x\n",
          (int)cycle_cnt, (int)pc, (int)fetched_instr);
#endif // proto je tento blok zobrazen jako komentář
```

Příkladem je testovací výpis znázorněný v příkladu 3.1. V případě, kdy konstanta `PRINT_TRACE` není definovaná, by měl být správně celý blok zobrazený jako komentář nebo nějakým způsobem odlišen od zbývajícího zdrojového kódu. Toto byl jeden z nedostatků bývalého editoru jazyka CodAL.



Obrázek 3.2: Nastavení zvýrazňování zdrojových kódů

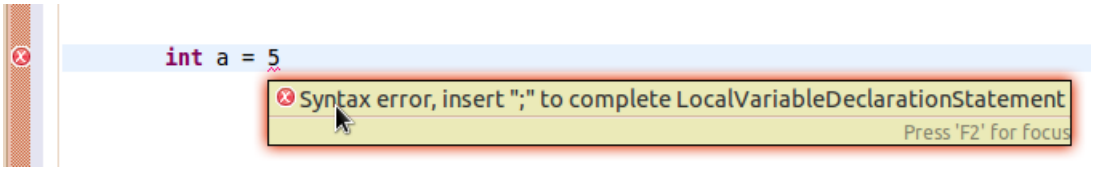
Formátování struktury kódu

Dalším nástrojem, který značně zpřehledňuje kód, je formátování zdrojového kódu (*code formatting*). Uživatel si může nadefinovat odsazení vnořených komponent a styl zalamování dlouhého textu. Tato činnost se pak děje dynamicky za běhu při tom, jak uživatel píše. Dále může být vyvolána také manuálně nad označeným zdrojovým kódem[3].

Pro odsazování vnořených bloků je nutné znát počty závorek (nebo jiných znaků pro zanořování). Tato informace je často odvozována z výsledku syntaktické analýzy.

Kontrola správnosti zdrojového kódu

Nejpodstatnějším doplňkem kvalitních editorů je analyzátor korektnosti zdrojového kódu. Smyslem analýzy je rozhodnutí, zda zdrojový kód patří do zvoleného jazyka. Pokud tuto



Obrázek 3.3: Upozornění na chybějící středník

podmínku zdrojový kód nesplňuje, je podstatné ukázat programátorovi, které úseky zdrojového kódu nesplňují požadovaná kritéria. Analýza přitom musí být dostatečně rychlá

[3]Například stiskem kláves `CTRL+SHIFT+F` ve vývojovém prostředí Eclipse.

a musí být schopna se zotavit při nálezu chyby (*domyslet* si správné řešení) tak, aby mohla být dokončena analýza pro zbytek zdrojového kódu. Přijatelným chováním je například podvlnkování chybných úseků kódu.

Zdrojový kód může být otestován třemi analýzami, které na sebe navazují:

1. Nejjednodušším způsobem je test na úrovni **lexikální** analýzy. Uživateli jsou zvýrazněny všechny lexémy, které jsou chybné a nepatří do daného jazyka. Na této úrovni jsou zaznamenány především překlepy. Uživatel například napíše neexistující operátor nebo pojmenuje proměnnou s číslicí na začátku.

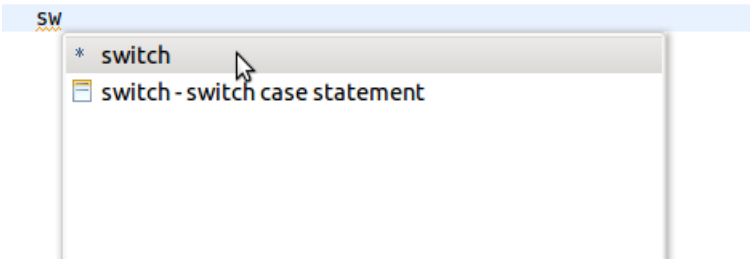
2. Pokročilejším nástrojem je kontrola **syntaktické** korektnosti zdrojového kódu. V takovém případě je testováno, zda jsou lexémy ve správném pořadí (odpovídají gramatice jazyka). Tato kontrola umí odhalit například nekonzistenci závorek.

3. Třetí typ analýzy testuje **sémantickou** správnost zdrojového kódu. Jedná se o nejvíce abstraktní analýzu, jejímž cílem je zkoumat význam zdrojového kódu. V této analýze jsou odhalena například špatná přetypovaní proměnných nebo pokusy inicializovat nedeklarované proměnné.

Kvalitní editory mají analýzu řešenu do třetí úrovně, o což bude snaha i v editoru jazyka CodAL. Problematika analýzy zdrojového kódu je podrobněji rozebrána v následující kapitole.

Content assist

Vhodným způsobem, jak předcházet chybám programátora, je předvídat kód, který programátor zamýšlí napsat. Při psaní jsou uživateli prostřednictvím vyskakovacího okna zobrazeného pod kurzorem znázorněny možnosti doplňující text, který uživatel napsal[4]. V nejjednodušší variantě se jedná o doplňování klíčových slov. Složitější verze podporují doplňování proměnných, které jsou viditelné v aktuálním bloku.



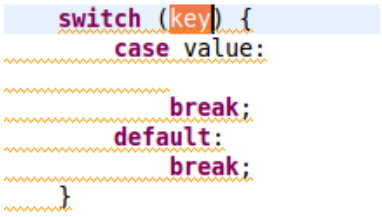
Obrázek 3.4: Ukázka nástroje content assist

Šablony

Vylepšením automatického doplňování popsaného v předchozím odstavci jsou šablony (*templates*). Prostřednictvím nich je možné definovat ucelené bloky kódu, jakými je například

[4]V prostředí Eclipse může uživatel vyvolat nápovědu prostřednictvím stisku kláves `CTRL+SPACE`.

konstrukce `switch`. Při pokusu o napsání klíčového slova `switch` je uživateli nabídnuto pomocí nástroje *content assist* vložení šablony pro celou konstrukci `switch`. Do ní pak může být snadno doplněn zbývající kód.

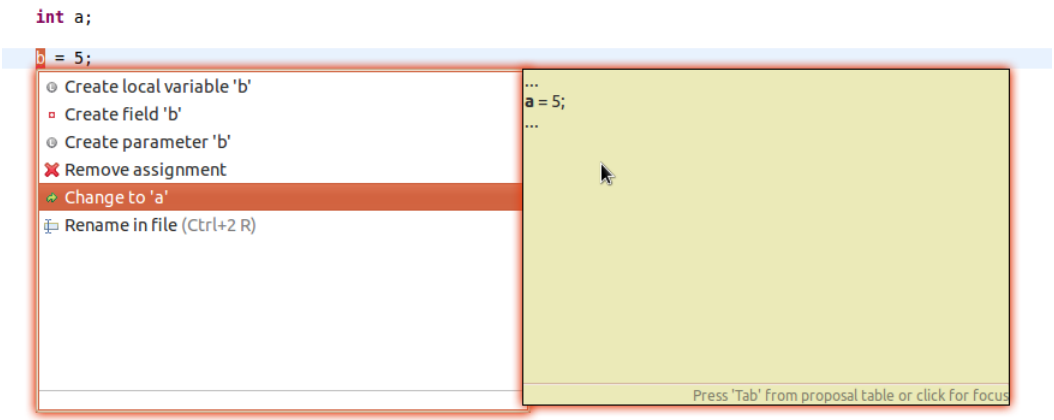


Obrázek 3.5: Ukázka vložení šablony konstrukce switch

Šablony jsou další možností, jak uživateli vnutit jednotný styl formátování. Úseky kódu jsou vkládány tak, aby dodržely použité odsazování a zalamování kódu. Kvalita tohoto nástroje je odvozena od kvality nápovědy. Ta rozhoduje o tom, jaké bloky je povoleno vložit na konkrétní místo v kódu.

Quick fix

Kvalitní editory (jakým je například editor jazyka Java v prostředí Eclipse) obsahují nástroj *Quick fix*. Jedná se o dodatečnou analýzu chybně napsaného zdrojového kódu. Editor se snaží najít co nejlepší řešení chyby. V případě nalezení doporučí nalezená východiska uživateli. V prostředí Eclipse je to provedeno pomocí ikony s motivem rozsvícené žárovky umístěné na levém okraji řádku obsahující chybný kód.

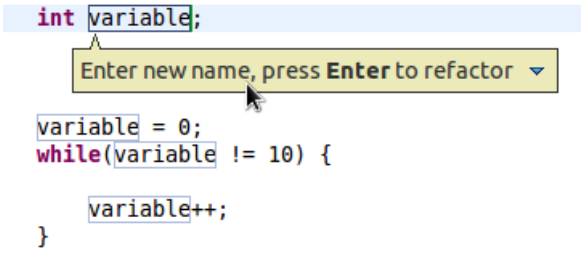


Obrázek 3.6: Seznam doporučení pro opravu sémantické chyby

Ne vždy je nalezeno optimální řešení. Jedná se o jednu z nejsložitějších oblastí problematiky editorů. K realizaci tohoto nástroje je potřebné dobře rozumět sémantice programovacího jazyka.

Správa elementů zdrojového kódu

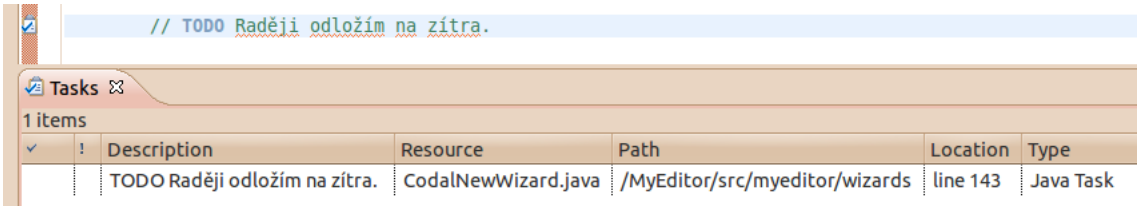
Pro lepší správu zdrojových kódů je vhodné vést informace o deklaracích proměnných, metod a jiných struktur podporovaných daným programovacím jazykem. Tyto deklarace je dále vhodné provázat se všemi dalšími výskyty těchto struktur. Díky tomu je možné provádět hromadné přejmenování identifikátorů (*refactoring*). Informace o výskytech struktur mohou být pak dále použity pro pohledy určené pro navigaci, hierarchii tříd, apod.



Obrázek 3.7: Hromadné přejmenování proměnné

Základní akce editoru

Eclipse usnadňuje tvorbu editorů v mnoha ohledech. K dispozici jsou další komponenty, které usnadňují programování. Jedná se o přidané hodnoty editoru a nesouvisí přímo se zpracováním zdrojového kódu. Příkladem je možnost označování míst v kódu záložkami (např. označení `TODO`) a umísťování breakpointů. Eclipse mimo jiné poskytuje základní funkcionalitu, jakou je možnost vrácení změn (*undo/redo*) nebo vyhledání řetězce ve zdrojovém kódu. Tyto komponenty jsou řešeny interně ve vývojovém prostředí a jsou součástí každého editoru.



Obrázek 3.8: Vytvoření úkolu

3.3.3 Pohledy

Dodatečné funkce editorů zmíněné v minulé kapitole nemusí být pro dnešní potřeby vývojářů dostačující. Při implementaci rozsáhlých projektů vzniká potřeba orientovat se ve zdrojových kódech a knihovnách. Pro odladění programů je nezbytné získávat informace ohledně překladu a běhu programu. Z toho důvodu jsou k dispozici pohledy.

Pohled je další typ okna. Na rozdíl od editorů neslouží k editaci zdrojových kódů, ale k zobrazování pomocných informací. Tyto informace jsou získávány zpravidla ze zdrojových

kódů, struktury projektů nebo běhu přeložených programů. Je používán především k navigaci ve stromové struktuře projektu (např. pohled *Package Explorer*), k zobrazení použitých metod, tříd a proměnných (*Outline*). Slouží pro orientaci v hierarchickém uspořádání prvků projektu (*Call/Type Hierarchy*). Pohledy jsou použity rovněž pro informace o překladu a pro nápovědu (*Console, Problems, Error Log*). Umístění pohledů není pevně stanoveno. Jsou zpravidla ukotveny okolo editorů. Uživatelé je mohou přemísťovat a seskupovat podle potřeb, případně i vyjmout ze základního pracovního prostředí a zobrazit jako samostatná okna.

V rámci této práce jim nebudu věnovat velkou pozornost. Více informací lze získat například v mé bakalářské práci, v rámci které jsem implementoval pohled zobrazující grafy volání funkcí [12].

3.3.4 Akce

Každý plugin může ve svých rozšířeních (*extension points*) definovat prvky, jakými jsou tlačítka rozšiřující nástrojové lišty (*toolbary*), kontextová nebo hlavní menu. Tyto prvky v sobě nesou akce, pomocí kterých je možné ovládat celé pracovní prostředí Eclipse.

Akce jsou využívány z velké části editory. Každý editor má k dispozici kontextové menu. V něm jsou tlačítka určená pro kopírování a vkládání, hromadné úpravy a porovnávání kódu. Pomocí nabídek kontextových menu je možné otevírat dialogy pro změnu nastavení editoru. Těmito nastaveními jsou především možnosti změn stylů zvýrazňování a konfigurací nástroje *Content assist*, šablon a jazyka pro kontrolu pravopisu komentářů.

3.4 Tvorba nového editoru

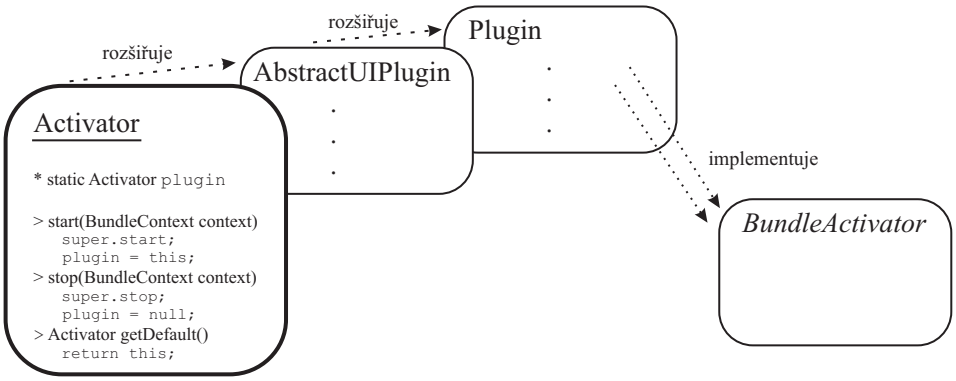
Vývojové prostředí Eclipse je možné rozšířit o vlastní sadu editorů. Podmínkou pro tvorbu editoru je vytvoření plug-inu, který musí dodržet předepsanou strukturu.

Eclipse poskytuje průvodce, pomocí kterého je možné vytvořit nový plug-in obsahující základní implementaci editoru spolu s příslušným průvodcem pro tvorbu souborů náležící novému editoru. V průvodci je možné stanovit přípony souborů, které budou automaticky asociovány s novým editorem. Výsledkem jsou všechny potřebné třídy a konfigurační soubory pro překlad a spuštění.

Aktivační třída

V souboru `MANIFEST.MF` je uvedena deklarace hlavní třídy plug-inu. Bývá zpravidla označována jménem `Activator` a slouží k inicializaci plug-inu. Třída poskytuje metody pro přístup ke zdrojům používaným v plug-inu. Je to první třída, na kterou se systém při aktivaci plug-inu odkáže. Při tom je vytvořena její instance, během jejíž existence nemůže být vytvořena žádná jiná instance této aktivační třídy. Často se proto do konstruktoru implementuje testování statické proměnné, v které je uložena reference na instanci třídy `Activator`. Pokud v době volání konstruktoru ukazuje na nějaký objekt, vygeneruje se výjimka `IllegalStateException`.

Každá aktivační třída plug-inu musí implementovat rozhraní `BundleActivator`. To je zajištěno rozšířením třídy `Plugin`, respektive `AbstractUIPlugin` v případě pluginu obsahující grafické uživatelské rozhraní. V takovém případě je nutné, aby konfigurační soubor `MANIFEST.MF` obsahoval v záložce závislostí knihovnu `org.eclipse.gui`.



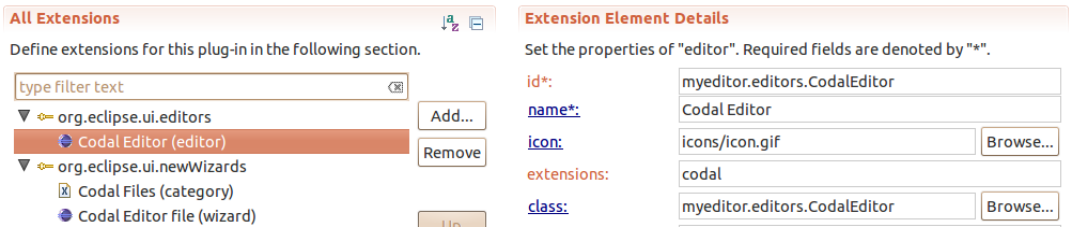
Obrázek 3.9: Aktivační třída

3.4.1 Rozšíření plug-inu

V souboru `plugin.xml` jsou v bloku rozšíření (*extensions*) uvedeny deklarace automaticky vytvořeného editoru a průvodce.

Přidání editoru

První z deklarací je předpis pro editor. Na následujícím obrázku 3.4.1 je ukázka specifikace vzorového editoru s názvem *Codal Editor*. Spolu názvem je povinné vyplnit také identifikátor. Editor má dále vyplněny volitelné atributy, jakými je ikona a přípona souborů, které budou asociovány s novým editorem (v tomto případě soubory s příponou `codal`).



Obrázek 3.10: Karta deklarace editoru v souboru `plugin.xml`

Ve formuláři je specifikována pod položkou *class* třída `CodalEditor`. Tato reference směřuje na třídu, ve které je uveden kód editoru. Jedná se o hlavní třídu nově vytvářeného editoru, která musí implementovat rozhraní `IEditorPart`. Typicky je to zajištěno rozšířením jedné z abstraktních tříd:

- Základní typ editoru poskytuje třída `EditorPart`. Jedná se o klasický typ editoru, ve kterém je možné přímo přistupovat ke zdrojovému kódu a editovat ho.
- Druhou možností je rozšířit třídu `MultiPageEditorPart`. V Eclipse existují editory, které se skládají z více stránek. Typickým příkladem je editor souborů formátu XML. Na jedné stránce je uveden zdrojový kód a na druhé je pomocí stromové struktury zobrazena přehlednější abstrakce zdrojového kódu. Uživatel může provádět editace ve stromové struktuře a ušetří si tím psaní XML značek. Soustředí se pouze na data.

- Vylepšením druhé možnosti je použití formulářů pro zadávání kódu. Takto je řešen například editor konfiguračních souborů plug-inu, ve kterém je možné zadávat vstupní data jak přímo do kódu, tak i přes uživatelsky příjemnější formulář. Formulářové typy editorů lze implementovat rozšířením třídy `FormEditor`.

Editor jazyka CodAL bude obsahovat pouze jednu stránku, ve které bude možné přímo editovat zdrojový kód[5]. Bude proto rozšiřovat třídu `EditorPart` (nepřímo třídu `WorkbenchPart`). Tato abstraktní třída deklaruje základní metody editoru:

- Metoda `createPartControl` vytváří samotný editor. V ní by měl být uveden kód pro fyzické vytvoření grafické komponenty. Může obsahovat například volání metod `createTextEditor` nebo `createTree`, které zajistí implementaci specifického druhu editoru.

- Inicializace editoru proběhne pomocí metody `init`. Přes tuto metodu jsou editoru předány vstupní data.

- Pro ukládání obsahu jsou k dispozici metody `doSave` a `doSaveAs`. Metody jsou volány v případě, že uživatel žádá uložit obsah. V nich je uvedena implementace fyzického zápisu do souboru nebo například implementace průvodce souborů.

- Implementovat je nutné další metody jako například `isDirty` (testuje, zda je soubor změněný) nebo `gotoMarker` (pro nastavení kurzoru). Jejich přesný výčet a popis je uveden v programové dokumentaci nástroje Eclipse.

Ve formuláři deklarující nový editor je dále možné specifikovat třídu `conributorClass`. Tato třída implementuje rozhraní `IEditorActionBarContributor`. Prostřednictvím této třídy je možné přidávat nové akce (viz sekce 3.3.4) do nástrojové lišty a menu. Obsahuje zpravidla akce zpět/vpřed (*undo/redo*), kopírování/vkládání (*copy/paste*), označování (*select all*) nebo vyhledávání. Tyto akce jsou automaticky vygenerovány průvodcem tvorby nového plug-inu.

Přidání průvodce

Plnohodnotný editor musí obsahovat průvodce pro tvorbu souborů asociovaných s daným editorem. V kartě rozšíření je možné tohoto průvodce deklarovat obdobným způsobem jako editor.

Obrázek 3.11: Karta deklarace průvodce pro vytvoření nového souboru

[5]Jelikož je jazyk CodAL z části deklarativní jazyk, bylo by zajímavé do budoucna popřemýšlet nad variantou podobnou editoru jazyka XML. Uživatel by mohl pomocí stromové struktury zadávat zdroje (paměti, registry, sběrnice) a prostřednictvím formulářů vyplňovat jejich parametry.

Na příkladu je znázorněna deklarace průvodce se jménem *Codal Editor File*. Obdobným způsobem, jako je tomu u editoru, je definován identifikátor, ikona a výchozí třída pro implementaci chování. Pro přehlednost byl průvodce asociován s kategorií *Codal Files*. Ve výchozí třídě `CodalNewWizard` je uveden kód průvodce. Třída je odvozena ze třídy `Wizard` dosupné z knihovny JFace.

Tvorba grafických průvodců překračuje rámec této práce, proto dále odkáži na dokumentaci toolkitu JFace [21].

3.4.2 Zpracování zdrojového kódu

Podstatnou částí tvorby nového editoru je implementace parseru, který bude zpracovávat vstupní data a ukládat je do vhodné struktury – zpravidla stromu. Tvorba této struktury je důležitá pro následnou analýzu zdrojového kódu. Využijí ji například nástroje pro zvýrazňování kódu, detekci chyb nebo statickou analýzu (viz podsekce 3.3.2). Stromová struktura prvků zdrojového kódu může být dále použita pohledy znázorňující seznamy deklarací, hierarchii volání a typů. Tvorba parserů je rozsáhlou problematikou a vymezil jsem ji proto samostatnou kapitolu 4. V této podsekci je rozebrán styl výsledného uložení dat získaných ze zdrojového kódu.

Kvalita nástrojů pro analýzu kódu závisí na množství informace obsažené ve stromové struktuře. Důležitá je kompatibilita stylu uložení dat s těmito nástroji. Pro uchování dat jazyka CodAL byl zvolen stejný styl, jako používá knihovna CDT. Díky tomu, že jazyk CodAL obsahuje v některých svých blocích obdobné elementy (například výrazy), jaké jsou obsaženy v datovém modelu CDT, bylo možné přímo vycházet z tohoto projektu.

CDT

Iniciály CDT jsou zkratkou pro název *C/C++ Development Toolkit*. Jedná se o projekt vývojového prostředí Eclipse, který rozšiřuje pracovní prostředí o sadu editorů a nástrojů pro analýzu zdrojových kódu jazyka C a C++. CDT nabízí možnost tvorby C/C++ projektů a správu nástrojů pro překlad. Knihovna je vybavena nástroji pro analýzu zdrojových kódů a dodatečné pohledy pro navigaci ve zdrojových kódech (seznamy metod a proměnných, typová hierarchie, call graph, ...).

Pro uchování dat získaných ze zdrojového kódu CDT používá CDT DOM (*Document Object Model*). Data jsou uložena ve stromové struktuře nazývané *abstraktní syntaktický strom* (zkráceně AST z anglického názvu *abstract syntax tree*). Pomocí tohoto modelu je možné na zdrojový kód nahlížet ve dvou pohledech – syntaktickém a sémantickém [19].

Syntaktický pohled CDT DOM

Smyslem tohoto pohledu je rozpoznávat jednotlivé prvky zdrojového kódu. Každý prvek je proto reprezentován samostatným uzlem stromu. Tyto uzly jsou v objektovém modelu knihovny CDT vedeny jako instance obecného uzlu `IASTNode`. Toto rozhraní deklaruje základní metody pro práci uzly, jakou je například zjišťování rodiče nebo potomků. Každý uzel je pak dále rozšířen o specifické metody a atributy závisející na typu uzlu.

- U listů se jedná o konkrétní vlastnosti lexémů, jakým je například název identifikátoru nebo hodnota proměnné.

- U nelistových uzlů se jedná především o druh dat (potomků), který zapouzdřují. Příkladem je deklarace funkce, která je složena z deklarátoru, modifikátorů a seznamu parametrů.

Každý uzel proto zpravidla implementuje specifické rozhraní deklarující metody pro daný typ (například `IASTName` pro identifikátor, `IASTExpression` pro výraz nebo `IASTSimpleDeclaration` pro deklaraci proměnných). Tyto rozhraní jsou odvozena ze základního rozhraní `IASTNode`. Na prvky je proto možné nahlížet obecně jako na strom uzlů typu `IASTNode` nebo specificky jako na strom konkrétních prvků. Toto je využíváno především pohledy pro navigaci ve zdrojovém kódu. V závislosti na typu dat, který zobrazují, si ze stromu vyfiltrují prvky požadovaných typů.

Podrobnější popis objektového modelu je ukázán v kapitole 7, která popisuje implementaci objektového modelu pro AST jazyka CodAL. V této kapitole jsou rovněž uvedeny některé nové typy uzlů vytvořené konkrétně pro jazyk CodAL.

Sémantický pohled CDT DOM

Druhým pohledem je sémantický pohled (někdy také nazývaný jako logický). Tento model je reprezentován sémantickými prvky programu, jakými jsou proměnné, typy a funkce. Hlavním smyslem je významově provázat jednotlivé sémantické prvky. Důležité jsou informace o deklaracích, které stanovují rozsah viditelnosti elementu. Tyto větve AST stromu jsou reprezentovány prvky implementující rozhraní `IBinding`.

Sémantický pohled je využíván například při testování, zda je proměnná deklarována. Z uzlu nesoucího výskyt proměnné (`IASTName`) se odkážeme na příslušný *binding*. Tyto bindingy dále mohou posloužit například při hromadném přejmenování proměnných (*refactoringu*). Více je popsáno v kapitole 8.

Kapitola 4

Analýza zdrojových kódů v editorech

Analýza zdrojových kódů představuje parsování vstupního souboru a následné vyhodnocení informace získané z parsování. Parsováním zdrojových kódů je v kontextu editorů myšlen proces převodu vstupního textu (posloupnosti znaků) do uspořádané stromové struktury – *abstraktního syntaktického stromu* (nebo zkráceně AST – *abstract syntax tree*). Účelem této struktury je významově odlišit jednotlivé části zdrojového kódu, protřídit je v závislosti na jejich důležitosti a znázornit mezi nimi logické závislosti. Vytvořená struktura je následně použita nástroji pro zvýrazňování a orientaci ve zdrojovém kódu. V editoru jazyka Eclipse bude vytvořen abstraktní syntaktický strom obsahující prvky odvozené z objektového modelu nástroje CDT (CDT DOM). Editor bude moci být propojený s existujícími nástroji knihovny CDT.

Jedním z hlavních účelů parsování zdrojového kódu je rozhodnutí, zda zdrojový kód obsahuje pouze povolené konstrukce daného jazyka (v našem případě jazyka CodAL). Pro formalizaci tohoto testu je nutné definovat některé základní pojmy z oblasti teoretické informatiky. Vycházel jsem přitom především ze skript Teoretická informatika [25], v kterých je možné najít přesné definice a doplňující informace k dané problematice.

4.1 Teorie formálních jazyků

Tato oblast zkoumá jazyky z teoretického hlediska a stanovuje formalismy pro jejich popis. Zabývá se problémem náležitosti jazyků do tříd, jejichž prvky je možné popsat určitým typem gramatiky. V závislosti na tom jsou zkoumány typy automatů, které dokáží přijímat jednotlivé třídy jazyků. Základ této teorie sestavil americký matematik Noam Chomsky, který se snažil matematicky popsat přirozený jazyk. V souvislosti s tím vytvořil členění gramatik formálních jazyků.

4.1.1 Jazyk

Hlavním úkolem k určení korektnosti zdrojového kódu je rozhodnutí, zda je zdrojový kód validní posloupnost znaků napsaná ve zvoleném zdrojovém jazyce. K tomu je nejprve nutné stanovit množiny všech přípustných symbolů. Tuto neprázdnou množinu označme symbolem Σ a pojmenujme *abeceda* [25]. Příkladem abecedy může být například množina symbolů zvolené znakové sady, v které programátor píše zdrojový kód (napíklad UTF-8).

Prostřednictvím symbolů z abecedy Σ je možné sestavovat *řetězce*. Řetězcem nad abecedou Σ je myšlena libovolná posloupnost symbolů z abecedy Σ (přesná definice v [25]). Pořadí symbolů v řetězci je významné. Prázdný řetězec je označován symbolem ϵ. Příkladem řetězce je klíčové slovo `memory` jazyka CodAL. Toto slovo je řetězcem nad abecedou znakové sady UTF-8.

Uvažujme nyní abecedu Σ a označme množinu všech možných řetězců nad touto abecedou jako Σ^*. Definice z [25] říká, že jazyk L je tvořen přípustnou množinou řetězců nad abecedou Σ. Tato množina tvoří podmnožinu všech možných řetězců nad abecedou Σ. V programovacím jazyce můžeme touto množinou označit jako množinu všech správně napsaných zdrojových kódů. Samotné klíčové slovo `memory` je sice řetězcem nad abecedou programovacího jazyka CodAL, ale není korektním zdrojovým kódem.

4.1.2 Gramatika

Nejjednodušším způsobem, jakým je možné definovat množinu přípustných řetězců jazyka L, je provedení množinového výčtu prvků. Vzhledem k tomu, že programovací jazyky jsou tvořeny nekonečnou množinou validních programů, je nutné zavést speciální popisný nástroj – *gramatiku*.

Definice 4.1 Gramatika G je dle [25] čtveřice $G = (N, \Sigma, P, S)$, kde

- N je konečná množina nonterminálních symbolů,
- Σ je konečná množina terminálních symbolů, $N \cap \Sigma = \emptyset$,
- P je konečná podmnožina kartézského součinu $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$,
- $S \in N$ je výchozí (také počáteční) symbol gramatiky.

Gramatika je základní nástroj pro popis jazyka. Obsahuje dvě množiny abeced, které jsou navzájem disjunktní. Množina neterminálních symbolů se skládá z pomocných symbolů. Musí obsahovat minimálně jeden symbol – *startovací neterminál* S. Tyto symboly jsou v závislosti na stanovené množině pravidel expandovány na řetězce tvořené terminálními nebo dalšími (stejnými nebo jinými) neterminálními symboly. Terminální symboly nelze dále expandovat a tvoří listy generovaného stromu. Množina těchto symbolů je ekvivalentní s abecedou, pomocí které jsou tvořeny řetězce jazyka popisovaného gramatikou G.

4.1.3 Chomského klasifikace

Dle chomského klasifikace lze gramatiky a tedy i formální jazyky rozdělit do čtyř tříd. Dělení je provedeno v závislosti na tom, jakých podob mohou nabývat pravidla gramatik (jakou sílu popisu představují). Přesný výčet a popis tříd je dostupný v [25]. Pro problematiku překladačů programovacích jazyků jsou podstatné pouze **gramatiky typu 3 a 2**.

Gramatiky typu 3

Pravidla gramatiky typu 3 jsou omezeny nejvíce. Mohou nabývat tvaru:

$$A \to xB \text{ nebo } A \to x; A, B \in N, x \in \Sigma^*.$$

Těmto gramatikám se říká pravé lineární[1]. V porovnání s gramatikami ostatních typů je pomocí této třídy gramatik možné popsat nejmenší množinu jazyků. Tyto jazyky se nazývají regulární. Jsou přijímány konečným automatem a mimo lineárních gramatik je možné je popsat regulárními výrazy, jejichž popisná síla je ekvivalentní [25]. Gramatika typu 3 může být použita pro popis lexikálních možností programovacího jazyka (z jakých lexémů je jazyk složen).

Gramatiky typu 2

Popisně silnější třídou gramatik jsou v chomského hierarchii gramatiky typu 2. Jejich pravidla mají tvar:

$$A \to y, A \in N, y \in (N \cup \Sigma)^*.$$

Na rozdíl od pravidel gramatik typu 3, může pravá strana pravidla nabývat libovolné posloupnosti terminálů a neterminálů. Díky tomu mohou být popsány základní konstrukce programovacích jazyků, jakou je například stejný počet levých a pravých závorek. Pro přijetí takového jazyka je nutné mít k dispozici zásobník, díky kterému je možné sledovat počty výskytů komplementárních symbolů. Gramatiky typu 2 hrají klíčovou roli při syntaktické analýze zdrojových kódů programovacích jazyků. Jazyky popsané těmito gramatikami se jmenují bezkontextové a jsou přijímány zásobníkovým automatem.

4.2 Etapy parsování

Ze sekce 4.1 vyplývá, že na parsování zdrojových kódu lze formálně nahlížet jako na test náležitosti vstupního řetězce (zdrojového kódu) do množiny obsahující všechny přípustné varianty řetězců (množiny zdrojových kódů jazyka CodAL). Analýza vstupního řetězce probíhá v několika etapách, které jsou vzájemně provázány. Jsou jimi *lexikální*, *syntaktická* a *sémantická* analýza.

Příklad 4.1 Pro ilustraci bude uvažován jazyk *Suma* umožňující tyto konstrukce:

- deklarace celočíselných proměnných klíčovým slovem `def` (při deklaraci jsou inicializovány na hodnotu 0);
- operace přiřazení celých čísel do deklarované proměnné pomocí operátoru '`=`';
- operace sčítání celých čísel operátorem '`+`';
- příkaz pro vypsání deklarované proměnné na standardní výstup reprezentovaný klíčovým slovem `print`;
- příkaz načítání do deklarovaného proměnné ze standardního vstupu reprezentovaný klíčovým slovem `scan`;
- řádkové komentáře uvozené znakem '`/`';
- znak '`;`' pro ukončení příkazu.

[1] Existují i levé lineární gramatiky, jejichž popisná síla je stejná jako u pravých lineárních gramatik. Tyto gramatiky je možné mezi sebou převádět.

Jazyk obsahuje klíčová slovo `def`, `scan` a `print`, binární operátory '`=`' a '`+`' a znak '`/`'. Ve výrazech je možné používat kulaté závorky. Příkazy je nutné zakončovat znakem '`;`'. Identifikátory mají stejný formát jako podporuje jazyk C.

Příklad 4.2 Příklad zdrojového kódu je následující:

```
def a; def b; def c; / deklarace proměnných a, b, c
scan a; / načtení celého čísla pro proměnné a
b = 1; / uložení konstanty do proměnné b
c = (a + b) + 1; / uložení součtu hodnot proměnných a, b
                   / inkrementováno o jedničku do proměnné c
print c; / výpis proměnné c
```

V příkladu 4.2 jsou ukázány všechny možné konstrukce, které je možné napsat v jazyce *Suma*. V následujících podsekcích budou na pokusném jazyce *Suma* ilustrovány tři zmíněné analýzy.

4.2.1 Lexikální analýza

Jedná se o první analýzu, jejíž vstupem je zdrojový kód. Jejím cílem je rozpoznat jednotlivá slova (lexémy), která se mohou vyskytovat ve zvoleném jazyce. Jazyk *Suma* můžeme v této analýze reprezentovat pomocným jazykem:

$$Suma' = \{\ def,\ scan,\ print,\ =,\ +,\ ;,\ (,\ )\ \} \cup ID \cup NUM \cup STR,$$

kde ID je množina všech identifikátorů, NUM je množina všech celých čísel a STR množina všech řetězců začínající symbolem '/' a končícím znakem nového řádku. Množiny ID, NUM a STR mohou být reprezentované regulárním výrazem, tudíž se jedná o jazyk typu 3. Pro tento jazyk je možné sestrojit konečný automat, který bude znak po znaku načítat vstup a rozpoznávat řetězce náležící jazyku $Suma'$. Tyto řetězce (*lexémy*) jsou ukládány do datových jednotek – *tokenů*, které obsahují typ a hodnotu lexému. Tokeny jsou dále využívány při syntaktické analýze. V případě rozpoznání chybného řetězce je předán speciální token reprezentující danou lexikální chybu. Komentáře jsou v této fázi analýzy zahazovány.

4.2.2 Syntaktická analýza

V rámci této analýzy jsou přijímány tokeny získané lexikální analýzou a řazeny do stromové struktury – *derivačního stromu*. Jazyk *Suma* nyní můžeme reprezentovat jazykem $Suma''$, který je popsaný gramatikou

$$G = (\{S,P,E,L\},\ \{def,\ scan,\ print,\ eq,\ plus,\\ semicln,\ left,\ right,\ id,\ num,\ lexerror\},\ P,\ S),$$

kde P obsahuje následující pravidla:

1. $S \rightarrow P\ semicln \mid SS \mid lexerror$
2. $P \rightarrow def\ id \mid scan\ id \mid print\ id \mid id\ eq\ E$
3. $E \rightarrow left\ E\ right \mid L\ plus\ E \mid L$
4. $L \rightarrow id \mid num$

Pravidla gramatiky G splňují podmínku pravidla gramatiky typu 2. Množina terminálů je tvořena množinou tokenů definovanou lexikální analýzou. Výčet prvků je ve stejném pořadí, jakým jsou definovány řetězce jazyka $Suma'$, doplněným o speciální token značící lexikální chybu[2]. Není zde obsažen token pro komentář.

Cílem syntaktické analýzy je otestovat řazení tokenů získaných lexikální analýzou. Pokud bude možné nalézt takovou posloupnost pravidel, které vygenerují z výchozího neterminálu posloupnost testovaných tokenů, je zdrojový kód syntakticky korektní. Například přiřazení `c = 2;` lze vygenerovat posloupností čtyř pravidel:

$$S \to P\ semicln,\ P \to id\ eq\ E,\ E \to L,\ L \to num.$$

Tuto skutečnost lze modelovat pomocí operací *přímé derivace* (definované v [25]) jako:

$$S \Rightarrow P\ semicln \Rightarrow id\ eq\ E\ semicln \Rightarrow id\ eq\ L\ semicln \Rightarrow id\ eq\ num\ semicln.$$

Jednotlivé řetězce terminálů a neterminálů, které lze vygenerovat z počátečního neterminálu, se nazývají *větné formy* [25]. Grafickým znázorněním odvození větné formy je *derivační strom* [25]. Jedná se o strukturu, která má ve svém kořenu počáteční neterminální symbol a ve svých listech symboly generované větné formy (skládající se zpravidla z terminálů). Činností syntaktického analyzátoru je nalézt derivační strom pro danou posloupnost tokenů.

Existují dva způsoby, jakými je možné sestavit derivační strom. První možností je stavět strom od výchozího neterminálu směrem k terminálům (*shora dolů*). Druhá možnost preferuje opačný směr (*zdola nahoru*). Společným úkolem obou přístupů je hledání prvního podřetězce větné formy, který může být redukovaný do nadřazeného neterminálu. Problémem je, pokud tuto redukci nelze jednoznačně určit a je nutné používat zpětného navracení (*backtrackingu*).

Programovací jazyky zpravidla spadají do třídy *deterministických bezkontextových jazyků*. Jedná se o podmnožinu bezkontextových jazyků. Jejich výhodou je, že mohou být popsány deterministickou gramatikou. Syntaktický analyzátor je schopný analyzovat zdrojový kód bez návratu. Existují tři typy deterministických bezkontextových gramatik:

1. Prvním typem jsou **precedenční gramatiky**. Parsery založené na těchto gramatikách používají metodu syntaktické analýzy zdola nahoru. Jejich podstatou je vytvoření tabulky precedencí (významností) jednotlivých terminálů. V závislosti na těchto pravidlech jsou terminály buď ukládány na zásobník nebo redukovány na neterminály dle stanovených pravidel. Gramatiky jsou použitelné například pro popis a tvorbu parserů výrazů.

2. Druhým typem jsou **LL gramatiky**. Zde je uplatňována syntaktická analýza zdola nahoru. Podstatou je pro každý neterminál gramatiky předpočítat pomocné množiny *First*, *Empty* a *Follow*[3]. Tyto množiny pomáhají určit, jaké nejlevější terminály je možné z daných neterminálů expandovat. Pomocí toho je možné sestavit deterministickou rozhodovací tabulku, která zohledňuje vstupní symbol a symbol na zásobníku. Metoda je využívána například pro parsování rekurzivním sestupem. LL Gramatiky jsou využívány například generátorem editorů Xtext.

[2]Do gramatiky se navíc na vhodná místa přidají pravidla simulující syntaktické chyby. V případě že není možné provést žádné jiné pravidlo, je provedeno výchozí pravidlo a v derivačním stromu vytvořen uzel znázorňující syntaktickou chybu.

[3]Přesné definice množin jsou uvedeny například v [24].

3. Posledním typem jsou **LR gramatiky**. Tyto gramatiky pojímají v porovnání s předchozími dvěma typy největší skupinu jazyků. Parsování probíhá stylem zdola nahoru. Pro jeho paměťovou náročnost je využívána zjednodušená varianta zvaná LALR parser (*Look-Ahead LR parser*). Tuto variantu implementuje například generátor LPG používaný pro tvorbu parseru nového editoru jazyka CodAL.

4.2.3 Sémantická analýza

Třetí analýzou je sémantická analýza. Vstupem této analýzy je derivační strom získaný syntaktickou analýzou. Výsledkem analýzy je *abstraktní syntaktický strom*. V této fázi je zkoumán význam dat zdrojového kódu. Jsou ověřovány například deklarace proměnných a probíhá typová kontrola. Jsou prováděny dodatečné kontroly, které nebylo možné (nebo bylo komplikované) popsat bezkontextovou gramatikou.

Příklad 4.3

```
/ def a;
a = 1; / uložení konstanty do proměnné a
```

Příklad 4.3 je syntakticky korektní, avšak z významového hlediska je chybný. Přiřazování hodnoty 1 se provádí do nedeklarované proměnné `a`. Aby byl příklad sémanticky správně, muselo by se provést odkomentování definice této proměnné.

4.3 Generátory parserů

V předchozí sekci byl ukázán postup zpracování zdrojového kódu. Jedná se o pracnou činnost náchylnou na vznik chyb. Vzhledem k tomu, že existuje matematický model popisující tuto činnost, je vhodné řešit tento problém automatizovaně. Existují k tomu generátory parserů. Činností těchto nástrojů je přijímat gramatiku implementovanou ve specifickém formátu a generovat zdrojový kód reprezentující překladač zdrojového kódu (*parser*).

4.3.1 Backus-Naurova forma

Pro popis bezkontextových gramatik přijímaných generátory parserů se používá *Backus-Naurova forma* (zkráceně BNF). BNF je notace využívaná pro popis syntaxe jazyků mezi programátory [13]. Notace převádí pravidla gramatiky do specifického formátu vhodného pro další zpracování. Příklad pravidla reprezentující výraz jazyka *Suma* v BNF může být zobrazeno následujícím způsobem:

```
<E> ::= "(" E ")" | <L> "+" <E> | <L>
```

Na levé straně pravidla může být pouze jeden neterminál. BNF je generátory využívána zpravidla v modifikovaných verzích.

4.3.2 Xtext

Editor jazyka CodAL využíval ve své předchozí verzi generátor Xtext [23]. Jedná se o svobodný framework, který je součástí projektu vývojového prostředí Eclipse. Generátor Xtext

přijímá gramatiku v Backus-Naurově formě a generuje plnohodnotný editor jazyka popsaného vstupní gramatikou. Součástí vygenerovaného editoru jsou uživatelské nástroje pro analýzu zdrojového kódu, statickou analýzu a pohledy pro orientaci ve zdrojovém kódu.

Výsledný parser přijímá jazyky popsané LL gramatikami, což je vzhledem k jazyku CodAL nevýhoda, protože gramatika jazyka CodAL, určená pro tvorbu překladače, je typu LALR. LL gramatiky mají menší popisnou sílu. Pro každé pravidlo gramatiky jazyka CodAL byla generátorem Xtext vygenerována speciální metoda. Vzhledem ke složitosti jazyka CodAL byl výsledný zdrojový kód velice objemný.

Nevýhodou nástroje Xtext je to, že neumí zohlednit direktivy preprocesoru. Před tím, než je zdrojový kód jazyka CodAL předán překladači, je zpracován preprocesorem jazyka C. Preprocesor modifikuje zdrojový kód v závislosti na vložených direktivách. Ve zdrojovém kódu jsou nahrazeny všechny konstanty (které mohou být víceřádkové). Jsou vyloučeny bloky, v kterých není splněna podmínka direktivy `#ifdef`. Řádky upraveného zdrojového kódu nemusí odpovídat řádkům původnímu souboru. Hlavní nevýhodou bývalého editoru bylo, že z důvodu výše popsaného problému zvýrazňoval i některé úseky kódu, které nebyly chybné.

4.3.3 LALR Parser Generator

V nové verzi editoru jazyka CodAL je parser generovaný pomocí svobodného nástroje LPG – LALR Parser Generator [16]. Generátor přijímá jazyky popsané LALR gramatikami a generuje na základě nich lexer a parser napsaný v jazyce Java, C, nebo C++. Gramatiky musí být implementované v Backus-Naurově formě. Verze BNF používaná v LPG umožňuje ke každému pravidlu dopsat sémantické pravidlo (konkrétně volání příslušné funkce), které se vykoná při vyvolání tohoto pravidla. Pomocí těchto metod může být konstruován abstraktní syntaktický strom.

Výhodou je, že v metodách reprezentující sémantická pravidla, může být uveden libovolný kód. V případě editoru jazyka CodAL zde budou uvedeny operace se zásobníkem a tvorba stromu skládajícího se z uzlů objektového modelu CDT. LPG zajistí vygenerování kódu, který se bude starat o správné volání metod v závislosti na přijímaném textu (vykonaných pravidlech gramatiky). Podrobnější popis jednotlivých činností, které jsou provedeny při tvorbě abstraktního syntaktického stromu, je uveden v sekci 7.1.

Kapitola 5

Návrh nového editoru jazyka CodAL

V rámci první části dokumentu byla rozebrána teoretická část diplomové práce. Popsal jsem syntaktické a sémantické vlastnosti jazyka CodAL sloužícího pro modelování architektur aplikačně specifických systémů. V souvislosti s tím jsem rozebral možnosti, které nabízí současné vývojové prostředí projektu Lissom určené pro souběžný návrh hardwaru a softwaru. Shrnul jsem aktuální stav editoru jazyka CodAL a porovnal ho s možnostmi editorů jiných programovacích jazyků. Vycházel jsem z funkcionality editorů vývojového prostředí Eclipse, na kterém je vývojové prostředí projektu Lissom založeno. Na teoretické úrovni jsem nastínil základní problematiku editorů týkající se především zpracování zdrojového textu a následný převod do vhodné struktury, která je použitelná pro další analýzu syntaktických a sémantických vlastností. V závěru čtvrté kapitoly jsem uvedl dva zástupce generátorů parserů.

Informace, které jsem získal teoretickou částí diplomové práce, jsem dále použil pro část praktickou. Nejprve jsem provedl návrh nového editoru jazyka CodAL, v kterém jsem shrnul jednotlivé technologie pro tvorbu tohoto editoru. Poté jsem naplánoval jednotlivé pracovní etapy, které bylo nutné provést pro dosažení stanovených cílů.

5.1 Použité technologie

Nový editor jazyka CodAL byl navržen jako plug-in vývojového prostředí Eclipse. Konkrétně byl začleněn mezi existující plug-iny vývojového prostředí Codasip Studio, které je na Eclipse založeno. Plug-in rozšířil existující projekt Eclipse CDT, který poskytuje editor jazyka C a C++ spolu s doplňky zajišťující jeho uživatelskou přívětivost. Parser editoru byl automatizovaně vygenerován pomocí LALR Parser Generatoru (LPG), jehož vstupem byla LALR gramatika jazyka CodAL implementovaná v Backus-Naurově formě. Pro uchování dat zpracovávaných zdrojových kódů byl vytvořen objektový model kompatibilní s objektovým modelem CDT (CDT DOM)[1]. Objektový model byl využit pro implementaci nástrojů známých z jiných editorů prostředí Eclipse, které slouží k hlubší analýze zdrojového kódu (popsáno v sekci 3.3.2).

[1] Vycházel jsem přitom především ze zdrojových kódů CDT a [18]

5.2 Etapy implementace

Realizaci navrhovaného editoru jsem rozdělil do několika na sebe navazujících implementačních etap. Výstupem každé naplánované etapy byl použitelný editor podporující stanovenou funkčnost. Tyto etapy jsem realizoval jako následující dílčí podprojekty:

1. Cílem prvního podprojektu bylo vytvořit základní editor, na kterém by bylo možné založit další rozšíření doplňků pro analýzu zdrojového kódu. Jeho požadavkem bylo, aby uměl barevně rozlišovat jednotlivé lexémy (*syntax highlighting*) a aby dokázal rozhodnout na úrovni lexikální a syntaktické analýzy, zda vstupní zdrojový kód patří nebo nepatří do jazyka CodAL. K dosažení tohoto cíle bylo nutné nejprve připravit vstup pro generátor parseru nesoucí informace o syntaktické stránce jazyka CodAL. Z těchto informací pak v druhém kroku automatizovaně vygenerovat parser, který by zajišťoval stanovené požadavky. Vstup generátoru parseru musel být v souladu s existující gramatikou jazyka CodAL. Musel být rovněž reprezentován v požadovaném formátu Backus-Naurovy formy.

2. Činností druhé etapy bylo rozšířit editor vytvořený v první etapě tak, aby uměl detekovat jednotlivé syntaktické a lexikální chyby, dokázal si zapamatovat jejich výskyt a vhodně poskytoval tuto informaci uživateli (zvýrazněním chybně napsaného zdrojového kódu). Pro dosažení takového výstupu jsem rozšířil gramatiku jazyka CodAL v Backus-Naurově formě. K jednotlivým pravidlům gramatiky jsem dopsal volání sémantických akcí, které jsou parserem zavolány při aplikování těchto pravidel. Jejich činností je vytvářen abstraktní syntaktický strom složený z patřičných instancí objektového modelu jazyka CodAL. Etapa se proto dále zaměřovala na tvorbu tohoto modelu, který rozšiřoval objektový model projektu CDT. S jeho implementací bylo možné vygenerovat nový parser, který uměl převádět zdrojový kód do struktury vhodné pro další analýzu.

3. V třetí etapě jsem se zaobíral sémantickou stránkou jazyka CodAL. Mojí činností bylo zpracovat získaný abstraktní syntaktický strom a vybrat z něho pouze podstatné uzly reprezentující uživateli logický celek (deklarace, volání, apod.). Tyto uzly bylo dále nutné mezi sebou provázat (konkrétní volání je vždy vztaženo ke konkrétní deklaraci). Pro tyto struktury jsem implementoval nové modely, které jsou využity nástroji editoru, jakými je například pohled Outline nebo refaktorizace proměnných.

Vzhledem k tomu, že se jazyk CodAL skládá z velkého množství přípustných konstrukcí, i jednotlivé etapy tvorby editoru zpracovávající tento jazyk představovaly rozsáhlý problém. V každém výše zmíněném bodu bylo nutné namodelovat celý jazyk na určité úrovni popisu. Před začátkem každé etapy bylo proto nutné zvážit přínosy a rizika daného postupu. Návrat o fázi zpět z důvodu špatně zvoleného postupu by znamenal velkou ztrátu času. Na práci jsem proto spolupracoval členem vědeckého týmu Lissom – Ing. Ondřejem Ilčíkem, který již měl zkušenosti s danou problematikou a pomáhal mi proto s některými náročnějšími úkoly. Podrobný popis jednotlivých etap je uveden v následujících třech kapitolách.

Kapitola 6

Popis gramatiky jazyka CodAL

Prvním implementačním milníkem bylo vytvoření editoru, který by uměl:

- rozpoznávat jednotlivé lexémy vstupního zdrojového kódu a barevně je rozlišovat dle jejich typu,
- testovat řazení lexémů zdrojového kódu podle gramatiky jazyka CodAL a rozhodnout, zda zdrojový kód patří do jazyka CodAL.

Pro zajištění této funkcionality jsem pomocí LALR Parser Generatoru vygeneroval parser. Tento parser uměl na základě stanovené gramatiky jazyka CodAL přebírat tokeny z existujího lexeru projektu CDT a aplikovat na ně pravidla této gramatiky. **Stěžejním bodem bodem této etapy bylo popsat gramatiku jazyka CodAL ve formátu vyžadovaným generátorem parseru.**

K dispozici jsem měl existující gramatiku jazyka CodAL používanou pro tvorbu kompilátoru jazyka CodAL. Tuto gramatiku jsem převedl do validního zápisu rozšířené varianty Backus-Naurovy formy, který je přijímán nástrojem LPG. Tento úkol jsem si rozdělil do tří kroků:

1. Prvním krokem bylo připravení struktury souborů, zajištění jejich provázání s generátorem parseru a stanovení cesty určující, kam se má parser vygenerovat. Jedním z požadavků bylo zajištění, aby nový nový parser vycházel z existujícího parseru projektu CDT. Výsledkem činnosti byl projekt obsahující prázdnou gramatiku, do které bylo možné implementovat pravidla.

2. Další krok úkolu představoval prozkoumat vstupní gramatiku v zápisu nástroje Bison [11] a vymyslet vhodný způsob převodu do požadovaného formátu v zápisu pro LPG. V rámci rozšíření projektu CDT jsem měl k dispozici gramatiku jazyka C standardu C99 ve validním formátu nástroje LPG. Na ni jsem navazoval.

3. Posledním krokem, který se částečně prolínal s bodem 2, bylo zajištění provázání generovaného parseru s existujícím lexikálním analyzátorem projektu CDT. Tento analyzátor byl použit z důvodu lexikální podobnosti jazyků CodAL a C. Cílem činnosti bylo zajistit, aby lexer projektu CDT správně zpracoval všechny lexémy jazyka CodAL a předal je v žádané formě syntaktickému analyzátoru jazyka CodAL.

6.1 Tvorba projektu

Generátor parseru LPG je binární soubor, který je možné ovládat z příkazové řádky. Prostřednictvím parametrů přebírá cestu k souboru gramatiky spolu s dodatečnými konfiguračními atributy, které stanovují způsob zpracování vstupu. Generátor umožňuje vygenerovat lexer i parser. V tomto projektu bude generován pouze parser. Pro zpracování lexémů byl využit existující lexer projektu CDT, protože umí zpracovávat většinu operátorů a literálů použitých v jazyce CodAL. Nová klíčová slova byla dodefinována prostřednictvím úprav v mapování tokenů lexeru na typy tokenů, s kterými pracuje parser (popsáno v 6.3.1). Další drobné odlišnosti byly realizovány změnou v existujícím lexeru.

6.1.1 Struktura základního vstupního souboru gramatiky

Obecný vstup pro LALR Parser Generator obsahuje dvě části. V první části se nachází sekce obsahující příkazy `%options`, pomocí kterých je definován způsob zpracování vstupní gramatiky. Je zde nastaven například typ výsledného parseru spolu s nastavením způsobu řešení případných konfliktů v pravidlech gramatiky (nastavení zpětného navracení). Dále je stanoven programovací jazyk, který je využit generátorem parseru pro vygenerování výsledných akcí a tabulek parseru. V tomto programovacím jazyce je parserem generován abstraktní syntaktický strom. Nakonec je nutné specifikovat jména výstupních souborů a balík (*package*), v kterém budou výstupní soubory uchovány.

Příklad 6.1

