CENG 242

Programming Language Concepts

Spring 2023-2024

Programming Exam 5

Due date: May 19 2023, Sunday, 23:59

Overview

In this assignment, you are going to complete an implementation for **Turing Machine** using C++11 with an emphasis on **template metaprogramming**, i.e. proving turing-completeness of C++ templates.

C++ Template Metaprogramming (TMP) can significantly impact software architecture design. It enables features to be configured at compile-time, improving flexibility for different builds. TMP enforces static type safety and allows complex validation of architectural components to catch errors early. Additionally, TMP facilitates generating optimized data structures and algorithms at compile-time, potentially enhancing performance. However, TMP code can be complex and requires careful design due to compiler limitations. In practice, TMP is valuable for building component frameworks, plugin systems, and even Domain-Specific Languages (DSLs) within an architecture. Overall, TMP offers a powerful tool for crafting robust and adaptable software architectures in C++, but its use requires a measured approach.

When discussing Turing-completeness in the context of C++ templates, it is important to recognize that C++ templates work in functional programming paradigm. That is, instead of loops and conditional statements; considering the templates like a Lambda calculus, with its emphasis on recursion and pattern matching, provides a more relevant lens through which to view Turing completeness in C++ templates.

1 Introduction

This section explores C++ templates, powerful tools for creating generic functions and data structures. We'll start with a classic example: calculating factorials using C++ templates. We'll see how templates can mimic functional programming concepts like recursion, leading to more concise and expressive code in Section 1.1. Next, we'll explore templates that accept a variable number of type arguments (type lists) for even greater flexibility in generic programming in Section 1.2. Then, we'll delve into some basic functionalities specific to type lists the functionalities for element access, manipulation, and transformation in Section 1.3. Finally, you'll be tasked to implement remaining functionalities specific to type lists, as described in Section 1.4. You can implement a Transition Function templates δ to complete the implementation Turing Machine evaluators using `List` in Section 2.2.

1.1 Computing factorial via C++ templates

Pattern Matching and Recursion Understanding C++ templates through a functional lens is particularly fruitful when considering their similarities to languages like Haskell. Haskell, a purely functional programming language, heavily relies on recursion and pattern matching for computations. By viewing C++ templates with this perspective, you can leverage your knowledge of meta-programming concepts to write more elegant and expressive codes. First, let's consider a factorial function in Haskell:

Factorial Example Haskell

```haskell
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

In the corresponding C++ code, the Factorial template uses recursion to calculate the factorial of a number. The `Factorial<N>` specialization defines the value as `N` multiplied by the factorial of `N-1` (achieved through `Factorial<N - 1>::value`). The base case, Factorial, is defined as 1. This demonstrates how recursion and template specialization work together to achieve a functional approach to calculating the factorial.

Factorial Example C++ Templates

```cpp
// @brief inductive case
template <int N>
struct Factorial {
    // C++11 and C++14
    enum { value = N * Factorial<N - 1>::value };
    // alternatively only in C++14
    static constexpr auto value = N * Factorial<N - 1>::value;
};

// @brief base case
template <>
struct Factorial<0> {
    // C++11 and C++14
    enum { value = 1 };
    // alternatively only in C++14
    static constexpr auto value = 1;
};
```

1.2 Type Lists

In this Section, you will see the declaration of the template `List`. Then, The following Sections 1.3 and 1.4 will focus on enhancing the functionalities of `List` by implementing methods for element access, manipulation, and transformation. They will be crucial for building more intricate data structures later on. Finally, the Section 2.2 will then utilize the enhanced `List` to implement a Turing Machine, demonstrating the power and versatility of this custom data structure.

Variadic Arguments C++ templates, since C++11, allow functions to accept a variable number of arguments. To leverage this functionality for building data structures, we can introduce the `List` template. This template represents an ordered sequence of types, providing the foundation for more complex data structures and manipulations.

Declarations The following code snipped box declares a **primary template** named `List`. The ellipsis (...) following `typename` indicates that the template can accept zero or more types as (variadic) arguments (or a.k.a. parameter packs). These types are captured in the parameter pack named `Ts`.

Type Declaration C++ Templates

