ASSIGNMENT OF BACHELOR’S THESIS

Title: C++ compile-time generated lexical analyzer
Student: Boris Rúra
Supervisor: Ing. Jan Trávníček
Study Programme: Informatics
Study Branch: Computer Science
Department: Department of Theoretical Computer Science
Validity: Until the end of summer semester 2019/20

Instructions

Study the lexical analyser generator flex.

Study the possibilities of C++ compile-time code evaluation available in the C++17 standard and the C++20 standard draft.
Propose a suitable flex features subset and propose its definition purely using C++ language suitable for later compile-time processing.
Implement a functional prototype of a library designed to recognise lexical elements according to the selected flex feature subset with functionality equivalent to flex.
Measure the space difference of lexical analyser generated by flex and your implementation on appropriately designed test inputs.
Measure time and space complexity of lexical element recognition using lexical analyser generated by flex and your implementation on appropriately designed test inputs.

References

Will be provided by the supervisor.
Bachelor’s thesis

C++ compile-time generated lexical analyzer

Boris Rúra

Department of Computer Science
Supervisor: Ing. Jan Trávniček, Ph.D.

May 15, 2019
Acknowledgements

I would like to thank Hana Dusíková for showing me C++ magic and for her continuous help throughout this whole thesis. I would also like to thank my supervisor Jan Trávníček for being patient while I fiddled with optimizations when I promised I was writing the thesis and always finding time for consultations. Last but not least I would like to thank my parents who always believe in me and push me to be the best I can be. Without these people I could never have done this.
I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 15, 2019
Užitečnost lexikální analýzy je nepopíratelná, ať už se jedná o zpracování přirozených jazyků, jako jsou textové zprávy, nebo zpracování zdrojových kódů kompilátory. Zjednodušení jejich tvorby a zvýšení jejich výkonnosti je proto vždy vítáno. Tato práce si klade za cíl ukázat odlišný přístup v tvorbě lexikálních analyzátorů řešením problematiky z metaprogramovací perspektivy s využitím připraveného standardu C++20. Dokazuje, že metaprogramovací přístup může přinést přinejmenším stejné kompaktní spustitelné soubory jako standartní a zároveň konkurenceschopnou rychlost lexikální analýzy. Dochází k závěru, že se jedná o schodný přístup k problematice a odstraňuje potřebu externích programů, které generují lexikální analyzáory, přestože se překladače v této oblasti stále vyvíjejí a časy kompilace mohou působit potíže.

Klíčová slova C++, C++20, template metaprogramming, lexikální analýza, generátor lexikálních analýzérů
Abstract

Usefullness of lexical analysis is undeniable, be it in processing of natural language such as text messages or in processing of source code by compilers. As such, simplifying the creation and boosting the performance of lexical analyzers is always welcome. This thesis aims to show a different approach to generating lexical analyzers by tackling this issue as a metaprogramming one, while leveraging the features of upcoming C++20 standard. It demonstrates that the metaprogramming approach can yield at least as compact binaries as the standard one while being competitive in the speed of the lexical analysis. It concludes that this is a viable approach and removes the need for external programs generating lexical analyzers, albeit compilers are still evolving in this area and compilation times may be a concern.

Keywords C++, C++20, template metaprogramming, lexical analysis, lexical analyzer generator
Contents

Introduction ................................................................................. 1
  Goals ................................................................................... 1
  Organization of this thesis ...................................................... 1

1 Lexical analysis ........................................................................... 3
  1.1 Token ............................................................................... 3
  1.2 Regular expressions ........................................................... 4
  1.3 Lexical analyzer generator ................................................... 4
  1.4 Table driven approach ........................................................ 4

2 Current solutions .......................................................................... 5
  2.1 Flex .................................................................................. 5
  2.2 Antlr ................................................................................ 8
  2.3 Selected feature set ............................................................. 10

3 C++ metaprogramming features .................................................. 11
  3.1 Constant expressions .......................................................... 11
  3.2 Class non-type template parameters (cnttp) ......................... 12
  3.3 Fold expressions ................................................................. 13

4 CTLE - Interface ...................................................................... 15
  4.1 states ............................................................................... 15
  4.2 rule ............................................................................... 16
  4.3 lexer .............................................................................. 17

5 Design ...................................................................................... 19
  5.1 Specifying a rule ................................................................. 19
  5.2 Regular expressions ........................................................... 22
  5.3 Using the result in an action .................................................. 23
  5.4 The result of a rule matching ............................................... 24
5.5 Storing rules and state definitions ........................................... 24
5.6 Making the lexing function .................................................... 24
5.7 Generating lexing function for states ..................................... 25
5.8 Allowing user to change the state of the lexer ...................... 25
5.9 Making the lexer extendable ............................................... 26
5.10 Modularity ........................................................................ 27

6 Comparing the generated lexers ............................................. 29
   6.1 Benchmarking .................................................................. 30
   6.2 Inspecting assembly generated by CTLE ............................ 31

7 Conclusion ............................................................................ 35

Bibliography ............................................................................ 37

A Contents of the attached CD .................................................. 39
List of Figures

1.1 compilation steps [2] ........................................... 3
5.1 algorithm for creating lexing functions .................... 25
6.1 sizes of binaries .............................................. 29
6.2 readelf output for CTLE ...................................... 29
6.3 sizes of binaries after strip command ...................... 30
6.4 benchmark results ............................................ 30
6.5 the analyzed lexer ........................................... 31
6.6 assembly of empty action .................................... 31
6.7 assembly of an action returning a token .................. 32
6.8 assembly of action using a lexeme ......................... 32
Introduction

Usefulness of lexical analysis is undeniable, be it in processing of natural language such as text messages or source code in compilers. As with all things that are widely used an increase in their performance and simplification of their creation is always welcome. This is not a new idea and it has been taken on from multiple angles. From handcoding the whole lexer to creating a new syntax for specifying the lexer and transpiling to a programming language. This thesis takes a different approach, tackling the issue as a metaprogramming problem and using new C++20 features to implement a lexer generator/metclass.

