Generating Parsers in C^{++} with Maphoon (1)

Hans de Nivelle

Nazarbayev University, Astana, Kazakhstan

Online, 15.07.2022

Kazakhstan and Nazarbayev University

- The present state of Kazakhstan exists since 1991, but Kazakh as national identity exists much longer, since 15th century AD.
- Nazarbayev University was established in 2010. Before that, the best students of Kazakhstan were sent to universities abroad.
- NU has 4687 undergraduate students, 1636 graduate students, and 584 students in the foundational year.
- Proficiency in English is a requirement for everyone, students and staff alike.
- All important decisions are based on merit only.

School of Engineering and Digital Sciences (SEDS)

- The School of Engineering and Digital Sciences has 2000 undergraduate students, ≥ 500 graduate students.
- The CS department has ≥ 1000 students, and is growing.
- There are vacancies at the CS department.

Why I created Maphoon

I am developing a programming language for implementation of logic (verification of mathematical proofs and theorem proving). For this language, I need a parser.

I have been teaching compiler construction, using Java as implementation language.

There are very nice tools for Java, namely JFlex (`www.jflex.de/`) for automated tokenizer generation, and CUP (`www2.cs.tum.edu/projects/cup/`) for automated parser generation.

I want to use C^{++} for the implementation of my programming language, I wanted to use tools similar to JFLex and CUP, but they were lacking.

In future version of the compiler construction course, I want to use C^{++}.

Reasons for Creating

For big, established languages like C^{++} or C, one probably does not need automatic parser generators.

Syntax does not change often, and the resources spent to implementing the parser are small, compared to the total resources spent on language and compiler development.

For experimental languages, like mine, they are useful. It is very easy to make changes in syntax with a parser generator.

I decided to write a tokenizer generator and parser generator by myself.

I am a university teacher. I want to be able to show the constructions in class.

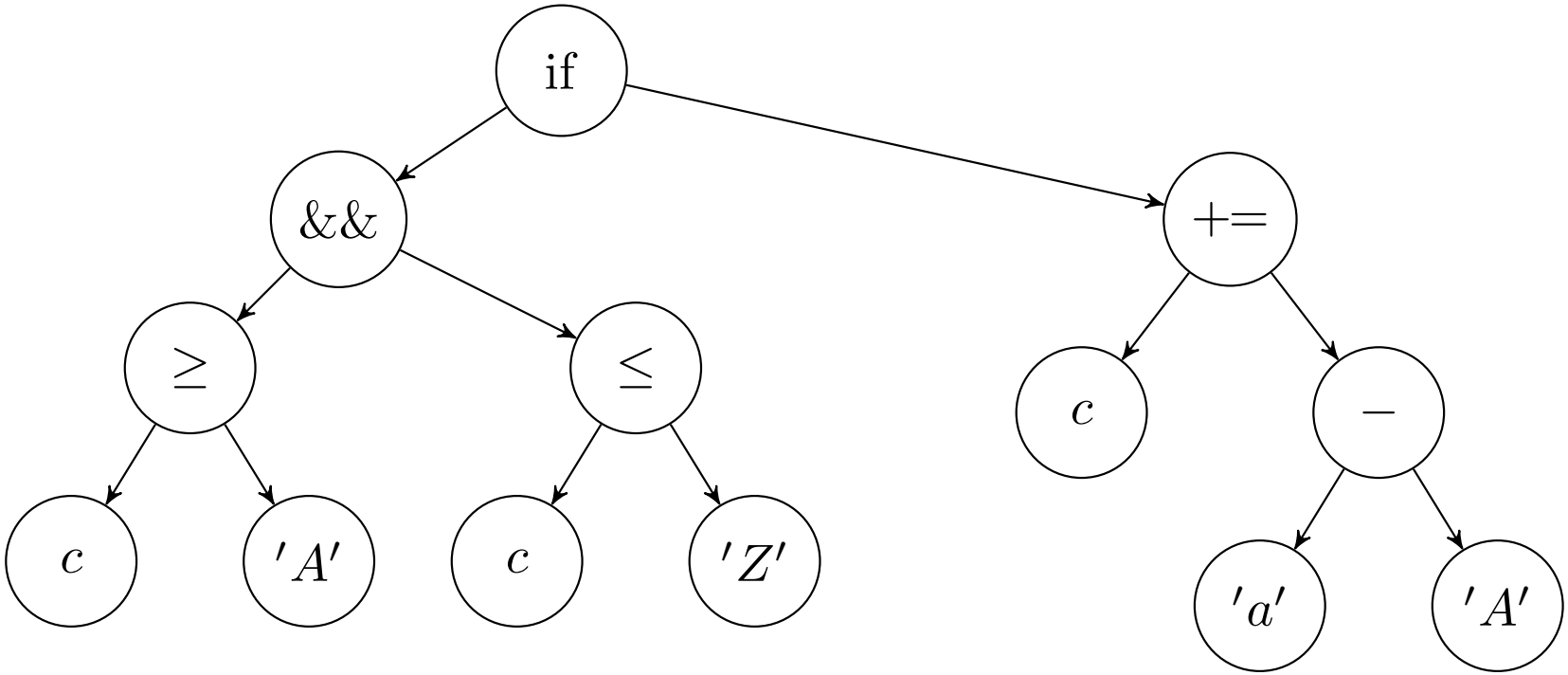
What is Parsing?