```
%options ast_directory=./ExprAst, automatic_ast=toplevel,
                                                var=nt, visitor=default
%options programming_language=java
%options package=expr1
%options template=dtParserTemplateD.g
%options import_terminals=ExprLexer.g
```

Na příkladu 6.1 je znázorněna ukázka základní hlavičky vzorové gramatiky z dokumentu Using LALR Parser Generator [15].

1. Na prvním řádku je stanovena cesta určující, kde budou uloženy generované třídy a rozhraní abstraktního syntaktického stromu a jakým způsobem se má zacházet s terminály[1].

2. Druhý řádek určuje, že bude použit programovací jazyk Java.

3. Na třetím řádku je stanoveno, že balík generovaných tříd bude pojmenován `expr1`.

4. Podstatná je definice šablony (*template*) `dtParserTemplateD.g` na čtvrtém řádku. Tento soubor obsahuje dodatečná nastavení spolu se zdrojovým kódem pro zjednodušení generování parseru. Šablony jsou volně dostupné součásti projektu LPG.

5. Poslední položka specifikuje lexer.

[1] V tomto projektu budou tyto třídy a rozhraní vytvářeny ručně rozšířením objektového modelu CDT.

S podrobnějším popisem možností konfigurace generování parseru je možné se seznámit v [14] a [15].

V druhé části gramatiky jsou uvedeny pravidla gramatiky v Backus-Naurově formě a terminály, které vznikají aplikováním těchto pravidel. Tyto bloky jsou logicky odděleny konstrukcemi `$Terminals` a `$Rules`. V příkladu 6.2 je znázorněno pokračování vzorové ukázky z dokumentu Using LALR Parser Generator [15].

Příklad 6.2

```
$Terminals
  IntegerLiteral
  PLUS ::= +
  MULTIPLY ::= *
  LPAREN ::= (
  RPAREN ::= )
$End

$Rules
  E ::= E + T | T
  T ::= T * F | F
  F ::= IntegerLiteral | ( E )
$End
```

Znak přepisu levé strany pravidla na pravou je reprezentován pomocí sekvence `::=`, oddělení pravidel pak pomocí symbolu `|`. Příklad 6.2 popisuje jednoduchý jazyk sestavený z výrazu, který v sobě může zahrnovat sčítání a násobení celých čísel s možným zanořováním pomocí závorek.

6.1.2 Výstup generovaný pomocí LPG

Nástroj LPG, který je spuštěn se zadaným vstupem, analyzuje vstupní gramatiku a poskytuje jako odezvu seznam statistik týkající se obsahu zadané gramatiky. Příkladem těchto informací jsou počty terminálů, neterminálů, akcí rozdělených do typů (redukce, uložení na zásobník, ...) a případných konfliktů mezi akcemi. Konflikty mohou nastat v případě nejednoznačných gramatik, kdy generátor nemůže deterministicky rozhodnout, zda má například provést redukci nebo ukládat na zásobník. Vyřešení konfliktů je možné provést zpětným navracením (definováno v části nastavení gramatiky). Generátor zkusí jednu variantu a v případě neúspěchu se navrátí do pozice, v které konflikt vznikl a pokusí se aplikovat jinou variantu.

V případě, že je generátoru parseru zadán programovací jazyk, je vygenerován parser. Výsledkem tohoto procesu jsou tři soubory:

- Prvním souborem je třída `Parser`. V této třídě jsou definovány metody volané při běhu parseru. Nachází se zde metoda `ruleAction`, která v závislosti na aktuálním číslu pravidla vyvolá metodu reprezentující akci spojenou s daným pravidlem gramatiky.
- Druhým souborem je rozhraní `Parsersym`. V něm jsou definovány konstanty reprezentující tokeny, které byly deklarovány v gramatice. Hodnoty tokenů jsou rovny celým číslům, pomocí kterých je možné se zaindexovat do pole názvů tokenů. Názvy vygenerovaných konstant jsou v základním nastavení shodné se jmény tokenů. V nastavení gramatiky je možné názvům konstant volitelně nastavit prefix.

- Posledním souborem je třída `Parserprs`. V této třídě jsou definovány veškeré proměnné určující aktuální stav parseru a překladové tabulky, které jsou parserem používány za běhu pro odvozování následujících stavů.

Vzhledem k tomu, že se jedná o automaticky generované soubory, není doporučeno tyto soubory ručně měnit. Tohoto pravidla jsem se držel a upravoval vždy pouze pravidla v gramatice.

6.1.3 Tvorba souborů popisující gramatiku jazyka CodAL

Styl popisu gramatiky pro nástroj LPG umožňuje modularizovat vstup do několika souborů. Pro tento účel je poskytnuta konstrukce `$Import`, pomocí které je možné začlenit dodatečné soubory. V gramatice jazyka CodAL je tato konstrukce využita pro import gramatiky jazyka C standardu C99 (příklad 6.3). Pro přehlednost označme tyto gramatiky názvem *LPG_Codal* a *LPG_C99*. Díky modularizaci jsem mohl v gramatice *LPG_Codal* využít existujících terminálních symbolů a některých pravidel definovaných v gramatice *LPG_C99*.

Příklad 6.3

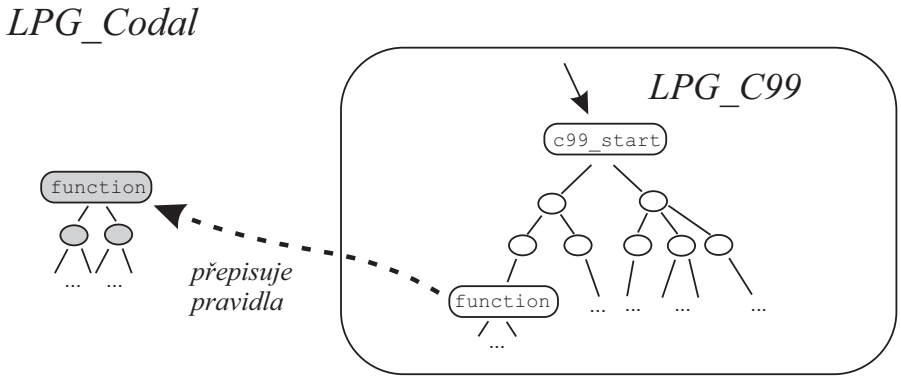
```
$Import
        C99Grammar.g
$End
```

Obdobným způsobem bylo možné přidat další soubory. Zejména výhodné se ukázalo vyčlenit tělo gramatiky *LPG_Codal* obsahující terminály a pravidla do speciálního souboru – `CodalGrammarExtensions.g`. Dodatečné změny jazyka bylo díky tomu možné provádět pouze v tomto souboru bez zásahu do celkového nastavení gramatiky.

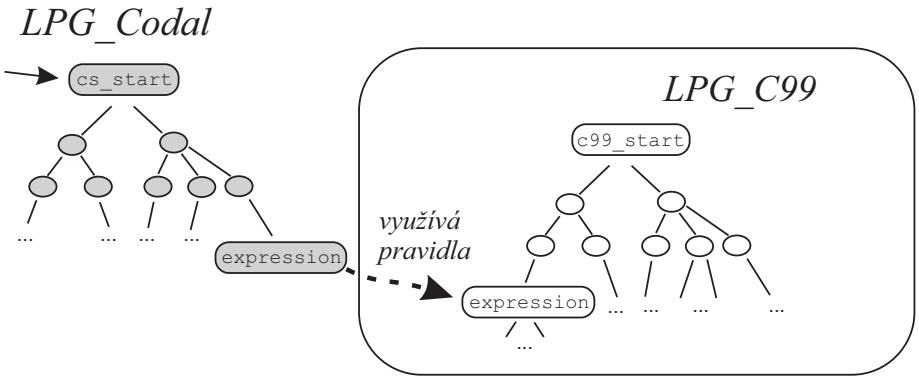
Z důvodu rozčlenění gramatiky do několika souborů bylo nutné explicitně stanovit výchozí neterminál, který představuje kořen, od kterého začne generátor analyzovat strom pravidel. V aktuálním pojetí rozšíření jazyka C ho bylo možné stanovit dvěma způsoby:

1. První možností bylo stanovit za výchozí bod počáteční neterminál začleněné gramatiky *LPG_C99*. Rozšíření této gramatiky by bylo implementováno v souboru `CodalGrammarExtension.g` prostřednictvím přetížení některých pravidel této gramatiky (obdobně jako je tomu u přetížení děděných metod u objektově orientovaných jazyků). Tím by byla zajištěna modifikace některých konstrukcí jazyka C nebo případné dodefinování konstrukcí nových. Tato možnost byla využita například v gramatice jazyka UPC (*Unified Parallel C*). Tento jazyk rozšiřuje jazyk C o některé konstrukce umožňující paralelní výpočty (například paralelní cyklus `for` apod.).

2. Druhou možností bylo vytvořit nový počáteční neterminál v gramatice *LPG_Codal* a z gramatiky *LPG_C99* využívat pouze některé větve stromu pravidel. V souboru `CodalGrammarExtension.g` by byly všechny základní pravidla a od určitých neterminálů by se odkazovalo na názvy neterminálů gramatiky *LPG_C99*. Tento způsob by byl implementačně složitější v případě, kdy bychom chtěli rozšířit jazyk C pouze o některé nové konstrukce, jako je tomu v jazyce UPC. Pokud by byl ovšem nový jazyk ze syntaktického hlediska výrazně odlišný oproti jazyku C (obsahoval by pouze některé základní prvky jazyka C), je tento způsob implementace výhodnější.



Obrázek 6.1: První způsob rozšíření gramatiky *LPG_C99*



Obrázek 6.2: Druhý způsob rozšíření gramatiky *LPG_C99*

Strukutra pravidel gramatiky jazyka CodAL se výrazně liší od struktury pravidel gramatiky jazyka C standardu C99. Pouze v některých svých větvích stromu pravidel gramatiky povoluje konstrukce obdobné syntaxi jazyka C. Na rozdíl od jazyka UPC[2] jsem proto zvolil druhou možnost rozšíření. Gramatice *LPG_Codal* jsem stanovil vlastní startovací neterminál spolu se základní strukturou pravidel, které byly doplněny o pravidla z gramatiky *LPG_C99*. V hlavním souboru gramatiky *LPG_Codal* je uveden úsek kódu odkazující se na nový startovací neterminál `cs_start` (příklad 6.4). Na tento neterminál v souboru `CodalGrammarExtension.g` navazují pravidla gramatiky *LPG_Codal*.

Příklad 6.4

```
$Start
    cs_start
$End
```

V poslední řadě bylo nutné stanovit šablonu, která obsahuje nastavení parseru a implementaci pomocných úseků kódu v jazyce Java, které jsou generátorem vkládány na patřičná místa třídy `Parser`. Díky tomu bylo později možné k pravidlům gramatiky *LPG_Codal*

[2]Popis implementace gramatiky pro kazyk UPC je přiblížen v [9]

vkládat volání metod v jazyce Java, které zajišťují vykonávání dodatečných akcí při parsování vstupu (např. práce se zásobníkem). Tyto volání byly generátorem vloženy na zvolená místa třídy `Parser` určené šablonou. Jako šablona byl využit existující template `LRParserTemplate` uplatněný rovněž v rozšiřované gramatice *LPG_C99*.

6.1.4 Integrace gramatiky se strukturou projektu editoru

Umístění gramatiky *LPG_Codal* do projektové struktury implementující editor jazyka CodAL vyžadovalo mimo jiné stáhnutí spustitelného souboru generátoru parseru a přidání tohoto programu do projektu. Vstupní gramatiku v BNF lze překládat z příkazové řádky, avšak pro potřeby snadné práce vytvořil člen vědecké skupiny Lissom – Ing. Ilčík skript nástroje Ant pro automatizovaný překlad. Skript obsahuje potřebné cesty vstupů a výstupů spolu s dodatečnými konfiguracemi. Překlad je díky tomu možné spouštět automatizovaně pomocí nástroje Eclipse. Skript se přizpůsobí operačnímu systému, na kterém je vývoj editoru aktuálně prováděn a vyvolá správný binární soubor generátoru parseru. Není přitom nutné provádět žádná dodatečná nastavení.

Vytvořením struktury projektu bylo zajištěno systematické členění jednotlivých částí projektu. Byly založeny soubory a jmenné prostory pro další rozšiřování. Projekt byl připraven pro implementaci pravidel gramatiky jazyka CodAL v zápisu Backus-Naurovy formy.

6.2 Převod gramatiky do formátu pro LPG

V dalším kroku tvorby vstupu pro generátor parseru jsem se pohyboval pouze v souboru `CodalGrammarExtension.g`. Cílem bylo vyjít ze startovacího neterminálu gramatiky *LPG_Codal* pojmenovaném jako `cs_start`. Na tento neterminál musela být navázána taková pravidla, aby z něho bylo možné pomocí těchto pravidel vygenerovat terminály řazené do posloupností odpovídající syntakticky validním zdrojovým kódům jazyka CodAL. Formát pravidel musel odpovídat formátu znázorněném ve vzorovém příkladu 6.2.

K dispozici jsem měl soubor `codalc.ypp` popisující existující gramatiku jazyka CodAL v zápisu nástroje Bison (označme názvem *Bison_Codal*). Tento nástroj přijímá bezkontextové gramatiky typu LALR stejně tak jako nástroj LPG. Díky tomu jsem mohl předpokládat, že pravidla odpovídají požadovanému tvaru a mohu z nich vycházet. Jedinou nevýhodou bylo, že syntaxe zápisu nástroje Bison neodpovídala požadované syntaxi v Backus-Naurově formě nástroje LPG. Úkolem této etapy proto bylo najít spojitost mezi těmito dvěma zápisy a provést převod.

Gramatika *Bison_Codal* slouží jako vstup pro generátor parseru překladače jazyka CodAL. Jedná se o referenční gramatiku jazyka CodAL. Pokud je vývojáři vědecké skupiny Lissom rozhodnuto provést změnu v jazyce CodAL, je jako první modifikována gramatika *Bison_Codal*. Jedním z požadavků této práce bylo, aby případnou změnu jazyka CodAL bylo možné snadno aplikovat i v editoru tohoto jazyka. Rozhodl jsem se proto vytvořit skript v jazyce Python, který umí automatizovaně provést převod mezi gramatikami *Bison_Codal* a *LPG_Codal*.

6.2.1 Převod základní struktury pravidel

Syntaxe nástroje Bison je podobná syntaxi vstupního souboru nástroje LPG. Struktura zápisu pravidel se podobá Backus-Naurově formě. Odlišnosti lze nalézt především u symbolů

pro členění pravidel. Ukázka implementace pravidla v zápisu nástroje Bison je znázorněna na příkladu 6.5.

Příklad 6.5

```
parameter_list
  :  parameter_list ',' id
       // sémantické akce v jazyce C
  |  id
       // sémantické akce v jazyce C
  ;
```

Jednotlivá pravidla jsou oddělovány středníkem. Pro oddělení pravé a levé strany pravidla je použitý symbol `:`, dále pak symbol `|`. Na příkladu 6.5 je možné vidět, že s pravými stranami pravidla jsou asociovány dodatečné sémantické akce. Tyto akce jsou parserem vyvolány při aplikaci pravidla, ke kterému se váží. Jedná se o kód v jazyce C, který zajišťuje tvorbu abstraktního syntaktického stromu, modifikaci tabulek deklarací apod. Tyto akce nebylo možné využít z důvodu toho, že pracují s objektovým modelem, který neodpovídá objektovému modelu CDT. Při konverzi byly proto sémantické akce zahazovány a v dalších etapách tvorby editoru bylo nutné s touto skutečností počítat.

Na příkladu 6.6 je znázorněno převedené pravidlo z příkladu 6.5 do formátu určeném pro nástroj LPG.

Příklad 6.6

```
parameter_list
  ::= parameter_list ',' id
  |   id
```

6.2.2 Zpracování epsilon pravidel

Speciální případem pravdila je *ϵ-pravidlo*. V zápisu nástroje Bison je pravá strana takového pravidla ponechána prázdná. V gramatice nástroje LPG musí být toto pravidlo označené pomocí konstrukce `$empty`. Na příkladu 6.7 je ukázána souvislost mezi implementacemi *ϵ-pravidla* v nástroji Bison a LPG.

Příklad 6.7

```
modifier                          modifier
  : SHARED                          ::= 'shared'
  |                                 |   $empty
  ;
```

V příkladu 6.7 je rovněž ukázán rozdíl v popisu klíčových slov, který je podrobněji rozebrán v sekci 6.3.1 týkající se mapování terminálů.

6.2.3 Mapování neterminálů

V sekci 6.1 popisující tvorbu projektu bylo nastíněno, že gramatika jazyka CodAL bude rozšiřovat gramatiku jazyka C (označenou jako *LPG_C99*). Z důvodu možných kolizí názvů neterminálů gramatiky *Bison_Codal* a *LPG_C99* jsem zajistil, aby skript automaticky

přidal vybraným neterminálům gramatiky *Bison_Codal* prefix `cs_` (*Codasip Studio*). Například počáteční neterminál `start` byl namapován na existující neterminál `cs_start`, který byl explicitně označen jako výchozí.

Nepřejmenovával jsem neterminály použité v pravidlech, které popisující konstrukce jazyka C[3]. V gramatice *Bison_Codal* již tyto neterminály mají vlastní prefix `c_`. Pravidla, jejichž levá strana obsahuje tyto neterminály, jsou do jisté míry[4] stejné pravidlům rozšiřované gramatiky *LPG_C99*. Skript umí řešit mapování těchto neterminálů mezi gramatikami *Bison_Codal* a *LPG_Codal* dvěma způsoby:

1. První varianta zreprodukuje pravidla obsahující tyto neterminály do nové gramatiky *LPG_Codal*. Neterminálům obsahující prefix `c_` je tento prefix zachován a prefix `cs_` nepřidáván. Pravidla rozšiřované gramatiky *LPG_C99* zůstanou nevyužitá a neterminály nacházející se na levých stranách těchto pravidel se stanou v rámci gramatiky *LPG_Codal* nedostupnými.

2. Druhá varianta se snaží nalézt spojitosti mezi pravidly gramatiky *Bison_Codal* obsahující na své levé straně neterminály s prefixem `c_` a pravidly gramatiky *LPG_C99*. Pro tento účel jsem vytvořil slovník, v kterém je možné pro neterminály gramatiky *Bison_Codal* nalézt odpovídající neterminály gramatiky *LPG_C99*. Skript pak nejprve převede pravidla, jejichž pravá strana neobsahuje neterminál s prefixem `c_`. V souvislosti s tím projde neterminály pravých stran, které obsahují názvy s prefixem `c_` a pokusí se je nahradit ekvivalentním neterminálem gramatiky *LPG_C99*. Pokud k nim ve slovníku neexistuje asociovaný neterminál, dogeneruje danou větev do místa, kde již ve slovníku existují všechny potřebné ekvivalenty (případně do terminálů).

Druhá varianta generuje jednodušší verzi gramatiky *LPG_Codal*. Díky tomu je možné vynechat celou větev gramatiky *Bison_Codal* a využít existující implementaci gramatiky *LPG_C99*. Při podrobnějším zkoumání syntaxe jazyka C popsaného gramatikou *Bison_Codal* jsem ovšem zjistil, že některá pravidla nejsou úplně ekvivalentní s asociovanými pravidly gramatiky *LPG_C99*[5]. Tento problém by bylo možné vyřešit pomocí dodatečného přetěžování odlišných pravidel. V souvislosti s možnými budoucími změnami jsem se ovšem rozhodl použít první variantu konverze gramatiky *Bison_Codal*. Výsledná gramatika *LPG_Codal* je čitelnější, přehlednější a tím pádem i lépe modifikovatelná při budoucích požadavcích na změnu.

Pravidla gramatiky *LPG_C99* jsou provázána s metodami provádějící sémantické akce, které jsou napojeny na objektový model CDT. Druhou variantu převodu pravidel gramatiky *Bison_Codal* jsem proto ponechal pro testovací účely objektového modelu. Mohl jsem například ihned zpracovávat výrazy bez jejich nutné implementace, které jsou v modifikované formě součástí mnoha konstrukcí jazyka CodAL.

6.3 Propojení s lexikálním analyzátorem projektu CDT

V posledním kroku tvorby vstupu pro generátor parseru jsem zajistil, aby parser spolupracoval s existujícím lexikálním analyzátorem projektu CDT. Musel jsem přemapovat terminály gramatiky *Bison_Codal* na terminály gramatiky *LPG_C99*, pro kterou je lexer

[3] Tyto konstrukce jsou v jazyce CodAL přípustné například v bloku `semantics` popisující chování událostí (viz 2.3.6).

[4] Přesný výčet odlišností je možné nalézt v sekci Semantics 2.3.6 nebo v dokumentaci jazyka CodAL.

[5] To plyne především z jistých modifikací popsaných v sekci 2.3.6.

projektu CDT primárně určen. Dále jsem musel ověřit, že použitý lexer bude ve zdrojovém kódu uživatele rozpoznávat veškeré povolené lexémy jazyka CodAL. To znamená, že jsem musel ověřit, zda každý token, který je použitý v gramatice *Bison_Codal*, má ekvivalentní zastoupení v gramatice *LPG_C99*.

Pro analýzu tohoto problému jsem vytvořil další slovník, který ve svých položkách obsahoval existující terminály gramatiky *LPG_C99*. Na tyto položky je možné se odkazovat přes názvy terminálů gramatiky *Bison_Codal*. Výsledný skript pro převod mezi gramatikami *Bison_Codal* a *LPG_Codal* ověřuje s využitím tohoto slovníku existenci každého aktuálně dosaženého terminálu gramatiky *Bison_Codal* a nahradí ho odpovídajícím terminálem gramatiky *LPG_C99*. V případě, že skript nenalezne odpovídající položku, vypíše danou skutečnost chybovým hlášením. Tyto situace jsem musel řešit ručně.

6.3.1 Mapování terminálů

Seznam všech terminálů gramatiky *Bison_Codal* jsem získal ze souboru `codalc.l`, který obsahuje dodatečné lexikální informace této gramatiky. Tento soubor slouží jako vstup pro nástroj Flex [10], který generuje lexikální analyzátor. Seznam všech terminálů gramatiky *LPG_C99* byl k dispozici například v sekci `$Terminals`, která provádí mapování terminálů na tokeny. Na následujících příkladech je pro oba typy zápisů znázorněna ukázka vzorového pravidla popisujícího deklaraci a současnou inicializaci proměnné. Na pravé straně pravidel se pro ukázku vyskytují různé typy terminálů.

Příklad 6.8

```
declaration
  : USE ID '=' CONST ';'
  ;
```

Na příkladu 6.8 máme znázorněno pravidlo v zápisu nástroje Bison, které přepisuje neterminál `declaration` na posloupnost terminálů[6]. Terminály jsou v zápisu nástroje Bison reprezentovány tokeny. Jejich význam je nutné zjistit z pomocného vstupního souboru nástroje Bison popisující lexikální stránku jazyka. Takový soubor by mohl obsahovat například následující úsek kódu z příkladu 6.9.

Příklad 6.9

```
"use"                                RETURN(USE);
[1-9][0-9]*                          YYLVALTEXT(); RETURN(CONST);
([a-zA-Z_])([a-zA-Z_]|[0-9])*        YYLVALTEXT(); RETURN(ID);
"="                                  RETURN('=');
";"                                  RETURN(';');
```

Každý token použitý v pravidlech gramatiky zápisu Bison reprezentuje nějaký řetězec nebo množinou řetězců reprezentovaných regulárním výrazem. Regulární množinu zde tvoří token `CONST`, který reprezentuje celá čísla a token `ID`, který popisuje identifikátory. Množina `ID` je explicitně zmenšena o klíčová slova, jakým je například klíčové slovo `use`.

[6] Příklad 6.8 (stejně tak i příklad 6.10) byl pro jednoduchost znázorněn bez sémantických pravidel. Toto pravidlo by bylo v praxi rozděleno na tři samostatná pravidla, kde identifikátor a konstanta by byly reprezentovány vlastním neterminálem s pomocným pravidlem, které by bylo asociováno se sémantickou akcí.

Příklad 6.10

```
cs_declaration
  ::= 'use' 'identifier_token' '=' 'integer' ';'
```

Na příkladu 6.10 je znázorněno převedené pravidlo do vstupního formátu LALR Parser Generatoru. Všechny tokeny jsou zde na rozdíl od zápisu nástroje Bison obaleny jednoduchými uvozovkami. Tokeny `identifier_token` a `integer` jsou zděděny z gramatiky *LPG_C99*. V případě, že se ve zdrojovém kódu vyskytne identifikátor nebo celé číslo, lexikální analyzátor je zpracuje a parseru předá tokeny reprezentující tyto lexémy. Problém nastává s tokenem `use`, který lexikální analyzátor neumí rozpoznat jako klíčové slovo, a tudíž ani neumí vytvořit odpovídající token.

6.3.2 Tvorba nových tokenů

Ne pro všechny tokeny gramatiky *Bison_Codal* existuje ekvivalentní token v gramatice *LPG_C99*. Jedná se například o nová klíčová slova, kterých je v jazyce CodAL velké množství. V takových případech je nutné nové tokeny nejprve nadefinovat, aby s nimi parser uměl pracovat. Dále je nutné provést úpravu v lexikální analýze, aby bylo zajištěno, že v případě výskytu posloupnosti symbolů odpovídající nově definovanému tokenu, bude parseru tento token předán.

Tokeny je v gramatice zápisu nástroje LPG možné nadefinovat do bloku `$Terminals`.

Příklad 6.11

