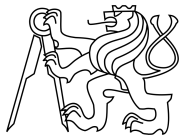
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Integrace syntaxe založené na jazyku Python pro Clang/LLVM překladač jazyka C++
Student:	Jan Červený
Vedoucí:	Ing. Radomír Polách
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Seznamte se s překladačem LLVM se zaměřením na přední část překladače.
Analyzujte a navrhněte zjednodušený zápis syntaktických konstrukcí jazyka C++ inspirovaný jazykem Python na základě ukázek dodaných vedoucím práce, například přidání vytváření bloku pomocí odsazování, odstranění blokových a podmínkových závorek, zjednodušení zápisu šablon a jiné, tak, aby vznikl jazyku Python podobný zápis, který bude čitelnější a kratší než standardní syntaxe.
Navržené rozšíření jazyka C++ implementujte do překladače Clang/LLVM.
Překladač s implementovaným rozšířením otestujte na vhodném souboru jednoduchých programů.
Rozšíření překladače a novou rozšířenou syntaxi a její sémantiku důkladně zdokumentujte a subjektivně zhodnoťte její vlastnosti.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 8. ledna 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Integrace syntaxe založené na jazyku Python pro Clang/LLVM překladač jazyka C++

Jan Červený

Vedoucí práce: Ing. Radomír Polách

9. ledna 2017

Poděkování

Na tomto místě bych chtěl poděkovat panu Ing. Radomíru Poláchovi za pomoc při tvorbě této práce. Dále bych chtěl poděkovat své rodině za podporu při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. ledna 2017 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Červený, Jan. *Integrace syntaxe založené na jazyku Python pro Clang/LLVM překladač jazyka C++*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato práce se zabývá návrhem a implementací rozšíření jazyka C++ pro překladač Clang/LLVM. Cílem tohoto rozšíření je zjednodušení a zlepšení čitelnosti. Hlavní inspirací jsou jazyky Python a CoffeeScript. Obsahem práce je také zhodnocení nové syntaxe a její porovnání s původní.

Klíčová slova C++, Python, CoffeeScript, Clang/LLVM, návrh rozšíření syntaxe, implementace rozšíření syntaxe

Abstract

This thesis deals with the desing and implementation of C++ language extension for Clang/LLVM compiler. The purpose is to simplify C++ syntax and make it easier to read. The main inspirations are Python and CoffeeScript languages. It also contains evaluation of the extension and its comparison to the original syntax.

Keywords C++, Python, CoffeeScript, Clang/LLVM, syntax extension design, syntax extension implementation

Obsah

Seznam obrázků

Úvod

C++ patří mezi hojně využívané programovací jazyky. V jeho syntaxi se objevují nadbytečné prvky, kvůli kterým se stává kód hůře čitelným a zbytečně dlouhým. Tyto prvky je možné po vzoru jiných jazyků odstranit a získat tak kratší a lépe čitelný zápis. Dobrými příklady jsou Python a CoffeeScript. Inspirace na toto rozšíření pochází z článku dodaným vedoucím práce [1]. Motivací je ulehčení práce při psaní kódu a zlepšení následné čitelnosti.

Kapitola 1

Cíl práce

Cílem této práce je analýza překladače LLVM, zejména jeho přední části Clang. Dále návrh rozšíření syntaxe programovacího jazyka C++ o zjednodušený zápis inspirovaný jazyky Python a CoffeeScript. Výsledkem má být snáze zapisovatelná syntaxe, která je i lépe čitelná. Navržené rozšíření má být implementováno do již zmíněného překladače Clang/LLVM. Součástí této práce je i dokumentace nově vzniklé syntaxe a zhodnocení vlastností tohoto rozšíření.

1.1 Struktura práce

Práce je rozdělena následujícím způsobem.

Cílem rešeršní části bakalářské práce je základní popis překladače, seznámení se syntaxemi programovacích jazyků C++, Python a CoffeeScript a také s překladačem Clang/LLVM, do které bude rozšíření implementováno.

Druhou částí je popis návrhu samotného rozšíření syntaxe programovacího jazyka C++ inspirovaný zejména jazykem Python, ale také jazykem CoffeScript.

Třetí část se věnuje popisu implementace tohoto rozšíření do překladače Clang/LLVM.

V poslední části jsou zhodnoceny vlastnosti nového rozšíření syntaxe C++ a jeho porovnání se syntaxí původní.

Kapitola 2

Analýza

2.1 Překladač

Překladač je počítačový program, který transformuje zdrojový kód napsaný v programovacím jazyce do spustitelného programu.

Typický překladač je rozdělen na dvě části, přední a zadní. Přední část je závislá na vstupním jazyku a sestává z těchto částí: lexikální analyzátor, syntaktický analyzátor, sémantický analyzátor a generátor mezikódu. Zadní část je závislá na cílovém jazyku a tvoří ji generátor kódu. Mezi těmito prvky přední i zadní části mohou být vloženy optimalizace. [2]

Následuje popis částí, které byly upravovány v rámci této práce, tedy lexikálního a syntaktického analyzátoru.

2.1.1 Lexikální analyzátor

Lexikální analyzátor čte vstupní program a jeho výstupem je posloupnost lexikálních elementů tzv. tokenů. Tokeny jsou terminálními symboly gramatiky, která popisuje syntaxi překládaného jazyka. Lexikální analyzátor je realizován konečným automatem (jazyk lexikálních tokenů lze popsat regulární gramatikou). Token v sobě nese, kromě svého typu, také například textovou podobu identifikátoru nebo hodnotu číselnou. Tokeny jsou z lexikálního analyzátoru předávány do analyzátoru syntaktického. [3]

V Clang/LLVM token obsahuje navíc například ještě svou přesnou polohu ve zdrojovém kódu, díky čemuž může být při chybovém hlášení přesně označeno místo chyby.

2.1.2 Syntaktický analyzátor

Syntaktický analyzátor (dále SA) zpracovává tokeny dodané lexikálním analyzátorem a vytváří abstraktní syntaktický strom. Clang používá pro analýzu tzv. metodu rekurzivního sestupu [4].

2.2 LLVM

LLVM bylo původně zkratkou pro Low Level Virtual Machine [5]. V současné době se ale jedná o kolekci modulárního a znovu použitelného překladače a dalších pomocných technologií.

Ke zveřejnění první verze 1.0 došlo v říjnu 2003 [6].

V následující části je čerpáno z webu LLVM [7]. Na počátcích se jednalo o výzkumný projekt University of Illinois, který si kladl za cíl přinést moderní, statický i dynamický překlad využívající SSA. Z původního projektu se stal projekt zastřešující mnoho podprojektů. Ty jsou využívány v dalších komerčních a open source projektech. Jedná se například o společnosti Apple, Intel nebo SONY [8]. LLVM je také využíváno v akademickém výzkumu.

Mezi hlavní podprojekty patří:

- LLVM Core, což je zadní část překladače. Kromě generování kódu pro mnoho platforem přináší i optimalizaci. Využívá LLVM mezikód (LLVM IR). Tuto část je možné využít při tvorbě překladače pro nový jazyk, případně pro jeho port.
- Clang viz 2.3
- dragonegg, který integruje optimalizátor a generátor kódu LLVM do SA GCC. Díky tomu může LLVM překládat Ada, Fortran a další jazyky podporované přední částí GCC.
- LLDB je debugger postavený jak na knihovnách samotného LLVM, tak i Clangu. Využívá Clang AST a SA výrazů, z LLVM například JIT a disassembler. Mezi jeho výhody patří velká rychlost.
- Projekt lld, který má za cíl stát se vestavěným linkerem pro Clang/LLVM. V současné době používá Clang pro vytvoření spustitelných souborů systémový linker.

2.3 Clang

Clang tvoří přední část překladače LLVM pro jazyky C, C++, Objective C a Objective C++. Důraz je kladen na jeho rychlost, jednoduchost, škálovatelnost a nízkou paměťovou náročnost.

Jednotlivé jeho části jsou rozděleny do knihoven, které mohou být použity pro různé účely. Toto dělení pomáhá i nově příchozím vývojářům, kterým pak stačí porozumět pouze konkrétní části. Mezi tyto knihovny patří například *liblex* pro lexikální analýzu a preprocesor, *libparse* pro syntaktickou analýzu, *libast*, který obsahuje třídy pro reprezentaci abstraktním syntaktickým stromem (AST), nebo samotnou knihovnou *clang*, která se stará o řízení překladu. Díky tomuto rozdělení může být Clang použit k vytvoření extrémně rychlého

generátoru kódu, který vůbec nevytváří AST nebo naopak může být využit pouze k vytvoření AST a následnou práci s ním.

