

**MATEMATICKO-FYZIKÁLNÍ  
FAKULTA**  
Univerzita Karlova

## **BAKALÁŘSKÁ PRÁCE**

Filip Kliber

# **Návrh a implementace jazyka pro code-golf challenge**

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Pavel Parízek, Ph.D.

Studijní program: Informatika

Studijní obor: Programování a softwarové systémy

Praha 2016

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne .....

Podpis autora

Název práce: Návrh a implementace jazyka pro code-golf challenge

Autor: Filip Kliber

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Pavel Parízek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Code-golf je jednou z kategorií programátorských soutěží, ve které je cílem implementovat řešení relativně snadné programátorské úlohy na co nejmenší počet bytů zdrojového kódu. Cílem mé práce je návrh a implementace programovacího jazyka, který je vhodný pro řešení code-golf úloh. Jazyk je navržený s ohledem na nově začínající řešitele code-golf challenge a zachovává čitelnost i srozumitelnost. Zakládá se na běžně používaných jazycích, ale zároveň umožňuje stručný zápis algoritmických řešení úloh z oblasti code-golf. Součástí práce je také implementace standardní knihovny jakožto běhové podpory pro tento jazyk.

Klíčová slova: překladač, programovací jazyk, code-golf, c++

Title: Design and implementation of a language for code-golf challenge

Author: Filip Kliber

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Code-golf is one of types of programming competitions. Primary objective of such competition is to provide a solution to relatively simple programming task, such that the length of the source code is minimal in terms of bytes. The goal of my thesis is to design and implement a programming language that is suited for code-golf challenges. The language is designed with respect to beginners in the code-golf domain. It preserves readability and intelligibility, and it is based on popular programming languages, but still provides facilities for concise writing of algorithmical solutions for code-golf tasks. The work also contains implementation of a standard library as support for runtime environment of the language.

Keywords: compiler, programming language, code-golf, c++

Rád bych poděkoval vedoucímu své bakalářské práce, RNDr. Pavlu Parízkovi, Ph.D., za jeho cenné rady, připomínky a čas obětovaný konzultacím.

# Obsah

<b>Úvod</b>	<b>2</b>
<b>1 Analýza existujících jazyků</b>	<b>5</b>
1.1 Standardní knihovny . . . . .	5
1.2 Funkce main . . . . .	5
1.3 Výstupní mechanismy . . . . .	6
1.4 Délka standardních identifikátorů . . . . .	6
1.5 Typ textového literálu . . . . .	6
<b>2 Specifikace jazyka Swer</b>	<b>7</b>
2.1 Literál . . . . .	7
2.2 Typ . . . . .	10
2.3 Deklarace . . . . .	10
2.4 Výraz . . . . .	12
2.5 Příkaz . . . . .	14
2.6 Program . . . . .	16
<b>3 Architektura řešení</b>	<b>17</b>
<b>4 Překladač Swer</b>	<b>18</b>
4.1 Lexikální analýza . . . . .	18
4.2 Syntaktická analýza . . . . .	18
4.3 Modul <code>swer</code> . . . . .	20
4.4 Modul <code>swer::compiler</code> . . . . .	27
<b>5 Standardní knihovna Swer</b>	<b>32</b>
5.1 Číselné typy . . . . .	32
5.2 Textový řetězec – <code>swerstl::String</code> . . . . .	33
5.3 Výstup . . . . .	34
5.4 Vstup . . . . .	34
5.5 Pole – <code>swerstl::Vector</code> . . . . .	35
5.6 Mapa – <code>swerstl::Map</code> . . . . .	36
5.7 Algoritmy . . . . .	37
<b>6 Příklady</b>	<b>39</b>
<b>Závěr</b>	<b>45</b>
<b>Seznam použité literatury</b>	<b>47</b>
<b>Přílohy</b>	<b>48</b>
A Uživatelská dokumentace . . . . .	48
B Překladač Swer . . . . .	49

# Úvod

Code-golf je druh soutěže mezi programátory, ve které jsou účastníkům prezentovány relativně jednoduché úlohy, jejichž algoritmické řešení je více méně přímočaré. Kritérium pro zvolení nejlepšího řešení je počet bytů zdrojového kódu – nejkratší kód vyhrává<sup>1</sup>.

*Příklad.* Pro zadání „Rozhodnout, zda je číslo prvočíslo nebo ne“, mohou řešení v různých jazycích vypadat následovně:

## Hexagony

```
.?' ).@@/' / . ! . > + = ( < . ! ) } ( $ > ( < %
```

## Pyth

```
}QPQ
```

## Java

```
class P{public static void main(String[] a){int i=2,n=Short.valueOf(a[0]);for(;i<n;)n=n%i++<1?0:n;System.out.print(n>1);}}
```

## APL

```
2=+/0=x|~x←⌈
```

Kvůli některým programovacím jazykům se občas definice code-golf mírně upravuje. Programovací jazyk *APL* pro základní funkce používá symboly, jejichž bytová reprezentace překračuje jeden byte (v kódování UTF-8) a proto se řešitelé, používající *APL* cítí znevýhodnění, a vyžadují, aby kritérium nebylo počet bytů, nýbrž počet znaků. Navíc také existují programovací jazyky, kde se zdrojový kód nedá vyjádřit počtem bytů, a vyžadují zvláštní pravidla (např. jazyk *Foldery*).

Existuje značné množství internetových portálů zabývajících se zadáváním i řešením code-golf úloh. Některé vyžadují, aby řešení bylo v konkrétním jazyce, jiné připouští pro jedno zadání řešení v různých jazycích. Jedním z populárních portálů je *Programming Puzzles & Code Golf* hostovaný na *StackExchange*<sup>2</sup>. Zadavatelé úloh na tomto portálu jsou stejní uživatelé, kteří je řeší – každý má možnost zadat svůj vlastní challenge. Zadavatel může taktéž nastavit nějaká pravidla pro řešení, pokud to úloha vyžaduje (např. explicitně zakázat použití nějaké techniky nebo počítat skóre trochu jiným způsobem). I když zadavatel explicitně nenapíše žádná pravidla, očekává se dodržování implicitních pravidel. Respektive je zakázáno následující (výčet obsahuje pouze hlavní body):

<sup>1</sup>Odtud plyne název code-**golf** – Jako v golfu vyhrává ten s nejmenším skóre.

<sup>2</sup>Portál obsahuje i jiné typy úloh, code-golf challenge jsou označeny tagem *code-golf*.

1. „Vykládat si zadání doslovně.“ Pokud by zadání říkalo „Napište program, který pro dané  $N$  vypíše  $N$ -té prvočíslo.“, tak řešení, které vypíše text  $N$ -té prvočíslo, není validní.
2. „Použit programovací jazyk vytvořený pouze pro tuto úlohu.“ (často zobecněno jako „Použitá verze programovacího jazyka musela být k dispozici dříve než byla zveřejněna úloha.“)
3. „Použití názvu programu pro uchování dat.“
4. „Použit externí zdroj pro získání výstupu.“

Díky tomu, že má uživatel volnou ruku ve volbě jazyka, ve kterém chce konkrétní úlohu řešit, vznikly programovací jazyky primárně určené k řešení code-golf challenge. Mezi často používané jazyky patří *CJam* – zásobníkove orientovaný jazyk, který pracuje (nejen) na zásobníku pomocí operátorů, a *Pyth* – procedurální jazyk, také hojně využívající operátory. Tyto jazyky jsou populární, protože mají největší potenciál zvítězit [1]. Principem těchto jazyků je co nejvíce minimalizovat velikost zdrojového kódu, i za cenu klesání či dokonce ztráty čitelnosti. Tato vlastnost je nejspíše nutná pro navržení kvalitního (co se týče minimalizace zdrojového kódu) jazyka. Nicméně to může být překážkou pro začínajícího řešitele. Po rozkukání se v prostředí code-golf se uživatel pokusí vyřešit nějakou úlohu, pravděpodobně v jazyce, ve kterém často programuje. Ukažme si příklad v jazyce C++.

*Příklad.* „Rozhodnout, zda je číslo prvočíslo nebo ne“

```
#include <iostream>
void main(int argc, char* argv[])
{
    int input = std::stoi(argv[0]);
    bool isprime = true;
    for (int i = 2; i < input; ++i)
        if (input % i == 0)
            isprime = false;
    std::cout << isprime;
}
```

Uživatel si pak uvědomí, že metoda `main` může vracet `int`, že identifikátory mohou být jen jednopísmenné, že některé deklarace se dají spojit, že většinu bílých znaků lze odstranit, případně ho napadne lepší myšlenka a přijde s lepším řešením.

```
#include<ios>
int main(int c,char**a){int n=atoi(a[1]),i=2;for(;i<n;)c*=n%i++;
puts(c&&">1?"1":"0");}
```

Toto řešení má 100 bytů a zdá se dobré. Uživatel jej zveřejní a podívá se, jak si vedou ostatní jazyky. Zjistí, že se mu podařilo porazit řešení v *C#*, *Javě* a ve spoustě dalších programovacích jazyků, ale s nejlepšími řešeními se nemůže vůbec srovnávat. V jazyce *Pyth* je řešení `}>PQ`, v jazyce *CJam* pak `qimp`.

Uživatel se pak podívá, jak se programuje v takovýchto jazycích a zjistí, že styl programování je jiný. Možná se mu nebude chtít učit nový jazyk a na code-golf nadobro zanevře. Možným řešením tohoto problému je navrhnout programovací jazyk, který se podobá běžným jazykům, ale umožňuje stručnější zápis jednotlivých konstruktů daného jazyka.

## Cíle práce

Cílem mé práce je navrhnout programovací jazyk, který je použitelný pro řešení code-golf challenge a byl by převážně určen začínajícím uživatelům v této oblasti. Protože cílová skupina jsou začínající uživatelé, je nezbytné, aby jazyk byl čitelný, lehce uchopitelný a zvládnutelný a aby nekladal velký důraz na ovládnutí nové syntaxe a sémantiky. Tento jazyk jsem nazval **Swer**.

Dalším cílem práce je implementovat překladač *Swer* a standardní knihovnu jazyka, jakožto běhovou podporu.



# 1. Analýza existujících jazyků

Můžeme předpokládat, že lidé, kteří začínají s řešením code-golf challenge, již mají zkušenosti s programováním. Chápu rozdíly mezi různými metodami programování (imperativní, logické, funkcionální) a znají koncept objektově orientovaného programování. Většina z nich bude ovládat některý z jazyků *Java*, *C*, *C++*, *Python*, *C#*, *PHP*, *Javascript*. [2] Všechny tyto jazyky mají společné to, že jsou procedurální. Z tohoto důvodu je vhodné, aby i Swer byl procedurální a aby syntaxí i sémantikou připomínal tyto jazyky.

Pokud se podíváme na příklad z motivace („Rozhodnutí, zda je číslo prvočíslo“) v jazyce C++

```
#include <ios>
int main(int c, char**a)
{
    int n = atoi(a[1]),
        i = 2;
    for (; i < n;)
        c *= n % i++;
    puts(c && n>1 ? "1" : "0");
}
```

tak zjistíme, že `#include` by bylo vhodné odstranit. Pokud by se jednalo o obtížnější úlohu, vyžadující ještě něco z hlavičkového souboru `algorithm`, máme navíc 18 bytů, které netvoří výpočetní část programu.

To samé platí o nutnosti deklarovat funkci `main` (a v případě že je to potřeba, včetně jejích parametrů). Mohli bychom předpokládat, že se náš kód automaticky zabalí do funkce `main` a budeme mít zaručeno, že argumenty příkazové řádky jsou k dispozici v proměnné s konkrétním názvem.

Následující sekce ukazují často používané konstrukty, které je vhodné v jazyce Swer zkrátit.

## 1.1 Standardní knihovny

Pro přístup do standardní knihovny vyžaduje C++ použití `#include` hlavičkových souborů. Java a C# potřebují buď příkaz `import`, respektive `using`, nebo použití jména třídy ze standardní knihovny musí být plně kvalifikované (to bývá zpravidla poměrně dlouhé – např. `System.Collections.Generic.Dictionary`). Proto je vhodné, aby Swer automaticky připojoval všechny potřebné části standardní knihovny bez nutnosti psaní dodatečných příkazů.

## 1.2 Funkce `main`

Nejkratší zápis signatury funkce `main` v C++ je `int main()`. Některé překladače dovolují vynechat typ návratové hodnoty, ale nevyhovuje to standardu C++. Java a C# navíc vyžadují, aby funkce byla ve třídě. Python má sice jednoduché `def main:`, ale vyžaduje odsazování. Javascript dovoluje psát příkazy rovnou. Swer by mohl předpokládat, že za poslední deklarací následuje kód funkce `main`.

## 1.3 Výstupní mechanismy

Často se hodí zkombinovat vypsání výstupu s posledními kroky algoritmu. Např. místo kódu

```
vystup -> "Text"
if podminka
    vystup -> "1"
else
    vystup -> "2"
```

je možné napsat výrazně kratší

```
vypis -> "Text" + podminka ? "1" : "2"
```

C++ používá `std::ostream` a `operator<<`, který, díky své prioritě, často vyžaduje dodatečné uzavorkování argumentů. Některé jazyky využijí formátovací funkci podobnou `printf` z C, ale ta vyžaduje formátovací řetězec. C# umožňuje použití interpolovaných textových řetězců.

## 1.4 Délka standardních identifikátorů

Protože jsou zmíněné jazyky používány pro vývoj rozsáhlého softwaru, u kterého je nutná udržitelnost, jsou použité pro názvy typů, klíčových slov a standardních tříd výřečná jména. Swer by měl umožňovat použití standardních mechanismů pomocí identifikátorů o délce ideálně jednoho znaku.

## 1.5 Typ textového literálu

Tento problém se vyskytuje u jazyka C++, kde typ textového literálu je `const char[N]`, kde  $N - 1$  je délka literálu (jeden byte za ukončovací `\0`). To nedovoluje použití metod definovaných na třídě `string` přímo. Od verze C++14 je díky `operator""s` možné použít syntaxi `"text"s.size()`. Jazyk C má sice stejnou vlastnost, ale protože není objektově orientovaný, tak se v jeho případě nejedná o problém.