```
$Terminals
  use
$End
```

Nově definovaný token `use` je od této chvíle možné používat obdobným způsobem jako tokeny `identifier_token` a `integer` v příkladu 6.10.

Dále je nutné provést úpravu lexikální analýzy. Vzhledem k tomu, že klíčová slova jsou vytvořena vyčleněním některých identifikátorů, není nutné měnit existující lexer. Stačí přidat mezi lexer a parser dodatečné testování identifikátorů. V případě, že se aktuální identifikátor bude rovnat novému klíčovému slovu, bude parseru předán token reprezentující odpovídající klíčové slovo.

Pro tuto možnost je v CDT dostupné rozhraní `IDOMTokenMap`, které deklaruje metodu `mapKind`. Úkolem této metody je mapovat tokeny získané lexerem na tokeny obsažené v parseru. Tímto způsobem jsem dodefinoval přibližně 60 nových klíčových slov jazyka CodAL.

6.3.3 Chybějící tokeny

V případě, kdy nový token nereprezentuje klíčové slovo ale například operátor složený z posloupnosti znaků, které lexer neumí rozpoznat, je nutné provést změnu přímo v lexeru. Taková situace v jazyce CodAL nastala pouze jednou – v případě posloupnosti dvou teček používané v rozsazích (např. `bit[0..31]`). Tento problém byl vyřešen dočasným řešením. Parser místo tokenu reprezentující dvojtečku očekává dva tokeny reprezentující jednu tečku, kterou lexer umí rozpoznat. Problém s možnými bílými znaky mezi dvěma tečkami je přenechán dodatečné statické analýze zdrojového kódu.

6.4 Využití gramatiky jazyka CodAL

Z vytvořené gramatiky v Backus Naurově formě jsem mohl vygenerovat parser určený pro základní verzi editoru.

6.4.1 Syntax highliting v editoru CodAL

Využití existujícího lexikálního analyzátoru a existujících deklarací tokenů bylo výhodné i z důvodu toho, že nový editor díky tomu ihned uměl automaticky zvýrazňovat veškeré lexémy reprezentované tokeny z gramatiky *LPG_C99*. Nová klíčová slova jazyka CodAL jsem explicitně dodefinoval do výčtu `CodalKeyword`.

Výsledný syntax highlighting funguje stejně jako v editorech projektu CDT. Uživatel může v nastavení měnit styly zvýrazňování, které se mu automaticky ukládají v daném profilu workspace. V základním nastavení je styl shodný s stylem editoru jazyka C.

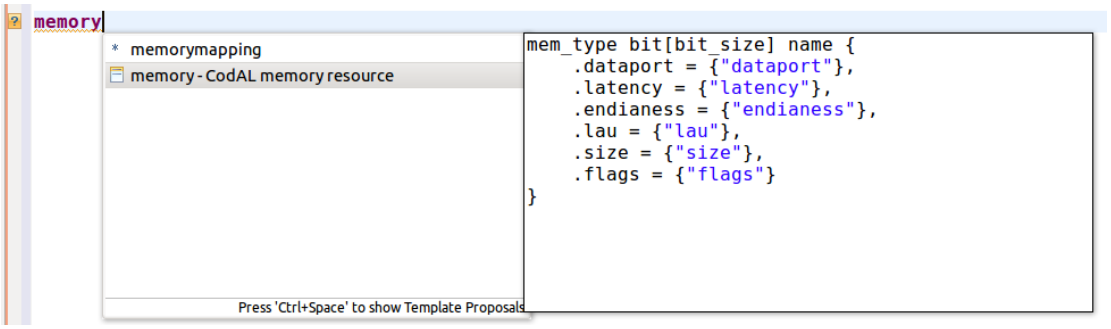
```
/**is zero-extended to the left and combined in bitwise logical AND operation with*/
element op_andi  { assembler { "ANDI" };  binary { OP_ANDI:6 }; return {OP_ANDI; }; }
/**is zero-extended to the left and combined in bitwise logical OR operation with*/
element op_ori   { assembler { "ORI" };   binary { OP_ORI:6 }; return {OP_ORI; }; }
/**is zero-extended to the left and combined in bitwise logical Exclusive OR operation with*/
element op_xori  { assembler { "XORI" };  binary { OP_XORI:6 }; return {OP_XORI; }; }

set op_arithm_imm_unsigned = op_andi, op_ori, op_xori;
```

Obrázek 6.3: Zvýraňování syntaxe ve zdrojovém kódu jazyka CodAL

6.4.2 Šablony

Dalším nástrojem, který mohl být použit již v první verzi editoru, byly šablony (*templates*). V nastavení editoru bylo nutné ručně předpřipravit jednotlivé konstrukce jazyka CodAL, jakými jsou například zdroje a operace. Programátorům vědeckého týmu Lissom je díky šablonám ušetřena spousta času při deklarování jednotlivých bloků (kterých zdrojový kód obsahuje mnoho).



Obrázek 6.4: Content assist nabízející šablonu pro paměť

Kapitola 7

Tvorba objektového modelu jazyka CodAL

Druhým implementačním milníkem bylo rozšíření editoru implementovaného v předchozí etapě tak, aby:

- dokázal nalézt syntaktické chyby a zapamatoval si jejich přesné pozice,
- při nálezu syntaktické chyby se uměl zotavit a provést analýzu ve zbytku zdrojového souboru,
- zvýraznil uživateli nalezené chyby podvlnkováním chybných konstrukcí ve zdrojovém kódu.

Pro tyto požadavky bylo nutné zajistit, aby parser editoru převáděl data vstupního souboru do vhodné datové struktury použitelné pro další analýzu – *abstraktního syntaktického stromu* (AST). Dále bylo nutné zajistit, aby jednotlivé uzly výsledného AST byly rozlišeny v závislosti na tom, jaký typ tokenu nebo neterminálu reprezentují. Díky tomu je možné snadno nalézt všechny uzly, které reprezentují syntaktické chyby a zjistit jejich přesnou pozici.

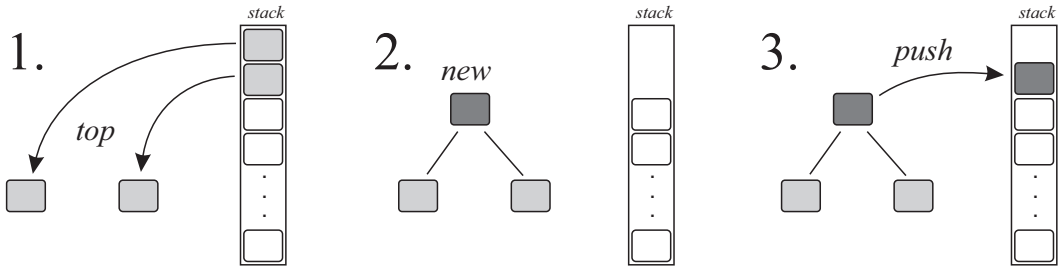
Hlavní činností této etapy bylo vytvořit model pro abstraktní syntaktický strom, jehož součástí jsou rovněž metody pro tvorbu AST odpovídající tomuto modelu. Prostřednictvím těchto metod jsem zajistil propojení mezi parserem a objektovým modelem AST. Podstatným požadavkem na objektový model bylo, aby odpovídal objektovému modelu CDT a mohl být v budoucnu využit existujícími nástroji editorů CDT.

7.1 Proces tvorby AST

Tvorba abstraktního syntaktického stromu probíhá v opačném směru expandování pravidel gramatiky – zdola nahoru. Nejprve jsou vytvořeny listové uzly reprezentující tokeny (např. identifikátory nebo literály). Pro ně jsou postupně v závislosti na pravidlech gramatiky vytvářeny rodičovské uzly (např. příkazy nebo deklarace). Celý proces tvorby probíhá, dokud není vytvořen kořenový uzel reprezentující celý program.

Pro uchování aktuálně zpracovávaných uzlů je využíván zásobník. Manipulace s ním probíhá ve třech opakovaných krocích:

1. Vyber uzly očekávané z vrcholu zásobníku.
2. Vytvoř nový uzel a jako jeho potomky nastav vybrané uzly ze zásobníku.
3. Ulož nově vytvořený prvek na zásobník.



Obrázek 7.1: Jeden krok tvorby abstraktního syntaktického stromu

To, kolik uzlů se má aktuálně vyjmout ze zásobníku a jaký typ uzlu se bude vytvářet, je stanoveno v konkrétní *consume funkci*, která implementuje kroky 1 - 3. Consume funkce jsou vybírány podle toho, jaké pravidlo gramatiky je parserem aktuálně aplikováno. Jejich volání je implementováno pod jednotlivá pravidla jako dodatečná sémantická akce.

Příklad 7.1

```
cs_id
  ::= 'identifier_token'
    /. $Build  consumeIdentifierName();  $EndBuild ./
```

Na příkladu 7.1 je ukázáno volání funkce `consumeIdentifierName` při aplikování pravidla pro zpracování identifikátoru. Dodatečná sémantická akce je obalena mezi konstrukce „`/. $Build`“ a „`$EndBuild ./`“. Formát označení těchto bloků je definován v šabloně gramatiky. Generátor parseru si před začátkem tvorby parseru tuto informaci zjistí a podle nastavení šablony tyto úseky kódu vloží na správná místa zdrojových kódů parseru. Tím je zajištěno provázání parseru s metodami pro tvorbu abstraktního syntaktického stromu.

Veškeré nové consume funkce jsem implementoval do třídy `CodalParserAction`. Tato třída je odvozena ze třídy `C99BuildASTParserAction`, v které jsou implementovány consume funkce objektového modelu jazyka C. Díky tomu jsem mohl v mnoha případech využít existující implementace modelu AST.

7.1.1 Vyjmutí položek ze zásobníku

Zásobník je reprezentovaný kontejnerem `ScopedStack` projektu CDT. Součástí tohoto kontejneru jsou základní známé metody pro práci se zásobníkem, jakou je například metoda `pop`. Tuto metodu jsem použil v případě, kdy jsem na vrcholu zásobníku očekával jednu konkrétní položku.

Hlavní výhodou kontejneru `ScopeStack` je, že zahrnuje metody pro práci se skupinou položek obsažených ve vymezeném prostoru (*scope*). V gramatice totiž existují pravidla, kdy

je některý neterminál expandován na (teoreticky nekonečně dlouhý) seznam neterminálů nebo terminálů stejného typu. Příkladem může být například seznam deklarátorů registru:

Příklad 7.2

```
register bit[32] r1, r2, r3, r4 /* ... */ ;
```

V gramatice zápisu nástroje Bison by tato konstrukce mohla být popsána například následujícím způsobem[1]:

Příklad 7.3

```
register_decl
        ::= <openscope-ast> register_decl_list
                /. $Build  consumeRegisterDeclarators();  $EndBuild  ./

register_decl_list
        ::= register_decl_id ',' register_decl_list
        |   register_decl_id

register_decl_id
        ::= id
                /. $Build  consumeCodalDeclarator();  $EndBuild  ./
```

Pomocí zápisu `<openscope-ast>` je vymezen prostor ohraničující seznam položek (v tomto příkladě deklarátorů). Ve funkci `consumeRegisterDeclarators` je pak místo metody `pop` použita metoda `closeScope`, která vrátí z vrcholu zásobníku seznam všech položek vztažených k danému seznamu. Bez této možnosti by seznam musel být reprezentován rekurzivně zanořenými položkami v binárním stromu, což by bylo méně praktické pro další práci s AST.

7.1.2 Tvorba nového uzlu AST

Po vyjmutí položek ze zásobníku je nutné vytvořit nový uzel, který se stane rodičem těchto položek. Potomci jsou mu předány jako parametry jeho konstruktoru nebo přes metodu `set`, případně `add` u seznamů. V těchto metodách musí být zajištěno obousměrně provázání rodiče a potomka.

Pro vyvolávání konstruktorů nových uzlů jsem použil styl abstrakce z návrhu CDT. Nové uzly jsou vytvářeny prostřednictvím rozhraní `ICodalNodeFactory`. Implementaci tohoto rozhraní zajišťuje třída `CodalASTNodeFactory`. Metody pro tvorbu uzlů stejných typů bývají zpravidla přetížené a lze je volat s různými typy parametrů v závislosti na tom, jaké uzly jsou aktuálně na vrcholu zásobníku.

Rozhraní `ICodalNodeFactory` rozšiřuje rozhraní `ICNodeFactory` z knihovny CDT (a tedy nepřímo i základní rozhraní `INodeFactory`). Díky tomu jsem mohl znovu implementovat některé metody těchto rodičovských rozhraní a přinutit tím existující consume funkce projektu CDT vytvářet mé vlastní verze již existujících typů uzlů[2].

[1] Pro jednoduchost uvádím pouze pravidla vztahující se k seznamu deklarátorů.

[2] Toho jsem využil například pro tvorbu vlastních uzlů reprezentujících identifikátory, které používají vlastní *bindings*. Podrobněji v kapitole 8.

7.1.3 Vložení nové položky na zásobník

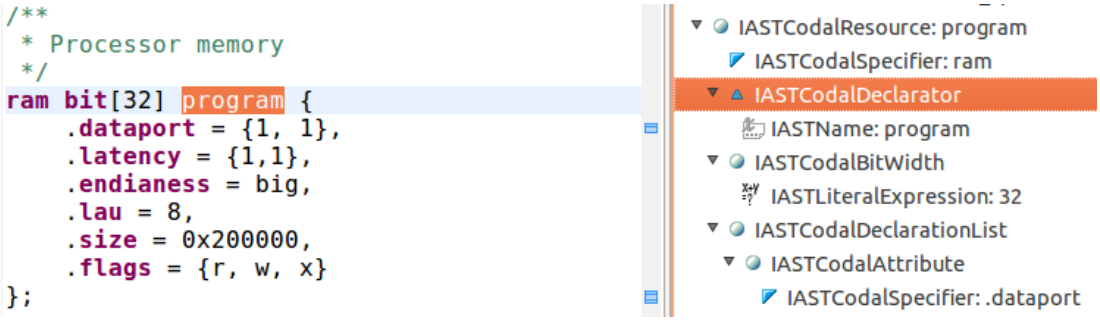
Třetí akcí, která musí být provedena v rámci consume funkcí, je uložení nově vytvořeného uzlu na zásobník. Díky tomu může být nový uzel spolu s jeho potomky dále zakomponován do vytvářeného AST. Pro činnost uložení položky na zásobník poskytuje kontejner metodu `push`. Spolu s voláním této metody jsou prováděny některé dodatečné činnosti, kterými jsou například nastavení počtu znaků a pozice ve zdrojovém kódu (metoda `setLengthAndOffset`).

7.1.4 Testování korektnosti tvorby AST

Celá činnost tvorby AST je prováděna, dokud nejsou zpracovány všechny uzly a na zásobník není vložen kořenový uzel – *translation unit*. Činnost tvorby AST je velice důležitá. Bez správně vytvořeného AST není možné provést analýzu zdrojového kódu a uživateli tedy nemůže být poskytnuta žádná informace o jejím výsledku. Během vytváření AST proto nesmí pro žádný (i nevalidní) vstup dojít k chybě vytváření AST.

Nejčastěji jsem se setkával s chybou, při které volaná consume funkce očekávala na vrcholu zásobníku jiné uzly, než které se tam ve skutečnosti nacházely. Příčinou bývala špatná implementace dané consume funkce nebo špatné provázání pravidel gramatiky s consume funkcemi. Tyto chyby jsem se snažil maximálně eliminovat.

Pro účely testování jsem si vytvořil vstupní soubor obsahující všechny typy povolených konstrukcí jazyka CodAL. Dodatečně jsem navíc do tohoto souboru zanášel syntaktické chyby, díky kterým docházelo častěji k chybě vytváření AST. Pokud se AST podařilo vytvořit, ověřil jsem dodatečně jeho správnost pomocí pohledu *DOM AST Viewer*. Tento pohled umí vizualizovat AST aktuálně otevřeného zdrojového kódu. Pomocí tohoto pohledu se mi podařily odhalit zbylé chyby, které přímo nevyplynuly z procesu tvorby AST.



Obrázek 7.2: Ukázka použití pohledu DOM AST Viewer

7.2 Objektový model pro AST

Aby parser mohl sestavit abstraktní syntaktický strom, bylo nutné vytvořit objektový model popisující tuto strukturu. Objektový model AST poskytuje odpovědi na dvě otázky:

1. ***Jaké typy uzlů může AST obsahovat?*** Tato informace je modelována prostřednictvím rozhraní. V závislosti na tom, jaké rozhraní uzel implementuje, je možné

při práci s AST zjišťovat, co daný uzel reprezentuje (zda se jedná o deklaraci, příkaz, identifikátor, apod.)[3]. Tato informace je využita při dalším zpracování AST.

2. ***Jaká je hierarchie mezi stanovenými typy uzlů AST?*** Jinými slovy bylo nutné specifikovat, jaké potomky může každý typ uzlu obsahovat (například deklarace může obsahovat specifikátor, seznam deklarátorů a případně i tělo inicializující danou deklaraci). Toho je dosaženo pomocí metod *get* a *set* deklarovaných v jednotlivých rozhraních. Tyto metody umožňují pracovat s potomky uzlů.

V projektu CDT je implementován objektový model určený pro AST jazyka C. Z tohoto modelu jsem vycházel. Objektový model jazyka CodAL je odvozen z objektového modelu jazyka C. Při propojování pravidel gramatiky jazyka CodAL s consume funkcemi jsem proto mohl využít tři způsoby implementace:

- První možností bylo implementovat vlastní consume funkci, která při zavolání vytvoří instanci vlastního typu uzlu. Tento typ uzlu jsem musel přidat do objektovém modelu jazyka CodAL.
- Druhou možností bylo implementovat vlastní consume funkci, která při zavolání vytvoří instanci existujícího typu uzlu. Tento typ uzlu jsem získal ze zděděného objektového modelu jazyka C projektu CDT.
- Implementačně nejjednodušší možností bylo přímo využít existující consume funkci projektu CDT. Tato varianta mohla být použita pouze pro konstrukce jazyka CodAL odpovídající konstrukcím jazyka C.

Primárně jsem se snažil využívat existující consume funkce pracující s existujícími typy uzlů AST. V některých případech jsem si musel naimplementovat vlastní consume funkci, která inicializovala existující typ uzlu AST vlastním způsobem (např. očekávala na zásobníku potomky uzlu v opačném pořadí). Pro specifické deklarace jazyka CodAL jsem musel vytvořit vlastní typy uzlů.

7.2.1 Objektový model projektu CDT

Objetový model projektu CDT obsahuje rozhraní popisující model na abstraktní úrovni. Je rozčleněn do několika vrstev:

1. Základní vrstvu tvoří ***obecný objektový model AST***. Tento model obsahuje rozhraní, jejichž cílem je reprezentovat základní konstrukce používané v libovolných programovacích jazycích (výrazy, literály, identifikátory, apod.).
2. Druhou vrstvu tvoří ***objektové modely pro AST konkrétních programovacích jazyků*** – konkrétně objektové modely pro AST jazyků C a C++. Tyto modely jsou složeny z rozhraní modelující specifické konstrukce daného jazyka (jakými jsou například struktury v jazyka C).

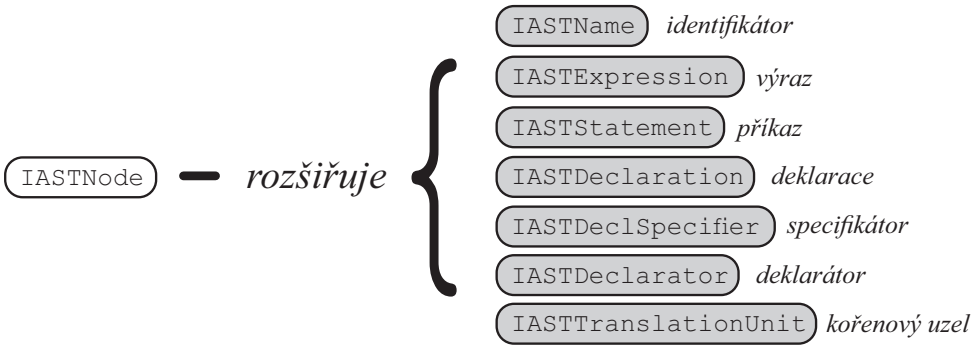
Rozhraní objektového modelu CDT jsou v každém jazyce implementovány vlastními třídami. Nástroje editorů CDT pak pracují s uzly AST obecně prostřednictvím metod rozhraní objektového modelu AST. V následujícím textu jsou vyjmenovány rozhraní základních typů uzlů používaných v projektu CDT.

[3] V dalším textu jsou proto rozhraní objektového modelu AST nazývána jako *typy* uzlů.

Rozhraní IASTNode

Rozhraní `IASTNode` představuje nejobecnější pohled na uzel v AST. Všechny další rozhraní popisující typy uzlů AST jsou z tohoto typu uzlu odvozeny. Rozhraní `IASTNode` deklaruje základní metody pro práci uzlem. Obsahuje metody pro nastavení a získání referencí na potomky nebo rodiče, pomocí kterých je možné procházet stromem. Dále obsahuje metodu `getFileLocation` pro získání pozice odpovídající konstrukce ve zdrojovém souboru. Pro vytvoření kopie dané instance uzlu deklaruje metodu `copy`.

Důležitou metodou rozhraní `IASTNode` je metoda `accept`. Tato metoda slouží jako filtr. V parametru je jí předán objekt *visitor*, který obsahuje seznam hledaných typů uzlů a prázdný seznam pro ukládání nalezených výsledků. Uzel, jehož metoda `accept` je vyvolána, ověří, zda splňuje kritéria hledaných uzlů. Pokud ano, přidá se do seznamu nalezených výsledků. Metoda `accept` je dále volána u jeho potomků. Tímto způsobem je možné získat například všechny identifikátory v daném prostoru (*scope*), což je využito například u přejmenování proměnných[4]. Implementace metod *accept* je mimo jiné vyžadována pohledem DOM AST Viewer, který je použitý pro testování.



Obrázek 7.3: Důležitá rozhraní rozšiřující základní typ uzlu AST

Rozhraní IASTName

Rozhraní `IASTName` přímo rozšiřuje základní typ uzlu `IASTNode`. Jeho cílem je reprezentovat identifikátory. Obsahuje proto metody pro práci s řetězcem znaků, který představuje název daného identifikátoru.

Další důležitou součástí rozhraní `IASTName` jsou metody, jejichž cílem je vracet objekt zvaný *binding*. Tento objekt reprezentuje deklaraci daného identifikátoru. Identifikátory se mohou ve zdrojovém kódu vyskytovat v různých situacích:

1. Mohou být buď součásti deklarace. V takovém případě není pro vyhledání deklarace nutné prohledávat AST.

2. Častěji se identifikátory vyskytují ve výrazech. V těchto situacích je pro vyhledání deklarace nutné prohledat AST. Využívá toho například nástroj pro přejmenování proměnných nebo nástroj pro odkázání se na deklaraci (více v kapitole 8).

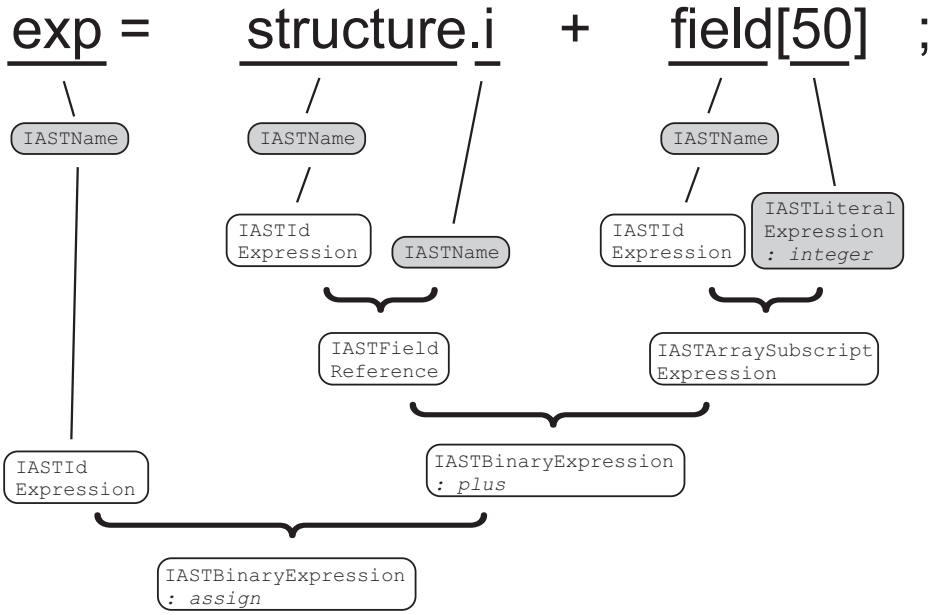
[4]Více je popsáno v sekci 8.1.2, která se zabývá činností visitorů.

3. Existují i výjimky, kdy identifikátor tvoří například specifikátor deklarace (proměnná je programátorem definovaného typu). V tomto případě je nutné řešit bindings dle konkrétní situace.

Metody pro získání bindings jsou implementovány tím způsobem, že zavolají visitor, který se pokusí najít danou deklaraci v AST. Princip této činnosti je vysvětlen v sekci 8.1 zabývající se bindings.

Rozhraní IASTExpression

Rozhraní `IASTExpression` slouží jako výchozí typ pro všechny typy výrazů. Nepřímo rozšiřuje (prostřednictvím rozhraní `IASTInitializerClause`) základní typ uzlu `IASTNode`. Jeho typickými metodami jsou metody pro získání typu výsledné hodnoty, kterou reprezentují a dále metody pro zjištění, zda se výraz může nacházet na levé straně přiřazení.



Obrázek 7.4: Příklad interpretace výrazu v AST

Jak je znázorněno na obrázku 7.4, rozhraní `IASTExpression` je dle konkrétního typu výrazu rozšiřováno dalšími rozhraními:

- Pro uchování literálu poskytuje objektový model projektu CDT rozhraní `IASTLiteralExpression` (a jeho implementaci `CASTLiteralExpression` pro jazyk C). Toto rozhraní se od základního rozhraní `IASTExpression` liší v tom, že obsahuje konstanty odlišující jednotlivé druhy literálů (celé číslo, znak, řetězec, ...). Consume funkci je pak předáván v parametru konkrétní druh literálu a v závislosti na něm vytvořen uzel konkrétního druhu.
- Ve výrazech je možné používat identifikátory (například chceme-li nějakou proměnnou sečíst s jinou). Výskyty těchto identifikátorů jsou reprezentovány rozhraním `IASTIdExpression`. Instance tohoto rozhraní pak slouží jako obálka pro uzel typu `IASTName`.

- Pro výskyty složených identifikátorů (posloupností identifikátorů oddělených např. tečkami) poskytuje objektový model CDT rozhraní `IASTFieldReference`. Uzel tohoto typu obsahuje jako své potomky identifikátor `IASTName` a výraz `IASTExpression` (který může implementovat rozhraní `IASTIdExpression` nebo opět rozhraní `IASTFieldReference`). Díky tomu je možné modelovat reference na prvky struktur (které jsou v CDT nazývány pole – *field*), v jazyce CodAL potom reference na zdroje zanořené v entitách (viz 2.3.5).

- V jazyce CodAL je možné se ve výrazech odkazovat na konkrétní registry z pole registrů – např: `reg[0]`. Tuto konstrukci knihovna CDT reprezentuje pomocí rozhraní `IASTArraySubscriptExpression`. Pomocí tohoto rozhraní jsou modelovány reference na konkrétní prvky pole v jazyce C. Uzel implementující toto rozhraní obsahuje dva potomky typu `IASTExpression` (prvním je identifikátor a druhým výraz v hranatých závorkách).

- Výsledkem nějaké operace nad výrazem je opět výraz. Pro modelování operace s výrazy jsou v projektu CDT k dispozici rozhraní `IASTUnaryExpression` pro unární operace, `IASTBinaryExpression` pro binární operace a `IASTConditionalExpression` pro ternární operace. Consume funkcím zpracovávajícím tyto operace je v parametru předáván druh operace obdobným způsobem, jako tomu je při odlišení literálů. Priority operací se modelují prostřednictvím pravidel gramatiky.

  Příklad 7.4