Kvůli jednodušší správě Clang používá jednotný parser pro všechny zmíněné jazyky z rodiny C. Je ručně psaný a využívá metodu rekurzivního sestupu. Implementován je v C++ a díky zmíněnému rekurzivnímu sestupu je kód relativně snadno pochopitelný a umožňuje jednoduchou implementaci diagnostiky a zotavování po chybě. [4]

2.3.1 Porovnání s GCG

Srovnání výhod a nevýhod s GCC [9].

Výhody Clangu:

- Byl od začátku navržen pro jasnou a obsáhlou diagnostiku varování a chyb.
- Mezi jeho výhodu oproti GCC patří systém varovných a chybových hlášek, který kromě samotného popisu chyby zobrazuje i přesné místo ve zdrojovém kódu, kde k chybě došlo.

```
$ gcc-4.2 -fsyntax-only t.c
t.c:7: error: invalid operands to binary + (have 'int'
       and 'struct A')

$ clang -fsyntax-only t.c
t.c:7:39: error: invalid operands to binary expression
                 ('int' and 'struct A')
  return y + func(y ? ((SomeA.X + 40) + SomeA) / 42
                       ~~~~~~~~~~~~~~ ^ ~~~~~
  + SomeA.X : SomeA.X);
```

- AST Clangu je snadno pochopitelné.
- Clang může být díky své architektuře využit pro různé nástroje při analýze kódu, refaktoringu nebo třeba v integrovaných vývojových prostředích. Oproti tomu je monolitické GCC jen obtížně znovupoužitelné.
- Clang implicitně nezjednodušuje kód při parsování jako to dělá GCC, což je dobré při jeho analýze.
- Clang může uložit serializovaný AST na disk a poté ho načíst do jiného programu.
- Clang je rychlejší a potřebuje méně paměti.
- Clang může díky použití LLVM jako své zadní části používat jeho funkce, například Just-In-Time kompilaci.

- Clang podporuje mnoho jazykových rozšíření, z nichž ne všechny jsou implementovány v GCC, například atributy pro kontrolu bezpečné práce s vlákny.

Výhody GCC:

- Podporuje jazyky, které Clang ne. Jedná se například o jazyky Java, FORTRAN, Go.
- GCC podporuje více platforem než LLVM.
- Ne všechna jazyková rozšíření, které podporuje GCC, podporuje i Clang, například vnořené funkce v C.

2.4 C++

C++ je objektově orientovaný programovací jazyk navržen Bjarne Stroustrupem s úmyslem zkombinovat objektově orientovaný jazyk Simula s efektivním jazykem C používaným pro systémové programování. První verze C++, ještě pojmenovaná *C with Classes*, byla poprvé použita v roce 1980. Samotná podpora pro objektové programování byla přidána v roce 1983. Podpora šablon byla přidána mezi lety 1987 až 1989. Jazyk se začal šířit a vzniklo několik na sobě nezávislých implementací. Proto se roku 1990 začalo pod záštitou American National Standards Institute (ANSI) a později International Standards Organization (ISO) pracovat na standardizaci. Tato snaha vyústila roku 1998 v mezinárodní standard známý jako C++98. [10]

Dodnes prodělalo C++ mnoho změn, přibyla například možnost automatického zjišťování návratového typu funkce, lambda výrazy nebo klíčové slovo final. Poslední dokončenou verzí standardu je C++14 a pracuje se na verzi C++17. [11]

2.4.1 Ukázka zdrojového kódu

```
#include <iostream>
using namespace std;

class Obdelnik{

  public:
  Obdelnik(double s, double v)
    : sirka(s), vyska(v) {}

  double obsah(){ return sirka * vyska;}
  double obvod(){ return 2*(sirka+vyska);}

  private:
  double sirka, vyska;

};


int main(){

  Obdelnik a(5.1,7.9);
  cout << "Obsah obdelniku je " << a.obsah();
  cout << '\n';
  cout << "Obvod obdelniku je " << a.obvod();
  cout << '\n';

  int d = 1;

  for(int i = 1; i < 10; i++){
    cout << d << '\n';
    d *= 2;
  }

  return 0;
}
```

Výstup programu:

```
Obsah obdelniku je 40.29
Obvod obdelniku je 26
1
2
4
8
16
32
64
128
256
```

2.5 Python

Python je interpretovaný objektově orientovaný programovací jazyk. Jeho jednoduchá, lehce naučitelná syntaxe je dobře čitelná. Podporuje moduly a balíčky a tedy i modularitu programu a znovupoužitelnost kódu. [12]

Během svého vývoje prodělal mnoho změn. První verzi, označenou jako 0.9.0 publikoval Guido Van Rossum v únoru 1991. V lednu 1994 následovala verze 1.0. V říjnu 2001 byla představena verze 2.0. Aktuální verzí je Python 3, který není zpětně kompatibilní s verzí 2. Vývoj Pythonu se řídí 19 hlavními pravidly, známými jako *The Zen of Python*. Mezi tyto pravidla patří například *Čitelnost se počítá* nebo *Hezký je lepší než ošklivý*. [13]

2.5.1 Syntaxe

Hlavní zjednodušení oproti C++ je v blocích (složených příkazech), které jsou tvořeny pomocí odsazování stejné šířky namísto složených závorek. Dále odpadá nutnost psát závorky okolo podmínek u *if*, *while*, *for* a dalších příkazů. Také není nutné psát středník na konci příkazu (pokud jej od dalšího odděluje nový řádek). Ukázka syntaxe použité při návrhu rozšíření C++ v BNF [14]:

```
suite         ::=  stmt_list NEWLINE
                 | NEWLINE INDENT statement+ DEDENT
statement     ::=  stmt_list NEWLINE | compound_stmt
stmt_list     ::=  simple_stmt (";" simple_stmt)* [";"]

target_list     ::=  target ("," target)* [","]
target          ::=  identifier
                 | "(" target_list ")"
                 | "[" target_list "]"
```

```
classdef    ::=  [decorat.] "class" classname [inher.] ":" suite
inheritance ::=  "(" [parameter_list] ")"
classname   ::=  identifier

if_stmt ::=  "if" expression ":" suite
             ( "elif" expression ":" suite )*
             ["else" ":" suite]


while_stmt ::=  "while" expression ":" suite
                ["else" ":" suite]

for_stmt ::=  "for" target_list "in" expression_list ":" suite
              ["else" ":" suite]
```

2.5.2 Ukázka zdrojového kódu

Ukázka jednoduchého programu napsaného v jazyce Python.

```
class Obdelnik:

    def __init__(self, sirka, vyska):
        self.sirka = sirka
        self.vyska = vyska

    def obsah(self):
        return self.sirka * self.vyska

    def obvod(self):
        return 2*(self.sirka + self.vyska)


a = Obdelnik(5.1,7.9)
print("Obsah obdelniku je {}".format(a.obsah()))
print("Obvod obdelniku je {}".format(a.obvod()))

for i in range(9): // mocniny dvojky
    print(pow(2,i))
```

Výstup programu:

```
Obsah obdelniku je 40.29
Obvod obdelniku je 26
1
2
4
8
16
32
64
128
256
```

2.6 CoffeeScript

CoffeeScript je jazyk, který se jedna ku jedné překládá do JavaScriptu, zjednodušuje zápis a zlepšuje jeho čitelnost. Díky tomuto překladu je možné ho využít s již existujícími knihovnami pro JavaScript. Přeložený výstup je dobře čitelný a funguje v každém běhovém prostředí JavaScriptu.

Pro potřeby nové syntaxe z něj byl přejat jednoduchý zápis tříd, kdy za samotným jménem a případnou dědičností následuje nový řádek.

```
class Snake extends Animal
    move: ->
    alert "Slithering..."
    super 5
```

CoffeeScript u funkcí automaticky vrací poslední příkaz, což bylo také využito v této práci.

```
changeNumbers = ->
    inner = -1
    outer = 10
```

Výsledek po přeložení do JavaScriptu [15]:

```
changeNumbers = function() {
    var inner;
    inner = -1;
    return outer = 10;
};
```

Kapitola 3

Návrh

Není zde kladen důraz na popis celé nové rozšířené syntaxe, ale spíše na rozdíly mezi původní syntaxí. Nejsou zde tedy rozepisovány detaily na místech, kde je syntaxe shodná. Pro popis je použita BNF podle [16]. Navíc je přidán symbol „+“ pro více než jedno opakování.

3.1 Způsob použití

Rozšířenou syntaxi je možné použít s již existujícím kódem v C++. Analyzátor se přepíná na novou syntaxi pomocí direktivy preprocesoru *#pragma PPlusPlus ON*. Je třeba ji uvádět po direktivách preprocesoru *#include* a její platnost trvá do konce souboru.

Pro oddělení příkazů není třeba používat středník. Může však být nadále použit pro oddělení více příkazů na jednom řádku.

