# Qt

Qt je multi-platformní open-source knihovna sloužící k vývoji především GUI aplikací v jazyce C++. Aplikace se vyznačují především nativním vzhledem GUI. Jednou z výhod této knihovny je, že dosahuje výkonu nativních aplikací. Součástí Qt je Qt Quick, což je soubor technologií sloužících k tvorbě deklarativního GUI.

Qt poskytuje vývojové prostředí Qt creator, které lze použít pro vývoj C++ aplikací využívající prvky Qt nebo aplikací Qt Quick. Součástí Qt creator je vizuální debuger C++

## Qt Quick

Qt Quick je soubor technologií sloužících k tvorbě deklarativního GUI. Poskytuje sadu GUI prvků, deklarativní jazyk QML a modul Qt declarative. Qt declarative je interpret jazyka JavaScript, který umožňuje spuštění QML aplikací s backendem založeným na Qt. Qt declarative odděluje logiku QML definovaného GUI od aplikační logiky C++. K datům vytvořených v QML lze přistupovat, v C++ části aplikace a naopak.

## QML

QML je jazyk vycházející z jazyka JavaScript. Jedná se o jazyk pro deklaraci hierarchického GUI, na vrcholu hierarchie je vždy jediný prvek a všechny ostatní prvky jsou na něj navázány ve stromové struktuře. Každý element může mít teoreticky neomezené množství potomků (elementy, jež se nacházejí v hierarchii přímo pod ním), avšak může mít nanejvýš jednoho rodiče (element, který se nachází v hierarchii přímo nad ním). Pouze kořenový prvek na vrcholu hierarchie nemá žádného rodiče.

### Syntaxe

Syntaxe jazyka QML vychází z jazyka JavaScript a je podobná notaci JSON. Základní syntaxe je ilustrována v následujícím kódu.

Rectangle

{

width : 100; height : 100

Image

{

source : „img.jpg“

}

}

Soubor s tímto kódem vytvoří dva GUI elementy. Kořenovým prvkem je element typu Rectangle, jehož potomkem je element typu Image. Element typu Rectangle má přiřazenu hodnotu dvěma atributům width a height, zatímco element typu Image má nastaven atribut source řetězcem „img.jpg“.

### Přidávání nových atributů

QML umožňuje rozšířit existující typ elementu o přídavné atributy, pomocí klíčového slova property a zadáním datového typu atributu viz. následující kód:

Rectangle

{

property int newProp01

property int newProp02 : 5

}

### Výrazy

Hodnoty jednotlivých atributů lze definovat pomocí výrazů. Hodnoty výrazů se přepočítávají za běhu aplikace, pokud se jejich hodnota může změnit, například:

1. width : 5
2. width : height/2
3. width : parent.width/3

V prvním výrazu je hodnota nastavena na konstantní hodnotu 5.

V druhém výrazu je atribut nastaven pomocí jiného atributu height, kdykoli se tudíž za běhu programu změní hodnota atributu height, pak se změní i hodnota atributu width.

V posledním výrazu je použito klíčové slovo parent, pomocí nějž lze přistupovat k atributům nadřazeného prvku v hierarchii (tzv. rodiče). Pokud se v takovémto případě změní hodnota rodičova atributu width, dojde i k opětovnému spočtení výrazu a tak i k úpravě hodnoty width potomka. Tímto způsobem je umožněno dynamicky přizpůsobovat velikosti prvků v hierarchii v závislosti na změnách velikostí jiných prvků.

### Identifikátory objektů

Každý objekt může mít přiřazen speciální identifikátor, pomocí nějž může být k němu přistupováno. K tomu slouží speciální atribut „id“, do nějž může být přiřazen daný identifikátor. Identifikátor musí být unikátní v dané komponontě. V následujícím příkladu je ukázka použití daného atributu.

Rectangle

{

id : identifier01

width : 100

}

Rectangle

{

width : identifier01.width

}

V tomto případě je hornímu prvku nastaven atribut width na hodnotu 100 a atribut width spodního prvku je nastaven na totožnou hodnotu.

### Komponenty

GUI definované uvnitř jednoho QML souboru se označuje jako komponenta. Každou komponentu je možné znovu použít jako stavební prvky jiných komponent, obdobně jako je tomu u standardních typů elementů. Komponenty se importují automaticky, při použití jména souboru jako typu GUI elementu.

Například existuje-li soubor CustomButton.qml, pak následující kód vytvoří uvnitř obdélníka komponentu definovanou v importovaném souboru CustomButton.qml.