In computer science, nearly everything can be represented as a tree.

We are given some input as sequence of characters. The task of the parser is to extract the tree structure. The resulting tree is called abstract syntax tree.

Example of an AST

```
if( c >= 'A' && c <= 'Z' ) c += 'a' - 'A';
   // If c is upper case, make it lower case.
```



Tokenizing

Tokenizing is the first step.

Tokens typically have the following forms:

- Numbers (integers or floating point)
- Strings (can be quite complicated, when there are escape codes)
- Reserved words, operators
- Comments
- Indentation changes (In Python, these result in tokens)

Automatic Generation of Tokenizers

There exists a very nice theory for recognizing and classifying tokens:

Regular Expressions ⇒ Non-Determinstic Finite Automata ⇒ Deterministic Finite Automata ⇒ Minimal Deterministic Finite Automaton.

There exist many excellent tools that implement this theory, also for C or C^{++} :

flex: (`github.com/westes/flex`),

re2c: (`re2c.org/`)

(There are more)

Why not use an existing implementation?

The problem is lack of flexibility.

There is always something that doesn't fit in:

- My logic language uses Python-style indentation.
- It also uses comments of form `#< ... >#` , which can be nested. (They play the role of `#if 0 ... #endif`). These tokens are non-regular.
- Long comments of form `/* ..... */` should not be stored in the buffer. It is better to handle these separately.
- For some languages (I will not mention their names), the tokenizer must have access to type definitions.
- Some languages allow a certain token `>>` only in certain contexts. (Again, I am not mentioning any names.)

I hesitated long if it is worth automating tokenizing. In the end, I did it, keeping the following goals in mind:

- Flexibility: It must be possible to handle some tokens by hand.

  The tokenizer must not dictate how source information is handled.

- Usable in education. I am a university professor and I want to show the automata to students.

- Efficiency (up to the point where flexibility has to be sacrificed)

I created the building blocks for the tokenizer, but not the tokenizer itself.

On the next slide, I show a **filereader** class. It forms the buffer between the tokenizer and the input source.

Since my tokenizer generator does not generate the complete tokenizer, the user has to interact with this class.

Class Filereader

A filereader contains a pointer to a file. In addition to that, it keeps track of line number and column position. It has an unbounded buffer of characters that is initially empty.

The main methods are:

- `bool has( size_t n )` : Read characters from the source until the buffer has size n and return `true` on success.
- `char peek( size_t i ) const` : Get the i-th character from the buffer.
- `string_view view( size_t i ) const` : Get the first i characters in the buffer as `string_view`.
- `commit( size_t i )` : Remove the first i characters from the buffer. The buffer must contain $n \geq i$ characters, and after committing, it contains $n - i$ characters.