Pro odsazení může být použit libovolný počet mezer a tabulátorů větší nebo roven jedné. Blok (složený příkaz) vzniká při zvětšení odsazení oproti předchozímu neprázdnému řádku a končí při použití menšího odsazení. Je tedy nutné zachovat počet mezer (případně tabulátorů) stejný pro celý zamýšlený blok.

3.2 Použité symboly

```
<deklarace_příkaz> ::= <deklarace>
                     | <příkaz>
                     | <prázdný_příkaz>

<složený_příkaz>
::= { "\n" <odsazení> <deklarace_příkaz> }+
```

3.3 Třídy

U tříd byla nejprve zkoušena syntaxe, kdy na konci řádku s jménem třídy a případnou dědičností byl znak „>“ a později „->“, jako tomu je v CoffeeScriptu. Nakonec se ukázalo jako nejlepší řešení úplné vypuštění těchto znaků, které zkrátilo zápis a zlepšilo jeho čitelnost.

Pro struktury platí to samé. Syntaxe dědičnosti zůstala zachována.

```
<třída> ::= class <jméno_třídy> [<dědičnost>]
            { "\n" <odsazení> <specifikace_členů> }+

<struktura> ::= struct <jméno_třídy> [<dědičnost>]
                { "\n" <odsazení> <specifikace_členů> }+
```

3.4 Funkce

U funkcí (metod) byly, stejně jako u tříd/struktur, zkoušeny znaky „>“ a „->“ za uzavírací závorkou po parametrech, nakonec jsou stejně tak vypuštěny.

```
<funkce> ::=
      <návratový_typ> <jméno> "(" <parametry> ")" <tělo_funkce>

<tělo_funkce> ::=  <složený_příkaz> | <try_catch>
```

3.4.1 Autoreturn

Bylo přidáno automatické vracení posledního příkazu u funkcí a metod bez nutnosti uvedení klíčového slova *return* po vzoru CoffeeScriptu. Stane se tak, pokud má funkce návratový typ jiný než *void* a pokud je poslední příkaz zároveň výraz (třída *Expr* dědí z třídy *Stmt*), který má smysl vracet (není typu *void*). To platí například pro číselnou konstantu, identifikátor nebo volání funkce. Klasický *return* je možné používat i nadále.

3.5 Cykly

Cykly *for*, *while* a *do while* mají odstraněny závorky okolo podmínek. Naopak pro zlepšení čitelnosti byla přidána dvojtečka za podmínku.

3.5.1 For

Došlo k odstranění závorky za klíčovým slovem *for*, mezi ním a dvojtečkou dochází k vypnutí lexikální analýzy a je proto možné tuto část rozdělit na více řádků.

```
<for_cyklus> ::=
    for [<inicializace_proměnné>] ";" [<podmínka>] ";" <výraz> ":"
    <složený_příkaz>

<for_cyklus> ::= for auto <jméno> ":" <proměnná> ":"
                 <složený_příkaz>
```

3.5.2 While

Došlo zde k odstranění závorek okolo podmínky, podmínka může být rozdělena na více řádků, odřádkování v ní, stejně jako u *for* cyklu, nehraje roli.

```
<while_cyklus> ::= while <podmínka> ":"
                   <složený_příkaz>
```

3.5.3 Do while

I zde dochází k odstranění závorky okolo podmínky a i zde v ní dochází dočasnému vypnutí lexikální analýzy pomocí odsazování. Klíčové slovo *while* musí mít stejné odsazení jako *do*.

```
<do/while_cyklus> ::= do ":"
                      <složený_příkaz> "\n"
                      while <podmínka>
```

3.6 Switch

Zde došlo také k odstranění výrazu okolo závorky. Pro rozlišení jednotlivých případů (case) je nutné zvětšit odsazení.

```
<switch> ::= switch <výraz> ":"
             <switch_tělo>

<switch_tělo> ::=
    { "\n" <odsazení> <case_deklarace_příkaz> }+

<case_deklarace_příkaz> ::=  <deklarace_příkaz> | <case>

<case> ::= case <konstantní_výraz> ":"
           <složený_příkaz>
```

3.7 If, else if, else

U *if* a *else if* došlo k odstranění závorky okolo podmínek, v nich je vypnuta lexikální analýza pomocí odsazení a lze tedy delší podmínky rozdělit na více řádků pro zlepšení čitelnosti.

```
<if> ::= if <podmínka> ":"
         <složený_příkaz>

        { "\n"
          else if <podmínka> ":"
          <složený_příkaz> }+

        [ "\n"
          else ":"
          <složený_příkaz> ]
```

3.8 Namespace

Syntaxe *namespace* má pouze odstraněné složené závorky v důsedku použití odsazení, již ji není možné více zjednodušit.

```
<namespace> ::= namespace <jméno>
                <složený_příkaz>
```

3.9 Enum

U výčtů je čárka za jednotlivými položkami nahrazena odřádkováním a stejným odsazením.

```
<enum> ::= enum [<jméno>]
           <enum_tělo>

<enum_tělo> ::= { "\n" <odsazení> <enum_položka> }+

<enum_položka> ::=
       <identifikátor> | <identifikátor> "=" <konstanta>
```

3.10 Try catch

Za *try* byla přidána dvojtečka, po *catch* je vynechána závorka. Inicializační seznam je možné použít pouze u konstruktorů ve *function-try* bloku.

```
<try_catch> ::= try ":" [<inicializační_seznam>]
                <složený_příkaz> "\n"
                 { catch <deklarace_proměnné_elipsa> ":"
                   <složený_příkaz> }+

<deklarace_proměnné_elipsa> ::= <deklarace_proměnné> | "..."
```

3.11 Šablony

U šablon došlo k nahrazení závorek „<“ a „>“ za odsazení. Dále byl přidán zjednodušený zápis pro *typenamy* a šablony, které mají jako parametr další šablonu. U deklarace (třídy nebo funkce) následující po samotné šabloně jsou závorky „<“/„>“ použity normálně jako u původní syntaxe (například pro částečnou specializaci).

Pro úplnou specializaci šablon bylo přidáno nové klíčové slovo „spec“, které nahrazuje „<>“. Jinak je jejich syntaxe stejná. Ufvedené změny se samozřejmě dotkly i částečně specializace, ta ale jinak zůstává také stejná.

```
<template> ::= template <temp_parametry> "\n"
               <deklarace>

<temp_parametry ::= { "\n" <odsazení> <temp_parametry> }+

<temp_parametry> ::= <deklarace>
                   | <typename_zkratka>
                   | <temp_temp_zkratka>
                   | spec
```

Namísto vypisování jednotlivých *typenamů* je možné použít zkrácený zápis, např. *typename T1, typename T2* je možné zapsat jako *typename: T1, T2*.

```
<typename_zkratka> ::= typename ":" <identifikátory>

<identifikátory> ::= <identifikátor>
                   | <identifikátory> "," <identifikátor>
```

Pokud šablona přijímá jako parametr šablonu, která má parametry pouze *typenamy*, lze využít zkrácený zápis, kdy je výčet *typenamů* nahrazen jejich počtem.

Tedy místo *template<typename,typename> Trida* lze psát *template 2 Trida*.

```
<temp_temp_zkratka> ::= template <číslo> <název_třídy>
```

3.12 Prázdný příkaz

Bylo přidáno nové klíčové slovo *pass*, které slouží jako prázdný příkaz na místo, kde by v původní syntaxi bylo „{}“ (to je nutné, aby se měl pro lexikální analyzátor o co „opřít“ a vrátil správně závorky).

```
<prázdný_příkaz> ::= pass
```

3.13 Nepoužitý návrh

Myšlenkou bylo vytvořit automatické *gettery* a *settery*, které měly fungovat tak, že za novým klíčovým slovem *getters* (případně *setters*) by následoval v závorce výčet atributů, ke kterým se mají automaticky vygenerovat jednoduché metody vracející/nastavující tento atribut. Po neúspěšných pokusech bylo s přihlédnutím k tomu, že takovéto metody nemají moc velký význam (kromě zachování konzistence v projektech, kde se již používají), od tohoto nápadu upuštěno ve prospěch jiných změn.

Kapitola 4

Implementace

Implementace byla provedena do Clang/LLVM verze 3.8.1. Upraveny byly části zabývající se lexikální a syntaktickou analýzou (třídy *Lex* a *Parse*) a definice tokenů.

4.1 Změna způsobu syntaktické analýzy

Bylo nutné vymyslet způsob, jak dát vědět syntaktickému analyzátoru, že zpracovává novou rozšířenou syntaxi a ne původní. Prvotní myšlenkou bylo přepínání na úrovni souboru podle jeho koncovky. To se ale ukázalo jako těžko providitelné. Na koncovku souboru je totiž navázáno mnoho nastavení a když už to vypadalo, že je vše správně upravené, objevil se problém například s *#include* direktivou preprocesoru, kdy přestala správně fungovat. Tato cesta tedy byla opuštěna.