```
additive_expression
  ::=  multiplicative_expression
    |  additive_expression '+' multiplicative_expression
                /. $Build  consumeExpressionBinaryOperator
                    (IASTBinaryExpression.op_plus);  $EndBuild ./

multiplicative_expression
  ::=  unary_expression
    |  multiplicative_expression '*' unary_expression
          /. $Build  consumeExpressionBinaryOperator
              (IASTBinaryExpression.op_multiply);  $EndBuild ./

unary_expression
  ::= -- další pravidlo
```

  V příkladu 7.4 je ukázáno, jakým způsobem je možné stanovit, že násobení má vyšší prioritu jak sčítání. V gramatice *Bison_Codal* byly priority řešeny v některých případech jiným způsobem. Tyto pravidla jsem proto musel ručně přepsat do správného formátu. Nemohl jsem přitom využít existující pravidla z gramatiky *LPG_C99*, jelikož výrazy v jazyce CodAL jsou odlišné od výrazů v jazyce C. Mohl jsem však využít existující consume funkce.

Knihovna CDT nabízí spoustu dalších typů výrazů, jakými jsou například lambda výrazy nebo výrazy přetypování. Pro jazyk CodAL jsem si však vystačil s výše zmíněnými.

Rozhraní IASTStatement

Výrazy bývají ve většině případů součástí příkazů. Pro příkazy je v knihovně CDT vyčleněno rozhraní `IASTStatement`. Toto rozhraní přímo rozšiřuje základní typ uzlu `IASTNode` a slouží obdobně jako rozhraní `IASTExpression` jako základní vzor pro konkrétní typy příkazů. V jazyce CodAL jsem použil následující skupiny příkazů:

1. První skupinou jsou příkazy, které do sebe zanořují jiné typy uzlů, než jsou příkazy. Příkladem je rozhraní `IASTExpressionStatement` (obaluje výraz – `IASTExpression`) a rozhraní `IASTDeclarationStatement` (obaluje deklaraci – `IASTDeclaration`).

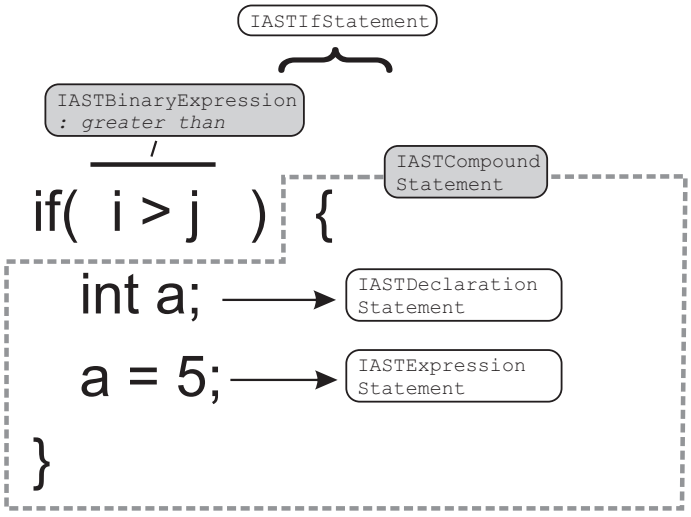
2. Dalším typem jsou příkazy větvení. Z této skupiny jsem využil rozhraní `IASTIfStatement` a `IASTSwitchStatement`, který dále využívá pomocné typy příkazů – `IASTCaseStatement` a `IASTBreakStatement`.

3. Třetí skupinu představují příkazy pro cykly. Zástupci z této skupiny jsou rozhraní `IASTForStatement`, `IASTWhileStatement` a `IASTDoStatement`.

4. Čtvrtou skupinou jsou příkazy skoku. Zde jsem využil rozhraní `IASTContinueStatement`, `IASTBreakStatement` `IASTReturnStatement`. Tyto rozhraní modelují příkazy `continue`, `break` a `return`. Rozhraní pro příkaz skoku `goto` nebylo použito, jelikož je tato konstrukce v jazyce CodAL zakázána.

5. Prázdný příkaz je reprezentovaný prostřednictvím rozhraní `IASTNullStatement`.

6. Pro seskupení příkazů do seznamu jsou určeny složené příkazy. Modelovány jsou pomocí rozhraní `IASTCoumpoundStatement`. Instance tohoto rozhraní obsahuje potomky stejného typu – `IASTStatement`. Díky tomu je možné si v seznamu uchovávat veškeré typy příkazů bez ohledu na jejich konkrétní typ. Složený příkaz může rekurzivně obsahovat další složené příkazy. Stejným stylem jsou řešena například těla příkazů větvení.



Obrázek 7.5: Příklad interpretace příkazu v AST

Rozhraní IASTDeclaration

Pokud nebudeme zohledňovat fakt, že příkazy mohou být potomky jiných příkazů, jsou příkazy zpravidla obsaženy v tělech deklarací (nejčastěji definicí funkcí). Tyto typy uzlů jsou v knihovně CDT modelovány pomocí rozhraní `IASTDeclaration`.

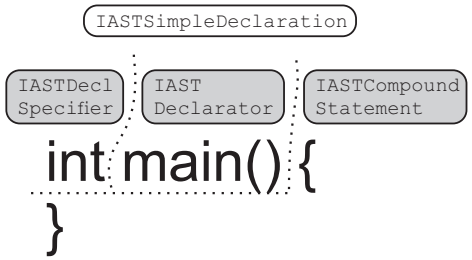
Společným rysem uzlů, které implementují toto rozhraní je, že musí povinně obsahovat ***specifikátor deklarace*** (`IASTDeclSpecifier`), který určuje typ proměnné. Další potomci závisí na typu odvozeného rozhraní. Většina deklarací obsahuje zpravidla tyto dva další druhy potomků:

1. ***deklarátor*** (`IASTDeclarator`) sloužící pro stanovení názvu deklarace,
2. a ***tělo deklarace***, obsahující například příkazy (`IASTCompoundStatement`) nebo seznam dalších deklarací.

Volitelně pak deklarace mohou obsahovat další potomky. V objektovém modelu jazyka CodAL jsem implementoval nové typy deklarací, které obsahují například bitovou šířku zdroje.

Z objektového modelu knihovny CDT jsem použil dva odvozené typy deklarací:

1. Základním odvozeným typem deklarace je rozhraní `IASTSimpleDeclaration`. Tento typ je používán pro deklarace proměnných jazyka C. Místo jednoho deklarátoru obsahuje seznam deklarátorů. Tento typ uzlu jsem použil jako výchozí rozhraní pro implementaci nových rozhraní modelující deklarace jazyka CodAL.
2. Pro reprezentaci definicí funkcí je vyčleněno rozhraní `IASTFunctionDefinition`. Oproti rozhraní `IASTSimpleDeclaration` povoluje pouze jeden deklarátor. Vyžaduje ovšem tělo definice funkce, které implementuje rozhraní `IASTStatement`. Ve všech případech se ale jedná o složený příkaz `IASTCompoundStatement`.



Obrázek 7.6: Příklad interpretace deklarace v AST

Rozhraní IASTDeclSpecifier

Instance rozhraní `IASTDeclSpecifier` je hlavním vyžadovaným potomkem deklarací (nastavuje se vždy již prostřednictvím konstruktoru). Slouží jako *specifikátor deklarace*. Pomocí těchto uzlů je stanoven typ proměnné nebo například návratový typ funkce. Přesný výčet těchto typů je uveden v konkrétních rozhraních odvozených z `IASTDeclSpecifier`.

- Základním odvozeným rozhraním je `IASTSimpleDeclSpecifier`, které obsahuje známé typy jako `int`, `char` nebo také `void`. Při volání consume funkce je v parametru funkce

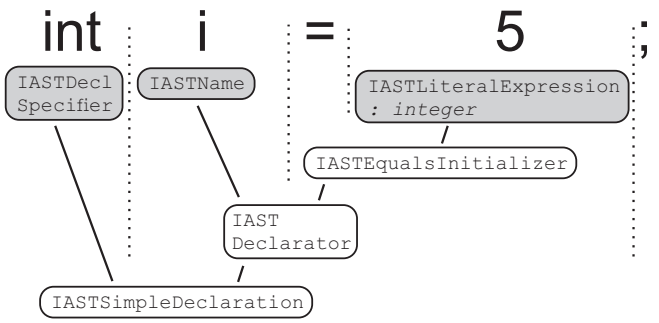
předán tento typ. To, o jaký typ se jedná, je určeno podle toho, jaké pravidlo gramatiky se vykonalo.

- Dalším důležitým rozhraním je `IASTNamedTypeSpecifier`. Toto rozhraní nemá přesně stanovený výčet možných specifikátorů, ale obsahuje jednoho potomka implementujícího rozhraní `IASTName`[5]. Díky tomu je možné stanovit vlastní druh specifikátoru (např. struktury v jazyce C).

Rozhraní IASTDeclarator

Rozhraní `IASTDeclarator` je dalším používaným typem uzlu v deklaracích. Reprezentuje *deklarátor*, který slouží jako obálka pro název deklarace. Vyžadovaným potomkem deklarátoru je proto identifikátor – uzel implementující rozhraní `IASTName`. Při analýze identifikátoru pak můžeme snadno určit, o jaké použití proměnné se jedná. Pokud je to deklarace, rodičem identifikátoru bude vždy deklarátor. Pokud je to použití proměnné, rodičem bude výraz.

Volitelně může deklarátor mimo jiné obsahovat ***inicializátor*** (`IASTInitializer`). Tento typ uzlu slouží pro uchování výrazu, kterým je proměnná inicializována při deklaraci. Nejčastěji se jedná o výraz přiřazení (například: `int i = 0`), který lze modelovat pomocí rozhraní `IASTEqualsInitializer`.



Obrázek 7.7: Příklad interpretace deklarátoru v AST

Rozhraní IASTTranslationUnit

Rozhraní `IASTTranslationUnit` modeluje kořenový uzel abstraktního syntaktického stromu. Rozšiřuje rozhraní `IASTDeclarationListOwner` (odvozené z `IASTNode`). Toto rozhraní předepisuje metody pro práci s seznamem deklarací (`IASTDeclaration`), které tvoří potomky kořenového uzlu.

Prostřednictvím metody `accept` kořenového uzlu se nejčastěji provádějí analýzy abstraktního syntaktického stromu.

7.2.2 Rozšíření o konstrukce jazyka CodAL

Pro potřeby uchovávání konstrukcí odpovídající gramatice jazyka CodAL v abstraktním syntaktickém stromu jsem implementoval nové typy uzlů. Abych zároveň mohl využít existující nástroje projektu CDT pro editor jazyka CodAL, musel jsem implementovat nové

[5] Jedná se o jednu z výjimek použití uzlu typu `IASTName`.

typy uzlů AST tak, aby rozšiřovaly rozhraní objektového modelu CDT. Navázal jsem konkrétně na objektový model pro AST jazyka C. Tímto rozšířením jsem vytvořil třetí vrstvu objektového modelu CDT.

V první verzi objektového modelu jazyka CodAL jsem vytvořil přibližně 100 nových typů uzlů. Hotový model mi poskytl lepší pohled na jazyk CodAL a mohl jsem díky němu snadněji hledat souvislosti mezi jednotlivými konstrukcemi jazyků CodAL a C. Do další verze objektového modelu se mi podařilo zmenšit počet typů uzlů přibližně na třetinu[6]. Spousta z nich by se i nyní dala seskupit nebo nahradit existujícími uzly objektového modelu CDT. V některých případech jsem ovšem shledal vhodnější využívat typové odlišení pro stejné syntaktické konstrukce z důvodu odlišného sémantického významu. Další zpracování takových konstrukcí je výhodnější řešit odděleně.

V následujícím textu jsou popsány některá důležitá rozhraní modelující nové typy uzlů objektového modelu. Pro přehlednost jsem tyto rozhraní rozčlenil do skupin.

Modifikované typy uzlů objektového modelu CDT

Jazyk CodAL ve svých částech povoluje úseky kódu implementované v jazyce C. Bylo výhodné se snažit namodelovat tyto úseky kódu pomocí uzlů implementující existující rozhraní modelu CDT. V případě drobných odlišností v syntaxi stačilo mírně zmodifikovat existující typy uzlů.

1. Odlišností oproti jazyku C jsou způsoby deklarace funkcí nebo proměnných. V jazyce CodAL je povoleno za deklaraci uvést seznam identifikátorů ohraničený dvěma závorkami (např. `((param1,param2))`). Pro modelování těchto konstrukci jsem vytvořil rozhraní `IASTCodalSimpleDeclaration`, které rozšiřuje základní rozhraní `IASTSimpleDeclaration`. Rozdílem mezi rozhraními jsou dodatečné metody pro práci se zmíněným seznamem parametrů.

2. Druhý příklad odlišnosti mezi jazyky C a CodAL představovaly složené příkazy – `IASTCompoundStatement`. V objektovém modelu jazyka CodAL jsem pro složené příkazy implementoval vlastní rozhraní `IASTCodalCompoundStatement`. Oproti původní verzi může složený příkaz obsahovat specifikátor `SIM` nebo `SEM`.

   Příklad 7.5

   ```
   SIM {
     int a = 1;
     // atd...
   }
   ```

Pomocné typy uzlů objektového modelu jazyka CodAL

Jazyk CodAL obsahuje spoustu konstrukcí, které v jazyce C nejsou povoleny. Pro tyto konstrukce bylo nutné vytvořit nové typy uzlů, aby je bylo možné zpracovávat v AST.

1. Typickým zástupcem je ***definice bitové šířky*** (např. `bit[32]` nebo `bit[0..31]`). Pro tuto konstrukci jsem vytvořil rozhraní `IASTCodalBitWidth`. Uzly implementující toto rozhraní slouží jako obálka pro výraz `IASTExpression`. Pro výrazy reprezentující rozsahy (např. `0..31`) jsem použil existující výraz pro binární operátor. Jako druh

[6] Rozšíření objektového modelu CDT obsahovalo v březnu 2013 přesně 29 nových typů uzlů.

binární operace jsem zvolil existující druh určený pro rozsahy v jazyce C (značí se posloupností tří teček). V jazyce CodAL tyto konstrukce nejsou povoleny, tudíž nedojde k záměně.

2. Dalším novým rozhraním objektového modelu AST jazyka CodAL je `IASTCodalBinaryLiteral`. Cílem tohoto rozhraní je rozšířit existující literály jazyka C o ***literál reprezentující binární číslo*** (např. 00000000:11111111). Uzel implementující toto rozhraní se skládá ze dvou čísel, přičemž levé z nich je volitelné.

3. Dalšími významnými typy uzlů jazyka CodAL jsou ***modifikátory operací***. Jedná se o jednoduché výrazy obsažené v hlavičce deklarací operací. Jejich účelem je poskytovat referenci na konkrétní deklaraci (např. fázi zřetězené linky). Tyto výrazy bylo nutné pro další zpracování odlišit od běžných výrazů. Vytvořil jsem pro ně proto rozhraní `IASTCodalOperationModifier` a `IASTCodalRepresentsSection`. Uzly implementující tyto rozhraní slouží jako obálka pro tyto výrazy.

4. V jazyce CodAL jsou přípustné ***specializované příkazy*** jakým je například `delay` používaný v sekci `timing` (Viz 2.3.6). Pro tyto příkazy jsem rovněž vytvořil vlastní rozhraní.

Specifikátory deklarací jazyka CodAL

Jazyk CodAL obsahuje oproti jazyku C mnoho nových klíčových slov. Velká část z nich slouží jako specifikátor deklarace (např. `register` nebo `event`). V rozhraních modelujících specifikátory jazyka C tyto druhy specifikátorů nejsou zahrnuty. Implementoval jsem proto tři nová rozhraní:

1. Základním typem specifikátoru jazyka CodAL je `IASTCodalSpecifier`. Toto rozhraní je založeno na podobném principu jako rozhraní `IASTSimpleDeclSpecifier` určené pro specifikátory jazyka C. V rozhraní `IASTCodalSpecifier` jsou definovány konstanty jednotlivých druhů specifikátorů jazyka CodAL. Spolu s novým rozhraním jsem vytvořil novou consume funkci, která přebírá v parametru typ aktuálně zpracovávaného specifikátoru a vytváří instanci s patřičným nastavením druhu specifikátoru.

2. Druhým rozhraním je `IASTCodalNameSpecifier`. Toto rozhraní modeluje situaci, kdy je specifikátor reprezentován identifikátorem. Je vytvořeno rozšířením ekvivalentního rozhraní objektového modelu CDT. Důvodem, proč jsem provedl rozšíření je ten, že rozhraní `IASTCodalNameSpecifier` mimo jiné rozšiřuje základní rozhraní `IASTCodalSpecifier` zmíněné v předchozí odrážce. Díky tomu mohu se všemi typy specifikátorů jazyka CodAL pracovat hromadně.

3. V modelu jazyka CodAL přibyl speciální typ specifikátoru. Je označován jako *modifikátor specifikátoru deklarace* a slouží jako dodatečná modifikace uvedeného specifikátoru (např. `extern bus`). Pro modifikátory jsem vytvořil vlastní rozhraní `IASTCodalModifier`. Jeho podoba je shodná s podobou rozhraní `IASTCodalSpecifier`. Liší se pouze v konstantách určujících druh modifikátoru.

Deklarátory jazyka CodAL

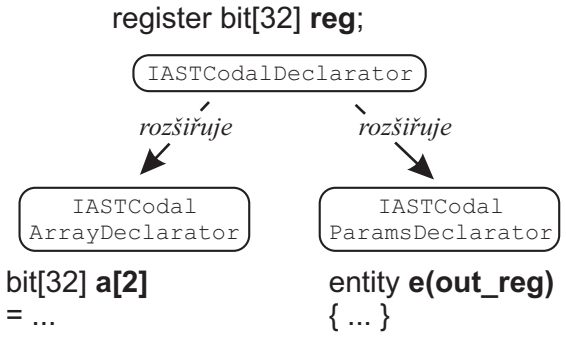
Obdobně, jako jsem vytvořil vlastní rozhraní pro specifikátory, jsem vytvořil vlastní rozhraní pro deklarátory. Hlavním důvodem bylo odlišení od deklarátorů objektového modelu

jazyka C. V případě, že se při analýze identifikátoru zjistí, že jeho rodič je deklarátor, může být otestováno, zda se jedná o deklaraci jazyka CodAL nebo deklaraci jazyka C. V závislosti na této informaci je možné zavolat odpovídající metody pro tvorbu bindings.

1. Základním rozhraním reprezentujícím deklarátory jazyka CodAL je rozhraní `IASTCodalDeclarator`. Prostřednictvím tohoto rozhraní je možné pracovat se všemi typy deklarátorů jazyka CodAL.

2. Pro deklarátory obsahující identifikátor spolu výrazem jsem implementoval rozhraní `IASTCodalArrayDeclarator`. Tento typ deklarátoru je odvozen z existujícího rozhraní `IASTArrayDeclarator`, které se v C používá k deklaraci polí.

3. Speciálním typem deklarátoru je `IASTCodalParamsDeclarator`. Toto rozhraní je vymezeno pro deklarátory, které mají jako seznam potomků stanoven seznam identifikátorů. Je používáno například deklarátory pamětí nebo registrů.



Obrázek 7.8: Deklarátory v jazyce CodAL

Základní deklarace jazyka CodAL

Hlavní uzel (*translation unit*) obsahuje jako své potomky deklarace. Na rozdíl od jazyka C se na této úrovni nacházejí deklarace paměťových zdrojů a operací. V objektovém modelu jazyka CodAL jsem pro tyto deklarace vytvořil následující nová rozhraní:

1. **Paměťové zdroje** jsou reprezentovány rozhraním `IASTCodalResource`. Uzly implementující toto rozhraní obsahují jako své potomky specifikátor a seznam deklarátorů (obdobně, jako je tomu u rozhraní `IASTSimpleDeclaration`). Mimo jiné tento typ uzlu obsahuje modifikátor deklarace (`IASTCodalModifier`), uzel definující bitovou šířku (`IASTCodalBitWidth`) a tělo skládající se z pomocných deklarací (seznamu atributů zdroje). Aktivní mohou být přitom pouze někteří potomci. Díky tomu jsem pomocí rozhraní `IASTCodalResource` namodeloval více syntakticky podobných konstrukcí jazyka CodAL:

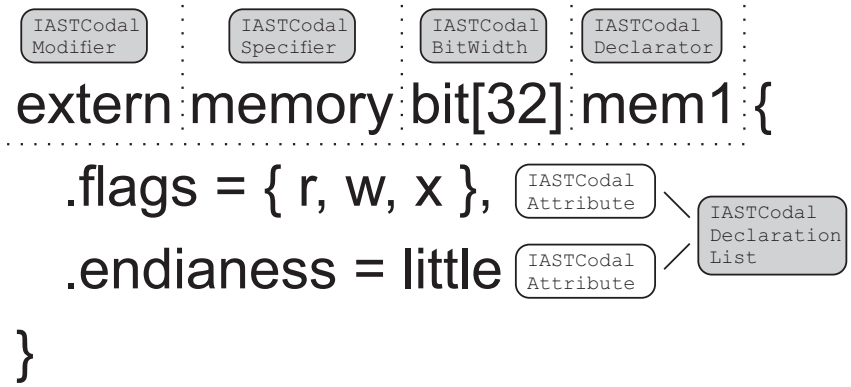
   (a) Hlavní skupinu tvoří základní zdroje, jakými jsou ***registry, paměti a sběrnice***. Tyto konstrukce se vyznačují tím, že musí povinně obsahovat specifikaci bitové šířky (např. `bit[32]`). Konstrukce této skupiny mohou dále volitelně obsahovat modifikátor (viz `IASTCodalModifier`). Třetí položkou je tělo obsahující seznam pomocných deklarací, které reprezentují atributy zdrojů. Atributy jsou

modelovány rozhraním `IASTCodalAttribute`, které obsahuje specifikátor (druh atributu) a příkaz `IASTStatement` (tělo atributu, které v sobě obsahuje výrazy odpovídající danému druhu atributu).

(b) Další skupinou jsou ***porty, signály, logické obvody a programový čítač***. Uzly s těmito specifikátory se od skupiny základních zdrojů liší v tom, že nemají žádné tělo a nemůže jim být stanoven modifikátor.

(c) Konstrukce popisující ***zřetězenou linku a mapování paměťového prostoru (memorymapping)*** neobsahují definici bitové šířky a modifikátor. Skládají se pouze ze specifikátoru, jednoho deklarátoru (který u konstrukce `memorymapping` není povinný) a těla, které obsahuje pomocné deklarace. U zřetězených linek se jedná o deklarace jednotlivých fází zřetězené linky a u mapování paměťového prostoru (`IASTCodalPipelineStage`) se jedná o deklarace jednotlivých mapování (`IASTCodalMemoryMap`).

(d) Posledním prvek modelovaným pomocí rozhraní `IASTCodalResource` je ***instance entity***. Uzly tohoto typu obsahují pouze specifikátor (reprezentovaný identifikátorem – `IASTCodalNameSpecifier`) a seznam deklarátorů.



Obrázek 7.9: Příklad interpretace zdrojů v AST

2. Speciálním typem deklarace jsou **aliasy**. Tento typ uzlů je podobný deklaraci proměnné v jazyce C obsahující inicializaci této proměnné. Rozdílem je, že deklarace obsahují bitovou šířku a specifikátor je odvozen od zdroje, kterým je nová proměnná inicializována. Tento typ uzlů je modelován rozhraním `IASTCodalAlias`. Pravá strana deklarace je uložena v inicializátoru deklarátoru.

3. Dále jsem vytvořil rozhraní `IASTCodalEntity` modelující **entity**. Tyto typy uzlů jsem modeloval obdobně jako jsou řešeny struktury jazyka C v projektu CDT. Entity mají stanoven specifikátor, deklarátor a tělo obsahující deklarace – `IASTDeclaration` (gramatika jazyka CodAL pak tyto deklarace omezuje pouze na zdroje a aliasy).

4. Pro **operace** (instrukce a události) je určeno rozhraní `IASTCodalOperation`. Deklarace obsahuje specifikátor (`event` nebo `element`), jeden deklarátor, volitelně modifikátory operace a povinně tělo obsahující příkazy (`IASTCompoundStatement`). Příkazy mohou nabývat následujících podob:

(a) Obalují do sebe deklaraci instance operace (`IASTCodalInstance`), která zviditelňuje deklarace jiných operací v těle dané operace.

(b) Obalují do sebe deklaraci atributu (`IASTCodalAttribute`), v jehož těle je specifikováno konkrétní chování nebo vlastnosti operace.

(c) Jsou reprezentovány příkazy větvení, pomocí kterých je možné povolit aktivitu atributů operace pouze za určitých podmínek.

5. Instrukce jsou seskupeny v **instrukčních sadách** modelovaných pomocí deklarace `IASTCodalGroup`. Tyto deklarace obsahují tělo se seznamem identifikátorů reprezentující instrukce.

6. Posledním typem deklarace, která může být obsažena v seznamu deklarací kořenového uzlu, je `IASTModelType`. Jedná se o jednoduchou deklaraci, která obsahuje identifikátor stanovující nastavení konkrétního modelu.

7.3 Analýza syntaktických chyb

V minulé sekci byly popsány důležitá rozhraní modelující validní konstrukce jazyka CodAL. Aby bylo možné lokalizovat syntaktické chyby ve zdrojovém kódu, bylo nutné použít další typ uzlu AST, jehož cílem je nést informaci o konkrétních chybách.

Činnost detekce syntaktické chyby probíhá v následujících třech krocích:

1. V případě, že parser nemůže použít žádné pravidlo příslušné gramatiky, aplikuje implicitní pravidlo.

2. Vykonáním implicitního pravidla se zavolá consume funkce, která vytvoří uzel nesoucí informaci o syntaktické chybě a připojí ho do AST.

3. Editor pomocí metod `accept` vyhledá všechny uzly, které reprezentují chybu a získá informace o tom, jaký úsek kódu tyto uzly reprezentují. Ze získaných informací je provedeno zvýraznění chybných částí zdrojového kódu.

```
element simm16
{
    assembler {immval=signed};
    binary {immval=0bs[16]};
    tato konstrukce neni povolena
    return {immval;};
}
```

Obrázek 7.10: Ukázka odhalení chybného kódu

7.3.1 Implementace implicitních pravidel

Implicitní pravidla je nutné vkládat na vhodná místa gramatiky spojená například s generováním seznamů. Systém jejich umísťování je možné ilustrovat na následujícím příkladu.

Vezměme uzel typu `IASTCodalResource`, který modeluje základní zdroje. Jako jednoho ze svých potomků může obsahovat seznam deklarací (`IASTDeclarationList`). Tento seznam je složen z atributů vztahujících se k danému zdroji. Gramatika určuje, jaké typy atributů může konkrétní typ zdroje (paměť, registr, sběrnice, ...) obsahovat. Na příkladu 7.6 jsou znázorněny pravidla popisující tělo zdroje typu paměť (`memory`).

Příklad 7.6

```
cs_mem_body
        ::=  '{' <openscope-ast> cs_mem_param_list '}'
                  /. $Build  consumeDeclarationList();  $EndBuild  ./

cs_mem_param_list
        ::=  cs_mem_param_list ',' cs_mem_param
        |    cs_mem_param

cs_mem_param
        ::=  cs_dataport
        |    cs_size
        |    ...
        |    cs_declaration_error
```

Tělo paměti je reprezentované neterminálem `cs_mem_body`. Tento neterminál může být expandován na posloupnost atributů (`cs_mem_param`) obalenou ve složených závorkách. V metodě `consumeDeclarationList` je pak na vrcholu zásobníku očekáván seznam o libovolném počtu položek (stanoveno pomocí konstrukce `<openscope-ast>`).

Pravidlo expandující neterminál `cs_mem_param` určuje, jakých atributů může tělo paměti nabývat[7]. Tyto atributy mají svoji syntaxi popsanou dalšími pravidly, které nyní nejsou podstatné. Důležitý je neterminál `cs_declaration_error`. Pravidlo expandující tento neterminál má následující podobu.

Příklad 7.7

```
cs_declaration_error
        ::=  ERROR_TOKEN
               /. $Build  consumeDeclarationProblem();  $EndBuild ./
```

`ERROR_TOKEN` je speciální typ tokenu, který je definován v šabloně gramatiky. Je tvořen doplňkem k množině tokenů, které je možné jako první vygenerovat z aktuálního místa gramatiky. V případě, že lexer pošle parseru token, který nelze vygenerovat žádným aktuálně použitelným pravidlem gramatiky, použije se toto implicitní pravidlo a daný token je zpracován jako chybný token.

7.3.2 Uchování neplatného tokenu v AST

Objetový model AST projektu CDT obsahuje rozhraní `IASTProblem`. V uzlech implementující toto rozhraní jsou uchovány všechny důležité informace o chybě. Rodičem tohoto uzlu je vždy uzel implementující rozhraní `IASTProblemHolder`, který slouží jako jeho obálka.

[7] Pro jednoduchost nejsou znázorněny všechny možné typy atributů.

Toto zapouzdření je použito proto, jelikož consume funkce očekávávají na vrcholu zásobníku konkrétní typ uzlu (například deklaraci nebo příkaz). Výskyt neočekávaného typu uzlu na vrcholu zásobníku by vedl k chybě tvorby AST.

Rozhraní `IASTProblemHolder` je rozšířeno rozhraními, které jsou mimo jiné vždy odvozeny z některého konkrétního typu uzlu. Z modelu CDT jsem použil typy `IASTProblemDeclaration` (pro deklarace), `IASTProblemStatement` (pro příkazy) a `IASTProblemExpression` (pro výrazy). V příkladu 7.7 je v souvislosti s implicitním pravidlem volána funkce `consumeDeclarationProblem`, která vytvoří uzel implementující rozhraní `IASTProblemDeclaration`. Díky tomu, že je toto rozhraní odvozeno z rozhraní `IASTDeclaration`, je uzel představující syntaktickou chybu přidán do seznamu deklarací stejně jako ostatní uzly reprezentující atributy paměti.

7.3.3 Problém s určením rozsahu chyby

Při implementaci implicitních pravidel jsem se setkal s problémem, kdy nebylo možné přesně namodelovat rozsah syntaktické chyby. Problém se týkal především chybějících závorek. Příklad problému je znázorněný na obrázku 7.3.3 .