Goals

The first goal of this thesis is to show current solutions to this problem, discuss their strengths and weaknesses and usage.

The second goal is to introduce features of C++, mainly the C++20 standard, that allow for metaprogramming.

The main goal of this thesis is to design and implement a compile-time generated lexical analyzer with an interface similar to Flex.

Organization of this thesis

Chapter 1 introduces lexical analysis and lexical analyzer generators. Chapter 2 discusses current solutions to generating lexical analyzers. Chapter 3 provides an overview of C++ features used to generate the lexical analyzer. Chapter 4 contains an overview of user exposed classes and their usage in creating a lexer. Chapter 5 explains the design decisions and shows how some of the internals work. Chapter 6 compares the current solutions to this one, benchmarking its speed and size of the binaries produced. Chapter 7 concludes this thesis summarizing the learnings.
Lexical analysis is the first step of compilation process. It turns the source code into a stream of tokens. Lexical analyzers are also sometimes called scanners because they “scan” the input for tokens.\[2\]

1.1 Token

A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.\[2\]

```
int foo = 3;
```

The example above, would be lexed as these tokens.\[6\]

\(<\text{INT}|->, \text{<IDENTIFIER|"foo"}>, \text{<ASSIGN,-}>, \\
\text{<NUM, 3}>, \text{<SEMICOLON,-}>
```

The input that was mapped into a token is called a lexeme, e.g. ”int” is the lexeme of token \(<\text{INT}|->\).
1. **Lexical analysis**

1.2 **Regular expressions**

Definition of regular expressions [3]:

1. $\emptyset$, $\lambda$ and $a \in A$ are regular expressions.
2. If $\alpha$ and $\beta$ are regular expressions then $\alpha + \beta$, $\alpha \cdot \beta$, $\alpha^*$ and $(\alpha)$ are also regular expressions.
3. All regular expressions can be obtained by applying the rules 1 and 2 finitely many times.

Regular expressions are a type of language-defining notation. They define exactly the same languages that the various forms of automata describe: the regular languages. However, they offer something that automata do not: a declarative way to express the strings we want to accept. Thus regular expressions serve as the input language for many systems that process strings, for example lexical analyzer generators. [2] Even though the regular expressions used in this thesis allow more syntactical constructs than the definition above, they still can be translated to just those 4 operations.

- $[abc]$ is equivalent to $(a+b+c)$
- $a+$ is equivalent to $a \cdot a^*$
- etc.

1.3 **Lexical analyzer generator**

A lexical analyzer generator is a transpiler. It receives an input, written in the language of the generator and produces an output, the source code of a lexical analyzer in the specified programming language. While the input languages of generators vary, the idea is mostly the same. Their input is a list of rule definitions similar to:

```
REGEX -> TOKEN ACTION
```

In which **REGEX** is a regular expression that must match if the rule is to be used, **TOKEN** is the identifier which is returned to the user and **ACTION** is an optional statement in the output language, that is to be executed if the rule is matched.

1.4 **Table driven approach**

We can represent an NFA by a transition table, whose rows correspond to states, and whose columns correspond to the input symbols and $\epsilon$. The entry for a given state and input is the value of the transition function applied to those arguments. If the transition function has no information about that state-input pair, we put $Q)$ in the table for the pair. [2]
Current solutions

There are two approaches to creating lexical analyzers. They are either being hand coded, which is the approach most compilers take today. Or we use lexical analyzer generators.

The issue with the former is, that while it brings the desired performance, it requires a lot of boilerplate and is error-prone, while the latter requires the user learns a whole new syntax of the generator itself. Furthermore, the user has little to no way of debugging the generated lexer, as most lexers use the table driven approach. Which makes the source code highly unreadable. This thesis focuses on the latter approach, comparing analyzers used today\(^2\).

2.1 Flex

The most common lexer generator, called Flex, is a spiritual successor to AT&T’s Lex. Its manpage reads: flex is a tool for generating scanners: programs which recognized lexical patterns in text. It was derived from original Lex and rewritten into C around 1987.

2.1.1 Installation

The installation of this generator is as straightforward as can be. It is available in aptitude package repositories.

```
apt install flex
```

\(^2\)The comparison is done on Ubuntu 18.04.2 LTS - bionic.
2. Current solutions

2.1.2 Syntax

The input file consists of three parts only separated by `%%`:

- **definitions**
- **rules**
- **user code**

Contents of each part will be discussed in a dedicated section.

2.1.2.1 Definitions

The definitions section contains declarations of simple name definitions to simplify the scanner specification, and declarations of start conditions. Name definitions have the form:

```
name definition
```

The definition can subsequently be referred to using `{name}`, which will expand to `(definition)`.

The section may also contain definitions of start conditions. The flex manual says: 'flex provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with "<sc>" will only be active when the scanner is in the start condition named "sc".' The syntax for specifying a start condition is as follows:

```
%x NAME // this condition is exclusive
%s NAME // this one is inclusive
```

A start condition may have one of two types *inclusive* marked as `s` or *exclusive* marked as `x`. Flex manual describes the difference: *If the start condition is inclusive, then rules with no start conditions at all will also be active. If it is exclusive, then only rules qualified with the start condition will be active.*

2.1.2.2 Rules

The format of a basic rule is:

```
<START_CONDITIONS> REGEX <ACTION>
```

- **START_CONDITIONS** is an optional `<>` enclosed list of start conditions defined in previous section, or `*` to specify a rule which is active in every start condition.
- **REGEX** is a regular expression to be matched against. Flex uses a subset of regular expressions which often behaves unexpectedly, for example lookahead is not a part of match but is counted into the length of match when deciding which rule is to be the matched one.
- **ACTION** is an optional code in C or C++ language. It may contain a return statement, specifying a value returned to the caller. If no return statement is encountered the rule is non-returning, an example of non-returning rule would be a rule which skips whitespace.
2.1.2.3 User code

This section is just appended to the output file.

2.1.3 Interfacing with the generated lexer

While the generated lexer is highly configurable, only the default interface on a very simple example is shown below. The first part is the `yyin` global variable which sets the input, it accepts a `FILE*`. The second part is the int `yylex()` function, which starts lexing the file provided in the `yyin` variable and returns as soon as it encounters a returning rule, or reaches `EOF`.

2.1.4 Usage

A simple `.lex` file:

```
%( // this is simply copied to the outfile
  enum tokens {
    TOK_FLOAT
  };
}
NUMBER [[:digits:]]+
%
NUMBER "."NUMBER { return tokens::TOK_FLOAT; }
%
int main(int argc, char* argv[]) {
  yyin = fopen(argv[1]); // set input
  while (auto token = yylex()) // ... use the token
}
```

To create a C/C++ source code with Flex, a `.lex` source file is provided and a `.c/.cpp` file produced which then can be compiled.

```
flex input.lex -o output.cpp

g++ output.cpp -o lexer
```

2.1.5 Conclusion

Flex is an established generator, it is fairly easy to use, easily available and its manpage is excellent. It is capable of generating a C++ class instead of the usual C-style interface which alleviates some design flaws such as global variables etc. One of its biggest strengths is the ability to interface with `yacc/bison` (parsers) natively.
2. Current solutions

2.2 Antlr

ANTLR is not just a lexical analyzer generator. Even its manpage would suggest that as it reads: "ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It’s widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees." But it is perfectly capable of generating lexical analyzers.

2.2.1 Installation

While antlr is available in most package managers, it is Java based, implying it has to be installed as well. Antlr also supports outputing to more than just C/C++ which are not even its main targets and do not come with the installation.

The C++ target has to be downloaded from their website and compiled, which takes about 5-10 minutes. It is a CMake project making the setup easier, but still the compilation takes away the ease of use Flex provides.

2.2.2 Syntax

As with Flex, ANTLR provides the ability to create aliases for regular expressions, which are called fragments, allowing for reuse and simplifying the code.

```plaintext
fragment DIGIT : [0-9];
```

The rule format in ANTLR is:

```plaintext
RULE_NAME : REGEX { CODE } ACTIONS ;
```

- `RULE_NAME` is a string which represents the token for this rule, it can be later obtained from the generated lexer class.
- `REGEX` is a simple regular expression, with some ANTLR specific syntactical constructs (negation being one).
- `CODE` is an optional part which contains code in the target language which is copied to the generated output.
- `ACTIONS` are optional, it is a comma separated list of ANTLR specific actions, executed after rule is matched, for example skip, which specifies that its rule should not return a token even if it is the one that matched. They can be completely ignored and used as a function calls in the CODE section instead.

ANTLR also provides a mechanism to optionally activate rules, with `modes`.

```plaintext
mode example:
RULE : '...';
// other rules active in this mode.
```
2.2.3 Usage

ANTLR takes a .g4 file as its input and, when configured properly, produces C++ code, a header and a source file (already different to flex), which needs to be linked against their runtime, built while installing.

A .g4 file for a small lexer:

```java
lexer grammar lexer;
// similar to Flex definitions.
fragment DIGIT : [0-9];
fragment DIGITS : ( DIGIT+ );

TOK_FLOAT : DIGITS \.' DIGITS;
```

Then a small main.cpp:

```cpp
// the name of the file is the same as the name of our grammar.
#include "lexer.h"
...
int main(int argc, char* argv[]) {
    antlr4::ANTLRFileStream file{argv[1]};
    lexer l{file};
    
    while (!l.hit_eof) {
        auto token = lexer.nextToken();
        // use the token.
    }
}
```

The token is not an enumeration, it is a custom Token class, returned as a unique_ptr. This approach, while user friendly, does add some overhead as it is an allocation per token.

To generate and compile the lexer:

```
antlr4 -Dlanguage=Cpp input.g4
g++ main.cpp lexer.cpp -lantlr4-runtime -o lexer
```
2. Current solutions

2.3 Selected feature set

Upon inspecting the features of Flex and ANTLR a feature set which needs to be implemented has been selected.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Flex</th>
<th>ANTLR4</th>
</tr>
</thead>
<tbody>
<tr>
<td>User defined actions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Non-returning actions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Optional activation of rules</td>
<td>Yes (start conditions)</td>
<td>Yes (modes)</td>
</tr>
<tr>
<td>User-defined return type</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Scanning from memory</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Extendable interface</td>
<td>Yes (inheritance)</td>
<td>Yes (inheritance)</td>
</tr>
<tr>
<td>Lexeme part of return value</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
The term “metaprogramming” is used to describe programs that perform code generation, code introspection, or both. The task of generating a lexer might be also viewed as a metaprogramming task. Instead of transpiling a lexer definition to C++ source a lexer can be defined in C++ itself leveraging its code-generation and compile-time evaluation features. This section is an overview of the concepts needed to implement the lexer.

3.1 Constant expressions

Certain contexts require expressions that satisfy additional requirements as detailed in this subclause; other contexts have different semantics depending on whether or not an expression satisfies these requirements. Expressions that satisfy these requirements, assuming that copy elision (11.9.5) is not performed, are called constant expressions. [Note: Constant expressions can be evaluated during translation.] 

The important part is "can be evaluated". If the evaluation makes only sense at compile time (e.g. a template parameter) then the expression will either be evaluated, or the compilation will fail because the value was not provided. In C++20 the keyword `consteval` will be introduced, which implies compile-time evaluation only. Example:

```cpp
// a constexpr constructible class.
class state {
  int m_identifier;
public:
  // a class may be constructed at compile-time - a literal.
  constexpr state(int id);
};
```
// a function may be constexpr.
constexpr void sort(auto& indexable, auto&& comparator) {
    ...
}
// an example which lets the type system deduce
// whether we should return or not.
template<typename RetTy, typename F>
std::optional<RetTy> execute(F&& foo) {
    // get the return value from compiler.
    using foo_ret_t = decltype(foo());
    // let the compiler check if the type is valid in this context.
    // if not we get a hard error at compile time and an error message.
    static_assert(std::is_same_v<RetTy, foo_ret_t> ||
                std::is_void_v<foo_ret_t>,
                "Function must return the specified type or void.");
    // if the function is not void, return its value (in an optional).
    if constexpr (!std::is_void_v<foo_ret_t>)
        return foo();
    // else return an empty optional.
    return std::nullopt;
}

3.2 Class non-type template parameters (cnttp)

C++ has long been able to have non-type template parameters, they though,
were restricted to primitive types: integers, pointers, references (even doubles
are not allowed).

Until C++20, which lifted this restriction on class types (unions still disal-
lowed)[6]. The parameters must be literals (constexpr constructible example
above). This change is quite significant and one of key changes to C++ in
the upcoming C++20 standard that allow for implementation of this lexer.
Using the state class from example above, cnttp usage can be demonstrated:

// a class using cnttp.
template<class>
class bar {};
// we can use bar like this.
bar<state{12, false}> foo;
// or we could use a different class representing a string *wink*
rule<"[[:alpha:]][[:alnum:]]"> r;
3.3 Fold expressions

A fold expression performs a fold of a pack over a binary operator.\[4\]

\[fold-expression: \]
( cast-expression fold-operator ... )
( ... fold-operator cast-expression )
( cast-expression fold-operator ... fold-operator cast-expression )

fold-operator: one of
+ - * / % ^ & | << >>
+= -= *= /= %= ^= &= |= <<= >>= =
== != < > <= >= && || , .* ->*

Example:

\[template<typename... Rule>\]
\[bool match(auto&& input, list<Rule...>) {\]
\[ \quad return (false || ... || Rule::match(input)); \]
\[} \]
\[match("int", list<rule<"int">,rule<".*">,rule<"[:digit:]+">}{\}); \]

An fold operation can be explained on the example above. Suppose a rule class has a match static method returning a bool. The fold expression executes that static method for each rule and computes whether any one of them matched. Folds which use logical operators can short-circuit, others cannot. It can be described as:

\((false || rule1::match(input)) || rule2::match(input)) || ...\)
This chapter aims to explain some of the classes exposed to the user of CTLE. It does not go into the implementation itself, it is supposed to be a "manual" explaining how to use the interface to create a lexer.

4.1 states

```
template<typename StateT = std::nullptr_t,
         typename StateList = ctll::list<>>
struct states
```

states is a container for specifying states in CTLE. The states in CTLE serve the same purpose as mode in ANTLR4 or START_CONDITION in Flex.

The StateT is the type that is used to represent states, it has to be convertible to int for CTLE to work correctly so enums are the best match, but only convertibility to int is required so the type is not otherwise restricted. This parameter is optional.

StateList is a ctll::list of class state. This parameter is optional.

4.1.1 state

```
template<StateT Identifier, bool Exclusive = false,
         typename Actions = actions<>>
struct state
```

The state is a container for data needed to specify a state in CTLE.

The Identifier is the value used as the identifier for this state. It is later used in rule definitions, its type should be StateT. Its value must be bigger that of state_reserved otherwise a compiler error occurs.

Exclusive parameter is a bool specifying whether this state is to be exclusive or inclusive, mimicing the behaviour of Flex. This parameter is optional and states are inclusive by default. false - inclusive true - exclusive.
Actons is a specification of the actions class. This parameter is optional.

4.1.1.1 actions

template<callable Eof, callable NoMatch>
struct actions

actions is a container for actions to be executed when an event is encountered in a certain state. Actions in CTLE are discussed in subsection 5.1.2, but the actions used in this context do not receive a lexeme as a parameter, which is the only difference. Both of these actions can be specified as empty_callable in which case the defaults for a lexer are used.

Eof is the action executed when lexer encounters an EOF while lexing in this state. NoMatch is the action executed if none of the rules active in this state match.

4.1.2 Example

Usage:
// define the state identifier.
enum class state_ids {
  IN_COMMENT,
  MULTILINE_STRING
};
// set a state specification
using state_def = states<
  state_ids,  // StateT
cwl::list<  // StateList
    state<    // state_ids::IN_COMMENT, // id
      true,   // exclusive
      actions<[]{throw std::runtime_error("EOF while reading comment");}>,
    // the last two parameters are optional
    state<state_ids::MULTILINE_STRING>
  >;

4.2 rule

template <ctll::basic_fixed_string Pattern,
           callable Action = empty Callable,
           std::array States = std::array(state_initial)>
class rule

rule is a container for all the information needed to specify a rule in CTLE.
Pattern is a regular expression which needs to match if this rule is to be used. Action is the action which should be executed if this rule is chosen as best match (actions are further discussed in subsection 5.1.2). A sentinel empty_callable may be used to signal no action is to be executed. States is an array of identifiers, in which this rule is supposed to be active. There are two special state identifiers that are always defined, even if user defined no states: state_initial and all_states. The former is the default for all rules and represents the initial state of the lexer, e.g. rules with no other state specified, the latter signalizes that this rule shall be active across all the states.

4.2.1 Example

```cpp
// a simple rule matching all identifiers
using rule1 = rule<"[[:alpha:]][[:alnum:]]*",
   [](auto &&...){ return tokens::IDENTIFIER; }>
// a rule skipping non * characters in comment.
using rule2 = rule<"[^\*]+", empty_callable,
   std::array{state_ids::IN_COMMENT}>
// a rule skipping all whitespace
using rule3 = rule<"[ \t\r\n]+">;
```

4.3 lexer