Rectangle

{

CutomButton {}

}

### Funkce

V QML lze pomocí klíčového slova function definovat funkci se zdrojovým kódem v jazyce JavaScript, při vyhodnocování výrazu v takovýchto funkcích platí totožná pravidla s těmi pro vyhodnocování výrazů přiřazených atributům.

Pokud místo výrazu je atributu přiřazen kód těla funkce v jazyce JavaScript ohraničený složenými závorky, bude pro vyhodnocení hodnoty atributu použita tato funkce, výsledná hodnota atributu se v takovém případě vrací pomocí klíčového slova return.

### Kontrola atributů

Kontrola existence atributů nebo kontrola datových typů je částečně řešena při zpracování QML souboru a částečně až za běhu výsledného programu. V případě přiřazení hodnot uvnitř QML

Pokud je nějakému atributu přiřazena hodnota mimo JavaScriptový kód nějaké funkce. Je kontrolována existence daného atributu již při zoracování QML souboru. Ve všech ostatních případech, ať už se jedná o přiřazení hodnoty atributu, získání hodnoty nějakého atributu jsou existence atributů kontrolovány až při jejich čtení nebo pokusu o změnu za běhu programu.

v : w.x + d;

Pokud element nemá atribut se jménem v, nastane chyba už při zpracování QML souboru. V opačném případě mohou nastat chyby až během spuštění programu. Aby byl kód validní, musí existovat buď element identifikátorem w, mající atribut x typu kompatibilním s v nebo musí element mít atribut w, který obsahuje objekt, jež má atribut x typově kompatibilní s v. Zároveň musí existovat atribut d, který je typově kompatibilní s v, přičemž musí tento typ podporovat operaci sčítání.

V případě, že existuje zároveň atribut w a element s identifikátorem w, bere se v potaz pouze element s identifikátorem w. Tudíž pokud existuje atribut w obsahující atribut x, ale zároveň existuje nějaký element s identifikátorem w, který neobsahuje atribut x, dojde k chybě, kvůli neexistenci atributu x.

# CQML – Návrh a vlastnosti

Cílem při návrhu jazyka CQML bylo zachovat, co největší podobnost s QML, co se týče jeho funkčnosti a možností, ale zároveň umožnit, jednoduchou integraci do aplikací v jazycích C,C++ a Objective C. Z tohoto důvodu byl pro výstupní jazyk vybrán jazyk C. Pro jednodušší převod do výstupního formátu je pro kódy funkcí pro obsluhu událostí a výpočtů hodnot atributů oproti jazyku QML (využívající JavaScript) použita syntaxe jazyka C. Což umožní ve výsledném kódu vygenerovaným překladačem  například zavolat funkce či přistupovat ke globálním proměnným.

Syntaxe byla zvolena podobná jazyku QML. Rozdílem je ukončující znak středníku na konci každého výrazu. Ačkoli ukončovací znak není nutný (viz. QML a JavaScript), je tímto dosaženo toho, že výsledná gramatika bude nevypouštěcí, což bude mít za následek, jednodušší zpracování CQML kódu a také přesnější chybová hlášení při syntaktické analýze.

Podobně jako v QML, GUI komponenty definované uživatelem mohou být uloženy v oddělených souborech a následně importovány a opětovně používány jako samostatné prvky v jiných GUI komponentech. Na rozdíl od QML se soubory nebudou imporrtovat automaticky, ale začátku každého CQML souboru budou definovány soubory, z nichž budou importovány GUI hierarchie a také identifikátor, jehož použitím jako typ elementu bude na dané místo umístěna importovaná komponenta.

import „CutomButton.cqml“ as Button;

Rectangle

{

Button {};

}

Programovací jazyky zpravidla nebývají bezkontextové, nicméně jejich syntaktická analýza (nikoli však sémantická) lze provést pomocí bezkontextové gramatiky. Toto platí například pro jazyk C, z jehož gramatiky se bude vycházet pro syntaktickou analýzu kódu funkcí a výrazů v CQML. Z tohoto hlediska bude i gramatika CQML bezkontextová, protože typová kontrola a kontrola existence atributů či identifikátorů nebude prováděna během syntaktické analýzy. To má za výhodu, že nebude během syntaktické analýzy potřeba použití vyhledávací tabulky pro kontrolu symbolů.

Pro gramatiku CQML byla zvolena YACC notace, vzhledem k dostupnosti technologií pro generování parseru a existenci gramatiky pro jazyk C ve formátu YACC.