## 2. Specifikace jazyka Swer

Jazyk Swer je procedurální jazyk s prvky objektově-orientovaného programování. Jeho zdrojový kód je překládán do C++, který pak lze přeložit do spustitelného souboru. Syntaxe je case-sensitivní, tedy na velikosti písmen záleží. Jednotlivé tokeny mohou být odděleny libovolným počtem bílých znaků (i prázdným řetězcem), pokud by nedošlo ke spojení dvou tokenů do jednoho, který by byl validní. Například mezi identifikátory `string` a `str` musí být bílý znak, jinak by se jednalo o jeden identifikátor `stringstr`.

Následující sekce popisují gramatiku a sémantiku jednotlivých fragmentů gramatiky. Nejedná se o skutečnou gramatiku použitou při konstrukci překladače, jak se dozvíme později v sekci 4.2.

Popis gramatiky používá podobný formát jako specifikace standardu C++ (dokument *ISO International Standard ISO/IEC 14882:2014(E) – Programming Language C++*). Pokud je čtenář znalý formátu používaného ve zmíněném standardu, může následující vysvětlení přeskočit.

Název sekce koresponduje se základním elementem jazyka. Následuje krátké vysvětlení sémantiky daného fragmentu. Poté je ukázán fragment gramatiky popisující syntaktický zápis. *Kurzívou* se identifikují jednotlivé syntaktické kategorie. Text s **pevnou šířkou** je použitý jako notace pro klíčová slova a znaky, které se mají v kódu objevit přesně v dané podobě. Alternativní přepisy se vyskytují na jednotlivých řádcích, nebo jsou označeny textem *jeden z*. Pokud je terminální nebo neterminální symbol nepovinný, má ve svém dolním indexu vyznačeno <sub>opt</sub>. Pokud za syntaktickou kategorií následuje *-seq* (kurzívou), znamená to, že daná syntaktická kategorie může být opakována za sebou (bez oddělovače, alespoň jedno opakování). Podobně *-list* za neterminálem značí možnost opakování, kde se mezi jednotlivými iteracemi vyskytuje čárka (,). Za poslední iterací čárka není.

*Příklad.* Ilustrace použití zmíněných konvencí.

*statement :*

```
{ statement-seqopt }  
-> expressionopt ;
```

Značí, že *statement* je buď sekvencí *statement* (bez oddělovače) ve složených závorkách ({, }), kde sekvence může být prázdná (<sub>opt</sub>), nebo *statement* začíná textem *->*, za kterým následuje nepovinný *expression* ukončený středníkem (;).

### 2.1 Literál

Swer obsahuje několik druhů literálů. Sémantika každého z nich je popsána v následujících sekcích.

*literal :*

```
integer-literal  
floating-literal  
character-literal  
string-literal  
identifier
```

*array-literal*

## Celé číslo

Prvním z druhů literálů je celé číslo, zapsané buď dekadickým, binárním, oktalovým nebo hexadecimálním zápisem. Je ekvivalentní zápisu čísla v C++ se suffixem `ll` (`long long`).

*integer-literal* :

*nonzero-decimal-digit decimal-digit-seq*<sub>opt</sub>

`0b` *binary-digit-seq*

`0B` *binary-digit-seq*

`0` *octal-digit-seq*

`0x` *hexadecimal-digit-seq*

`0X` *hexadecimal-digit-seq*

*nonzero-decimal-digit* :

Jeden z 1 2 3 4 5 6 7 8 9

*decimal-digit* :

0

*nonzero-decimal-digit*

*binary-digit* :

Jeden z 0 1

*octal-digit* :

Jeden z 0 1 2 3 4 5 6 7

*hexadecimal-digit* : Jeden z

0 1 2 3 4 5 6 7 8 9

a A b B c C d D e E f F

## Desetinné číslo

Dalším druhem literálu je desetinné číslo (s plovoucí desetinnou čárkou). To je ekvivalentní desetinnému číslu v C++ se suffixem `d` (`double`).

*floating-literal* :

*decimal-digit-seq* <sub>opt</sub> *exponent*

*decimal-digit-seq*<sub>opt</sub> `.` *decimal-digit-seq* *exponent*

*exponent* :

`e` <sub>opt</sub> *decimal-digit-seq*

`E` <sub>opt</sub> *decimal-digit-seq*

`e` `-` *decimal-digit-seq*

E – *decimal-digit-seq*

## Znak

Mezi literály patří také znak. Sémantika je stejná jako v C++. Swer umožňuje zapsat znak pomocí jeho kódu jako C++ (např. `\xABCD`).

*character-literal* :

`' character '`

*character* :

*alpha-character*

Jeden z 0 1 2 3 4 5 6 7 8 9

Jeden z < > { } [ ] ( ) ~ ! @ # \$ % ^ & \* - = + \_ ; | " : , . ?

*alpha-character* :

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

*character* nesmí být `'`, pokud před ním nepředchází znak `\`. Znaky jako tabulátor nebo nový řádek se píšou pomocí skutečných znaků (např. `'\t'`).

## Textový řetězec

Dalším důležitým literálem je textový řetězec. Metody užitečné při práci s textovými řetězci jsou detailně popsány v kapitole 5.

*string-literal* :

`" character-seqopt "`

Mezi sekvencí znaků se nesmí vyskytovat `"`, pokud jemu nepředchází znak `\`. Textový řetězec může přesahovat přes konec řádku.

## Identifikátor

Identifikátor v programu slouží pro identifikaci jmen – názvů proměnných, tříd, a metod.

*identifier* :

*alpha-character identifier-character-seq<sub>opt</sub>*

*\_ identifier-character-seq<sub>opt</sub>*

*identifier-character* :

*alpha-character*

*decimal-digit*

-

Použití identifikátoru `_` (samotné podtržítko) je ve funkci `main` rezervováno pro argumenty příkazové řádky.

## Pole

Posledním druhem literálu je pole. Metody užitečné při práci s poli jsou detailněji popsány v kapitole 5.

*array-literal* :  
[ *expression-list* ]

Typ jednotlivých elementů pole je odvozen podle prvního výrazu v hranatých závorkách. Swer neumožňuje napsat literál pro prázdné pole.

## 2.2 Typ

Dalším syntaktickým elementem jazyka je použití typu. Typ je používán při deklaraci proměnné, v parametrech metod, a jako generický argument.

*type-specifier* :  
*type-name* *ref-or-ptr-qualifier-seq*<sub>opt</sub>

*ref-or-ptr-qualifier* :  
#  
~

*type-name* :  
\$  
*nested-name-specifier*<sub>opt</sub> *identifier* { *type-specifier-seq* }<sub>opt</sub>

*nested-name-specifier* :  
::  
*type-name* ::

*nested-name-specifier* je ekvivalentní *scope resolution* operátoru (`::`) v C++. Nepovinný *type-specifier-seq* značí generické argumenty, ekvivalentní `<T, ...>` z C++, kde `...` značí další argumenty.

Na rozdíl od C++ je deklarace ukazatele nebo reference součástí typu a nikoliv identifikátoru. Takže `integer# a,b;` deklaruje dvě proměnné `a` a `b`, obě typu ukazatel na `integer`.

## 2.3 Deklarace

Swer dovoluje deklaraci následujících typů: třída, metoda a proměnná.

## Třída

Deklarace třídy slouží pro zavedení nového typu. Jednotlivé deklarace ve třídě mají veřejnou viditelnost a Swer nemá prostředky pro změnu viditelnosti některých prvků.

*class-declaration* :

$\{ identifier-seq \}_{opt} identifier \{ class-member-seq_{opt} \}$

*class-member* :

*class-declaration*

*function-declaration*

*variable-declaration* ;

Generické argumenty není možné použít na takové úrovni, kterou umožňuje mechanismus nazývaný *C++ template metaprogramming*.

## Metoda

Další z deklarací je deklarace metody. Syntakticky není rozdíl mezi deklarací metody mimo třídu nebo ve třídě.

*function-declaration* :

*type-specifier identifier ( formal-parameter-seq<sub>opt</sub> ) compound-statement*

*formal-parameter* :

*type-specifier identifier<sub>opt</sub>*

*type-specifier identifier = expression*

Tělo metody je povinné, což zamezuje použití *forward declaration*, což by znemožňovalo cyklické volání funkcí. Tento problém je vyřešen díky tomu, že překladač Swer zajišťuje, že všechny funkce jsou nejprve deklarovány a pak až definovány. Implementační detaily této vlastnosti jsou popsány v sekci 4.2. Typ žádného argumentu funkce nesmí být automaticky dedukovaný – pomocí klíčového symbolu \$. Návratová hodnota může být dedukovaná z kontextu, pokud se nejedná o rekurzivně definovanou funkci.

## Proměnná

Posledním typem deklarace je proměnná. Jedna deklarace umožňuje deklarovat více proměnných stejného typu. Proměnná může být ihned inicializována.

*variable-declaration* :

*type-specifier identifier-with-opt-initialization-seq*

*identifier-with-opt-initialization* :

*identifier initialization<sub>opt</sub>*

```

initialization :
    = expression
    { initializer-clause-list }
initializer-clause :
    expression
    { initializer-clause-list }

```

Swier neumožňuje deklarovať konštruktor triedy, ale pojem konštruktoru chápe. Inicializace proměnné umožňuje volat konštruktor tříd ze standardní knihovny.

## 2.4 Výraz

Dalším elementem jazyka je výraz. Výraz vzniká spojením více výrazů pomocí operátorů nebo z literálu.

```

expression :
    literal
    ( expression )
    expression binary-operator expression
    unary-operator expression
    expression postfix-operation
    lambda
    operator-object

```

*binary-opreator* : Jeden z  
 $\wedge$  \* / % + -  $\wedge\wedge$  << >> < > <= >= == !=  
 = += -= \*= /= %= ^= <<= >>= &&= ||= ^ ^=

*unary-operator* :  
 Jeden z + - \* & ~ ! ++ --

```

postfix-operation :
    ( initializer-listopt )
    [ expression ]
    . identifier
    -> identifier
    Jeden z ++ -- !

```

Gramatika v této formě je nejednoznačná, protože výraz  $a - b - c$  může být interpretován jako  $(a - b) - c$  nebo  $a - (b - c)$ . Řešení tohoto problému je popsáno v kapitole 4.2.

Asociativita a priorita operátorů je definovaná podle následující tabulky. Sloupec *ekvivalent v C++* naznačuje příkaz, ke kterému je *operator* ekvivalentní. Pokud je sloupec prázdný, znamená to, že *operator* by měl stejný význam i v C++. Asociativita zleva doprava znamená, že  $a \text{ op } b \text{ op } c$  je vyhodnocován v pořadí  $(a \text{ op } b) \text{ op } c$ .



priorita	operátor	ekvivalent v C++	asociativita
1	v++ v-- v() v[] v.a v->a v!	Faktoriál $v$	Zleva doprava
2	++v --v +v -v ++v --v !v ~v *v &v		Zprava doleva
3	$a \wedge b$	Umocnění $a^b$	Zleva doprava
4	$a * b$ $a / b$ $a \% b$		Zleva doprava
5	$a + b$ $a - b$		Zleva doprava
6	$a \ll b$ $a \gg b$		Zleva doprava
7	$a < b$ $a \leq b$ $a == b$ $a > b$ $a \geq b$ $a != b$		Zleva doprava
8	$a \&\& b$	$a \& b$	Zleva doprava
9	$a \wedge\wedge b$	$a \wedge b$	Zleva doprava
10	$a \ \  b$	$a   b$	Zleva doprava
11	$a \& b$	$a \&\& b$	Zleva doprava
12	$a   b$	$a \ \  b$	Zleva doprava
13	$a = b$ $a += b$ $a -= b$ $a *= b$ $a /= b$ $a \% = b$ $a \ll = b$ $a \gg = b$ $a \wedge = b$ $a \&\& = b$ $a \wedge\wedge = b$ $a \ \  = b$	$a = a^b$ $a \& = b$ $a \wedge = b$ $a   = b$	Zprava doleva

## Lambda výraz

Speciálním případem výrazu je lambda výraz. Typ každého lambda výrazu je unikátní, jediný způsob jak vytvořit proměnnou obsahující lambda výraz je pomocí automatické dedukce z kontextu (pomocí klíčového symbolu `$` – ekvivalentní C++ `auto`).

*lambda :*

*( ) => compound\_statement*

*identifier => compound\_statement*

*( identifier , identifier-list ) => compound\_statement*

Pokud má lambda výraz jeden parametr, tak syntaxe vyžaduje, aby nebyl v kulatých závorkách.

## Objekt operátor

Pro zjednodušení volání některých funkcí ze standardní knihovny dává jazyk Swer možnost jednoduše syntakticky zapsat binární operátor, který se vyhodnotí na objekt. K dispozici jsou operátory pro  $+$   $-$   $\times$   $\div$   $\exp$ .

*operator-object :*

Jeden z `+`? `-`? `*`? `/`? `%`? `^`?

Primární použití je očekáváno v situaci `funkce(3, 2, /*)`, ale jazyk Swer nebrání ani použití `/(3, 2)`. Objekt operátor je lambda výraz, který přijímá dva argumenty a použije na ně daný operátor.

## 2.5 Příkaz

Základním prvkem jazyka Swer je příkaz. Příkazy jsou vykonávány sekvenčně, jeden za druhým.

*statement :*

```
{ statement-seqopt }  
expression ;  
if-statement  
switch-statement  
iteration-statement  
declaration-statement  
brk ;  
cnt ;  
-> expressionopt ;
```

`brk` je ekvivalentní C-like `break`. `cnt` je ekvivalentní C-like `continue`. Symbol `->` je ekvivalentní C-like `return`.

### If-else

Prvním z konstruktů, které jazyk Swer nabízí, je klasický *if-else*.

*if-statement :*

```
if ( condition ) statement  
if ( condition ) statement el statement
```

*condition :*

```
expression  
type-specifier identifier initialization
```

Pokud je v podmínce deklarace proměnné, tak není proměnná vidět mimo *if-else* příkaz.

### Switch

Další možností, jak při běhu programu zajistit vykonávání jen některé větve algoritmu, je *switch* příkaz.