Souběžně s tímto nápadem bylo zkoušeno přepínání na úrovni deklarace třídy, funkce, případně při analýze podmínky u příkazů *if*, *while*, *for* apod. K rozpoznání nové syntaxe došlo při nenalezení otevírací závorky. To nebylo považováno za chybu, ale byla zavolána metoda SA, která ho nastavila na rozšířenou syntaxi. U tříd (to samé platí pro struktury, jediný rozdíl při analýze je zde v implicitním nastavení viditelnosti členských proměnných a metod z *private* na *public*) a funkcí (případně metod tříd) to bylo použití „>“ a později „->“ před samotným tělem třídy/funkce, tedy tam, kde by měla být složená závorka. Při nalezení tokenu s tímto znakem se zavolala metoda SA, která přepnula způsob analýzy, tento token se zahodil a přečetl se další, již podle nových pravidel. To sice dovolovalo použití rozšířené syntaxe uvnitř standardní, ale toto míchání bylo zbytečně zmatečné a nebylo možné použít novou syntaxi na nejvyšší úrovni souboru, pouze uvnitř zmíněných příkazů, funkcí a tříd.

Další zkoušenou možností bylo přepínání po přečtení slova *pplusplus*, po jeho nalezení na nejvyšší úrovni souboru se volala metoda SA, která nastavovala analýzu aktuálního souboru na novou syntaxi a načetla nový token.

Toto klíčové slovo muselo následovat po direktivách *#include* a jeho platnost trvala až do konce souboru. Výhodou byla možnost využití nové syntaxe již na nejvyšší úrovni souboru, například při deklaraci globálních proměnných. Také to bylo výhodnější pro práci se zabudovanou diagnostikou chyb a varování Clangu. Nové klíčové slovo bylo vytvořeno přidáním makra definující toto slovo do souboru s definicí tokenů[1]. Dále byla přidána podmínka do metody *Parser::ParseTopLevelDecl*[2], pokud bylo toto klíčové slovo nalezeno, analyzátor byl přepnut do režimu rozšířené syntaxe.

Nakonec byl zvolen standardně používaný způsob nastavení překladače pomocí direktivy preprocesoru *pragma*. Při implementaci bylo vycházeno z článku *Implement Pragmas in Clang/LLVM* [17].

Byla přidána struktura *NewSyntaxPragmaHandler* (dědí z třídy *PragmaHandler*) starající se o obsluhu nové pragma direktivy[3]. Dále byla v metodě *Parser::initializePragmaHandlers* tato obsluha (*handler*) přidána do preprocesoru. Obdobně bylo v metodě *Parser::resetPragmaHandlers* přidáno odstranění z preprocesoru. Samotná obsluha je řešena zastíněním metody *HandlePragma*. Zde je kontrolováno, zda nejsou na řádku před znakem „#“ mezery (to je řešeno přes preprocesor, kdy je v metodě *Lexer::SkipWhitespace*, do které bylo přidána implementace odsazování, přidán příznak preprocesoru, ten je pak odnastaven v *Preprocessor::DiscardUntilEndOfDirective*[4]). Pokud byly mezery nalezeny, je vrácena chyba. Dále je čten další token, kterým by měl být identifikátor (*ON* nebo *OFF*) a podle něj je nastaven preprocesor, případně je vypsána chyba (*err_pragma*, přidáno do diagnostiky[5]).

Výsledek této implementace je hodně podobný použití klíčového slova *pplusplus*, narozdíl od něj je ale možné v rámci souboru přepnout na původní syntaxi. Přepínání bylo zamýšleno na nejvyšší úrovni souboru (*top level deklarace*), ale nebylo nalezeno řešení, jak na úrovni lexikálního analyzátoru a preprocesoru při zapnutí syntaxe kontrolovat, zda na této nejvyšší úrovni jsme. Ohlídání tedy bylo ponecháno na programátorovi, při změně na novou syntaxi například uprostřed funkce může dojít k chybě překladu. Změna na původní syntaxi je hlídána (viz předchozí odstavec). Pragma direktiva musí být zarovnaná se zbytkem kódu, jinak dojde k chybě analýzy. Pokud je správně zarovnaná a má před sebou mezery, dojde k vrácení chyby z obshluhy *NewSyntaxPragmaHandler::HandlePragma*. Díky přepínání na nejvyšší úrovni není například nutné psát středník za deklaracemi funkcí, stačí pouze odřádkovat a zachovat nulové odsazení.

Omezením je nemožnost použití direktivy *#include* po změně na novou syntaxi (ani když je přepnuto zpět). Většinou jsou další soubory vkládány na začátek souboru. Nicméně se mohou najít případy, kdy by se hodilo vložit

[1] soubor `llvm/tools/clang/include/clang/Basic/TokenKinds.def`
[2] soubor `llvm/tools/clang/lib/Parse/Parser.cpp`
[3] soubor `llvm/tools/clang/lib/Parse/ParsePragma.cpp`
[4] soubor `llvm/tools/clang/lib/Lex/PPDirectives.cpp`
[5] soubor `llvm/tools/clang/include/clang/Basic/DiagnosticCommonKinds.td`

soubor i jinam, proto byla snaha tento problém řešit. Při nalezení direktivy *include* se vytváří nový objekt lexikálního analyzátoru, ale je použit stejný objekt preprocesoru. Tento LA vstupuje do vkládaného souboru, proto byly pokusy o uložení stavu preprocesoru, jeho resetování a opětovném vrácení do původního stavu při návratu z vkládaného souboru, bohužel se tento postup ukázal jako nefunkční.

4.2 Bloky pomocí odsazování

Při čtení znaku „\n“ (nový řádek) se volá metoda lexikálního analyzátoru *Lexer::SkipWhitespace*[6], která má za úkol rychlé odstranění bílých znaků při čtení souboru. Tato funkce se hodí pro použití při zjišťování odsazení. Původní funkce ve smyčce přeskakovala horizontální bílé znaky. Zde byla přidána proměnná, do které se ukládá jejich počet. Pokud již nejsou na vstupu další horizontální bílé znaky, kontroluje se, zda následuje vertikální (nový řádek). Pokud ano, přeskočí se, vynuluje se proměnná počítající mezery a tabulátory a opět se přeskakují horizontální bílé znaky. Pokud další znak již není nový řádek, dojde k vyskočení ze smyčky. V tuto chvíli je spočítaný počet mezer/tabulátorů na řádku, který neobsahuje pouze bílé znaky.

Pro zjišťování hloubky zanoření bloku byl do lexikálního analyzátoru přidán zásobník (*std::stack<int>*). Pokud je SA nastaven na rozšířenou syntaxi, je tento zásobník použit spolu s napočítanými mezerami/tabulátory následujícím způsobem.

```
pokud počet mezer je větší než nula
   pokud zásobník je prázdný nebo je jeho vrchol < odsazení
      přidej na vrchol počet mezer
      vrať token levá závorka

jinak pokud zásobník není prázdný a vrchol zásobníku > odsazení
   dokud zásobník není prázdný a vrchol zásobníku > odsazení
      přičti k~počtu zbývajících pravých závorek 1
      sniž vrchol zásobníku o~jedna
   vrať token "oddělovač"

jinak
   vrať token "oddělovač"
```

Levá závorka je nastavena podle potřeby, může to být „{“ (pro složené bloky) nebo „<“ (pro šablony), oddělovač může být středník (za příkazem), čárka (v *enumu*) nebo nový token *template_new_param* (v šablonách). Kvůli

[6] soubor `llvm/tools/clang/lib/Lex/Lexer.cpp`

tomu byly přidány metody do třídy *Preprocessor*[7], které mění nastavení a vracejí nastavené znaky/tokeny.

Tedy platí, že pokud se odsazení zvětšilo, lexikální analyzátor vrátí znak levou závorku, pokud je odsazení stejné, vrátí oddělovač. Pokud je odsazení menší, spočítá, kolik bloků je třeba uzavřít a nejdříve ještě může vrátí oddělovač (kvůli poslednímu příkazu).

Na začátku metody *Lexer::LexTokenInternal*[8], která vrací tokeny, se kontroluje, zda nezbývají ještě nějaké závorky ke vrácení, pokud ano, vrátí příslušný token. Pokud ne, pokračuje normálně dále. Při přečtení konce souboru se vracejí závorky tak, aby došlo k uzavření všech otevřených.