```cpp
template<typename ReturnType, typename Rules,
   typename States = states<>,
   typename Extensions = extensions<>,
   typename Actions = defaults<typename Rules::type>,
   typename IteratorT = const char*>
class lexer : public DerivesFrom

This is the container for all the data needed to construct the lexer, only the first two parameters are mandatory. ReturnType the return type of one call to lexing function of this lexer. This is usually an enumeration type in which the tokens are specified. If the user wishes to use default actions for handling EOF and NoMatch events in the lexer, it must be an enum which contains eof and no_match fields (or else the compilation will fail). Rules must be a ctau::list of rule class. States a specialization of states class, see section 4.1. Extensions see section 5.9. Actions are the actions for the state_initial see subsubsection 4.1.1.1 IteratorT is a forward iterator, which is used to specify the input of lexing.
4. CTLE - Interface

4.3.1 Example

```cpp
enum tokens {
    integer,
    eof,
    no_match
};

enum state_id {
    // all state ids must be bigger than the
    // reserved value
    comment = state_reserved
};

using rules = ctll::list<
    rule<"\[[\[:digits:\]]\]*\">>, [](auto&&...){
        return tokens::integer;
    },
    rule<"\[\]">,
    empty_callable,
    std::array{state_id::comment}
>; // only rules and ReturnType are mandatory, everything after is
    // completely optional.
using state_defs = states<state_id, ctll::list<
    state<comment>
    >>;
...
using number_lexer = lexer<
tokens, rules, state_defs, ...>;
```
This chapter goes through all the challenges and decisions that had to be made and shows the thought process behind the design itself. Each section describes a goal, and how this goal was implemented.

5.1 Specifying a rule

In flex, there are three parts to a rule. The regular expression, the action, and a set of start conditions the rule itself is active in. CTLE mimics this, but he start conditions are called states \texttt{states}. Each part is discussed in a dedicated section.

5.1.1 Storing the string representation of regular expressions

\textbf{Goal:}

\begin{verbatim}
rule<"[[:alpha:]]+">;
\end{verbatim}

While a similar behaviour can be achieved by using const char* as ntp in C++17, the syntax is not as convenient as the proposed one. The pointer must have static linkage so in C++17 the syntax would be.

\begin{verbatim}
template<const char*> class rule
Usage:
constexpr char pattern[] = "[[alpha:+]]";
rule<pattern>;
\end{verbatim}

In C++20, cnttp are allowed.

\begin{verbatim}
template<string_class> class rule
Usage:
rule<"[[alpha:+]]">;
\end{verbatim}
5. Design

Which leads to the desired syntax. An idea of how `string_class` might be implemented follows.

```cpp
template<typename CharT, size_t N>
class string_class {
    CharT m_data[N]; // static arrays are constexpr constructible.
public:
    // actual implementation ommited.
    constexpr string_class(const CharT (&input)[N]);
};
```

`fixed_string` from ctre library is used in the actual implementation, from which this example was "paraphrased".

5.1.2 Specifying an action

Flex with its actions has an advantage here, it can copy blocks of code specified as actions into the output code and document a set of variables/function calls/method calls available to that block of code. In C++, however, code cannot be copied at compile time. That means an action must be a callable, be it functor or a function. That raises 2 big questions.

**Question 1:** What should the parameters passed to an action be? The answer to this question is pretty straightforward, an action must be able to access the matched lexeme as well as the lexer its rule belongs to. Which implies the signature this signature for an action should be:

```
return_t (lexer_t &lexer, lexeme_t lexeme)
```

The type of the lexer however, is influenced by its rules. So when the action is specified the `lexer_t` is not yet known. That means, the instantiation of the action must be delayed until `lexer_t` is known. Put shortly, the first parameter has to be templated. By doing so, functions cannot be used anymore, as templated functions cannot be used as template-template parameters. That leaves functors and a signature:

```
return_t (auto &lexer, lexeme_t lexeme)
```

If CTLE did not suport captures in the regular expressions, the previous signature would be the one used for all actions. However, CTLE does support captures, and that means, an action must somehow recieve those as well. That implies that the signature must change with the regular expression to accommodate for the captures. So a rule without captures could use the signature described above. But as captures are added the signature becomes.

```
return_t (auto &lexer, lexeme_t lexeme,
          lexeme_t capture1, ..., lexeme_t captureN)
```
Question 2: How should the value returned from an action returned to the caller of the lexing function? In flex, this is a non-issue, as actions are simply blocks of code copied into the output source code, so when there is a return statement in the action, then there is a return statement in the generated lexing function.

As stated before, actions must be functors. That means, the action is being called from another function (lets call it the lexing function) and even if it does return, that does not mean that the lexing function returns. Furthermore a way to specify actions which do not return values needs to be provided, f.x. an action that signals that counts lines in the comment but does not return a token as comments are ignored.

A way to do this would be to force the user to have a no_return constant and return from the lexing function only if the value returned from action is different to that of no_return. Which would lead to a lot of code bloat for the user.

Another way to implement such behaviour would be to force the action to return a std::optional and make the lexing function return only if it is not empty, which is a little bit more user-friendly as user does not have to define constants. It is a good start but it still leads to some boilerplate code when specifying an action.

The it is implement in CTLE by leveraging the type-system of the compiler and combining it with the std::optional approach. The return_t of the signature has not been discussed yet and that is because the return type of the action specifies which rules are returning and which are not. There are three types of return types allowed.

```cpp
// this action is not returning a value.
void (lexer_t&, lexeme_t)
// this action may return a value.
std::optional<lexer_t::return_t> (lexer_t&, lexeme_t)
// this action always returns.
lexer_t::return_t (lexer_t&, lexeme_t)
```

An idea about how this behaviour is implemented might be obtained from section 3.1.

Combining all the previously discussed decision with the fact that lambas are also functors, leads to a syntax very similar to the one Flex uses.