*switch-statement* :

**sw** ( *condition* ) { *case-statement-seq* }

*case-statement* :

: *expression* : *statement-seq*<sub>opt</sub>

: **el** : *statement-seq*<sub>opt</sub>

Narozdíl od C/C++ není vyžadován příkaz **brk**; na konci **case** příkazu pro zamezení propadnutí do další větve výpočtu. Pokud bychom ale chtěli propadnout do další větve, musíme jako poslední příkaz v dané **case** větvi uvést **cnt**;. Toto neplatí, pokud je sekvence příkazů u dané **case** větve prázdná.

## Iterace

Příkazů, které umožňují vytvořit cyklus, je v jazyce Swer několik.

*iteration-statement* :

@ *identifier*<sub>opt</sub> ( *expression* .. *expression* ) *statement* (1)

@ *identifier*<sub>opt</sub> ( *expression* ) *statement* (2)

@ ( *expression* ; ) *statement* (3)

@ ( *expression* ; *expression* ; *expression-list* ) *statement* (4)

@ ( *type-specifier identifier initialization* ; *expression* ; *expression-list* ) *statement* (5)

V případě, že je identifikátor nepovinný a je vynechaný, tak má implicitní jméno, podle úrovně zanoření příkazů *for* s implicitní řídicí proměnnou. Jméno identifikátoru na první úrovni je **i**, v druhé úrovni zanoření je **j** a ve třetí je **k**. Při pokusu překládat zdrojový soubor s větším množstvím zanořených *for* cyklů s implicitní řídicí proměnnou, nastane chyba překladu. Chyba překladu nastane i v případě, že jméno identifikátoru nelze použít, protože uživatel již deklaroval proměnnou se stejným názvem.

(1) Představuje cyklus s pevným počtem opakování. Je podobný *for* cyklu z jazyka Pascal. Cyklus **@var(expr1 .. expr2) stmt**; před prvním krokem iterace přiřadí do **var** hodnotu **expr1**. Po dokončení příkazu **stmt** se přičte k **var** buď +1 (v případě **expr1 < expr2**), nebo -1 (v opačném případě).

(2) Je sémanticky ekvivalentní cyklu *for-each* přes nějakou kolekci (v případě Swer pole – **swerstl::vector** nebo mapa – **swerstl::map**). Použití příkazu **@var([1,2,3]) stmt**; postupně vykoná příkaz **stmt**;, kde řídicí proměnná **var** postupně nabývá hodnot 1, 2, 3.

(3) Umožňuje napsat *while* z C-like jazyků. Zápis **@(expr;) stmt**; je ekvivalentní s C/C++ **while (expr) stmt**;

(4) Dovoluje zapsat *for* cyklus, kde první část (*expression*) určuje inicializaci, druhá část koncovou podmínku cyklu a třetí část krok prováděný na konci každé iterace cyklu. Je ekvivalentní s C-like *for* cyklem v tomto syntaktickém formátu.

(5) Poslední možnost jak napsat *for* cyklus se příliš neliší od té předchozí. Jediný rozdíl je, že tato verze umožňuje v první části deklarovat (jednu) novou proměnnou.

## Deklarace

Deklarace proměnné má stejný syntaktický zápis jako v sekci 2.3. Za tím následuje středník pro ukončení deklarčního příkazu. Pokud proměnná nemá inicializační část, Swer automaticky doplní inicializaci implicitní hodnotou.

*declaration-statement* :  
*variable-declaration* ;

## 2.6 Program

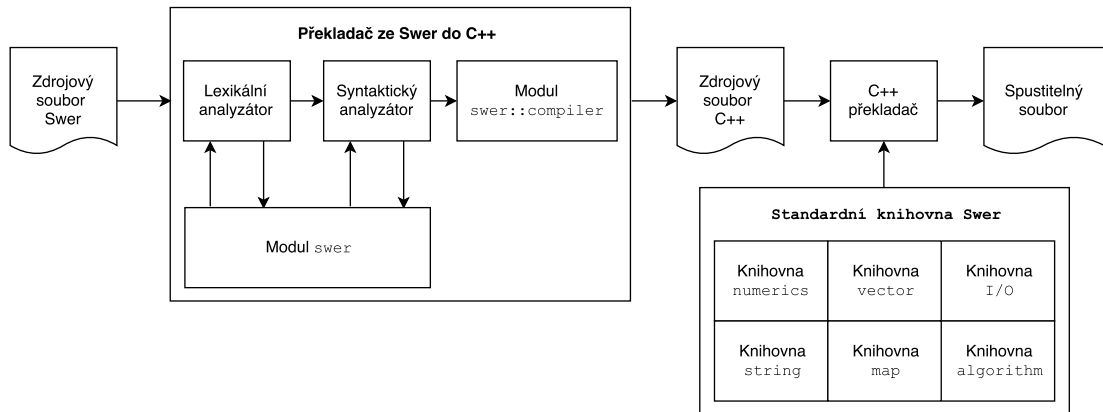
Zdrojový kód programu v jazyce Swer se skládá ze dvou, bezprostředně po sobě navazujících sekcí – sekce deklarací a sekce příkazů. Každá ze sekcí může být potenciálně prázdná. Tento fakt popisuje následující část gramatiky.

*program* :  
*declaration-seq*<sub>opt</sub> *statement-seq*<sub>opt</sub>

*declaration* :  
*class-declaration*  
*function-declaration*

Deklační sekce neumožňuje deklarovat proměnné. Deklarace proměnných už je brána jako *declaration-statement* a je v sekci příkazů.

### 3. Architektura řešení



Obrázek 3.1: Přehled architektury řešení

Výše uvedený obrázek představuje architekturu implementace jazyka Swer. Uživatel napíše dle specifikace (kapitola 2) kód v jazyce Swer a předá jej překladači. Ten program s pomocí lexikální a syntaktické analýzy a několika modulů zpracuje a vygeneruje z něj textový soubor, který obsahuje kód v jazyce C++ se stejnou sémantikou jako vstupní soubor překladače Swer. Tento kód, spolu se standardní knihovnou jazyka Swer je možné přeložit překladačem C++, který vygeneruje spustitelný soubor. Volání překladače C++ je možné delegovat na překladač Swer. Ten podporuje volání tří běžně používaných C++ překladačů — *GCC*, *Clang* a *Microsoft C++ compiler*.

Generování zdrojového kódu Swer do C++ oproti přímému generování do strojového kódu má několik výhod. Hlavní výhodou je přenositelnost. Překladač C++ existuje na velkém množství platforem i operačních systémů. Podobnou vlastnost by nám zajišťovala například také Java.

Další výhodou je elegantní integrace standardní knihovny jazyka Swer, která může být implementovaná prostřednictvím C++. Standardní knihovna C++ poskytuje uživatelům velké množství prostředků pro běhovou podporu a standardní knihovna jazyka Swer jí aktivně využívá.

Jazyk C++ umožňuje predefinování operátorů. To zjednodušuje implementaci použití netriviálních operátorů v jazyce Swer a pro ladící účely zpřehledňuje výstup v C++.

Při spuštění překladače Swer dojde v první řadě ke zpracování argumentů příkazové řádky. O to se stará POSIX-ová knihovna *getopt*. Použité volby si pamatuje třída `swer::options`, která je předána třídě `swer::compiler` při jejím vytvoření. Hlavní algoritmus překladače zajišťuje běh syntaktické analýzy (metoda `yyparse`), která dostane ukazatel na instanci třídy `swer::compiler`. Syntaktická analýza od lexikální analýzy dostává jednotlivé tokeny, z kterých buduje derivační strom. Po redukci jednotlivých pravidel volá syntaktická analýza kód odpovídající danému pravidlu, který konstruuje jednotlivé elementy jazyka (deklarace třídy, volání funkce, ...). Z jednotlivých elementů je postupně odspoda budován strom. Po redukci posledního pravidla (počátečního pravidla gramatiky) si modul `swer::compiler` uloží kořen stromu a syntaktická analýza končí. Následuje generování výstupního kódu, který zajišťuje třída `swer::compiler`.

## 4. Překladač Swer

### 4.1 Lexikální analýza

Lexikální analýzu vstupu ve formátu jazyka Swer zajišťuje nástroj *flex* (The Fast Lexical Analyzer). V souboru `src/swer.1` jsou pomocí regulárních výrazů definované tokeny, které má rozpoznávat flex. Před sestavováním projektu Swer se volá nástroj *flex*, který ze souboru `src/swer.1` vygeneruje nový soubor `src/autogen/gen_flex.cpp`. Tento soubor mimo jiné obsahuje definici funkce `yylex`, jejíž zavolání vrátí následující token ze vstupního souboru.

Flex se vždy ve vstupním souboru snaží nalézt nejdelší posloupnost znaků, která vyhovuje některému z pravidel. To v některých případech způsobuje potíže. Jak víme z kapitoly 2.5, je možné v jazyce Swer použít iterační příkaz připomínající `for` z jazyka Pascal. `for(1..5) prikaz;` by znamenal 5 iterací příkazu `prikaz` s řídicí proměnnou nabývající postupně hodnot 1,2,3,4,5. V definici tokenů pro flex by se měly vyskytovat pravidla pro tokeny `for`, `(` (levá závorka), pravidlo pro přirozené číslo, `..` a pravá závorka `)`. Definiční soubor pro flex obsahuje také definici pro desetinné číslo, které vyhovují jak `1.`, tak `.5`. Při analýze textu mezi závorkami ve výše zmíněném `for` cyklu, bychom očekávali, že flex bude postupně vracet tokeny `1`, `..`, `5`. Ale protože se flex snaží nalézt nejdelší posloupnost znaků vyhovující nějakému z pravidel, bude vracet tokeny `1.`, `.5`. Z tohoto důvodu je do pravidel přidána definice, která vyhovuje textu `..`, kterému předchází přirozené číslo. V tomto případě flex vrátí číslo a do fronty uloží token `..`. Následné volání funkce `yylex` nejprve vrací tokeny z fronty a pak až dělá nutné zpracování vstupního souboru.

Pro správný běh syntaktické analýzy stačí u tokenu znát jeho typ. Výraz `1 + 1` je pro syntaktickou analýzu stejný jako `1547 + 2101`, pro správnou sémantiku si ale musíme v některých případech zapamatovat i hodnotu. To platí pro tokeny *číslo*, *desetinné číslo* a *řetězec*. Funkce `yylex`, která ze vstupního souboru vrátí následující token, má ve skutečnosti parametr, který ukazuje na strukturu `swer::swer_lval`. Důležité prvky této struktury pro lexikální analýzu jsou právě `number_i`, `number_d` a `string`, které nesou hodnotu tokenu.

### 4.2 Syntaktická analýza

Pro zjednodušení syntaktické analýzy vstupního souboru je použitý nástroj *Bison*. Soubor `src/swer.y` obsahuje popis gramatiky jazyka Swer. Bison z pravidel gramatiky generuje soubory `src/autogen/gen_bison.hpp` a `src/autogen/gen_bison.cpp`, které obsahují mimo jiné definice identifikátorů tokenů pro flex a definici funkce `yyparse`, která se stará o parsování vstupního textu (volá flexem generovanou funkci `yylex`).

S gramatikou, jak je popsána v kapitole 2, by měl nástroj Bison problémy. To proto, že Bison zvládá zpracovávat pouze bezkontextový jazyk, respektive LALR(1) [3]. Proto je gramatika v souboru `src/swer.y` popsána detailněji.

Jazyky C a C++ pro deklaraci proměnné typu ukazatel (resp. reference) na typ používají syntaxi `typ * promenna` (resp. `typ & promenna`). Jazyk Swer používá pro deklaraci ukazatele (resp. reference) symbol `#` (resp. `&`). To je z toho

důvodu, že část zdrojového kódu `a * b;` je validní a může znamenat dvě věci v závislosti na tom, jestli `a` reprezentuje typ nebo hodnotu. Z tohoto důvodu, pro zjednodušení syntaktické analýzy jsou v jazyce Swer použity symboly, které nezanášejí do gramatiky tyto problémy.

Další podobná situace nastává v C++ při použití generických tříd. Syntaxe `typ<arg> promenna` může mít dva významy v závislosti na tom, co ve skutečnosti reprezentují identifikátory `typ` a `arg`. Pokud by se jednalo o proměnné, může se jednat o porovnávací výraz se sémantikou `(typ < arg) > promenna`. Z tohoto důvodu používá Swer složené závorky pro deklaraci i definici typu.

V kapitole 2 jsou výrazy definovány v jednom pravidle ve stylu *expression : expression operator expression*. Takovéto pravidlo je nejednoznačné. Žádným způsobem neurčuje asociativitu ani prioritu jednotlivých operátorů. Nástroj bison umožňuje vyřešit tuto situaci pomocí definování operátorů, spolu s jejich asociativitou a precedencí. Tento mechanismus není v překladači Swer využit. Jedním z důvodů proč tento mechanismus nevyužít je to, že podobného efektu lze docílit rozdělením operátorů na jednotlivá pravidla, kde výraz s operátorem s nižší prioritou se smí přepsat na operátor s vyšší prioritou, ale naopak jen s použitím uzávorkování.

Dalším problémem, který přináší gramatika popsaná v kapitole 2 je tzv. *dangling-else problem*. Ten nastává z důvodu, že větev *else* podmínkového příkazu *if* není povinná a díky tomu příkaz `if (podminka) if (podminka) prikaz1; else prikaz2;` může být chápán jako:

```
if (podminka1)
    if (podminka2)
        prikaz1;
    else
        prikaz2;
```

nebo také

```
if (podminka1)
    if (podminka2)
        prikaz1;
else
    prikaz2;
```

Kde v prvním případě se `prikaz2` provádí pouze za předpokladu, že platí `podminka1 && !podminka2`. V druhém případě vykonání `prikaz2` vůbec nesouvisí s hodnotou `podminka2`.

Tento problém je vyřešen duplikováním některých pravidel pro příkazy, které v případě přítomné *else* větve ve vnějším *if* příkazu nedovolí použít vnořený *if* příkaz bez *else* větví. To také znamená, že *else* větve jsou vždy spojené s nejbližším vyšším použitým *if* příkazem.