```cpp
// @brief a primary template
template<typename... Ts>
struct List;
```

Specializations The code snipped box below line defines a **partial specialization** of the `List` template. It specifies that the list can hold at least one element (`T`) followed by zero or more elements (`Ts...`). This specialization essentially defines the structure of the list, with the first element being of type `T` and the remaining elements being captured in the `Ts...` parameter pack.

Partial Specialization — C++ Templates

```
// @brief a partial specialization
template<typename T, typename... Ts>
struct List<T, Ts...> {};
```

The code snipped box below defines a **full specialization** of the `List` template for the empty case. It specifies that an empty list is represented by an empty struct (`List<>`). This is necessary to terminate the recursion when no further elements are added to the list.

Full Specialization — C++ Templates

```
// @brief the full specialization
template<>
struct List<> {};
```

1.3 Basic Functionalities

The type `List` is supposed to have the following functionalities below:

1. `Head` – retrieves the first element of the type `List`.
2. `Tail` – retrieves the remaining elements of the type `List`, excluding the first element.
3. `Size` – retrieves the size of the type `List`, indicating the total number of elements.
4. `At` – retrieves an element at a certain `index` in the `List`. (`index` starts from 0)
5. `Find` – retrieves the `index` of a certain element in the `List`, or 'Nothing' if the element is not found.
6. `Replace` – replaces an element at a certain `index` with a certain element in the `List`.
7. `Append` – adds an element to the `end` of the `List`.
8. `Prepend` – adds an element to the `front` (beginning) of the `List`.
9. `Map` – applies a (function) type to each element of a `List` and returns a new `List` with the wrapped elements.
10. `Filter` – creates a new `List` containing only elements from the original `List` that satisfy a certain `predicate` (a condition) type.

The remaining of this section will guide you through implementing the some basic functionalities like `Head`, `Tail`, `Size`, and `At` for the `List` template. These functions provide the foundation for working with elements within the list. In essence, you'll be provided the C++ code defining the behavior of these functions, specifying how they access and manipulate elements based on their position or retrieve the overall size of the list. Then, the following Section 1.4 dives deeper into advanced functionalities for the `List` template. You'll be tasked with implementing the following features using C++ template metaprogramming: Find, Replace, Append, Prepend, Map, and Filter. By implementing these features, you'll significantly enhance the capabilities of the `List` template, making it a versatile tool for manipulating and transforming data structures in your C++ programs.

Let's Get Functional with Head and Tail! The following C++ code snippets bring the power of functional programming to your lists! Inspired by Haskell, they both implement a List template with built-in access to the first element (head) and the remaining elements (tail).

Head and Tail — C++11 Templates

```
template<typename T, typename... Ts>
struct List<T, Ts...> {
    typedef T head;
    typedef List<Ts...> tail;
};
template<>
struct List<> {
    typedef List<> tail; // empty list
};
```

Head and Tail — C++14 Templates

```
template<typename T, typename... Ts>
struct List<T, Ts...> {
    using head = T;
    using tail = List<Ts...>;
};
template<>
struct List<> {
    using tail = List<>; // empty list
};
```

They equal to the following Haskell code snipped:

Head and Tail — Haskell

```
head t:ts = t
tail [] = []
tail t:ts = ts
```

Now you can manipulate your lists! Assume we created a type `List` with three types, `int`, `char`, and `float` (i.e. `List<int, char, float>`). Then, we accessed the `tail`, which is equal to which is a `List<int, char, float>::tail == List<char, float>`. Finally, we used `tail::head` to get the first element (i.e. `char`) of the tail of the type `List`.

Test Case for Head and Tail — C++ Templates

```
static_assert(std::is_same<List<int,char,float>::tail::head, char>); // pass
```

Size The following C++ code snipped computes the number of elements in a `List`. Alternative to recursive approach, `sizeof...` operator yields the intended result without recursion.

Size — C++ Templates

```
template<typename T, typename... Ts>
struct List<T, Ts...> {
    ...
    enum { value = sizeof...(Ts) };
    // alternatively only in C++14
    static constexpr auto value = sizeof...(Ts);
};
template<>
struct List<> {
    ...
    enum { value = 0 }; // unnecessary
};
```

consider implementing it in recursively! Pseudocode in Haskell would be like below:

size function — Haskell