5.1.3 Storing an action

Goal:
rule<"int", [](auto&&...) { return tokens::INT; }>

To implement this behaviour cttpp can be leveraged again. An idea of how CTLE implements callable:
5. Design

```cpp
template<typename FunctorT>
class callable {
    FunctorT m_functor;
public:
    constexpr callable(FunctorT&& functor);
    // calls the operator() of the contained functor.
    auto operator() (auto&& ...);
};
// and we extend our rule class
template<string_type, callable>
class rule

A rule may signal that no action is needed through using a special sentinel empty_callable as its action.

5.1.4 Storing a list of valid states

Goal:
rule="[[:alpha:]]", empty_action, { states::IN_COMMENT }

This is by far the easiest of the three parts of a rule. std::array is a constexpr constructible container so it can be used as cnttp, extending the rule class signature for the last time.

```cpp
template<string_type, callable, std::array>
class rule
Usage:
rule="[[:alpha:]]", empty_action,
    std::array{ states::IN_COMMENT }
```

5.2 Regular expressions

The construction of regular expressions at compile-time is out of scope of this thesis and this task is delegated to CTRE. A Compile-Time Regular Expressions library made by Hana Dusíková. Its usage is:

```cpp
ctre::match<"REGEX">(input);
```

Where the match takes a ctre::fixed_string as its template parameter. However, a match of regular expression means, that the whole input matches the pattern. This is not the desired behaviour while lexing, search on the other hand, searches for a pattern anywhere in the input, not necessarily a pattern starting at the start of the input. Example of the behaviours:

Regex : "[[:alpha:]]+"
Match : "foo" true, "foo1" false
Search : "foo1" true, "1foo" true

22
Lexical analyzers however match input differently, it can be described as a regular match in which input is modified to "^input.*". In other words a search anchored to the start of the input. This can be achieved with CTRE, although it is not the part of the standard interface so it is added by CTLE.

Another interesting characteristic of CTRE is that it allows captures in the regular expressions and returns a tuple-like object from match that adapts to the regular expression passed as its template parameter, which combined with a std::apply-like approach is how CTLE allows different action signatures for different number of captures.

### 5.2.1 Storing the results of a regex match

A return type of the regular expression matching depends on the regular expression itself, a regular expression with no captures will return a tuple-like object of size one, with each capture making the size grow by one.

C++ runtime polymorphism can be used to store different types behind the same interface, but that is slow, as it requires allocation and the method dispatch must go through vtables.

std::variant could also be used, which is faster but generates a lot of code.

This problem however, can be approached from a different perspective. The it can be defined as storing up to $N$ lexemes, $N$ being the maximum number of captures between all rules. $N$ can be obtained at compile-time and thus std::array of size $N$ can be used to store the lexemes, filling $M$ first elements returned from the match, $M$ being the size of tuple-like object returned from the matching function.

```cpp
using storage_t = std::array<lexeme_t, N>;
```

### 5.3 Using the result in an action

The rule itself, has the information about how many captures its pattern has, so the information about $M$ elements filled in the storage is available within the rule. Only those $M$ first elements out of the storage are forwarded to an action along with the reference to the lexer itself thus satisfying the signature of the provided action.

```cpp
static return_t do_action(lexer_t& l, storage_t t) {
    return ctle::partial_apply<M>(action, l, t);
}
using action_t = return_t(*)(lexer_t&, storage_t);
```

The wrapper around the actual call to the action has a stable signature, which means any action can be stored as a pointer to its wrapper.
5. Design

5.4 The result of a rule matching

By creating a structure containing `storage_t` which can store the lexemes and `action_t`, a polymorphic behaviour can be achieved with value semantics.

```c
struct match_result {
    storage_t storage;
    action_t action;
    ...
}
```

This solution can be thought of as if a type-erasure was used. It is unknown how many elements are filled in the storage as well as which action is stored inside, but they are from the same rule, hence the action itself uses only the correct $M$ first elements from the storage.

5.5 Storing rules and state definitions

Rules and definitions are completely defined by their type. In the previous chapter `ctll::list` from CTRE library has been used to store any number of types. The whole implementation fits on a single line:

```c
template<typename... Ts> struct list{};
```

Accessing the stored types

```c
// a function using said types
template<typename... Ts> void foo_impl(ctll::list<Ts...>) {
    // use the types as an argpack again.
}
```

```c
template<typename L>
void foo() {
    foo_impl(L{});
}
```

```c
// called with a list.
foo<ctll::list<int, char, double>>();
```

5.6 Making the lexing function

First, the `match_result` structure and add a binary operator, which is going to be used in a fold expression, this operator is also the one responsible for handling the colisions in the lexer. A simple implementation:

```c
match_result operator|(match_result other) {
    return (length() < other.length()) ? other : *this;
}
```
5.7. Generating lexing function for states

Where length is the length of the first element in the storage. This element is always present because it represents the whole matched lexeme. CTLE mimics Flex and returns the longest matched lexeme, or the one which came first if collision occurs.

The rules stored in a ctll::list can be accessed as an argpack. Combined with fold expressions (discussed in section 3.3) a lexing function can be implemented:

```cpp
template<typename... Rule>
auto lex(ctll::list<Rule...>) {
    return (match_result{} | ... | Rule::match(begin, end));
}
```

5.7 Generating lexing function for states

The lexing function is specified by the list of rules provided as its parameter. Different lexing functions for different states can be obtained by providing different lists of rules. Ignoring that rules are defined as types, the algorithm for creating lexing functions for different states can be described as:

```cpp
for (state : states) {
    rule_list r;
    for (rule : rules) {
        if (rule.active_in(state))
            r += rule;
    }
    create lex(r);
}
```

This algorithm can be applied to types, but implementation is non-trivial, so it is not shown in this text, but describe the idea. The definitions of rules and states provide enough information to decide which rule is active in which states, which can be done for each rule and if the rule is active it is added to the list for a state. The resulting calling convention is:

```cpp
lex(get_active_in_state<state>());
```

5.8 Allowing user to change the state of the lexer

Changing the state of the lexer means changing the list of rules that are active. That means changing the lexing function itself. The caller however, cannot pass the list as it is generated internally by the lexer. To get around this issue, a wrapper is defined around the lexing function.
5. Design

```cpp
template<typename Rules>
result_t lex_wrapper() {
    return lex(Rules());
}
```

Call syntax:

```cpp
lex_wrapper<get_active_in_state<state>>();
```

This wrapper a stable signature, uninfluenced by the list of rules passed into it, meaning pointers to such wrappers can be stored as:

```cpp
using lex_ptr = result_t(*);
lex_ptr lex_function = &lex_wrapper<get_active_in_state<state>>;
```

This pointer to a function represents a state in the lexer. It can be stored in any container, for example `std::array` which is constexpr constructible and has random access as required size is known at compilation time (number of states + 1 for initial state). The change of state, now means finding the correct function for a requested state in the container and setting the `lex_function` value in the lexer.

### 5.9 Making the lexer extendable

The biggest obstacle in extending the lexer is the reference to the lexer being available to every action. If the approach Flex and ANTLR was used, a base class from which the lexer would have to inherit would need to be defined. While this approach is viable, and the action will get the correct and complete type, it has one flaw. The base class itself is not able to access the methods of its derived class, the lexer itself. This becomes an issue when a functionality that needs to access the API of the lexer needs to be implemented, for example an ability to lex from file, which implies setting the input. If the relationship is reversed and the extension class inherits from the generated lexer, the issues of accessing the lexer methods are resolved but the action, will not get the reference to the extended class but only to the lexer itself.

#### 5.9.1 CRTP

The need to know the type of Derived class in the Base class is not a new one. It can be solved through the use of templates. The Curiously Recurring Template Pattern (CRTP) is an idiom in C++ in which a class X derives from a class template instantiation using X itself as template argument\[^1\]. An example of this idiom in use:

```cpp
// base can now cast its this pointer and access
// members of derived.
template<typename DerivedT>
struct base {};
```
5.10. Modularity

// use derived as template parameter for base.
struct derived : base<derived> {};

This alleviates the issue with the base class approach. CTLE provides an `ctle::extension` class to inherit from, which performs the needed cast internally and provides a `lexer()` method to access the correct type, but extensions need not inherit from it, the only requirement is for extensions to take exactly one template parameter.

```
template<typename LexerT>
struct from_file : ctle::extension<LexerT>
...
```

5.10 Modularity

Where both approaches fall short is modularity. While the base class could be reused in multiple lexers, adding functionality to this base class would lead to the interface growing and being unusable in the long run, making it very hard to write any tests and leaving parts of it unused most of the time. This is the reason why the CTLE was designed to handle many small extensions. They have to be passed to the lexer via the `ctle::extensions` class.

Usage:
```
using extensions = ctle::extensions<from_file, line_counter,...>;
using lexer = ctle::lexer<rules,states,extensions>;
```

The lexer inherits from these extensions publicaly, so all of their interfaces are available in the interface of the generated `lexer`.
Comparing the generated lexers

Three equivalent lexical analyzers have been created with Flex, ANTLR and CTLE and compiled with the same flags.

```c
$ g++ -Ofast -std=c++2a
```

First, the size of generated executables is compared. Linux `stat` command is used first, to get the on-disk size, and then the `size` command, which discards sections not loaded into memory when executable is ran.

<table>
<thead>
<tr>
<th></th>
<th>Flex</th>
<th>ANTLR4</th>
<th>CTLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>on-disk size</td>
<td>97680</td>
<td>750304</td>
<td>3267536</td>
</tr>
<tr>
<td>size</td>
<td>87510</td>
<td>555255</td>
<td>71076</td>
</tr>
</tbody>
</table>

Figure 6.1: sizes of binaries

CTLE binary is order of magnitude bigger than ANTLR and two orders behind flex when on-disk layout is considered, but when loaded into memory, it is the smallest.

```bash
> readelf -a ctle
...[28] .symtab SYMTAB 00000000000000000000000000000000 00015170
      00000000000000000000000000000000 00016b98
      00000000000000000000000000000000 0032f38
      00000000000000000000000000000000 00015170
[29] .strtab STRTAB 00000000000000000000000000000000 00015170
      00000000000000000000000000000000 00015170
      00000000000000000000000000000000 00015170
      00000000000000000000000000000000 00015170
...[30] .shstrtab STRTAB 00000000000000000000000000000000 00015170
      00000000000000000000000000000000 00015170
      00000000000000000000000000000000 00015170
      00000000000000000000000000000000 00015170
...```

Figure 6.2: readelf output for CTLE

When the actual content of the binary is inspected approximately 3.2MB out of the whole size it occupies on disk are being used by `.strtab` section. This section contains string representations of all its symbols. In CTLE, whole
6. Comparing the generated lexers

lexer is specified as a type, which gets (along with other generated symbols) stored here as a mangled string, implying this section grows as the definition of the lexer grows. The reported size from `size` indicates, that the section itself is not used and can be removed, making CTLE binary the smallest.

```bash
> strip --strip-unneded *
```

<table>
<thead>
<tr>
<th>B</th>
<th>Flex</th>
<th>ANTLR4</th>
<th>CTLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>on-disk size</td>
<td>92336</td>
<td>609544</td>
<td>76168</td>
</tr>
</tbody>
</table>

Figure 6.3: sizes of binaries after `strip` command

6.1 Benchmarking

Clang frontend from LLVM was chosen as the testsuite. It is large (12K files), and contains a wide variety of C++ constructs. The benchmark was ran 30 times, while swapping the order in which the files were being lexed to avoid one lexer having the advantage of said file being cached from previous runs. The best and worst results have been removed for each lexer and a file and a mean of the remaining runs used as the result.