Při konání redukce jednotlivých pravidel je nutné si někde ukládat výsledek – instanci třídy reprezentující sémantiku daného pravidla. K tomu slouží (stejně jako pro přenášení dat z lexikální do syntaktické analýzy) struktura `swer::swer_lval`. Fakt, že se všechno přenáší prostřednictvím jedné struktury má nevýhodu v tom, že z několika položek struktury je vždy použita jen jedna.

To způsobuje zbytečné volání konstruktorů a destruktorů jednotlivých objektů a zabírá to místo v paměti. Nástroj bison ale požaduje, aby se mezi jednotlivými pravidly předávala právě jedna struktura. Zbytečné nároky na paměť by šlo vyřešit použitím unie (`union` z jazyků C nebo C++). Ta však vyžaduje použití „jednoduchých“ typů, které nemají ne-triviální funkce (např. destruktor). Od C++11 je tento požadavek vylepšen a unie může obsahovat i prvky, které mají netriviální destruktor. Pak je ale vyžadována vlastní správa aktivních prvků unie. Existuje implementace takto popsané vlastnosti např. v knihovně *Boost* (`boost::variant`). Ta není pro implementaci `swer_lval` použita z důvodu zachování jednoduchosti.

## 4.3 Modul `swer`

Modulem `swer` rozumíme systém tříd, metod a proměnných ve jmenném prostoru `swer` ze zdrojového kódu. Tyto mechanismy slouží při běhu kódu syntaktické analýzy pro tvoření sémantiky výsledného programu. Ke většině důležitých pravidel gramatiky existuje odpovídající třída, reprezentující sémantiku daného fragmentu. Následující sekce popisují chování jednotlivých pravidel gramatiky ze souboru `src/swer.y`. Název sekce reprezentuje sémantiku skupiny pravidel, které jsou zmíněny ke konci dané sekce. Názvy typů proměnných v jednotlivých třídách bývají zpravidla `name_ptr`, odpovídající `std::shared_ptr<name>`, a `name_seq` odpovídající `std::vector<name_ptr>`.

### `swer::type`

Tato třída slouží pro reprezentaci typu, nebo jeho části, při běhu syntaktické analýzy. Pro typ je typické, že má jméno (`m_identifier`), případně předka, a tedy sám je podtypem jiného typu (`m_nested_type`) a volitelně má seznam generických argumentů (`m_generics`), kde každý generický argument je také typ. Typ také obsahuje seznam kvalifikátorů (`qualifier`), mezi které patří `pointer`, `reference` a `nested`, které mohou být před (`m_prefixes`) nebo za (`m_suffixes`) typem. Třída sama nekontroluje, zda je aktuálně konstruovaný typ validní. Spoléhá na to, že syntakticky nevalidní typy neprojdou syntaktickou analýzou.

`type_name` je pravidlo, které vytváří instanci třídy `swer::type`. K typu jsou postupně doplňovány další vlastnosti pomocí `nested_name_specifier` nebo `ref_or_ptr_qualifier`. Výsledný typ vznikne redukcí `type_specifier`.

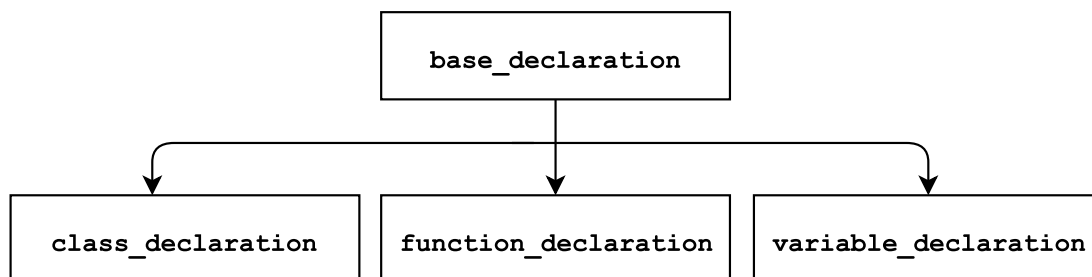
### `swer::base_declaration`

Třída `base_declaration` je společným předkem všech deklarácí. Při běhu syntaktické analýzy se potomci této třídy používají pro reprezentaci deklarácí jednotlivých elementů jazyka.

### `swer::class_declaration`

Slouží pro reprezentaci deklarace třídy. Třída má jméno (`m_identifier`), seznam identifikátorů generických argumentů (`m_identifiers`) a seznam deklarácí prvků třídy (`m_declarations`).





Obrázek 4.1: Závislost deklaračních tříd

`class_declaration` tvoří instanci třídy `swer::class_declaration` ze sekvence deklarací `class_body` (`function_variable_class_declaration_seq`).

`swer::function_declaration`

Reprezentuje deklaraci metody. Ta obsahuje jméno metody (`m_identifier`), typ návratové hodnoty (`m_ret_type`), seznam formálních parametrů (`m_formal_parameters`) a tělo metody (`m_body`). Formální parametr je reprezentovaný třídou `swer::formal_parameter`, která sdružuje typ parametru (`m_type`) a jméno s případnou implicitní hodnotou (`m_ident_init`).

Jméno parametru s případnou implicitní hodnotou je uloženo ve struktuře `identifier_initialization`, která si kromě jména identifikátoru pamatuje zda byla inicializace vykonána prostřednictvím `identifier = initialization` nebo `identifier{initialization}`. To je důležité pro rozlišení volání konstruktoru nebo kopírovacího konstrukturu třídy ze standardní knihovny Swer.

Struktura `identifier_initialization` má navíc proměnnou `location`, která obsahuje číslo řádku výskytu ve vstupním souboru formátu Swer. To je kvůli lepšímu chybovému výstupu v případě, že se uživatel snaží na jedné úrovni zásobníku definovat dvě různé proměnné se stejným názvem.

Z pravidel `formal_parameter` se složí `formal_parameter_list`, který se redukuje na `formal_parameters`. Z toho se pak tvoří `function_declaration`.

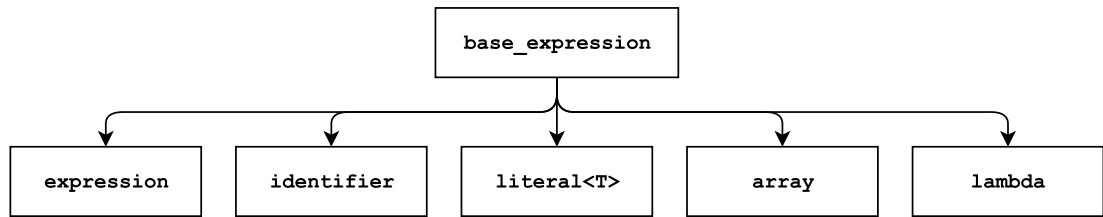
`swer::variable_declaration`

Tato třída podporuje deklaraci více proměnných stejného typu v jednom příkaze. Deklarace proměnné (resp. proměnných) se skládá z typu (`m_type`) a seznamu identifikátorů s případnou inicializací. To je taktéž zajištěno strukturou `identifier_initialization`.

Pravidel pro tvoření deklarace proměnných je více. `variable_declaration` používá `identifier_with_opt_initialization_seq`, která má nepovinnou inicializační část a proměnných může být deklarovaných více najednou. Oproti tomu `variable_declaration_with_initialization` vyžaduje deklaraci právě jedné proměnné a má povinnou inicializační část.

`swer::base_expression`

Slouží jako společný předek všech výrazů, které vznikají v průběhu syntaktické analýzy. Následující sekce popisují sémantiku odvozených tříd.

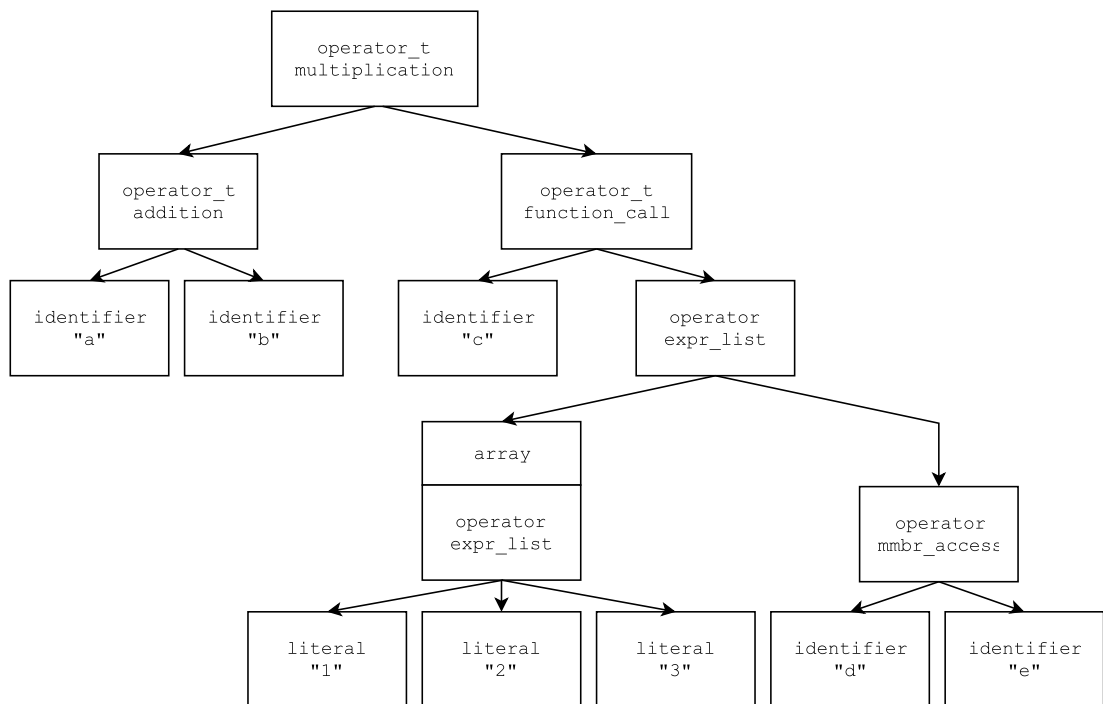


Obrázek 4.2: Závislost výrazových tříd

`swer::expression`

Připomínáme, že syntaktická analýza buduje odspodu strom. I výraz je v paměti reprezentován jako strom, kde listy jsou jednotlivé, dále již nedělitelné, konstanty (identifikátor, literál, pole nebo lambda). Vnitřní uzly stromu jsou právě typu `swer::expression`. Tato třída, kromě svých synů (`m_expressions`), obsahuje také `m_operator`. To je proměnná typu `operator_t`, která říká, jaký operátor byl použit pro syny.

Následuje diagram, který ilustruje jak vypadá v paměti výrazový strom. Obrázek popisuje výraz  $(a + b) * c([1, 2, 3], d.e)$ .



Obrázek 4.3: Visualizace výrazového stromu

Následující tabulka ukazuje přehled operátorů a jejich funkcí.

operátor	sémantika	operátor	sémantika
addition	$v_1 + v_2$	logical_not	$\neg v$
subtraction	$v_1 - v_2$	logical_and	$v_1 \wedge v_2$
multiplication	$v_1 \times v_2$	logical_or	$v_1 \vee v_2$
division	$v_1 \div v_2$	bitwise_not	$\sim v$
modulation	$v_1 \bmod v_2$	bitwise_and	$v_1 \& v_2$
exponentiation	$v_1^{v_2}$	bitwise_or	$v_1   v_2$
unary_plus	$+v$	bitwise_xor	$v_1 \wedge v_2$
unary_minus	$-v$	left_shift	$v_1 \ll v_2$
pre_increment	$++v$	right_shift	$v_1 \gg v_2$
pre_decrement	$--v$	address_of	$\&v$
post_increment	$v++$	dereference	$*v$
post_decrement	$v--$	member_access_dot	$v_1.v_2$
factorial	$v!$	member_access_right_arrow	$v_1 \rightarrow v_2$
conditional	$c ? v_1 : v_2$	assignment	$v_1 = v_2$
less	$v_1 < v_2$	addition_assignment	$v_1 += v_2$
greater	$v_1 > v_2$	subtraction_assignment	$v_1 -= v_2$
less_equals	$v_1 \leq v_2$	multiplication_assignment	$v_1 *= v_2$
greater_equals	$v_1 \geq v_2$	division_assignment	$v_1 /= v_2$
equals	$v_1 = v_2$	modulation_assignment	$v_1 \%= v_2$
not_equals	$v_1 \neq v_2$	exponentiation_assignment	$v_1 = v_1^{v_2}$
subscript	$v_1[v_2]$	left_shift_assignment	$v_1 \ll= v_2$
function_call	$v_1(\dots)$	right_shift_assignment	$v_1 \gg= v_2$
expr_list	$v_1, \dots$	bitwise_and_assignment	$v_1 \&= v_2$
braced_list	$\{v_1, \dots\}$	bitwise_or_assignment	$v_1  = v_2$
expression	$(v)$	bitwise_xor_assignment	$v_1 \wedge= v_2$

`swer::identifier`

Jedním ze základních výrazů (v řeči stromu listových výrazů), se kterým se potýkáme při běhu syntaktické analýzy je identifikátor. Ve složeném výrazu reprezentuje všechna jména (funkce, třídy, proměnné). Jediným prvkem této třídy je jméno identifikátoru.

`expression_base` a `expression_postfix` jsou pravidla, která vytváří instance této třídy.

`swer::literal`

Je generická třída reprezentující literál (kromě pole, to používá vlastní třídu). `literal` může být textový, celočíselný nebo desetinný. Jediným prvkem této třídy je `m_literal` obsahující hodnotu literálu.

Tato třída se vytváří v pravidle `literal`.

`swer::array`

Speciálním literálem je pole. Pole není možné implementovat jako `swer::literal<T[]>`, protože vzniká v několika krocích syntaktické analýzy.

Pole je totiž skupina výrazů (`operator_t::expr_list`), které vznikají v pravidle `expression_list` a tedy jsou typu `swer::expression`. Prvky pole jsou uloženy v položce `m_elements`.

Tato třída se vytváří v pravidle `literal`.