4.3 Klíčové slovo pass

Vložení nového klíčového slova bylo docíleno přidáním makra definující toto klíčové slovo do definice klíčových slov[9] a přidáním podmínky v metodě *Parser::ParseStatementOrDeclarationAfterAttributes*. Ta se volá při analýze výrazů a deklarací. Při nalezení tokenu s klíčovým slovem *pass* se pouze zavolá metoda *Parser::ConsumeToken*, která načte následující token, a pokračuje se dále.

4.4 Autoreturn

V metodě *Parser::ParseFunctionStatementBody*[10], která se stará o syntaktickou analýzu těla funkce, bylo přidáno zjištění návratového typu (tato informace je předána této metodě při jejím volání). Pokud není *void*, nastaví se proměnná *Autoreturn* objektu SA na *true*. Tato proměnná se poté kontroluje při analýze samotného těla funkce a zajistí, že se nebude automaticky přidávat příkaz *return* jinam než do funkcí a metod tříd. V případě *function-try* bloku bylo třeba tuto kontrolu přidat do *Parser::ParseFunctionTryBlock* a nastavení proměnně *Autoreturn* také do *ParseCXXTryBlockCommon*.

Dále se volá metoda *Parser::ParseCompoundStatementBody*. Ta má na starost syntaktickou analýzu složených výrazů. Zde se kontroluje, zda tělo již neobsahuje příkaz *return* (pouze na nejvyšší úrovni těla funkce, pokud bude *return* například v těle příkazu *if*, nebude na něj brán zřetel).

Pokud nebyl *return* nalezen a proměnná *Autoreturn* je nastavena na *true*, vezme se poslední příkaz (v Clangu je třída pro výrazy *Expr* potomek třídy *Stmt*) z těla funkce a zkontroluje se, že se jedná o výraz (*Expr::classof()*[11]). To vyřadí příkazy, které nemá smysl vracet, jako třeba *if* nebo *while*. Tuto

[7] soubor `llvm/tools/clang/include/clang/Lex/Preprocessor.h`
[8] soubor `llvm/tools/clang/lib/Lex/Lexer.cpp`
[9] soubor `llvm/tools/clang/include/clang/Basic/TokenKinds.def`
[10] soubor `llvm/tools/clang/lib/Parse/ParseStmt.cpp`
[11] soubor `llvm/tools/clang/include/clang/AST/Expr.h`

podmínku ale splňuje například i příkaz *throw* (třída *CXXThrowExpr*), proto je ještě dále nutné ohlídat, jakou hodnotu má výsledek výrazu. Je volána metoda *Expr::getType*, která vrací objekt třídy *QualType*. Výsledek metody *QualType::getAsString* můžeme jednoduše porovnat s řetězcem „void“ a tím eliminovat výrazy, u kterých se nemá cenu pokoušet o vytvoření *returnu*. U vyhovujících příkazů (výrazů) se z tohoto pomocí volání *Sema::ActOnReturnStmt* příkaz *return* a zařadí se do seznamu příkazů namísto původního. Pokud je výsledek typu *void*, vypíše se varování (*warn_not_autoreturn* přidáno do diagnostiky[12]).

O kontrolu typu výrazu se dále postará sémantický analyzátor, který při použití výrazu, který nemá určenou implicitní konverzi na návratový typ funkce, vypíše chybové hlášení. Ten také standardně vypíše varovnou hlášku, pokud je v *non-void* funkci možná cesta bez vrácení nějaké hodnoty.

Předtím bylo neúspěšně testováno řešení na úrovni sémantického analyzátoru. Dále také řešení, kdy byla na poslední příkaz použita metoda *Stmt::getStmtClassName* a její výsledek byl porovnáván s následujícími řetězci reprezentující typy výrazů, pro které se má využít automatické vrácení: *BinaryOperator*, *DeclRefExpr*, *CallExpr*, *IntegerLiteral*, *CharacterLiteral*, *MemberExpr*, *CXXMemberCallExpr*. Z třídy *Expr* dědí přes 90 tříd a při použití výše uvedeného řešení by bylo snadné pominout některý výraz, který má být vrácen, proto se konečné řešení jeví jako mnohem lepší.

4.5 Třídy a struktury

Vzhledem ke konečné zvolené syntaxi, kdy nebylo do definic tříd a struktur nic přidáno, ale bylo pouze využito odsazovací syntaxe, stačilo odebrat nutnost psát za definicí třídy (struktury) středník.

4.6 Funkce

U funkcí a metod nakonec také nebylo třeba nic upravovat. Kromě automatického návratu posledního příkazu viz 4.4.

4.7 For

For cyklus je analyzován metodou *Parser::ParseForStatement*[13]. Zde bylo třeba upravit (odebrat) kontrolu, zda po klíčovém slově *for* následuje otevírací závorka. Dále bylo zabráněno objektu *BalancedDelimiterTracker*, který se stará o závorky, aby odebral tokeny „(“ a „)“. Bylo přidáno odebrání tokenu „:“ za podmínkou a případné hlášení chyby, pokud dvojtečka chybí.

[12] soubor `llvm/tools/clang/include/clang/Basic/DiagnosticCommonKinds.td`
[13] soubor `llvm/tools/clang/lib/Parse/ParseStmt.cpp`

Dochází zde k vypnutí analýzy pomocí odsazování a jejímu opětovnému zapnutí (pokud byla původně zapnuta), aby bylo možné rozdělit část *for* cyklu před samotným tělem na více částí.

4.8 While

U *while* cyklu bylo také nutné upravit kontrolu v *Parser::ParseWhileStatement*, jestli po klíčovém slově následuje otevírací závorka. Používá se zde metoda *Parser::ParseParenExprOrCondition*, která se stará o syntaktickou analýzu výrazů v závorkách. Té byl přidán parametr, který říká, zda je skutečně výraz v závorkách či nikoli. Zde je tedy *ParseParenExprOrCondition* volána bez závorek a části kódu v ní, které se starají o práci se závorkami, jsou podmíněny tímto parametrem. Dále je v této metodě přidáno dočasné vypnutí analýzy pomocí odsazování, aby bylo možné rozdělit podmínku na více řádků. To se samozřejmě týká i ostatních konstrukcí, které používají *ParseParenExprOrCondition*, mezi ně patří *do while*, *if* a *else if*, Bylo přidáno odebrání „:“ za podmínkou a hlášení chyby, pokud tato chybí.

4.9 Do while

Zde bylo v metodě *Parser::ParseDoStatement* přidáno odebrání tokenu s dvojtečkou a chybové hlášení, úpravy u analýzy podmínky po klíčovém slovu *while* využívají stejné metody jako u samotného *while* cyklu.

4.10 If, else if, else

I u příkazu *if* byla upravena část se závorkami a odebrání tokenu „:“ v metodě *Parser::ParseIfStatement* stejně jako u *while* cyklu. Odebrání tohoto tokenu bylo přidáno i za klíčové slovo *else*, případně za podmínku u *else if*.

4.11 Namespace

Vzhledem k původní syntaxi a již implementovaným tvoření bloků pomocí odsazování, není třeba u *namespace* SA nijak upravovat.

4.12 Enum

U výčtu je nastaven token vracený při odřádkování se stejným odsazením na čárku a poté opět zpět na středník. Na konci metody byl ještě přidán kód starající se o středník, který v původní syntaxí za *enumem* je.

4.13 Try catch

Do metody *Parser::ParseCXXTryBlock*[14] byla přidána kontrola, zda po *try* následuje „:“. V *Parser::ParseCXXCatchBlock* byla odstraněna kontrola závorek okolo deklarace v *catch* části. Po ní byla přidána nutnost psát za ní dvojtečku.

4.14 Šablony

Analýza šablon je implementována v souboru `llvm/tools/clang/lib/Parse/ParseTemplate.cpp`. Je využito vracení závorek při zvýšení odsazení. V metodě *Parser::ParseDeclarationStartingWithTemplate* je nastavena na „<“. Token vrácený při odřádkování s následným stejným odsazením byl nastaven na nově přidaný token *template_new_param*. Ten v rozšířené syntaxi částečně nahrazuje token čárky, tedy rozlišuje jednotlivé parametry při analýze seznamu parametrů šablony, jeho použití je objasněno dále. Na konci metody *Parser::ParseTemplateDeclarationOrSpecialization* jsou znaky opět nastaveny na „;“ a „{“, které jsou většinou používány.

4.14.1 Úplná specializace

Pokud je v metodě *ParseTemplateDeclarationOrSpecialization* za klíčovým slovem *template* nalezeno nové klíčové slovo *spec* (bylo přidáno do definice tokenů[15]), je token *spec* přečten a zároveň je zkonzumován i další token (jedná se o *template_new_param*, který je vrácen lexikálním analyzátorem kvůli stejnému odsazení, které má následující deklarace). Díky tomu se vyhneme zbytečnému voláni metody *Parser::ParseTemplateParameters*, seznam parametrů zůstane prázdný a díky tomu analyzátor pozná, že se jedná o specializaci.