```
size i [] = i
size i (x:xs) = size (i+1) xs
```

Selection The following C++ code snippets leverages **recursion** to implement selection operation. To achieve, let's first define a type **At** retrieving an element at a certain index in recursion:

At C++ Templates

```
// base case
// @brief get the 0th element of a list
template<typename T, typename... Ts>
struct At<0, List<T, Ts...>> {
    typedef T type;
    // alternatively only in C++14
    using type = T;
};

// inductive case
// @brief gets the Nth element of a list
template<int index, typename T, typename... Ts>
struct At<index, List<T, Ts...>> {
    static_assert(index > 0, "index cannot be negative");
    typedef typename At<index - 1, List<Ts...>>::type type;
    // alternatively only in C++14
    using type = typename At<index - 1, List<Ts...>>::type;
};
```

Here is how to write the corresponding code in Haskell:

at Haskell

```
at 0 (x:xs) = x
at i (x:xs) = at (i-1) xs
```

in order to derive selection functionality in List type, we need to add the following code:

Embedding At into List C++ Templates

```
template<typename T, typename... Ts>
struct List<T, Ts...> {
    ...
    template <int index>
    typedef typename At<index, List<T, Ts...>>::type at;
    // alternatively only in C++14
    using at = typename At<index, List<T, Ts...>>::type;
};
template<>
struct List<> {
    ...
};
```

Lastly, in order to test selection operation implementation. Let's first assume we created a type List with three types, `int`, `char`, and `float` (i.e. `List<int, char, float>`). Then, the compilation of `List<int, char, float>::at<2>` should yield the second element in the list (starting at index zero), which is equal to `float`.

Test Case for At C++ Templates

```
static_assert(std::is_same<List<int,char,float>::at<2>, float>); // pass
```

1.4 Advanced Functionalities

In this section, you are asked to implement the following features using C++ template metaprogramming: Find, Replace, Append, Prepend, Map, and Filter. For each task, you are given a short explanation, C++ primary template, and Haskell pseudocode.

Find This type must find a certain item in a type list. Assume we have a list `List<int,float,char>` called `L3`. Instantiating a type `Find<char,0,L3>` will create a new type where we purposefully store a `value` (like in `Factorial` example) for computing the index of the first occurence of `char`. Thus, accessing the `value` in this type via `Find<char,0,L3>::value` should yield 2. Here is its primary template in C++ and pseudocode in Haskell:

Find Primary Template — C++

```cpp
template<typename Q, int index,
         typename T> // T is List
struct Find;
```

Find Pseudocode — Haskell

```haskell
find q _ [] = -1
find q i (q:xs) = i
find q i (x:xs) = find q (i+1) xs
```

Replace This type must replace a certain item at a certain index in a type list. Assume we have the same list called `L3` in previous example. Instantiating a type `Replace<double,2,L3>` will create a new type where we purposefully store another `type` storing the replaced list. Therefore, accessing `type` like `Replace<double,2,L3>::type` should yield the replaced list as a type `List<int,float,double>`. Here is its primary template in C++ and pseudocode in Haskell:

Replace Primary Template — C++

```cpp
template<typename Q, int index,
         typename T> // T is List
struct Replace;
```

Replace Pseudocode — Haskell

```haskell
replace q 0 (x:xs) = q:xs
replace q i (x:xs) = x:(replace q (i-1) xs)
```

Append This type must adds a certain item at the end (largest) index in a type list. Assume we have the same list called `L3` in previous example. Instantiating a type `Append<double,L3>` will create a new type where we purposefully store another `type` storing the appended list. Therefore, accessing `type` like `Append<double,L3>::type` should yield the appended list as a type `List<int,float,char,double>`. Here is its primary template in C++ and pseudocode in Haskell:

Append Primary Template — C++

```cpp
template<typename NewItem, typename T>
struct Append;
```

Append Pseudocode — Haskell