`swer::lambda`

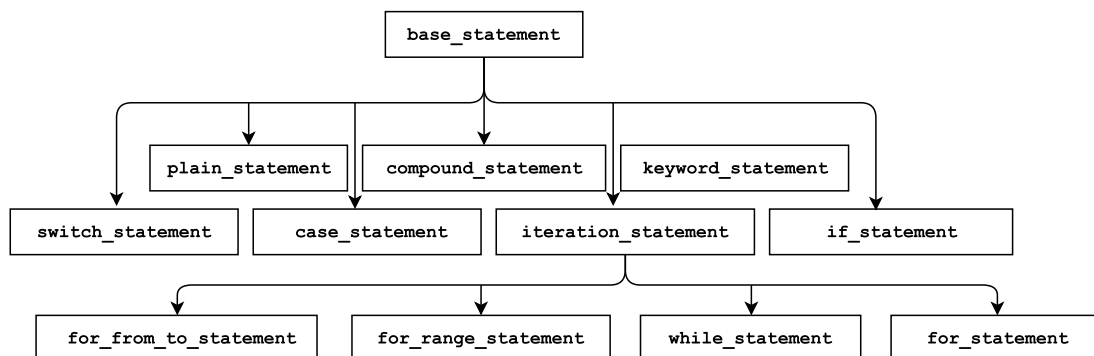
Dalším ze základních výrazů je lambda. Ten se skládá z příkazu (`m_statement`) a seznamu identifikátorů reprezentujících jména vstupních argumentů (`m_identifiers`). Protože se předpokládá primární užití lambda výrazů v metodách akceptujících jako jeden z parametrů funkci, není nutné znát typy argumentů lambda výrazu (díky tomu, že standard C++14 podporuje polymorfní lambda výrazy).

Pravidla související s vytvářením lambda výrazů jsou `lambda_introducer` (seznam argumentů) a `expression_base` (vytvoření instance třídy `lambda`).

`swer::operator_object`

Posledním výrazem je objekt operátor. Technicky se jedná pouze o konkrétní lambda výraz ve tvaru `(lhs, rhs) => {return lhs op rhs}`, kde `op` je z položky `m_type` typu `operator_type`. Ten může nabývat hodnot: PLUS (+), MINUS (−), MULTI (×), DIVIDE (÷), MODULO (funkce `mod`) nebo EXPONENT (funkce `exp`).

`swer::base_statement`



Obrázek 4.4: Závislost příkazových tříd

`swer::plain_statement`

Je generickou třídou, která slouží pro zabalení základních konstruktů jazyka Swer. Mezi ty patří `variable_declaration` (sekce 4.3) a `base_expression` (sekce 4.3). To jsou konstrukty, které mohou být používány i mimo příkaz (např. jako inicializace proměnné u `base_expression` nebo použití `variable_declaration` ve třídě). Tato třída obsahuje pouze zabalený objekt (`m_base`) určený generickým argumentem.

Mezi pravidla patřící k `plain_statement` patří `statement_dangling_else`, `statement` a `declaration_statement`.

## `swer::compound_statement`

Slouží pro seskupování více příkazů do jednoho (v kódu Swer, stejně jako v C-like jazycích, pomocí syntaxe { `stmt`; `stmt`; }). Jedinou položkou této třídy je sekvence příkazů (`m_statements`).

Instance této třídy je vytvářena redukcí pravidla `compound_statement`.

## `swer::keyword_statement`

Reprezentuje příkaz, který je samotným klíčovým slovem. Klíčová slova jazyka Swer sdružuje enumerace `swer::keywords`.

<code>swer::keywords</code>	Swer syntaxe	C-like ekvivalent
RETURN	->; ->expr;	return; return expr;
BREAK	brk;	break;
CONTINUE	cnt;	continue;

Třída `keyword_statement` obsahuje jako prvek identifikaci klíčového slova (`m_keyword`) a případný argument klíčového slova (`m_expression` pro RETURN).

Pravidla `statement` a `statement_dangling_else` zajišťují vznik instance této třídy.

## `swer::if_statement`

Sémantiku podmínkového příkazu zajišťuje třída `if_statement`. Ta obsahuje položky `m_condition`, `m_if_branch` a `m_else_branch`. `m_condition` reprezentuje podmínku, na základě které se vyhodnocuje která větev bude použita. Jazyk Swer, podobně jako C++, umožňuje v podmínce příkazu `if` syntakticky zapsat deklaraci proměnné (vyžaduje povinnou inicializaci). Deklarace proměnné (sekce 4.3) není výraz (sekce 4.3). Proto je podmínka zabalená ve vlastní třídě `swer::condition`.

Instance `if_statement` vzniká v pravidlech `if_statement_dangling_else` a `if_statement`.

## `swer::switch_statement`

Druhým příkazem, který by šel označit jako podmínkový, je *switch*. Třída `switch_statement` obsahuje tyto položky: `m_condition` – hodnota, podle které se *switch* provádí, a `m_statements` – jednotlivé větve příkazu *switch*. Ačkoliv typem `m_statements` je ve skutečnosti sekvence ukazatelů na `base_statement` (sekce 4.3), skutečné typy za běhu překladače jsou instance třídy `case_statement`.

Příkazy *switch* vznikají v pravidle `switch_statement`.

## `swer::case_statement`

Slouží jakožto větve příkazu *switch*. Třída `case_statement` obsahuje položky `m_case` (výraz porovnávaný vůči podmínce) a `m_statements`. Pokud je `m_case` prázdný, resp jeho hodnota je `nullptr`, jedná se o `default` větev.

Tato třída obsahuje také metodu `ends_with_continue`, která vrací `true` v případě, že posledním příkazem daného `case` labelu je *continue*. Ta je užitečná při generování kódu, díky sémantice příkazu *switch* v jazyce Swer.

Pravidlo `case_statement` zajišťuje vytvoření instance `case_statement`, `case_statement_seq` pak jeho sdružování.

`swer::iteration_statement`

Je společným předkem všech iteračních příkazů. Diagram znázorňující tuto závislost je na obrázku 4.4. Tato třída obsahuje pouze `m_body`, protože tělo iteračního příkazu je jediná společná položka zbylých příkazů.

`swer::for_from_to_statement`

Reprezentuje iterační příkaz s pevným počtem opakování. Třída obsahuje položky `m_identifier` (řídící proměnná cyklu), `m_from`, `m_to` (výrazy pro začínající a koncovou hodnotu cyklu). Kromě těchto položek, jejichž hodnoty třída dostává od gramatiky, obsahuje třída ještě `m_from_var` a `m_to_var`. Ty reprezentují identifikátory nutné pro generování kódu cyklu *for*, které si vyžádá od třídy `swer::compiler`.

Pravidla `iteration_statement`, `iteration_statement_dangling_else`, `implicit_for_statement`, `implicit_for_statement_dangling_else` vytvářejí instance této třídy.

`swer::for_range_statement`

Mechanismus *for-each* iterující přes rozsah je reprezentovaný touto třídou. Mezi položky této třídy patří `m_identifier`, identifikátor řídící proměnné, a `m_range`, výraz přes který se iteruje.

Instance této třídy jsou vytvářeny ve stejných pravidlech, jako v předchozí třídě, tedy `iteration_statement`, `iteration_statement_dangling_else`, `implicit_for_statement`, `implicit_for_statement_dangling_else`.

`swer::while_statement`

Dalším iteračním příkazem je *while* cyklus. Jedinou položkou této třídy je `m_end_condition`, což je výraz, jehož hodnota signalizuje konec provádění iterací.

Pravidla gramatiky související s cyklem *while* jsou `iteration_statement` a `iteration_statement_dangling_else`

`swer::for_statement`

Posledním iteračním příkazem je klasický *for* cyklus z C-like jazyků. Tato třída obsahuje položky pro inicializaci (`m_initialization`), podmínku (`m_condition`) a krok po provedení iterovaného příkazu (`m_step`). Třída obsahuje ještě položku `m_variable`, která je inicializovaná v případě, že v inicializační části cyklu byla použita deklarace proměnné. V tomto případě je `m_initialization` nastaveno na `nullptr`.

Jako u ostatních iteračních příkazů i tady platí, že vytváření instancí této třídy se odehrává v pravidlech `iteration_statement_dangling_else` a `iteration_statement`.

## Řešení chybových stavů

V případě, že při běhu překladače zdrojového kódu `Swer` dojde k nějaké chybě, používá se globální proměnná `global_logger` ukazující na instanci třídy `simple_logger`. Instance této třídy se vytvoří ještě před zpracováváním argumentů příkazové řádky. To je z důvodu, že i během této operace může nastat chybový stav. Třída `simple_logger` má několik metod na oznámení nějakého stavu.

Metoda `error` slouží pro oznámení chyby, která souvisí s během programu a ne přímo s překladem (během syntaktické analýzy). Používá se v případě nevalidních argumentů příkazové řádky, nemožnosti otevřít soubor pro čtení nebo jiné závažné chyby.

Metoda `compilation_error` oznamuje chyby při běhu syntaktické analýzy – nevyhovující formát vstupního souboru, neukončený textový řetězec, deklarace již existující proměnné. Metoda kromě vlastního oznámení chyby ještě chybové stavy počítá. To proto, aby se zbytečně negeneroval program, pokud syntaktická analýza skončí s chybami.

Překladač `Swer` nedetekuje všechny chyby. Velké množství sémantických chyb nechává na následném běhu překladače C++, jehož mechanismus detekcí a oznámení chybových stavů bývá zpravidla velmi kvalitní.

## 4.4 Modul `swer::compiler`

Po skončení běhu syntaktické analýzy má třída `compiler` k dispozici program (poslední redukované pravidlo *program*). To je sekvence deklarací a příkazů. Většina tříd z předchozí sekce *modul swer* disponuje metodou `as_string`, která definuje způsob, jak se z dané třídy generuje zdrojový kód C++. V případě, že syntaktická analýza došla bez chyby, spustí se generování kódu C++. To je iniciováno voláním funkce `generate_program`. Ta v závislosti na nastavení (třída `options`) volá buď funkci `generate_program_to_command`, která spustí příkaz a na jeho standardní vstup zapisuje vygenerovaný kód C++, nebo funkci `generate_program_to_stream`, která generuje výstup do proudu (standardní výstup nebo soubor).

Obě výše zmíněné funkce používají jedinou funkci, která zajišťuje vlastní generování kódu. Jedná se o funkci `generate_program_by_function`, která jako parametr přijímá funkci `F`, kterou volá s parametrem reprezentujícím textový řetězec s právě generovanou částí kódu. Od funkce `F` se očekává, že zajistí vypsání textového řetězce na správný výstup.

Pro jednodušší zápis implementace metod `as_string` ze tříd modulu `swer` jsou v souboru `src/utilities.hpp` definované pomocné funkce. Mezi tyto funkce patří `concat`, která variabilní počet argumentů spojí a vrátí jako jeden textový řetězec. Další funkcí je `string_join`, která jako parametr dostane rozsah (buď pomocí iterátorů z jazyka C++, nebo přímo celý kontejner), oddělovač a případný funktor. Funkce vrací dohromady spojené všechny prvky z daného rozsahu, kde prvek rozsahu se na textový řetězec převádí pomocí funktoru a prvky jsou oddělené daným oddělovačem.

Protože častým funktorem používaným ve funkci `string_join` je funkce, která na svém argumentu volá `as_string`, obsahují utility také definici pomocné funkce

`as_string_decay`, která na svém argumentu volá právě `as_string`.

Poslední pomocné funkce jsou `template_args_decl` a `template_args_use` používané při generování definic vnořených tříd nebo funkcí.

Implementace funkcí `as_string` u většiny tříd z modulu `swer` je triviální a jedná se jen o syntaktický převod mezi různými klíčovými slovy v jazycích Swer a C++. Proto se v následujícím textu zaměříme pouze na některé implementace funkcí `as_string`. Nejprve se ale podívejme na některé pomocné funkce třídy `swer::compiler`.

### `get_tmp_variable_identififier`

Generování některých fragmentů gramatiky do kódu C++ vyžaduje zavedení a použití nových dočasných proměnných. Jména těchto proměnných je třeba volit s uvážením. Nesmí kolidovat s jinými proměnnými, které definoval uživatel a zároveň musí být unikátní i v rámci následujících volání této funkce. Identifikátor má tvar `zakladindex`, kde `zaklad` je fixní textový řetězec a `index` je s každým následujícím voláním zvětšován o jedničku.

Aby nedošlo ke kolizím mezi identifikátory, které vrací tato funkce a případnými uživatelsky definovanými proměnnými, je `zaklad` zvolen jako textový řetězec, který v jazyce Swer není validní pro identifikátor. Jazyk C++ od verze C++11 umožňuje použití *universal-character-name* v identifikátoru [4]. Identifikátor `\uAAAA` je validním v C++, ale nejedná se o identifikátor jazyka Swer. Proto není možné, aby došlo ke kolizím s uživatelsky definovanými proměnnými.

### `swer::compiler::declaration_map`

Tato třída se stará o správu deklarovaných proměnných. V případě, že syntaktická analýza narazí na deklaraci proměnné nebo deklaraci vyžaduje implicitní proměnná některých iteračních cyklů, volá se funkce `declare` této třídy. Ta si pamatuje pro úroveň zásobníku množinu deklarovaných proměnných a je schopná vrátit informaci o tom, zda je proměnná deklarovaná. Úroveň zásobníku je řízena funkcemi `enter_block` a `leave_block` třídy `compiler`. Ty jsou volány při redukci pravidel gramatiky, které mění hloubku zásobníku.

### `get_iteration_identififier`

Jak víme ze sekce 2.5, některé iterační cykly používají implicitně definovanou proměnnou. Její hodnotu vrací právě funkce `get_iteration_identififier`. Ta zjistí počet vnořených iteračních cyklů bez explicitně definované řídicí proměnné, podívá se, jestli již náhodou není taková proměnná deklarovaná jinde, a pokud ne, tak vrátí její identifikátor.

Nyní si již můžeme popsat netriviální implementace metod `as_string`.