4.14.2 Typename zkratka

V metodě *Parser::ParseTemplateParameterList*, která má za úkol analýzu seznamu parametrů byla přidána kontrola, zda se za klíčovým slovem *typename* nenachází dvojtečka. Pokud tomu tak je, je ve smyčce volána metoda *Parser::ParseTemplateParameter* s nově přidaným argument *Typenames* nastaveným na *true* (defaultně je v deklaraci metody nastaven na *false*).

Tato metoda v případě nastavení *Typenames* na *true* volá metodu *Parser::ParseTypeParameter*, která vrací jeden přečtený parametr, s tím je v tomto případě zacházeno jako s *typenamem*, tedy je nastavena proměnná *TypenameKeyword* na *true*, i když je toto klíčové slovo čteno pouze jednou na začátku a ne u každého parametru.

Výsledek volání je zařazen do seznamu parametrů.

[14] `llvm/tools/clang/lib/Parse/ParseStmt.cpp`

[15] soubor `/llvm/tools/clang/include/clang/Basic/TokenKinds.def`

V této části je důležité použití vracení nového tokenu *template_new_param*. Smyčka v *ParseTemplateParameterList* totiž čte při nalezení této „typename zkratky“ parametry, které jsou oddělené čárkou. Pokud je nalezen token *template_new_param*, ví, že následující kód již není součástí. Při čtení dalšího parametru za touto zkratkou, je ke kontrole, zda je další token čárka, přidána možnost, že používáme novou syntaxi a místo tokenu čárky je token *template_new_param*. Kvůli této úpravě byl také tento token přidán k čárce do metody *Parser::isStartOfTemplateTypeParameter*, která rozhoduje, jestli jsme narazili na „type parametr“.

4.14.3 Šablona jako parametr šablony

V metodě *Parser::ParseTemplateTemplateParameter* je, po zkonzumování klíčového slova *template*, přidána kontrola, zde je následující token číselná konstanta, pokud ano, je dále přečtena její hodnota. Toto číslo je poté použito jako počet *typename* parametrů, které se mají vytvořit pomocí metody *Sema::ActOnTypeParameter*[16] a vložit do seznamu parametrů. Dále je v tomto případě vynecháno čtení klíčového slova *class*. O zbytek se stará již existující kód.

[16] `llvm/tools/clang/lib/Sema/SemaTemplate.cpp`

Kapitola 5

Zhodnocení

Následuje porovnání nové syntaxe s původní a moje subjektivní zhodnocení jejich výhod či nevýhod.

5.1 Porovnání s původní syntaxí

První je vždy uvedena ukázka kódu s původní syntaxí, pod ní s implementovaným rozšířením.

5.1.1 Třídy

Třídy bez složených závorek vypadají dle mého názoru lépe, zápis je přehlednější. Velmi se zde hodí automatické vracení posledního výrazu, která zkracuje psaní get metod.

```
class Trida{

   public:
   Trida(int x) : cislo(x)       {}

   int get(){
      return cislo;
   }

   private:
   int cislo;

};
```

```
class Trida

    public:
    Trida(int x) : cislo(x)
        pass

    int get()
        cislo

    private:
    int cislo
```

5.1.2 Funkce

Myslím, že zjednodušení zápisu funkcím (případně metodám tříd) prospělo, co se čitelnosti týká.

```
int nasob(int a, int b){
    return a * b;
}

int nasob(int a, int b)
    a * b
```

5.1.3 Cykly

Cykly by mohly být bez jakéhokoli symbolu za podmínkou, po vzoru tříd a funkcí, kde žádný přidán není. Myslím si ale, že narozdíl od tříd, kde hlavička bývá bez odsazení a funkce, kde ji na první pohled jasně definuje závorka okolo parametrů, je zde lepší použít dvojtečku za podmínkou. Uvození cyklu a jeho tělo takto zůstane jasněji oddělené, než kdyby dvojtečka chyběla.

```
// nasobilka
for(int i = 1; i <= 10; i++){

    for(int j = 1; j <= 10; j++){
        cout << i*j;
        cout << "\t";
    }
    cout << endl;

}
```

```
for int i = 1; i <= 10; i++:

    for int j = 1; j <= 10; j++:
        cout << i*j
        cout << "\t"

    cout << endl


while (x < y){
    delejNeco();
    x++;
}

while x < y:
    delejNeco()
    x++


do{
    delejNeco();
    x++;
}while(x < y);

do:
    delejNeco(x)
    x++
while x < y
```

5.1.4 Switch

Pro *switch* platí to samé, co pro cykly. Dvojtečka je zde v zájmu lepší čitelnosti. Odstraněné složené závorky také prospívají.

```
switch(mesic){

    case 1: case 3: case 5: case 7: case 8:
    case 10: case 12:
        pocet_dni = 31;
            break;

    case 4: case 6: case 9: case 11:
        pocet_dni = 30;
        break;

    case 2:
        pocet_dni = unor(rok);
        break;

}

switch mesic:

    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
        dni = 31
        break

    case 4: case 6: case 9: case 11:
        dni = 30
        break

    case 2:
        dni = unor(rok)
        break
```

5.1.5 If, else if, else

If není výjimka, odstranění závorek okolo podmínek a přidání dvojtečky za podmínku zlepšuje čitelnost, zejména pokud se kód více rozvětvuje.

```
if(x < 0){
    cout << "x␣bylo␣mensi␣nez␣0\n";
    x = 0;
}else if(x == 0){
    cout << "x␣bylo␣0\n";
}else{
    cout << "x␣bylo␣vetsi␣nez␣0\n";
    x = 0;
}
```

```
if x < 0:
    cout << "x␣bylo␣mensi␣nez␣0\n"
    x = 0
else if x == 0:
    cout << "x␣bylo␣0\n"
else:
    cout << "x␣bylo␣vetsi␣nez␣0\n"
    x = 0
```

5.1.6 Namespace

Zápis *namespace* nešel, kromě nahrazení složených závorek odsazením, více zjednodušit.

```
namespace jmeno{
    int nasob(int a, int b){
            return a * b
    }
}
```

```
namespace jmeno
    int nasob(int a, int b)
       a * b
```

5.1.7 Enum

Enumu přidává na lepší čitelnosti nahrazení čárek mezi jednotlivými položkami stejným odsazením.

```
enum Cisla{
  jedna = 1,
  dva,
  tri,
  deset = 10,
  jedenact
};

enum Cisla
  jedna = 1
  dva
  tri
  deset = 10
  jedenact
```

5.1.8 Try catch

Zde došlo také ke zjednodušení a odlehčení.

```
try{
   throw "A12";
}
catch (const char * x){
   cout << "Exc.: " << x << endl;
}

try:
   throw "A12"

catch const char * x:
   cout << "Exc.: " << x << endl
```

5.1.9 Šablony

U šablon byla zpřehledněn zápis, kdy je místo závorek použito odsazení. Dále bylo přidáno zkrácení některých konstrukcí, viz níže.

```
template<typename T>
struct S{
 T val;
};

template
 typename T
struct
S~T val
```

Bylo přidáno zjednodušené zapisování opakujících se *typenamů*. Zároveň ale může být zapsán každý zvlášť.

```
template<typename T1, typename T2>
struct S{
 T1 val1;
 T2 val2;
};

template
 typename: T1, T2
struct
S~T1 val1
 T2 val2

// Je mozne pouzit i toto
template
 typename T1
 typename T2
struct
S~T1 val1
 T2 val2
```

Také přibyl zkrácený zápis *template templatů*, který nahrazuje *typenamy* v parametru pouze jejich počtem. Spolu s výše uvedeným dle mého podstatně zlepšuje čitelnost a délku kódu.

```
template<typename T1, typename T2,
         template <typename,typename> class  C>
struct SS{
 C<T1,T2> s;
};

template
 typename: T1, T2
 template 2 C
struct SS
 C<T1,T2> s
```

5.2 Ukázka zdrojových kódů

Následuje ukázka zdrojových kódů vybraných pro testování. Vlevo je vždy původní syntaxe, vpravo nová. Byl kladen důraz na stejné formátování u obou syntaxí tak, aby nebyla jedna zvýhodněna. Pokud je například v jedné vynechán řádek pro lepší čitelnost, je tak učiněno i v druhé. V původní syntaxi je také na místech, kde to lze, použita zkrácená forma cyklů a *ifu* bez složených závorek.