```
memorymapping defaultmap {
    0x0.. 0x007FFFFF = program[31..0];
;

ram bit[32] program {
    .dataport = {1, 1},
    .latency = {1,1},
    .endianess = big,
    .lau = 8,
    .size = 0x200000,
    .flags = {r, w, x}
};
```

Obrázek 7.11: Problém určení rozsahu syntaktické chyby

V případě, kdy uživatel zapomene ukončit tělo mapování paměťového prostoru závorkou, parser pokračuje v aplikování pravidel modelující atributy bloku `memorymapping` (obdobných, jako byly uvedeny na příkladu 7.6). Jelikož se jedná o neplatné konstrukce těla, následující tokeny jsou rozpoznávány jako chybné až do doby výskytu první ukončující složené závorky, která zapříčiní vynoření z těla konstrukce `memorymapping`. Tento problém způsobí, že je uživateli zvýrazněna velká část kódu jako chybná. Chybu ovšem z logického hlediska způsobuje pouze jeden chybějící znak.

Implementace těchto situací pomocí dodatečných pravidel je komplikovaná a gramatiku by výrazně zesložitila. Problém se vyskytuje i v jiných editorech. V rámci diplomové práce jsem ho ponechal jako otevřený. Jeho řešení by bylo možné provést například dodatečnými kontrolami ve statické analýze a zvýrazněním prvního chybného řádku.

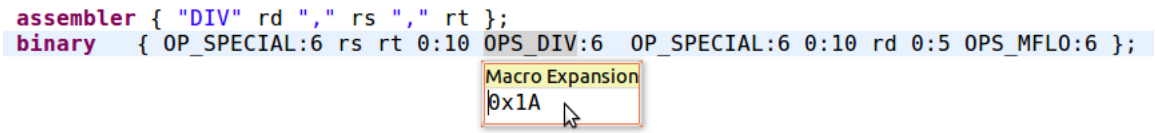
7.4 Direktivy preprocesoru

Nevýhodou původního editoru jazyka CodAL implementovaného pomocí nástroje Xtext bylo, že neuměl pracovat s direktivami preprocesoru. Projekt CDT byl navržen tak, aby tyto

konstrukce uměl pro jazyk C a C++ zpracovávat. Direktivy preprocesoru jazyka CodAL odpovídají syntaxi direktiv preprocesoru jazyka C. V následujícím textu jsou popsány některé důležité direktivy uplatněné v jazyce CodAL.

7.4.1 Makra preprocesoru

Makra jsou první skupinou direktiv. Pro jejich definici slouží direktiva `#define`. V objektovém modelu CDT je pro makra vyčleněno rozhraní `IASTPreprocessorMacroDefinition`. Preprocesor projektu CDT provede při parsování souboru nahrazení všech výskytů konstant použitých v `#define` za výrazy, které tyto konstanty definují. Výrazy jsou díky tomu parsovány spolu se zbytkem zdrojového kódu. Je tedy otestována jejich syntaktická správnost v kontextu úseků zdrojového kódu, v kterých jsou použity. Ve výsledném AST se pak nachází jak definice makra tak uzly reprezentující zpracovaný výraz. Při výběru konstanty myší je uživateli zobrazeno okno s původním výrazem.



Obrázek 7.12: Ukázka makra v jazyce CodAL

7.4.2 Podmínkové příkazy preprocesoru

Druhou skupinu představují direktivy preprocesoru `#ifdef`, `#elif` a `#endif`. Podmínka `#ifdef` testuje, zda byla definována konkrétní konstanta. V případě, že ne, chová se jako komentář. Zdrojový kód, který se nachází v tomto bloku, není zpracován a začleněn v AST. Uživateli jsou tyto bloky speciálně odlišeny.

7.4.3 Direktivy pro modularizaci

V hlavičce zdrojového kódu se může vyskytnout direktiva `#include` začleňující jiný soubor se zdrojovým kódem. Parser si tuto direktivu virtuálně nahradí zdrojovým kódem obsaženým v souboru, na který direktiva ukazuje. Tento zdrojový kód je pak zpracován spolu se zdrojovým kódem zbytku souboru, v kterém se direktiva nachází.

V jazyce CodAL se nachází dva typy tohoto druhu direktiv – základní varianta `#include` a rozšířená varianta `INCLUDE`. Druhý typ je určen pro simulátory. Direktivy neobsahují znak `#` proto, aby byly preprocesorem jazyka CodAL vynechány. Projekt CDT ovšem tento styl zápisu nerozpozná jako direktivu. Z tohoto důvodu byla syntaxe této direktivy změněna na univerzální direktivu `#pragma include`. Preprocesor tuto direktivu rozpoznává, ale ničím ji nenahrazuje. Je proto přenechána simulátorům.

Kapitola 8

Analýza abstraktního syntaktického stromu

Třetím implementačním úkolem bylo rozšířit editor vytvořený v druhé etapě tak, aby uměl:

- vyhledat deklaraci libovolné aktuálně vybrané proměnné,
- vyhledat výskyty všech proměnných libovolné deklarace a dokázal je hromadně přejmenovat (refaktorizace),
- poskytnout uživateli hierarchickou strukturu deklarací ve zdrojovém kódu a umožnil se na tyto deklarace odkázat (nástroj *Outline*).

Pro dosažení těchto požadavků jsem musel provést analýzu abstraktního syntaktického stromu, vybrat z něho důležité uzly a ty pak logicky provázat a reprezentovat v nových strukturách určených konkrétním nástrojům pro správu zdrojových kódů.

Projekt CDT obsahuje implementaci všech požadovaných nástrojů, jakými jsou nástroj Outline, nástroj pro refaktorizaci a nástroj pro odkázání se na proměnnou. Tyto součásti editoru pracují s pevně stanovenými strukturami, které jsou popsány objektovými modely. Musel jsem pro ně proto vytvořit dva typy modelů:

1. První model je určený pro modelovaní vazeb mezi proměnnými. Účelem modelu je vymezit všechny deklarace proměnných, vymezit jejich rozsah viditelnosti a provázat je s výskyty použití těchto proměnných. Model je určený pro hromadné přejmenování proměnných a pro nástroj, pomocí kterého se můžeme odkazovat na deklarace. Nazvěme tento model jako ***model vazeb***.
2. Druhý model (nazvěme ho ***model objektů***) slouží k modelování přehledu všech jednotek definovaných ve zdrojovém kódu. Abstraktní syntaktický strom obsahuje velké množství uzlů, kde významné jsou pro programátora pouze některé. Struktura modelovaná tímto modelem je oproti AST výrazně menší. V mnoha případech spojuje několik uzlů do jednoho a vytváří z nich logické objekty jakými jsou například definice funkcí spolu s parametry.

Tato kapitola se zabývá tvorbou obou zmíněných modelů, popisuje práci s těmito modely a jejich propojení s nástroji pro správu zdrojových kódů.

8.1 Tvorba modelu vazeb

Struktura popsaná modelem vazeb je složená z prvků, které se nazývají *bindings*. Každý binding reprezentuje právě jednu deklaraci některého objektu (proměnné, funkce, ...). Má vymezený prostor viditelnosti (*scope*), který určuje tu část AST, v které se mohou vyskytovat identifikátory přidružené k této deklaraci. Pro každý identifikátor by mělo být možné najít odpovídající binding. Pokud tomu tak není, znamená to, že identifikátor nebyl řádně deklarován a není možné provést například hromadné přejmenování nebo odkázání se na deklaraci.

8.1.1 Princip provázání proměnných

Každý identifikátor reprezentovaný rozhraním `IASTName` obsahuje metodu `resolveBinding`. Tato metoda vrací pro daný identifikátor odpovídající binding. Pokud se budeme chtít odkázat na deklaraci výskytu libovolné proměnné, musíme nejprve získat binding této proměnné. Získaný binding nám poskytne referenci na odpovídající uzel v AST (zpravidla se jedná o identifikátor[1]).

Pro identifikátory jazyka C (implementovaných třídou `CASTName`) je v případě zavolání metody `resolveBinding` předáno řízení statické metodě třídy `CVisitor` pojmenované stejným názvem. Úkolem této metody je analyzovat druh použití identifikátoru (zda se jedná o deklarace nebo výraz) a v závislosti na tom vytvořit nebo vyhledat odpovídající binding.

1. V případě, že rodič identifikátoru je deklarátor, jedná se o deklaraci. Pro tento případ je vytvořen nový binding. Jeho typ je odvozen z typu deklarace, v které je deklarátor umístěn.

2. Pokud je identifikátor obsažen ve výrazu, je nutné odpovídající binding vyhledat. To je provedeno následujícími kroky:

   (a) Nejprve je vyhledán scope, v kterém se proměnná nachází. Scope je získán z nejbližšího rodičovského uzlu v AST reprezentující (složený) příkaz nebo kořenový uzel.

   (b) V získaném scope jsou prozkoumány všechny existující bindings. Pokud některý z nich reprezentuje deklaraci identifikátoru stejného jména jako zkoumaný identifikátor, je vrácen nalezený binding[2]. V opačném případě je operace opakována pro scope umístěný výše v hierarchii AST.

Nalezený binding je uložen v atributu daného identifikátoru a v případě nového volání `resolveBinding` je z důvodu optimalizace výpočetního výkonu vrácen tento uložený prvek. Bindings bývají hromadně vyhledávány zpravidla při načtení souboru.

8.1.2 Činnost visitorů

Třída `CVisitor` slouží jako *visitor*. Úkolem visitorů je procházet uzly AST a vyhledávat objekty splňující stanovená kritéria. Tato činnost je prováděna dvěma způsoby:

[1] Voláním metody `resolveBinding` identifikátoru A můžeme získat binding obsahující jako deklaraci opět identifikátor A. Tato situace nastává v případě, kdy analyzujeme deklaraci.

[2] Dodatečně bývá prováděna typová kontrola.

1. Prvním způsobem je vyhledávání směrem *shora dolů* prostřednictvím metod `accept`. Visitor se snaží pro daný scope najít všechny výskyty uzlů AST splňující dané požadavky (například identifikátory s konkrétním jménem).

2. Druhým způsobem je vyhledávání *zdola nahoru*. Tímto způsobem jsou prostřednictvím metody `resolveBindings` vyhledávány bindings pro identifikátory AST (více v sekci 8.1.4).

AST vytvořený pro zdrojový kód jazyka CodAL obsahuje typy uzlů, s kterými `CVisitor` neumí pracovat. Musel jsem proto vytvořit vlastní visitor – `CodalVisitor`. Tato třída rozšiřuje třídu `CVisitor` a přetěžuje její statickou metodu `resolveBinding`. V nové verzi této metody je nejprve otestováno zda, je analyzovaný identifikátor součástí konstrukce popsané pomocí objektového modelu jazyka CodAL:

1. Pokud ano, použije se vlastní implementace tvorby/hledání bindings.

2. V záporném případě je tvorba/hledání bindings přenecháno rodičovské metodě `resolveBinding` třídy `CVisitor`.

Správa bindings pro konstrukce jazyka C tedy zůstala kompatibilní s editory C.

Aby byl editorem jazyka CodAL využíván visitor `CodalVistor`, musel jsem přetížit rovněž metodu `resolveBindings` třídy `CASTName`, která implementuje identifikátory jazyka C. K tomu jsem musel vytvořit rovněž vlastní implementaci identifikátorů – třídu `CodalASTName`.

8.1.3 Tvorba bindings pro deklarace v jazyce CodAL

Potřeba vlastní implementace správy bindings je vyžadována ve všech případech, kdy rodičovský uzel identifikátoru implementuje rozhraní `ICodalDeclarator`. V této situaci se jedná o deklaraci v jazyce CodAL. V závislosti na typu deklarace je nutné vytvořit nový binding, který musí implementovat některé z rozhraních modelu vazeb jazyka CodAL.

Základním rozhraním reprezentující bindings je v knihovně CDT rozhraní `IBinding`. Od tohoto rozhraní jsou odvozeny rozhraní určené pro konkrétní typy deklarací. Pro deklarace v jazyce CodAL jsem použil dva typy:

1. U **běžných deklarací** jsem vycházel z rozhraní `IVariable`. V modelu vazeb jsem vytvořil 4 nové bindings: `ICodalResource` (pro zdroje a entity), `ICodalOperation` (pro operace), `ICodalGroup` (pro instrukční sady) a `ICodalInstance` (pro lokální zviditelnění operací v tělech jiných operací). Tyto rozhraní jsem vytvořil především z důvodu vzájemného odlišení deklarací, které reprezentují. Bez dopadu na změnu funkcionality editoru by bylo možné provést sjednocení zmíněných rozhraní do jednoho univerzálního rozhraní nebo dokonce použít pouze základní rozhraní `IVariable`[3].

2. Druhý typ bindings, který jsem použil, je reprezentován rozhraním `IField`. Toto rozhraní modeluje **proměnné, které jsou součástí deklarace jiné proměnné nebo datového typu**. Příkladem tohoto druhu bindings jsou proměnné obsažené ve struktuře jazyka C. Reference na složky struktury probíhá prostřednictvím instance struktury, v které jsou tyto složky obsaženy (např. `struktura1.slozka1`). V jazyce CodAL existují dva druhy bindings implementující rozhraní `IField`:

[3]Tato otázka zůstává prozatím otevřená a její odpověď bude odvozena z další implementace.

(a) Prvním druhem jsou ***složky entit***. Jedná se o vnořené zdroje, které jsou deklarovány uvnitř entit[4]. Pro modelování těchto deklarací jsem v modelu vazeb vytvořil rozhraní `ICodalNestedResource`. Toto rozhraní modeluje stejné druhy deklarací jako `ICodalResource`, které ovšem nejsou součástí entit.

(b) Druhým rozhraním, které jsem přidal do modelu vazeb, je rozhraní `ICodalPipelineStage`. Toto rozhraní modeluje ***fáze zřetězené linky*** (*pipeline stage* – viz 2.3.3). Na rozdíl od vnořených zdrojů jsou deklarace fází zřetězené linky součástí deklarace jiné proměnné (zřetězené linky), nikoliv definice datového typu. V příkladu 8.1 je znázorněn rozdíl mezi sémantikou entity a zřetězených linek.

Příklad 8.1

```
entity entityType {                pipeline pipeline1 {
  register bit[8] reg;              stage1;
}                                  }
entityType entity1;
// používám entity1.reg            // používám pipeline1.stage1
```

V případě, že je hledán binding identifikátoru, jehož rodičovský uzel je deklarátor, který neimplementuje rozhraní `IASTCodalDeclarator`, je zavolána rodičovská metoda `resolveBinding` visitoru `CVisitor`.

8.1.4 Hledání bindings pro výrazy v jazyce CodAL

Druhou situací v metodě `resolveBindings` je případ, kdy je analyzovaný identifikátor součástí výrazu. V takové situaci je nutné vyhledat binding, který reprezentuje deklaraci. Ve visitoru jazyka CodAL jsem musel vyřešit dvě situace:

1. V případě, že je identifikátor součástí výrazu obsaženého v atributu zdrojů, není identifikátoru nastavován žádný binding. Příkladem je nastavení endianity u paměti.

   Příklad 8.2

   ```
   register bit[8] little;

   memory bit[32] memory1 {
     .endianess = little
   }
   ```

   Řetězec `little`, který je přiřazen atributu `endianess`, zde neslouží jako identifikátor a nemá proto nic společného s identifikátorem `little` v deklaraci osmibitového registru. Slouží jako literál[5].

2. Druhou situaci představovaly výrazy obsažené v modifikátoru operací. Součástí těchto výrazů mohou být reference na fáze zřetězených linek. Pro tyto reference jsem implementoval vlastní vyhledávání bindings. Princip algoritmu je zjednodušeně popsán pseudokódem v příkladu 8.3.

[4] Entity mají stejný sémanticky význam jako struktury v jazyce C s tím rozdílem, že jako své složky obsahují zdroje – viz 2.3.5.

[5] Z logického hlediska by bylo vhodnější identifikátory obsažené v atributech obalit do uvozovek (obdobně jako řetězce). Tato změna syntaxe by ovšem znamenala, že by bylo nutné překonvertovat atributy zdrojů všech existujících modelů architektur.

Příklad 8.3