Recognizing Tokens by Hand (1)

For the tokens that you want to recognize by hand, write a function of form:

`std::pair< tokentype, size_t > try_X( filereader& inp ).`

If the attempt failed, the second field must be 0.

It is possible to recognize more than one type of token in a single function.

The following function may return different token types dependent on the type of number:

```
std::pair< tokentype, size_t >
                        try_number( filereader& inp )
```

Recognizing Tokens by Hand (2)

This is how one interacts with `filereader`:

```
std::pair< tokentype, size_t >
tokenizer::try_identifier( filereader& inp )
{
   if( inp. has(1) && starts_ident( inp. peek(0)) )
   {
      size_t i = 1;
      while( inp. has(i+1) &&
                continues_ident( inp. peek(i) ))
         ++ i;
      return std::pair( sym_IDENT, i );
   }
   else
      return std::pair( sym_IDENT, 0 );
}
```

Automatic Recognition

Most of the tokens can be recognized by a finite automaton.

For the time being, we recompute automaton every time the program is started:

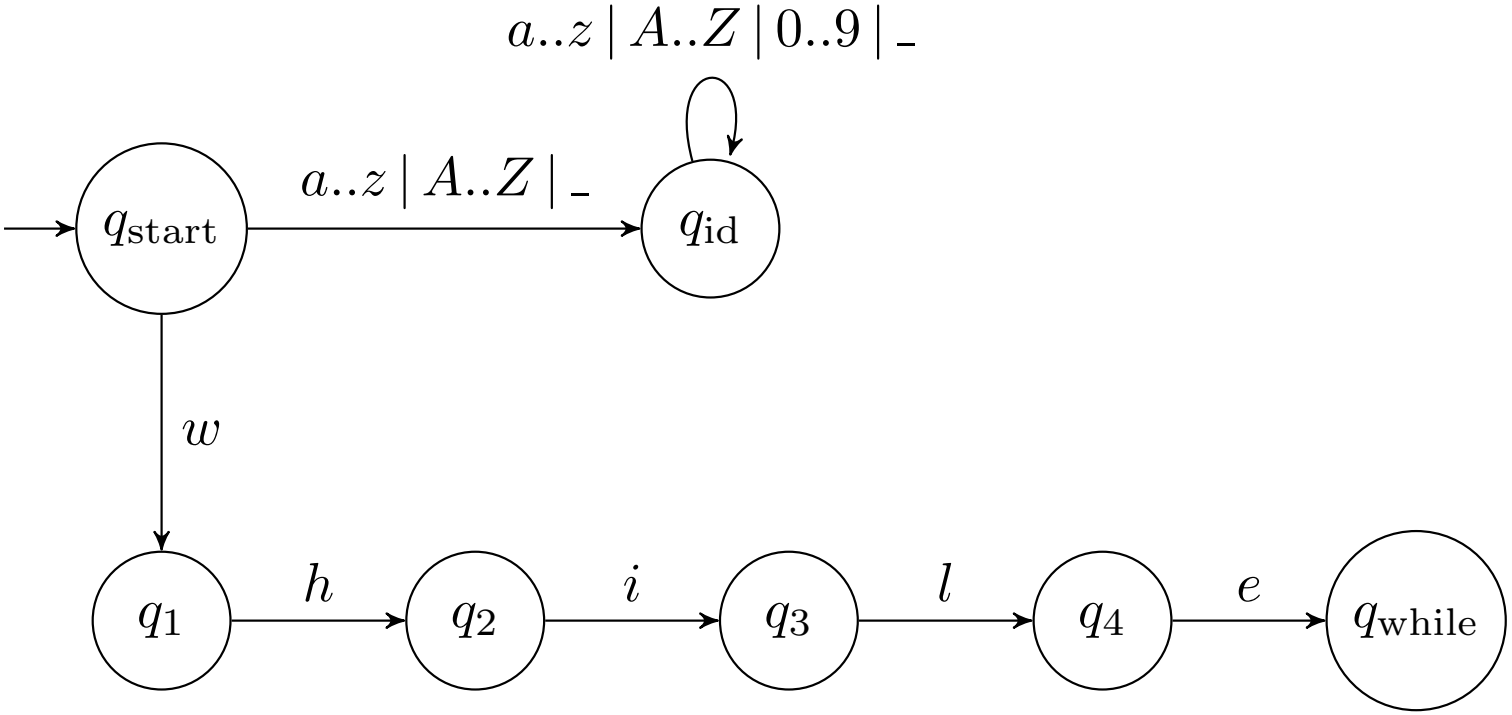
```
static lexing::classifier< char, tokentype >
   cls = buildclassifier( );
```