5.2.1 Výpočet Eulerovy funkce

Výsledek Eulerovi funkce pro číslo x je počet menších čísel, které jsou s tímto číslem nesoudělné.

```
#include <iostream>
#include <math.h>
using namespace std;

int rozklad(int zb, int i,
            int * prvoc, int * exp){

 *exp = 0;
 while (zb % i == 0){
  *prvoc = i;
  *exp = *exp +1;
  zb /= i;
 }

 return zb;
}

int main(){

 int prvoc = 0, exp = 0;
 int i, vysl = 1, meziv = 0;
 int n, zb;
 cout << "Zadejte n:\n";

 if ((cin >> n) < 0){
  cout << "Nespravny vstup.\n";
  return 1;
 }

 zb = n;
 for (i = 2; i < sqrt(n); i++){
  zb = rozklad(zb, i, &prvoc, &exp);
  if (exp){
   meziv = (prvoc-1)*pow(prvoc,exp-1);
   vysl *= meziv;
  }
 }

 if (zb != 1)
  vysl = vysl*(zb-1);

 cout << "phi(" << n << ") = ";
 cout << vysl << "\n";

 return 0;
}
```

```
#include <iostream>
#include <math.h>
#pragma PPlusPlus ON
using namespace std

int rozklad(int zb, int i,
            int * prvoc, int * exp)

 *exp = 0
 while zb % i == 0:
  *prvoc = i
  *exp = *exp +1
  zb /= i

 zb

int main()

 int prvoc = 0, exp = 0
 int i, vysl = 1, meziv = 0
 int n, zb
 cout << "Zadejte n:\n"

 if (cin >> n) < 0:
  cout << "Nespravny vstup.\n"
  return 1

 zb = n
 for i =     2; i < sqrt(n); i++:
  zb = rozklad(zb, i, &prvoc, &exp)
  if exp:
   meziv = (prvoc-1)*pow(prvoc,exp-1)
   vysl *= meziv;

 if zb != 1:
  vysl = vysl*(zb-1)

 cout << "phi(" << n << ") = "
 cout << vysl << "\n"

 0
```

5.2.2 Bubble a quick sort

Porovnání zdrojových kódů pro řazení bubble [18] a quick sortem [19].

```
#include <iostream>
using namespace std;

void bubbleSort(int * a, int s){
 for(int i = 0; i < s~- 1; i++)
  for(int j = 0; j < s-i - 1; j++)
   if(a[j+1] < a[j]){
    int tmp = a[j + 1];
    a[j + 1] = a[j];
    a[j] = tmp;
  }
}

void swap(int a[], int l, int r){
 int tmp = a[r];
 a[r] = a[l];
 a[l] = tmp;
}

void quicksort(int a[], int l, int r){
 if(l < r){
  int bound = l;
  for(int i = l + 1; i < r; i++)
   if(a[i] > a[l])
    swap(a, i, ++bound);

  swap(a, l, bound);
  quicksort(a, l, bound);
  quicksort(a, bound + 1, r);
  }
}

void print(int x[], int s){
 for(int i = 0; i < s; i++){
  cout << x[i];
  cout << ' ';
 }
 cout << endl;
}

int main(){

 int x[] = {5, 6, 8, 9, 156, 23, -2};
 int y[] = {5, 6, 8, 9, 156, 23, -2};
 bubbleSort(x, 7);
 quicksort(y, 0, 7);
 print(x,7);
 print(y,7);

  return 0;
}
```

```
#include <iostream>
#pragma PPlusPlus ON
using namespace std

void bubbleSort(int * a, int size)
 for int i = 0; i < size - 1; i++:
  for int j = 0; j < size-i - 1; j++:
   if a[j+1] < a[j]:
    int tmp = a[j + 1]
    a[j + 1] = a[j]
    a[j] = tmp

void swap(int a[], int l, int r)
 int tmp = a[r]
 a[r] = a[l]
 a[l] = tmp

void quicksort(int a[], int l, int r)
 if l < r:
  int bound = l
  for int i = l + 1; i < r; i++:
   if a[i] > a[l]:
    swap(a, i, ++bound)

  swap(a, l, bound)
  quicksort(a, l, bound)
  quicksort(a, bound + 1, r)

void print(int x[], int s)
 for int i = 0; i < s; i++:
  cout << x[i]
  cout << ' '
 cout << endl

int main()

 int x[] = {5, 6, 8, 9, 156, 23, -2}
 int y[] = {5, 6, 8, 9, 156, 23, -2}
 bubbleSort(x, 7)
 quicksort(y, 0, 7)
 print(x,7)
 print(y,7)

 0
```

5.2.3 Heap sort

Ukázka řazení pomocí haldy [20].

```
#include <iostream>
using namespace std;

void swap(int a[], int l, int r){
 int tmp = a[r];
 a[r] = a[l];
 a[l] = tmp;
}

void repairTop(int a[],int bot,int i){
 int tmp = a[i];
 int succ = i*2 + 1;
 if (succ<bot && a[succ]>a[succ+1])
  succ++;

 while (succ <= bot && tmp > a[succ]){
  a[i] = a[succ];
  i = succ;
  succ = succ*2 + 1;
  if (succ<bot && a[succ]>a[succ+1])
   succ++;
  }

 a[i] = tmp;
}

void heapsort(int a[], int size){
 for (int i=size/2 - 1; i>=0; i--)
  repairTop(a, size - 1, i);

 for (int i = size - 1; i>0; i--){
  swap(a, 0, i);
  repairTop(a, i - 1, 0);
 }
}

int main(){
 int a[] = {5, -5, 6, 9, 12, 7, 55};
 heapsort(a, 7);
 for(int i = 0; i < 7; i++){
   cout << a[i];
   cout << ' ';
 }
 std::cout << std::endl;
 return 0;
}
```

```
#include <iostream>
#pragma PPlusPlus ON
using namespace std

void swap(int a[], int l, int r)
 int tmp = a[r]
 a[r] = a[l]
 a[l] = tmp

void repairTop(int a[],int bot,int i)
 int tmp = a[i]
 int succ = i*2 + 1
 if succ<bot && a[succ]>a[succ+1]:
  succ++

 while succ <= bot && tmp > a[succ]:
  a[i] = a[succ]
  i = succ
  succ = succ*2 + 1
  if succ < bot && a[succ]>a[succ+1]:
   succ++

 a[i] = tmp;

void heapsort(int a[], int size)
 for int i=size/2 - 1; i>=0; i--:
  repairTop(a, size - 1, i)

 for int i = size - 1; i>0; i--:
  swap(a, 0, i)
  repairTop(a, i - 1, 0)

int main()
 int a[] = {5, -5, 6, 9, 12, 7, 55}
 heapsort(a, 7)
 for int i = 0; i < 7; i++:
   cout << a[i];
   cout << ' '

  std::cout << std::endl
 0
```

5.2.4 Eratosthenovo síto

Ukázka tzv. Eratosthenova síta [21], algoritmu pro hledání prvočísel menších než n.

```
#include <math.h>
#include <iostream>
using namespace std;

void printSieve(bool * sieve,
                int    size) {
 for (int i = 2; i < size; i++)
  if(sieve[i] == false)
   cout << i << " ";
}

void sieveOfEratosthenes(int uBnd){
 bool * sieve = new bool[uBnd];

 for(int i = 0; i < uBnd; i++)
  sieve[i] = false;

 sieve[0] = sieve[1] = true;
 for(int i = 2;
     i <= sqrt((double)uBnd); i++){
  if(sieve[i] == true)
   continue;
  for(int j = 2*i; j < uBnd; j += i)
   sieve[j] = true;
 }

 printSieve(sieve, uBnd);
 delete[] sieve;
}

int main(){
 sieveOfEratosthenes(100);
 cout << endl;
 return 0;
}
```

```
#include <math.h>
#include <iostream>
#pragma PPlusPlus ON
using namespace std

void printSieve(bool * sieve,
                int    size)
 for int i = 2; i < size; i++:
  if sieve[i] == false:
   cout << i << " "

void sieveOfEratosthenes(int uBnd)
 bool * sieve = new bool[uBnd]

 for int i = 0; i < uBnd; i++:
  sieve[i] = false

 sieve[0] = sieve[1] = true
 for int i = 2;
     i <= sqrt((double)uBnd); i++:
  if sieve[i] == true:
   continue
  for int j = 2*i; j < uBnd; j += i:
   sieve[j] = true

 printSieve(sieve, uBnd)
 delete[] sieve

int main()
 sieveOfEratosthenes(100)
 cout << endl
 0
```