```
function resolvePipelineBinding(IASTName name) :
  // pokud analyzuji nejpravější část složeného identifikátoru
  // obsaženého v modifikátoru operace:
  if(name == id[n], where
    exp == id[1].id[2]...id[n] && exp instanceof IASTFieldExpression
    && exp.parent instanceof IASTCodalOperationModifier) :
      // získám binding předposledního id
      binding = node[n-1].resolveBinding();
      // pokud binding reprezentuje deklaraci zřetězené linky:
      if(binding.decl instanceof IASTCodalPipeline) :
        // projdu všechny fáze zřetězené linky
        for(IASTCodalPipelineStage stage in binding.decl.stages) :
          // vyhledám deklaraci hledané fáze:
          if(stage.name equals id[n]) :
            // nastavím binding
            id[n].setBinding(stage.getBinding());
            return;
  // ostatní případy umí vyřešit visitor CDT
  CVisitor.resolveBinding(name);
```

Konstrukce `instanceof` otestuje, zda je objekt implementuje konkrétní rozhraní. Konstrukce `equals` porovná hodnoty objektů (obdobně jako metoda `equals` v jazyce Java).

Pro všechny ostatní výskyty identifikátorů ve výrazech je volána rodičovská metoda `resolveBinding` visitoru `CVisitor`. Z tohoto důvodu bylo výhodné, že objektový model AST byl založen na objektovém modelu projektu CDT. Tvorba bindings prověřila pečlivost implementace objektového modelu AST a jeho kompatibilitu s objektovým modelem CDT. V některých případech bylo nutné provést dodatečné změny v typech uzlů AST tak, aby byly zpracovávány visitorem jazyka C.

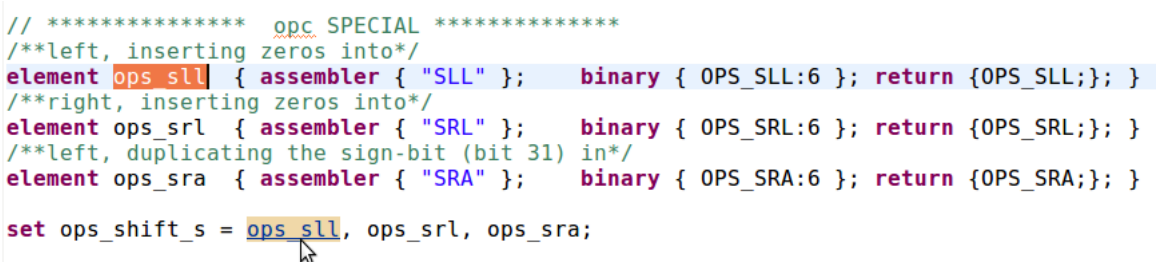
8.2 Využití modelu vazeb

Model vazeb je využit vždy, když je potřeba provázat konkrétní výskyt identifikátoru s jeho deklarací.

8.2.1 Vyhledání deklarace ve zdrojovém kódu

První nástroj CDT, který jsem propojil modelem vazeb, byl vyhledávač deklarací. Vyvolání tohoto nástroje probíhá označením konkrétního výskytu identifikátoru a stisku klávesy `F3` (případně současným podržením klávesy `CTRL` a vybráním identifikátoru levým tlačítkem myši). Výsledkem akce je přesun ve zdrojovém kódu na místo, kde se nachází deklarace tohoto identifikátoru. V případě nenalezení deklarace je ve stavové liště vypsáno chybové hlášení.

Při vyvolání této akce je zavolána metoda `runOnAST` třídy `OpenDeclarationsJob` projektu CDT. Akce běží jako samostatné vlákno. Jejím cílem je získat aktuálně označený



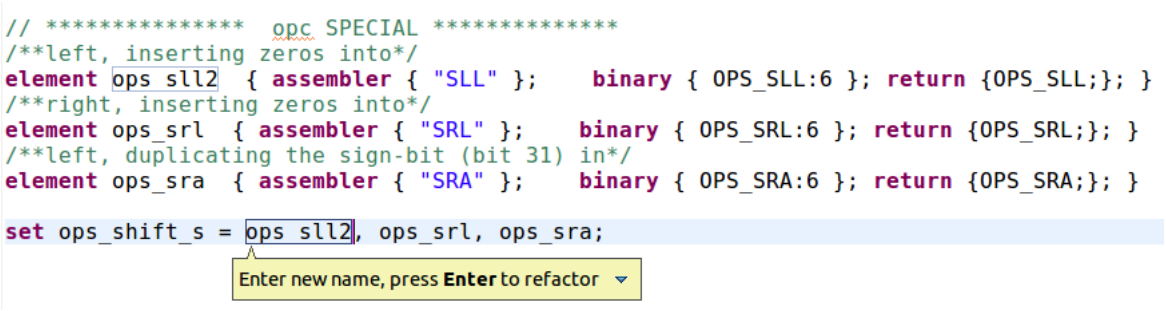
Obrázek 8.1: Odkázání se na deklaraci instrukce `ops_sll`

lexém a zjistit, jakým uzlem AST je tento lexém reprezentovaný. V případě, že se jedná o uzel `IASTName`, pokusí se vyhledat jeho binding a odpovídající deklaraci. V případě nálezu provede navigaci k dané deklaraci (respektive deklarátoru).

8.2.2 Propojení výskytů stejných proměnných a jejich refaktorizace

Dalším přínosem modelu vazeb je možnost vyhledávání všech výskytů konkrétní proměnné. Při označení proměnné jsou ve zdrojovém kódu podsvíceny všechny proměnné stejné deklarace. V postranní liště editoru jsou rovněž zvýrazněny všechny místa, kde je proměnná v aktuálním zdrojovém kódu použita. Tento doplněk přináší uživateli větší přehlednost ve zdrojovém kódu.

Díky tomu, že je možné hromadně vybrat proměnné stejné deklarace, může být provedeno jejich hromadné přejmenování. Akce pro refaktorizaci je vyvolána například prostřednictvím klávesové zkratky `SHIFT+ALT+R`. Uživatel má poté možnost provést přejmenování aktuálně vybrané proměnné. Při zadávání nového názvu se mění název všech identifikátorů provázaných s aktuálně přejmenovávanou proměnnou. Po stisku klávesy `ENTER/ESC` je akce dokončena/vrácena zpět.



Obrázek 8.2: Refaktorizace názvu instrukce `ops_sll`

Hromadné přejmenování zajišťuje akce `RefactoringAction`, která volá metodu `run` třídy `CRefactoringAction`. Cílem metody `run` je vyhledat s pomocí modelu vazeb deklaraci přejmenovávané proměnné, zjistit její rozsah (scope), vyhledat v něm veškeré výskyty identifikátorů dané deklarace a nahradit text těchto identifikátorů za text zadaný uživatelem.

Pro úspěšnou činnost refaktorizace, jsem musel modifikovat projekt CDT tak, aby uměl pracovat i se soubory s příponami jazyka CodAL. Nástroj pro hromadné přejmenování zkoumá závislosti mezi soubory a případné konflikty způsobené přejmenováním.

8.3 Tvorba modelu objektů

Druhý rozšiřující model popisuje strukturu, která nese přehled o všech jednotkách, které jsou definovány ve zdrojovém kódu. Tato struktura představuje strom podobný AST s tím rozdílem, že uzly jsou tvořeny pouze objekty, které reprezentují logický celek (deklarace funkce, definice funkce, makra, ...). To umožňuje pracovat se zdrojovým kódem na větší úrovni abstrakce.

Tvorba struktury probíhá ve třídě implementující rozhraní `IContributedModelBuilder`. Toto rozhraní deklaruje metodu `parse`, která je zavolána vždy, když je nutné provést aktualizaci datové struktury. Pro každý programovací jazyk je nutné vytvořit vlastní implementaci této metody. Jejím účelem je získat abstraktní syntaktický strom a vybrat z něho uzly reprezentující deklarace nebo definice důležitých prvků a aplikovat je do nové datové struktury. Každému uzlu struktury je explicitně nastavena pozice ve zdrojovém kódu.

8.3.1 Model objektů v jazyce C

V editoru jazyka C je metoda `parse` implementovaná ve třídě `CModelBuiler2`. Do datové struktury modelu objektů jsou ukládány uzly reprezentující direktivy preprocesoru a globální deklarace a definice.

Základním objektem modelu objektů je rozhraní `ICElement`. Toto rozhraní má podobný význam jako rozhraní `IASTNode` v modelu AST popsané v 7.2.1. Obsahuje základní metody pro práci se všemi typy uzlů modelu objektů (nastavení jména, typu, rodiče atd.). Dále obsahuje konstanty, které tvoří výčet konkrétních typů odvozených ze základního rozhraní `ICElement`.

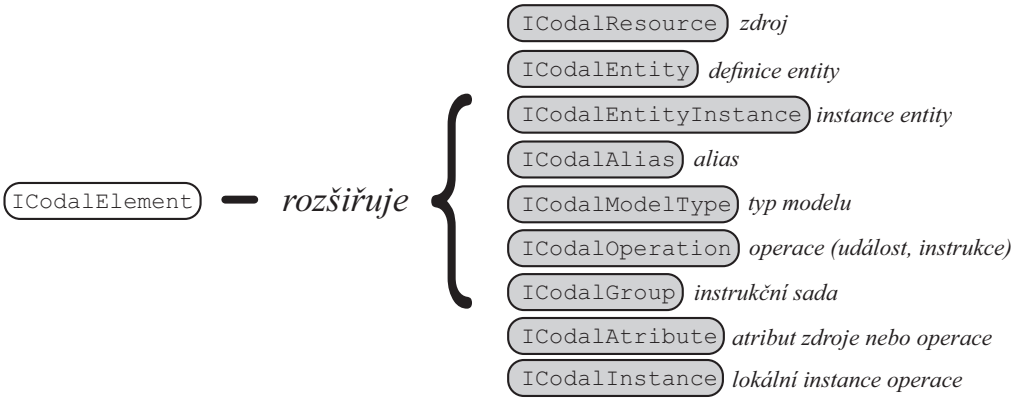
Z modelu objektů jazyka C jsem použil následující odvozená rozhraní:

- Pro direktivy preprocesoru jsem převzal typy uzlů `IInclude` a `IMacro`.
- Pro deklarace funkcí a globálních proměnných, které se mohou vyskytovat například v hlavičkových souborech jazyka CodAL, jsem využil rozhraní `IFunctionDeclaration` a `IVariableDeclaration`.

8.3.2 Model objektů v jazyce CodAL

Model objektů je v jazyce CodAL vytvářen ve třídě `CodalModelBuilder`. Tato třída rozšiřuje třídu `CModelBuilder2`, prostřednictvím které vytváří uzly pro direktivy preprocesoru a deklarace funkcí a globálních proměnných jazyka C. Pro potřeby jazyka CodAL založil člen vědecké skupiny Lissom – Ing. Ilčík rozhraní `ICodalElement` rozšiřující základní rozhraní `ICElement`. Toto rozhraní jsem použil jako výchozí pro implementaci odvozených typů uzlů modelu struktury určeného pro jazyka CodAL.

Oproti modelu vazeb se model objektů liší v tom, že modeluje všechny globální deklarace. V modelu vazeb byly důležité pouze ty, které obsahují deklarátor. Model objektů obsahuje například atributy zrojů a operací. Jednotlivé typy rozhraní jsou zobrazeny na obrázku 8.3.2.



Obrázek 8.3: Rozhraní rozšiřující základní rozhraní `ICodalElement`

8.3.3 Propojení s modelem vazeb

Při tvorbě struktury modelu objektů je v jazyce CodAL vyžadována struktura modelu vazeb. Konkrétně se tak děje při inicializaci uzlu repzentujícího zdroje: `IResource`. Jedním z atributů uzlu implementujícího toto rozhraní je typ zdroje (např. `register`). Tato informace je zjištěna ze specifikátoru deklarace v AST.

Výjimku tvoří aliasy, u kterých není typ zdroje uveden. Je roven typu zdroje, který je mu přiřazen v inicializátoru. V takovém případě je nutné specifikátor přiřazovaného zdroje dohledat. K tomu je nutné nejprve najít deklaraci tohoto zdroje. To je provedeno prostřednictvím modelu vazeb vyhledáním patřičného binding.

8.4 Využití modelu objektů

Model objektů je využit v situacích, kdy je nutné mít přehled o aktuálních globálních objektech zdrojového kódu. Ve většině případů je požadavek spojen s vyhledáním konkrétního objektu.

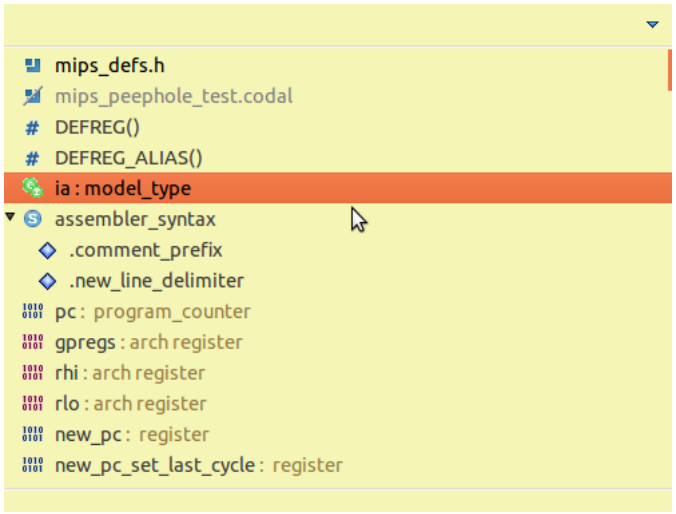
8.4.1 Nástroj Outline

Nástroj Outline je primárně reprezentován pohledem, který je podobný pohledu DOM AST Viewer (viz 7.1.4). Na rozdíl od tohoto pohledu nezobrazuje všechny uzly AST, ale pouze základní strukturu zdrojového kódu skládající se z globálních deklarací, definicí a direktiv preprocesoru. Uživatel má možnost si řadit jednotlivé položky pohledu abecedně a třídit, což mu pomáhá se rychleji orientovat ve zdrojovém kódu. Vybraním některé z položek je přesměrován na konkrétní konstrukci ve zdrojovém kódu.

Outline je dále k dispozici ve formě vyskakovacího okna, které je možné vyvolat stiskem klávesové zkratky `CTRL+O`. Vyskakovací okno se objeví na místě kurzoru. Obsahuje textové pole zadání objektu, který má být vyhledán. Díky tomu se může uživatel rychle dostat ke konkrétnímu objektu.

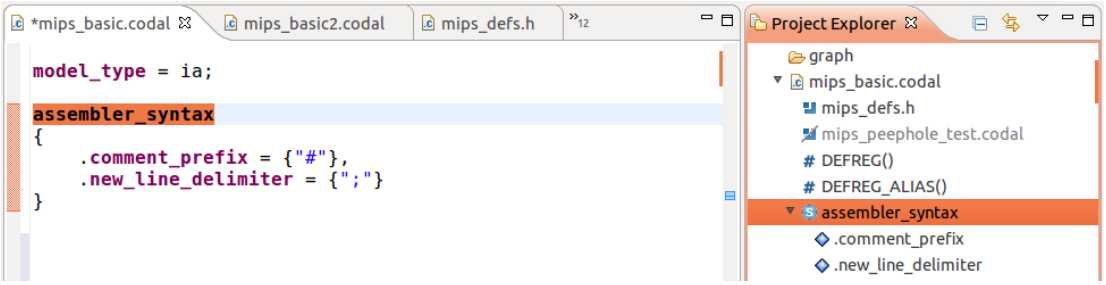
8.4.2 Programové nasměrování na objekt

Velkou výhodou modelu objektů je, že každý prvek struktury má nastaveny pozice ve zdrojovém kódu. V závislosti na typu jsou nastavovány například pozice identifikátorů, speci-



Obrázek 8.4: Nástroj Outline vyvolaný klávesovou zkratkou `CTRL+O`

fikátorů, těla apod. Díky tomu je možné vytvářet programové směrování ke konkrétním blokům kódu. Editor může být proto propojen s jinými součásti Eclipse. Využito je toho například ve výpisech konzole, v které jsou u každého výpisu dostupné odkazy na konkrétní objekt nebo řádek. Touto problematikou se zabýval člen vědecké skupiny Lissom – Ing. Ilčík.



Obrázek 8.5: Propojení pohledu Package Explorer s objekty zdrojového kódu

Kapitola 9

Dosažené výsledky

V rámci diplomové práce jsem implementoval editor, který:

1. zpracovává zdrojové kódy jazyka CodAL,
2. barevně odlišuje různé typy lexémů,
3. nabízí klíčová slova pomocí nástroje Content assist a doplňuje je do zdrojového kódu,
4. poskytuje šablony (*templates*) pro vkládání složitých konstrukcí,
5. automaticky odsazuje zanořené bloky zdrojového kódu,
6. zohledňuje direktivy preprocesoru,
7. zvýrazňuje lexikálně a syntakticky chybné konstrukce,
8. umožňuje se odkazovat na deklarace identifikátorů (zkratka `F3`),
9. propojuje výskyty identifikátorů stejné deklarace a umožňuje jejich refaktorizaci,
10. poskytuje přehled struktury zdrojového kódu (pohled Outline, zkratka `CTRL+O`),
11. umožňuje vyhledávat a programově se odkazovat na logické objekty zdrojového kódu.

V prosinci 2012 byl ve vývojovém prostředí Codasip Studio integrován editor podporující funkcionalitu 1-7. Zbytek byl implementován do verze vydané v březnu 2013. Testování proběhlo členy vědecké skupiny Lissom, kteří mi poskytnuli zpětnou vazbu a pomohli odhalit některé skryté chyby.

V následujíci sekci je editor zhodnocen z hlediska jeho výkonu.

9.1 Zhodnocení časové a paměťové složitosti

Nový editor jazyka CodAL se pohybuje ve stejné třídě časové a paměťové složitosti jako editor jazyka C projektu CDT. Editor reaguje v přijatelném čase na podněty uživatele (běžné činnosti jakými je například zvýraznění všech výskytů jedné proměnné ve zdrojovém kódu trvá méně jak jednu sekundu).

Mým hlavním cílem implementace bylo co nejvíce využít existujících implementací projektu CDT, což se mi povedlo. Editor jazyka CodAL je propojen s existujícími nástroji

projektu CDT (mírně modifikovanými), jakými je pohled Outline nebo nástroj pro hromadné přejmenování identifikátorů. V případě vyvolání těchto nástrojů je vykonán stejný algoritmus jako pro editor jazyka C.

Do projektu CDT jsem implementoval 3 nové objektové modely (model abstraktního syntaktického stromu, model vazeb – *bindings* a model objektů). Tyto modely vycházejí z odpovídajících objektových modelů projektu CDT. Pro rychlou práci se zdrojovým kódem je využíván model objektů, který značně redukuje abstraktní syntaktický strom. Projekt CDT pro tento model využívá dodatečně cachování a optimalizace, kterých jsem využil.

9.1.1 Měření časové složitosti parsování

Editor jazyka CodAL se od editoru jazyka C liší v parseru, který používá pro vytváření abstraktního syntaktického stromu. Parser jazyka CodAL umí mimo konstrukcí jazyka CodAL parsovat podmnožinu jazyka C. Cílem měření bylo ověřit, zda časová složitost parseru jazyka CodAL není vyšší než složitost parseru jazyka C.

Pro měření jsem použil dva vzorové soubory (zdrojový kód architektury MIPS v jazyce CodAL a soubor obsahující základní výpočty s maticemi v jazyce C) a sledoval, jak vzroste čas při zvyšování počtu řádků. Neporovnával jsem přímo časy zpracování obou typů souborů pro stejný počet řádků, jelikož obsahují naprosto odlišné konstrukce[1]. Rovněž mě nezajímaly ani konkrétní hodnoty, jelikož jsem měření prováděl pomocí systémové funkce jazyka Java pro měření času a nemohl jsem tedy získat naprosto přesné hodnoty. Zajímal jsem se pouze o nárůst průměrné doby. Časy trvání parsování jsou pro oba typy souborů uvedeny v tabulkách 9.1 a 9.2.