### `for_from_to_statement`

Připomínáme, že se jedná o pevný cyklus *od-do*. Protože C++ nemá zápis pro takovou formu cyklu jako např. Pascal, je nutné použít běžný *for* cyklus. V první řadě je nutné generovat kód, který vyhodnotí argumenty *for* cyklu do dočasných proměnných. Jména těchto proměnných získáme pomocí funkce



`get_tmp_variable_identifier`. Deklarace těchto proměnných jsou samostatné příkazy a je nutné je spolu s vygenerovaným *for* cyklem zabalit do složeného příkazu pro případ, že by se *for* cyklus vyskytoval na místě, kde se očekává jen jeden příkaz (například větev *if*). Pak je nutné brát ohled na fakt, že použití, kde *od* > *do*, je také validní a vyžaduje iteraci směrem dolů (řídící proměnná se bude snižovat). Generovaný kód musí proto pokrývat obě možnosti.

*Příklad.* Pro příkaz `@i(5..10) stmt;` se generuje

```
{
    const auto& tmp1 = 5;
    const auto& tmp2 = 10;
    for (
        auto i = tmp1;
        tmp1 < tmp2 ? i < tmp2 : i > tmp2;
        tmp1 < tmp2 ? ++i : --i
    )
        stmt;
}
```

## case\_statement

Při generování každé větve příkazu *case* je třeba zjistit jaký je poslední z příkazů dané větve (pokud příkaz existuje) a na základě toho vygenerovat případný `break;`, případně negenerovat poslední příkaz `continue;`.

## swer::literal

Každý literál se generuje jinak. Celé číslo vyžaduje doplnění suffixu `ll`, desetinné nevyžaduje žádný suffix. U znaku je potřeba jej zabalit do apostrofů a textový řetězec voláním konstruktora třídy `String` ze standardní knihovny `Swer`. Tohoto by bylo možné dosáhnout parciálními specializacemi pro některé literály, ale vyžadovalo by to velké množství kódu. Proto se místo toho používá přetížená metoda `to_string_hlp`, která zajišťuje správné generování jednotlivých literálů.

## function\_declaration

Překladač C++ je jednopřechodový – zdrojový kód čte jen jednou (pokud pomineme šablony). Z tohoto důvodu je nutné v některých případech oddělit deklaraci od definice. Jedním z takových případů je cyklické volání funkcí. Pokud `A` volá funkci `B`, je nutné deklarovat `A` před `B`. Pokud ale zároveň funkce `B` volá funkci `A`, dostaneme cyklickou závislost, kterou je nutné rozbít pomocí oddělené deklarace a definice. Funkce se do zdrojového kódu proto generuje ve dvou částech – deklarace a pak definice. `Swer` umožňuje deklarovat funkce uvnitř tříd a vnořené třídy. To způsobuje problémy s generováním definic funkcí. Obecná definice funkce má v C++ následující zápis:

```

template<typename T11, typename T12, ... typename T1N1>
template<typename T21, typename T22, ... typename T2N2>
...
template<typename TM1, typename TM2, ... typename TMNM>
return_type
T1<T11, T12, ...T1N1>::
T2<T21, T22, ...T2N2>::
...
TN<TM1, TM2, ...TMNM>::
identifier(params){ code; }

```

Kde TI je I-tá vnořená třída s generickými argumenty T11, ..., TIN1. Deklarace nevyžaduje použití generických argumentů ani vnořených tříd, pokud se buď v žádné třídě nevyskytuje a nebo je deklarovaná přímo v kódu dané třídy. Definici získáme voláním funkce `define` s parametrem obsahujícím seznam nadřazených tříd v pořadí, ve kterém jsou zanořené. Metoda `declare` vrací deklaraci.

## class\_declaration

Větší problém nastává u deklarací tříd. Na nejvyšší úrovni programu není možné správně oddělit deklaraci a definici tak, aby každá metoda každé třídy viděla vše ostatní. Například pro následující kód

```

struct A { void fncA(){ B b; b.fncB(); } };
struct B { void fncB(){ A a; a.fncA(); } };

```

je nutné oddělit deklarace do různých hlavičkových a zdrojových souborů. Swer ale vždy generuje jen jeden soubor a kvůli tomu není možné předchozí problém vyřešit. Nicméně stejné sémantiky můžeme dosáhnout alternativním způsobem, například

```

A{}
B{}

$ fncA(A~a) { B b; fncB(b); }
$ fncB(B~b) { A a; fncA(a); }

```

Tento problém je možné vyřešit mezi dvěma třídami, které jsou vnořené v jedné společné. C++ totiž umožňuje deklarovat třídu uvnitř jiné třídy a její definici specifikovat mimo vnořenou třídu. Obecná definice třídy má podobnou syntaxi jako funkce:

```

template<typename T11, typename T12, ... typename T1N1>
template<typename T21, typename T22, ... typename T2N2>
...
template<typename TM1, typename TM2, ... typename TMNM>
struct
T1<T11, T12, ...T1N1>::
T2<T21, T22, ...T2N2>::
...
TN<TM1, TM2, ...TMNM>::
identifier { declarations };

```

Implementace metody `as_string` tedy prochází deklarace (`m_declarations`), prvně generuje deklarace vnořených tříd, pak deklarace funkcí, poté případné položky třídy. Tímto končí deklarace třídy a pokračuje definice vnitřních tříd a definice vnitřních funkcí.

Nyní již víme jak generovat kód jednotlivých elementů jazyka Swer. Modul `compiler` pak generuje program takto:

`#include <swer_stl>` – připojení standardní knihovny Swer.

Deklarace tříd

Deklarace funkcí

Definice tříd

Definice funkcí

Signatura funkce `main`, vytvoření pole z argumentů příkazové řádky

Začátek bloku `try`.

Příkazy

Ukončení bloku `try`, generování bloku `catch` a konec funkce `main`.

Zmíněný blok `try-catch` slouží pro lepší diagnostiku chyb, které jsou způsobené nesprávným použitím mechanismů ze standardní knihovny. Blok `catch` zachytává výjimky typu `std::exception`, které sice nepokryjí všechny objekty, které lze příkazem `throw` v C++ vyhodit, ale většinu chybových stavů způsobených standardní knihovnou to zahrnuje.

## 5. Standardní knihovna Swer

Standardní knihovna jazyka Swer je implementována v jazyce C++. Vygenerovaný kód C++ při překladu vyžaduje připojení této knihovny. Knihovna obsahuje mechanismy pro práci s čísly (sekce 5.1), textovými řetězci (sekce 5.2), poli (sekce 5.5) a mapami (sekce 5.6). Součástí standardní knihovny jsou také metody zpřístupňující práci s výstupem (sekce 5.3) a vstupem (sekce 5.4), a další užitečné algoritmy (sekce 5.7) převážně z oblasti funkcionálního programování. Většina metod, tříd a typů ze standardní knihovny má dvě naprosto ekvivalentní verze. Jedna má nezkrácený, výřečný název, ze kterého je zřejmá sémantika. Druhá má název zkrácený na 1-3 znaky nebo se její použití vyvolá pomocí operátoru. V následujících sekcích jsou jednotlivé mechanismy označovány jejich plným názvem. Pokud daný mechanismus má zkrácenou verzi, je uveden v dané sekci zpravidla ke konci.

Celá standardní knihovna je implementovaná ve jmenném prostoru `swerstl`. Ačkoliv jazyk Swer nezná pojem jmenný prostor, je schopen funkce používat bez nutnosti uvedení plně kvalifikovaného jména (např. `swerstl::String`). To je zajištěno tím, že hlavní soubor celé standardní knihovny – `src/swer_stl`, který obsahuje `#include` všech částí knihovny, obsahuje také příkaz `using namespace swerstl;`.

### 5.1 Číselné typy

Swer obsahuje dva číselné typy a jeden znakový, implicitně převeditelný na číslo.

#### Integer

Jedná se o celé číslo, velikosti alespoň 64 bitů, ekvivalentní typu `long long int` z C++. Literál tohoto typu je ze specifikace Swer (kapitola 2) definován pravidlem `integer-literal`.

Typ `I` je ekvivalentní typu `Integer`.

#### Floating

Představuje desetinné číslo, representované typem `double` z C++. Z toho plyne, že má vlastnosti *IEEE-754 64-bit floating point type*. Pravidlo gramatiky `floating-literal` definuje literál typu `Floating`.

Zkráceně lze tento typ označovat jako `F`.

#### Character

Reprezentuje jeden znak. Odpovídá přesně typu `char` z jazyka C++. Typ `Character` je implicitně převeditelný na číslo. Definice literálu je specifikována pravidlem `character-literal`.

`C` je zkrácená verze pro `Character`.

Než si představíme další z typů ze standardní knihovny, ukažme si nejprve následující metodu.

```
detail::circular_indexer(Container c, Integer pos)
```

Tato metoda se používá při indexování `String` nebo `Vector`. Pro `pos` z rozsahu `[0, c.size())` má stejné chování jako `c[pos]`. Pokud je `pos < 0`, vrátí `c[c.size() + pos]`, tedy element na pozici `pos` od konce. Pokud je `pos` mimo rozsah, je chování ekvivalentní `circular_indexer(c, pos ± c.size())`, kde  $\pm$  je  $+$  v případě, že `pos > 0`, jinak  $-$ .

Tyto vlastnosti vytvářejí iluzi cyklického kontejneru, ve kterém za posledním elementem následuje opět první.

Chování `circular_indexer(c, pos)` lze také popsat takto:

Pokud `c.size() == 0`, dojde k výjimce.

Pokud `pos >= 0`, vrátí `c[pos % c.size()]`.

Pokud `pos < 0`, vrátí `c[c.size() - 1 - pos % c.size()]`.

## 5.2 Textový řetězec – `swerstl::String`

Dalším základním typem, který poskytuje knihovna `Swer`, je textový řetězec. Velmi se podobá typu `std::string` a jeho použití je podobné. Kromě následujících funkcí obsahuje třída `String` ještě všechny metody z `std::string`, především relační operátory, `operator+` a `operator+=`, konstruktory a metody vracející iterátory – `begin`, `end` a jejich `r-` a `c-` verze.

`Integer size()`

Vrátí délku řetězce (počet znaků, které řetězec obsahuje). Je k dispozici jako položka `c` (z anglického *count*).

`String operator*(String str, Integer n)`

Vrátí `n` krát zduplikovaný řetězec `str` spojený do jednoho. Třída obsahuje také stejný operátor, který přijímá argumenty v opačném pořadí.

`Character& operator[](Integer pos)`

Vrací referenci na znak na pozici `pos` pomocí `detail::circular_indexer`.

`String reverse()`

Otočí textový řetězec tak, že vymění první a poslední znak, druhý a předposlední, atd., a vrátí otočený `String`. Je k dispozici ve zkrácené verzi `String rv()` a jako položka třídy `r`.

`Vector<String> split(String delimiters = ",")`

Rozdělí textový řetězec na slova, která jsou oddělená znaky předanými v argumentu `delimiters`. Jako oddělovač je brán libovolný znak z `delimiters`. Je ekvivalentní funkci `Vector<String> sp(String delimiters = ",")` nebo položce `s`, která má jako oddělovač definovaný pouze znak `' '`.

`Integer to_integer()`

Převede text ze `String` na `Integer`. Pokud konverze není možná, vrátí `Integer` s hodnotou 0.

`Vector<Character> to_vector()`

Z textového řetězce vytvoří pole znaků, tak, že *i*-ty znak řetězce bude na *i*-té pozici v poli.

## 5.3 Výstup

Swer dovoluje vypisovat libovolný text pouze na standardní výstup. Vypsání textu se iniciuje pomocí funkce `print`, která má variabilní počet argumentů typu `String`, `Integer`, `Floating`, `Character`, `Vector` nebo `Map`.

Výstup pro typ `Vector<T>` je definovaný jako `t1 t2 ... tN`, kde `ti` je textová reprezentace *i*-tého prvku pole typu `T`.

Pro typ `Map<K, V>` vypadá definice výstupu takto

```
k1 v1
k2 v2
...
kN vN
```

kde `ki` je klíč a `vi` je hodnota odpovídající tomuto klíči (tedy pro `Map<K, V>` mapa `vi` odpovídá výrazu `mapa[ki]`).

Metoda `print` má dvě zkrácené verze – funkce `pr` je ekvivalentní funkci `print`. Funkce `p` navíc vytiskne za posledním argumentem znak pro nový řádek (`\n`).

## 5.4 Vstup

Podobně jako výstup i pro vstup lze použít jen standardní vstup. Načítání dat ze vstupu zajišťují následující metody:

`String read_to_end()`

Přečte celý obsah standardního vstupu do textového řetězce, který metoda vrátí. Také je k dispozici prostřednictvím metody `String re()`.

`String read_line()`

Tato metoda čte standardní vstup do té doby, než narazí na znak konce řádku. Poté aktuálně načtený text vrátí. Zkrácená verze této metody má formát `String rl()`.

`T read()`

Přečte následující element ze standardního vstupu na základě typového argumentu. Swer neumožňuje volat funkce, které požadují dedukovaný typový argument, a proto existují následující verze této funkce.

`String rs()` načte následující slovo (text až do následujícího bílého znaku).

`String ri()` načte následující číslo.

`String rf()` načte následující desetinné číslo.

`String rc()` načte následující znak.

`String rv(Vector<T>&)` čte prvky typu `T` dokud to jde, a vrátí z nich vytvořený vektor jako návratovou hodnotu a také v argumentu předaného referencí.

`String rm(Map<K, V>&)` čte dvojice prvků `K` a `V` a vytváří z nich mapu, kterou následně vrátí jakožto návratovou hodnotu a také jako argument předaný referencí.

Poslední dvě funkce z výše uvedeného seznamu mají kromě návratové hodnoty ještě argument reference proto, že C++ neumí dedukovat typový argument jen na základě návratové hodnoty, a tedy výraz `Vector<Integer> vec = rv();` v jazyce Swer by nebyl přeložitelný následným překladem do C++.

## 5.5 Pole – `swerstl::Vector`

Základním kontejnerem, který nabízí jazyk Swer, je pole. Implementace pole je řešena pomocí C++ třídy `std::deque`, která umožňuje přidávat elementy na začátek i na konec, a indexovat do pole. `swerstl::Vector` nabízí většinu metod, které implementuje `std::deque`. Navíc pak zpřístupňuje následující metody.