5.3 Měření velikosti

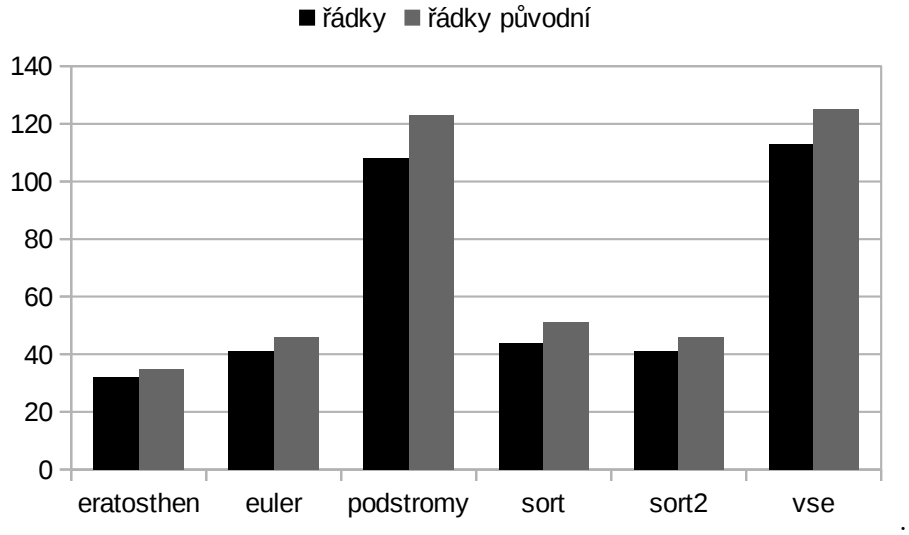
Velikost byla porovnávána u výše uvedených příkladů a dále u dalších dvou (ty jsou všechny na přiloženém CD). Bohužel se nepodařilo zprovoznit nástroj *clang-format*, který by byl na toto měření ideální, proto bylo prováděno pomocí programu *wc*, měřeny byly počty řádků, viz 5.1 a počty znaků, viz 5.2. Oboje bez započítání komentářů.

Měření u vybraných zdrojových kódů ukázalo, že původní syntaxe má o 9 až 16 % více řádků a o 2 až 11 % více znaků.

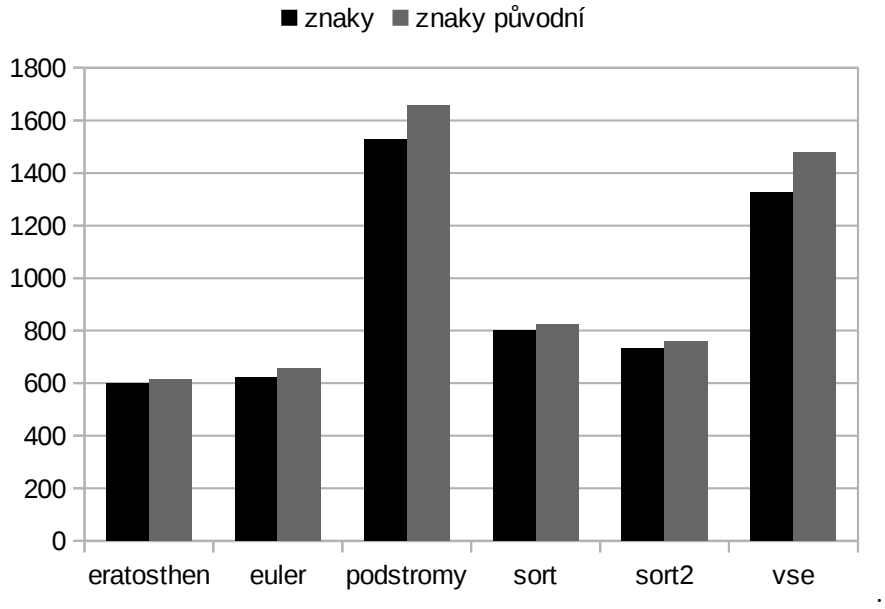
5.4 Zhodnocení rozšíření syntaxe

Myslím, že rozšíření syntaxe zlepšilo výrazně čitelnost kódu. Díky odebrání závorek okolo podmínek a současnému přidání dvojtečky, nahrazení složených závorek u bloků za odsazování, se v zdrojovém kódu lépe orientuje. Odebrání středníků jako oddělovače také výrazně pomohlo v odlehčení.

Za velmi praktické považuji automatické přidání příkazu return za poslední výraz u funkcí a metod a zkrácený zápis šablon.



Obrázek 5.1: Porovnání počtu řádků



Obrázek 5.2: Porovnání počtu znaků

Závěr

Cílem práce bylo navrhnout, implementovat a otestovat rozšíření syntaxe C++ inspirované jazyky Python a CoffeeScript, které ulehčí psaní kódu a zlepší jeho čitelnost. V rešeršní části byly zpracovány informace o překladači Clang/LLVM, jazycích C++, Python a CoffeeScript. Na základě této části bylo navrženo rozšíření syntaxe C++. Jeho implementace byla popsána v následující části. Nakonec bylo provedeno zhodnocení tohoto rozšíření.

Myslím, že tato práce ukázala, že kombinace síly C++ a syntaxe jazyků Python a CoffeeScript má potenciál. Rozšíření základních konstrukcí C++ bylo navrženo, implementováno a otestováno. Vzhledem k rozsáhlosti jazyka C++ je zde ale jistě velký prostor pro další vylepšení.

Literatura

[1] Korban, A.: What if C++ looked more like Python or CoffeeScript? 2014, [online]. [cit. 2016-05-12]. Dostupné z: `http://cpprocks.com/what-if-c-looked-more-like-python-or-coffeescript/`

[2] Janoušek, J.: BI-PJP, FIT ČVUT, Přednáška č.1 - Úvod, struktura překladače. 2014.

[3] Janoušek, J.: BI-PJP, FIT ČVUT, Přednáška č.2 - Lexikální analýza. 2014.

[4] Clang - Features and Goals. [online]. [cit. 2016-05-12]. Dostupné z: `http://clang.llvm.org/features.html`

[5] Lattner, C.; Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, Kalifornie, 2004.

[6] LLVM Download Page. [online]. [cit. 2016-05-12]. Dostupné z: `http://llvm.org/releases/`

[7] Lattner, C.: LLVM Overview. [online]. [cit. 2016-05-12]. Dostupné z: `http://llvm.org/`

[8] Lattner, C.: LLVM Users. [online]. [cit. 2016-05-12]. Dostupné z: `http://llvm.org/Users.html`

[9] Clang vs Other Open Source Compilers. [online]. [cit. 2016-05-12]. Dostupné z: `http://clang.llvm.org/comparison.html`

[10] Zamir, S.: *Handbook of object technology*. Boca Raton: CRC Press, c1999, ISBN 08-493-3135-8.

[11] History of C++. 2016, [online]. [cit. 2016-05-12]. Dostupné z: http://en.cppreference.com/w/cpp/language/history

[12] Python Software Foundation: What is Python? Executive Summary. [online]. [cit. 2016-05-12]. Dostupné z: https://www.python.org/doc/essays/blurb/

[13] Klein, B.: History of Python. [online]. [cit. 2016-05-12]. Dostupné z: http://www.python-course.eu/python3_history_and_philosophy.php

[14] Python Software Foundation: The Python Language Reference. [online]. [cit. 2016-05-12]. Dostupné z: https://docs.python.org/3/reference/compound_stmts.html

[15] CoffeeScript. 2015, [online]. [cit. 2016-05-12]. Dostupné z: http://coffeescript.org/

[16] Estier, T.: What is BNF notation? [online]. [cit. 2016-05-12]. Dostupné z: http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html

[17] van Teijlingen, W.: Implement Pragmas in Clang/LLVM. [online]. [cit. 2016-12-12]. Dostupné z: http://wvanteijlingen.github.io/notes/clang_pragmas

[18] Neckář, J.: Bubble sort. [online]. [cit. 2016-12-29]. Dostupné z: www.algoritmy.net/article/3/Bubble-sort

[19] Neckář, J.: Quicksort. [online]. [cit. 2016-12-29]. Dostupné z: www.algoritmy.net/article/10/Quicksort

[20] Neckář, J.: Heapsort. [online]. [cit. 2016-12-29]. Dostupné z: www.algoritmy.net/article/17/Heapsort

[21] Neckář, J.: Eratosthenovo síto. [online]. [cit. 2016-12-29]. Dostupné z: www.algoritmy.net/article/65/Eratosthenovo-sito

Příloha **A**

Seznam použitých zkratek

SSA Static single assignment form

GCC GNU Compiler Collection

BNF Backusova-Naurova forma

SA Syntaktický analyzátor/analýza

LA Lexikální analyzátor/analýza

Příloha **B**

Obsah přiloženého CD

- `ctime.txt` ..... stručný popis obsahu CD
- `src`
  - `instalace.txt` ..... popis instalace Clang/LLVM
  - `llvm` ..... adresář s upravenými zdrojovými kódy Clang/LLVM
  - `ukazky` ..... testované zdrojové kódy
- `text` ..... text práce
  - `BP_Cerveny_Jan_2017.pdf` ..... text práce ve formátu PDF
  - `src` ..... zdrojové soubory práce