<table>
<thead>
<tr>
<th>S</th>
<th>Flex</th>
<th>ANTLR4</th>
<th>CTLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>194.51</td>
<td>206.78</td>
<td>220.74</td>
</tr>
</tbody>
</table>

Figure 6.4: benchmark results

The graph in Figure 6.4 shows only the files <25KB, those however make up 95% of all files in the benchmark.

---

391% of all files are smaller than 17.5KB.
6.2 Inspecting assembly generated by CTLE

A small lexer has been created to analyze the generated assembly.

```cpp
enum class tokens { INT = state_reserved, PRINT, no_match, eof };

using rules = ctle::list<
    rule<"\[[:.\d:.\d:]+", [](auto&...){ return tokens::INT; }>,
    rule<"[\t\r\n]+">,
    rule<"print:'(.*?)'", [](auto& lexer, auto lexeme, auto to_print)
    -> std::optional {
        lexer.set_state(state_initial);
        if (!to_print.length())
            return std::nullopt;
        printf("%s", to_print.begin());
        return tokens::PRINT;
    }>
>
using lexer = ctle::lexer<tokens, rules>;
```

Figure 6.5: the analyzed lexer

6.2.1 Actions

An action in CTLE is a callable literal, which is stored in a ctle::callable literal. To execute an action the operator() of said literal has to be called. Furthermore this literal is passed into another functor that wraps its call and decides if an action is returning or not. This last literal is then passed into a modified version of std::apply. All in all, three levels of indirection:

input → ctle::callable → ctle::action → ctle::apply.

First, an empty action is inspected (sentinel empty_callable).

```asm
EMPTY_ACTION:
    movb $0x0,-0x4(%rsp)
    mov -0x8(%rsp),%rax
    retq
```

Figure 6.6: assembly of empty action

This action is no action at all. std::optional is used internally in CTLE to signal returning/non-returning actions so CTLE cheats a little here because this does not go through the same indirections as the other actions but immediately returns an empty optional. Notice the movb setting a flag in optional to false(0).
Next, the most common action in any lexer, returning a token.

```c
[](auto&&...) { return tok::INT; }
```

**TOKEN ACTION:**

```assembly
movl $0x1,-0x8(%rsp)
movb $0x1,-0x4(%rsp)
mov -0x8(%rsp),%rax
retq
```

Figure 6.7: assembly of an action returning a token

This looks quite similar to the first action. There are only 2 differences here. First, the value of optional is set to `tok::INT` by `movl` instruction. Then the `movb` sets the flag to `true` this time.

Analyzing an action that uses its lexeme.

```c
[](auto lexer, auto lexeme) {
    printf("%s", lexeme.begin());
    return tok::INT;
}
```

**PRINT_INT:**

```assembly
sub $0x18,%rsp
mov 0x8(%rsi),%rsi
mov $0x402624,%edi
xor %eax,%eax
callq 400e60 <printf@plt>
movl $0x1,0x8(%rsp)
movb $0x1,0xc(%rsp)
mov 0x8(%rsr),%rax
add $0x18,%rax
retq
```

Figure 6.8: assembly of action using a lexeme

A pattern can be observed in all of these actions (and in all actions in CTLE). The compiler sees the indirections and optimizes them away. This implies that while it is not possible to copy blocks of code at compilation time, as Flex can when generating the lexer, it is possible to produce just as efficient code by inlining the indirections.
6.2. Inspecting assembly generated by CTLE

6.2.2 Matching function

Only the important parts of assembly for the matching function will be inspected. The whole assembly is only 216 instructions and can be found in the attached CD.

MATCH:

```
... movq $0x400f70,0xe0(%rsp) ...
... test %r8,%r8 je 401290 MATCH+0x300 ...
... callq *%r8 ...
```

There are three important parts in the assembly above. First is the movq instruction which sets the function pointer to an action which should be performed. This is repeated for every part of the fold expression along with the call to the matching function. The assembly of the matching function may also be inlined, but it is not included as it is out of scope of this thesis. The test checks if the pointer to an action is not a nullptr in which case the call to an action is avoided in favor of a jump. The last important call is the callq which executes the action stored previously.
This thesis studied current approaches to generating lexical analyzers, namely Flex and ANTLR4 lexical analyzer generators. Its goal was to select a feature set from these two solutions and completely reimplement it in a C++20 only solution - CTLE. The design of CTLE allows for better extensibility than that of ANTLR4 or Flex generated lexer while keeping the interface familiar to Flex whenever possible.

The generated lexer has been shown to produce smaller binaries than both Flex and ANTLR4 sitting in between ANTLR4 and Flex in the speed benchmarks. This study proves that use of new C++20 features does not necessarily lead to big binaries and can produce at least as efficient code as if it was written by hand. Where C++ even in its C++20 version still lacks behind other languages supporting metaprogramming is that most of it is done through types, leading to big symbol tables and by extension to slow compilation times and big .strtab sections in resulting binaries.

The main area CTLE can be improved is the pattern matching itself. While it is reasonably fast even in its current state, all the patterns provided in rules are matched against the input independently, which causes the complexity to be $O(nm)$ where $n$ is the length of the input being matched and $m$ is the number of rules. This overhead can be avoided by building a Deterministic Finite Automata out of the provided patterns, thus making the complexity $O(n)$, $n$ being the length of matched input. And while this can be done at compile-time it is out of scope of this thesis.
Bibliography


Appendix A

Contents of the attached CD

- readme.txt ......................... description of the contents of the CD
- exe ................................ directory containing executables
- src
  - impl................................. source code of implementation
  - thesis.............................. source code of the thesis in LaTeX
- text ........................................ text of the thesis
  - thesis.pdf ......................... text of the thesis in PDF format
  - thesis.ps ......................... text of the thesis in PS format