Následuje soubor pravidel pro gramatiku ve formátu YACC pro jazyk CQML. Tokeny (terminály) jsou psané velkým písmem a neterminální symboly malým. Neterminály compound\_statement, type\_specifier a conditional\_expression jsou převzaty z gramatiky pro jazyk C, a jejich zpracování bude provedeno podle pravidel dané gramatiky.

start\_point

: element\_or\_import\_list

;

element\_or\_import\_list

: import\_list element

| element

;

import\_list

: import import\_list

| import

;

import

: IMPORT STRING\_LITERAL AS IDENTIFIER

;

element

: IDENTIFIER '{' attribute\_or\_subelement\_list '}'

| IDENTIFIER '{' '}'

;

attribute\_or\_subelement\_list

: attribute\_or\_element ';' attribute\_or\_subelement\_list

| attribute\_or\_element ';'

;

attribute\_or\_element

: element

| event\_handler

| attribute

| property

;

event\_handler

: IDENTIFIER ':' compound\_statement

;

property

: PROPERTY IDENTIFIER attribute

| PROPERTY type\_specifier attribute

| PROPERTY IDENTIFIER IDENTIFIER

| PROPERTY type\_specifier IDENTIFIER

;

attribute

: IDENTIFIER ':' conditional\_expression

| IDENTIFIER '.' IDENTIFIER ':' conditional\_expression

;

Gramatika jazyka CQML je víceznačná, protože pro definici funkcí a výrazů používá syntaxi jazyka C, jehož gramatika je víceznačná díky pravidlu pro podmínku IF-ELSE (viz. níže), jedná se o tzv. „dangling else problem“.

selection\_statement

: IF '(' expression ')' statement

| IF '(' expression ')' statement ELSE statement

Následující výrok může být rozparsován dvěma způsoby.

if (a) if (b) s; else s2;

1.

if(a){

if(b)

s;

else

s2;

}

2.

if(a){

if(b)

s;

}

else

s2;

Tato víceznačnost je vyřešena pomocí konvence tak, že se ELSE přiřadí k nejvnitřnějšímu IF, tudíž se použije první varianta.

# Návrh překladače

Cílem je vytvořit aplikaci překladače, která přemění kód z jazyka CQML pro deklarativní GUI, kód importovatelný do aplikace v jazyce C. Tudíž na vstupu bude soubor se zdrojovým kódem v jazyce CQML a výstupem bude zdrojový kód v jazyce C, ve formě definicí datových struktur pro GUI a funkcí. Vzhledem k tomu, že jazyk C na rozdíl od vyšších programovacích jazyků (C++, Java) nepodporuje členské metody u struktur (resp. tříd), je tento nedostatek obejit tím, že se použije funkce, která je rozšířena o jeden parametr, jako který přijímá ukazatel na objekt, který by danou metodu volal. V případě virtuálních metod je jako člen struktury použit ukazatel na funkci, která se bude volat.

Výsledný překladač je navržen tak, že lze rozdělit do několika bloků (viz. obrázek).

Cílem prvního bloku je využít zdrojový kód z CQML souboru a ověřit jeho syntaktickou správnost. V případě, že je kód syntakticky v pořádku, bude výstupem tohoto bloku syntaktický strom. Pro tento blok budou použity zdrojové kódy vygenerované pomocí aplikací Bison a Flex, z YACC gramatiky pro CQML.

V syntaktickém stromu budou nalezeny příkazy pro import souborů. Tyto soubory budou následně otevřeny a postupně zpracovány totožným způsobem.

Jakmile jsou všechny importované soubory zpracovány a jsou jejich zpracováním vytvořeny jejich syntaktické stromy, bude zjištěno, zda jsou některé soubory cyklicky závislé. To se bude realizovat pomocí konstrukce orientovaného grafu, kde každý vrchol představuje soubor a závislost mezi soubory je znázorněna hranou (směřující k importovanému souboru), a následným zjištěním zda v grafu existuje orientovaný cyklus. Pokud v grafu takový cyklus existuje, pak jsou soubory cyklicky závislé, což vyústí v chybu a ukončení programu. Je-li graf acyklický, jsou podle něho topologicky seřazeny syntaktické stromy jednotlivých souborů (vrcholy) pro následné zpracování.

V dalším bloku se zpracuje syntaktický strom a podle něj se vytvoří GUI hierarchie. Pro každý uzel představující GUI element, atribut, funkci, či nový atribut se vytvoří instance příslušné třídy. Každému elementu jsou podle jeho potomků v syntaktickém stromu přiřazeny jeho potomci v GUI hierarchii a jeho atributy a funkce.