Using the automaton:

```
std::pair< tokentype, size_t >
       p = readandclassify( cls, inp );
```

`inp` is a `filereader&`, and the return value is the same as for hand-written functions.

Refreshing Automata: Identifier and a reserved Word "while"



Flat Automata

We use a nicer and simpler way for representing finite automata (see GandALF 2022).

- Transitions are always labeled with intervals, but the intervals are broken up into step functions.
- We use relative addressing for the states.

Our representation simplifies the representation, the standard algorithms (combining, determinization and minimization) and works in practice without change: 'what you teach is what you use'.

A state is represented a triple (Λ, ϕ, T), in which

- Λ is a set of integers, representing the ϵ transitions.
- ϕ is an ordered map from `char` to (integer or #), representing the transitions from this state. It must have at least have a value for the minimal `char`. (I will assume it is \$80.) In order to apply it on character c, find the $(c', n) \in \phi$ with maximal $c' \leq c$.
- T is the classification of the state.

A classifier is a vector of states. We use relative addressing for state references.

nr	Λ	Φ	T
0 :	$\{1, 4\}$	$\{\ (\$80, \#)\ \}$	**err**
1 :	$\{\,\}$	$\{\,(\$80, \#),\ (A, 1), (Z^{+1}, \#),\ (a, 1), (z^{+1}, \#)\,\}$	**err**
2 :	$\{1\}$	$\{\,(\$80, \#),\ (0, 0), (9^{+1}, \#),\ (A, 0), (Z^{+1}, \#),$	
		$(\_, 0), (\_^{+1}, \#), (0, 0), (9^{+1}, \#)\,\}$	**err**
3 :	$\{\,\}$	$\{\,(\$80, \#)\,\}$	**ident**
4 :	$\{\,\}$	$\{\,(\$80, \#), (w, 1), (w^{+1}, \#)\,\}$	**err**
5 :	$\{\,\}$	$\{\,(\$80, \#), (h, 1), (h^{+1}, \#)\,\}$	**err**
6 :	$\{\,\}$	$\{\,(\$80, \#), (i, 1), (i^{+1}, \#)\,\}$	**err**
7 :	$\{\,\}$	$\{\,(\$80, \#), (l, 1), (l^{+1}, \#)\,\}$	**err**
8 :	$\{\,\}$	$\{\,(\$80, \#), (e, 1), (e^{+1}, \#)\,\}$	**err**
9 :	$\{\,\}$	$\{\,(\$80, \#)\,\}$	**while**

Remaining Topics

I show you in code

- How the classifier is created, using code.
- The classifier.
- How to make the classifier deterministic.
- How to minimize the classifier.
- How to deal with EOF and bad files.
- How to ignore whitespace and comments.
- How to compute attributes.
- How to obtain a maximally(?) efficient tokenizer, using truly dirty trickery.

This completes the topic of tokenizing.

Questions, Possible Future Directions

- Should one use `constexpr`?

  Requires `constexpr` vector and map. Currently not available. Moreover RE2C claims that tables are an order of magnitude slower than direct coding.

- Theory works in 99% of cases, but you need an escape for the remaining 1%.

- Other automata constructions? (Intersection).