```haskell
append n ts = ts ++ [n]
```

Prepend This type must adds a certain item at the beginning (zero) index in a type list. Assume we have the same list called `L3` in previous example. Instantiating a type `Prepend<double,L3>` will create a new type where we purposefully store another `type` storing the appended list. Therefore, accessing `type` like `Prepend<double,L3>::type` should yield the prepended list as a type `List<double,int,float,char>`. Here is its primary template in C++ and pseudocode in Haskell:

Prepend Primary Template — C++

```cpp
template<typename NewItem, typename T>
struct Prepend;
```

Prepend Pseudocode — Haskell

```haskell
prepend n ts = n:ts
-- or prepend n ts = [n] ++ ts
```

Map This type must apply a (function) type to each element of a list. Assume we have the same list called `L3` in previous example. Also assume we have a referenceable type `std::add_pointer`. Instantiating a type `Map<std::add_pointer,L3>` will wrap each type in the list `L3` with `std::add_pointer` and purposefully access `type` like `std::add_pointer<char>::type`. Therefore, accessing `Map<std::add_pointer, L3>::type` should yield the mapped list as a type `List<int*,float*,char*>`. Here is its primary template in C++ and pseudocode in Haskell:

Map Primary Template — C++

```cpp
template<template<typename,typename...>
         typename F, typename Ret,
         typename T>
struct Map;
```

Map Pseudocode — Haskell

```haskell
map f ret [] = ret
map f ret (x:xs) = map f (f(x):ret) xs
```

Filter This type must creates a new list containing only elements from the original list that satisfy a certain `predicate` (a condition) type. Assume we have the same list called `L3` in previous example. Also assume we have a referenceable type `std::is_floating_point`. Instantiating a type `Filter<std::is_floating_point,L3>` will wrap each type in the list `L3` with `std::is_floating_point` and purposefully access `value` of wrapped elements like `std::is_floating_point<char>::value` and append them into the new list (which is stored in `Filter<...>::type`) if they satisfy the condition. Therefore, accessing `Filter<std::is_floating_point,L3>::type` should yield the filtered list as a type list `List<float>`. Here is its primary template in C++ and pseudocode in Haskell:

Filter Primary Template — C++

```cpp
template<template<typename,typename...>
         typename F, typename Ret,
         typename T>
struct Filter;
```

Filter Pseudocode — Haskell

```haskell
conditional True a b = a
conditional False a b = b

filter f ret [] = ret
filter f ret (x:xs) = conditional f(x) \
    (filter f (x:ret) xs) (filter f ret xs)
```

2 Turing Machine

In this section, you will be familiar with implementing Turing Machine evaluator using `List` template from the previous section. First, let's briefly introduce the notation and concepts: A Turing machine is an idealised model of a central processing unit (CPU) that controls all data manipulation done by a computer, with the canonical machine using sequential memory to store data. Mathematically, a Turing Machine M (automaton) is defined as a 7-tuple $M = (Q, \Gamma, b, \Sigma, \delta, q_0, F)$ where;

- Q is finite, non-empty set of states;
- Γ is a finite, non-empty set of tape alphabet symbols;
- $b \in \Gamma$ is the blank symbol (the only symbol allowed to occur on the tape infinitely often at any step during the computation);
- $\Sigma \subseteq \Gamma / \{b\}$ is the set of input symbols, that is, the set of symbols allowed to appear in the initial tape contents;
- $\delta : (Q \backslash F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a partial function called the transition function, where L is left shift, R is right shift. If δ is not defined on the current state and the current tape symbol, then the machine halts; intuitively, the transition function specifies the next state transited from the current state, which symbol to overwrite the current symbol pointed by the head, and the next head movement.

- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is the set of final states. The initial tape contents is said to be accepted by M if it eventually halts in a state from F;

This section builds upon the C++ declarations defined earlier in the Sections 1.2, 1.3, and 1.4. In this part of the homework, we'll first introduce States, Input Symbols, Configuration, and Transition Rules templates on top of `List` in the Section 2.1. Next, you'll be focusing on implementing the `TransitionFunction` template in the Section 2.2, a core component of the Turing Machine. Finally, we'll bring everything together by providing a complete Turing Machine implementation that incorporates your `TransitionFunction` and the previously defined templates in the section 2.3.

2.1 Structures

States the finite non-empty set of states Q is represented with the following `State` type. Each state must have a unique integer id (called `value`) as their collection is countable.

State Template C++