Následně se zpracují všechny elementy s definovaným atributem ID, a tyto identifikátory se použijí ke tvorbě mapy, pomocí níž bude umožněno přistupovat k danému elementu podle definovaného identifikátoru.

V další části programu se zpracují kódy funkcí do formy potřebné pro závěrečný výstup v jazyce C.

## Zpracování kódu v jazyce C

### Zpracování identifikátorů

Kódy (resp. syntaktické stromy) jednotlivých funkcí se projdou, a naleznou se identifikátory, jež se nachází v mapě identifikátorů elementů (id atributy elementů), a tyto identifikátory se v kódu nahradí skutečným názvem elementu uvnitř výsledné GUI aplikace.

// todo návrh struktur pro zpracování C kódu

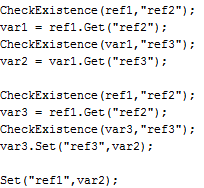
### Dynamická kontrola typů a přiřazování hodnot

Jelikož bude do atributů některých typů umožněno ukládat reference na objekty různého typu a pomocí této reference bude možné přistupovat k atributu daného objektu, nemůže být zaručeno, že atribut, ke kterému se aplikace snaží přistoupit je v daném objektu přítomen. Proto bude potřeba za běhu výsledné GUI aplikace kontrolovat, přítomnost takového atributu

Uvažujme kód z obrázku níže. Z kódu je zřejmé, že u atributu ref budepotřeba zkontrolovat, zda má objekt referovaný tímto atributem atribut ref2 a objekt referovaný ref2 obsahuje ref3.



Oproti nahrazení identifkátoru názvem elementu nestačí pouze nahradit část textového řetězce ve výrazu. Vzhledem k tomu, že kromě ověření existence atributu bude potřeba daný atribut i získat pro další zpracování, bude nutné kód rozdělit do jednotlivých částí a každý operátor tečky nahradit voláním funkce pro získání (resp. nastavení) hodnoty atributu a kontrolou existence daného atributu. Jak je znázorněno v následujícím kódu.



Díky možnosti zřetězit za sebou několik operátoru přiřazení, je potřeba rozdělit výraz na jednotlivé l-hodnoty a r-hodnoty a podle nich vybrat zda bude použit funkce pro získání nebo nastavení atributu, neboli l-hodnoty se nastaví setterem a r-hodnoty se získají pomocí getteru.

## Výstup

Poslední blok se postará o výpis do výstupních souborů. Pro každý zpracovaný vstupní soubor se vytvoří hlavičkový soubor s deklarací struktur a hlaviček funkcí. Každý soubor bude mít vlastní inicializační funkci pro inicializaci hierarchie a počátečnímu přiřazení hodnot a ukazatelů na potřebné funkce.

Pro každý GUI element, kterému je pomocí příkazu přidán nový atribut, se definuje nový typ struktury, která dědí z původního typu struktury. Každý přidaný atribut se deklaruje v nové struktuře, spolu s ukazatelem na funkci, která bude sloužit k jeho updatování. Pro každou takovou strukturu se také vytvoří funkce, které takovou strukturu alokují a inicializují, přičemž se v dané funkci zavolá funkce pro inicializaci struktury, ze které se dědí.

Pro všechny elementy se vytvoří funkce pro jejich update, která slouží k updatování hodnot v atributech elementů. Pro každý atribut, jemuž je přiřazena nějaká hodnota v CQML souborech, bude definována funkce pro jeho update danou hodnotou. Ukazatel na danou funkci, bude přiřazena v inicializační funkci daného souboru, do členské proměnné struktury určené pro daný atribut.

# Implementace

Pomocí generátoru Bison byl podle gramatiky jazyka CQML vytvořen parser. Pro vygenerování lexikografického analyzátoru byl použit FLEX. Zdrojový kód parseru a lexikografického analyzátoru je pak použit v parsovací aplikaci.

Po spuštění program načte výchozí typy GUI prvků, uloží si jejich názvy a seznam jejich atributů, včetně jejich typů a výchozích hodnot, do instancí třídy „ClassContainer“.

Program na vstupu přijímá jméno souboru, se zdrojovým kódem v jazyce CQML. Pokud daný soubor existuje, je otevřen a jeho obsah je předán parseru. Pokud parser detekuje syntaktickou chybu ve zdrojovém souboru, vypíše chybu a program se ukončí. Výstupem parseru je syntaktický strom. V první fázi se ve stromu vyhledají příkazy pro import elementu z jiných souborů. Totéž program cyklicky provede pro všechny soubory, z nichž se elementy importují.