Počet řádků souboru	**Doba parsování**
1 669 (1x)	15 907 594 ns (1x)
8 345 (5x)	51 466 314 ns (3,2x)
16 690 (10x)	108 370 052 ns (6,8x)
25 035 (15x)	194 620 970 ns (12,2x)
33 380 (20x)	304 649 313 ns (19,15x)
83 450 (50x)	519 572 637 ns (32,66x)

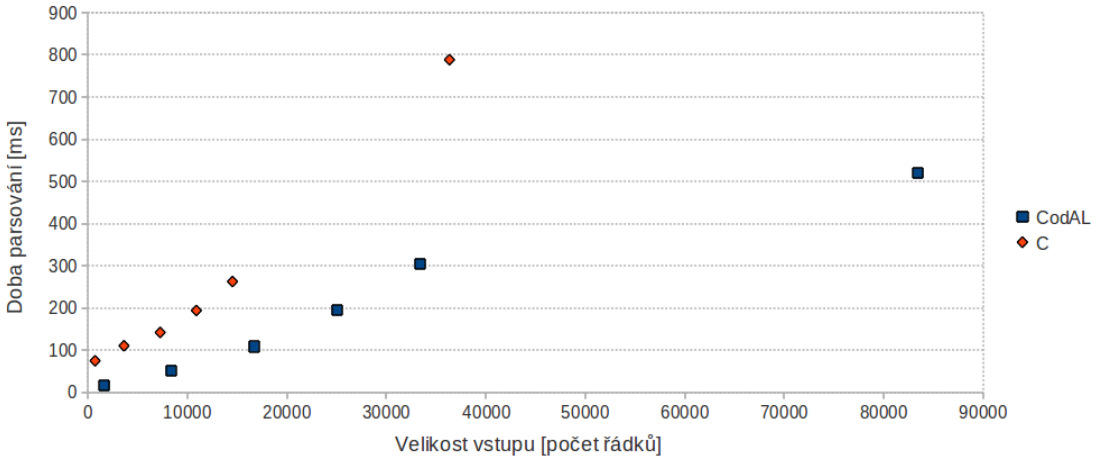
Tabulka 9.1: Časy parsování souboru v jazyce CodAL

Počet řádků souboru	**Doba parsování**
727 (1x)	74 809 938 ns (1x)
3 635 (5x)	110 411 510 ns (1,4x)
7 270 (10x)	142 015 457 ns (1,9x)
10 905 (15x)	194 154 717 ns (2,6x)
14 540 (20x)	262 621 513 ns (3,5x)
36 350 (50x)	788 485 067 ns (10,5x)

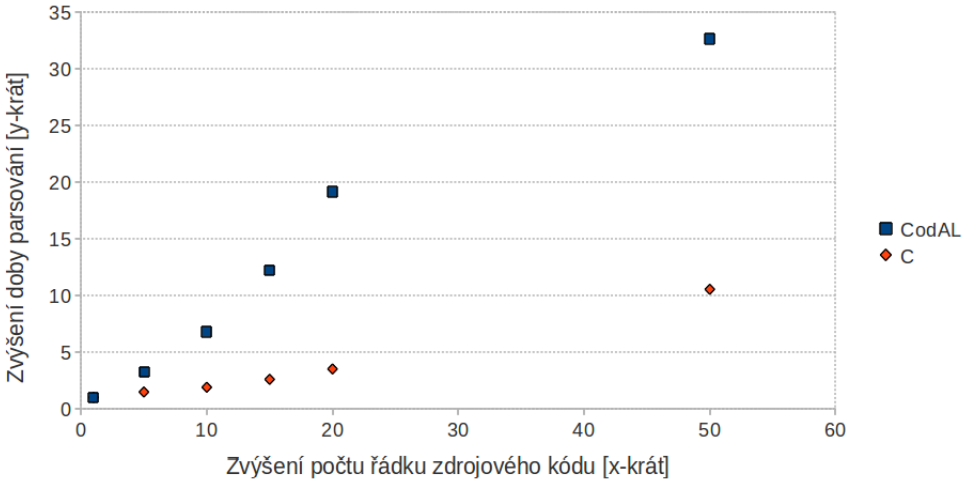
Tabulka 9.2: Časy parsování souboru v jazyce C

[1] Při porovnání rychlosti zpracování dvou souborů jazyků CodAL a C o stejné délce je zpracován rychleji zpravidla soubor v jazyce CodAL. Je to způsobeno tím, že soubor v jazyce C obsahuje průměrně na jednom řádku více informací (více uzlů AST). Zdrojový kód jazyka CodAL se skládá převážně z deklarací, které jsou ve většině případů rozděleny na několik řádků pro lepší přehlednost.

Zvyšováním počtu řádků jsem provedl duplikováním všech řádků zdrojového kódu. Z naměřených hodnot pro jednotlivé délky souborů jsem vypočítal aritmetický průměr a určil, kolikrát se tato doba zvýšila oproti průměrné době parsování zdrojového kódu bez duplikovaných řádků. Získané hodnoty jsem vynesl do grafů, které je znázorněny na obrázcích 9.1 a 9.2.



Obrázek 9.1: Doba parsování v závislosti na počtu řádků



Obrázek 9.2: Závislost nárůstu doby parsování na zvyšování délky zdrojového kódu

Doba parsování stoupá lineárně se vzrůstajícím počtem řádku pro oba typy souborů. Byla tedy zachována třída složitosti. Doba parsování u souborů jazyka CodAL však narůstá o něco rychleji. To je způsobeno tím, že gramatika jazyka CodAL je složitější oproti gramatice jazyka C (gramatika jazyka CodAL obsahuje podmnožinu gramatiky jazyka C).

9.1.2 Práce s velkými soubory

Přestože doba parsování zdrojového kódu narůstá lineárně se zvyšující se velikostí vstupu, editor může mít problémy s velmi velkými soubory. Je to způsobeno operacemi, které provádějí analýzu zdrojového kódu (především pak dodatečnou statickou analýzu).

Editor jazyka CodAL v tomto případě obstál lépe jak editor jazyka C, jelikož podporuje pouze podmnožinu těchto nástrojů. Je však i tak výhodné využít optimalizací CDT, které od určitého počtu řádků zdrojového kódu budou ignorovat některé akce nástrojů editoru. Tyto optimalizace je možné zadat v nastavení editoru.

9.1.3 Měření využití paměti

Pro měření velikosti využité paměti jsem použil nástroj Eclipse MAT – Memory Analyser [8]. Jedná se o plug-in, který umí zaznamenat aktuální obsah paměti běžící aplikace a provést jeho analýzu. Prováděl jsem analýzu běžící aplikace Eclipse a z naměřených výsledků jsem si filtroval pouze ty, které mají souvislost s běžícím editorem. Jednalo se především o instance tříd balíků:

- `com.codasip.editor.codal.*` (dále jen `codal.*`),
- `org.eclipse.cdt.*` (dále jen `cdt.*`),
- `java.*`
- a řetězce znaků – `char[]`.

Některé řetězce znaků a instance tříd balíku `java.*` mohou být alokovány jinými běžícími plug-iny. Mým cílem však bylo sledovat změnu využívané paměti při změně velikosti vstupu. Tato skutečnost mě proto nevadila.

Nejprve jsem provedl měření pro tři situace – pro prázdný soubor, pro soubor jazyka CodAL modelující architekturu MIPS (1 669 řádků) a pro soubor jazyka C určený pro základní operace s maticemi (727 řádků). Naměřené hodnoty jsou znázorněny v tabulkách 9.3, 9.4 a 9.5.

Skupina objektů	Počet objektů	Velikost v paměti
`codal.*`	70	1 976 bytů
`cdt.*`	10 434	708 224 bytů
`java.*`	438 381	16 470 832 bytů
`char[]`	231 771	13 439 120 bytů

Tabulka 9.3: Analýza využité paměti pro prázdný soubor

Skupina objektů	Počet objektů	Velikost v paměti
`codal.*`	70	1 976 bytů
`cdt.*`	107 793	4 702 840 bytů
`java.*`	471 839	17 522 592 bytů
`char[]`	242 752	15 346 136 bytů

Tabulka 9.4: Analýza využité paměti pro soubor jazyka C

Skupina objektů	Počet objektů	Velikost v paměti
`codal.*`	8 040	430 664 bytů
`cdt.*`	47 713	2 174 328 bytů
`java.*`	470 387	17 557 464 bytů
`char[]`	241 460	14 548 608 bytů

Tabulka 9.5: Analýza využité paměti pro soubor jazyka CodAL

Naměřené výsledky potvrzují, že jeden řádek jazyka CodAL obsahuje v průměru méně uzlů AST jak jeden řádek souboru jazyka C. Zpracovaný soubor jazyka CodAL zabírá méně místa v paměti jak soubor jazyka C i přesto, že je více jak 2x delší. U výsledků souboru jazyka CodAL je dále možné sledovat, jak moc je využívána knihovna CDT.

Další měření jsem prováděl pouze na souboru jazyka CodAL o velikosti 1 669 řádků. Sledoval jsem, jak moc vzroste využití paměti při zvyšování počtu řádků vstupního souboru. Využití paměti jednotlivých typů objektů je znázorněno v tabulkách 9.6, 9.7, 9.8 a 9.9.

Velikost souboru	Počet objektů	Velikost v paměti
1x	8 040	430 664 bytů
2x	16 005	859 272 bytů
4x	31 935	1 716 488 bytů
8x	63 795	3 430 920 bytů
16x	127 515	6 859 784 bytů

Tabulka 9.6: Analýza využité paměti balíkem `codal.*`

Velikost souboru	Počet objektů	Velikost v paměti
1x	47 078	2 148 736 bytů
2x	85 932	3 668 328 bytů
4x	163 062	6 693 872 bytů
8x	317 217	12 742 152 bytů
16x	625 475	24 841 048 bytů

Tabulka 9.7: Analýza využité paměti balíkem `cdt.*`

Velikost souboru	Počet objektů	Velikost v paměti
1x	540 467	19 685 360 bytů
2x	581 848	21 289 696 bytů
4x	590 765	21 672 464 bytů
8x	606 228	22 349 744 bytů
16x	624 536	23 283 192 bytů

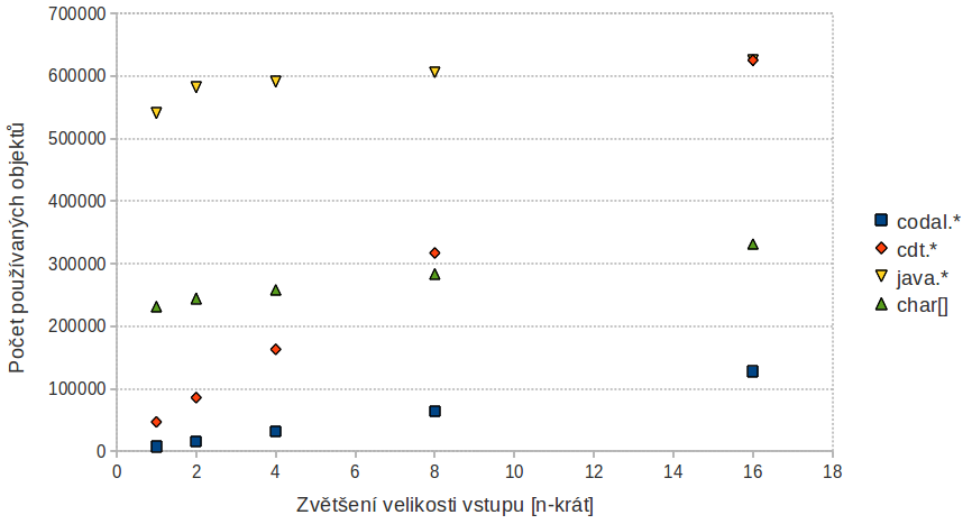
Tabulka 9.8: Analýza využité paměti balíkem `java.*`

Velikost souboru	Počet objektů	Velikost v paměti
1x	231 284	17 111 152 bytů
2x	244 892	18 918 136 bytů
4x	257 969	20 831 728 bytů
8x	283 646	24 624 368 bytů
16x	331 212	31 918 304 bytů

Tabulka 9.9: Analýza využité paměti řetězci – `char[]`

Hodnoty pro řetězce `char[]` a balíky `java.*` se u základní velikosti souboru liší oproti naměřeným výsledkům tabulky 9.5. Je to způsobeno nezávislostí obou měření. V obou případech byl jinými plug-iny prostředí Eclipse alokován odlišný počet objektů.

Naměřené hodnoty jsem vynesl do grafu který je znázorněný na obrázku 9.3.



Obrázek 9.3: Závislost nárůstu využití paměti na zvyšování délky zdrojového kódu

Počet instancí vzrůstá lieárně se zvyšováním počtu řádků vstupního souboru. Vždy je nutné zohlednit fixní počet instancí, které jsou vytvořeny i v případě, kdy je vstupní soubor prázdný.

Kapitola 10

Závěr

Tato diplomová práce se zabývala návrhem a tvorbou nového editoru jazyka CodAL v prostředí Eclipse.

V teoretické části jsem popsal jazyk CodAL určený k rychlému modelování víceprocesorových systémů na čipu (MPSoC). Představil jsem vývojové prostředí projektu Lissom, který obsahoval nevyhovující editor jazyka CodAL. Popsal jsem architekturu vývojového prostředí Eclipse, na kterém je vývojové prostředí projektu Lissom založeno. Rozebral jsem problematiku editorů a zpracování zdrojového kódu.

Ve druhé části diplomové práce jsem se soustředil na praktickou část. Navrhl jsem nový editor založený na projektu Eclipse CDT. Implementaci jsem rozdělil do tří etap. V první etapě jsem vytvořil vstup pro generátor parseru (LALR Parser Generator) a vygeneroval základní verzi parseru. V druhé etapě jsem se soustředil na modelování abstraktního syntaktického stromu. AST mi posloužil ve třetí etapě, v které jsem ho dále analyzoval. Editor jsem propojil s existujícími nástroji projektu CDT sloužící pro analýzu zdrojového kódu.

10.1 Další možná rozšíření

Implementované objektové modely editoru hrají důležitou roli při analýze zdrojových kódů. Díky nim je možné provádět dodatečné analýzy hodnotící především sémantické vlastnosti jazyka. Dalším nástrojem, který je plánován přidat do editoru jazyka CodAL, je pokročilý Content assist. Tento nástroj bude provádět statickou analýzu zdrojového kódu [2]. Při chybách budou uživateli nabízeny možnosti opravy těchto chyb (prostřednictvím nástroje Quick fix). Člen vědecké skupiny Lissom – Ing. Ilčík již prováděl experimenty s knihovnou Codan, která je pro tuto činnost určená.

Díky tomu, že je editor jazyka CodAL založen na projektu CDT, je možné využívat rovněž aktualizací tohoto projektu. Je ovšem nutné dbát na to, aby byly objektové modely jazyka CodAL stále kompatibilní s objektovými modely projektu CDT.

10.2 Přínos diplomové práce

Díky diplomové práci jsem si rozšířil znalosti v oblasti teorie překladačů, editorů a analýzy zdrojových kódů. Seznámil jsem se s projekty a nástroji pro implementaci vlastního editoru podporujícího pokročilé funkce analýzy zdrojových kódu. Podílel jsem se na zajímavém projektu, který bude dále využíván a rozšiřován.

Literatura

[1] APS BRNO s.r.o.: *CodAL Manual : reference guide, version 3.2*. Brno, 2011.

[2] Binh, L. T.; Viet, H. A.; Toan, P. X.; aj.: *C/C++ source code analyzing with CDT*. 2007 [cit. 2013-03-31].

[3] Calypto Design Systems, I.: Catapult: Product Family Overview [online]. http://calypto.com/en/products/catapult/overview, 2013 [cit. 2013-04-20].

[4] Clayberg, E.; Rubel, D.: *Eclipse : Building Commercial-Quality Plugins*. Boston: Addison-Wesley Professional, druhé vydání, 2008, ISBN 987-0-321-42672-7.

[5] Codasip: CodAL Architecture Description Language [online]. http://www.codasip.com/products/codal/, 2013 [cit. 2013-04-20].

[6] Codasip: Codasip Studio [online]. http://www.codasip.com/products/studio/, 2013 [cit. 2013-04-20].

[7] Eclipse: Eclipse [online]. http://www.eclipse.org, 2013 [cit. 2013-04-20].

[8] Eclipse Foundation: Eclipse Memory Analyser [online]. http://www.eclipse.org/mat/, 2013 [cit. 2013-04-26].

[9] Eclipsepedia: CDT/designs/C99 and UPC Parser Overview [online]. http://wiki.eclipse.org/CDT/designs/C99_and_UPC_Parser_Overview, 2013 [cit. 2013-03-31].

[10] Flex: The Fast Lexical Analyzer [online]. http://flex.sourceforge.net/, 2008 [cit. 2013-04-20].

[11] GNU: Bison – GNU parser generator [online]. http://www.gnu.org/software/bison/, 2012 [cit. 2013-04-20].

[12] Hynek, J.: *Profilovací perspektiva v prostředí Eclipse*. Brno: Vysoké učení technické, Fakulta informačních technologií, 2011.

[13] Kolář, D.: *Principy programovacích jazyků a OOP : IPP*. Brno: Vysoké učení technické, Fakulta informačních technologií, 2008.

[14] LALR Parser Generator: *Getting Started with LPG*. 2006.

[15] LALR Parser Generator: *Using LALR Parser Generator*. 2006.

[16] LALR Parser Generator: About project [online]. http://lpg.sourceforge.net/, 2013 [cit. 2013-04-20].

[17] Lissom: Lissom [online]. http://www.fit.vutbr.cz/research/groups/lissom/, 2013 [cit. 2013-04-20].

[18] Scarpino, M.: Building a CDT-based editor [online]. http://www.ibm.com/developerworks/opensource/library/os-ecl-cdt1/index.html, 2006 [cit. 2013-03-31].

[19] The Eclipse Foundation: CDT Programmer's Guide [online]. http://help.eclipse.org/helios/topic/org.eclipse.cdt.doc.isv/guide/index.html, 2012 [cit. 2013-01-01].

[20] The Eclipse Foundation: Eclipse Documentation [online]. http://help.eclipse.org/, 2012 [cit. 2013-01-01].

[21] The Eclipse Foundation: The JFace UI framework [online]. http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/guide/jface.htm, 2012 [cit. 2013-01-01].

[22] The Eclipse Foundation: Eclipse CDT (C/C++ Development Tooling) [online]. http://www.eclipse.org/cdt/, 2013 [cit. 2013-04-20].

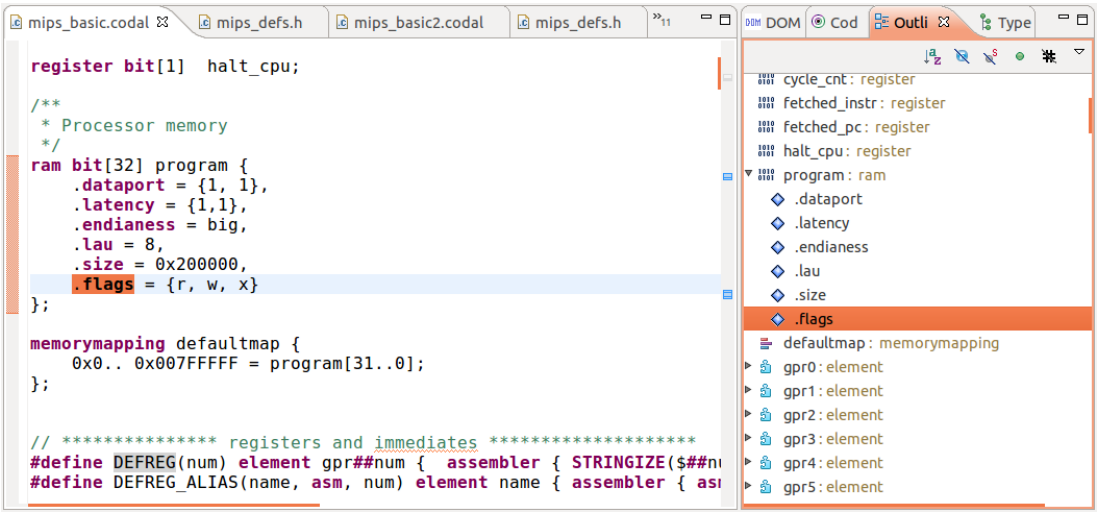
[23] Xtext: Language Development made Easy! [online]. http://www.eclipse.org/Xtext/, 2013 [cit. 2013-04-20].

[24] Češka, M.: *Sktipta Teoretická informatika*. Brno: Vysoké učení technické, Fakulta informačních technologií, 2002.

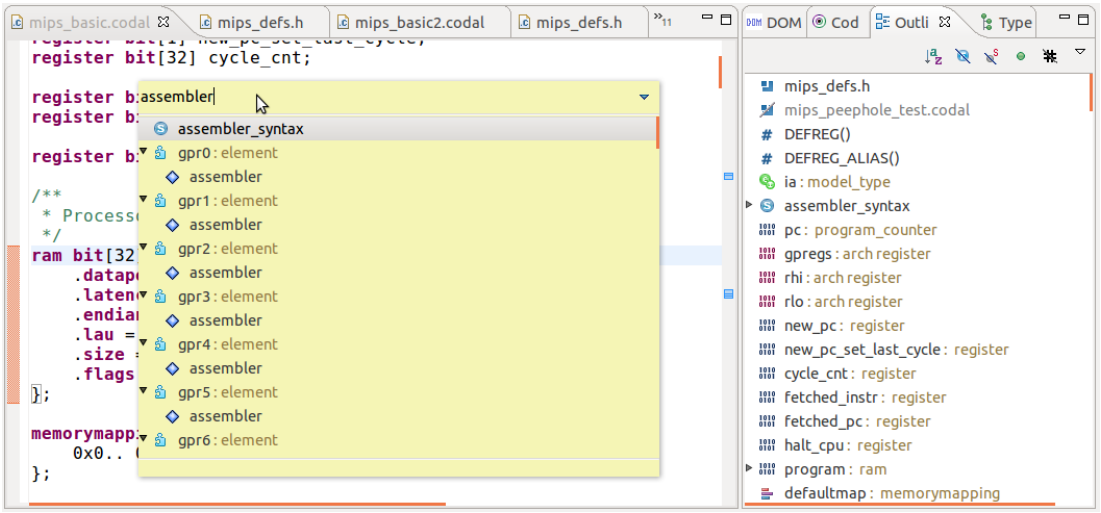
[25] Češka, M.; Vojnar, T.; Smrčka, A.: *Teoretická informatika*. Brno: Vysoké učení technické, Fakulta informačních technologií, 2011.

Příloha A

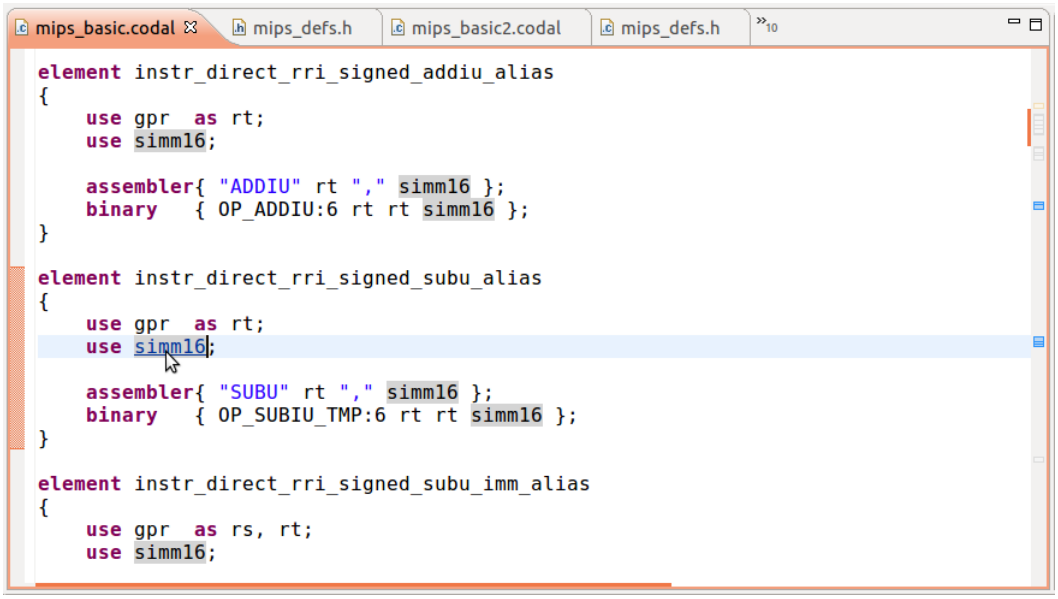
Snímky editoru



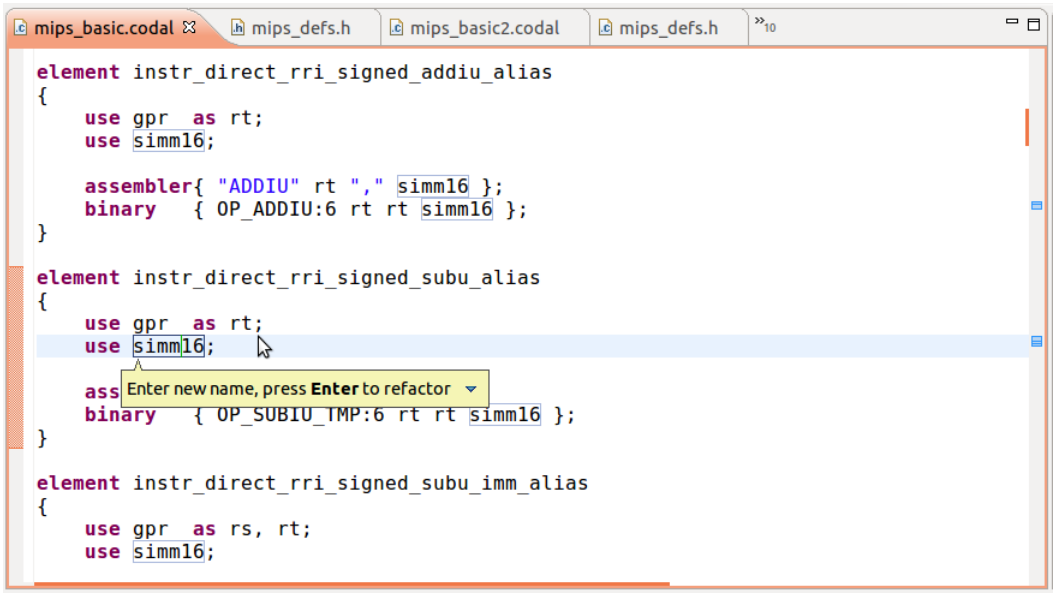
Obrázek A.1: Nástroj Outline



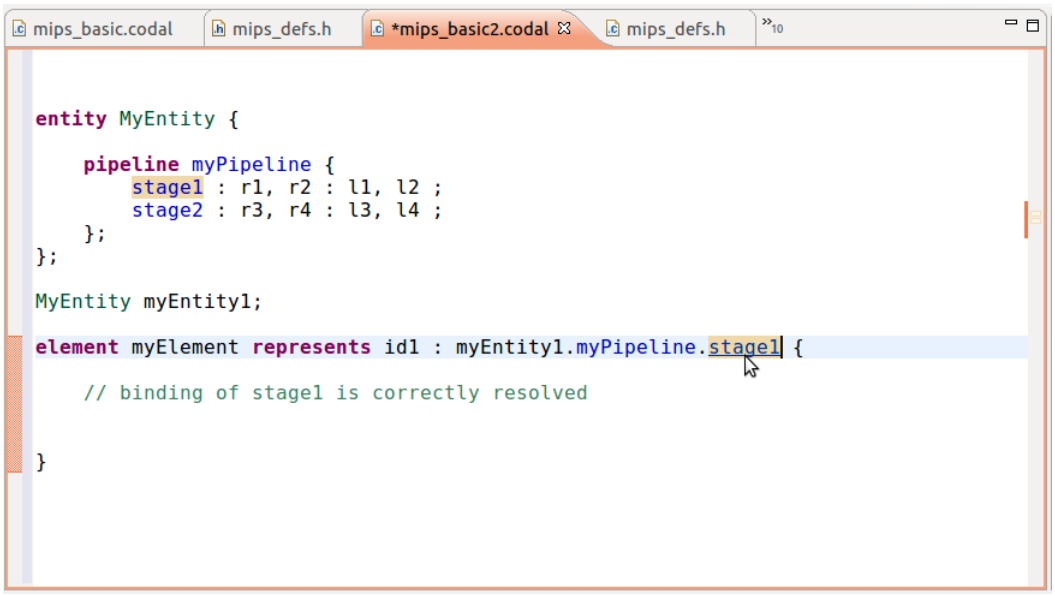
Obrázek A.2: Nástroj Quick Outline (zkratka `CTRL+O`)



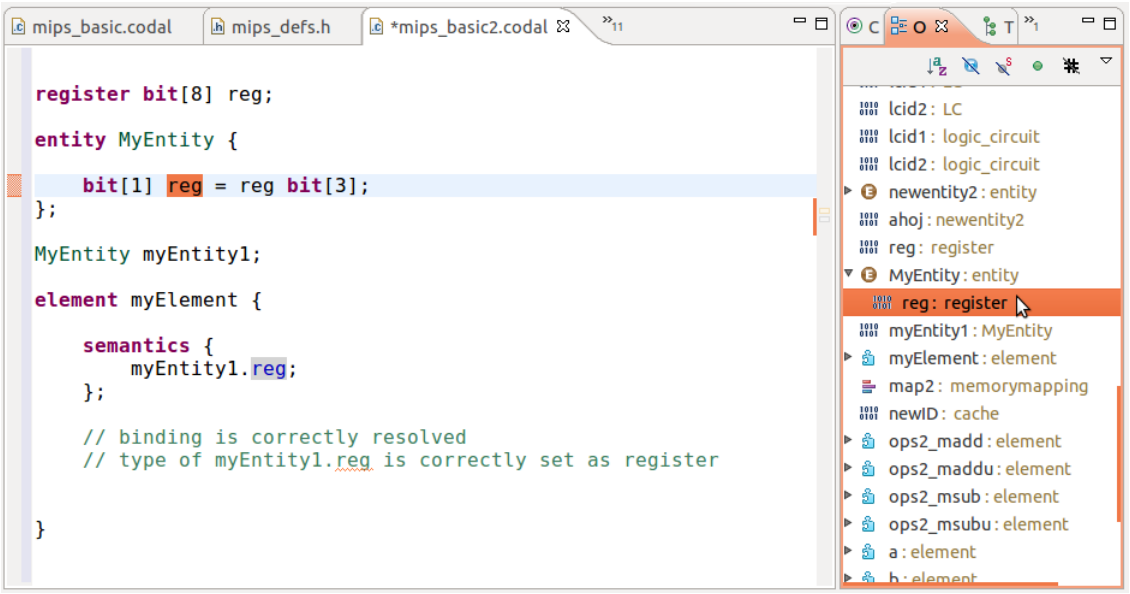
Obrázek A.3: Provázání výskytů proměnných stejné deklarace.



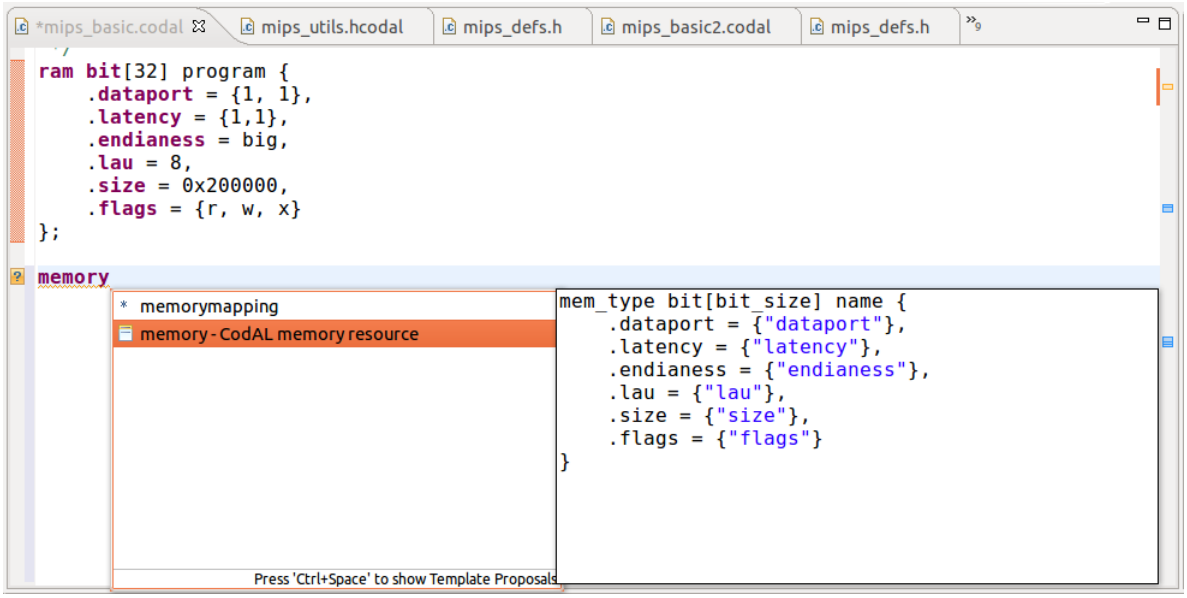
Obrázek A.4: Hromadné přejmenování proměnných



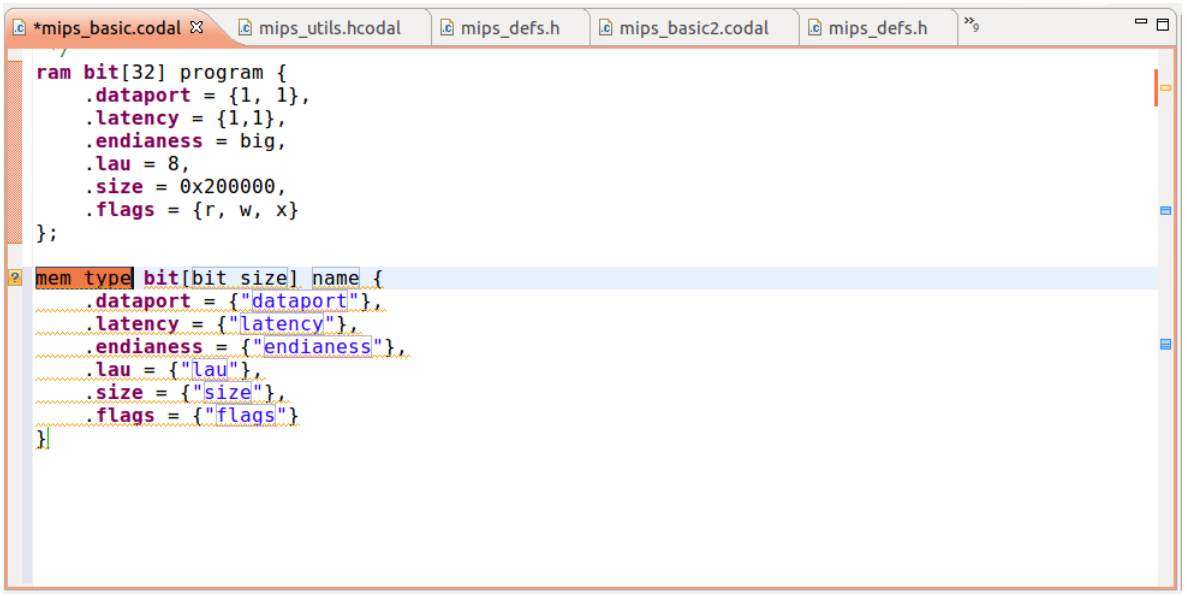
Obrázek A.5: Provázání výskytu fáze zřetězené linky s deklarací v entitě



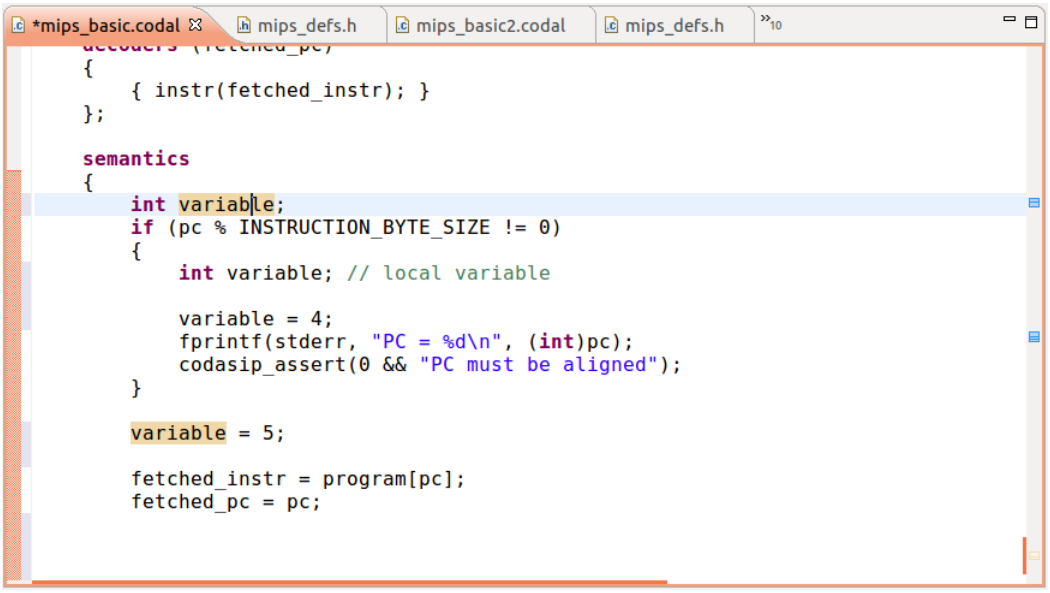
Obrázek A.6: Provázání výskytu registru s deklarací aliasu registru



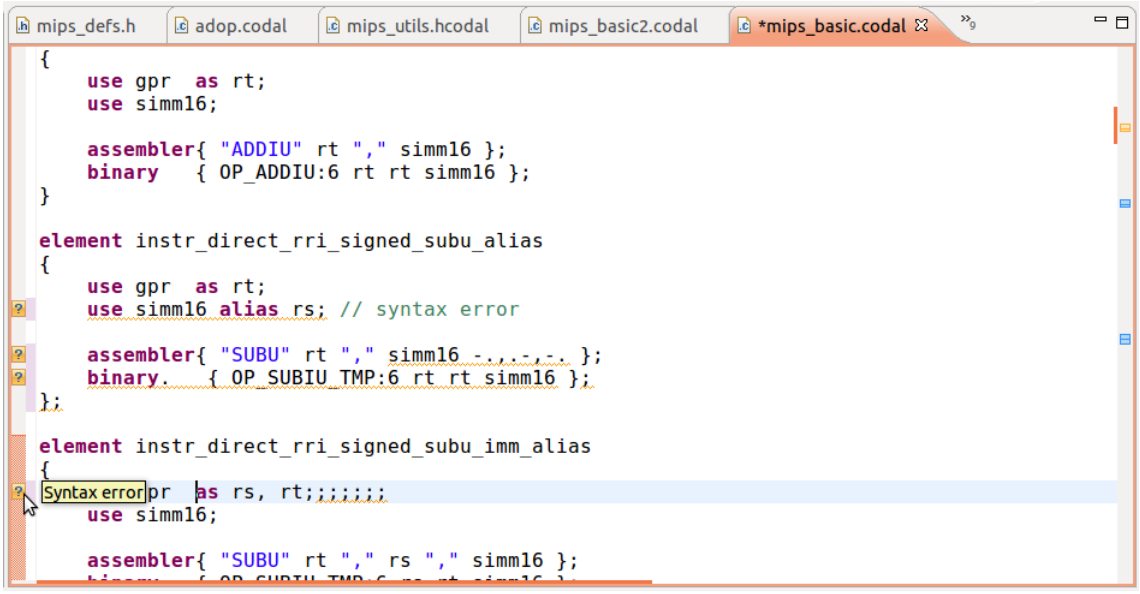
Obrázek A.7: Nabídnutí šablony pomocí nástroje Content assist



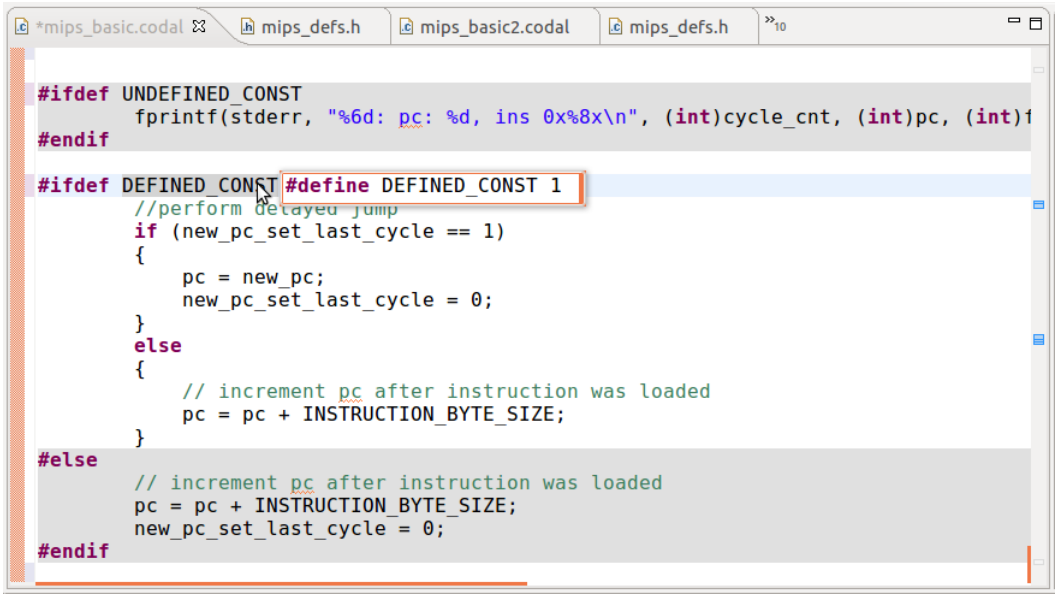
Obrázek A.8: Vložení šablony do zdrojového kódu



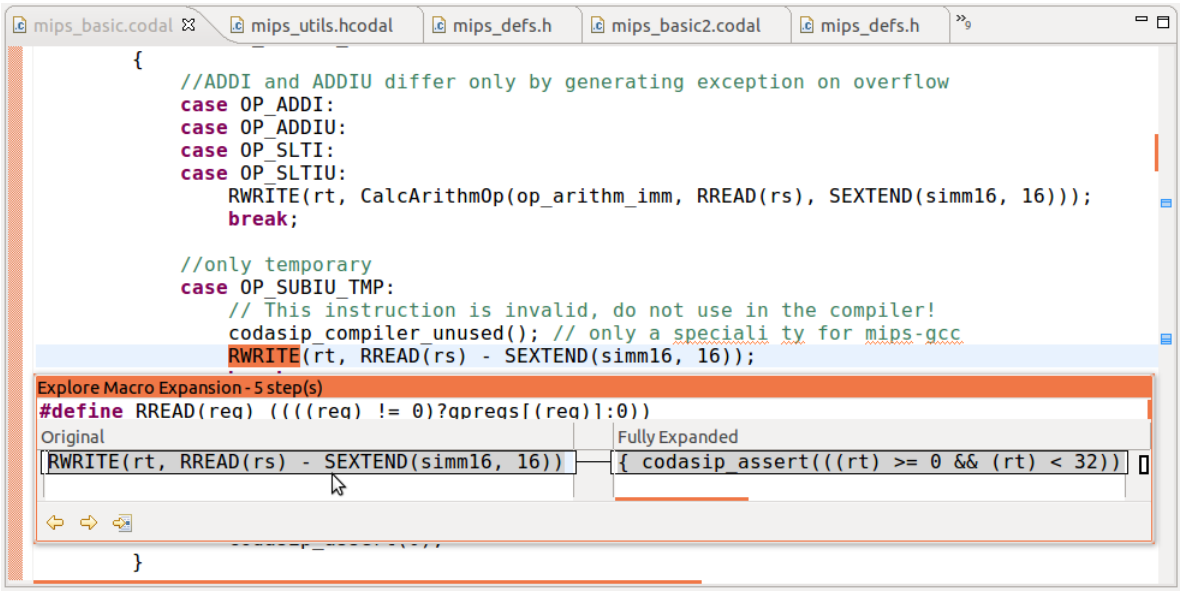
Obrázek A.9: Ukázka zohlednění rozsahu vititelnosti proměnné.



Obrázek A.10: Zvýraznění chyb ve zdrojovém kódu



Obrázek A.11: Použití direktivy preprocesoru `#ifdef`



Obrázek A.12: Ukázka expanze makra