```cpp
template<int n>
struct State {
    static int constexpr value = n;
    static char const * name;
};
template<int n>
char const* State<n>::name = "";
```

QStart is a special state with value 0 representing the starting state q_0. The Turing machine assumed to start running at this state.

QAccept is a special state with value -1 representing one of the final states $q_a \in F$. The Turing machine eventually halts when it reaches to q_a.

QReject is a special state with value -2 representing one of the final states $q_r \in F$. The Turing machine eventually halts when it reaches to q_r.

Input Symbols the set of inputs Σ is represented with the following `Input` type. Each input in the alphabet have a unique integer id (called `value`).

Input Template C++

```cpp
template <int n>
struct Input {
    static constexpr int value = n;
    static char const * name;
};
```

InputBlank is a special input with value -1 representing the blank symbol b. It is expected that the input tape, `List<Input...>`, is initially filled with this blank symbol everywhere except for the designated input section.

Note that: Since $b \notin \Sigma$ input values shall be positive integers only (you can make such assertion in your implementation but no need).

Configuration the set of all possible configurations for a turing machine is represented with the following `Configuration` type. A `Configuration` should contain a 3-tuple (`InputTape`, `State`, `Position`). A Turing machine will contain a configuration to indicate its current state, input tape, and position.

Configuration Template C++

```
template<typename InputTape, typename State, int Position>
struct Configuration {
    using input_tape = InputTape;
    using state = State;
    static constexpr int position = Position;
};
```

Note that: Without loss of generality, the transition function δ maps the configuration set to itself e.g. $\delta : (\sigma_t, q_t, p_t) \to (\sigma_{t+1}, q_{t+1}, p_{t+1})$ where $\delta(\sigma_t, q_t) = (\sigma_{t+1}, q_{t+1}, D)$, and $p_{t+1} = p_t \oplus D$.

Transition Rules the graph of the transition function is represented with `Rule` type. A `Rule` is a 5-tuple (`OldState`, `Input`, `NewState`, `Output`, `Move`). The transition $\delta(q_t, \sigma_t) = (q_{t+1}, \sigma_{t+1}, D)$ is represented by a `Rule` with `OldState` $= q_t$, `Input` $= \sigma_t$, `NewState` $= \sigma_{t+1}$, `Output` $= \sigma_{t+1}$, and `Move` $= D$.

Rule Template C++

```
enum Direction { LEFT = -1, STOP = 0, RIGHT = +1 };

template<typename OldState, typename Input,
         typename NewState, typename Output, Direction Move>
struct Rule {
    typedef OldState old_state;
    typedef Input input;
    typedef NewState new_state;
    typedef Output output;
    static constexpr Direction direction = Move;
};
```

2.2 Transition Function

The transition function δ can be represented by a typelist and its manipulations. The `Configuration` contains 3-tuple (`InputTape`, `State`, `Position`) and the `Transitions` stores all the possible state transitions, i.e. `List<Rule...>`, for the Turing machine. In this (and final) part of the assignment, you are expected to fill the `TransitionFunction` template so that accessing the `end_config` yields the final configuration after Turing Machine halted and so on.

TransitionFunction Template C++

```
// @brief inductive case
template<typename Config, typename Transitions, typename = void>
struct TransitionFunction { /* @TODO */
    using end_config = ...;
    using end_input = ...;
    using end_state = ...;
    static constexpr auto end_position = ...;
};

template<bool C, typename = void> struct EnableIf;
template<typename Type> struct EnableIf<true, Type> { using type = Type; };
```

TransitionFunction Template C++

```
template<typename A, typename B> struct is_same { static constexpr auto value = false; };
template<typename A> struct is_same<A, A> { static constexpr auto value = true; };