Následně je vytvořen orientovaný graf vzájemných závislostí mezi jednotlivými soubory, kde každý vrchol představuje soubor a každá hrana představuje vazbu mezi soubory, zatímco počátečním vrcholem hrany je vrchol představující soubor, do něhož je prvek importován, a koncovým vrcholem je vrchol představující soubor, z nějž byl prvek importován. Pomocí algoritmu prohledávání do hloubky je zjištěno, zda se v grafu nachází cyklus. Pokud se v grafu nachází cyklus, je vypsána chybová hláška a program se ukončí.

Jelikož je graf acyklický, pak každý vrchol představuje komponentu a tak lze jednotlivé vrcholy grafu a tudíž i soubory topologicky seřadit. Soubory (resp. vrcholy v grafu) jsou seřazeny pomocí Tarjanova algoritmu. Syntaktické stromy jednotlivých seřazených souborů jsou postupně zpracovány následujícím způsobem.

Během průchodu stromem se postupně alokují instance třídy Element, přičemž každá instance představuje určitý prvek v hierarchii GUI. Každému prvku je přiřazen jeho typ a seznam jeho potomků resp. prvků, které se v hierarchii nachází níže. Dále je každému prvku, u nějž jsou definované změny některých z jejich atributů, přiřazena množina dvojic názvů atributů a jejich hodnot. Hodnotou v tomto případě nemusí být konstanta, ale i výraz v jazyce C, jehož výpočtem se získá daná hodnota.

Dále program vytvoří mapu identifikátorů, pro všechny prvky, u nichž byl nastaven atribut „id“. Tento atribut musí mít unikátní hodnotu, pokud je v souboru více než jeden prvek nastaven na stejnou hodnotu atributu „id“, dojde k chybě.

Každému elementu je podle jeho specifikovaného typu přiřazen ukazatel na instanci „ClassContainer“. Pokud byly nějakému elementu přiřazeny nové atributy, bude muset být ve výstupním kódu daný element reprezentován speciální třídou, tudíž se vytvoří nová instance „ClassContainer“, do které jsou mimo výchozích atributů přidány i atributy nové. Všechny nové atributy musí být buď nějakého výchozího typu, nebo musí typem nějakého importovaného či výchozího elementu, jinak dojde k chybě.

Vzhledem k tomu, že hodnoty některých atributů jsou definovány jako výrazy v jazyce C, v nichž je možné se odkazovat na jiné elementy pomocí jejich identifikátorů „id“ anebo je možné se odkázat na prvek výše v hierarchii klíčovým slovem „parent“, je nutné analyzovat výrazy a nahradit ve výsledném kódu klíčová slova referencemi na dané prvky.

// todo

Bezkontextové gramatiky

Bezkontextová gramatika je definována čtveřicí (N,Σ,S,P). Kde N je konečná množina neterminálních symbolů. Σ je konečná neprázdná množina terminálních symbolů. S je počáteční symbol. P je konečná množina pravidel α -> β, kde α je neterminální symbol a β je slovo, pro které platí β ϵ (N ᴜ Σ)\*.

Z počátečního symbolu lze použitím pravidel odvodit všechna slova přijímaná danou bezkontextovou gramatikou. Pro každé slovo takto odvozené existuje derivace (neboli odvození) existuje posloupnost pravidel z množiny P, jejichž postupnou aplikací na neterminální symboly lze, z počátečního symbolu vytvořit dané slovo. Pro každou takovou derivaci existuje derivační strom, pro který platí:

1. Každý vrchol, který není listem, je ohodnocen neterminálem
2. Každý list je ohodnocen terminálem nebo prázdným slovem, přičemž list ohodnocený prázdným slovem je jediným následníkem svého předchůdce.
3. Jestliže některý vrcholek, který není listem je ohodnocen neterminálem A a má následníky X1, X2,…XK(v tomto pořadí z leva do prava), pak A->X1X2…XK je pravidlo gramatiky.

Pokud žádný list není ohodnocen prázdným slovem, pak je gramatika nevypouštěcí, v opačném případě se jedná o gramatiku vypouštěcí.

Použité technologie

Bison

Bison je víceúčelový generátor parserů, který z dané bezkontextové gramatiky vygeneruje deterministický LR parser nebo generalizovaný LR parser ve formě zdrojového kódu k programu v jazyce C nebo C++. [Bison]

Flex

Flex je nástroj pro tvorbu skenerů pro lexikální analýzu. Jeho cílem je rozpoznat lexikální šablony v textovém vstupu a převést je na výstup ve formě posloupnosti symbolů (tokenů), které mohou být zpracovány dále například pomocí parseru.