`Vector<T> operator+(const Vector<T>& lhs, const Vector<T>& rhs)`

Spojí dohromady dvě pole tak, že za poslední prvek `lhs` postupně vloží od začátku všechny prvky `rhs` a nově vytvořené pole vrátí jako návratovou hodnotu.

`Vector<T> operator+=(const T& value, Vector<T>& vec)` (1)

`Vector<T> operator+=(Vector<T>& vec, const T& value)` (2)

`Vector<T> operator+=(const Vector<T>& values, Vector<T>& vec)` (3)

`Vector<T> operator+=(Vector<T>& vec, const Vector<T>& values)` (4)

Připojí element (ve variantách 1 a 2), resp. elementy (varianty 3 a 4), na začátek (1 a 3), resp. na konec (2 a 4) pole.

`T& operator[](Integer pos)`

Vrací referenci prvku znak na pozici `pos` pomocí `circular_indexer` (sekce 5.1).

`void insert(Integer index, const T& item)`

`void insert(Integer index, const Vector<T>& items)`

Vloží za pozici `index` element, resp. elementy. V tomto případě je `index` skutečná pozice a nedochází k použití mechanismu `detail::circular_indexer`. Zkráceně lze volat tuto funkci jako `in`.

`void erase(Integer index)`

`void erase(Integer from, Integer to)`

`Vector<T>& operator-(const T& item)`

Z pole odstraní prvek na pozici `index`, resp. prvky z rozsahu `[from, to)`. Zkrácená verze této funkce má název `rm` (z *remove*). Další možnost odstranění je podle hodnoty prvku a to zajišťuje `operator-`. Díky asociativitě operátoru `-` je možné

odstranění prvků 1, 2, 3 z pole `vec` zajistit voláním `vec - 1 - 2 - 3`.

`Vector<T> sort()`

Seřadí prvky pole vzestupně, kde porovnání dvou prvků vykonává `std::less<T>`, resp `operator<`. Další možností, jak docílit seřazení pole, je funkcí `sr` nebo položkou `s`.

`Vector<T> reverse()`

Vrátí nové pole, kde jsou prohozené prvky první a poslední ; druhý a předposlední atd. Je k dispozici také pod názvem `rv()` nebo jako položka `r`.

`Integer size()`

Vrátí počet prvků pole. Zkrácená verze umožňuje použití pomocí `sz()` nebo položky `c` (z anglického *count*).

## 5.6 Mapa – `swerstl::Map`

Posledním kontejnerem, který nabízí knihovna jazyka Swer, je mapa. Ta umožňuje definovat dvojice klíč–hodnota. Třída `std::unordered_map` z C++ je použita pro implementaci `Map`. Hlavní rozdíl mezi těmito třídami je v iterátorech. Dereferencí `unordered_map<K, V>::iterator` získáme `std::pair<K, V>`, a pro `map<K, V>::iterator` dostaneme `swerstl::pair<K, V>`. Třída `pair` ze standardní knihovny Swer má ekvivalentní použití jako třída `std::pair`. Rozdíl je v tom, že místo `first` se používá `f` a místo `second` se používá `s`.

`V& operator[] (const K& key)`

Pokud existuje, tak vrátí mapovanou hodnotu pro klíč `key`, jinak vytvoří nový prvek v mapě na pozici `key` s defaultní hodnotou typu `V` a vrátí na něj referenci. Použití `circular_indexer` nedává v tomto případě smysl.

`Map<K, V>& operator-(const Key& item)`

Z mapy odebere prvek s klíčem `item` a vrátí referenci na sebe.

`bool contains(const key_type& key)`

Vrací `true` pokud mapa obsahuje prvek s klíčem `key`, jinak `false`. Jazyk Swer sice přímo nepodporuje typ `bool`, ale jeho význam chápe a použití výrazu typu `bool` v podmínce je validní. Je k dispozici také ve zkrácené verzi `ct`.

`Integer size()`

Vrátí počet prvků mapy. Ekvivalentní přístup je pomocí `sz()` nebo položky `c` (z anglického *count*).

`Vector<V> values()`

Vrátí pole, které obsahuje hodnoty z mapy. Jedná se o prvky `map[key]` takové, že `contains(key) == true`. Pořadí prvků v poli je nespecifikované. Zkrácená verze nese název `vl`.



## 5.7 Algoritmy

Knihovní metody ze souboru `src/algorithm.hpp` obsahují převážně funkce z oblasti funkcionálního programování a funkce, které používají operátory jazyka Swer.

`Integer pow(Integer base, Integer exp)`

Funkce vrací číslo `base` umocněné na `exp` ( $\text{base}^{\text{exp}}$ ). Existuje přetížená metoda pro typ `Floating`. Použití je možné také pomocí vestavěného operátoru `base ^ exp` jazyka Swer.

`Integer& pow_assign(Integer& base, Integer exp)`

A opět existuje také přetížená verze pro `Floating`. Na rozdíl od předchozí funkce je přímo modifikovaný argument `base`. Jazyk Swer disponuje operátorem pro použití této metody jednodušeji, pomocí syntaxe `base ^= exp`.

`Integer factorial(Integer value)`

Vrací faktoriál argumentu `value`. Unární postfixový operátor `!` volá tuto metodu.

`Vector<T> generate(const T& from, const T& to)`

Generuje pole prvků `from, from+1, ..., to` včetně. V případě, že `to < from`, vrací prázdné pole prvků typu `T`. Zkrácená verze má signaturu `Vector<T> gen(const T& from, const T& to)`.

`Vector<T> generate(const T& to)`

Je ekvivalentní volání `generate(1, to);`. Také existuje zkrácená verze této metody s názvem `gen`.

`Vector<T> zip(const T& v1, const T& v2, F f)`

Má podobné chování jako funkcionální metoda z jazyka Haskell `zipWith f v1 v2`. Vrací pole, ve kterém `i`-tý prvek má hodnotu `f(v1[i], v2[i])`. Pro jednodušší zápis funkcí je vhodné použít lambda výrazy (sekce 2.4), nebo dokonce objekt operátory (sekce 2.4).

Funkce `zip([1,2,3], [4,5,6], *)` vrátí `[4,10,18]`.

`Vector<T> map(const T& vec, F fnc)`

Další z funkcionálních metod je funkce `map`. Ta dostane pole a vrátí nové pole, kde na každý prvek je volaná funkce `fnc`.

`map([1,2,3], x => {->x+1;})` vrátí `[2,3,4]`.

`Vector<T> take_while(const T& vec, F fnc)`

Metoda `take_while` vrací pole, které obsahuje první elementy z argumentu `vec`, pro které platí podmínka `fnc`. Zkrácená verze má název `tkw`.

Například `tkw([1,2,3,1,2,3], x => {->x<3})` vrátí `[1,2]`.

`Vector<T> filter(const T& vec, F fnc)`

Metoda `filter` má velmi podobné chování funkci `take_while`, až na to, že první neplatnost podmínky `fnc` nekončí průchod polem `vec`. Pro vyvolání zkrácené

verze se používá jméno `flt`.

Volání funkce `flt([1,2,3,1,2,3], x => {->x<3})` vrátí `[1,2,1,2]`.

`T fold(const Container<T> vec, F fnc)`

Poslední funkcí z kolekce algoritmů je `fold`. Ta pro prázdné pole vrací defaultní hodnotu typu `T`. Pro jednoprvkové pole vrací přímo tu jedinou hodnotu a pro větší pole vrací hodnotu, která vznikne postupným aplikováním funkce `fnc` na prvky pole způsobem `fnc(... (fnc(fnc(fnc(vec[0], vec[1]), vec[2]), vec[3]), ...)`. Zkrácený název této funkce je `fld`.

## 5.8 Implementační detaily

Jeden z implementačních detailů je již popsán v sekci 5.1 – cyklický indexer. Některé typy ze standardní knihovny jazyka Swer obsahují položky, které se tváří jako proměnné, které zpřístupňují například velikost kontejneru (metoda `size()` použitelná jako `c`). Jedná se o mechanismus, který bývá v jazyce C# označován jako *getter*, nicméně ačkoliv sémantika je velmi podobná, tak implementace v C++ se liší.

Generická třída `swerstl::detail::getter<T>` je implicitně převeditelná na typ `T` pomocí `operator T()`. Kromě toho má konstruktor, který očekává funktor, který si `getter` uloží a následná konverze jej volá. To umožňuje použití v případech, kdy C++ dokáže odhadnout, že má konverzi použít. To například nejde při použití šablonové metody a tam je vyžadováno explicitní přetypování (resp. volání konstruktoru).

Typický příklad definice proměnné typu `getter` ve třídě `T` pro volání `T::size` by vypadala jako `getter<Integer>(std::bind(&size, this))`. To způsobuje potíže při defaultní implementaci kopírovacího konstruktoru a přiřazovacího operátoru. Ty zkopírují zabalenou funkci se „starým“ ukazatelem `this`. Proto implementace všech metod, které vytváří instance třídy s položkou typu `getter` ještě navíc volají metodu `update_getters`, která zajišťuje znovu-vytvoření zabalené funkce se správnou hodnotou `this`. Tuto metodu je nutné volat i v implementaci metody `swap`.

Některé funkce při nesprávném použití vyhazují výjimky. Swer sice nepodporuje vyhazování ani zachytávání výjimek, ale generovaný kód funkce `main` je obalený *try-catch* blokem, který se snaží zachytit všechny případné výjimky, které způsobí nesprávné použití mechanismů ze standardní knihovny.

## 6. Příklady

Tato kapitola má za cíl ukázat řešení některých úloh z oblasti code-golf v jazyce Swer. Ke každému příkladu je navíc přiložený i kód v C++ pro srovnání. Zhodnocením se zabývá až následující kapitola – Závěr. Pro lepší pochopení chování nejsou uváděny kódy programů ve zkrácené podobě (po odstranění redundantních bílých znaků). Počet bytů, které by program obsahoval po smazání redundantních znaků bývá uveden vpravo, nad ukázkou kódu.

### cat goes “Meow”

*Zadání:* Program, který na standardní výstup vypíše obsah standardního vstupu, pokud vstup neodpovídá přímo textu cat. Pak se na výstup vypíše řetězec cat goes "Meow".

*Řešení ve Swer :*

48 bytes

```
$s = re();
s == "cat"
  ? p(s," goes \"Meow\"")
  : p(s);
```

*Řešení v C++ :*

143 bytes

```
#include <cstdio>
#include <string>
int main()
{
    std::string s;
    while (int i = ~getchar())
        s += ~i;
    printf("%s", (s == "cat" ? s + " goes \"Meow\"" : s).c_str());
}
```

### Fibonacci + Fizz Buzz = Fibo Nacci!

*Zadání:* Program, který vypíše na standardní výstup prvních 100 Fibonacciho čísel (začínajících číslem 1) oddělených novým řádkem za následujících podmínek.

Tento příklad má vadu v tom, že sté Fibonacciho číslo se nevejde do číselného typu `Integer` v jazyce Swer, ani do typu `unsigned long long int` v C++ ( $F(100) > 2^{64}$ ). Z tohoto důvodu by splnění zadání vyžadovalo implementaci aritmetiky nad libovolně dlouhými čísly. Protože to nebyla hlavní myšlenka úlohy, jsou poslední řádky výstupu (pro hodnoty Fibonacciho čísla, které se nevlezly do datového typu) nevalidní (došlo k přetečení).

Pokud je Fibonacciho číslo dělitelné číslem 6, vypíše se místo čísla text FiboNacci.

Jinak, pokud je Fibonacciho číslo dělitelné číslem 2, vypíše se místo čísla text *Fibo*.

Jinak, pokud je Fibonacciho číslo dělitelné číslem 3, vypíše se místo čísla text *Nacci*.

*Řešení ve Swer:*

112 bytes

```
$f(I c)
{
    -> c < 2 ? 1 : f(c-1) + f(c-2);
}

@(0..100)
{
    $l=f(i);
    pr(1 % 2 ? "" : "Fibo");
    pr(1 % 3 ? "" : "Nacci");
    if (1 % 2 & 1 % 3)
        p(1);
    el
        p();
}
```

*Řešení v C++:*

185 bytes

```
#include <iostream>

int f(int c)
{
    return c < 2 ? 1 : f(c-1) + f(c-2);
}

int main()
{
    auto&s = std::cout;
    for(int i = 0; i < 101; ++i)
    {
        int l=f(i);
        s << (1 % 2 ? "" : "Fibo")
          << (1 % 3 ? "" : "Nacci");
        (1 % 2 && 1 % 3 ? s << l : s) << "\n";
    }
}
```

*Poznámka:* Implementace funkce *f* je velmi neefektivní a už pro velmi malá čísla poběží i na moderním zařízení velmi dlouho (řádově roky až stovky let). To ale pro splnění zadání code-golf nevádí.

## String Zip and Sort

*Zadání:* Ze zadaného seznamu řetězců vytvořte textový řetězec, který vznikne tak, že vezmeme znak z každého řetězce postupně z každé pozice, seřadíme je

podle jejich ASCII hodnoty a připojíme na konec vytvářeného řetězce. Můžete předpokládat, že seznam řetězců obsahuje stejně dlouhé řetězce a ty obsahují pouze znaky, které mají ASCII kód v rozsahu [32,127].

*Příklady:*

"abc", "cba" -> "acbbac"

"HELLO", "world", "!!!!!" -> "!Hw!Eo!Lr!Ll!Od"

Pro řešení této úlohy není nutné psát celý program, stačí napsat definici funkce, lambda výrazu nebo podobného mechanismu, který umožňuje jazyk, ve kterém úlohu řešíme.

*Řešení ve Swer:*

51 bytes

```
s => {
  @($j = 0; j < s[0].c; ++j)
  {
    S r;
    @(s)
      r += i[j];
    pr(r.s);
  }
}
```

*Řešení v C++:*

152 bytes

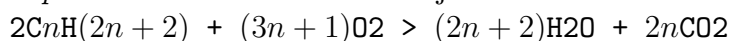
```
#include <iostream>
#include <string>

[](auto s)
{
  for (int i = 0; i < s[0].size(); ++i)
  {
    auto r = "s;
    for (auto k : s)
      r += k[i];
    std::sort(begin(r), end(r));
    std::cout << r;
  }
};
```

## Back to the chemistry class

*Zadání:* Napsat celý program, který ze standardního vstupu přečte text uhlovodíku (alkanu) ve formátu  $C_nH(n+2)$  a na standardní výstup napíše textový řetězec, reprezentující reakci daného uhlovodíku s kyslíkem, vytvářející oxid uhličitý a vodu.