// @brief base case
template<typename InputTape, typename State, int Position, typename Transitions>
struct TransitionFunction<Configuration<InputTape, State, Position>, Transitions,
                  typename EnableIf<is_same<State, QAccept>::value ||
                                    is_same<State, QReject>::value>::type>
{ /* @TODO */
    using end_config = ...;
    using end_input = ...;
    using end_state = ...;
    static constexpr auto end_position = ...;
};
```

In order to implement `TransitionFunction` template, it is a good practice to use pattern matching and recursion. Assume we have a configuration (`InputTape`, `State`, `Position`) and transitions `List<Rule...>`. We can define the `TransitionFunction` template to recursively determine the final configuration after the machine halts;

- The **base case** is when the Turing Machine's current `State` is in the terminating states, either `QAccept` or `QReject` for our case. In the `TransitionFunction` templates above, it can be seen that this termination is handled via pattern matching mechanism. The Base case should define `end_config`, `end_input`, `end_state`, and `end_position` properly.
- The **inductive case** is when the Turing Machine's current `State` is neither `QAccept` nor `QReject`. This time, `TransitionFunction` should first compute determine δ(`State`, `Input`) and define `end_config`, `end_input`, `end_state`, and `end_position` recursively.
  - To determine the next state (q_{t+1}) and output symbol (σ_{t+1}) based on the current state (q_t) and input symbol (σ_t), you can filter out the `Transitions` with a custom lambda function. The function should check if the rule's `OldState` and `Input` fields match q_t and σ_t respectively.
  - The (first) rule found in the filtered list, is the matching rule i.e.e $\delta(q_t, \sigma_t) = (q_{t+1}, \sigma_{t+1}, D)$. If no rule is found (i.e. the filtered list is empty), the matching rule should be defined as $\delta(q_t, \sigma_t) = (q_f, \sigma_t, 0)$, where q_f is the rejection state (`QReject`).
  - Once the matching rule is obtained, the next `Configuration` can be defined by the `NewState` (q_{t+1}), `Output` (σ_{t+1}), and `Move`. In essence, the next `InputTape` is same as the current input tape except that the input character at the current `Position` is `Output` (which is `Input` in the current `InputTape`). Likewise, the next `Position` is the sum of current `Position` and `Move`.
  - Lastly, the next `TransitionFunction` should be defined by the next `Configuration` recursively. In order to initiate recursion, The `end_config`, `end_input`, `end_state`, and `end_position` should be defined by that of the next `TransitionFunction`.

2.3 Turing Machine

The Turing Machine M can be represented by an `InputTape` and `Transitions`. The `InputTape` is a typelist of accepted characters in the alphabet, i.e. `List<Input...>`. The `Transitions` is a typelist of transition rules, i.e. `List<Rule...>`.

Turing Machine Template C++

```cpp
template<typename InputTape, typename Transitions>
struct TuringMachine {
    using input_tape = InputTape;
    using transitions = Transitions;
    using start_state = QStart;

    using start_config = Configuration<input_tape, start_state, 0>;
    using transition_function = TransitionFunction<start_config, Transitions>;

    using end_config = typename transition_function::end_config;
    using end_input = typename transition_function::end_input;
    using end_state = typename transition_function::end_state;
    static constexpr auto end_position = transition_function::end_position;
};
```

3 Specifications and Notes

1. **Implementation and Submission:** The initial files are available in the Virtual Programming Lab (VPL) activity called "PE5" on odtuclass. You can download them and work on your local device, or directly work on the VPL's editor. The last saved versions of your `list.cpp` and `turing-machine.cpp` files will be used for final grading. Make sure that your implementation is compatible with the provided primary templates. Ensure your code compiles using the following command.

```
$ g++ -std=c++11 main.cpp
```

2. Keep in mind that we will not run your code, all test cases will be checked in compile-time.

3. **Cheating: We have zero tolerance policy for cheating.** People involved in cheating (any kind of code sharing and codes taken from internet included) will be punished according to the university regulations.

4. **Evaluation:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. No erroneous input will be used. Also, none of the inputs will include empty-transition loops. Thus, you **do not need to** detect and eliminate computation loops; you can safely assume all string processing will take a finite amount of time.

5. The given sample inputs are only to ease your debugging process and are **not extensive**. Furthermore, it is not guaranteed that they cover all the cases for required functions. As a programmer, it is **your responsibility** to consider such extreme cases for the functions. Your implementations will be evaluated on a more comprehensive set of test cases to determine your **final** grade after the deadline.