*Tip:* Každá reakce má následující tvar



*Poznámka:* Mezery ve výstupu nejsou nutné, ale pokud se budou konzistentně vyskytovat, je možné si z celkového počtu bytů odečíst 10% jako bonus.

*Řešení ve Swer:*

101 × 0.9 bytes

```
I n=rc();
!(n = ri()) & (n=1);
$t = 2 * n;
p("2C", n>1 ? S(n) : "", "H", t + 2, " + ", t + n + 1, "02 > ", t + 2,
"H2O + ", t, "CO2");
```

*Řešení v C++:*

160 × 0.9 bytes

```
#include <iostream>
int main()
{
    int n = 1, t;
    std::cin.get();
    std::cin >> n;
    t = 2 * n;
    printf("2C");
    n - 1 && printf("%i", n);
    printf("H%i + %i02 > %iH2O + %iCO2", t + 2, t + n + 1, t + 2, t);
}
```

## 16-bit binary grid

*Zadání:* Je zadáno 16 bitové číslo. Převeďte jeho dekadický zápis do 4 × 4 ASCII mřížky jeho bitů, kde nejvýznamnější bit je vlevo nahoře, nejméně významný vpravo dole a bity se zapisují směrem doprava, dolů (jako prostý text).

*Příklady:*

**4242**

```
+---+---+---+---+
|   |   |   | # |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
| # |   |   | # |
+---+---+---+---+
|   |   | # |   |
+---+---+---+---+
```

**33825**

```
+---+---+---+---+
| # |   |   |   |
+---+---+---+---+
|   | # |   |   |
+---+---+---+---+
|   |   | # |   |
+---+---+---+---+
|   |   |   | # |
+---+---+---+---+
```

*Řešení ve Swer:*

96 bytes

```
$n=ri(),k=2^16;
@(0..8)
{
    @(1..4)
    if (i % 2)
        pr("| ", n && (k /= 2) ? "# " : " ");
    el
        pr("+---");
    p(i % 2 ? "|" : "+");
}
```

*Řešení v C++:*

172 bytes

```
#include <iostream>
int n, i = 65536, j;

int main()
{
    std::cin >> n;

    for (; j < 9;)
    {
        for(int k:{0,0,0,0})
            if (j % 2)
                printf("| %s", n & (i /= 2) ? "# " : " ");
            else
                printf("+---");
            puts(j++ % 2 ? "|" : "+");
    }
}
```

## Gamma Function Golf

*Zadání:* Pro zadané  $t \in (-10^9, 13)$ , vypište  $\Gamma(t)$  – Gamma funkce – která je definovaná jako

$$\Gamma(t) = \int_0^{\infty} x^{t-1} e^{-x} dx.$$

Pro vyřešení této úlohy není možné použít žádný zabudovaný algoritmus, který počítá Gamma funkci (např. z matematické standardní knihovny). Taktéž není možné použít žádný mechanismus, který umí počítat integrál. Výstup musí být přesný na alespoň šest desetinných míst. Můžete předpokládat, že pro vstup  $t$  je  $\Gamma(t)$  definované, tedy  $t$  je buď kladné, nebo záporné, ale ne celé.

*Poznámka:* Pro řešení v obou jazycích je použita Eulerova definice Gamma funkce:

$$\Gamma(t) = \frac{1}{t} \prod_{n=1}^{\infty} \frac{(1 + \frac{1}{n})^t}{1 + \frac{1}{n}}$$

*Poznámka:* Výpočet funkce chvíli trvá (v řádu desítek sekund), protože iteruje přes množinu  $\{1, 2, \dots, 10^9\}$ . Nicméně vrací validní řešení, takže je validním řešením této úlohy.

*Řešení ve Swer:*

49 bytes

```
t=>{
    $v=1.;
    @{1..1^9}
        v *= (1+1./i)^t / (1 + t/i);
    -> v / t;
};
```

*Řešení v C++:*

81 bytes

```
[](auto t)
{
    auto v = 1.;
    for (int x = 1; x < 1e9; ++x)
        v *= pow(1 + 1. / x, t) / (1 + t / x);
    return v / t;
};
```



# Závěr

Jazyk Swer poskytuje svým uživatelům nástroje pro řešení úloh z oblasti code-golf. V kapitole 2 jsme se seznámili se specifikací jazyka a popsali jsme sémantiku konstruktů, které jazyk Swer nabízí. Poté následovaly dvě technické kapitoly popisující implementaci překladače, včetně řešení problémů ve specifikaci. V kapitole 5 jsme si představili standardní knihovnu jazyka Swer a poukázali jsme na některé její pěkné i nešikovné vlastnosti. Poslední kapitola 6 se zabývala ukázkami kódu v jazyce Swer a jeho srovnání s jazykem C++. Z této kapitoly bylo vidět, že jazyk Swer má zápis výrazně kratší, než ekvivalentní program v C++. I přesto, že zápis je o dost kratší, tak si nemyslím, že by došlo ke snížení čitelnosti. Pokud uživatel ovládne standardní knihovnu jazyka Swer a zapamatuje si drobné rozdíly mezi Swer a C++ (pokud zná C++), tak může velmi rychle zapadnout mezi řešitele code-golf úloh. Šance, že by nějakou úlohu vyhrál je sice pořád velmi malá, díky jazykům Pyth nebo CJam, jejichž zápis je velmi komprimovaný, a pokud člověk nezná daný jazyk, tak i velmi nečitelný a nepochopitelný. Následující tabulka ukazuje srovnání mezi některými jazyky. Pokud ve sloupci chybí počet bytů, tak daná úloha nebyla v daném jazyce řešena.

Úloha	Swer	C++	Pyth	CJam	Java	C#
Cat goes "Meow"	48	143	-	24	-	-
String Zip and Sort	51	152	5	5	-	-
Fibo Nacci!	112	185	37	44	308	320
Back to the chemistry class	90.9	144	40	44	181.8	-
16-bit binary grid	96	172	37	41	-	227
Gamma Function Golf	49	81	21	-	-	-

Kód překladače a standardní knihovny je přenositelný, díky tomu se jazyk Swer neomezuje jen na jednu platformu. Testování překladače probíhalo na platformách Windows (Microsoft Visual Studio 2015, MinGW), Unix (Debian, Ubuntu). Implementace překladače bohužel neobsahuje žádné testy, které by ověřovaly, zda v implementaci není chyba. Jedním z možných způsobů testování aplikace by bylo porovnávat výstup překladače Swer – tedy kód v C++ – s očekávaným zdrojovým kódem C++. To má ale několik problémů. Očekávaný výstup by musel přesně odpovídat výstupu překladače i přesto, že například některé bílé znaky nejsou nutné a Swer je generuje jen pro lepší pochopení vygenerovaného kódu. Dal by se napsat testovací framework, který by dělal lexikální analýzu výstupu z překladače Swer do nějaké lépe zpracovatelné formy (například sekvence tokenů), která by se porovnávala s očekávaným výstupem.

Jinou možností testování by bylo testovat jednotlivé metody, či dokonce třídy překladače, a simulovat tak parsování zdrojového kódu. Pokrytí kódu těmito testy by bylo ale velmi malé, protože možných vstupů zdrojového kódu jazyka Swer je nezměrné množství a vyžadovalo by to implementaci testovacího frameworku (nebo nastudování a použití existujícího pro C++).

Na mou práci by mohla navazovat implementace pluginu do nějakého textového editoru pro zjednodušení psaní kódu v jazyce Swer. Nebo dokonce implementace vývojového prostředí pro jazyk Swer obsahující lexikální, či dokonce

syntaktický highlighting, doplňování názvů funkcí či integrace s dokumentací. Vývojový nástroj by také mohl obsahovat funkcionalitu, která by převedla kód z jazyka Swer do tzv. „golfed“ tvaru, který je co nejvíce komprimovaný (odstraněny redundantní bílé znaky, zkráceny identifikátory, apod.).

*Příklad.* Porovnání přehledného a „golfed“ kódu.

```
$ funkce(I parametr)
{
    $ prm = 7;
    -> prm + 1;
}
```

```
$f(I p){$o=7;->o+1;}
```

Další z možných návazností na mou práci by bylo vylepšení překladače o lepší hlášení chyb jak syntaktických tak sémantických (použití nedefinované funkce, nekompatibilita typů, apod.), které momentálně překladač Swer neřeší a hlásí je až následné volání překladače C++.

Ačkoliv standardní knihovna jazyka Swer obsahuje základní mechanismy pro práci s poli, textovými řetězci i pokročilejším kontejnerem jako mapa, či funkce z oblasti funkcionálního programování, stále mu chybí některé mechanismy, které by umožňovaly jednodušší zápis některých algoritmů. Mezi takové vlastnosti patří například knihovna regulárních výrazů, knihovna pro práci s časem a datem, či případně práce se soubory a souborovým systémem.

I přes tyto překážky je možné jazyk Swer použít v oblasti code-golf bez větších obtíží. Zadání práce splňuje a řešení úloh v jazyce Swer je, dle mého názoru, pro začátečníky v oblasti code-golf přehledné a snadno pochopitelné.

# Seznam použité literatury

- [1] Golfscript, cjam, or pyth? <http://codegolf.stackexchange.com/questions/54385/golfscript-cjam-or-pyth>. [2016.06.27].
- [2] Tiobe index for july 2016. [http://www.tiobe.com/tiobe\\_index](http://www.tiobe.com/tiobe_index). [2016.07.07].
- [3] The concepts of bison. [http://dinosaur.compilertools.net/bison/bison\\_4.html](http://dinosaur.compilertools.net/bison/bison_4.html). [2016.07.11].
- [4] International Organization for Standardization, <http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4594.pdf>. *Working Draft, Standard for Programming Language C++*. [2016.05.30].

# Přílohy

## A Uživatelská dokumentace

### A.1 Překlad projektu Swer

Pro správný překlad je vyžadována přítomnost následujících nástrojů:

Flex (generátor tokenizeru)

Bison (generátor parseru)

Překladač C++ podporující standard C++11

### Platforma Windows

Pomocí vývojového nástroje *Microsoft Visual Studio 2015* otevřeme soubor `Swer/Swer.sln`. V nástroji *Visual Studio* změníme cestu pro nástroje Flex a Bison v *Custom build tool* pro soubory `src/swer.l` a `swer.y` (V *Solution Explorer* → *Properties*). Alternativní možností je přidat složky s nástroji do systémové proměnné `$PATH`.

Následně pomocí tlačítka *Build* → *Build Solution* sestavíme projekt. Spustitelný soubor `Swer.exe` se nachází ve složce `bin/[x86|x64]/[Debug|Release]` dle zvolené konfigurace.

### Platforma UNIX

Překlad na platformě UNIX vyžaduje navíc přítomnost GNU nástrojů (*make*, *sed*). V kořenovém adresáři s projektem Swer spustíme příkaz

```
$ make
```

Ten vytvoří v adresáři `Swer` spustitelný soubor `swer`.

### A.2 Použití jazyka Swer

Sestavený program Swer je řádkový program, který přijímá vstupní data z argumentů příkazové řádky. Pomocí příkazu

```
> swer -h
```

můžeme získat základní informace o ovládání programu. Ty obsahují informace o programu a volby, které je možné specifikovat z příkazové řádky. Možnosti jsou následující

Volba	Ekvivalent	Parametry	Popis
<code>-h</code>	<code>--help</code>	žádné	Zobrazí nápovědu.
<code>-c</code>	<code>--compiler</code>	<code>[gcc, clang, cl]</code>	Přesměruje výstup do překladače C++.
<code>-C</code>	<code>--command</code>	<i>příkaz</i>	Přesměruje výstup do volání <i>příkaz</i> .
<code>-o</code>	<code>--output</code>	<i>soubor</i>	Přesměruje výstup do <i>soubor</i> .

Volby `-c` a `-C` nesmí být použity najednou. V případě kombinace volby `-o` a `-c` je sémantika volby `-o` správně přenesena do volání konkrétního překladače.

Při použití volby `-c cl` je vytvořen soubor s náhodným (neobsazeným) názvem, do kterého je generován výstup překladače Swer. Nad tímto souborem je pak volám překladač *Microsoft C++ Compiler*.

Následující tabulka ukazuje příkazy použité při přímém volání překladače:

gcc nebo g++	<code>g++ -std=c++14 -x c++ -</code>
clang nebo clang++	<code>clang++ -std=c++14 -x c++ -</code>
cl	<code>cl /EHsc /TP /Fea.exe input_file</code>

`input_file` je jméno souboru vygenerovaného v mezikroku. Výstup z překladače Swer je třeba směřovat do souboru (ze kterého pak překladač `cl` čte z důvodu, že `cl` neumí číst zdrojový kód ze standardního vstupu).

Použití příkazu `swer file_name` přeloží soubor `file_name` ve zdrojovém kódu jazyka Swer do zdrojového kódu C++, který je implicitně vypsán na standardní výstup.

Jazyk Swer je popsán v kapitolách 2 a 5, a ukázky programů v tomto jazyce pak obsahuje kapitola 6

Pro správné použití standardní knihovny je nutné, aby překladač C++ hledal připojované soubory (`#include`) také ve složce `SwerSTL/src`. Ta obsahuje soubor `swer_stl`, který sdružuje připojování všech částí standardní knihovny.

## B Překladač Swer

Swer (kořenový adresář)

- └ Swer (překladač Swer)
  - └ lib/ (dodatečné knihovny pro Swer)
  - └ src/ (zdrojový kód Swer)
  - └ Makefile (Makefile pro UNIX)
  - └ Swer.vcxproj (Visual studio projekt pro Windows)
- └ SwerSTL (standardní knihovna Swer)
  - └ src/ (zdrojový kód standardní knihovny)
  - └ Makefile (Makefile pro UNIX)
  - └ SwerSTL.vcxproj (Visual studio projekt pro Windows)
- └ Examples/ (příklady, ukázky kódu)
- └ Swer.sln (Solution soubor pro Visual studio)